Dr. Joseph Teguh Santoso, S.Kom, M.Kom

STRUKTUR DATA dan ALGORITMA (Bagian 2)

YAYASAN PRIMA AGUS TEKNIK

STRUKTUR DATA dan ALGORITMA (Bagian 2)

Dr. Joseph Teguh Santoso, S.Kom, M.Kom

BIODATA PENULIS

Dr. Joseph Teguh Santoso, S.Kom, M.Kom adalah Rektor dari Universitas Sains & Teknologi Komputer (Universitas STEKOM) Semarang yang memiliki banyak pengalaman praktis dalam bidang *e-commerce*sejak Tahun 2002. Beliau mempunyai 3 (tiga) toko *Official Online Store* di China untuk merek Sepeda Raleigh, dengan omzet tahunan pada Tahun 2019 mencapai lebih dari Rp. 35 Milyar rupiah dan terus meningkat. Dr. Joseph T.S memiliki lisensi tunggal sepeda merek “Raleigh” untuk penjualan *Online* di seluruh China. Di samping itu beliau juga memiliki pabrik sepeda dan sepeda listrik merek “Fengjiu”, yaitu Pabrik Sepeda Listrik yang masih tergolong kecil di China. Pengalaman beliau malang melintang di dunia *online store* di China seperti Alibaba, Tmall, Taobao, JD, Aliexpress sangat membantu mahasiswa untuk memiliki pengalaman teknis dan praktis untuk membuka toko *online* bersama beliau.

YAYASAN PRIMA AGUS TEKNIK

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

ISBN 978-623-5734-17-0 (jil.2 PDF)

STRUKTUR DATA dan ALGORITMA (Bagian 2)

Dr. Joseph Teguh Santoso, S.Kom, M.Kom

YAYASAN PRIMA AGUS TEKNIK

PENERBIT :
YAYASAN PRIMA AGUS TEKNIK
Jl. Majapahit No. 605 Semarang
Telp. (024) 6723456. Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

STRUKTUR DATA dan ALGORITMA (Bagian 2)

Penulis :
Dr. Joseph Teguh Santoso, S.Kom., M.Kom

ISBN : 9 786235 734170

Editor :
Muhammad Sholikan, M.Kom

Penyunting :
Dr. Mars Caroline Wibowo. S.T., M.Mm.Tech

Desain Sampul dan Tata Letak :
Irdha Yunianto, S.Ds., M.Kom

Penebit :
Yayasan Prima Agus Teknik Bekerja sama dengan
Universitas Sains & Teknologi Komputer (Universitas STEKOM)

Redaksi :
Jl. Majapahit no 605 Semarang
Telp. (024) 6723456
Fax. 024-6710144
Email : penerbit_ypat@stekom.ac.id

Distributor Tunggal :
Universitas STEKOM
Jl. Majapahit no 605 Semarang
Telp. (024) 6723456
Fax. 024-6710144
Email : info@stekom.ac.id

KATA PENGANTAR

Puji syukur pada Tuhan Yang Maha Esa bahwa buku yang berjudul "*Struktur Data dan Algoritma (Bagian 2)*" ini dapat diselesaikan dengan baik. Dalam pemrograman atau aplikasi sebuah komputer, hal penting untuk dipahami ialah bagaimana logika kita dalam mengolah cara pikir untuk mendapatkan solusi, inovasi, dan bahkan untuk menyelesaikan masalah pemrograman yang akan dibangun secara kompleks dan berurutan. Struktur data adalah cara mengatur, menyimpan, maupun mengelola data di media penyimpanan (Warehouse) sehingga data dapat dimanfaatkan secara efisien. Dalam teknik pemrograman, struktur data bisa diartikan tata letak data yang berisi kolom data, baik kolom data yang hanya digunakan untuk tujuan pemrograman maupun kolom data yang ditampilkan oleh aplikasi sehingga dapat digunakan pengguna.

Dalam buku ini dijelaskan bahwa Struktur data meliputi Data Sederhana dan Data Majemuk. Data Majemuk terdiri dari Data Majemuk Linier dan Non Linier. Yang masing-masing struktur data akan dijelaskan mendetail dalam buku ini.

Definisi Algoritma menurut buku ini adalah Runtutan pengambilan putusan secara logis untuk pemecahan masalah. Tiap bab dalam buku ini membahas detail tentang struktur data secara lengkap, karena dilengkapi dengan contoh soal, kasus dan penyelesaian.

Belajar bahasa pemrograman sangat penting, kita dituntut untuk bisa memahami dan mengerti jenis-jenis data yang akan dipergunakan dalam membangun sebuah Program atau Aplikasi, jadi buku ini sangat cocok untuk pembaca yang ingin belajar tentang database.

Algoritma atau dengan kata lain Logic, merupakan komponen dasar dalam pembuatan sebuah Aplikasi Program. Ciri-ciri algoritma yang baik meliputi: *Input, Output, Definite, Efective* dan *Terminate.* Maksudnya adalah sebuah Algoritma harus memiliki Masukan dan Hasil, Juga memiliki kejelasan apa yang dilakukan, serta langkah penyelesaian yang efektif dan langkah tersebut dapat berhenti atau dapat diberhentikan secara jelas. Semoga buku ini dapat memberi manfaat yang besar pada para pembacanya.

Semarang, Desember 2021
Penulis

Dr. Joseph Teguh Santoso, S.Kom, M.Kom

DAFTAR ISI

BAB 1
ALGORITMA GRAFIK

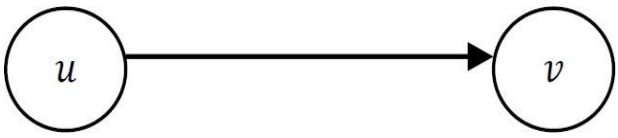
1.1 PENDAHULUAN

Di dunia nyata, banyak masalah direpresentasikan dalam bentuk objek dan hubungan di antara mereka. Misalnya, di peta rute maskapai, kita mungkin tertarik dengan pertanyaan seperti: “Apa cara tercepat untuk pergi dari Hyderabad ke New York?” atau “Apa cara termurah untuk pergi dari Hyderabad ke New York?” Untuk menjawab pertanyaan-pertanyaan ini kita memerlukan informasi tentang koneksi (rute maskapai penerbangan) antar objek (kota). Grafik adalah struktur data yang digunakan untuk memecahkan masalah semacam ini.

1.2 GLOSARIUM

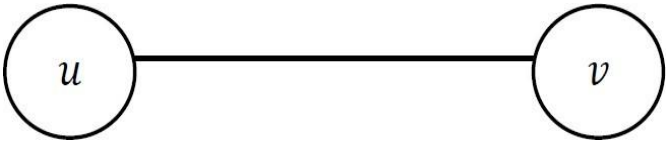
Grafik: Grafik adalah pasangan (V, E), di mana V adalah himpunan simpul, yang disebut simpul, dan £ adalah kumpulan pasangan simpul, yang disebut tepi.

- Vertikal dan tepi adalah posisi dan menyimpan elemen
- Definisi yang kita gunakan:
    - Tepi terarah:
        - pasangan terurut dari simpul (u, v)
        - simpul pertama u adalah asal
        - simpul kedua v adalah tujuan
        - Contoh: lalu lintas jalan satu arah



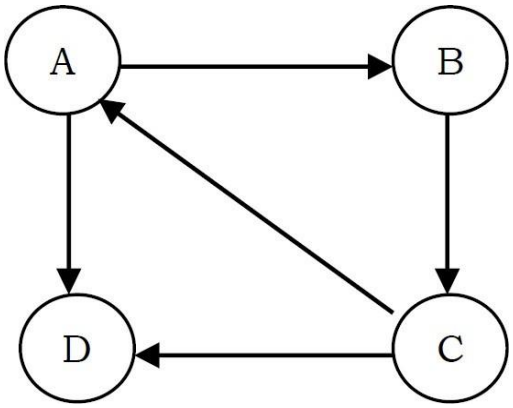
Gambar 1.1 tepi terarah

    - Tepi tidak terarah:
        - pasangan simpul yang tidak berurutan (u, v)
        - Contoh: jalur kereta api



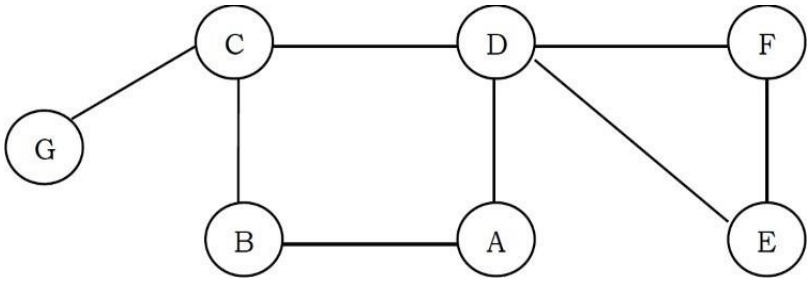
Gambar 1.2 tepi tidak terarah

    - Grafik terarah:
        - semua tepi diarahkan
        - Contoh: jaringan rute



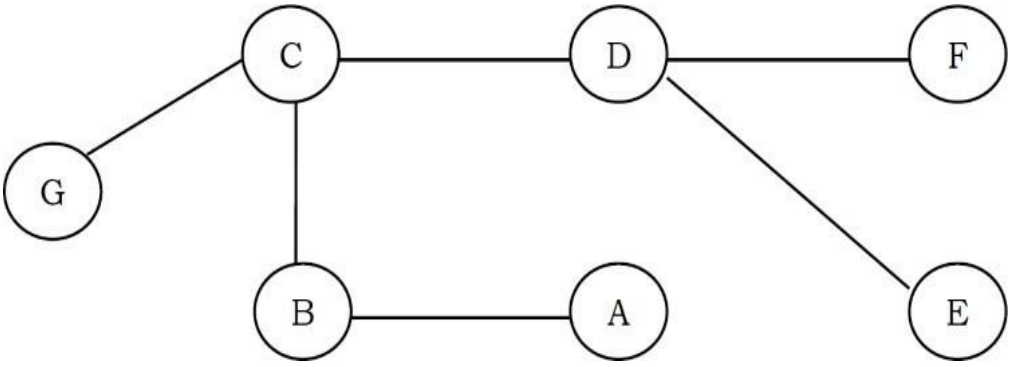
Gambar 1.3 Grafik terarah

- Grafik tak berarah:
  - semua ujungnya tidak terarah
  - Contoh: jaringan penerbangan



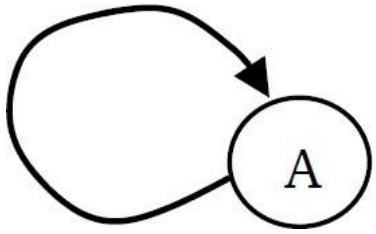
Gambar 1.4 Grafik tak berarah

- Ketika sebuah edge menghubungkan dua vertex, vertex tersebut dikatakan saling bertetangga dan edge tersebut bersisian pada kedua vertex tersebut.
- Grafik tanpa siklus disebut pohon. Pohon adalah grafik terhubung asiklik.



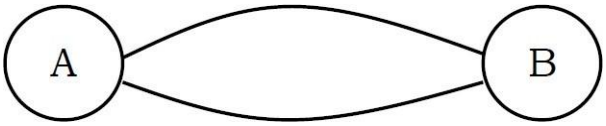
Gambar 1.5 Vertek saling bertetangga

- Self loop adalah edge yang menghubungkan vertex dengan dirinya sendiri.



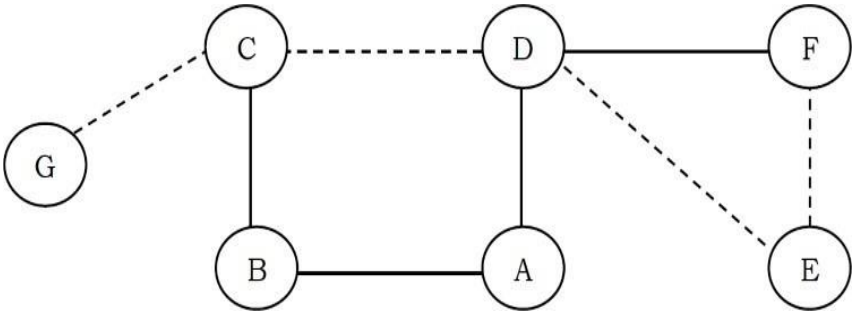
Gambar 1.6 self loop

- Dua sisi sejajar jika menghubungkan pasangan simpul yang sama.

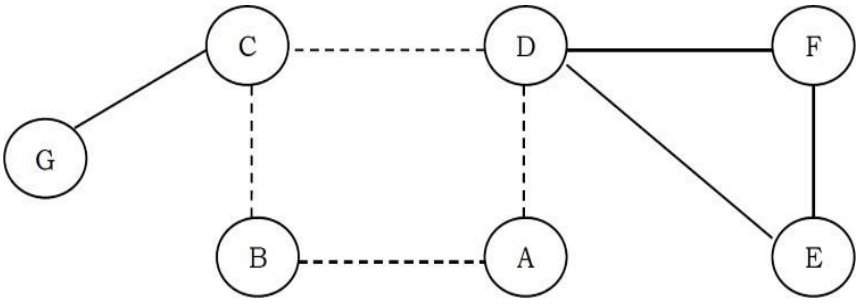


Gambar 1.7 dua sisi sejajar pasangan simpul sama

- Derajat suatu simpul adalah jumlah sisi yang bersisian pada simpul tersebut.
- Subgrafik adalah himpunan bagian dari sisi grafik (dengan simpul terkait) yang membentuk grafik.
- Lintasan dalam suatu grafik adalah barisan simpul-simpul yang bertetangga. Lintasan sederhana adalah lintasan yang tidak memiliki simpul berulang. Pada Grafik di bawah ini, garis putus-putus mewakili jalur dari G ke E.



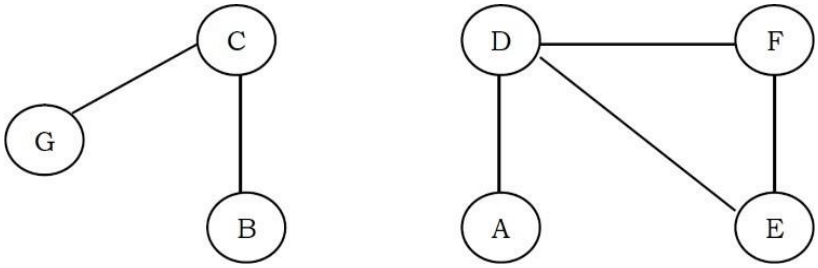
Gambar 1.8 barisan simpul bertetangga



Gambar 1.9 siklus sederhana

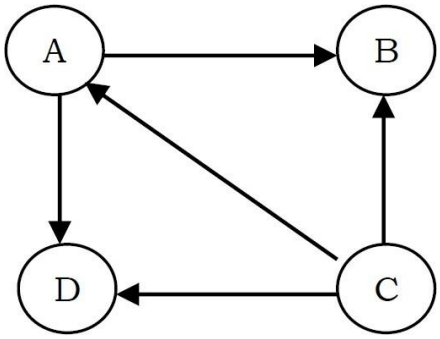
Siklus adalah lintasan yang simpul pertama dan terakhirnya sama. Siklus sederhana adalah siklus tanpa simpul atau tepi yang berulang (kecuali simpul pertama dan terakhir).

- Kita mengatakan bahwa satu simpul terhubung ke simpul lain jika ada jalur yang memuat keduanya.
- Suatu grafik terhubung jika terdapat lintasan dari setiap simpul ke setiap simpul lainnya.
- Jika suatu grafik tidak terhubung maka terdiri dari sekumpulan komponen yang terhubung.



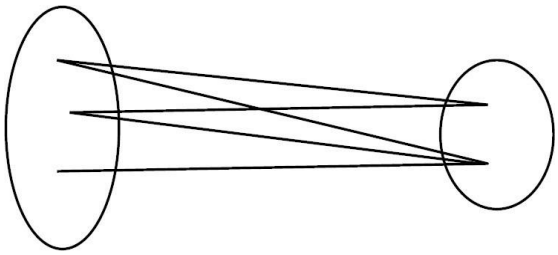
Gambar 1.10 sekumpulan komponen terhubung

- Grafik asiklik berarah [DAG] adalah grafik berarah tanpa siklus.



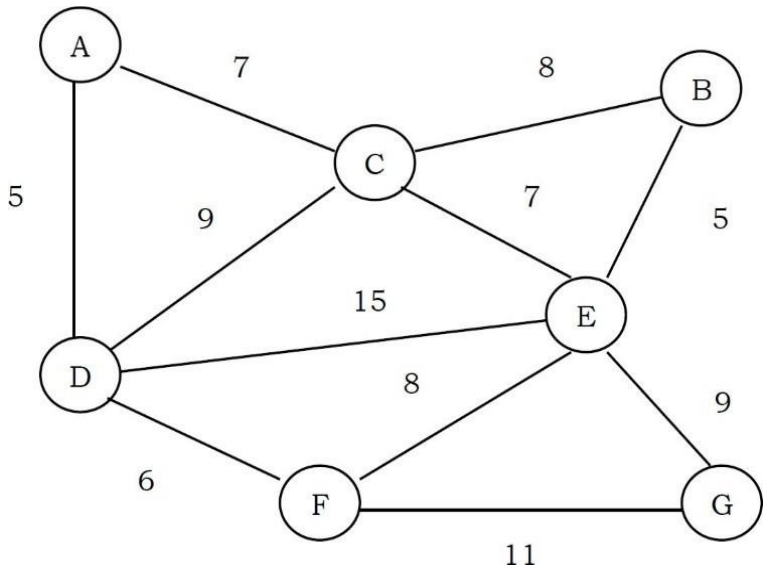
Gambar 1.11 grafik asiklik berarah

- Hutan adalah kumpulan pohon yang terpisah-pisah.
- Pohon merentang dari grafik terhubung adalah subgrafik yang berisi semua simpul grafik tersebut dan merupakan pohon tunggal. Hutan merentang dari suatu grafik adalah penyatuan pohon merentang dari komponen-komponennya yang terhubung.
- Grafik bipartit adalah grafik yang simpul-simpulnya dapat dibagi menjadi dua himpunan sehingga semua sisinya menghubungkan sebuah simpul dalam satu himpunan dengan satu simpul pada himpunan lainnya.



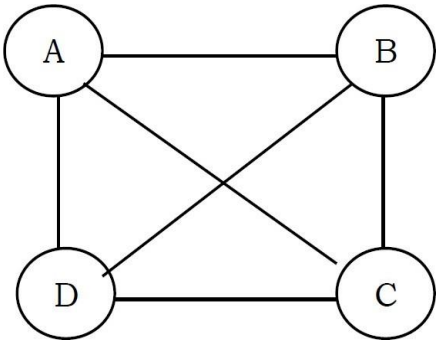
Gambar 1.12 grafik bipartit

- Dalam grafik berbobot, bilangan bulat (bobot) ditetapkan ke setiap sisi untuk direpresentasikan (jarak atau biaya).



Gambar 1.13 grafik berbobot

- Grafik yang semua sisinya ada disebut grafik lengkap.



Gambar 1.14 grafik lengkap

- Grafik dengan sisi yang relatif sedikit (umumnya jika sisinya < |V| log |V|) disebut

- Grafik jarang.
- Grafik dengan kemungkinan sisi yang hilang relatif sedikit disebut densitas.
- Grafik berbobot terarah kadang-kadang disebut jaringan.
- Kita akan menyatakan jumlah simpul dalam grafik tertentu dengan |V|, dan jumlah sisi dengan |E|. Perhatikan bahwa E dapat berkisar dari 0 hingga |V|(|V| – l)/2 (dalam Grafik tak berarah). Hal ini karena setiap node dapat terhubung ke setiap node lainnya.

1.3 APLIKASI GRAFIK

- Mewakili hubungan antar komponen dalam rangkaian elektronika
- Jaringan transportasi: Jaringan jalan raya, Jaringan penerbangan
- Jaringan komputer: Jaringan area lokal, Internet, Web
- Database: Untuk merepresentasikan diagram ER (Entity Relationship) dalam database, untuk merepresentasikan ketergantungan tabel dalam database

1.4 REPRESENTASI GRAFIK

Seperti pada ADT lainnya, untuk memanipulasi Grafik kita perlu merepresentasikannya dalam beberapa bentuk yang berguna. Pada dasarnya, ada tiga cara untuk melakukan ini:

- Matriks Kedekatan
- Daftar Kedekatan
- Set Kedekatan

Matriks Kedekatan

Deklarasi Grafik untuk Matriks Ketetanggaan

Pertama, mari kita lihat komponen struktur data grafik. Untuk merepresentasikan grafik, kita membutuhkan jumlah simpul, jumlah sisi dan juga interkoneksinya. Jadi, Grafik dapat dideklarasikan sebagai:

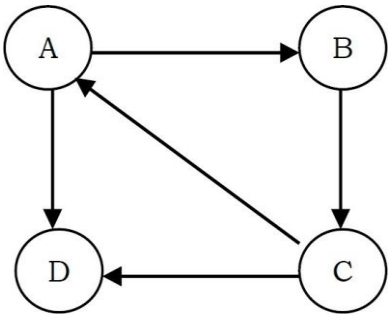
```
struct Graph {
  int V;
  int E;
  int **Adj; //Since we need two dimensional matrix
};
```

Keterangan

Dalam metode ini, kita menggunakan matriks dengan ukuran V × V. Nilai matriks adalah boolean. Mari kita asumsikan matriksnya adalah Adj. Nilai Adj[u, v] diset ke 1 jika ada edge dari vertex u ke vertex v dan 0 sebaliknya.

Dalam matriks, setiap sisi diwakili oleh dua bit untuk grafik tak berarah. Artinya, sisi dari u ke v diwakili oleh 1 nilai pada Adj[u,v] dan Adj[u,v]. Untuk menghemat waktu, kita hanya dapat memproses setengah dari matriks simetris ini. Juga, kita dapat mengasumsikan bahwa ada "ujung" dari setiap simpul ke dirinya sendiri. Jadi, Adj[u, u] diset ke 1 untuk semua simpul.

Jika grafik tersebut merupakan grafik berarah maka kita hanya perlu menandai satu entri dalam matriks ketetanggaan. Sebagai contoh, perhatikan Grafik berarah di bawah ini.



Gambar 1.15 grafik berarah

Matriks ketetanggaan untuk grafik ini dapat diberikan sebagai:

Tabel 1.1 Matriks ketetanggaan

	A	B	C	D
A	0	1	0	1
B	0	0	1	0
C	1	0	0	1
D	0	0	0	0

Sekarang, mari kita berkonsentrasi pada implementasinya. Untuk membaca suatu grafik, salah satu caranya adalah dengan terlebih dahulu membaca nama simpulnya kemudian membaca pasangan nama simpul (sisi). Kode di bawah ini membaca Grafik tidak berarah.

```
//This code creates a graph with adj matrix representation
struct Graph *adjMatrixOfGraph() {
    int i, u, v;
    struct Graph *G = (struct Graph *) malloc(sizeof(struct Graph));
    if(!G) {
        printf("Memory Error");
        return;
    }
    scanf("Number of Vertices: %d, Number of Edges:%d", &G→V, &G→E);
    G→Adj = malloc(sizeof(G→V  * G→V));
    for(u = 0; u < G→V; u++)
            for(v = 0; v < G→V; v++)
                    G→Adj[v][v] = 0;
    for(i = 0;  i < G→E; i++) {
        //Read an edge
        scanf("Reading Edge: %d %d", &u, &v);
        //For undirected graphs set both the bits
        G→ Adj[u][v] = 1;
        G→ Adj[v][u] = 1;
    }
    return G;
}
```

Representasi matriks ketetanggaan baik jika grafiknya padat. Matriks membutuhkan O(V2) bit penyimpanan dan O(V2) waktu untuk inisialisasi. Jika jumlah sisi sebanding dengan

V2, maka tidak ada masalah karena langkah V2 diperlukan untuk membaca sisi. Jika Grafik jarang, inisialisasi matriks mendominasi waktu berjalan algoritma karena membutuhkan O(V2).

Daftar Kedekatan

Deklarasi Grafik untuk Adjacency List

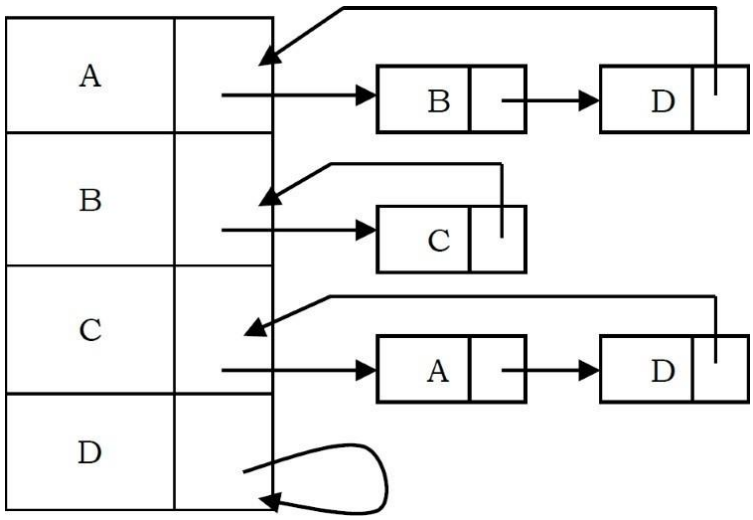
Dalam representasi ini semua simpul yang terhubung ke simpul v terdaftar pada daftar ketetanggaan untuk simpul v itu. Ini dapat dengan mudah diimplementasikan dengan daftar tertaut. Itu berarti, untuk setiap simpul v kita menggunakan daftar tertaut dan simpul daftar mewakili koneksi antara v dan simpul lain di mana v memiliki tepi.

Jumlah total daftar tertaut sama dengan jumlah simpul dalam Grafik. Grafik ADT dapat dideklarasikan sebagai:

```
struct Graph {
    int V;
    int E;
    int *Adj; //head pointers to linked list
};
```

Keterangan

Mempertimbangkan contoh yang sama dengan matriks ketetanggaan, representasi daftar ketetanggaan dapat diberikan sebagai:



Gambar 1.16 daftar ketetanggaan

Karena simpul A memiliki sisi untuk B dan D, kita telah menambahkannya dalam daftar ketetanggaan untuk A. Hal yang sama juga terjadi pada simpul lainnya.

```
//Nodes of the Linked List
struct ListNode {
    int vertexNumber;
    struct ListNode *next;
}

//This code creates a graph with adj list representation
struct Graph *adjListOfGraph() {
    int i, x, y;
    struct ListNode *temp;
    struct Graph *G = (struct Graph *) malloc(sizeof(struct Graph));
    if(!G) {
        printf("Memory Error");
        return;
    }
    scanf("Number of Vertices: %d, Number of Edges:%d", &G→V, &G→E);
    G→Adj = malloc(G→V * sizeof(struct ListNode));

    for(i = 0; i  < G→V; i++) {
        G→Adj[i] = (struct ListNode *) malloc(sizeof(struct ListNode));
        G→Adj[i]→vertexNumber = i;
        G→Adj[i]→next = G→ Adj[i];
    }
    for(i = 0;  i < E; i++) {
        //Read an edge
        scanf("Reading Edge: %d %d", &x, &y);
        temp = (struct ListNode *) malloc(struct ListNode);
        temp→vertexNumber = y;
        temp→next = G→Adj[x];
        G→ Adj[x]→next = temp;
        temp = (struct ListNode *) malloc(struct ListNode);
        temp→vertexNumber = y;
        temp→next = G→ Adj[y];
        G→Adj[y]→ next= temp;
    }

    retutn G;
}
```

Untuk representasi ini, urutan tepi dalam input penting. Ini karena mereka menentukan urutan simpul pada daftar ketetanggaan. Grafik yang sama dapat direpresentasikan dalam banyak cara berbeda dalam daftar ketetanggaan. Urutan munculnya edge pada daftar adjacency mempengaruhi urutan edge yang diproses oleh Algoritma.

Kekurangan dari *Adjacency Lists*

Menggunakan representasi daftar adjacency, kita tidak dapat melakukan beberapa operasi secara efisien. Sebagai contoh, pertimbangkan kasus menghapus sebuah node. . Dalam representasi daftar adjacency, tidak cukup jika kita hanya menghapus sebuah node dari representasi daftar, jika kita menghapus sebuah node dari daftar adjacency maka itu sudah cukup. Untuk setiap simpul pada daftar adjacency dari simpul tersebut menentukan simpul lain. Kita perlu mencari node lain yang terhubung list juga untuk menghapusnya. Masalah ini dapat diselesaikan dengan menghubungkan dua list node yang berhubungan dengan edge tertentu dan membuat daftar adjacency tertaut ganda. Tetapi semua tautan tambahan ini berisiko untuk diproses.

Set Kedekatan

Ini sangat mirip dengan daftar adjacency tetapi alih-alih menggunakan daftar Tertaut, *Set Disjoint [Union-Find]* digunakan. Untuk detail lebih lanjut, lihat bab ADT Set Terpisah.

Perbandingan Representasi Grafik

Grafik berarah dan tidak berarah direpresentasikan dengan struktur yang sama. Untuk grafik berarah, semuanya sama, kecuali bahwa setiap sisi hanya diwakili satu kali. Sisi dari x ke y diwakili oleh nilai 1 pada Agj[x][y] dalam matriks ketetanggaan, atau dengan menambahkan y pada daftar ketetanggaan x. Untuk grafik berbobot, semuanya sama, kecuali mengisi matriks ketetanggaan dengan bobot alih-alih nilai boolean.

Tabel 1.2 perbandingan representasi grafik

Perwakilan	Jarak	Memeriksa tepi antara v dan w?	Iterasi insiden tepi ke v?
Daftar tepi	E	E	E
Matriks Adj	V^2	1	V
Daftar Penyesuaian	E + V	Degree(v)	Degree(v)
Set Penyesuaian	E + V	log(Degree(v))	Degree(v)

1.5 GRAFIK TRAVERSAL

Untuk menyelesaikan masalah pada grafik, diperlukan suatu mekanisme untuk melintasi grafik tersebut. Algoritma traversal grafik juga disebut algoritma pencarian grafik. Seperti algoritma traversal pohon (Inorder, Preorder, Postorder, dan Level-Order traversal), algoritma pencarian grafik dapat dianggap dimulai dari beberapa simpul sumber dalam grafik dan "mencari" grafik dengan menelusuri tepi dan menandai simpul. Sekarang, kita akan membahas dua algoritma tersebut untuk melintasi Grafik.

- Pencarian Pertama Kedalaman [DFS]
- Pencarian Pertama yang Luas [BFS]

Pencarian Pertama Kedalaman [DFS]

Algoritma DFS bekerja dengan cara yang mirip dengan praorder traversal pohon. Seperti halnya traversal preorder, secara internal algoritma ini juga menggunakan stack.

Mari kita perhatikan contoh berikut. Misalkan seseorang terjebak di dalam labirin. Untuk keluar dari labirin itu, orang tersebut mengunjungi setiap jalan dan setiap persimpangan (dalam kasus terburuk). Katakanlah orang tersebut menggunakan dua warna cat untuk menandai persimpangan yang sudah dilewati. Saat menemukan persimpangan baru, itu ditandai abu-abu, dan dia terus masuk lebih dalam.

Setelah mencapai "jalan buntu" orang tersebut tahu bahwa tidak ada lagi jalan yang belum dijelajahi dari persimpangan abu-abu, yang sekarang sudah selesai, dan dia

menandainya dengan warna hitam. Jalan buntu ini bisa berupa persimpangan yang sudah ditandai abu-abu atau hitam, atau sekadar jalan yang tidak mengarah ke persimpangan.

Persimpangan labirin adalah simpul dan jalur antara persimpangan adalah tepi Grafik. Proses kembali dari "jalan buntu" disebut backtracking. Kita mencoba untuk pergi dari titik awal ke dalam Grafik sedalam mungkin, sampai kita harus mundur ke titik abu-abu sebelumnya. Dalam algoritma DFS, kita menemukan jenis tepi berikut.

Tabel 1.3 jenis tepi

Tepi pohon: temukan simpul baru
Tepi belakang: dari keturunan ke leluhur
Tepi depan: dari leluhur ke keturunan
Tepi silang: antara pohon atau subpohon

Untuk kebanyakan algoritma klasifikasi boolean, unvisited/visited sudah cukup (untuk implementasi tiga warna lihat bagian masalah). Artinya, untuk beberapa masalah kita perlu menggunakan tiga warna, tetapi untuk pembahasan kita dua warna sudah cukup.

Salah ⟶ **Vertex belum dikunjungi**
Benar ⟶ **Vertex dikunjungi**

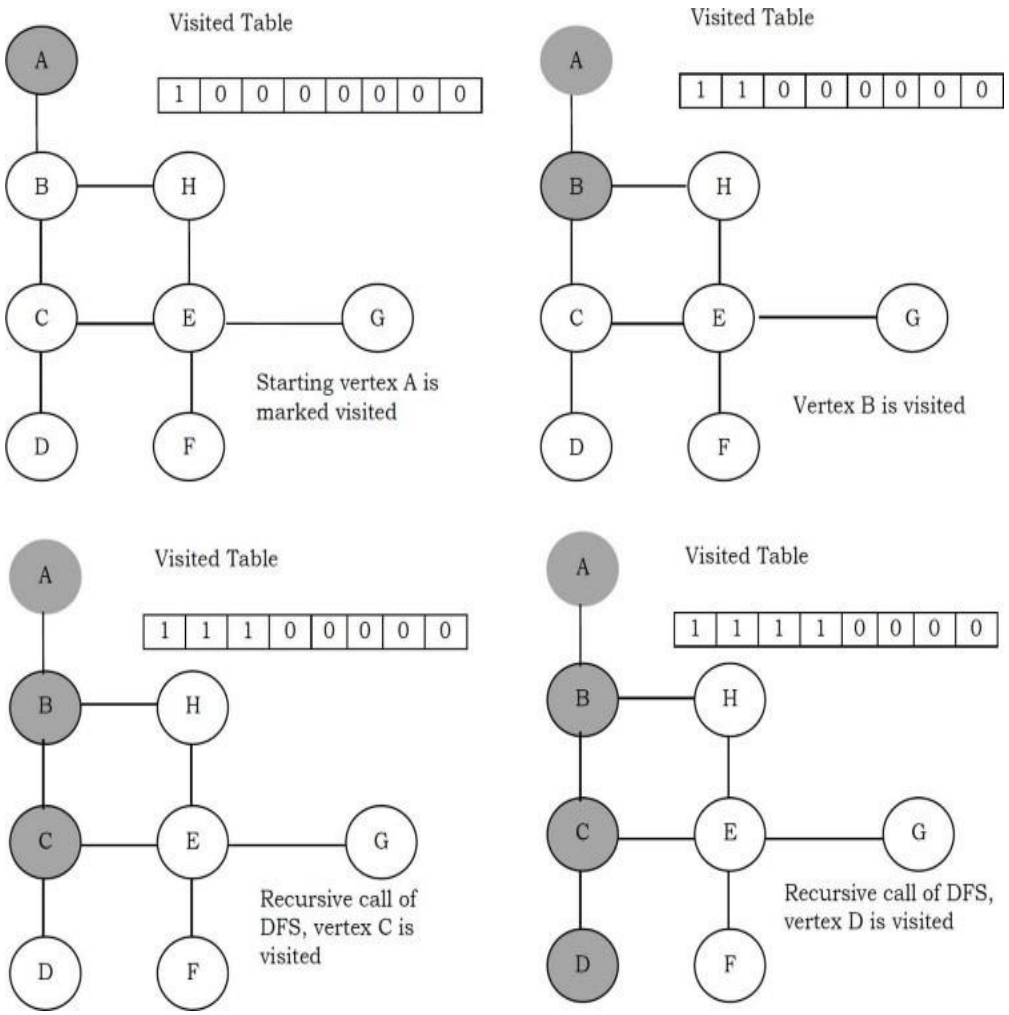
Awalnya semua simpul ditandai belum dikunjungi (salah). Algoritma DFS dimulai pada titik u dalam Grafik. Dengan memulai dari simpul u itu mempertimbangkan tepi dari u ke simpul lainnya. Jika tepi mengarah ke simpul yang sudah dikunjungi, maka lacak kembali ke simpul u saat ini. Jika sebuah edge mengarah ke sebuah vertex yang belum dikunjungi, maka pergilah ke vertex tersebut dan mulailah memproses dari vertex tersebut. Itu berarti simpul baru menjadi simpul saat ini. Ikuti proses ini sampai kita mencapai jalan buntu. Pada titik ini mulai mundur.

Proses berakhir ketika backtracking mengarah kembali ke titik awal. Algoritma berdasarkan mekanisme ini diberikan di bawah ini: anggap Visited[] adalah array global.

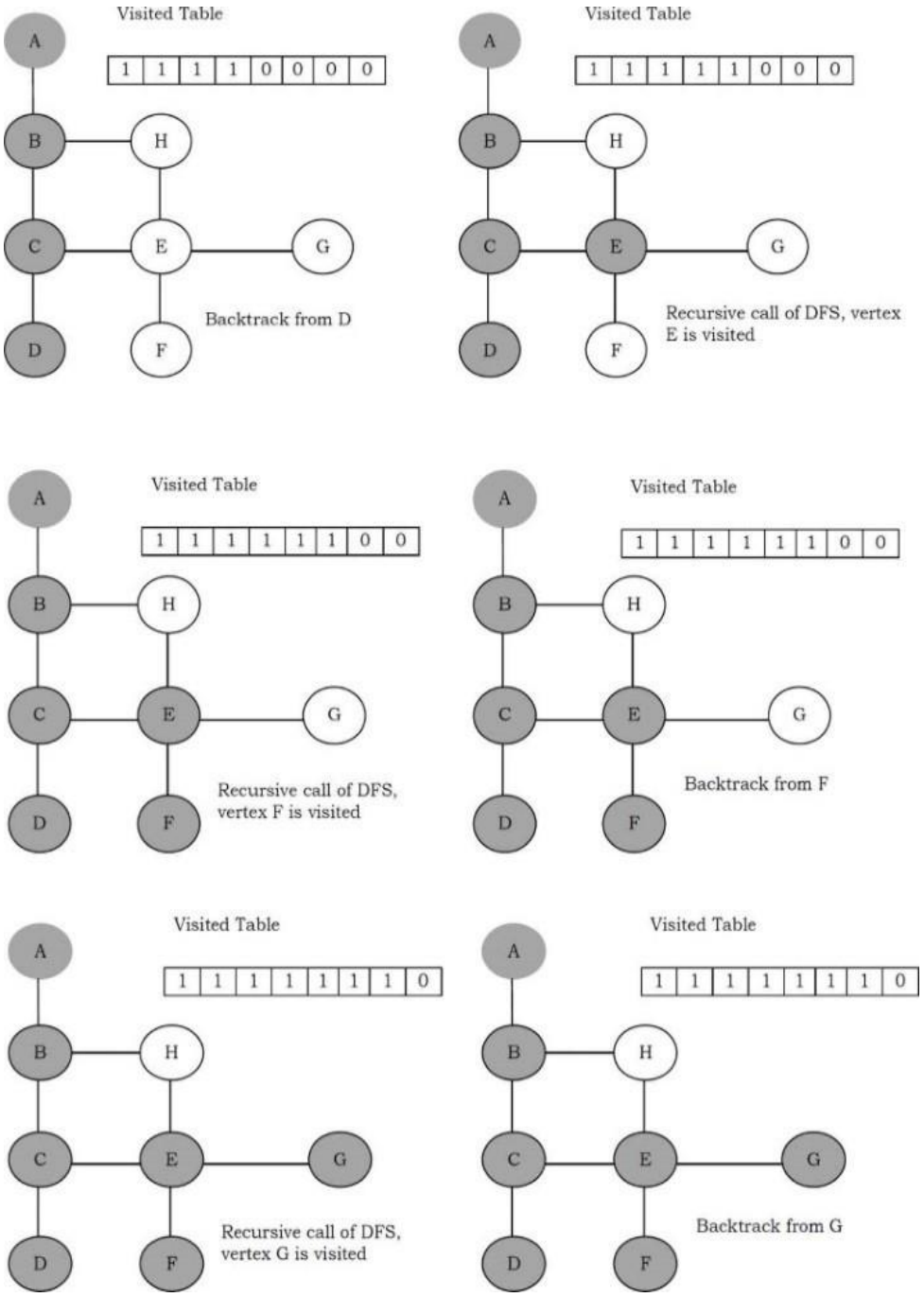
Sebagai contoh perhatikan Grafik berikut. Kita dapat melihat bahwa terkadang sebuah edge mengarah ke vertex yang sudah ditemukan. Tepi ini disebut tepi belakang, dan tepi lainnya disebut tepi pohon karena menghapus tepi belakang dari Grafik menghasilkan pohon.

Pohon yang dihasilkan akhir disebut pohon DFS dan urutan di mana simpul diproses disebut nomor DFS dari simpul. Pada Grafik di bawah ini, warna abu-abu menunjukkan bahwa simpul dikunjungi (tidak ada signifikansi lain). Kita perlu melihat kapan tabel Visited diperbarui.

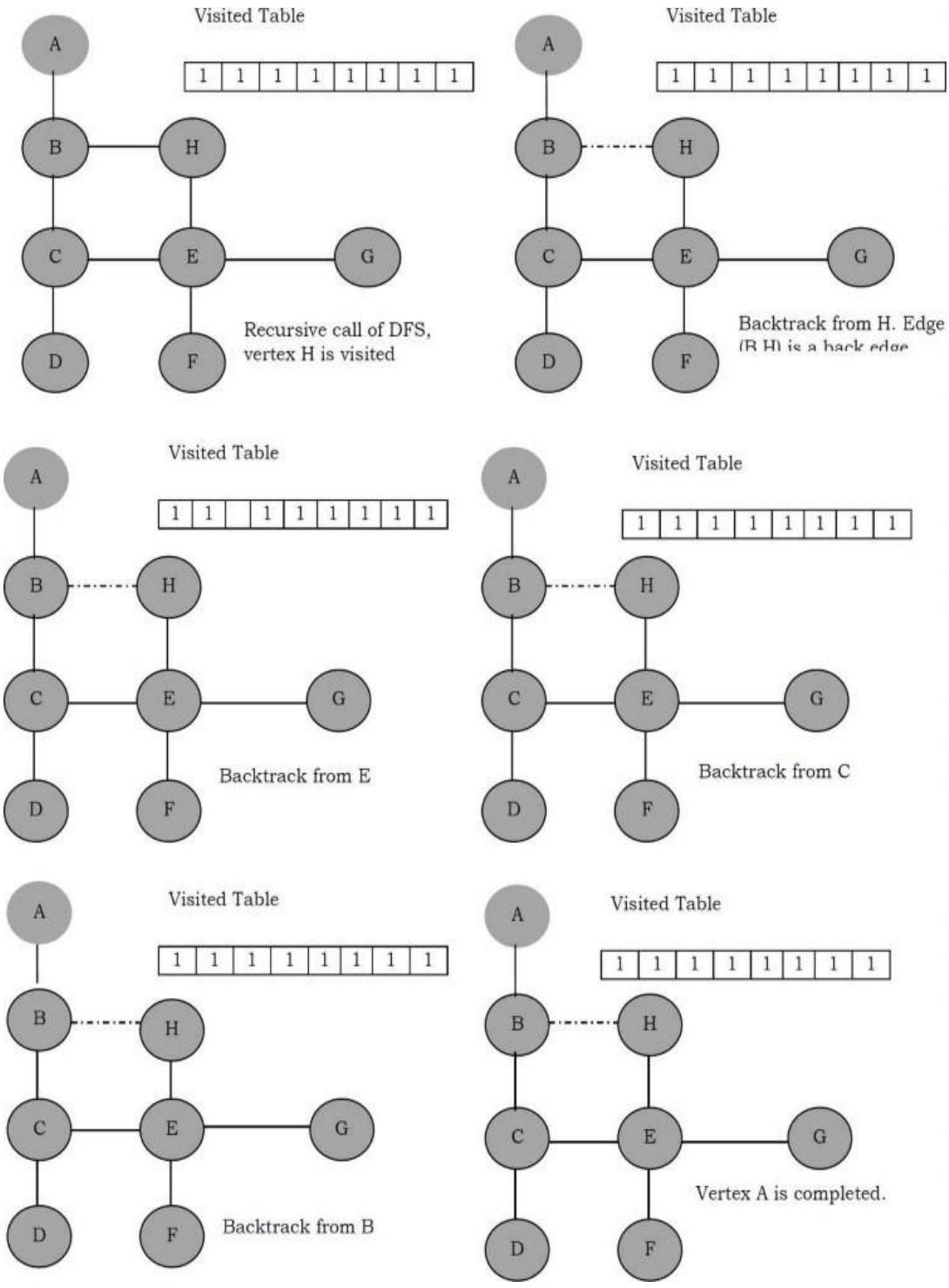
```
int Visited[G→V];
void DFS(struct Graph *G, int u) {
    Visited[u] = 1;
    for( int v = 0; v < G→V; v++ ) {

        /* For example, if the adjacency matrix is used for representing the
           graph, then the condition to be used for finding unvisited adjacent
           vertex of u  is: if( !Visited[v] && G→Adj[u][v] ) */

        for each unvisited adjacent node v of u {
            DFS(G, v);
        }
    }
}
void DFSTraversal(struct Graph *G) {
    for (int i = 0; i< G→V;i++)
        Visited[i]=0;

    //This loop is required if the graph has more than one component
    for (int i = 0; i< G→V;i++)
        if(!Visited[i])
            DFS(G, i);
}
```



Gambar 1.17 Contoh grafik



Gambar 1.18 Contoh grafik (2)



Gambar 1.19 contoh grafik

Dari diagram di atas, terlihat bahwa traversal DFS membuat pohon (tanpa tepi belakang) dan kita menyebut pohon tersebut sebagai pohon DFS. Algoritma di atas bekerja bahkan jika Grafik yang diberikan memiliki komponen yang terhubung.

Kompleksitas waktu DFS adalah O(V + E), jika kita menggunakan daftar ketetanggaan untuk merepresentasikan Grafik. Ini karena kita memulai dari sebuah simpul dan memproses simpul-simpul yang berdekatan hanya jika simpul-simpul tersebut tidak dikunjungi. Demikian pula, jika matriks ketetanggaan digunakan untuk representasi grafik, maka semua tepi yang berdekatan dengan simpul tidak dapat ditemukan secara efisien, dan ini memberikan kompleksitas O(V2).

Aplikasi DFS

- Penyortiran topologi
- Menemukan komponen yang terhubung
- Menemukan titik artikulasi (titik potong) dari Grafik
- Menemukan komponen yang terhubung kuat
- Memecahkan teka-teki seperti labirin

Untuk algoritma lihat Bagian Masalah.

Pencarian Pertama yang Luas [BFS]

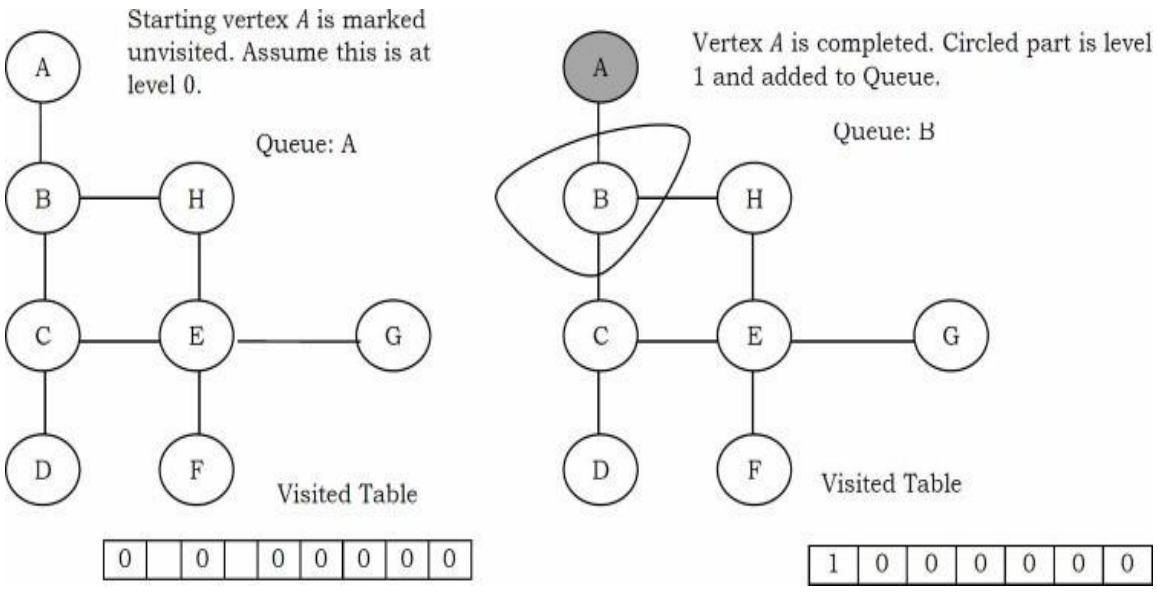
Algoritma BFS bekerja mirip dengan level – order traversal dari pohon. Seperti level – order traversal, BFS juga menggunakan antrian. Faktanya, level – order traversal terinspirasi dari BFS. BFS bekerja tingkat demi tingkat. Awalnya, BFS dimulai pada simpul tertentu, yaitu pada level 0. Pada tahap pertama, BFS mengunjungi semua simpul pada level 1 (artinya, simpul yang jaraknya 1 dari simpul awal Grafik). Pada tahap kedua, ia mengunjungi semua simpul di tingkat kedua. Simpul-simpul baru ini adalah simpul-simpul yang berdekatan dengan simpul-simpul level 1. BFS melanjutkan proses ini sampai semua level Grafik selesai. Umumnya struktur data antrian digunakan untuk menyimpan simpul dari suatu level.

Seperti halnya DFS, asumsikan bahwa awalnya semua simpul ditandai belum dikunjungi (salah). Vertex yang telah diproses dan dikeluarkan dari antrian ditandai sebagai dikunjungi (true). Kita menggunakan antrian untuk mewakili set yang dikunjungi karena akan menjaga simpul dalam urutan saat mereka pertama kali dikunjungi. Implementasi untuk pembahasan di atas dapat diberikan sebagai:

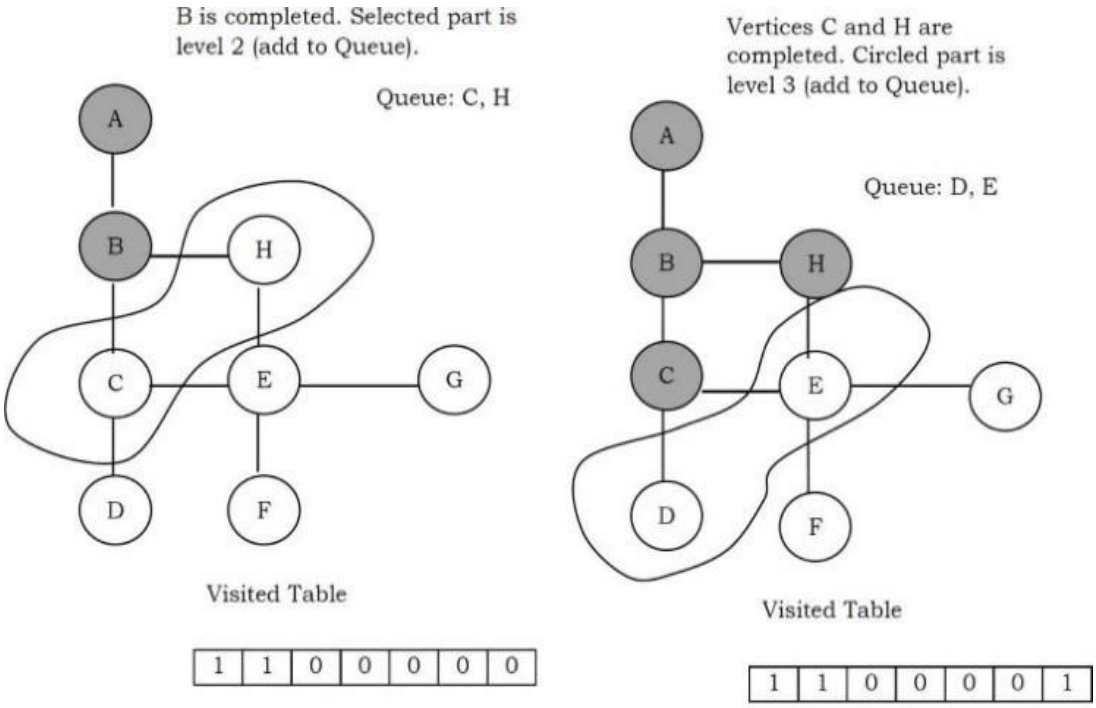
```
void BFS(struct Graph *G, int u) {
   int v;
   struct Queue *Q = CreateQueue();
   EnQueue(Q, u);
   while(!IsEmptyQueue(Q)) {
      u = DeQueue(Q);
      Process u; //For example, print
      Visited[s]=1;
      /* For example, if the adjacency matrix is used for representing the  graph,
```

```
        then the condition be used for finding unvisited adjacent vertex of u  is:
        if( !Visited[v] && G→Adj[u][v] ) */
        for each unvisited adjacent node v of u {
            EnQueue(Q, v);
        }
    }
}
void BFSTraversal(struct Graph *G) {
    for (int i = 0; i< G→V;i++)
        Visited[i]=0;
    //This loop is required if the graph has more than one component
    for (int i = 0; i< G→V;i++)
        if(!Visited[i])
            BFS(G. i):
}
```

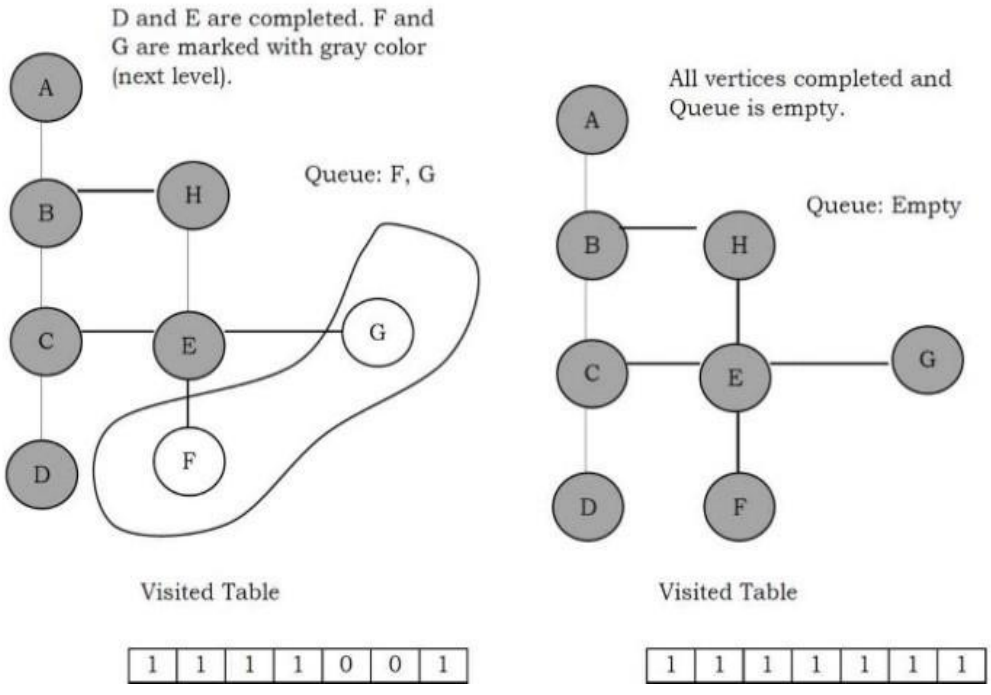
Sebagai contoh, mari kita perhatikan Grafik yang sama dengan contoh DFS. Traversal BFS dapat ditunjukkan sebagai:



Gambar 1.20 contoh traversal BFS



Gambar 1.21 memilih bagian level 2



Gambar 1.22 F dan G ditandai dengan warna abu – abu

Kompleksitas waktu BFS adalah O(V + E), jika kita menggunakan daftar ketetanggaan untuk merepresentasikan Grafik, dan $O(V^2)$ untuk representasi matriks ketetanggaan.

Aplikasi BFS

- Menemukan semua komponen yang terhubung dalam Grafik
- Menemukan semua node dalam satu komponen yang terhubung
- Menemukan jalur terpendek antara dua node
- Menguji Grafik bipartit

Membandingkan DFS dan BFS

Membandingkan BFS dan DFS, keuntungan besar DFS adalah bahwa ia memiliki persyaratan memori yang jauh lebih rendah daripada BFS karena tidak diperlukan untuk menyimpan semua pointer anak di setiap level. Tergantung pada data dan apa yang kita cari, baik DFS atau BFS dapat menguntungkan. Misalnya, dalam silsilah keluarga jika kita mencari seseorang yang masih hidup dan jika kita menganggap orang itu berada di bawah pohon, maka DFS adalah pilihan yang lebih baik. BFS akan membutuhkan waktu yang sangat lama untuk mencapai level terakhir itu.

Algoritma DFS menemukan tujuan lebih cepat. Sekarang, jika kita mencari anggota keluarga yang sudah lama meninggal, maka orang itu akan lebih dekat ke puncak pohon. Dalam hal ini, BFS menemukan lebih cepat daripada DFS. Jadi, kelebihan keduanya berbeda-beda tergantung data dan apa yang kita cari.

DFS terkait dengan traversal preorder pohon. Seperti traversal preorder, DFS mengunjungi setiap node sebelum anak-anaknya. Algoritma BFS bekerja mirip dengan level – order traversal dari pohon.

Jika seseorang bertanya apakah DFS lebih baik atau BFS lebih baik, jawabannya tergantung pada jenis masalah yang kita coba selesaikan. BFS mengunjungi setiap level satu per satu, dan jika kita tahu solusi yang kita cari ada di kedalaman rendah, maka BFS bagus. DFS adalah pilihan yang lebih baik jika solusinya berada pada kedalaman maksimum. Tabel di bawah ini menunjukkan perbedaan antara DFS dan BFS dalam hal aplikasinya.

Tabel 1.4 perbedaan DFS dan BFS

Aplikasi	DFS	BFS
Mencakup hutan, komponen terhubung, jalur, siklus	Ya	Ya
Jalur terpendek		Ya
Penggunaan ruang memori minimal	Ya	

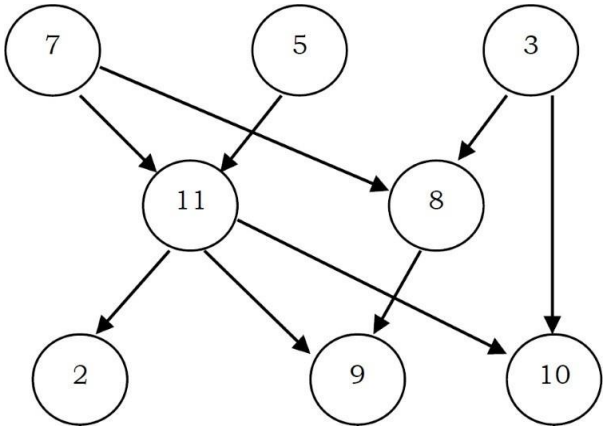
1.6 PENGURUTAN TOPOLOGIS

Pengurutan topologi adalah pengurutan simpul dalam grafik asiklik berarah [DAG] di mana setiap simpul datang sebelum semua simpul yang memiliki tepi keluar. Sebagai contoh, perhatikan struktur prasyarat mata kuliah di universitas. Tepi berarah (v,w) menunjukkan bahwa lintasan v harus diselesaikan sebelum lintasan w. Urutan topologi untuk contoh ini adalah urutan yang tidak melanggar persyaratan prasyarat. Setiap DAG mungkin memiliki satu

atau lebih urutan topologi. Pengurutan topologi tidak dimungkinkan jika grafik memiliki siklus, karena untuk dua simpul v dan w pada siklus, v mendahului w dan w mendahului v.

Jenis topologi memiliki sifat yang menarik. Semua pasangan simpul berurutan dalam urutan terurut dihubungkan oleh tepi; kemudian tepi-tepi ini membentuk jalur Hamiltonian berarah [lihat Bagian Masalah] di DAG. Jika jalur Hamiltonian ada, urutan pengurutan topologi adalah unik. Jika jenis topologi tidak membentuk jalur Hamilton, DAG dapat memiliki dua atau lebih urutan topologi. Pada Grafik di bawah ini: 7, 5, 3, 11, 8, 2, 9, 10 dan 3, 5, 7, 8, 11, 2, 9, 10 keduanya merupakan urutan topologi.

Awalnya, indegree dihitung untuk semua simpul, dimulai dengan simpul yang memiliki indegree 0. Itu berarti pertimbangkan simpul yang tidak memiliki prasyarat. Untuk melacak simpul dengan indegree nol kita dapat menggunakan antrian.



Gambar 1.23 pengurutan topologis

Semua simpul dengan derajat masuk 0 ditempatkan pada antrian. Sementara antrian tidak kosong, sebuah simpul v dihilangkan, dan semua sisi yang berdekatan dengan v mengalami penurunan derajat. Sebuah simpul ditempatkan pada antrian segera setelah derajat masuknya turun ke 0. Urutan topologi adalah urutan di mana simpul DeQueue. Kompleksitas waktu dari algoritma ini adalah O(|E| + |V|) jika daftar ketetanggaan digunakan.

```
void TopologicalSort( struct Graph *G ) {
    struct Queue *Q;
    int counter;
    int v, w;
    Q = CreateQueue();
    counter =  0;
    for (v = 0; v< G→V; v++)
        if( indegree[v] ==   0 )
            EnQueue( Q, v );
    while( !IsEmptyQueue( Q ) ) {
        v = DeQueue( Q );
        topologicalOrder[v] = ++counter;
        for each w adjacent to v
            if( --indegree[w] ==  0 )
                EnQueue ( Q, w );
    }
    if( counter != G→V)
        printf("Graph has cycle");
    DeleteQueue( Q );
}
```

Total waktu berjalan dari pengurutan topologi adalah O(V + E).

Catatan: *Masalah penyortiran Topologi dapat diselesaikan dengan DFS. Lihat Bagian Masalah untuk algoritma.*

Aplikasi Penyortiran Topologi

- Mewakili prasyarat kursus
- Mendeteksi kebuntuan
- Jalur pekerjaan komputasi
- Memeriksa loop tautan simbolik
- Mengevaluasi rumus dalam spreadsheet

1.7 ALGORITMA JALUR TERPENDEK

Mari kita pertimbangkan masalah penting lainnya dari Grafik. Diberikan sebuah grafik G = (V, E) dan sebuah simpul s yang berbeda, kita perlu mencari jalur terpendek dari s ke setiap simpul lain di G. Ada variasi dalam algoritma jalur terpendek yang bergantung pada jenis grafik input dan diberikan di bawah ini.

Variasi Algoritma Jalur Terpendek

Tabel 1.5 variasi algoritma jalur terpendek

Jalur terpendek pada grafik tak berbobot
Jalur terpendek dalam grafik berbobot
Jalur terpendek dalam grafik berbobot dengan sisi negatif

Aplikasi Algoritma Jalur Terpendek

- Menemukan cara tercepat untuk pergi dari satu tempat ke tempat lain

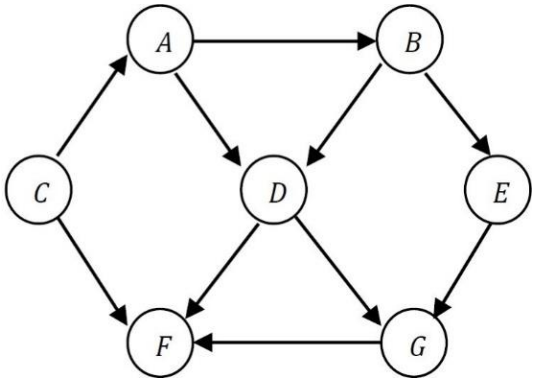
- Menemukan cara termurah untuk terbang/mengirim data dari satu kota ke kota lain

Jalur Terpendek pada Grafik Tak Berbobot

Biarkan s menjadi simpul input dari mana kita ingin menemukan jalur terpendek ke semua simpul lainnya. Grafik tidak berbobot adalah kasus khusus dari masalah jalur terpendek berbobot, dengan semua sisi berbobot 1. Algoritma ini mirip dengan BFS dan kita perlu menggunakan struktur data berikut:

- Tabel jarak dengan tiga kolom (setiap baris sesuai dengan simpul):
  - Jarak dari simpul sumber.
  - Path – berisi nama simpul yang melaluinya kita mendapatkan jarak terpendek.
- Antrian digunakan untuk mengimplementasikan pencarian luas-pertama. Ini berisi simpul yang jaraknya dari simpul sumber telah dihitung dan simpul yang berdekatan akan diperiksa.

Sebagai contoh, perhatikan Grafik berikut dan representasi daftar kedekatannya.



Gambar 1.24 contoh grafik dan representasi daftar kedekatannya

Daftar ketetanggaan untuk Grafik ini adalah:

$$\boldsymbol{A}: B \rightarrow D$$
$$\boldsymbol{B}: D \rightarrow E$$
$$\boldsymbol{C}: A \rightarrow F$$
$$\boldsymbol{D}: F \rightarrow G$$
$$\boldsymbol{E}: G$$
$$\boldsymbol{F}: -$$
$$\boldsymbol{G}: F$$

Misalkan s = C. Jarak dari C ke C adalah 0. Awalnya, jarak ke semua node lain tidak dihitung, dan kita menginisialisasi kolom kedua dalam tabel jarak untuk semua simpul (kecuali C) dengan -1 seperti di bawah ini.

Tabel 1.6 tabel analisis vertex dan jarak

Vertex	Jarak [v]	Titik sebelumnya yang memberikan Jarak [v]
A	-1	-
B	-1	-
C	0	-
D	-1	-
E	-1	-
F	-1	-
G	-1	-

Algoritma

```
void UnweightedShortestPath(struct Graph *G, int s) {
  struct Queue *Q = CreateQueue();
  int v, w;
  EnQueue(Q, s);
  for (int i = 0; i< G→V;i++)
        Distance[i]=-1;
  Distance[s]= 0;
  while (!IsEmptyQueue(Q)) {
        v = DeQueue(Q);
        for each w adjacent to v          ← Each vertex examined at most once
              if(Distance[w] == -1)  {
                    Distance[w] = Distance[v] + 1;
                    Path[w] = v;
                    EnQueue(Q, w);   ← Each vertex EnQueue'd at most once
              }
  }
  DeleteQueue(Q);
}
```

Waktu berjalan: O(|E| + |V|), jika daftar kedekatan digunakan. Dalam perulangan for, kita memeriksa sisi keluar untuk simpul tertentu dan jumlah semua sisi yang diperiksa dalam perulangan while sama dengan jumlah sisi yang menghasilkan O(|E|).

Jika kita menggunakan representasi matriks, kompleksitasnya adalah $O(|V|^2)$, karena kita perlu membaca seluruh baris dalam matriks dengan panjang |V| untuk menemukan simpul yang berdekatan untuk simpul tertentu.

Jalur terpendek dalam Grafik Berbobot [Dijkstra's]

Solusi terkenal untuk masalah jalur terpendek dikembangkan oleh Dijkstra. Algoritma Dijkstra merupakan generalisasi dari algoritma BFS. Algoritma BFS reguler tidak dapat menyelesaikan masalah jalur terpendek karena tidak dapat menjamin bahwa simpul di depan antrian adalah simpul yang paling dekat dengan sumber s.

Sebelum pergi ke kode mari kita memahami bagaimana algoritma bekerja. Seperti pada algoritma jalur terpendek tidak berbobot, di sini juga kita menggunakan tabel jarak. Algoritma bekerja dengan menjaga jarak terpendek dari simpul v dari sumber pada tabel Jarak.

Nilai Distance[v] menahan jarak dari s ke v. Jarak terpendek dari sumber ke dirinya sendiri adalah nol. Tabel Jarak untuk semua simpul lainnya diatur ke -1 untuk menunjukkan bahwa simpul tersebut belum diproses.

Tabel 1.7 tabel analisis vertex dan jarak

Vertex	Jarak [v]	Titik sebelumnya yang memberikan Jarak [v]
A	-1	-
B	-1	-
C	0	-
D	-1	-
E	-1	-
F	-1	-
G	-1	-

Setelah algoritma selesai, tabel Jarak akan memiliki jarak terpendek dari sumber s ke satu sama lain simpul v. Untuk menyederhanakan pemahaman algoritma Dijkstra, mari kita asumsikan bahwa simpul yang diberikan dipertahankan dalam dua set. Awalnya set pertama hanya berisi elemen sumber dan set kedua berisi semua elemen yang tersisa. Setelah iterasi ke-k, himpunan pertama berisi k simpul yang paling dekat dengan sumber. K simpul ini adalah simpul yang telah kita hitung jarak terpendeknya dari sumber.

Catatan tentang Algoritma Dijkstra

- Menggunakan metode tamak: Selalu pilih simpul terdekat berikutnya dengan sumber.
- Menggunakan antrian prioritas untuk menyimpan simpul yang belum dikunjungi berdasarkan jarak dari s.
- Tidak bekerja dengan bobot negatif.

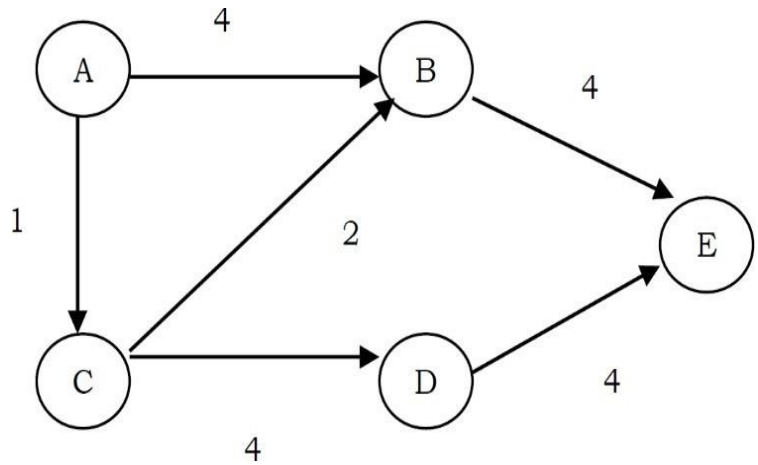
Perbedaan antara Unweighted Shortest Path dan Algoritma Dijkstra

1) Untuk merepresentasikan bobot dalam daftar adjacency, setiap simpul berisi bobot dari sisi-sisinya (selain pengenalnya).
2) Alih-alih antrian biasa kita menggunakan antrian prioritas [jarak adalah prioritas] dan simpul dengan jarak terkecil dipilih untuk diproses.
3) Jarak ke suatu simpul dihitung dengan jumlah bobot sisi-sisi pada lintasan dari sumber ke simpul tersebut.
4) Kita memperbarui jarak jika jarak yang baru dihitung lebih kecil dari jarak lama yang telah kita hitung.

```
void Dijkstra(struct Graph *G, int s) {
      struct PriorityQueue *PQ = CreatePriorityQueue();
      int v, w;
      EnQueue(PQ, s);
      for (int i = 0; i< G→V;i++)
          Distance[i]=-1;
      Distance[s] = 0;
      while ((!IsEmptyQueue(PQ)) {
         v = DeleteMin(PQ);
         for all adjacent vertices w of v {
            Compute new distance d= Distance[v] + weight[v][w];
            if(Distance[w] == -1) {
                Distance[w] = new distance d;
                Insert w in the priority queue with priority d
                Path[w] = v;
            }
            if(Distance[w]  > new distance d) {
                Distance[w] = new distance d;
              Update priority of vertex w to be d;
              Path[w] = v;
            }
         }
      }
}
```

Algoritma di atas dapat lebih dipahami melalui sebuah contoh, yang akan menjelaskan setiap langkah yang diambil dan bagaimana Jarak dihitung. Grafik berbobot di bawah ini memiliki 5 simpul dari A – E.

Nilai antara dua simpul dikenal sebagai biaya tepi antara dua simpul. Misalnya, biaya tepi antara A dan C adalah 1. Algoritma Dijkstra dapat digunakan untuk mencari jalur terpendek dari sumber A ke simpul yang tersisa dalam grafik.



Gambar 1.25 garfik simpul yang tersisa

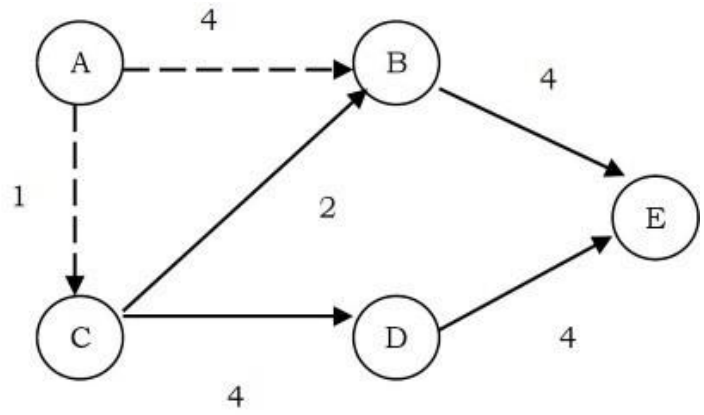
awalnya tabel Jarak adalah:

Tabel 1.8 tabel analisis vertex dan jarak

Vertex	Jarak [v]	Titik sebelumnya yang memberikan Jarak [v]
A	0	-
B	-1	-
C	-1	-
D	-1	-
E	-1	-

Setelah langkah pertama, dari titik A, kita dapat mencapai B dan C. Jadi, dalam tabel Jarak, kita memperbarui jangkauan B dan C dengan biayanya dan hal yang sama ditunjukkan di bawah ini.

A	0	-
B	4	A
C	1	A
D	-1	-
E	-1	-

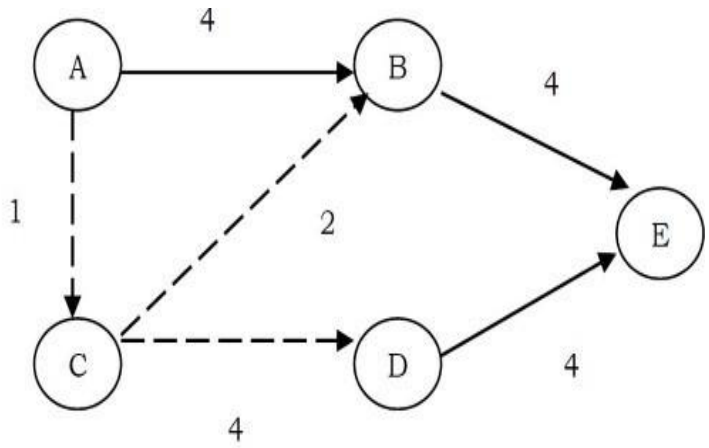


Gambar 1.26 memperbarui jangkauan B dan C

Jalur terpendek dari B, C dari A

Sekarang, mari kita pilih jarak minimum di antara semuanya. Jarak minimum simpul adalah C. Artinya, kita harus mencapai simpul lain dari dua simpul ini (A dan C). Misalnya, B dapat dicapai dari A dan juga dari C. Dalam hal ini kita harus memilih salah satu yang memberikan biaya terendah. Karena mencapai B melalui C memberikan biaya minimum (1 + 2), kita memperbarui tabel Jarak untuk simpul B dengan biaya 3 dan simpul dari mana kita mendapatkan biaya ini sebagai C.

A	0	-
B	3	C
C	1	A
D	5	C
E	-1	-

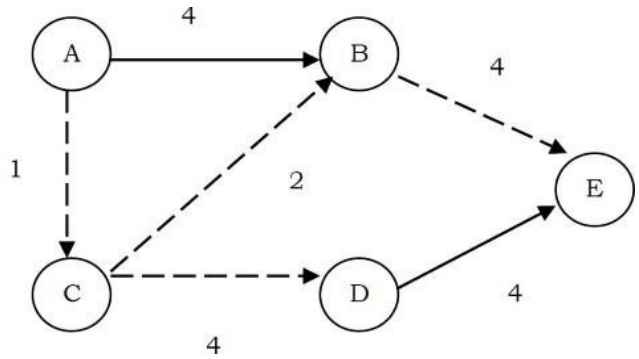


Gambar 1.27 jarak simpul B dengan biaya 3

Jalur terpendek ke B, D menggunakan C sebagai simpul perantara

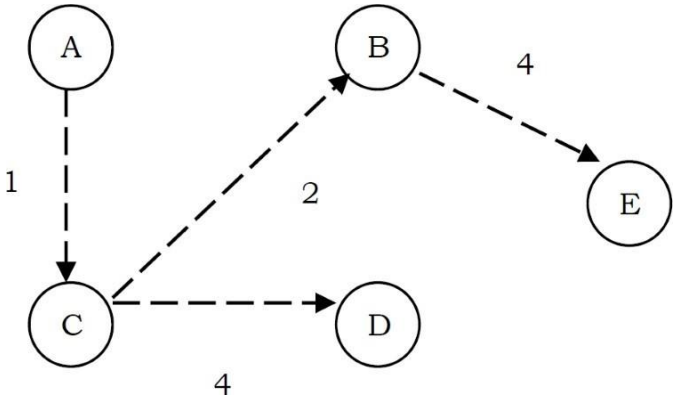
Satu-satunya simpul yang tersisa adalah E. Untuk mencapai E, kita harus melihat semua jalur yang melaluinya kita dapat mencapai E dan memilih salah satu yang memberikan biaya minimum. Kita dapat melihat bahwa jika kita menggunakan B sebagai titik perantara melalui C kita mendapatkan biaya minimum.

A	0	-
B	3	C
C	1	A
D	5	C
E	7	B



Gambar 1.28 B sebagai titik perantara melalui C

Pohon biaya minimum terakhir yang dihasilkan oleh algoritma Dijkstra adalah:



Gambar 1.29 pohon biaya minimum terakhir yang dihasilkan oleh algoritma Dijkstra

Kinerja

Dalam algoritma Dijkstra, efisiensi tergantung pada jumlah DeleteMins (V DeleteMins) dan update untuk antrian prioritas (E update) yang digunakan. Jika tumpukan biner standar digunakan maka kompleksitasnya adalah O(ElogV).

Istilah ElogV berasal dari pembaruan E (setiap pembaruan membutuhkan logV) untuk heap standar. Jika himpunan yang digunakan adalah array maka kompleksitasnya adalah O(E + V2).

Kekurangan Algoritma Dijkstra

- Seperti yang dibahas di atas, kelemahan utama dari algoritma ini adalah bahwa ia melakukan pencarian buta, sehingga membuang-buang waktu dan sumber daya yang diperlukan.
- Kelemahan lainnya adalah tidak dapat menangani sisi negatif. Hal ini menyebabkan Grafik asiklik dan paling sering tidak dapat memperoleh jalur terpendek yang benar.

Kerabat Algoritma Dijkstra

- Algoritma Bellman-Ford menghitung jalur terpendek sumber tunggal dalam digrafik berbobot. Ini menggunakan konsep yang sama dengan algoritma Dijkstra tetapi dapat menangani tepi negatif juga. Ini memiliki lebih banyak waktu berjalan daripada algoritma Dijkstra.
- Algoritma Prim menemukan pohon merentang minimum untuk grafik berbobot terhubung. Ini menyiratkan bahwa subset dari tepi yang membentuk pohon di mana bobot total semua tepi di pohon diminimalkan.

Algoritma Bel man-Ford

Jika grafik memiliki biaya tepi negatif, maka algoritma Dijkstra tidak berfungsi. Masalahnya adalah begitu sebuah simpul u dinyatakan diketahui, ada kemungkinan bahwa dari beberapa simpul lain yang tidak diketahui v ada jalur kembali ke u yang sangat negatif. Dalam kasus seperti itu, mengambil jalur dari s ke v kembali ke u lebih baik daripada pergi dari s ke u tanpa menggunakan v. Kombinasi algoritma Dijkstra dan algoritma tidak berbobot akan menyelesaikan masalah. Inisialisasi antrian dengan s. Kemudian, pada setiap tahap, kita DeQueue a simpul v. Kita menemukan semua simpul W berdekatan dengan v sedemikian rupa sehingga,

jarak ke v + bobot (v,w) < jarak lama ke w

Kita memperbarui jarak dan jalur lama, dan menempatkan w pada antrian jika belum ada di sana. Sedikit dapat diatur untuk setiap simpul untuk menunjukkan kehadiran dalam antrian. Kita ulangi prosesnya sampai antrian kosong.

```
void BellmanFordAlgorithm(struct Graph *G, int s) {
    struct Queue *Q = CreateQueue();
    int v, w;
    EnQueue(Q, s);
    Distance[s] = 0;                    // assume the Distance table is filled with INT_MAX
    while ((!IsEmptyQueue(Q)) {
        v = DeQueue(Q);
        for all adjacent vertices w of v {
            Compute new distance d= Distance[v] + weight[v][w];
            if(old distance to w > new distance d ) {
                Distance[v] = (distance to v) + weight[v][w]);
                Path[w] = v;
                if(w is there in queue)
                    EnQueue(Q, w)
            }
        }
    }
}
```

Algoritma ini bekerja jika tidak ada siklus biaya negatif. Setiap simpul paling banyak dapat DeQueue | V| kali, jadi waktu berjalannya adalah O(|E|. |V|) jika daftar ketetanggaan digunakan.

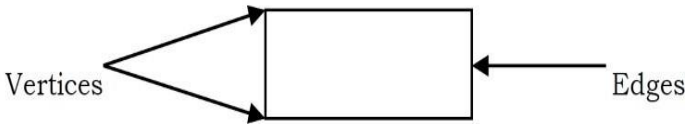
Ikhtisar Algoritma Jalur Terpendek

Tabel 1.9 ikhtisar algroritma jalur terpendek

Jalur terpendek dalam grafik tak berbobot [Modified BFS]	O(*|E| + |V|*)
Jalur terpendek dalam grafik berbobot [Dijkstra's]	O(*|E| log |V|*)
Jalur terpendek dalam grafik berbobot dengan sisi negatif [Bellman – Ford]	O(*|E|.|V|*)
Jalur terpendek dalam grafik asiklik berbobot	O(*|E| + |V|*)

1.8 POHON RENTANG MINIMAL

Pohon merentang dari suatu grafik adalah subgrafik yang berisi semua simpul dan juga merupakan pohon. Suatu grafik dapat memiliki banyak pohon merentang. Sebagai contoh, perhatikan Grafik dengan 4 simpul seperti yang ditunjukkan di bawah ini. Mari kita asumsikan bahwa sudut-sudut Grafik adalah simpul.



Gambar 1.30 pohon rentang minimal

Untuk Grafik sederhana ini, kita dapat memiliki beberapa pohon merentang seperti yang ditunjukkan di bawah ini.

Gambar 1.31 pohon merentang dari grafik sederhana

Algoritma yang akan kita bahas sekarang adalah pohon merentang minimum pada grafik tak berarah. Kita berasumsi bahwa Grafik yang diberikan adalah Grafik berbobot. Jika grafik adalah grafik tidak berbobot maka kita masih dapat menggunakan algoritma grafik berbobot dengan memperlakukan semua bobot sebagai sama. Pohon merentang minimum dari grafik tak-berarah G adalah pohon yang dibentuk dari tepi-tepi grafik yang menghubungkan semua simpul G dengan biaya total (bobot) minimum. Pohon merentang minimum hanya ada jika grafik terhubung. Ada dua algoritma terkenal untuk masalah ini:

- Algoritma Prim
- Algoritma Kruskal

Algoritma Prim

Algoritma Prim hampir sama dengan algoritma Dijkstra. Seperti pada algoritma Dijkstra, pada algoritma Prim kita menyimpan nilai jarak dan jalur dalam tabel jarak. Satu-satunya pengecualian adalah karena definisi jarak berbeda, pernyataan pemutakhiran juga sedikit berubah. Pernyataan pembaruan lebih sederhana dari sebelumnya.

```
void Prims(struct Graph *G, int s) {
    struct PriorityQueue *PQ = CreatePriorityQueue();
    int v, w;
    EnQueue(PQ, s);
    Distance[s] = 0;                    // assume the Distance table is filled with -1
    while ((!IsEmptyQueue(PQ)) {
    v = DeleteMin(PQ);
        for all adjacent vertices w of v {
            Compute new distance d= Distance[v] + weight[v][w];
            if(Distance[w] == -1) {
                Distance[w] = weight[v][w];
                Insert w in the priority queue with priority d
                Path[w] = v;
            }
            if(Distance[w]  > new distance d) {
                Distance[w] = weight[v][w];
                Update priority of vertex w to be d;
                Path[w] = v;
            }
        }
    }
}
```

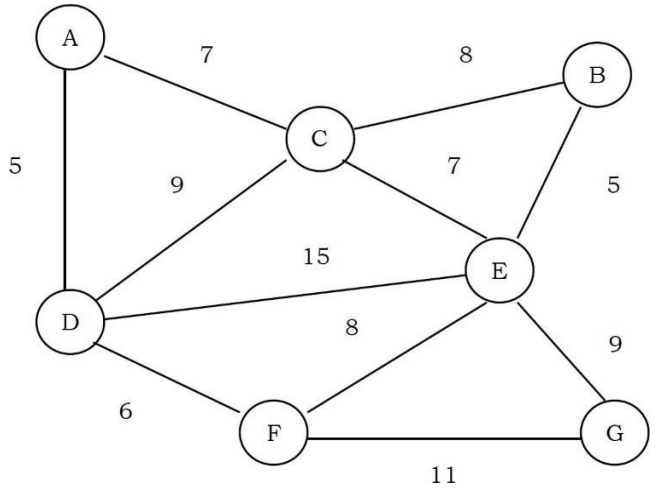
Keseluruhan implementasi dari algoritma ini identik dengan algoritma Dijkstra. Waktu berjalan adalah O(|V|2) tanpa tumpukan [baik untuk Grafik padat], dan O (ElogV) menggunakan tumpukan biner [baik untuk Grafik jarang].

Algoritma Kruskal

Algoritma dimulai dengan V pohon yang berbeda (V adalah simpul dalam Grafik). Saat membangun pohon merentang minimum, setiap kali algoritma Kruskal memilih tepi yang memiliki bobot minimum dan kemudian menambahkan tepi itu jika tidak membuat siklus. Jadi, awalnya ada | V | pohon simpul tunggal di hutan. Menambahkan tepi menggabungkan dua pohon menjadi satu. Ketika algoritma selesai, hanya akan ada satu pohon, dan itu adalah pohon merentang minimum. Ada dua cara untuk mengimplementasikan algoritma Kruskal:

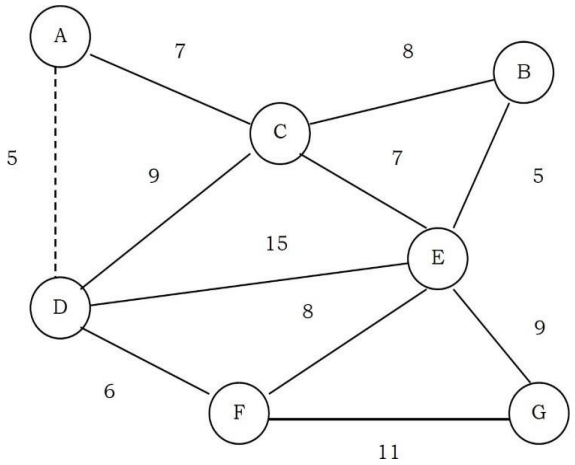
- Dengan menggunakan Set Terpisah: Menggunakan operasi UNION dan FIND
- Dengan menggunakan Antrian Prioritas: Mempertahankan bobot dalam antrian prioritas

Struktur data yang sesuai adalah algoritma UNION/FIND [untuk mengimplementasikan hutan]. Dua simpul termasuk dalam himpunan yang sama jika dan hanya jika mereka terhubung dalam hutan merentang saat ini. Setiap simpul pada awalnya berada dalam himpunannya sendiri. Jika u dan v berada pada himpunan yang sama, rusuknya ditolak karena membentuk suatu siklus. Jika tidak, tepi diterima, dan UNION dilakukan pada dua set yang berisi u dan v. Sebagai contoh, perhatikan Grafik berikut (tepi menunjukkan bobot).



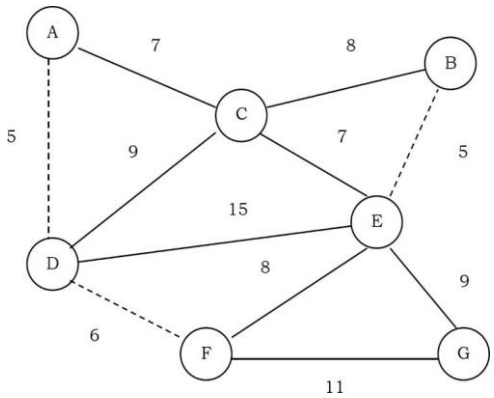
Gambar 1.32 grafik tepi menunjukkan bobot

Sekarang mari kita lakukan algoritma Kruskal pada Grafik ini. Kita selalu memilih tepi yang memiliki bobot minimum.



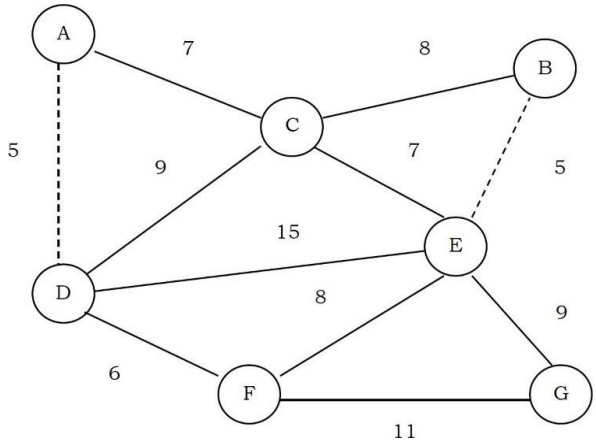
Gambar 1.33 algoritma kruskal

Dari Grafik di atas, sisi-sisi yang memiliki bobot (biaya) minimum adalah: AD dan BE. Dari keduanya kita dapat memilih salah satunya dan mari kita asumsikan bahwa kita memilih AD (garis putus-putus).



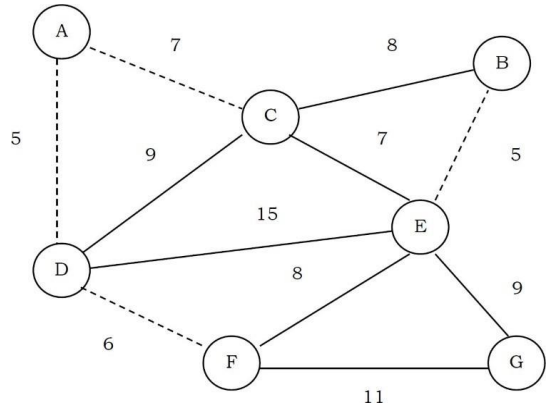
Gambar 1.34 memilih AD garis putus putus

DF adalah tepi berikutnya yang memiliki biaya terendah (6).



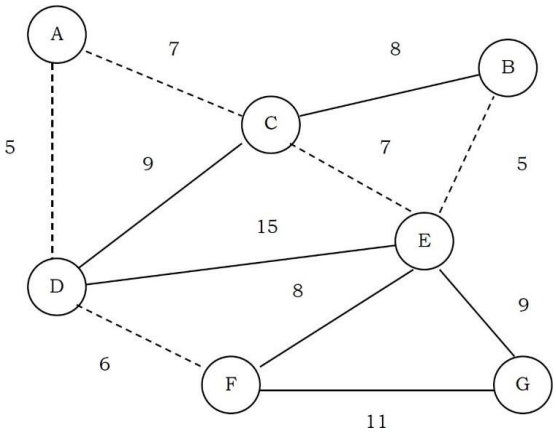
Gambar 1.35 DF biaya terendah (6)

BE sekarang memiliki biaya terendah dan kita memilihnya (garis putus-putus menunjukkan tepi yang dipilih).



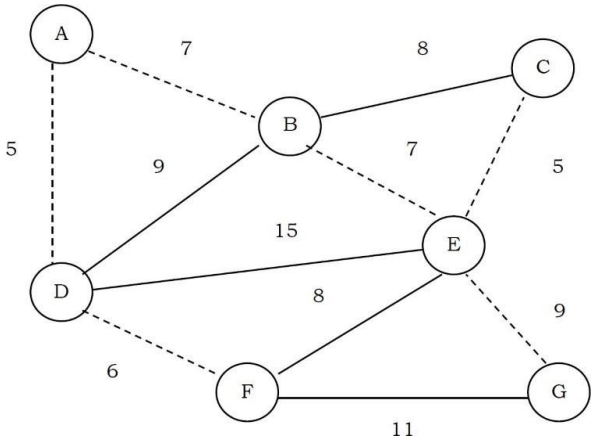
Gambar 1.36 BE memiliki biaya terendah

Selanjutnya, AC dan CE memiliki biaya rendah 7 dan kita memilih AC.



Gambar 1.37 memilih AC biaya rendah (7)

Kemudian kita memilih CE karena biayanya adalah 7 dan tidak membentuk siklus.



Gambar 1.38 CE tidak membentuk

Tepi biaya rendah berikutnya adalah CB dan EF. Tetapi jika kita memilih CB, maka itu membentuk siklus. Jadi kita buang. Hal ini juga terjadi pada EF. Jadi sebaiknya kita tidak memilih keduanya. Dan biaya rendah berikutnya adalah 9 (BD dan EG). Memilih BD membentuk siklus jadi kita membuangnya. Menambahkan EG tidak akan membentuk siklus dan oleh karena itu dengan tepi ini kita menyelesaikan semua simpul dari Grafik.

```
void Kruskal(struct Graph *G) {
    S = ϕ;   // At the end S will contains the edges of minimum spanning trees
    for (int v = 0; v< G→V; v++)
        MakeSet (v);
    Sort edges of E by increasing weights w;
    for each edge (u, v) in E { //from sorted list
        if(FIND (u) ≠ FIND (v)) {
            S = S ∪  {(u, v)};
            UNION (u, v);
        }
    }
    return S;
}
```

Catatan: *Untuk implementasi operasi UNION dan FIND, lihat Disjoint Sets ADT Bab.*

Waktu berjalan terburuk dari algoritma ini adalah O(ElogE), yang didominasi oleh operasi heap. Itu berarti, karena kita sedang membangun heap dengan tepi E, kita membutuhkan waktu O(ElogE) untuk melakukannya.

1.9 ALGORITMA GRAFIK: MASALAH & SOLUSI

Soal-1 Pada grafik sederhana tak berarah dengan n simpul, berapakah jumlah rusuk maksimum? Self-loop tidak diperbolehkan.

Solusi: Karena setiap node dapat terhubung ke semua node lainnya, node pertama dapat terhubung ke n – 1 node. Node kedua dapat terhubung ke n – 2 node [karena satu sisi sudah ada dari node pertama]. Jumlah total rusuk adalah: 1 + 2 + 3 + … + n – $= \frac{n(n-1)}{2}$tepi.

Soal-2 Berapa banyak matriks ketetanggaan berbeda yang dimiliki grafik dengan n titik dan sisi E?

Solusi Ini sama dengan jumlah permutasi dari n elemen, yaitu, n!.

Soal-3 Berapa banyak daftar ketetanggaan berbeda yang dimiliki grafik dengan n simpul?

Solusi Sama dengan jumlah permutasi sisi, yaitu E!.

Soal-4 Representasi grafik tak berarah manakah yang paling tepat untuk menentukan apakah suatu simpul terisolasi atau tidak (tidak terhubung dengan simpul lain)?

Solusi: Daftar Kedekatan. Jika kita menggunakan matriks adjacency, maka kita perlu memeriksa baris lengkap untuk menentukan apakah simpul tersebut memiliki tepi atau tidak. Dengan menggunakan adjacency list, sangat mudah untuk memeriksanya, dan dapat dilakukan hanya dengan memeriksa apakah simpul tersebut memiliki NULL untuk penunjuk berikutnya atau tidak [NULL menunjukkan bahwa simpul tersebut tidak terhubung ke simpul lainnya].

Soal-5 Untuk memeriksa apakah ada jalur dari sumber s ke target t, mana yang terbaik antara himpunan lepas dan DFS?

Solusi: Tabel di bawah ini menunjukkan perbandingan antara himpunan disjoint dan DFS. Entri dalam tabel mewakili kasus untuk setiap pasangan node (untuk s dan t).

Metode	Waktu pengerjaan	Waktu Kueri	Ruang angkasa
Union-Temukan	*V + E logV*	*logV*	*V*
DFS	*E + V*	*1*	*E + V*

Soal-6 Berapakah jumlah maksimum sisi yang dapat dimiliki oleh grafik berarah dengan n simpul dan masih tidak mengandung siklus berarah?

Solusi: Angkanya adalah V (V – 1)/2. Setiap grafik berarah dapat memiliki paling banyak n2 sisi. Namun, karena grafik tersebut tidak memiliki siklus, maka grafik tersebut tidak dapat memuat loop sendiri, dan untuk sembarang pasangan x,y dari simpul, paling banyak satu sisi dari (x,y) dan (y,x) dapat dimasukkan. Oleh karena itu jumlah tepi dapat paling banyak (V2 – V)/2 sesuai keinginan. Dimungkinkan untuk mencapai tepi V(V – 1)/2. Beri label n simpul 1,2... n dan tambahkan sisi (x, y) jika dan hanya jika x < y. Grafik ini memiliki jumlah tepi yang sesuai dan tidak dapat berisi siklus (setiap jalur mengunjungi urutan node yang meningkat).

Soal-7 Berapa banyak grafik berarah sederhana tanpa sisi paralel dan loop-sendiri yang mungkin dalam bentuk V?

Solusi: (V) × (V – 1). Karena, setiap simpul dapat terhubung ke V – 1 simpul tanpa loop sendiri.

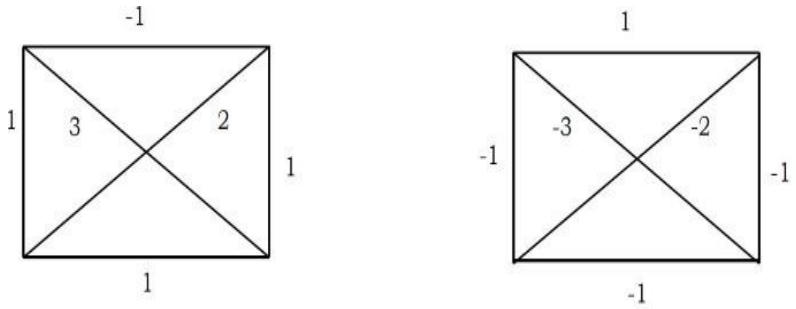
Soal-8 Apa perbedaan antara DFS dan BFS?

Solusi:

DFS	BFS
Mundur dimungkinkan dari jalan buntu.	Pelacakan mundur tidak dimungkinkan.
Simpul dari mana eksplorasi tidak lengkap diproses dalam urutan LIFO	Simpul yang akan dieksplorasi diatur sebagai antrian FIFO.
Pencarian dilakukan dalam satu arah tertentu	Simpul pada tingkat yang sama dipertahankan secara paralel.

Soal-9 Sebelumnya dalam bab ini, kita telah membahas algoritma pohon merentang minimum. Sekarang, berikan algoritma untuk menemukan pohon merentang dengan bobot maksimum dalam Grafik.

Solusi:



Grafik yang diberikan **Grafik yang diubah dengan bobot sisi negatif**

Dengan menggunakan grafik yang diberikan, buat grafik baru dengan simpul dan sisi yang sama. Tetapi alih-alih menggunakan bobot yang sama, ambil negatif bobotnya. Artinya, bobot sisi = negatif bobot sisi yang bersesuaian pada grafik yang diberikan. Sekarang, kita dapat menggunakan algoritma pohon merentang minimum yang ada pada grafik baru ini. Hasilnya, kita akan mendapatkan pohon rentang dengan bobot maksimum pada pohon aslinya.

Soal-10 Berikan algoritma untuk memeriksa apakah grafik tertentu G memiliki jalur sederhana dari sumber s ke tujuan d. Asumsikan grafik G direpresentasikan menggunakan matriks yang berdekatan.

Solusi Mari kita asumsikan bahwa struktur untuk Grafik adalah:

```
struct Graph {
    int V;                  //Number of vertices
    int E;                  //Number of edges
    int ** adjMatrix; //Two dimensional array for storing the connections
};
```

Untuk setiap simpul panggil DFS dan periksa apakah simpul saat ini sama dengan simpul tujuan atau tidak. Jika mereka sama, maka kembalikan 1. Jika tidak, panggil DFS pada tetangganya yang belum dikunjungi. Satu hal penting

yang perlu diperhatikan di sini adalah, kita memanggil algoritma DFS pada simpul-simpul yang belum dikunjungi.

```
void HasSimplePath(struct Graph *G, int s, int d) {
    int t;
    Viisited[s] = 1;
    if(s == d)
        return 1;
    for(t = 0; t < G→V; t++) {
        if(G→adjMatrix[s][t] && !Viisited[t])
            if(DFS(G, t, d))
                return 1;
    }
    return 0;
}
```

Kompleksitas Waktu: O(E). Dalam algoritma di atas, untuk setiap node, karena kita tidak memanggil DFS pada semua tetangganya (membuang melalui kondisi if), Kompleksitas Ruang: O(V).

Soal-11 Hitung jalur sederhana untuk grafik tertentu G memiliki jalur sederhana dari sumber s ke tujuan d? Asumsikan Grafik direpresentasikan menggunakan matriks yang berdekatan.

Solusi: Mirip dengan diskusi di Soal-10, mulai dari satu node dan panggil DFS pada node itu. Sebagai hasil dari panggilan ini, ia mengunjungi semua node yang dapat dijangkau dalam Grafik yang diberikan. Itu berarti ia mengunjungi semua simpul dari komponen yang terhubung dari simpul itu. Jika ada node yang belum dikunjungi, maka mulai lagi dari salah satu node tersebut dan panggil DFS.

Sebelum DFS pertama di setiap komponen yang terhubung, tambahkan jumlah komponen yang terhubung. Lanjutkan proses ini sampai semua simpul Grafik dikunjungi. Akibatnya, pada akhirnya kita akan mendapatkan jumlah total komponen yang terhubung. Implementasi berdasarkan logika ini diberikan di bawah ini:

```
void CountSimplePaths(struct Graph * G, int s, int d) {
    int t;
    Viisited[s] = 1;
    if(s == d) {
        count++;
        Visited[s] = 0;
        return;
    }
    for(t = 0; t < G→V; t++) {
        if(G→adjMatrix[s][t] && !Viisited[t]) {
            DFS(G, t, d);
            Visited[t] = 0;
        }
    }
}
```

Soal-12 Semua pasangan jalur terpendek Masalah: Temukan jarak grafik terpendek antara setiap pasangan simpul dalam grafik tertentu. Mari kita asumsikan bahwa Grafik yang diberikan tidak memiliki sisi negatif.

Solusi: Masalah dapat diselesaikan dengan menggunakan n aplikasi algoritma Dijkstra. Itu berarti kita menerapkan algoritma Dijkstra pada setiap simpul dari grafik yang diberikan. Algoritma ini tidak bekerja jika grafik memiliki sisi dengan bobot negatif.

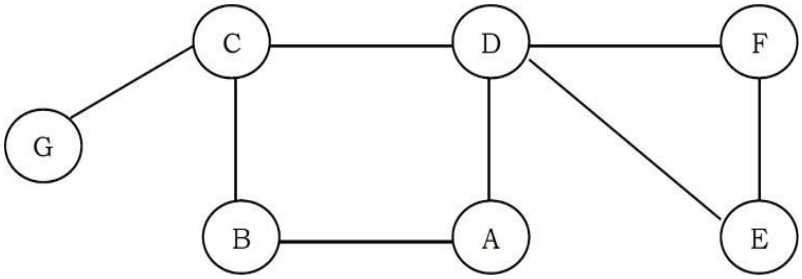
Soal-13 Dalam Soal-12, bagaimana kita menyelesaikan masalah jalur terpendek semua pasangan jika grafik memiliki sisi dengan bobot negatif?

Solusi: Ini dapat diselesaikan dengan menggunakan algoritma Floyd – Warshall. Algoritma ini juga berfungsi dalam kasus grafik berbobot di mana sisi-sisinya memiliki bobot negatif. Algoritma ini adalah contoh dari Pemrograman Dinamis - lihat bab Pemrograman Dinamis.

Soal-14 Aplikasi DFS: Potong Vertex atau Titik Artikulasi

Solusi: Dalam grafik tak berarah, sebuah titik potong (atau titik artikulasi) adalah sebuah titik, dan jika kita menghilangkannya, maka grafik tersebut terbagi menjadi dua komponen yang tidak terhubung. Sebagai contoh perhatikan gambar berikut. Penghapusan simpul "D" membagi grafik menjadi dua komponen terhubung ({E,F} dan {A,B,C,G}).

Demikian pula, penghapusan simpul "C" membagi Grafik menjadi ({G} dan {A, B,D,E,F}). Untuk grafik ini, A dan C adalah titik potong.



Catatan: *Grafik terhubung dan tidak berarah disebut bi – terhubung jika grafik tersebut masih terhubung setelah menghilangkan sembarang simpul.*

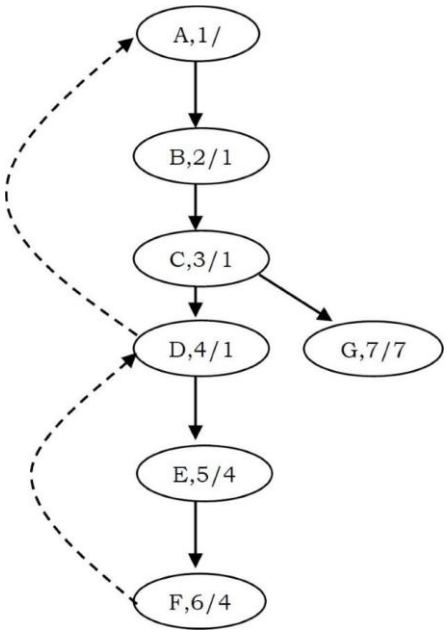
DFS menyediakan algoritma waktu-linear (O(n)) untuk menemukan semua simpul terpotong dalam grafik terhubung. Mulai dari simpul mana pun, panggil DFS dan beri nomor simpul saat mereka dikunjungi. Untuk setiap simpul v, kita menyebut nomor DFS ini dfsnum(v). Pohon yang dihasilkan dengan traversal DFS disebut pohon rentang DFS. Kemudian, untuk setiap simpul v dalam pohon rentang DFS, kita menghitung simpul bernomor terendah, yang kita

sebut rendah(v), yang dapat dicapai dari v dengan mengambil nol atau lebih tepi pohon dan kemudian mungkin satu tepi belakang (dalam urutan itu).

Berdasarkan pembahasan di atas, kita memerlukan informasi berikut untuk algoritma ini: dfsnum dari setiap simpul di pohon DFS (setelah dikunjungi), dan untuk setiap simpul v, kedalaman tetangga terendah dari semua keturunan v di DFS pohon, yang disebut rendah.

Dfsnum dapat dihitung selama DFS. Rendahnya v dapat dihitung setelah mengunjungi semua turunan v (yaitu, tepat sebelum v dikeluarkan dari tumpukan DFS) sebagai minimum dfsnum dari semua tetangga v (selain induk v di pohon DFS) dan rendah dari semua anak v di pohon DFS.

Titik akar adalah titik potong jika dan hanya jika memiliki paling sedikit dua anak. Sebuah simpul non-root u adalah simpul potong jika dan hanya jika ada anak v dari u sedemikian rupa sehingga rendah(v) dfsnum(u). Properti ini dapat diuji setelah DFS dikembalikan dari setiap anak u (itu berarti, tepat sebelum u dikeluarkan dari tumpukan DFS), dan jika benar, u memisahkan Grafik menjadi komponen bi-connected yang berbeda. Ini dapat direpresentasikan dengan menghitung satu komponen bi-connected dari setiap v tersebut (komponen yang berisi v akan berisi sub-pohon v, ditambah u), dan kemudian menghapus sub-pohon v dari pohon.

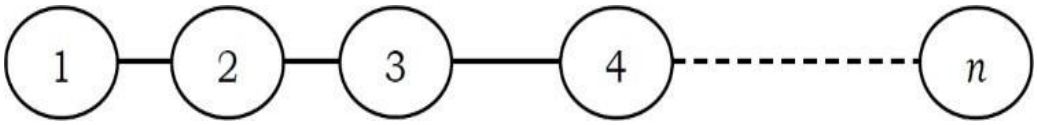


Untuk Grafik yang diberikan, pohon DFS dengan dfsnum/low dapat diberikan seperti yang ditunjukkan pada gambar di bawah ini. Implementasi dari pembahasan di atas adalah:

```
int adjMatrix [256] [256] ;
int dfsnum [256], num = 0, low [256];
void CutVertices( int u ) {
      low[u] = dfsnum[u] = num++;

      for (int v = 0 ; v < 256; ++v ) {
            if(adjMatrix[u][v] && dfsnum[v] == -1) {
                  CutVertices( v ) ;
                  if(low[v] > dfsnum[u])
                        printf("Cut Vetex:%d",u);
                  low[u] = min ( low[u] , low[v] ) ;
            }
            else      // (u,v) is a back edge
                  low[u ] = min(low[u] , dfsnum[v]) ;
      }
}
```

Soal-15 Misalkan G adalah grafik terhubung dengan orde n. Berapa jumlah maksimum cut-vertices yang dapat ditampung G?

Solusi: n – 2. Sebagai contoh, perhatikan Grafik berikut. Pada grafik di bawah ini, kecuali untuk simpul 1 dan n, semua simpul yang tersisa adalah simpul terpotong. Ini karena menghapus 1 dan n simpul tidak membagi Grafik menjadi dua. Ini adalah kasus di mana kita bisa mendapatkan jumlah maksimum titik potong.

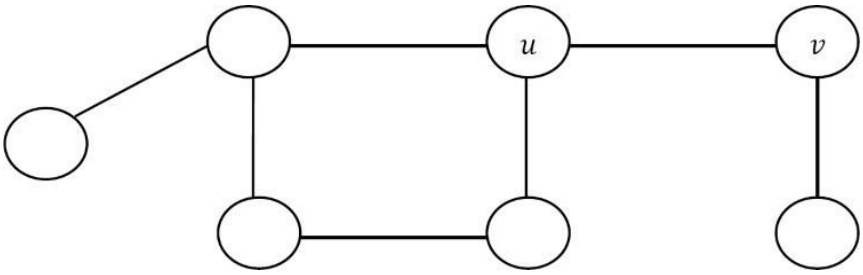


Soal-16 Aplikasi DFS: Memotong Jembatan atau Memotong Tepi

Solusi:

Definisi: Misalkan G adalah grafik terhubung. Sebuah tepi uv di G disebut jembatan G jika G – uv terputus.

Sebagai contoh perhatikan Grafik berikut.



Pada grafik di atas, jika kita menghilangkan sisi uv maka grafik tersebut terbagi menjadi dua komponen. Untuk Grafik ini, uv adalah jembatan. Diskusi yang kita lakukan untuk simpul potong juga berlaku untuk jembatan. Satu-satunya perubahan adalah, alih-alih mencetak titik, kita memberikan tepi. Pengamatan utama adalah bahwa tepi (u, v) tidak dapat menjadi jembatan jika merupakan bagian dari siklus. Jika (u, v) bukan bagian dari siklus, maka itu adalah jembatan.

Kita dapat mendeteksi siklus di DFS dengan adanya tepi belakang, (u, v) adalah jembatan jika dan hanya jika tidak ada anak v atau v yang memiliki tepi belakang ke u atau nenek moyang u. Untuk mendeteksi apakah ada anak v yang memiliki tepi belakang ke induk u, kita dapat menggunakan ide yang sama seperti di atas untuk melihat dfsnum terkecil yang dapat dijangkau dari subpohon yang berakar di v.

```
int dfsnum[256], num = 0, low [256];
void Bridges( struct Graph *G, int u ) {
      low[u] = dfsnum[u] = num++;

      for (int v = 0 ; G→V; ++v ) {
            if(G→adjMatrix[u][v] && dfsnum[v] == -1) {
                  cutVertices( v ) ;

                  if(low[v] > dfsnum[u])
                        print (u,v)  as a bridge

                  low[u] = min ( low[u] , low[v] ) ;
            }
            else      // (u,v) is a back edge
                  low[u ] = min(low[u] , dfsnum[v]) ;
      }
}
```

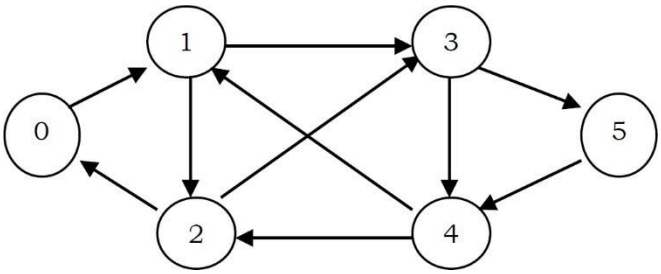
Soal-17 Aplikasi DFS: Diskusikan Sirkuit Euler

Solusi Sebelum membahas masalah ini mari kita lihat terminologinya:

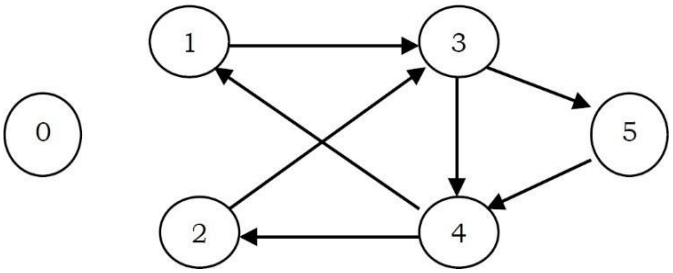
- Eulerian tour- jalur yang berisi semua sisi tanpa pengulangan.
- Sirkuit Euler – jalur yang memuat semua sisi tanpa pengulangan dan berawal dan berakhir pada simpul yang sama.
- Grafik Euler – grafik yang memuat sirkuit Euler.
- Titik genap: sebuah titik yang memiliki banyak sisi datang.
- Odd vertex: sebuah vertex yang memiliki jumlah edge datang yang ganjil.

Sirkuit Euler: Untuk Grafik yang diberikan, kita harus merekonstruksi sirkuit menggunakan pena, menggambar setiap garis tepat satu kali. Kita tidak boleh mengangkat pena dari kertas saat menggambar. Artinya, kita harus menemukan lintasan dalam grafik yang mengunjungi setiap sisi tepat satu kali dan masalah ini disebut lintasan Euler (juga disebut tur Euler) atau masalah sirkuit Euler. Teka-teki ini memiliki solusi sederhana berdasarkan DFS.

Sirkuit Euler ada jika dan hanya jika grafik terhubung dan jumlah tetangga setiap simpul genap. Mulailah dengan simpul apa pun, pilih tepi keluar yang tidak dilalui, dan ikuti. Ulangi sampai tidak ada lagi tepi keluar yang tidak dipilih. Sebagai contoh, perhatikan Grafik berikut: Sirkuit Euler yang sah dari Grafik ini adalah 0 1 3 4 1 2 3 5 4 2 0.

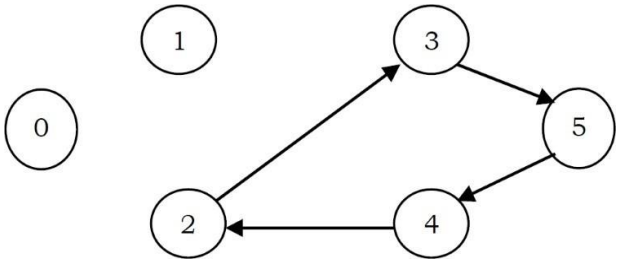


Jika kita mulai dari vertex 0, kita dapat memilih edge ke vertex 1, kemudian memilih edge ke vertex 2, lalu memilih edge ke vertex 0. Sekarang tidak ada edge unchosen yang tersisa dari vertex 0:



Kita sekarang memiliki sirkuit 0,1,2,0 yang tidak melintasi setiap tepi. Jadi, kita memilih beberapa simpul lain yang ada di sirkuit itu, katakanlah simpul 1. Kita kemudian melakukan pencarian kedalaman pertama dari tepi yang tersisa.

Katakanlah kita memilih tepi ke simpul 3, lalu 4, lalu 1. Sekali lagi kita terjebak. Tidak ada lagi tepi yang tidak dipilih dari simpul 1. Sekarang kita sambung jalur ini 1,3,4,1 ke jalur lama 0,1,2,0 untuk mendapatkan: 0,1,3,4,1,2,0. Tepi yang tidak dipilih sekarang terlihat seperti ini:



Kita dapat memilih simpul lain untuk memulai DFS lain. Jika kita memilih simpul 2, dan menyambung jalur 2,3,5,4,2, maka kita mendapatkan rangkaian akhir 0,1,3,4,1,2,3,5,4,2,0.

Masalah serupa adalah menemukan siklus sederhana dalam grafik tak berarah yang mengunjungi setiap simpul. Ini dikenal sebagai masalah siklus Hamilton. Meskipun tampaknya hampir identik dengan masalah rangkaian Euler, tidak ada algoritma yang efisien untuk itu yang diketahui.

Catatan:

- Grafik tak-berarah terhubung adalah Euler jika dan hanya jika setiap simpul grafik berderajat genap, atau tepat dua simpul berderajat ganjil.
- Grafik berarah disebut Euler jika grafik tersebut terhubung kuat dan setiap simpulnya sama gelar masuk dan keluar.

Aplikasi: Seorang tukang pos harus mengunjungi serangkaian jalan untuk mengirimkan surat dan paket. Dia perlu menemukan jalan yang dimulai dan diakhiri di kantor pos, dan yang melewati setiap jalan (tepi) tepat satu kali. Dengan cara ini tukang pos akan mengirimkan surat dan paket ke semua jalan yang diperlukan, dan pada saat yang sama akan menghabiskan waktu/usaha minimum di jalan.

Soal-18 Aplikasi DFS: Menemukan Komponen yang Sangat Terhubung.

Solusi: Ini adalah aplikasi lain dari DFS. Pada grafik berarah, dua simpul u dan v terhubung kuat jika dan hanya jika terdapat lintasan dari u ke v dan terdapat lintasan dari v ke u. Keterkaitan yang kuat adalah relasi ekivalensi.

- Sebuah simpul terhubung kuat dengan dirinya sendiri
- Jika simpul u terhubung kuat ke simpul v, maka v terhubung kuat ke u
- Jika simpul u terhubung kuat ke simpul v, dan v terhubung kuat ke simpul x, maka u terhubung kuat ke x

Artinya, untuk grafik berarah tertentu kita dapat membaginya menjadi komponen-komponen yang terhubung kuat. Masalah ini dapat diselesaikan dengan melakukan dua pencarian mendalam-pertama. Dengan dua pencarian DFS, kita dapat menguji apakah grafik berarah tertentu terhubung kuat atau tidak. Kita juga dapat menghasilkan himpunan bagian dari simpul yang terhubung kuat.

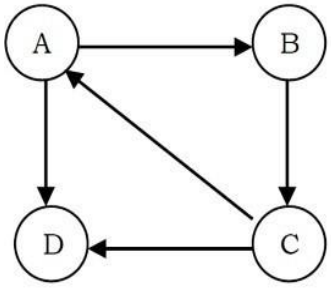
Algoritma

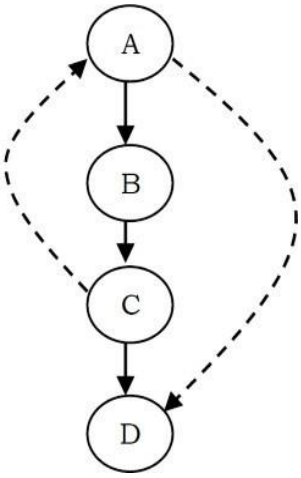
- Lakukan DFS pada grafik G yang diberikan.
- Jumlah simpul dari grafik G yang diberikan menurut traversal post-order dari hutan merentang kedalaman-pertama.
- Bangun grafik Gr dengan membalik semua sisi di G.
- Lakukan DFS di Gr: Selalu mulai DFS baru (panggilan awal untuk Mengunjungi) di vertex bernomor tertinggi.
- Setiap pohon dalam hasil hutan bentang kedalaman pertama sesuai dengan komponen yang terhubung kuat.

Mengapa algoritma ini bekerja?

Mari kita pertimbangkan dua simpul, v dan w. Jika mereka berada dalam komponen terhubung kuat yang sama, maka ada jalur dari v ke W dan dari w ke

v dalam grafik awal G, dan karenanya juga dalam Gr. Jika dua simpul v dan w tidak berada dalam pohon merentang kedalaman pertama yang sama dari Gr, jelas mereka tidak dapat berada dalam komponen terhubung kuat yang sama. Sebagai contoh, perhatikan Grafik yang ditunjukkan di bawah ini di sebelah kiri. Mari kita asumsikan Grafik ini adalah G.

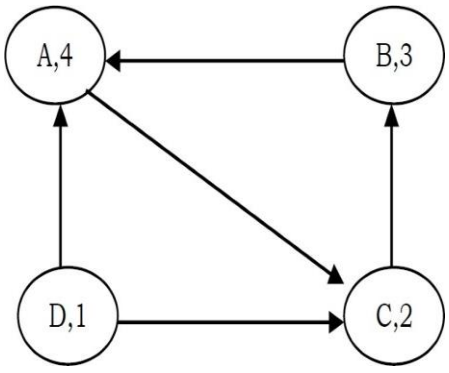




Sekarang, sesuai algoritma, melakukan DFS pada Grafik G ini memberikan diagram berikut. Garis putus-putus dari C ke A menunjukkan tepi belakang. Sekarang, melakukan traversal post order pada pohon ini memberikan: D,C,B dan A.

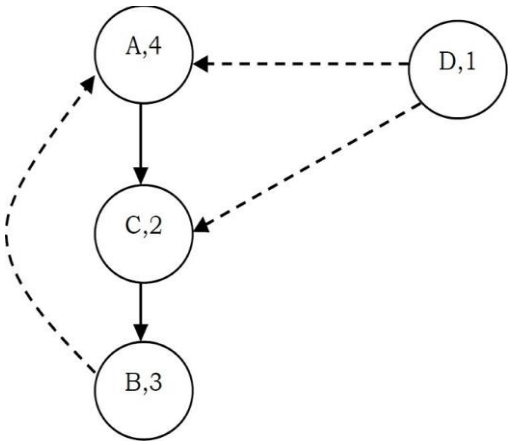
Vertex	Nomor Pesanan Pos
A	4
B	3
C	2
D	1

Sekarang balikkan grafik G yang diberikan dan beri nama G_r dan pada saat yang sama berikan nomor postorder ke simpul. Grafik terbalik G_r akan terlihat seperti:



Langkah terakhir adalah melakukan DFS pada Grafik terbalik G_r. Saat melakukan DFS, kita perlu mempertimbangkan vertex yang memiliki jumlah DFS terbesar. Jadi, pertama kita mulai dari A dan dengan DFS kita pergi ke C dan kemudian B. Di B, kita tidak bisa bergerak lebih jauh. Ini mengatakan bahwa {A, B, C} adalah komponen yang terhubung kuat. Sekarang satu-satunya elemen yang tersisa adalah D dan kita mengakhiri DFS kedua kita di D. Jadi komponen yang terhubung adalah: {A, B, C} dan {D}.

Implementasi berdasarkan pembahasan ini dapat ditunjukkan sebagai berikut:




```
//Graph represented in adj matrix.
int adjMatrix [256][256], table[256];
vector <int> st ;
int counter = 0 ;
//This table contains the DFS Search number
int dfsnum [256], num = 0, low[256] ;
void StronglyConnectedComponents( int u ) {
      low[u] = dfsnum[ u ] = num++;
      Push(st, u) ;
      for( int v = 0 ; v < 256; ++v ) {
            if(graph[u][v] && table[v] == -1) {
                  if( dfsnum[v] == -1)
                      StronglyConnectedComponents(v) ;
                  low[u] = min(low[u] , low[v]) ;
            }
      }
      if(low[u] == dfsnum[u]) {
            while( table[u] != counter) {
                  table[st.back()] = counter;
                  Push(st) ;
            }
            ++ counter;
      }
}
```

Soal-19 Hitung jumlah komponen terhubung dari Grafik G yang diwakili dalam matriks yang berdekatan.

Solusi Masalah ini dapat diselesaikan dengan satu penghitung tambahan di DFS.

```
//Visited[] is a global array.
int Visited[G→V];
void DFS(struct Graph *G, int u) {
    Visited[u] = 1;
    for( int v = 0; v < G→V; v++ ) {
        /* For example, if the adjacency matrix is used for representing the
             graph, then the condition to be used for finding unvisited adjacent
             vertex of u  is: if( !Visited[v] && G→Adj[u][v] ) */
        for each unvisited adjacent node v of u {
            DFS(G, v);
        }
    }
}
void DFSTraversal(struct Graph *G) {
      int count = 0;
      for (int i = 0; i< G→V;i++)
              Visited[i]=0;
      //This loop is required if the graph has more than one component
      for (int i = 0; i< G→V;i++)
              if(!Visited[i]) {
                      DFS(G, i);
                      count++;
              }
      return count;
}
```

Kompleksitas Waktu: Sama seperti DFS dan tergantung pada implementasi. Dengan matriks ketetanggaan kompleksitasnya adalah O(|E| + |V|) dan dengan matriks ketetanggaan kompleksitasnya adalah $O(|V|^2)$.

Soal-20 Bisakah kita menyelesaikan Soal-19, menggunakan BFS?

Solusi: Ya. Masalah ini dapat diselesaikan dengan satu counter tambahan di BFS.

```
void BFS(struct Graph *G, int u) {
    int v,
    Queue Q = CreateQueue();
    EnQueue(Q, u);
    while(!IsEmptyQueue(Q)) {
          u = DeQueue(Q);
          Process u; //For example, print
          Visited[s]=1;
          /* For example, if the adjacency matrix is used for representing the
               graph, then the condition be used for finding unvisited adjacent
               vertex of u  is: if( !Visited[v] && G→Adj[u][v] ) */
          for each unvisited adjacent node v of u {
              EnQueue(Q, v);
          }
    }
}
```

```
void BFSTraversal(struct Graph *G) {
    for (int i = 0; i< G→V;i++)
        Visited[i]=0;
    //This loop is required if the graph has more than one component
    for (int i = 0; i< G→V; i++)
        if(!Visited[i])
            BFS(G, i);
}
```

Kompleksitas Waktu: Sama seperti BFS dan itu tergantung pada implementasi. Dengan matriks ketetanggaan kompleksitasnya adalah O(|E| + |V|) dan dengan matriks ketetanggaan kompleksitasnya adalah O(|V|2).

Soal-21 Mari kita asumsikan bahwa G(V,E) adalah grafik tak berarah. Berikan algoritma untuk menemukan pohon rentang yang membutuhkan kompleksitas waktu O(|E|) (tidak harus pohon rentang minimum).

Solusi: Pengujian sebuah siklus dapat dilakukan dalam waktu yang konstan, dengan menandai simpul-simpul yang telah ditambahkan ke himpunan S. Suatu sisi akan memperkenalkan sebuah siklus, jika kedua simpulnya telah ditandai.

Algoritma:

```
S = {}; //Assume S is a set
for each edge e ∈ E {
    if(adding e to S doesn't form a cycle) {
        add e to S;
        mark e;
    }
}
```

Soal-22 Apakah ada cara lain untuk menyelesaikan 0?

Solusi: Ya. Kita dapat menjalankan BFS dan menemukan pohon BFS untuk Grafik (pohon urutan level dari Grafik). Kemudian mulailah dari elemen root dan terus bergerak ke level berikutnya dan pada saat yang sama kita harus mempertimbangkan node di level berikutnya hanya sekali. Itu berarti, jika kita memiliki simpul dengan banyak sisi input, maka kita harus mempertimbangkan hanya salah satunya; jika tidak mereka akan membentuk siklus.

Soal-23 Mendeteksi sebuah siklus dalam grafik tak berarah

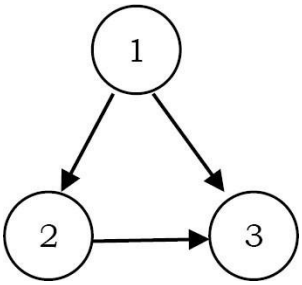
Solusi: Grafik tak berarah adalah asiklik jika dan hanya jika DFS tidak menghasilkan sisi belakang, sisi (u, v) di mana v telah ditemukan dan merupakan ancestor dari u.

- Jalankan DFS pada Grafik.
- Jika ada sisi belakang – Grafik memiliki siklus.

Jika grafik tidak mengandung siklus, maka |E| < |V| dan biaya DFS O(|V|). Jika grafik tersebut mengandung sebuah siklus, maka sisi belakang ditemukan setelah 2|V| langkah paling banyak.

Soal-24 Mendeteksi sebuah siklus di DAG

Solusi:



Deteksi siklus pada Grafik berbeda dari pada pohon. Hal ini karena dalam sebuah grafik, sebuah simpul dapat memiliki banyak orang tua. Dalam sebuah pohon, algoritma untuk mendeteksi sebuah siklus adalah dengan melakukan pencarian pertama yang mendalam, menandai simpul-simpul yang ditemui. Jika simpul yang ditandai sebelumnya terlihat lagi, maka ada siklus. Ini tidak akan berfungsi pada Grafik. Mari kita perhatikan Grafik yang ditunjukkan pada gambar di bawah ini. Jika kita menggunakan algoritma pendeteksian siklus pohon, maka akan melaporkan hasil yang salah. Itu berarti bahwa Grafik ini memiliki siklus di dalamnya. Tetapi Grafik yang diberikan tidak memiliki siklus di dalamnya. Ini karena node 3 akan terlihat dua kali dalam DFS mulai dari node1.

Algoritma deteksi siklus untuk pohon dapat dengan mudah dimodifikasi agar berfungsi untuk Grafik. Kuncinya adalah bahwa dalam DFS dari grafik asiklik, sebuah simpul yang semua turunannya telah dikunjungi dapat dilihat kembali tanpa menyiratkan sebuah siklus. Namun, jika sebuah node terlihat untuk kedua kalinya sebelum semua turunannya dikunjungi, maka harus ada siklus. Dapatkah Anda melihat mengapa ini? Misalkan ada siklus yang berisi node A. Ini berarti A harus dapat dijangkau dari salah satu turunannya. Jadi ketika DFS mengunjungi keturunan tersebut, ia akan melihat A lagi, sebelum selesai mengunjungi semua keturunan A. Jadi ada siklusnya. Untuk mendeteksi siklus, kita dapat memodifikasi pencarian mendalam terlebih dahulu.

```
int DetectCycle(struct Graph *G) {
      for (int i = 0; i< G→V; i++) {
            Visited[s]=0;
            Predecessor[i] = 0;
      }
      for (int i = 0; i < G→V;i++) {
               if(!Visited[i] && HasCycle(G, i))
                        return 1;
      }
      return false;;
}
int HasCycle(struct Graph *G, int u) {
      Visited[u]=1;
      for (int i = 0; i< G→V; i++) {
        if(G→Adj[s][i]) {
            if(Predecessor[i] != u && Visited[i])
                return 1;
             else {
                    Predecessor[i] = u;
                    return  HasCycle(G, i);
            }
        }
    }
    return 0;
}
```

Kompleksitas Waktu: O(V + E).

Soal-25 Diberikan grafik asiklik berarah, berikan algoritma untuk mencari kedalamannya.

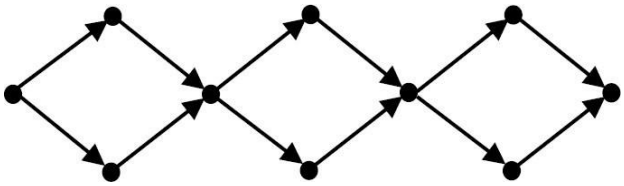
Solusi: Jika merupakan grafik tak berarah, kita dapat menggunakan algoritma jalur terpendek tak berbobot sederhana (periksa bagian Algoritma Jalur Terpendek). Kita hanya perlu mengembalikan angka tertinggi di antara semua jarak. Untuk grafik asiklik berarah, kita dapat menyelesaikannya dengan mengikuti pendekatan serupa yang kita gunakan untuk mencari kedalaman pohon. Di pohon, kita telah memecahkan masalah ini menggunakan traversal urutan level (dengan satu simbol ekstra khusus untuk menunjukkan akhir level).

```
//Assuming the given  graph is a DAG
int DepthInDAG( struct Graph *G ) {
      struct Queue *Q;
      int counter;
      int v, w;
      Q = CreateQueue();

      counter =  0;
      for (v = 0; v< G→V; v++)
               if( indegree[v] ==   0 )
                        EnQueue( Q , v );
      EnQueue( Q, '$' );
```

```
    while( !IsEmptyQueue( Q )) {
            v = DeQueue( Q );
            if(v == '$') {
                    counter++;
                    if(!IsEmptyQueue( Q ))
                            EnQueue( Q , '$' );

            }
            for each w adjacent to v
                    if( --indegree[w] ==  0 )
                            EnQueue ( Q , w );
    }
    DeleteQueue( Q );
    return counter;
}
```

Total waktu berjalan adalah O(V + E).

Soal-26 Ada berapa macam topologi dag berikut?



Solusi: Jika kita mengamati Grafik di atas ada tiga tahap dengan 2 simpul. Dalam diskusi awal bab ini, kita melihat bahwa pengurutan topologi memilih elemen-elemen dengan derajat nol pada setiap titik waktu. Pada masing-masing dari dua tahap simpul, pertama-tama kita dapat memproses simpul atas atau simpul bawah. Akibatnya, pada setiap tahap ini kita memiliki dua kemungkinan. Jadi jumlah total kemungkinan adalah perkalian dari kemungkinan pada setiap tahap dan yaitu, $2 \times 2 \times 2 = 8$.

Soal-27 Urutan topologi unik: Rancang algoritma untuk menentukan apakah grafik berarah memiliki urutan topologi unik.

Solusi: Grafik berarah memiliki urutan topologi unik jika dan hanya jika ada sisi berarah antara setiap pasangan simpul berurutan dalam urutan topologi. Ini juga dapat didefinisikan sebagai: grafik berarah memiliki urutan topologi unik jika dan hanya jika memiliki lintasan Hamilton. Jika digrafik memiliki beberapa urutan topologi, maka urutan topologi kedua dapat diperoleh dengan menukar sepasang simpul berurutan.

Soal-28 Mari kita pertimbangkan prasyarat untuk kursus di IIT Bombay. Misalkan semua prasyarat adalah wajib, setiap mata kuliah ditawarkan setiap semester, dan

tidak ada batasan jumlah mata kuliah yang bisa kita ambil dalam satu semester. Kita ingin mengetahui jumlah minimum semester yang diperlukan untuk menyelesaikan jurusan tersebut. Jelaskan struktur data yang akan kita gunakan untuk mewakili masalah ini, dan garis besar algoritma waktu linier untuk menyelesaikannya.

Solusi: Gunakan Grafik asiklik terarah (DAG). Simpul mewakili mata kuliah dan tepi mewakili hubungan prasyarat antara mata kuliah di IIT Bombay. Ini adalah DAG, karena hubungan prasyarat tidak memiliki siklus.

Jumlah semester yang dibutuhkan untuk menyelesaikan jurusan tersebut lebih banyak satu dari jalur terpanjang di dag. Ini dapat dihitung pada pohon DFS secara rekursif dalam waktu linier. Jalur terpanjang keluar dari simpul x adalah 0 jika x memiliki derajat keluar 0, selain itu adalah 1 + max {jalur terpanjang dari y | (x,y) adalah sisi dari G}.

Soal-29 Di sebuah universitas, katakanlah IIT Bombay), ada daftar mata kuliah beserta prasyaratnya. Itu berarti, dua daftar diberikan:

A – Daftar kursus

B – Prasyarat: B berisi pasangan (x,y) dimana x,y A menunjukkan bahwa mata kuliah x tidak dapat diambil sebelum mata kuliah y.

Mari kita pertimbangkan seorang siswa yang ingin mengambil hanya satu mata kuliah dalam satu semester. Rancang jadwal untuk siswa ini.

Contoh A = {C-Lang, Data Structures, OS, CO, Algorithms, Design Patterns, Programming}. B = { (C-Lang, CO), (OS, CO), (Struktur Data, Algoritma), (Pola Desain, Pemrograman) }. Salah satu jadwal yang mungkin adalah:

Semester 1:	Struktur Data
Semester 2:	Algoritma
Semester 3:	C-Lang
Semester 4:	OS
Semester 5:	CO
Semester 6:	Pola Desain
Semester 7:	Pemrograman

Solusi: Solusi untuk masalah ini persis sama dengan solusi topologi. Asumsikan bahwa nama mata kuliah adalah bilangan bulat dalam rentang [1..n], n diketahui (n tidak konstan). Hubungan antara mata kuliah akan direpresentasikan oleh grafik berarah G = (V,E), dimana V adalah himpunan mata kuliah dan jika mata

kuliah i merupakan prasyarat tentu j, E akan memuat sisi (i,j). Mari kita asumsikan bahwa Grafik akan direpresentasikan sebagai daftar Adjacency.

Pertama, mari kita amati algoritma lain untuk mengurutkan DAG secara topologi dalam O(|V| + |E|).

- Temukan derajat-dalam semua simpul - O(|V| + |E|)
- Mengulang:
  Cari titik v dengan derajat dalam=0 - O(|V|)
  Keluarkan v dan keluarkan dari G, beserta tepinya - O(|V|)
  Kurangi derajat masuk setiap simpul u seperti (v, u) adalah sisi di G dan simpan daftar simpul dengan derajat dalam=0 – O(derajat(v))
  Ulangi proses sampai semua simpul dihapus

Kompleksitas waktu dari algoritma ini juga sama dengan jenis topologi yaitu O(|V|+ |E|).

Soal-30 Pada Soal-29, seorang mahasiswa ingin mengambil semua mata kuliah di A, dalam jumlah semester minimal. Itu berarti mahasiswa siap untuk mengambil sejumlah mata kuliah dalam satu semester. Rancang jadwal untuk skenario ini. Salah satu jadwal yang mungkin adalah:

Semester 1: C-Lang, OS, Pola Desain
Semester 2: Struktur Data, CO, Pemrograman
Semester 3: Algoritma

Solusi: Variasi dari algoritma pengurutan topologi di atas dengan sedikit perubahan: Dalam setiap semester, alih-alih mengambil satu mata pelajaran, ambil semua mata pelajaran dengan derajat nol. Itu berarti, jalankan algoritma pada semua node dengan derajat 0 (daripada berurusan dengan satu sumber di setiap tahap, semua sumber akan ditangani dan dicetak).
Kompleksitas Waktu: O(|V| + |E|).

Soal-31 LCA dari sebuah DAG: Diberikan sebuah DAG dan dua simpul v dan w, carilah leluhur bersama (LCA) terendah dari v dan w. LCA dari v dan w adalah ancestor dari v dan w yang tidak memiliki keturunan yang juga ancestor dari v dan w.

Petunjuk Tentukan tinggi simpul v dalam DAG sebagai panjang lintasan terpanjang dari akar ke v. Di antara simpul yang merupakan nenek moyang dari v dan w, simpul dengan tinggi terbesar adalah LCA dari v dan w .

Soal-32 Jalur leluhur terpendek: Diberikan sebuah DAG dan dua simpul v dan w, temukan jalur leluhur terpendek antara v dan w. Lintasan ancestral antara v

dan w adalah ancestor x bersama dengan lintasan terpendek dari v ke x dan lintasan terpendek dari w ke x. Jalur leluhur terpendek adalah jalur leluhur yang panjang totalnya diminimalkan.

Petunjuk Jalankan BFS dua kali. Jalankan pertama dari v dan kedua kalinya dari w. Temukan DAG di mana jalur ancestral terpendek menuju ke ancestor x yang sama yang bukan LCA.

Soal-33 Mari kita asumsikan bahwa kita memiliki dua grafik G1 dan G2. Bagaimana kita memeriksa apakah mereka isomorfik atau tidak?

Solusi: Ada banyak cara untuk merepresentasikan grafik yang sama. Sebagai contoh, perhatikan Grafik sederhana berikut. Dapat dilihat bahwa semua representasi di bawah ini memiliki jumlah simpul yang sama dan jumlah rusuk yang sama.

Definisi: Grafik G1 = {V1, E1} dan G2 = {V2, E2} isomorfik jika

1) Ada korespondensi satu-satu dari V1 ke V2 dan
2) Ada korespondensi satu-satu dari E1 ke E2 yang memetakan setiap tepi G1 ke G2.

Sekarang, untuk Grafik yang diberikan bagaimana kita memeriksa apakah mereka isomorfik atau tidak?

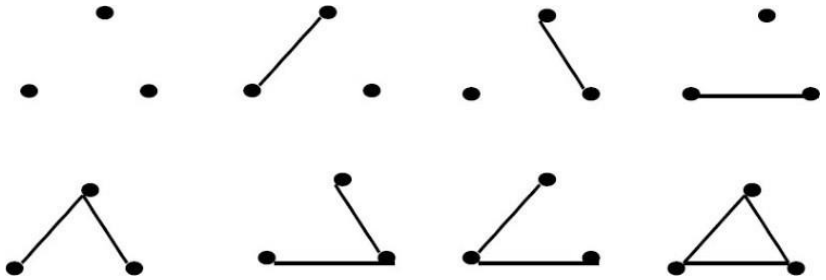
Secara umum, bukanlah tugas yang mudah untuk membuktikan bahwa dua grafik isomorfik. Untuk alasan itu kita harus mempertimbangkan beberapa sifat Grafik isomorfik. Itu berarti sifat-sifat itu harus dipenuhi jika Grafiknya isomorfik. Jika grafik yang diberikan tidak memenuhi sifat-sifat ini maka kita katakan grafik tersebut bukan grafik isomorfik.

Sifat: Dua grafik isomorfik jika dan hanya jika untuk beberapa urutan simpulnya, matriks ketetanggaannya sama.

Berdasarkan sifat di atas, kita putuskan apakah grafik yang diberikan isomorfik atau tidak. Saya memesan untuk memeriksa properti, kita perlu melakukan beberapa operasi transformasi matriks.

Soal-34 Ada berapa grafik tak berarah sederhana non-isomorfik dengan n simpul?

Solusi: Kita akan mencoba menjawab pertanyaan ini dalam dua langkah. Pertama, kita menghitung semua Grafik berlabel. Asumsikan semua representasi di bawah ini diberi label dengan {1,2,3} sebagai simpul. Himpunan semua Grafik tersebut untuk n = 3 adalah:



Hanya ada dua pilihan untuk setiap sisi: itu ada atau tidak. Oleh karena itu, karena jumlah maksimum tepi adalah ($\binom{n}{2}$dan karena jumlah maksimum tepi dalam grafik tak berarah dengan n simpul adalah $\frac{n\,(n-1)}{2} = n_{c_2} = \binom{n}{2}$), jumlah total grafik berlabel tak berarah adalah $2^{\binom{n}{2}}$.

Soal-35 Lintasan Hamilton dalam DAG: Diberikan DAG, rancang algoritma waktu linier untuk menentukan apakah ada lintasan yang mengunjungi setiap simpul tepat satu kali.

Solusi: Masalah jalur Hamiltonian adalah masalah NP-complete (untuk detail lebih lanjut, lihat bab Kelas Kompleksitas). Untuk mengatasi masalah ini, kita akan mencoba memberikan algoritma aproksimasi (yang menyelesaikan masalah, tetapi mungkin tidak selalu menghasilkan solusi yang optimal).

Mari kita pertimbangkan algoritma pengurutan topologi untuk menyelesaikan masalah ini. Pengurutan topologi memiliki sifat yang menarik: bahwa jika semua pasang simpul berurutan dalam urutan terurut dihubungkan oleh tepi, maka tepi ini membentuk jalur Hamilton berarah di DAG. Jika jalur Hamiltonian ada, urutan pengurutan topologi adalah unik. Juga, jika jenis topologi tidak membentuk jalur Hamilton, DAG akan memiliki dua atau lebih urutan topologi.

Algoritma Perkiraan: Hitung jenis topologi dan periksa apakah ada tepi antara setiap pasangan simpul berurutan dalam urutan topologi.

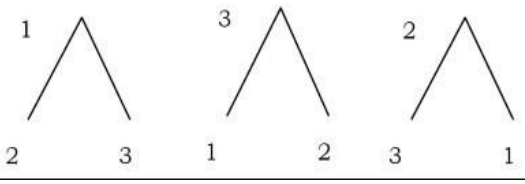
Dalam grafik tak berbobot, temukan lintasan dari s ke t yang mengunjungi setiap simpul tepat satu kali. Solusi dasar berdasarkan backtracking adalah, kita mulai dari s dan mencoba semua tetangganya secara rekursif, memastikan kita tidak pernah mengunjungi simpul yang sama dua kali. Algoritma berdasarkan implementasi ini dapat diberikan sebagai:

```
bool seenTable[32];
void HamiltonianPath( struct Graph *G, int u ) {
      if( u == t )
            /* Check that we have seen all vertices. */
      else {
          for( int v = 0; v < n; v++ )
            if( !seenTable[v] && G→Adj[u][v] ) {
                seenTable[v] = true;
                HamiltonianPath( v );
                seenTable[v] = false;
            }
      }
}
```

Perhatikan bahwa jika kita memiliki jalur parsial dari s ke u menggunakan simpul s = v1, v2,..., vk = u, maka kita tidak peduli dengan urutan kunjungan kita ke simpul ini untuk mengetahui simpul mana untuk mengunjungi berikutnya. Yang perlu kita ketahui hanyalah himpunan simpul yang telah kita lihat (array seenTable[]) dan simpul mana kita berada sekarang (u). Ada 2n himpunan simpul yang mungkin dan n pilihan untuk u. Dengan kata lain, ada 2^n kemungkinan array seenTable[] dan n parameter berbeda untuk Hamiltonian_path(). Apa yang dilakukan Hamiltonian_path() selama panggilan rekursif tertentu sepenuhnya ditentukan oleh array seenTable[ ] dan parameter u.

Soal-36 Untuk grafik tertentu G dengan n simpul berapa banyak pohon yang dapat kita bangun?

Solusi: Ada rumus sederhana untuk masalah ini dan dinamai menurut Arthur Cayley. Untuk grafik tertentu dengan n simpul berlabel, rumus untuk mencari jumlah pohon adalah nn–2. Di bawah ini, jumlah pohon dengan nilai n yang berbeda ditampilkan.

n value	Formula value: n^{n-2}	Number of Trees
2	1	
3	3	

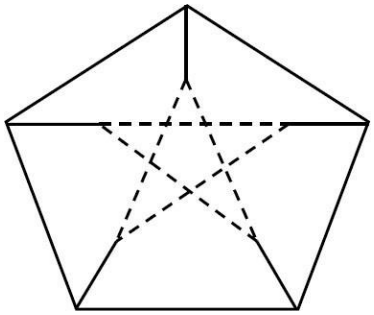
Soal-37 Untuk suatu grafik G dengan n simpul berapa banyak pohon merentang yang dapat kita bangun?

Solusi: Solusi untuk masalah ini sama dengan Soal-36. Ini hanyalah cara lain untuk mengajukan pertanyaan yang sama. Karena jumlah rusuk pada pohon biasa dan pohon merentang adalah sama.

Soal-38 Masalah siklus Hamilton: Apakah mungkin untuk melintasi setiap simpul dari suatu grafik tepat satu kali, dimulai dan berakhir pada simpul yang sama?

Solusi: Karena masalah jalur Hamilton adalah masalah NP-complete, masalah siklus Hamilton adalah masalah NP-complete. Siklus Hamilton adalah siklus yang melintasi setiap titik pada grafik tepat satu kali. Tidak ada kondisi yang diketahui di mana keduanya perlu dan cukup, tetapi ada beberapa kondisi yang cukup.

- Agar grafik memiliki siklus Hamilton, derajat setiap simpul harus dua atau lebih.
- Grafik Petersen tidak memiliki siklus Hamilton dan grafiknya diberikan di bawah ini.



- Secara umum, semakin banyak sisi yang dimiliki suatu grafik, semakin besar kemungkinan memiliki Hamiltonian siklus.
- Misalkan G adalah grafik sederhana dengan n 3 simpul. Jika setiap simpul memiliki derajat paling sedikit $\frac{n}{2}$, maka G memiliki siklus Hamilton.
- Algoritma yang paling terkenal untuk menemukan siklus Hamilton memiliki kompleksitas kasus terburuk eksponensial.

Catatan: Untuk algoritma aproksimasi jalur Hamiltonian, lihat bab Pemrograman Dinamis.

Soal-39 Apa perbedaan antara algoritma Dijkstra dan Prim?

Solusi: Algoritma Dijkstra hampir identik dengan algoritma Prim. Algoritma dimulai pada titik tertentu dan meluas ke luar dalam Grafik sampai semua titik telah tercapai. Satu-satunya perbedaan adalah bahwa algoritma Prim menyimpan tepi biaya minimum sedangkan algoritma Dijkstra menyimpan biaya total dari simpul sumber ke simpul saat ini. Lebih sederhana, algoritma Dijkstra menyimpan penjumlahan tepi biaya minimum sedangkan algoritma Prim menyimpan paling banyak satu tepi biaya minimum.

Soal-40 Grafik Pembalikan: : Berikan algoritma yang mengembalikan kebalikan dari grafik berarah (setiap sisi dari v ke w diganti dengan sisi dari w ke v).

Solusi: Dalam teori grafik, kebalikan (juga disebut transpos) dari grafik berarah G adalah grafik ada himpunan simpul yang sama dengan semua sisi dibalik. Artinya, jika G berisi sisi (u, v) maka kebalikan dari G berisi sisi (v, u) dan sebaliknya.

Algoritma:

```
Graph ReverseTheDirectedGraph(struct Graph *G) {
        Create new graph with name ReversedGraph and
                let us assume that this will contain the reversed graph.
        //The reversed graph also will contain same number of vertices and edges.
        for each vertex of given graph G {
                for each vertex w adjacent to v {
                        Add the w to v edge in ReversedGraph;
                        //That means we just need to reverse the bits in adjacency matrix.
                }
        }
        return ReversedGraph;
}
```

Soal-41 Penjual Bepergian Soal: Temukan jalur terpendek dalam grafik yang mengunjungi setiap simpul setidaknya sekali, dimulai dan diakhiri pada simpul yang sama?

Solusi: Travelling Salesman Problem (TSP) terkait dengan pencarian siklus Hamilton. Diberikan grafik berbobot G, kita ingin mencari siklus terpendek (mungkin tidak sederhana) yang mengunjungi semua simpul.

Algoritma aproksimasi: Algoritma ini tidak menyelesaikan masalah tetapi memberikan solusi yang berada dalam faktor 2 dari optimal (dalam kasus terburuk).

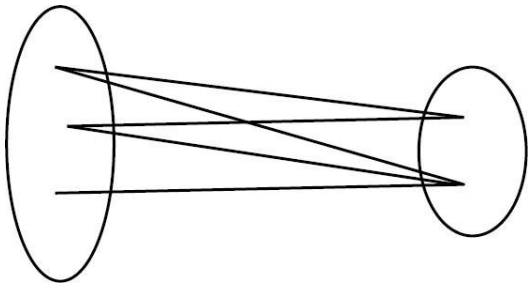
1) Temukan Minimal Spanning Tree (MST).
2) Lakukan DFS dari MST.

Untuk detailnya, lihat bab tentang Kelas Kompleksitas.

Soal-42 Diskusikan Pencocokan Bipartit?

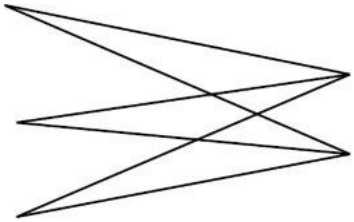
Solusi: Pada grafik Bipartit, kita membagi grafik menjadi dua himpunan lepas, dan setiap sisi menghubungkan simpul dari satu himpunan ke simpul di himpunan bagian lain (seperti yang ditunjukkan pada gambar).

Definisi **:** Grafik sederhana G = (V, E) disebut grafik bipartit jika simpul-simpulnya dapat dibagi menjadi dua himpunan lepas $V = V_1 \cup V_2$, sehingga setiap rusuk berbentuk e = (a,b) dimana a V1 dan b $\in V_2$. Salah satu syarat penting adalah tidak ada simpul baik di V1 atau keduanya di V2 yang terhubung.

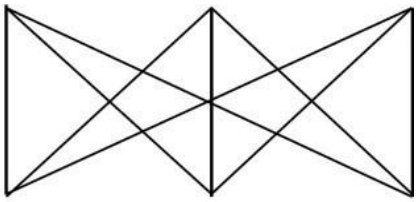


Sifat-sifat Grafik Bipartit

- Suatu grafik disebut bipartit jika dan hanya jika grafik tersebut tidak memiliki panjang siklus ganjil.
- Grafik bipartit lengkap Km,n adalah grafik bipartit yang setiap simpul dari suatu himpunan bertetangga dengan setiap simpul dari himpunan lainnya.

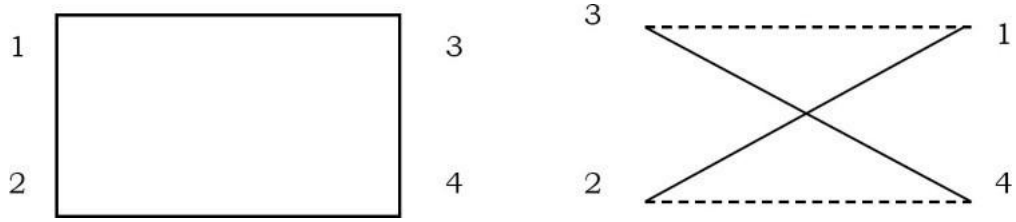


$K_{2,3}$



$K_{3,3}$

- Suatu himpunan bagian dari sisi M E adalah bersesuaian jika tidak ada dua sisi yang memiliki simpul yang sama. Sebagai contoh, set tepi yang cocok direpresentasikan dengan garis putus-putus. M yang cocok disebut maksimum jika memiliki jumlah tepi yang mungkin paling banyak. Dalam Grafik, tepi putus-putus mewakili pencocokan alternatif untuk Grafik yang diberikan.



- M yang cocok adalah sempurna jika cocok dengan semua simpul. Kita harus memiliki V1 = V2 untuk mendapatkan kecocokan yang sempurna.
- Jalur bolak-balik adalah jalur yang ujung-ujungnya bergantian antara cocok dan tepi yang tak tertandingi. Jika kita menemukan jalur alternatif, maka kita dapat meningkatkan pencocokan. Ini karena jalur bolak-balik terdiri dari tepi yang cocok dan tidak cocok. Jumlah tepi yang tidak cocok melebihi jumlah tepi yang cocok satu per satu.

Oleh karena itu, jalur bolak-balik selalu meningkatkan pencocokan satu per satu.

Pertanyaan selanjutnya adalah, bagaimana kita menemukan pasangan yang cocok? Berdasarkan teori dan definisi di atas, kita dapat menemukan pencocokan sempurna dengan algoritma aproksimasi berikut.

Algoritma Pencocokan (algoritma Hungaria):

1) Mulai dari simpul yang tidak cocok.
2) Temukan jalur alternatif.
3) Jika ada, ubah tepi yang cocok menjadi tidak ada tepi yang cocok dan sebaliknya. Jika tidak ada, pilih simpul lain yang tidak cocok.
4) Jika jumlah rusuk sama dengan V/2, berhenti. Jika tidak, lanjutkan ke langkah 1 dan ulangi, selama semua simpul telah diperiksa tanpa menemukan jalur alternatif.

Kompleksitas Waktu dari Algoritma Pencocokan: Jumlah iterasi dalam O(V). Kompleksitas menemukan jalur bolak-balik menggunakan BFS adalah O(E). Oleh karena itu, total kompleksitas waktu adalah O(V × E).

Soal-43 Masalah Pernikahan dan Personalia?

Masalah Pernikahan: Ada X pria dan Y wanita yang ingin menikah. Peserta menunjukkan siapa di antara lawan jenis yang bisa menjadi pasangan potensial bagi mereka. Setiap wanita dapat menikah dengan paling banyak satu pria, dan setiap pria dengan paling banyak satu wanita. Bagaimana kita bisa menikahkan semua orang dengan seseorang yang mereka sukai?

Masalah Personalia: Anda adalah bos dari sebuah perusahaan. Perusahaan memiliki M pekerja dan N pekerjaan. Setiap pekerja memenuhi syarat untuk melakukan beberapa pekerjaan, tetapi tidak yang lain. Bagaimana Anda akan menetapkan pekerjaan untuk setiap pekerja?

Solusi: Kedua kasus ini hanyalah cara lain untuk menanyakan Grafik bipartit, dan solusinya sama dengan Soal-42.

Soal-44 Berapa banyak rusuk pada grafik bipartit lengkap Km,n?

Solusi: m × n. Hal ini karena setiap simpul pada himpunan pertama dapat menghubungkan semua simpul pada himpunan kedua.

Soal-45 Suatu grafik disebut grafik beraturan jika tidak memiliki loop dan sisi berganda di mana setiap titik memiliki jumlah tetangga yang sama; yaitu, setiap simpul memiliki derajat yang sama. Sekarang, jika Km,n adalah grafik beraturan, bagaimana hubungan antara m dan n?

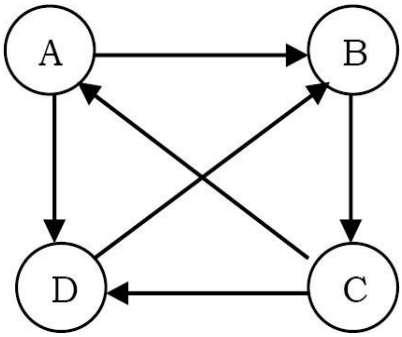
Solusi Karena setiap simpul harus memiliki derajat yang sama, relasinya harus m = n.

Soal-46 Berapa jumlah maksimum sisi dalam pencocokan maksimum grafik bipartit dengan n simpul?

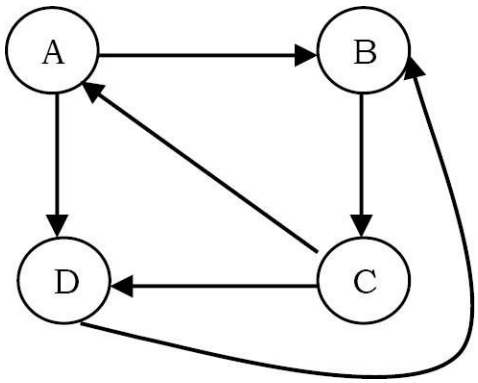
Solusi: Dari definisi pencocokan, kita seharusnya tidak memiliki tepi dengan simpul yang sama. Jadi dalam grafik bipartit, setiap simpul hanya dapat terhubung ke satu simpul. Karena kita membagi total simpul menjadi dua set, kita bisa mendapatkan jumlah tepi maksimum jika kita membaginya menjadi dua. Akhirnya jawabannya adalah $\frac{n}{2}$.

Soal-47 Diskusikan Grafik Planar. Grafik planar: Apakah mungkin menggambar sisi-sisi suatu grafik sedemikian rupa sehingga sisi-sisinya tidak bersilangan?

Solusi: Suatu grafik G dikatakan planar jika dapat digambarkan pada bidang sedemikian rupa sehingga tidak ada dua sisi yang saling bertemu kecuali pada titik di mana mereka datang. Gambar seperti itu disebut gambar bidang G. Sebagai contoh perhatikan Grafik di bawah ini:



Grafik ini dapat dengan mudah kita ubah menjadi grafik planar seperti di bawah ini (tanpa ada sisi yang bersilangan).



Bagaimana kita memutuskan apakah suatu grafik yang diberikan adalah planar atau tidak?

Solusi untuk masalah ini tidak sederhana, tetapi para peneliti telah menemukan beberapa sifat menarik yang dapat kita gunakan untuk memutuskan apakah grafik yang diberikan adalah grafik planar atau bukan.

Sifat-sifat Grafik Planar

- Jika grafik G adalah grafik sederhana planar terhubung dengan V simpul, di mana V = 3 dan E sisi, maka E = 3V – 6.
- K_5 tidak planar. [K_5 adalah singkatan dari grafik lengkap dengan 5 simpul].
- Jika grafik G adalah grafik sederhana planar terhubung dengan simpul V dan rusuk E, dan tidak ada segitiga, maka E = 2V – 4.
- $K_{3,3}$ tidak planar. [$K_{3,3}$ adalah singkatan dari grafik bipartit dengan 3 simpul di satu sisi dan 3 simpul lainnya di sisi lain. $K_{3,3}$ berisi 6 simpul].
- Jika suatu grafik G adalah grafik sederhana planar terhubung, maka G memuat paling sedikit satu simpul yang 5 derajat atau kurang.
- Suatu grafik planar jika dan hanya jika tidak mengandung subgrafik yang memiliki K5 dan $K_{3,3}$ sebagai kontraksi.
- Jika suatu grafik G memuat grafik nonplanar sebagai subgrafik, maka G adalah nonplanar.
- Jika grafik G adalah grafik planar, maka setiap subgrafik G adalah planar.
- Untuk setiap grafik planar terhubung G = (V,E), rumus berikut harus berlaku: V + F – E = 2, di mana F adalah jumlah wajah.
- Untuk sembarang grafik planar G = (V, E) dengan komponen K, rumus berikut berlaku: V + F – E = 1 + K.

Untuk menguji planaritas dari grafik yang diberikan, kita menggunakan sifat-sifat ini dan memutuskan apakah itu grafik planar atau bukan. Perhatikan bahwa semua properti di atas hanya kondisi yang diperlukan tetapi tidak cukup.

Soal-48 Berapa banyak wajah yang dimiliki K2,3?

Solusi: Dari pembahasan di atas, kita mengetahui bahwa V + F – E = 2, dan dari soal sebelumnya kita mengetahui bahwa E = m × n = 2 × 3 = 6 dan V = m + n = 5. 5 + F – 6 = 2 F = 3.

Soal-49 Diskusikan Pewarnaan Grafik

Solusi: Pewarnaan k dari suatu grafik G adalah pemberian satu warna untuk setiap simpul G sedemikian sehingga tidak lebih dari k warna yang digunakan dan tidak ada dua simpul bertetangga yang menerima warna yang sama. Suatu grafik disebut k –dapat diwarnai jika dan hanya jika memiliki k –pewarnaan.

Aplikasi Pewarnaan Grafik Masalah pewarnaan grafik memiliki banyak aplikasi seperti penjadwalan, alokasi register pada compiler, penetapan frekuensi pada radio bergerak, dll.

Clique: Clique pada grafik G adalah subgrafik lengkap maksimum dan dilambangkan dengan (G).

Bilangan kromatik: Bilangan kromatik dari grafik G adalah bilangan terkecil k sehingga G adalah k –berwarna, dan dilambangkan dengan X (G).

Batas bawah untuk X (G) adalah (G), dan itu berarti (G) X (G).

Sifat-sifat Bilangan Kromatik: Misalkan G adalah grafik dengan n simpul dan G' adalah komplemennya. Kemudian,

- X (G) $\leq$ Δ (G) + 1, di mana Δ (G) adalah derajat maksimum G.
- X(G) ω (G') $\geq$ n
- X(G) + ω (G') $\leq$ n + 1
- X(G) + (G') $\leq$ n + 1

Masalah K-warna: Diberikan Grafik G = (V,E) dan bilangan bulat positif k $\leq$ V. Periksa apakah G adalah k –warna?

Masalah ini NP-complete dan akan dibahas secara rinci dalam bab tentang Kelas Kompleksitas.

Algoritma pewarnaan grafik: Seperti yang telah dibahas sebelumnya, masalah ini adalah NP-Complete. Jadi kita tidak memiliki algoritma waktu polinomial untuk menentukan X(G). Mari kita perhatikan algoritma aproksimasi (tidak efisien) berikut.

- Pertimbangkan grafik G dengan dua simpul yang tidak berdekatan a dan b. Sambungan G1 diperoleh dengan menggabungkan dua simpul yang tidak berdekatan a dan b dengan sebuah sisi. Kontraksi G2 diperoleh dengan mengecilkan {a,b} menjadi satu simpul c(a, b) dan dengan menggabungkannya ke setiap tetangga di G dari simpul a dan simpul b (dan menghilangkan banyak sisi).
- Pewarnaan G di mana a dan b memiliki warna yang sama menghasilkan pewarnaan G1. Pewarnaan G di mana a dan b memiliki warna yang berbeda menghasilkan pewarnaan G2.
- Ulangi operasi koneksi dan kontraksi di setiap Grafik yang dihasilkan, sampai Grafik yang dihasilkan semua klik. Jika klik terkecil yang dihasilkan adalah K –klik, maka (G) = K.

Catatan penting tentang Pewarnaan Grafik

- Grafik planar sederhana G dapat diwarnai dengan 6 warna.
- Setiap grafik planar sederhana dapat diwarnai dengan kurang dari atau sama dengan 5 warna.

Soal-50 Apa masalah empat pewarnaan itu?

Solusi: Sebuah Grafik dapat dibangun dari peta apapun. Daerah peta diwakili oleh simpul Grafik, dan dua simpul dihubungkan oleh tepi jika daerah yang bersesuaian dengan simpul bertetangga. Grafik yang dihasilkan adalah planar. Itu berarti dapat digambar dalam bidang tanpa ada sisi yang bersilangan.

Masalah Empat Warna adalah apakah titik-titik pada grafik planar dapat diwarnai dengan paling banyak empat warna sehingga tidak ada dua titik bertetangga yang menggunakan warna yang sama.

Sejarah: Masalah Empat Warna pertama kali diberikan oleh Francis Guthrie. Dia adalah seorang mahasiswa di University College London di mana dia belajar di bawah Augusts De Morgan. Setelah lulus dari London ia belajar hukum, tetapi beberapa tahun kemudian saudaranya Frederick Guthrie telah menjadi murid De Morgan. Suatu hari Francis meminta saudaranya untuk membicarakan masalah ini dengan De Morgan.

Soal-51 Ketika representasi matriks ketetanggaan digunakan, kebanyakan algoritma grafik membutuhkan waktu $O(V^2)$. Tunjukkan bahwa menentukan apakah grafik berarah yang direpresentasikan dalam matriks ketetanggaan yang mengandung sink dapat dilakukan dalam waktu O(V). Wastafel adalah simpul dengan derajat dalam |V| – 1 dan out-degree 0 (Hanya satu yang bisa eksis dalam grafik).

Solusi: Sebuah simpul i adalah sink jika dan hanya jika M[i,j] = 0 untuk semua j dan M[j, i] = 1 untuk semua j i. Untuk sembarang pasangan simpul i dan j:

$$M[i,j] = 1 \rightarrow \text{vertex i can't be a sink}$$
$$M[i,j] = 0 \rightarrow \text{vertex j can't be a sink}$$

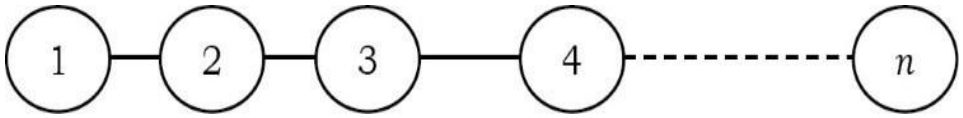
Algoritma:

- Mulai dari i = 1,j = 1
- Jika M[i,j] = 0 → saya menang, j + +
- Jika M[i,j] = 1 → j menang, i + +
- Lanjutkan proses ini sampai j = n atau i = n + 1
- Jika i == n + 1, Grafik tidak mengandung sink
- Jika tidak, periksa baris i – semuanya harus nol; dan centang kolom i – harus semuanya kecuali yang M[i, i]; – jika demikian, t adalah wastafel.

Kompleksitas Waktu: O(V), karena paling banyak 2|V| sel-sel dalam matriks diperiksa.

Soal -52 Apa yang terburuk – penggunaan memori kasus DFS?

Solusi: Ini terjadi ketika O(|V|), yang terjadi jika Grafik sebenarnya adalah daftar. Jadi algoritma ini hemat memori pada grafik dengan diameter kecil.



Soal-53 Apakah DFS menemukan jalur terpendek dari node awal ke beberapa node w ?

Solusi: Tidak. Dalam DFS tidak wajib untuk memilih tepi bobot terkecil.

Soal-54 Benar atau Salah: Algoritma Dijkstra tidak menghitung jalur terpendek "semua pasangan" dalam grafik berarah dengan bobot tepi positif karena, menjalankan Algoritma satu kali, mulai dari beberapa simpul tunggal x, ia hanya akan menghitung jarak min dari x ke y untuk semua node y dalam Grafik.

Solusi: Benar.

Soal-55 Benar atau Salah: Algoritma Prim dan Kruskal dapat menghitung pohon merentang minimum yang berbeda ketika dijalankan pada grafik yang sama.

Solusi: Benar.

BAB 2
PENYORTIRAN

2.1 APA ITU PERNYOTIRAN?

Sorting adalah algoritma yang mengatur elemen-elemen daftar dalam urutan tertentu (baik menaik atau menurun). *Output*nya adalah permutasi atau penataan ulang input.

2.2 MENGAPA PERNYOTIRAN DIPERLUKAN?

Penyortiran adalah salah satu kategori penting dari algoritma dalam ilmu komputer dan banyak penelitian telah masuk ke dalam kategori ini. Penyortiran dapat secara signifikan mengurangi kompleksitas masalah, dan sering digunakan untuk algoritma dan pencarian basis data.

2.3 KLASIFIKASI ALGORITMA PENYORTIRAN

Algoritma pengurutan umumnya dikategorikan berdasarkan parameter berikut.

Berdasarkan Jumlah Perbandingan

Dalam metode ini, algoritma pengurutan diklasifikasikan berdasarkan jumlah perbandingan. Untuk algoritma pengurutan berdasarkan perbandingan, perilaku kasus terbaik adalah O(nlogn) dan perilaku kasus terburuk adalah O(n2). Algoritma pengurutan berbasis perbandingan mengevaluasi elemen daftar dengan operasi perbandingan kunci dan membutuhkan setidaknya perbandingan O(nlogn) untuk sebagian besar input. Kemudian dalam bab ini kita akan membahas beberapa algoritma pengurutan non-perbandingan (linier) seperti Counting sort, Bucket sort, Radix sort, dll. Algoritma Linear Sorting memberlakukan sedikit batasan pada input untuk meningkatkan kompleksitas.

Berdasarkan Jumlah Swap

Dalam metode ini, algoritma pengurutan dikategorikan berdasarkan jumlah swap (juga disebut inversi).

Dengan Penggunaan Memori

Beberapa algoritma pengurutan "di tempat" dan mereka membutuhkan memori O(1) atau O(logn) untuk membuat lokasi tambahan untuk menyortir data sementara.

Dengan Rekursi

Algoritma pengurutan dapat bersifat rekursif [pengurutan cepat] atau non-rekursif [pengurutan pemilihan, dan pengurutan penyisipan], dan ada beberapa Algoritma yang menggunakan keduanya (pengurutan gabungan).

Dengan Stabilitas

Algoritma pengurutan stabil jika untuk semua indeks i dan j sedemikian sehingga kunci A[i] sama dengan kunci A[j], jika *record* R[i] mendahului *record* R[j] dalam file asli, *record* R[i]

mendahului *record* R[j] dalam daftar yang diurutkan. Beberapa algoritma pengurutan mempertahankan urutan relatif elemen dengan kunci yang sama (elemen yang setara mempertahankan posisi relatifnya bahkan setelah pengurutan).

Dengan kemampuan beradaptasi

Dengan beberapa Algoritma pengurutan, kompleksitas berubah berdasarkan pra-penyortiran [penyortiran cepat]: pengurutan awal input memengaruhi waktu berjalan. Algoritma yang memperhitungkan hal ini dikenal adaptif.

2.4 KLASIFIKASI LAINNYA

Metode lain untuk mengklasifikasikan algoritma pengurutan adalah:

- Sortir Internal
- Sortir Eksternal

Sortir Internal

Algoritma pengurutan yang menggunakan memori utama secara eksklusif selama pengurutan disebut algoritma pengurutan internal. Jenis algoritma ini mengasumsikan akses acak berkecepatan tinggi ke semua memori.

Sortir Eksternal

Algoritma pengurutan yang menggunakan memori eksternal, seperti pita atau disk, selama pengurutan termasuk dalam kategori ini.

2.5 GELOMBANG SORTIR

Bubble sort adalah algoritma pengurutan yang paling sederhana. Ini bekerja dengan mengulangi array input dari elemen pertama hingga terakhir, membandingkan setiap pasangan elemen dan menukarnya jika diperlukan. Bubble sort melanjutkan iterasinya sampai tidak ada lagi swap yang diperlukan. Algoritma mendapatkan namanya dari elemen yang lebih kecil "gelembung" ke bagian atas daftar. Secara umum, insertion sort memiliki kinerja yang lebih baik daripada bubble sort. Beberapa peneliti menyarankan agar kita tidak mengajarkan bubble sort karena kesederhanaan dan kompleksitas waktu yang tinggi.

Satu-satunya keuntungan signifikan yang dimiliki bubble sort dibandingkan implementasi lain adalah ia dapat mendeteksi apakah daftar input sudah diurutkan atau belum.

Penerapan

Algoritma membutuhkan $O(n^2)$ (bahkan dalam kasus terbaik). Kita dapat meningkatkannya dengan menggunakan satu bendera tambahan. Tidak ada lagi swap yang menunjukkan selesainya penyortiran.

```
void BubbleSortImproved(int A[], int n) {
    int pass, i, temp, swapped = 1;
    for (pass = n - 1; pass >= 0 && swapped; pass--) {
        swapped = 0;
        for (i = 0; i <= pass - 1 ; i++) {
            if(A[i] > A[i+1]) {
                // swap elements
                temp = A[i];
                A[i] = A[i+1];
                A[i+1] =  temp;
                swapped = 1;
            }
        }
    }
}
```

Jika daftar sudah diurutkan, kita dapat menggunakan tanda ini untuk melewati sisa lintasan. Versi yang dimodifikasi ini meningkatkan kasus terbaik dari bubble sort menjadi O(n).

```
void BubbleSort(int A[], int n) {
    for (int pass = n - 1; pass >= 0; pass--){
        for (int i = 0; i <= pass - 1 ; i++)      {
            if(A[i] > A[i+1]) {
                // swap elements
                int temp = A[i];
                A[i] = A[i+1];
                A[i+1] = temp;
            }
        }
    }
}
```

Kinerja

Kompleksitas kasus terburuk: $O(n^2)$
Kompleksitas kasus terbaik (Versi yang ditingkatkan): O(n)
Kompleksitas kasus rata-rata (Versi dasar) : $O(n^2)$
Kompleksitas ruang kasus terburuk: O(1) tambahan

2.6 SORTIR SELEKSI

Seleksi sortir adalah algoritma pengurutan di tempat. Pengurutan pilihan berfungsi dengan baik untuk file kecil. Ini digunakan untuk menyortir file dengan nilai yang sangat besar dan kunci kecil. Ini karena pemilihan dibuat berdasarkan kunci dan swap dilakukan hanya jika diperlukan.

Keuntungan

- Mudah diterapkan
- Penyortiran di tempat (tidak memerlukan ruang penyimpanan tambahan)

Kekurangan

- Skala tidak baik: $O(n^2)$

Algoritma

1. Temukan nilai minimum dalam daftar
2. Tukar dengan nilai di posisi saat ini
3. Ulangi proses ini untuk semua elemen sampai seluruh array diurutkan

Algoritma ini disebut selection sort karena berulang kali memilih elemen terkecil.

Penerapan

```
void Selection(int A [], int n) {
    int i, j, min, temp;
    for (i = 0; i < n - 1; i++) {
        min = i;
        for (j = i+1; j < n; j++) {
            if(A [j] < A [min])
                min = j;
        }
        // swap elements
        temp = A[min];
        A[min] = A[i];
        A[i] = temp;
    }
}
```

Kinerja

Kompleksitas kasus terburuk: $O(n^2)$
Kompleksitas kasus terbaik : $O(n^2)$
Kompleksitas kasus rata-rata : $O(n^2)$
Kompleksitas ruang kasus terburuk: O(1) tambahan

2.7 PENGURUTAN PENYISIPAN

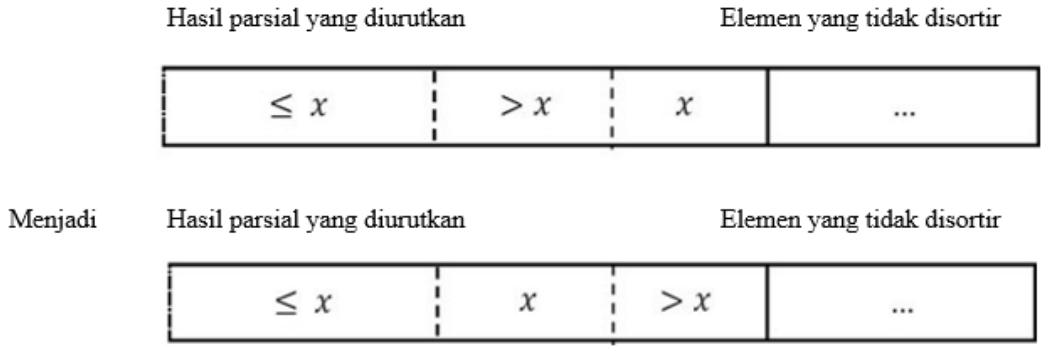
Jenis penyisipan adalah jenis perbandingan yang sederhana dan efisien. Dalam algoritma ini, setiap iterasi menghapus elemen dari data input dan memasukkannya ke posisi yang benar dalam daftar yang sedang diurutkan. Pilihan elemen yang akan dihapus dari input adalah acak dan proses ini diulang sampai semua elemen input selesai.

Keuntungan

- Implementasi sederhana
- Efisien untuk data kecil
- Adaptive: Jika daftar input diurutkan sebelumnya [mungkin tidak sepenuhnya] maka pengurutan penyisipan mengambil O(n + d), di mana d adalah jumlah inversi
- Praktis lebih efisien daripada seleksi dan bubble sort, meskipun semuanya memiliki kompleksitas kasus terburuk $O(n^2)$
- Stabil: Mempertahankan urutan relatif data input jika kuncinya sama
- Di tempat: Hanya membutuhkan ruang memori tambahan O(1) dalam jumlah yang konstan
- Online: Pengurutan penyisipan dapat mengurutkan daftar saat menerimanya

Algoritma

Setiap pengulangan pengurutan penyisipan menghapus elemen dari data input, dan memasukkannya ke posisi yang benar dalam daftar yang sudah diurutkan hingga tidak ada elemen input yang tersisa. Penyortiran biasanya dilakukan di tempat. Array yang dihasilkan setelah k iterasi memiliki properti di mana k + 1 entri pertama diurutkan.



Gambar 2.1 k + 1 entri pertama diurutkan

Setiap elemen yang lebih besar dari x disalin ke kanan karena dibandingkan dengan x.

Penerapan

```
void InsertionSort(int A[], int n) {
    int i, j, v;
    for (i = 1; i <= n - 1; i++) {
        v = A[i];
        j = i;
        while (A[j-1] > v && j >= 1) {
            A[j] = A[j-1];
            j--;
        }
        A[j] = v;
    }
}
```

Contoh

Diberikan sebuah array: 6 8 1 4 5 3 7 2 dan tujuannya adalah untuk menempatkan mereka dalam urutan menaik.

Tabel 2.1 array diurutkan

6 8 1 4 5 3 7 2	(Pertimbangkan indeks 0)
6 8 1 4 5 3 7 2	(Pertimbangkan indeks 0 -1)
1 6 8 4 5 3 7 2	(Pertimbangkan indeks 0 - 2: tempat penyisipan 1 di depan 6 dan 8)
1 4 6 8 5 3 7 2	(Proses yang sama seperti di atas diulang sampai array diurutkan)
1 4 5 6 8 3 7 2	
1 3 4 5 6 7 8 2	
1 2 3 4 5 6 7 8	
	(Array diurutkan!)

Analisis

Analisis kasus terburuk

Kasus terburuk terjadi ketika untuk setiap i loop dalam harus memindahkan semua elemen A[1], . . . , A[i – 1] (yang terjadi ketika A[i] = kunci lebih kecil dari semuanya), yang membutuhkan waktu Θ (i – 1).

$$T(n) = \Theta(1) + \Theta(2) + \Theta(2) + \ldots\ldots + \Theta(n-1)$$
$$= \Theta(1 + 2 + 3 + \ldots\ldots + n - 1) = \Theta(\frac{n(n-1)}{2}) \approx \Theta(n^2)$$

Analisis kasus rata-rata

Untuk kasus rata-rata, loop dalam akan menyisipkan A[i] di tengah A[1], . . . , A[i – 1]. Ini membutuhkan waktu Θ (i/2).

$$T(n) = \sum_{i=1}^{n} \Theta(i/2) \approx \Theta(n^2)$$

Kinerja

Jika setiap elemen lebih besar dari atau sama dengan setiap elemen di sebelah kirinya, waktu berjalannya insertion sort adalah (n). Situasi ini terjadi jika larik dimulai sudah diurutkan, dan larik yang sudah diurutkan adalah kasus terbaik untuk pengurutan penyisipan.

Kompleksitas kasus terburuk: Θ (n^2)
Kompleksitas kasus terbaik: Θ (n)
Kompleksitas kasus rata-rata: Θ (n^2)
Kompleksitas ruang kasus terburuk: O(n^2) total, O(1) tambahan

Perbandingan dengan Algoritma Penyortiran Lainnya

Insertion sort adalah salah satu algoritma pengurutan dasar dengan waktu kasus terburuk O(n2). Insertion sort digunakan ketika data hampir diurutkan (karena adaptifnya) atau ketika ukuran input kecil (karena overhead yang rendah). Untuk alasan ini dan karena stabilitasnya, pengurutan penyisipan digunakan sebagai kasus dasar rekursif (ketika ukuran masalah kecil) untuk algoritma pengurutan pembagian dan penaklukan overhead yang lebih tinggi, seperti pengurutan gabungan atau pengurutan cepat.

Catatan:

- Bubble sort mengambil $\frac{n^2}{2}$ perbandingan dan $\frac{n^2}{2}$ swap (inversi) baik dalam kasus rata-rata dan dalam kasus terburuk.
- Seleksi sort mengambil $\frac{n^2}{2}$ perbandingan dan n swap.
- Jenis penyisipan mengambil $\frac{n^2}{4}$ perbandingan dan $\frac{n^2}{8}$ swap dalam kasus rata-rata dan dalam kasus terburuk mereka ganda.
- Pengurutan penyisipan hampir linier untuk input yang diurutkan sebagian.
- Pengurutan pilihan paling cocok untuk elemen dengan nilai lebih besar dan kunci kecil.

2.8 PENYORTIRAN RAK

Jenis cangkang (juga disebut pengurutan peningkatan yang semakin berkurang) ditemukan oleh Donald Shell. Algoritma sorting ini merupakan generalisasi dari insertion sort. Pengurutan penyisipan bekerja secara efisien pada input yang sudah hampir diurutkan. Jenis shell juga dikenal sebagai jenis penyisipan n-gap. Alih-alih hanya membandingkan pasangan yang berdekatan, pengurutan shell membuat beberapa lintasan dan menggunakan berbagai celah antara elemen yang berdekatan (diakhiri dengan celah 1 atau pengurutan penyisipan klasik).

Dalam jenis penyisipan, perbandingan dibuat antara elemen yang berdekatan. Paling banyak 1 inversi dihilangkan untuk setiap perbandingan yang dilakukan dengan insertion sort. Variasi yang digunakan dalam shell sort adalah untuk menghindari membandingkan elemen yang berdekatan sampai langkah terakhir dari algoritma. Jadi, langkah terakhir dari shell sort secara efektif adalah algoritma insertion sort. Ini meningkatkan pengurutan penyisipan dengan memungkinkan perbandingan dan pertukaran elemen yang jauh. Ini adalah algoritma pertama yang memiliki kompleksitas kurang dari kuadratik di antara algoritma-algoritma pengurutan perbandingan.

Shellsort sebenarnya adalah ekstensi sederhana untuk insertion sort. Perbedaan utama adalah kemampuannya untuk bertukar elemen yang berjauhan, membuatnya jauh lebih cepat bagi elemen untuk mencapai tempat yang seharusnya. Sebagai contoh, jika elemen terkecil berada di akhir array, dengan insertion sort akan membutuhkan langkah-langkah array lengkap untuk menempatkan elemen ini di awal array. Namun, dengan shell sort, elemen ini dapat melompat lebih dari satu langkah dalam satu waktu dan mencapai tujuan yang tepat dalam pertukaran yang lebih sedikit.

Ide dasar dalam shellsort adalah untuk menukar setiap elemen ke-h dalam array. Sekarang ini bisa membingungkan jadi kita akan berbicara lebih banyak tentang ini, h menentukan seberapa jauh pertukaran elemen dapat terjadi, misalnya ambil h sebagai 13, elemen pertama (indeks-0) ditukar dengan elemen ke-14 (indeks-13) jika perlu (tentu saja). Elemen kedua dengan elemen ke-15, dan seterusnya. Sekarang jika kita ambil memiliki 1, itu persis sama dengan jenis penyisipan biasa.

Shellsort bekerja dengan memulai dengan cukup besar (tetapi tidak lebih besar dari ukuran array) sehingga memungkinkan pertukaran elemen yang memenuhi syarat yang berjauhan. Setelah pengurutan selesai dengan h tertentu, array dapat dikatakan sebagai pengurutan-h. Langkah selanjutnya adalah mereduksi h dengan urutan tertentu, dan kembali melakukan pengurutan-h lengkap lainnya. Setelah h adalah 1 dan h-diurutkan, array sepenuhnya diurutkan. Perhatikan bahwa urutan terakhir untuk ft adalah 1 sehingga pengurutan terakhir selalu merupakan pengurutan penyisipan, kecuali saat ini array sudah terbentuk dengan baik dan lebih mudah untuk diurutkan.

Shell sort menggunakan urutan h1,h2, ...,ht yang disebut urutan kenaikan. Setiap urutan kenaikan baik-baik saja selama h1 = 1, dan beberapa pilihan lebih baik daripada yang lain. Penyortiran shell membuat beberapa lintasan melalui daftar input dan mengurutkan sejumlah set berukuran sama menggunakan pengurutan penyisipan. Penyortiran shell meningkatkan efisiensi pengurutan penyisipan dengan menggeser nilai dengan cepat ke tujuannya.

Penerapan

```
void ShellSort(int A[], int array_size) {
    int i, j, h, v;
    for (h = 1; h = array_size/9; h = 3*h+1);
    for ( ; h > 0; h = h/3) {
        for (i = h+1; i = array_size; i += 1) {
            v = A[i];
            j = i;
            while (j > h && A[j-h] > v) {
                A[j] = A[j-h];
                j -= h;
            }
            A[j] = v;
        }
    }
}
```

Perhatikan bahwa ketika h == 1, Algoritma melewati seluruh daftar, membandingkan elemen yang berdekatan, tetapi melakukan sangat sedikit pertukaran elemen. Untuk h == 1, shell sort bekerja seperti insertion sort, kecuali jumlah inversi yang harus dihilangkan sangat berkurang oleh langkah-langkah algoritma sebelumnya dengan h > 1.

Analisis

Penyortiran shell efisien untuk daftar ukuran sedang. Untuk daftar yang lebih besar, Algoritma bukanlah pilihan terbaik. Ini adalah yang tercepat dari semua algoritma pengurutan $O(n^2)$.

Kerugian dari Shell sort adalah algoritma yang kompleks dan hampir tidak seefisien penggabungan, tumpukan, dan pengurutan cepat. Penyortiran shell secara signifikan lebih lambat daripada penggabungan, tumpukan, dan pengurutan cepat, tetapi merupakan Algoritma yang relatif sederhana, yang menjadikannya pilihan yang baik untuk menyortir daftar kurang dari 5000 item kecuali kecepatan penting. Ini juga merupakan pilihan yang baik untuk penyortiran berulang dari daftar yang lebih kecil.

Kasus terbaik dalam pengurutan Shell adalah ketika array sudah diurutkan dalam urutan yang benar. Jumlah perbandingan lebih sedikit. Waktu berjalannya Shell sort tergantung pada pilihan urutan kenaikan.

Kinerja

Kompleksitas kasus terburuk tergantung pada urutan celah. Paling terkenal: $O(n\log^2 n)$
Kompleksitas kasus terbaik: O(n)
Kompleksitas kasus rata-rata tergantung pada urutan celah

Kompleksitas ruang kasus terburuk: O(n)

2.9 GABUNGKAN SORTIR

Merge sort adalah contoh dari strategi membagi dan menaklukkan.

Catatan penting

- Penggabungan adalah proses menggabungkan dua file terurut menjadi satu file terurut yang lebih besar.
- Seleksi adalah proses membagi sebuah file menjadi dua bagian: k elemen terkecil dan n – k elemen terbesar.
- Pemilihan dan penggabungan adalah operasi yang berlawanan
    - seleksi membagi daftar menjadi dua daftar
    - menggabungkan menggabungkan dua file menjadi satu file
- Merge sort adalah pelengkap Quick sort
- Merge sort mengakses data secara berurutan
- Algoritma ini digunakan untuk menyortir daftar tertaut
- Merge sort tidak sensitif terhadap urutan awal inputnya
- Dalam Quick sort sebagian besar pekerjaan dilakukan sebelum panggilan rekursif. Penyortiran cepat dimulai dengan subfile terbesar dan diakhiri dengan subfile kecil dan sebagai hasilnya perlu tumpukan. Selain itu, algoritma ini tidak stabil. Merge sort membagi daftar menjadi dua bagian; kemudian setiap bagian ditaklukkan secara individual. Merge sort dimulai dengan subfile kecil dan diakhiri dengan subfile terbesar. Akibatnya tidak perlu stack. Algoritma ini stabil.

Penerapan

```
void Mergesort(int A[], int temp[], int left, int right) {
    int mid;
    if(right > left) {
            mid = (right + left) / 2;
            Mergesort(A, temp, left, mid);
            Mergesort(A, temp, mid+1, right);
            Merge(A, temp, left, mid+1, right);
    }
}
void Merge(int A[], int temp[], int left, int mid, int right) {
    int i, left_end, size, temp_pos;
    left_end = mid - 1;
    temp_pos = left;
    size = right - left + 1;
    while ((left <= left_end) && (mid <= right)) {
            if(A[left] <= A[mid]) {
                    temp[temp_pos] = A[left];
                    temp_pos = temp_pos + 1;
                    left = left +1;
```

```
            }
            else {
                    temp[temp_pos] = A[mid];
                    temp_pos = temp_pos + 1;
                    mid = mid + 1;
            }
    }
    while (left <= left_end) {
            temp[temp_pos] = A[left];
            left = left + 1;
            temp_pos = temp_pos + 1;
    }
    while (mid <= right) {
            temp[temp_pos] = A[mid];
            mid = mid + 1;
            temp_pos = temp_pos + 1;
    }
    for (i = 0; i <= size; i++) {
            A[right] = temp[right];
            right = right - 1;
    }
}
```

Analisis

Dalam Merge sort daftar input dibagi menjadi dua bagian dan ini diselesaikan secara rekursif. Setelah menyelesaikan sub masalah, mereka digabungkan dengan memindai sub masalah yang dihasilkan. Mari kita asumsikan T(n) adalah kompleksitas dari Merge sort dengan n elemen. Pengulangan untuk Merge Sort dapat didefinisikan sebagai:

Recurrence for Mergesort is $T(n) = 2T(\frac{n}{2}) + \Theta(n)$.
Using Master theorem, we get, $T(n) = \Theta(nlogn)$.

Catatan: *Untuk lebih jelasnya, lihat bab Divide and conquer.*

Kinerja:

Kompleksitas kasus terburuk: Θ (nlogn)
Kompleksitas kasus terbaik : Θ (nlogn)
Kompleksitas kasus rata-rata : Θ (nlogn)
Kompleksitas ruang kasus terburuk: Θ (n) tambahan

2.10 PENGURUTAN TUMPUKAN

Heapsort adalah algoritma pengurutan berbasis perbandingan dan merupakan bagian dari keluarga pengurutan seleksi. Meskipun praktiknya agak lebih lambat pada kebanyakan mesin daripada implementasi Quick sort yang baik, ia memiliki keuntungan dari runtime (nlogn) kasus terburuk yang lebih menguntungkan. Heapsort adalah Algoritma di tempat tetapi bukan jenis yang stabil.

Kinerja

Performa kasus terburuk: (nlogn)
Performa kasus terbaik: (nlogn)

Performa kasus rata-rata: (nlogn)
Kompleksitas ruang kasus terburuk: (n) total, (1) tambahan

Untuk detail lainnya tentang Heapsort, lihat bab Antrian Prioritas.

2.11 SORTIR CEPAT

Penyortiran cepat adalah contoh teknik Algoritma bagi-dan-taklukkan. Ini juga disebut semacam pertukaran partisi. Ini menggunakan panggilan rekursif untuk menyortir elemen, dan ini adalah salah satu algoritma terkenal di antara algoritma pengurutan berbasis perbandingan.

Divide: Array A[low ...high] dipartisi menjadi dua sub array yang tidak kosong A[low ...q] dan A[q + 1... tinggi], sehingga setiap elemen A[rendah ... tinggi] kurang dari atau sama dengan setiap elemen A[q + 1... tinggi]. Indeks q dihitung sebagai bagian dari prosedur partisi ini.

Conquer: Dua sub-array A[low ...q] dan A[q + 1 ...high] diurutkan berdasarkan panggilan rekursif ke Quick sort.

Algoritma

Algoritma rekursif terdiri dari empat langkah:

1) Jika ada satu atau tidak ada elemen dalam array yang akan diurutkan, kembalikan.
2) Pilih elemen dalam array untuk dijadikan sebagai titik "poros". (Biasanya elemen paling kiri dalam array digunakan.)
3) Pisahkan array menjadi dua bagian – satu dengan elemen yang lebih besar dari pivot dan yang lainnya dengan elemen yang lebih kecil dari pivot.
4) Ulangi algoritma secara rekursif untuk kedua bagian dari array asli.

Penerapan

```
void Quicksort( int A[], int low, int high ) {
   int pivot;
   /* Termination condition! */
   if( high > low ) {
          pivot = Partition( A, low, high );
          Quicksort( A, low, pivot-1 );
          Quicksort( A, pivot+1, high );
   }
}
int Partition( int A, int low, int high ) {
   int left, right, pivot_item = A[low];
   left = low;
   right = high;
   while ( left < right ) {
          /* Move left while item < pivot */
          while( A[left] <= pivot_item )
                 left++;
          /* Move right while item > pivot */
          while( A[right] > pivot_item )
```

```
                right--;
        if( left < right )
                swap(A,left,right);
    }
    /* right is final position for the pivot */
    A[low] = A[right];
    A[right] = pivot_item;
    return right;
}
```

Analisis

Mari kita asumsikan bahwa T(n) adalah kompleksitas dari Quick sort dan juga mengasumsikan bahwa semua elemen berbeda. Perulangan untuk T(n) bergantung pada dua ukuran submasalah yang bergantung pada elemen partisi. Jika pivot adalah elemen terkecil ke-i maka tepat (i – 1) item akan berada di bagian kiri dan (n – i) di bagian kanan. Mari kita menyebutnya sebagai i –split. Karena setiap elemen memiliki peluang yang sama untuk dipilih sebagai pivot, maka peluang untuk memilih elemen ke-i adalah $\frac{1}{n}$.

Kasus Terbaik: Setiap partisi membagi array menjadi dua dan memberikan

T(n) = 2T(n/2) + Θ (n) = Θ (nlogn), [menggunakan teorema utama *Divide and conquer*]

Kasus Terburuk: Setiap partisi memberikan pemisahan yang tidak seimbang dan kita mendapatkan

T(n) = T(n – 1) + Θ (n) = Θ (n^2)[menggunakan teorema master Pengurangan dan Penaklukan]

Kasus terburuk terjadi ketika daftar sudah diurutkan dan elemen terakhir dipilih sebagai pivot.

Kasus Rata-Rata: Dalam kasus rata-rata pengurutan Cepat, kita tidak tahu di mana pemisahan terjadi. Untuk alasan ini, kita mengambil semua nilai yang mungkin dari lokasi split, menambahkan semua kompleksitasnya dan membaginya dengan n untuk mendapatkan kompleksitas kasus rata-rata.

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n} \frac{1}{n}(runtime\ with\ i-split) + n + 1 \\
&= \frac{1}{n}\sum_{i=1}^{N}\big(T(i-1) + T(n-i)\big) + n + 1 \\
&\quad //\text{since we are dealing with best case we can assume } T(n-i) \text{ and } T(i-1) \text{ are equal} \\
&= \frac{2}{n}\sum_{i=1}^{n} T(i-1) + n + 1 \\
&= \frac{2}{n}\sum_{i=0}^{n-1} T(i) + n + 1
\end{aligned}
$$

Kalikan kedua ruas dengan n.

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + n^2 + n$$

Rumus yang sama untuk n – 1.

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + (n-1)^2 + (n-1)$$

Kurangi rumus n – 1 dari n.

$$nT(n) - (n-1)T(n-1) = 2\sum_{i=0}^{n-1} T(i) + n^2 + n - (2\sum_{i=0}^{n-2} T(i) + (n-1)^2 + (n-1))$$
$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n$$
$$nT(n) = (n+1)T(n-1) + 2n$$

Bagi dengan n(n + 1).

$$\begin{aligned}
\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2}{n+1} \\
&= \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\cdot \\
&\cdot \\
&= \mathrm{O}(1) + 2\textstyle\sum_{i=3}^{n} \frac{1}{i} \\
&= \mathrm{O}(1) + \mathrm{O}(2logn) \\
\frac{T(n)}{n+1} &= \mathrm{O}(logn) \\
T(n) &= \mathrm{O}\big((n+1)\,logn\big) = \mathrm{O}(nlogn)
\end{aligned}$$

Kompleksitas Waktu, T(n) = O(nlogn).

Kinerja

Kompleksitas kasus terburuk: O(n^2)
Kompleksitas kasus terbaik: O(nlogn)
Kompleksitas kasus rata-rata: O(nlogn)
Ruang kasus terburuk Kompleksitas: O(1)

Pengurutan cepat secara acak

Dalam perilaku kasus rata-rata dari *Quick sort*, kita berasumsi bahwa semua permutasi dari angka input memiliki kemungkinan yang sama. Namun, kita tidak bisa selalu mengharapkannya untuk bertahan. Kita dapat menambahkan pengacakan ke suatu algoritma untuk mengurangi kemungkinan mendapatkan kasus terburuk dalam *Quick sort*.

Ada dua cara untuk menambahkan pengacakan di *Quick sort:* baik dengan menempatkan data input secara acak dalam array atau dengan memilih elemen secara acak dalam data input untuk pivot. Pilihan kedua lebih mudah untuk dianalisis dan diimplementasikan. Perubahan hanya akan dilakukan pada algoritma partisi.

Dalam *Quick sort* normal, elemen pivot selalu menjadi elemen paling kiri dalam daftar yang akan diurutkan. Alih-alih selalu menggunakan A[rendah] sebagai pivot, kita akan menggunakan elemen yang dipilih secara acak dari subarray A[rendah..tinggi] dalam versi acak dari *Quick sort.* Hal ini dilakukan dengan menukar elemen A[rendah] dengan elemen yang dipilih secara acak dari A[rendah..tinggi]. Ini memastikan bahwa elemen pivot memiliki kemungkinan yang sama untuk menjadi salah satu elemen tinggi – rendah + 1 di subarray.

Karena elemen pivot dipilih secara acak, kita dapat mengharapkan pembagian array input menjadi seimbang secara rata-rata. Ini dapat membantu dalam mencegah perilaku terburuk dari quick sort yang terjadi pada partisi yang tidak seimbang. Meskipun versi acak meningkatkan kompleksitas kasus terburuk, kompleksitas kasus terburuknya masih O(n2).

Salah satu cara untuk meningkatkanAcak – Pengurutan cepat adalah memilih pivot untuk mempartisi dengan lebih hati-hati daripada memilih elemen acak dari larik. Salah satu pendekatan yang umum adalah memilih pivot sebagai median dari kumpulan 3 elemen yang dipilih secara acak dari array.

2.12 PENGURUTAN POHON

Pengurutan pohon menggunakan pohon pencarian biner. Ini melibatkan pemindaian setiap elemen input dan menempatkannya ke posisi yang tepat di pohon pencarian biner. Ini memiliki dua fase:

- Tahap pertama adalah membuat pohon pencarian biner menggunakan elemen array yang diberikan.
- Fase kedua melintasi pohon pencarian biner yang diberikan secara berurutan, sehingga menghasilkan array yang diurutkan.

Kinerja

Jumlah rata-rata perbandingan untuk metode ini adalah O(nlogn). Tetapi dalam kasus terburuk, jumlah perbandingan dikurangi dengan O(n2), kasus yang muncul ketika pohon pengurutan adalah pohon miring.

2.13 PERBANDINGAN ALGORITMA PENGURUTAN

Tabel 2.2 perbandingan algoritma pengurutan

Nama	Kasus Rata-rata	Kasus terburuk	Bantu	Apakah Stabil?	Catatan lainnya
Gelembung	$O(n^2)$	$O(n^2)$	1	Ya	Kode kecil
Pilihan	$O(n^2)$	$O(n^2)$	1	Tidak	Stabilitas tergantung pada implementasi
Insersi	$O(n^2)$	$O(n^2)$	1	Ya	Kasus rata-rata juga O(n+d), di mana d adalah bilangan inversi.
Kerang	-	$O(n\log^2 n)$	1	Tidak	
Gabungkan sort	O(nlogn)	O(nlogn)	Bergantung	Ya	
Sortir tumpukan	O(nlogn)	O(nlogn)	1	Tidak	

Sortir cepat	O(nlogn)	$O(n^2)$	O(nlogn)	Bergantung	Dapat diimplementasikan sebagai jenis yang stabil tergantung pada bagaimana pivot ditangani.
Jenis pohon	O(nlogn)	$O(n^2)$	O(n)	Bergantung	Dapat diimplementasikan sebagai jenis yang stabil.

Catatan: n menunjukkan jumlah elemen dalam input.

2.14 ALGORITMA PENGURUTAN LINIER

Di bagian sebelumnya, kita telah melihat banyak contoh algoritma pengurutan berbasis perbandingan. Di antara mereka, penyortiran berbasis perbandingan terbaik memiliki kompleksitas O(nlogn). Pada bagian ini, kita akan membahas jenis algoritma lainnya: Algoritma Pengurutan Linier. Untuk meningkatkan kompleksitas waktu pengurutan algoritma ini, kita membuat beberapa asumsi tentang input. Beberapa contoh Algoritma Pengurutan Linier adalah:

- Menghitung Sortir
- Sortir Keranjang
- Urutkan Radix

2.15 MENGHITUNG SORTIR

Menghitung pengurutan bukan algoritma pengurutan perbandingan dan memberikan kompleksitas O(n) untuk pengurutan. Untuk mencapai kompleksitas O(n), pengurutan pencacahan mengasumsikan bahwa setiap elemen adalah bilangan bulat dalam rentang 1 hingga K, untuk beberapa bilangan bulat K. Ketika if = O(n), pengurutan pencacahan berjalan dalam waktu O(n). Ide dasar dari Counting sort adalah untuk menentukan, untuk setiap elemen input X, jumlah elemen yang kurang dari X. Informasi ini dapat digunakan untuk menempatkannya langsung ke posisi yang benar. Misalnya, jika 10 elemen kurang dari X, maka X termasuk ke posisi 11 dalam *output*.

Pada kode di bawah ini, A[0 ..n – 1] adalah array input dengan panjang n. Dalam Counting sort kita membutuhkan dua array lagi: mari kita asumsikan array B[0 ..n – 1] berisi *output* yang diurutkan dan array C[0 ..K – 1] menyediakan penyimpanan sementara.

```
void CountingSort (int A[], int n, int B[], int K) {
    int C[K], i, j;
    //Complexity: O(K)
    for (i =0 ; i<K; i++)
        C[i] = 0;
    //Complexity: O(n)
    for (j =0 ; j<n; j++)
        C[A[j]] = C[A[j]] + 1;
    //C[i] now contains the number of elements equal to i
    //Complexity: O(K)
    for (i =1 ; i<K; i++)
        C[i] = C[i] + C[i-1];
    // C[i] now contains the number of elements ≤ i
    //Complexity: O(n)
    for (j = n-1; j>=0; j--) {
        B[C[A[j]]] = A[j];
        C[A[j]] = C[A[j]] - 1;
    }
}
```

Kompleksitas Total:

$$O(K) + O(n) + O(K) + O(n) = O(n) \text{ jika } K = O(n).$$

Kompleksitas Ruang:

$$O(n) \text{ jika } K = O(n).$$

Catatan: *Penghitungan bekerja dengan baik jika K =O(n). Jika tidak, kerumitannya akan lebih besar.*

2.16 PENGURUTAN KERANJANG (ATAU PENGURUTAN BIN)

Seperti Counting sort, Bucket sort juga memberlakukan batasan pada input untuk meningkatkan kinerja. Dengan kata lain, Bucket sort berfungsi dengan baik jika input diambil dari set tetap. Bucket sort adalah generalisasi dari Counting Sort. Misalnya, asumsikan bahwa semua elemen input dari {0, 1, . . . , K – 1}, yaitu himpunan bilangan bulat dalam interval [0, K – 1]. Itu berarti, K adalah jumlah elemen jauh di input. Bucket sort menggunakan K counter. Penghitung ke-i melacak jumlah kemunculan elemen ke-i. Penyortiran ember dengan dua ember secara efektif merupakan versi Penyortiran cepat dengan dua ember.

Untuk bucket sort, fungsi hash yang digunakan untuk mempartisi elemen harus sangat baik dan harus menghasilkan hash terurut: if i < k maka hash(i) < hash(k). Kedua, elemen yang akan diurutkan harus terdistribusi secara merata.

Selain yang disebutkan di atas, pengurutan ember sebenarnya sangat bagus mengingat pengurutan penghitungan secara wajar berbicara tentang batas atasnya. Dan menghitung sort sangat cepat. Perbedaan khusus untuk bucket sort adalah bahwa ia menggunakan fungsi hash untuk mempartisi kunci dari array input, sehingga beberapa kunci

dapat di-hash ke bucket yang sama. Oleh karena itu setiap ember harus secara efektif menjadi daftar yang dapat ditumbuhkan; mirip dengan radix sort.

Pada kode di bawah ini insertionsort digunakan untuk mengurutkan setiap bucket. Ini untuk menanamkan bahwa algoritma pengurutan ember tidak menentukan teknik pengurutan mana yang akan digunakan pada ember. Seorang programmer dapat memilih untuk terus menggunakan sortir ember pada setiap ember sampai koleksi diurutkan (dengan cara program sortir radix di bawah). Apapun metode pengurutan yang digunakan pada , bucket sort masih cenderung ke arah O(n).

```
#define BUCKETS 10
void BucketSort(int A[], int array_size) {
 int i, j, k;
 int buckets[BUCKETS];
 for(j =0; j < BUCKETS; j++)
        buckets[j] = 0;
 for(i =0; i < array_size; i++)
        ++ buckets[A[i]];
 for(i =0, j=0; j < BUCKETS; j++)
        for(k = buckets[j];k > 0; --k)
            A[i++] = j;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

2.17 PENGURUTAN RADIX

Mirip dengan Counting sort dan Bucket sort, algoritma pengurutan ini juga mengasumsikan beberapa jenis informasi tentang elemen input. Misalkan nilai input yang akan diurutkan berasal dari basis d. Itu berarti semua angka adalah angka d-digit.

Dalam pengurutan Radix, pertama-tama urutkan elemen berdasarkan digit terakhir [digit paling tidak signifikan]. Hasil ini sekali lagi diurutkan berdasarkan digit kedua [digit paling tidak signifikan berikutnya]. Lanjutkan proses ini untuk semua digit hingga kita mencapai digit paling signifikan. Gunakan semacam stabil untuk mengurutkannya berdasarkan digit terakhir. Kemudian stabil mengurutkannya berdasarkan digit signifikan kedua, kemudian dengan yang ketiga, dll. Jika kita menggunakan pengurutan penghitung sebagai pengurutan stabil, total waktunya adalah O(nd) O(n).

Algoritma:

1) Ambil angka penting terkecil dari setiap elemen.
2) Urutkan daftar elemen berdasarkan digit itu, tetapi pertahankan urutan elemen dengan digit yang sama (ini adalah definisi pengurutan stabil).
3) Ulangi pengurutan dengan setiap digit yang lebih signifikan.

Kecepatan pengurutan Radix tergantung pada operasi dasar bagian dalam. Jika operasinya tidak cukup efisien, Radix sort bisa lebih lambat dari algoritma lain seperti Quick sort dan Merge sort. Operasi ini mencakup fungsi insert dan delete dari sub-daftar dan proses

mengisolasi digit yang kita inginkan. Jika angka-angka tersebut tidak memiliki panjang yang sama maka tes diperlukan untuk memeriksa digit tambahan yang perlu disortir. Ini bisa menjadi salah satu bagian paling lambat dari jenis Radix dan juga salah satu yang paling sulit untuk dibuat efisien.

Karena pengurutan Radix bergantung pada angka atau huruf, jenis ini kurang fleksibel dibandingkan jenis lainnya. Untuk setiap jenis data yang berbeda, pengurutan Radix perlu ditulis ulang, dan jika urutan pengurutan berubah, pengurutan perlu ditulis ulang lagi. Singkatnya, pengurutan Radix membutuhkan lebih banyak waktu untuk menulis, dan sangat sulit untuk menulis pengurutan Radix tujuan umum yang dapat menangani semua jenis data.

Untuk banyak program yang membutuhkan pengurutan cepat, pengurutan Radix adalah pilihan yang baik. Namun, ada jenis yang lebih cepat, yang merupakan salah satu alasan mengapa jenis Radix tidak digunakan sebanyak jenis lainnya. Kompleksitas Waktu: O(nd) O(n), jika d kecil.

2.18 PENYORTIRAN EKSTERNAL

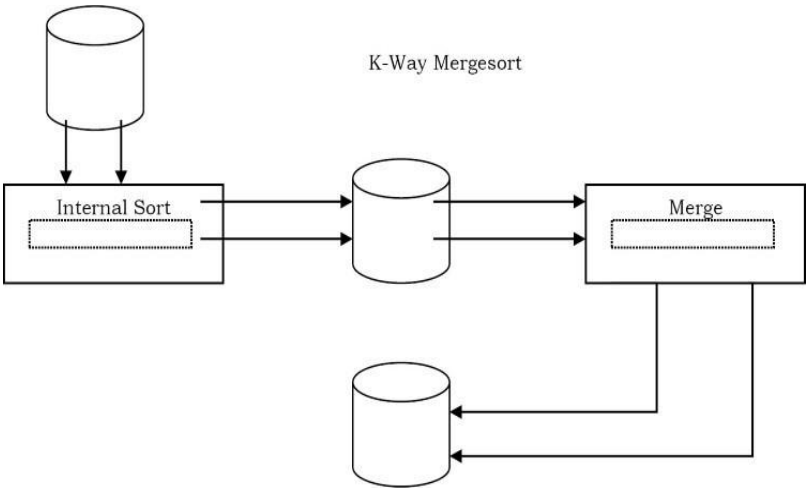
Penyortiran eksternal adalah istilah umum untuk kelas Algoritma pengurutan yang dapat menangani sejumlah besar data. Algoritma pengurutan eksternal ini berguna ketika file terlalu besar dan tidak dapat masuk ke memori utama.

Seperti algoritma pengurutan internal, ada sejumlah algoritma untuk pengurutan eksternal. Salah satu algoritma tersebut adalah External Mergesort. Dalam praktiknya, Algoritma pengurutan eksternal ini dilengkapi dengan pengurutan internal.

Penggabungan Eksternal Sederhana

Sejumlah *record* dari setiap tape dibaca ke memori utama, disortir menggunakan internal sort, dan kemudian dikeluarkan ke tape. Demi kejelasan, mari kita asumsikan bahwa 900 megabyte data perlu diurutkan hanya menggunakan 100 megabyte RAM.

1) Baca 100MB data ke dalam memori utama dan urutkan dengan beberapa metode konvensional (misalkan Quick sort).
2) Tulis data yang diurutkan ke disk.
3) Ulangi langkah 1 dan 2 sampai semua data diurutkan dalam potongan 100MB. Sekarang kita perlu menggabungkannya menjadi satu file *output* yang diurutkan.
4) Baca 10MB pertama dari setiap potongan yang diurutkan (sebut saja buffer input) di memori utama (total 90MB) dan alokasikan 10MB sisanya untuk buffer *output*.
5) Lakukan 9-way Mergesort dan simpan hasilnya di *output* buffer. Jika buffer *output* penuh, tulis ke file yang diurutkan terakhir. Jika salah satu dari 9 buffer input kosong, isi dengan 10MB berikutnya dari potongan terurut 100MB yang terkait; atau jika tidak ada lagi data dalam potongan yang diurutkan, tandai sebagai habis dan jangan gunakan untuk menggabungkan.



Gambar 2.2 *K-Way Mergesort*

Algoritma di atas dapat digeneralisasi dengan mengasumsikan bahwa jumlah data yang akan diurutkan melebihi memori yang tersedia dengan faktor K. Kemudian, K potongan data perlu diurutkan dan penggabungan K-way harus diselesaikan.

Jika X adalah jumlah memori utama yang tersedia, akan ada K buffer input dan 1 buffer *output* dengan ukuran X/(K + 1) masing-masing. Tergantung pada berbagai faktor (seberapa cepat hard drive?) kinerja yang lebih baik dapat dicapai jika buffer *output* dibuat lebih besar (misalnya, dua kali lebih besar dari satu buffer input).

Kompleksitas pengurutan Penggabungan Eksternal 2 arah: Di setiap lintasan kita membaca + menulis setiap halaman dalam file. Mari kita asumsikan bahwa ada n halaman dalam file. Itu berarti kita membutuhkan log$n\rceil$ + 1 jumlah operan. Total biayanya adalah 2n($\lceil \log n \rceil$ + 1).

2.19 PENYORTIRAN: MASALAH & SOLUSI

Soal-1 Diberikan sebuah larik A[0...n– 1] dari n bilangan yang berisi pengulangan beberapa bilangan. Berikan algoritma untuk memeriksa apakah ada elemen yang berulang atau tidak. Asumsikan bahwa kita tidak diperbolehkan untuk menggunakan ruang tambahan (yaitu, kita dapat menggunakan beberapa variabel sementara, penyimpanan O(1)).

Solusi: Karena kita tidak diperbolehkan menggunakan ruang ekstra, salah satu cara sederhana adalah memindai elemen satu per satu dan untuk setiap elemen periksa apakah elemen tersebut muncul di elemen yang tersisa. Jika kita menemukan kecocokan, kita mengembalikan true.

```
int CheckDuplicatesInArray(in A[], int n) {
    for (int i  =  0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            if(A[i]==A[j])
                reutrn true;
    return false;
}
```

Setiap iterasi bagian dalam, loop berindeks j menggunakan ruang O(1), dan untuk nilai tetap i, loop j dieksekusi n – i kali. Loop luar dieksekusi n – 1 kali, sehingga seluruh fungsi menggunakan waktu yang sebanding dengan

$$\sum_{i=1}^{n-1} n - i = n(n-1) - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} = O(n^2)$$

Kompleksitas Waktu: $O(n^2)$.

Kompleksitas Ruang: O(1).

Soal-2 Bisakah kita meningkatkan kompleksitas waktu Soal-1?

Solusi: Ya, menggunakan teknik penyortiran.

```
int CheckDuplicatesInArray(in A[], int n) {
    //for heap sort algorithm refer Priority Queues chapter
    Heapsort( A, n );
    for (int i  =  0; i < n-1; i++)
            if(A[i]==A[i+1])
                reutrn true;
    return false;
}
```

Fungsi Heapsort membutuhkan waktu O(nlogn), dan membutuhkan ruang O(1). Pemindaian jelas membutuhkan n – 1 iterasi, setiap iterasi menggunakan waktu O(1). Waktu keseluruhan adalah O(nlogn + n) = O(nlogn).

Kompleksitas Waktu: O(nlogn).

Kompleksitas Ruang: O(1).

Catatan: *Untuk variasi masalah ini, lihat bab Pencarian.*

Soal-3 Diberikan sebuah larik A[0 ...n – 1], di mana setiap elemen larik mewakili satu suara dalam pemilihan. Asumsikan bahwa setiap suara diberikan sebagai bilangan bulat yang mewakili ID calon terpilih. Berikan algoritma untuk menentukan siapa yang memenangkan pemilihan.

Solusi: Masalah ini tidak lain adalah menemukan elemen yang berulang kali paling banyak. Solusinya mirip dengan solusi Soal-1: lacak penghitung.

```
int CheckWhoWinsTheElection(in A[], int n) {
    int i, j, counter = 0, maxCounter = 0, candidate;
    candidate = A[0];
    for (i  =  0; i < n; i++) {
            candidate = A[i];
            counter = 0;
            for (j   = i + 1; j < n; j++) {
                    if(A[i]==A[j]) counter++;
            }
            if(counter > maxCounter) {
                    maxCounter = counter;
                    candidate = A[i];
            }
    }
    return candidate;
}
```

Kompleksitas Waktu: $O(n^2)$.

Kompleksitas Ruang: O(1).

Catatan: *Untuk variasi masalah ini, lihat bab Pencarian.*

Soal-4 Bisakah kita meningkatkan kompleksitas waktu Soal-3? Asumsikan kita tidak memiliki ruang ekstra.

Solusi: Ya. Pendekatannya adalah mengurutkan suara berdasarkan ID kandidat, kemudian memindai array yang diurutkan dan menghitung kandidat mana yang sejauh ini memiliki suara terbanyak. Kita hanya perlu mengingat pemenangnya, jadi kita tidak membutuhkan struktur data yang cerdas. Kita dapat menggunakan Heapsort karena ini adalah algoritma pengurutan di tempat.

```
int CheckWhoWinsTheElection(in A[], int n) {
    int i, j, currentCounter = 1, maxCounter = 1;
    int currentCandidate, maxCandidate;
    currentCandidate = maxCandidate= A[0];
    //for heap sort algorithm refer Priority Queues Chapter
    Heapsort( A, n );
    for (int i  =  1; i <= n; i++) {
            if( A[i] == currentCandidate)
                    currentCounter ++;
            else {
                    currentCandidate = A[i];
                    currentCounter = 1;
            }
            if(currentCounter > maxCounter)
                    maxCounter = currentCounter;
            else {
                    maxCandidate = currentCandidate;
                    maxCounter = currentCounter;
            }
    }
    return candidate;
}
```

Karena kompleksitas waktu Heapsort adalah O(nlogn) dan di tempat, ia hanya menggunakan tambahan O(1) penyimpanan selain array input. Pemindaian

array yang diurutkan melakukan pengkondisian waktu-konstan n – 1 kali, sehingga menggunakan waktu O(n). Batas waktu keseluruhan adalah O(nlogn).

Soal-5 Bisakah kita lebih meningkatkan kompleksitas waktu Soal-3?

Solusi: Pada soal yang diberikan, jumlah kandidat lebih sedikit tetapi jumlah suara sangat besar. Untuk masalah ini kita dapat menggunakan pengurutan penghitungan.
Kompleksitas Waktu: O(n), n adalah jumlah suara (elemen) dalam array.
Kompleksitas Ruang: O(k), k adalah jumlah calon peserta pemilu.

Soal-6 Diberikan sebuah larik A dari n elemen, yang masing-masing merupakan bilangan bulat dalam rentang [1, n2], bagaimana kita mengurutkan larik dalam waktu O(n)?

Solusi: Jika kita mengurangi setiap angka dengan 1, maka kita mendapatkan rentang [0, n2 – 1]. Jika kita menganggap semua bilangan sebagai basis 2 –digit n. Setiap digit berkisar dari 0 hingga n2 – 1. Urutkan ini menggunakan radix sort. Ini hanya menggunakan dua panggilan untuk menghitung sort. Terakhir, tambahkan 1 ke semua angka. Karena ada 2 panggilan, kompleksitasnya adalah $O(2n) \approx O(n)$.

Soal-7 Untuk Soal-6, bagaimana jika range [1... n3]?

Solusi: Jika kita mengurangi setiap angka dengan 1 maka kita mendapatkan rentang [0, n3 – 1]. Mempertimbangkan semua angka sebagai basis 3 digit n: setiap digit berkisar dari 0 hingga n3 – 1. Urutkan ini menggunakan pengurutan radix. Ini hanya menggunakan tiga panggilan untuk menghitung sortir. Terakhir, tambahkan 1 ke semua angka. Karena ada 3 panggilan, kompleksitasnya adalah $O(3n) \approx O(n)$.

Soal-8 Diberikan sebuah larik dengan n bilangan bulat, masing-masing bernilai kurang dari n100, dapatkah ia diurutkan dalam waktu linier?

Solusi: Ya. Alasannya sama seperti pada Soal-6 dan Soal-7.

Soal-9 Misalkan A dan B masing-masing dua larik dengan n elemen. Diberikan nomor K, berikan algoritma waktu O(nlogn) untuk menentukan apakah ada a A dan b B sedemikian rupa sehingga a + b = K

Solusi: Karena kita membutuhkan O(nlogn), itu memberi kita pointer yang perlu kita urutkan. Jadi, kita akan melakukannya.

```
int Find( int A[], int B[], int n, K ) {
    int i, c;
    Heapsort( A, n );                   // O(nlogn)
    for (i =0; i< n; i++) {             // O(n)
            c = k-B[i];                 // O(1)
            if(BinarySearch(A, c))      // O(logn)
                    return 1;
    }
    return 0;
}
```

Catatan: *Untuk variasi masalah ini, lihat bab Pencarian.*

Soal-10 Misalkan A,B dan C masing-masing terdiri dari tiga larik dengan n elemen. Diberikan nomor K, berikan algoritma waktu O(nlogn) untuk menentukan apakah ada a A, b B dan c C sedemikian rupa sehingga a + b + c = K.

Solusi: Lihat bab Pencarian.

Soal-11 Diberikan sebuah array dengan n elemen, dapatkah kita menampilkan elemen K dalam urutan terurut mengikuti median dalam urutan terurut dalam waktu O(n + KlogK).

Solusi: Ya. Cari median dan partisi median. Dengan ini kita dapat menemukan semua elemen yang lebih besar dari itu. Sekarang temukan elemen terbesar ke-K di set ini dan partisi itu; dan dapatkan semua elemen kurang dari itu. Keluarkan daftar yang diurutkan dari set elemen terakhir. Jelas, operasi ini membutuhkan waktu O(n + KlogK).

Soal-12 Pertimbangkan Algoritma pengurutan: Bubble sort, Insertion sort, Selection sort, Merge sort, Heap sort, dan Quick sort. Manakah dari ini yang stabil?

Solusi: Mari kita asumsikan bahwa A adalah array yang akan diurutkan. Juga, katakanlah R dan S memiliki kunci yang sama dan R muncul lebih awal dalam array daripada S. Itu berarti, R berada di A[i] dan S berada di A[j], dengan i < j. Untuk menunjukkan algoritma yang stabil, dalam *output* yang diurutkan R harus mendahului S.

Jenis gelembung: Ya. Elemen mengubah urutan hanya ketika catatan yang lebih kecil mengikuti yang lebih besar. Karena S tidak lebih kecil dari R, maka S tidak dapat mendahuluinya.

Sortir pilihan: Tidak. Ini membagi larik menjadi bagian yang diurutkan dan tidak diurutkan dan secara iteratif menemukan nilai minimum di bagian yang tidak diurutkan. Setelah menemukan minimum x, jika Algoritma memindahkan x ke bagian array yang diurutkan melalui swap, maka elemen yang ditukar adalah R yang kemudian dapat dipindahkan ke belakang S. Ini akan membalikkan posisi

R dan S, jadi secara umum tidak stabil. Jika swapping dihindari, itu bisa dibuat stabil tetapi biaya pada waktunya mungkin akan sangat signifikan.

Jenis penyisipan: Ya. Seperti yang disajikan, ketika S akan dimasukkan ke dalam subarray terurut A[1..j – 1], hanya *record* yang lebih besar dari S yang digeser. Jadi R tidak akan digeser selama penyisipan S dan karenanya akan selalu mendahuluinya.

Merge sort: Ya, Dalam kasus *record* dengan kunci yang sama, *record* di subarray kiri mendapat preferensi. Itu adalah catatan yang datang lebih dulu dalam array yang tidak disortir. Akibatnya, mereka akan mendahului catatan selanjutnya dengan kunci yang sama.

Heap sort: Tidak. Misalkan i = 1 dan R dan S merupakan dua *record* dengan kunci terbesar di input. Kemudian R akan tetap berada di lokasi 1 setelah array di-heapified, dan akan ditempatkan di lokasi n pada iterasi pertama Heapsort. Jadi S akan mendahului R dalam *output*.

Sortir cepat: Tidak. Langkah partisi dapat menukar lokasi *record* berkali-kali, dan dengan demikian dua *record* dengan kunci yang sama dapat bertukar posisi di hasil akhir.

Soal-13 Pertimbangkan algoritma pengurutan yang sama dengan Soal-12. Manakah dari mereka yang ada di tempat?

Solusi:

Bubble sort: Ya, karena hanya dua bilangan bulat yang diperlukan.

Jenis penyisipan: Ya, karena kita perlu menyimpan dua bilangan bulat dan satu catatan.

Jenis pilihan: Ya. Algoritma ini kemungkinan akan membutuhkan ruang untuk dua bilangan bulat dan satu catatan.

Merge sort: Tidak. Array perlu melakukan penggabungan. (Jika data dalam bentuk daftar tertaut, penyortiran dapat dilakukan di tempat, tetapi ini adalah modifikasi yang tidak sepele.)

Heap sort: Ya, karena heap dan array yang diurutkan sebagian menempati ujung yang berlawanan dari array input.

Quicksort: Tidak, karena bersifat rekursif dan menyimpan catatan aktivasi O(logn) di tumpukan. Memodifikasinya menjadi non-rekursif layak tetapi nontrivial.

Soal-14 Di antara algoritma Quick sort, Insertion sort, Selection sort, dan Heap sort, mana yang membutuhkan jumlah swap minimum?

Solusi: Sortir seleksi – hanya perlu n swap (lihat bagian teori).

Soal-15 Berapa jumlah minimum perbandingan yang diperlukan untuk menentukan apakah suatu bilangan bulat muncul lebih dari n/2 kali dalam susunan n bilangan bulat yang diurutkan?

Solusi: Lihat bab Pencarian.

Soal-16 Mengurutkan array dari 0, 1 dan 2: Diberikan sebuah array A[] yang terdiri dari 0, 1 dan 2, berikan algoritma untuk pengurutan A[]. Algoritma harus menempatkan semua 0 terlebih dahulu, lalu semua 1 dan semua 2 terakhir.

Contoh: Masukan = {0,1,1,0,1,2,1,2,0,0,0,1}, Keluaran = {0,0,0,0,0,1,1,1,1,2,2}

Solusi: Gunakan Counting sort. Karena hanya ada tiga elemen dan nilai maksimumnya adalah 2, kita memerlukan array sementara dengan 3 elemen.

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1).

Catatan: *Untuk variasi masalah ini, lihat bab Pencarian.*

Soal-17 Apakah ada cara lain untuk menyelesaikan Soal-16?

Solusi: Menggunakan Quick dort. Karena kita tahu bahwa hanya ada 3 elemen, 0,1 dan 2 dalam array, kita dapat memilih 1 sebagai elemen pivot untuk Quick sort. Sortir cepat menemukan tempat yang tepat untuk 1 dengan memindahkan semua 0 ke kiri 1 dan semua 2 ke kanan 1. Untuk melakukan ini, hanya menggunakan satu pemindaian.

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1).

Catatan: *Untuk algoritma yang efisien, lihat bab Pencarian.*

Soal-18 Bagaimana kita menemukan angka yang muncul paling banyak dalam sebuah array?

Solusi: Salah satu pendekatan sederhana adalah mengurutkan array yang diberikan dan memindai array yang diurutkan. Saat memindai, lacak elemen yang muncul dalam jumlah maksimum.

Algoritma:

```
QuickSort(A, n);
int i, j, count=1, Number=A[0], j=0;
for(i=0;i<n;i++) {
        if(A[j]==A) {
                count++;
                Number=A[j];
        }
        j=i;
}
printf("Number:%d, count:%d", Number, count);
```

Kompleksitas Waktu = Waktu Penyortiran + Waktu Pemindaian = O(nlogn) +O(n) = O(nlogn).
Kompleksitas Ruang: O(1).
Catatan: *Untuk variasi masalah ini, lihat bab Pencarian.*

Soal-19 Apakah ada cara lain untuk menyelesaikan Soal-18?

Solusi: Menggunakan Pohon Biner. Buat pohon biner dengan jumlah bidang tambahan yang menunjukkan berapa kali elemen muncul di input. Katakanlah kita telah membuat Pohon Pencarian Biner [BST]. Sekarang, lakukan traversal In-Order dari pohon. Traversal In-Order dari BST menghasilkan daftar yang diurutkan. Saat melakukan traversal In-Order, lacak elemen maksimum.
Kompleksitas Waktu: O(n) + O(n) ≈ O(n). Parameter pertama adalah untuk membangun BST dan parameter kedua adalah untuk Inorder Traversal.
Kompleksitas Ruang: O(2n) ≈ O(n), karena setiap node di BST membutuhkan dua pointer tambahan.

Soal-20 Apakah ada cara lain untuk menyelesaikan Soal-18?

Solusi: Menggunakan Tabel Hash. Untuk setiap elemen dari array yang diberikan, kita menggunakan penghitung, dan untuk setiap kemunculan elemen, kita menambahkan penghitung yang sesuai. Pada akhirnya kita bisa mengembalikan elemen yang memiliki penghitung maksimum.
Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n). Untuk membangun tabel hash kita membutuhkan O(n).
Catatan: *Untuk algoritma yang efisien, lihat bab Pencarian.*

Soal-21 Diberikan file 2 GB dengan satu string per baris, algoritma pengurutan mana yang akan kita gunakan untuk mengurutkan file dan mengapa?

Solusi: Ketika kita memiliki batas ukuran 2GB, itu berarti kita tidak dapat membawa semua data ke dalam memori utama.

Algoritma: Berapa banyak memori yang kita miliki? Mari kita asumsikan kita memiliki memori X MB yang tersedia. Bagilah file menjadi potongan K, di mana X * K ~ 2 GB.

- Bawa setiap potongan ke dalam memori dan urutkan baris seperti biasa (algoritma O(nlogn) apa pun).
- Simpan baris kembali ke file.
- Sekarang bawa potongan berikutnya ke dalam memori dan urutkan.
- Setelah selesai, gabungkan satu per satu; dalam kasus penyelesaian satu set, bawa lebih banyak data dari potongan tertentu.

Algoritma di atas juga dikenal sebagai pengurutan eksternal. Langkah 3 – 4 dikenal sebagai K-way merge. Gagasan di balik pengurutan eksternal adalah ukuran data. Karena datanya sangat besar dan kita tidak dapat membawanya ke memori, kita harus menggunakan Algoritma pengurutan berbasis disk.

Soal-22 Hampir diurutkan: Diberikan sebuah larik n elemen, masing-masing dengan posisi paling banyak K dari posisi targetnya, buatlah algoritma yang mengurutkan dalam waktu O(n logK).

Solusi: Bagi elemen menjadi n/K grup berukuran K, dan urutkan setiap bagian dalam waktu O(KlogK), misalkan menggunakan Mergesort. Ini mempertahankan properti bahwa tidak ada elemen yang lebih dari K elemen di luar posisinya. Sekarang, gabungkan setiap blok elemen K dengan blok di sebelah kirinya.

Soal-23 Apakah ada cara lain untuk menyelesaikan Soal-22?

Solusi: Masukkan elemen K pertama ke dalam tumpukan biner. Masukkan elemen berikutnya dari array ke dalam heap, dan hapus elemen minimum dari heap. Mengulang.

Soal-24 Menggabungkan K daftar terurut: Diberikan K daftar terurut dengan total n elemen, berikan algoritma O(nlogK) untuk menghasilkan daftar terurut dari semua n elemen.

Solusi: Algoritma Sederhana untuk menggabungkan K daftar terurut: Pertimbangkan grup yang masing-masing memiliki $\frac{n}{K}$ elemen. Ambil daftar pertama dan gabungkan dengan daftar kedua menggunakan Algoritma waktu linier untuk menggabungkan dua daftar yang diurutkan, seperti Algoritma penggabungan yang digunakan dalam pengurutan gabungan. Kemudian, gabungkan daftar elemen yang dihasilkan $\frac{2n}{\mathrm{K}}$ dengan daftar ketiga, lalu gabungkan daftar elemen

yang dihasilkan $\frac{3n}{K}$ dengan daftar keempat. Ulangi ini sampai kita mendapatkan satu daftar terurut dari semua n elemen.

Kompleksitas Waktu: Dalam setiap iterasi kita menggabungkan elemen K.

$$T(n) = \frac{2n}{K} + \frac{3n}{K} + \frac{4n}{K} + \cdots \frac{Kn}{K}(n) = \frac{n}{K}\sum_{i=2}^{K} i$$

$$T(n) = \frac{n}{K}\left[\frac{K(K+1)}{2}\right] \approx O(nK)$$

Soal-25 Dapatkah kita meningkatkan kompleksitas waktu Soal-24?

Solusi: Salah satu metode adalah dengan berulang kali memasangkan daftar dan kemudian menggabungkan setiap pasangan. Metode ini juga dapat dilihat sebagai komponen ekor dari jenis penggabungan eksekusi, di mana analisisnya jelas. Ini disebut Metode Turnamen. Kedalaman maksimum Metode Turnamen adalah logK dan dalam setiap iterasi kita memindai semua n elemen.

Kompleksitas Waktu; O(nlogK).

Soal-26 Apakah ada cara lain untuk menyelesaikan Soal-24?

Solusi: Metode lainnya adalah dengan menggunakan antrian prioritas hujan untuk elemen minimum dari setiap daftar if. Pada setiap langkah, kita mengeluarkan minimum yang diekstraksi dari antrian prioritas, menentukan dari daftar K mana ia berasal, dan memasukkan elemen berikutnya dari daftar itu ke dalam antrian prioritas. Karena kita menggunakan antrian prioritas, kedalaman maksimum antrian prioritas adalah logK.

Kompleksitas Waktu; O(nlogK).

Soal-27 Metode penyortiran mana yang lebih baik untuk Daftar Tertaut?

Solusi: Merge Sort adalah pilihan yang lebih baik. Pada tampilan pertama, merge sort mungkin bukan pilihan yang baik karena node tengah diperlukan untuk membagi daftar yang diberikan menjadi dua sub-daftar dengan panjang yang sama. Kita dapat dengan mudah menyelesaikan masalah ini dengan memindahkan node secara alternatif ke dua daftar (lihat bab Daftar Tertaut). Kemudian, menyortir dua daftar ini secara rekursif dan menggabungkan hasilnya ke dalam satu daftar akan mengurutkan yang diberikan.

```
typedef struct ListNode {
    int data;
    struct ListNode *next;
};
```

```
struct ListNode * LinkedListMergeSort(struct ListNode * first) {
    struct ListNode * list1HEAD = NULL;
    struct ListNode * list1TAIL = NULL;
    struct ListNode * list2HEAD = NULL;
    struct ListNode * list2TAIL = NULL;
    if(first==NULL || first→next==NULL)
        return first;
    while (first != NULL) {
        Append(first, list1HEAD, list1TAIL);
        if(first != NULL)
            Append(first, list2HEAD, list2TAIL);
    }
    list1HEAD = LinkedListMergeSort(list1HEAD);
    list2HEAD = LinkedListMergeSort(list2HEAD);
    return Merge(list1HEAD, list2HEAD);
}
```

Catatan: *Append() menambahkan argumen pertama ke ekor dari daftar tertaut tunggal yang kepala dan ekornya ditentukan oleh argumen kedua dan ketiga.*

Semua Algoritma pengurutan eksternal dapat digunakan untuk menyortir daftar tertaut karena setiap file yang terlibat dapat dianggap sebagai daftar tertaut yang hanya dapat diakses secara berurutan. Kita dapat mengurutkan daftar tertaut ganda menggunakan bidang berikutnya seolah-olah itu adalah daftar yang ditautkan secara tunggal dan merekonstruksi bidang sebelumnya setelah menyortir dengan pemindaian tambahan.

Soal-28 Bisakah kita mengimplementasikan Penyortiran Linked Lists dengan Quick Sort?

Solusi: Quick Sort yang asli tidak dapat digunakan untuk menyortir Single Linked Lists. Ini karena kita tidak bisa mundur dalam Daftar Tertaut Tunggal. Tapi kita bisa memodifikasi Quick Sort yang asli dan membuatnya bekerja untuk Single Linked Lists.

Mari kita perhatikan implementasi Quick Sort yang dimodifikasi berikut ini. Node pertama dari daftar input dianggap sebagai poros dan dipindahkan ke sama. Nilai setiap node dibandingkan dengan pivot dan dipindahkan ke kurang (masing-masing, sama atau lebih besar) jika nilai node lebih kecil dari (masing-masing, sama dengan atau lebih besar dari) pivot. Kemudian, less dan large diurutkan secara rekursif. Akhirnya, menggabungkan lebih sedikit, sama, dan lebih besar ke dalam satu daftar menghasilkan daftar yang diurutkan.

Append() menambahkan argumen pertama ke ekor dari daftar tertaut tunggal yang kepala dan ekornya ditentukan oleh argumen kedua dan ketiga. Saat kembali, argumen pertama akan dimodifikasi sehingga menunjuk ke simpul berikutnya dari daftar. Join() menambahkan daftar yang kepala dan ekornya ditentukan oleh argumen ketiga dan keempat ke daftar yang kepala dan

ekornya ditentukan oleh argumen pertama dan kedua. Untuk kesederhanaan, argumen pertama dan keempat menjadi kepala dan ekor dari daftar yang dihasilkan.

```
typedef struct ListNode {
      int data;
      struct ListNode *next;
};
void Qsort(struct ListNode *first, struct ListNode * last) {
      struct ListNode *lesHEAD=NULL, lesTAIL=NULL;
      struct ListNode *equHEAD=NULL, equTAIL=NULL;
      struct ListNode *larHEAD=NULL, larTAIL=NULL;
      struct ListNode *current = *first;
      int pivot, info;
      if(current == NULL) return;
      pivot = current→data;
      Append(current, equHEAD, equTAIL);
      while (current != NULL) {
            info = current→data;
            if(info < pivot)
                  Append(current, lesHEAD, lesTAIL)
            else if(info > pivot)
                  Append(current, larHEAD, larTAIL)
            else      Append(current, equHEAD, equTAIL);
      }
      Quicksort(&lesHEAD, &lesTAIL);
      Quicksort(&larHEAD, &larTAIL);
      Join(lesHEAD, lesTAIL,equHEAD, equTAIL);
      Join(lesHEAD, equTAIL,larHEAD, larTAIL);
      *first = lesHEAD;
      *last = larTAIL;
}
```

Soal-29 Diberikan sebuah larik dengan nilai warna 100.000 piksel, yang masing-masing merupakan bilangan bulat dalam kisaran [0,255]. Algoritma pengurutan mana yang lebih disukai untuk menyortirnya?

Solusi: Menghitung Sortir. Hanya ada 256 nilai kunci, jadi array tambahan hanya akan berukuran 256, dan hanya akan ada dua lintasan data, yang akan sangat efisien dalam ruang dan waktu.

Soal-30 Mirip dengan Soal-29, jika kita memiliki direktori telepon dengan 10 juta entri, algoritma pengurutan mana yang terbaik?

Solusi: Sortir Bucket. Di Bucket Sort, bucket ditentukan oleh 7 digit terakhir. Ini membutuhkan array tambahan berukuran 10 juta dan memiliki keuntungan hanya membutuhkan satu kali melewati data pada disk. Setiap ember berisi semua nomor telepon dengan 7 digit terakhir yang sama tetapi dengan kode

area yang berbeda. Bucket kemudian dapat diurutkan berdasarkan kode area dengan pemilihan atau pengurutan penyisipan; hanya ada beberapa kode area.

Soal-31 Berikan algoritma untuk menggabungkan daftar yang diurutkan K.

Solusi Lihat bab Antrian Prioritas.

Soal-32 Diberikan sebuah file besar yang berisi milyaran angka. Temukan maksimum 10 angka dari file ini.

Solusi Lihat bab Antrian Prioritas.

Soal-33 Ada dua larik terurut A dan B. Yang pertama berukuran m + n yang hanya berisi m elemen. Satu lagi berukuran n dan mengandung n elemen. Gabungkan kedua larik ini ke dalam larik pertama berukuran m + n sedemikian rupa sehingga keluarannya diurutkan.

Solusi: Trik untuk masalah ini adalah mulai mengisi array tujuan dari belakang dengan elemen terbesar. Kita akan berakhir dengan array tujuan yang digabungkan dan diurutkan.

```
void Merge(int[] A[], int m, int B[], int n) {
    int count = m;
    int i = n – 1, j = count – 1, k = m - 1;
    for(;k>=0;k--) {
            if(B[i] > A[j] || j < 0) {
                A[k] =B[i];
                i--;
                if(i<0)
                    break;
            }
            else {
                A[k] = A[j];
                j--;
            }
    }
}
```

Kompleksitas Waktu: O(m + n).

Kompleksitas Ruang: O(1).

Soal-34 Mur dan Baut Soal: Diberikan satu set n mur dengan ukuran dan n baut yang berbeda sehingga terdapat korespondensi satu-satu antara mur dan baut, temukan untuk setiap mur baut yang sesuai. Asumsikan bahwa kita hanya dapat membandingkan mur dengan baut: kita tidak dapat membandingkan mur dengan mur dan baut dengan baut.

Cara alternatif untuk membingkai pertanyaan: Kita diberikan sebuah kotak yang berisi baut dan mur. Asumsikan ada n mur dan n baut dan setiap mur cocok dengan tepat satu baut (dan sebaliknya). Dengan mencoba mencocokkan

baut dan mur kita bisa melihat mana yang lebih besar, tetapi kita tidak bisa membandingkan dua baut atau dua mur secara langsung. Rancang algoritma yang efisien untuk mencocokkan mur dan baut.

Solusi: Pendekatan Brute Force: Mulailah dengan baut pertama dan bandingkan dengan masing-masing mur sampai kita menemukan kecocokan. Dalam kasus terburuk, kita membutuhkan n perbandingan. Ulangi ini untuk baut berturut-turut pada semua yang tersisa memberikan kompleksitas $O(n^2)$.

Soal-35 Untuk Soal-34, dapatkah kita meningkatkan kompleksitasnya?

Solusi: Dalam Soal-34, kita mendapatkan kompleksitas $O(n^2)$ dalam kasus terburuk (jika baut dalam urutan menaik dan mur dalam urutan menurun). Analisisnya sama dengan Quick Sort. Peningkatannya juga sejalan. Untuk mengurangi kompleksitas kasus terburuk, alih-alih memilih baut pertama setiap kali, kita dapat memilih baut acak dan mencocokkannya dengan mur. Pemilihan acak ini mengurangi kemungkinan mendapatkan kasus terburuk, tapi tetap saja kasus terburuknya adalah $O(n^2)$.

Soal-36 Untuk Soal-34, dapatkah kita meningkatkan kompleksitas lebih lanjut?

Solusi: Kita dapat menggunakan teknik bagi-dan-taklukkan untuk memecahkan masalah ini dan solusinya sangat mirip dengan Quick Sort secara acak. Untuk mempermudah, mari kita asumsikan bahwa baut dan mur diwakili dalam dua larik B dan N.

Algoritma pertama-tama melakukan operasi partisi sebagai berikut: pilih baut acakB[t]. Dengan menggunakan baut ini, atur ulang susunan mur menjadi tiga kelompok elemen:

- Pertama mur yang lebih kecil dari B[i]
- Kemudian mur yang cocok dengan B[i], dan
- Terakhir, mur lebih besar dari B[i].

Selanjutnya, dengan menggunakan mur yang cocok dengan B[i], lakukan partisi serupa pada susunan baut. Sepasang operasi partisi ini dapat dengan mudah diimplementasikan dalam waktu O(n), dan membiarkan baut dan mur dipartisi dengan baik sehingga baut dan mur "poros" sejajar satu sama lain dan semua baut dan mur lainnya berada di sisi yang benar dari pivot ini – mur dan baut yang lebih kecil mendahului pivot, dan mur dan baut yang lebih besar mengikuti pivot. Algoritma kita kemudian diselesaikan dengan menerapkan dirinya secara rekursif ke subarray di kiri dan kanan posisi pivot untuk mencocokkan baut dan mur yang tersisa ini. Kita dapat mengasumsikan dengan induksi pada n bahwa panggilan rekursif ini akan cocok dengan baut yang tersisa.

Untuk menganalisis waktu berjalan dari Algoritma kita, kita dapat menggunakan analisis yang sama dengan Quick Sort secara acak. Oleh karena itu, dengan menerapkan analisis dari Quick Sort, kompleksitas waktu dari algoritma kita adalah O(nlogn).

Analisis Alternatif: Kita dapat memecahkan masalah ini dengan membuat perubahan kecil pada Quick Sort. Mari kita asumsikan bahwa kita memilih elemen terakhir sebagai poros, katakanlah itu adalah kacang. Bandingkan mur dengan hanya baut saat kita berjalan menuruni larik. Ini akan mempartisi array untuk baut. Setiap baut yang kurang dari mur partisi akan berada di sebelah kiri. Dan setiap baut yang lebih besar dari mur partisi akan berada di sebelah kanan.

Saat menelusuri daftar, temukan baut yang cocok untuk mur partisi. Sekarang kita melakukan partisi lagi menggunakan baut yang cocok. Akibatnya, semua mur yang lebih kecil dari baut yang cocok akan berada di sisi kiri dan semua mur yang lebih besar dari baut yang cocok akan berada di sisi kanan. Panggil secara rekursif pada array kiri dan kanan.

Kompleksitas waktunya adalah O(2nlogn) O(nlogn).

Soal-37 Diberikan pohon biner, dapatkah kita mencetak elemen-elemennya dalam urutan terurut dalam waktu O(n) dengan melakukan traversal pohon In-order?

Solusi: Ya, jika pohonnya adalah Pohon Pencarian Biner [BST]. Untuk lebih jelasnya lihat bab Pohon.

Soal-38 Diberikan sebuah larik elemen, ubahlah menjadi larik sedemikian rupa sehingga A < B > C < D > E < F dan seterusnya.

Solusi: Urutkan array, lalu tukar setiap elemen yang berdekatan untuk mendapatkan hasil akhir.

```
#include<algorithm>
convertArraytoSawToothWave(){
   int A[] = {0,-6,9,13,10,-1,8,12,54,14,-5};
   int n = sizeof(A)/sizeof(A[0]), i = 1, temp;
   sort(A, A+n);
    for(i=1; i < n; i+=2){
      if(i+1 < n){
         temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
      }
   }
   for(i=0; i < n; i++){
      printf("%d ", A[i]);
   }
}
```

Kompleksitas waktu adalah O(nlogn+n) O(nlogn), untuk pengurutan dan pemindaian.

Soal-39 Bisakah kita mengerjakan Soal-38 dengan waktu O(n)?

Solusi: Pastikan semua elemen berposisi genap lebih besar dari elemen ganjil yang berdekatan, dan kita tidak perlu khawatir tentang elemen berposisi ganjil. Lintasi semua elemen array input yang diposisikan genap, dan lakukan hal berikut:

- Jika elemen saat ini lebih kecil dari elemen ganjil sebelumnya, tukar sebelumnya dan saat ini.
- Jika elemen saat ini lebih kecil dari elemen ganjil berikutnya, tukar berikutnya dan saat ini.

```
convertArraytoSawToothWave(){
    int A[] = {0,-6,9,13,10,-1,8,12,54,14,-5};
    int n = sizeof(A)/sizeof(A[0]), i = 1, temp;
    sort(A, A+n);
    for(i=1; i < n; i+=2){
        if (i>0 && A[i-1] > A[i] ){
           temp = A[i]; A[i] = A[i-1]; A[i-1] = temp;
        }
        if (i<n-1 && A[i] < A[i+1] ){
            temp = A[i]; A[i] = A[i+1]; A[i+1] = temp;
        }
    }
    for(i=0; i < n; i++){
       cout<<A[i]<< " ";
    }
}
```

Kompleksitas waktu adalah O(n).

Soal-40 Merge sort menggunakan

(a) Bagi dan taklukkan strategi
(b) Pendekatan mundur
(c) Pencarian heuristik
(d) Pendekatan tamak

Solusi: (a). Lihat bagian teori.

Soal-41 Manakah dari teknik desain algoritma berikut yang digunakan dalam algoritma quicksort?

(a) Pemrograman dinamis
(b) Mundur
(c) Bagi dan taklukkan
(d) Metode tamak

Solusi: (c). Lihat bagian teori.

Soal-42 Untuk menggabungkan dua daftar terurut ukuran m dan n ke dalam daftar terurut ukuran m+n, kita memerlukan perbandingan

(a) O(m)
(b) O(n)
(c) O(m + n)
(d) O(logm + logn)

Solusi: (c). Kita bisa menggunakan logika merge sort. Lihat bagian teori.

Soal-43 Pengurutan cepat dijalankan pada dua input yang ditunjukkan di bawah ini untuk mengurutkan dalam urutan menaik

(i) 1,2,3 n
(ii) n, n- 1, n-2, 2, 1

Biarkan C1 dan C2 menjadi jumlah perbandingan yang dibuat untuk input (i) dan (ii) masing-masing. Kemudian,

(a) C1 < C2
(b) C1 > C2
(c) C1 = C2
(d) kita tidak bisa mengatakan apa-apa untuk n sewenang-wenang.

Solusi: (b). Karena masalah yang diberikan membutuhkan *output* dalam urutan menaik, Quicksort pada urutan yang sudah diurutkan memberikan kasus terburuk (O(n2)). Jadi, (i) menghasilkan kasus terburuk dan (ii) membutuhkan lebih sedikit perbandingan.

Soal-44 Berikan pasangan yang benar untuk pasangan berikut:

(A) O(masuk)
(B) O(n)
(C) O(nlogn)
(D) O(n2)
(P) Seleksi
(Q) Jenis penyisipan
(R) Pencarian biner
(S) Menggabungkan sort

(a) A – R B – P C – Q – D – S
(b) A – R B – P C – S D – Q
(c) A – P B – R C – S D – Q
(d) A – P B – S C – R D – Q

Solusi: (b). Lihat bagian teori.

Soal-45 Misalkan s adalah larik terurut dari n bilangan bulat. Misalkan t(n) menunjukkan waktu yang dibutuhkan untuk algoritma yang paling efisien untuk menentukan apakah ada dua elemen dengan jumlah kurang dari 1000 dalam s. manakah dari pernyataan berikut yang benar?

a) t(n) adalah O(1)
b) $n < t(n) < nlog_2^n$
c) $nlog_2^n < t(n) < (_2^n)$
d) $t(n) = (_2^n)$

Solusi: (a). Karena array yang diberikan sudah diurutkan, cukup jika kita memeriksa dua elemen pertama dari array.

Soal-46 Implementasi Insertion Sort $\Theta(n^2)$ biasa untuk mengurutkan array menggunakan pencarian linier untuk mengidentifikasi posisi di mana elemen akan dimasukkan ke dalam bagian array yang sudah diurutkan. Jika, sebaliknya, kita menggunakan pencarian biner untuk mengidentifikasi posisi, waktu berjalan kasus terburuk akan

(a) tetap (n^2)
(b) menjadi ($n(\log n)^2$)
(c) menjadi (nlogn)
(d) menjadi (n)

Solusi: (a). Jika kita menggunakan pencarian biner maka akan ada perbandingan dalam kasus terburuk, yaitu (nlogn). Tetapi algoritma secara keseluruhan masih akan memiliki waktu berjalan rata-rata (n^2) karena serangkaian swap yang diperlukan untuk setiap penyisipan.

Soal-47 Dalam quick sort, untuk mengurutkan n elemen, elemen terkecil ke-n/4 dipilih sebagai pivot menggunakan algoritma waktu O(n). Apa kompleksitas waktu kasus terburuk dari quick sort?

(A) Θ (n)
(B) Θ (nLog)
(C) Θ (n^2)
(D) Θ (n^2logn)

Solusi: Ekspresi rekursi menjadi: T(n) = T(n/4) + T(3n/4) + en. Memecahkan rekursi menggunakan varian teorema master, kita mendapatkan Θ (nLogn).

Soal-48 Pertimbangkan algoritma Quicksort. Misalkan ada prosedur untuk menemukan elemen pivot yang membagi daftar menjadi dua sub-daftar yang masing-masing berisi setidaknya seperlima dari elemen. Misalkan T(n) adalah jumlah perbandingan yang diperlukan untuk mengurutkan n elemen. Kemudian

A) T (n) 2T (n/5) + n
B) T (n) T (n/5) + T (4n/5) + n
C) T (n) 2T (4n/5) + n
D) T (n) 2T (n / 2) + n

Solusi: (C). Untuk kasus di mana n/5 elemen berada dalam satu subset, T(n/5) diperlukan perbandingan untuk subset pertama dengan n/5 elemen, T(4n/5) untuk elemen 4n/5 sisanya, dan n adalah untuk menemukan poros. Jika ada lebih dari n/5 elemen dalam satu himpunan maka himpunan lainnya akan memiliki kurang dari 4n/5 elemen dan kompleksitas waktu akan kurang dari T(n/5) + T(4n/5) + n.

Soal-49 Manakah dari algoritma pengurutan berikut yang memiliki kompleksitas kasus terburuk terendah?

(A) Menggabungkan sort
(B) Jenis gelembung
(C) Sortir cepat
(D) Jenis seleksi

Solusi: (A). Lihat bagian teori.

Soal-50 Manakah dari algoritma penyortiran di tempat berikut yang membutuhkan jumlah swap minimum?

(A) Sortir cepat
(B) Jenis penyisipan
(C) Jenis seleksi
(D) Jenis tumpukan

Solusi: (C). Lihat bagian teori.

Soal-51 Anda memiliki larik n elemen. Misalkan Anda mengimplementasikan quicksort dengan selalu memilih elemen pusat array sebagai pivot. Maka batas atas yang paling ketat untuk kinerja kasus terburuk adalah

(A) $O(n^2)$
(B) O(nlogn)
(C) Θ (nlogn)
(D) O(n3)

Solusi: (A). Saat kita memilih elemen pertama sebagai pivot, kasus terburuk dari quick sort muncul jika input diurutkan - baik dalam urutan menaik atau menurun.

Soal-52 Biarkan P menjadi Program Quicksort untuk mengurutkan angka dalam urutan menaik menggunakan elemen pertama sebagai pivot. Misalkan t1 dan t2 adalah banyaknya perbandingan yang dibuat oleh P untuk input {1, 2, 3, 4, 5} dan {4, 1, 5, 3, 2} secara berurutan. Manakah dari berikut ini yang berlaku?

(A) t1 = 5
(B) t1 < t2
(C) t1 > t2
(D) t1 = t2

Solusi: (C). Kasus terburuk Quick Sort terjadi ketika elemen pertama (atau terakhir) dipilih sebagai pivot dengan array yang diurutkan.

Soal-53 Jumlah minimum perbandingan yang diperlukan untuk menemukan minimum dan maksimum 100 angka adalah ——

Solusi: 147 (Rumus untuk jumlah minimum perbandingan yang diperlukan adalah 3n/2 – 3 dengan n angka).

Soal-54 Banyaknya elemen yang dapat diurutkan dalam waktu T(logn) menggunakan heap sort adalah

(A) Θ (1)
(B) Θ (kuadrat(logn))
(C) Θ (log n/(log log n))
(D) Θ (masuk)

Solusi: (D). Mengurutkan array dengan k elemen membutuhkan waktu Θ (k log k) seiring bertambahnya k. Kita ingin memilih k sedemikian rupa sehingga Θ (k log k) = Θ (logn). Memilih k = (logn) tidak selalu berhasil, karena Θ (k log k) = Θ (logn loglogn) ≠ Θ (logn). Di sisi lain, jika Anda memilih k = T(log n / log log n), maka runtime semacam itu akan menjadi:

$$= \Theta((\log n / \log\log n) \log (\log n / \log\log n))$$
$$= \Theta((\log n / \log\log n) (\log\log n - \log\log\log n))$$
$$= \Theta(\log n - \log n \log\log\log n / \log\log n)$$
$$= \Theta(\log n (1 - \log\log\log n / \log\log n))$$

Perhatikan bahwa 1 – logloglogn / loglogn cenderung ke 1 saat n menuju tak terhingga, jadi ekspresi di atas sebenarnya adalah Θ (log n), seperti yang diperlukan. Oleh karena itu, jika Anda mencoba mengurutkan larik berukuran Θ (logn / loglogn) menggunakan heap sort, sebagai fungsi n, runtimenya adalah Θ (logn).

Soal-55 Manakah dari berikut ini yang merupakan batas atas paling ketat yang mewakili jumlah swap yang diperlukan untuk mengurutkan n angka menggunakan sortir pilihan?

(A) O(masuk)
(B) O(n)
(C) O(nlogn)
(D) $O(n^2)$

Solusi: (B). Jenis seleksi hanya membutuhkan O(n) swap.

Soal-56 Manakah dari berikut ini yang merupakan persamaan perulangan untuk kompleksitas waktu kasus terburuk dari algoritma Quicksort untuk mengurutkan n(≥ 2) angka? Dalam persamaan pengulangan yang diberikan dalam opsi di bawah ini, c adalah konstanta.

(A) T(n) = 2T (n/2) + cn
(B) T(n) = T(n – 1) + T(0) + cn
(C) T(n) = 2T (n – 2) + cn
(D) T(n) = T(n/2) + cn

‘

Solusi: (B). Ketika pivot adalah elemen terkecil (atau terbesar) yang dipartisi pada blok berukuran n, hasilnya menghasilkan satu sub-blok kosong, satu elemen (poros) di tempat yang benar dan sub-blok berukuran n – 1.

Soal-57 Benar atau Salah. Dalam quicksort acak, setiap kunci terlibat dalam jumlah perbandingan yang sama.

Solusi: Salah.

Soal-58 Benar atau Salah: Jika Quicksort ditulis sehingga algoritma partisi selalu menggunakan nilai median segmen sebagai pivot, maka performa terburuknya adalah O(nlogn).

Solusi: Benar.

BAB 3
PENCARIAN (*SEARCH*)

3.1 APA ITU PENCARIAN?

Dalam ilmu komputer, pencarian adalah proses menemukan item dengan properti tertentu dari kumpulan item. Item dapat disimpan sebagai catatan dalam database, elemen data sederhana dalam array, teks dalam file, node di pohon, simpul dan tepi dalam Grafik, atau mungkin elemen dari ruang pencarian lainnya.

3.2 MENGAPA KITA MEMBUTUHKAN PENCARIAN?

Pencarian adalah salah satu algoritma inti ilmu komputer. Kita tahu bahwa komputer saat ini menyimpan banyak informasi. Untuk mengambil informasi ini dengan mahir kita membutuhkan algoritma pencarian yang sangat efisien. Ada cara tertentu untuk mengatur data yang meningkatkan proses pencarian. Artinya, jika kita menyimpan data dalam urutan yang benar, akan mudah untuk mencari elemen yang diperlukan. Sortasi adalah salah satu teknik untuk membuat elemen tersusun. Dalam bab ini kita akan melihat algoritma pencarian yang berbeda.

3.3 JENIS PENCARIAN

Berikut ini adalah jenis-jenis pencarian yang akan kita bahas dalam buku ini.

- Pencarian Linier Tidak Terurut
- Pencarian Linier Diurutkan/Diurutkan
- Pencarian Biner
- Pencarian interpolasi
- Pohon Pencarian Biner (beroperasi pada pohon dan merujuk bab Pohon)
- Tabel Simbol dan Hashing
- Algoritma Pencarian String: Percobaan, Pencarian Ternary dan Pohon Sufiks

3.4 PENCARIAN LINIER TIDAK TERURUT

Mari kita asumsikan kita diberikan sebuah array di mana urutan elemen tidak diketahui. Itu berarti elemen array tidak diurutkan. Dalam hal ini, untuk mencari elemen kita harus memindai array lengkap dan melihat apakah elemen tersebut ada dalam daftar yang diberikan atau tidak

```
int UnOrderedLinearSearch (int A[], int n, int data) {
    for (int i  = 0; i < n; i++) {
        if(A[i] == data)
            return i;
    }
    return -1;
}
```

Kompleksitas waktu: O(n), dalam kasus terburuk kita perlu memindai array lengkap.
Kompleksitas ruang: O(1).

3.5 PENCARIAN LINIER DIURUTKAN

Jika elemen array sudah diurutkan, maka dalam banyak kasus kita tidak perlu memindai seluruh array untuk melihat apakah elemen tersebut ada dalam array yang diberikan atau tidak. Pada algoritma di bawah ini, dapat dilihat bahwa, pada sembarang titik jika nilai pada A[i] lebih besar dari data yang akan dicari, maka kita hanya mengembalikan -1 tanpa mencari array yang tersisa.

```
int OrderedLinearSearch(int A[], int n, int data) {
    for (int i  = 0; i < n; i++) {
            if(A[i] == data)
                    return i;
            else if(A[i] > data)
                    return -1;
    }
    return -1;
}
```

Kompleksitas waktu dari algoritma ini adalah O(n). Hal ini karena dalam kasus terburuk kita perlu memindai array yang lengkap. Tetapi dalam kasus rata-rata itu mengurangi kompleksitas meskipun tingkat pertumbuhannya sama.

Kompleksitas ruang: O(1).

Catatan: Untuk Algoritma di atas, kita dapat melakukan peningkatan lebih lanjut dengan meningkatkan indeks pada tingkat yang lebih cepat (katakanlah, 2). Ini akan mengurangi jumlah perbandingan untuk pencarian dalam daftar yang diurutkan.

3.6 PENCARIAN BINER

Mari kita pertimbangkan masalah pencarian kata dalam kamus. Biasanya, kita langsung menuju ke beberapa halaman perkiraan [misalnya, halaman tengah] dan mulai mencari dari titik itu. Jika nama yang kita cari sama maka pencarian selesai. Jika halaman berada sebelum halaman yang dipilih, maka terapkan proses yang sama untuk paruh pertama; jika tidak menerapkan proses yang sama untuk babak kedua. Pencarian biner juga bekerja dengan cara yang sama. Algoritma yang menerapkan strategi seperti itu disebut sebagai algoritma pencarian biner.

$$mid = low + \frac{(high-low)}{2} \; or \; \frac{low+high}{2}$$

Gambar 3.1 pencarian biner

```
//Iterative Binary Search Algorithm
int BinarySearchIterative(int A[], int n, int data) {
    int low =  0;
    int high = n-1;
    while (low <= high) {
        mid  = low + (high-low)/2; //To avoid overflow
        if(A[mid] == data)
            return mid;
        else if(A[mid] < data)
            low =  mid + 1;
        else high =  mid - 1;
    }
    return -1;
}
//Recursive Binary Search Algorithm
int BinarySearchRecursive(int A[], int low, int high, int data) {
    int mid  = low + (high-low)/2; //To avoid overflow
    if (low>high)
      return -1;
    if(A[mid] == data)
        return mid;
    else if(A[mid] < data)
        return BinarySearchRecursive (A, mid + 1, high, data);
    else   return BinarySearchRecursive (A, low, mid - 1 , data);
    return -1;
}
```

Pengulangan untuk pencarian biner adalah . Ini karena kita selalu mempertimbangkan hanya setengah dari daftar input dan membuang setengah lainnya. Menggunakan teorema utama *Divide and conquer*, kita mendapatkan, T(n) = O(logn).

Kompleksitas Waktu: O(logn).

Kompleksitas Ruang: O(1) [untuk algoritma iteratif].

3.7 PENCARIAN INTERPOLASI

Tidak diragukan lagi pencarian biner adalah algoritma yang bagus untuk pencarian dengan kompleksitas waktu berjalan rata-rata dari logn. Itu selalu memilih tengah dari ruang pencarian yang tersisa, membuang satu setengah atau yang lain, sekali lagi tergantung pada perbandingan antara nilai kunci yang ditemukan pada posisi perkiraan (tengah) dan nilai kunci yang dicari. Ruang pencarian yang tersisa dikurangi menjadi bagian sebelum atau sesudah perkiraan posisi.

Dalam matematika, interpolasi adalah proses membangun titik-titik data baru dalam jangkauan kumpulan titik-titik data yang diketahui secara diskrit. Dalam ilmu komputer, seseorang sering memiliki sejumlah titik data yang mewakili nilai suatu fungsi untuk sejumlah nilai variabel independen yang terbatas. Seringkali diperlukan untuk menginterpolasi (yaitu memperkirakan) nilai fungsi itu untuk nilai antara dari variabel independen.

Misalnya, kita memiliki tabel seperti ini, yang memberikan beberapa nilai fungsi yang tidak diketahui f. Interpolasi menyediakan sarana untuk memperkirakan fungsi pada titik-titik antara, seperti x = 55.

Tabel 3.1 Data pencarian interpolasi

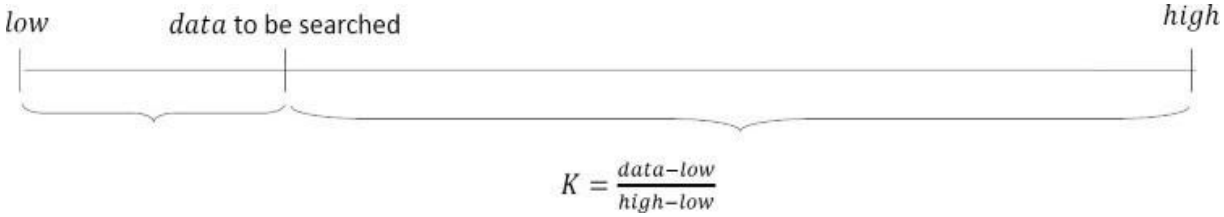
x	***f(x)***
1	10
2	20
3	30
4	40
5	50
6	60
7	70

Ada banyak metode interpolasi yang berbeda, dan salah satu metode yang paling sederhana adalah interpolasi linier. Karena 55 berada di tengah antara 50 dan 60, masuk akal untuk mengambil f(55) di tengah antara f(5) = 50 dan f(6) = 60, yang menghasilkan 55.

Interpolasi linier mengambil dua titik data, katakanlah (x1; y2) dan (x2, y2), dan interpolannya diberikan oleh:

$$y = y_1 + (y_2 - y_1)\frac{x - x_1}{x_2 - x_1} \; at \; point \; (x, y)$$

Dengan input di atas, apa yang akan terjadi jika kita tidak menggunakan konstanta , tetapi konstanta “K” lain yang lebih akurat, yang dapat membawa kita lebih dekat ke item yang dicari.



Gambar 3.2 Pencarian Interpolasi

Algoritma ini mencoba mengikuti cara kita mencari nama di buku telepon, atau kata di kamus. Kita, manusia, tahu sebelumnya bahwa jika nama yang kita cari dimulai dengan "m", seperti "biksu" misalnya, kita harus mulai mencari di dekat bagian tengah buku telepon. Jadi jika kita mencari kata "karir" dalam kamus, Anda tahu bahwa itu harus ditempatkan di suatu tempat di awal. Ini karena kita tahu urutan hurufnya, kita tahu intervalnya (a-z), dan entah bagaimana kita secara intuitif tahu bahwa kata-kata itu tersebar secara merata. Fakta-fakta ini cukup untuk menyadari bahwa pencarian biner bisa menjadi pilihan yang buruk. Memang algoritma pencarian biner membagi daftar dalam dua sub-daftar yang sama, yang tidak berguna jika kita tahu sebelumnya bahwa item yang dicari ada di suatu tempat di awal atau akhir daftar. Ya, kita juga dapat menggunakan pencarian melompat jika item berada di awal, tetapi tidak jika berada di akhir, dalam hal ini algoritma ini tidak begitu efektif.

Algoritma pencarian interpolasi mencoba untuk meningkatkan pencarian biner. Pertanyaannya adalah bagaimana menemukan nilai ini? Nah, kita tahu batas-batas interval dan melihat lebih dekat ke gambar di atas kita bisa mendefinisikan rumus berikut.

$$K = \frac{data - low}{high - low}$$

Konstanta K ini digunakan untuk mempersempit ruang pencarian. Untuk pencarian biner, konstanta K ini adalah (rendah + tinggi)/2.

Sekarang kita dapat yakin bahwa kita lebih dekat dengan nilai yang dicari. Rata-rata pencarian interpolasi membuat perbandingan log (logn) (jika elemen terdistribusi seragam), di mana n adalah jumlah elemen yang akan dicari. Dalam kasus terburuk (misalnya di mana nilai numerik dari kunci meningkat secara eksponensial) dapat membuat hingga O(n) perbandingan. Dalam pencarian interpolasi-sekuensial, interpolasi digunakan untuk menemukan item yang dekat dengan yang dicari, kemudian pencarian linier digunakan untuk menemukan item yang tepat. Agar Algoritma ini memberikan hasil terbaik, kumpulan data harus diurutkan dan didistribusikan secara merata.

```
int InterpolationSearch(int A[], int data){
    int low = 0, mid, high = sizeof(A) – 1;
    while (low <= high) {
            mid = low + (((data – A[low]) * (high – low))/(A[high] – A[low]));
            if (data == A[mid])
                return mid + 1;
            if (data < A[mid])
                high = mid - 1;
            else
                low = mid + 1;
    }
    return -1;
}
```

3.8 MEMBANDINGKAN ALGORITMA PENCARIAN DASAR

Tabel 3.2 membandingkan algoritma dasar

Penerapan	Pencarian-Kasus Terburuk	Cari - Rata-Rata Kasus
Array Tidak Terurut	n	n/2
Array Terurut (Pencarian Biner)	logn	logn
Daftar Tidak Terurut	n	n/2
Daftar pesanan	n	n/2
Pohon Pencarian Biner (untuk pohon miring)	n	logn
Pencarian interpolasi	n	Log(logn)

Catatan: Untuk diskusi tentang pohon pencarian biner lihat bab Pohon.

3.9 PENCARIAN: MASALAH & SOLUSI

Soal-1 Diberikan array n angka, berikan algoritma untuk memeriksa apakah ada elemen duplikat dalam array atau tidak?

Solusi: Ini adalah salah satu masalah paling sederhana. Satu jawaban yang jelas untuk ini adalah mencari duplikat dalam array secara mendalam. Artinya, untuk setiap elemen input periksa apakah ada elemen dengan nilai yang sama. Ini bisa kita selesaikan hanya dengan menggunakan dua loop for sederhana. Kode untuk solusi ini dapat diberikan sebagai:

```
void CheckDuplicatesBruteForce(int A[], int n) {
    for(int i = 0; i < n; i++) {
        for(int j = i+1; j < n; j++) {
            if(A[i] == A[j])        {
                printf("Duplicates exist: %d", A[i]);
                return;
            }
        }
    }
    printf("No duplicates in given array.");
}
```

Kompleksitas Waktu: O(n2), untuk dua loop for bersarang.
Kompleksitas Ruang: O(1).

Soal-2 Bisakah kita meningkatkan kompleksitas solusi Soal-1?

Solusi: Ya. Urutkan array yang diberikan. Setelah disortir, semua elemen dengan nilai yang sama akan berdekatan. Sekarang, lakukan pemindaian lain pada array yang diurutkan ini dan lihat apakah ada elemen dengan nilai yang sama dan berdekatan.

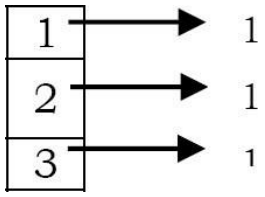
```
void CheckDuplicatesSorting(int A[], int n) {
    Sort(A, n);                          //sort the array
    for(int i = 0; i < n-1; i++) {
        if(A[i] == A[i+1]) {
            printf("Duplicates exist: %d", A[i]);
            return;
        }
    }
    printf("No duplicates in given array.");
}
```

Kompleksitas Waktu: O(nlogn), untuk pengurutan (dengan asumsi algoritma pengurutan nlogn). Kompleksitas Ruang: O(1).

Soal-3 Apakah ada cara alternatif untuk menyelesaikan Soal-1?

Solusi: Ya, menggunakan tabel hash. Tabel hash adalah metode sederhana dan efektif yang digunakan untuk mengimplementasikan kamus. Waktu rata-rata untuk mencari elemen adalah O(1), sedangkan waktu kasus terburuk adalah O(n). Lihat bab Hashing untuk detail lebih lanjut tentang algoritma hashing. Sebagai contoh, perhatikan array, A = {3,2,1,2,2,3}.

Pindai array input dan masukkan elemen ke dalam hash. Untuk setiap elemen yang dimasukkan, pertahankan counter sebagai 1 (asumsikan awalnya semua keseluruhan diisi dengan nol). Ini menunjukkan bahwa elemen yang sesuai telah terjadi. Untuk array yang diberikan, tabel hash akan terlihat seperti (setelah memasukkan tiga elemen pertama 3,2 dan 1):



Sekarang jika kita mencoba memasukkan 2, karena nilai counter dari 2 sudah 1, kita dapat mengatakan bahwa elemen tersebut telah muncul dua kali.

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-4 Bisakah kita lebih meningkatkan kompleksitas solusi Soal-1?

Solusi: Mari kita asumsikan bahwa elemen array adalah bilangan positif dan semua elemen berada dalam rentang 0 hingga n – 1. Untuk setiap elemen A[i], pergi ke elemen array yang indeksnya adalah A[i]. Itu berarti pilih A[A[i]] dan tandai -A[A[i]] (negasikan nilai pada A[A[i]]). Lanjutkan proses ini sampai kita menemukan elemen yang nilainya sudah dinegasikan. Jika salah satu elemen

tersebut ada maka kita katakan elemen duplikat ada dalam array yang diberikan. Sebagai contoh, perhatikan array, A = {3,2,1,2,2,3}.

Mulanya,

3	2	1	2	2	3
0	1	2	3	4	5

Pada langkah-1, hilangkan A[abs(A[0])],

3	2	1	-2	2	3
0	1	2	3	4	5

Pada langkah-2, hilangkan A[abs(A[l])],

3	2	-1	-2	2	3
0	1	2	3	4	5

Pada langkah-3, hilangkan A[abs(A[2])],

3	-2	- 1	-2	2	3
0	1	2	3	4	5

Pada langkah-4, hilangkan A[abs(A[3])],

3	-2	- 1	-2	2	3
0	1	2	3	4	5

Pada langkah-4, amati bahwa A[abs(A[3])] sudah negatif. Itu berarti kita telah menemukan nilai yang sama dua kali.

```
void CheckDuplicates(int A[], int n) {
    for(int i = 0; i < n; i++) {
        if(A[abs(A[i])] < 0) {
            printf("Duplicates exist:%d", A[i]);
            return;
        }
        else A[A[i]] = - A[A[i]];
    }
    printf("No duplicates in given array.");
}
```

Kompleksitas Waktu: O(n). Karena hanya satu pemindaian yang diperlukan.

Kompleksitas Ruang: O(1).

Catatan:

- Solusi ini tidak bekerja jika array yang diberikan hanya bisa dibaca.
- Solusi ini hanya akan bekerja jika semua elemen array bernilai positif.
- Jika rentang elemen tidak dalam 0 sampai n – 1 maka dapat memberikan pengecualian.

Soal-5 Diberikan sebuah array n angka. Berikan algoritma untuk menemukan elemen yang muncul paling banyak dalam array?

Solusi *Brute Force:* Salah satu solusi sederhana untuk ini adalah, untuk setiap elemen input periksa apakah ada elemen dengan nilai yang sama, dan untuk setiap kejadian tersebut, tambahkan penghitung. Setiap kali, periksa penghitung saat ini dengan penghitung maks dan perbarui jika nilai ini lebih besar dari penghitung maks. Ini bisa kita selesaikan hanya dengan menggunakan dua loop for sederhana.

```
int MaxRepititionsBruteForce(int A[], int n) {
    int counter =0, max=0;
    for(int i = 0; i < n; i++) {
        counter=0;
        for(int j = 0; j < n; j++) {
            if(A[i] == A[j])
                counter++;
        }
        if(counter > max) max = counter;
    }
    return max;
}
```

Kompleksitas Waktu: O(n2), untuk dua loop for bersarang.
Kompleksitas Ruang: O(1).

Soal-6 Bisakah kita meningkatkan kompleksitas solusi Soal-5?

Solusi: Ya. Urutkan array yang diberikan. Setelah disortir, semua elemen dengan nilai yang sama akan berdekatan. Sekarang, lakukan pemindaian lain pada array yang diurutkan ini dan lihat elemen mana yang muncul paling banyak.

Kompleksitas Waktu: O(nlogn). (untuk menyortir).
Kompleksitas Ruang: O(1).

Soal-7 Apakah ada cara lain untuk menyelesaikan Soal-5?

Solusi: Ya, menggunakan tabel hash. Untuk setiap elemen input, catat berapa kali elemen tersebut muncul di input. Itu berarti nilai penghitung mewakili jumlah kemunculan untuk elemen itu.

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-8 atau Soal-5, dapatkah kita meningkatkan kompleksitas waktu? Asumsikan bahwa rentang elemen adalah 1 hingga n. Itu berarti semua elemen berada dalam kisaran ini saja.

Solusi: Ya. Kita dapat memecahkan masalah ini dalam dua pemindaian. Kita tidak dapat menggunakan teknik negasi dari Soal-3 untuk soal ini karena banyaknya pengulangan. Pada pemindaian pertama, alih-alih meniadakan, tambahkan nilai n. Itu berarti untuk setiap kemunculan elemen tambahkan ukuran array ke elemen itu. Pada pemindaian kedua, periksa nilai elemen dengan membaginya dengan n dan kembalikan elemen yang memberikan nilai maksimum. Kode berdasarkan metode ini diberikan di bawah ini.

```
void MaxRepititions(int A[], int n) {
    int i =  0, max = 0, maxIndex;
    for(i = 0; i < n; i++)
        A[A[i]%n] +=n;
    for(i = 0; i < n; i++)
        if(A[i]/n > max) {
            max = A[i]/n;
            maxIndex =i;
        }
    return maxIndex;
}
```

Catatan:

- Solusi ini tidak bekerja jika array yang diberikan hanya bisa dibaca.
- Solusi ini hanya akan bekerja jika elemen arraynya positif.
- Jika rentang elemen tidak dalam 1 sampai n maka dapat memberikan pengecualian.

Kompleksitas Waktu: O(n). Karena tidak diperlukan perulangan for bersarang.
Kompleksitas Ruang: O(1).

Soal-9 Diberikan sebuah larik n angka, berikan algoritma untuk menemukan elemen pertama dalam larik yang berulang. Misalnya, dalam larik A = {3,2,1,2,2,3}, angka pertama yang berulang adalah 3 (bukan 2). Itu berarti, kita perlu mengembalikan elemen pertama di antara elemen yang diulang.

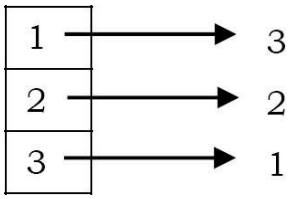
Solusi: Kita dapat menggunakan solusi brute force yang kita gunakan untuk Soal-1. Untuk setiap elemen, karena memeriksa apakah ada duplikat untuk elemen itu atau tidak, elemen mana yang lebih dulu digandakan akan dikembalikan.

Soal-10 Untuk Soal-9, dapatkah kita menggunakan teknik pengurutan?

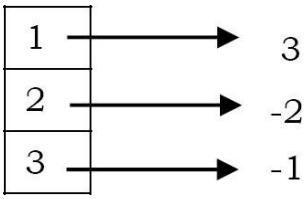
Solusi: Tidak. Untuk membuktikan kasus gagal, mari kita perhatikan larik berikut. Misal A = {3, 2, 1, 2, 2, 3}. Setelah diurutkan kita mendapatkan A = {1,2,2,2,3,3}. Dalam array yang diurutkan ini, elemen pertama yang diulang adalah 2 tetapi jawaban sebenarnya adalah 3.

Soal-11 Untuk Soal-9, dapatkah kita menggunakan teknik hashing?

Solusi: Ya. Tetapi teknik hashing sederhana yang kita gunakan untuk Soal-3 tidak akan berhasil. Misalnya, jika kita menganggap array input sebagai A = {3,2,1,2,3}, maka elemen yang diulang pertama adalah 3, tetapi menggunakan teknik hashing sederhana, kita mendapatkan jawabannya sebagai 2. Ini karena 2 adalah datang dua kali sebelum 3. Sekarang mari kita ubah perilaku tabel hashing sehingga kita mendapatkan elemen berulang pertama. Katakanlah, alih-alih menyimpan 1 nilai, awalnya kita menyimpan posisi elemen dalam array. Hasilnya tabel hash akan terlihat seperti (setelah memasukkan 3,2 dan 1):



Sekarang, jika kita melihat 2 lagi, kita hanya meniadakan nilai 2 saat ini di tabel hash. Artinya, kita jadikan nilai counter-nya sebagai -2. Nilai negatif dalam tabel hash menunjukkan bahwa kita telah melihat elemen yang sama dua kali. Demikian pula, untuk 3 (elemen berikutnya dalam input) juga, kita meniadakan nilai tabel hash saat ini dan akhirnya tabel hash akan terlihat seperti:



Setelah memproses array input lengkap, pindai tabel hash dan kembalikan nilai indeks negatif tertinggi darinya (yaitu, -1 dalam kasus kita). Nilai negatif tertinggi menunjukkan bahwa kita telah melihat elemen itu terlebih dahulu (di antara elemen yang berulang) dan juga berulang. Bagaimana jika elemen diulang lebih dari dua kali? Dalam hal ini, lewati saja elemen tersebut jika nilai yang sesuai i sudah negatif.

Soal-12 Untuk Soal-9, dapatkah kita menggunakan teknik yang kita gunakan untuk Soal-3 (teknik negasi)?

Solusi: Tidak. Sebagai contoh kontradiksi, untuk larik A = {3,2,1,2,2,3} elemen yang diulang pertama adalah 3. Tetapi dengan teknik negasi hasilnya adalah 2.

Soal-13 Menemukan Bilangan yang Hilang: Kita diberikan daftar n – 1 bilangan bulat dan bilangan bulat ini berada dalam kisaran 1 hingga n. Tidak ada duplikat dalam daftar. Salah satu bilangan bulat tidak ada dalam daftar. Diberikan algoritma untuk menemukan bilangan bulat yang hilang. Contoh: I/P: [1,2,4,6,3,7,8] O/P: 5

Solusi Brute Force: Salah satu solusi sederhana untuk ini adalah, untuk setiap angka dalam 1 hingga n, periksa apakah angka itu ada dalam larik yang diberikan atau tidak.

```
int FindMissingNumber(int A[], int n) {
    int i, j, found=0;
    for (i = 1; i < =n; i ++) {
        found = 0;
        for (j = 0; j < n; j ++)
            if(A[j]==i)
                found = 1;
        if(!found) return i;
    }
    return -1;
}
```

Kompleksitas Waktu: O(n2).
Kompleksitas Ruang: O(1).

Soal-14 Untuk Soal-13, dapatkah kita menggunakan teknik pengurutan?

Solusi: Ya. Mengurutkan daftar akan memberikan elemen dalam urutan yang meningkat dan dengan pemindaian lain kita dapat menemukan nomor yang hilang.

Kompleksitas Waktu: O(nlogn), untuk menyortir.
Kompleksitas Ruang: O(1).

Soal-15 Untuk Soal-13, dapatkah kita menggunakan teknik hashing?

Solusi: Ya. Pindai array input dan masukkan elemen ke dalam hash. Untuk elemen yang disisipkan, pertahankan penghitung sebagai 1 (asumsikan awalnya semua keseluruhan diisi dengan nol). Ini menunjukkan bahwa elemen yang sesuai telah terjadi. Sekarang, pindai tabel hash dan kembalikan elemen yang memiliki nilai penghitung nol.
Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-16 Untuk Soal-13, dapatkah kita meningkatkan kompleksitasnya?

Solusi: Ya. Kita bisa menggunakan rumus penjumlahan.

1) Dapatkan jumlah angka, jumlah = n × (n + l)/2.
2) Kurangi semua angka dari jumlah dan Anda akan mendapatkan angka yang hilang.

Kompleksitas Waktu: O(n), untuk memindai array lengkap.

Soal-17 Dalam Soal-13, jika jumlah bilangan melebihi bilangan bulat maksimum yang diperbolehkan, maka dapat terjadi kelebihan bilangan bulat dan kita mungkin tidak mendapatkan jawaban yang benar. Bisakah kita memecahkan masalah ini?

Solusi:

1) XOR semua elemen array, biarkan hasil XOR menjadi X.
2) XOR semua bilangan dari 1 sampai n, misalkan XOR adalah Y.
3) XOR dari X dan Y memberikan nomor yang hilang.

```
int FindMissingNumber(int A[], int n) {
    int i, X, Y;
    for (i = 0; i < n; i ++)
        X ^= A[i];
    for (i = 1; i <= n; i ++)
        Y ^= i;
    //In fact, one variable is enough.
    return X ^ Y;
}
```

Kompleksitas Waktu: O(n), untuk memindai array lengkap.
Kompleksitas Ruang: O(1).

Soal-18 Menemukan Bilangan yang Terjadi Beberapa Kali Ganjil: Diberikan sebuah larik bilangan bulat positif, semua bilangan muncul beberapa kali genap kecuali satu bilangan yang muncul beberapa kali ganjil. Temukan nomor dalam O(n) waktu & ruang konstan. Contoh : I/P = [1,2,3,2,3,1,3] O/P = 3

Solusi: Lakukan XOR bitwise dari semua elemen. Kita mendapatkan nomor yang memiliki kejadian ganjil. Ini karena, A XOR A = 0.

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-19 Temukan dua elemen berulang dalam array yang diberikan: Diberikan array dengan ukuran, semua elemen array berada dalam rentang 1 hingga n dan juga semua elemen hanya muncul sekali kecuali dua angka yang muncul dua kali. Temukan dua angka berulang itu. Contoh: jika array adalah 4,2,4,5,2,3,1 dengan

ukuran = 7 dan n = 5. Input ini memiliki n + 2 = 7 elemen dengan semua elemen muncul sekali kecuali 2 dan 4 yang muncul dua kali . Jadi *output*nya harus 4 2.

Solusi: Salah satu cara sederhana adalah memindai array lengkap untuk setiap elemen elemen input. Itu berarti menggunakan dua loop. Di loop luar, pilih elemen satu per satu dan hitung jumlah kemunculan elemen yang dipilih di loop dalam. Untuk kode di bawah ini, asumsikan bahwa PrintRepeatedElements dipanggil dengan n + 2 untuk menunjukkan ukurannya.

```
void PrintRepeatedElements(int A[], int size) {
    for(int i = 0; i < size; i++)
        for(int j = i+1; j < size; j++)
            if(A[i] == A[j])
                printf("%d", A[i]);
}
```

Kompleksitas Waktu: O(n2).
Kompleksitas Ruang: O(1).

Soal-20 Untuk Soal-19, dapatkah kita meningkatkan kompleksitas waktu?

Solusi: Urutkan array menggunakan algoritma pengurutan perbandingan dan lihat apakah ada elemen yang bersebelahan dengan nilai yang sama.
Kompleksitas Waktu: O(nlogn).
Kompleksitas Ruang: O(1).

Soal-21 Untuk Soal-19, dapatkah kita meningkatkan kompleksitas waktu?

Solusi: Gunakan Hitungan Array. Solusi ini seperti menggunakan tabel hash. Untuk mempermudah kita dapat menggunakan array untuk menyimpan hitungan. Lintasi array sekali dan lacak jumlah semua elemen dalam array menggunakan array temp count[] berukuran n. Saat kita melihat elemen yang hitungannya sudah diatur, cetak sebagai duplikat. Untuk kode di bawah ini, asumsikan bahwa PrintRepeatedElements dipanggil dengan n + 2 untuk menunjukkan ukuran.

```
void PrintRepeatedElements(int A[], int size) {
    int *count = (int *)calloc(sizeof(int), (size - 2));
    for(int i = 0; i < size; i++) {
        count[A[i]]++;
        if(count[A[i]] == 2)
            printf("%d", A[i]);
    }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-22 Pertimbangkan Soal-19. Mari kita asumsikan bahwa angka-angka tersebut berada dalam rentang 1 hingga n. Apakah ada cara lain untuk memecahkan masalah?

Solusi: Ya, dengan menggunakan Operasi XOR. Misalkan bilangan yang berulang adalah X dan Y, jika kita XOR semua elemen dalam array dan juga semua bilangan bulat dari 1 sampai n, maka hasilnya adalah X XOR Y. Angka 1 dalam representasi biner dari X XOR Y sesuai dengan bit yang berbeda antara X dan Y. Jika bit ke-k dari X XOR Y adalah 1, kita dapat meng-XOR semua elemen dalam array dan juga semua bilangan bulat dari 1 hingga n yang bit ke-knya adalah 1. Hasilnya adalah salah satu dari X dan Y.

```
void PrintRepeatedElements (int A[], int size) {
    int XOR = A[0];
    int i, right_most_set_bit_no, X= 0, Y = 0;
    for(i = 0; i < size; i++)                /* Compute XOR of all elements in A[]*/
            XOR ^= A[i];
    for(i = 1; i <= n; i++)                  /* Compute XOR of all elements {1, 2 ..n} */
            XOR ^= i;
    right_most_set_bit_no = XOR & ~( XOR -1);    // Get the rightmost set bit in right_most_set_bit_no
    /* Now divide elements in two sets by comparing rightmost set */
    for(i = 0; i < size; i++) {
            if(A[i] & right_most_set_bit_no)
                    X = X^ A[i];        /*XOR of first set in A[] */
            else    Y = Y ^ A[i];       /*XOR of second set in A[] */
    }
    for(i = 1; i <= n; i++) {
            if(i & right_most_set_bit_no)
                    X = X ^ i;              /*XOR of first set in A[] and {1, 2, ...n }*/
            else    Y = Y ^ i;              /*XOR of second set in A[] and {1, 2, ...n } */
    }
    printf("%d and %d",X, Y);
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-23 Pertimbangkan Soal-19. Mari kita asumsikan bahwa angka-angka tersebut berada dalam rentang 1 hingga n. Apakah masih ada cara lain untuk menyelesaikan masalah?

Solusi: Kita dapat menyelesaikannya dengan membuat dua persamaan matematika sederhana. Mari kita asumsikan bahwa dua bilangan yang akan kita temukan adalah X dan Y. Kita tahu jumlah n bilangan adalah n(n + l)/2 dan hasil kali adalah n!. Buat dua persamaan menggunakan rumus jumlah dan hasil kali ini, dan dapatkan nilai dari dua yang tidak diketahui menggunakan kedua persamaan tersebut. Misalkan penjumlahan semua bilangan dalam larik adalah S dan hasil kali adalah P dan bilangan yang diulang adalah X dan Y.

$$X + Y = S - \frac{n(n+1)}{2}$$
$$XY = P/n!$$

Dengan menggunakan dua persamaan di atas, kita dapat menemukan X dan Y. Dapat ada masalah luapan penjumlahan dan perkalian dengan pendekatan ini.

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-24 Serupa dengan Soal-19, mari kita asumsikan bahwa bilangan-bilangan tersebut berada dalam rentang 1 sampai n. Juga, n – 1 elemen berulang tiga kali dan elemen yang tersisa diulang dua kali. Temukan elemen yang berulang dua kali.

Solusi: Jika kita XOR semua elemen dalam array dan semua bilangan bulat dari 1 hingga n, maka semua elemen yang diulang tiga kali akan menjadi nol. Ini karena, karena elemen tersebut berulang tiga kali dan XOR lain kali dari jangkauan membuat elemen itu muncul empat kali. Hasilnya, keluaran dari a XOR a XOR a XOR a = 0. Ini adalah kasus yang sama dengan semua elemen yang diulang tiga kali.

Dengan logika yang sama, untuk elemen yang berulang dua kali, jika kita XOR elemen input dan juga range, maka jumlah kemunculan elemen tersebut adalah 3. Akibatnya, *output* dari a XOR a XOR a = a. Akhirnya, kita mendapatkan elemen yang diulang dua kali.

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-25 Diberikan sebuah array dari n elemen. Temukan dua elemen dalam array sedemikian rupa sehingga jumlah mereka sama dengan elemen K yang diberikan.

Solusi Brute Force: Salah satu solusi sederhana untuk ini adalah, untuk setiap elemen input, periksa apakah ada elemen yang jumlahnya K. Ini dapat kita selesaikan hanya dengan menggunakan dua loop for sederhana. Kode untuk solusi ini dapat diberikan sebagai:

```
void BruteForceSearch[int A[], int n, int K) {
    for (int i = 0; i < n; i++) {
        for(int j = i; j < n; j++) {
            if(A[i]+A[j] == K) {
                printf("Items Found:%d %d", i, j);
                return;
            }
        }
    }
    printf("Items not found: No such elements");
}
```

Kompleksitas Waktu: O(n2). Ini karena dua loop for bersarang.
Kompleksitas Ruang: O(1).

Soal-26 Untuk Soal-25, dapatkah kita meningkatkan kompleksitas waktu?

Solusi: Ya. Mari kita asumsikan bahwa kita telah mengurutkan array yang diberikan. Operasi ini membutuhkan O(nlogn). Pada array yang diurutkan, pertahankan indeks loIndex = 0 dan hiIndex = n – 1 dan hitung A[loIndex] + A[hiIndex]. Jika jumlahnya sama dengan K, maka kita selesai dengan solusinya. Jika jumlahnya kurang dari K, turunkan hiIndex, jika jumlahnya lebih besar dari K, naikkan loIndex.

```
void Search[int A[], int n, int K) {
    int loIndex, hiIndex, sum;
    Sort(A, n);
    for(loIndex = 0, hiIndex = n-1; loIndex < hiIndex) {
        sum = A[loIndex] + A[hiIndex];
        if(sum == K) {
            printf("Elements Found: %d  %d", loIndex, hiIndex);
            return;
        }
        else if(sum < K)
            loIndex = loIndex + 1;
        else    hiIndex = hiIndex - 1;
    }
    return;
}
```

Kompleksitas Waktu: O(nlogn). Jika array yang diberikan sudah diurutkan maka kompleksitasnya adalah O(n).
Kompleksitas Ruang: O(1).

Soal-27 Apakah solusi Soal-25 bekerja meskipun array tidak diurutkan?

Solusi: Ya. Karena kita memeriksa semua kemungkinan, Algoritma memastikan bahwa kita mendapatkan pasangan angka jika ada.

Soal-28 Apakah ada cara lain untuk menyelesaikan Soal-25?

Solusi: Ya, menggunakan tabel hash. Karena tujuan kita adalah untuk menemukan dua indeks dari array yang jumlahnya adalah K. Misalkan indeks tersebut adalah X

dan Y. Artinya, A[X] + A[Y] = K. Yang kita butuhkan adalah, untuk setiap elemen dari array input A[X], periksa apakah K – A[X] juga ada di array input. Sekarang, mari kita sederhanakan pencarian itu dengan tabel hash.

Algoritma:

- Untuk setiap elemen array input, masukkan ke dalam tabel hash. Katakanlah elemen saat ini adalah A[X].
- Sebelum melanjutkan ke elemen berikutnya kita periksa apakah K – A[X] juga ada di tabel hash atau tidak.
- Adanya angka tersebut menunjukkan bahwa kita dapat menemukan indeks.
- Jika tidak, lanjutkan ke elemen input berikutnya.

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-29 Diberikan sebuah larik A dengan n elemen. Temukan tiga indeks, i,j & k sedemikian rupa sehingga A[i]2 + A[j]2 = A[k]2?

Solusi:

Algoritma:

- Urutkan array yang diberikan di tempat.
- Untuk setiap indeks array i hitung A[i]2 dan simpan dalam array.
- Cari 2 angka dalam larik dari 0 hingga i – 1 yang menambah A[i] mirip dengan Soal- 25. Ini akan memberi kita hasil dalam waktu O(n). Jika kita menemukan jumlah seperti itu, kembalikan true, jika tidak lanjutkan.

```
Sort(A); // Sort the input array
for (int i=0; i < n; i++)
      A[i] = A[i]*A[i];
for (i=n; i > 0; i--) {
      res = false;
      if(res) {
            //Problem-11/12 Solution
      }
}
```

Kompleksitas Waktu: Waktu untuk menyortir + n × (Waktu untuk mencari jumlah) = O(nlogn) + n × O(n)= n2.

Kompleksitas Ruang: O(1).

Soal-30 Dua elemen yang jumlahnya paling dekat dengan nol. Diberikan sebuah array dengan bilangan positif dan negatif, temukan dua elemen sedemikian rupa sehingga jumlah mereka paling dekat dengan nol. Untuk array di bawah ini, algoritma harus memberikan -80 dan 85. Contoh: 1 60 – 10 70 – 80 85

Solusi Brute Force: Untuk setiap elemen, temukan jumlah dengan setiap elemen lain dalam array dan bandingkan jumlahnya. Akhirnya, kembalikan jumlah minimum.

```
void TwoElementsWithMinSum(int A[], int n) {
    int i, j, min_sum, sum, min_i, min_j, inv_count = 0;
    if(n < 2) {
            printf("Invalid Input");
            return;
    }
    /* Initialization of values */
    min_i = 0;
    min_j = 1;
    min_sum = A[0] + A[1];
    for(i= 0; i < n - 1; i ++)      {
            for(j = i + 1; j < n; j++)      {
                    sum = A[i] + A[j];
                    if(abs(min_sum) > abs(sum)) {
                            min_sum = sum;
                            min_i = i;
                            min_j = j;
                    }
            }
    }
    printf(" The two elements are %d and %d", arr[min_i], arr[min_j]);
}
```

Kompleksitas waktu: O(n2).
Kompleksitas Ruang: O(1).

Soal-31 Bisakah kita meningkatkan kompleksitas waktu Soal-30?

Solusi: Gunakan Penyortiran.

Algoritma:

1. Urutkan semua elemen dari array input yang diberikan.
2. Pertahankan dua indeks, satu di awal (i = 0) dan yang lainnya di akhir (j = n – 1). Juga, pertahankan dua variabel untuk melacak jumlah positif terkecil yang paling dekat dengan nol dan jumlah negatif terkecil yang paling dekat dengan nol.
3. Sedangkan i < j:
    A. Jika jumlah pasangan saat ini > nol dan < postiveClosest maka perbarui postiveClosest. pengurangan j.
    B. Jika jumlah pasangan saat ini < nol dan > negatifClosest maka perbarui negatifClosest. Kenaikan i.
    C. Jika tidak, cetak pasangan

```
void TwoElementsWithMinSum(int A[], int n) {
    int i = 0, j = n-1, temp, postiveClosest = INT_MAX, negativeClosest = INT_MIN;
    Sort(A, n);
    while(i < j) {
        temp  = A[i] + A[j];
        if(temp  > 0) {
            if (temp < postiveClosest)
                postiveClosest = temp;
            j--;
        }
        else if (temp  < 0) {
            if (temp > negativeClosest)
                negativeClosest = temp;
            i++;
        }
        else printf("Closest Sum: %d ", A[i] + A[j]);
    }
    return (abs(negativeClosest)> postiveClosest: postiveClosest: negativeClosest);
}
```

Kompleksitas Waktu: O(nlogn), untuk menyortir.
Kompleksitas Ruang: O(1).

Soal-32 Diberikan sebuah array dari n elemen. Temukan tiga elemen dalam array sedemikian rupa sehingga jumlah mereka sama dengan elemen K yang diberikan?

Solusi Brute Force: Solusi default untuk ini adalah, untuk setiap pasangan elemen input periksa apakah ada elemen yang jumlahnya K. Ini dapat kita selesaikan hanya dengan menggunakan tiga loop for sederhana. Kode untuk solusi ini dapat diberikan sebagai:

```
void BruteForceSearch[int A[], int n, int data) {
    for (int i = 0; i < n; i++) {
        for(int j = i+1; j < n; j++) {
            for(int k = j+1; k < n; k++)        {
                if(A[i] + A[j] + A[k]== data) {
                    printf("Items Found:%d %d %d", i, j, k);
                    return;
                }
            }
        }
    }
    printf("Items not found: No such elements");
}
```

Kompleksitas Waktu: O(n3), untuk tiga loop for bersarang.
Kompleksitas Ruang: O(1).

Soal-33 Apakah solusi Soal-32 bekerja meskipun array tidak diurutkan?

Solusi: Ya. Karena kita memeriksa semua kemungkinan, Algoritma memastikan bahwa kita dapat menemukan tiga angka yang jumlahnya K jika ada.

Soal-34 Bisakah kita menggunakan teknik pengurutan untuk menyelesaikan Soal-32?

Solusi: Ya.

```
void Search[int A[], int n, int data) {
        Sort(A, n);
        for(int k = 0; k < n; k++) {
                for(int i =  k + 1, j = n-1; i < j;  ) {
                        if(A[k] + A[i] + A[j]  == data) {
                                printf("Items Found:%d %d %d", i, j, k);
                                return;
                        }
                        else if(A[k] + A[i] + A[j]  < data)
                                i =  i + 1;
                        else    j =  j - 1;
                }
        }
        return;
}
```

Kompleksitas Waktu: Waktu untuk menyortir + Waktu untuk mencari dalam daftar terurut = O(nlogn) + $O(n^2)$ $O(n^2)$. Ini karena dua loop for bersarang.
Kompleksitas Ruang: O(1).

Soal-35 Bisakah kita menggunakan teknik hashing untuk menyelesaikan Soal-32?

Solusi: Ya. Karena tujuan kita adalah menemukan tiga indeks larik yang jumlahnya K. Misalkan indeks tersebut adalah X,Y dan Z. Artinya, A[X] + A[Y] + A[Z] = K.
Mari kita asumsikan bahwa kita telah menyimpan semua jumlah yang mungkin beserta pasangannya dalam tabel hash. Itu berarti kunci tabel hash adalah K – A[X] dan nilai untuk K – A[X] adalah semua kemungkinan pasangan input yang jumlahnya jika – A[X].

Algoritma:

- Sebelum memulai pencarian, masukkan semua jumlah yang mungkin dengan pasangan elemen ke dalam tabel hash.
- Untuk setiap elemen dari array input, masukkan ke dalam tabel hash. Katakanlah elemen saat ini adalah A[X].
- Periksa apakah ada entri hash dalam tabel dengan kunci: K – A[X].
- Jika elemen tersebut ada maka pindai pasangan elemen K – A[X] dan kembalikan semua pasangan yang mungkin dengan memasukkan A[X] juga.
- Jika tidak ada elemen seperti itu (dengan K – A[X] sebagai kuncinya), lanjutkan ke elemen berikutnya.

Kompleksitas Waktu: Waktu untuk menyimpan semua kemungkinan pasangan dalam tabel Hash + pencarian = $O(n^2)$ + $O(n^2)$ $O(n^2)$.
Kompleksitas Ruang: O(n).

Soal-36 Diberikan sebuah array n bilangan bulat, masalah 3 – jumlah adalah menemukan tiga bilangan bulat yang jumlahnya paling dekat dengan nol.

Solusi: Ini sama dengan Soal-32 dengan nilai K adalah nol.

Soal-37 Misalkan A adalah larik dari n bilangan bulat berbeda. Misalkan A memiliki properti berikut: terdapat indeks 1 k ≤ n sedemikian rupa sehingga A[l],..., A[k] adalah barisan naik dan A[k + 1],..., A[n] adalah barisan menurun. Merancang dan menganalisis algoritma yang efisien untuk menemukan k.

Pertanyaan serupa: Mari kita asumsikan bahwa array yang diberikan diurutkan tetapi dimulai dengan angka negatif dan diakhiri dengan angka positif [fungsi seperti itu disebut fungsi yang meningkat secara monoton]. Dalam larik ini temukan indeks awal dari bilangan positif. Asumsikan bahwa kita mengetahui panjang array input. Rancang algoritma O(logn).

Solusi: Mari kita gunakan varian dari pencarian biner.

```
int Search (int A[], int n, int first, int last) {
int mid, first = 0, last = n-1;
while(first <= last) {
      // if the current array has size 1
      if(first == last)
            return A[first];
      // if the current array has size 2
      else if(first == last-1)
            return max(A[first], A[last]);
      // if the current array has size 3 or more
      else {
            mid = first + (last-first)/2;
            if(A[mid-1] < A[mid] && A[mid] > A[mid+1])
                  return A[mid];
            else if(A[mid-1] < A[mid] && A[mid] < A[mid+1])
                  first = mid+1;
            else if(A[mid-1] > A[mid] && A[mid] > A[mid+1])
                  last = mid-1;
            else      return INT_MIN ;
       } // end of else
    } // end of while
}
```

Persamaan rekursinya adalah T(n) = 2T(n/2) + c. Menggunakan teorema master, kita mendapatkan O(logn).

Soal-38 Jika kita tidak tahu n, bagaimana kita menyelesaikan Soal-37?

Solusi: Hitung berulang kali A[1],A[2],A[4],A[8],A[16] dan seterusnya, sampai kita menemukan nilai n sehingga A[n] > 0.

Kompleksitas Waktu: O(logn), karena kita bergerak dengan laju 2. Lihat bab Pengantar Analisis Algoritma untuk detailnya.

Soal-39 Diberikan array input dengan ukuran yang tidak diketahui dengan semua 1 di awal dan 0 di akhir. Temukan indeks dalam array dari mana 0 dimulai. Pertimbangkan ada jutaan 1 dan 0 dalam array. Misalnya. isi array 1111111……..1100000 0000000.

Solusi: Soal ini hampir mirip dengan Soal-38. Periksa bit pada tingkat 2K di mana k = 0,1,2 Karena kita bergerak dengan laju 2, kompleksitasnya adalah O(logn).

Soal-40 Diberikan sebuah larik terurut dari n bilangan bulat yang telah diputar berkali-kali yang tidak diketahui, berikan algoritma O(logn) yang menemukan elemen dalam larik tersebut.

Contoh: Cari 5 dalam array (15 16 19 20 25 1 3 4 5 7 10 14) *Output*: 8 (indeks 5 dalam array)

Solusi: Mari kita asumsikan bahwa array yang diberikan adalah A[]dan gunakan solusi Soal-37 dengan ekstensi. Fungsi di bawah FindPivot mengembalikan nilai k (mari kita asumsikan bahwa fungsi ini mengembalikan indeks, bukan nilai). Temukan titik pivot, bagi array menjadi dua sub-array dan panggil pencarian biner.

Ide utama untuk menemukan titik pivot adalah – untuk array yang diurutkan (dalam urutan meningkat) dan berporos, elemen pivot adalah satu-satunya elemen yang elemen berikutnya lebih kecil darinya. Dengan menggunakan kriteria di atas dan metodologi pencarian biner, kita bisa mendapatkan elemen pivot dalam waktu O(logn).

Algoritma:

1) Cari tahu titik pivot dan bagi array menjadi dua sub-array.
2) Sekarang panggil pencarian biner untuk salah satu dari dua sub-array.
   A. jika elemen lebih besar dari elemen pertama maka cari di subarray kiri.
   B. pencarian lain di subarray kanan.
3) Jika elemen ditemukan dalam sub-array yang dipilih, maka kembalikan indeks jika tidak, kembalikan -1.

```
int FindPivot(int A[], int start, int finish) {
      if(finish - start == 0)
            return start;
      else if(start == finish - 1) {
            if(A[start] >= A[finish])
                  return start;
            else      return finish;
      }
      else {
            mid = start + (finish-start)/2;
            if(A[start] >= A[mid])
                  return FindPivot(A, start, mid);
            else      return FindPivot(A, mid, finish);
      }
}
int Search(int A[], int n, int x) {
      int pivot = FindPivot(A, 0, n-1);
    if(A[pivot] == x)
        return pivot;
    if(A[pivot] <= x)
        return BinarySearch(A, 0, pivot-1, x);
    else return BinarySearch(A, pivot+1, n-1, x);
}
int BinarySearch(int A[], int low, int high, int x) {
      if(high >= low)   {
            int mid = low + (high - low)/2;
            if(x == A[mid])
                  return mid;
            if(x > A[mid])
                  return BinarySearch(A, (mid + 1), high, x);
            else      return BinarySearch(A, low, (mid -1), x);
      }
      return -1;        //-1 if element is not found
}
```

Kompleksitas waktu: O(logn).

Soal-41 Untuk Soal-40, dapatkah kita menyelesaikannya dengan rekursi?

Solusi Ya.

```
int BinarySearchRotated(int A[], int start, int finish, int data) {
      int mid = start + (finish - start) / 2;
      if(start > finish)
            return -1;
      if(data == A[mid])
            return mid;
      else if(A[start] <= A[mid]) {         // start half is in sorted order.
            if(data >= A[start] && data < A[mid])
                  return BinarySearchRotated(A, start, mid - 1, data);
            else      return BinarySearchRotated(A, mid + 1, finish, data);
      }
      else {    // A[mid] <= A[finish], finish half is in sorted order.
            if(data > A[mid] && data <= A[finish])
                  return BinarySearchRotated(A, mid + 1, finish, data);
            else      return BinarySearchRotated(A, start, mid - 1, data);
      }
}
```

Kompleksitas waktu: O(logn).

Soal-42 Pencarian bitonic: Suatu array dikatakan bitonic jika terdiri dari barisan bilangan bulat yang meningkat diikuti segera oleh barisan bilangan bulat yang menurun. Diberikan array bitonic A dari n bilangan bulat yang berbeda, jelaskan cara menentukan apakah bilangan bulat yang diberikan ada dalam larik dalam langkah-langkah O(logn).

Solusi Solusinya sama dengan Soal-37.

Soal-43 Namun, cara lain untuk membingkai Soal-37.
Biarkan A[] menjadi array yang mulai meningkat, mencapai maksimum, dan kemudian menurun. Rancang algoritma O(logn) untuk menemukan indeks dari nilai maksimum.

Soal-44 Berikan algoritma O(nlogn) untuk menghitung median dari barisan n bilangan bulat.

Solusi Urutkan dan kembalikan elemen di .

Soal-45 Diberikan dua daftar terurut berukuran m dan n, cari median semua elemen dalam O(log (m+ n)) waktu.

Solusi Lihat bab *Divide and conquer*.

Soal-46 Diberikan array A yang diurutkan dari n elemen, mungkin dengan duplikat, temukan indeks kemunculan pertama dari suatu angka dalam waktu O(logn).

Solusi: Untuk menemukan kemunculan pertama suatu bilangan, kita perlu memeriksa kondisi berikut.
Kembalikan posisi jika salah satu dari berikut ini benar:

```
mid == low && A[mid] == data  ||  A[mid] == data && A[mid-1] < data
```

```
int BinarySearchFirstOccurrence(int A[], int low, int high, int data) {
    int mid;
    if(high >= low) {
        mid = low + (high-low) / 2;
        if((mid == low && A[mid] == data) || (A[mid] == data && A[mid - 1] < data))
            return mid;
        // Give preference to left half of the array
        else if(A[mid] >= data)
            return BinarySearchFirstOccurrence (A, low, mid - 1, data);
        else    return BinarySearchFirstOccurrence (A, mid + 1, high, data);
    }
    return -1;
}
```

Kompleksitas Waktu: O(logn).

Soal-47 Diberikan array A yang diurutkan dari n elemen, mungkin dengan duplikat. Temukan indeks kemunculan terakhir angka dalam waktu O(logn).

Solusi: Untuk menemukan kemunculan terakhir dari suatu angka, kita perlu memeriksa kondisi berikut. Kembalikan posisi jika salah satu dari berikut ini benar:

```
mid == high && A[mid] == data  ||  A[mid] == data && A[mid+1] > data
```

```
int BinarySearchLastOccurrence(int A[], int low, int high, int data) {
    int mid;
    if(high >= low) {
        mid = low + (high-low) / 2;
        if((mid == high && A[mid] == data) || (A[mid] == data && A[mid + 1] > data))
            return mid;
        // Give preference to right half of the array
        else if(A[mid] <= data)
            return BinarySearchLastOccurrence (A, mid + 1, high, data);
        else    return BinarySearchLastOccurrence (A, low, mod - 1, data);
    }
    return -1;
}
```

Kompleksitas Waktu: O(logn).

Soal-48 Diberikan array n elemen yang diurutkan, mungkin dengan duplikat. Menemukan jumlah kemunculan suatu bilangan.

Solusi Brute Force: Lakukan pencarian linier dari array dan hitungan kenaikan saat dan ketika kita menemukan data elemen dalam array.

```
int LinearSearchCount(int A[], int n, int data) {
    int count = 0;
    for (int i = 0; i < n; i++)
        if(A[i] == data)
            count++;
    return count;
}
```

Kompleksitas Waktu: O(n).

Soal-49 Bisakah kita meningkatkan kompleksitas waktu Soal-48?

Solusi: Ya. Kita dapat menyelesaikan ini dengan menggunakan satu panggilan pencarian biner diikuti dengan pemindaian kecil lainnya.

Algoritma:

- Lakukan pencarian biner untuk data dalam array. Mari kita asumsikan posisinya adalah K.
- Sekarang telusuri ke kiri dari K dan hitung jumlah kemunculan data. Biarkan hitungan ini menjadi leftCount.

- Demikian pula, traverse ke kanan dan hitung jumlah kemunculan data. Biarkan hitungan ini menjadi rightCount.
- Jumlah total kemunculan = leftCount + 1 + rightCount

Kompleksitas Waktu – O(logn + S) di mana 5 adalah jumlah kemunculan data.

Soal-50 Apakah ada cara alternatif untuk menyelesaikan Soal-48?

Solusi:

Algoritma:

- Temukan kemunculan pertama data dan panggil indeksnya sebagai FirstOccurrence (untuk algoritma lihat Soal-46)
- Temukan kemunculan terakhir data dan panggil indeksnya sebagai LastOccurrence (untuk algoritma lihat Soal-47)
- Kembalikan kejadian terakhir – kejadian pertama + 1 Kompleksitas Waktu = O(logn + logn) = O(logn).

Soal-51 Berapakah bilangan selanjutnya dari barisan 1,11,21 dan mengapa?

Solusi: Baca nomor yang diberikan dengan keras. Ini hanya masalah yang menyenangkan.

One One
Two Ones
One two, one one→ 1211

Jadi jawabannya adalah: angka berikutnya adalah representasi dari angka sebelumnya dengan membacanya dengan keras.

Soal-52 Menemukan bilangan terkecil kedua secara efisien.

Solusi: Kita dapat membuat tumpukan elemen yang diberikan hanya dengan menggunakan kurang dari n perbandingan (Lihat bab Antrian Prioritas untuk Algoritma). Kemudian kita menemukan yang terkecil kedua menggunakan perbandingan logn untuk operasi GetMax(). Secara keseluruhan, kita mendapatkan n + logn + konstan.

Soal-53 Apakah ada solusi lain untuk Soal-52?

Solusi: Sebagai alternatif, bagi n angka menjadi kelompok 2, lakukan perbandingan n/2 berturut-turut untuk menemukan yang terbesar, menggunakan metode seperti turnamen. Putaran pertama akan menghasilkan maksimum dalam n – 1 perbandingan. Putaran kedua akan dilakukan pada pemenang putaran pertama dan yang paling banyak muncul. Ini akan menghasilkan logn – 1 perbandingan untuk total n + logn – 2. Solusi di atas disebut masalah turnamen.

Soal-54 Suatu elemen dikatakan mayoritas jika muncul lebih dari n/2 kali. Berikan suatu algoritma mengambil array elemen n sebagai argumen dan mengidentifikasi mayoritas (jika ada).

Solusi: Solusi dasarnya adalah memiliki dua loop dan melacak jumlah maksimum untuk semua elemen yang berbeda. Jika jumlah maksimum menjadi lebih besar dari n/2, maka putuskan loop dan kembalikan elemen yang memiliki jumlah maksimum. Jika jumlah maksimum tidak lebih dari n/2, maka elemen mayoritas tidak ada.

Kompleksitas Waktu: $O(n^2)$.

Kompleksitas Ruang: O(1).

Soal-55 Bisakah kita meningkatkan kompleksitas waktu Soal-54 menjadi O(nlogn)?

Solusi: Menggunakan pencarian biner kita dapat mencapai ini. Node Pohon Pencarian Biner (digunakan dalam pendekatan ini) adalah sebagai berikut.

```
struct TreeNode {
    int element;
    int count;
    struct TreeNode *left;
    struct TreeNode *right;
} BST;
```

Masukkan elemen di BST satu per satu dan jika elemen sudah ada, tambahkan jumlah node. Pada tahap apa pun, jika jumlah simpul menjadi lebih dari n/2, maka kembalilah. Metode ini bekerja dengan baik untuk kasus di mana n/2 +1 kemunculan elemen mayoritas hadir di awal array, misalnya {1,1,1,1,1,2,3, dan 4}.

Kompleksitas Waktu: Jika pohon pencarian biner digunakan maka kompleksitas waktu terburuk adalah O(n2). Jika pohon pencarian biner seimbang digunakan maka O(nlogn).

Kompleksitas Ruang: O(n).

Soal-56 Apakah ada cara lain untuk mencapai kompleksitas O(nlogn) untuk Soal-54?

Solusi: Urutkan array input dan pindai array yang diurutkan untuk menemukan elemen mayoritas.

Kompleksitas Waktu: O(nlogn).

Kompleksitas Ruang: O(1).

Soal-57 Bisakah kita meningkatkan kompleksitas Soal-54?

Solusi: Jika suatu elemen muncul lebih dari n/2 kali di A maka itu pasti median dari A. Tapi, kebalikannya tidak benar, jadi setelah median ditemukan, kita harus memeriksa untuk melihat berapa kali muncul di A. Kita dapat menggunakan

seleksi linier yang membutuhkan waktu O(n) (untuk algoritma, lihat bab Algoritma Seleksi). int CheckMajority(int A[], dalam n) {

1) Gunakan seleksi linier untuk mencari median m dari A.
2) Lakukan satu kali lagi melewati A dan hitung jumlah kemunculan m.
   A. Jika m muncul lebih dari n/2 kali maka kembalikan true;
   B. Jika tidak, kembalikan false.

Soal-58 Apakah ada cara lain untuk menyelesaikan Soal-54?

Solusi: Karena hanya satu elemen yang berulang, kita dapat menggunakan pemindaian sederhana dari array input dengan melacak jumlah elemen. Jika hitungannya adalah 0, maka kita dapat mengasumsikan bahwa elemen dikunjungi untuk pertama kalinya, sebaliknya elemen yang dihasilkan.

```
int MajorityNum(int[] A, int n) {
  int count = 0, element = -1;
  for(int i = 0; i < n; i++) {
     // If the counter is 0 then set the current candidate to majority num and set the counter to 1.
     if(count == 0) {
              element = A[i];
              count = 1;
     }
     else if(element == A[i]) {
              // Increment counter If the counter is not 0 and element is same as current candidate.
              count++;
     }
     else {
            // Decrement counter If the counter is not 0 and element is different from current candidate.
            count--;
     }
  }
  return element;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-59 Diberikan sebuah array dengan 2n elemen yang n elemennya sama dan n elemen lainnya semuanya berbeda. Temukan elemen mayoritas.

Solusi: Elemen yang diulang akan menempati setengah dari array. Apa pun pengaturannya, hanya satu dari yang di bawah ini yang benar:

- Semua elemen duplikat akan berada pada jarak relatif 2 dari satu sama lain.
  Contoh: n, 1, n, 100, n, 54, n...
- Setidaknya dua elemen duplikat akan bersebelahan.
  Contoh: n,n, 1.100, n, 54, n,....
  n, 1,n,n,n,54,100...
  1.100,54, n.n.n.n....

Dalam kasus terburuk, kita akan membutuhkan dua lintasan di atas array:

- Pass Pertama: bandingkan A[i] dan A[i+1]
- Pass Kedua: bandingkan A[i] dan A[i + 2]

Sesuatu akan cocok dan itulah elemen Anda. Ini akan memakan biaya O(n) dalam waktu dan O(1) dalam ruang.

Soal-60 Diberikan sebuah array dengan 2n + 1 elemen integer, n elemen muncul dua kali di sembarang tempat dalam array dan satu integer hanya muncul sekali di suatu tempat di dalam.

Temukan bilangan bulat tunggal dengan operasi O(n) dan memori tambahan O(1).

Solusi: Kecuali satu elemen, semua elemen diulang. Kita tahu bahwa A XOR A = 0. Berdasarkan ini jika kita XOR semua elemen input maka kita mendapatkan elemen yang tersisa.

```
int Solution(int* A) {
      int i, res;
      for (i = res = 0; i < 2n+1; i++)
         res = res ^ A[i];
      return res;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1).

Soal-61 Melempar telur dari gedung berlantai n: Misalkan kita memiliki gedung berlantai n dan sejumlah telur. Juga asumsikan bahwa telur pecah jika dilempar dari lantai F atau lebih tinggi, dan sebaliknya tidak akan pecah. Rancang strategi untuk menentukan lantai F, sambil memecahkan telur O(logn).

Solusi: Lihat bab *Divide and conquer*.

Soal-62 Minimum lokal dari sebuah array: Diberikan sebuah array A dari n bilangan bulat yang berbeda, rancang sebuah algoritma O(logn) untuk menemukan minimum lokal: sebuah indeks i sedemikian rupa sehingga A[i – 1] < A[i] < A[saya+1].

Solusi: Periksa nilai tengah A[n/2], dan dua tetangga A[n/2 – 1] dan A[n/2 + 1]. Jika A[n/2] adalah minimum lokal, stop; jika tidak cari di setengah dengan tetangga yang lebih kecil.

Soal-63 Berikan array n × n elemen sedemikian rupa sehingga setiap baris dalam urutan menaik dan setiap kolom dalam urutan menaik, buatlah algoritma O(n) untuk

menentukan apakah elemen x yang diberikan ada dalam array. Anda dapat menganggap semua elemen dalam array n × n berbeda.

Solusi: Mari kita asumsikan bahwa matriks yang diberikan adalah A[n][n]. Mulailah dengan baris terakhir, kolom pertama [atau baris pertama, kolom terakhir]. Jika elemen yang kita cari lebih besar dari elemen di A[1][n], maka kolom pertama dapat dihilangkan. Jika elemen pencarian lebih kecil dari elemen pada A[1][n], maka baris terakhir dapat dihilangkan seluruhnya. Setelah kolom pertama atau baris terakhir dihilangkan, mulailah proses lagi dengan ujung kiri-bawah larik yang tersisa. Dalam algoritma ini, akan ada n elemen maksimum yang akan dibandingkan dengan elemen pencarian.

Kompleksitas Waktu: O(n). Ini karena kita akan melintasi paling banyak 2n titik.
Kompleksitas Ruang: O(1).

Soal-64 Diberikan sebuah array n × n a dari n2 bilangan, berikan algoritma O(n) untuk menemukan pasangan indeks i dan j sedemikian rupa sehingga A[i][j] < A[i + 1][j]. A[i][j] < A[i][j + 1],A[i][j] < A[i – 1][j], dan A[i][j] < A[i][j-1].

Solusi: Masalah ini sama dengan Soal-63.

Soal-65 Diberikan matriks n × n, dan di setiap baris semua 1 diikuti oleh 0. Temukan baris dengan jumlah maksimum 0.

Solusi: Mulailah dengan baris pertama, kolom terakhir. Jika elemennya 0 maka pindah ke kolom sebelumnya di baris yang sama dan pada saat yang sama tingkatkan penghitung untuk menunjukkan jumlah maksimum 0. Jika elemennya 1 maka pindah ke baris berikutnya di kolom yang sama. Ulangi proses ini sampai Anda mencapai baris terakhir, kolom pertama.

Kompleksitas Waktu: O(2n) O(n) (mirip dengan Soal-63).

Soal-66 Diberikan sebuah array input dengan ukuran yang tidak diketahui, dengan semua angka di awal dan simbol khusus di akhir. Temukan indeks dalam array dari mana simbol khusus dimulai.

Solusi: Lihat bab *Divide and conquer*.

Soal-67 Pisahkan bilangan genap dan ganjil: Diberikan sebuah larik A[], tulislah sebuah fungsi yang memisahkan bilangan genap dan ganjil. Fungsi harus menempatkan semua angka genap terlebih dahulu, dan kemudian angka ganjil. Contoh: Masukan = {12,34,45,9,8,90,3} Keluaran = {12,34,90,8,9,45,3}

Catatan: *Pada output, urutan angka dapat diubah, yaitu, pada contoh di atas 34 dapat datang sebelum 12, dan 3 dapat datang sebelum 9.*

Solusi Soal ini sangat mirip dengan Pisahkan 0 dan 1 (Soal-68) dalam sebuah array, dan kedua masalah tersebut merupakan variasi dari masalah bendera nasional Belanda yang terkenal.

Algoritma Logikanya mirip dengan Quick sort.

1) Inisialisasi dua variabel indeks kiri dan kanan: kiri = 0, kanan = n – 1
2) Terus bertambah indeks kiri sampai Anda melihat angka ganjil.
3) Terus kurangi indeks yang tepat sampai Anda melihat angka genap.
4) Jika kiri < kanan maka tukar A[kiri] dan A[kanan]

```
void DutchNationalFlag(int A[], int n) {
    int left = 0, right = n-1;
    while(left < right) {
        // Increment left index while we see 0 at left
        while(A[left]%2 == 0 && left < right)
            left++;
        // Decrement right index while we see 1 at right
        while(A[right]%2 == 1 && left < right)
            right--;
        if(left < right) {
            // Swap A[left] and A[right]
            swap(&A[left], &A[right]);
            left++;
            right--;
        }
    }
}
```

Kompleksitas Waktu: O(n).

Soal-68 Berikut ini adalah cara lain untuk menyusun Soal-67, tetapi dengan sedikit perbedaan.

Pisahkan 0 dan 1 dalam sebuah array: Kita diberi array 0 dan 1 dalam urutan acak. Pisahkan 0 di sisi kiri dan 1 di sisi kanan array. Lintasi array hanya sekali.

Larik masukan = [0,1,0,1,0,0,1,1,1,0] Larik keluaran = [0,0,0,0,0,1,1,1,1,1]

Solusi: Menghitung 0 atau 1

1. Hitung jumlah 0. Biarkan hitungannya menjadi C.
2. Setelah kita menghitung, letakkan C 0 di awal dan 1 di sisa n- C posisi dalam array.

Kompleksitas Waktu: O(n). Solusi ini memindai array dua kali.

Soal-69 Bisakah kita memecahkan Masalah-68 dalam satu pemindaian?

Solusi: Ya. Gunakan dua indeks untuk melintasi: Pertahankan dua indeks. Inisialisasi indeks pertama kiri sebagai 0 dan indeks kedua kanan sebagai n – 1. Lakukan hal berikut saat kiri < kanan:

1) Pertahankan indeks kenaikan yang tersisa saat ada Os di dalamnya
2) Pertahankan indeks penurunan dengan benar saat ada Is di dalamnya
3) Jika kiri < kanan maka tukar A[kiri] dan A[kanan]

```
//Function to put all 0s on left and all 1s on right
void Separate0and1(int A[], int n) {
    /* Initialize left and right indexes */
    int left = 0, right = n-1;
    while(left < right) {
        /* Increment left index while we see 0 at left */
        while(A[left] == 0 && left < right)
            left++;
        /* Decrement right index while we see 1 at right */
        while(A[right] == 1 && left < right)
            right--;
        /* If left is smaller than right then there is a 1 at left
        and a 0 at right.  Swap A[left] and A[right]*/
        if(left < right) {
            A[left] = 0;
            A[right] = 1;
            left++;
            right--;
        }
    }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-70 Urutkan sebuah array dari 0, 1 dan 2 [atau R, G dan B]: Diberikan sebuah array A[] yang terdiri dari 0, 1 dan 2, berikan algoritma untuk pengurutan A[].Algoritma harus menempatkan semua 0 pertama, lalu semua 1 dan akhirnya semua 2 di akhir. Contoh Masukan = {0,1,1,0,1,2,1,2,0,0,0,1}, Keluaran = {0,0,0,0,0,1,1,1,1,1,2 ,2}

Solusi:

```
void Sorting012sDutchFlagProblem(int A[],int n){
    int low=0,mid=0,high=n-1;
    while(mid <=high){
        switch(A[mid]){
            case 0:
                swap(A[low],A[mid]);
                low++;mid++;
                break;
            case 1:
                mid++;
                break;
            case 2:
                swap(A[mid],A[high]);
                high--;
                break;
        }
    }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-71 Perbedaan maksimum antara dua elemen: Diberikan sebuah array A[] bilangan bulat, temukan perbedaan antara dua elemen sedemikian rupa sehingga elemen yang lebih besar muncul setelah angka yang lebih kecil di A[].
Contoh: Jika array adalah [2,3,10,6,4,8,1] maka nilai yang dikembalikan harus 8 (Perbedaan antara 10 dan 2). Jika array adalah [7,9,5,6,3,2] maka nilai yang dikembalikan harus 2 (Perbedaan antara 7 dan 9)

Solusi Lihat bab *Divide and conquer*.

Soal-72 Diberikan array 101 elemen. Dari 101 elemen, 25 elemen diulang dua kali, 12 elemen diulang 4 kali, dan satu elemen diulang 3 kali. Temukan elemen yang diulang 3 kali dalam O(1).

Solusi: Sebelum menyelesaikan masalah ini, mari kita perhatikan sifat operasi XOR berikut: a XOR a = 0. Artinya, jika kita menerapkan XOR pada elemen yang sama maka hasilnya adalah 0.

Algoritma:

- XOR semua elemen dari array yang diberikan dan anggap hasilnya adalah A.
- Setelah operasi ini, 2 kemunculan angka yang muncul 3 kali menjadi 0 dan satu kemunculan tetap sama.
- 12 elemen yang muncul 4 kali menjadi 0.
- 25 elemen yang muncul 2 kali menjadi 0.
- Jadi hanya XOR'ing semua elemen memberikan hasil.

Kompleksitas Waktu: O(n), karena kita hanya melakukan satu pemindaian.
Kompleksitas Ruang: O(1).

Soal-73 Diberikan sebuah bilangan n, berikan algoritma untuk mencari jumlah nol yang tertinggal di n!.

Solusi:

```
int NumberOfTrailingZerosInNumber(int n) {
    int i, count = 0;
    if(n < 0)  return -1;
    for (i = 5; n / i > 0; i *= 5)
            count += n / i;
    return count;
}
```

Kompleksitas Waktu: O(logn).

Soal-74 Diberikan sebuah larik 2n bilangan bulat dalam format berikut a1 a2 a3 ...an b1 b2 b3... bn. Acak array menjadi a1 b1 a2 b2 a3 b3 ... an bn tanpa memori tambahan.

Solusi: Solusi brute force melibatkan dua loop bersarang untuk memutar elemen di paruh kedua larik ke kiri. Loop pertama berjalan n kali untuk mencakup semua elemen di paruh kedua array. Loop kedua memutar elemen ke kiri. Perhatikan bahwa indeks awal pada loop kedua bergantung pada elemen mana yang kita putar dan indeks akhir bergantung pada berapa banyak posisi yang perlu kita pindahkan ke kiri.

```
void ShuffleArray() {
    int n = 4;
    int A[] = {1,3,5,7,2,4,6,8};
    for (int i = 0, q =1, k = n; i < n; i++, k++, q++) {
        for (int j = k; j > i + q; j--) {
            int tmp = A[j-1];
            A[j-1] = A[j];
            A[j] = tmp;
        }
    }
    for (int i = 0; i  < 2*n; i++)
        printf("%d", A[i]);
}
```

Kompleksitas Waktu: $O(n^2)$.

Soal-75 Bisakah kita meningkatkan solusi Soal-74?

Solusi: Lihat bab *Divide and conquer*. Solusi kompleksitas waktu yang lebih baik O(nlogn) dapat dicapai dengan menggunakan teknik Divide and Concur. Mari kita lihat sebuah contoh

1. Mulailah dengan larik: a1 a2 a3 a4 b1 b2 b3 b4
2. Bagi array menjadi dua bagian: a1 a2 a3 a4 : b1 b2 b3 b4
3. Tukar elemen di sekitar pusat: tukar a3 a4 dengan b1 b2 dan Anda mendapatkan: a1 a.2 b1 b2 a3 a4 b3 b4
4. Bagi a1 a2 b1 b2 menjadi a1 a2 : b1 b2. Kemudian bagi a3 a4 b3 b4 menjadi a3 a4 : b3 b4
5. Tukar elemen di sekitar pusat untuk setiap subarray yang Anda dapatkan: a1 b1 a2 b2 dan a3 b3 a4 b4

Perhatikan bahwa solusi ini hanya menangani kasus ketika n = 2i di mana i = 0,1,2,3, dll. Dalam contoh kita n = 22 = 4 yang memudahkan untuk membagi array secara rekursif menjadi dua bagian. Ide dasar di balik menukar elemen di sekitar pusat sebelum memanggil fungsi rekursif adalah untuk menghasilkan masalah ukuran yang lebih kecil. Solusi dengan kompleksitas waktu linier dapat dicapai jika elemen-elemennya bersifat spesifik. Misalnya, jika Anda dapat

menghitung posisi baru elemen menggunakan nilai elemen itu sendiri. Ini tidak lain adalah teknik hashing.

Soal-76 Diberikan sebuah larik A[], carilah j – i maksimum sedemikian hingga A[j] > A[i]. Misalnya, Input: {34, 8, 10, 3, 2, 80, 30, 33, 1} dan *Output*: 6 (j = 7, i = 1).

Solusi: Pendekatan Brute Force: Jalankan dua loop. Di loop luar, pilih elemen satu per satu dari kiri. Di loop dalam, bandingkan elemen yang dipilih dengan elemen yang dimulai dari sisi kanan. Hentikan loop dalam saat Anda melihat elemen yang lebih besar dari elemen yang dipilih dan terus perbarui j – i maksimum sejauh ini.

```
int maxIndexDiff(int A[], int n){
   int maxDiff = -1;
   int i, j;
    for (i = 0; i < n; ++i){
       for (j = n-1; j > i; --j){
          if(A[j] > A[i] && maxDiff < (j - i))
             maxDiff = j - i;
       }
    }
    return maxDiff;
}
```

Kompleksitas Waktu: $O(n^2)$.
Kompleksitas Ruang: O(1).

Soal-77 Bisakah kita meningkatkan kompleksitas Soal-76?

Solusi: Untuk mengatasi masalah ini, kita perlu mendapatkan dua indeks optimal A[]: indeks kiri i dan indeks kanan j. Untuk elemen A[i], kita tidak perlu mempertimbangkan A[i] untuk indeks kiri jika ada elemen yang lebih kecil dari A[i] di sisi kiri A[i]. Demikian pula, jika ada elemen yang lebih besar di sisi kanan A[j] maka kita tidak perlu mempertimbangkan j ini untuk indeks yang tepat.

Jadi kita membangun dua Array tambahan LeftMins[] dan RightMaxs[] sedemikian rupa sehingga LeftMins[i] memegang elemen terkecil di sisi kiri A[i] termasuk A[i], dan RightMaxs[j] memegang elemen terbesar di sebelah kanan sisi A[j] termasuk A[j]. Setelah membangun dua array tambahan ini, kita menelusuri kedua array ini dari kiri ke kanan.

Saat melintasi LeftMins[] dan RightMaxs[], jika kita melihat bahwa LeftMins[i] lebih besar dari RightMaxs[j], maka kita harus bergerak maju di LeftMins[] (atau melakukan i++) karena semua elemen di sebelah kiri LeftMins[i] lebih besar dari atau sama dengan LeftMins[i]. Jika tidak, kita harus bergerak maju di RightMaxs[j] untuk mencari nilai y – i yang lebih besar.

```
int maxIndexDiff(int A[], int n){
  int maxDiff, i, j;
  int *LeftMins = (int *)malloc(sizeof(int)*n);
  int *RightMaxs = (int *)malloc(sizeof(int)*n);
  LeftMins[0] = A[0];
  for (i = 1; i < n; ++i)
     LeftMins[i] = min(A[i], LeftMins[i-1]);
  RightMaxs[n-1] = A[n-1];
  for (j = n-2; j >= 0; --j)
     RightMaxs[j] = max(A[j], RightMaxs[j+1]);
  i = 0, j = 0, maxDiff = -1;
  while (j < n && i < n){
     if (LeftMins[i] < RightMaxs[j]){
        maxDiff = max(maxDiff, j-i);
        j = j + 1;
     }
     else
        i = i+1;
  }
   return maxDiff;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-78 Mengingat array elemen, bagaimana Anda memeriksa apakah daftar diurutkan berpasangan atau tidak? Sebuah daftar dianggap diurutkan berpasangan jika setiap pasangan angka berurutan dalam urutan (tidak menurun).

Solusi:

```
int checkPairwiseSorted(int A[], int n) {
  if (n == 0 || n == 1)
    return 1;
  for (int i = 0; i < n - 1; i += 2){
    if (A[i] > A[i+1])
      return 0;
  }
  return 1;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-79 Diberikan array n elemen, bagaimana Anda mencetak frekuensi elemen tanpa menggunakan ruang ekstra. Asumsikan semua elemen positif, dapat diedit dan kurang dari n.

Solusi Gunakan teknik negasi.

```
void frequencyCounter(int A[],int n){
  int pos = 0;
  while(pos < n){
     int expectedPos = A[pos] - 1;
     if(A[pos] > 0 && A[expectedPos] > 0){
        swap(A[pos], A[expectedPos]);
        A[expectedPos] = -1;
```

```
        }
        else if(A[pos] > 0){
            A[expectedPos] --;
            A[pos ++] = 0;
        }
        else{
            pos ++;
        }
    }
    for(int i = 0; i < n; ++i){
        printf("%d frequency is %d\n", i + 1 ,abs(A[i]));
    }
}
int main(int argc, char* argv[]){
    int A[] = {10, 10, 9, 4, 7, 6, 5, 2, 3, 2, 1};
    frequencyCounter(A, sizeof(A)/ sizeof(A[0]));
    return 0;
}
```

Array harus memiliki angka dalam rentang [1, n] (di mana n adalah ukuran array). Kondisi jika (A[pos] > 0 && A[expectedPos] > 0) berarti bahwa kedua angka pada indeks pos dan expectPos adalah angka aktual dalam array tetapi bukan frekuensinya. Jadi kita akan menukarnya sehingga angka pada pos indeks akan pergi ke posisi yang seharusnya jika angka 1, 2, 3, ..., n disimpan di 0, 1, 2, ... , n – 1 indeks. Dalam contoh array input di atas, awalnya pos = 0, jadi 10 pada indeks 0 akan masuk ke indeks 9 setelah swap. Karena ini adalah kemunculan pertama dari 10, buatlah menjadi -1. Perhatikan bahwa kita menyimpan frekuensi sebagai angka negatif untuk membedakan antara angka dan frekuensi sebenarnya.

Kondisi else if (A[pos] > 0) berarti A[pos] adalah bilangan dan A[expectedPos] adalah frekuensinya tanpa menyertakan kemunculan A[pos]. Jadi tingkatkan frekuensinya dengan 1 (yaitu pengurangan 1 dalam hal angka negatif). Saat kita menghitung kemunculannya, kita perlu pindah ke pos berikutnya, jadi pos + +, tetapi sebelum pindah ke posisi berikutnya kita harus membuat frekuensi angka pos + 1 yang sesuai dengan indeks pos nol, karena angka seperti itu belum terjadi. Bagian lain yang terakhir berarti pos indeks saat ini sudah memiliki frekuensi angka pos +1, jadi pindah ke pos berikutnya, maka pos ++.

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1).

Soal-80 Mana yang lebih cepat dan seberapa banyak, pencarian linier hanya 1000 elemen pada komputer 5 GHz atau pencarian biner 1 juta elemen pada komputer 1 GHz. Asumsikan bahwa eksekusi setiap instruksi pada komputer 5-GHz lima kali lebih cepat daripada pada komputer 1-GHz dan bahwa setiap

iterasi dari algoritma pencarian linier adalah dua kali lebih cepat dari setiap iterasi dari algoritma pencarian biner.

Solusi: Pencarian biner dari 1 juta elemen akan membutuhkan $log_2^{1,000,000}$ atau paling banyak sekitar 20 iterasi (yaitu, kasus terburuk). Pencarian linier 1000 elemen akan membutuhkan rata-rata 500 iretasi (yaitu, setengah jalan melalui array). Oleh karena itu, pencarian biner akan $\frac{500}{20} = 25$ lebih cepat (dalam hal iterasi) daripada pencarian linier. Namun, karena iterasi pencarian linier dua kali lebih cepat, pencarian biner akan menjadi $\frac{25}{2}$ atau sekitar 12 kali lebih cepat daripada pencarian linier secara keseluruhan, pada mesin yang sama. Karena kita menjalankannya pada mesin yang berbeda, di mana instruksi pada mesin 5-GhZ 5 kali lebih cepat daripada instruksi pada mesin 1-GHz, pencarian biner akan $\frac{12}{5}$ atau sekitar 2 kali lebih cepat daripada pencarian linier! Ide utamanya adalah bahwa peningkatan perangkat lunak dapat membuat algoritma berjalan lebih cepat tanpa harus menggunakan perangkat lunak yang lebih kuat.

BAB 4
ALGORITMA SELEKSI [MEDIAN]

4.1 APA ITU ALGORITMA SELEKSI?

Algoritma seleksi adalah algoritma untuk mencari bilangan terkecil/terbesar ke-k dalam suatu daftar (disebut juga statistik orde ke-k). Ini termasuk menemukan elemen minimum, maksimum, dan median. Untuk menemukan statistik orde ke-k, ada beberapa solusi yang memberikan kompleksitas yang berbeda, dan dalam bab ini kita akan menghitung kemungkinan-kemungkinan tersebut.

4.2 PEMILIHAN BERDASARKAN PENYORTIRAN

Masalah pemilihan dapat diubah menjadi masalah pengurutan. Dalam metode ini, pertama-tama kita mengurutkan elemen input dan kemudian mendapatkan elemen yang diinginkan. Efisien jika kita ingin melakukan banyak seleksi.

Sebagai contoh, katakanlah kita ingin mendapatkan elemen minimum. Setelah menyortir elemen input, kita cukup mengembalikan elemen pertama (dengan asumsi array diurutkan dalam urutan menaik). Sekarang, jika kita ingin menemukan elemen terkecil kedua, kita cukup mengembalikan elemen kedua dari daftar yang diurutkan.

Artinya, untuk elemen terkecil kedua kita tidak melakukan pengurutan lagi. Hal yang sama juga terjadi dengan pertanyaan berikutnya. Bahkan jika kita ingin mendapatkan elemen terkecil ke-k, hanya satu pemindaian dari daftar yang diurutkan sudah cukup untuk menemukan elemen (atau kita dapat mengembalikan nilai indeks ke-k jika elemen berada dalam array).

Dari pembahasan di atas yang dapat kita sampaikan adalah, dengan pengurutan awal kita dapat menjawab pertanyaan apa pun dalam satu pemindaian, O(n). Secara umum, metode ini membutuhkan waktu O(nlogn) (untuk pengurutan), di mana n adalah panjang daftar input. Misalkan kita melakukan n kueri, maka biaya rata-rata per operasi hanya $\frac{n\,logn}{n} \approx O(logn)$. Analisis semacam ini disebut analisis diamortisasi.

4.3 ALGORTIMA SELEKSI LINEAR – MEDIAN DARI ALGORITMA MEDIAN

Performa terburuk	O(n)
Performa terbaik	O(n)
Kompleksitas ruang kasus terburuk	O(1) bantu

4.4 MENEMUKAN K ELEMEN TERKECIL DALAM URUTAN TERURUT

Untuk algoritma, periksa Soal-6. Algoritma ini mirip dengan Quick sort.

4.5 ALGORITMA SELEKSI: MASALAH & SOLUSI

Soal-1 Temukan elemen terbesar dalam larik A berukuran n.

Solusi: Pindai array lengkap dan kembalikan elemen terbesar.

```
void FindLargestInArray(int n, const int A[]) {
    int large = A[0];
    for (int i = 1; i <= n-1; i++)
        if(A[i] > large)
            large = A[i];
    printf("Largest:%d", large);
}
```

Kompleksitas Waktu - O(n).

Kompleksitas Ruang - O(1).

Catatan: Setiap algoritma deterministik yang dapat menemukan n kunci terbesar dengan membandingkan kunci membutuhkan setidaknya n -1 perbandingan.

Soal-2 Temukan elemen terkecil dan terbesar dalam array A berukuran n.

Solusi:

```
void FindSmallestAndLargestInArray (int A[], int n) {
    int small = A[0];
    int large =A[0];
    for (int i = 1; i <= n-1; i++)
        if(A[i] < small)
            small = A[i];
        else if(A[i] > large)
            large = A[i];
    printf("Smallest:%d, Largest:%d", small, large);
}
```

Kompleksitas Waktu - O(n).

Kompleksitas Ruang - O(1).

Jumlah kasus terburuk dari perbandingan adalah 2(n – 1).

Soal-3 Bisakah kita meningkatkan algoritma sebelumnya?

Solusi: Ya. Kita dapat melakukan ini dengan membandingkan berpasangan.

```
// n is assumed to be even. Compare in pairs.
void FindWithPairComparison (int A[], int n) {
    int large = small = -1;
    for (int i = 0; i <= n - 1; i = i + 2) {          // Increment i by 2.
        if(A[i] < A[i + 1]) {
            if(A[i] < small)
                small = A[i];
            if(A[i + 1] > large)
                large = A[i + 1];
        }
```

```
                else {
                        if(A[i + 1] < small)
                                small = A[i + 1];
                          if(A[i] > large)
                                large = A[i];
                }
        }
        printf("Smallest:%d, Largest:%d", small, large);
    }
```

Kompleksitas Waktu - O(n).
Kompleksitas Ruang - O(1).

Jumlah perbandingan:

$$\begin{cases} \frac{3n}{2} - 2, if\ n\ is\ even \\ \frac{3n}{2} - \frac{3}{2}\ if\ n\ is\ odd \end{cases}$$

Ringkasan:

Perbandingan langsung – 2(n – 1) perbandingan
Bandingkan untuk min hanya jika perbandingan untuk maks gagal
Kasus terbaik: urutan meningkat – n – 1 perbandingan
Kasus terburuk: urutan menurun – 2(n – 1) perbandingan
Kasus rata-rata: 3n/2 – 1 perbandingan

Catatan *Untuk teknik Divide and conquer, lihat bab Divide and conquer.*

Soal-4 Berikan algoritma untuk menemukan elemen terbesar kedua dalam daftar input elemen yang diberikan.

Solusi: Metode Brute Force

Algoritma:

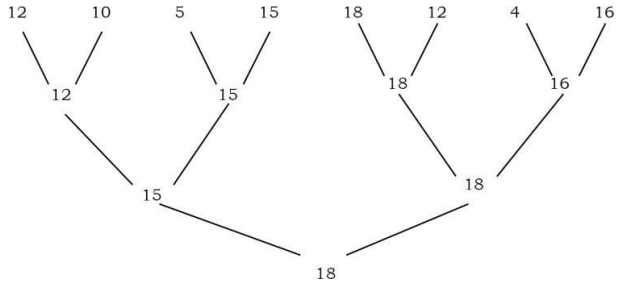
- Temukan elemen terbesar: membutuhkan perbandingan n – 1
- Hapus (buang) elemen terbesar
- Sekali lagi temukan elemen terbesar: kebutuhan n – 2 perbandingan

Jumlah total perbandingan: n – 1 + n – 2 = 2n – 3

Soal-5 Bisakah kita mengurangi jumlah perbandingan dalam solusi Soal-4?

Solusi: Metode Turnamen: Untuk mempermudah, asumsikan bahwa angka-angkanya berbeda dan n adalah pangkat 2. Kita memasangkan kunci dan membandingkan pasangan dalam putaran hingga hanya tersisa satu putaran. Jika input memiliki delapan kunci, ada empat perbandingan di putaran pertama, dua di putaran kedua, dan satu di putaran terakhir. Pemenang putaran terakhir adalah kunci terbesar. Gambar di bawah ini menunjukkan metodenya.

Metode turnamen secara langsung hanya berlaku jika n adalah pangkat 2. Jika tidak demikian, kita dapat menambahkan item yang cukup ke akhir larik untuk membuat ukuran larik pangkat 2. Jika pohon selesai maka maksimum tinggi pohon dicatat. Jika kita membangun pohon biner lengkap, kita membutuhkan n – 1 perbandingan untuk menemukan yang terbesar. Kunci terbesar kedua harus berada di antara yang hilang dibandingkan dengan yang terbesar. Artinya, elemen terbesar kedua harus menjadi salah satu lawan dari elemen terbesar. Jumlah kunci yang hilang hingga kunci terbesar adalah tinggi pohon, yaitu logn [jika pohon adalah pohon biner lengkap]. Kemudian menggunakan algoritma seleksi untuk menemukan yang terbesar di antara mereka, ambil logn – 1 perbandingan. Jadi jumlah total perbandingan untuk menemukan kunci terbesar dan terbesar kedua adalah n + logn – 2.



Soal-6 Temukan k-elemen terkecil dalam array S dari n elemen menggunakan metode partisi.

Solusi: Pendekatan Brute Force: Pindai angka sebanyak k kali untuk mendapatkan elemen yang diinginkan. Metode ini adalah yang digunakan dalam bubble sort (dan selection sort), setiap kali kita menemukan elemen terkecil di seluruh urutan dengan membandingkan setiap elemen. Dalam metode ini, urutan harus dilalui k kali. Jadi kompleksitasnya adalah O(n × k).

Soal-7 Bisakah kita menggunakan teknik pengurutan untuk menyelesaikan Soal-6?

Solusi: Ya. Urutkan dan ambil k elemen pertama.

1. Urutkan angka.
2. Pilih k elemen pertama.

Perhitungan kompleksitas waktu sepele. Pengurutan n angka adalah dari O(nlogn) dan memilih k elemen adalah dari O(k). Kompleksitas totalnya adalah O(nlogn + k) = O(nlogn).

Soal-8 Bisakah kita menggunakan teknik pengurutan pohon untuk menyelesaikan Soal-6?

Solusi: Ya.

1. Masukkan semua elemen dalam pohon pencarian biner.
2. Lakukan traversal InOrder dan cetak k elemen yang akan menjadi elemen terkecil. Jadi, kita memiliki k elemen terkecil.

Biaya pembuatan pohon pencarian biner dengan n elemen adalah O(nlogn) dan traversal hingga k elemennya adalah O(k). Oleh karena itu kompleksitasnya adalah O(nlogn + k) = O(nlogn).

Kerugian: Jika nomor diurutkan dalam urutan menurun, kita akan mendapatkan pohon yang miring ke kiri. Dalam hal ini, konstruksi pohon akan menjadi 0 + l + 2 + ... + (n– 1) yaitu O(n2). Untuk menghindarinya, kita bisa menjaga keseimbangan pohon, sehingga biaya pembuatan pohon hanya nlogn.

Soal-9 Bisakah kita meningkatkan teknik pengurutan pohon untuk menyelesaikan Soal-6?

Solusi: Ya. Gunakan pohon yang lebih kecil untuk memberikan hasil yang sama.

1. Ambil k elemen pertama dari urutan untuk membuat pohon yang seimbang dari k node (ini akan dikenakan biaya klogk).
2. Ambil angka yang tersisa satu per satu, dan
   A. Jika jumlahnya lebih besar dari elemen terbesar dari pohon, kembali.
   B. Jika jumlahnya lebih kecil dari elemen terbesar dari pohon, hapus elemen terbesar dari pohon dan tambahkan elemen baru. Langkah ini untuk memastikan bahwa elemen yang lebih kecil menggantikan elemen yang lebih besar dari pohon. Dan tentu saja biaya operasi ini adalah logk karena pohon tersebut merupakan pohon seimbang dari k elemen.

Setelah Langkah 2 selesai, pohon seimbang dengan k elemen akan memiliki k elemen terkecil. Satu-satunya tugas yang tersisa adalah mencetak elemen terbesar dari pohon.

Kompleksitas Waktu:

1. Untuk k elemen pertama, kita buat pohonnya. Makanya biayanya klok.
2. Untuk n – k elemen lainnya, kompleksitasnya adalah O(logk).

Langkah 2 memiliki kompleksitas (n – k) logk. Total biayanya adalah klogk + (n – k) logk = nlogk yaitu O(nlogk). Ikatan ini sebenarnya lebih baik daripada yang disediakan sebelumnya.

Soal-10 Bisakah kita menggunakan teknik partisi untuk menyelesaikan Soal-6?

Solusi: Ya. algoritma

1. Pilih pivot dari larik.

2. Partisi array sehingga: A[low...pivotpoint – 1] <= pivotpoint <= A[pivotpoint + 1..tinggi].
3. jika k < pivotpoint maka harus berada di sebelah kiri pivot, maka lakukan cara yang sama secara rekursif pada bagian kiri.
4. jika k = pivotpoint maka harus pivot dan cetak semua elemen dari rendah ke titik sumbu.
5. jika k > pivotpoint maka harus di sebelah kanan pivot, jadi lakukan cara yang sama secara rekursif di bagian kanan.

Panggilan tingkat atas adalah kthSmallest = Selection(1, n, k).

```
int Selection (int low, int high, int k) {
    int pivotpoint;
    if(low == high)
        return S[low];
    else {
        pivotpoint = Partition(low, high);
        if(k == pivotpoint)
            return S[pivotpoint];  //we can print all the elements from low to pivotpoint.
        else if(k < pivotpoint)
            return Selection (low, pivotpoint - 1, k);
        else return Selection (pivotpoint + 1, high, k);
    }
}
void Partition (int low, int high) {
    int i, j, pivotitem;
    pivotitem = S[low];
    j = low;
    for (i = low + 1; i <= high; i++)
        if(S[i] < pivotitem) {
            j++;
            Swap S[i] and S[j];
        }
    pivotpoint = j;
    Swap S[low] and S[pivotpoint];
    return pivotpoint;
}
```

Kompleksitas Waktu: $O(n^2)$ dalam kasus terburuk mirip dengan Quicksort. Meskipun kasus terburuknya sama dengan Quicksort, kinerjanya jauh lebih baik pada rata-rata [O(nlogk) – Rata-rata kasus].

Soal-11 Temukan elemen terkecil ke-k dalam array S dari n elemen dengan cara terbaik.

Solusi: Masalah ini mirip dengan Masalah-6 dan semua solusi yang dibahas untuk Masalah-6 berlaku untuk masalah ini. Satu-satunya perbedaan adalah bahwa alih-alih mencetak semua elemen k, kita hanya mencetak elemen ke-k. Kita dapat meningkatkan solusi dengan menggunakan algoritma median median. Median adalah kasus khusus dari algoritma seleksi. Algoritma Pemilihan(A,k)

untuk mencari elemen terkecil ke-k dari himpunan A dari n elemen adalah sebagai berikut:

Algoritma Seleksi (A, k)

1. Bagilah A ke dalam $ceil\left(\frac{length(A)}{5}\right)$kelompok-kelompok, dengan masing-masing kelompok memiliki lima item (kelompok terakhir mungkin memiliki lebih sedikit item).
2. Urutkan setiap grup secara terpisah (mis., Insertion sort).
3. Temukan median dari masing-masing $\frac{n}{5}$ grup dan simpan dalam beberapa larik (misalkan A').
4. Gunakan Seleksi secara rekursif untuk mencari median dari A' (median median). Misalkan median median adalah m.

$$m = Selection(A', \frac{\frac{length(A)}{5}}{2});$$

5. Misalkan q = # elemen A lebih kecil dari m;
6. Jika (k == q + 1)

return m;
/* Partition with pivot */

7. Partisi lain A menjadi X dan Y

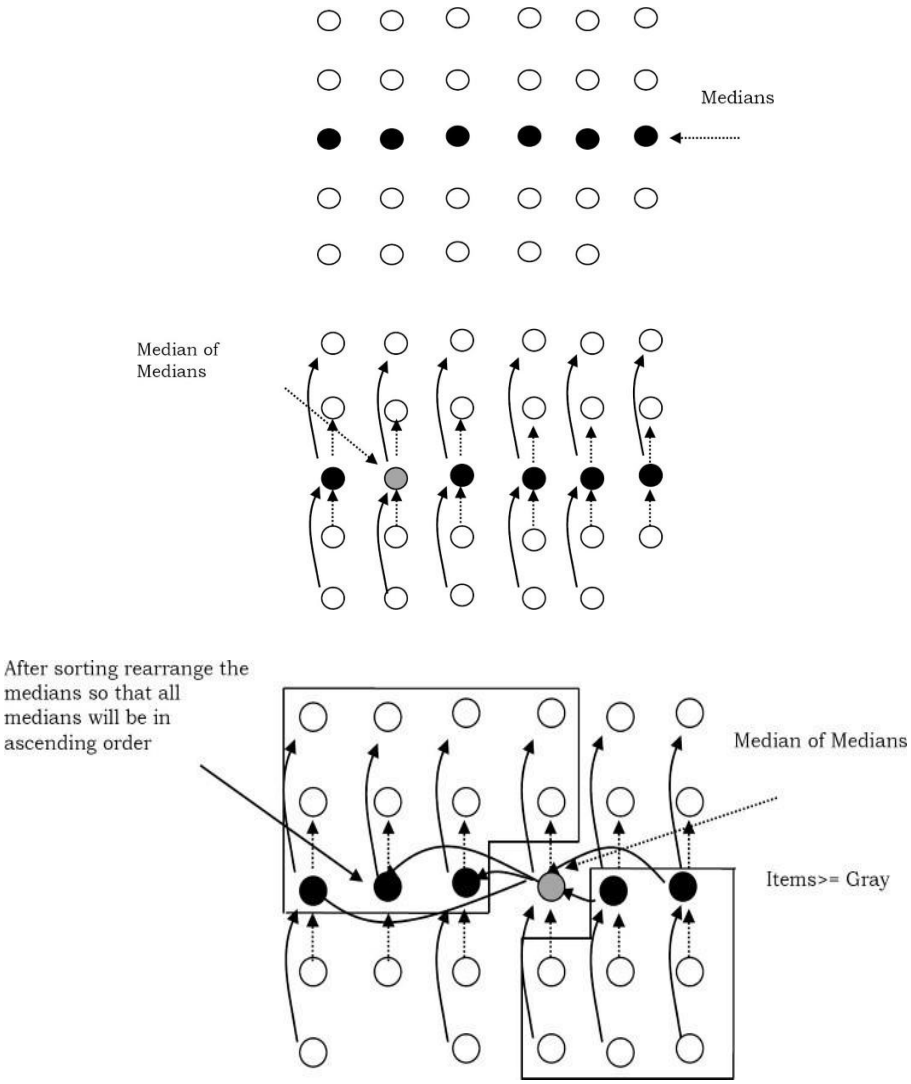
- X = {item lebih kecil dari m)
- Y = {item lebih besar dari m}

/* Next,form a subproblem */

8. Jika (k < q + 1)
9. Lainnya

return Selection(X, k);
return Selection(Y, k – (q+1));

Sebelum mengembangkan kekambuhan, mari kita pertimbangkan representasi input di bawah ini. Pada gambar, setiap lingkaran adalah elemen dan setiap kolom dikelompokkan dengan 5 elemen. Lingkaran hitam menunjukkan median di setiap kelompok 5 elemen. Seperti yang telah dibahas, urutkan setiap kolom menggunakan pengurutan penyisipan waktu konstan.



Pada gambar di atas item yang dilingkari abu-abu adalah median dari median (sebut saja ini m). Dapat dilihat bahwa paling sedikit 1/2 dari 5 median kelompok elemen m. Juga, 1/2 dari 5 kelompok elemen ini menyumbangkan 3 elemen yang m kecuali 2 grup [grup terakhir yang mungkin berisi kurang dari 5 elemen, dan grup lain yang berisi m]. Demikian pula, setidaknya 1/2 dari 5 kelompok elemen menyumbangkan 3 elemen yang m seperti yang ditunjukkan di atas. 1/2 dari 5 kelompok elemen menyumbang 3 elemen, kecuali 2 kelompok memberikan:

$$3(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2) \approx \frac{3n}{10} - 6$$

Sisanya adalah $n - \frac{3n}{10} - 6 \approx \frac{7n}{10} + 6$

$$\frac{7n}{10} + 6 \qquad \frac{3n}{10} - 6$$

Sejak lebih besar dari perlu kita pertimbangkan untuk yang terburuk.

Komponen dalam pengulangan:

- Dalam Algoritma pemilihan kita, kita memilih m, yang merupakan median median, untuk menjadi pivot, dan mempartisi A menjadi dua himpunan X dan Y. Kita perlu memilih himpunan yang memberikan ukuran maksimum (untuk mendapatkan kasus terburuk).
- Waktu dalam pemilihan fungsi saat dipanggil dari partisi prosedur. Jumlah tombol dalam input untuk panggilan ke Seleksi ini adalah $\frac{n}{5}$.
- Jumlah perbandingan yang diperlukan untuk mempartisi array. Angka ini adalah panjang (S), mari kita katakan n.

Kita telah menetapkan pengulangan berikut:

$$T(n) = T\left(\frac{n}{5}\right) + \Theta(n) + Max\{T(X), T(Y)\}$$

Dari pembahasan di atas kita telah melihat bahwa, jika kita memilih median dari median m sebagai pivot, ukuran partisinya adalah: $\frac{3n}{10} - 6$ dan $\frac{7n}{10} + 6$.

Soal-12 Pada Soal-11, kita membagi array input menjadi kelompok-kelompok yang terdiri dari 5 elemen. Konstanta 5 memainkan peran penting dalam analisis. Bisakah kita membagi dalam kelompok 3 yang bekerja dalam waktu linier?

$$\begin{aligned} T(n) &= T\left(\tfrac{n}{5}\right) + \Theta(n) + T\left(\tfrac{7n}{10} + 6\right) \\ &\approx T\left(\tfrac{n}{5}\right) + \Theta(n) + T\left(\tfrac{7n}{10}\right) + O(1) \\ &\leq c\tfrac{7n}{10} + c\tfrac{n}{5} + \Theta(n) + O(1) \\ &\text{Finally, } T(n) = \Theta(n). \end{aligned}$$

Solusi: Dalam hal ini modifikasi menyebabkan rutinitas memakan waktu lebih dari waktu linier. Dalam kasus terburuk, setidaknya setengah dari $\lceil \frac{n}{3} \rceil$ median yang ditemukan pada langkah pengelompokan lebih besar dari median median m, tetapi dua dari grup tersebut berkontribusi kurang dari dua elemen yang lebih besar dari m. Jadi sebagai batas atas, jumlah elemen yang lebih besar dari titik pivot setidaknya:

Demikian juga ini adalah batas bawah. Jadi hingga $n - (\frac{n}{3} - 4) = \frac{2n}{3} + 4$ elemen dimasukkan ke dalam panggilan rekursif ke Select. Langkah rekursif yang menemukan median median berjalan pada masalah ukuran $\lceil \frac{n}{3} \rceil$, dan akibatnya pengulangan waktu adalah:

$$T(n) = T(\lceil \tfrac{n}{3} \rceil) + T(2n/3 + 4) + \Theta(n).$$

$$2(\lceil \tfrac{1}{2} \lceil \tfrac{n}{3} \rceil \rceil - 2) \geq \frac{n}{3} - 4$$

Dengan asumsi bahwa T(n) meningkat secara monoton, kita dapat menyimpulkan bahwa

$T(\frac{2n}{3} + 4) \geq T(\frac{2n}{3}) \geq 2T(\frac{n}{3})$
$T(n) \geq 3T(\frac{n}{3}) + \Theta(n)$

, dan kita dapat mengatakan batas atas untuk ini sebagai , yaitu O(nlogn). Oleh karena itu, kita tidak dapat memilih 3 sebagai grup ukuran.

Soal-13 Seperti pada Soal-12, dapatkah kita menggunakan kelompok ukuran 7?

Solusi: Mengikuti alasan yang sama, kita sekali lagi memodifikasi rutinitas, sekarang menggunakan grup 7 daripada 5. Dalam kasus terburuk, setidaknya setengah $\lceil \frac{n}{7} \rceil$median yang ditemukan pada langkah pengelompokan lebih besar dari median median m, tetapi dua dari kelompok-kelompok tersebut menyumbang kurang dari empat elemen yang lebih besar dari m. Jadi sebagai batas atas, jumlah elemen yang lebih besar dari titik pivot setidaknya:

$$4(\lceil 1/2 \lceil n/7 \rceil \rceil\text{-}2) \geq \frac{2n}{7} - 8.$$

Demikian juga ini adalah batas bawah. Jadi hingga $n - (\frac{2n}{7} - 8) = \frac{5n}{7} + 8$ elemen dimasukkan ke dalam panggilan rekursif ke Select. Langkah rekursif yang menemukan median median berjalan pada masalah ukuran $\lceil \frac{n}{7} \rceil$, dan akibatnya pengulangan waktu adalah

$$T(n) = T(\lceil \tfrac{n}{7} \rceil) + T(\tfrac{5n}{7} + 8) + O(n)$$
$$T(n) \leq c\lceil \tfrac{n}{7} \rceil + c(\tfrac{5n}{7} + 8) + O(n)$$
$$\leq c\frac{n}{7} + c\frac{5n}{7} + 8c + an, a \text{ is a constant}$$
$$= cn - c\frac{n}{7} + an + 9c$$
$$= (a + c)n - (c\frac{n}{7} - 9c).$$

Ini dibatasi di atas oleh (a + c) n asalkan $c\frac{n}{7} - 9c \geq 0$. Oleh karena itu, kita dapat memilih 7 sebagai ukuran grup.

Soal-14 Diberikan dua larik yang masing-masing berisi n elemen terurut, berikan algoritma waktu-O(logn) untuk mencari median dari semua elemen 2n.

Solusi: Solusi sederhana untuk masalah ini adalah menggabungkan dua daftar dan kemudian mengambil rata-rata dari dua elemen tengah (perhatikan gabungan selalu berisi jumlah nilai genap). Namun, penggabungannya adalah (n), sehingga tidak memenuhi rumusan masalah. Untuk mendapatkan kompleksitas logn, biarkan medianA dan medianB menjadi median dari daftar masing-masing

(yang dapat dengan mudah ditemukan karena kedua daftar diurutkan). Jika medianA == medianB, maka itu adalah median keseluruhan dari gabungan dan kita selesai. Jika tidak, median gabungan harus berada di antara medianA dan medianB. Misalkan medianA < medianB (kasus sebaliknya sepenuhnya serupa). Kemudian kita perlu mencari median dari gabungan dua himpunan berikut:

$$\{x \; in \; A \mid x >= medianA\} \{x \; in \; B \mid x <= medianB\}$$

Jadi, kita dapat melakukan ini secara rekursif dengan mengatur ulang batas kedua array. Algoritma melacak kedua array (yang diurutkan) menggunakan dua indeks. Indeks ini digunakan untuk mengakses dan membandingkan median dari kedua array untuk menemukan di mana letak median keseluruhan.

```
void FindMedian(int A[], int alo , int ahi, int B[], int blo int bhi) {
    amid = alo + (ahi-alo)/2;
    amed = a[amid];
    bmid = blo + (bhi-blo)/2;
    bmed = b[bmid];
    if( ahi - alo + bhi - blo < 4) {
            Handle the boundary cases and solve it smaller problem in O(1) time.
            return;
    }
    else if(amed < bmed)
            FindMedian(A, amid, ahi, B, blo, bmid+1);
    else    FindMedian(A, alo, amid+1,B, bmid+1, bhi);
}
```

Kompleksitas Waktu: O(logn), karena kita mengurangi ukuran masalah hingga setengahnya setiap kali.

Soal-15 Misalkan A dan B adalah dua larik terurut yang masing-masing terdiri dari n elemen. Kita dapat dengan mudah menemukan elemen terkecil ke-k di A dalam waktu O(1) hanya dengan mengeluarkan A[k]. Demikian pula, kita dapat dengan mudah menemukan elemen terkecil ke-k di B. Berikan algoritma waktu O(logk) untuk menemukan elemen terkecil ke-k secara keseluruhan {yaitu, yang terkecil ke-k dalam gabungan A dan B.

Solusi Ini hanyalah cara lain untuk menanyakan Masalah-14.

Soal-16 Menemukan k elemen terkecil dalam urutan terurut: Diberikan himpunan n elemen dari domain yang terurut total, temukan k elemen terkecil, dan daftarkan dalam urutan terurut. Analisis waktu berjalan kasus terburuk dari implementasi terbaik dari pendekatan.

Solusi: Urutkan angka-angka, dan daftarkan k terkecil.

T(n) = Kompleksitas waktu pengurutan + daftar k elemen terkecil = Θ (nlogn) + Θ (n) = (nlogn).

Soal-17 Untuk Soal-16, jika kita mengikuti pendekatan di bawah ini, lalu apa kerumitannya?

Solusi: Menggunakan struktur data antrian prioritas dari heap sort, buat min-heap di atas set, dan lakukan ekstrak-min k kali. Lihat bab Antrian Prioritas (Tumpukan) untuk lebih jelasnya.

Soal-18 Untuk Soal-16, jika kita mengikuti pendekatan di bawah ini lalu apa kerumitannya?

Temukan elemen terkecil ke-k dari himpunan, partisi di sekitar elemen pivot ini, dan urutkan k elemen terkecil.

Solusi T (n) = Kompleksitas waktu ke-k – terkecil + Menemukan pivot + Awalan pengurutan

$$= \Theta(n) + \Theta(n) + \Theta(k \log k) = \Theta(n + k \log k)$$

Karena, k n, pendekatan ini lebih baik daripada Soal-16 dan Soal-17.

Soal-19 Temukan k tetangga terdekat ke median dari n bilangan berbeda dalam waktu O(n).

Solusi: Mari kita asumsikan bahwa elemen array diurutkan. Sekarang temukan median dari n angka dan panggil indeksnya sebagai X (karena array diurutkan, median akan berada di $\frac{n}{2}$ lokasi). Yang perlu kita lakukan adalah memilih k elemen dengan perbedaan absolut terkecil dari median, bergerak dari X – 1 ke 0, dan X + 1 ke n – 1 ketika median berada pada indeks m.

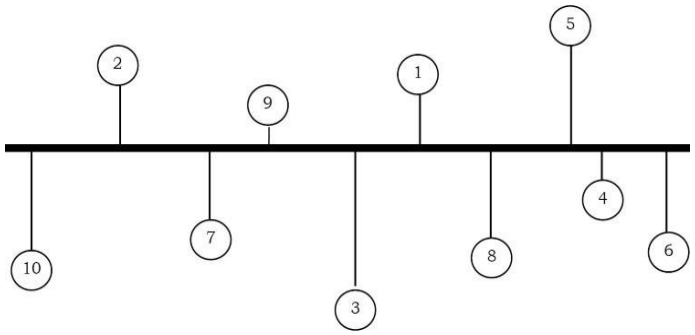
Kompleksitas Waktu: Setiap langkah membutuhkan (n). Jadi total kompleksitas waktu dari algoritma adalah (n).

Soal-20 Apakah ada cara lain untuk menyelesaikan Soal-19?

Solusi: Asumsikan untuk kesederhanaan bahwa n ganjil dan k genap. Jika himpunan A diurutkan, median berada pada posisi n/2 dan bilangan k pada A yang paling dekat dengan median berada pada posisi (n – k)/2 sampai (n + k)/2.

Kita pertama-tama menggunakan pemilihan waktu linier untuk menemukan elemen (n – k)/2, n/2, dan (n + k)/2 dan kemudian melewati himpunan A untuk menemukan bilangan yang lebih kecil dari (n + k)/2 elemen, lebih besar dari elemen (n – k)/2, dan tidak sama dengan elemen n/2. Algoritma membutuhkan waktu O(n) karena kita menggunakan pemilihan waktu linier tepat tiga kali dan melintasi n angka di A satu kali.

Soal-21 Diberikan koordinat (x,y) dari n rumah, di mana Anda harus membangun jalan yang sejajar dengan sumbu x untuk meminimalkan biaya konstruksi bangunan jalan masuk?



Solusi: Pembangunan jalan tidak memerlukan biaya apa pun. Ini adalah jalan masuk yang membutuhkan uang. Biaya jalan masuk sebanding dengan jaraknya dari jalan. Jelas, mereka akan tegak lurus. Solusinya adalah dengan menempatkan jalan pada median koordinat y.

Soal-22 Diberikan sebuah file besar yang berisi milyaran angka, temukan maksimal 10 angka dari file tersebut.

Solusi Lihat bab Antrian Prioritas.

Soal-23 Misalkan ada perusahaan susu. Perusahaan mengumpulkan susu setiap hari dari semua agennya. Agen terletak di tempat yang berbeda. Untuk mengumpulkan susu, di mana tempat terbaik untuk memulai sehingga jarak total yang ditempuh paling sedikit?

Solusi: Memulai dari median mengurangi total jarak yang ditempuh karena itu adalah tempat yang merupakan pusat dari semua tempat.

BAB 5
SIMBOL TABEL

5.1 PENDAHULUAN

Sejak kecil, kita semua telah menggunakan kamus, dan banyak dari kita memiliki pengolah kata (misalnya, Microsoft Word) yang dilengkapi dengan pemeriksa ejaan. Pemeriksa ejaan juga merupakan kamus tetapi cakupannya terbatas. Ada banyak contoh real time untuk kamus dan beberapa di antaranya adalah:

- Pemeriksa ejaan
- Kamus data yang terdapat dalam aplikasi manajemen basis data
- Tabel simbol yang dihasilkan oleh loader, assembler, dan compiler
- Tabel perutean dalam komponen jaringan (pencarian DNS)

Dalam ilmu komputer, kita umumnya menggunakan istilah 'tabel simbol' daripada 'kamus' ketika mengacu pada tipe data abstrak (ADT).

5.2 APA ITU TABEL SIMBOL?

Kita dapat mendefinisikan tabel simbol sebagai struktur data yang mengaitkan nilai dengan kunci. Ini mendukung operasi berikut:

- Cari apakah nama tertentu ada di tabel
- Dapatkan atribut dari nama itu
- Ubah atribut nama itu
- Masukkan nama baru dan atributnya
- Hapus nama dan atributnya

Hanya ada tiga operasi dasar pada tabel simbol: mencari, menyisipkan, dan menghapus.

Contoh: Pencarian DNS. Mari kita asumsikan bahwa kunci dalam kasus ini adalah URL dan nilainya adalah alamat IP.

- Masukkan URL dengan alamat IP yang ditentukan
- URL yang diberikan, temukan alamat IP yang sesuai

Tabel 5.1 *website* dan *IP address*

Key[Website]	Value [IP Address]
www.CareerMonks.com	128.112.136.11
www.AuthorsInn.com	128.112.128.15
www.AuthInn.com	130.132.143.21
www.klm.com	128.103.060.55
www.CareerMonk.com	209.052.165.60

5.3 IMPLEMENTASI TABEL SIMBOL

Sebelum menerapkan tabel simbol, mari kita menghitung kemungkinan implementasi. Tabel simbol dapat diimplementasikan dalam banyak cara dan beberapa di antaranya tercantum di bawah ini.

Implementasi Array Tidak Terurut

Dengan metode ini, hanya mempertahankan array sudah cukup. Dibutuhkan O(n) waktu untuk pencarian, penyisipan dan penghapusan dalam kasus terburuk.

Memerintahkan Implementasi Array [Diurutkan]

Dalam hal ini kita mempertahankan array kunci dan nilai yang diurutkan.

- Simpan dalam urutan yang diurutkan berdasarkan kunci
- kunci[i] = kunci terbesar ke-i
- values[i] = nilai yang terkait dengan kunci terbesar ke-i

Karena elemen diurutkan dan disimpan dalam array, kita dapat menggunakan pencarian biner sederhana untuk menemukan elemen. Dibutuhkan O(logn) waktu untuk pencarian dan O(n) waktu untuk penyisipan dan penghapusan dalam kasus terburuk.

5.4 TABEL PERBANDINGAN SIMBOL UNTUK IMPLEMENTASI

Mari kita perhatikan tabel perbandingan berikut untuk semua implementasi.

Tabel 5.2 perbandingan symbol untuk implementasi

Penerapan	Pencarian	Memasukkan	Menghapus
Array Tidak Terurut	n	n	n
Ordered Array (dapat diimplementasikan dengan pencarian biner array)	logn		n
Daftar Tidak Terurut	n	n	n
Daftar pesanan	n	n	n
Pohon Pencarian Biner (O(logn) rata-rata	logn	logn	logn
Pohon Pencarian Biner Seimbang (O(logn) dalam kasus terburuk)	logn	logn	logn
Pencarian Ternary (hanya perubahan dalam basis logaritma)	logn	logn	logn
Hashing (O(1) rata-rata)	1	1	1

Catatan:

- Pada tabel di atas, n adalah ukuran input.
- Tabel menunjukkan kemungkinan implementasi yang dibahas dalam buku ini. Tapi, mungkin ada implementasi lain.

BAB 6
HASHING

6.1 APA ITU HASHING?

Hashing adalah teknik yang digunakan untuk menyimpan dan mengambil informasi secepat mungkin. Ini digunakan untuk melakukan pencarian yang optimal dan berguna dalam mengimplementasikan tabel simbol.

6.2 MENGAPA HASHING?

Dalam bab Pohon kita melihat bahwa pohon pencarian biner seimbang mendukung operasi seperti menyisipkan, menghapus dan mencari dalam waktu O(logn). Dalam aplikasi, jika kita membutuhkan operasi ini di O(1), maka hashing menyediakan caranya. Ingat bahwa kompleksitas hashing kasus terburuk masih O(n), tetapi memberikan O(1) rata-rata.

6.3 HASHTABLE ADT

Operasi umum untuk tabel hash adalah:

- CreatHashTable: Membuat tabel hash baru
- HashSearch: Mencari kunci di tabel hash
- HashInsert: Menyisipkan kunci baru ke dalam tabel hash
- HashDelete: Menghapus kunci dari tabel hash
- DeleteHashTable: Menghapus tabel hash

6.4 MEMAHAMI HASHING

Secara sederhana kita dapat memperlakukan array sebagai tabel hash. Untuk memahami penggunaan tabel hash, mari kita perhatikan contoh berikut: Berikan algoritma untuk mencetak karakter pertama yang diulang jika ada elemen duplikat di dalamnya. Mari kita pikirkan solusi yang mungkin. Cara penyelesaian yang sederhana dan kasar adalah: diberi string, untuk setiap karakter periksa apakah karakter itu diulang atau tidak. Kompleksitas waktu dari pendekatan ini adalah O(n2) dengan kompleksitas ruang O(1).

Sekarang, mari kita cari solusi yang lebih baik untuk masalah ini. Karena tujuan kita adalah menemukan karakter pertama yang berulang, bagaimana jika kita mengingat karakter sebelumnya dalam beberapa array?

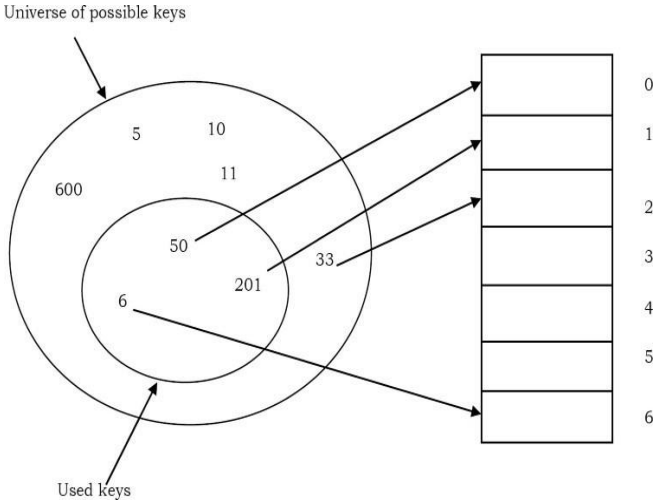
Kita tahu bahwa jumlah karakter yang mungkin adalah 256 (untuk kesederhanaan asumsikan karakter ASCII saja). Buat array ukuran 256 dan inisialisasi dengan semua nol. Untuk setiap karakter input, pergi ke posisi yang sesuai dan tingkatkan hitungannya. Karena kita menggunakan array, dibutuhkan waktu yang konstan untuk mencapai lokasi mana pun. Saat

memindai input, jika kita mendapatkan karakter yang penghitungnya sudah 1 maka kita dapat mengatakan bahwa karakter tersebut adalah karakter yang berulang untuk pertama kalinya.

```
char FirstRepeatedChar ( char *str ) {
    int i, len=strlen(str);
    int count[256]; //additional array
    for(i=0; i<256; ++i)
        count[i] = 0;
    for(i=0; i<len; ++i) {
        if(count[str[i]]==1) {
            printf("%c", str[i]);
            break;
        }
        else    count[str[i]]++;
    }
    if(i==len)
        printf("No Repeated Characters");
    return 0;
}
```

Mengapa tidak Array?

Pada soal sebelumnya, kita telah menggunakan array berukuran 256 karena kita telah mengetahui jumlah kemungkinan karakter yang berbeda [256] sebelumnya. Sekarang, mari kita pertimbangkan sedikit varian dari masalah yang sama. Misalkan array yang diberikan memiliki angka, bukan karakter, lalu bagaimana kita menyelesaikan masalah



Gambar 6.1 array dengan angka bukan karakter

Dalam hal ini himpunan nilai yang mungkin adalah tak terhingga (atau setidaknya sangat besar). Membuat array besar dan menyimpan counter tidak mungkin. Itu berarti ada satu set kunci universal dan lokasi terbatas di memori. Jika kita ingin memecahkan masalah ini, kita perlu memetakan semua kemungkinan kunci ini ke lokasi memori yang mungkin. Dari pembahasan dan diagram di atas terlihat bahwa kita membutuhkan pemetaan kemungkinan kunci ke salah satu lokasi yang tersedia. Akibatnya menggunakan array sederhana bukanlah pilihan yang tepat untuk memecahkan masalah di mana kemungkinan kuncinya sangat besar. Proses pemetaan kunci ke lokasi disebut hashing.

Catatan: *Untuk saat ini, jangan khawatir tentang bagaimana kunci dipetakan ke lokasi. Itu tergantung pada fungsi yang digunakan untuk konversi. Salah satu fungsi sederhana tersebut adalah kunci % ukuran tabel.*

6.5 KOMPONEN HASHING

Hashing memiliki empat komponen utama:

1) Tabel Hash
2) Fungsi Hash
3) Tabrakan
4) Teknik Resolusi Tabrakan

6.6 TABEL HASH

Tabel hash adalah generalisasi dari array. Dengan sebuah array, kita menyimpan elemen yang kuncinya adalah k pada posisi k dari array. Artinya, dengan diberikan kunci k, kita menemukan elemen yang kuncinya adalah k hanya dengan melihat posisi ke-k dari array. Ini disebut pengalamatan langsung.

Pengalamatan langsung dapat diterapkan ketika kita mampu mengalokasikan array dengan satu posisi untuk setiap kunci yang mungkin. Tetapi jika kita tidak memiliki cukup ruang untuk mengalokasikan lokasi untuk setiap kemungkinan kunci, maka kita memerlukan mekanisme untuk menangani kasus ini. Cara lain untuk mendefinisikan skenario adalah: jika kita memiliki lebih sedikit lokasi dan lebih banyak kemungkinan kunci, maka implementasi array sederhana tidak cukup.

Dalam kasus ini, satu opsi adalah menggunakan tabel hash. Tabel hash atau peta hash adalah struktur data yang menyimpan kunci dan nilai terkaitnya, dan tabel hash menggunakan fungsi hash untuk memetakan kunci ke nilai terkaitnya. Konvensi umum adalah bahwa kita menggunakan tabel hash ketika jumlah kunci yang sebenarnya disimpan relatif kecil dibandingkan dengan jumlah kunci yang mungkin.

6.7 FUNGSI HASH

Fungsi hash digunakan untuk mengubah kunci menjadi indeks. Idealnya, fungsi hash harus memetakan setiap kunci yang mungkin ke indeks slot yang unik, tetapi sulit untuk dicapai dalam praktiknya.

Diberikan kumpulan elemen, fungsi hash yang memetakan setiap item ke dalam slot unik disebut sebagai fungsi hash sempurna. Jika kita tahu elemen dan koleksinya tidak akan pernah berubah, maka dimungkinkan untuk membangun fungsi hash yang sempurna. Sayangnya, mengingat kumpulan elemen yang berubah-ubah, tidak ada cara sistematis untuk

membangun fungsi hash yang sempurna. Untungnya, kita tidak memerlukan fungsi hash yang sempurna untuk tetap mendapatkan efisiensi kinerja.

Salah satu cara untuk selalu memiliki fungsi hash yang sempurna adalah dengan memperbesar ukuran tabel hash sehingga setiap nilai yang mungkin dalam rentang elemen dapat diakomodasi. Ini menjamin bahwa setiap elemen akan memiliki slot unik. Meskipun ini praktis untuk sejumlah kecil elemen, itu tidak layak ketika jumlah elemen yang mungkin besar. Misalnya, jika elemennya adalah sembilan digit nomor Jaminan Sosial, metode ini akan membutuhkan hampir satu miliar slot. Jika kita hanya ingin menyimpan data untuk kelas yang terdiri dari 25 siswa, kita akan membuang banyak memori.

Tujuan kita adalah membuat fungsi hash yang meminimalkan jumlah tabrakan, mudah dihitung, dan mendistribusikan elemen dalam tabel hash secara merata. Ada sejumlah cara umum untuk memperluas metode sisa sederhana. Kita akan mempertimbangkan beberapa di antaranya di sini.

Metode folding untuk membangun fungsi hash dimulai dengan membagi elemen menjadi potongan-potongan berukuran sama (potongan terakhir mungkin tidak berukuran sama). Potongan-potongan ini kemudian ditambahkan bersama untuk memberikan nilai hash yang dihasilkan. Misalnya, jika elemen kita adalah nomor telepon 436-555-4601, kita akan mengambil angka dan membaginya menjadi kelompok 2 (43,65,55,46,01). Setelah penambahan, 43+65+55+46+01, kita mendapatkan 210. Jika kita menganggap tabel hash kita memiliki 11 slot, maka kita perlu melakukan langkah ekstra membagi dengan 11 dan menyimpan sisanya. Dalam hal ini 210 % 11 adalah 1, jadi nomor telepon 436-555-4601 hash ke slot 1. Beberapa metode pelipatan melangkah lebih jauh dan membalikkan setiap bagian sebelum penambahan. Untuk contoh di atas, kita mendapatkan 43+56+55+64+01=219 yang menghasilkan 219 % 11 = 10.

Bagaimana Memilih Fungsi Hash?

Masalah dasar yang terkait dengan pembuatan tabel hash adalah:

- Fungsi hash yang efisien harus dirancang sehingga mendistribusikan nilai indeks dari objek yang disisipkan secara seragam di seluruh tabel.
- Algoritma resolusi tabrakan yang efisien harus dirancang sedemikian rupa sehingga menghitung indeks alternatif untuk kunci yang indeks hashnya sesuai dengan lokasi yang sebelumnya dimasukkan dalam tabel hash.
- Kita harus memilih fungsi hash yang dapat dihitung dengan cepat, mengembalikan nilai dalam rentang lokasi di tabel kita, dan meminimalkan tabrakan.

Karakteristik Fungsi Hash yang Baik

Fungsi hash yang baik harus memiliki karakteristik sebagai berikut:

- Minimalkan tabrakan
- Mudah dan cepat untuk menghitung
- Distribusikan nilai kunci secara merata di tabel hash

- Gunakan semua informasi yang tersedia di kunci
- Memiliki faktor beban tinggi untuk satu set kunci tertentu

6.8 FAKTOR BEBAN

Faktor beban tabel hash yang tidak kosong adalah jumlah item yang disimpan dalam tabel dibagi dengan ukuran tabel. Ini adalah parameter keputusan yang digunakan ketika kita ingin mengulang atau memperluas entri tabel hash yang ada. Ini juga membantu kita dalam menentukan efisiensi fungsi hashing. Itu berarti, ia memberitahu apakah fungsi hash mendistribusikan kunci secara seragam atau tidak.

$$Load\ factor = \frac{Number\ of\ elements\ in\ hash\ table}{Hash\ Table\ size}$$

6.9 *COLLISION*

Fungsi hash digunakan untuk memetakan setiap kunci ke ruang alamat yang berbeda, tetapi secara praktis tidak mungkin untuk membuat fungsi hash seperti itu dan masalahnya disebut tabrakan. *Collision* adalah kondisi dimana dua *record* disimpan di lokasi yang sama.

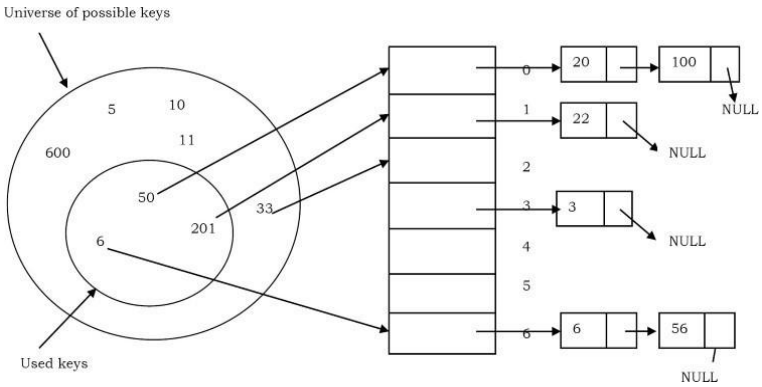
6.10 TEKNIK RESOLUSI *COLLISION*

Proses menemukan lokasi alternatif disebut resolusi tumbukan. Meskipun tabel hash memiliki masalah tabrakan, mereka lebih efisien dalam banyak kasus dibandingkan dengan semua struktur data lainnya, seperti pohon pencarian. Ada sejumlah teknik resolusi tabrakan, dan yang paling populer adalah rantai langsung dan pengalamatan terbuka.

- **Rantai Langsung:** Serangkaian aplikasi daftar tertaut
  - Rantai terpisah
- **Pengalamatan Terbuka:** Implementasi berbasis array
  - Penyelidikan linier (pencarian linier)
  - Penyelidikan kuadrat (pencarian nonlinier)
  - Hashing ganda (gunakan dua fungsi hash)

6.11 RANTAI TERPISAH

Resolusi tabrakan dengan rantai menggabungkan representasi terkait dengan tabel hash. Ketika dua atau lebih *record* di-hash ke lokasi yang sama, *record-record* ini dibentuk menjadi sebuah daftar yang ditautkan secara tunggal yang disebut rantai.



Gambar 6.2 rantai terpisah

6.12 PENGALAMATAN TERBUKA

Dalam pengalamatan terbuka semua kunci disimpan dalam tabel hash itu sendiri. Pendekatan ini juga dikenal sebagai hashing tertutup. Prosedur ini didasarkan pada probing. Tabrakan diselesaikan dengan menyelidik.

Penyelidikan Linier

Interval antara probe ditetapkan pada 1. Dalam probing linier, kita mencari tabel hash secara berurutan, mulai dari lokasi hash asli. Jika suatu lokasi ditempati, kita memeriksa lokasi berikutnya. Kita membungkus dari lokasi meja terakhir ke lokasi meja pertama jika perlu. Fungsi pengulangan adalah sebagai berikut:

rehash(key) = (n + 1)% ukuran table

Salah satu masalah dengan probing linier adalah item tabel cenderung mengelompok bersama dalam tabel hash. Ini berarti bahwa tabel berisi kelompok lokasi yang diduduki secara berurutan yang disebut pengelompokan.

Cluster bisa dekat satu sama lain, dan bergabung menjadi cluster yang lebih besar. Dengan demikian, satu bagian meja mungkin cukup padat, meskipun bagian lain memiliki item yang relatif sedikit. Clustering menyebabkan pencarian probe yang lama dan oleh karena itu menurunkan efisiensi secara keseluruhan.

Lokasi berikutnya yang akan diperiksa ditentukan oleh ukuran langkah, di mana ukuran langkah lainnya (lebih dari satu) dimungkinkan. Ukuran langkah harus relatif prima terhadap ukuran tabel, yaitu pembagi persekutuan terbesarnya harus sama dengan 1. Jika kita memilih ukuran tabel sebagai bilangan prima, maka ukuran langkah apa pun relatif prima terhadap ukuran tabel. Pengelompokan tidak dapat dihindari dengan ukuran langkah yang lebih besar.

Penyelidikan Kuadrat

Interval antara probe meningkat secara proporsional dengan nilai hash (interval sehingga meningkat secara linier, dan indeks dijelaskan oleh fungsi kuadrat). Masalah Clustering dapat dihilangkan jika kita menggunakan metode quadratic probing.

Dalam penyelidikan kuadrat, kita mulai dari lokasi hash asli i. Jika lokasi ditempati, kita memeriksa lokasi i + 1^2 , i +2^2, i + 3^2, i + 4^2 ... Kita membungkus dari lokasi tabel terakhir ke lokasi tabel pertama jika perlu. Fungsi pengulangan adalah sebagai berikut:

rehash(key) = (n + k^2)% ukuran tabel

Contoh: Mari kita asumsikan bahwa ukuran tabel adalah 11 (0..10)
Fungsi Hash: h(kunci) = mod kunci 11
Masukkan kunci
31 mod 11 = 9
19 mod 11 = 8
2 mod 11 = 2
13 mod 11 = 2 → 2 + 1^2 = 3
25 mod 11 = 3 → 3 + 1^2=4
24 mod 11 = 2 → 2 + 1^2, 2 + 2^2 = 6
21 mod 11 = 10
9 mod 11 = 9 → 9 + 1^2, 9 + 2^2 mod 11, 9 + 3^2 mod 11=7

0	
1	
2	2
3	13
4	25
5	5
6	24
7	9
8	19
9	31
10	21

Meskipun clustering dihindari dengan quadratic probing, masih ada kemungkinan clustering. Pengelompokan disebabkan oleh beberapa kunci pencarian yang dipetakan ke kunci hash yang sama. Dengan demikian, urutan probing untuk kunci pencarian tersebut diperpanjang oleh konflik berulang sepanjang urutan probing. Baik probing linier dan kuadrat menggunakan urutan probing yang independen dari kunci pencarian.

Hashing Ganda

Interval antara probe dihitung dengan fungsi hash lain. Hashing ganda mengurangi pengelompokan dengan cara yang lebih baik. Kenaikan untuk urutan probing dihitung dengan menggunakan fungsi hash kedua. Fungsi hash kedua h2 seharusnya:

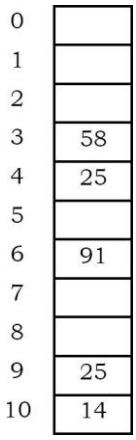
$h2(key) \neq 0$ and $h2 \neq h1$

Kita pertama menyelidiki lokasi h1(key). Jika lokasi ditempati, kita menyelidiki lokasi h1(key) + h2(key), h1(key) + 2 * h2(key), ...

Contoh:

Ukuran meja adalah 11 (0..10)

Fungsi Hash: asumsikan h1(key) = kunci mod 11 dan h2(key) = 7- (key mod 7)



Masukkan kunci:

58 mod 11 = 3

14 mod 11 = 3 → 3 + 7 = 10

91 mod 11 = 3 → 3+ 7,3+ 2* 7 mod 11 = 6

25 mod 11 = 3 → 3 + 3,3 + 2*3 = 9

6.13 PERBANDINGAN TEKNIK RESOLUSI *COLLISION*

Perbandingan: Linear Probing vs. Double Hashing

Pilihan antara probing linier dan hashing ganda tergantung pada biaya komputasi fungsi hash dan pada faktor beban (jumlah elemen per slot) dari tabel. Keduanya menggunakan sedikit probe tetapi hashing ganda membutuhkan lebih banyak waktu karena hash untuk membandingkan dua fungsi hash untuk kunci yang panjang.

Perbandingan: Pengalamatan Terbuka vs. Rantai Terpisah

Agak rumit karena kita harus memperhitungkan penggunaan memori. Rantai terpisah menggunakan memori ekstra untuk tautan. Pengalamatan terbuka membutuhkan memori ekstra secara implisit di dalam tabel untuk mengakhiri urutan probe. Tabel hash dengan alamat terbuka tidak dapat digunakan jika data tidak memiliki kunci unik. Alternatifnya adalah menggunakan tabel hash berantai yang terpisah.

Perbandingan: Metode Pengalamatan Terbuka

Tabel 6.1 metode pengalamatan terbuka

Penyelidikan Linier	Penyelidikan Kuadrat	hashing ganda
Tercepat di antara tiga.	Paling mudah untuk diterapkan dan disebarkan.	Membuat penggunaan memori lebih efisien
Menggunakan sedikit probe.	Menggunakan memori ekstra untuk tautan dan tidak memeriksa semua lokasi dalam tabel.	Menggunakan beberapa probe tetapi membutuhkan lebih banyak waktu
Terjadi masalah yang dikenal sebagai primary clustering.	Terjadi masalah yang dikenal sebagai pengelompokan sekunder.	Lebih rumit untuk diterapkan
Interval antara probe tetap - sering pada 1.	Interval antara probe meningkatkan tp proporsional memiliki nilai	Interval antara probe dihitung dengan yang lain memiliki fungsi

6.14 BAGAIMANA HASHING MENDAPAT KOMPLEKSITAS O(1)?

Dari diskusi sebelumnya, orang meragukan bagaimana hashing mendapatkan O(1) jika beberapa elemen dipetakan ke lokasi yang sama...

Jawaban untuk masalah ini sederhana. Dengan menggunakan faktor beban, kita memastikan bahwa setiap blok (misalnya, daftar tertaut dalam pendekatan rantai terpisah) rata-rata menyimpan jumlah elemen maksimum yang lebih sedikit daripada faktor beban. Juga, dalam praktiknya, faktor beban ini adalah konstan (umumnya, 10 atau 20). Akibatnya, pencarian di 20 elemen atau 10 elemen menjadi konstan.

Jika jumlah rata-rata elemen dalam satu blok lebih besar dari faktor beban, kita mengulangi elemen dengan ukuran tabel hash yang lebih besar. Satu hal yang harus kita ingat adalah bahwa kita mempertimbangkan hunian rata-rata (jumlah total elemen dalam tabel hash dibagi dengan ukuran tabel) ketika memutuskan pengulangan.

Waktu akses tabel tergantung pada faktor beban yang pada gilirannya tergantung pada fungsi hash. Ini karena fungsi hash mendistribusikan elemen ke tabel hash. Untuk alasan ini, kita mengatakan tabel hash memberikan kompleksitas O(1) rata-rata. Juga, kita biasanya menggunakan tabel hash dalam kasus di mana pencarian lebih dari operasi penyisipan dan penghapusan.

6.15 TEKNIK HASHING

Ada dua jenis teknik hashing: hashing statis dan hashing dinamis

Hashing Statis

Jika data diperbaiki maka hashing statis berguna. Dalam hashing statis, set kunci disimpan tetap dan diberikan sebelumnya, dan jumlah halaman utama dalam direktori tetap diperbaiki.

Hashing Dinamis

Jika data tidak diperbaiki, hashing statis dapat memberikan kinerja yang buruk, dalam hal ini hashing dinamis adalah alternatifnya, dalam hal ini kumpulan kunci dapat berubah secara dinamis.

6.16 MASALAH YANG TABEL HASHNYA TIDAK SESUAI

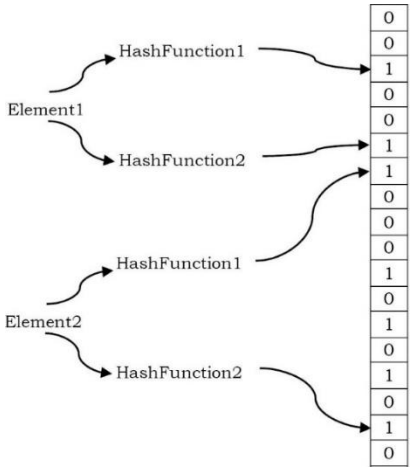
- Masalah yang memerlukan pemesanan data
- Masalah memiliki data multidimensi
- Pencarian awalan, terutama jika kuncinya panjang dan panjangnya bervariasi
- Masalah yang memiliki data dinamis
- Masalah di mana data tidak memiliki kunci unik.

6.17 FILTER MEKAR

Filter *Bloom* adalah struktur data probabilistik yang dirancang untuk memeriksa apakah suatu elemen ada dalam satu set dengan memori dan efisiensi waktu. Ini memberitahu kita bahwa elemen baik pasti tidak di set atau mungkin di set. Struktur data dasar filter *Bloom* adalah Vektor Bit. Algoritma ini ditemukan pada tahun 1970 oleh Burton *Bloom* dan bergantung pada penggunaan sejumlah fungsi hash yang berbeda.

Bagaimana itu bekerja?

Sekarang bit dalam vektor bit telah ditetapkan untuk Element1 dan Element2; kita dapat meminta filter mekar untuk memberi tahu kita jika sesuatu telah terlihat sebelumnya. Elemen di-hash tetapi alih-alih mengatur bit, kali ini pemeriksaan dilakukan dan jika bit yang akan disetel sudah disetel, filter mekar akan mengembalikan true bahwa elemen telah terlihat sebelumnya.



Gambar 6.3 elemen 1 dan 2

Filter *Bloom* dimulai dengan array bit yang diinisialisasi ke nol. Untuk menyimpan nilai data, kita cukup menerapkan k fungsi hash yang berbeda dan memperlakukan nilai k yang dihasilkan sebagai indeks dalam array, dan kita menetapkan setiap k elemen array ke 1. Kita ulangi ini untuk setiap elemen yang kita temui.

Sekarang anggaplah sebuah elemen muncul dan kita ingin tahu apakah kita pernah melihatnya sebelumnya. Apa yang kita lakukan adalah menerapkan k fungsi hash dan mencari elemen array yang ditunjukkan. Jika salah satu dari mereka adalah 0 kita dapat 100% yakin bahwa kita belum pernah menemukan elemen sebelumnya - jika kita pernah, bit akan disetel ke 1. Namun, bahkan jika semuanya adalah satu, kita masih tidak dapat menyimpulkan bahwa kita telah melihat elemen sebelumnya karena semua bit dapat disetel oleh k fungsi hash yang diterapkan ke beberapa elemen lainnya. Yang dapat kita simpulkan hanyalah bahwa kemungkinan besar kita telah menemukan elemen sebelumnya.

Perhatikan bahwa tidak mungkin untuk menghapus elemen dari filter *Bloom*. Alasannya sederhana karena kita tidak dapat menghapus sedikit yang tampaknya milik suatu elemen karena mungkin juga disetel oleh elemen lain.

Jika larik bit sebagian besar kosong, yaitu, disetel ke nol, dan fungsi hash k independen satu sama lain, maka probabilitas positif palsu (yaitu, menyimpulkan bahwa kita telah melihat item data padahal sebenarnya belum) rendah. Misalnya, jika hanya ada k bit yang ditetapkan, kita dapat menyimpulkan bahwa probabilitas positif palsu sangat dekat dengan nol karena satu-satunya kemungkinan kesalahan adalah bahwa kita memasukkan item data yang menghasilkan nilai hash k yang sama - yang tidak mungkin selama fungsi 'memiliki' independen.

Saat array bit terisi, kemungkinan positif palsu perlahan meningkat. Tentu saja ketika bit array penuh, setiap elemen yang ditanyakan diidentifikasi telah dilihat sebelumnya. Jadi jelas kita bisa menukar ruang untuk akurasi dan juga waktu.

Penghapusan satu kali elemen dari filter *Bloom* dapat disimulasikan dengan memiliki filter *Bloom* kedua yang berisi elemen yang telah dihapus. Namun, positif palsu pada filter kedua menjadi negatif palsu pada filter komposit, yang mungkin tidak diinginkan. Dalam pendekatan ini, menambahkan kembali item yang sebelumnya dihapus tidak dimungkinkan, karena seseorang harus menghapusnya dari filter yang dihapus.

Memilih fungsi hash

Persyaratan merancang k fungsi hash independen yang berbeda dapat menjadi penghalang untuk besar k. Untuk fungsi hash yang baik dengan *output* yang lebar, harus ada sedikit jika ada korelasi antara bidang bit yang berbeda dari hash tersebut, sehingga jenis hash ini dapat digunakan untuk menghasilkan beberapa fungsi hash yang berbeda dengan mengiris *output*nya menjadi beberapa bidang bit. Atau, seseorang dapat meneruskan k nilai awal yang berbeda (seperti 0, 1, ..., k - 1) ke fungsi hash yang mengambil nilai awal – atau menambahkan (atau menambahkan) nilai-nilai ini ke kunci. Untuk m dan/atau k yang lebih besar,

independensi di antara fungsi hash dapat dilonggarkan dengan peningkatan tingkat positif palsu yang dapat diabaikan.

Memilih ukuran bit vektor

Filter *Bloom* dengan kesalahan 1% dan nilai optimal k, sebaliknya, hanya membutuhkan sekitar 9,6 bit per elemen – terlepas dari ukuran elemen. Keuntungan ini sebagian berasal dari kekompakannya, diwarisi dari array, dan sebagian dari sifat probabilistiknya. Tingkat positif palsu 1% dapat dikurangi dengan faktor sepuluh dengan menambahkan hanya sekitar 4,8 bit per elemen.

Keuntungan Ruang

Sementara mempertaruhkan positif palsu, filter *Bloom* memiliki keunggulan ruang yang kuat atas struktur data lain untuk mewakili set, seperti pohon pencarian biner self-balancing, percobaan, tabel hash, atau array sederhana atau daftar entri yang ditautkan. Sebagian besar memerlukan penyimpanan setidaknya item data itu sendiri, yang dapat memerlukan di mana saja dari sejumlah kecil bit, untuk bilangan bulat kecil, hingga jumlah bit yang berubah-ubah, seperti untuk string (mencoba adalah pengecualian, karena mereka dapat berbagi penyimpanan antara elemen dengan awalan yang sama). Struktur terkait menimbulkan overhead ruang linier tambahan untuk pointer.

Namun, jika jumlah nilai potensial kecil dan banyak di antaranya dapat diset, filter *Bloom* mudah dilampaui oleh larik bit deterministik, yang hanya membutuhkan satu bit untuk setiap elemen potensial.

Keuntungan Waktu

Filter *Bloom* juga memiliki properti yang tidak biasa yaitu waktu yang dibutuhkan untuk menambahkan item atau untuk memeriksa apakah item dalam himpunan adalah konstanta tetap, O(k), sepenuhnya independen dari jumlah item yang sudah ada di himpunan. Tidak ada struktur data kumpulan ruang-konstan lainnya yang memiliki properti ini, tetapi waktu akses rata-rata tabel hash jarang dapat membuatnya lebih cepat dalam praktiknya daripada beberapa filter *Bloom*. Namun, dalam implementasi perangkat keras, filter *Bloom* bersinar karena pencarian k-nya independen dan dapat diparalelkan.

Penerapan

Lihat Bagian Masalah.

6.18 HASHING: MASALAH & SOLUSI

Soal-1 Menerapkan teknik resolusi tabrakan berantai yang terpisah. Juga, diskusikan kompleksitas waktu dari setiap fungsi.

Solusi: Untuk membuat hashtable dengan ukuran tertentu, katakanlah n, kita mengalokasikan array n/L (yang nilainya biasanya antara 5 dan 20) pointer ke daftar, diinisialisasi ke NULL. Untuk melakukan operasi Cari/Sisipkan/Hapus, pertama-tama kita menghitung indeks tabel dari kunci yang diberikan dengan

menggunakan fungsi hash dan kemudian melakukan operasi yang sesuai dalam daftar linier yang dipertahankan di lokasi itu. Untuk mendapatkan distribusi kunci yang seragam pada tabel hash, pertahankan ukuran tabel sebagai bilangan prima.

```
#define LOAD_FACTOR 20
struct ListNode {
        int key;
        int data;
        struct ListNode *next;
};
struct HashTableNode {
        int bcount;                     //Number of elements in block
        struct ListNode *next;
};
struct HashTable {
        int tsize;
        int count;                      //Number of elements in table
        struct HashTableNode **Table;
};
struct HashTable *CreatHashTable(int size) {
        struct HashTable *h;
        h = (struct HashTable *)malloc(sizeof(struct HashTable));
        if(!h)
                return NULL;
        h→tsize = size/ LOAD_FACTOR;
        h→count = 0;
        h→Table = (struct HashTableNode **) malloc( sizeof(struct HashTableNode *) * h→tsize);
        if(!h→Table) {
                printf("Memory Error");
                return NULL;
        }
        for(int i=0; i < h→tsize; i++) {
            h→Table[i]→next = NULL;
            h→Table[i]→bcount = 0;
        }
        return h;
}
int HashSearch(Struct HashTable *h, int data) {
        struct ListNode *temp;
        temp = h→Table[Hash(data, h→tsize)]→next;               //Assume Hash is a built-in function
        while(temp) {
                if(temp→data == data)
                        return 1;
                temp = temp→next;
        }
        return 0;
}
int HashInsert(Struct HashTable *h, int data) {
        int index;
        struct ListNode *temp, *newNode;
        if(HashSearch(h, data))
              return 0;
        index = Hash(data, h→tsize);                    //Assume Hash is a built-in function
        temp = h→Table[index]→next;
        newNode = (struct ListNode *) malloc(sizeof(struct ListNode));
        if(!newNode) {
                printf("Out of Space");
                return -1;
        }
        newNode→key = index;
        newNode→data = data;
        newNode→next = h→Table[index]→next;
```

```
        h→Table[index]→next = newNode;
        h→Table[index]→bcount++;
        h→count++;
        if(h→count / h→tsize  > LOAD_FACTOR)
                Rehash(h);
        return 1;
}
int HashDelete(Struct HashTable *h, int data) {
        int index;
        struct ListNode *temp, *prev;
        index = Hash(data, h→tsize);
        for(temp = h→Table[index]→next, prev = NULL; temp; prev = temp, temp = temp→next) {
                if(temp→data == data) {
                        if(prev != NULL)
                                prev→next = temp→next;
                        free(temp);
                        h→Table[index]→bcount--;
                        h→count--;
                        return 1;
                }
        }
        return 0;
}
void Rehash(Struct HashTable *h) {
        int oldsize, i, index;
        struct ListNode *p, * temp, *temp2;
        struct HashTableNode **oldTable;
        oldsize = h→tsize;
        oldTable = h→Table;
        h→tsize = h→tsize * 2;
        h→Table = (struct HashTableNode **) malloc(h→tsize * sizeof(struct HashTableNode *));
        if(!h→Table) {
                printf( "Allocation Failed");
                return;
        }
        for(i = 0; i < oldsize; i++) {
                for(temp = oldTable[i]→next; temp; temp = temp→next) {
                        index = Hash(temp→data, h→tsize);
                        temp2 = temp; temp = temp→next;
                        temp2→next = h→Table[index]→next;
                        h→Table[index]→next = temp2;
                }
        }
}
```

CreatHashTable – O(n). HashSearch - rata-rata O(1). HashInsert - O(1) rata-rata. HashDelete - O(1) rata-rata.

Soal-2 Diberikan array karakter, berikan algoritma untuk menghapus duplikat.

Solusi: Mulailah dengan karakter pertama dan periksa apakah karakter tersebut muncul di bagian string yang tersisa menggunakan pencarian linier sederhana. Jika berulang, bawa karakter terakhir ke posisi itu dan kurangi ukuran string satu per satu. Lanjutkan proses ini untuk setiap karakter berbeda dari string yang diberikan.

```
int elem(int *A, size_t n, int e){
   for (int i = 0; i < n; ++i)
      if (A[i] == e)
         return 1;
    return 0;
}
int RemoveDuplicates(int *A, int n){
   int m = 0;
    for (int i = 0; i < n; ++i)
      if (!elem(A, m, A[i]))
         A[m++] = A[i];
    return m;
}
```

Kompleksitas Waktu: $O(n^2)$.

Kompleksitas Ruang: O(1).

Soal-3 Bisakah kita menemukan ide lain untuk menyelesaikan masalah ini dalam waktu yang lebih baik daripada $O(n^2)$?

Perhatikan bahwa urutan karakter dalam solusi tidak penting.

Solusi: Gunakan pengurutan untuk menyatukan karakter yang berulang. Terakhir, pindai array untuk menghapus duplikat di posisi berurutan.

```
int Compare(const void* a, const void *b) {
      return *(char*)a - *(char*)b;
}
void RemoveDuplicates(char s[]) {
      int last, current;
      QuickSort(s, strlen(s), sizeof(char), Compare);
      current = 0, last = 0;
      for(; s[current]; i++) {
                if(s[last] != s[current])
                          s[++last] = s[current];
      }
      s[last] = '\0';
}
```

Kompleksitas Waktu: (nlogn).

Kompleksitas Ruang: O(1).

Soal-4 Bisakah kita menyelesaikan masalah ini dalam satu kali pass over array yang diberikan?

Solusi: Kita dapat menggunakan tabel hash untuk memeriksa apakah suatu karakter berulang dalam string yang diberikan atau tidak. Jika karakter saat ini tidak tersedia di tabel hash, maka masukkan ke dalam tabel hash dan simpan karakter itu di string yang diberikan juga. Jika karakter saat ini ada di tabel hash, lewati karakter itu.

```
void RemoveDuplicates(char s[]) {
        int src, dst;
        struct HastTable *h;
        h = CreatHashTable();
        current = last = 0;
        for(; s[current]; current++) {
                if( !HashSearch(h, s[current]))     {
                        s[last++] = s[current];
                        HashInsert(h, s[current]);
                }
        }
        s[last] = '\0';
}
```

Kompleksitas Waktu: (n) rata-rata.
Kompleksitas Ruang: O(n).

Soal-5 Diberikan dua larik bilangan tak berurut, periksa apakah kedua larik memiliki himpunan bilangan yang sama?

Solusi: Mari kita asumsikan bahwa dua array yang diberikan adalah A dan B. Solusi sederhana untuk yang diberikan masalahnya adalah: untuk setiap elemen A, periksa apakah elemen itu ada di B atau tidak. Masalah muncul dengan pendekatan ini jika ada duplikat. Misalnya pertimbangkan input berikut:

A = {2,5,6,8,10,2,2}
B = {2,5,5,8,10,5,6}

Algoritma di atas memberikan hasil yang salah karena untuk setiap elemen A ada elemen di B juga. Tetapi jika kita melihat jumlah kemunculannya, mereka tidak sama. Masalah ini dapat kita selesaikan dengan memindahkan elemen-elemen yang sudah dibandingkan dengan akhir daftar. Artinya, jika kita menemukan sebuah elemen di B, maka kita pindahkan elemen tersebut ke ujung B, dan pada pencarian selanjutnya kita tidak akan menemukan elemen tersebut. Tetapi kelemahannya adalah perlu swap ekstra. Kompleksitas Waktu pendekatan ini adalah $O(n^2)$, karena untuk setiap elemen A kita harus memindai B.

Soal-6 Bisakah kita meningkatkan kompleksitas waktu Soal-5?

Solusi: Ya. Untuk meningkatkan kompleksitas waktu, mari kita asumsikan bahwa kita telah mengurutkan kedua daftar. Karena ukuran kedua array adalah n, kita

membutuhkan waktu O(n log n) untuk menyortirnya. Setelah menyortir, kita hanya perlu memindai kedua array dengan dua pointer dan melihat apakah mereka menunjuk ke elemen yang sama setiap saat, dan terus memindahkan pointer sampai kita mencapai akhir array.

Kompleksitas Waktu dari pendekatan ini adalah O(n log n). Ini karena kita membutuhkan O(n log n) untuk menyortir array. Setelah penyortiran, kita membutuhkan O(n) waktu untuk pemindaian tetapi lebih sedikit dibandingkan dengan O(n log n).

Soal-7 Bisakah kita lebih meningkatkan kompleksitas waktu Soal-5?

Solusi: Ya, dengan menggunakan tabel hash. Untuk ini, pertimbangkan algoritma berikut.

Algoritma:

- Membangun tabel hash dengan elemen array A sebagai kunci.
- Saat memasukkan elemen, lacak frekuensi nomor untuk setiap nomor. Itu berarti, jika ada duplikat, maka tingkatkan penghitung kunci yang sesuai.
- Setelah membuat tabel hash untuk elemen A, sekarang pindai array B.
- Untuk setiap kemunculan elemen B, kurangi nilai penghitung yang sesuai.
- Pada akhirnya, periksa apakah semua penghitung nol atau tidak.
- Jika semua counter adalah nol, maka kedua array sama jika tidak maka array berbeda.

Kompleksitas Waktu; O(n) untuk memindai array.

Kompleksitas Ruang; O(n) untuk tabel hash.

Soal-8 Diberikan daftar pasangan bilangan; jika pasangan(i,j) ada, dan pasangan(j,i) ada, laporkan semua pasangan tersebut. Misalnya, dalam {{1,3},{2,6},{3,5},{7,4},{5,3},{8,7}}, kita melihat bahwa {3,5} dan {5,3} hadir. Laporkan pasangan ini saat Anda menemukan {5,3}. Kita menyebut pasangan seperti itu 'pasangan simetris'. Jadi, berikan algoritma yang efisien untuk menemukan semua pasangan tersebut.

Solusi: Dengan menggunakan hashing, kita bisa menyelesaikan masalah ini hanya dalam sekali scan. Perhatikan algoritma berikut.

Algoritma:

- Baca pasangan elemen satu per satu dan masukkan ke dalam tabel hash. Untuk setiap pasangan, pertimbangkan elemen pertama sebagai kunci dan elemen kedua sebagai nilai.
- Saat memasukkan elemen, periksa apakah hashing elemen kedua dari pasangan saat ini sama dengan nomor pertama dari pasangan saat ini.

- Jika mereka sama, maka itu menunjukkan pasangan simetris keluar dan mengeluarkan pasangan itu.
- Jika tidak, masukkan elemen itu ke dalamnya. Itu berarti, gunakan angka pertama dari pasangan saat ini sebagai kunci dan angka kedua sebagai nilai dan masukkan ke dalam tabel hash.
- Pada saat kita menyelesaikan pemindaian semua pasangan, kita memiliki *output* semua pasangan simetris.

Kompleksitas Waktu; O(n) untuk memindai array. Perhatikan bahwa kita hanya melakukan pemindaian input.
Kompleksitas Ruang; O(n) untuk tabel hash.

Soal-9 Diberikan daftar tertaut tunggal, periksa apakah ada loop di dalamnya atau tidak.

Solusi: Menggunakan Algoritma Tabel Hash:

- Lintasi node daftar tertaut satu per satu.
- Periksa apakah alamat node ada di tabel hash atau tidak.
- Jika sudah ada di tabel hash, itu menunjukkan kita mengunjungi node yang sudah dikunjungi. Ini hanya mungkin jika daftar tertaut yang diberikan memiliki loop di dalamnya.
- Jika alamat node tidak ada di tabel hash, maka masukkan alamat node tersebut ke tabel hash.
- Lanjutkan proses ini sampai kita mencapai akhir dari linked list atau kita menemukan loop.

Kompleksitas Waktu; O(n) untuk memindai daftar tertaut. Perhatikan bahwa kita hanya melakukan pemindaian input.
Kompleksitas Ruang; O(n) untuk tabel hash.
Catatan: *untuk solusi yang efisien, lihat bab Daftar Tertaut.*

Soal-10 Diberikan array 101 elemen. Dari mereka 50 elemen berbeda, 24 elemen diulang 2 kali, dan satu elemen diulang 3 kali. Temukan elemen yang diulang 3 kali di O(1).

Solusi Menggunakan Algoritma Tabel Hash:

- Pindai larik masukan satu per satu.
- Periksa apakah elemen tersebut sudah ada di tabel hash atau belum.
- Jika sudah ada di tabel hash, tambah nilai counter-nya [ini menunjukkan jumlah kemunculan elemen].
- Jika elemen tersebut tidak ada pada tabel hash, masukkan node tersebut ke dalam tabel hash dengan nilai counter 1.

- Lanjutkan proses ini hingga mencapai akhir array.

Kompleksitas Waktu: O(n), karena kita melakukan dua pemindaian.
Kompleksitas Ruang: O(n), untuk tabel hash.
Catatan: *Untuk solusi yang efisien, lihat bab Pencarian.*

Soal-11 Diberikan m himpunan bilangan bulat yang memiliki n elemen di dalamnya, berikan algoritma untuk menemukan elemen yang muncul dalam jumlah himpunan maksimum?

Solusi Menggunakan Algoritma Tabel Hash:

- Pindai set input satu per satu.
- Untuk setiap elemen melacak counter. Penghitung menunjukkan frekuensi kemunculan di semua set.
- Setelah menyelesaikan pemindaian semua set, pilih yang memiliki nilai penghitung maksimum.

Kompleksitas Waktu: O(mn), karena kita perlu memindai semua set.
Kompleksitas Ruang: O(mn), untuk tabel hash. Karena, dalam kasus terburuk semua elemen mungkin berbeda.

Soal-12 Diberikan dua himpunan A dan B, dan sebuah bilangan K, Berikan algoritma untuk menemukan apakah terdapat sepasang elemen, satu dari A dan satu dari B, yang berjumlah K.

Solusi: Untuk mempermudah, mari kita asumsikan bahwa ukuran A adalah m dan ukuran B adalah n.

Algoritma:

- Pilih set yang memiliki elemen minimum.
- Untuk set yang dipilih, buat tabel hash. Kita dapat menggunakan kunci dan nilai sebagai hal yang sama.
- Sekarang scan array kedua dan periksa apakah (elemen K-selected) ada di tabel hash atau tidak.
- Jika ada maka kembalikan pasangan elemen.
- Jika tidak, lanjutkan sampai kita mencapai akhir set.

Kompleksitas Waktu: O(Max(m,n)), karena kita melakukan dua pemindaian.
Kompleksitas Ruang: O(Min(m,n)), untuk tabel hash. Kita dapat memilih set kecil untuk membuat tabel hash.

Soal-13 Berikan algoritma untuk menghapus karakter tertentu dari string tertentu yang diberikan dalam string lain?

Solusi: Untuk mempermudah, mari kita asumsikan bahwa jumlah maksimum karakter yang berbeda adalah 256. Pertama kita membuat array tambahan yang diinisialisasi ke 0. Pindai karakter yang akan dihapus, dan untuk masing-masing karakter tersebut kita atur nilainya menjadi 1, yang menunjukkan bahwa kita perlu menghapus karakter itu.

Setelah inisialisasi, pindai string input, dan untuk setiap karakter, kita memeriksa apakah karakter itu perlu dihapus atau tidak. Jika flag sudah diatur maka kita cukup melompat ke karakter berikutnya, jika tidak kita menyimpan karakter di string input. Lanjutkan proses ini sampai kita mencapai akhir string input. Semua operasi ini dapat kita lakukan di tempat seperti yang diberikan di bawah ini.

```
void RemoveChars(char str[], char removeTheseChars[]) {
    int srcInd, destInd;
    int auxi[256]; //additional array
    for(srcInd =0; srcInd<256; srcIndex++)
        auxi[srcInd]=0;
    //set true for all characters to be removed
    srcIndex=0;
    while(removeTheseChars[srcInd]) {
        auxi[removeTheseChars[srcInd]]=1;
        srcInd++;
    }
    //copy chars unless it must be removed
    srcInd=destInd=0;
    while(str[srcInd++]) {
        if(!auxi[str[srcInd]])
            str[destInd++]=str[srcInd];
    }
}
```

Kompleksitas Waktu: Waktu untuk memindai karakter yang akan dihapus + Waktu untuk memindai array input= O(n) +O(m) ≈ O(n). Di mana m adalah panjang karakter yang akan dihapus dan n adalah panjang string input.

Kompleksitas Ruang: O(m), panjang karakter yang akan dihapus. Tetapi karena kita mengasumsikan jumlah maksimum karakter yang berbeda adalah 256, kita dapat memperlakukan ini sebagai konstanta. Tetapi kita harus ingat bahwa ketika kita berurusan dengan karakter multi-byte, jumlah total karakter yang berbeda jauh lebih dari 256.

Soal-14 Berikan algoritma untuk menemukan karakter pertama yang tidak berulang dalam sebuah string. Misalnya, karakter pertama yang tidak berulang dalam string "abzddab" adalah 'z'.

Solusi: Solusi untuk masalah ini adalah sepele. Untuk setiap karakter dalam string yang diberikan, kita dapat memindai string yang tersisa jika karakter itu muncul di dalamnya. Jika tidak muncul maka kita selesai dengan solusinya dan kita

kembalikan karakter tersebut. Jika karakter muncul di string yang tersisa, maka lanjutkan ke karakter berikutnya.

```
char FirstNonRepeatedChar( char *str ) {
        int i, j, repeated = 0;
        int len = strlen(str);
        for(i = 0; i < len; i++ )     {
                repeated = 0;
                for( j = 0; j < len; j++ )     {
                        if( i != j && str[i] == str[j] ) {
                                repeated = 1;
                                break;
                        }
                }
                if( repeated == 0 )          // Found the first non-repeated character
                        return str[i];
        }
        return '';
}
```

Kompleksitas Waktu: $O(n^2)$, untuk dua perulangan.
Kompleksitas Ruang: O(1).

Soal-15 Bisakah kita meningkatkan kompleksitas waktu Soal-13?

Solusi **Ya**. Dengan menggunakan tabel hash kita dapat mengurangi kompleksitas waktu. Buat tabel hash dengan membaca semua karakter dalam string input dan menghitung berapa kali setiap karakter muncul. Setelah membuat tabel hash, kita dapat membaca entri tabel hash untuk melihat elemen mana yang memiliki jumlah sama dengan 1. Pendekatan ini membutuhkan ruang O(n) tetapi juga mengurangi kompleksitas waktu menjadi O(n).

```
char FirstNonRepeatedCharUsinghash( char * str ) {
        int i, len=strlen(str);
        int count[256]; //additional array
        for(i=0;i<len;++i)
                count[i] = 0;
        for(i=0;i<len;++i)
                count[str[i]]++;
        for(i=0; i<len; ++i) {
                if(count[str[i]]==1) {
                        printf("%c",str[i]);
                        break;
                }
        }
        if(i==len)
                printf("No Non-repeated Characters");
        return 0;
}
```

Kompleksitas Waktu; Kita memiliki O(n) untuk membuat tabel hash dan O(n) lainnya untuk membaca entri tabel hash. Jadi total waktunya adalah O(n) + O(n) = O(2n) ≈ O(n).
Kompleksitas Ruang: O(n) untuk menjaga nilai hitungan.

Soal-16 Diberikan sebuah string, berikan algoritma untuk menemukan huruf pertama yang berulang dalam sebuah string?

Solusi: Solusi untuk masalah ini agak mirip dengan Soal-13 dan Soal-15. Satu-satunya perbedaan adalah, alih-alih memindai tabel hash dua kali, kita dapat memberikan jawabannya hanya dalam satu pemindaian. Ini karena saat memasukkan ke dalam tabel hash kita bisa melihat apakah elemen itu sudah ada atau belum. Jika sudah ada maka kita tinggal mengembalikan karakter tersebut.

```
char FirstRepeatedCharUsinghash( char * str ) {
    int i, len=strlen(str);
    int count[256]; //additional array
    for(i=0;i<len;++i)
        count[i] = 0;
    for(i=0; i<len; ++i) {
        if(count[str[i]]==1) {
            printf("%s",str[i]);
            break;
        }
        else count[str[i]]++;
    }
    if(i==len)
        printf("No Repeated Characters");
    return 0;
}
```

Kompleksitas Waktu: Kita memiliki O(n) untuk memindai dan membuat tabel hash. Perhatikan bahwa kita hanya perlu satu pemindaian untuk masalah ini. Jadi total waktunya adalah O(n). Kompleksitas Ruang: O(n) untuk menjaga nilai hitungan.

Soal-17 Diberikan sebuah array n angka, buatlah algoritma yang menampilkan semua pasangan yang jumlahnya S.

Solusi: Masalah ini mirip dengan Masalah-12. Tetapi alih-alih menggunakan dua set, kita hanya menggunakan satu set.

Algoritma:

- Pindai elemen array input satu per satu dan buat tabel hash. Baik kunci dan nilai bisa sama.
- Setelah membuat tabel hash, pindai lagi array input dan periksa apakah (S –
- elemen yang dipilih) keluar di tabel hash atau tidak.
- Jika keluar maka kembalikan pasangan elemen.
- Jika tidak, lanjutkan dan baca semua elemen array.

Kompleksitas Waktu; Kita memiliki O(n) untuk membuat tabel hash dan O(n) lainnya untuk membaca entri dari tabel hash. Jadi total waktunya adalah O(n) + O(n) = O(2n) ≈ O(n).
Kompleksitas Ruang: O(n) untuk menjaga nilai hitungan.

Soal-18 Apakah ada cara lain untuk menyelesaikan Soal-17?

Solusi: Ya. Solusi alternatif untuk masalah ini melibatkan penyortiran. Pertama urutkan array input. Setelah menyortir, gunakan dua penunjuk, satu di awal dan satu lagi di akhir. Setiap kali tambahkan nilai kedua indeks dan lihat apakah jumlahnya sama dengan S. Jika sama, cetak pasangan itu. Jika tidak, tambah penunjuk kiri jika jumlahnya lebih kecil dari S dan kurangi penunjuk kanan jika jumlahnya lebih besar dari S.
Kompleksitas Waktu:
Waktu untuk menyortir + Waktu untuk memindai = O(nlogn) + O(n)≈O(nlogn).
Kompleksitas Ruang: O(1).

Soal-19 Kita memiliki file dengan jutaan baris data. Hanya dua garis yang identik; selebihnya unik. Setiap baris sangat panjang sehingga mungkin tidak muat di memori. Apa solusi paling efisien untuk menemukan garis yang identik?

Solusi: Karena satu baris lengkap mungkin tidak muat ke dalam memori utama, baca sebagian baris dan hitung hash dari sebagian baris itu. Kemudian baca bagian berikutnya dari baris dan hitung hash. Kali ini gunakan hash sebelumnya juga saat menghitung nilai hash baru. Lanjutkan proses ini sampai kita menemukan hash untuk baris yang lengkap. Lakukan ini untuk setiap baris dan simpan semua nilai hash dalam file [atau pertahankan tabel hash dari hash ini]. Jika suatu saat Anda mendapatkan nilai hash yang sama, baca baris yang sesuai bagian demi bagian dan bandingkan.

Catatan Lihat bab Pencarian untuk masalah terkait.

Soal-20 Jika h adalah fungsi hashing dan digunakan untuk hash n kunci ke dalam tabel ukuran s, di mana n <= s, jumlah tumbukan yang diharapkan yang melibatkan kunci X tertentu adalah :

(A) kurang dari 1.
(B) kurang dari n.
(C) kurang dari s.
(D) kurang dari $\frac{n}{2}$.

Solusi: A

Soal-21 Menerapkan Filter *Bloom*

Solusi: Filter *Bloom* adalah struktur data yang dirancang untuk memberi tahu, dengan cepat dan hemat memori, apakah suatu elemen ada dalam suatu himpunan. Ini didasarkan pada mekanisme probabilistik di mana hasil pengambilan positif palsu dimungkinkan, tetapi negatif palsu tidak. Pada akhirnya kita akan melihat bagaimana menyetel parameter untuk meminimalkan jumlah hasil positif palsu.

Mari kita mulai dengan sedikit teori. Ide di balik filter *Bloom* adalah untuk mengalokasikan bit vektor dengan panjang m, awalnya semua diatur ke 0, dan kemudian memilih k fungsi hash independen, h1, h2, ..., hk , masing-masing dengan rentang [1..m]. Ketika elemen a ditambahkan ke himpunan maka bit pada posisi h1(a), h2(a), ..., h_k (a) dalam bit vektor diset ke 1.

Diberikan elemen kueri q kita dapat menguji apakah itu di set menggunakan bit pada posisi h1(q), h2(q), ..., hk (q) dalam vektor. Jika salah satu dari bit ini adalah 0 kita melaporkan bahwa q tidak di set jika tidak, kita melaporkan bahwa q adalah. Hal yang harus kita perhatikan adalah bahwa dalam kasus pertama masih ada beberapa kemungkinan bahwa q tidak ada dalam himpunan yang dapat membawa kita ke respons positif palsu.

```
typedef unsigned int (*hashFunctionPointer)(const char *);
struct Bloom{
    int bloomArraySize;
    unsigned char *bloomArray;
    int nHashFunctions;
    hashFunctionPointer *funcsArray;
};
#define SETBLOOMBIT(a, n) (a[n/CHAR_BIT] |= (1<<(n%CHAR_BIT)))
#define GETBLOOMBIT(a, n) (a[n/CHAR_BIT] & (1<<(n%CHAR_BIT)))
struct Bloom *createBloom(int size, int nHashFunctions, ...){
    struct Bloom *blm;
    va_list l;
    int n;
    if(!(blm=malloc(sizeof(struct Bloom))))
        return NULL;
    if(!(blm->bloomArray=calloc((size+CHAR_BIT-1)/CHAR_BIT, sizeof(char)))) {
        free(blm);
        return NULL;
    }
    if(!(blm->funcsArray=(hashFunctionPointer*)malloc(nHashFunctions*sizeof(hashFunctionPointer)))) {
        free(blm->bloomArray);
        free(blm);
        return NULL;
    }
    va_start(l, nHashFunctions);
    for(n=0; n<nHashFunctions; ++n) {
        blm->funcsArray[n]=va_arg(l, hashFunctionPointer);
    }
    va_end(l);
```

```
    blm→nHashFunctions=nHashFunctions;
    blm→bloomArraySize=size;
    return blm;
}
int deleteBloom(struct Bloom *blm){
    free(blm→bloomArray);
    free(blm→funcsArray);
    free(blm);
    return 0;
}
int addElementBloom(struct Bloom *blm, const char *s){
    for(int n=0; n<blm→nHashFunctions; ++n) {
        SETBLOOMBIT(blm→bloomArray, blm→funcsArray[n](s)%blm→bloomArraySize);
    }
    return 0;
}
int checkElementBloom(struct Bloom *blm, const char *s){
    for(int n=0; n<blm→nHashFunctions; ++n) {
        if(!(GETBLOOMBIT(blm→bloomArray, blm→funcsArray[n](s)%blm→bloomArraySize))) return 0;
    }
    return 1;
}
unsigned int shiftAddXORHash(const char *key){
    unsigned int h=0;
    while(*key) h^=(h<<5)+(h>>2)+(unsigned char)*key++;
    return h;
}
unsigned int XORHash(const char *key){
    unsigned int h=0;
    hash_t h=0;
    while(*key) h^=*key++;
    return h;
}
int test(){
    FILE *fp;
    char line[1024];
    char *p;
    struct Bloom *blm;
    if(!(blm=createBloom(1500000, 2, shiftAddXORHash, XORHash))) {
        fprintf(stderr, "ERROR: Could not create Bloom filter\n");
        return -1;
    }
    if(!(fp=fopen("path", "r"))) {
        fprintf(stderr, "ERROR: Could not open file %s\n", argv[1]);
        return -1;
    }
    while(fgets(line, 1024, fp)) {
        if((p=strchr(line, '\r'))) *p='\0';
        if((p=strchr(line, '\n'))) *p='\0';
        addElementBloom(blm, line);
    }
    fclose(fp);
    while(fgets(line, 1024, stdin)) {
        if((p=strchr(line, '\r'))) *p='\0';
        if((p=strchr(line, '\n'))) *p='\0';
        p=strtok(line, " \t,.;:\r\n?!-/()");
        while(p) {
            if(!checkBloom(blm, p)) {
                printf("No match for ford \"%s\"\n", p);
            }
            p=strtok(NULL, " \t,.;:\r\n?!-/()");
        }
    }
    deleteBloom(blm);
    return 1;
}
```

BAB 7
ALGORITMA STRING

7.1 PENDAHULUAN

Untuk memahami pentingnya algoritma string mari kita pertimbangkan kasus memasukkan URL (Uniform Untuk memahami pentingnya algoritma string mari kita pertimbangkan kasus memasukkan URL (Uniform Resource Locator) di browser apapun (misalnya, Internet Explorer, Firefox, atau Google Chrome). Anda akan melihat bahwa setelah mengetik awalan URL, daftar semua kemungkinan URL akan ditampilkan. Artinya, browser sedang melakukan beberapa pemrosesan internal dan memberi kita daftar URL yang cocok. Teknik ini terkadang disebut otomatis – selesai.

Demikian pula, pertimbangkan kasus memasukkan nama direktori di antarmuka baris perintah (di Windows dan UNIX). Setelah mengetik awalan nama direktori, jika kita menekan tombol tab, kita mendapatkan daftar semua nama direktori yang cocok yang tersedia. Ini adalah contoh lain dari penyelesaian otomatis.

Untuk mendukung operasi semacam ini, kita memerlukan struktur data yang menyimpan data string secara efisien. Dalam bab ini, kita akan melihat struktur data yang berguna untuk mengimplementasikan algoritma string.

Kita memulai diskusi kita dengan masalah dasar string: diberikan string, bagaimana kita mencari substring (pola)? Ini disebut masalah pencocokan string. Setelah membahas berbagai algoritma pencocokan string, kita akan melihat struktur data yang berbeda untuk menyimpan string.

7.2 ALGORITMA PENCOCOKAN

Di bagian ini, kita berkonsentrasi untuk memeriksa apakah pola P adalah substring dari string lain T (T singkatan dari teks) atau tidak. Karena kita mencoba untuk memeriksa string P yang tetap, terkadang algoritma ini disebut algoritma pencocokan string yang tepat. Untuk menyederhanakan diskusi kita, mari kita asumsikan bahwa panjang teks T yang diberikan adalah n dan panjang pola P yang kita coba cocokkan memiliki panjang m. Artinya, T memiliki karakter dari 0 hingga n – 1 (T[0 ...n – 1]) dan P memiliki karakter dari 0 hingga m – 1 (T[0 ...m – 1]). Algoritma ini diimplementasikan dalam C++ sebagai strstr().

Di bagian selanjutnya, kita mulai dengan metode brute force dan secara bertahap bergerak menuju algoritma yang lebih baik.

- Metode Brute Force
- Algoritma Pencocokan String Rabin-Karp
- Pencocokan String dengan Finite Automata
- Algoritma KMP

- Algoritma Boyer-Moore
- Pohon Sufiks

7.3 METODE BRUTE FORCE

Dalam metode ini, untuk setiap kemungkinan posisi dalam teks T kita periksa apakah pola P cocok atau tidak. Karena panjang T adalah n, kita memiliki n – m + 1 kemungkinan pilihan untuk perbandingan. Ini karena kita tidak perlu memeriksa lokasi m – 1 terakhir dari T karena panjang polanya adalah m. Algoritma berikut mencari kemunculan pertama dari string pola P dalam string teks T.

Algoritma

```
int BruteForceStringMatch (int T[], int n,  int P[], int m) {
    for (int i  = 0; i <=n - m; i++) {
            int j  = 0;
            while (j < m && P[j] == T[i + j])
                j  = j + 1;
            if(j == m )
                return i;
    }
    return -1;
}
```

Kompleksitas Waktu: O((n – m + 1) × m) ≈ O(n × m). Kompleksitas Ruang: O(1).

7.4 ALGORTIMA PENCOCOKAN STRING RABIN-KARP

Dalam metode ini, kita akan menggunakan teknik hashing dan alih-alih memeriksa setiap kemungkinan posisi di T, kita hanya memeriksa apakah hashing P dan hashing m karakter T memberikan hasil yang sama.

Awalnya, terapkan fungsi hash ke m karakter pertama T dan periksa apakah hasil ini dan hasil hashing P sama atau tidak. Jika tidak sama, lanjutkan ke karakter T berikutnya dan terapkan lagi fungsi hash ke m karakter (dengan memulai dari karakter kedua). Jika sama maka kita bandingkan m karakter dari T dengan P.

Memilih Fungsi Hash

Pada setiap langkah, karena kita menemukan hash dari m karakter T, kita membutuhkan fungsi hash yang efisien. Jika fungsi hash mengambil kompleksitas O(m) di setiap langkah, maka kompleksitas totalnya adalah O(n × m). Ini lebih buruk daripada metode brute force karena pertama kita menerapkan fungsi hash dan juga membandingkan.

Tujuan kita adalah untuk memilih fungsi hash yang membutuhkan kompleksitas O(1) untuk menemukan hash dari m karakter T setiap waktu. Hanya dengan begitu kita dapat mengurangi kompleksitas total dari algoritma. Jika fungsi hash tidak bagus (kasus terburuk), kompleksitas algoritma Rabin-Karp adalah O(n – m + 1) × m) O(n × m). Jika kita memilih fungsi hash yang baik, kompleksitas dari kompleksitas algoritma Rabin-Karp adalah O(m + n). Sekarang mari kita lihat bagaimana memilih fungsi hash yang dapat menghitung hash dari m karakter T pada setiap langkah dalam O(1).

Untuk mempermudah, mari kita asumsikan bahwa karakter yang digunakan dalam string T hanya bilangan bulat. Artinya, semua karakter dalam T {0,1,2,...,9 }. Karena semuanya adalah bilangan bulat, kita dapat melihat string m karakter berurutan sebagai angka desimal. Misalnya, string 61815' sesuai dengan angka 61815. Dengan asumsi di atas, pola P juga merupakan nilai desimal, dan mari kita asumsikan bahwa nilai desimal P adalah p. Untuk teks yang diberikan T[0..n – 1], misalkan t(i) menyatakan decimal nilai panjang–m substring T[i..i + m – 1] untuk i = 0,1, ...,n – m– 1. Jadi, t(i) == p jika dan hanya jika T[i..i + m – 1] == P[0..m – 1]. Kita dapat menghitung p dalam waktu O(m) menggunakan Aturan Horner sebagai:

$$p = P[m-1] + 10(P[m-2] + 10(P[m-3] + \ldots + 10\,(P[1] + 10\,P[0])\ldots))$$

Kode untuk asumsi di atas adalah:

```
value = 0;
for (int i = 0; i < m-1; i++) {
    value = value * 10;
    value = value + P[i];
}
```

Kita dapat menghitung semua nilai t(i), untuk nilai i = 0,1,..., n – m – 1 dalam total waktu O(n). Nilai t(0) dapat dihitung dengan cara yang sama dari T[0.. m – 1] dalam waktu O(m). Untuk menghitung nilai sisa t(0), t(1),..., t(n – m – 1), pahami bahwa t(i + 1) dapat dihitung dari t(i) dalam waktu konstan.

$$t(i+1) = 10 * (t(i) - 10^{m-1} * T[i]) + T[i + m - 1]$$

Misalnya, jika T = 123456" dan m = 3

$$t(0) = 123$$
$$t(1) = 10 * (123 - 100 * 1) + 4 = 234$$

Penjelasan langkah demi langkah

Pertama : hilangkan digit pertama : 123 – 100 * 1 = 23

Kedua: Kalikan dengan 10 untuk menggesernya : 23 * 10 = 230

Ketiga: Tambahkan digit terakhir : 230 + 4 = 234

Algoritma berjalan dengan membandingkan, t(i) dengan p.

Ketika t(i) == p, maka kita telah menemukan substring P di T, mulai dari posisi i.

7.5 PENCOCOKAN STRING DENGAN FINITE AUTOMATA ALGORITMA

Dalam metode ini kita menggunakan finite automata yang merupakan konsep dari Theory of Computation (ToC). Sebelum melihat algoritmanya, mari kita lihat dulu definisi dari finite automata.

Automata Terbatas

Sebuah robot berhingga F adalah sebuah 5-tupel (Q,q0,A,∑,δ), di mana

- Q adalah himpunan berhingga keadaan
- q0 Q adalah keadaan awal
- A Q adalah himpunan keadaan menerima
- adalah alfabet input berhingga

- adalah fungsi transisi yang memberikan keadaan berikutnya untuk keadaan saat ini dan input

Bagaimana Cara Kerja Finite Automata?

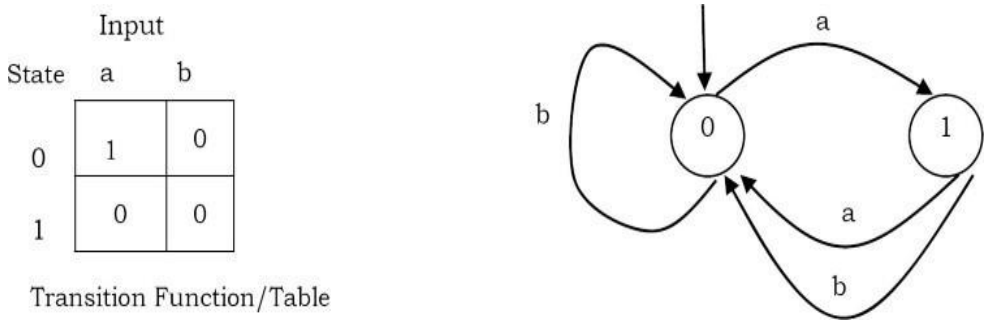
- Otomat berhingga F dimulai pada keadaan q0
- Membaca karakter dari satu per satu
- Jika F dalam keadaan q dan membaca karakter input a, F berpindah ke keadaan (q,d)
- Pada akhirnya, jika keadaannya di A, maka kita katakan, F menerima string input yang dibaca sejauh ini
- Jika string input tidak diterima, ini disebut string ditolak

Contoh:

Mari kita asumsikan bahwa ,

$$Q = \{0,1\{,q0 = 0,A = \{1\},\sum = \{a, b\}. (q,d)$$

seperti terlihat pada tabel/diagram transisi. Ini menerima string yang diakhiri dengan angka ganjil; misalnya, abbaaa diterima, aa ditolak.



Gambar 7.1 Finite Automata

Catatan Penting untuk Membangun Automata Hingga

Untuk membangun automata, pertama kita mulai dengan keadaan awal. FA akan berada dalam keadaan k jika k karakter pola telah cocok. Jika karakter teks berikutnya sama dengan karakter pola c, kita telah mencocokkan k + 1 karakter dan FA memasuki keadaan k + 1. Jika karakter teks berikutnya tidak sama dengan karakter pola, maka FA masuk ke keadaan 0 ,1,2, atau k, tergantung pada berapa banyak karakter pola awal yang cocok dengan karakter teks yang diakhiri dengan c.

Algoritma Pencocokan

Sekarang, mari kita berkonsentrasi pada algoritma pencocokan.

- Untuk pola yang diberikan P[0.. m – 1], pertama-tama kita perlu membuat otomat berhingga F
    - Himpunan keadaan adalah Q = {0,1,2, ,m}

- Keadaan awal adalah 0
- Satu-satunya negara yang menerima adalah m
- Waktu untuk membangun F bisa besar jika besar

- Pindai string teks T[0.. n – 1] untuk menemukan semua kemunculan pola P[0.. m – 1]
- Pencocokan string efisien: (n)
  - Setiap karakter diperiksa tepat satu kali
  - Waktu yang konstan untuk setiap karakter
  - Tetapi waktu untuk menghitung (fungsi transisi) adalah O(m|∑|). Ini karena memiliki entri O(m|∑|). Jika kita asumsikan |∑| konstan maka kompleksitasnya menjadi O(m).

Algoritma:

```
//Input: Pattern string P[0..m-1], δ and F
//Goal: All valid shifts displayed
FiniteAutomataStringMatcher(int P[], int m, F, δ) {
    q = 0;
    for (int i = 0; i < m; i++)
        q = δ(q,T[i]);
        if(q == m)
            printf("Pattern occurs with shift: %d", i-m);
}
```

Kompleksitas Waktu: O(m).

7.6 ALGORITMA KMP

Seperti sebelumnya, mari kita asumsikan bahwa T adalah string yang akan dicari dan P adalah pola yang akan dicocokkan. Algoritma ini dipresentasikan oleh Knuth, Morris dan Pratt. Dibutuhkan O(n) kompleksitas waktu untuk mencari pola. Untuk mendapatkan kompleksitas waktu O(n), ia menghindari perbandingan dengan elemen T yang sebelumnya terlibat dibandingkan dengan beberapa elemen pola P.

Algoritma ini menggunakan tabel dan secara umum kita sebut prefix function atau prefix table atau fail function F. Pertama kita akan melihat bagaimana cara mengisi tabel ini dan kemudian bagaimana mencari pola menggunakan tabel ini. Fungsi awalan F untuk sebuah pola menyimpan pengetahuan tentang bagaimana pola itu cocok dengan pergeseran itu sendiri. Informasi ini dapat digunakan untuk menghindari pergeseran pola P yang tidak berguna. Artinya tabel ini dapat digunakan untuk menghindari backtracking pada string T.

Tabel Awalan

Sebagai contoh, asumsikan bahwa P = *a b a b a c a*. Untuk pola ini, mari kita ikuti petunjuk langkah demi langkah untuk mengisi tabel awalan F. Awalnya: m = length[P] = 7,F[0] = 0 dan F[1] = 0.

```
int F[]; //assume F is a global array
void Prefix-Table(int P[], int m) {
    int i=1,j=0, F[0]=0;
    while(i<m) {
          if(P[i]==P[j]) {
                F[i]=j+1;
                i++;
                j++;
          }
          else if(j>0)
                j=F[j-1];
          else {
                F[i]=0;
                i++;
          }
    }
}
```

Langkah 1: i = 1,j = 0,F[1] =0

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0					

Langkah 2: i = 2,j = 0,F[2] = 1

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1				

Langkah 3: i = 3,j = 1,F[3] =2

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2			

Langkah 4: i = 4,j = 2,F[4] =3

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3		

Langkah 5: i = 5,j = 3,F[5] = 1

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	

Langkah 6: i = 6,j = 1,F[6] =1

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	1

Pada langkah ini pengisian tabel awalan sudah selesai.

Algoritma Pencocokan

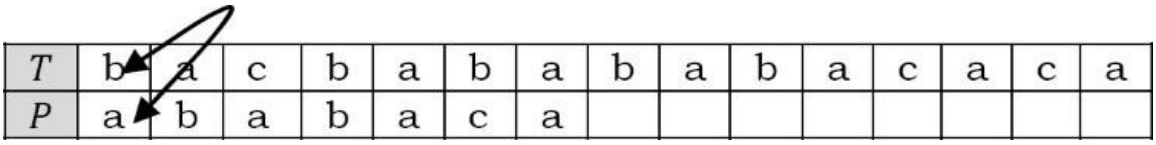
Algoritma KMP mengambil pola P, string T dan fungsi awalan F sebagai input, dan menemukan kecocokan P di T

```
int KMP(char T[], int n, int P[], int m) {
    int i=0,j=0;
    Prefix-Table(P,m);
    while(i<n) {
        if(T[i]==P[j]) {
            if(j==m-1)
                return i-j;
            else {
                i++;
                j++;
            }
        }
        else if(j>0)
            j=F[j-1];
        else    i++;
    }
    return -1;
}
```

Kompleksitas Waktu: O(m + n), di mana m adalah panjang pola dan n adalah panjang teks yang akan dicari. Kompleksitas Ruang: O(m).

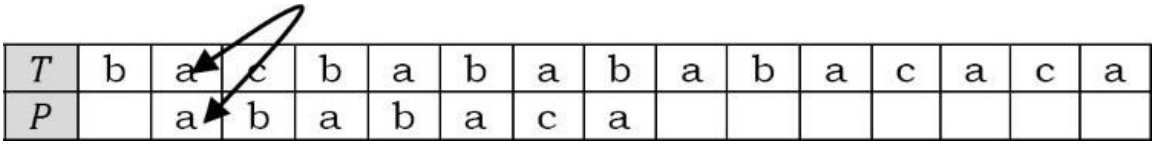
Sekarang, untuk memahami prosesnya, mari kita lihat sebuah contoh. Asumsikan bahwa T = *b a c b a b a b a b a c a c a* & P = *a b a b a c a*. Karena kita telah mengisi tabel awalan, mari kita gunakan dan pergi ke algoritma pencocokan. Awalnya: n = ukuran T = 15; m = ukuran P = 7.

Langkah 1: i = 0, j = 0, membandingkan P[0] dengan T[0]. P[0] tidak cocok dengan T[0]. P akan digeser satu posisi ke kanan.



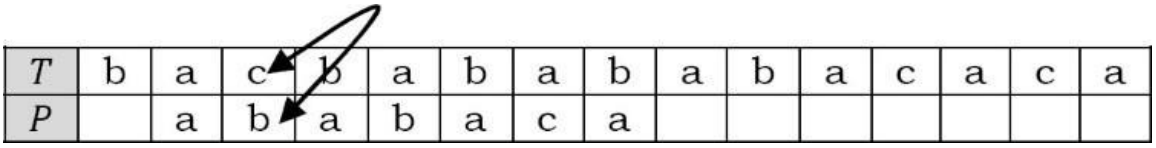
T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P	a	b	a	b	a	c	a								

Langkah 2 :i = 1, j = 0, membandingkan P[0] dengan T[1]. P[0] cocok dengan T[1]. Karena ada kecocokan, P tidak digeser.



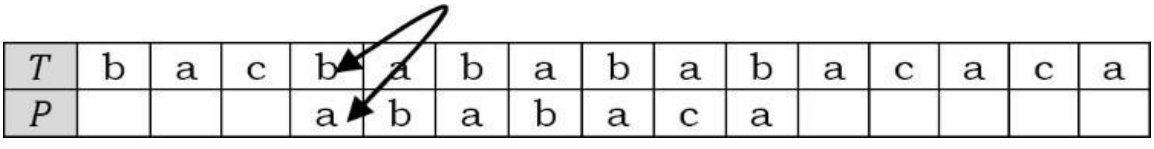
T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P		a	b	a	b	a	c	a							

Langkah 3: i = 2, j = 1, membandingkan P[1] dengan T[2]. P[1] tidak cocok dengan T[2]. Mundur pada P, membandingkan P[0] dan T[2].



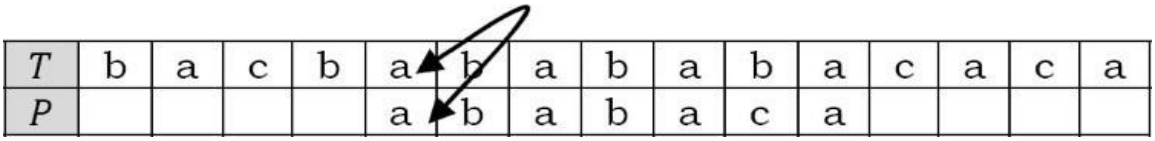
T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P		a	b	a	b	a	c	a							

Langkah 4: i = 3, j = 0, membandingkan P[0] dengan T[3]. P[0] tidak cocok dengan T[3].

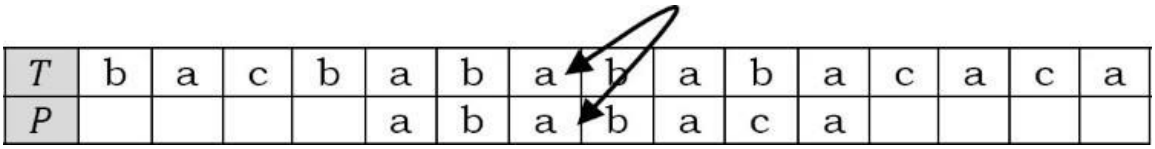


T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P				a	b	a	b	a	c	a					

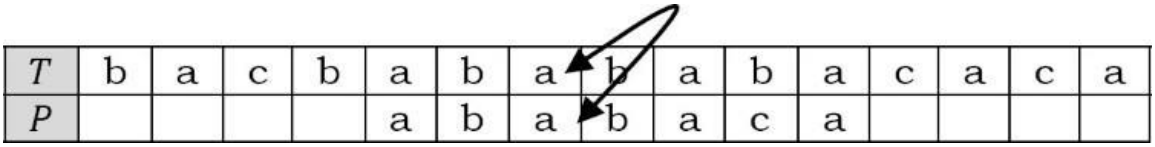
Langkah 5: i = 4, j = 0, membandingkan P[0] dengan T[4]. P[0] cocok dengan T[4].



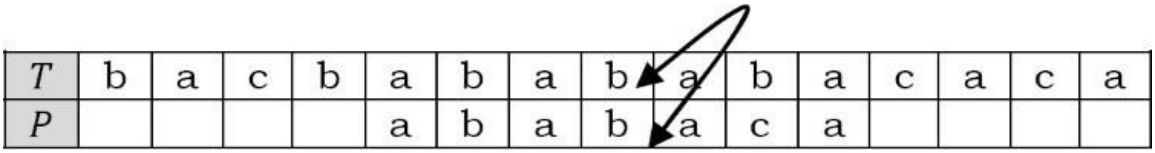
Langkah 6: i = 5, j = 1, membandingkan P[1] dengan T[5]. P[1] cocok dengan T[5].



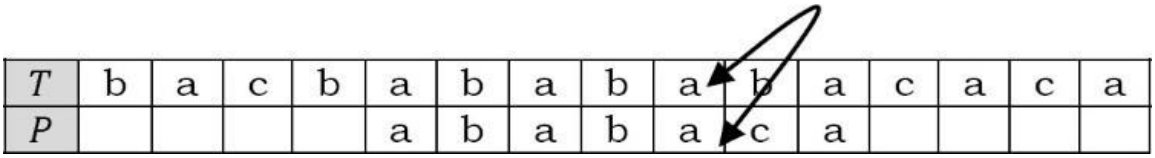
Langkah 7: i = 6, j = 2, membandingkan P[2] dengan T[6]. P[2] cocok dengan T[6].



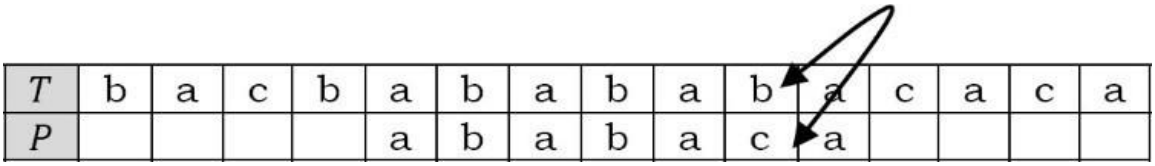
Langkah 8: i = 7, j = 3, membandingkan P[3] dengan T[7]. P[3] cocok dengan T[7].



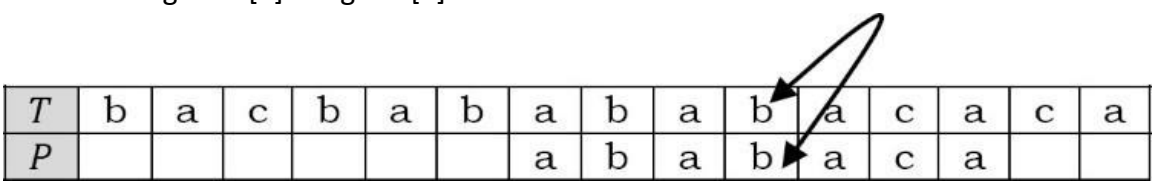
Langkah 9: i = 8, j = 4, membandingkan P[4] dengan T[8]. P[4] cocok dengan T[8].



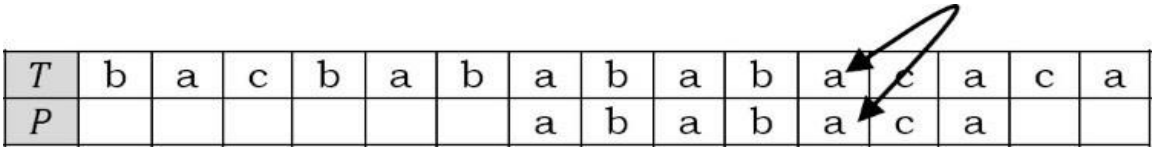
Langkah 10: i = 9, j = 5, membandingkan P[5] dengan T[9]. P[5] tidak cocok dengan T[9].



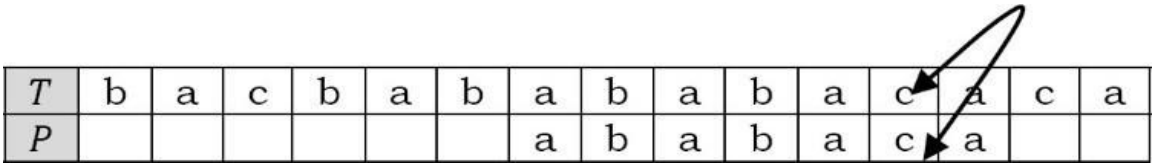
Mundur pada P, membandingkan P[4] dengan T[9] karena setelah mismatch ; = F[4] = 3. Membandingkan P[3] dengan T[9].



Langkah 11: i = 10, j = 4, membandingkan P[4] dengan T[10]. P[4] cocok dengan T[10].

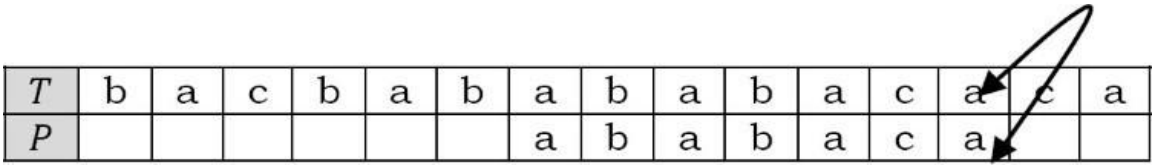


Langkah 12: i = 11, j = 5, membandingkan P[5] dengan T[11]. P[5] cocok dengan T[11].



T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P							a	b	a	b	a	c	a		

Langkah 13: i = 12, j = 6, membandingkan P[6] dengan T[12]. P[6] cocok dengan T[12].



T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P							a	b	a	b	a	c	a		

Pola P telah ditemukan sepenuhnya terjadi pada string T. Jumlah total pergeseran yang terjadi untuk kecocokan yang dapat ditemukan adalah: i – m= 13 – 7 = 6 shift.

Catatan:

- KMP melakukan perbandingan dari kiri ke kanan
- Algoritma KMP membutuhkan preprocessing (fungsi awalan) yang membutuhkan kompleksitas ruang dan waktu O(m)
- Pencarian membutuhkan kompleksitas waktu O(n + m) (tidak tergantung pada ukuran alfabet)

7.7 ALGORITMA BOYER-MOORE

Seperti algoritma KMP, ini juga melakukan beberapa pra-pemrosesan dan kita menyebutnya fungsi terakhir. Algoritma memindai karakter pola dari kanan ke kiri dimulai dengan karakter paling kanan. Selama pengujian kemungkinan penempatan pola P di T, ketidakcocokan ditangani sebagai berikut: Mari kita asumsikan bahwa karakter saat ini yang dicocokkan adalah T[i] = c dan karakter pola yang sesuai adalah P[j]. Jika c tidak terdapat di mana pun dalam P, maka geser pola P sepenuhnya melewati T[i]. Jika tidak, geser P hingga kemunculan karakter c di P sejajar dengan T[i]. Teknik ini menghindari perbandingan yang tidak perlu dengan menggeser pola relatif terhadap teks.

Fungsi terakhir membutuhkan waktu O(m + |∑|) dan pencarian aktual membutuhkan waktu O(nm). Oleh karena itu waktu berjalan kasus terburuk dari algoritma Boyer-Moore adalah O(nm + |∑|). Hal ini menunjukkan bahwa kasus terburuk waktu berjalan adalah kuadrat, dalam kasus n == m, sama dengan algoritma brute force.

- Algoritma Boyer-Moore sangat cepat pada alfabet besar (relatif terhadap panjang pola).
- Untuk alfabet kecil, Boyer-Moore tidak disukai.
- Untuk string biner, algoritma KMP direkomendasikan.
- Untuk pola terpendek, algoritma brute force lebih baik.

7.8 STRUKTUR DATA UNTUK MENYIMPAN STRING

Jika kita memiliki satu set string (misalnya, semua kata dalam kamus) dan kata yang ingin kita cari di set itu, untuk melakukan operasi pencarian lebih cepat, kita membutuhkan cara yang efisien untuk menyimpan string. Untuk menyimpan set string kita dapat menggunakan salah satu dari struktur data berikut.

- Tabel Hash
- Pohon Pencarian Biner
- Mencoba
- Pohon Pencarian Terner

7.9 TABEL HASH UNTUK STRING

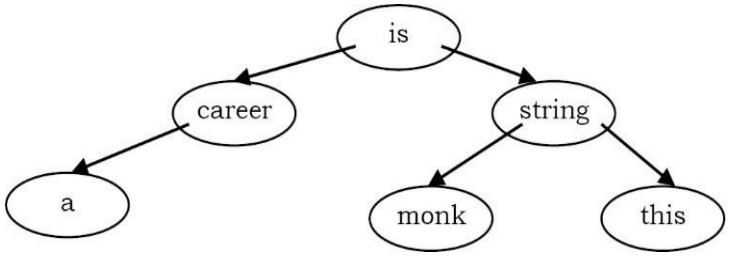
Seperti yang terlihat pada bab Hashing, kita dapat menggunakan tabel hash untuk menyimpan bilangan bulat atau string. Dalam hal ini, kunci tidak lain adalah senar. Masalah dengan implementasi tabel hash adalah bahwa kita kehilangan informasi pemesanan – setelah menerapkan fungsi hash, kita tidak tahu ke mana peta itu akan dipetakan. Akibatnya, beberapa kueri membutuhkan lebih banyak waktu. Misalnya, untuk menemukan semua kata yang dimulai dengan huruf “K”, dengan representasi tabel hash kita perlu memindai tabel hash secara lengkap. Ini karena fungsi hash mengambil kunci lengkap, melakukan hash di atasnya, dan kita tidak tahu lokasi setiap kata.

7.10 POHON PENCARIAN BINER UNTUK STRING

Dalam representasi ini, setiap node digunakan untuk mengurutkan string menurut abjad. Ini dimungkinkan karena string memiliki urutan alami: A datang sebelum B, yang datang sebelum C, dan seterusnya. Ini karena kata dapat diurutkan dan kita dapat menggunakan Binary Search Tree (BST) untuk menyimpan dan mengambilnya. Sebagai contoh, mari kita asumsikan bahwa kita ingin menyimpan string berikut menggunakan BST:

ini adalah string biksu karir

Untuk string yang diberikan ada banyak cara untuk merepresentasikannya dalam BST. Salah satu kemungkinan tersebut ditunjukkan pada pohon di bawah ini.



Gambar 7.2 *binary search tree*

Masalah dengan Representasi Pohon Pencarian Biner

Metode ini bagus dalam hal efisiensi penyimpanan. Tetapi kelemahan dari representasi ini adalah, pada setiap node, operasi pencarian melakukan pencocokan lengkap dari kunci yang diberikan dengan data node, dan sebagai hasilnya kompleksitas waktu dari operasi pencarian meningkat. Jadi, dari sini kita dapat mengatakan bahwa representasi string BST baik dalam hal penyimpanan tetapi tidak dalam hal waktu.

7.11 MENCOBA "TRIE"

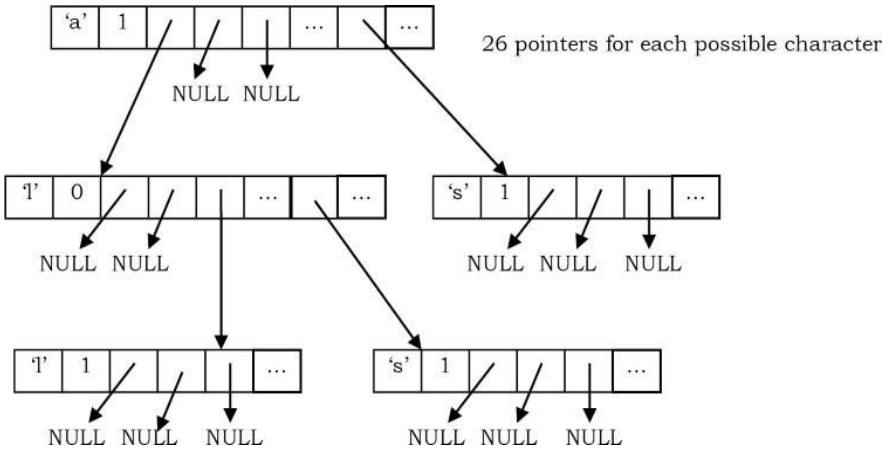
Sekarang, mari kita lihat representasi alternatif yang mengurangi kompleksitas waktu dari operasi pencarian. Nama trie diambil dari kata re"trie".

Apa itu Trie?

Trie adalah pohon dan setiap simpul di dalamnya berisi jumlah pointer sama dengan jumlah karakter alfabet. Misalnya, jika kita berasumsi bahwa semua string dibentuk dengan karakter alfabet bahasa Inggris "a" hingga "z" maka setiap simpul dari trie berisi 26 pointer. Struktur data trie dapat dideklarasikan sebagai:

```
struct TrieNode {
    char data;                  // Contains the current node character.
    int is_End_Of_String;       // Indicates whether the string formed from root to
                                // current node is a string or not
    struct TrieNode *child[26]; // Pointers to other tri nodes
};
```

Misalkan kita ingin menyimpan string "a", "all", "als", dan "as" ": trie untuk string ini akan terlihat seperti:



Gambar 7.3 string trie

Mengapa Trie?

Percobaan dapat menyisipkan dan menemukan string dalam waktu O(L) (di mana L mewakili panjang satu kata). Ini jauh lebih cepat daripada tabel hash dan representasi pohon pencarian biner.

Deklarasi Trie

Struktur TrieNode memiliki data (char), is_End_Of_String (boolean), dan memiliki kumpulan node anak (Collection of TrieNodes). Ini juga memiliki satu metode lagi yang disebut subNode(char). Metode ini mengambil karakter sebagai argumen dan akan mengembalikan simpul anak dari tipe karakter tersebut jika ada. Elemen dasar - TrieNode dari struktur data TRIE terlihat seperti ini:

```
struct TrieNode {
    char data;
    int is_End_Of_String;
    struct TrieNode *child[];
};
struct TrieNode *TrieNode subNode(struct TrieNode *root, char c){
    if(root! = NULL){
        for(int i=0; i < 26; i++){
            if(root.child[i]→data == c)
                return root.child[i];
        }
    }
    return NULL;
}
```

Sekarang kita telah mendefinisikan TrieNode kita, mari kita lanjutkan dan melihat operasi TRIE lainnya. Untungnya, struktur data TRIE mudah diimplementasikan karena memiliki dua metode utama: insert() dan search(). Mari kita lihat implementasi dasar dari kedua metode ini.

Memasukkan String di Trie

Untuk menyisipkan string, kita hanya perlu memulai dari simpul akar dan mengikuti jalur yang sesuai (jalur dari akar menunjukkan awalan dari string yang diberikan). Setelah kita mencapai pointer NULL, kita hanya perlu membuat simpul simpul ekor untuk karakter yang tersisa dari string yang diberikan.

```
void InsertInTrie(struct TrieNode *root, char *word) {
    if(!*word) return;
    if(!root) {
        struct TrieNode *newNode = (struct TrieNode *) malloc (sizeof(struct TrieNode *));
          newNode→data=*word;
          for(int i =0; i<26; i++)
              newNode→child[i]=NULL;
          if(!*(word+1))
              newNode→is_End_Of_String = 1;
          else newNode→child[*word] = InsertInTrie(newNode→child[*word], word+1);
          return newNode;
    }
    root→child[*word] = InsertInTrie(root→child[*word], word+1);
    return  root;
}
```

Kompleksitas Waktu: O(L), di mana L adalah panjang string yang akan disisipkan.

Catatan: Untuk implementasi kamus yang sebenarnya, kita mungkin memerlukan beberapa pemeriksaan lagi seperti memeriksa apakah string yang diberikan sudah ada di kamus atau tidak.

Mencari String di Trie

Sama halnya dengan operasi pencarian: kita hanya perlu memulai dari root dan mengikuti petunjuknya. Kompleksitas waktu operasi pencarian sama dengan panjang string tertentu yang ingin dicari.

```
int SearchInTrie(struct TrieNode *root, char *word) {
    if(!root)
        return -1;
    if(!*word) {
        if(root→ is_End_Of_String)
            return 1;
        else return -1;
    }
    if(root→data == *word)
        return  SearchInTrie(root→child[*word], word+1);
    else  return -1;
}
```

Kompleksitas Waktu: O(L), di mana L adalah panjang string yang akan dicari.

Masalah dengan Representasi Percobaan

Kerugian utama dari percobaan adalah mereka membutuhkan banyak memori untuk menyimpan string. Seperti yang telah kita lihat di atas, untuk setiap node kita memiliki terlalu banyak pointer node. Dalam banyak kasus, hunian setiap node lebih sedikit. Kesimpulan terakhir mengenai struktur data percobaan adalah bahwa mereka lebih cepat tetapi membutuhkan memori yang besar untuk menyimpan string.

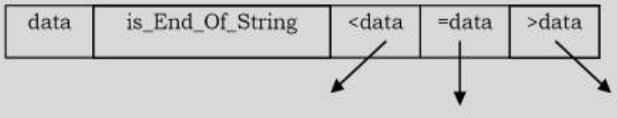
Catatan: Ada beberapa representasi percobaan yang ditingkatkan yang disebut teknik kompresi trie. Tapi, bahkan dengan teknik itu kita bisa mengurangi memori hanya di daun dan bukan di node internal.

7.12 POHON PENCARIAN TERNER

Representasi ini awalnya disediakan oleh Jon Bentley dan Sedgewick. Pohon pencarian ternary mengambil keuntungan dari pohon pencarian biner dan mencoba. Itu berarti menggabungkan efisiensi memori BST dan efisiensi waktu percobaan.

Deklarasi Pohon Pencarian Terner

```
struct TSTNode {
    char data;
    int is_End_Of_String;
    struct TSTNode *left;
    struct TSTNode *eq;
    struct TSTNode *right;
};
```



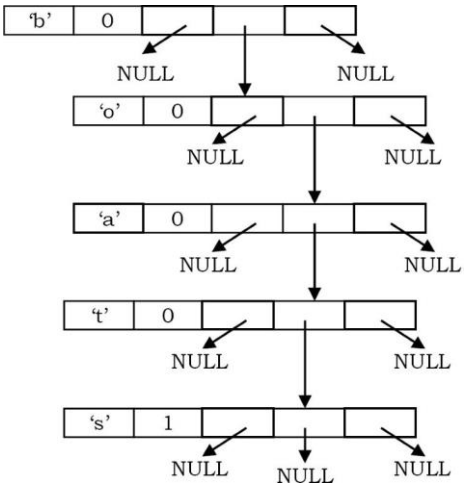
Pohon Pencarian Ternary (TST) menggunakan tiga petunjuk:

- Pointer kiri menunjuk ke TST yang berisi semua string yang menurut abjad kurang dari data.

- Pointer kanan menunjuk ke TST yang berisi semua string yang menurut abjad lebih besar dari data.
- Pointer eq menunjuk ke TST yang berisi semua string yang menurut abjad sama dengan data. Artinya, jika kita ingin mencari string, dan jika karakter string input saat ini dan data node saat ini di TST sama, maka kita perlu melanjutkan ke karakter berikutnya dalam string input dan mencarinya di subpohon yang ditunjuk oleh persamaan.

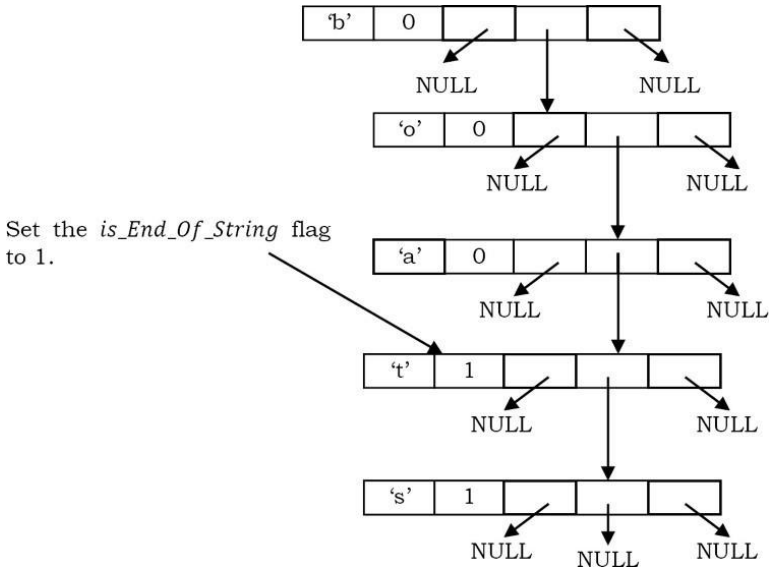
Memasukkan string di Pohon Pencarian Terner

Untuk mempermudah, mari kita asumsikan bahwa kita ingin menyimpan kata-kata berikut dalam TST (juga mengasumsikan urutan yang sama): boats, boat, bat dan bat. Awalnya, mari kita mulai dengan tali perahu.



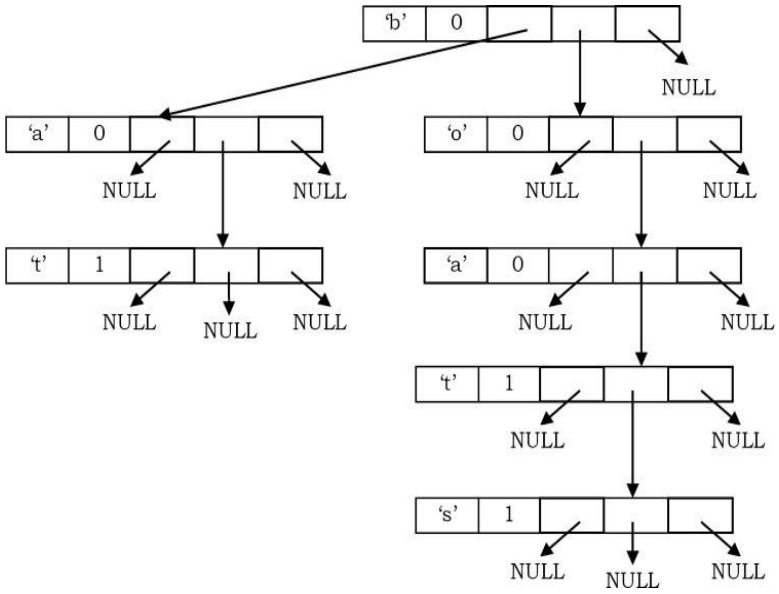
Gambar 7.4 input string ke pohon pencarian biner

Nah jika kita ingin memasukkan string boat, maka TST menjadi [ubahannya hanya setting is_End_Of_String flag dari simpul "t" ke 1]:



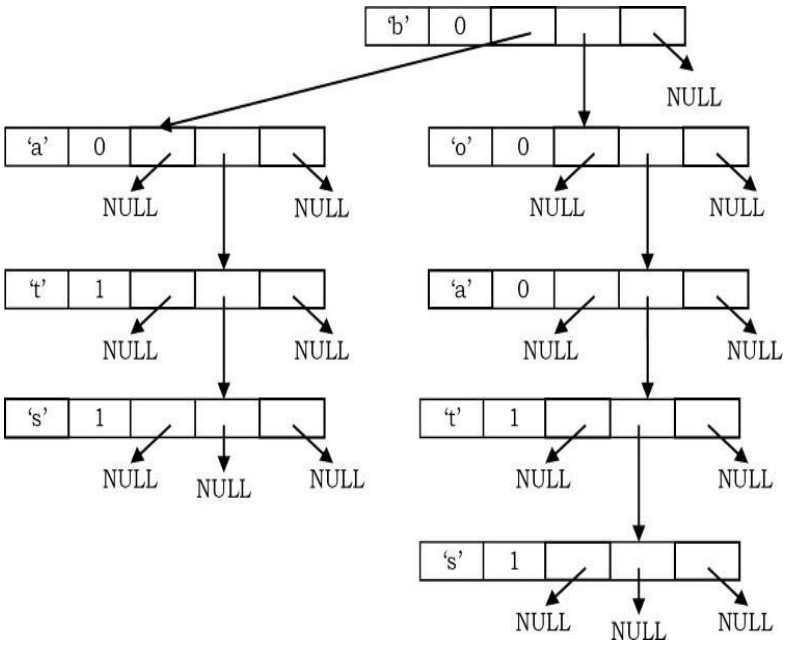
Gambar 7.5 input string boat

Sekarang, mari kita masukkan string berikutnya: bat



Gambar 7.6 input string bat

Sekarang, mari kita masukkan kata terakhir: bat.



Gambar 7.7 input kata terakhir bat

Berdasarkan contoh-contoh ini, kita dapat menulis algoritma penyisipan seperti di bawah ini. Kita akan menggabungkan operasi penyisipan BST dan mencoba.

```
struct TSTNode  *InsertInTST(struct TSTNode  *root, char *word) {
    if(root == NULL) {
            root = (struct TSTNode  *) malloc(sizeof(struct TSTNode));
            root→data = *word;
            root→is_End_Of_String = 1;
            root→left = root→eq = root→right = NULL;
    }
    if(*word < root→data)
            root→left = InsertInTST (root→left, word);
    else if(*word == root→data) {
            if(*(word+1))
                root→eq = InsertInTST (root→eq, word+1);
            else root→is_End_Of_String = 1;
    }
    else   root→right = InsertInTST (root→right, word);
    return root;
}
```

Kompleksitas Waktu: O(L), di mana L adalah panjang string yang akan disisipkan.

Mencari di Pohon Pencarian Terner

Jika setelah memasukkan kata-kata yang ingin kita cari, maka kita harus mengikuti aturan yang sama dengan pencarian biner. Satu-satunya perbedaan adalah, jika cocok, kita harus memeriksa karakter yang tersisa (dalam subpohon eq) alih-alih kembali. Juga, seperti BST kita akan melihat versi rekursif dan non-rekursif dari metode pencarian.

```
int SearchInTSTRecursive(struct TSTNode  *root, char *word) {
    if(!root)
        return -1;
    if(*word < root→data)
        return SearchInTSTRecursive(root→left, word);
    else if(*word > root→data)
        return SearchInTSTRecursive(root→right, word);
    else {
         if(root→is_End_Of_String && *(word+1)==0)
            return 1;
         return SearchInTSTRecursive(root→eq, ++word);
    }
}
int SearchInTSTNon-Recursive(struct TSTNode  *root, char *word) {
    while (root) {
            if(*word < root→data)
                root = root→left;
            else if(*word == root→data) {
                if(root→is_End_Of_String && *(word+1) == 0)
                    return 1;
                word++;
                root = root→eq;
            }
            else root = root→right;
    }
    return -1;
}
```

Kompleksitas Waktu: O(L), di mana L adalah panjang string yang akan dicari.

Menampilkan Al Words of Ternary Search Tree

Jika kita ingin mencetak semua string TST kita dapat menggunakan algoritma berikut. Jika kita ingin mencetaknya dalam urutan terurut, kita harus mengikuti traversal inorder dari TST.

```
char word[1024];
void DisplayAllWords(struct TSTNode  *root) {
    if(!root)
        return;
    DisplayAllWords(root→left);
    word[i] = root→data;
    if(root→is_End_Of_String) {
            word[i] = '\0';
            printf("%c", word);
    }
    i++;
    DisplayAllWords(root→eq);

    i--;
    DisplayAllWords(root→right);
}
```

Menemukan Panjang Kata Terbesar di TST

Ini mirip dengan mencari ketinggian BST dan dapat ditemukan sebagai:

```
int MaxLengthOfLargestWordInTST(struct TSTNode  *root) {
    if(!root)
        return 0;
    return Max(MaxLengthOfLargestWordInTST(root→left),
            MaxLengthOfLargestWordInTST(root→eq)+1,
            MaxLengthOfLargestWordInTST(root→right)));
}
```

7.13 MEMBANDINGKAN BST, TRIE DAN TST

- Tabel hash dan implementasi BST menyimpan string lengkap di setiap node. Akibatnya mereka membutuhkan lebih banyak waktu untuk mencari. Tapi mereka hemat memori.
- TST dapat tumbuh dan menyusut secara dinamis tetapi tabel hash mengubah ukuran hanya berdasarkan faktor beban.
- TST memungkinkan pencarian parsial sedangkan BST dan tabel hash tidak mendukungnya.
- TST dapat menampilkan kata-kata dalam urutan yang diurutkan, tetapi dalam tabel hash kita tidak bisa mendapatkan urutan yang diurutkan.
- Mencoba melakukan operasi pencarian dengan sangat cepat tetapi membutuhkan memori yang besar untuk menyimpan string.

TST menggabungkan keunggulan BST dan Tries. Itu berarti mereka menggabungkan efisiensi memori BST dan efisiensi waktu percobaan

7.14 POHON SUFIKS

Pohon sufiks adalah struktur data penting untuk string. Dengan pohon sufiks kita dapat menjawab pertanyaan dengan sangat cepat. Tetapi ini membutuhkan beberapa pemrosesan awal dan konstruksi pohon sufiks. Meskipun konstruksi pohon sufiks rumit, ia memecahkan banyak masalah terkait string lainnya dalam waktu linier.

Catatan: *Pohon sufiks menggunakan pohon (pohon sufiks) untuk satu string, sedangkan tabel Hash, BST, Tries, dan TST menyimpan satu set string. Itu berarti, pohon sufiks menjawab pertanyaan yang terkait dengan satu string.*

Mari kita lihat terminologi yang kita gunakan untuk representasi ini.

Awalan dan Akhiran

Diberikan string T = T1T2 … Tn, awalan T adalah string T1 ...Ti di mana saya dapat mengambil nilai dari 1 hingga n. Misalnya, jika T = banana, maka awalan dari T adalah: b, ba, ban, bana, banan, banana.

Demikian pula, diberikan string T = T1T2 … Tn, akhiran dari T adalah string Ti ...Tn di mana saya dapat mengambil nilai dari n ke 1. Misalnya, jika T = banana, maka sufiks dari T adalah: a, na, ana, nana, anana, pisang.

Pengamatan

Dari contoh di atas, kita dapat dengan mudah melihat bahwa untuk teks T dan pola P tertentu, masalah pencocokan string yang tepat juga dapat didefinisikan sebagai:

- Temukan sufiks dari T sedemikian sehingga P adalah prefiks dari sufiks ini atau
- Temukan awalan T sedemikian rupa sehingga P adalah akhiran dari awalan ini.

Contoh: Misalkan teks yang akan dicari adalah T = acebkkbac dan polanya menjadi P = kkb. Untuk contoh ini, P adalah awalan dari akhiran kkbac dan juga akhiran dari awalan acebkkb.

Apa itu Pohon Sufiks?

Secara sederhana, pohon sufiks untuk teks T adalah struktur data seperti Trie yang mewakili sufiks dari T. Definisi pohon sufiks dapat diberikan sebagai: Pohon sufiks untuk string karakter n T[1 ...n] adalah pohon berakar dengan properti berikut.

- Sebuah pohon sufiks akan berisi n daun yang diberi nomor dari 1 sampai n
- Setiap simpul internal (kecuali root) harus memiliki minimal 2 anak
- Setiap sisi dalam pohon diberi label oleh substring tak kosong dari T
- Tidak ada dua sisi dari sebuah simpul (sisi anak-anak) yang dimulai dengan karakter yang sama
- Jalur dari akar ke daun mewakili semua sufiks dari T

Konstruksi Pohon Sufiks

Algoritma

1. Misalkan S adalah himpunan semua sufiks dari T. Tambahkan $ ke setiap sufiks.
2. Urutkan sufiks dalam S berdasarkan karakter pertamanya.
3. Untuk setiap kelompok Sc (c):

(i) Jika grup Sc hanya memiliki satu elemen, maka buat simpul daun.

(ii) Jika tidak, temukan prefiks umum terpanjang dari sufiks dalam grup Sc, buat simpul internal, dan lanjutkan secara rekursif dengan Langkah 2, S menjadi himpunan sufiks yang tersisa dari Sc setelah memisahkan awalan umum terpanjang.

Untuk pemahaman yang lebih baik, mari kita lihat sebuah contoh. Biarkan teks yang diberikan menjadi T = tatat. Untuk string ini, berikan nomor untuk masing-masing sufiks.

Tabel 7.1 data suffix

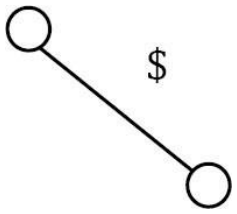
Index	Suffix
1	$
2	*t$*
3	*at$*
4	*tat$*
5	*atat$*
6	*tatat$*

Sekarang, urutkan sufiks berdasarkan karakter awalnya.

Index	Suffix	
1	$	Group S_1 based on a
3	at	Group S_2 based on a
5	$atat$	
2	t	Group S_3 based on t
4	tat	
6	$tatat$	

Gambar 7.8 sufiks berdasarkan karakter

Dalam tiga kelompok, kelompok pertama hanya memiliki satu elemen. Jadi, sesuai Algoritma, buat simpul daun untuknya, seperti yang ditunjukkan di bawah ini.



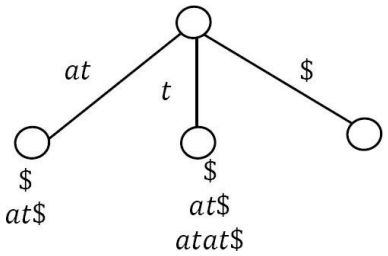
Gambar 7.9 simpul daun

Sekarang, untuk S_2 dan S_3 (karena memiliki lebih dari satu elemen), mari kita cari awalan terpanjang dalam grup, dan hasilnya ditunjukkan di bawah ini.

Tabel 7.2 kelompok S_2 dan S_3

Kelompok	Indeks untuk grup ini	Awalan Terpanjang dari Sufiks Grup
S_2	3, 5	*at*
S_3	2, 4, 6	*t*

Untuk S_2 dan S_3, buat simpul internal, dan tepi berisi awalan umum terpanjang dari grup tersebut.



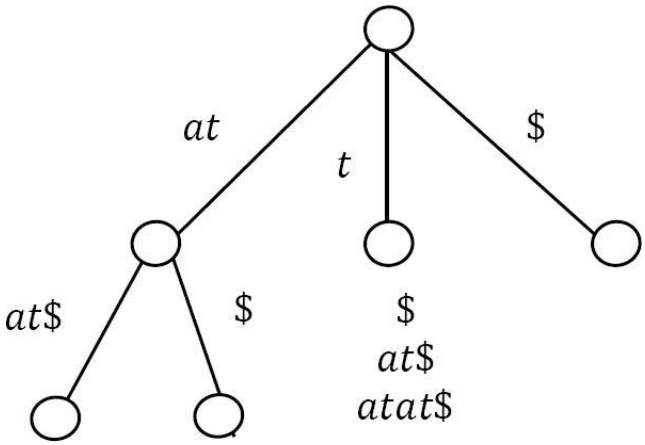
Gambar 7.10 simpul internal

Sekarang kita harus menghapus awalan umum terpanjang dari elemen grup S_2 dan S_3.

Tabel 7.3 kelompok S_2 dan S_3

Kelompok	Indeks untuk grup ini	Awalan Terpanjang dari Sufiks Grup	Sufiks yang dihasilkan
S_2	3, 5	*at*	*$, at $*
S_3	2, 4, 6	*t*	*$, at$, atat$*

Keluar langkah selanjutnya adalah menyelesaikan S_2 dan S_3 secara rekursif. Pertama mari kita ambil S_2. Dalam grup ini, jika kita mengurutkannya berdasarkan karakter pertamanya, mudah untuk melihat bahwa grup pertama hanya berisi satu elemen $, dan grup kedua juga hanya berisi satu elemen, di $. Karena kedua grup hanya memiliki satu elemen, kita dapat langsung membuat simpul daun untuk mereka.



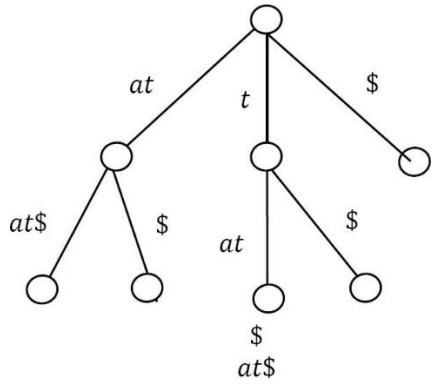
Gambar 7.11 simpul daun

Pada langkah ini, elemen S_1 dan S_2 selesai dan satu-satunya grup yang tersisa adalah S3. Seperti langkah sebelumnya, di grup S_3, jika kita mengurutkannya berdasarkan karakter pertamanya, mudah untuk melihat bahwa hanya ada satu elemen di grup pertama dan itu adalah $. Untuk elemen S3 yang tersisa, hapus awalan umum terpanjang.

Tabel 7.4 data kelompok S_3

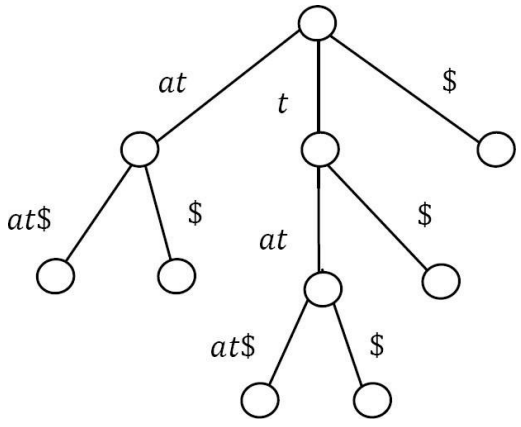
Kelompok	Indeks untuk grup ini	Awalan Terpanjang dari Sufiks Grup	Sufiks yang dihasilkan
S_3	4, 6	*at*	*$, at $*

Di grup kedua S_3, ada dua elemen: $ dan at$. Kita bisa langsung menambahkan simpul daun untuk elemen grup pertama $. Mari kita tambahkan subtree S_3 seperti yang ditunjukkan di bawah ini.



Gambar 7.12 tambah subtree S_3

Sekarang, S_3 berisi dua elemen. Jika kita mengurutkannya berdasarkan karakter pertamanya, mudah untuk melihat bahwa hanya ada dua elemen dan di antaranya adalah $ dan yang lainnya adalah $. Kita bisa langsung menambahkan simpul daun untuk mereka. Mari kita tambahkan subtree S_3 seperti yang ditunjukkan di bawah ini.



Gambar 7.13 tambah subtree S_3

Karena tidak ada elemen lagi, ini adalah penyelesaian konstruksi pohon akhiran untuk string T = tatat. Kompleksitas waktu dari konstruksi pohon sufiks menggunakan algoritma di atas adalah O(n2) di mana n adalah panjang string input karena ada n sufiks yang berbeda. Yang terpanjang memiliki panjang n, terpanjang kedua memiliki panjang n – 1, dan seterusnya.

Catatan:

- Ada algoritma O(n) untuk membangun pohon sufiks.
- Untuk meningkatkan kompleksitas, kita dapat menggunakan indeks sebagai ganti string untuk cabang.

Aplikasi Pohon Sufiks

Semua masalah di bawah ini (tetapi tidak terbatas pada ini) pada string dapat diselesaikan dengan pohon sufiks dengan sangat efisien (untuk Algoritma lihat bagian Masalah).

- Pencocokan String yang Tepat: Diberikan teks T dan pola P, bagaimana kita memeriksa apakah P muncul di T atau tidak?
- Substring Terulang Terpanjang: Diberikan teks T bagaimana kita menemukan substring T yang merupakan substring berulang maksimum?
- Palindrome Terpanjang: Diberikan sebuah teks T bagaimana kita menemukan substring dari T yang merupakan palindrome terpanjang dari T?
- Substring Umum Terpanjang: Diberikan dua string, bagaimana kita menemukan substring umum terpanjang?
- Awalan Umum Terpanjang: Diberikan dua string X[i ...n] dan Y[j ...m], bagaimana kita menemukan awalan umum terpanjang?
- Bagaimana kita mencari ekspresi reguler dalam teks T yang diberikan?
- Diberikan sebuah teks T dan sebuah pola P, bagaimana kita menemukan kemunculan pertama dari P dalam T?

7.15 ALGORITMA STRING: MASALAH & SOLUSI

Soal-1 Diberikan sebuah paragrafik kata, berikan algoritma untuk menemukan kata yang muncul paling banyak. Jika paragrafik di-scroll ke bawah (beberapa kata menghilang dari frame pertama, beberapa kata masih muncul, dan beberapa kata baru), berikan kata yang muncul secara maksimal. Jadi, harus dinamis.

Solusi: Untuk masalah ini kita dapat menggunakan kombinasi antrian prioritas dan percobaan. Kita mulai dengan membuat trie di mana kita memasukkan kata seperti yang muncul, dan di setiap daun trie. Nodenya berisi kata itu bersama dengan pointer yang menunjuk ke node di heap [antrian prioritas] yang juga kita buat. Heap ini berisi node yang strukturnya berisi counter. Ini adalah frekuensinya dan juga penunjuk ke daun trie itu, yang berisi kata itu sehingga tidak perlu menyimpan kata dua kali.

Setiap kali kata baru muncul, kita menemukannya di trie. Jika sudah ada, kita meningkatkan frekuensi simpul itu di heap yang sesuai dengan kata itu, dan kita menyebutnya heapify. Hal ini dilakukan agar setiap saat kita bisa mendapatkan kata frekuensi maksimum. Saat menggulir, ketika sebuah kata keluar dari ruang lingkup, kita mengurangi penghitung di tumpukan. Jika frekuensi baru masih lebih besar dari nol, heapify heap untuk memasukkan modifikasi. Jika frekuensi baru adalah nol, hapus node dari heap dan hapus dari trie.

Soal-2 Diberikan dua string, bagaimana kita dapat menemukan substring umum terpanjang?

Solusi: Mari kita asumsikan bahwa dua string yang diberikan adalah T1 dan T2. Substring umum terpanjang dari dua string, T1 dan T2, dapat ditemukan dengan membangun pohon sufiks umum untuk T1 dan T2. Itu berarti kita perlu membangun pohon sufiks tunggal untuk kedua string. Setiap node ditandai untuk menunjukkan jika itu mewakili akhiran T1 atau T2 atau keduanya. Ini menunjukkan bahwa kita perlu menggunakan penanda yang berbeda simbol untuk kedua string (misalnya, kita dapat menggunakan $ untuk string pertama dan # untuk simbol kedua). Setelah membangun pohon sufiks umum, simpul terdalam yang ditandai untuk T1 dan T2 mewakili substring umum terpanjang.
Cara lain untuk melakukannya adalah: Kita dapat membuat pohon sufiks untuk string T1$T2#. Ini setara dengan membangun pohon sufiks umum untuk kedua string.
Kompleksitas Waktu: O(m + n), di mana m dan n adalah panjang string input T1 dan T2.

Soal-3 Palindrom Terpanjang: Diberikan teks T bagaimana kita menemukan substring dari T yang merupakan palindrom terpanjang dari T?

Solusi: Palindrom terpanjang dari T[1..n] dapat ditemukan dalam waktu O(n). Algoritmanya adalah: pertama buat pohon sufiks untuk T$reverse(T)# atau buat pohon sufiks umum untuk T dan reverse(T). Setelah membangun pohon sufiks, temukan simpul terdalam yang ditandai dengan $ dan #. Pada dasarnya itu berarti menemukan substring umum terpanjang.

Soal-4 Diberikan string (kata), berikan algoritma untuk menemukan kata berikutnya dalam kamus.

Solusi: Mari kita asumsikan bahwa kita menggunakan Trie untuk menyimpan kata-kata kamus. Untuk menemukan kata berikutnya di Tries kita dapat mengikuti

pendekatan sederhana seperti yang ditunjukkan di bawah ini. Mulai dari karakter paling kanan, tambahkan karakter satu per satu. Setelah kita mencapai Z, pindah ke karakter berikutnya di sisi kiri.

Setiap kali kita menambah, periksa apakah kata dengan karakter yang bertambah ada di kamus atau tidak. Jika ada, maka kembalikan kata, jika tidak tambahkan lagi. Jika kita menggunakan TST, maka kita dapat menemukan penerus berurutan untuk kata saat ini.

Soal-5 Berikan algoritma untuk membalik string.

Solusi:

```
//If the str is editable
char *ReversingString(char str[]) {
        char temp, start, end;
        if(str == NULL || *str == '\0')
                return str;
        for (end = 0; str[end]; end++);
        end--;
        for (start = 0; start < end; start++, end--) {
                temp = str[start]; str[start] = str[end]; str[end] = temp;
        }
        return str;
}
```

Kompleksitas Waktu: O(n), di mana n adalah panjang string yang diberikan.

Kompleksitas Ruang: O(n).

Soal-6 Jika string tidak dapat diedit, bagaimana kita membuat string yang merupakan kebalikan dari string yang diberikan?

Solusi: Jika string tidak dapat diedit, maka kita perlu membuat array dan mengembalikan pointernya.

```
//If str is a const string (not editable)
char* ReversingString(char* str) {
        int start, end, len;
        char temp, *ptr=NULL;
        len=strlen(str);
        ptr=malloc(sizeof(char)*(len+1));
        ptr=strcpy(ptr,str);
        for (start=0, end=len-1; start<=end; start++, end--) {        //Swapping
                temp=ptr[start]; ptr[start]=ptr[end]; ptr[end]=temp;
        }
        return ptr;
}
```

Kompleksitas Waktu: O $(\frac{n}{2})$ $\approx$ O (n), di mana n adalah panjang string yang diberikan.

Kompleksitas Ruang: O(1).

Soal-7 Bisakah kita membalikkan string tanpa menggunakan variabel sementara?

Solusi: Ya, kita dapat menggunakan logika XOR untuk menukar variabel.

```
char* ReversingString(char *str) {
	int start = 0, end= strlen(str)-1;
	while( start<end ) {
		str[start] ^= str[end];	str[end] ^= str[start];	str[start] ^= str[end];
		++start;
		--end;
	}
	return str;
}
```

Kompleksitas Waktu: $O\left(\frac{n}{2}\right) \approx O(n)$, di mana n adalah panjang string yang diberikan.

Kompleksitas Ruang: O(1).

Soal-8 Diberikan sebuah teks dan sebuah pola, berikan algoritma untuk mencocokkan pola dalam teks tersebut. Menganggap ? (pencocokan karakter tunggal) dan * (pencocokan multi karakter) adalah karakter wild card.

Solusi: Metode Brute Force. Untuk metode yang efisien, lihat bagian teori.

```
int PatternMatching(char *text, char *pattern) {
	if(*pattern == 0)
		return 1;
	if(*text == 0)
		return *p == 0;
	if('?' == *pattern)
		return PatternMatching(text+1,pattern+1) || PatternMatching(text,pattern+1);
	if('*' == *pattern)
		return PatternMatching(text+1,pattern) || PatternMatching(text,pattern+1);
	if(*text == *pattern)
		return PatternMatching(text+1,pattern+1);
	return -1;
}
```

Kompleksitas Waktu: O(mn), di mana m adalah panjang teks dan n adalah panjang pola.

Kompleksitas Ruang: O(1).

Soal-9 Berikan algoritma untuk membalik kata-kata dalam sebuah kalimat.

Contoh: Input: “Ini adalah String Biksu Karir”, *Output*: “String Biksu Karir a adalah Ini”

Solusi: Mulai dari awal dan terus membalik kata-kata. Implementasi di bawah ini mengasumsikan bahwa '' (spasi) adalah pembatas untuk kata-kata dalam kalimat yang diberikan.

```
void ReverseWordsInSentences(char *text) {
	int wordStart, wordEnd, length;
	length = strlen(text);
	ReversingString(text, 0, length-1);
	for(wordStart = wordEnd = 0; wordEnd < length; wordEnd ++) {
		if(text[wordEnd] ! = ' ') {
```

```
                    wordStart = wordEnd;
                    while (text[wordEnd] != ' ' && wordEnd < length)
                            wordEnd ++;
                    wordEnd--;
                    ReversingString(text, wordStart, wordEnd); //Found current word, reverse it now.
            }
        }
}
void ReversingString(char text[], int start, int end) {
        for (char temp; start < end; start++, end--) {
                temp = str[end];
                str[end] = str[start];
                strstart] = temp;
        }
}
```

Kompleksitas Waktu: O(2n) ≈ O(n), di mana n adalah panjang string.
Kompleksitas Ruang: O(1).

Soal-10 Permutasi string [anagram]: Berikan algoritma untuk mencetak semua kemungkinan permutasi karakter dalam string. Tidak seperti kombinasi, dua permutasi dianggap berbeda jika mengandung karakter yang sama tetapi dalam urutan yang berbeda. Untuk mempermudah, asumsikan bahwa setiap kemunculan karakter yang diulang adalah karakter yang berbeda. Artinya, jika inputnya adalah "aaa", *output*nya harus enam pengulangan "aaa". Permutasi dapat ditampilkan dalam urutan apa pun.

Solusi: Solusinya dicapai dengan membangkitkan n! string, masing-masing panjang n, di mana n adalah panjang string input.

```
void Permutations(int depth, char *permutation, int *used, char *original) {
        int length = strlen(original);
        if(depth == length)
                printf("%s",permutation);
        else {
                for (int i = 0; i < length; i++) {
                        if(!used[i]) {
                                used[i] = 1;
                                permutation[depth] = original[i];
                                Permutations(depth + 1, permutation, used, original);
                                used[i] = 0;
                        }
                }
        }
}
```

Soal-11 Kombinasi Kombinasi String: Tidak seperti permutasi, dua kombinasi dianggap sama jika mengandung karakter yang sama, tetapi mungkin dalam urutan yang berbeda. Berikan algoritma yang mencetak semua kemungkinan kombinasi karakter dalam string. Misalnya, "ac" dan "ab" adalah kombinasi yang berbeda dari string input "abc", tetapi "ab" sama dengan "ba".

Solusi: Solusi dicapai dengan membangkitkan n!/r! (n – r)! string, masing-masing panjangnya antara 1 dan n di mana n adalah panjang string input yang diberikan.

Algoritma:

Untuk setiap karakter input

A. Masukkan karakter saat ini ke dalam string keluaran dan cetak.
B. Jika ada karakter yang tersisa, buat kombinasi dengan karakter yang tersisa.

```
void Combinations(int depth, char *combination, int start, char *original) {
    int length = strlen(original);
    for (int i = start; i < length; i++) {
            combination[depth] = original[i];
          combination[depth +1] = '\0';
            printf("%s", combination);
            if(i < length -1)
              Combinations(depth + 1, combination, start + 1, original);
    }
}
```

Soal-12 Diberikan string "ABCCBCBA", berikan algoritma untuk menghilangkan karakter yang berdekatan secara rekursif jika mereka sama. Misalnya, ABCCBCBA nnnnnn> ABBCBA- >ACBA

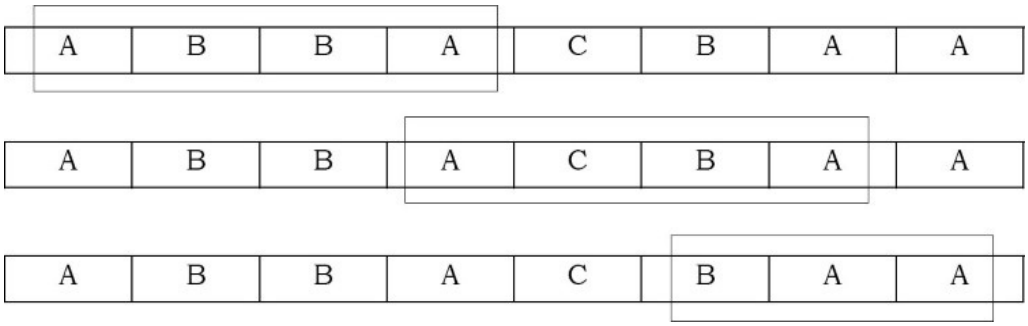
Solusi: Pertama kita perlu memeriksa apakah kita memiliki pasangan karakter; jika ya, maka batalkan. Sekarang periksa karakter berikutnya dan elemen sebelumnya. Terus batalkan karakter hingga kita mencapai awal larik, mencapai akhir larik, atau tidak menemukan pasangan.

```
void RremoveAdjacentPairs(char* str)  {
    int len = strlen(str), i, j = 0;
    for (i=1; i <= len; i++) {
            while ((str[i] == str[j]) && (j >= 0)){          //Cancel pairs
                    i++;
                    j--;
            }
            str[++j] = str[i];
    }
    return;
}
```

Soal-13 Diberikan satu set karakter CHARS dan input string INPUT, temukan jendela minimum di str yang akan berisi semua karakter dalam CHARS dengan kompleksitas O(n). Misalnya, INPUT = ABBACBAA dan CHARS = AAB memiliki jendela BAA minimum.

Solusi: Algoritma ini didasarkan pada pendekatan jendela geser. Dalam pendekatan ini, kita mulai dari awal array dan bergerak ke kanan. Segera setelah kita memiliki jendela yang memiliki semua elemen yang diperlukan, coba geser jendela sejauh mungkin ke kanan dengan semua elemen yang diperlukan. Jika

panjang jendela saat ini kurang dari panjang minimum yang ditemukan hingga sekarang, perbarui panjang minimum. Misalnya, jika array input adalah ABBACBAA dan jendela minimum harus mencakup karakter AAB, maka jendela geser akan bergerak seperti ini:



Algoritma Input adalah array yang diberikan dan chars adalah array karakter yang perlu ditemukan.

1. Buatlah array integer shouldfind[] dari len 256. Elemen ke-i dari array ini akan memiliki hitungan berapa kali kita perlu mencari elemen nilai ASCII i.
2. Buat array lain yang telah ditemukan 256 elemen, yang akan memiliki jumlah elemen yang diperlukan ditemukan sampai sekarang.
3. Hitung <= 0
4. Saat memasukkan[i]
   A. Jika elemen input[i] tidak ditemukan→ lanjutkan
   B. Jika elemen input[i] diperlukan => tingkatkan jumlah sebanyak 1.
   C. Jika count adalah panjang array chars[], geser jendela ke kanan sebanyak mungkin.
   D. Jika panjang jendela saat ini kurang dari panjang min yang ditemukan sampai sekarang, perbarui panjang min.

```
#define MAX 256
void MinLengthWindow(char input[], char chars[]) {
        int shouldfind[MAX] = {0,}, hasfound[MAX] = {0,};
        int j=0, cnt = 0, start=0, finish, minwindow = INT_MAX;
        int charlen = strlen(chars), iplen = strlen(input);
        for (int i=0; i< charlen; i++)
                shouldfind[chars[i]] += 1;
        finish = iplen;
        for (int i=0; i< iplen; i++) {
                if(!shouldfind[input[i]])
                        continue;
                hasfound[input[i]] += 1;
                if(shouldfind[input[i]] >= hasfound[input[i]])
                        cnt++;
                if(cnt == charlen) {
                        while (shouldfind[input[j]] == 0 || hasfound[input[j]] > shouldfind[input[j]])          {
                                if(hasfound[input[j]] > shouldfind[input[j]])
                                        hasfound[input[j]]--;
                                j++;
                        }
```

```
                    if(minwindow > (i - j +1)) {
                        minwindow = i - j +1;
                        finish = i;
                        start = j;
                    }
                }
            }
            printf("Start:%d and Finish: %d", start, finish);
    }
```

Kompleksitas: Jika kita menelusuri kode, i dan j dapat melintasi paling banyak n langkah (di mana n adalah ukuran input) dalam kasus terburuk, menambah total 2n kali. Oleh karena itu, kompleksitas waktu adalah O(n).

Soal-14 Kita diberikan array karakter 2D dan pola karakter. Berikan algoritma untuk menemukan apakah pola ada dalam larik 2D. Polanya bisa dalam urutan apa pun (semua 8 tetangga harus dipertimbangkan) tetapi kita tidak dapat menggunakan karakter yang sama dua kali saat mencocokkan. Kembalikan 1 jika kecocokan ditemukan, 0 jika tidak. Contoh: Temukan “MICROSOFT” pada matriks di bawah ini.

A	C	P	**R**	C
X	**S**	**O**	P	**C**
V	**O**	V	N	**I**
W	G	**F**	**M**	N
Q	A	**T**	I	T

Solusi: Menemukan solusi masalah ini secara manual relatif intuitif; kita hanya perlu menjelaskan algoritma untuk itu. Ironisnya, menggambarkan algoritma bukanlah bagian yang mudah.

Bagaimana kita melakukannya secara manual? Pertama kita mencocokkan elemen pertama, dan ketika dicocokkan, kita mencocokkan elemen kedua di 8 tetangga dari pertandingan pertama. Kita melakukan proses ini secara rekursif, dan ketika karakter terakhir dari pola input cocok, kembalikan true.

Selama proses di atas, berhati-hatilah untuk tidak menggunakan sel apa pun dalam larik 2D dua kali. Untuk tujuan ini, Anda menandai setiap sel yang dikunjungi dengan beberapa tanda. Jika pencocokan pola Anda gagal di beberapa titik, mulailah mencocokkan dari awal (pola) di sel yang tersisa. Saat kembali, Anda menghapus tanda sel yang dikunjungi.

Mari kita ubah metode intuitif di atas menjadi sebuah algoritma. Karena kita melakukan pemeriksaan serupa untuk pencocokan pola setiap saat, solusi rekursif adalah yang kita butuhkan. Dalam solusi rekursif, kita perlu memeriksa apakah substring yang dilewatkan cocok dengan matriks yang diberikan atau tidak. Syaratnya adalah tidak menggunakan sel yang sudah digunakan, dan

untuk menemukan sel yang sudah digunakan, kita perlu menambahkan larik 2D lain ke fungsi (atau kita dapat menggunakan bit yang tidak digunakan dalam larik input itu sendiri.) Juga, kita memerlukan arus posisi matriks input dari mana kita harus mulai. Karena kita perlu meneruskan lebih banyak informasi daripada yang sebenarnya diberikan, kita harus memiliki fungsi pembungkus untuk menginisialisasi informasi tambahan yang akan diteruskan.

Algoritma:

Jika kita melewati karakter terakhir dalam pola

Kembalikan benar

Jika kita mendapatkan sel bekas lagi

Kembalikan salah jika kita berhasil melewati matriks 2D

Kembalikan salah

Jika mencari elemen dan sel pertama tidak cocok

FindMatch dengan sel berikutnya dalam urutan baris-pertama (atau urutan kolom-pertama)

Jika tidak, jika karakter cocok,

tandai sel ini sebagai digunakan

res = FindMatch dengan posisi pola berikutnya di 8 tetangga

menandai sel ini sebagai tidak digunakan

Kembalikan res

```
#define MAX 100
boolean FindMatch_wrapper(char mat[MAX][MAX], char *pat, int nrow, int ncol) {
      if(strlen(pat) > nrow*ncol) return false;
      int used[MAX][MAX] = {{0,},};
      return FindMatch(mat, pat, used, 0, 0, nrow, ncol, 0);
}
//level: index till which pattern is matched & x, y: current position in 2D array
boolean FindMatch(char mat[MAX][MAX], char *pat, int used[MAX][MAX],
                                        int x, int y, int nrow, int ncol, int level) {
   if(level == strlen(pat)) //pattern matched
      return true;
   if(nrow == x || ncol == y)   return false;
   if(used[x][y])         return false;
   if(mat[x][y] != pat[level] && level == 0) {
      if(x < (nrow - 1))
         return FindMatch(mat, pat, used, x+1, y, nrow, ncol, level); //next element in same row
      else if(y < (ncol - 1))
         return FindMatch(mat, pat, used, 0, y+1, nrow, ncol, level); //first element from same column
      else  return false;
   }
   else if(mat[x][y] == pat[level]) {
      boolean res;
      used[x][y] = 1;       //marking this cell as used
```

```
    //finding subpattern in 8 neighbors
    res = (x > 0  ? FindMatch(mat, pat, used, x-1, y, nrow, ncol, level+1) :   false) ||
       (res = x < (nrow - 1)  ? FindMatch(mat, pat, used, x+1, y, nrow, ncol, level+1) :   false) ||
       (res = y > 0 ? FindMatch(mat, pat, used, x, y-1, nrow, ncol, level+1) :   false) ||
       (res = y < (ncol - 1)  ? FindMatch(mat, pat, used, x, y+1, nrow, ncol, level+1) :   false) ||
        (res = x < (nrow - 1)&&(y < ncol -1)?FindMatch(mat, pat, used, x+1, y+1, nrow, ncol, level+1) :  false) ||
       (res = x < (nrow - 1) && y > 0  ?  FindMatch(mat, pat, used, x+1, y-1, nrow, ncol, level+1) : false) ||
       (res = x > 0 && y < (ncol - 1) ?    FindMatch(mat, pat, used, x-1, y+1, nrow, ncol, level+1) : false) ||
       (res = x > 0 && y > 0 ? FindMatch(mat, pat, used, x-1, y-1, nrow, ncol, level+1) : false);
    used[x][y] = 0;          //marking this cell as unused
    return res;
  }
  else  return false;
}
```

Soal-15 Diberikan dua string str1 dan str2, tulislah sebuah fungsi yang mencetak semua sisipan dari dua string yang diberikan. Kita dapat berasumsi bahwa semua karakter di kedua string berbeda. Contoh: Input: str1 = “AB”, str2 = “CD” dan *Output*: ABCD ACBD ACDB CABD CADB CDAB. Sebuah string yang disisipkan dari dua string yang diberikan mempertahankan urutan karakter dalam string individu. Misalnya, dalam semua interleaving dari contoh pertama di atas, 'A' datang sebelum 'B' dan 'C datang sebelum 'D'.

Solusi: Biarkan panjang str1 menjadi m dan panjang str2 menjadi n. Mari kita asumsikan bahwa semua karakter dalam str1 dan str2 berbeda. Biarkan Count(m,n) menjadi jumlah semua string yang disisipkan dalam string tersebut. Nilai Count(m,n) dapat ditulis sebagai berikut.

```
Count(m, n) = Count(m-1, n) + Count(m, n-1)
Count(1, 0) = 1 and Count(1, 0) = 1
```

Untuk mencetak semua interleaving, pertama-tama kita dapat memperbaiki karakter pertama dari strl[0..m-1] dalam string *output*, dan secara rekursif memanggil str1[1..m-1] dan str2[0..n-1]. Dan kemudian kita dapat memperbaiki karakter pertama dari str2[0..n-1] dan secara rekursif memanggil str1[0..m-1] dan str2[1..n-1].

```
void PrintInterleavings(char *str1, char *str2, char *iStr, int m, int n, int i){
   // Base case: If all characters of str1 & str2 have been included in output string,
   // then print the output string
   if ( m==0 && n ==0 )
      printf("%s\n", iStr) ;
   // If some characters of str1 are left to be included, then include the
   // first character from the remaining characters and recur for rest
   if ( m != 0 )   {
      iStr[i] = str1[0];
      PrintInterleavings(str1 + 1, str2, iStr, m-1, n, i+1);
   }
   // If some characters of str2 are left to be included, then include the
   // first character from the remaining characters and recur for rest
   if ( n != 0 ) {
      iStr[i] = str2[0];
      PrintInterleavings(str1, str2+1, iStr, m, n-1, i+1);
   }
}
```

```
// Allocates memory for output string and uses PrintInterleavings() for printing all interleaving's
void Print(char *str1, char *str2, int m, int n){
   // allocate memory for the output string
   char *iStr= (char*)malloc((m+n+1)*sizeof(char));
         // Set the terminator for the output string
   iStr[m+n] = '\0';
   // print all interleaving's using PrintInterleavings()
   PrintInterleavings(str1, str2, iStr, m, n, 0);
   free(iStr);
}
```

Soal-16 Diberikan sebuah matriks dengan ukuran n × n yang berisi bilangan bulat acak. Berikan algoritma yang memeriksa apakah baris cocok dengan kolom atau tidak. Misalnya, jika baris ke-i cocok dengan kolom ke-j, dan baris ke-i berisi elemen - [2,6,5,8,9]. Kemudian;''1 kolom juga akan berisi elemen - [2,6,5,8,9].

Solusi: Kita dapat membangun sebuah trie untuk data di kolom (baris juga akan berfungsi). Kemudian kita dapat membandingkan baris dengan trie. Ini akan memungkinkan kita untuk keluar segera setelah awal baris tidak cocok dengan kolom mana pun (mundur). Juga ini akan memungkinkan kita memeriksa baris terhadap semua kolom dalam satu lintasan.

Jika kita tidak ingin membuang memori untuk pointer kosong maka kita dapat lebih meningkatkan solusi dengan membangun pohon sufiks.

Soal-17 Tulis metode untuk mengganti semua spasi dalam string dengan '%20'. Asumsikan string memiliki ruang yang cukup di akhir string untuk menampung karakter tambahan.

Solusi: Temukan jumlah spasi. Kemudian, mulai dari akhir (dengan asumsi string memiliki cukup ruang), ganti karakter. Mulai dari akhir mengurangi penimpaan.

```
void encodeSpaceWithString(char* A){
    char *space = "%20";
    int stringLength = strlen(A);
    if(stringLength ==0){
        return;
    }
    int i, numberOfSpaces = 0;
    for(i = 0; i < stringLength; i++){
        if(A[i] == ' ' || A[i] == '\t'){
            numberOfSpaces ++;
        }
    }
    if(!numberOfSpaces)
        return;
    int newLength = len + numberOfSpaces * 2;
    A[newLength] = '\0';
    for(i = stringLength-1; i >= 0; i--){
        if(A[i] == ' ' || A[i] == '\t'){
            A[newLength--] = '0';
            A[newLength--] = '2';
            A[newLength--] = '%';
        }
        else{
            A[newLength--] = A[i];
        }
    }
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1). Di sini, kita tidak perlu khawatir tentang ruang yang dibutuhkan untuk karakter tambahan.

Soal-18 Pengkodean panjang berjalan: Tulis algoritma untuk mengompresi string yang diberikan dengan menggunakan hitungan karakter berulang dan jika panjang string baru yang ditekan jagung tidak lebih kecil dari string asli, kembalikan string asli.

Solusi:

```
string CompressString(string inputStr){
        char last = inputStr.at(0);
        int size = 0, count = 1;
        char temp[2];
        string str;
        for (int i = 1; i < inputStr.length(); i++){
                if(last == inputStr.at(i))
                        count ++;
                else{
                        itoa(count, temp, 10);
                        str += last;
                        str += temp;
                        last = inputStr.at(i);
                        count = 1;
                }
        }
```

```
        str = str + last + temp;
        // If the compresed string size is greater than input string, return input string
        if(str.length() >= inputStr.length())
                return inputStr;
        else return str;
}
```

Dengan ruang ekstra O(2):

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1), tetapi menggunakan array sementara berukuran dua.

Tanpa ruang ekstra (di tempat):

```
char CompressString(char *inputStr, char currentChar, int lengthIndex, int& countChar, int& index){
        if(lengthIndex == -1)
                return currentChar;
        char lastChar = CompressString(inputStr, inputStr[lengthIndex], lengthIndex-1, countChar, index);
        if(lastChar == currentChar)
                countChar++;
        else {
                inputStr[index++] = lastChar;
                for(int i =0; i< NumToString(countChar).length(); i++)
                        inputStr[index++] = NumToString(countChar).at(i);
                countChar = 1;
        }
        return currentChar;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1).

BAB 8
TEKNIK DESAIN ALGORITMA

8.1 PENDAHULUAN

Dalam bab-bab sebelumnya, kita telah melihat banyak algoritma untuk memecahkan berbagai jenis masalah. Sebelum memecahkan masalah baru, kecenderungan umum adalah mencari kesamaan masalah saat ini dengan masalah lain yang kita miliki solusinya. Ini membantu kita dalam mendapatkan solusi dengan mudah.

Dalam bab ini, kita akan melihat cara yang berbeda untuk mengklasifikasikan algoritma dan dalam bab berikutnya kita akan fokus pada beberapa di antaranya (*Greedy*, *Divide and conquer*, Dynamic Programming).

8.2 KLASIFIKASI

Ada banyak cara untuk mengklasifikasikan algoritma dan beberapa di antaranya ditunjukkan di bawah ini:

- Metode Implementasi
- Metode Desain
- Klasifikasi Lainnya

8.3 KLASIFIKASI BERDASARKAN METODE IMPLEMENTASI

Sebuah algoritma rekursif adalah salah satu yang memanggil dirinya sendiri berulang kali sampai kondisi dasar terpenuhi. Ini adalah metode umum yang digunakan dalam bahasa pemrograman fungsional seperti C, C ++, dll. Algoritma iteratif menggunakan konstruksi seperti loop dan terkadang struktur data lain seperti tumpukan dan antrian untuk menyelesaikan masalah.

Beberapa masalah cocok untuk rekursif dan yang lain cocok untuk iteratif. Misalnya, masalah Towers of Hanoi dapat dengan mudah dipahami dalam implementasi rekursif. Setiap versi rekursif memiliki versi iteratif, dan sebaliknya.

Prosedural atau Deklaratif (non-Prosedural)

Dalam bahasa pemrograman deklaratif, kita mengatakan apa yang kita inginkan tanpa harus mengatakan bagaimana melakukannya. Dengan pemrograman prosedural, kita harus menentukan langkah-langkah yang tepat untuk mendapatkan hasilnya. Misalnya, SQL lebih bersifat deklaratif daripada prosedural, karena kueri tidak menentukan langkah-langkah untuk menghasilkan hasil. Contoh bahasa prosedural antara lain: C, PHP, dan PERL.

Serial atau Paralel atau Didistribusikan

Secara umum, saat membahas Algoritma, kita berasumsi bahwa komputer menjalankan satu instruksi pada satu waktu. Ini disebut algoritma serial. Algoritma paralel

memanfaatkan arsitektur komputer untuk memproses beberapa instruksi sekaligus. Mereka membagi masalah menjadi submasalah dan menyajikannya ke beberapa prosesor atau utas. Algoritma iteratif umumnya dapat diparalelkan. Jika algoritma paralel didistribusikan ke mesin yang berbeda maka kita sebut algoritma tersebut algoritma terdistribusi.

Deterministik atau Non-Deterministik

Algoritma deterministik memecahkan masalah dengan proses yang telah ditentukan, sedangkan algoritma non-deterministik menebak solusi terbaik pada setiap langkah melalui penggunaan heuristik.

Tepat atau Perkiraan

Seperti yang telah kita lihat, untuk banyak masalah kita tidak dapat menemukan solusi yang optimal. Artinya, Algoritma di mana kita dapat menemukan solusi optimalnya disebut Algoritma eksak. Dalam ilmu komputer, jika kita tidak memiliki solusi optimal, kita memberikan algoritma aproksimasi. Algoritma aproksimasi umumnya terkait dengan masalah NP-hard (lihat bab Kelas Kompleksitas untuk lebih jelasnya).

8.4 KLASIFIKASI BERDASARKAN METODE DESAIN

Cara lain untuk mengklasifikasikan algoritma adalah dengan metode desainnya.

Metode Tamak

Algoritma tamak bekerja secara bertahap. Di setiap tahap, keputusan dibuat dengan baik pada saat itu, tanpa memikirkan konsekuensi masa depan. Secara umum, ini berarti bahwa beberapa lokal terbaik dipilih. Diasumsikan bahwa pemilihan lokal terbaik juga menghasilkan solusi optimal global.

Memecah dan menaklukkan

Strategi D & C memecahkan masalah dengan:

1) Divide: Memecah masalah menjadi sub-sub masalah yang merupakan contoh yang lebih kecil dari jenis masalah yang sama.
2) Rekursi: Memecahkan sub masalah ini secara rekursif.
3) Conquer: Menggabungkan jawaban mereka dengan tepat.

Contoh: merge sort dan algoritma pencarian biner.

Pemrograman Dinamis

Pemrograman dinamis (DP) dan memoisasi bekerja sama. Perbedaan antara DP dan *Divide and conquer* adalah pada kasus yang terakhir tidak ada ketergantungan antar sub masalah, sedangkan pada DP akan terjadi tumpang tindih sub masalah. Dengan menggunakan memoisasi [mempertahankan tabel untuk sub masalah yang sudah diselesaikan], DP mengurangi kompleksitas eksponensial menjadi polynomial kompleksitas (O(n2), O(n3), dll.) untuk banyak masalah.

Perbedaan antara pemrograman dinamis dan rekursi adalah dalam memoisasi panggilan rekursif. Ketika sub masalah independen dan jika tidak ada pengulangan, memoisasi

tidak membantu, maka pemrograman dinamis bukanlah solusi untuk semua masalah. Dengan menggunakan memoisasi [mempertahankan tabel sub masalah yang sudah diselesaikan], pemrograman dinamis mengurangi kompleksitas dari eksponensial ke polinomial.

Pemrograman Linier

Dalam pemrograman linier, ada ketidaksetaraan dalam hal input dan memaksimalkan (atau meminimalkan) beberapa fungsi linier dari input. Banyak masalah (contoh: aliran maksimum untuk grafik berarah) dapat didiskusikan dengan menggunakan program linier.

Pengurangan [Transformasi dan Taklukkan]

Dalam metode ini, kita memecahkan masalah yang sulit dengan mengubahnya menjadi masalah yang diketahui yang memiliki algoritma optimal asimtotik. Dalam metode ini, tujuannya adalah untuk menemukan algoritma pereduksi yang kompleksitasnya tidak didominasi oleh algoritma reduksi yang dihasilkan. Misalnya, Algoritma pemilihan untuk menemukan median dalam daftar melibatkan pengurutan daftar terlebih dahulu dan kemudian menemukan elemen tengah dalam daftar yang diurutkan. Teknik ini juga disebut mengubah dan menaklukkan.

8.5 KLASIFIKASI LAINNYA

Dalam ilmu komputer setiap bidang memiliki masalah sendiri dan membutuhkan algoritma yang efisien. Contoh: algoritma pencarian, algoritma pengurutan, algoritma penggabungan, algoritma numerik, algoritma Grafik, algoritma string, algoritma geometris, algoritma kombinatorial, pembelajaran mesin, kriptografiki, algoritma paralel, algoritma kompresi data, teknik parsing, dan banyak lagi.

Klasifikasi berdasarkan Kompleksitas

Dalam klasifikasi ini, algoritma diklasifikasikan berdasarkan waktu yang dibutuhkan untuk menemukan solusi berdasarkan ukuran inputnya. Beberapa algoritma mengambil kompleksitas waktu linier (O(n)) dan yang lain membutuhkan waktu eksponensial, dan beberapa tidak pernah berhenti. Perhatikan bahwa beberapa masalah mungkin memiliki beberapa algoritma dengan kompleksitas yang berbeda.

Algoritma Acak

Beberapa algoritma membuat pilihan secara acak. Untuk beberapa masalah, solusi tercepat harus melibatkan keacakan. Contoh: Sortir Cepat.

Pencacahan Cabang dan Terikat dan Pelacakan Balik

Ini digunakan dalam Kecerdasan Buatan dan kita tidak perlu mengeksplorasi ini sepenuhnya. Untuk metode Backtracking lihat bab Recusion dan Backtracking.

Catatan: *Dalam beberapa bab berikutnya kita membahas metode desain Greedy, Divide and conquer, dan Dynamic Programming]. Metode-metode ini ditekankan karena lebih sering digunakan daripada metode-metode lain untuk memecahkan masalah.*

BAB 9
ALGORITMA *GREEDY*

9.1 PENDAHULUAN

Mari kita mulai diskusi kita dengan teori sederhana yang akan memberi kita pemahaman tentang teknik *Greedy*. Dalam permainan Catur, setiap kali kita membuat keputusan tentang suatu langkah, kita juga harus memikirkan konsekuensi di masa depan. Padahal, dalam permainan Tenis (atau Bola Voli), tindakan kita didasarkan pada situasi langsung.

Ini berarti bahwa dalam beberapa kasus membuat keputusan yang terlihat tepat pada saat itu memberikan solusi terbaik (*Greedy*), tetapi dalam kasus lain tidak. Teknik *Greedy* paling cocok untuk melihat situasi langsung.

9.2 STRATEGI *GREEDY*

Algoritma tamak bekerja secara bertahap. Di setiap tahap, keputusan dibuat dengan baik pada saat itu, tanpa memikirkan masa depan. Ini berarti bahwa beberapa lokal terbaik dipilih. Diasumsikan bahwa seleksi lokal yang baik menghasilkan solusi optimal global.

9.3 ELEMEN ALGORITMA *GREEDY*

Dua sifat dasar dari algoritma *Greedy* yang optimal adalah:

1) Properti pilihan tamak
2) Substruktur yang optimal

Properti pilihan *greedy*

Sifat ini menyatakan bahwa solusi optimal global dapat diperoleh dengan membuat solusi optimal lokal (*Greedy*). Pilihan yang dibuat oleh algoritma *Greedy* mungkin bergantung pada pilihan sebelumnya tetapi tidak pada masa depan. Itu secara iteratif membuat satu pilihan *Greedy* demi satu dan mengurangi masalah yang diberikan menjadi yang lebih kecil.

Substruktur yang optimal

Suatu masalah menunjukkan substruktur optimal jika solusi optimal untuk masalah tersebut berisi solusi optimal untuk submasalah. Itu berarti kita dapat memecahkan submasalah dan membangun solusi untuk memecahkan masalah yang lebih besar.

9.4 APAKAH *GREEDY* SELALU BERHASIL?

Membuat pilihan lokal yang optimal tidak selalu berhasil. Oleh karena itu, algoritma *Greedy* tidak akan selalu memberikan solusi terbaik. Kita akan melihat contoh khusus di bagian Masalah dan di bab Pemrograman Dinamis.

9.5 KEUNTUNGAN DAN KERUGIAN DARI METODE *GREEDY*

Keuntungan utama dari metode *Greedy* adalah mudah, mudah dimengerti dan mudah dikodekan. Dalam algoritma *Greedy*, begitu kita membuat keputusan, kita tidak perlu menghabiskan waktu untuk memeriksa kembali nilai-nilai yang sudah dihitung. Kerugian utamanya adalah bahwa untuk banyak masalah tidak ada algoritma tamak. Artinya, dalam banyak kasus tidak ada jaminan bahwa melakukan perbaikan optimal lokal dalam solusi optimal lokal memberikan solusi global optimal.

9.6 APLIKASI *GREEDY*

- Sorting: Sortir seleksi, Sortir topologi
- Antrian Prioritas: Heap sort
- Algoritma kompresi pengkodean Huffman
- Algoritma Prim dan Kruskal
- Jalur terpendek dalam Grafik Berbobot [Dijkstra's]
- Masalah penukaran koin
- Masalah Ransel Pecahan
- Himpunan terpisah-UNION menurut ukuran dan UNION menurut tinggi (atau peringkat)
- Algoritma penjadwalan pekerjaan
- Teknik tamak dapat digunakan sebagai Algoritma aproksimasi untuk masalah yang kompleks

9.7 MEMAHAMI TEKNIK *GREEDY*

Untuk pemahaman yang lebih baik, mari kita lihat sebuah contoh.

Algoritma Pengkodean Huffman

Diberikan himpunan n karakter dari abjad A [setiap karakter c A] dan frekuensi terkaitnya freq(c), temukan kode biner untuk setiap karakter c A, sehingga c A freq(c)|kode biner(c)| adalah minimum, di mana /binarycode(c)/mewakili panjang kode biner karakter c. Itu berarti jumlah panjang semua kode karakter harus minimum [jumlah frekuensi setiap karakter dikalikan dengan jumlah bit dalam representasi].

Ide dasar di balik algoritma pengkodean Huffman adalah menggunakan lebih sedikit bit untuk karakter yang lebih sering muncul. Algoritma pengkodean Huffman memampatkan penyimpanan data menggunakan kode panjang variabel. Kita tahu bahwa setiap karakter membutuhkan 8 bit untuk representasi. Namun secara umum, kita tidak menggunakan semuanya. Juga, kita menggunakan beberapa karakter lebih sering daripada yang lain. Saat membaca file, sistem umumnya membaca 8 bit sekaligus untuk membaca satu karakter. Tetapi skema pengkodean ini tidak efisien. Alasan untuk ini adalah bahwa beberapa karakter lebih sering digunakan daripada karakter lain. Katakanlah karakter e’ digunakan 10 kali lebih sering

daripada karakter q. Akan lebih baik jika kita menggunakan kode 7 bit untuk e dan kode 9 bit untuk q karena itu dapat mengurangi panjang pesan kita secara keseluruhan.

Rata-rata, menggunakan pengkodean Huffman pada file standar dapat menguranginya dari 10% hingga 30% tergantung pada frekuensi karakter. Gagasan di balik pengkodean karakter adalah untuk memberikan kode biner yang lebih panjang untuk karakter dan kelompok karakter yang lebih jarang. Juga, pengkodean karakter dibangun sedemikian rupa sehingga tidak ada dua kode karakter yang merupakan awalan satu sama lain.

Sebuah contoh

Mari kita asumsikan bahwa setelah memindai file, kita menemukan frekuensi karakter berikut:

Tabel 9.1 data karakter dan frekuensi

Karakter	***Frekuensi***
a	12
b	2
c	7
d	13
e	14
f	85

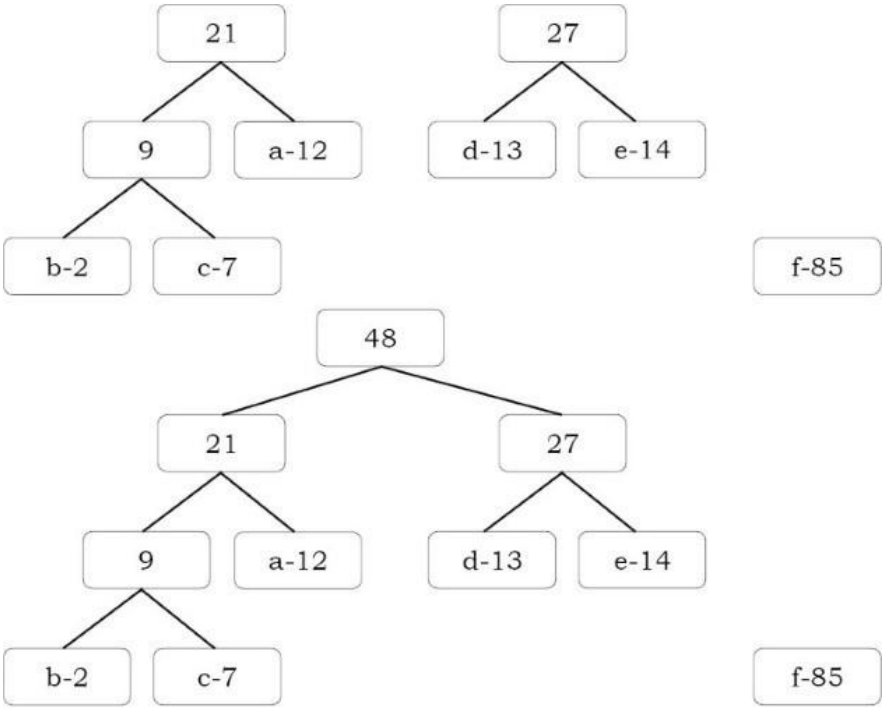
Dengan ini, buat pohon biner untuk setiap karakter yang juga menyimpan frekuensi kemunculannya (seperti yang ditunjukkan di bawah).

Gambar 9.1 kemunculan frekuensi

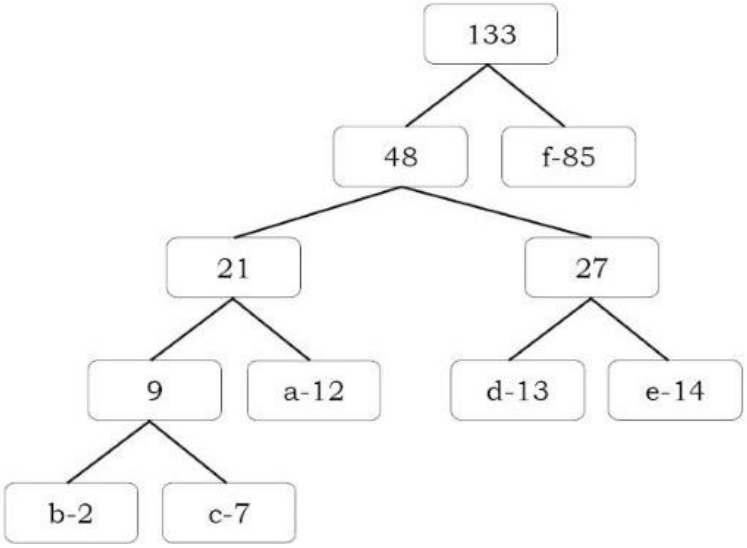
Algoritma bekerja sebagai berikut: Dalam daftar, temukan dua pohon biner yang menyimpan frekuensi minimum pada simpulnya. Hubungkan dua node ini pada node umum yang baru dibuat yang tidak akan menyimpan karakter tetapi akan menyimpan jumlah frekuensi dari semua node yang terhubung di bawahnya. Jadi gambar kita terlihat seperti ini:

Gambar 9.2 pohon biner

Ulangi proses ini sampai hanya satu pohon yang tersisa:



Gambar 9.3 pohon biner



Gambar 9.4 pohon biner

Setelah pohon dibangun, setiap simpul daun sesuai dengan huruf dengan kode. Untuk menentukan kode untuk simpul tertentu, telusuri dari akar ke simpul daun. Untuk setiap gerakan ke kiri, tambahkan 0 ke kode, dan untuk setiap gerakan ke kanan, tambahkan 1. Hasilnya, untuk pohon yang dihasilkan di atas, kita mendapatkan kode berikut:

Tabel 9.2 data surat dan kode

Surat	Kode
a	001
b	0000
c	0001
d	010
e	011
f	1

Menghitung Bit yang Disimpan

Sekarang, mari kita lihat berapa banyak bit yang disimpan oleh algoritma pengkodean Huffman. Yang perlu kita lakukan untuk perhitungan ini adalah melihat berapa banyak bit yang awalnya digunakan untuk menyimpan data dan mengurangi jumlah bit yang digunakan untuk menyimpan data menggunakan kode Huffman. Dalam contoh di atas, karena kita memiliki enam karakter, mari kita asumsikan setiap karakter disimpan dengan kode tiga bit. Karena ada 133 karakter seperti itu (kalikan total frekuensi dengan 3), jumlah total bit yang digunakan adalah 3 * 133 = 399. Dengan menggunakan frekuensi pengkodean Huffman, kita dapat menghitung jumlah total bit baru yang digunakan:

Tabel 9.2 jumlah total bit

Surat	Kode	Frekuensi	Total Bit
a	001	12	36
b	0000	2	8
c	0001	7	28
d	010	13	39
e	011	14	42
f	1	85	85
Total			238

Jadi, kita menghemat 399 – 238 = 161 bit, atau hampir 40% dari ruang penyimpanan.

```
HuffmanCodingAlgorithm(int A[], int n) {
    Initialize a priority queue, PQ, to contain the n elements in A;
    struct BinaryTreeNode *temp;
    for (i = 1; i<n; i++) {
        temp  = (struct *)malloc(sizeof(BinaryTreeNode));
        temp→left = Delete-Min(PQ);
        temp→right = Delete-Min(PQ);
        temp→data = temp→left→data + temp→right→data;
        Insert temp to PQ;
    }
    return PQ;
}
```

Kompleksitas Waktu: O(nlogn), karena akan ada satu build_heap, 2n – 2 delete_mins, dan n – 2 insert, pada antrian prioritas yang tidak pernah memiliki lebih dari n elemen. Lihat bab Antrian Prioritas untuk detailnya.

9.8 ALGORITMA *GREEDY*: MASALAH & SOLUSI

Soal-1 Diberikan sebuah array F dengan ukuran n. Asumsikan konten array F[i] menunjukkan panjang file ke-i dan kita ingin menggabungkan semua file ini menjadi satu file tunggal. Periksa apakah algoritma berikut memberikan solusi terbaik untuk masalah ini atau tidak?

Algoritma: Menggabungkan file secara berurutan. Itu berarti pilih dua file pertama dan gabungkan. Kemudian pilih *output* dari penggabungan sebelumnya dan gabungkan dengan file ketiga, dan teruskan ...

Catatan: Diberikan dua file A dan B dengan ukuran m dan n, kompleksitas penggabungan adalah O(m + n).

Solusi: Algoritma ini tidak akan menghasilkan solusi yang optimal. Untuk contoh penghitung, mari kita perhatikan array ukuran file berikut.

$$F = \{10,5,100,50,20,15\}$$

Sesuai algoritma di atas, kita perlu menggabungkan dua file pertama (10 dan 5 file ukuran), dan sebagai hasilnya kita mendapatkan daftar file berikut. Dalam daftar di bawah, 15 menunjukkan biaya penggabungan dua file dengan ukuran 10 dan 5.

$$\{15,100,50,20,15\}$$

Demikian pula, menggabungkan 15 dengan file 100 berikutnya menghasilkan: {115,50,20,15}. Untuk langkah selanjutnya daftarnya menjadi

{165,20,15}, {185,15}

Akhirnya,

{200}

Total biaya penggabungan = Biaya semua operasi penggabungan = 15 + 115 + 165 + 185 + 200 = 680.

Untuk melihat apakah hasil di atas sudah optimal atau tidak, perhatikan urutannya: {5,10,15,20,50,100}. Untuk contoh ini, mengikuti pendekatan yang sama, total biaya penggabungan = 15 + 30 + 50 + 100 + 200 = 395. Jadi, algoritma yang diberikan tidak memberikan solusi terbaik (optimal).

Soal-2 Mirip dengan Soal-1, apakah algoritma berikut memberikan solusi optimal?
Algoritma: Menggabungkan file secara berpasangan. Itu berarti setelah langkah pertama, algoritma menghasilkan n/2 file perantara. Untuk langkah selanjutnya, kita perlu mempertimbangkan file perantara ini dan menggabungkannya secara berpasangan dan terus berjalan.
Catatan: Terkadang algoritma ini disebut penggabungan 2 arah. Alih-alih dua file sekaligus, jika kita menggabungkan K file sekaligus maka kita menyebutnya penggabungan K-way.

Solusi: Algoritma ini tidak akan menghasilkan solusi yang optimal dan mempertimbangkan contoh sebelumnya sebagai contoh counter. Sesuai algoritma di atas, kita perlu menggabungkan pasangan file pertama (10 dan 5 file ukuran), pasangan file kedua (100 dan 50) dan pasangan file ketiga (20 dan 15). Hasilnya, kita mendapatkan daftar file berikut.

{15.150,35}

Demikian pula, gabungkan *output* berpasangan dan langkah ini menghasilkan [di bawah, elemen ketiga tidak memiliki elemen pasangan, jadi tetap sama]:

{165,35}

Akhirnya,

{185}

Total biaya penggabungan = Biaya semua operasi penggabungan = 15 + 150 + 35 + 165 + 185 = 550. Ini jauh lebih dari 395 (dari masalah sebelumnya). Jadi, algoritma yang diberikan tidak memberikan solusi terbaik (optimal).

Soal-3 Dalam Soal-1, apa cara terbaik untuk menggabungkan semua file menjadi satu file?

Solusi: Dengan menggunakan algoritma *Greedy*, kita dapat mengurangi total waktu untuk menggabungkan file yang diberikan. Mari kita perhatikan algoritma berikut.

Algoritma:

1. Simpan ukuran file dalam antrian prioritas. Kunci dari elemen adalah panjang file.
2. Ulangi langkah berikut sampai hanya ada satu file:
   A. Ekstrak dua elemen terkecil X dan Y.
   B. Gabungkan X dan Y dan masukkan file baru ini ke dalam antrian prioritas.

Varian dari algoritma yang sama:

1. Urutkan ukuran file dalam urutan menaik.
2. Ulangi langkah berikut sampai hanya ada satu file:
   A. Ambil dua elemen pertama (terkecil) X dan Y.
   B. Gabungkan X dan Y dan masukkan file baru ini ke dalam daftar yang diurutkan.

Untuk memeriksa algoritma di atas, mari kita lacak dengan contoh sebelumnya. Array yang diberikan adalah:

F = {10,5,100,50,20,15}

Sesuai algoritma di atas, setelah mengurutkan daftar menjadi: {5,10,15,20,50,100}. Kita perlu menggabungkan dua file terkecil (5 dan 10 file ukuran) dan sebagai hasilnya kita mendapatkan daftar file berikut. Dalam daftar di bawah, 15 menunjukkan biaya penggabungan dua file dengan ukuran 10 dan 5.

{15,15,20,50,100}

Demikian pula, menggabungkan dua elemen terkecil (15 dan 15) menghasilkan: {20,30,50,100}. Untuk langkah selanjutnya daftarnya menjadi

{50,50,100} // menggabungkan 20 dan 30
{100.100} // menggabungkan 20 dan 30

Akhirnya,

{200}

Total biaya penggabungan = Biaya semua operasi penggabungan = 15 + 30 + 50 + 100 + 200 = 395. Jadi, algoritma ini menghasilkan solusi optimal untuk masalah penggabungan ini.

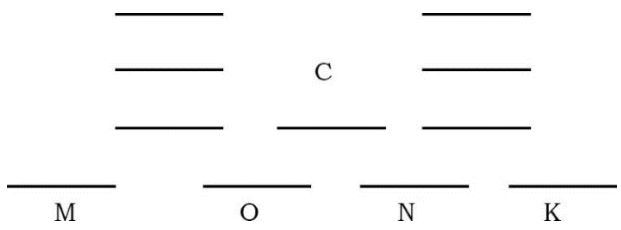
Kompleksitas Waktu: O(nlogn) waktu menggunakan tumpukan untuk menemukan pola penggabungan terbaik ditambah biaya optimal untuk menggabungkan file.

Soal-4 Algoritma Penjadwalan Interval: Diberikan himpunan n interval S = {(starti, endj)|1 ≤ i ≤ n}. Mari kita asumsikan bahwa kita ingin mencari subset maksimum S' dari S sehingga tidak ada pasangan interval dalam S' yang tumpang tindih. Periksa apakah algoritma berikut berfungsi atau tidak.

Algoritma:

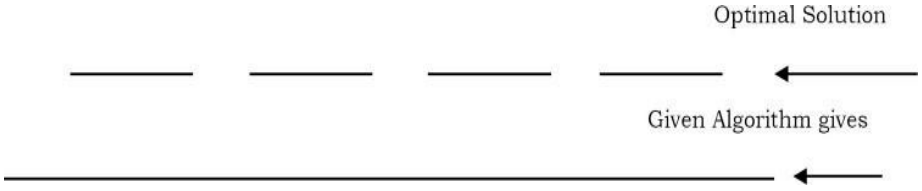
```
while (S is not empty) {
        Select the interval I that overlaps the least number of other intervals.
        Add I to final solution set S'.
        Remove all intervals from S that overlap with I.
}
```

Solusi: Algoritma ini tidak menyelesaikan masalah pencarian subset maksimum dari interval yang tidak tumpang tindih. Perhatikan interval berikut. Solusi optimalnya adalah {M,O,N,K}. Namun, interval yang tumpang tindih dengan yang lain paling sedikit adalah C, dan algoritma yang diberikan akan memilih C terlebih dahulu.



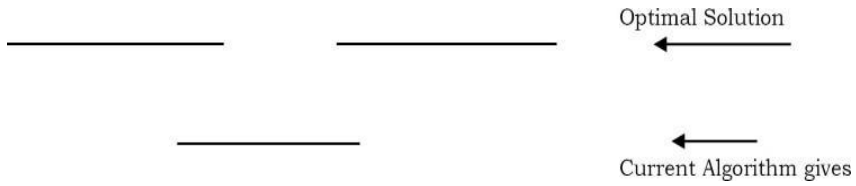
Soal-5 Dalam Soal-4, jika kita memilih interval yang dimulai paling awal (juga tidak tumpang tindih dengan interval yang sudah dipilih), apakah itu memberikan solusi optimal?

Solusi: Tidak. Itu tidak akan memberikan solusi optimal. Mari kita perhatikan contoh di bawah ini. Dapat dilihat bahwa solusi optimal adalah 4 sedangkan algoritma yang diberikan memberikan 1.



Soal-6 Dalam Soal-4, jika kita memilih interval terpendek (tetapi tidak tumpang tindih dengan interval yang sudah dipilih), apakah itu memberikan solusi optimal?

Solusi: Ini juga tidak akan memberikan solusi yang optimal. Mari kita perhatikan contoh di bawah ini. Dapat dilihat bahwa solusi optimal adalah 2 sedangkan algoritma memberikan 1.



Soal-7 Untuk Soal-4, apa solusi optimalnya?

Solusi: Sekarang, mari kita berkonsentrasi pada solusi serakah yang optimal.

Algoritma:

Sort intervals according to the right-most ends [end times];
for every consecutive interval {
- If the left-most end is after the right-most end of the last selected interval then we select this interval
- Otherwise we skip it and go to the next interval

}

Kompleksitas waktu = Waktu untuk menyortir + Waktu untuk memindai = O(nlogn + n) = O(nlogn).

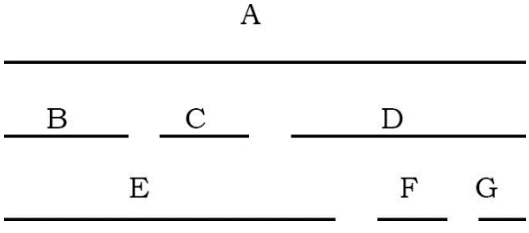
Soal-8 Perhatikan masalah berikut.

Masukan: S = {(starti,endi)|1 ≤ i ≤ n} interval. Interval (starti,endi) dapat kita perlakukan sebagai permintaan kamar untuk kelas dengan waktu mulai; ke waktu endi.

Output: Temukan tugas kelas ke ruangan yang menggunakan jumlah ruangan paling sedikit. Perhatikan algoritma iteratif berikut. Tetapkan sebanyak mungkin kelas ke ruang pertama, lalu tugaskan sebanyak mungkin kelas ke ruang kedua, lalu berikan sebanyak mungkin kelas ke ruang ketiga, dst. Apakah algoritma ini memberikan solusi terbaik?

Catatan: Sebenarnya, masalah ini mirip dengan algoritma penjadwalan interval. Perbedaannya hanya pada aplikasinya.

Solusi: Algoritma ini tidak menyelesaikan masalah pewarnaan interval. Perhatikan interval berikut:



Memaksimalkan jumlah kelas di ruang pertama menghasilkan {B, C, F, G} dalam satu ruang, dan kelas A, D, dan E masing-masing di kamar mereka sendiri, dengan total 4. Solusi optimalnya adalah letakkan A di satu ruangan, { B, C, D } di kamar lain, dan {E, F, G} di kamar lain, sehingga totalnya ada 3 kamar.

Soal-9 Untuk Soal-8, perhatikan algoritma berikut. Memproses kelas dalam urutan waktu mulai yang meningkat. Asumsikan bahwa kita sedang memproses kelas C. Jika ada ruang R sedemikian rupa sehingga R telah ditetapkan ke kelas sebelumnya, dan C dapat ditugaskan ke R tanpa tumpang tindih dengan kelas yang ditugaskan sebelumnya, maka tetapkan C ke R. Jika tidak, masukkan C ke dalam a ruangan baru. Apakah algoritma ini menyelesaikan masalah?

Solusi: Algoritma ini memecahkan masalah pewarnaan interval. Perhatikan bahwa jika algoritma serakah membuat ruang baru untuk kelas ci saat ini, maka karena itu memeriksa kelas dalam urutan waktu mulai, titik awal ci harus berpotongan dengan kelas terakhir di semua ruang saat ini. Jadi ketika serakah membuat ruang terakhir, n, itu karena waktu mulai kelas saat ini berpotongan dengan n – 1 kelas lainnya. Tetapi kita tahu bahwa untuk sembarang titik di kelas mana pun ia hanya dapat berpotongan dengan paling banyak s kelas lain, jadi pasti n $\leq$ S. Karena s adalah batas bawah pada jumlah total yang dibutuhkan, dan serakah layak, itu demikian juga optimal.

Catatan *Untuk solusi optimal lihat Soal-7 dan untuk kode lihat Soal-10.*

Soal-10 Misalkan kita diberikan dua larik Start[1 ..n] dan Finish[1 ..n] yang mencantumkan waktu mulai dan selesai setiap kelas. Tugas kita adalah memilih subset X terbesar yang mungkin $\in$ {1,2,...,n} sehingga untuk setiap pasangan i,j X, baik Mulai [i] > Selesai[j] atau Mulai [j] > Selesai [i]

Solusi: Tujuan kita adalah menyelesaikan kelas pertama sedini mungkin, karena itu membuat kita memiliki kelas yang tersisa.

```
int LargestTasks(int Start[], int n,  int Finish []) {
    sort Finish[];
    rearrange Start[] to match;
    count = 1;
    X[count]  = 1;
    for (i  = 2;  i<n;  i++) {
            if(Start[i] > Finish[X[count]])      {
                    count  = count + 1;
                    X[count] =  I;
            }
    }
    return X[1 .. count];
}
```

Kita memindai kelas sesuai urutan waktu selesai, dan setiap kali kita menemukan kelas yang tidak bertentangan dengan kelas terbaru sejauh ini, maka kita mengambil kelas itu. Algoritma ini jelas berjalan dalam waktu O(nlogn) karena penyortiran.

Soal-11 Pertimbangkan masalah membuat perubahan di negara India. Input untuk masalah ini adalah bilangan bulat M. *Output*nya harus berupa jumlah koin minimum untuk menghasilkan uang kembalian sebesar M rupee. Di India, asumsikan koin yang tersedia adalah 1,5,10,20,25,50 rupee. Asumsikan bahwa kita memiliki jumlah koin yang tidak terbatas untuk setiap jenis.

Untuk masalah ini, apakah algoritma berikut menghasilkan solusi optimal atau tidak? Ambil koin sebanyak mungkin dari denominasi tertinggi. Jadi misalnya, untuk membuat perubahan untuk 234 rupee, algoritma serakah akan mengambil empat koin 50 rupee, satu koin 25 rupee, satu koin 5 rupee, dan empat koin 1 rupee.

Solusi: Algoritma *greedy* tidak optimal untuk masalah membuat perubahan dengan jumlah koin minimum ketika denominasi 1,5,10,20,25, dan 50. Untuk menghasilkan 40 rupee, algoritma serakah akan menggunakan tiga uang logam 25,10, dan 5 rupiah. Solusi optimal adalah dengan menggunakan dua koin 20-shilling.

Catatan: *Untuk solusi optimal, lihat bab Pemrograman Dinamis.*

Soal-12 Mari kita asumsikan bahwa kita akan melakukan perjalanan jauh antara kota A dan B. Sebagai persiapan untuk perjalanan kita, kita telah mengunduh peta yang berisi jarak dalam mil antara semua pompa bensin di rute kita. Asumsikan tangki mobil kita dapat menampung bensin sejauh n mil. Asumsikan bahwa nilai n diberikan. Misalkan kita berhenti di setiap titik. Apakah itu memberikan solusi terbaik?

Solusi: Di sini algoritma tidak menghasilkan solusi optimal. Alasan Jelas: pengisian di setiap SPBU tidak menghasilkan solusi yang optimal.

Soal-13 Untuk Soal-12, berhentilah jika dan hanya jika Anda tidak memiliki cukup bensin untuk pergi ke SPBU berikutnya, dan jika Anda berhenti, isi tangki sampai penuh. Buktikan atau bantah bahwa algoritma ini menyelesaikan masalah dengan benar.

Solusi: Pendekatan serakah berhasil: Kita memulai perjalanan kita dari A dengan tangki penuh. Kita memeriksa peta kita untuk menentukan pompa bensin terjauh di rute kita dalam n mil. Kita berhenti di pom bensin itu, mengisi tangki kita dan

memeriksa peta kita lagi untuk menentukan pom bensin terjauh di rute kita dalam n mil dari perhentian ini. Ulangi proses sampai kita sampai ke B.

Catatan *Untuk kode, lihat bab Pemrograman Dinamis.*

Soal-14 Soal Ransel Pecahan: Soal yang diberikan t_1: t_2, ...,tn (barang yang mungkin ingin kita bawa di ransel) dengan bobot terkait s_1; s_2, ... , s_n dan nilai manfaat vx, v_2, …, vn, bagaimana kita bisa memaksimalkan manfaat total mengingat kita tunduk pada batas bobot mutlak C?

Solusi:

Algoritma:

1) Hitung nilai per kepadatan ukuran untuk setiap item $d_i = \frac{v_i}{s_i}$.
2) Urutkan setiap item berdasarkan kepadatan nilainya.
3) Ambil sebanyak mungkin item kepadatan yang belum ada di tas Kompleksitas Waktu: O(nlogn) untuk menyortir dan O(n) untuk pilihan serakah.

Catatan *Item dapat dimasukkan ke dalam antrian prioritas dan diambil satu per satu sampai tas penuh atau semua item telah dipilih. Ini sebenarnya memiliki runtime yang lebih baik dari O(n + clogn) di mana c adalah jumlah item yang benar-benar dipilih dalam solusi. Ada penghematan runtime jika c = O(n), tetapi sebaliknya tidak ada perubahan dalam kompleksitas.*

Soal-15 Jumlah peron kereta api: Di stasiun kereta api, kita memiliki jadwal kedatangan dan keberangkatan kereta api. Kita perlu mencari jumlah minimum peron agar semua kereta dapat diakomodasi sesuai jadwalnya.

Contoh: Jadwalnya seperti di bawah ini, jawabannya adalah 3. Jika tidak, stasiun kereta api tidak akan dapat menampung semua kereta api.

Rel	Kedatangan	Keberangkatan
Rel A	Jam 0900	Jam 0930
Rel B	Jam 0915	Jam 1300
Rel C	Jam 1030	Jam 1100
Rel D	Jam 1045	Jam 1145

Solusi: Mari kita ambil contoh yang sama seperti yang dijelaskan di atas. Perhitungan jumlah peron dilakukan dengan menentukan jumlah maksimum kereta api di stasiun kereta api setiap saat.

Pertama, urutkan semua waktu kedatangan(A) dan keberangkatan(D) dalam sebuah larik. Kemudian, simpan kedatangan dan keberangkatan yang sesuai dalam array juga. Setelah disortir, array kita akan terlihat seperti ini:

0900	0915	0930	1030	1045	1100	1145	1300
A	A	D	A	A	D	D	D

Sekarang ubah array dengan menempatkan 1 untuk A dan -1 untuk D. Array baru akan terlihat seperti ini:

1	1	-1	1	1	-1	-1	-1

Akhirnya buat array kumulatif dari ini:

1	2	1	2	3	2	1	0

Solusi kita akan menjadi nilai maksimum dalam array ini. Ini dia 3.

Catatan: *Jika kita memiliki kereta yang tiba dan yang lain berangkat pada waktu yang sama, maka tempatkan waktu keberangkatan terlebih dahulu dalam larik yang diurutkan.*

Soal-16 Pertimbangkan sebuah negara dengan jalan yang sangat panjang dan rumah-rumah di sepanjang jalan. Asumsikan bahwa penghuni semua rumah menggunakan telepon seluler. Kita ingin menempatkan menara ponsel di sepanjang jalan, dan setiap menara ponsel mencakup jarak 7 kilometer. Buat algoritma efisien yang memungkinkan menara ponsel paling sedikit.

Solusi:

Algoritma untuk menemukan jumlah menara ponsel paling sedikit:

1) Mulai dari awal jalan
2) Temukan rumah pertama yang ditemukan di jalan
3) Jika tidak ada rumah seperti itu, hentikan algoritma ini. Jika tidak, lanjutkan ke langkah berikutnya
4) Temukan menara ponsel 7 mil jauhnya setelah kita menemukan rumah ini di sepanjang jalan
5) Pergi ke langkah 2

Soal-17 Mempersiapkan Kaset Lagu: Misalkan kita memiliki satu set n lagu dan ingin menyimpannya di kaset. Di masa depan, pengguna akan ingin membaca lagu-lagu itu dari kaset. Membaca lagu dari kaset tidak seperti membaca dari disk; pertama kita harus maju cepat melewati semua lagu lainnya, dan itu membutuhkan banyak waktu. Biarkan A[1 .. n] menjadi larik yang mencantumkan panjang setiap lagu, khususnya, lagu i memiliki panjang A[i]. Jika lagu disimpan secara berurutan dari 1 sampai n, maka biaya untuk mengakses lagu ke-k adalah:

$$C(k) = \sum_{i=1}^{k} A[i]$$

Biaya mencerminkan fakta bahwa sebelum kita membaca lagu k kita harus terlebih dahulu memindai melewati semua lagu sebelumnya di kaset. Jika kita mengubah urutan lagu dalam kaset, kita mengubah biaya akses lagu, sehingga beberapa lagu menjadi lebih mahal untuk dibaca, tetapi yang lain menjadi lebih murah. Urutan lagu yang berbeda cenderung menghasilkan perkiraan biaya yang berbeda. Jika kita berasumsi bahwa setiap lagu memiliki kemungkinan yang sama untuk diakses, urutan mana yang harus kita gunakan jika kita ingin biaya yang diharapkan sekecil mungkin?

Solusi: Jawabannya sederhana. Kita harus menyimpan lagu dalam urutan dari terpendek ke terpanjang. Menyimpan lagu-lagu pendek di awal akan mengurangi waktu penerusan untuk pekerjaan yang tersisa.

Soal-18 Mari kita pertimbangkan serangkaian acara di HITEX (Pusat Konvensi Hyderabad). Asumsikan bahwa terdapat n kejadian dimana masing-masing kejadian membutuhkan satu satuan waktu. Peristiwa i akan memberikan keuntungan sebesar P [i] rupee (P [i] > 0) jika dimulai pada atau sebelum waktu T[i], di mana T[i] adalah bilangan arbitrer. Jika suatu acara tidak dimulai oleh T[i] maka tidak ada gunanya menjadwalkannya sama sekali. Semua acara dapat dimulai sedini waktu 0. Berikan algoritma yang efisien untuk menemukan jadwal yang memaksimalkan keuntungan.

Solusi:

Algoritma:

- Urutkan pekerjaan menurut lantai(T[i]) (diurutkan dari terbesar ke terkecil).
- Biarkan waktu t menjadi waktu saat ini sedang dipertimbangkan (di mana awalnya t = lantai(T[i])).
- Semua pekerjaan i di mana lantai(T[i]) = t dimasukkan ke dalam antrian prioritas dengan keuntungan g, digunakan sebagai kunci.
- DeleteMax dilakukan untuk memilih pekerjaan yang akan dijalankan pada waktu t.
- Kemudian t dikurangi dan proses dilanjutkan.

Jelas kompleksitas waktu adalah O(nlogn). Pengurutan membutuhkan O(nlogn) dan paling banyak n operasi insert dan DeleteMax dilakukan pada antrian prioritas, yang masing-masing membutuhkan waktu O(logn).

Soal-19 Mari kita pertimbangkan server layanan pelanggan (misalnya, layanan pelanggan seluler) dengan n pelanggan yang akan dilayani dalam antrian. Untuk mempermudah asumsikan bahwa waktu pelayanan yang dibutuhkan oleh setiap pelanggan diketahui sebelumnya dan itu adalah menit berat untuk pelanggan i. Jadi jika, misalnya, pelanggan dilayani dalam urutan kenaikan i, maka pelanggan ke-i harus tunggu: $\sum_{j=1}^{n-1} W_j$ menit. Total waktu tunggu semua pelanggan dapat diberikan sebagai = $\sum_{i=1}^{n} \sum_{j=1}^{n-1} W_j$. Apa cara terbaik untuk melayani pelanggan sehingga total waktu tunggu dapat dikurangi?

Solusi: Masalah ini dapat dengan mudah diselesaikan dengan menggunakan teknik serakah. Karena tujuan kita adalah mengurangi total waktu tunggu, yang dapat kita lakukan adalah memilih pelanggan yang waktu layanannya lebih sedikit. Artinya, jika kita memproses pelanggan dalam urutan waktu pelayanan yang meningkat maka kita dapat mengurangi total waktu tunggu.

Kompleksitas Waktu: O(nlogn).

BAB 10
ALGORITMA *DIVIDE AND CONQUER*

10.1 PENDAHULUAN

Dalam bab *Greedy*, kita telah melihat bahwa untuk banyak masalah strategi *Greedy* gagal memberikan solusi yang optimal. Di antara masalah tersebut, ada beberapa yang dapat dengan mudah diselesaikan dengan menggunakan teknik *Divide and conquer* (D&C). *Divide and conquer* adalah teknik desain algoritma yang penting berdasarkan rekursi.

Algoritma D&C bekerja dengan secara rekursif memecah suatu masalah menjadi dua atau lebih sub masalah yang bertipe sama, hingga menjadi cukup sederhana untuk diselesaikan secara langsung. Solusi untuk sub masalah kemudian digabungkan untuk memberikan solusi untuk masalah asli.

10.2 APA ITU STRATEGI *DIVIDE AND CONQUER*?

Strategi D & C memecahkan masalah dengan:

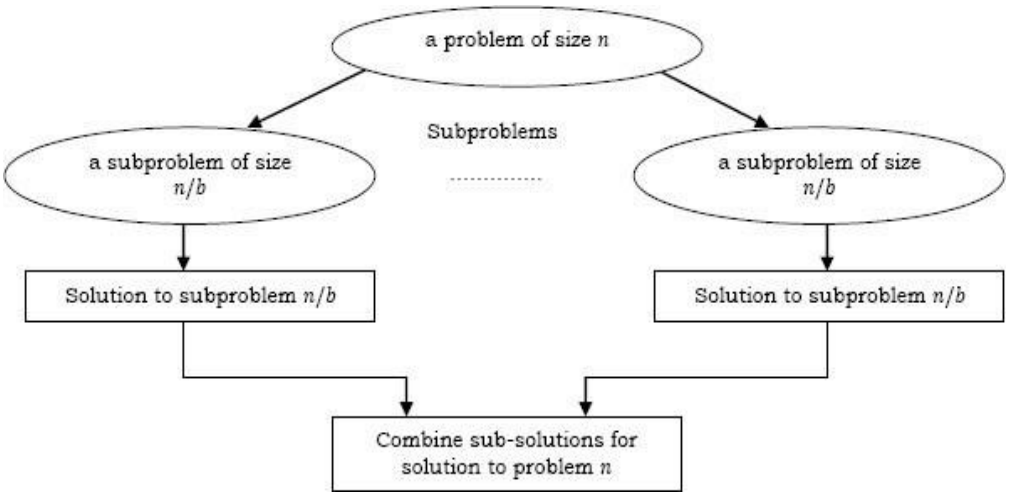
1) Divide: Memecah masalah menjadi sub-sub masalah yang merupakan contoh yang lebih kecil dari jenis masalah yang sama.
2) Rekursi: Memecahkan sub masalah ini secara rekursif.
3) Conquer: Menggabungkan jawaban mereka dengan tepat.

10.3 APAKAH *DIVIDE AND CONQUER* SELALU BERHASIL?

Tidak mungkin menyelesaikan semua masalah dengan teknik Divide & Conquer. Sesuai definisi D & C, rekursi memecahkan submasalah yang bertipe sama. Untuk semua masalah tidak mungkin menemukan submasalah yang berukuran sama dan D&C bukan pilihan untuk semua masalah.

10.4 VISUALISASI *DIVIDE AND CONQUER*

Untuk pemahaman yang lebih baik, perhatikan visualisasi berikut. Asumsikan bahwa n adalah ukuran masalah awal. Seperti dijelaskan di atas, kita dapat melihat bahwa masalah dibagi menjadi sub masalah dengan masing-masing ukuran n/b (untuk beberapa konstanta b). Kita memecahkan sub masalah secara rekursif dan menggabungkan solusi mereka untuk mendapatkan solusi untuk masalah asli.



Gambar 10.1 Visualisasi Divide And Conquer

```
DivideAndConquer ( P ) {
    if( small ( P ) )
        // P is very small so that a solution is obvious
        return solution ( n );
        divide the problem P into k sub problems P1, P2, ..., Pk;
        return (
            Combine (
            DivideAndConquer ( P1 ),
            DivideAndConquer ( P2 ),
            ...
            DivideAndConquer ( Pk )
            )
    );
}
```

10.5 MEMAHAMI *DIVIDE AND CONQUER*

Untuk pemahaman yang jelas tentang D & C, mari kita pertimbangkan sebuah cerita. Ada seorang lelaki tua yang adalah seorang petani kaya dan memiliki tujuh putra. Dia takut ketika dia meninggal, tanah dan hartanya akan dibagi di antara tujuh putranya, dan mereka akan bertengkar satu sama lain.

Jadi, dia mengumpulkan mereka dan menunjukkan kepada mereka tujuh batang kayu yang telah dia ikat dan memberi tahu mereka bahwa siapa pun yang dapat memecahkan ikatan itu akan mewarisi segalanya. Mereka semua mencoba, tetapi tidak ada yang bisa memecahkan bungkusan itu. Kemudian lelaki tua itu melepaskan ikatannya dan mematahkan tongkatnya satu per satu. Saudara-saudara memutuskan bahwa mereka harus tetap bersama dan bekerja bersama dan berhasil bersama. Moral untuk pemecah masalah berbeda. Jika kita tidak dapat menyelesaikan masalah, bagilah menjadi beberapa bagian, dan selesaikan satu bagian pada satu waktu.

Dalam bab-bab sebelumnya, kita telah memecahkan banyak masalah berdasarkan strategi D & C: seperti Pencarian Biner, Pengurutan Gabung, Pengurutan Cepat, dll.... Lihat topik tersebut untuk mendapatkan gambaran tentang cara kerja D & C. Di bawah ini adalah

beberapa masalah real-time lainnya yang dapat dengan mudah diselesaikan dengan strategi D & C. Untuk semua masalah ini kita dapat menemukan submasalah yang mirip dengan masalah aslinya.

- Mencari nama di buku telepon: Kita memiliki buku telepon dengan nama dalam urutan abjad. Diberi nama, bagaimana kita mengetahui apakah nama itu ada di buku telepon atau tidak?
- Memecah batu menjadi debu: Kita ingin mengubah batu menjadi debu (batu yang sangat kecil).
- Menemukan pintu keluar di sebuah hotel: Kita berada di ujung lobi hotel yang sangat panjang dengan serangkaian pintu yang panjang, dengan satu pintu di sebelah kita. Kita mencari pintu yang mengarah ke pintu keluar.
- Menemukan mobil kita di tempat parkir.

10.6 KEUNTUNGAN *DIVIDE AND CONQUER*

Memecahkan masalah yang sulit: D & C adalah metode yang ampuh untuk memecahkan masalah yang sulit. Sebagai contoh, perhatikan masalah Menara Hanoi. Hal ini membutuhkan pemecahan masalah menjadi submasalah, memecahkan kasus-kasus sepele dan menggabungkan submasalah untuk memecahkan masalah asli. Membagi masalah menjadi submasalah sehingga submasalah dapat digabungkan kembali merupakan kesulitan utama dalam merancang algoritma baru. Untuk banyak masalah seperti itu, D & C memberikan solusi sederhana.

Paralelisme: Karena D & C memungkinkan kita untuk menyelesaikan subproblem secara mandiri, ini memungkinkan untuk dieksekusi di mesin multiprosesor, terutama sistem shared-memory dimana komunikasi data antar prosesor tidak perlu direncanakan sebelumnya, karena subproblem yang berbeda dapat dieksekusi pada prosesor yang berbeda.

Akses memori: Algoritma D & C secara alami cenderung menggunakan cache memori secara efisien. Ini karena sekali submasalah kecil, semua submasalahnya dapat diselesaikan di dalam cache, tanpa mengakses memori utama yang lebih lambat.

10.7 KERUGIAN *DIVIDE AND CONQUER*

Salah satu kelemahan dari pendekatan D&C adalah rekursinya lambat. Ini karena overhead dari panggilan submasalah yang berulang. Juga, pendekatan D & C membutuhkan tumpukan untuk menyimpan panggilan (status pada setiap titik dalam rekursi). Sebenarnya ini tergantung pada gaya implementasi. Dengan kasus dasar rekursif yang cukup besar, overhead rekursi dapat diabaikan untuk banyak masalah.

Masalah lain dengan D & C adalah, untuk beberapa masalah, mungkin lebih rumit daripada pendekatan berulang. Misalnya, untuk menambahkan n angka, pengulangan sederhana untuk menjumlahkannya secara berurutan jauh lebih mudah daripada pendekatan

D & C yang memecah himpunan angka menjadi dua bagian, menambahkannya secara rekursif, dan kemudian menjumlahkannya.

10.8 TEOREMA UTAMA

Seperti yang dinyatakan di atas, dalam metode D & C, kita menyelesaikan sub masalah secara rekursif. Semua masalah umumnya didefinisikan dalam definisi rekursif. Masalah rekursif ini dapat dengan mudah diselesaikan dengan menggunakan teorema Master. Untuk detail tentang teorema Master, lihat bab Pengantar Analisis Algoritma. Hanya untuk kesinambungan, mari kita pertimbangkan kembali teorema Master.

Jika perulangan dalam bentuk $T(n) = aT(\frac{n}{b}) + \Theta(n^k log^p n)$, di mana a 1, b > 1, k 0 dan p adalah bilangan real, maka kompleksitasnya dapat langsung diberikan sebagai:

1) Jika a > b^k, maka $T(n) = \Theta\left(n^{log_b^a}\right)$
2) Jika a = b^k
   a. Jika p > -1, mak$T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$
   b. Jika p = -1, mak$T(n) = \Theta\left(n^{log_b^a} loglogn\right)$
   c. Jika p < -1, mak$T(n) = \Theta\left(n^{log_b^a}\right)$
3) Jika a < b^k
   a. Jika p > 0, maka T(n) = ($n^k \log^p n$)
   b. Jika p < 0, maka T(n) = O(n^k)

10.9 APLIKASI *DIVIDE AND CONQUER*

- Pencarian Biner
- Gabungkan Sortir dan Sortir Cepat
- Temuan Median
- Temuan Min dan Maks
- Perkalian Matriks
- Masalah Pasangan Terdekat

10.10 *DIVIDE AND CONQUER*: MASALAH & SOLUSI

Soal-1 Mari kita pertimbangkan sebuah algoritma A yang memecahkan masalah dengan membaginya menjadi lima subproblem dengan ukuran setengah, secara rekursif menyelesaikan setiap subproblem, dan kemudian menggabungkan solusi dalam waktu linier. Apa kompleksitas dari algoritma ini?

Solusi: Mari kita asumsikan bahwa ukuran input adalah n dan T(n) mendefinisikan solusi untuk masalah yang diberikan. Sesuai deskripsi, algoritma membagi masalah menjadi 5 sub masalah dengan ukuran masing-masing $\frac{n}{2}$. Jadi kita

perlu memecahkan $5\mathrm{T}(\frac{n}{2})$ submasalah. Setelah menyelesaikan sub masalah ini, array yang diberikan (waktu linier) dipindai untuk menggabungkan solusi ini. Algoritma perulangan total untuk masalah ini dapat diberikan sebagai $T(n) = 5T\left(\frac{n}{2}\right) + O(n)$. Menggunakan teorema Master (dari D & C), kita mendapatkan kompleksitas $O\left(n^{log_2^5}\right) \approx O(n^{2+}) \approx O(n^3)$.

Soal-2 Mirip dengan Soal-1, algoritma B memecahkan masalah berukuran n dengan memecahkan dua submasalah berukuran n – 1 secara rekursif dan kemudian menggabungkan solusi dalam waktu yang konstan. Apa kompleksitas dari algoritma ini?

Solusi: Mari kita asumsikan bahwa ukuran input adalah n dan T(n) mendefinisikan solusi untuk masalah yang diberikan. Sesuai deskripsi algoritma, kita membagi masalah menjadi 2 sub masalah dengan masing-masing ukuran n – 1. Jadi, kita harus menyelesaikan sub masalah 2T(n – 1). Setelah menyelesaikan sub masalah ini, algoritma hanya membutuhkan waktu yang konstan untuk menggabungkan solusi ini. Algoritma perulangan total untuk masalah ini dapat diberikan sebagai:

$$T(n) = 2T(n-1) + \mathrm{O}(1)$$

Menggunakan teorema Master (Pengurangan dan Penaklukan), kita mendapatkan kompleksitas sebagai $\mathrm{O}\left(n^0 2^{\frac{n}{1}}\right) = \mathrm{O}(2^n)$. (Lihat bab Pendahuluan untuk lebih jelasnya).

Soal-3 Sekali lagi mirip dengan Soal-1, algoritma lain C memecahkan masalah berukuran n dengan membaginya menjadi sembilan submasalah berukuran $\frac{n}{3}$, memecahkan setiap submasalah secara rekursif, dan kemudian menggabungkan solusi dalam waktu $O(n^2)$. Apa kompleksitas dari algoritma ini?

Solusi: Mari kita asumsikan bahwa ukuran input adalah n dan T(n) mendefinisikan solusi untuk masalah yang diberikan. Sesuai deskripsi algoritma kita membagi masalah menjadi 9 sub masalah dengan ukuran masing-masing $\frac{n}{3}$. Jadi kita perlu memecahkan $9\mathrm{T}(\frac{n}{3})$sub masalah. Setelah menyelesaikan sub masalah, algoritma mengambil waktu kuadrat untuk menggabungkan solusi ini. Algoritma perulangan total untuk masalah ini dapat diberikan sebagai:

$T(n) = 9T\left(\frac{n}{3}\right) + O(n^2)$. Menggunakan teorema Master D & C, kita mendapatkan kompleksitas sebagai $O(n^2 \log n)$.

Soal-4 Tulis perulangan dan selesaikan.

```
void function(n) {
    if(n > 1) {
        printf("*");
        function(n/2);
        function(n/2);
    }
}
```

Solusi: Mari kita asumsikan bahwa ukuran input adalah n dan T(n) mendefinisikan solusi untuk masalah yang diberikan. Sesuai kode yang diberikan, setelah mencetak karakter dan membagi masalah menjadi 2 submasalah dengan masing-masing ukuran $\frac{n}{2}$ dan menyelesaikannya. Jadi kita perlu memecahkan $2T(\frac{n}{2})$ submasalah. Setelah menyelesaikan submasalah ini, Algoritma tidak melakukan apa pun untuk menggabungkan solusi. Algoritma perulangan total untuk masalah ini dapat diberikan sebagai:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

Menggunakan teorema Master (dari D & C), kita mendapatkan kompleksitas sebagai $O\left(n^{log_2^2}\right) \approx O(n^1) = O(n)$.

Soal-5 Diberikan sebuah array, berikan algoritma untuk mencari maksimum dan minimum.

Solusi: Lihat bab Algoritma Seleksi.

Soal-6 Diskusikan Pencarian Biner dan kompleksitasnya.

Solusi: Lihat bab Pencarian untuk diskusi tentang Pencarian Biner.

Analisis Mari kita asumsikan bahwa ukuran input adalah n dan T(n) mendefinisikan solusi untuk masalah yang diberikan. Elemen-elemennya dalam urutan yang diurutkan. Dalam pencarian biner kita mengambil elemen tengah dan memeriksa apakah elemen yang akan dicari sama dengan elemen tersebut atau tidak. Jika sama maka kita kembalikan elemen tersebut.

Jika elemen yang akan dicari lebih besar dari elemen tengah maka kita mempertimbangkan sub-array kanan untuk menemukan elemen dan membuang sub-array kiri. Demikian pula, jika elemen yang akan dicari lebih

kecil dari elemen tengah maka kita mempertimbangkan sub-array kiri untuk menemukan elemen dan membuang sub-array kanan.

Artinya, dalam kedua kasus kita membuang setengah dari sub-array dan mempertimbangkan tersisa setengahnya saja. Juga, pada setiap iterasi kita membagi elemen menjadi dua bagian yang sama.

Sesuai pembahasan di atas setiap kali kita membagi masalah menjadi 2 sub masalah dengan ukuran masing-masing $\frac{n}{2}$ dan memecahkan $\mathrm{T}(\frac{n}{2})$ satu sub masalah. Algoritma perulangan total untuk masalah ini dapat diberikan sebagai:

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathrm{O}(1)$$

Menggunakan teorema Master (dari D & C), kita mendapatkan kompleksitas sebagai O(logn).

Soal-7 Pertimbangkan versi modifikasi dari pencarian biner. Mari kita asumsikan bahwa array dibagi menjadi 3 bagian yang sama (pencarian ternary) alih-alih 2 bagian yang sama. Tulis pengulangan untuk pencarian ternary ini dan temukan kompleksitasnya.

Solusi: Dari pembahasan Soal-5, pencarian biner memiliki relasi perulangan: $\mathrm{T(n)} = \mathrm{T}\left(\frac{\mathrm{n}}{2}\right) + \mathrm{O}(1)$. Serupa dengan pembahasan Soal-5, alih-alih 2

$$T(n) = T\left(\frac{n}{3}\right) + \mathrm{O}(1)$$

dalam relasi perulangan kita menggunakan “3”. Itu menunjukkan bahwa kita membagi array menjadi 3 sub-array dengan ukuran yang sama dan hanya mempertimbangkan salah satunya. Jadi, pengulangan untuk pencarian terner dapat diberikan sebagai:

Menggunakan teorema Master (dari D & C), kita mendapatkan kompleksitas sebagai $\mathrm{O}(log_3^n) \approx \mathrm{O}(logn)$ ≈ O(logn) (kita tidak perlu khawatir tentang basis log karena mereka adalah konstanta).

Soal-8 Dalam Soal-5, bagaimana jika kita membagi array menjadi dua set ukuran kira-kira sepertiga dan dua pertiga.

Solusi: Kita sekarang mempertimbangkan versi pencarian ternary yang sedikit dimodifikasi di mana hanya satu perbandingan yang dibuat, yang membuat dua partisi, satu dari $\frac{n}{3}$ elemen kasar dan yang lainnya dari $\frac{2n}{3}$. Di sini kasus

terburuk datang ketika panggilan rekursif ada di bagian elemen yang lebih besar $\frac{2n}{3}$. Jadi pengulangan yang sesuai dengan kasus terburuk ini adalah:

$$T(n) = T\left(\frac{2n}{3}\right) + \mathrm{O}(1)$$

Menggunakan teorema Master (dari D & C), kita mendapatkan kompleksitas sebagai O(logn). Menarik untuk dicatat bahwa kita akan mendapatkan hasil yang sama untuk pencarian k-ary umum (selama k adalah konstanta tetap yang tidak bergantung pada n) saat n mendekati tak terhingga.

Soal-9 Diskusikan Merge Sort dan kompleksitasnya.

Solusi: Lihat bab Sorting untuk diskusi tentang Merge Sort. Dalam Merge Sort, jika jumlah elemen lebih besar dari 1, kemudian membaginya menjadi dua himpunan bagian yang sama, Algoritma dipanggil secara rekursif pada himpunan bagian, dan himpunan bagian yang diurutkan yang dikembalikan digabungkan untuk memberikan daftar yang diurutkan dari himpunan asli. Persamaan perulangan dari algoritma Merge Sort adalah:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O(n), if\ n > 1 \\ 0 \qquad\qquad\qquad\quad , if n = 1 \end{cases}$$

Jika kita menyelesaikan perulangan ini menggunakan teorema Master D & C, ini memberikan kompleksitas O(nlogn).

Soal-10 Diskusikan Quick Sort dan kompleksitasnya.

Solusi: Lihat bab Sorting untuk diskusi tentang Quick Sort. Untuk Quick Sort kita memiliki kompleksitas yang berbeda untuk kasus terbaik dan kasus terburuk.

Kasus Terbaik: Dalam Quick Sort, jika jumlah elemen lebih besar dari 1 maka mereka dibagi menjadi dua himpunan bagian yang sama, dan algoritma dipanggil secara rekursif pada himpunan bagian. Setelah menyelesaikan sub masalah, kita tidak perlu menggabungkannya. Ini karena di Quick Sort mereka sudah diurutkan. Tapi, kita perlu memindai elemen lengkap untuk mempartisi elemen. Persamaan rekurensi kasus terbaik Quick Sort adalah

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O(n), if\ n > 1 \\ 0 \qquad\qquad\qquad\quad , if n = 1 \end{cases}$$

Jika kita memecahkan perulangan ini menggunakan teorema Master D & C memberikan kompleksitas O(nlogn).

Kasus Terburuk: Dalam kasus terburuk, Quick Sort membagi elemen input menjadi dua set dan salah satunya hanya berisi satu elemen. Itu berarti himpunan lain memiliki n – 1 elemen untuk diurutkan. Mari kita asumsikan bahwa ukuran input adalah

n dan T(n) mendefinisikan solusi untuk masalah yang diberikan. Jadi kita perlu menyelesaikan submasalah T(n – 1), T(1). Tetapi untuk membagi input menjadi dua set Quick Sort membutuhkan satu scan elemen input (ini membutuhkan O(n)).

Setelah menyelesaikan sub masalah ini, algoritma hanya membutuhkan waktu yang konstan untuk menggabungkan solusi ini. Algoritma perulangan total untuk masalah ini dapat diberikan sebagai:

$$T(n) = T(n-1) + \mathrm{O}(1) + \mathrm{O}(n).$$

Ini jelas merupakan persamaan pengulangan penjumlahan. Jadi,

$$T(n) = \frac{n(n+1)}{2} = \mathrm{O}(n^2)$$

Catatan: *Untuk analisis kasus rata-rata, lihat bab Sorting.*

Soal-11 Diberikan array tak hingga di mana n sel pertama berisi bilangan bulat dalam urutan yang diurutkan dan sel lainnya diisi dengan beberapa simbol khusus (misalnya, $). Asumsikan kita tidak mengetahui nilai n. Berikan Algoritma yang menggunakan bilangan bulat K sebagai input dan menemukan posisi dalam larik yang berisi K, jika posisi seperti itu ada, dalam waktu O(logn).

Solusi: Karena kita membutuhkan algoritma O(logn), kita tidak boleh mencari semua elemen dari daftar yang diberikan (yang memberikan kompleksitas O(n)). Untuk mendapatkan kompleksitas O(logn) satu kemungkinan adalah dengan menggunakan pencarian biner. Tetapi dalam skenario yang diberikan, kita tidak dapat menggunakan pencarian biner karena kita tidak mengetahui akhir dari daftar. Masalah pertama kita adalah menemukan akhir dari daftar. Untuk melakukan itu, kita bisa mulai dari elemen pertama dan terus mencari dengan indeks ganda. Itu artinya kita cari dulu di indeks 1 lalu, 2,4,8 ...

```
int FindInInfiniteSeries(int A[]) {
    int mid, l = r = 1;
    while( A[r] != '$') {
            l = r;
            r = r * 2;
    }
    while( (r – l > 1 ) {
            mid = (r – l)/2 + l;
            if( A[mid] == '$')
                    r = mid;
            else    l = mid;
    }
}
```

Jelas bahwa, setelah kita mengidentifikasi kemungkinan interval A[i,...,2i] di mana K mungkin, panjangnya paling banyak n (karena kita hanya memiliki n angka dalam larik A), jadi cari K menggunakan pencarian biner membutuhkan waktu O(logn).

Soal-12 Diberikan array yang diurutkan dari bilangan bulat tak berulang A[1.. n], periksa apakah ada indeks i yang A[i] = i. Berikan algoritma bagi-dan-taklukkan yang berjalan dalam waktu O(logn).

Solusi: Kita tidak dapat menggunakan pencarian biner pada array sebagaimana adanya. Jika kita ingin mempertahankan properti O(logn) dari solusi, kita harus mengimplementasikan pencarian biner kita sendiri. Jika kita memodifikasi array (di tempat atau dalam salinan) dan mengurangi i dari A[i], kita dapat menggunakan pencarian biner. Kompleksitas untuk melakukannya adalah O(n).

Soal-13 Kita diberikan dua daftar terurut dengan ukuran n. Berikan algoritma untuk menemukan elemen median dalam gabungan dua daftar.

Solusi: Kita menggunakan proses Merge Sort. Gunakan prosedur penggabungan pengurutan gabungan (lihat bab Pengurutan). Melacak hitungan sambil membandingkan elemen dari dua array. Jika hitungan menjadi n (karena ada 2n elemen), kita telah mencapai median. Ambil rata-rata elemen pada indeks n – 1 dan n dalam array gabungan.
Kompleksitas Waktu: O(n).

Soal-14 Bisakah kita memberikan Algoritma jika ukuran kedua daftar tidak sama?

Solusi: Solusinya mirip dengan masalah sebelumnya. Mari kita asumsikan bahwa panjang dari dua daftar adalah m dan n. Dalam hal ini kita perlu berhenti ketika penghitung mencapai (m + n)/2.
Kompleksitas Waktu: O((m + n)/2).

Soal-15 Bisakah kita meningkatkan kompleksitas waktu Soal-13 menjadi O(logn)?

Solusi: Ya, menggunakan pendekatan D&C. Mari kita asumsikan bahwa dua daftar yang diberikan adalah L1 dan L2.

Algoritma:

1. Temukan median dari array input terurut yang diberikan L1[] dan L2[]. Asumsikan median tersebut adalah m1 dan m2.
2. Jika m1 dan m2 sama maka kembalikan m1 (atau m2).
3. Jika m1 lebih besar dari m2, maka median akhir akan berada di bawah dua sub array.
4. Dari elemen pertama L1 ke m1.
5. Dari m2 ke elemen terakhir L2.
6. Jika m2 lebih besar dari m1, maka median ada di salah satu dari dua sub-array di bawah ini.

7. Dari m1 ke elemen terakhir L1.
8. Dari elemen pertama L2 ke m2.
9. Ulangi proses di atas hingga ukuran kedua sub array menjadi 2.
10. Jika ukuran kedua array adalah 2, maka gunakan rumus di bawah ini untuk mendapatkan median. 11. Median = (maks(L1[0],L2[0]) + min(L1[1],L2[1])/2

Kompleksitas Waktu: O(logn) karena kita hanya mempertimbangkan setengah dari input dan membuang setengah sisanya.

Soal-16 Diberikan array input A. Mari kita asumsikan bahwa mungkin ada duplikat dalam daftar. Sekarang cari elemen dalam daftar sedemikian rupa sehingga kita mendapatkan indeks tertinggi jika ada duplikat.

Solusi: Lihat bab Pencarian.

Soal-17 Diskusikan Algoritma Perkalian Matriks Strassen menggunakan *Divide and conquer*. Artinya, jika diberikan dua matriks n × n, A dan B, hitunglah matriks n × n C = A × B, di mana elemen-elemen C diberikan oleh

$$C_{i,j} = \sum_{k=0}^{n-1} A_{i,k}\, B_{k,j}$$

Solusi: Sebelum algoritma Strassen, pertama-tama mari kita lihat algoritma dasar bagi dan taklukkan. Pendekatan umum yang kita ikuti untuk memecahkan masalah ini diberikan di bawah ini. Untuk menentukan, C[i,j] kita perlu mengalikan baris ke-i dari A dengan kolom ke-j dari B.

```
// Initialize C.
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
        C[i, j] += A[i, k] * B[k, j];
```

Masalah perkalian matriks dapat diselesaikan dengan teknik D&C. Untuk mengimplementasikan algoritma D&C kita perlu memecah masalah yang diberikan menjadi beberapa submasalah yang mirip dengan yang asli. Dalam contoh ini kita melihat masing-masing matriks n × n sebagai matriks 2 × 2, yang elemen-elemennya adalah submatriks. Jadi, perkalian matriks asli, C = A × B dapat ditulis sebagai:

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

where each $A_{i,j}$, $B_{i,j}$, and $C_{i,j}$ is a $\frac{n}{2} \times \frac{n}{2}$ matrix.

Dari definisi $C_{i,j}$ yang diberikan, kita mendapatkan bahwa hasil submatriks dapat dihitung sebagai berikut:

$$C_{1,1} = A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1}$$
$$C_{1,2} = A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2}$$
$$C_{2,1} = A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1}$$
$$C_{2,2} = A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2}$$

Di sini simbol + dan × diartikan sebagai penjumlahan dan perkalian (masing-masing) dari $\frac{n}{2} \times \frac{n}{2}$ matriks.

Untuk menghitung perkalian matriks n × n asli, kita harus menghitung delapan produk $\frac{n}{2} \times \frac{n}{2}$ matriks (membagi) diikuti dengan empat $\frac{n}{2} \times \frac{n}{2}$ jumlah matriks (menaklukkan). Karena penambahan matriks adalah operasi $O(n^2)$, total waktu berjalan untuk operasi perkalian diberikan oleh pengulangan:

$$T(n) = \begin{cases} O(1) & , for\ n = 1 \\ 8T\left(\frac{n}{2}\right) + O(n^2) & , for\ n > 1 \end{cases}$$

Menggunakan teorema utama, kita mendapatkan $T(n) = O(n^3)$.

Untungnya, ternyata salah satu dari delapan perkalian matriks adalah redundan (ditemukan oleh Strassen). Perhatikan deret tujuh matriks berikut $\frac{n}{2} \times \frac{n}{2}$:

$$M_0 = (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2})$$
$$M_1 = (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2})$$
$$M_2 = (A_{1,1} - A_{2,1}) \times (B_{1,1} + B_{1,2})$$
$$M_3 = (A_{1,1} + A_{1,2}) \times B_{2,2}$$
$$M_4 = A_{1,1} \times (B_{1,2} - B_{2,2})$$
$$M_5 = A_{2,2} \times (B_{2,1} - B_{1,1})$$
$$M_6 = (A_{21} + A_{2,2}) \times B_{1,1}$$

Setiap persamaan di atas hanya memiliki satu perkalian. Sepuluh penambahan dan tujuh perkalian diperlukan untuk menghitung M_0 hingga M_6. Mengingat M_0 hingga M_6, kita dapat menghitung elemen-elemen dari matriks produk C sebagai berikut:

$$C_{1,1} = M_0 + M_1 - M_3 + M_5$$
$$C_{1,2} = M_3 + M_4$$
$$C_{2,1} = M_5 + M_6$$
$$C_{2,2} = M_0 - M_2 + M_4 - M_6$$

Pendekatan ini membutuhkan tujuh perkalian $\frac{n}{2} \times \frac{n}{2}$ matriks dan 18 $\frac{n}{2} \times \frac{n}{2}$ penambahan. Oleh karena itu, waktu berjalan kasus terburuk diberikan oleh pengulangan berikut:

$$T(n) = \begin{cases} O(1) & ,for\ n = 1 \\ 7T\left(\frac{n}{2}\right) + O(n^2) & ,for\ n = 1 \end{cases}$$

Dengan menggunakan teorema master, kita dapatkan, $T(n) = O\left(n^{log_2^7}\right) = O(n^{2.81})$

Soal-18 Soal Harga Saham: Pertimbangkan harga saham CareerMonk.com dalam n hari berturut-turut. Itu berarti input terdiri dari array dengan harga saham perusahaan. Kita tahu bahwa harga saham tidak akan sama sepanjang hari. Dalam harga saham input mungkin ada tanggal di mana stok tinggi saat kita bisa menjual kepemilikan saat ini, dan mungkin ada hari di mana kita bisa membeli saham. Sekarang masalah kita adalah mencari hari di mana kita bisa membeli saham dan hari di mana kita bisa menjual saham sehingga kita bisa mendapatkan keuntungan maksimal.

Solusi: Seperti yang diberikan dalam soal, mari kita asumsikan bahwa inputnya adalah array dengan harga saham [bilangan bulat]. Katakanlah array yang diberikan adalah A[1],...,A[n]. Dari array ini kita harus mencari dua hari [satu untuk beli dan satu untuk sel1] sedemikian rupa sehingga kita bisa mendapatkan keuntungan maksimal. Juga, poin lain yang harus diperhatikan adalah tanggal beli harus sebelum tanggal jual. Salah satu pendekatan sederhana adalah dengan melihat semua kemungkinan tanggal jual dan beli.

```
void StockStrategy(int A[], int n, int *buyDateIndex, int *sellDateIndex) {
    int j, profit=0;
    *buyDateIndex =0; *sellDateIndex =0;
    for (int i  = 1; i < n; i++) //indicates buy date
            //indicates sell date
            for( j  =  i; j < n;  j++)
                    if(A[j] - A[i] > profit)        {
                            profit = A[j] - A[i];
                            *buyDateIndex = i;
                            *sellDateIndex = j;
                    }
}
```

Dua loop bersarang membutuhkan n(n + l)/2 komputasi, jadi ini membutuhkan waktu Θ (n^2).

Soal-19 Untuk Soal-18, dapatkah kita meningkatkan kompleksitas waktu?

Solusi: Ya, dengan memilih solusi Divide-and-Conquer Θ (nlogn). Bagilah daftar input menjadi dua bagian dan temukan solusi secara rekursif di kedua bagian. Di sini, kita mendapatkan tiga kasus:

- buyDateIndex dan sellDateIndex keduanya berada pada periode waktu sebelumnya.
- buyDateIndex dan sellDateIndex keduanya berada di periode waktu selanjutnya.
- buyDateIndex berada di bagian awal dan sellDateIndex berada di bagian akhir periode waktu.

Dua kasus pertama dapat diselesaikan dengan rekursi. Kasus ketiga membutuhkan perawatan. Ini karena buyDateIndex di satu sisi dan sellDateIndex di sisi lain. Dalam hal ini kita perlu menemukan harga minimum dan maksimum di dua sub-bagian dan ini dapat kita selesaikan dalam waktu linier.

```
void StockStrategy(int A[], int left, int right) {
    //Declare the necessary variables;
    if(left + 1 = right)
            return (0, left, left);
    mid   = left + (right - left) / 2;
    (profitLeft, buyDateIndexLeft, sellDateIndexLeft) =  StockStrategy(A, left, mid);
    (profitRight, buyDateIndexRight, sellDateIndexRight)  = StockStrategy(A, mid, right);
    minLeft  = Min(A, left, mid);
    maxRight  = Max(A, mid, right);
    profit  = A[maxRight] - A[minLeft];
    if(profitLeft > max{profitRight, profit})
            return (profitLeft, buyDateIndexLeft, sellDateIndexLeft);
    else if(profitRight > max{profitLeft, profit})
            return (profitRight, buyDateIndexRight, sellDateIndexRight);
    else    return (profit, minLeft, maxRight);
}
```

Algoritma StockStrategy digunakan secara rekursif pada dua masalah setengah ukuran input, dan sebagai tambahan Θ(n) waktu dihabiskan untuk mencari harga maksimum dan minimum. Jadi kompleksitas waktu dicirikan oleh pengulangan T(n) = 2T(n/2) + Θ (n) dan dengan teorema Master kita mendapatkan O(nlogn).

Soal-20 Kita sedang menguji laptop yang “tidak bisa dipecahkan” dan tujuan kita adalah untuk mengetahui seberapa tidak mudah pecahnya laptop tersebut. Secara khusus, kita bekerja di gedung n-lantai dan ingin mengetahui lantai terendah tempat kita dapat menjatuhkan laptop tanpa merusaknya (sebut ini "langit-langit"). Misalkan kita diberikan dua laptop dan ingin mencari plafon setinggi mungkin. Berikan algoritma yang meminimalkan jumlah percobaan yang kita perlukan untuk membuat f(n) (semoga, f(n) adalah sub-linear, karena f(n) linier menghasilkan solusi trivial).

Solusi: Untuk masalah yang diberikan, kita tidak dapat menggunakan pencarian biner karena kita tidak dapat membagi masalah dan menyelesaikannya secara

rekursif. Mari kita ambil contoh untuk memahami skenario. Katakanlah 14 adalah jawabannya. Itu berarti kita membutuhkan 14 tetes untuk menemukan jawabannya. Pertama kita turun dari ketinggian 14, dan kalau rusak kita coba semua lantai dari 1 sampai 13. Kalau tidak pecah kita tinggal 13 tetes, jadi kita turunkan dari lantai 14 + 13 + 1 = 28. Pasalnya jika rusak di lantai 28 kita bisa mencoba semua lantai dari 15 hingga 27 dalam 12 tetes (total 14 tetes). Jika tidak pecah, maka kita memiliki 11 tetes dan kita dapat mencoba mencari lantai dalam 14 tetes.

Dari contoh di atas terlihat bahwa kita pertama kali mencoba dengan celah 14 lantai, lalu diikuti 13 lantai, lalu 12 lantai dan seterusnya. Jadi jika jawabannya k maka kita coba intervalnya di k, k – 1, k – 2 ....1. Mengingat bahwa jumlah lantai adalah n, kita harus menghubungkan keduanya. Karena lantai maksimum dari mana kita dapat mencoba adalah n, total lompatan harus kurang dari n. Ini memberikan:

$$\begin{aligned} k + (k-1) + (k-2) + \cdots + 1 &\leq n \\ \frac{k(k+1)}{2} &\leq n \\ k &\leq \sqrt{n} \end{aligned}$$

Kompleksitas proses ini adalah $\mathrm{O}(\sqrt{n})$

Soal-21 Diberikan n angka, periksa apakah ada dua yang sama.

Solusi: Lihat bab Pencarian.

Soal-22 Berikan algoritma untuk mengetahui apakah suatu bilangan bulat adalah persegi? Misalnya. 16 adalah, 15 tidak.

Solusi: Awalnya mari kita katakan i = 2. Hitung nilai i × i dan lihat apakah itu sama dengan angka yang diberikan. Jika sama maka kita selesai; jika tidak, tingkatkan i vlaue. Lanjutkan proses ini sampai kita mencapai i × i lebih besar dari atau sama dengan bilangan yang diberikan.

Kompleksitas Waktu: $\mathrm{O}(\sqrt{n})$.

Kompleksitas Ruang: O(1).

Soal-23 Diberikan sebuah larik 2n bilangan bulat dengan format berikut a1 a2 a3 ...an b1 b2 b3... bn. Acak array menjadi a1 b1 a2 b2 a3 b3 ... an bn tanpa memori tambahan [MA].

Solusi: Mari kita ambil contoh (untuk solusi brute force lihat bab Pencarian)

1. Mulailah dengan larik: a1 a2 a3 a4 b1 b2 b3 b4
2. Bagi array menjadi dua bagian: a1 a2 a3 a4 : b1 b2 b3 b4

3. Tukar elemen di sekitar pusat: tukar a3 a4 dengan b1 b2 Anda mendapatkan: a1 a2 b1 b2 a3 a4 b3 b4
4. Bagi a1 a2 b1 b2 menjadi a1 a2 : b1 b2 lalu bagi a3 a4 b3 b4 menjadi a3 a4 : b3 b4
5. Tukar elemen di sekitar pusat untuk setiap subarray yang Anda dapatkan: a1 b1 a2 b2 dan a3 b3 a4 b4

Harap dicatat bahwa solusi ini hanya menangani kasus ketika n = 2^i di mana i = 0,1,2,3, dll. Dalam contoh kita n = 2^2 = 4 yang memudahkan untuk membagi array secara rekursif menjadi dua bagian. Ide dasar di balik menukar elemen di sekitar pusat sebelum memanggil fungsi rekursif adalah untuk menghasilkan masalah ukuran yang lebih kecil. Solusi dengan kompleksitas waktu linier dapat dicapai jika elemen-elemennya bersifat spesifik. Misalnya Anda dapat menghitung posisi baru elemen menggunakan nilai elemen itu sendiri. Ini adalah teknik hashing.

```
void ShuffleArray(int A[], int l, int r) {
    //Array center
    int c = l + (r-l)/2, q = 1 + l + (c-l)/2;
    if(l == r) //Base case when the array has only one element
            return;
    for (int k = 1, i = q; i <= c; i++, k++) {
            //Swap elements around the center
            int tmp = A[i];    A[i] = A[c + k];    A[c + k] = tmp;
    }
    ShuffleArray(A, l, c);              //Recursively call the function on the left and right
    ShuffleArray(A, c + 1, r); );       //Recursively call the function on the right
}
```

Kompleksitas Waktu: O(nlogn).

Soal-24 Mur dan Baut Soal: Diberikan satu set n mur dengan ukuran dan n baut yang berbeda sehingga terdapat korespondensi satu-satu antara mur dan baut, temukan untuk setiap mur baut yang sesuai. Asumsikan bahwa kita hanya dapat membandingkan mur dengan baut (tidak dapat membandingkan mur dengan mur dan baut dengan baut).

Solusi: Lihat bab Menyortir.

Soal-25 Nilai Maksimum Barisan Bersebelahan: Diberikan barisan n bilangan A(1) ...A(n), berikan algoritma untuk menemukan barisan berurutan A(i) ...A(j) yang

jumlah dari elemen-elemen berikutnya adalah maksimum. Contoh : {-2, 11, -4, 13, -5, 2} → 20 dan {1, -3, 4, -2, -1, 6 } → 7.

Solusi: Bagilah input ini menjadi dua bagian. Jumlah maksimum berikutnya yang berdekatan dapat terjadi dalam salah satu dari 3 cara:

- Kasus 1: Bisa sepenuhnya di babak pertama
- Kasus 2: Itu bisa sepenuhnya di babak kedua
- Kasus 3: Dimulai di babak pertama dan berakhir di babak kedua

Kita mulai dengan melihat kasus 3. Untuk menghindari loop bersarang yang dihasilkan dari mempertimbangkan semua n/2 titik awal dan n/2 titik akhir secara independen, ganti dua loop bersarang dengan dua loop berurutan. Loop berurutan, masing-masing berukuran n/2, bergabung hanya membutuhkan kerja linier. Setiap urutan berurutan yang dimulai di babak pertama dan berakhir di babak kedua harus mencakup elemen terakhir dari babak pertama dan elemen pertama dari babak kedua. Apa yang bisa kita lakukan dalam kasus 1 dan 2 adalah menerapkan strategi yang sama untuk membagi menjadi lebih banyak bagian. Singkatnya, kita melakukan hal berikut:

1. Hitung secara rekursif urutan berurutan maksimum yang berada seluruhnya pada paruh pertama.
2. Hitung secara rekursif suburutan bersebelahan maksimum yang berada seluruhnya dalam babak kedua.
3. Hitung, melalui dua loop berturut-turut, jumlah maksimum berurutan berikutnya yang dimulai di babak pertama tetapi berakhir di babak kedua.
4. Pilih yang terbesar dari tiga jumlah.

Biaya kasus dasar adalah 1. Program melakukan dua panggilan rekursif ditambah pekerjaan linier yang terlibat dalam menghitung jumlah maksimum untuk kasus 3. Relasi perulangan adalah:

$$T(1) = 1$$
$$T(n) = 2T(n/2) + n$$

Menggunakan teorema Master D & C, kita mendapatkan kompleksitas waktu sebagai T(n) = O(nlogn).

Catatan: *Untuk solusi yang efisien, lihat bab Pemrograman Dinamis.*

```
int MaxSumRec(int A[], int left, int right) {
    int MaxLeftBorderSum = 0, MaxRightBorderSum = 0, LeftBorderSum = 0, RightBorderSum = 0;
    int mid = left + (right - left) / 2;
    if(left == right) // Base Case
        return A[left] > 0 ? A[left] : 0;
    int MaxLeftSum = MaxSumRec(A, left, mid);
    int MaxRightSum = MaxSumRec(A, mid + 1, right);
    for (int i = mid; i >= left; i--) {
        LeftBorderSum += A[i];
        if(LeftBorderSum > MaxLeftBorderSum)
            MaxLeftBorderSum = LeftBorderSum;
    }
    for (int j = mid + 1; j <= right; j++) {
        RightBorderSum += A[j];
        if(RightBorderSum > MaxRightBorderSum)
            MaxRightBorderSum = RightBorderSum;
    }
    return Max(MaxLeftSum, MaxRightSum,MaxLeftBorderSum + MaxRightBorderSum);
}
int MaxSubsequenceSum(int A, int n) {
    return n > 0 ? MaxSumRec(A, 0, n – 1) : 0;
}
```

Soal-26 Pasangan Titik Terdekat: Diberikan himpunan n titik, S = {p_1,p_2,p_3,…,pn}, di mana pi = (x_i,y_i). Temukan pasangan titik yang memiliki jarak terkecil di antara semua pasangan (asumsikan bahwa semua titik berada dalam satu dimensi).

Solusi: Mari kita asumsikan bahwa kita telah mengurutkan poin. Karena titik-titik berada dalam satu dimensi, semua titik berada dalam satu garis setelah kita mengurutkannya (baik pada sumbu X atau sumbu Y). Kompleksitas pengurutan adalah O(nlogn). Setelah menyortir, kita dapat menelusurinya untuk menemukan titik berurutan dengan perbedaan terkecil. Jadi masalah dalam satu dimensi diselesaikan dalam waktu O(nlogn) yang terutama didominasi oleh waktu pengurutan.

Kompleksitas Waktu: O(nlogn).

Soal-27 Untuk Soal-26, bagaimana kita menyelesaikannya jika titik-titik berada dalam ruang dua dimensi?

Solusi: Sebelum masuk ke algoritma, mari kita perhatikan persamaan matematika berikut:

$$distance(p_1, p_2) = \sqrt{(x_1 - x_2)^2 - (y_1 - y_2)^2}$$

Persamaan di atas menghitung jarak antara dua titik $p_1 = (x_1,y_1)$ dan $p_2 = (x_2,y_2)$.

Solusi Brute Force:

- Hitung jarak antara semua pasangan titik. Dari n titik ada cara untuk memilih 2 titik. .

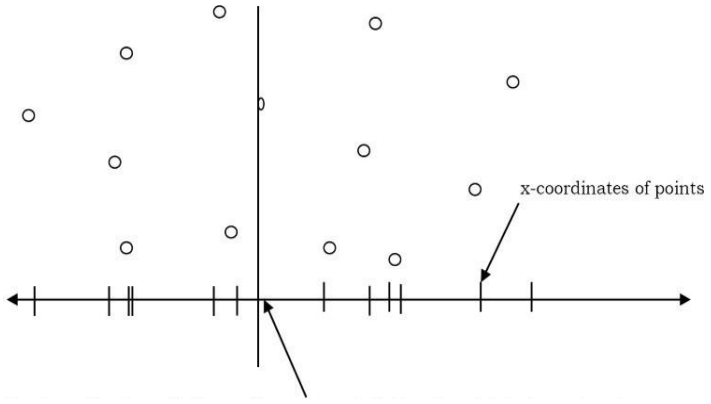
- Setelah menemukan jarak untuk semua n2 kemungkinan, kita memilih salah satu yang memberikan jarak minimum dan ini membutuhkan $O(n^2)$. Kompleksitas waktu keseluruhan adalah $O(n^2)$.

Soal-28 Berikan solusi O(nlogn) untuk masalah pasangan terdekat (Masalah-27)?

Solusi: Untuk mencari solusi O(nlogn), kita dapat menggunakan teknik D & C. Sebelum memulai proses bagi-dan-taklukkan, mari kita asumsikan bahwa titik-titik diurutkan berdasarkan peningkatan koordinat x. Bagilah titik-titik tersebut menjadi dua bagian yang sama berdasarkan median koordinat x. Itu berarti masalahnya dibagi menjadi menemukan pasangan terdekat di masing-masing dari dua bagian. Untuk kesederhanaan mari kita pertimbangkan algoritma berikut untuk memahami prosesnya.

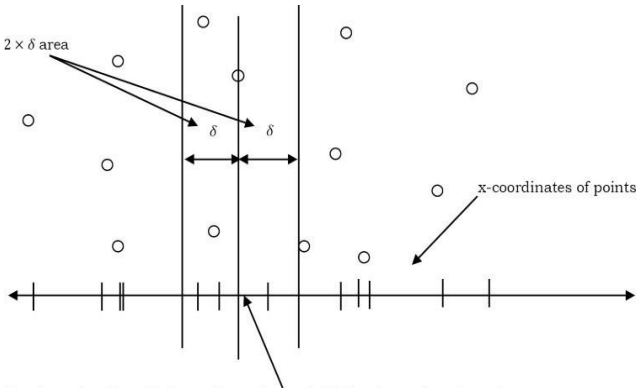
Algoritma:

1) Urutkan titik-titik yang diberikan dalam S (kumpulan titik tertentu) berdasarkan koordinat x-nya. Partisi S menjadi dua himpunan bagian, S_1 dan S_2, tentang garis l melalui median S. Langkah ini adalah bagian Divide dari teknik D & C.
2) Temukan pasangan terdekat di S_1 dan S_2 dan panggil mereka L dan R secara rekursif.
3) Sekarang, langkah 4 sampai 8 membentuk komponen Menggabungkan teknik D & C.
4) Mari kita asumsikan bahwa δ = min (L,R).
5) Hilangkan titik-titik yang berjarak δ lebih jauh dari dari l.
6) Pertimbangkan titik-titik yang tersisa dan urutkan berdasarkan koordinat y-nya.
7) Pindai titik-titik yang tersisa dalam urutan y dan hitung jarak setiap titik ke semua tetangganya yang berjarak tidak lebih dari 2 × δ (itulah alasan pengurutan menurut y).
8) Jika salah satu dari jarak ini kurang δ dari maka perbarui δ.



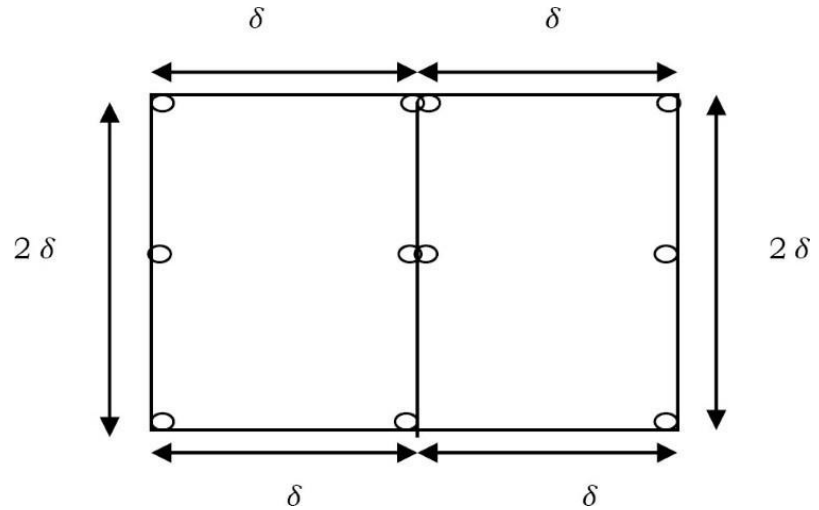
Garis l melalui titik median dan membagi himpunan menjadi 2 bagian yang sama.

Menggabungkan hasil dalam waktu linier



Garis l melalui titik median dan membagi himpunan menjadi 2 bagian yang sama

Misalkan δ = min(L,R), di mana L adalah solusi untuk sub masalah pertama dan R adalah solusi untuk sub masalah kedua. Kandidat yang mungkin untuk pasangan terdekat, yang berada di seberang garis pemisah, adalah mereka yang jaraknya kurang dari dari garis. Jadi kita hanya membutuhkan titik-titik yang berada di dalam area 2 × δ melintasi garis pemisah seperti yang ditunjukkan pada gambar. Sekarang, untuk memeriksa semua titik dalam jarak δ dari garis, perhatikan gambar berikut.



Dari diagram di atas kita dapat melihat bahwa maksimum 12 titik dapat ditempatkan di dalam alun-alun dengan jarak tidak kurang dari δ. Artinya, kita hanya perlu memeriksa jarak yang berada dalam 11 posisi dalam daftar yang diurutkan. Ini mirip dengan yang di atas, tetapi dengan perbedaan bahwa dalam penggabungan submasalah di atas, tidak ada batas vertikal. Jadi kita bisa menerapkan taktik kotak 12 poin di semua kotak yang mungkin di area 2 × δ

dengan garis pemisah sebagai garis tengah. Karena terdapat maksimum n kotak seperti itu di area tersebut, total waktu untuk menemukan pasangan terdekat di koridor adalah O(n).

Analisis:

1) Langkah-1 dan Langkah-2 ambil O(nlogn) untuk menyortir dan mencari minimum secara rekursif.
2) Langkah-4 membutuhkan O(1).
3) Langkah-5 membutuhkan O(n) untuk memindai dan menghilangkan.
4) Langkah-6 membutuhkan O(nlogn) untuk menyortir.
5) Langkah-7 membutuhkan O(n) untuk pemindaian.

Kompleksitas total: T(n) = O(nlogn) + O(1) + O(n) + O(n) + O(n) O(nlogn).

Soal-29 Untuk menghitung kn, berikan algoritma dan diskusikan kompleksitasnya.

Solusi: Algoritma naif untuk menghitung k^n adalah: mulai dengan 1 dan kalikan dengan k hingga mencapai kn. Untuk pendekatan ini; ada n – 1 perkalian dan masing-masing membutuhkan waktu yang konstan memberikan algoritma (n).

Tetapi ada cara yang lebih cepat untuk menghitung k^n. Sebagai contoh,

$$9^{24} = (9^{12})^2 = ((9^6)^2)^2 = (((9^3)^2)^2)^2 = (((9^2.9)^2)^2)^2$$

Perhatikan bahwa mengambil kuadrat dari suatu angka hanya membutuhkan satu perkalian; dengan cara ini, untuk menghitung 924 kita hanya membutuhkan 5 perkalian, bukan 23.

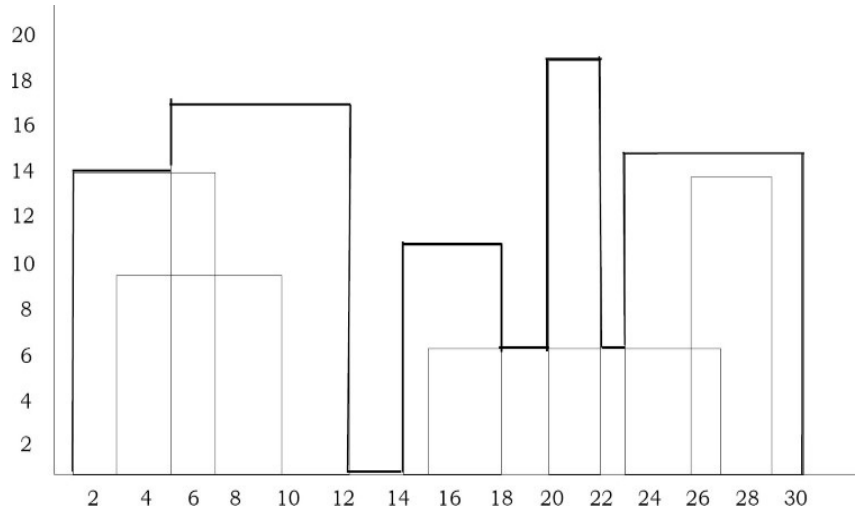
```
int Exponential(int k, int n) {
    if (k == 0)
        return 1;
    else{
        if (n%2 == 1){
            a = Exponential(k, n-1);
            return a*k;
        }
        else{
            a= Exponential(k, n/2);
            return a*a;
        }
    }
}
```

Misalkan T(n) adalah jumlah perkalian yang diperlukan untuk menghitung k^n. Untuk mempermudah, asumsikan $k = 2^i$ untuk beberapa $i \geq 1$.

$$T(n) = T(\frac{n}{2}) + 1$$

Menggunakan teorema master kita mendapatkan T(n) = O(logn).

Soal-30 The Skyline Soal: Diberikan lokasi dan bentuk yang tepat dari n bangunan persegi panjang di kota 2 dimensi. Tidak ada urutan khusus untuk bangunan persegi panjang ini. Asumsikan bahwa bagian bawah semua bangunan terletak pada garis horizontal yang tetap (tepi bawah adalah collinear). Inputnya adalah daftar tiga kali lipat; satu per gedung. Sebuah bangunan B_i diwakili oleh triple (l_i, h_i, r_i) di mana l_i menunjukkan posisi x dari tepi kiri dan r_i menunjukkan posisi x dari tepi kanan, dan hi menunjukkan tinggi bangunan. Berikan algoritma yang menghitung cakrawala (dalam 2 dimensi) dari bangunan ini, menghilangkan garis tersembunyi. Pada diagram di bawah ini ada 8 bangunan, diwakili dari kiri ke kanan oleh kembar tiga (1, 14, 7), (3, 9, 10), (5, 17, 12), (14, 11, 18), (15, 6, 27), (20, 19, 22), (23, 15, 30) dan (26, 14, 29).



*Output*nya adalah kumpulan titik-titik yang menggambarkan jalur kaki langit. Dalam beberapa versi soal, kumpulan poin ini diwakili oleh urutan angka p_1; p_2, ..., p_n, sehingga titik pi menyatakan garis horizontal yang ditarik pada ketinggian pi jika i genap, dan titik p_i menyatakan garis vertikal yang ditarik pada posisi pi jika i ganjil. Dalam kasus kita, kumpulan titik akan menjadi urutan p_1, p_2, ..., p_n pasang (x_i, h_i) di mana p_i(x_i, h_i) mewakili ketinggian tinggi kaki langit di posisi x_i. Pada diagram di atas kaki langit digambar dengan garis tebal mengelilingi bangunan dan diwakili oleh urutan pasangan posisi-tinggi (1, 14), (5, 17), (12, 0), (14, 11) , (18, 6), (20, 19), (22, 6), (23, 15) dan (30, 0). Juga, asumsikan bahwa R_i gedung paling kanan bisa maksimal 1000. Itu berarti, koordinat L_i gedung kiri bisa minimal 1 dan R_i gedung paling kanan bisa maksimal 1000.

Solusi: Informasi yang paling penting adalah bahwa kita tahu bahwa koordinat kiri dan kanan setiap bangunan adalah bilangan bulat non-negatif kurang dari 1000. Sekarang mengapa ini penting? Karena kita dapat menetapkan nilai tinggi untuk setiap koordinat xi yang berbeda di mana i berada di antara 0 dan 9.999.

Algoritma:

- Alokasikan array untuk 1000 elemen dan inisialisasi semua elemen ke 0. Sebut saja array ini auxHeights.
- Lakukan iterasi pada semua bangunan dan untuk setiap gedung B_i iterasi pada rentang $[l_i.. r_i)$ di mana l_i adalah kiri, r_i adalah koordinat kanan gedung B_i.
- Untuk setiap elemen x_j dari rentang ini, periksa apakah h_i>auxHeights[x_j], yaitu jika gedung B_i lebih tinggi dari nilai ketinggian saat ini di posisi x_j. Jika demikian, ganti auxHeights[x_j] dengan h_i.

Setelah kita memeriksa semua bangunan, susunan auxHeights menyimpan ketinggian bangunan tertinggi di setiap posisi. Ada satu hal lagi yang harus dilakukan: mengonversi larik auxHeights ke format *output* yang diharapkan, yaitu ke urutan pasangan posisi-tinggi. Ini juga mudah: cukup petakan setiap indeks i ke pasangan (i, auxHeights[i]).

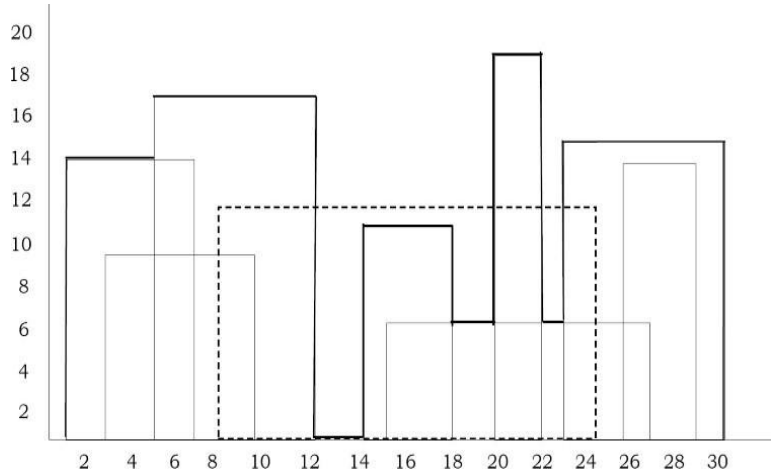
```
#include<stdio.h>
#define MaxRightMostBuildingRi 1000
int auxHeights[MaxRightMostBuildingRi];
int SkyLineBruteForce(){
        int left,h,right,i,prev;
        int rightMostBuildingRi=0;
        while(scanf("%d %d %d", &left, &h, &right)==3){
           for(i=left;i<right;i++)
              if(auxHeights[i]<h)
                 auxHeights[i]=h;
              if(rightMostBuildingRi<right)
                 rightMostBuildingRi=right;
        }
        prev = 0;
        for(i=1;i<rightMostBuildingRi;i++){
           if(prev!=auxHeights[i]){
              printf("%d %d ", i, auxHeights[i]);
              prev=auxHeights[i];
           }
        }
        printf("%d %d\n", rightMostBuildingRi, auxHeights[rightMostBuildingRi]);
        return 0;
}
```

Mari kita lihat kompleksitas waktu dari algoritma ini. Asumsikan bahwa, n menunjukkan jumlah bangunan dalam urutan input dan m menunjukkan koordinat maksimum (bangunan paling kanan r_i). Dari kode di atas, jelas bahwa untuk setiap bangunan input baru, kita melintasi dari kiri (l_i) ke kanan (r_i) untuk memperbarui ketinggian. Dalam kasus terburuk, dengan n bangunan berukuran sama, masing-masing memiliki koordinat l = 0 kiri dan r = m – 1 kanan, yaitu setiap bangunan membentang di seluruh interval [0.. m).

Jadi waktu berjalan dari pengaturan ketinggian setiap posisi adalah O(n × m). Kompleksitas waktu keseluruhan adalah O(n × m), yang jauh lebih besar daripada $O(n^2)$ jika m > n.

Soal-31 Bisakah kita meningkatkan solusi Soal-30?

Solusi: Akan menjadi percepatan yang sangat besar jika entah bagaimana kita bisa menentukan cakrawala dengan menghitung ketinggian untuk koordinat tersebut hanya di tempat yang penting, bukan? Intuisi memberitahu kita bahwa jika kita dapat memasukkan sebuah bangunan ke dalam skyline yang ada maka alih-alih semua koordinat bentang bangunan, kita hanya perlu memeriksa ketinggian di koordinat kiri dan kanan bangunan ditambah koordinat skyline yang tumpang tindih dengan bangunan tersebut. dan dapat memodifikasi.



Apakah menggabungkan dua kaki langit secara substansial berbeda dari menggabungkan bangunan dengan satu kaki langit? Jawabannya, tentu saja, Tidak. Ini menunjukkan bahwa kita menggunakan membagi-dan-menaklukkan. Bagilah input dari n bangunan menjadi dua set yang sama. Hitung (secara rekursif) skyline untuk setiap set kemudian gabungkan dua skyline. Memasukkan bangunan satu demi satu bukanlah cara tercepat untuk menyelesaikan masalah ini seperti yang telah kita lihat di atas. Namun, jika pertama-tama kita menggabungkan pasangan bangunan menjadi kaki langit, lalu kita menggabungkan pasangan kaki langit ini menjadi kaki langit yang lebih besar (dan bukan dua rangkaian bangunan), dan kemudian menggabungkan pasangan kaki langit yang lebih besar ini menjadi yang lebih besar, maka - karena masalahnya ukuran dibelah dua di setiap langkah -setelah langkah logn kita dapat menghitung cakrawala akhir.

```
class SkyLineDivideandConquer {
 public:
 vector<pair<int, int>> getSkyline(int start, int end, vector<vector<int>>& buildings) {
    if (start == end) {
       vector<pair<int, int>> result;
       result.push_back(pair<int, int>(buildings[start][0], buildings[start][1]));
       result.push_back(pair<int, int>(buildings[start][2], 0));
       return result;
    }
    int mid = (end + start) / 2;
    vector<pair<int, int>> leftSkyline = getSkyline(start, mid, buildings);
    vector<pair<int, int>> rightSkyline = getSkyline(mid + 1, end, buildings);
    vector<pair<int, int>> result = mergeSkylines(leftSkyline, rightSkyline);
    return result;
 }
 vector<pair<int, int>> mergeSkylines(vector<pair<int, int>>& leftSkyline, vector<pair<int, int>>& rightSkyline) {
    vector<pair<int, int>> result;
    int i = 0, j = 0, currentHeight1 = 0, currentHeight2 = 0;
    int maxH = max(currentHeight1, currentHeight2);
    while (i != leftSkyline.size() && j != rightSkyline.size()) {
       if (leftSkyline[i].first < rightSkyline[j].first) {
          currentHeight1 = leftSkyline[i].second;
          if (maxH != max(currentHeight1, currentHeight2))
             result.push_back(pair<int, int>(leftSkyline[i].first, max(currentHeight1, currentHeight2)));
          maxH = max(currentHeight1, currentHeight2);
          i++;
       }
       else if (leftSkyline[i].first > rightSkyline[j].first) {
          currentHeight2 = rightSkyline[j].second;
          if (maxH != max(currentHeight1, currentHeight2))
             result.push_back(pair<int, int>(rightSkyline[j].first, max(currentHeight1, currentHeight2)));
          maxH = max(currentHeight1, currentHeight2);
          j++;
       } else {
          if(leftSkyline[i].second >= rightSkyline[j].second) {
             currentHeight1 = leftSkyline[i].second;
             currentHeight2 = rightSkyline[j].second;
             if(maxH != max(currentHeight1, currentHeight2))
                result.push_back(pair<int, int>(leftSkyline[i].first, leftSkyline[i].second));
             maxH = max(currentHeight1, currentHeight2);
             i++;
             j++;
          } else {
             currentHeight1 = leftSkyline[i].second;
             currentHeight2 = rightSkyline[j].second;
             if(maxH != max(currentHeight1, currentHeight2))
                result.push_back(pair<int, int>(rightSkyline[j].first, rightSkyline[j].second));
             maxH = max(currentHeight1, currentHeight2);
             i++;
             j++;
          }
       }
    }
    while (j < rightSkyline.size()) {
       result.push_back(rightSkyline[j]);
       j++;
    }
    while (i != leftSkyline.size()) {
       result.push_back(leftSkyline[i]);
       i++;
    }
    return result;
 }
};
```

Misalnya, diberikan dua skylines A=(a_1, ha_1, a_2, ha_2, ..., an, 0) dan B=(b_1, hb_1, b_2, hb_2, ..., bm, 0), kita menggabungkan daftar ini sebagai daftar baru: (c_1, hc_1, c_2, hc_2, ..., c_{n+m}, 0). Jelas, kita menggabungkan daftar a dan b seperti dalam algoritma Merge standar. Tapi, selain itu, kita harus memutuskan ketinggian yang tepat di antara nilai-nilai batas ini. Kita menggunakan dua variabel currentHeight1 dan currentHeight2 (perhatikan bahwa ini adalah ketinggian sebelum bertemu kepala daftar) untuk menyimpan ketinggian saat ini dari kaki langit pertama dan kedua, masing-masing. Saat membandingkan entri head (currentHeight1, currentHeight2) dari dua skyline, kita memperkenalkan strip baru (dan menambahkan ke skyline *output*) yang koordinat x adalah minimum dari entri koordinat x dan yang tingginya maksimum currentHeight1 dan tinggi saat ini2. Algoritma ini memiliki struktur yang mirip dengan Mergesort. Jadi waktu berjalan keseluruhan dari pendekatan membagi dan menaklukkan akan menjadi O(nlogn).

BAB 11
PEMROGRAMAN DINAMIS

11.1 PENDAHULUAN

Dalam bab ini kita akan mencoba memecahkan masalah yang gagal mendapatkan solusi optimal dengan menggunakan teknik lain (misalnya, metode Divide & Conquer dan *Greedy*). Pemrograman Dinamis (DP) adalah teknik sederhana tetapi sulit untuk dikuasai. Salah satu cara mudah untuk mengidentifikasi dan menyelesaikan masalah DP adalah dengan memecahkan masalah sebanyak mungkin. Istilah Pemrograman tidak terkait dengan pengkodean tetapi dari literatur, dan berarti mengisi tabel (mirip dengan Pemrograman Linier).

11.2 APA ITU STRATEGI PEMROGRAMAN DINAMIS

Pemrograman dinamis dan memoisasi bekerja sama. Perbedaan utama antara pemrograman dinamis dan bagi dan taklukkan adalah bahwa dalam kasus yang terakhir, sub masalah independen, sedangkan dalam DP bisa ada sub masalah yang tumpang tindih. Dengan menggunakan memoisasi [mempertahankan tabel sub masalah yang sudah diselesaikan], pemrograman dinamis mengurangi
kompleksitas eksponensial ke kompleksitas polinomial (O(n2), O(n3), dll.) untuk banyak masalah. Komponen utama DP adalah:

- Rekursi: Memecahkan sub masalah secara rekursif.
- Memoization: Menyimpan nilai yang sudah dihitung dalam tabel (Memoization berarti caching).

Pemrograman Dinamis = Rekursi + Memoisasi

11.3 SIFAT STRATEGI PEMROGRAMAN DINAMIS

Dua properti pemrograman dinamis yang dapat mengetahui apakah itu dapat menyelesaikan masalah yang diberikan atau tidak adalah:

- Substruktur optimal: solusi optimal untuk suatu masalah berisi solusi optimal untuk sub masalah.
- Sub masalah yang tumpang tindih: solusi rekursif berisi sejumlah kecil sub masalah berbeda yang diulang berkali-kali.

11.4 DAPATKAH PEMROGRAMAN DINAMIS MENYELESAIKAN SEMUA MASALAH?

Seperti teknik *Greedy* and *Divide and conquer*, DP tidak dapat menyelesaikan setiap masalah. Ada masalah yang tidak dapat diselesaikan dengan teknik algoritmik apapun [*Greedy*, *Divide and conquer* and *Dynamic Programming*].

Perbedaan antara Pemrograman Dinamis dan rekursi langsung adalah dalam memoisasi panggilan rekursif. Jika sub masalah independen dan tidak ada pengulangan maka memoisasi tidak membantu, sehingga pemrograman dinamis bukanlah solusi untuk semua masalah.

11.5 PENDEKATAN PEMROGRAMAN DINAMIS

Pada dasarnya ada dua pendekatan untuk menyelesaikan masalah DP:

- Pemrograman dinamis dari bawah ke atas
- Pemrograman dinamis *top-down*

Pemrograman Dinamis Bawah-atas

Dalam metode ini, kita mengevaluasi fungsi yang dimulai dengan nilai argumen input terkecil yang mungkin dan kemudian kita melangkah melalui nilai yang mungkin, perlahan-lahan meningkatkan nilai argumen input. Saat menghitung nilai, kita menyimpan semua nilai yang dihitung dalam tabel (memori). Saat argumen yang lebih besar dievaluasi, nilai yang telah dihitung sebelumnya untuk argumen yang lebih kecil dapat digunakan.

Pemrograman Dinamis *Top-down*

Dalam metode ini, masalah dipecah menjadi sub masalah; masing-masing sub masalah ini diselesaikan; dan solusi diingat, jika mereka perlu dipecahkan. Juga, kita menyimpan setiap nilai yang dihitung sebagai tindakan akhir dari fungsi rekursif, dan sebagai tindakan pertama kita memeriksa apakah ada nilai yang dihitung sebelumnya.

Pemrograman Bawah-atas versus Atas-bawah

Dalam pemrograman bottom-up, programmer harus memilih nilai untuk menghitung dan memutuskan urutan perhitungan. Dalam hal ini, semua sub masalah yang mungkin diperlukan diselesaikan terlebih dahulu dan kemudian digunakan untuk membangun solusi untuk masalah yang lebih besar. Dalam pemrograman top-down, struktur rekursif dari kode asli dipertahankan, tetapi perhitungan ulang yang tidak perlu dihindari. Masalah dipecah menjadi sub masalah, sub masalah ini dipecahkan dan solusinya diingat, jika perlu dipecahkan lagi.

Catatan: *Beberapa masalah dapat diselesaikan dengan kedua teknik tersebut dan kita akan melihat contohnya di bagian selanjutnya.*

11.6 CONTOH ALGORITMA PEMROGRAMAN DINAMIS

- Banyak algoritma string termasuk urutan umum terpanjang, urutan peningkatan terpanjang, substring umum terpanjang, jarak edit.
- Algoritma pada Grafik dapat diselesaikan secara efisien: Algoritma Bellman-Ford untuk menemukan jarak terpendek dalam Grafik, algoritma jalur terpendek Floyd's All-Pairs, dll.

- Perkalian matriks rantai
- Jumlah Subset
- 0/1 Ransel
- Masalah salesman keliling, dan masih banyak lagi

11.7 MEMAHAMI PEMROGRAMAN DINAMIS

Sebelum membahas masalah, mari kita pahami cara kerja DP melalui contoh.

Seri Fibonacci

Dalam deret Fibonacci, angka saat ini adalah jumlah dari dua angka sebelumnya. Deret Fibonacci didefinisikan sebagai berikut:

$$\begin{aligned} Fib(n) &= 0, && for\ n = 0 \\ &= 1, && for\ n = 1 \\ &= Fib(n-1) + Fib(n-2), && for\ n > 1 \end{aligned}$$

Implementasi rekursif dapat diberikan sebagai:

```
int RecursiveFibonacci(int n) {
    if(n == 0) return 0;
    if(n == 1) return 1;
    return RecursiveFibonacci(n -1) + RecursiveFibonacci(n -2);
}
```

Memecahkan pengulangan di atas memberikan:

$$T(n) = T(n-1) + T(n-2) + 1 \approx \left(\frac{1+\sqrt{5}}{2}\right)^n \approx 2^n = O(2^n)$$

Catatan: *Sebagai bukti, lihat bab Pendahuluan.*

Bagaimana Memoisasi membantu?

Memanggil fib(5) menghasilkan pohon panggilan yang memanggil fungsi pada nilai yang sama berkali-kali:

fib(5)
fib(4) + *fib*(3)
(*fib*(3) + *fib*(2)) + (*fib*(2) + *fib*(1))
((*fib*(2) + *fib*(1)) + (*fib*(1) + *fib*(0))) + ((*fib*(1) + *fib*(0)) + *fib*(1))
(((*fib*(1) + *fib*(0)) + *fib*(1)) + (*fib*(1) + *fib*(0))) + ((*fib*(1) + *fib*(0)) + *fib*(1))

Dalam contoh di atas, fib(2) dihitung tiga kali (tumpang tindih submasalah). Jika n besar, maka lebih banyak nilai fib (sub masalah) dihitung ulang, yang mengarah ke algoritma waktu eksponensial. Alih-alih memecahkan sub masalah yang sama lagi dan lagi, kita dapat menyimpan nilai yang dihitung sebelumnya dan mengurangi kerumitannya.

Memoisasi bekerja seperti ini: Mulai dengan fungsi rekursif dan tambahkan tabel yang memetakan nilai parameter fungsi ke hasil yang dihitung oleh fungsi. Kemudian jika fungsi ini dipanggil dua kali dengan parameter yang sama, kita cukup mencari jawabannya di tabel.

Meningkatkan: Sekarang, kita melihat bagaimana DP mengurangi kompleksitas masalah ini dari eksponensial menjadi polinomial. Seperti yang telah dibahas sebelumnya, ada dua cara untuk melakukan ini. Salah satu pendekatannya adalah dari bawah ke atas: metode

ini dimulai dengan nilai input yang lebih rendah dan terus membangun solusi untuk nilai yang lebih tinggi.

```
int fib[n];
int fib(int n) {
    // Check for base cases
    if(n == 0 || n == 1) return 1;
    fib[0] = 1;
    fib[1] = 1;
    for (int i = 2; i < n; i++)
        fib[i] = fib[i - 1] + fib[i - 2];
    return fib[n - 1];
}
```

Pendekatan lainnya adalah top-down. Dalam metode ini, kita mempertahankan panggilan rekursif dan menggunakan nilainya jika sudah dihitung. Implementasi untuk ini diberikan sebagai:

```
int fib[n];
int fibonacci( int n ) {
    if(n == 1)
        return 1;
    if(n == 2)
        return 1;
    if( fib[n] != 0) return fib[n] ;
    return fib[n] = fibonacci(n-1) + fibonacci(n -2) ;
}
```

Catatan: *Untuk semua masalah, mungkin tidak mungkin menemukan solusi pemrograman top-down dan bottom-up.*

Kedua versi implementasi deret Fibonacci jelas mengurangi kompleksitas masalah menjadi O(n). Ini karena jika suatu nilai sudah dihitung maka kita tidak akan memanggil subproblem lagi. Sebagai gantinya, kita langsung mengambil nilainya dari tabel.

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n), untuk tabel.

Peningkatan Lebih Lanjut: Satu pengamatan lagi dari deret Fibonacci adalah: Nilai saat ini adalah jumlah dari dua perhitungan sebelumnya saja. Ini menunjukkan bahwa kita tidak harus menyimpan semua nilai sebelumnya. Sebaliknya, jika kita hanya menyimpan dua nilai terakhir, kita dapat menghitung nilai saat ini. Implementasi untuk ini diberikan di bawah ini:

```
int fibonacci(int n) {
    int a = 0, b = 1, sum, i;
    for (i=0;i < n;i++) {
            sum = a + b;
            a = b;
            b = sum;
    }
    return sum;
}
```

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(1).

Catatan: *Metode ini mungkin tidak berlaku (tersedia) untuk semua masalah.*

Pengamatan

Saat memecahkan masalah menggunakan DP, coba cari tahu hal-hal berikut:

- Lihat bagaimana masalah didefinisikan dalam hal submasalah secara rekursif.
- Lihat apakah kita dapat menggunakan beberapa tabel [memoisasi] untuk menghindari perhitungan yang berulang.

Faktorial dari suatu Angka

Sebagai contoh lain, perhatikan masalah faktorial: n! adalah produk dari semua bilangan bulat antara n dan 1. Definisi faktorial rekursif dapat diberikan sebagai:

$$n! = n * (n-1)!$$
$$1! = 1$$
$$0! = 1$$

Definisi ini dapat dengan mudah diubah menjadi implementasi. Di sini masalahnya adalah menemukan nilai n!, dan sub-masalahnya adalah menemukan nilai (n – l)!.

```
int fact(int n) {
    if(n == 1) return 1;
    else if(n == 0) return 1;
    else      // recursive case: multiply n by (n -1) factorial
        return n *fact(n -1);
}
```

Dalam kasus rekursif, ketika n lebih besar dari 1, fungsi memanggil dirinya sendiri untuk mencari nilai (n – l)! dan mengalikannya dengan n. Dalam kasus dasar, ketika n adalah 0 atau 1, fungsi hanya mengembalikan 1. Pengulangan untuk implementasi di atas dapat diberikan sebagai: T(n) = n × T(n – 1) O(n)

Kompleksitas Waktu: O(n). Kompleksitas Ruang: O(n), panggilan rekursif membutuhkan setumpuk ukuran n.

Dalam relasi perulangan dan implementasi di atas, untuk setiap nilai n, tidak ada perhitungan berulang (tidak ada sub masalah yang tumpang tindih) dan fungsi faktorial tidak mendapatkan manfaat apa pun dengan pemrograman dinamis. Sekarang, katakanlah kita ingin menghitung deret m! untuk beberapa nilai arbitrer m. Dengan menggunakan algoritma di atas, untuk setiap panggilan tersebut kita dapat menghitungnya dalam O(m). Misalnya, untuk menemukan keduanya n! dan m! kita dapat menggunakan pendekatan di atas, di mana kompleksitas total untuk menemukan n! dan m! adalah O(m + n).

Kompleksitas Waktu: O(n + m).

Kompleksitas Ruang: O(max(m,n)), panggilan rekursif membutuhkan tumpukan ukuran yang sama dengan maksimum m dan n.

Meningkatkan: Sekarang mari kita lihat bagaimana DP mengurangi kerumitan. Dari definisi rekursif di atas dapat dilihat bahwa fakta(n) dihitung dari fakta(n -1) dan n dan tidak ada yang lain. Alih-alih memanggil fact(n) setiap saat, kita dapat menyimpan nilai yang dihitung sebelumnya dalam sebuah tabel dan menggunakan nilai ini untuk menghitung nilai baru. Implementasi ini dapat diberikan sebagai:

```
int facto[n];
int fact(int n) {
    if(n == 1) return 1;
    else if(n == 0)
          return 1;
    //Already calculated case
    else if(facto[n]!=0)
          return facto[n];
    else      // recursive case: multiply n by (n -1) factorial
    return facto[n]= n *fact(n -1);
}
```

Untuk mempermudah, mari kita asumsikan bahwa kita telah menghitung n! dan ingin menemukan m!. Untuk mencari m!, kita hanya perlu melihat tabel dan menggunakan entri yang ada jika sudah dihitung. Jika m < n maka kita tidak perlu menghitung ulang m!. Jika m > n maka kita dapat menggunakan n! dan panggil faktorial pada nomor yang tersisa saja.

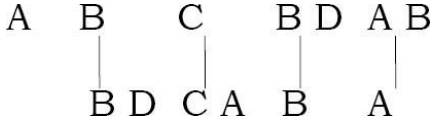
Implementasi di atas jelas mengurangi kompleksitas menjadi O(max(m,n)). Ini karena jika fakta(n) sudah ada, maka kita tidak menghitung ulang nilainya lagi. Jika kita mengisi nilai-nilai yang baru dihitung ini, maka panggilan berikutnya semakin mengurangi kompleksitas. Kompleksitas Waktu: O(maks(m,n)). Kompleksitas Ruang: O(max(m,n)) untuk tabel.

11.8 BARISAN UMUM TERPANJANG

Diberikan dua string: string X dengan panjang m [X(1..m)], dan string Y dengan panjang n [Y(1..n)], carilah turunan umum terpanjang: urutan karakter terpanjang yang muncul di kiri-ke-kanan (tetapi tidak harus dalam blok yang berdekatan) di kedua string. Misalnya, jika X = “ABCBDAB” dan Y = “BDCABA”, LCS(X, Y) = {“BCBA”, “BDAB”, “BCAB”}. Kita dapat melihat ada beberapa solusi optimal.

Pendekatan Brute Force: Satu ide sederhana adalah memeriksa setiap barisan dari X[1.. m] (m adalah panjang barisan X) untuk melihat apakah ia juga merupakan barisan dari Y[1..n] (n adalah panjangnya dari urutan Y). Pengecekan membutuhkan waktu O(n), dan ada 2^m suburutan X. Dengan demikian, waktu berjalan adalah eksponensial O(n. 2^m) dan tidak baik untuk urutan besar.

Solusi Rekursif: Sebelum pergi ke solusi DP, mari kita bentuk solusi rekursif untuk ini dan nanti kita dapat menambahkan memoisasi untuk mengurangi kompleksitas. Mari kita mulai dengan beberapa pengamatan sederhana tentang masalah LCS. Jika kita memiliki dua string, ucapkan "ABCBDAB" dan "BDCABA", dan jika kita menggambar garis dari huruf-huruf di string pertama ke huruf yang sesuai di string kedua, tidak ada dua garis yang bersilangan:



Gambar 11.1 solusi rekursif

Dari pengamatan di atas, kita dapat melihat bahwa karakter X dan Y saat ini mungkin cocok atau tidak. Artinya, misalkan dua karakter pertama berbeda. Maka tidak mungkin

keduanya menjadi bagian dari urutan yang sama - satu atau yang lain (atau mungkin keduanya) harus dihilangkan. Akhirnya, amati bahwa setelah kita memutuskan apa yang harus dilakukan dengan karakter pertama dari string, sub masalah yang tersisa lagi-lagi merupakan masalah LCS, pada dua string yang lebih pendek. Oleh karena itu kita dapat menyelesaikannya secara rekursif.

Solusi untuk LCS harus menemukan dua barisan di X dan Y dan katakanlah indeks awal barisan di X adalah i dan indeks awal barisan di Y adalah j. Juga, asumsikan bahwa X[i ...m] adalah substring dari X yang dimulai dari karakter i dan berlanjut hingga akhir X, dan bahwa Y[j ...n] adalah substring dari Y yang dimulai dari karakter j dan seterusnya sampai akhir Y.

Berdasarkan pembahasan di atas, di sini kita mendapatkan kemungkinan seperti yang dijelaskan di bawah ini:

1) Jika X[i] == Y[j] : 1 + LCS(i + 1,j + 1)
2) Jika X[i] Y[j]. LCS(i,j + 1) // melewatkan karakter ke-j dari Y
3) Jika X[i] Y[j]. LCS(i + 1,j) // melewatkan karakter ke-i dari X

Dalam kasus pertama, jika X[i] sama dengan Y[j], kita mendapatkan pasangan yang cocok dan dapat menghitungnya terhadap panjang total LCS. Jika tidak, kita perlu melewatkan karakter ke-i dari X atau karakter ke-j dari Y dan menemukan turunan umum terpanjang. Sekarang, LCS(i,j) dapat didefinisikan sebagai:

$$LCS(i,j) = \begin{cases} 0, & if\ i = m\ or\ j = n \\ Max\{LCS(i,j+1), LCS(i+1,j)\}, & if\ \text{X[i]} \neq \text{Y[j]} \\ 1 + LCS[i+1, j+1], & if\ \text{X[i]} == \text{Y[j]} \end{cases}$$

LCS memiliki banyak aplikasi. Dalam pencarian web, jika kita menemukan jumlah perubahan terkecil yang diperlukan untuk mengubah satu kata menjadi kata lain. Perubahan di sini adalah penyisipan, penghapusan atau penggantian satu karakter.

Ini adalah solusi yang tepat tetapi sangat memakan waktu. Misalnya, jika dua string tidak memiliki karakter yang cocok, baris terakhir selalu dieksekusi yang memberikan (jika m == n) mendekati $O(2^n)$.

```
//Initial Call: LCSLength(X, 0, m-1, Y, 0, n-1);
int LCSLength( int X[], int i, int m, int Y[], int j, int n) {
    if (i == m || j == n)
        return 0;
    else if (X[i] == Y[j]) return 1 + LCSLength(X, i+1, m, Y, j+1, n);
    else return max( LCSLength(X, i+1, m, Y, j, n),  LCSLength(X, i, m, Y, j+1, n));
}
```

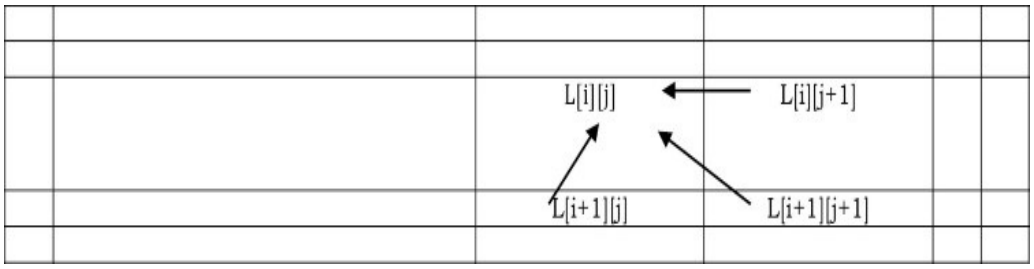
Solusi DP: Menambahkan Memoisasi: Masalah dengan solusi rekursif adalah bahwa submasalah yang sama dipanggil berkali-kali. Subproblem terdiri dari panggilan ke LCS_length, dengan argumen berupa dua sufiks X dan Y, jadi ada persis (i + 1)(j + 1) kemungkinan subproblem (jumlah yang relatif kecil). Jika ada hampir 2^n panggilan rekursif, beberapa submasalah ini harus diselesaikan berulang-ulang.

Solusi DP adalah memeriksa, setiap kali kita ingin menyelesaikan sub masalah, apakah kita sudah melakukannya sebelumnya. Jadi kita mencari solusi daripada menyelesaikannya lagi. Diimplementasikan dengan cara yang paling langsung, kita hanya menambahkan

beberapa kode ke solusi rekursif kita. Untuk melakukan ini, cari kodenya. Ini dapat diberikan sebagai:

```
int LCS[1024][1024];
int LCSLength( int X[], int m, int Y[], int n) {
    for( int i = 0; i <= m; i++ )
        LCS[i][n] = 0;
    for( int j = 0; j <= n; j++ )
        LCS[m][j] = 0;
    for( int i = m – 1; i >= 0; i--) {
        for( int j = n – 1; j >= 0; j--) {
            LCS[i][j] = LCS[i + 1][j + 1]; // matching X[i] to Y[j]
            if( X[i] == Y[j] )
                LCS[i][j]++; // we get a matching pair
            // the other two cases – inserting a gap
            if(LCS[i][j + 1] > LCS[i][j] )
                LCS[i][j] = LCS[i][j + 1];
            if(LCS[i + 1][j] > LCS[i][j] )
                LCS[i][j] = LCS[i + 1][j];
        }
    }
    return LCS[0][0];
}
```

Pertama, urus kasus dasar. Kita telah membuat tabel LCS dengan satu baris dan satu kolom lebih besar dari panjang dua string. Kemudian jalankan loop DP berulang untuk mengisi setiap sel dalam tabel. Ini seperti melakukan rekursi mundur, atau *bottom up*.



Gambar 11.2 Tabel LCS

Nilai LCS[i][j] bergantung pada 3 nilai lainnya (LCS[i+1][j+1], LCS[i][j+1] dan LCS[i+1][j]), semua yang memiliki nilai i atau j yang lebih besar. Mereka melewati tabel dalam urutan penurunan nilai i dan j. Ini akan menjamin bahwa ketika kita perlu mengisi nilai LCS[i][j], kita sudah mengetahui nilai semua sel yang bergantung padanya.

Kompleksitas Waktu: O(mn), karena i mengambil nilai dari 1 hingga m dan dan j mengambil nilai dari 1 hingga n.

Kompleksitas Ruang: O(mn).

Catatan: Pada pembahasan di atas, kita telah mengasumsikan LCS(i,j) adalah panjang LCS dengan X[i ...m] dan Y[j ...n]. Kita dapat menyelesaikan masalah dengan mengubah definisi sebagai LCS(i,j) adalah panjang dari LCS dengan X[1 ...i] dan Y[1...j].

Mencetak urutan berikutnya: Algoritma di atas dapat menemukan panjang dari urutan umum terpanjang tetapi tidak dapat memberikan urutan terpanjang yang sebenarnya. Untuk

mendapatkan urutannya, kita menelusurinya melalui tabel. Mulai dari sel (0,0). Kita tahu bahwa nilai LC5[0][0] adalah nilai maksimum dari 3 sel tetangga. Jadi kita cukup menghitung ulang LC5[0][0] dan perhatikan sel mana yang memberikan nilai maksimum. Kemudian kita pindah ke sel itu (itu akan menjadi salah satu dari (1,1), (0,1) atau (1,0)) dan ulangi ini sampai kita mencapai batas tabel. Setiap kali kita melewati sel (i,j') di mana X[i] == Y[j], kita memiliki pasangan yang cocok dan mencetak X[i]. Pada akhirnya, kita akan mencetak barisan persekutuan terpanjang dalam waktu O(mn).

Cara alternatif untuk mendapatkan jalur adalah dengan menyimpan tabel terpisah untuk setiap sel. Ini akan memberi tahu kita dari arah mana kita berasal saat menghitung nilai sel itu. Pada akhirnya, kita mulai lagi dari sel (0,0) dan ikuti petunjuk ini sampai sudut meja yang berlawanan.

Dari contoh di atas, saya harap Anda memahami ide di balik DP. Sekarang mari kita lihat lebih banyak masalah yang dapat diselesaikan dengan mudah menggunakan teknik DP.

Catatan: *Seperti yang telah kita lihat di atas, dalam DP komponen utamanya adalah rekursi. Jika kita mengetahui pengulangannya, maka mengonversinya menjadi kode adalah tugas yang minimal. Untuk masalah di bawah ini, kita berkonsentrasi untuk mendapatkan pengulangan.*

11.9 PEMROGRAMAN DINAMIS: MASALAH & SOLUSI

Soal-1 Ubah perulangan berikut menjadi kode.

$$T(0) = T(1) = 2$$

$$T(n) = \sum_{i=1}^{n-1} 2 \times \mathrm{T(i)} \times \mathrm{T(i-1)}, \text{for n} > 1$$

Solusi: Kode untuk rumus rekursif yang diberikan dapat diberikan sebagai:

```
int f(int n) {
    int sum = 0;
    if(n==0 || n==1) //Base Case
        return 2;
    //recursive case
    for(int i=1; i < n;i++)
       sum += 2 * f(i) * f(i-1);
    return sum;
}
```

Soal-2 Bisakah kita meningkatkan solusi Soal-1 menggunakan memoisasi DP?

Solusi: Ya. Sebelum menemukan solusi, mari kita lihat bagaimana nilai dihitung.

T(0) = T(1) = 2

T(2) = 2 * T(1) * T(0)

T(3) = 2 * T(1) * T(0) + 2 * T(2) * T(1)

T(4) = 2 * T(1) * T(0) + 2 * T(2) * T(1) + 2 * T(3) * T(2)

Dari perhitungan di atas jelas bahwa ada banyak perhitungan berulang dengan nilai input yang sama. Mari kita gunakan tabel untuk menghindari perhitungan berulang ini, dan implementasinya dapat diberikan sebagai:

```
int f(int n) {
    T[0] = T[1] = 2;
    for(int i=2; i <= n; i++) {
        T[i] = 0;
        for (int j=1; j < i; j++)
            T[i] +=2 * T[j] * T[j-1];
    }
    return T[n];
}
```

Kompleksitas Waktu: O(n^2), dua untuk loop.

Kompleksitas Ruang: O(n), untuk tabel.

Soal-3 Bisakah kita lebih meningkatkan kompleksitas Soal-2?

Solusi: Ya, karena semua perhitungan sub masalah hanya bergantung pada perhitungan sebelumnya, kode dapat dimodifikasi sebagai:

```
int f(int n) {
    T[0] = T[1] = 2;
    T[2] = 2 * T[0] * T[1];
    for(int i=3; i <= n; i++)
        T[i]=T[i-1] + 2 * T[i-1] * T[i-2];
    return T[n];
}
```

Kompleksitas Waktu: O(n), karena hanya satu for loop.

Kompleksitas Ruang: O(n).

Soal-4 Nilai Maksimum Barisan Bersebelahan: Diberikan sebuah larik n bilangan, berikan algoritma untuk menemukan barisan berurutan A(i)... A(j) yang jumlah elemennya maksimum. Contoh: {-2, 11, -4, 13, -5, 2} → 20 dan {1, -3, 4, -2, -1, 6} → 7

Solusi:

Masukan: Array. A(1) ... A(n) dari n angka.

Sasaran: Jika tidak ada bilangan negatif, maka solusinya hanyalah jumlah dari semua elemen dalam larik yang diberikan. Jika ada angka negatif, maka tujuan kita adalah memaksimalkan jumlah [bisa ada angka negatif dalam jumlah yang berdekatan].

Salah satu pendekatan yang sederhana dan kasar adalah melihat semua jumlah yang mungkin dan memilih salah satu yang memiliki nilai maksimum.

```
int MaxContigousSum(int A[], in n) {
    int maxSum = 0;
    for(int i = 0; i < n; i++)                          // for each possible start point
        for(int j = i; j < n; j++)    {                 // for each possible end point
            int currentSum = 0;
            for(int k = i; k <= j; k++)
                currentSum += A[k];
            if(currentSum > maxSum)
                maxSum = currentSum;
        }
    }
    return maxSum;
}
```

Kompleksitas Waktu: $O(n^3)$.
Kompleksitas Ruang: O(1).

Soal-5 Bisakah kita meningkatkan kompleksitas Soal-4?

Solusi: Ya. Satu pengamatan penting adalah, jika kita telah menghitung jumlah dari barisan i,...,j – 1, maka kita hanya perlu satu tambahan lagi untuk mendapatkan jumlah dari barisan i,...,j. Namun, algoritma Problem-4 mengabaikan informasi ini. Jika kita menggunakan fakta ini, kita bisa mendapatkan algoritma yang ditingkatkan dengan waktu berjalan $O(n^2)$.

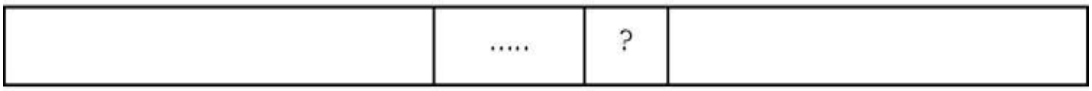
```
int MaxContigousSum(int A[], int n) {
    int maxSum = 0;
    for( int i = 0; i < n;   i++) {
        int currentSum = 0;
        for( int j = i; j < n; j++)    {
            currentSum += a[j];
            if(currentSum > maxSum)
                maxSum = currentSum;
        }
    }
    return maxSum;
}
```

Kompleksitas Waktu: $O(n^2)$.
Kompleksitas Ruang: O(1).

Soal-6 Bisakah kita menyelesaikan Soal-4 menggunakan Pemrograman Dinamis?

Solusi: Ya. Untuk mempermudah, katakanlah, M(i) menunjukkan jumlah maksimum untuk semua jendela yang berakhiran i.

Given Array, A: recursive formula considers the case of selecting i^{th} element



$A[i]$

Untuk menemukan jumlah maksimum kita harus melakukan salah satu dari berikut ini dan memilih maksimum di antara mereka.

- Perpanjang jumlah lama dengan menambahkan A[i]
- atau mulai jendela baru yang dimulai dengan satu elemen A[i]

$$M(i) = Max\begin{cases} M(i-1) + A[i] \\ 0 \end{cases}$$

Dimana, M(i – 1) + A[i] menunjukkan kasus perpanjangan jumlah sebelumnya dengan menambahkan A[i] dan 0 menunjukkan jendela baru yang dimulai dari A[i].

```
int MaxContigousSum(int A[], int n) {
    int M[n] = 0, maxSum = 0;
    if(A[0] > 0)
        M[0] = A[0];
    else M[0] = 0;
    for( int i = 1; i < n; i++) {
        if( M[i-1] + A[i] > 0)
            M[i] = M[i-1] + A[i];
        else    M[i] = 0;
    }
    for( int i = 0; i < n; i++)
        if(M[i] > maxSum)
            maxSum = M[i];
    return maxSum;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n), untuk tabel.

Soal-7 Apakah ada cara lain untuk menyelesaikan Soal-4?

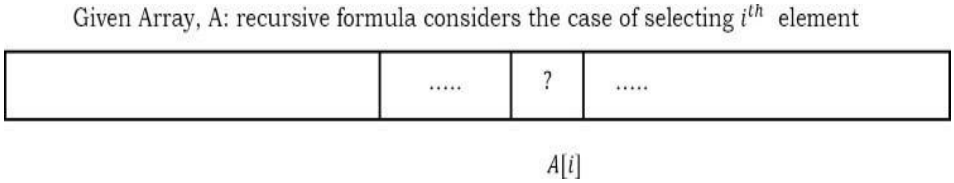
Solusi: Ya. Kita dapat menyelesaikan masalah ini tanpa DP juga (tanpa memori). Algoritmanya sedikit rumit. Salah satu cara sederhana adalah dengan mencari semua segmen bersebelahan positif dari larik (sumEndingHere) dan melacak jumlah segmen bersebelahan maksimum di antara semua segmen positif (sumSoFar). Setiap kali kita mendapatkan jumlah positif, bandingkan (sumEndingHere) dengan sumSoFar dan perbarui sumSoFar jika lebih besar dari sumSoFar. Mari kita perhatikan kode berikut untuk pengamatan di atas.

```
int MaxContigousSum(int A[], int n) {
    int sumSoFar = 0, sumEndingHere = 0;
    for(int i = 0; i < n; i++)    {
        sumEndingHere = sumEndingHere + A[i];
        if(sumEndingHere < 0) {
            sumEndingHere = 0;
            continue;
        }
        if(sumSoFar < sumEndingHere)
            sumSoFar = sumEndingHere;
    }
    return sumSoFar;
}
```

Catatan: *Algoritma tidak berfungsi jika input berisi semua angka negatif. Ini mengembalikan 0 jika semua angka negatif. Untuk mengatasinya, kita bisa menambahkan pemeriksaan ekstra sebelum implementasi yang sebenarnya. Fase akan terlihat jika semua angka negatif, dan jika itu akan mengembalikan maksimum (atau terkecil dalam hal nilai absolut).*
Kompleksitas Waktu: O(n), karena kita hanya melakukan satu pemindaian.
Kompleksitas Ruang: O(1), untuk tabel.

Soal-8 Dalam penyelesaian Soal-7, kita mengasumsikan bahwa M(i) menunjukkan jumlah maksimum untuk semua jendela yang berakhiran i. Bisakah kita berasumsi M(i) menunjukkan jumlah maksimum untuk semua jendela mulai dari i dan berakhir di n?

Solusi: Ya. Untuk mempermudah, katakanlah, M(i) menunjukkan jumlah maksimum untuk semua jendela mulai dari i.



Untuk menemukan jendela maksimum, kita harus melakukan salah satu dari berikut ini dan memilih maksimum di antara mereka.

- Perpanjang jumlah lama dengan menambahkan A[i]
- Atau mulai jendela baru yang dimulai dengan satu elemen A[i]

$$M(i) = Max\begin{cases} M(i+1) + A[i], & if\ M(i+1) + A[i] > 0 \\ 0 \quad , & if\ M(i+1) + A[i] <= 0 \end{cases}$$

Dimana, M(i + 1) + A[t] menunjukkan kasus perpanjangan jumlah sebelumnya dengan menambahkan A[i], dan 0 menunjukkan jendela baru yang dimulai dari A[i].

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n), untuk tabel.
Catatan: *Untuk solusi O(nlogn), lihat bab Divide and conquer.*

Soal-9 Diberikan barisan n bilangan A(1) ...A(n), berikan algoritma untuk mencari barisan berurutan A(i) ...A(j) yang jumlah elemennya adalah maksimum. Di sini syaratnya adalah kita tidak boleh memilih dua angka yang bersebelahan.

Solusi: Mari kita lihat bagaimana DP menyelesaikan masalah ini. Asumsikan bahwa M(i) mewakili jumlah maksimum dari 1 hingga i angka tanpa memilih dua angka yang bersebelahan. Saat menghitung M(i), keputusan yang harus kita buat adalah, apakah akan memilih elemen ke-i atau tidak. Ini memberi kita dua kemungkinan dan berdasarkan ini kita dapat menulis rumus rekursif sebagai:

$$M(i) = \begin{cases} Max\{A[i] + M(i-2), M(i-1)\}, if\ i > 2 \\ A[1], \quad if\ i = 1 \\ Max\{A[1], A[2]\}, \quad if\ i = 2 \end{cases}$$

- Kasus pertama menunjukkan apakah kita memilih elemen ke-i atau tidak. Jika kita tidak memilih elemen ke-i maka kita harus memaksimalkan jumlah menggunakan elemen 1 ke i – 1. Jika elemen ke-i dipilih maka kita tidak harus memilih i – elemen 1 dan perlu memaksimalkan jumlah menggunakan 1 ke i – 2 elemen.
- Dalam representasi di atas, dua kasus terakhir menunjukkan kasus dasar.

Given Array, A: recursive formula considers the case of selecting i^{th} element

A[i-2] A[i-1] A[i]

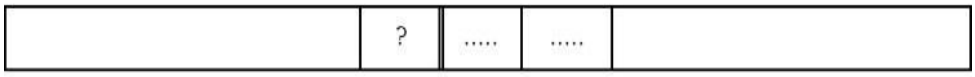
```
int maxSumWithNoTwoContinuousNumbers(int A[], int n) {
    int M[n+1];
    M[0]=A[0];
    M[1]=(A[0]>A[1]?A[0]:A[i]);
    for(i=2, i<n; i++)
        M[i]= (M[i-1]>M[i-2]+A[i]? M[i-1]: M[i-2]+A[i]);
    return M[n-1];
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-10 Dalam Soal-9, kita mengasumsikan bahwa M(i) mewakili jumlah maksimum dari 1 hingga i angka tanpa memilih dua angka yang bersebelahan. Bisakah kita memecahkan masalah yang sama dengan mengubah definisi sebagai: M(i) mewakili jumlah maksimum dari i ke n angka tanpa memilih dua angka yang bersebelahan?

Solusi: Ya. Mari kita asumsikan bahwa M(i) mewakili jumlah maksimum dari i ke n angka tanpa memilih dua angka yang bersebelahan:

Given Array, A: recursive formula considers the case of selecting i^{th} element



A[i] A[i+1] A[i+2]

Seperti solusi Soal-9, kita dapat menulis rumus rekursif sebagai:

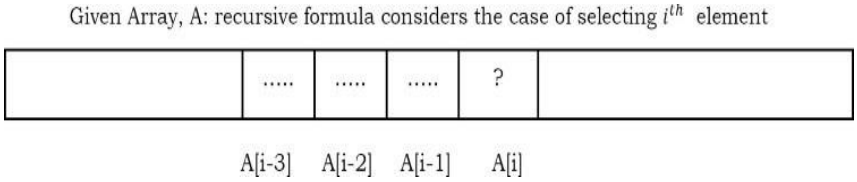
$$M(i) = \begin{cases} Max\{A[i] + M(i+2), M(i+1)\}, if\ i > 2 \\ A[1], \quad if\ i = 1 \\ Max\{A[1], A[2]\}, \quad if\ i = 2 \end{cases}$$

- Kasus pertama menunjukkan apakah kita memilih elemen ke-i atau tidak. Jika kita tidak memilih elemen ke-i maka kita harus memaksimalkan jumlah menggunakan elemen i + 1 hingga n. Jika elemen ke-i dipilih maka kita tidak boleh memilih elemen ke-i + 1 perlu memaksimalkan jumlah menggunakan elemen i + 2 hingga n.
- Dalam representasi di atas, dua kasus terakhir menunjukkan kasus dasar.

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-11 Diberikan barisan n bilangan A(1) ...A(n), berikan algoritma untuk mencari barisan berurutan A(i) ...A(j) yang jumlah elemennya adalah maksimum. Di sini syaratnya adalah kita tidak boleh memilih tiga angka bersambungan.

Solusi: Input: Array A(1) ...A(n) dari n angka.



Asumsikan bahwa M(i) mewakili jumlah maksimum dari 1 hingga i angka tanpa memilih tiga angka yang bersebelahan. Saat menghitung M(i), keputusan yang harus kita buat adalah, apakah akan memilih elemen ke-i atau tidak. Ini memberi kita kemungkinan berikut:

$$M(i) = Max \begin{cases} A[i] + A[i-1] + M(i-3) \\ A[i] + M(i-2) \\ M(i-1) \end{cases}$$

- Pada soal yang diberikan batasannya bukan untuk memilih tiga angka bersambungan, tetapi kita dapat memilih dua elemen secara terus-menerus dan melewatkan yang ketiga. Itulah yang dikatakan kasus pertama dalam rumus rekursif di atas. Itu berarti kita melewatkan A[i – 2].
- Kemungkinan lainnya adalah, memilih elemen ke-i dan melewatkan elemen ke-i kedua – ke-1. Ini adalah kasus kedua (melewatkan A[i – 1]).
- Istilah ketiga mendefinisikan kasus tidak memilih elemen ke-i dan sebagai hasilnya kita harus menyelesaikan masalah dengan elemen i – 1.

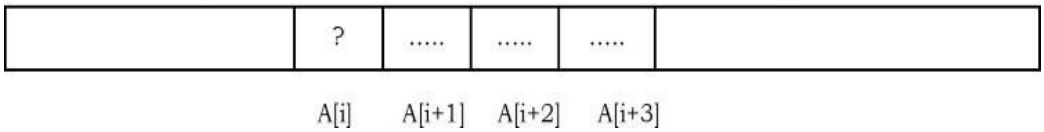
Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(n).

Soal-12 Dalam Soal-11, kita mengasumsikan bahwa M(i) mewakili jumlah maksimum dari 1 hingga i angka tanpa memilih tiga angka yang bersebelahan. Bisakah kita memecahkan masalah yang sama dengan mengubah definisi sebagai: M(i)

mewakili jumlah maksimum dari i ke n angka tanpa memilih tiga angka yang bersebelahan?

Solusi: Ya. Alasannya sangat mirip. Mari kita lihat bagaimana DP memecahkan masalah ini. Asumsikan bahwa M(i) mewakili jumlah maksimum dari i hingga n angka tanpa memilih tiga angka yang bersebelahan.

Given Array, A: recursive formula considers the case of selecting i^{th} element



Saat menghitung M(i), keputusan yang harus kita buat adalah, apakah akan memilih elemen ke-i atau tidak. Ini memberi kita kemungkinan berikut:

$$M(i) = Max \begin{cases} A[i] + A[i+1] + M(i+3) \\ A[i] + M(i+2) \\ M(i+1) \end{cases}$$

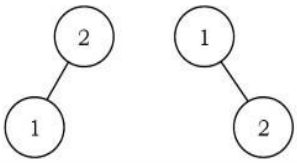
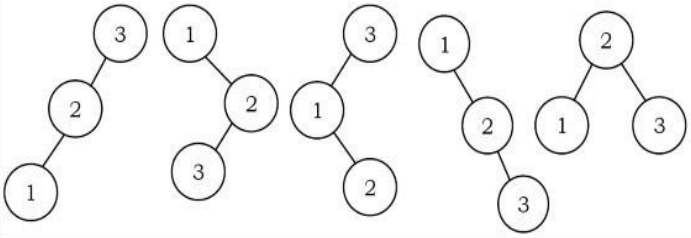
- Pada soal yang diberikan batasannya adalah untuk tidak memilih tiga angka bersambungan, tetapi kita dapat memilih dua elemen secara terus-menerus dan melewatkan yang ketiga. Itulah yang dikatakan kasus pertama dalam rumus rekursif di atas. Itu berarti kita melewatkan A[i + 2].
- Kemungkinan lainnya adalah, memilih elemen ke-i dan melewatkan elemen ke-i kedua – ke-1. Ini adalah kasus kedua (melewatkan A[i + 1]).
- Dan kasus ketiga tidak memilih elemen ke-i dan sebagai hasilnya kita harus menyelesaikan masalah dengan elemen i + 1.

Kompleksitas Waktu: O(n).

Kompleksitas Ruang: O(n).

Soal-13 Bilangan Catalan: Ada berapa pohon pencarian biner dengan n simpul?

Solusi: Binary Search Tree (BST) adalah pohon di mana elemen subtree kiri lebih kecil dari elemen root, dan elemen subtree kanan lebih besar dari elemen root. Properti ini harus dipenuhi di setiap simpul di pohon. Jumlah BST dengan n node disebut Catalan Number dan dilambangkan dengan Cn. Misalnya, ada 2 BST dengan 2 node (2 pilihan untuk root) dan 5 BST dengan 3 node.

Number of nodes, n	Number of Trees
1	
2	
3	

Mari kita asumsikan bahwa simpul pohon diberi nomor dari 1 hingga n. Di antara node, kita harus memilih beberapa node sebagai root, dan kemudian membagi node yang lebih kecil dari root node menjadi subtree kiri, dan elemen yang lebih besar dari root node menjadi subtree kanan. Karena kita telah memberi nomor pada simpul, mari kita asumsikan bahwa elemen akar yang kita pilih adalah elemen ke-i.

Jika kita memilih elemen ke-i sebagai root maka kita mendapatkan i – 1 elemen pada subpohon kiri dan n – i elemen pada subpohon kanan. Karena Cn adalah bilangan Catalan untuk n elemen, C_{i-1} mewakili bilangan Catalan untuk elemen subpohon kiri (i – 1 elemen) dan C_{n-i} mewakili bilangan Catalan untuk elemen subpohon kanan. Kedua sub pohon tidak bergantung satu sama lain, jadi kita cukup mengalikan kedua bilangan tersebut. Artinya, bilangan Catalan untuk nilai i tetap adalah $C_{i-1} \times C_{n-i}$.

Karena ada n node, untuk i kita akan mendapatkan n pilihan. Jumlah total Catalan dengan n node dapat diberikan sebagai:

$$C_n = \sum_{i=1}^{n} C_{i-1} \times C_{n-i}$$

```
int CatalanNumber( int n ) {
    if( n == 0 )
        return 1;
    int count = 0;
    for( int i = 1; i <= n; i++ )
            count += CatalanNumber (i -1) * CatalanNumber (n -i);
    return count;
}
```

Kompleksitas Waktu: O(4n). Untuk bukti, lihat Bab Pendahuluan.

Soal-14 Bisakah kita meningkatkan kompleksitas waktu Soal-13 menggunakan DP?

Solusi: Panggilan rekursif Cn hanya bergantung pada angka C0 hingga C_{n-1} dan untuk setiap nilai i, ada banyak perhitungan ulang. Kita akan menyimpan tabel nilai C_i yang dihitung sebelumnya. Jika fungsi CatalanNumber() dipanggil dengan parameter i, dan jika sudah dihitung sebelumnya, maka kita dapat menghindari penghitungan ulang submasalah yang sama.

Kompleksitas waktu implementasi ini $O(n^2)$, karena untuk menghitung CatalanNumber(n), kita perlu menghitung semua nilai CatalanNumber(i) antara 0 dan n – 1, dan masing-masing akan dihitung tepat satu kali, dalam waktu linier. Dalam matematika, Catalan Number dapat diwakili oleh persamaan langsung sebagai:

```
int Table[1024];
int CatalanNumber( int n ) {
    if( Table[n] ) != 1 )
        return Table[n];
    Table[n] = 0;
    for( int i = 1; i <= n; i++ )
            Table[n] += CatalanNumber( i -1) * CatalanNumber(n -i);
    return Table[n];
}
```

Soal-15 Kurung Produk Matriks: Diberikan deret matriks: $A_1 \times A_2 \times A_3 \times \ldots \times An$ dengan dimensinya, bagaimana cara terbaik untuk mengurungnya sehingga menghasilkan jumlah perkalian total minimum. Asumsikan bahwa kita menggunakan matriks standar dan bukan algoritma perkalian matriks Strassen.

Solusi: Masukan: Barisan matriks $A_1 \times A_2 \times A_3 \times \ldots \times An$, di mana Ai adalah $P_{i-1} \times P_i$. Dimensi diberikan dalam array P.

Sasaran Mengkurung matriks yang diberikan sedemikian rupa sehingga menghasilkan jumlah perkalian optimal yang diperlukan untuk menghitung $A_1 \times A_2 \times A_3 \times \ldots \times$ Sebuah.

Untuk masalah perkalian matriks, ada banyak kemungkinan. Karena perkalian matriks bersifat asosiatif. Tidak masalah bagaimana kita mengkurung produk, hasilnya akan sama. Sebagai contoh, untuk empat matriks A, B, C, dan D, kemungkinannya adalah:

(ABC)D = (AB)(CD) = A(BCD) = A(BC)D =..

Mengalikan matriks (p × q) dengan matriks (q × r) membutuhkan perkalian pqr. Setiap kemungkinan di atas menghasilkan jumlah produk yang berbeda selama perkalian. Untuk memilih yang terbaik, kita dapat melalui setiap tanda kurung (brute force), tetapi ini membutuhkan waktu O(2n) dan sangat lambat. Sekarang mari kita gunakan DP untuk meningkatkan kompleksitas waktu ini.

Asumsikan bahwa, M[i,j] mewakili paling sedikit perkalian yang diperlukan untuk mengalikan $A_i \ldots A_j$.

$$M[i,j] = \begin{cases} 0 & ,if\ i = j \\ Min\{M[i,k] + M[k+1,j] + P_{i-1}P_kP_j\}, & if\ i < j \end{cases}$$

Rumus rekursif di atas mengatakan bahwa kita harus menemukan titik k sedemikian rupa sehingga menghasilkan jumlah perkalian minimum. Setelah menghitung semua nilai yang mungkin untuk k, kita harus memilih nilai k yang memberikan nilai minimum. Kita dapat menggunakan satu tabel lagi (misalnya, S[i,j]) untuk merekonstruksi kurung optimal. Hitung M[i,j] dan S[i,j] secara bottom-up.

```
/* P is the sizes of the matrices, Matrix i has the dimension P[i-1] x P[i].
M[i,j] is the best cost of multiplying matrices i through j
S[i,j] saves the multiplication point and we use this for back tracing */
void MatrixChainOrder(int P[], int length) {
    int n = length – 1, M[n][n], S[n][n];
    for (int i = 1; i <= n; i++)
        M[i][i] = 0;
    // Fills in matrix by diagonals
    for (int l=2; l<= n; l++)   {        // l is chain length
        for (int i=1; i<= n -l+1; i++) {
            int j = i+l-1;
            M[i][j] = MAX_VALUE;
            // Try all possible division points i..k and k..j
            for (int k=i; k<=j-1; k++) {
                int thisCost = M[i][k] + M[k+1][j] + P[i-1]*P[k]*P[j];
                if(thisCost < M[i][j]) {
                    M[i][j] = thisCost;
                    S[i][j] = k;
                }
            }
        }
    }
}
```

Ada berapa sub masalah? Dalam rumus di atas, i dapat berkisar dari 1 hingga n dan j dapat berkisar dari 1 hingga n. Jadi ada total n2 submasalah, dan juga kita melakukan n – 1 operasi tersebut [karena jumlah total operasi yang kita butuhkan untuk A1 × A2 ×A3 ×. . . × Sebuah ise n – 1]. Jadi

Kompleksitas waktunya adalah $O(n^3)$.
Kompleksitas Ruang: $O(n^2)$.

Soal-16 Untuk Soal-15, dapatkah kita menggunakan metode *greedy*?

Solusi: Metode *Greedy* bukanlah cara yang optimal untuk menyelesaikan masalah ini. Mari kita lihat beberapa contoh kontra untuk ini. Seperti yang telah kita lihat, metode serakah membuat keputusan yang baik secara lokal dan tidak

mempertimbangkan solusi optimal di masa depan. Dalam hal ini, jika kita menggunakan *Greedy*, maka kita selalu melakukan perkalian termurah terlebih dahulu. Terkadang mengembalikan tanda kurung yang tidak optimal.

Contoh Perhatikan $A_1 \times A_2 \times A_3$ dengan dimensi 3 × 100, 100 × 2 dan 2 × 2. Berdasarkan *greedy* kita kurung sebagai: $A_1 \times (A_2 \times A_3)$ dengan 100 • 2 • 2 + 3 • 100 • 2 = 1000 perkalian. Tetapi solusi optimal untuk masalah ini adalah: $(A_1 \times A_2) \times A_3$ dengan 3 • 100 • 2 + 3 • 2 • 2 = 612 perkalian. kita tidak bisa menggunakan serakah untuk memecahkan masalah ini.

Soal-17 Integer Knapsack Soal [Duplikat Item Diizinkan]: Diberikan n tipe item, dimana tipe item ke-i memiliki ukuran integer si dan nilai vi. Kita perlu mengisi knapsack dengan kapasitas total C dengan item dengan nilai maksimum. Kita dapat menambahkan beberapa item dengan tipe yang sama ke knapsack.

Solusi *Input*: n jenis item dimana item tipe i memiliki ukuran si dan nilai vi. Juga, asumsikan jumlah item yang tidak terbatas untuk setiap jenis item.

Catatan *Untuk masalah Fractional Knapsack lihat bab Algoritma Greedy.*

Sasaran Isi knapsack dengan kapasitas C dengan menggunakan n jenis item dan dengan nilai maksimum.

Satu catatan penting adalah tidak wajib mengisi knapsack sampai habis. Artinya, mengisi knapsack dengan lengkap [ukuran C] jika kita mendapatkan nilai V dan tanpa mengisi knapsack sepenuhnya [1katakanlah C – 1] dengan nilai U dan jika V < U maka kita pertimbangkan yang kedua. Dalam hal ini, kita pada dasarnya mengisi knapsack ukuran C – 1. Jika kita mendapatkan situasi yang sama untuk C – 1 juga, maka kita mencoba mengisi knapsack dengan ukuran C – 2 dan mendapatkan nilai maksimum.

Katakanlah M(j) menunjukkan nilai maksimum yang dapat kita kemas ke dalam ransel ukuran j. Kita dapat menyatakan M(j) secara rekursif dalam hal solusi untuk sub masalah sebagai berikut:

$$M(j) = \begin{cases} max\{M(j-1), max_{i=1\ to\ n}(M(j-s_i)) + v_i\}, & if\ j \geq 1 \\ 0, & if\ j \leq 0 \end{cases}$$

Untuk masalah ini keputusannya tergantung pada apakah kita memilih item ke-i tertentu atau tidak untuk knapsack ukuran j.

- Jika kita memilih item ke-i, maka kita menambahkan nilainya vi ke solusi optimal dan mengurangi ukuran knapsack yang akan diselesaikan menjadi j – s_i.
- Jika kita tidak memilih item tersebut maka periksa apakah kita bisa mendapatkan solusi yang lebih baik untuk knapsack ukuran j – 1.

Nilai M(C) akan berisi nilai solusi optimal. Kita dapat menemukan daftar item dalam solusi optimal dengan mempertahankan dan mengikuti "back pointer".
Kompleksitas Waktu: Menemukan setiap nilai M(j) akan membutuhkan waktu Θ (n), dan kita perlu menghitung C nilai-nilai tersebut secara berurutan. Oleh karena itu, total waktu berjalan adalah Θ (nC).
Kompleksitas Ruang: (C).

Soal-18 0-1 Knapsack Soal: Untuk Soal-17, bagaimana kita menyelesaikannya jika item tidak digandakan (tidak memiliki jumlah item yang tidak terbatas untuk setiap jenis, dan setiap item diperbolehkan untuk digunakan untuk 0 atau 1 kali)?
Contoh real-time: Misalkan kita pergi dengan penerbangan, dan kita tahu bahwa ada batasan berat bagasi. Juga, barang-barang yang kita bawa bisa dari berbagai jenis (seperti laptop, dll.). Dalam hal ini, tujuan kita adalah untuk memilih item dengan nilai maksimum. Artinya, kita perlu memberitahu petugas bea cukai untuk memilih barang yang memiliki bobot lebih dan nilai lebih kecil (keuntungan).

Solusi: Input adalah sekumpulan n item dengan ukuran si dan nilai vi dan sebuah Knapsack berukuran C yang perlu kita isi dengan subset item dari set yang diberikan. Mari kita coba mencari rumus rekursif untuk masalah ini menggunakan DP. Misalkan M(i,j) mewakili nilai optimal yang bisa kita peroleh untuk mengisi ransel ukuran j dengan item 1... i. Rumus rekursif dapat diberikan sebagai:

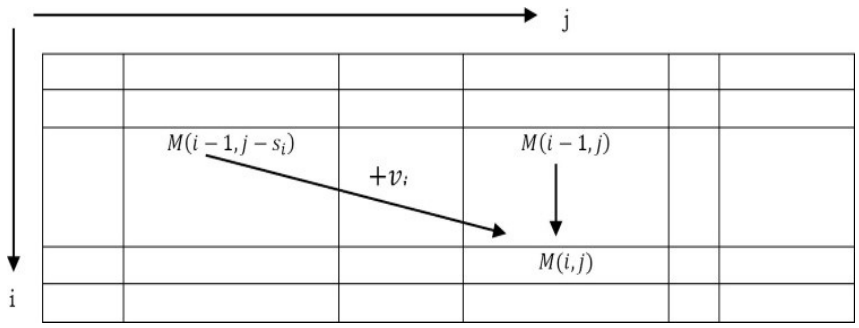
$$M(i,j) = Max\{M(i-1,j), M(i-1,j-s_i) + v_i\}$$

i^{th} item is not used

i^{th} item is used

Kompleksitas Waktu: O(nC), karena ada submasalah nC yang harus diselesaikan dan masing-masing submasalah membutuhkan O(1) untuk dihitung. Kompleksitas Ruang: O(nC), sedangkan Integer Knapsack hanya membutuhkan O(C).

Sekarang mari kita perhatikan diagram berikut yang membantu kita dalam merekonstruksi solusi optimal dan juga memberikan pemahaman lebih lanjut. Ukuran matriks di bawah adalah M.



Karena i mengambil nilai dari 1 ...n dan j mengambil nilai dari 1... C, ada total submasalah nC. Sekarang mari kita lihat apa yang dikatakan rumus di atas:

- M(i – 1,j): Menunjukkan kasus tidak memilih item ke-i. Dalam hal ini, karena kita tidak menambahkan ukuran apa pun ke knapsack, kita harus menggunakan ukuran knapsack yang sama untuk submasalah tetapi mengecualikan item ke-i. Item yang tersisa adalah i – 1.
- $M(i – 1,j – s_i) + v_i$ menunjukkan kasus di mana kita telah memilih item ke-i. Jika kita menambahkan item ke-i maka kita harus mengurangi ukuran knapsack subproblem menjadi j – si dan pada saat yang sama kita perlu menambahkan nilai vi ke solusi optimal. Item yang tersisa adalah i – 1.

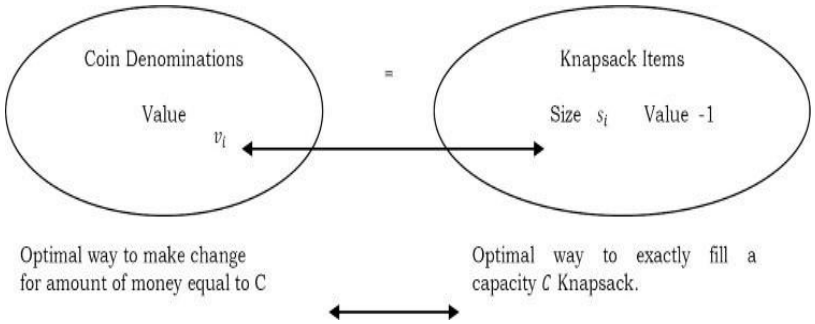
Sekarang, setelah menemukan semua nilai M(i,j), nilai tujuan optimal dapat diperoleh sebagai:

Maksj{M(n,j)} Ini karena kita tidak tahu berapa jumlah kapasitas yang memberikan solusi terbaik.

Untuk menghitung beberapa nilai M(i,j), kita mengambil maksimum M(i – 1,j) dan $M(i – 1,j – s_i) + vi$. Kedua nilai ini (M(i,j) dan $M(i – 1,j – s_i)$) muncul di baris sebelumnya dan juga di beberapa kolom sebelumnya. Jadi, M(i,j) dapat dihitung hanya dengan melihat dua nilai pada baris sebelumnya dalam tabel.

Soal-19 Membuat Perubahan: Diberikan n jenis nilai pecahan koin $v_1 < v_2 <...< v_n$ (bilangan bulat). Asumsikan $v_1 = 1$, sehingga kita selalu dapat membuat perubahan untuk sejumlah uang C. Berikan algoritma yang membuat perubahan untuk sejumlah uang C dengan koin sesedikit mungkin.

Solusi: Masalah ini identik dengan masalah Integer Knapsack. Dalam masalah kita, kita memiliki denominasi koin, masing-masing bernilai vi. Kita dapat membangun sebuah instance dari masalah Knapsack untuk setiap item yang memiliki ukuran si, yang sama dengan nilai denominasi vi koin. Dalam Knapsack kita dapat memberikan nilai setiap item sebagai -1.



Sekarang mudah untuk memahami cara optimal untuk menghasilkan uang C dengan koin paling sedikit sama dengan cara optimal untuk mengisi Knapsack ukuran C. Ini karena karena setiap nilai memiliki nilai -1, dan algoritma Knapsack menggunakan item sesedikit mungkin yang sesuai dengan koin sesedikit mungkin.

Mari kita coba merumuskan perulangan. Misalkan M(j) menunjukkan jumlah uang logam minimum yang diperlukan untuk membuat perubahan untuk jumlah uang yang sama dengan j.

$$M(j) = Mini\{M(j - v_j)\} + 1$$

Artinya, jika pecahan koin i adalah pecahan terakhir yang ditambahkan ke dalam solusi, maka cara optimal untuk menyelesaikan solusi tersebut adalah dengan secara optimal membuat perubahan untuk jumlah uang $j - v_i$ dan kemudian menambahkan satu koin tambahan sebesar nilai v_i.

```
int Table[128] ; //Initialization
int MakingChange(int n) {
    if(n < 0) return -1;
    if(n == 0)
            return 0;
    if(Table[n] != -1)
            return Table[n];
    int ans = -1;
    for ( int i = 0 ; i < num_denomination ; ++i )
            ans = Min( ans , MakingChange(n - denominations [ i ] ) ) ;

    return Table[ n ] = ans + 1 ;
}
```

Kompleksitas Waktu: O(nC). Karena kita memecahkan sub-masalah C dan masing-masing sub-masalah membutuhkan minimalisasi n suku.

Kompleksitas Ruang: O(nC).

Soal-20 **Barisan Bertambah Terpanjang:** Diberikan barisan n bilangan A_1 . . . A_n, tentukan suatu barisan (tidak harus bersebelahan) dengan panjang maksimum di mana nilai-nilai dalam barisan tersebut membentuk barisan yang meningkat secara ketat.

Solusi:

Masukan: Urutan n angka $A_1 \ldots A_n$.

Tujuan: Untuk menemukan barisan yang hanya merupakan himpunan bagian dari elemen dan tidak kebetulan bersebelahan. Tetapi elemen-elemen di urutan berikutnya harus membentuk urutan yang meningkat secara ketat dan pada saat yang sama urutannya harus mengandung elemen sebanyak mungkin.

Misalnya, jika barisannya adalah (5,6,2,3,4,1,9,9,8,9,5), maka (5,6), (3,5), (1,8,9) adalah semua sub-urutan meningkat. Yang terpanjang adalah (2,3,4,8,9), dan kita ingin algoritma untuk menemukannya.

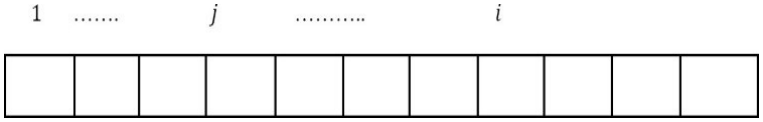
Pertama, mari kita berkonsentrasi pada algoritma untuk menemukan urutan terpanjang. Kemudian, kita dapat mencoba mencetak urutan itu sendiri dengan menelusuri tabel. Langkah pertama kita adalah menemukan rumus rekursif.

Pertama, mari kita buat kondisi dasar. Jika hanya ada satu elemen dalam urutan input maka kita tidak perlu menyelesaikan masalah dan kita hanya perlu mengembalikan elemen itu. Untuk urutan apa pun kita bisa mulai dengan elemen pertama (A[1]). Karena kita mengetahui bilangan pertama dalam LIS, mari kita cari bilangan kedua (A[2]). Jika A[2] lebih besar dari A[1] maka sertakan juga A[2]. Jika tidak, kita selesai - LIS adalah urutan satu elemen (A[1]).

Sekarang, mari kita menggeneralisasi diskusi dan memutuskan tentang elemen ke-i. Misalkan L(i) mewakili barisan optimal yang dimulai dari posisi A[1] dan berakhir di A[i]. Cara optimal untuk mendapatkan barisan yang meningkat secara ketat yang berakhir pada posisi i adalah dengan memperpanjang beberapa barisan yang dimulai pada beberapa posisi sebelumnya j. Untuk ini rumus rekursif dapat ditulis sebagai:

$$L(i) = Maks_{j < i \text{ dan } A[j] < A[i]} \{L(j)\} + 1$$

Pengulangan di atas mengatakan bahwa kita harus memilih beberapa posisi awal j yang memberikan urutan maksimum. 1 dalam rumus rekursif menunjukkan penambahan elemen ke-i.



Sekarang setelah menemukan urutan maksimum untuk semua posisi kita harus memilih satu di antara semua posisi yang memberikan urutan maksimum dan didefinisikan sebagai:

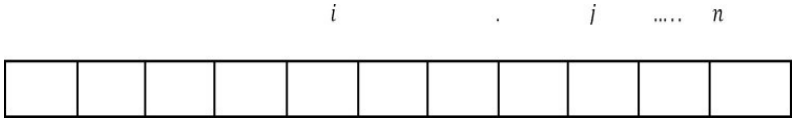
$$Max_i\{L(i)\}$$

```
int LISTable [1024];
int LongestIncreasingSequence( int A[], int n ) {
    int i, j, max = 0;
    for ( i = 0; i < n; i++ )
        LISTable[i] = 1;
    for ( i = 0; i< n; i++ ) {
        for ( j = 0; j < i; j++ ) {
            if( A[i] > A[j] && LISTable[i] < LISTable[j] + 1 )
                LISTable[i] = LISTable[j] + 1;
        }
    }
    for ( i = 0; i < n; i++) {
        if( max < LISTable[i] )
            max = LISTable[i];
    }
    return max;
}
```

Kompleksitas Waktu: $O(n^2)$, karena dua for loop.
Kompleksitas Ruang: O(n), untuk tabel.

Soal-21 Barisan Kenaikan Terpanjang: Pada Soal-20, kita asumsikan bahwa L(i) mewakili barisan optimal yang dimulai dari posisi A[1] dan berakhir di A[i]. Sekarang, mari kita ubah definisi L(i) menjadi: L(i) mewakili barisan optimal yang dimulai dari posisi A[i] dan berakhir di A[n]. Dengan pendekatan ini, bisakah kita menyelesaikan masalah?

Solusi: Ya.



Misalkan L(i) mewakili barisan optimal yang dimulai dari posisi A[i] dan berakhir di A[n]. Cara optimal untuk mendapatkan urutan yang meningkat secara ketat mulai dari posisi i adalah dengan memperpanjang beberapa urutan yang dimulai pada beberapa posisi kemudian j. Untuk ini rumus rekursif dapat ditulis sebagai:

$$L(i) = Maks_{j < i \text{ dan } A[j] < A[i]} \{L(j)\} + 1$$

Kita harus memilih beberapa posisi kemudian j yang memberikan urutan maksimum. 1 dalam rumus rekursif adalah penambahan elemen ke-i. Setelah menemukan urutan maksimum untuk semua posisi pilih satu di antara semua posisi yang memberikan urutan maksimum dan didefinisikan sebagai:

$$Max_i\{L(i)\}$$

```
int LISTable [1024];
int LongestIncreasingSequence( int A[], int n ) {
    int i, j, max = 0;
    for ( i = 0; i < n; i++ )
        LISTable[i] = 1;
    for(i = n – 1; i >= 0; i++) {
        // try picking a larger second element
        for( j = i + 1; j < n; j++ ) {
            if( A[i] < A[j] && LISTable [i] < LISTable [j] + 1)
                LISTable[i] = LISTable[j] + 1;
        }
    }
    for ( i = 0; i < n; i++ ) {
        if( max < LISTable[i] )
            max = LISTable[i];
    }
    return max;
}
```

Kompleksitas Waktu: $O(n^2)$ karena dua loop for bersarang.
Kompleksitas Ruang: O(n), untuk tabel.

Soal-22 Apakah ada cara alternatif untuk menyelesaikan Soal-21?

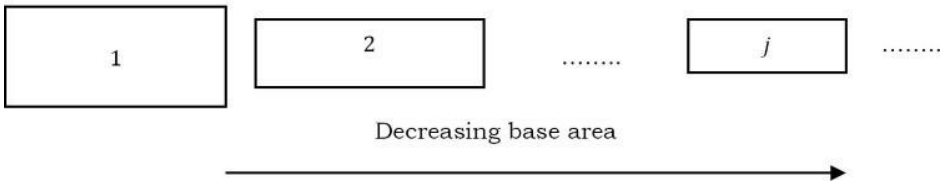
Solusi: Ya. Metode lainnya adalah mengurutkan urutan yang diberikan dan menyimpannya ke dalam array lain dan kemudian mengambil "Longest Common Subsequence" (LCS) dari dua array. Metode ini memiliki kompleksitas O(n2). Untuk masalah LCS lihat bagian teori bab ini.

Soal-23 Penumpukan Kotak: Asumsikan bahwa kita diberikan satu set n kotak persegi panjang 3 – D. Dimensi kotak ke-i adalah tinggi hi, lebar wi dan kedalaman di. Sekarang kita ingin membuat tumpukan kotak yang setinggi mungkin, tetapi kita hanya dapat menumpuk sebuah kotak di atas kotak lain jika dimensi dasar 2 –D dari kotak bawah masing-masing benar-benar lebih besar daripada dimensi 2 –D dasar kotak yang lebih tinggi. Kita dapat memutar kotak sehingga setiap sisi berfungsi sebagai alasnya. Dia mungkin untuk menggunakan beberapa contoh dari jenis kotak yang sama.

Solusi: Masalah penumpukan kotak dapat dikurangi menjadi LIS [Soal-21].

Input: n kotak di mana i dengan tinggi hai, lebar wi dan kedalaman di. Untuk semua n kotak kita harus mempertimbangkan semua orientasi terhadap rotasi. Artinya, jika kita memiliki, dalam himpunan aslinya, sebuah kotak dengan dimensi 1 × 2 × 3, maka kita mempertimbangkan 3 kotak,

$$1 \times 2 \times 3 \Rightarrow \begin{cases} 1 \times (2 \times 3), with\ height\ 1, base\ 2\ and\ width\ 3 \\ 2 \times (1 \times 3), with\ height\ 2, base\ 1\ and\ width\ 3 \\ 3 \times (1 \times 2), with\ height\ 3, base\ 1\ and\ width\ 2 \end{cases}$$



Penyederhanaan ini memungkinkan kita untuk melupakan rotasi kotak dan kita hanya fokus pada penumpukan n kotak dengan tinggi masing-masing sebagai hi dan luas dasar (w_i x d_i). Juga asumsikan bahwa $w_i \leq d_i$. Sekarang yang kita lakukan adalah membuat tumpukan kotak yang setinggi mungkin dan memiliki tinggi yang maksimal. Kita mengizinkan kotak i di atas kotak j hanya jika kotak i lebih kecil dari kotak j di kedua dimensi. Artinya, jika $w_i < w_j$ && $d_i < d_j$. Sekarang mari kita selesaikan ini menggunakan DP. Pertama pilih kotak dalam urutan penurunan luas dasar.

Sekarang, katakanlah H(j) mewakili tumpukan kotak tertinggi dengan kotak j di atasnya. Hal ini sangat mirip dengan masalah LIS karena tumpukan n kotak dengan akhiran kotak j sama dengan mencari barisan dengan kotak j pertama karena pengurutan dengan mengecilkan luas alas. Urutan kotak di tumpukan akan sama dengan urutan urutan.

Sekarang kita dapat menulis H(j) secara rekursif. Untuk membentuk tumpukan yang berakhir di kotak j, kita perlu memperluas tumpukan sebelumnya yang berakhir di i. Itu berarti, kita perlu meletakkan j box di bagian atas tumpukan [i box adalah bagian atas tumpukan saat ini]. Untuk menempatkan kotak j di atas tumpukan kita harus memenuhi kondisi $w_i > w_j$ dan $d_i > d_j$ [ini memastikan bahwa kotak tingkat rendah memiliki alas lebih banyak daripada kotak di atasnya]. Berdasarkan logika ini, kita dapat menulis rumus rekursif sebagai:

$$H(j) = Max_{i<j\ and\ \mathrm{w_i}>\mathrm{w_j} \mathrm{and}\ \mathrm{d_i}>\mathrm{d_j}} \{H(i)\} + h_i$$

Mirip dengan masalah LIS, pada akhirnya kita harus memilih j terbaik dari semua nilai potensial. Ini karena kita tidak yakin kotak mana yang mungkin berakhir di atas.

$$Max_j\{H(j)\}$$

Kompleksitas Waktu: $O(n^2)$.

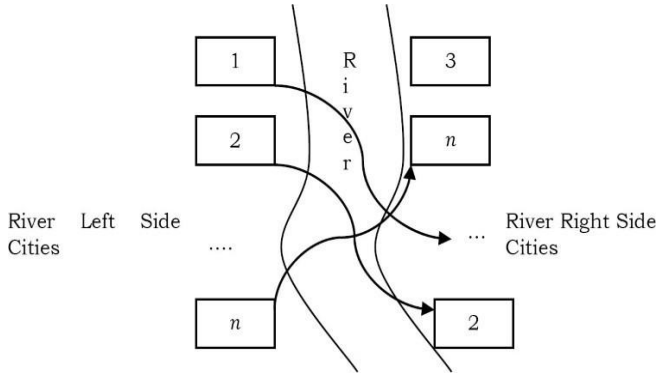
Soal-24 Membangun Jembatan di India: Perhatikan sebuah sungai lurus yang sangat panjang yang bergerak dari utara ke selatan. Asumsikan ada n kota di kedua sisi sungai: n kota di sebelah kiri sungai dan n kota di sebelah kanan sungai. Juga, asumsikan bahwa kota-kota ini diberi nomor dari 1 hingga n tetapi urutannya tidak diketahui. Sekarang kita ingin menghubungkan sebanyak kiri-pasangan

kota yang tepat mungkin dengan jembatan sedemikian rupa sehingga tidak ada dua jembatan yang saling bersilangan. Saat menghubungkan kota, kita hanya dapat menghubungkan kota i di sisi kiri ke kota i di sisi kanan.

Solusi:

Masukan: Dua pasang himpunan dengan masing-masing bernomor dari 1 sampai n.

Tujuan: Bangun jembatan sebanyak mungkin tanpa persilangan antara kota sisi kiri ke kota sisi kanan sungai.



Untuk lebih memahami mari kita perhatikan diagram di bawah ini. Pada diagram dapat dilihat bahwa terdapat n kota di sebelah kiri sungai dan n kota di sebelah kanan sungai. Juga, perhatikan bahwa kita menghubungkan kota-kota yang memiliki nomor yang sama [persyaratan dalam masalah]. Tujuan kita adalah untuk menghubungkan kota-kota maksimum di sisi kiri sungai ke kota-kota di sisi kanan sungai, tanpa tepi yang bersilangan. Sederhananya, mari kita urutkan kota-kota di satu sisi sungai.

Jika kita amati dengan cermat, karena kota-kota di sebelah kiri sudah diurutkan, masalahnya dapat disederhanakan untuk menemukan barisan kenaikan maksimum. Itu berarti kita harus menggunakan solusi LIS untuk menemukan barisan kenaikan maksimum di kota-kota sisi kanan sungai.

Kompleksitas Waktu: $O(n^2)$, (sama seperti LIS).

Soal-25 Jumlah Subset: Diberikan barisan n bilangan positif A1 . . . An, berikan algoritma yang memeriksa apakah ada himpunan bagian dari A yang jumlah semua bilangannya adalah T?

Solusi: Ini adalah variasi dari masalah Knapsack. Sebagai contoh, perhatikan array berikut:

A = [3,2,4,19,3,7,13,10,6,11]

Misalkan kita ingin memeriksa apakah ada himpunan bagian yang jumlahnya 17. Jawabannya ya, karena jumlah 4 + 13 = 17 dan oleh karena itu {4,13} adalah himpunan bagian tersebut.

Mari kita coba selesaikan masalah ini menggunakan DP. Kita akan mendefinisikan matriks n × T, di mana n adalah jumlah elemen dalam array input kita dan T adalah jumlah yang ingin kita periksa.
Misal, M[i,j] = 1 jika dimungkinkan untuk mencari himpunan bagian dari bilangan 1 sampai i yang menghasilkan jumlah/ dan M[i,j] = 0 sebaliknya.

$$M[i, j] = Maks(M[i – 1,j], M[i – 1, j – A_i])$$

Menurut rumus rekursif di atas yang mirip dengan masalah Knapsack, kita memeriksa apakah kita bisa mendapatkan jumlah j dengan tidak memasukkan elemen i di subset kita, dan kita memeriksa apakah kita bisa mendapatkan jumlah j dengan memasukkan i dan memeriksa apakah jumlah j – Ai ada tanpa elemen ke-i. Ini identik dengan Knapsack, kecuali bahwa kita menyimpan 0/1 bukan nilai. Dalam implementasi di bawah ini kita dapat menggunakan operasi biner OR untuk mendapatkan nilai maksimum antara M[i – 1,j] dan M[i – 1,j – A_i].

```
int SubsetSum( int A[], int n, int  T ) {
    int i, j, M[n+1][T +1];
    M[0][0]=0;
    for (i=1; i<= T; i++)
        M[0][i]= 0;
    for (i=1; i<=n; i++) {
        for (j = 0; j<= T; j++) {
            M[i][j] = M[i-1][j] || M[i-1][j - A[i]];
        }
    }
    return M[n][T];
}
```

Ada berapa submasalah? Dalam rumus di atas, i dapat berkisar dari 1 hingga n dan j dapat berkisar dari l hingga T. Ada total nT submasalah dan masing-masing membutuhkan O(1). Jadi kompleksitas waktu adalah O(nT) dan ini bukan polinomial karena waktu berjalan bergantung pada dua variabel [n dan T], dan kita dapat melihat bahwa keduanya merupakan fungsi eksponensial dari variabel lainnya.
Kompleksitas Ruang: O(nT).

Soal-26 Diberikan himpunan n bilangan bulat dan jumlah semua bilangan paling banyak jika. Temukan himpunan bagian dari n elemen ini yang jumlahnya tepat setengah dari jumlah total n bilangan.

Solusi: Asumsikan bahwa angka-angkanya adalah $A_1 \ldots A_n$. Mari kita gunakan DP untuk menyelesaikan masalah ini. Kita akan buat array boolean T dengan ukuran sama

dengan K + 1. Asumsikan bahwa T[x] adalah 1 jika terdapat subset dari n elemen yang jumlahnya x. Artinya, setelah algoritma selesai, T[K] akan menjadi 1, jika dan hanya jika ada himpunan bagian dari bilangan yang memiliki jumlah K. Setelah kita mendapatkan nilai tersebut maka kita hanya perlu mengembalikan T[K/2]. Jika 1, maka ada subset yang menambahkan hingga setengah jumlah total.

Awalnya kita set semua nilai T ke 0. Kemudian kita set T[0] ke 1. Ini karena kita selalu bisa membangun 0 dengan mengambil set kosong. Jika kita tidak memiliki angka di A, maka kita selesai! Jika tidak, kita memilih nomor pertama, A[0]. Kita bisa membuangnya atau memasukkannya ke dalam subset kita. Ini berarti bahwa T[] baru harus memiliki T[0] dan T[A[0]] disetel ke 1. Ini menciptakan kasus dasar. Kita melanjutkan dengan mengambil elemen berikutnya dari A.

Misalkan kita telah menangani elemen i – 1 pertama dari A. Sekarang kita ambil A[i] dan lihat tabel kita T[]. Setelah memproses elemen i – 1, array T memiliki 1 di setiap lokasi yang sesuai dengan jumlah yang dapat kita buat dari angka yang telah kita proses. Sekarang kita tambahkan nomor baru, A[i]. Seperti apa seharusnya tabel itu? Pertama-tama, kita bisa mengabaikan A[i]. Itu berarti, tidak ada yang harus menghilang dari T[] - kita masih bisa membuat semua jumlah itu. Sekarang perhatikan beberapa lokasi T[j] yang memiliki 1 di dalamnya. Ini sesuai dengan beberapa subset dari angka sebelumnya yang berjumlah j. Jika kita menambahkan A[i] ke subset itu, kita akan mendapatkan subset baru dengan jumlah total j + A[i]. Jadi kita harus mengatur T[j + A[i]] ke 1 juga. Itu saja. Berdasarkan pembahasan di atas, kita dapat menulis algoritma sebagai:

```
bool T[10240];
bool SubsetHalfSum( int  A[], int n ) {
    int K = 0;
    for( int i = 0; i < n; i++ )
        K += A[i];
    T[0] = 1;           // initialize the table
    for( int i = 1; i <= K; i++ )
        T[i] = 0;
    // process the numbers one by one
    for( int i = 0; i < n; i++ ) {
        for( int j = K - A[i]; j >= 0; j--)     {
            if( T[j] )
                T[j + A[i]] = 1;
        }
    }
    return T[K / 2];
}
```

Dalam kode di atas, j loop bergerak dari kanan ke kiri. Ini mengurangi masalah penghitungan ganda. Artinya, jika kita bergerak dari kiri ke kanan, maka kita dapat melakukan perhitungan berulang.

Kompleksitas Waktu: O(nK), untuk dua loop for.

Kompleksitas Ruang: O(K), untuk tabel boolean T.

Soal-27 Bisakah kita meningkatkan kinerja Soal-26?

Solusi: Ya. Dalam kode di atas yang kita lakukan adalah, loop j bagian dalam dimulai dari K dan bergerak ke kiri. Itu berarti, tidak perlu memindai seluruh tabel setiap saat.

Apa yang sebenarnya kita inginkan adalah menemukan semua 1 entri. Pada awalnya, hanya entri ke 0 yang bernilai 1. Jika kita menyimpan lokasi entri 1 paling kanan dalam sebuah variabel, kita selalu dapat mulai di tempat itu dan ke kiri alih-alih mulai dari ujung kanan tabel.

Untuk memanfaatkan ini sepenuhnya, kita dapat mengurutkan A[] terlebih dahulu. Dengan begitu, 1 entri paling kanan akan bergerak ke kanan sepelan mungkin. Akhirnya, kita tidak terlalu peduli dengan apa yang terjadi di bagian kanan tabel (setelah T[K/2]) karena jika T[x] adalah 1, maka T[Kx] akhirnya juga harus 1 – ini sesuai dengan komplemen dari subset yang memberi kita x. Kode berdasarkan diskusi di atas diberikan di bawah ini.

```
int T[10240];
int SubsetHalfSumEfficient( int  A[], int n ) {
    int K = 0;
    for( int i = 0; i < n; i++ )
        K += A[i];
    Sort(A,n));
    T[0] = 1;           // initialize the table
    for( int i = 1; i <= sum; i++ )
        T[i] = 0;
    int R = 0; // rightmost 1 entry
    for( int i = 0; i < n; i++)  {          // process the numbers one by one
        for( int j = R; j >= 0; j--) {
            if( T[j] )
                T[j + A[i]] = 1;
            R = min(K / 2, R + C[i] );
        }
    }
    return T[K / 2];
}
```

Setelah perbaikan, kompleksitas waktu masih O(nK), tetapi kita telah menghapus beberapa langkah yang tidak berguna.

Soal-28 Masalah partisi partisi adalah untuk menentukan apakah suatu himpunan dapat dipartisi menjadi dua himpunan bagian sehingga jumlah elemen pada kedua himpunan bagian adalah sama [sama dengan soal sebelumnya tetapi

cara menanyakannya berbeda]. Misalnya, jika A[] = {1, 5, 11, 5}, array dapat dipartisi sebagai {1, 5, 5} dan {11}. Demikian pula, jika A[] = {1, 5, 3}, array tidak dapat dipartisi menjadi kumpulan jumlah yang sama.

Solusi: Mari kita coba memecahkan masalah ini dengan cara lain. Berikut adalah dua langkah utama untuk mengatasi masalah ini:

1. Hitung jumlah array. Jika jumlahnya ganjil, tidak mungkin ada dua himpunan bagian dengan jumlah yang sama, jadi kembalikan false.
2. Jika jumlah elemen larik genap, hitung jumlah/2 dan temukan himpunan bagian dari larik dengan jumlah sama dengan jumlah/2.

Langkah pertama sederhana. Langkah kedua sangat penting, dan dapat diselesaikan baik menggunakan rekursi atau Pemrograman Dinamis.

Solusi Rekursif: Berikut ini adalah properti rekursif dari langkah kedua yang disebutkan di atas. Misalkan subsetSum(A, n, sum/2) adalah fungsi yang mengembalikan nilai true jika ada subset dari A[0..n-1] dengan jumlah sama dengan jumlah/2. Masalah isSubsetSum dapat dibagi menjadi dua sub masalah:

a) isSubsetSum() tanpa mempertimbangkan elemen terakhir (mengurangi n menjadi n – 1)

b) isSubsetSum mempertimbangkan elemen terakhir (mengurangi jumlah/2 oleh A[n-1] dan n menjadi n – 1)

Jika salah satu dari sub masalah di atas kembali benar, maka kembalikan benar.

subsetSum (A,n,sum/2) = isSubsetSum (A,n – 1,sum/2) \\ subsetSum (A,n – 1,sum/2 – A[n – 1])

```
// A utility function that returns true if there is a subset of A[] with sum equal to given sum
bool subsetSum (int A[], int n, int sum){
  if (sum == 0)
    return true;
  if (n == 0 && sum != 0)
    return false;
  // If last element is greater than sum, then ignore it
  if (A[n-1] > sum)
    return subsetSum (A, n-1, sum);
  return subsetSum (A, n-1, sum) || subsetSum (A, n-1, sum-A[n-1]);
}

// Returns true if A[] can be partitioned in two subsets of equal sum, otherwise false
bool findPartition (int A[], int n){
   // calculate sum of all elements
   int sum = 0;
   for (int i = 0; i < n; i++)
     sum += A[i];
   // If sum is odd, there cannot be two subsets with equal sum
   if (sum%2 != 0)
     return false;
   // Find if there is subset with sum equal to half of total sum
   return subsetSum (A, n, sum/2);
}
```

Kompleksitas Waktu: O(2^n) Dalam kasus terburuk, solusi ini mencoba dua kemungkinan (apakah menyertakan atau mengecualikan) untuk setiap elemen.

Solusi Pemrograman Dinamis: Masalah dapat diselesaikan dengan menggunakan pemrograman dinamis ketika jumlah elemennya tidak terlalu besar. Kita dapat membuat bagian array 2D[][] dengan ukuran (jumlah/2)*(n + 1). Dan kita dapat membangun solusinya dengan cara bottom-up sehingga setiap entri yang diisi memiliki properti berikut:

bagian [i][j] = benar jika himpunan bagian dari {A[0],A[1],..A[j – 1]} memiliki jumlah sama dengan jumlah/2, jika tidak salah

```
// Returns true if A[] can be partitioned in two subsets of equal sum, otherwise false
bool findPartition (int A[], int n){
   int sum = 0;
   int i, j;
   // calculate sum of all elements
   for (i = 0; i < n; i++)
     sum += A[i];
   if (sum%2 != 0)
     return false;
   bool part[sum/2+1][n+1];
   // initialize top row as true
   for (i = 0; i <= n; i++)
     part[0][i] = true;
   // initialize leftmost column, except part[0][0], as 0
   for (i = 1; i <= sum/2; i++)
     part[i][0] = false;
   // Fill the partition table in bottom up manner
   for (i = 1; i <= sum/2; i++) {
     for (j = 1; j <= n; j++) {
       part[i][j] = part[i][j-1];
       if (i >= A[j-1])
         part[i][j] = part[i][j] || part[i - A[j-1]][j-1];
     }
   }
   return part[sum/2][n];
}
```

Kompleksitas Waktu: O(jumlah × n).

Kompleksitas Ruang: O(jumlah × n). Harap dicatat bahwa solusi ini tidak akan layak untuk array dengan jumlah besar.

Soal-29 Menghitung Kurung Boolean: Mari kita asumsikan bahwa kita diberikan ekspresi boolean yang terdiri dari simbol 'benar', 'salah', 'dan', 'atau', dan 'xor'. Temukan banyak cara untuk mengurung ekspresi sedemikian rupa sehingga akan bernilai benar. Misalnya, hanya ada 1 cara untuk memberi tanda kurung 'benar dan salah xor benar' sehingga bernilai benar.

Solusi: Biarkan jumlah simbol menjadi n dan di antara simbol ada operator boolean seperti dan, atau, xor, dll. Misalnya, jika n = 4, T atau F dan T xor F. Tujuan kita adalah menghitung jumlah cara untuk mengurung ekspresi dengan operator boolean sehingga bernilai true. Dalam kasus di atas, jika kita menggunakan T atau ((F dan T) xor F) maka bernilai true.

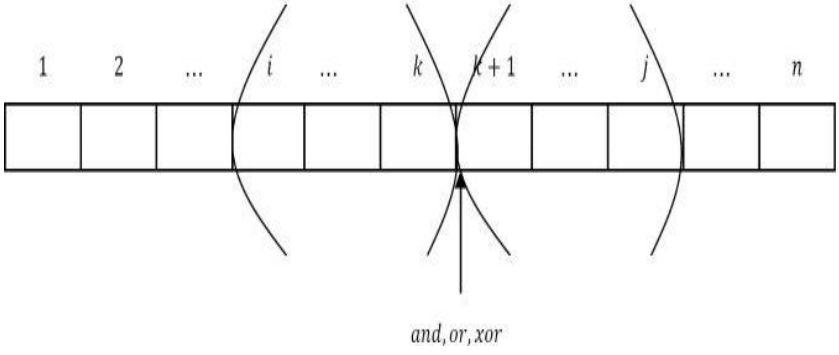
T atau{ (F dan T)xor F) = Benar

Sekarang mari kita lihat bagaimana DP menyelesaikan masalah ini. Misalkan T(i,j) menyatakan banyaknya cara untuk mengurung sub ekspresi dengan simbol i ...j [simbol berarti hanya T dan F dan bukan operator] dengan operator boolean sehingga bernilai true. Juga, i dan j mengambil nilai dari 1 hingga n. Misalnya, dalam kasus di atas, T(2,4) = 0 karena tidak ada cara untuk mengurung ekspresi F dan T x atau F untuk menjadikannya benar.

Sekadar untuk penyederhanaan dan kesamaan, misalkan F(i,j) menyatakan banyaknya cara untuk mengurung sub ekspresi dengan simbol i ...j dengan operator boolean sehingga bernilai false. Kasus dasarnya adalah T(i,i) dan F(i,i). Sekarang kita akan menghitung T(i, i + 1) dan F(i, i + 1) untuk semua nilai i. Demikian pula, T(i, i + 2) dan F(i, i + 2) untuk semua nilai i dan seterusnya. Sekarang mari kita generalisasikan solusinya.

$$T(i,j) = \sum_{k=i}^{j-1} \begin{cases} T(i,k)T(k+1,j), & for\ "and" \\ Total(i,k)Total(k+1,j) - F(i,k)F(k+1,j), & for\ "or" \\ T(i,k)F(k+1,j) + F(i,k)T(k+1,j), & for\ "xor" \end{cases}$$

Where, $Total(i,k) = T(i,k) + F(i,k)$.



Apa yang dikatakan rumus rekursif di atas adalah, T(i,j) menunjukkan jumlah cara untuk mengurung ekspresi. Mari kita asumsikan bahwa kita memiliki beberapa sub masalah yang berakhir di k. Maka banyaknya cara pengurungan dari i ke j adalah jumlah jumlah cara pengurungan dari i ke k dan dari k + 1 ke j. Untuk mengurung antara k dan k + 1 ada tiga cara: "dan", "atau" dan "xor".

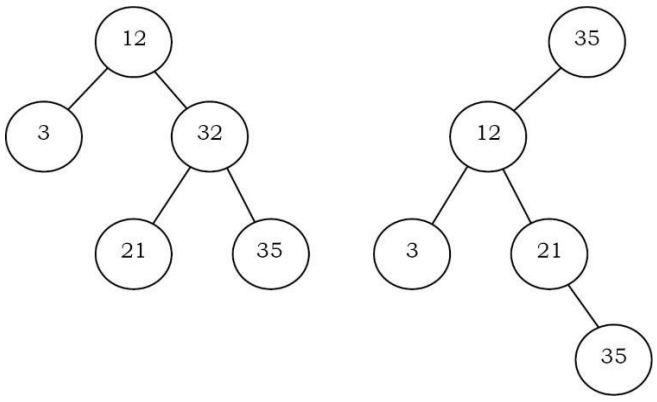
- Jika kita menggunakan “dan” antara k dan k + 1, maka ekspresi akhir menjadi benar hanya jika keduanya benar. Jika keduanya benar maka kita dapat memasukkannya untuk mendapatkan hitungan akhir.
- Jika kita menggunakan “atau”, maka jika setidaknya salah satunya benar, hasilnya menjadi benar. Alih-alih memasukkan ketiga kemungkinan untuk "atau", kita memberikan satu alternatif di mana kita mengurangi kasus "salah" dari total kemungkinan.
- Sama halnya dengan “xor”. Percakapannya seperti dalam dua kasus di atas.

Setelah menemukan semua nilai, kita harus memilih nilai k, yang menghasilkan jumlah maksimum, dan untuk k ada i hingga j – 1 kemungkinan.

Ada berapa submasalah? Dalam rumus di atas, i dapat berkisar dari 1 hingga n, dan j dapat berkisar dari 1 hingga n. Jadi ada total n^2 submasalah, dan juga kita melakukan penjumlahan untuk semua nilai tersebut. Jadi kompleksitas waktunya adalah $O(n^3)$.

Soal-30 Pohon Pencarian Biner Optimal: Diberikan satu set n kunci (diurutkan) A[1..n], buat pohon pencarian biner terbaik untuk elemen A. Juga asumsikan bahwa setiap elemen dikaitkan dengan frekuensi yang menunjukkan angka berapa kali item tertentu dicari di pohon pencarian biner. Itu berarti kita perlu membangun pohon pencarian biner sehingga total waktu pencarian akan berkurang.

Solusi: Sebelum menyelesaikan masalah, mari kita pahami masalahnya dengan sebuah contoh. Mari kita asumsikan bahwa array yang diberikan adalah A = [3,12,21,32,35]. Ada banyak cara untuk mewakili elemen-elemen ini, dua di antaranya tercantum di bawah ini.



Dari keduanya, representasi mana yang lebih baik? Waktu pencarian untuk suatu elemen tergantung pada kedalaman simpul. Jumlah rata-rata perbandingan untuk pohon pertama adalah: $\frac{1+2+2+3+3}{5} = \frac{11}{5}$ dan untuk pohon kedua, jumlah rata-rata perbandingan adalah: $\frac{1+2+3+3+4}{5} = \frac{13}{5}$ pohon memberikan hasil yang lebih baik.

Jika frekuensi tidak diberikan dan jika kita ingin mencari semua elemen, maka sederhana di atas perhitungan sudah cukup untuk menentukan pohon terbaik. Jika frekuensi diberikan, maka pemilihannya tergantung pada frekuensi elemen dan juga kedalaman elemen. Untuk mempermudah, mari kita asumsikan bahwa larik yang diberikan adalah A dan frekuensi yang sesuai berada dalam larik F. F[i] menunjukkan frekuensi elemen ke-i A[i]. Dengan ini, total waktu pencarian S(root) dari pohon dengan root dapat didefinisikan sebagai:

$$S(root) = \sum_{i=1}^{n} (depth(root, i) + 1) \times F[i])$$

Dalam ekspresi di atas, depth(root, i) + 1 menunjukkan jumlah perbandingan untuk mencari elemen ke-i. Karena kita mencoba membuat pohon pencarian biner, elemen subpohon kiri lebih kecil dari elemen akar dan elemen subpohon kanan lebih besar dari elemen akar. Jika kita memisahkan waktu subpohon kiri dan waktu subpohon kanan, maka ekspresi di atas dapat ditulis sebagai:

$$S(root) = \sum_{i=1}^{r-1} (depth(root, i) + 1) \times F[i]) + \sum_{i=1}^{n} F[i] + \sum_{i=r+1}^{n} (depth(root, i) + 1) \times F[i])$$

Where r indicates the position of the root element in the array.

Jika kita mengganti waktu subpohon kiri dan subpohon kanan dengan panggilan rekursif yang sesuai, maka ekspresinya menjadi:

$$S(root) = S(root \rightarrow left) + S(root \rightarrow right) + + \sum_{i=1}^{n} F[i]$$

Deklarasi simpul Pohon Pencarian Biner

Lihat bab Pohon.

Penerapan:

```
struct BinarySearchTreeNode *OptimalBST(int A[], int F[], int low, int high) {
    int r, minTime = 0;
    struct BinarySearchTreeNode *newNode=(struct BinarySearchTreeNode *)
                                                malloc(sizeof(struct BinarySearchTreeNode));
    if(!newNode) {
            printf("Memory Error");
            return;
    }
    for (r =0, r <= n-1; r++)  {
            root→left =  OptimalBST(A, F, low, r-1);
            root→left =  OptimalBST(A, F, r+1, high)
            root→data = A[r];
            if(minTime > S(root)) minTime = S(root);
    }
    return minTime;
}
```

Soal-31 Jarak Edit: Diberikan dua string A dengan panjang m dan B dengan panjang n, ubah A menjadi B dengan jumlah minimum operasi dari jenis berikut: hapus karakter dari A, masukkan karakter ke A, atau ubah beberapa karakter di A menjadi karakter baru. Jumlah minimal operasi yang diperlukan untuk mengubah A menjadi B disebut jarak edit antara A dan B.

Solusi:

Input: Dua string teks A dengan panjang m dan B dengan panjang n.
Sasaran: Mengonversi string A menjadi B dengan konversi minimal.
Sebelum beralih ke solusi, mari kita pertimbangkan kemungkinan operasi untuk mengubah string A menjadi B.

- Jika m > n, kita perlu menghapus beberapa karakter A
- Jika m == n, kita mungkin perlu mengonversi beberapa karakter A
- Jika m < n, kita perlu menghapus beberapa karakter dari A

Jadi operasi yang kita butuhkan adalah penyisipan karakter, penggantian karakter dan penghapusan karakter, dan kode biaya yang sesuai didefinisikan di bawah ini.

Biaya operasi:

Penyisipan karakter	c_i
Penggantian karakter	c_r
Penghapusan karakter	c_d

Sekarang mari kita berkonsentrasi pada perumusan masalah rekursif. Misalkan, T(i,j) mewakili biaya minimum yang diperlukan untuk mengubah i karakter pertama dari A menjadi yang pertama; karakter B. Artinya, A[1... i] sampai B[1...j].

$$T(i,j) = min \begin{cases} c_d + T(i-1,j) \\ T(i,j-1) + c_i \\ \begin{cases} T(i-1,j-1), & if\ A[i] == B[j] \\ T(i-1,j-1) + c_r & if\ A[i] \neq B[j] \end{cases} \end{cases}$$

Berdasarkan pembahasan di atas kita memiliki kasus berikut.

- Jika kita menghapus karakter ke-i dari A, maka kita harus mengubah sisa i – 1 karakter dari A menjadi j karakter dari B
- Jika kita memasukkan karakter ke-i di A, maka ubah karakter ke-i dari A ini menjadi j – 1 karakter dari B
- Jika A[i] == B[j], maka kita harus mengubah sisa i – 1 karakter dari A menjadi j – 1 karakter dari B
- Jika A[i] ≠ B[j], maka kita harus mengganti karakter ke-i dari A menjadi karakter ke-j dari B dan mengubah sisa i – 1 karakter dari A menjadi j – 1 karakter dari B

Setelah menghitung semua kemungkinan kita harus memilih salah satu yang memberikan biaya terendah.

Ada berapa submasalah? Dalam rumus di atas, i dapat berkisar dari l hingga m dan j dapat berkisar dari 1 hingga n. Ini memberikan mn submasalah dan masing-masing mengambil O(1) dan kompleksitas waktu adalah O(mn). Kompleksitas Ruang: O(mn) di mana m adalah jumlah baris dan n adalah jumlah kolom dalam matriks yang diberikan.

Soal-32 Semua Pasangan Jalur Terpendek Soal: Algoritma Floyd: Diberikan grafik berarah berbobot G = (V,E), di mana V = {1,2,...,n}. Temukan jalur terpendek antara setiap pasangan node dalam Grafik. Asumsikan bobot direpresentasikan dalam matriks C[V][V], di mana C[i] [j] menunjukkan bobot (atau biaya) antara node i dan j. Juga, C[i][j] = atau -1 jika tidak ada jalur dari simpul i ke simpul j.

Solusi: Mari kita coba mencari solusi DP (algoritma Floyd) untuk masalah ini. Algoritma Floyd untuk semua pasangan masalah jalur terpendek menggunakan matriks A[1. .n][1..n] untuk menghitung panjang jalur terpendek. Mulanya,

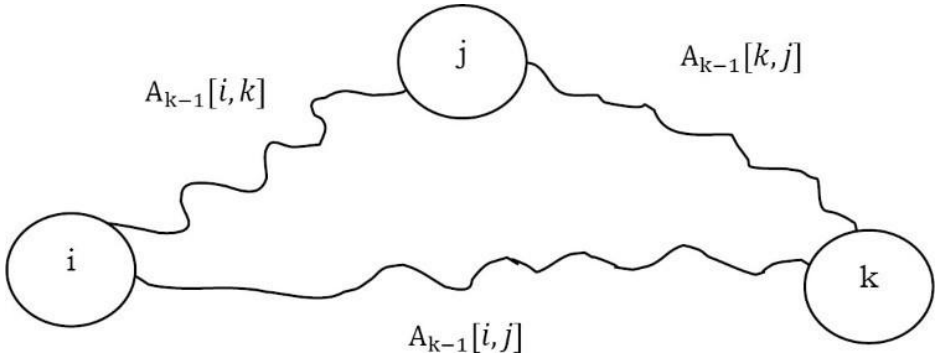
$$A[i,j] = C[i,j] \;\; if \;\; i \neq j$$
$$= 0 \qquad if \;\; i = j$$

Dari definisi tersebut, C[i,j] = jika tidak ada jalur dari i ke j. Algoritma membuat n melewati A. Misalkan A0,A1, ...,An adalah nilai A pada n lintasan, dengan A0 sebagai nilai awal. Tepat setelah k– iterasi ke-1, A_{k-1}[i,j] = panjang terkecil dari setiap lintasan dari simpul i ke simpul j yang tidak melalui simpul {k + 1, k + 2,.... n} . Artinya, melewati simpul mungkin melalui {1,2,3,..., k – 1}.

Pada setiap iterasi, nilai A[i][j] diperbarui dengan minimum A_{k-1} [i,j] dan A_{k-1} [i, k] + A_{k-1}[k,j].

$$A[i,j] = \min \begin{cases} \mathrm{A}_{\mathrm{k}-1}[i,j] \\ \mathrm{A}_{\mathrm{k}-1}[i,k] + \mathrm{A}_{\mathrm{k}-1}[k,j] \end{cases}$$

Lintasan ke-k mengeksplorasi apakah simpul k terletak pada jalur optimal dari i ke j, untuk semua i,j. Hal yang sama ditunjukkan pada diagram di bawah ini.



```
void Floyd(int C[][], int A[][], int n) {
    int i, j, k;
    for(i = 0, i <= n - 1;i + +)
        for(j = 0; j <= n - 1, j + +)
            A[i][j] = C[i][j];
    for(i = 0;i <= n - 1;i + +)
        A[i][i] = 0;
    for(k = 0;k <= n - 1;k + +) {
            for(i = 0;i <= n - 1;i + +) {
                    for(j = 0;j <= n - 1, j + +)
                            if(A[i][k] + A[k][j] < A[i][j])
                                    A[i][j] = A[i][k] + A[k][j];
            }
    }
}
```

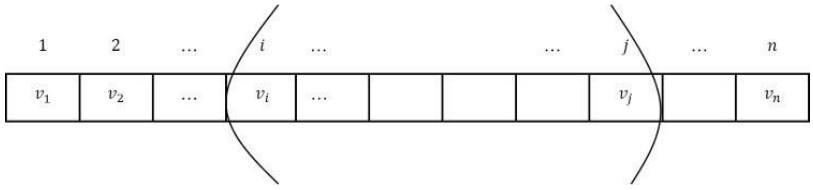
Kompleksitas Waktu: $O(n^3)$.

Soal-33 Strategi Optimal untuk Sebuah Game: Pertimbangkan deretan n koin dengan nilai v_1 ... v_n, di mana n genap [karena ini adalah permainan dua pemain]. Kita memainkan game ini dengan lawan. Di setiap giliran, pemain memilih koin pertama atau terakhir dari baris, menghapusnya dari baris secara permanen, dan menerima nilai koin. Tentukan jumlah uang maksimum yang mungkin bisa kita menangkan jika kita bergerak terlebih dahulu.

Cara alternatif untuk membingkai pertanyaan: Diberikan n pot, masing-masing dengan sejumlah koin emas, disusun dalam satu garis. Anda bermain game melawan pemain lain. Anda bergiliran memilih pot emas. Anda dapat memilih pot dari kedua ujung garis, mengeluarkan pot, dan menyimpan potongan emas. Pemain dengan emas paling banyak pada akhirnya menang. Kembangkan strategi untuk memainkan game ini.

Solusi: Mari kita selesaikan masalah menggunakan teknik DP kita. Untuk setiap giliran kita atau lawan kita memilih koin hanya dari ujung baris. Mari kita mendefinisikan submasalah sebagai:

V(i,j): menunjukkan kemungkinan nilai maksimum yang pasti bisa kita menangkan jika giliran kita dan satu-satunya koin yang tersisa adalah v_i ... v_j.



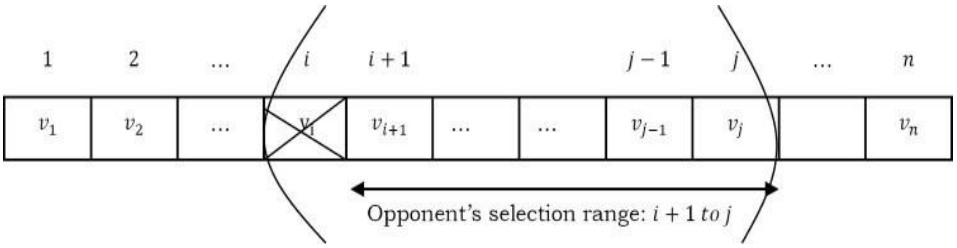
Kasus Dasar: V(i,i),V(i, i + 1) untuk semua nilai i.

Dari nilai-nilai ini, kita dapat menghitung V(i, i + 2),V(i,i + 3) dan seterusnya. Sekarang mari kita definisikan V(i,j) untuk setiap sub masalah sebagai:

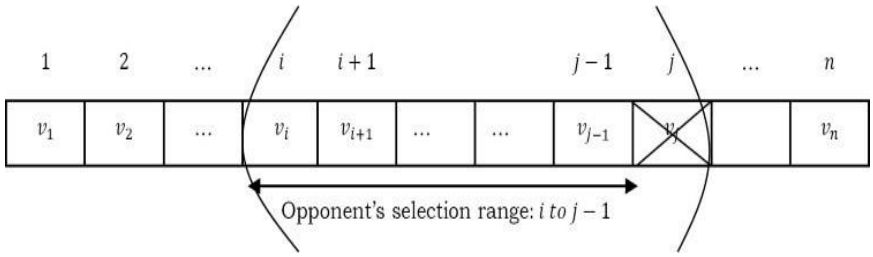
$$V(i,j) = Max\left\{Min\begin{Bmatrix} V(i+1,j-1) \\ V(i+2,j) \end{Bmatrix} + v_i, Min\begin{Bmatrix} V(i,j-2) \\ V(i+1,j-1) \end{Bmatrix} + v_j\right\}$$

Dalam panggilan rekursif kita harus fokus pada koin ke-i ke koin ke-j (v_i... v_j). Karena giliran kita untuk mengambil koin, kita memiliki dua kemungkinan: apakah kita dapat memilih vi atau vj. Suku pertama menunjukkan kasus jika kita memilih koin ke-i (v_i) dan suku kedua menunjukkan kasus jika kita memilih koin ke-j (v_j). Max luar menunjukkan bahwa kita harus memilih koin yang memberikan nilai maksimum. Sekarang mari kita fokus pada ketentuan:

- Memilih koin ke-i: Jika kita memilih koin ke-i maka range yang tersisa adalah dari i + 1 sampai j. Karena kita memilih koin ke-i, kita mendapatkan nilai vi untuk itu. Dari kisaran yang tersisa i + 1 hingga j, lawan dapat memilih koin ke-i + 1 atau koin ke-j. Tetapi pemilihan lawan harus seminimal mungkin [suku Min]. Hal yang sama dijelaskan pada gambar di bawah ini.



- Memilih koin ke-j: Di sini juga argumennya sama seperti di atas. Jika kita memilih koin ke-j, maka rentang yang tersisa adalah fromitoj-1. Karena kita memilih koin ke-j, kita mendapatkan nilai v_j untuk itu. Dari rentang i hingga j - 1 yang tersisa, lawan dapat memilih koin ke-i atau koin ke-1 j. Tetapi pemilihan lawan harus diminimalkan semaksimal mungkin [istilah Min].



Ada berapa submasalah? Dalam rumus di atas, i dapat berkisar dari 1 hingga n dan j dapat berkisar dari 1 hingga n. Ada total n^2 submasalah dan masing-masing membutuhkan O(1) dan total kompleksitas waktu adalah $O(n^2)$.

Soal-34 **Penataan:** Asumsikan bahwa kita menggunakan kartu domino berukuran 2 × 1 untuk memasang papan dengan tinggi tak terhingga 2. Berapa banyak cara seseorang dapat memasang pita 2 × n sel persegi dengan kartu domino 1x2?

Solusi: Perhatikan bahwa kita dapat menempatkan ubin baik secara vertikal maupun horizontal. Untuk menempatkan ubin vertikal, kita membutuhkan celah minimal 2 × 2. Untuk menempatkan ubin horizontal, kita membutuhkan celah 2 × 1. Dengan cara ini, masalahnya dikurangi menjadi menemukan banyak cara untuk mempartisi n menggunakan angka 1 dan 2 dengan urutan yang dianggap relevan [1]. Misalnya: 11 = 1 + 2 + 2+1+2 + 2 + 1.

Jika kita harus menemukan pengaturan seperti itu untuk 12, kita dapat menempatkan 1 di akhir atau kita dapat menambahkan 2 dalam pengaturan yang mungkin dengan 10. Demikian pula, katakanlah kita memiliki F_n pengaturan yang mungkin untuk n. Kemudian untuk (n + 1), kita dapat menempatkan hanya 1 di akhir atau kita dapat menemukan kemungkinan pengaturan untuk (n – 1) dan beri angka 2 di akhir. Mengikuti teori di atas:

$$F_{n+1} = F_n + F_{n-1}$$

Mari kita verifikasi teori di atas untuk masalah awal kita:

- Dalam berapa cara kita dapat mengisi strip 2 × 1: 1 → Hanya satu ubin vertikal.
- Dalam berapa cara kita dapat mengisi strip 2 × 2: 2 → Baik 2 ubin horizontal atau 2 ubin vertikal.
- Dalam berapa cara kita dapat mengisi strip 2 × 3: 3 → Entah menempatkan ubin vertikal di 2 solusi yang mungkin untuk strip 2 × 2, atau menempatkan 2 ubin horizontal di satu-satunya solusi yang mungkin untuk strip 2 × 1. (2 + 1 = 3).
- Demikian pula, dalam berapa banyak cara kita dapat mengisi strip 2 × n: Letakkan ubin vertikal dalam solusi yang memungkinkan untuk 2 X (n – 1) strip atau letakkan 2 ubin horizontal dalam solusi yang mungkin untuk 2 × (n – 2) strip. (F_{n-1} + Fn_{-2}).
- Begitulah cara kita memverifikasi bahwa solusi akhir kita adalah: Fn = F_{n-1} + F_{n-2} dengan F1 = 1 dan F_2 = 2.

Soal-35 Barisan Palindrom Terpanjang: Barisan adalah palindrom jika dibaca sama baik kita membacanya dari kiri ke kanan atau kanan ke kiri. Misalnya A, C, G, G, G, G, C, A. Diberikan barisan dengan panjang n, buatlah algoritma untuk menampilkan panjang barisan palindrom terpanjang. Misalnya, string A,G,C,T,C,B,M,A,A,C,T,G,G,A,M memiliki banyak palindrom sebagai turunannya, misalnya: A,G,T,C ,M,C,T,G,A memiliki panjang 9.

Solusi: Mari kita gunakan DP untuk menyelesaikan masalah ini. Jika kita melihat sub-string A[i,..,j] dari string A, maka kita dapat menemukan barisan palindrom dengan panjang minimal 2 jika A[i] == A[j]. Jika tidak sama, maka kita harus mencari palindrom dengan panjang maksimum pada barisan A[i + 1,..., j] dan A[i,..., j – 1].

Juga, setiap karakter A[i] adalah palindrom dengan panjang 1. Oleh karena itu, kasus dasarnya diberikan oleh A[i, i] = 1. Mari kita definisikan palindrom dengan panjang maksimum untuk substring A[i,...,j] sebagai L(i,j).

$$L(i,j) = \begin{cases} L(i+1, j-1) + 2, & if\ A[i] == A[j] \\ Max\{L(i+1, j), L(i, j-1)\}, & otherwise \end{cases}$$

$$L(i,i) = 1\ for\ all\ i = 1\ to\ n$$

```
int LongestPalindromeSubsequence(int A[], int n) {
    int max  = 1;
    int i,k, L[n][ n];
    for (i  = 1; i<= n -1; i++) {
        L[i][i] =1;
        if(A[i]==A[i+1]) {
            L[i][i + 1]  = 1;
            max = 2;
        }
        else
            L[i][i + 1]  = 0;
    }
    for (k=3;k<= n;k++) {
            for (i  = 1;i <= n-k +1; i++) {
                    j =  i + k - 1;
                    if(A[i] == A[j])     {
                        L[i, j] =  2 + L[i + 1][j - 1];
                        max = k;
                    }
                    else
                        L[i, j]  = max(L[i + 1][j - 1], L[i][j - 1]);
            }
    }
    return max;
}
```

Kompleksitas Waktu: Perulangan 'for' pertama membutuhkan waktu O(n) sedangkan perulangan 'for' kedua membutuhkan waktu O(n – k) yang juga O(n). Oleh karena itu, total waktu berjalan dari algoritma diberikan oleh $O(n^2)$.

Soal-36 Substring Palindrome Terpanjang: Diberikan sebuah string A, kita perlu mencari sub-string terpanjang dari A sedemikian rupa sehingga kebalikannya persis sama.

Solusi: Perbedaan mendasar antara substring palindrom terpanjang dan turunan palindrom terpanjang adalah bahwa, dalam kasus substring palindrom terpanjang, string keluaran harus berupa karakter bersebelahan, yang

memberikan palindrom maksimum; dan dalam kasus barisan palindrom terpanjang, *output*nya adalah urutan karakter di mana karakter mungkin tidak bersebelahan tetapi mereka harus dalam urutan yang meningkat sehubungan dengan posisinya dalam string yang diberikan.

Solusi brute force secara mendalam memeriksa semua n (n + 1) / 2 kemungkinan substring dari string n-panjang yang diberikan, menguji masing-masing jika itu adalah palindrom, dan melacak yang terpanjang yang terlihat sejauh ini. Ini memiliki kompleksitas kasus terburuk $O(n^3)$, tetapi kita dapat dengan mudah melakukan lebih baik dengan menyadari bahwa palindrom berpusat pada huruf (untuk palindrom dengan panjang ganjil) atau spasi di antara huruf (untuk genap- palindrom panjang). Oleh karena itu kita dapat memeriksa semua n + 1 pusat yang mungkin dan menemukan palindrom terpanjang untuk pusat tersebut, dengan melacak keseluruhan palindrom terpanjang. Ini memiliki kompleksitas kasus terburuk $O(n^2)$.

Mari kita gunakan DP untuk menyelesaikan masalah ini. Perlu dicatat bahwa tidak ada lebih dari $O(n^2)$ substring dalam string dengan panjang n (sementara persis ada 2^n suburutan). Oleh karena itu, kita dapat memindai setiap substring, memeriksa palindrom, dan memperbarui panjang substring palindrom terpanjang yang ditemukan sejauh ini. Karena uji palindrom membutuhkan waktu linier dalam panjang substring, ide ini membutuhkan algoritma $O(n^3)$. Kita dapat menggunakan DP untuk meningkatkan ini. Untuk $1 \leq I \leq j \leq n$, tentukan

$$L(i,j) = \begin{cases} 1, & if\ A[i] \ldots . A[j] \text{ is a palindrome substring,} \\ 0, & \text{otherwise} \end{cases}$$

$$L[i,i] = 1,$$

$$L[i,j] = L[i,i+1], if\ A[i] == A[i+1], for\ 1 \leq i \leq j \leq n-1.$$

Juga, untuk string dengan panjang setidaknya 3,

$$L[i,j] = (L[i+1, j-1]\ and\ A[i] = A[j]).$$

Perhatikan bahwa untuk mendapatkan perulangan yang terdefinisi dengan baik, kita perlu secara eksplisit menginisialisasi dua diagonal berbeda dari larik boolean L[i,j], karena perulangan untuk entri [i,j] menggunakan nilai [i – 1,j – 1], yang berjarak dua diagonal dari [i,j] (artinya, untuk substring dengan panjang k, kita perlu mengetahui status substring dengan panjang k – 2).

```
int LongestPalindromeSubstring(int A[], int n) {
    int max = 1;
    int i,k, L[n][n];
    for (i = 1; i<=n-1; i++) {
        L[i][i] =1;
        if(A[i]==A[i+1]) {
            L[i][i + 1] = 1;
            max = 2;
        }
```

```
            else
                L[i][i + 1]  = 0;
        }
        for (k=3;k<=n;k++) {
           for (i  = 1;i <= n-k +1; i++) {
              j =  i + k - 1;
              if(A[i] == A[j] && L[i + 1][j - 1]) {
                 L[i][j] =  1;
                 max = k;
              }
              else
                 L[i][j]  = 0;
           }
        }
        return max;
    }
```

Kompleksitas Waktu: Perulangan for pertama membutuhkan waktu O(n) sedangkan perulangan for kedua membutuhkan waktu O(n – k) yang juga O(n). Oleh karena itu total waktu berjalan dari algoritma diberikan oleh $O(n^2)$.

Soal-37 Diberikan dua string S dan T, berikan algoritma untuk mencari berapa kali S muncul di T. Tidak wajib bahwa semua karakter S harus muncul bersebelahan dengan T. Misalnya, jika S = ab dan T = abadcb maka solusinya adalah 4, karena ab muncul 4 kali di abadcb.

Solusi:

Input: Diberikan dua senar S[1..m] dan T[1 ...m].
Sasaran: Hitung berapa kali S muncul di T.
Asumsikan L(i,j) mewakili hitungan berapa kali i karakter S muncul di j karakter T.

$$L(i,j) = Max \begin{cases} 0, & if\ j = 0 \\ 1, & if\ i = 0 \\ L(i-1,j-1) + L(i,j-1), & if\ S[i] == T[j] \\ L(i-1,j), & if\ S[i] \neq T[j] \end{cases}$$

Jika kita berkonsentrasi pada komponen dari rumus rekursif di atas,

- Jika j = 0, maka karena T kosong maka hitungan menjadi 0.
- Jika i = 0, maka kita dapat memperlakukan string kosong S juga muncul di T dan kita dapat memberikan hitungan sebagai 1.
- Jika S[i] == T[i], berarti karakter ke-i dari S dan karakter ke-j dari T adalah sama. Dalam hal ini kita harus memeriksa submasalah dengan i – 1 karakter dari S dan j – 1 karakter dari T dan juga kita harus menghitung hasil dari i karakter dari S dengan – 1 karakter dari T. Hal ini karena semua i karakter dari S mungkin muncul di j – 1 karakter dari T.
- Jika S[i] ≠ T[i], maka kita harus mendapatkan hasil submasalah dengan i – 1 karakter dari S dan j karakter dari T.

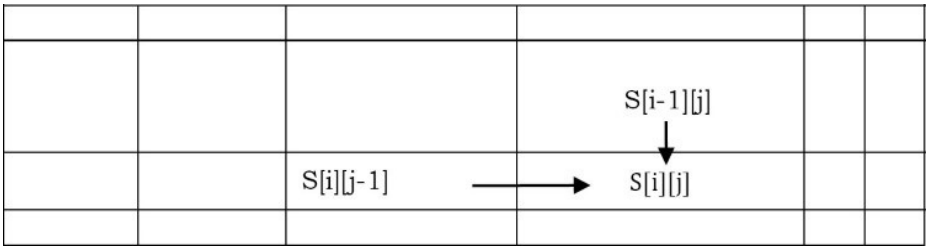
Setelah menghitung semua nilai, kita harus memilih salah satu yang memberikan jumlah maksimum.

Ada berapa submasalah? Dalam rumus di atas, i dapat berkisar dari 1 hingga m dan j dapat berkisar dari 1 hingga n. Ada total subproblem yang dijalankan dan masing-masing membutuhkan O(1). Kompleksitas Waktu adalah O(mn).

Kompleksitas Ruang: O(mn) di mana m adalah jumlah baris dan n adalah jumlah kolom dalam matriks yang diberikan.

Soal-38 Diberikan matriks dengan n baris dan m kolom (n × m). Di setiap sel ada sejumlah apel. Kita mulai dari sudut kiri atas matriks. Kita bisa turun atau kanan satu sel. Akhirnya, kita harus sampai di pojok kanan bawah. Temukan jumlah apel maksimum yang dapat kita kumpulkan. Ketika kita melewati sel, kita mengumpulkan semua apel yang tersisa di sana.

Solusi: Mari kita asumsikan bahwa matriks yang diberikan adalah A[n][m]. Hal pertama yang harus diperhatikan adalah paling banyak ada 2 cara kita bisa sampai ke sebuah sel - dari kiri (jika tidak terletak di kolom pertama) dan dari atas (jika tidak terletak di baris paling atas).



Untuk menemukan solusi terbaik untuk sel itu, kita harus sudah menemukan solusi terbaik untuk semua sel dari mana kita bisa sampai ke sel saat ini. Dari atas, hubungan berulang dapat dengan mudah diperoleh sebagai:

$$S(i,j) = \left\{ A[i][j] + Max \begin{Bmatrix} S(i,j-1), & if\ j > 0 \\ S(i-1,j), & if\ i > 0 \end{Bmatrix} \right\}$$

S(i,j) harus dihitung dengan terlebih dahulu dari kiri ke kanan di setiap baris dan memproses baris dari atas ke bawah, atau dengan pertama dari atas ke bawah di setiap kolom dan memproses kolom dari kiri ke kanan.

Ada berapa submasalah seperti itu? Dalam rumus di atas, i dapat berkisar dari 1 hingga n dan j dapat berkisar dari 1 hingga m. Ada total subproblem run dan masing-masing membutuhkan O(1). Kompleksitas Waktu adalah O(nm). Kompleksitas Ruang: O(nm), di mana m adalah jumlah baris dan n adalah jumlah kolom dalam matriks yang diberikan.

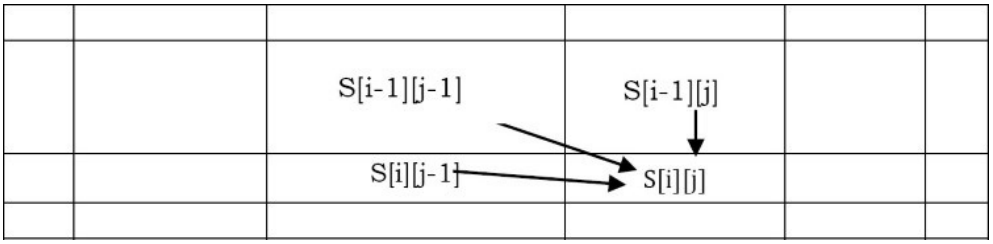
```
int FindApplesCount(int A[][], int n, int m) {
    int S[n][m];
    for( int i = 1;i<=n;i++) {
        for(int j = 1;i<=m;j++) {
            S[i][j] = A[i][j];
            if(j>0 && S[i][j] < S[i][j] + S[i][j-1])
                S[i][j] += S[i][j-1];
            if(i>0 && S[i][j] < S[i][j] + S[i-1][j])
                S[i][j] +=S[i-1][j];
        }
    }
    return S[n][m];
}
```

Soal-39 Mirip dengan Soal-38, asumsikan bahwa kita bisa turun, ke kanan satu sel, atau bahkan dalam arah diagonal. Kita harus tiba di pojok kanan bawah. Berikan solusi DP untuk mencari jumlah apel maksimum yang bisa kita kumpulkan.

Solusi: Ya. Pembahasannya sangat mirip dengan Soal-38. Mari kita asumsikan bahwa matriks yang diberikan adalah A[n][m]. Hal pertama yang harus diperhatikan adalah paling banyak ada 3 cara kita bisa datang ke sel - dari kiri, dari atas (jika tidak terletak di baris paling atas) atau dari diagonal atas. Untuk menemukan solusi terbaik untuk sel itu, kita harus sudah menemukan solusi terbaik untuk semua sel dari mana kita bisa sampai ke sel saat ini. Dari atas, hubungan berulang dapat dengan mudah diperoleh:

$$S(i,j) = \left\{ A[i][j] + Max \begin{Bmatrix} S(i,j-1), & if\ j>0 \\ S(i-1,j), & if\ i>0 \\ S(i-1,j-1), & if\ i>0\ and\ j>0 \end{Bmatrix} \right\}$$

S(i,j) harus dihitung dengan terlebih dahulu dari kiri ke kanan di setiap baris dan memproses baris dari atas ke bawah, atau dengan pertama dari atas ke bawah di setiap kolom dan memproses kolom dari kiri ke kanan.



Ada berapa submasalah seperti itu? Dalam rumus di atas, i dapat berkisar dari 1 hingga n dan j dapat berkisar dari 1 hingga m. Ada total mn submasalah dan masing-masing mengambil O(1).

Kompleksitas Waktu adalah O(nm).

Kompleksitas Ruang: O(nm) di mana m adalah jumlah baris dan n adalah jumlah kolom dalam matriks yang diberikan.

Soal-40 Sub-matriks kuadrat ukuran maksimum dengan semua 1: Diberikan matriks dengan 0 dan 1, berikan algoritma untuk menemukan sub-matriks kuadrat ukuran maksimum dengan semua Is. Sebagai contoh, perhatikan matriks biner di bawah ini.

$$\begin{matrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Sub-matriks kuadrat maksimum dengan semua bit yang ditetapkan adalah

$$\begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$$

Solusi: Mari kita coba selesaikan masalah ini menggunakan DP. Biarkan matriks biner yang diberikan menjadi B[m][m]. Ide dari algoritma ini adalah untuk membangun matriks sementara L[][] di mana setiap entri L[i][j] mewakili ukuran sub-matriks kuadrat dengan semua 1 termasuk B[i][j] dan B[i][j] adalah yang paling kanan dan entri paling bawah dalam sub-matriks.

Algoritma:

1) Bangun matriks penjumlahan L[m][n] untuk matriks B[m][n] yang diberikan.
   A. Salin baris pertama dan kolom pertama apa adanya dari B[ ][ ] ke L[ ][ ].
   B. Untuk entri lain, gunakan ekspresi berikut untuk membangun L[ ][ ]

$$\text{if}(B[i][j]\,)$$
$$L[i][j] \;=\; min(L[i][j-1], L[i-1][j], L[i-1][j-1]) \;+\; 1;$$
$$\text{else } L[i][j] \;=\; 0;$$

2) Temukan entri maksimum dalam L[m][n].
3) Menggunakan nilai dan koordinat entri maksimum di L[i], cetak submatriks B[][].

```
void MatrixSubSquareWithAllOnes(int B[][], int m, int n) {
    int i, j, L[m][n], max_of_s, max_i, max_j;
    // Setting first column of L[][]
    for(i = 0; i < m; i++)
        L[i][0] = B[i][0];
    // Setting first row of L[][]
    for(j = 0; j < n; j++)
        L[0][j] = B[0][j];
    // Construct other entries of L[][]
    for(i = 1; i < m; i++) {
        for(j = 1; j < n; j++) {
            if(B[i][j] == 1)
                L[i][j] = min(L[i][j-1], L[i-1][j], L[i-1][j-1]) + 1;
            else    L[i][j] = 0;
        }
    }
```

```
        max_of_s = L[0][0]; max_i = 0; max_j = 0;
        for(i = 0; i < m i++) {
                for(j = 0; j < n; j++) {
                        if(L[i][j] > max_of_s){
                            max_of_s = L[i][j];
                            max_i = i;
                            max_j = j;
                        }
                }
        }
        printf("Maximum sub-matrix");
        for(i = max_i; i > max_i - max_of_s; i--) {
                for(j = max_j; j > max_j - max_of_s; j--)
                        printf("%d",B[i][j]);
        }
    }
```

Ada berapa submasalah? Dalam rumus di atas, i dapat berkisar dari 1 hingga n dan j dapat berkisar dari 1 hingga m. Ada total nm submasalah dan masing-masing mengambil O(1). Kompleksitas Waktu adalah O(nm). Kompleksitas Ruang adalah O(nm), di mana n adalah jumlah baris dan m adalah jumlah kolom dalam matriks yang diberikan.

Soal-41 Sub-matriks ukuran maksimum dengan semua 1: Diberikan matriks dengan 0 dan 1, berikan algoritma untuk menemukan sub-matriks ukuran maksimum dengan semua Is. Sebagai contoh, perhatikan matriks biner di bawah ini.

1	1	0	0	1	0
0	1	1	1	1	1
1	1	1	1	1	0
0	0	1	1	0	0

Sub-matriks maksimum dengan semua bit yang ditetapkan adalah

1	1	1	1
1	1	1	1

Solusi: Jika kita menggambar histogram dari semua sel 1 pada baris di atas untuk baris tertentu, maka maksimum semua sub-matriks 1 yang berakhir pada baris tersebut akan sama dengan luas persegi panjang maksimum dalam histogram tersebut. Di bawah ini adalah contoh untuk baris ke-3 dalam matriks yang dibahas di atas [1]:

1	1	0	0	1	0
0	1	1	1	1	1
1	1	1	1	1	0
0	0	1	1	0	0

Jika kita menghitung luas ini untuk semua baris, luas maksimum akan menjadi jawaban kita. Kita dapat memperluas solusi kita dengan sangat mudah untuk menemukan koordinat awal dan akhir. Untuk ini, kita perlu membangkitkan matriks bantu S[][] dimana setiap elemen mewakili jumlah Is di atas dan memasukkannya, hingga 0 pertama. S[][] untuk matriks di atas akan seperti gambar di bawah ini:

1 1 0 0 1 0
0 2 1 1 2 1
1 3 2 2 3 0
0 0 3 3 0 0

Sekarang kita cukup memanggil persegi panjang maksimum kita dalam histogram pada setiap baris di S[][] dan memperbarui area maksimum setiap saat. Kita juga tidak memerlukan ruang ekstra untuk menyimpan S. Kita dapat memperbarui matriks asli (A) ke S dan setelah perhitungan, kita dapat mengonversi S kembali ke A.

```
#define ROW 10
#define COL 10
int find_max_matrix(int A[ROW][COL]) {
    int max, cur_max = 0;
    //Calculate Auxilary matrix
    for (int i=1; i<ROW; i++)
        for(int j=0; j<COL; j++) {
            if(A[i][j] == 1)
                A[i][j] = A[i-1][j] + 1;
        }
    //Calculate maximum area in S for each row
    for (int i=0; i<ROW; i++) {
        max = MaxRectangleArea(A[i], COL);          //Refer Stacks Chapter
        if(max > cur_max)
            cur_max = max;
    }
    //Regenerate Original matrix
    for (int i=ROW-1; i>0; i--)
        for(int j=0; j<COL; j++) {
            if(A[i][j])
                A[i][j] = A[i][j] - A[i-1][j];
        }
    return cur_max;
}
```

Soal-42 Jumlah maksimum sub-matriks: Diberikan matriks n × n M dari bilangan bulat positif dan negatif, berikan algoritma untuk menemukan sub-matriks dengan jumlah terbesar yang mungkin.

Solusi: Misalkan Aux[r, c] menyatakan jumlah subarray persegi panjang dari M dengan satu sudut di entri [1,1] dan yang lainnya di [r,c]. Karena ada n2 kemungkinan seperti itu, kita dapat menghitungnya dalam waktu O(n2). Setelah menghitung

semua jumlah yang mungkin, jumlah dari setiap subarray persegi panjang dari M dapat dihitung dalam waktu yang konstan. Ini memberikan algoritma O(n4): kita cukup menebak sudut kiri bawah dan sudut kanan atas dari subarray persegi panjang dan menggunakan tabel Aux untuk menghitung jumlahnya.

Soal-43 Bisakah kita meningkatkan kompleksitas Soal-42?

Solusi: Kita dapat menggunakan solusi Soal-4 dengan sedikit variasi, seperti yang telah kita lihat bahwa larik jumlah maksimum dari algoritma larik 1 – D memindai larik satu entri pada satu waktu dan menyimpan total entri yang berjalan. Kapan saja, jika total ini menjadi negatif, maka setel ke 0. Algoritma ini disebut algoritma Kadane. Kita menggunakan ini sebagai fungsi bantu untuk memecahkan masalah dua dimensi dengan cara berikut.

```
public void FindMaximumSubMatrix(int[][] A, int n){
   //computing the vertical prefix sum for columns
   int[][] M = new int[n][n];
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         if (j == 0)
            M[j][i] = A[j][i];
         else
            M[j][i] = A[j][i] + M[j - 1][i];
      }
   }
   int maxSoFar = 0, min, subMatrix;
   //iterate over the possible combinations applying Kadane's Alg.
   for (int i = 0; i < n; i++) {
      for (int j = i; j < n; j++) {
         min = 0;
         subMatrix = 0;
         for (int k = 0; k < n; k++) {
            if (i == 0)
               subMatrix += M[j][k];
            else   subMatrix += M[j][k] - M[i - 1 ][k];
            if(subMatrix < min)
               min = subMatrix;
            if((subMatrix - min) > maxSoFar)
               maxSoFar = subMatrix - min;
         }
      }
   }
}
```

Kompleksitas Waktu: $O(n^3)$.

Soal-44 Diberikan sebuah bilangan n, tentukan jumlah kuadrat minimum yang diperlukan untuk menjumlahkan bilangan n yang diberikan.

Contoh: min[1] = 1 = 1^2, min[2] = 2 = $1^2 + 1^2$, min[4] = 1 = 2^2, min[1^3] = 2 = $3^2 + 2^2$.

Solusi: Masalah ini dapat direduksi menjadi masalah penukaran koin. Denominasi adalah 1 sampai $\sqrt{n}$. Sekarang, kita hanya perlu melakukan perubahan untuk n dengan jumlah pecahan minimum.

Soal-45 Menemukan Jumlah Lompatan Optimal Untuk Mencapai Elemen Terakhir: Diberikan sebuah larik, mulai dari elemen pertama dan capai elemen terakhir dengan melompat. Panjang lompatan paling banyak dapat berupa nilai pada posisi saat ini dalam larik. Hasil optimal adalah ketika Anda mencapai tujuan dalam jumlah minimum lompatan. Contoh: Diberikan array A = {2,3,1,1,4}. Kemungkinan cara untuk mencapai akhir (daftar indeks) adalah:

- 0,2,3,4 (lompat 2 ke indeks 2, lalu lompat 1 ke indeks 3, lalu lompat 1 ke indeks 4)
- 0,1,4 (lompat 1 ke indeks 1, lalu lompat 3 ke indeks 4) Karena solusi kedua hanya memiliki 2 lompatan, ini adalah hasil yang optimal.

Solusi: Masalah ini adalah contoh klasik Pemrograman Dinamis. Meskipun kita bisa menyelesaikan ini dengan kekerasan, itu akan rumit. Kita dapat menggunakan pendekatan masalah LIS untuk menyelesaikan ini. Segera setelah kita melintasi larik, kita harus menemukan jumlah lompatan minimum untuk mencapai posisi itu (indeks) dan memperbarui larik hasil kita. Setelah kita mencapai akhir, kita memiliki solusi optimal pada indeks terakhir dalam array hasil.

Bagaimana kita dapat menemukan jumlah lompatan yang optimal untuk setiap posisi (indeks)? Untuk indeks pertama, jumlah lompatan optimal akan menjadi nol. Harap dicatat bahwa jika nilai pada indeks pertama adalah nol, kita tidak dapat melompat ke elemen apa pun dan mengembalikan tak terbatas. Untuk elemen n + 1, inisialisasi hasil[n + 1] sebagai tak hingga. Kemudian kita harus melalui loop dari 0 ... n, dan pada setiap indeks i, kita akan melihat apakah kita dapat melompat ke n + 1 dari i atau tidak. Jika memungkinkan, lihat apakah jumlah lompatan (hasil[i] + 1) kurang dari hasil[n + 1], lalu perbarui hasil[n + 1], jika tidak lanjutkan ke indeks berikutnya.

```
//Define MAX 1 less so that adding 1 doesn't make it 0
#define MAX 0xFFFFFFFE;
unsigned int jump(int *array, int n) {
    unsigned answer, int *result = new unsigned int[n];
    int i, j;
    //Boundary conditions
    if(n==0 || array[0] == 0)
        return MAX;
    result[0] = 0;  //no need to jump at first element
    for (i = 1; i < n; i++) {
            result[i] = MAX; //Initialization of result[i]
            for (j = 0; j < i; j++) {
                    //check if jump is possible from j to is
                    if(array[j] >= (i-j)) {
                            //check if better solution available
                            if((result[j] + 1) < result[i])
                                    result[i] = result[j] + 1;  //updating result[i]
                    }
            }
    }
    answer = result[n-1]; //return result[n-1]
    delete[] result;
    return answer;
}
```

Kode di atas akan mengembalikan jumlah lompatan yang optimal. Untuk menemukan indeks lompatan juga, kita dapat dengan mudah memodifikasi kode sesuai kebutuhan.

Kompleksitas Waktu: Karena kita menjalankan 2 loop di sini dan iterasi dari 0 ke i di setiap loop maka total waktu yang dibutuhkan adalah 1 + 2 + 3 + 4 + ... + n – 1. Jadi efisiensi waktu O(n) = O (n * (n – 1)/2) = $O(n^2)$.

Kompleksitas Ruang: O(n) ruang untuk larik hasil.

Soal-46 Jelaskan apa yang akan terjadi jika algoritma pemrograman dinamis dirancang untuk memecahkan masalah yang tidak memiliki sub-masalah yang tumpang tindih.

Solusi: Ini hanya akan membuang-buang memori, karena jawaban dari sub-masalah tidak akan pernah digunakan lagi. Dan *running time*nya akan sama dengan menggunakan algoritma *Divide & Conquer.*

Soal-47 Natal sudah dekat. Anda membantu Sinterklas membagikan hadiah kepada anak-anak. Untuk memudahkan pengiriman, Anda diminta untuk membagi n hadiah menjadi dua kelompok sedemikian rupa sehingga perbedaan berat kedua kelompok ini diminimalkan. Bobot setiap hadiah adalah bilangan bulat positif. Silakan rancang Algoritma untuk menemukan pembagian optimal yang meminimalkan nilai perbedaan. Algoritma harus menemukan perbedaan bobot minimal serta pengelompokan dalam waktu O(nS), di mana S adalah bobot total dari n hadiah ini. Jelaskan secara singkat kebenaran Algoritma Anda.

Solusi: Soal ini dapat diubah menjadi membuat satu set sedekat $\frac{S}{2}$ mungkin. Kita mempertimbangkan masalah yang setara dengan membuat satu set sedekat $W=\left\lfloor\frac{S}{2}\right\rfloor$ mungkin. Tentukan FD(i,w) sebagai jarak minimal antara berat tas dan W saat menggunakan hadiah pertama saja. WLOG, kita dapat mengasumsikan berat tas selalu kurang dari atau sama dengan W. Kemudian isi tabel DP untuk 0≤i≤ n dan 0≤ w W di mana F(0, w) = W untuk semua w, dan Ini membutuhkan waktu O(nS). FD(n,W) adalah celah minimum. Akhirnya, untuk merekonstruksi jawabannya, kita mundur dari (n,W). Selama backtracking, jika FD(i,j) = FD(i – 1,j) maka i tidak terpilih dalam bag dan kita pindah ke F(i – 1,j). Jika tidak, i dipilih dan kita pindah ke F(i – 1,j – w_i).

$$FD(i,w) = min\{FD(i-1,w-w_i)-w_i, FD(i-1,w)\}\ if\ \{FD(i-1,w-w_i) \geq w_i$$
$$= FD(i-1,w)\ \text{otherwise}$$

Soal-48 Sebuah sirkus sedang merancang menara rutin yang terdiri dari orang-orang yang berdiri di atas bahu satu sama lain. Untuk alasan praktis dan estetis, setiap orang harus lebih pendek dan lebih ringan daripada orang di bawahnya. Mengingat tinggi dan berat masing-masing orang di sirkus, tulislah metode untuk menghitung jumlah orang sebanyak mungkin di menara seperti itu.

Solusi: Sama dengan masalah Box stacking dan Longest meningkat subsequence (LIS).

BAB 12
KELAS KOMPLEKSITAS

12.1 PENDAHULUAN

Dalam bab-bab sebelumnya kita telah memecahkan masalah dengan kompleksitas yang berbeda. Beberapa algoritma memiliki tingkat pertumbuhan yang lebih rendah sementara yang lain memiliki tingkat pertumbuhan yang lebih tinggi. Masalah dengan tingkat pertumbuhan yang lebih rendah disebut masalah yang mudah (atau masalah yang mudah dipecahkan) dan masalah dengan tingkat pertumbuhan yang lebih tinggi disebut masalah yang sulit (atau masalah yang sulit dipecahkan). Klasifikasi ini dilakukan berdasarkan waktu berjalan (atau memori) yang dibutuhkan algoritma untuk memecahkan masalah.

Tabel 12.1 klasifikasi pemecahan masalah

Kompleksitas Waktu	Nama	Contoh	Masalah
O(1)	Konstan	Menambahkan elemen ke depan daftar tertaut	Masalah mudah dipecahkan
O (masuk)	Logaritma	Menemukan elemen dalam pohon pencarian biner	
Pada)	Linier	Menemukan elemen dalam array yang tidak disortir	
O (tidak masuk)	Logaritma Linier	Gabungkan sort	
$O(n^2)$	Kuadrat	Jalur terpendek antara dua node dalam Grafik	
$O(n^3)$	Kubik	Perkalian Matriks	
$O(2^n)$	Eksponensial	Masalah Menara Hanoi	Masalah yang sulit dipecahkan
$O(n^1)$	Faktorial	Permutasi dari sebuah string	

Ada banyak masalah yang kita tidak tahu solusinya. Semua masalah yang telah kita lihat sejauh ini adalah masalah yang dapat diselesaikan oleh komputer dalam waktu deterministik. Sebelum memulai diskusi, mari kita lihat terminologi dasar yang kita gunakan dalam bab ini.

12.2 WAKTU POLINOMIAL/EKSPONSIAL

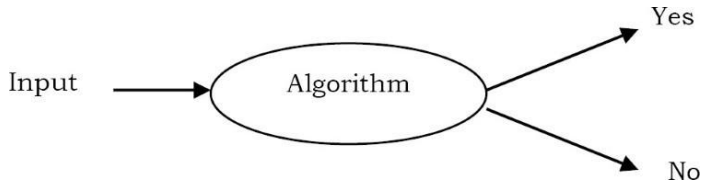
Waktu eksponensial berarti, pada dasarnya, mencoba setiap kemungkinan (misalnya, algoritma backtracking) dan sifatnya sangat lambat. Waktu polinomial berarti memiliki

beberapa algoritma pintar untuk memecahkan masalah, dan kita tidak mencoba setiap kemungkinan. Secara matematis, kita dapat merepresentasikannya sebagai:

- Waktu polinomial adalah $O(n^k)$, untuk beberapa k.
- Waktu eksponensial adalah $O(k^n)$, untuk beberapa k.

12.3 APA ITU MASALAH KEPUTUSAN?

Masalah keputusan adalah pertanyaan dengan jawaban ya/tidak dan jawabannya tergantung pada nilai input. Misalnya, masalah "Diberikan array n angka, periksa apakah ada duplikat atau tidak?" adalah masalah keputusan. Jawaban untuk masalah ini dapat berupa ya atau tidak tergantung pada nilai array input.



Gambar 12.1 Alur masalah keputusan

12.4 PROSEDUR KEPUTUSAN

Untuk masalah keputusan yang diberikan mari kita asumsikan kita telah memberikan beberapa algoritma untuk menyelesaikannya. Proses pemecahan masalah keputusan yang diberikan dalam bentuk algoritma disebut prosedur keputusan untuk masalah itu.

12.5 APA ITU ALUR KOMPLEKSITAS

Dalam ilmu komputer, untuk memahami masalah yang solusinya tidak ada, masalah dibagi ke dalam kelas dan kita menyebutnya sebagai kelas kompleksitas. Dalam teori kompleksitas, kelas kompleksitas adalah sekumpulan masalah dengan kompleksitas terkait. Ini adalah cabang teori komputasi yang mempelajari sumber daya yang dibutuhkan selama komputasi untuk memecahkan masalah yang diberikan. Sumber daya yang paling umum adalah waktu (berapa banyak waktu yang dibutuhkan algoritma untuk menyelesaikan masalah) dan ruang (berapa banyak memori yang dibutuhkan).

12.6 JENIS KELAS KOMPLEKSITAS

Kelas kompleksitas P adalah himpunan masalah keputusan yang dapat diselesaikan oleh mesin deterministik dalam waktu polinomial (P adalah waktu polinomial). Masalah P adalah sekumpulan masalah yang solusinya mudah ditemukan.

Kelas NP

Kelas kompleksitas NP (NP adalah singkatan dari waktu polinomial non-deterministik) adalah himpunan masalah keputusan yang dapat diselesaikan oleh mesin non-deterministik

dalam waktu polinomial. Masalah kelas NP mengacu pada serangkaian masalah yang solusinya sulit ditemukan, tetapi mudah diverifikasi.

Untuk pemahaman yang lebih baik, mari kita pertimbangkan sebuah perguruan tinggi yang memiliki 500 siswa. Juga, asumsikan bahwa ada 100 kamar yang tersedia untuk siswa. Pemilihan 100 mahasiswa harus berpasangan dalam satu ruangan, tetapi dekan mahasiswa memiliki daftar pasangan mahasiswa tertentu yang tidak dapat satu kamar karena suatu alasan.

Jumlah total kemungkinan pasangan terlalu besar. Tetapi solusi (daftar pasangan) yang diberikan kepada dekan, mudah untuk memeriksa kesalahan. Jika salah satu pasangan terlarang ada dalam daftar, itu adalah kesalahan. Dalam masalah ini, kita dapat melihat bahwa memeriksa setiap kemungkinan sangat sulit, tetapi hasilnya mudah untuk divalidasi.

Artinya, jika seseorang memberi kita solusi untuk masalah tersebut, kita dapat memberi tahu mereka apakah itu benar atau tidak dalam waktu polinomial. Berdasarkan pembahasan di atas, untuk soal kelas NP jika jawabannya ya, maka ada bukti dari fakta tersebut, yang dapat diverifikasi dalam waktu polinomial.

Kelas Co-NP

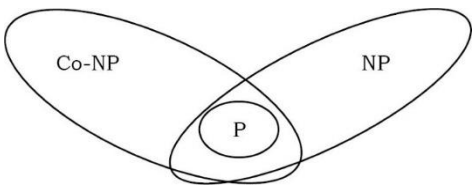
Co – NP adalah kebalikan dari NP (pelengkap NP). Jika jawaban soal Co – NP adalah tidak, maka ada bukti dari fakta ini yang dapat diperiksa dalam waktu polinomial.

Tabel 12.2 jawaban soal Co - NP

P	Dapat dipecahkan dalam waktu polinomial
NP	Ya, jawaban dapat diperiksa dalam waktu polinomial
Co-NP	Tidak ada jawaban yang dapat diperiksa dalam waktu polinomial

Hubungan antara P, NP dan Co-NP

Setiap masalah keputusan di P juga ada di NP. Jika masalah ada di P, kita dapat memverifikasi jawaban YA dalam waktu polinomial. Demikian pula, setiap masalah di P juga ada di Co – NP.



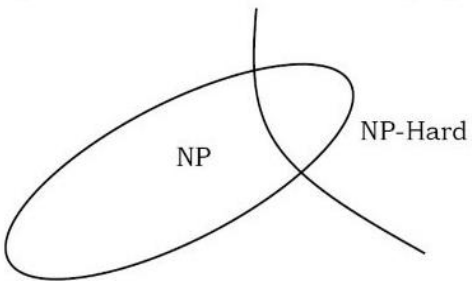
Gambar 12.2 hubungan antara P, NP, dan Co-NP

Salah satu pertanyaan terbuka yang penting dalam ilmu komputer teoretis adalah apakah P = NP atau tidak. Tidak ada yang tahu. Secara intuitif, harus jelas bahwa P NP, tetapi tidak ada yang tahu bagaimana membuktikannya. Pertanyaan terbuka lainnya adalah apakah NP dan Co – NP berbeda. Meskipun kita dapat memverifikasi setiap jawaban YA dengan cepat, tidak ada alasan untuk berpikir bahwa kita juga dapat memverifikasi jawaban TIDAK dengan cepat. Secara umum diyakini bahwa NP Co – NP, tetapi sekali lagi tidak ada yang tahu bagaimana membuktikannya.

Kelas NP-hard

Ini adalah kelas masalah sehingga setiap masalah di NP direduksi menjadi itu. Semua masalah NP-hard tidak ada di NP, jadi perlu waktu lama untuk memeriksanya. Artinya, jika seseorang memberi kita solusi untuk masalah NP-hard, kita membutuhkan waktu lama untuk memeriksa apakah itu benar atau tidak. Masalah K adalah NP-hard menunjukkan bahwa jika algoritma waktu polinomial (solusi) ada untuk K maka algoritma waktu polinomial untuk setiap masalah adalah NP. Dengan demikian:

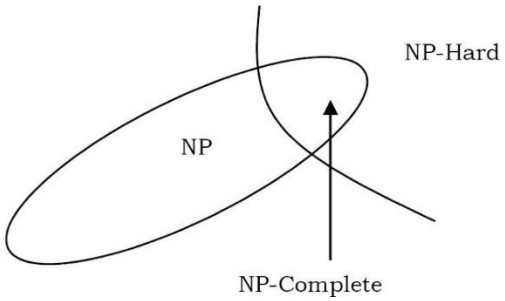
K adalah NP-hard menyiratkan bahwa jika K dapat diselesaikan dalam waktu polinomial, maka P = NP



Gambar 12.3 kelas NP-hard

Kelas NP-complete

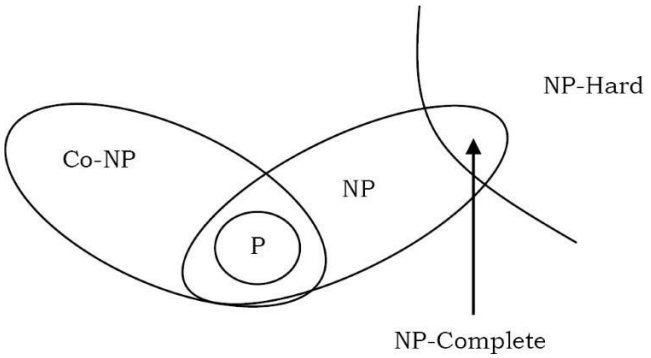
Akhirnya, sebuah masalah adalah NP-complete jika merupakan bagian dari NP-hard dan NP. Masalah NP-complete adalah masalah tersulit di NP. Jika ada yang menemukan algoritma waktu polinomial untuk satu masalah NP-complete, maka kita dapat menemukan algoritma waktu polinomial untuk setiap masalah NP-complete. Ini berarti bahwa kita dapat memeriksa jawaban dengan cepat dan setiap masalah di NP berkurang menjadi itu.



Gambar 12.4 kelas NP-complete

Hubungan antara P, NP Co-NP, NP-Hard dan NP-Complete

Dari pembahasan di atas, kita dapat menulis hubungan antara komponen yang berbeda seperti yang ditunjukkan di bawah ini (ingat, ini hanya asumsi).



Gambar 12.5 hubungan antar kelas diatas

Himpunan masalah yang NP-hard adalah superset ketat dari masalah yang NP-complete. Beberapa masalah (seperti masalah berhenti) adalah NP-hard, tetapi tidak di NP. Masalah NP-hard mungkin mustahil untuk dipecahkan secara umum. Kita dapat membedakan kesulitan antara masalah NP-hard dan NP-complete karena kelas NP mencakup segala sesuatu yang lebih mudah daripada masalah "terberat" - jika masalah tidak ada di NP, itu lebih sulit daripada semua masalah di NP.

Apakah P==NP?

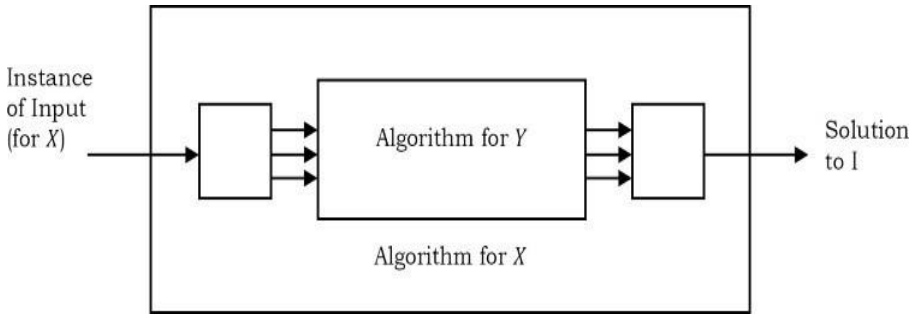
Jika P = NP, berarti setiap masalah yang dapat diperiksa dengan cepat dapat diselesaikan dengan cepat (ingat perbedaan antara memeriksa apakah suatu jawaban benar dan benar-benar menyelesaikan suatu masalah).

Ini adalah pertanyaan besar (dan tidak ada yang tahu jawabannya), karena saat ini banyak masalah NP-complete yang tidak dapat diselesaikan dengan cepat. Jika P = NP, berarti ada cara untuk menyelesaikannya dengan cepat. Ingatlah bahwa "cepat" berarti bukan coba-coba. Itu bisa memakan waktu satu miliar tahun, tetapi selama kita tidak menggunakan trial and error, itu cepat. Di masa depan, sebuah komputer akan dapat mengubah miliaran tahun itu menjadi beberapa menit.

12.7 PENGURANGAN

Sebelum membahas pengurangan, mari kita pertimbangkan skenario berikut. Asumsikan bahwa kita ingin menyelesaikan masalah X tetapi merasa itu sangat rumit. Dalam hal ini apa yang kita lakukan?

Hal pertama yang terlintas dalam pikiran adalah, jika kita memiliki masalah yang mirip dengan X (sebut saja Y), maka kita mencoba memetakan X ke Y dan menggunakan solusi Y untuk menyelesaikan X juga. Proses ini disebut reduksi.



Gambar 12.6 pemetaan X ke Y dengan solusi Y

Untuk memetakan masalah X ke masalah Y, kita memerlukan beberapa algoritma dan itu mungkin membutuhkan waktu linier atau lebih. Berdasarkan pembahasan ini biaya penyelesaian masalah X dapat diberikan sebagai:

Biaya penyelesaian X = Biaya penyelesaian Y + Waktu pengurangan

Sekarang, mari kita pertimbangkan skenario lainnya. Untuk memecahkan masalah X, terkadang kita mungkin perlu menggunakan Algoritma Y (solusi) beberapa kali. Dalam hal itu,

*Biaya penyelesaian X = Jumlah Kali * Biaya penyelesaian X + Waktu pengurangan*

Hal utama dalam NP-Complete adalah *reducibility*. Artinya, kita mereduksi (atau mentransformasi) masalah NP-Complete yang diberikan ke masalah NP-Complete lain yang diketahui. Karena masalah NP-complete sulit dipecahkan dan untuk membuktikan bahwa masalah NP-complete yang diberikan sulit, kita mengambil satu masalah sulit yang ada (yang dapat kita buktikan sulit) dan mencoba memetakan masalah yang diberikan untuk itu dan akhirnya kita membuktikan bahwa masalah yang diberikan sulit.

Catatan: *Tidak wajib mereduksi soal yang diberikan menjadi soal yang sulit diketahui untuk membuktikan kekerasannya. Terkadang, kita mengurangi masalah sulit yang diketahui menjadi masalah yang diberikan.*

Masalah Penting NP-complete (Pengurangan)

Soal Satisfiability: Rumus boolean berada dalam bentuk normal konjungtif (CNF) jika merupakan konjungsi (AND) dari beberapa klausa, yang masing-masing merupakan disjungsi (OR) dari beberapa literal, yang masing-masing merupakan variabel atau negasinya.

Contoh:

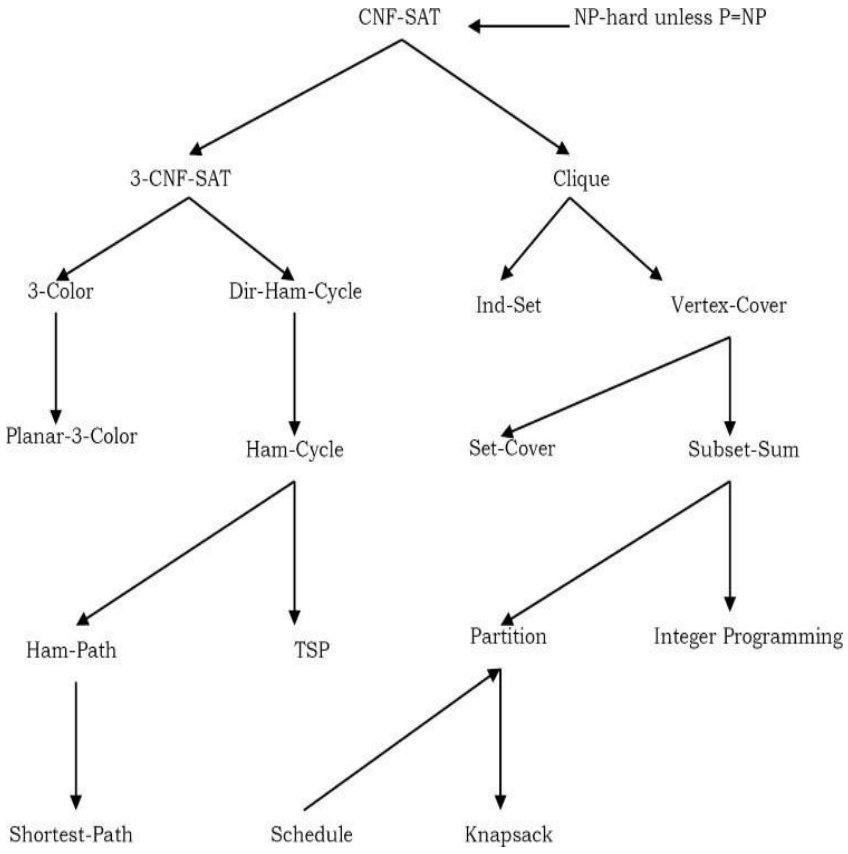
(a b c ∨ d e)∧(b ~c ∨ ~d) (~a ∨ c ∨ d) (a ~b)

Rumus 3-CNF adalah rumus CNF dengan tepat tiga literal per klausa. Contoh sebelumnya bukanlah rumus 3-CNF, karena klausa pertamanya memiliki lima literal dan klausa terakhirnya hanya memiliki dua.

2-SAT Soal: 3-SAT hanya SAT terbatas pada rumus 3-CNF: Diberikan rumus 3-CNF, apakah ada penugasan ke variabel sehingga rumus tersebut bernilai TRUE?

2-SAT Soal: 2-SAT hanyalah SAT terbatas pada rumus 2-CNF: Diberikan rumus 2-CNF, apakah ada penugasan ke variabel sehingga rumus tersebut bernilai TRUE?

Masalah Kepuasan Sirkuit: Diberikan sirkuit kombinasional boolean yang terdiri dari gerbang AND, OR dan NOT, apakah ini memenuhi?. Itu berarti, mengingat rangkaian boolean yang terdiri dari gerbang AND, OR dan NOT yang dihubungkan dengan benar oleh kabel, masalah Sirkuit-SAT adalah memutuskan apakah ada penugasan input yang *output*nya BENAR.



Gambar 12.7 sirkuit SAT

Masalah Jalur Hamilton (Ham-Path): Diberikan grafik tak berarah, apakah ada jalur yang mengunjungi setiap titik tepat satu kali?

Masalah Siklus Hamilton (Ham-Cycle): Diberikan grafik tak berarah, apakah ada siklus (dengan simpul awal dan akhir sama) yang mengunjungi setiap simpul tepat satu kali?

Masalah Siklus Hamiltonian Berarah (Dir-Ham-Cycle): Diberikan grafik berarah, apakah ada siklus (di mana simpul awal dan akhir sama) yang mengunjungi setiap simpul tepat satu kali?
Traveling Salesman Problem (TSP): Diberikan daftar kota dan jarak berpasangannya, masalahnya adalah menemukan tur sesingkat mungkin yang mengunjungi setiap kota tepat satu kali.
Masalah Jalur Terpendek (Jalur Terpendek): Diberikan grafik berarah dan dua simpul s dan t, periksa apakah ada jalur sederhana terpendek dari s ke t.
Pewarnaan Grafik: Pewarnaan k dari suatu grafik adalah memetakan salah satu dari k 'warna' ke setiap titik, sehingga setiap sisi memiliki dua warna berbeda pada titik ujungnya. Masalah pewarnaan grafik adalah menemukan jumlah warna terkecil yang mungkin dalam pewarnaan legal.
3- Masalah warna: Diberikan sebuah grafik, apakah mungkin untuk mewarnai grafik dengan 3 warna sedemikian rupa sehingga setiap sisi memiliki dua warna yang berbeda?
Clique (juga disebut Grafik lengkap): Diberikan Grafik, masalah CLIQUE adalah menghitung jumlah node dalam subgrafik lengkap terbesarnya. Artinya, kita perlu mencari subgrafik maksimum yang juga merupakan grafik lengkap.
Masalah Himpunan Independen (Ind_Set): Misalkan G adalah grafik arbitrer. Himpunan bebas di G adalah himpunan bagian dari simpul-simpul G yang tidak memiliki sisi di antara simpul-simpul tersebut. Masalah himpunan bebas maksimum adalah ukuran himpunan bebas terbesar dalam grafik tertentu.
Masalah Penutup Titik (Vertex-Cover): Penutup titik dari suatu grafik adalah himpunan titik-titik yang menyentuh setiap sisi dalam grafik. Masalah penutup simpul adalah menemukan penutup simpul terkecil dalam grafik tertentu.
Soal Jumlah Subset (Jumlah Subset): Diberikan himpunan S dari bilangan bulat dan bilangan bulat T, tentukan apakah 5 memiliki subset yang elemen-elemennya berjumlah T.
Pemrograman Integer: Diberikan bilangan bulat bi, aij temukan 0/1 variabel xi yang memenuhi sistem persamaan linier.

$$\sum_{j=1}^{N} \mathrm{a_{ij}x_j} = b_i \quad 1 \le i \le M$$
$$\mathrm{x_j} \in \{0,1\}\ 1 \le j \le N$$

Pada gambar, panah menunjukkan pengurangan. Misalnya, Siklus Ham (Masalah Siklus Hamilton) dapat direduksi menjadi CNF-SAT. Sama halnya dengan pasangan masalah apa pun. Untuk pembahasan kita, kita dapat mengabaikan proses reduksi untuk setiap masalah. Ada teorema yang disebut Teorema Cook yang membuktikan bahwa masalah Circuit satisfiability adalah NP-hard. Itu berarti, Kepuasan sirkuit adalah masalah NP-hard yang diketahui.

Catatan: *Karena soal-soal di bawah ini adalah NP-Complete, maka soal-soal tersebut juga NP dan NP-hard. Untuk kesederhanaan kita dapat mengabaikan bukti untuk pengurangan ini.*

12.8 KELAS KOMPLEKSITAS: MASALAH & SOLUSI

Soal-1 Apa yang dimaksud dengan algoritma cepat?

Solusi: Algoritma cepat (solusi) berarti bukan solusi coba-coba. Ini bisa memakan waktu satu miliar tahun, tetapi selama kita tidak menggunakan trial and error, itu efisien. Komputer masa depan akan mengubah miliaran tahun itu menjadi beberapa menit.

Soal-2 Apa yang dimaksud dengan algoritma yang efisien?

Solusi: Suatu algoritma dikatakan efisien jika memenuhi sifat-sifat berikut:

- Skala dengan ukuran input.
- Jangan pedulikan konstanta.
- Waktu berjalan asimtotik: waktu polinomial.

Soal-3 Bisakah kita menyelesaikan semua masalah dalam waktu polinomial?

Solusi: Tidak. Jawabannya sepele karena kita telah melihat banyak masalah yang membutuhkan waktu lebih dari polinomial.

Soal-4 Apakah ada masalah yang NP-hard?

Solusi: Menurut definisi, NP-hard menyiratkan bahwa itu sangat sulit. Itu berarti sangat sulit untuk membuktikan dan untuk memverifikasi bahwa itu sulit. Teorema Cook membuktikan bahwa masalah Circuit satisfiability adalah NP-hard.

Soal-5 Untuk masalah 2-SAT, manakah dari berikut ini yang dapat diterapkan?

(a) P
(b) NP
(c) CoNP
(d) NP-Hard
(e) CoNP-Hard
(f) NP-complete
(g) CoNP-complete

Solusi: 2-SAT dapat dipecahkan dalam waktu poli. Jadi P, NP, dan CoNP.

Soal-6 Untuk soal 3-SAT, manakah dari berikut ini yang dapat diterapkan?

(a) P
(b) NP
(c) CoNP
(d) NP-Hard
(e) CoNP-Hard

(f) NP-complete
(g) CoNP-complete

Solusi: 3-SAT adalah NP-complete. Jadi NP, NP-Hard, dan NP-complete.

Soal-7 Untuk masalah 2-Clique, manakah dari berikut ini yang dapat diterapkan?
(a) P
(b) NP
(c) CoNP
(d) NP-Hard
(e) CoNP-Hard
(f) NP-complete
(g) CoNP-complete

Solusi: 2-Clique dapat diselesaikan dalam waktu poli (periksa tepi antara semua pasangan titik dalam waktu O(n2)). Jadi P.NP, dan CoNP.

Soal-8 Untuk soal 3-Clique, manakah dari berikut ini yang dapat diterapkan?
(a) P
(b) NP
(c) CoNP
(d) NP-Hard
(e) CoNP-Hard
(f) NP-complete
(g) CoNP-complete

Solusi: 3-Clique dapat diselesaikan dalam poli-waktu (periksa segitiga antara semua vertex-triplet dalam waktu O(n3)). Jadi P, NP, dan CoNP.

Soal-9 Pertimbangkan masalah penentuan. Untuk rumus boolean yang diberikan, periksa apakah setiap penugasan ke variabel memenuhinya. Manakah dari berikut ini yang dapat diterapkan?
(a) P
(b) NP
(c) CoNP
(d) NP-Hard
(e) CoNP-Hard
(f) NP-complete
(g) CoNP-complete

Solusi: Tautologi adalah masalah pelengkap untuk Satisfiability, yaitu NP-complete, jadi Tautology adalah CoNP-complete. Jadi itu adalah CoNP, CoNP-hard, dan CoNP-complete.

Soal-10 Biarkan S menjadi masalah NP-complete dan Q dan R menjadi dua masalah lain yang tidak diketahui di NP. Q adalah waktu polinomial direduksi menjadi S dan S adalah waktu polinomial direduksi menjadi R. Manakah dari pernyataan berikut yang benar?

(a) R adalah NP-complete
(b) R adalah NP-hard
(c) Q adalah NP-complete
(d) Q adalah NP-hard

Solusi: R adalah NP-hard (b).

Soal-11 Misalkan A adalah masalah pencarian siklus Hamilton pada grafik G = (V ,E), dengan
|V| habis dibagi 3 dan B masalah menentukan apakah siklus Hamiltonian ada dalam Grafik tersebut. Manakah dari berikut ini yang benar?

(a) Baik A dan B adalah NP-hard
(b) A adalah NP-hard, tetapi B tidak
(c) A adalah NP-hard, tetapi B tidak
(d) Baik A maupun B bukan NP-hard

Solusi: Baik A dan B adalah NP-hard (a).

Soal-12 Biarkan A menjadi masalah yang termasuk dalam kelas NP. Nyatakan manakah dari berikut ini yang benar?

(a) Tidak ada algoritma waktu polinomial untuk A.
(b) Jika A dapat diselesaikan secara deterministik dalam waktu polinomial, maka P = NP.
(c) Jika A adalah NP-hard, maka A adalah NP-complete.
(d) A mungkin tidak dapat ditentukan.

Solusi: Jika A adalah NP-hard, maka A adalah NP-complete (c).

Soal-13 Misalkan kita mengasumsikan Vertex – Cover diketahui sebagai NP-complete. Berdasarkan reduksi kita, dapatkah kita mengatakan Independen – Set adalah NP-complete?

Solusi: Ya. Ini mengikuti dari dua kondisi yang diperlukan untuk melengkapi NP:

- Himpunan Independen ada di NP, seperti yang tertera pada soal.

- Pengurangan dari masalah NP-complete yang diketahui.

Soal-14 Misalkan Himpunan Independen diketahui NP-complete. Berdasarkan reduksi kita, apakah Vertex Cover NP-complete?

Solusi: Tidak. Dengan mereduksi dari Penutup-Simpul ke Himpunan Independen, kita tidak mengetahui kesulitan menyelesaikan Himpunan Independen. Ini karena Independent-Set masih bisa menjadi masalah yang jauh lebih sulit daripada Vertex-Cover. Kita belum membuktikan itu.

Soal-15 Kelas NP adalah kelas bahasa yang tidak dapat diterima dalam waktu polinomial. Apakah itu benar? Menjelaskan.

Solusi:

- Kelas NP adalah kelas bahasa yang dapat diverifikasi dalam waktu polinomial.
- Kelas P adalah kelas bahasa yang dapat ditentukan dalam waktu polinomial.
- Kelas P adalah kelas bahasa yang dapat diterima dalam waktu polinomial.

P NP dan “bahasa dalam P dapat diterima dalam waktu polinomial”, deskripsi “bahasa dalam NP tidak dapat diterima dalam waktu polinomial” salah.

Istilah NP berasal dari waktu polinomial nondeterministik dan diturunkan dari karakterisasi alternatif dengan menggunakan mesin Turing waktu polinomial nondeterministik. Ini tidak ada hubungannya dengan "tidak dapat diterima dalam waktu polinomial".

Soal-16 Pengkodean yang berbeda akan menyebabkan kompleksitas waktu yang berbeda untuk algoritma yang sama. Apakah itu benar?

Solusi: Benar. Kompleksitas waktu dari algoritma yang sama berbeda antara pengkodean unary dan pengkodean biner. Tetapi jika kedua pengkodean terkait secara polinomial (misalnya pengkodean basis 2 & basis 3), maka perubahan di antara keduanya tidak akan menyebabkan kompleksitas waktu berubah.

Soal-17 Jika P = NP, maka NPC (NP Complete) $\subseteq$ P. Benarkah?

Solusi: Benar. Jika P = NP, maka untuk bahasa apapun L $\in$ NP C (1) L $\in$ NPC (2) L adalah NP-hard. Dengan syarat pertama, L $\in$ NPC $\subseteq$ NP = P $\Rightarrow$NPC $\subseteq$ P.

Soal-18 Jika NPC $\subseteq$ P, maka P = NP. Apakah itu benar?

Solusi: Benar. Semua masalah NP dapat direduksi menjadi masalah NPC arbitrer dalam waktu polinomial, dan masalah NPC dapat diselesaikan dalam waktu polinomial karena NPC $\subseteq$ P. $\Rightarrow$Masalah NP dapat diselesaikan dalam waktu polynomial $\Rightarrow$ NP $\subseteq$ P dan sepele P $\subseteq$ NP menyiratkan NP = P.

BAB 13
KONSEP LAIN-LAIN

13.1 PENDAHULUAN

Dalam bab ini kita akan membahas topik-topik yang berguna untuk wawancara dan ujian.

13.2 PERETASAN PADA PEMROGRAMAN BIT-BIJAKSANA

Dalam C dan C++ kita dapat bekerja dengan bit secara efektif. Pertama mari kita lihat definisi dari setiap operasi bit dan kemudian beralih ke teknik yang berbeda untuk memecahkan masalah. Pada dasarnya, ada enam operator yang didukung C dan C++ untuk manipulasi bit:

Tabel 13.1 enam operator C dan C++

Simbol	Operasi
&	Sedikit demi sedikit AND
1	Sedikit demi sedikit OR
A	Bitwise Eksklusif-OR
≪	Pergeseran kiri sedikit demi sedikit
≫	Pergeseran kanan sedikit demi sedikit
~	Pelengkap sedikit demi sedikit

13.2.1 Sedikit demi sedikit AND

Bitwise AND menguji dua angka biner dan mengembalikan nilai bit 1 untuk posisi di mana kedua angka memiliki satu, dan nilai bit 0 di mana kedua angka tidak memiliki satu:

```
  01001011
& 00010101
----------
  00000001
```

13.2.2 Sedikit demi sedikit OR

Bitwise OR menguji dua bilangan biner dan mengembalikan nilai bit 1 untuk posisi di mana salah satu bit atau kedua bit adalah satu, hasil 0 hanya terjadi ketika kedua bit adalah 0:

```
  01001011
| 00010101
----------
  01011111
```

13.2.3 Bitwise Eksklusif-OR

Eksklusif-OR bitwise menguji dua bilangan biner dan mengembalikan nilai bit 1 untuk posisi di mana kedua bit berbeda; jika mereka sama maka hasilnya adalah 0:

```
  01001011
^ 00010101
----------
  01011110
```

13.2.4 Pergeseran Kiri Bitwise

Pergeseran kiri bitwise memindahkan semua bit dalam angka ke kiri dan mengisi posisi bit yang dikosongkan dengan 0.

```
  01001011
<< 2
--------
  00101100
```

13.2.5 Pergeseran Kanan Bitwise

Pergeseran kanan bitwise memindahkan semua bit dalam nomor ke kanan.

```
  01001011
>> 2
--------
  ??010010
```

Perhatikan penggunaan? untuk fill bit. Dimana shift kiri mengisi posisi yang dikosongkan dengan 0, shift kanan akan melakukan hal yang sama hanya ketika nilainya tidak ditandatangani. Jika nilai ditandatangani maka pergeseran ke kanan akan mengisi posisi bit yang dikosongkan dengan bit tanda atau 0, mana pun yang ditentukan implementasi. Jadi pilihan terbaik adalah jangan pernah menggeser nilai yang ditandatangani dengan benar.

13.2.6 Pelengkap Bitwise

Komplemen bitwise membalikkan bit dalam satu bilangan biner.

```
  01001011
~
---------
  10110100
```

13.2.7 Memeriksa Apakah Bit K-th Diset atau Tidak

Mari kita asumsikan bahwa bilangan yang diberikan adalah n. Kemudian untuk memeriksa bit Kth kita dapat menggunakan ekspresi:

n & (1 K ≪ 1). Jika ekspresinya benar maka kita dapat mengatakan bit Kth diset (artinya, diset ke 1).

Contoh:

$$n = 01001011 \text{ and } K = 4$$
$$1 \ll K - 1 \qquad 00001000$$
$$n \,\&\, (1 \ll K - 1) \quad 00001000$$

13.2.8 Mengatur Bit K-th

Untuk bilangan n yang diberikan, untuk menyetel bit ke-K kita dapat menggunakan ekspresi: n | 1 ≪ (K – 1)

Contoh:

$$\begin{aligned} n &= 01001011 \text{ and } K = 3 \\ 1 \ll K-1 &\quad 00000100 \\ n \mid (1 \ll K-1) &\quad 01001111 \end{aligned}$$

13.2.9 Menghapus K-th Bit

Untuk menghapus bit K dari bilangan n yang diberikan, kita dapat menggunakan ekspresi: n & ~(1 ≪ K – 1)

Contoh:

$$\begin{aligned} n &= 01001011 \text{ and } K = 4 \\ 1 \ll K-1 &\quad 00001000 \\ \sim(1 \ll K-1) &\quad 11110111 \\ n \,\&\, \sim(1 \ll K-1) &\quad 01000011 \end{aligned}$$

13.2.10 Mengalihkan Bit K-th

Untuk bilangan n yang diberikan, untuk mengubah bit Kth kita dapat menggunakan ekspresi: n ^(1 ≪ K – 1)

Contoh:

$$\begin{aligned} n &= 01001011 \text{ and } K = 3 \\ 1 \ll K-1 &\quad 00000100 \\ n \,\hat{}\, (1 \ll K-1) &\quad 01001111 \end{aligned}$$

13.2.11 Mengalihkan Satu Bit Paling Kanan

Untuk bilangan n yang diberikan, untuk mengubah satu bit paling kanan kita dapat menggunakan ekspresi: n & n – 1

Contoh:

$$\begin{aligned} n &= 01001011 \\ n-1 &\quad 01001010 \\ n \,\&\, n-1 &\quad 01001010 \end{aligned}$$

13.2.12 Mengisolasi Satu Bit Paling Kanan

Untuk bilangan n yang diberikan, untuk mengisolasi satu bit paling kanan kita dapat menggunakan ekspresi: n & – n

Contoh:

$$\begin{aligned} n &= 01001011 \\ -n &\quad 10110101 \\ n \,\&\, -n &\quad 00000001 \end{aligned}$$

Catatan: *Untuk menghitung –n, gunakan representasi komplemen dua. Itu berarti, alihkan semua bit dan tambahkan 1.*

13.2.13 Mengisolasi Bit Nol Paling Kanan

Untuk bilangan n yang diberikan, untuk mengisolasi bit nol paling kanan kita dapat menggunakan ekspresi: ~n & n + 1

Contoh:

$$\begin{array}{cc} n = & 01001011 \\ \sim n & 10110100 \\ n+1 & 01001100 \\ \sim n \,\&\, n+1 & 00000100 \end{array}$$

13.2.14 Memeriksa Apakah Angka Adalah Kekuatan 2 atau Tidak

Diberikan angka n, untuk memeriksa apakah angka tersebut dalam bentuk 2^n atau tidak, kita dapat menggunakan ekspresi:

jika(n & n – 1 == 0)

Contoh:

$$\begin{array}{cc} n = & 01001011 \\ n-1 & 01001010 \\ n \,\&\, n-1 & 01001010 \\ if(n \,\&\, n-1 == 0) & 0 \end{array}$$

13.2.15 Mengalikan Angka dengan Kekuatan 2

Untuk bilangan n yang diberikan, untuk mengalikan bilangan tersebut dengan 2^K kita dapat menggunakan ekspresi: n K

Contoh:

$$\begin{array}{cc} n = 00001011 \text{ and } K = 2 \\ n \gg K & 00010010 \end{array}$$

13.2.17 Menemukan Modulo dari Bilangan yang Diberikan

Untuk bilangan n yang diberikan, untuk menemukan %8 kita dapat menggunakan ekspresi: n & 0x7. Demikian pula, untuk menemukan %32, gunakan ekspresi: n & 0x1F

Catatan: *Demikian pula, kita dapat menemukan nilai modulo dari bilangan berapa pun.*

13.2.18 Membalikkan Angka Biner

Untuk bilangan n yang diberikan, untuk membalikkan bit (reverse (mirror) bilangan biner) kita dapat menggunakan potongan kode berikut:

```
unsigned int n, nReverse = n;
int s = sizeof(n);
for (; n; n >>= 1) {
    nReverse <<= 1;
    nReverse |= n & 1;
    s--;
}
nReverse <<= s;
```

Kompleksitas Waktu: Ini membutuhkan satu iterasi per bit dan jumlah iterasi tergantung pada ukuran nomor.

13.2.19 Menghitung Angka Satu dalam Angka

Untuk bilangan n tertentu, untuk menghitung jumlah 1 dalam representasi binernya, kita dapat menggunakan salah satu metode berikut.

Metode 1: Proses sedikit demi sedikit dengan bitwise dan operator

```
unsigned int n;
unsigned int count=0;
while(n) {
    count += n & 1;
    n >>= 1;
}
```

Kompleksitas Waktu: Pendekatan ini membutuhkan satu iterasi per bit dan jumlah iterasi tergantung pada sistem.

Metode 2: Menggunakan pendekatan modulo

```
unsigned int n;
unsigned int count=0;
while(n) {
    if(n%2 ==1)
        count++;
    n = n/2;
}
```

Kompleksitas Waktu: Ini membutuhkan satu iterasi per bit dan jumlah iterasi tergantung pada sistem.

Metode 3: Menggunakan pendekatan toggling: n & n – 1

```
unsigned int n;
unsigned int count=0;
while(n) {
    count++;
    n &= n - 1;
}
```

Kompleksitas Waktu: Jumlah iterasi tergantung pada jumlah 1 bit dalam nomor tersebut.

Metode 4: Menggunakan ide preprocessing. Dalam metode ini, kita memproses bit dalam kelompok. Misalnya jika kita memprosesnya dalam kelompok 4 bit sekaligus, kita membuat tabel yang menunjukkan jumlah satu untuk setiap kemungkinan tersebut (seperti yang ditunjukkan di bawah).

Tabel 13.2 tabel kelompok 4 bit

0000→0	0100→1	1000→1	1100→2
0001→1	0101→2	1001→2	1101→3
0010→1	0110→2	1010→2	1110→3
0011→2	0111→3	1011→3	1111→4

Berikut kode untuk menghitung jumlah Is dalam bilangan dengan pendekatan ini:

```
int Table = {0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4};
int count = 0;
for(; n; n >>= 4)
    count = count + Table[n & 0xF];
return count;
```

Kompleksitas Waktu: Pendekatan ini membutuhkan satu iterasi per 4 bit dan jumlah iterasi tergantung pada sistem.

13.2.20 Membuat Mask untuk Trailing Zero's

Untuk angka n yang diberikan, untuk membuat topeng untuk angka nol yang tertinggal, kita dapat menggunakan ekspresi: (n & – n) – 1

Contoh:

$$\begin{array}{rl} n = & 01001011 \\ -n & 10110101 \\ n \,\&\, -n & 00000001 \\ (n \,\&\, -n) - 1 & 00000000 \end{array}$$

Catatan: *Dalam kasus di atas kita mendapatkan topeng sebagai semua nol karena tidak ada nol yang tertinggal.*

13.2.21 Tukar bit ganjil dan genap

Contoh:

n =	01001011
Find even bits of given number (evenN) = n & 0xAA	00001010
Find odd bits of given number (oddN) = n & 0x55	01000001
evenN >>= 1	00000101
oddN <<= 1	10000010
Final Expression: evenN | oddN	10000111

13.2.22 Performa Rata-Rata Tanpa Pembagian

Apakah ada algoritma bit-twiddling untuk menggantikan mid = (low + high) / 2 (digunakan dalam Binary Search and Merge Sort) dengan sesuatu yang lebih cepat?

Kita dapat menggunakan mid = (low + high) >> 1. Perhatikan bahwa menggunakan (low + high) / 2 untuk perhitungan titik tengah tidak akan bekerja dengan benar ketika integer overflow menjadi masalah. Kita dapat menggunakan pemindahan bit dan juga mengatasi kemungkinan masalah luapan: rendah + ((tinggi – rendah)/2) dan operasi pemindahan bit untuk ini adalah rendah + ((tinggi – rendah) >> 1).

13.3 PERTANYAAN PEMROGRAMAN LAINNYA DENGAN SOLUSI

Soal-1 Berikan algoritma untuk mencetak elemen matriks dalam urutan spiral.

Solusi: Solusi non-rekursif melibatkan arah kanan, kiri, atas, bawah, dan menangani indeks yang sesuai. Setelah baris pertama dicetak, arah berubah (dari kanan) ke bawah, baris dibuang dengan menambah batas atas. Setelah kolom terakhir dicetak, arah berubah ke kiri, kolom dibuang dengan mengurangi batas tangan kanan.

```
void Spiral(int **A, int n) {
    int rowStart=0, columnStart=0;
    int rowEnd=n-1, columnEnd=n-1;
    while(rowStart<=rowEnd && columnStart<=columnEnd) {
        int i=rowStart, j=columnStart;
        for(j=columnStart; j<=columnEnd; j++)
            printf("%d ",A[i][j]);
        for(i=rowStart+1, j--; i<=rowEnd; i++)
            printf("%d ",A[i][j]);
        for(j=columnEnd-1, i--; j>=columnStart; j--)
            printf("%d ",A[i][j]);
        for(i=rowEnd-1, j++; i>=rowStart+1; i--)
            printf("%d ",A[i][j]);
        rowStart++; columnStart++; rowEnd--; columnEnd--;
    }
}
```

Kompleksitas Waktu: $O(n^2)$.
Kompleksitas Ruang: O(1).

Soal-2 Berikan algoritma untuk mengocok meja kartu.

Solusi: Asumsikan bahwa kita ingin mengocok larik 52 kartu, dari 0 hingga 51 tanpa pengulangan, seperti yang mungkin kita inginkan untuk setumpuk kartu.
Pertama isi array dengan nilai secara berurutan, lalu lanjutkan array dan tukar setiap elemen dengan elemen yang dipilih secara acak dalam rentang dari dirinya sendiri hingga akhir. Mungkin saja suatu elemen akan bertukar dengan dirinya sendiri, tetapi tidak ada masalah dengan itu.

```
void Shuffle(int cards[], int n){
    srand(time(0));                    // initialize seed randomly
    for (int i=0; i<n; i++)
        cards[i] = i;                  // filling the array with card number
    for (int i=0; i<n; i++) {
        int r = i + (rand() % (52-i)); // Random remaining position.
        int temp = cards[i];
        cards[i] = cards[r];
        cards[r] = temp;
    }
    printf("Shuffled Cards:" );
    for (int i=0; i<n; i++)
        printf("%d ", cards[i]);
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-3 Algoritma pembalikan untuk rotasi array: Tulis sebuah fungsi rotate(A[], d, n) yang memutar A[] dengan ukuran n elemen d. Misalnya, larik 1,2,3,4,5,6,7 menjadi 3,4,5,6,7,1,2 setelah 2 putaran.

Solusi: Perhatikan algoritma berikut.

Algoritma:

Putar(Array[], d, n)
Membalikkan (Array[], 1, d) ;
Membalikkan (Array[], d + 1, n);
Membalikkan (Array[], 1, n);

Biarkan AB menjadi dua bagian dari Array input di mana A = Array[0..d-1] dan B = Array[d..n-1]. Ide algoritmanya adalah:
Membalikkan A untuk mendapatkan ArB. /* Ar kebalikan dari A */
Membalikkan B untuk mendapatkan ArBr. /* Br kebalikan dari B */
Membalikkan semua untuk mendapatkan (ArBr) r = BA.
Misalnya, jika Array[] = [1, 2, 3, 4, 5, 6, 7], d =2 dan n = 7 maka, A = [1, 2] dan B = [3,4, 5, 6, 7]
Membalikkan A, kita dapatkan ArB = [2, 1, 3, 4, 5, 6, 7], Balik B, kita dapatkan ArBr = [2, 1, 7, 6, 5, 4, 3]
Membalikkan semua, kita dapatkan (ArBr)r = [3, 4, 5, 6, 7, 1, 2]

Penerapan:

```
//Function to left rotate Array[] of size n by d
void leftRotate(int Array[], int d, int n) {
    rvereseArrayay(Array, 0, d-1);
    rvereseArrayay(Array, d, n-1);
    rvereseArrayay(Array, 0, n-1);
}
//UTILITY FUNCTIONS: function to print an Arrays
void printArrayay(int Array[], int size){
    for(int i = 0; i < size; i++)
        printf("%d ", Array[i]);
    printf("%\n ");
}
//Function to reverse Array[] from index start to end
void rvereseArrayay(int Array[], int start, int end) {
  int i;
  int temp;
  while(start < end){
       temp = Array[start];
       Array[start] = Array[end];
       Array[end] = temp;
       start++;
       end--;
   }
}
```

```
do, where 1 < k <= n:
        reverse (s, 1, k);
        reverse (s, k + 1, n);
        reverse (s, 1, n);
```

Soal-4 Misalkan Anda diberikan sebuah array s[1...n] dan prosedur terbalik (s,i,j) yang membalik urutan elemen di antara posisi i dan j (keduanya inklusif). Apa urutan berikut?

a) Memutar s ke kiri sebanyak k posisi
b) Daun s tidak berubah
c) Membalikkan semua elemen s
d) Tidak satu pun di atas

Solusi: (b). Efek dari 3 pembalikan di atas untuk setiap k setara dengan rotasi kiri larik berukuran n oleh k [lihat Soal-3].

Soal-5 Menemukan Anagram dalam Kamus: Anda diberikan 2 file ini: dictionary.txt dan jumbles.txt

File thejumbles.txt berisi sekumpulan kata yang diacak. Tugas Anda adalah mencetak kata-kata campur aduk itu, 1 kata menjadi satu baris. Setelah setiap kata acak, cetak daftar kata-kata kamus asli yang dapat dibentuk dengan mengacak kata acak tersebut. Kata kamus yang harus Anda pilih ada di file dictionary.txt. Contoh konten jumbles.txt:

```
nwae: wean anew wane
eslyep: sleepy
rpeoims: semipro imposer promise
ettniner: renitent
ahicryrhe: hierarchy
dica: acid cadi caid
dobol: blood
 ......
%
```

Solusi Langkah-demi-Langkah

Langkah 1: Inisialisasi

- Buka file dictionary.txt dan baca kata-kata ke dalam larik (sebelum melangkah lebih jauh, verifikasi dengan menggemakan kata-kata kembali dari larik ke layar).
- Mendeklarasikan variabel tabel hash.

Langkah 2: Proses Kamus untuk setiap kata kamus dalam larik. Lakukan hal berikut: Kita sekarang memiliki tabel hash di mana setiap kunci adalah bentuk kata kamus yang diurutkan dan nilai yang terkait dengannya adalah string atau larik kata kamus yang diurutkan ke kunci yang sama.

- Hapus baris baru dari akhir setiap kata melalui chomp($word);
- Buat salinan kata yang diurutkan - yaitu mengatur ulang karakter individu dalam string untuk diurutkan menurut abjad

- Pikirkan kata yang diurutkan sebagai nilai kunci dan pikirkan kumpulan semua kata kamus yang mengurutkan kata kunci yang sama persis sebagai nilai kunci
- Minta hashtable untuk melihat apakah kata yang diurutkan sudah menjadi salah satu kuncinya
- Jika belum ada maka masukkan kata yang diurutkan sebagai kunci dan kata asli yang tidak diurutkan sebagai nilainya
- Jika tidak, gabungkan kata yang tidak disortir ke string nilai yang sudah ada di luar sana (beri spasi di antaranya)

Langkah 3: Proses file kata campur aduk

- Membaca file kata campur aduk satu kata pada satu waktu. Saat Anda membaca setiap kata yang campur aduk, potong dan buat salinan yang diurutkan (salinan yang diurutkan adalah kunci Anda)
- Cetak kata campur aduk yang tidak disortir
- Query hashtable untuk salinan diurutkan. Jika ditemukan, cetak nilai terkait pada baris yang sama dengan kunci dan kemudian baris baru.

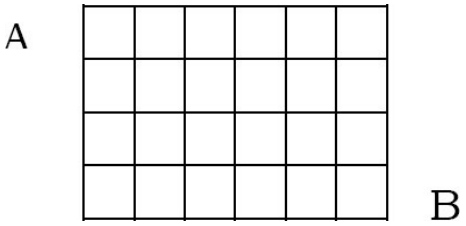
Langkah 4: Rayakan, kita semua sudah selesai

Contoh kode di Perl:

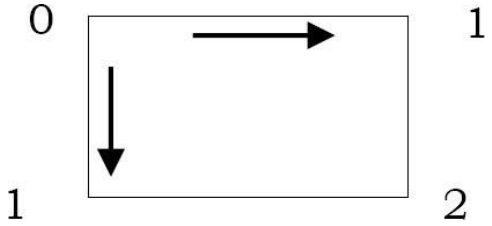
```
#step 1
open("MYFILE",<dictionary.txt>);
while(<MYFILE>){
   $row = $_;
   chomp($row);
   push(@words,$row);
}
my %hashdic = ();
#step 2
foreach $words(@words){
   @not_sorted=split (//, $words);

    @sorted = sort (@not_sorted);

   $name=join("",@sorted);
    if (exists $hashdic{$name})    {
      $hashdic{$name}.=" $words";
   }
```

```
    else {
        $hashdic{$name}=$words;
    }
}
$size=keys %hashdic;
#step 3
open("jumbled",<jumbles.txt>);
while(<jumbled>){
    $jum = $_;
    chomp($jum);
    @not_sorted1=split (//, $jum);
    @sorted1 = sort{@not_sorted1);
    $name1=join("",@sorted1);
    if(length($hashdic{$name1})<1) {
        print "\n$jum : NO MATCHES";
    }
    else {
        @value=split(/ /,$hashdic{$name1});
        print "\n$jum : @values";
    }
}
```

Soal-6 Jalur: Diberikan matriks seperti yang ditunjukkan di bawah ini, hitung banyak cara untuk mencapai tujuan B dari A.



Solusi: Sebelum menemukan solusi, kita mencoba memahami masalah dengan versi yang lebih sederhana. Masalah terkecil yang dapat kita pertimbangkan adalah jumlah kemungkinan rute dalam grid 1 × 1.



Dari gambar diatas dapat diketahui bahwa :

- Dari sudut kiri bawah dan kanan atas hanya ada satu kemungkinan rute ke tujuan.
- Dari sudut kiri atas ada dua kemungkinan rute yang sepele.

Demikian pula, untuk kisi 2x2 dan 3x3, kita dapat mengisi matriks sebagai:

0	1
1	**2**

0	1	1
1	2	3
1	3	**6**

Dari pembahasan di atas, jelas bahwa untuk mencapai sudut kanan bawah dari sudut kiri atas, jalurnya tumpang tindih. Karena jalur unik dapat tumpang tindih pada titik tertentu (sel kisi), kita dapat mencoba mengubah Algoritma sebelumnya, sebagai cara untuk menghindari mengikuti jalur yang sama lagi. Jika kita mulai mengisi 4x4 dan 5x5, kita dapat dengan mudah menemukan solusi berdasarkan konsep matematika masa kecil kita.

0	1	1	1
1	2	3	4
1	3	6	10
1	4	10	**20**

0	1	1	1	1
1	2	3	4	5
1	3	6	10	15
1	4	10	20	35
1	5	15	35	**70**

Apakah Anda dapat mengetahui polanya? Sama halnya dengan segitiga Pascal. Jadi, untuk menemukan banyak cara, kita cukup memindai tabel dan terus menghitungnya sambil bergerak dari kiri ke kanan dan dari atas ke bawah (dimulai dengan kiri-atas). Kita bahkan dapat menyelesaikan masalah ini dengan persamaan matematika segitiga Pascal.

Soal-7 Diberikan sebuah string yang memiliki sekumpulan kata dan spasi, tulislah sebuah program untuk memindahkan spasi ke depan string. Anda perlu melintasi array hanya sekali dan Anda perlu menyesuaikan string di tempatnya.

Input = "pindahkan spasi ini ke awal" Output = "pindahkan spasi ke awal"

Solusi: Pertahankan dua indeks i dan j; melintasi dari akhir ke awal. Jika indeks saat ini berisi char, tukar karakter di indeks i dengan indeks j. Ini akan memindahkan semua spasi ke awal array.

```
void mySwap(char A[],int i,int j){
    char temp=A[i];
    A[i]=A[j];
    A[j]=temp;
}
void moveSpacesToBegin(char A[]){
    int i=strlen(A)-1;
    int j=i;
    for(; j>=0; j--){
        if(!isspace(A[j]))
            mySwap(A,i--,j);
    }
}
```

```
void testCode(int argc, char * argv[]){
    char sparr[]="move these spaces to beginning";
    printf("Value of A is: %s\n", sparr);
    moveSpacesToBegin(sparr);
    printf("Value of A is: %s", sparr);
}
```

Kompleksitas Waktu: O(n) di mana n adalah jumlah karakter dalam array input.
Kompleksitas Ruang: O(1).

Soal-8 Untuk Soal-7, dapatkah kita meningkatkan kompleksitasnya?

Solusi: Kita dapat menghindari operasi swap dengan penghitung sederhana. Tapi, itu tidak mengurangi kompleksitas keseluruhan.

```
void moveSpacesToBegin(char A[]){
    int n=strlen(A)-1,count=n;
    int i=n;
    for(;i>=0;i--){
        if(A[i]!=' ')
            A[count--]=A[i];
    }
    while(count>=0)
        A[count--]=' ';
}
```

```
int testCode(){
    char sparr[]="move these spaces to beginning";
    printf("Value of A is: %s\n", sparr);
    moveSpacesToBegin(sparr);
    printf("Value of A is: %s", sparr);
}
```

Kompleksitas Waktu: O(n) di mana n adalah jumlah karakter dalam array input.
Kompleksitas Ruang: O(1).

Soal-9 Diberikan sebuah string yang memiliki sekumpulan kata dan spasi, tulislah sebuah program untuk memindahkan spasi ke akhir string. Anda perlu melintasi array hanya sekali dan Anda perlu menyesuaikan string di tempatnya.

Input = "pindahkan spasi ini ke akhir" Output = "pindahkan spasi ke akhir"

Solusi: Lintasi larik dari kiri ke kanan. Saat melintasi, pertahankan penghitung untuk elemen non-spasi dalam array. Untuk setiap karakter non-spasi A[i], letakkan elemen pada A[count] dan increment count. Setelah traversal lengkap, semua elemen non-spasi telah digeser ke ujung depan dan hitungan ditetapkan sebagai indeks 0 pertama. Sekarang, yang perlu kita lakukan adalah menjalankan loop yang mengisi semua elemen dengan spasi dari hitungan hingga akhir array .

```
void moveSpacesToEnd(char A[]){
    // Count of non-space elements
    int count = 0;
    int n =strlen(A)-1;
    int i =0;
    for (; i <= n; i++)
        if (!isspace(A[i]))
            A[count++] = A[i];
    while (count <= n)
        A[++count] = ' ';
}
```

```
void testCode(int argc, char * argv[]){
    char sparr[]="move these spaces to end";
    printf("Value of A is: %s\n", sparr);
    moveSpacesToEnd(sparr);
    printf("Value of A is: %s", sparr);
}
```

Kompleksitas Waktu: O(n) di mana n adalah jumlah karakter dalam array input.
Kompleksitas Ruang: O(1).

Soal-10 Memindahkan Nol ke Akhir: Diberikan array n bilangan bulat, pindahkan semua nol dari array yang diberikan ke akhir array. Misalnya, jika array yang diberikan adalah {1, 9, 8, 4, 0, 0, 2, 7, 0, 6, 0}, maka harus diubah menjadi {1, 9, 8, 4, 2, 7, 6, 0, 0, 0, 0}. Urutan semua elemen lainnya harus sama.

Solusi: Pertahankan dua variabel i dan j; dan inisialisasi dengan 0. Untuk setiap elemen array A[i], jika A[i] elemen bukan nol, maka ganti elemen A[j] dengan elemen A[i]. Variabel i akan selalu bertambah sampai n - 1 tetapi kita akan menaikkan j hanya jika elemen yang ditunjuk oleh i bukan nol.

```
void moveZerosToEnd(int A[], int size){
    int i=0,j=0;
    while (i <= size - 1){
        if (A[i] != 0){
            A[j++] = A[i];
        }
        i++;
    }
    while (j <= size - 1)
        A[j++] = 0;
}
```

```
int testCode(){
    int A[ ] = {1,9,8,4,0,0,2,7,0,6,0};
    int i;
    int size = sizeof(A) / sizeof(A[0]);
    moveZerosToEnd(A, size);
    for (i = 0; i <= size - 1; i++)
        printf("%d ", A[i]);
    return 0;
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-11 Untuk Soal-10, dapatkah kita meningkatkan kompleksitasnya?

Solusi: Dengan menggunakan teknik swap sederhana, kita dapat menghindari pengulangan while kedua yang tidak perlu dari kode di atas.

```
void mySwap(int A[],int i,int j){
    int temp=A[i]; A[i]=A[j]; A[j]=temp;
}
void moveZerosToEnd(int A[], int len){
    int i, j;
    for(i=0,j=0; i<len; i++)    {
        if (A[i] !=0)
            mySwap(A,j++,i);
    }
}
```

Kompleksitas Waktu: O(n).
Kompleksitas Ruang: O(1).

Soal-12 Varian Soal-10 dan Soal-11: Diberikan sebuah array yang berisi bilangan negatif dan positif; berikan algoritma untuk memisahkan bilangan positif dan negatif di dalamnya. Juga, pertahankan urutan relatif angka positif dan negatif. Masukan: -5, 3, 2, -1, 4, -8
Keluaran: -5-1 -8342

Solusi: Pada fungsi moveZerosToEnd, ganti saja kondisi A[i] !=0 dengan A[i] < 0.

Soal-13 Diberikan sebuah angka, tukar bit ganjil dan genap.

Solusi:

```
int swap(int num){
   int mask1 = 0xAAAAAAAA;
   int mask2 = 0x55555555;
   return (num << 1 & mask1) | ( num >> 1 & mask2);
}
```

Soal-14 Hitung jumlah bit yang ditetapkan dalam semua angka dari 1 hingga n

Solusi: Kita dapat menggunakan teknik bagian 21.2.19 dan mengulangi semua angka dari 1 hingga n.

```
int countingNumberofOnesIn1toN(unsigned int n){
   int count =0, i = 0, j;
   for (i = 1; i <= n; i++){
      j = i;
      while(j){
         j = j & (j-1);
         count++;
      }
   }
   return count;
}
```

Soal-15 Hitung jumlah bit yang ditetapkan dalam semua angka dari 1 hingga n

Solusi: Kita dapat menggunakan teknik bagian 21.2.19 dan mengulangi semua angka dari 1 hingga n.

```
int countingNumberofOnesIn1toN(unsigned int n){
   int count =0, i = 0, j;
   for (i = 1; i <= n; i++){
      j = i;
      while(j){
         j = j & (j-1);
         count++;
      }
   }
   return count;
}
```

Kompleksitas waktu: O (jumlah bit yang ditetapkan dalam semua angka dari 1 hingga n).

DAFTAR PUSTAKA

Akash. Programming Interviews, tech-queries.blogspot.com.

Alfred V.Aho,J. E. (1983). Data Structures and Algorithms. Addison-Wesley.

Algorithms.Retrieved from cs.princeton.edu/algs4/home

Anderson., S. E. Bit Twiddling Hacks. Retrieved 2010, from Bit Twiddling Hacks: graphics. Stanford. edu

Bentley, J. AT&T Bell Laboratories. Retrieved from AT&T Bell Laboratories.

Bondalapati, K. Interview Question Bank. Retrieved 2010, from Interview Question Bank: halcyon.usc.edu/~kiran/msqs.html

Chen. Algorithms hawaii.edu/~chenx.

Database, P.Problem Database. Retrieved 2010, from Problem Database: datastructures.net

Drozdek, A. (1996). Data Structures and Algorithms in C++.

Ellis Horowitz, S. S. Fundamentals of Data Structures.

Gilles Brassard, P. B. (1996). Fundamentals of Algorithmics.

Hunter., J. Introduction to Data Structures and Algorithms. Retrieved 2010, from Introduction to Data Structures and Algorithms.

James F. Korsh, L. J. Data Structures, Algorithms and Program Style Using C.

John Mongan, N. S. (2002). Programming Interviews Exposed. Wiley-India. .

Judges. Comments on Problems and Solutions. http://www.informatik.uni-ulm.de/acm/Locals/2003/html/judge, html.

Kalid. P, NP, and NP-Complete. Retrieved from P, NP, and NP-Complete.: cs.princeton.edu/~kazad

Knuth., D. E. (1973). Fundamental Algorithms, volume 1 of The Art of Computer Programming. Addison-Wesley.

Leon, J. S. Computer Algorithms. Retrieved 2010, from Computer Algorithms: math.uic.edu/~leon

Leon., J. S. Computer Algorithms, math.uic.edu/~leon/cs-mcs401-s08.

OCF. Algorithms. Retrieved 2010, from Algorithms: ocf.berkeley.edu

Parlante., N. Binary Trees. Retrieved 2010, from cslibrary.stanford.edu: cslibrary.stanford.edu

Patil., V. Fundamentals of data structures. Nirali Prakashan.

Poundstone., W. HOW WOULD YOU MOVE MOUNT FUJI? New York Boston.: Little, Brown and Company.

Pryor, M. Tech Interview. Retrieved 2010, from Tech Interview: techinterview.org

Questions, A. C. A Collection of Technical Interview Questions. Retrieved 2010, from A Collection of Technical Interview Questions

S. Dasgupta, C. P. Algorithms cs.berkeley.edu/~vazirani.

Sedgewick., R. (1988). Algorithms. Addison-Wesley.

Sells, C. (2010). Interviewing at Microsoft. Retrieved 2010, from Interviewing at Microsoft

Shene, C.-K. Linked Lists Merge Sort Implementation.

Sinha, P. Linux Journal. Retrieved 2010, from: linuxjournal.com/article/6828.

Structures., d. D. www.math-cs.gordon.edu. Retrieved 2010, from www.math- cs.gordon.edu

T. H. Cormen, C. E. (1997). Introduction to Algorithms. Cambridge: The MIT press.

Tsiombikas, J. Pointers Explained, nuclear.sdf-eu.org.

Warren., H. S. (2003). Hackers Delight. Addison-Wesley.

Weiss., M. A. (1992). Data Structures and Algorithm Analysis in C.

SANDRASI http://sandrasi-sw.blogspot.in/