YAYASAN PRIMA AGUS TEKNIK

Pemrograman Kompetitif

Dr. Joseph Teguh Santoso, M. Kom

Pemrograman Kompetitif

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 978-623-5734-01-9 (PDF)

9 786235 734019

Pemrograman Kompetitif

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

PEMROGRAMAN KOMPETITIF

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

ISBN : 9 786235 734019

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 kepada Tuhan yang Maha Kuasa, bahwa pada akhirnya buku yang berjudul "Pemrograman Komprehensif" telah selesai ditulis. Tujuan utama dari buku ini adalah untuk memperkenalkan secara komprehensif tentang jenis pemrograman kompetitif yang modern. Diasumsikan bahwa para pembaca buku ini sudah mengetahui dasar-dasar pemrograman, sehingga desain algoritma atau kontes pemrograman sebelumnya tidak diperlukan. Isi dari buku ini mencakup berbagai topik dengan berbagai kesulitan, oleh sebab itu buku ini cocok untuk pemula dan pembaca yang telah berpengalaman. Kontes pemrograman sudah memiliki sejarah yang cukup panjang. Kontes Pemrograman Perguruan Tinggi Internasional untuk mahasiswa dimulai pada tahun 1970-an, dan Olimpiade Internasional pertama di bidang Informatika untuk siswa sekolah menengah diselenggarakan pada tahun 1989. Kedua kompetisi tersebut sekarang merupakan acara yang diadakan setiap tahun dan diikuti oleh sejumlah besar peserta yang masih kuliah ataupun yang sudah lulus di seluruh dunia.

Saat ini, pemrograman kompetitif menjadi lebih populer dari sebelumnya. Internet telah memainkan peran penting dalam kemajuannya. Sekarang ada komunitas online yang aktif dari pemrogram kompetitif, dan banyak kontes diselenggarakan setiap minggu. Pada saat yang sama, kesulitan kontes meningkat. Teknik yang hanya dikuasai oleh peserta terbaik beberapa tahun yang lalu, sekarang menjadi alat standar yang dikenal oleh banyak orang. Pemrograman kompetitif berakar pada studi ilmiah tentang algoritma. Namun, ketika seorang ilmuwan komputer menulis bukti untuk menunjukkan bahwa algoritma mereka berfungsi, pemrograman kompetitif mengimplementasikan algoritma mereka dan mengirimkannya ke sistem kontes. Kemudian, algoritma diuji dengan menggunakan satu set uji kasus, dan jika lulus semuanya, maka dapat diterima. Hal ini adalah elemen penting dalam pemrograman kompetitif, karena menyediakan cara yang secara otomatis dapat mendapatkan bukti yang kuat bahwa suatu algoritma dapat bekerja dengan baik. Faktanya, pemrograman kompetitif telah terbukti menjadi cara terbaik untuk mempelajari algoritma, karena ia mendorong untuk merancang algoritma yang benar-benar berfungsi, dan juga membuat sketsa kemungkinan suatu ide itu berhasil atau tidak.

Manfaat lain pemrograman kompetitif adalah bahwa sebuah kontes membutuhkan pemikiran, maka secara khusus, tidak ada spoiler dalam pernyataan masalah. Ini sebenarnya adalah masalah yang sering terjadi saat mengikuti kuliah algoritma. Kita diberi masalah yang untuk dipecahkan, namun pada kalimat terakhir ada petunjuk cara memodifikasi algoritma dalam menyelesaikan masalah. Tentu saja setelah membaca ini, mahasiswa tidak perlu lagi banyak berpikir, karena sudah tahu bagaimana menyelesaikan masalah tersebut. Hal ini tidak akan pernah terjadi dalam pemrograman kompetitif. Sebaliknya, kita memiliki seperangkat alat lengkap yang tersedia harus mencari tahu sendiri alat mana yang akan kita gunakan. Permasalahan memecahkan masalah dalam pemrograman kompetitif juga meningkatkan keterampilan pemrograman dan *debugging* seseorang. Biasanya, sebuah solusi diberikan nilai jika solusi tersebut dapat menyelesaikan semua masalah yang diuji dengan benar, sehingga pemrogram kompetitif yang sukses, harus mampu mengimplementasikan program yang tidak memiliki bug. Ini adalah keterampilan yang berharga dalam rekayasa perangkat lunak, dan bukan suatu kebetulan bahwa perusahaan IT tertarik pada orang-orang yang memiliki latar belakang dalam pemrograman kompetitif.

Untuk mencetak seorang programmer yang handal membutuhkan waktu yang lama, terlebih untuk menjadi programmer kompetitif, tetapi hal ini juga merupakan kesempatan untuk belajar banyak. Jika kita dapat yakin bahwa kitaa akan mendapatkan pemahaman umum yang lebih baik tentang algoritma, maka dengan membaca buku ini, kita akan merasa tertantang untuk memecahkan masalah algoritma, dan siap ikut serta dalam kontes.

Demikian, semoga buku ini dapat memberi manfaat yang besar bagi kemajuan sumber daya manusia Indonesia, menjadi manusia yang cerdas dan unggul dalam menyelesaikan masalah pemrograman di Indonesia.

Semarang, 15 Oktober 2021

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

DAFTAR ISI

BAB 1
PENDAHULUAN

Bab ini menunjukkan tentang apa itu pemrograman kompetitif, dan membahas sumber belajar tambahan. Bab ini juga membahas elemen-elemen pemrograman kompetitif, memperkenalkan pilihan kontes pemrograman populer, dan memberikan saran tentang cara mempraktekkan pemrograman kompetitif, membahas tujuan dan topik buku ini, dan secara singkat menjelaskan isi setiap bab. Lalu dibagian akhir bab menyajikan Kumpulan Masalah CSES, yang berisi kumpulan masalah praktik. Memecahkan masalah sambil membaca buku adalah cara yang baik untuk mempelajari pemrograman kompetitif dan buku-buku lain yang berkaitan dengan pemrograman kompetitif dan desain algoritma.

1.1 Apa itu Pemrograman Kompetitif?

Pemrograman kompetitif menggabungkan dua topik: desain algoritma dan implementasi algoritma.

1.1.1 Desain Algoritma

Inti dari pemrograman kompetitif adalah tentang menciptakan algoritma yang efisien yang memecahkan masalah komputasi yang terdefinisi dengan baik. Desain algoritma membutuhkan pemecahan masalah dan keterampilan matematika. Seringkali solusi untuk suatu masalah adalah kombinasi dari metode terkenal dan wawasan baru. Matematika berperan penting dalam pemrograman kompetitif. Sebenarnya, tidak ada batasan yang jelas antara desain algoritma dan matematika. Buku ini telah ditulis sehingga tidak banyak latar belakang matematika yang dibutuhkan. Lampiran buku mengulas beberapa konsep matematika yang digunakan di seluruh buku, seperti himpunan, logika, dan fungsi, dan lampiran dapat digunakan sebagai referensi saat membaca buku.

1.1.2 Implementasi Algoritma

Pemrograman tidak kompetitif, solusi untuk masalah dievaluasi dengan menguji algoritma yang diimplementasikan menggunakan satu set kasus uji. Jadi, setelah menemukan algoritma yang memecahkan masalah, langkah selanjutnya adalah mengimplementasikannya dengan benar, yang membutuhkan keterampilan pemrograman yang baik. Pemrograman kompetitif sangat berbeda dari rekayasa perangkat lunak tradisional: program pendek (biasanya paling banyak beberapa ratus baris), program harus ditulis dengan cepat, dan tidak perlu mempertahankannya setelah kontes. Saat ini, bahasa pemrograman yang paling populer digunakan dalam kontes adalah C++, Python, dan Java. Misalnya, di Google Code Jam 2017, di antara 3.000 peserta terbaik, 79% menggunakan C++, 16% menggunakan Python, dan 8% menggunakan Java.

Banyak orang menganggap C++ sebagai pilihan terbaik untuk programmer yang kompetitif. Manfaat menggunakan C++ adalah bahasa yang sangat efisien dan pustaka standarnya berisi kumpulan besar struktur data dan algoritma. Semua contoh program dalam buku ini ditulis dalam C++, dan struktur data dan algoritma perpustakaan standar sering digunakan. Program mengikuti standar C++11, yang dapat digunakan di sebagian besar kontes saat ini. Jika Anda belum dapat memprogram dalam C++, sekarang adalah saat yang tepat untuk mulai belajar.

1.2 Kontes Pemrograman

International Olympiad in Informatics (IOI)

IOI *(International Olympiad in Informatics)* adalah kontes pemrograman tahunan untuk siswa sekolah menengah. Setiap negara diperbolehkan mengirimkan tim yang terdiri dari empat siswa untuk mengikuti kontes. Biasanya ada sekitar 300 peserta dari 80 negara. IOI terdiri dari dua kontes berdurasi lima jam. Dalam kedua kontes tersebut, para peserta diminta untuk menyelesaikan tiga tugas pemrograman yang sulit. Tugas dibagi menjadi subtugas, yang masing-masing memiliki inti yang ditugaskan. Sementara para kontestan dibagi menjadi beberapa tim, mereka bersaing sebagai individu. Peserta IOI dipilih melalui kontes nasional.

Sebelum IOI, banyak kompetisi regional yang diselenggarakan, seperti *Baltic Olympiad in Informatics* (BOI), Central European Olympiad in Informatics (CEOI), dan *Asia-Pacific Informatics Olympiad* (APIO). ICPC International Collegiate Programming Contest adalah kontes pemrograman tahunan untuk mahasiswa. Setiap tim dalam kontes terdiri dari tiga siswa, dan tidak seperti di IOI, siswa bekerja sama; hanya ada satu komputer yang tersedia untuk setiap tim. ICPC terdiri dari beberapa tahap, dan akhirnya tim terbaik diundang ke Final Dunia. Meskipun ada puluhan ribu peserta dalam kontes ini, hanya ada sedikit slot akhir[1] yang tersedia, jadi bahkan maju ke final adalah pencapaian yang luar biasa.

Dalam setiap kontes ICPC, tim memiliki waktu lima jam untuk memecahkan sepuluh masalah algoritma. Selama kontes, pesaing dapat melihat hasil tim lain, tetapi selama satu jam terakhir papan skor dibekukan dan tidak mungkin untuk melihat hasil pengiriman terakhir. Kontes Online Ada juga banyak kontes online yang terbuka untuk semua orang. Saat ini, situs kontes paling aktif adalah Codeforces, yang menyelenggarakan kontes mingguan. Situs kontes populer lainnya termasuk AtCoder, CodeChef, CSAcademy, HackerRank, dan Topcoder. Beberapa perusahaan menyelenggarakan kontes online dengan final di tempat. Contoh kontes tersebut adalah Facebook Hacker Cup, Google Code Jam, dan Yandex.Algorithm. Tentu saja, perusahaan juga menggunakan kontes tersebut untuk perekrutan: tampil baik dalam kontes adalah cara yang baik untuk membuktikan keterampilan seseorang dalam pemrograman.

1.2 Tips untuk Berlatih

Mempelajari pemrograman kompetitif membutuhkan banyak pekerjaan. Namun, ada banyak cara untuk berlatih, dan beberapa di antaranya lebih baik daripada yang lain. Ketika memecahkan masalah, kita harus ingat bahwa jumlah masalah yang diselesaikan tidak begitu penting daripada kualitas masalah. Sangat menggoda untuk memilih masalah yang terlihat bagus dan mudah dan menyelesaikannya, dan melewatkan masalah yang terlihat sulit dan membosankan. Namun, cara untuk benar-benar meningkatkan keterampilan seseorang adalah dengan fokus pada jenis masalah yang terakhir.

Pengamatan penting lainnya adalah bahwa sebagian besar masalah kontes pemrograman dapat diselesaikan dengan menggunakan algoritme sederhana dan pendek, tetapi bagian yang sulit adalah menemukan algoritme. Pemrograman kompetitif bukan tentang mempelajari algoritme yang kompleks dan tidak jelas, melainkan tentang mempelajari pemecahan masalah dan cara mendekati masalah yang sulit menggunakan alat sederhana.

[1] Jumlah pasti slot akhir bervariasi dari tahun ke tahun; pada 2020, ada 183 slot akhir.

Terakhir, beberapa orang membenci implementasi algoritma: mendesain algoritma itu menyenangkan tetapi membosankan untuk mengimplementasikannya. Namun, kemampuan untuk mengimplementasikan algoritma dengan cepat dan benar merupakan aset penting, dan keterampilan ini dapat dipraktikkan. Adalah ide yang buruk untuk menghabiskan sebagian besar waktu kontes untuk menulis kode dan menemukan bug, daripada memikirkan bagaimana memecahkan masalah.

1.3 Tentang Buku Ini

Silabus IOI mengatur topik-topik yang mungkin muncul di Olimpiade Internasional Informatika, dan silabus telah menjadi titik awal dalam memilih topik untuk buku ini. Namun, buku ini juga membahas beberapa topik lanjutan yang (per 2017) dikecualikan dari IOI tetapi mungkin muncul di kontes lain. Contoh topik tersebut adalah aliran maksimum, teori nim, dan susunan akhiran.

Sementara banyak topik pemrograman kompetitif dibahas dalam buku teks algoritme standar, ada juga perbedaannya. Misalnya, banyak buku teks fokus pada penerapan algoritma pengurutan dan struktur data dasar dari awal, tetapi pengetahuan ini tidak terlalu relevan dalam pemrograman kompetitif, karena fungsionalitas perpustakaan standar dapat digunakan. Lalu, ada topik yang terkenal di komunitas pemrograman kompetitif tetapi jarang dibahas di buku teks. Contoh dari topik tersebut adalah struktur data pohon segmen yang dapat digunakan untuk memecahkan sejumlah besar masalah yang membutuhkan algoritma rumit. Salah satu tujuan buku ini adalah untuk mendokumentasikan teknik pemrograman kompetitif yang biasanya hanya dibahas di forum online dan posting blog. Bila memungkinkan, referensi ilmiah telah diberikan untuk metode yang khusus untuk pemrograman kompetitif. Namun, hal ini sering tidak mungkin, karena banyak teknik sekarang menjadi bagian dari cerita rakyat pemrograman kompetitif dan tidak ada yang tahu siapa yang awalnya menemukannya. Struktur bukunya adalah sebagai berikut:

- Bab 2 mengulas fitur bahasa pemrograman C++, dan kemudian membahas algoritma rekursif dan manipulasi bit.
- Bab 3 berfokus pada efisiensi: cara membuat algoritme yang dapat memproses kumpulan data besar dengan cepat.
- Bab 4 membahas algoritma pengurutan dan pencarian biner, dengan fokus pada aplikasinya dalam desain algoritma.
- Bab 5 membahas pemilihan struktur data dari pustaka standar C++, seperti vektor, set, dan Maps.
- Bab 6 memperkenalkan teknik desain algoritma yang disebut pemrograman dinamis, dan menyajikan contoh masalah yang dapat diselesaikan dengan menggunakannya.
- Bab 7 membahas algoritma grafis dasar, seperti mencari jalur terpendek dan pohon merentang minimum.
- Bab 8 membahas beberapa topik desain algoritma tingkat lanjut, seperti bitparalelisme dan analisis diamortisasi.
- Bab 9 berfokus pada pemrosesan kueri rentang array secara efisien, seperti menghitung jumlah nilai dan menentukan nilai minimum.
- Bab 10 menyajikan algoritma khusus untuk pohon, termasuk metode untuk memproses kueri pohon.
- Bab 11 membahas topik matematika yang relevan dalam pemrograman kompetitif.
- Bab 12 menyajikan teknik grafik tingkat lanjut, seperti komponen terhubung kuat dan aliran maksimum.

- Bab 13 berfokus pada algoritma geometris dan menyajikan teknik yang menggunakan masalah geometris yang dapat diselesaikan dengan mudah.
- Bab 14 membahas teknik string, seperti hashing string, algoritma-Z, dan menggunakan array sufiks.
- Bab 15 membahas pilihan topik yang lebih maju, seperti algoritma akar kuadrat dan optimisasi pemrograman dinamis.

1.4 Kumpulan Masalah CSES

Kumpulan Masalah CSES menyediakan kumpulan masalah yang dapat digunakan untuk mempraktikkan pemrograman kompetitif. Masalah-masalah telah disusun dalam urutan kesulitan, dan semua teknik yang diperlukan untuk memecahkan masalah dibahas dalam buku ini. Kumpulan masalah tersedia di alamat berikut:
https://cses.fi/problemset/

Mari kita lihat bagaimana memecahkan masalah pertama dalam kumpulan masalah, yang disebut Algoritma Aneh. Rumusan masalahnya adalah sebagai berikut:
Pertimbangkan sebuah algoritma yang mengambil sebagai input bilangan bulat positif n. Jika n genap, algoritma membaginya dengan dua, dan jika n ganjil, algoritma mengalikannya dengan tiga dan menambahkan satu. Algoritma mengulangi ini, sampai n adalah satu. Sebagai contoh, barisan untuk n =3 adalah sebagai berikut:

3→10→5→16→8→4→2→1

Tugas Anda adalah mensimulasikan eksekusi algoritma untuk nilai n yang diberikan.

Input

Satu-satunya baris input berisi bilangan bulat n.

Output

Cetak baris yang berisi semua nilai n selama algoritma.

Kendala

$1 \leq n\ 10^6$

Contoh

Input:

3

Output :

3 10 5 16 8 4 2 1

Masalah ini terkait dengan dugaan Collatz yang terkenal yang menyatakan bahwa algoritma di atas berakhir untuk setiap nilai n. Namun, tidak ada yang bisa membuktikannya. Namun, dalam masalah ini, kita tahu bahwa nilai awal n akan mengalahkan hampir satu juta, yang membuat masalah lebih mudah dipecahkan. Soal ini merupakan soal simulasi sederhana yang tidak memerlukan banyak pemikiran. Berikut adalah cara yang mungkin untuk memecahkan masalah di C++:

```
#include <iostream>
using namespace std;
int main() {
int n;
cin >> n;
while (true){
cout << n << " ";
if (n == 1) break;
if (n%2 == 0) n /= 2;
```

```
else n = n*3+1;
}
cout << "\n";
}
```

Kode pertama membaca dalam nomor input n, dan kemudian mensimulasikan algoritma dan mencetak nilai n setelah setiap langkah. Sangat mudah untuk menguji bahwa algoritma dengan benar menangani contoh kasus n =3 yang diberikan dalam pernyataan masalah. Sekarang saatnya mengirimkan kode ke CSES. Kemudian kode akan dikompilasi dan diuji menggunakan satu set kasus uji. Untuk setiap test case, CSES akan memberitahu kita apakah kode kita lolos atau tidak, dan kita juga bisa memeriksa input, output yang diharapkan, dan output yang dihasilkan oleh kode kita. Setelah menguji kode kami, CSES memberikan laporan berikut:

Tes	Keterangan	Waktu (detik)
#1	ACCEPTED	0.06 / 1.00
#2	ACCEPTED	0.06 / 1.00
#3	ACCEPTED	0.07 / 1.00
#4	ACCEPTED	0.06 / 1.00
#5	ACCEPTED	0.06 / 1.00
#6	ACCEPTED	---- / 1.00
#7	ACCEPTED	---- / 1.00
#8	ACCEPTED	0.007 / 1.00
#9	ACCEPTED	---- / 1.00
#10	ACCEPTED	0.06 / 1.00

Ini berarti bahwa kode kami melewati beberapa kasus uji (DITERIMA), terkadang terlalu lambat (BATAS WAKTU DILEWATKAN), dan juga menghasilkan Output yang salah (JAWABAN SALAH). Ini cukup mengejutkan! Kasus uji pertama yang gagal memiliki n = 138367. Jika kita menguji kode kita secara lokal menggunakan input ini, ternyata kode tersebut memang lambat. Bahkan, itu tidak pernah berakhir. Oleh karena itu kode Anda gagal sehingga dapat menjadi cukup besar selama simulasi. Secara khusus, itu bisa menjadi lebih besar dari batas atas variabel int. Untuk memperbaiki masalah, cukup dengan mengubah kode kita sehingga tipe n panjang. Maka kita akan mendapatkan hasil yang diinginkan:

Tes	Keterangan	Waktu (detik)
#1	ACCEPTED	0.05 / 1.00
#2	ACCEPTED	0.06 / 1.00
#3	ACCEPTED	0.07 / 1.00
#4	ACCEPTED	0.06 / 1.00
#5	ACCEPTED	0.06 / 1.00
#6	ACCEPTED	0.05 / 1.00
#7	ACCEPTED	0.06 / 1.00
#8	ACCEPTED	0.05 / 1.00
#9	ACCEPTED	0.07 / 1.00
#10	ACCEPTED	0.06 / 1.00

Seperti yang ditunjukkan contoh ini, bahkan algoritme yang sangat sederhana mungkin

mengandung bug yang tidak kentara. Pemrograman kompetitif mengajarkan cara menulis algoritme yang benar-benar berfungsi.

1.5 Sumber Daya Lainnya

Selain buku ini, sudah ada beberapa buku lain tentang pemrograman kompetitif. Tantangan Pemrograman Skiena dan Revilla adalah buku perintis di bidang yang diterbitkan pada tahun 2003. Buku yang lebih baru adalah Pemrograman Kompetitif 3 oleh Halim dan Halim. Kedua buku di atas ditujukan untuk pembaca yang tidak memiliki latar belakang pemrograman kompetitif.

Mencari Tantangan? adalah buku lanjutan, yang menyajikan kumpulan masalah sulit dari kontes pemrograman Polandia. Fitur yang paling menarik dari buku ini adalah menyediakan analisis rinci tentang bagaimana memecahkan masalah. Buku ini ditujukan untuk programmer kompetitif yang berpengalaman.

Tentu saja, buku algoritme umum juga merupakan bacaan yang bagus untuk programmer yang kompetitif. Yang paling komprehensif adalah Pengantar Algoritma yang ditulis oleh Cormen, Leiserson, Rivest, dan Stein, juga disebut CLRS. Buku ini adalah sumber yang bagus jika Anda ingin memeriksa semua detail tentang algoritma dan bagaimana membuktikan dengan sungguh-sungguh bahwa itu benar. *Desain Algoritma* Kleinberg dan Tardos berfokus pada teknik desain algoritma, dan secara menyeluruh membahas metode membagi dan menaklukkan, algoritma serakah, pemrograman dinamis, dan algoritma aliran maksimum. *Algorithm Design Manual* dari Skiena adalah buku yang lebih praktis yang mencakup katalog besar masalah komputasi dan menjelaskan cara menyelesaikannya.

BAB 2
TEKNIK PEMROGRAMAN

Bab ini menyajikan beberapa fitur bahasa pemrograman C++ yang berguna dalam pemrograman kompetitif, dan memberikan contoh bagaimana menggunakan rekursi dan operasi bit dalam pemrograman. Bagian selanjutnya membahas pilihan topik yang terkait dengan C++, termasuk metode input dan output, bekerja dengan angka, dan cara mempersingkat kode, dan dilanjutkan pada algoritma rekursif. Pertama kita akan mempelajari cara yang elegan untuk menghasilkan semua himpunan bagian dan permutasi dari suatu himpunan menggunakan rekursi. Setelah ini, kita akan menggunakan backtracking untuk menghitung jumlah cara untuk menempatkan n ratu yang tidak menyerang pada papan catur n×n. Dasar-dasar operasi bit dan juga bagaimana menggunakannya untuk mewakili himpunan bagian dari himpunan akan dibahasa dalam bab ini.

2.1 Fitur Bahasa

Templat kode C++ khas untuk pemrograman kompetitif terlihat seperti ini:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    // solution comes here
}
```

Baris # include di awal kode adalah fitur kompiler g++ yang memungkinkan kita menyertakan seluruh pustaka standar. Dengan demikian, tidak perlu menyertakan pustaka secara terpisah seperti iostream, vektor, dan algoritme, melainkan tersedia secara otomatis. Baris using menyatakan bahwa kelas dan fungsi perpustakaan standar dapat digunakan secara langsung dalam kode. Tanpa garis using kita harus menulis, misalnya, std::cout, tetapi sekarang cukup menulis cout. Kode dapat dikompilasi menggunakan perintah berikut:

```
g++ -std=c++11 -O2 -Wall test.cpp -o test
```

Perintah ini menghasilkan pengujian file biner dari pengujian kode sumber .cpp. Kompiler mengikuti standar C++11 (-std=c++11), mengoptimalkan kode (-O2), dan menunjukkan peringatan tentang kemungkinan kesalahan (-Wall).

2.1.1 Input dan Output

Di sebagian besar kontes, aliran standar digunakan untuk membaca input dan menulis output. Dalam C++, aliran standar adalah cin untuk input dan cout untuk output. Juga fungsi C, seperti scanf dan printf, dapat digunakan. Input untuk program biasanya terdiri dari angka dan string yang dipisahkan dengan spasi dan baris baru. Mereka dapat dibaca dari aliran cin sebagai berikut:

```
int a, b;
string x;
cin >> a >> b >> x;
```

Kode semacam ini selalu berfungsi, dengan asumsi setidaknya ada satu spasi atau baris baru di antara setiap elemen dalam input. Misalnya, kode di atas dapat membaca kedua input berikut:

```
123 456 monkey
123 456 monkey
```

Aliran count digunakan untuk output sebagai berikut:

```
int a = 123, b = 456;
string x = "monkey";
cout << a << " " << b << " " << x << "\n";
```

Input dan output terkadang menjadi hambatan dalam program. Baris berikut di awal kode membuat input dan output lebih efisien:

```
ios::sync_with_stdio(0);
cin.tie(0);
```

Perhatikan bahwa baris baru "\n" bekerja lebih cepat daripada endl, karena endl selalu menyebabkan operasi flush. Fungsi C scanf dan printf adalah alternatif dari aliran standar C++. Mereka biasanya sedikit lebih cepat, tetapi juga lebih sulit untuk digunakan. Kode berikut membaca dua bilangan bulat dari input:

```
int a, b;
scanf("%d %d", &a, &b);
```

Kode berikut mencetak dua bilangan bulat:

```
int a = 123, b = 456;
printf("%d %d\n", a, b);
```

Terkadang program harus membaca seluruh baris input, mungkin berisi spasi. Ini dapat dicapai dengan menggunakan fungsi getline:

```
string s;
getline(cin, s);
```

Jika jumlah data tidak diketahui, loop berikut berguna:

```
while (cin >> x) {
// code
}
```

Loop ini membaca elemen dari input satu demi satu, sampai tidak ada lagi data yang tersedia di input. Dalam beberapa sistem kontes, file digunakan untuk input dan output. Solusi mudah untuk ini adalah menulis kode seperti biasa menggunakan aliran standar, tetapi tambahkan baris berikut ke awal kode:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

Setelah ini, program membaca input dari file "input.txt" dan menulis output ke file "*output.txt*".

2.1.2 Bekerja denganNomor

Integer

Tipe integer yang paling banyak digunakan dalam pemrograman kompetitif adalah int, yang merupakan tipe[2] 32-bit dengan rentang nilai -2^{31} ...2^{31} - 1 (sekitar$-2 \cdot x\ 10^9$ ...$2 \cdot x\ 10^9$). Jika tipe int tidak cukup, tipe long long 64-bit dapat digunakan. Ini memiliki rentang nilai -2^{63} ...2^{63} - 1 (sekitar $-9\ x \cdot 10^{18}$ ...$9\ x \cdot 10^{18}$). Kode berikut mendefinisikan variabel panjang yang panjang:

```
long long x = 123456789123456789LL;
```

Akhiran LL berarti jenis bilangan itu panjang. Kesalahan umum saat menggunakan tipe long long adalah tipe int masih digunakan di suatu tempat dalam kode. Misalnya, kode berikut berisi kesalahan halus:

```
int a = 123456789;
```

[2] Faktanya, standar C++ tidak secara tepat menentukan ukuran tipe angka, dan batasannya bergantung pada kompiler dan platform. Ukuran yang diberikan di bagian ini adalah ukuran yang kemungkinan besar akan Anda lihat saat menggunakan sistem modern

```
long long b = a*a;
cout << b << "\n"; // -1757895751
```

Meskipun variabel b bertipe long long, kedua bilangan dalam ekspresi a*a bertipe int, dan hasilnya juga bertipe int. Karena itu, variabel b akan memiliki hasil yang salah. Soal tersebut dapat diselesaikan dengan mengubah tipe a menjadi long long atau dengan mengubah ekspresi menjadi (long long) a*a. Biasanya masalah kontes diatur sedemikian rupa sehingga tipe long long sudah cukup. Namun, baik untuk mengetahui bahwa kompiler g++ juga menyediakan tipe 128-bit __int128_t dengan rentang nilai -2^{127} ...$2^{127}-1$ (about-10^{38} ...10^{38}). Namun, jenis ini tidak tersedia di semua sistem kontes.

Aritmatika Modular

Kadang-kadang, jawaban suatu masalah adalah bilangan yang sangat besar, tetapi cukup untuk mengeluarkannya "modulo m", yaitu, sisanya ketika jawabannya dibagi dengan m (misalnya, "modulo 10^9 +7"). Idenya adalah bahwa meskipun jawaban sebenarnya sangat besar, cukup menggunakan tipe int dan long long. Kami menyatakan dengan x mod m sisa ketika x dibagi dengan m. Misalnya, 17 mod 5=2, karena 17=3·5+2. Properti penting dari sisa adalah bahwa rumus berikut berlaku:

(a+b) mod m = (a mod m+b mod m) mod m
(a−b) mod m = (a mod m−b mod m) mod m
(a·b) mod m = (a mod m·b mod m) mod m

Dengan demikian, kita dapat mengambil sisanya setelah setiap operasi, dan jumlahnya tidak akan pernah menjadi terlalu besar.

Misalnya, kode berikut menghitung n!, faktorial dari n, modulo m:

```
long long x = 1;
for (int i = 1; i <= n; i++) {
        x = (x*i)%m;
}
cout << x << "\n";
```

Biasanya kita ingin sisanya selalu berada di antara 0...m−1. Namun, dalam C++ dan bahasa lain, sisa angka negatif adalah nol atau negatif. Cara mudah untuk memastikan tidak ada sisa negatif adalah pertama-tama menghitung sisa seperti biasa dan kemudian menambahkan m jika hasilnya negatif:

```
x = x%m;
if (x < 0) x += m;
```

Namun, ini hanya diperlukan ketika ada pengurangan dalam kode, dan sisanya mungkin menjadi negatif. Bilangan Floating Point Dalam kebanyakan masalah pemrograman kompetitif, cukup menggunakan bilangan bulat, tetapi terkadang bilangan floating point diperlukan. Tipe titik mengambang yang paling berguna dalam C++ adalah ganda 64-bit dan, sebagai ekstensi dalam kompiler g++, ganda panjang 80-bit. Dalam kebanyakan kasus, dobel sudah cukup, tetapi dobel panjang lebih akurat. Ketepatan jawaban yang dibutuhkan biasanya diberikan dalam pernyataan masalah. Cara mudah untuk menampilkan jawabannya adalah dengan menggunakan fungsi print f dan memberikan jumlah tempat desimal dalam string pemformatan. Misalnya, kode berikut mencetak nilai x dengan 9 tempat desimal:

```
printf("%.9f\n", x);
```

Kesulitan dalam menggunakan bilangan floating point adalah beberapa bilangan tidak dapat direpresentasikan secara akurat sebagai bilangan floating point, dan akan terjadi

kesalahan pembulatan. Misalnya, dalam kode berikut, nilai x sedikit lebih kecil dari 1, sedangkan nilai yang benar adalah 1.

```
double x = 0.3*3+0.1;
printf("%.20f\n", x); // 0.99999999999999988898
```

Membandingkan angka titik mengambang dengan operator = = sangat berisiko, karena ada kemungkinan bahwa nilainya harus sama tetapi bukan karena kesalahan presisi. Cara yang lebih baik untuk membandingkan angka titik mengambang adalah dengan mengasumsikan bahwa dua angka adalah sama jika perbedaan di antara keduanya kurang dari ε, di mana adalah angka kecil. Misalnya, dalam kode berikut ε =10^{-9}:

```
if (abs(a-b) < 1e-9)
{
// a and b are equal
}
```

Perhatikan bahwa meskipun angka titik mengambang tidak akurat, bilangan bulat hingga batas tertentu masih dapat direpresentasikan secara akurat. Misalnya, dengan menggunakan ganda, dimungkinkan untuk secara akurat mewakili semua bilangan bulat yang nilai absolutnya paling banyak 2^{53}.

2.1.3 Kode Singkatan

Type Names

Perintah *typedef* dapat digunakan untuk memberi nama pendek pada tipe data. Misalnya, nama panjang panjang adalah panjang, sehingga kita dapat mendefinisikan nama pendek ll sebagai berikut:

```
typedef long long ll;
```

Setelah ini, kode

```
long long a = 123456789;
long long b = 987654321;
cout << a*b << "\n";
```

dapat dipersingkat sebagai berikut:

```
ll a = 123456789;
ll b = 987654321;
cout << a*b << "\n";
```

Perintah typedef juga dapat digunakan dengan tipe yang lebih kompleks. Sebagai contoh, kode berikut memberikan nama vi untuk vektor bilangan bulat, dan nama pi untuk pasangan yang berisi dua bilangan bulat.

```
typedef vector<int> vi;
typedef pair<int,int> pi;
```

Makro Cara lain untuk mempersingkat kode adalah dengan mendefinisikan makro. Makro menentukan bahwa string tertentu dalam kode akan diubah sebelum kompilasi. Dalam C++, makro didefinisikan menggunakan kata kunci #define. Misalnya, kita dapat mendefinisikan makro berikut:

```
#define F first
#define S second
#define PB push_back
#define MP make_pair
```

Setelah ini, kode

```
v.push_back(make_pair(y1,x1));
v.push_back(make_pair(y2,x2));
int d = v[i].first+v[i].second;
```

dapat dipersingkat sebagai berikut:

```
v.PB(MP(y1,x1));
v.PB(MP(y2,x2));
int d = v[i].F+v[i].S;
```

Makro juga dapat memiliki parameter, yang memungkinkan untuk mempersingkat loop dan struktur lainnya. Misalnya, kita dapat mendefinisikan makro berikut:

```
#define REP(i,a,b) for (int i = a; i <= b; i++
```

Setelah ini, kode

```
for (int i = 1; i <= n; i++) {
search(i);
}
```

dapat dipersingkat sebagai berikut:

```
REP(i,1,n) {
search(i);
}
```

2.2 Algoritma Rekursif

Rekursi sering memberikan cara yang elegan untuk mengimplementasikan suatu algoritma. Pada bagian ini, kita membahas algoritma rekursif yang secara sistematis melalui kandidat solusi untuk suatu masalah. Pertama, kita fokus pada pembangkitan himpunan bagian dan permutasi dan kemudian membahas teknik backtracking yang lebih umum.

2.2.1 Menghasilkan Subset

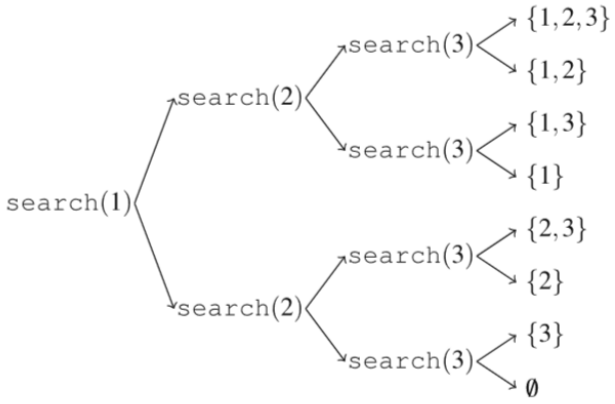
Aplikasi rekursi pertama kita adalah membangkitkan semua himpunan bagian dari himpunan n elemen. Misalnya, himpunan bagian dari {1,2,3} adalah , {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, dan {1, 2,3}. Pencarian fungsi rekursif berikut dapat digunakan untuk menghasilkan himpunan bagian. Fungsi mempertahankan vektor

```
vector<int> subset;
```

yang akan berisi elemen dari setiap subset. Pencarian dimulai ketika fungsi dipanggil dengan parameter 1.

```
void search(int k) {
        if (k == n+1) {
                // process subset
        } else {
// include k in the subset
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
// don't include k in the subset
        search(k+1);
        }
        }
```

Ketika pencarian fungsi dipanggil dengan parameter k, ia memutuskan apakah akan memasukkan elemen k dalam subset atau tidak, dan dalam kedua kasus, kemudian memanggil dirinya sendiri dengan parameterk+1. Kemudian, if*k* =n+1, fungsi memperhatikan bahwa semua elemen telah diproses dan subset telah dihasilkan. Gambar 2.1 mengilustrasikan pembangkitan himpunan bagian ketika n =3. Panggilan fungsi pengajaran, baik cabang atas (k termasuk dalam subset) atau cabang bawah (k tidak termasuk dalam subset) dipilih.



Gambar 2.1 Pohon rekursi saat menghasilkan himpunan bagian dari himpunan{1,2,3}

2.2.2 Menghasilkan Permutasi

Selanjutnya kita mempertimbangkan masalah pembangkitan semua permutasi dari himpunan n elemen. Misalnya, permutasi dari{1,2,3}adalah (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1 ,2), dan (3,2,1). Sekali lagi, kita dapat menggunakan rekursi untuk melakukan pencarian. Pencarian fungsi berikut mempertahankan vektor

vector<**int**> permutation;

yang akan berisi setiap permutasi, dan sebuah array

bool chosen[n+1];

yang menunjukkan untuk setiap elemen jika telah dimasukkan dalam permutasi. Pencarian dimulai ketika fungsi dipanggil tanpa parameter.

```
void search() {
if (permutation.size() == n) {
        // process permutation}
else {
        for (int i = 1; i <= n; i++) {
                if (chosen[i]) continue;
                chosen[i] = true;
                permutation.push_back(i);
                search();
                chosen[i] = false;
                permutation.pop_back();
        }
}
}
```

Setiap pemanggilan fungsi menambahkan elemen baru ke permutasi dan mencatat bahwa elemen tersebut telah disertakan dalam pilihan. Jika ukuran permutasi sama dengan ukuran himpunan, permutasi telah dihasilkan. Perhatikan bahwa pustaka standar C++ juga memiliki fungsi next_permutation yang dapat digunakan untuk menghasilkan permutasi. Fungsi tersebut diberikan permutasi, dan menghasilkan permutasi berikutnya dalam urutan leksikografis. Kode berikut melewati permutasi dari{1,2,...,n}:

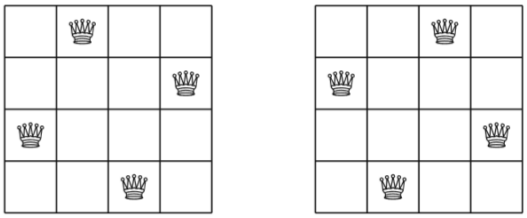
```
for (int i = 1; i <= n; i++ {
permutation.push_back(i);
}
do {
```

```
// process permutation
} while (next_permutation(permutation.begin(),
permutation.end()));
```

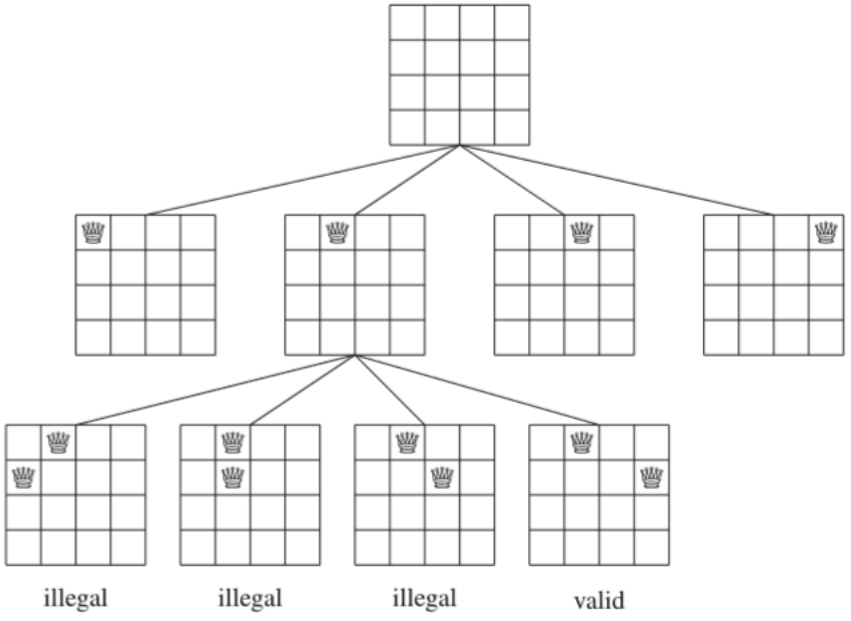
2.2.3 Bracktracking

Algoritma backtracking dimulai dengan solusi kosong dan memperluas solusi langkah demi langkah. Pencarian secara rekursif melewati semua cara yang berbeda bagaimana solusi dapat dibangun. Sebagai contoh, perhatikan masalah menghitung jumlah ratu yang dapat ditempatkan pada papan catur nxn sehingga tidak ada dua ratu yang saling menyerang. Misalnya, Gambar 2.2 menunjukkan dua solusi yang mungkin untuk n =4.

Masalah dapat diselesaikan dengan menggunakan backtracking dengan menempatkan ratu di papan baris demi baris. Lebih tepatnya, tepat satu ratu akan ditempatkan di setiap baris sehingga tidak ada ratu yang menyerang salah satu ratu yang ditempatkan sebelumnya. Sebuah solusi telah ditemukan ketika semua n ratu telah ditempatkan di papan tulis. Misalnya, Gambar 2.3 menunjukkan beberapa solusi parsial yang dihasilkan oleh algoritma pelacakan mundur ketika n =4. Di tingkat bawah, tiga konfigurasi pertama adalah ilegal, karena ratu saling menyerang. Namun, konfigurasi keempat valid, dan dapat diperluas ke solusi lengkap dengan menempatkan dua ratu lagi di papan. Hanya ada satu cara untuk menempatkan dua ratu yang tersisa.



Gambar 2.2 Kemungkinan cara untuk menempatkan 4 ratu di papan catur 4×4



Gambar 2.3 Solusi parsial untuk masalah ratu menggunakan backtracking

Algoritma tersebut dapat diimplementasikan sebagai berikut:

```
void search(int y) {
if (y == n) {
        count++;
        return;
}
for (int x = 0; x < n; x++) {
        if (col[x] || diag1[x+y] || diag2[x-y+n-1] continue;
        col[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
    search(y+1);
    col[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
    }
}
```

Pencarian dimulai dengan memanggil search(0). Ukuran papan adalah n, dan kode menghitung jumlah solusi yang akan dihitung. Kode mengasumsikan bahwa baris dan kolom papan diberi nomor dari 0 hingga n-1. Ketika pencarian dipanggil dengan parameter y, ia menempatkan ratu pada baris y dan kemudian memanggil dirinya sendiri dengan parameter y+1. Kemudian, jika y =n, solusi telah ditemukan, dan nilai count bertambah satu. Array col melacak kolom yang berisi ratu, dan array diag1 dan diag2 melacak diagonal. Tidak diperbolehkan menambahkan ratu lain ke kolom atau diagonal yang sudah berisi ratu.

Misalnya, Gambar 2.4 menunjukkan penomoran kolom dan diagonal papan 4x4. Algoritma pelacakan mundur di atas memberi tahu bahwa ada 92 cara untuk menempatkan 8 ratu di papan 8×8. Ketika n meningkat, pencarian dengan cepat menjadi lambat, karena jumlah solusi tumbuh secara eksponensial. Misalnya, dibutuhkan sekitar satu menit pada komputer modern untuk menghitung bahwa ada 14772512 cara untuk menempatkan 16 ratu di papan 16x16. Faktanya, tidak ada yang tahu cara yang efisien untuk menghitung jumlah kombinasi ratu untuk nilai n yang lebih besar. Saat ini, nilai n terbesar yang hasilnya diketahui adalah 27: ada 234907967154122528 kombinasi dalam kasus ini. Ini ditemukan pada tahun 2016 oleh sekelompok peneliti yang menggunakan sekelompok komputer untuk menghitung hasilnya.

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

col

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

diag1

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

diag2

Gambar 2.4 Penomoran array saat menghitung kombinasi pada papan 4x4

2.3 Manipulasi Bit

Dalam pemrograman, bilangan bulat n-bit disimpan secara internal sebagai bilangan biner yang terdiri dari n bit. Misalnya, tipe C++ int adalah tipe 32-bit, yang berarti bahwa setiap nomor int terdiri dari 32 bit. Misalnya, representasi bit dari int nomor 43 adalah

00000000000000000000000000101011.

Bit dalam representasi diindeks dari kanan ke kiri. Untuk mengubah representasi bit $b_k \ldots b_2 b_1 b_0$ menjadi angka, rumusnya

$$b_k 2^k + \cdots + b_2 2^2 + b_1 2^1 + b_0 2^0.$$

dapat digunakan. Sebagai contoh,

$$1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 43.$$

Representasi bit dari suatu nomor dapat ditandatangani atau tidak. Biasanya representasi bertanda digunakan, yang berarti bahwa angka negatif dan positif dapat diwakili. Sebuah variabel bertanda n bit dapat berisi bilangan bulat antara 2^{n-1} dan $2^{n-1} - 1$. Misalnya, tipe int di C++ adalah tipe yang ditandatangani, jadi variabel int dapat berisi bilangan bulat apa pun antara -2^{31} dan $2^{31} - 1$.

Bit pertama dalam representasi bertanda adalah tanda bilangan (0 untuk bilangan non negatif dan 1 untuk bilangan negatif), dan n 1 bit sisanya berisi besaran bilangan. Komplemen dua digunakan, yang berarti bahwa bilangan yang berlawanan dari suatu bilangan dihitung dengan terlebih dahulu membalik semua bit dalam bilangan tersebut dan kemudian menambah bilangan tersebut satu per satu. Misalnya, representasi bit dari nomor int−43 adalah

11111111111111111111111111010101.

Dalam representasi tidak bertanda, hanya angka non-negatif yang dapat digunakan, tetapi batas atas untuk nilainya lebih besar. Sebuah variabel unsigned n bit dapat berisi bilangan bulat antara 0 dan $2^n - 1$. Misalnya, dalam C++, variabel int yang tidak ditandatangani dapat berisi bilangan bulat apa pun antara 0 dan $2^{32} - 1$. Ada hubungan antara representasi: bilangan bertanda x sama dengan bilangan tak bertanda $2^n - x$. Misalnya, kode berikut menunjukkan bahwa bilangan bertanda $x = -43$ sama dengan bilangan tak bertanda $y = 2^{32} - 43$:

```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

Jika angka lebih besar dari batas atas representasi bit, angka tersebut akan meluap. Dalam representasi bertanda, angka berikutnya setelah $2^{n-1} - 1$ adalah -2^{n-1}, dan dalam representasi tidak bertanda, angka berikutnya setelah $2^n - 1$ adalah 0. Sebagai contoh, perhatikan kode berikut:

```
int x = 2147483647 cout << x << "\n";
// 2147483647
x++; cout << x << "\n";
// -2147483648
```

awalnya, nilai x adalah $2^{31} - 1$. Ini adalah nilai terbesar yang dapat disimpan dalam variabel int, jadi angka berikutnya setelah $2^{31} - 1$ adalah -2^{31}.

2.3.1 Operasi Bit

Operasi And

Operasi And x & y menghasilkan bilangan yang memiliki satu bit pada posisi dimana x dan y memiliki satu bit. Misalnya, 22 & 26=18, karena

$$\begin{array}{r} 10110\ (22) \\ \&\ 11010\ (26) \\ \hline = 10010\ (18) \end{array}$$

Dengan menggunakan operasi and, kita dapat memeriksa apakah suatu bilangan x genap karena $x\ \&1$=0jika x genap, dan $x\ \&1 = 1$ jika x ganjil. Lebih umum, x habis dibagi 2^k tepat ketika $x\ \&\ (2^k - 1) = 0$.

Operasi Or

Oroperasi x | y menghasilkan angka yang memiliki satu bit di posisi di mana setidaknya satu dari x dan y memiliki satu bit. Misalnya, 22|26=30, karena

$$\begin{array}{r} 10110\ (22) \\ |\ 11010\ (26) \\ \hline = 11110\ (30) \end{array}$$

Operasi Xor

Operasi xor x ˆ y menghasilkan bilangan yang memiliki satu bit pada posisi dimana tepat satu dari x dan y memiliki satu bit. Misalnya, 22 ˆ 26 = 12, karena

$$\begin{array}{r} 10110\ (22) \\ \hat{}\ 11010\ (26) \\ \hline = 01100\ (12) \end{array}$$

Operasi Not

Operasi not ~x menghasilkan angka di mana semua bit x telah dibalik. Rumus ~x =− x−1 berlaku, misalnya, ~29=−30. Hasil dari operasi not pada tingkat bit tergantung pada panjang representasi bit, karena operasi tersebut membalikkan semua bit. Misalnya, jika angkanya adalah angka int 32-bit, hasilnya adalah sebagai berikut:

$$x = 29\ 00000000000000000000000000011101$$
$$\sim x = 30\ 11111111111111111111111111100010$$

Pergeseran Shift

Left bit shift x x << k menambahkan k bit nol ke angka, dan pergeseran bit kanan x >> k menghapus k bit terakhir dari angka. Misalnya, 14 << 2=56, karena 14 dan 56 sesuai dengan 1110 dan 111000. Demikian pula, 49 >> 3=6, karena 49 dan 6 sesuai dengan 110001 dan 110. Perhatikan bahwa x << k sesuai dengan mengalikan x dengan 2^k, dan x >> k sesuai dengan pembagian x dengan 2^k yang dibulatkan ke bawah menjadi bilangan bulat.

Bit Mask

Sebuah bit mask dari bentuk 1 << k memiliki satu bit di posisi k, dan semua bit lainnya adalah nol, sehingga kita dapat menggunakan masker tersebut untuk mengakses bit tunggal angka. Secara khusus, bit ke-k dari suatu bilangan adalah tepat ketika x & (1 << k) bukan nol. Kode berikut mencetak representasi bit dari bilangan int x:

```
for (int k = 31; k >= 0; k--) {
        if (x&(1<<k)) cout << "1";
        else cout << "0";
}
```

Dimungkinkan juga untuk memodifikasi bit tunggal angka menggunakan ide serupa. Rumus x | (1 << k) menetapkan bit ke-k dari x menjadi satu, rumus x & ~(1 << k) menetapkan bit ke-k dari x nol, dan rumus x’(1 << k) membalikkan bit ke-k dari x. Kemudian, rumus x & (x−1) menetapkan satu bit terakhir dari x tozero, dan rumus x &−x menetapkan semua satu bit menjadi nol, kecuali satu bit terakhir. Rumus x | (x 1) membalikkan semua bit setelah bit terakhir. Akhirnya, bilangan positif x adalah pangkat dua tepat ketika x & (x 1) =0. Satu perangkap saat menggunakan topeng bit

adalah bahwa 1<<k selalu merupakan topeng bit int. Cara mudah untuk membuat topeng bit panjang panjang adalah 1LL<<k.

Fungsi Tambahan

Kompilator g++ juga menyediakan fungsi berikut untuk menghitung bit:

- __builtin_clz(x): jumlah nol di awal representasi bit
- __builtin_ctz(x): jumlah nol di akhir representasi bit
- __builtin_popcount(x): jumlah yang ada dalam representasi bit
- __builtin_parity(x): paritas (genap atau ganjil) dari jumlah satu dalam representasi bit

Fungsi-fungsi tersebut dapat digunakan sebagai berikut:

```
int x = 5328; // 00000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; //19
cout << __builtin_ctz(x) << "\n"; //4
cout << __builtin_popcount(x) << "\n";//5
cout << __builtin_parity(x) << "\n";//1
```

Perhatikan bahwa fungsi di atas hanya mendukung angka int, tetapi ada juga versi panjang dari fungsi yang tersedia dengan akhiran ll.

2.3.2 Representasi Set

Setiap himpunan bagian dari himpunan {0, 1, 2,..., n1} dapat direpresentasikan sebagai bilangan bulat n bit yang satu bitnya menunjukkan elemen mana yang termasuk dalam himpunan bagian. Ini adalah cara yang efisien untuk merepresentasikan himpunan, karena setiap elemen hanya memerlukan satu bit memori, dan operasi himpunan dapat diimplementasikan sebagai operasi bit. Misalnya, sejak intisa32-bittype, sebuah bilangan int dapat mewakili setiap subset dari himpunan{0,1,2,...,31}. Representasi bit dari himpunan {1,3,4,8} adalah

00000000000000000000000100011010,

yang sesuai dengan angka $2^8 + 2^4 + 2^3 + 2^1 = 282$. Kode berikut mendeklarasikan variabel int x yang dapat berisi subset {0,1,2,...,31}. Setelah ini, kode menambahkan elemen 1, 3, 4, dan 8 ke himpunan dan mencetak ukuran himpunan.

```
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; / / 4
```

Kemudian, kode berikut mencetak semua elemen yang termasuk dalam set:

```
for (int i = 0; i < 32; i++) {
if (x&(1<<i)) cout << i << " ";
}
// output :1348
```

Operasi Set

Tabel 2.1 menunjukkan bagaimana operasi himpunan dapat diimplementasikan sebagai operasi bit. Sebagai contoh, kode berikut pertama membangun himpunan x ={ 1,3,4,8}dan y ={ 3,6,8,9}dan kemudian membangun himpunan z = x y ={ 1,3,4 ,6,8,9}:

Tabel2.1 Menerapkan operasi yang ditetapkan sebagai operasi bit		
Operasi (Operation)	Sintak Set	Sintak Bit
Persimpangan (Intersection)	a ∩ b	a & b
Persatuan (Union)	a ∪ b	a | b
Melengkapi (Complement)	ā	~ a
Perbedaan (Difference)	a \ b	a & (~b)

```
int x = (1<<1)|(1<<3)|(1<<4)|(1<<8);
int y = (1<<3)|(1<<6)|(1<<8)|(1<<9);
int z = x|y;
cout << __builtin_popcount(z) << "\n"; // 6
```

Kode berikut melewati himpunan bagian dari{0,1,…,n−1}:

```
for (int b = 0; b < (1<<n); b++) {
    // process subset b
}
```

Kemudian, kode berikut melewati himpunan bagian dengan tepat k elemen:

```
for (int b = 0; b < (1<<n); b++) {
    if (__builtin_popcount(b) == k) {
        // process subset b
    }
}
```

Akhirnya, kode berikut melewati himpunan bagian dari himpunan x:

```
int b = 0;
do {
    // process subset b
} while (b=(b-x)&x);
```

C++ Bitset

Pustaka standar C++ juga menyediakan struktur bitset, yang sesuai dengan array yang nilainya masing-masing 0 atau 1. Misalnya, kode berikut membuat bitset dari 10 elemen:

```
bitset<10> s;
s[1] = 1;
s[3] = 1;
s[4] = 1;
s[7] = 1;
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

Hitungan fungsi mengembalikan jumlah satu bit dalam bitset:

```
cout << s.count() << "\n"; // 4
```

Juga operasi bit dapat langsung digunakan untuk memanipulasi bit:

```
bitset<10> a, b;
// ...
bitset<10> c = a&b;
bitset<10> d = a|b;
bitset<10> e = a^b;
```

BAB 3
EFISIENSI

Efisiensi algoritme memainkan peran sentral dalam pemrograman kompetitif. Dalam bab ini, kita mempelajari alat yang mempermudah merancang algoritme yang efisien. Bab ini akan memperkenalkan konsep kompleksitas waktu, yang memungkinkan untuk memperkirakan waktu berjalan dari algoritma tanpa menerapkannya. Kompleksitas waktu dari suatu algoritma menunjukkan seberapa cepat waktu berjalannya meningkat ketika ukuran input bertambah. Bagianselanjutnya menyajikan dua contoh masalah yang dapat diselesaikan dengan banyak cara. Dalam kedua masalah tersebut, kita dapat dengan mudah merancang solusi brute force yang lambat, tetapi ternyata kita juga dapat membuat algoritme yang jauh lebih efisien.

3.1 Kompleksitas Waktu

Kompleksitas waktu dari perkiraan algoritma menunjukkan banyak waktu yang akan digunakan oleh algoritma untuk input yang diberikan. Dengan menghitung kompleksitas waktu, kita sering dapat mengetahui apakah algoritma tersebut cukup cepat untuk menyelesaikan suatu masalah—tanpa mengimplementasikannya. Kompleksitas waktu dilambangkan dengan *O* (⋯) di mana tiga titik mewakili beberapa fungsi. Biasanya, variabel n menunjukkan ukuran input. Misalnya, jika inputnya adalah array angka, n akan menjadi ukuran array, dan jika inputnya adalah string, *n* akan menjadi panjang string.

3.1.1 Hukum Kalkulasi

Jika sebuah kode terdiri dari satu perintah, kompleksitas waktunya adalah *O* (1). Misalnya, kompleksitas waktu dari kode berikut adalah *O* (1).

```
a++;
b++;
c = a+b;
```

Kompleksitas waktu dari sebuah loop memperkirakan berapa kali kode di dalam loop dieksekusi. Misalnya, kompleksitas waktu dari kode berikut adalah *O(n),* karena kode di dalam loop dieksekusi *n* kali. Kami berasumsi bahwa "..." menunjukkan kode yang kompleksitas waktunya adalah *O*(1).

```
for (int i = 1; i <= n; i++) {
    ...
}
```

Maka, kompleksitas waktu dari kode berikut adalah O(n2):

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        ...
    }
}
```

Secara umum, jika ada *k* loop bersarang dan setiap loop melewati nilai *n*, kompleksitas waktunya adalah $O(n^k)$. Kompleksitas waktu tidak memberi tahu kita berapa kali kode di dalam loop dieksekusi, karena hanya menunjukkan urutan pertumbuhan dan

mengabaikan faktor konstan. Dalam contoh berikut, kode di dalam loop dieksekusi *3n, n +5, [n/2]* kali, tetapi kompleksitas waktu setiap kode adalah *O(n)*.

```
for (int i = 1; i <= 3*n; i++) {
    ...
}
```

```
for (int i = 1; i <= n+5; i++) {
    ...
}
```

```
for (int i = 1; i <= n; i += 2) {
    ...
}
```

Contoh lain, kompleksitas waktu dari kode berikut adalah $O(n^2)$, karena kode di dalam loop dieksekusi $1+2+...+n = ½ (n^2 +n)$ kali.

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
        ...
    }
}
```

Jika suatu algoritma terdiri dari fase-fase yang berurutan, kompleksitas waktu total adalah kompleksitas waktu terbesar dari satu fase. Alasan untuk ini adalah bahwa fase paling lambat adalah hambatan dari algoritma. Sebagai contoh, kode berikut terdiri dari tiga fase dengan kompleksitas waktu $O(n)$, $O(n^2)$, dan $O(n)$. Jadi, total kompleksitas waktu adalah $O(n^2)$.

```
for (int i = 1; i <= n; i++) {
    ...
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        ...
    }
}
for (int i = 1; i <= n; i++) {
    ...
}
```

Terkadang kompleksitas waktu tergantung pada beberapa faktor, dan rumus kompleksitas waktu mengandung beberapa variabel. Misalnya, kompleksitas waktu dari kode berikut adalah *O(nm):*

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        ...
    }
}
```

Kompleksitas waktu dari fungsi rekursif tergantung pada berapa kali fungsi dipanggil dan kompleksitas waktu dari satu panggilan. Kompleksitas waktu total adalah produk dari nilai-nilai ini. Sebagai contoh, perhatikan fungsi berikut:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        ...
    }
}
```

Pemanggilan *f(n)* menyebabkan n pemanggilan fungsi, dan kompleksitas waktu setiap pemanggilan adalah *O(1),* sehingga total kompleksitas waktu adalah *O(n).* Sebagai contoh lain, perhatikan fungsi berikut:

```
void f(int n) {
    if (n == 1) return;
    f(n-1);
}
```

Apa yang terjadi ketika fungsi dipanggil dengan parameter n? Pertama, ada dua panggilan dengan parameter n−1, lalu empat panggilan dengan parameter n−2, lalu delapan panggilan dengan parameter n−3, dan seterusnya. Secara umum akan ada 2^k panggilan dengan parameter n−k dimana k =0,1,...,n−1. Jadi, kompleksitas waktunya adalah

$$1+2+4+\cdots+2^{n-1} = 2^{n-1} = O(2^n).$$

3.1.2 Kompleksitas Waktu Umum

Daftar berikut berisi kompleksitas waktu umum dari algoritma:

- *O (1)* Waktu berjalan dari algoritma waktu-konstan tidak bergantung pada ukuran input. Algoritma waktu-konstan yang khas adalah formula langsung yang menghitung jawabannya.
- *O (log n)* Sebuah algoritma logaritmik sering membagi dua ukuran input pada setiap langkah. Waktu berjalan dari algoritma tersebut adalah logaritma, karena log2 n sama dengan berapa kali n harus dibagi 2 untuk mendapatkan 1. Perhatikan bahwa basis logaritma tidak ditampilkan dalam kompleksitas waktu.
- *O(√n)* Algoritma akar kuadrat lebih lambat dari O(log n) tetapi lebih cepat dari O(n) Sifat khusus akar kuadrat adalah bahwa n =n/√n, s pada elemen dapat dibagi menjadi O(√n) blok elemen O(√n).
- *O(n)* Algoritme linier melewati input beberapa kali secara konstan. Ini seringkali merupakan kompleksitas waktu terbaik, karena biasanya diperlukan untuk mengakses setiap elemen input setidaknya sekali sebelum melaporkan jawabannya.
- *O(n log n)* Kompleksitas waktu ini sering menunjukkan bahwa algoritma mengurutkan input, karena kompleksitas waktu dari algoritma pengurutan yang efisien adalah O(n log n). Kemungkinan lain adalah bahwa algoritma menggunakan struktur data di mana setiap operasi membutuhkan waktu O(log n).
- *O(n^2)* Sebuah algoritma kuadrat sering berisi dua loop bersarang. Dimungkinkan untuk melewati semua pasangan elemen input dalam waktu O(n^2).
- *O(n^3)* Sebuah algoritma kubik sering berisi tiga loop bersarang. Dimungkinkan untuk melewati semua triplet elemen input dalam waktu O(n^3).

- *O(2^n)* Kompleksitas waktu ini sering menunjukkan bahwa algoritme iterasi melalui semua himpunan bagian dari elemen input. Misalnya, himpunan bagian dari {1, 2, 3} adalah ∅, {1}, {2}, {3}, {1, 2}, {1, 3},{2, 3}, dan {1, 2, 3}.
- *O (n!)* Kompleksitas waktu ini sering menunjukkan bahwa algoritme iterasi melalui semua permutasi elemen input. Misalnya, permutasi {1, 2, 3} adalah (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1 , 2), dan (3, 2, 1).

Suatu algoritma disebut polinomial jika kompleksitas waktunya paling banyak *O(n^k)* di mana *k* adalah konstanta. Semua kompleksitas waktu di atas kecuali *O(2^n)* dan *O(n!)* adalah polinomial. Dalam praktiknya, konstanta biasanya kecil, dan oleh karena itu kompleksitas waktu polinomial secara kasar berarti bahwa algoritma dapat memproses input yang besar . Kebanyakan algoritma dalam buku ini adalah polinomial. Namun, ada banyak masalah penting yang algoritma polinomialnya tidak diketahui, yaitu, tidak ada yang tahu bagaimana menyelesaikannya secara efisien. Masalah NP-hard adalah kumpulan masalah penting, yang tidak diketahui algoritma polinomialnya.

3.1.3 Memperkirakan Efisiensi

Dengan menghitung kompleksitas waktu suatu algoritme, sebelum menerapkan algoritme, dimungkinkan untuk memeriksa apakah algoritme tersebut cukup efisien untuk memecahkan masalah. Titik awal untuk estimasi adalah kenyataan bahwa komputer modern dapat melakukan beberapa ratus juta operasi sederhana dalam satu detik. Misalnya, asumsikan bahwa batas waktu untuk suatu masalah adalah satu detik dan ukuran input adalah n = 10^5. Jika kompleksitas waktu adalah O(n^2), algoritme akan melakukan sekitar $(10^5)^2 = 10^{10}$ operasi. Ini akan memakan waktu setidaknya beberapa puluh detik, sehingga algoritme tampaknya terlalu lambat untuk menyelesaikan masalah. Namun, jika kompleksitas waktu O(n log n), hanya akan ada sekitar $10^5 \log 10^5 \approx 1.6 \cdot 10^6$ operasi, dan algoritma pasti akan sesuai dengan batas waktu.

Di sisi lain, mengingat ukuran input, kita dapat mencoba menebak kompleksitas waktu yang dibutuhkan dari algoritma yang memecahkan masalah. Tabel 3.1 berisi beberapa perkiraan yang berguna dengan asumsi batas waktu satu detik. Sebagai contoh, jika ukuran input adalah n = 10^5, mungkin diharapkan bahwa kompleksitas waktu dari algoritma adalah O(n) atau O(n log n). Informasi ini membuat lebih mudah untuk merancang algoritma, karena ada kebenaran dari pendekatan yang akan menghasilkan algoritma dengan kompleksitas waktu yang lebih buruk. Namun, penting untuk diingat bahwa kompleksitas waktu hanyalah perkiraan efisiensi, karena menyembunyikan faktor konstan. Misalnya, suatu algoritma yang berjalan dalam waktu O(n) dapat melakukan operasi n/2 atau 5 n, yang memiliki pengaruh penting pada waktu berjalan aktual dari algoritma tersebut.

Tabel 3.1 Memperkirakan kompleksitas waktu dari ukuran input

Ukuran Input	**Ekpektasi Perkiraan waktu**
n ≤10	*O(n!)*
n ≤20	*O(2^n)*
n ≤500	*O(2^3)*
n ≤5000	*O(2^2)*
n ≤10^6	*O(n log n) or O(n)*
N lebih besar	*O(1) or O(log n)*

3.1.4 Definisi Formal

Apa sebenarnya yang dimaksud dengan algoritma bekerja dalam waktu *O(f (n))*? Ini berarti bahwa ada konstanta *c* dan n_0 sehingga algoritma melakukan paling banyak operasi *cf (n)* untuk semua input di mana $n \geq n_0$. Dengan demikian, notasi *O* memberikan batas atas untuk waktu berjalan algoritma untuk input yang cukup besar. Misalnya, secara teknis benar untuk mengatakan bahwa kompleksitas waktu dari algoritma berikut adalah $O(n^2)$.

```
For (int i : 1 ; i <= n; i++)  {
. . .
}
```

Namun, batas yang lebih baik adalah *O(n),* dan akan sangat menyesatkan untuk memberikan batas $O(n^2)$, karena semua orang sebenarnya berasumsi bahwa notasi *O* digunakan untuk memberikan perkiraan kompleksitas waktu yang akurat. Ada juga dua notasi umum lainnya. Notasi *Ω* memberikan batas bawah untuk waktu berjalan dari algoritma analisis. Kompleksitas waktu dari algoritma analisis Ω *(f (n)),* jika ada konstanta *c* dan n_0 sedemikian rupa sehingga algoritma melakukan setidaknya *cf (n)* operasi untuk semua input di mana $n \geq n_0$. Akhirnya, notasi memberikan batas yang tepat: kompleksitas waktu dari suatu algoritma adalah Θ *(f (n))* jika keduanya adalah *O(f (n))* dan Ω (*f (n)).* Misalnya, karena kompleksitas waktu dari algoritma di atas adalah *O(n)* dan *Ω (n),* ia juga *Θ (n).* Kita dapat menggunakan notasi di atas dalam banyak situasi, tidak hanya untuk mengacu pada kompleksitas waktu dari algoritma. Misalnya, kita mungkin mengatakan bahwa array berisi nilai *O(n),* atau bahwa suatu algoritma terdiri dari putaran *O(log n).*

3.2 Contoh

Pada bagian ini kita membahas dua masalah desain algoritma yang dapat diselesaikan dengan beberapa cara yang berbeda. Kami mulai dengan algoritma brute force sederhana, dan kemudian membuat solusi yang lebih efisien dengan menggunakan berbagai ide desain algoritma.

3.2.1 Jumlah Subarray Maksimum

Diberikan array n angka, tugas pertama kita adalah menghitung jumlah subarray maksimum, yaitu, jumlah terbesar yang mungkin dari urutan nilai berurutan dalam array. Masalahnya menarik ketika mungkin ada nilai negatif dalam array. Sebagai contoh, Gambar 3.1 menunjukkan sebuah array dan subarray jumlah maksimumnya.

−1	2	4	−3	5	2	−5	2

−1	2	4	−3	5	2	−5	2

Gambar 3.1 Subarray jumlah maksimum dari array ini adalah [2,4,−3,5,2], yang jumlahnya 10

Solusi Waktu $O(n^3)$. Cara langsung untuk menyelesaikan masalah adalah dengan menelusuri semua kemungkinan sub-baris, menghitung jumlah nilai di setiap sub-baris dan mempertahankan jumlah maksimum. Kode berikut mengimplementasikan algoritma ini:

```
int best = 0 ;
for (int a = 0; a < n ; a++)  {
        for (int b = a; b < n; b++)  {
        int sum = 0;
        for (int k = a; k <= b; k++) {
        sum += array[k];
        }
        best = max(best,sum);
        }
}
cout << best << "\n";
```

Variabel a dan b memperbaiki indeks pertama dan terakhir dari subarray, dan jumlah nilai dihitung ke jumlah variabel. Variabel best berisi jumlah maksimum yang ditemukan selama pencarian. Kompleksitas waktu dari algoritme adalah $O(n^3)$, karena terdiri dari tiga loop bersarang yang melewati input.

Solusi Waktu $O(n^2)$ Sangat mudah untuk membuat algoritme lebih efisien dengan menghapus satu loop darinya. Ini dimungkinkan dengan menghitung jumlah pada saat yang sama ketika ujung kanan subarray bergerak. Hasilnya adalah kode berikut:

```
int best = 0;
for (int a = 0; a < n; a++) {
        int sum = 0;
        for (int b = a; b < n; b++) {
        sum += array[b];
        best = max(best,sum);
        }
}
cout << best << "\n";
```

Setelah perubahan ini, kompleksitas waktu adalah $O(n^2)$. O(n) Solusi Waktu. Ternyata masalah dapat diselesaikan dalam waktu O(n), yang berarti bahwa satu loop saja sudah cukup. Idenya adalah untuk menghitung, untuk setiap posisi array, jumlah maksimum dari subarray yang berakhir pada posisi itu. Setelah ini, jawaban dari masalah adalah jumlah maksimum dari jumlah tersebut.

Pertimbangkan submasalah untuk menemukan jumlah subarray maksimum yang berakhir pada posisi k. Ada dua kemungkinan:

1. Subarray hanya berisi elemen pada posisi k.
2. Subarray terdiri dari subarray yang berakhir pada posisi k 1, diikuti oleh elemen pada posisi k.

Dalam kasus terakhir, karena kita ingin mencari subarray dengan jumlah maksimum, subarray yang berakhir pada posisi *k−1* juga harus memiliki jumlah maksimum. Jadi, kita dapat menyelesaikan masalah secara efisien dengan menghitung jumlah subarray maksimum untuk setiap posisi akhir dari kiri ke kanan. Kode berikut mengimplementasikan algoritma:

```
int best = 0, sum = 0;
for (int k = 0; k < n; k++) {
        sum = max(array[k],sum+array[k]);
        best = max(best,sum);
}
cout << best << "\n";
```

Algoritme hanya berisi satu loop yang melewati input, sehingga kompleksitas waktunya adalah O(n). Ini juga merupakan kompleksitas waktu terbaik, karena algoritma apa pun untuk masalah tersebut harus memeriksa semua elemen array setidaknya sekali.

Perbandingan Efisiensi

Seberapa efisien algoritme di atas dalam praktiknya? Tabel 3.2 menunjukkan waktu berjalan dari algoritme di atas untuk nilai yang berbeda dari komputer non modern. Dalam setiap pengujian, input dibangkitkan secara acak, dan waktu yang dibutuhkan untuk membaca input tidak diukur. Perbandingan menunjukkan bahwa semua algoritma bekerja dengan cepat ketika ukuran input kecil, tetapi input yang lebih besar menghasilkan perbedaan yang luar biasa dalam waktu berjalan. Algoritma $O(n^3)$ menjadi lambat ketika $n = 10^4$, dan algoritma $O(n^2)$ menjadi lambat ketika $n = 10^5$. Hanya algoritma O(n) yang mampu memproses input terbesar sekalipun secara instan.

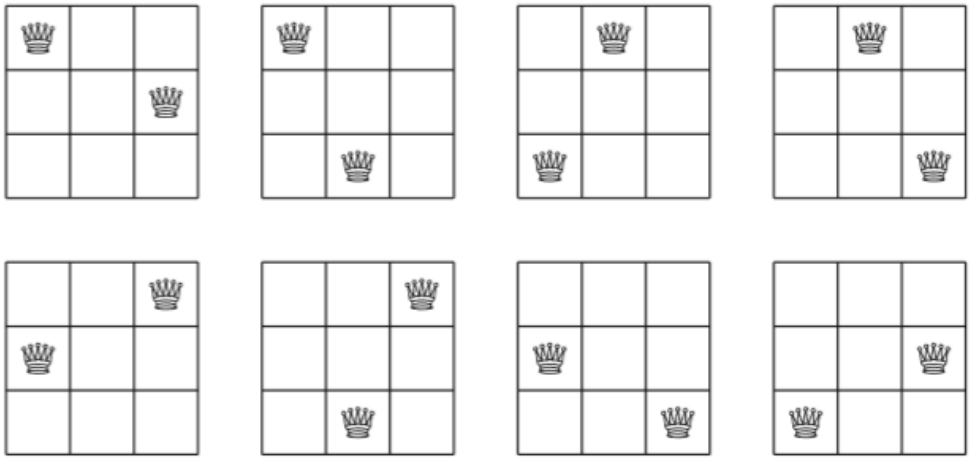
Tabel 3.2 Membandingkan waktu berjalan dari algoritma penjumlahan subarray maksimum

Ukuran aray *n*	*$O(n^3)$ (s)*	*$O(n^2)$ (s)*	*O(n) (s)*
10^2	0.0	0.0	0.0
10^3	0.1	0.0	0.0
10^4	>10.0	0.1	0.0
10^5	>10.0	5.3	0.0
10^6	>10.0	>10.0	0.0
10^7	>10.0	>10.0	0.0

3.2.2 Masalah Dua Ratu

Diberikan papan catur n × n, masalah kita berikutnya adalah menghitung jumlah cara kita dapat menempatkan dua ratu di papan sedemikian rupa sehingga mereka tidak saling menyerang. Misalnya, seperti yang ditunjukkan Gambar 3.2, ada delapan cara untuk menempatkan dua ratu di papan *3x3.* Misalkan q(n) menunjukkan jumlah kombinasi yang valid untuk papan *n x* n.

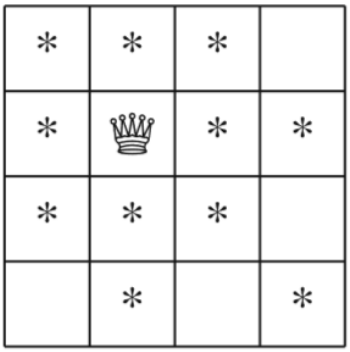
Misalnya, *q(3) =8,* dan Tabel 3.3 menunjukkan nilai *q(n*) untuk *1≤n 10*. Untuk memulainya, cara sederhana untuk menyelesaikan masalah adalah melalui semua cara yang mungkin untuk menempatkan dua ratu di papan dan menghitung kombinasi di mana ratu tidak saling menyerang. Algoritma seperti itu bekerja dalam waktu $O(n^4)$, karena ada *n2* cara untuk memilih posisi ratu pertama, dan untuk setiap posisi seperti itu, ada *n2 1* cara untuk memilih posisi ratu kedua. Karena jumlah kombinasi bertambah cepat, algoritma yang menghitung kombinasi satu per satu pasti akan terlalu lambat untuk memproses nilai n yang lebih besar. Jadi, untuk membuat algoritma yang efisien, kita perlu mencari cara untuk menghitung kombinasi dalam kelompok. Satu pengamatan yang berguna adalah cukup mudah untuk menghitung jumlah kotak yang diserang oleh satu ratu (Gambar 3.3). Pertama, selalu menyerang *n−1* kotak secara horizontal dan *n−1* kotak secara vertikal. Kemudian, untuk kedua diagonal, menyerang kotak *d−1* di mana *d* adalah jumlah kotak pada diagonal. Dengan menggunakan informasi ini, kita dapat menghitung dalam waktu *O(1)* jumlah kotak di mana ratu lainnya dapat ditempatkan, yang menghasilkan algoritma waktu $O(n^2)$.



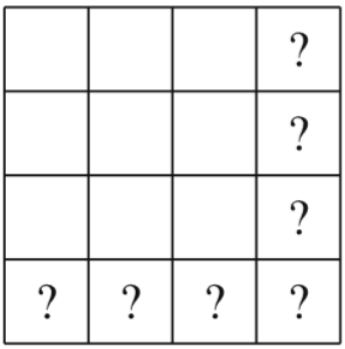
Gambar 3.2 Semua cara yang mungkin untuk menempatkan dua ratu yang tidak menyerang di papan catur 3×3

Tabel 3.3 Nilai pertama dari fungsi q(n): banyaknya cara untuk menempatkan dua ratu yang tidak menyerang pada papan catur n×n

Ukuran papan *n*	**Jumlah Jalan *q(n)***
1	0
2	0
3	8
4	44
5	140
6	340
7	700
8	1288
9	2184
10	3480



Gambar 3.3 Ratu menyerang semua kotak bertanda "*" di papan



Gambar 3.4 Kemungkinan posisi ratu pada baris dan kolom terakhir

Cara lain untuk mendekati masalah adalah dengan mencoba merumuskan fungsi rekursif yang menghitung jumlah kombinasi. Pertanyaannya adalah: jika kita mengetahui nilai *q(n),* bagaimana kita dapat menggunakannya untuk menghitung nilai *q(n+1)?* Untuk mendapatkan solusi rekursif, kita dapat memfokuskan pada baris terakhir dan kolom terakhir dari papan n×n (Gambar 3.4). Pertama, jika tidak ada ratu pada baris atau kolom terakhir, jumlah kombinasinya adalah *q(n−1).* Kemudian, ada *2n−1* posisi ratu di baris atau kolom terakhir. Ini menyerang *3(n−1)* kotak, jadi ada n^2 - *3(n−1)−1* posisi untuk ratu lainnya. Terakhir, terdapat kombinasi *(n−1)(n−2)* dimana kedua ratu berada pada baris atau kolom terakhir. Karena kami menghitung kombinasi tersebut dua kali, kami harus menghapus nomor ini dari hasil. Dengan menggabungkan semua ini, kami mendapatkan rumus rekursif

$$q(n) = q(n-1) + (2n-1)(n2\ -3(n-1)-1) - (n-1)(n-2)$$
$$= q(n-1) + 2(n-1)2(n-2),$$

yang memberikan solusi O(n) untuk masalah tersebut. Akhirnya, ternyata ada juga rumus bentuk tertutup

$$q(n) = \frac{n^4}{2} - \frac{5n^3}{3} + \frac{3n^2}{2} - \frac{n}{3}$$

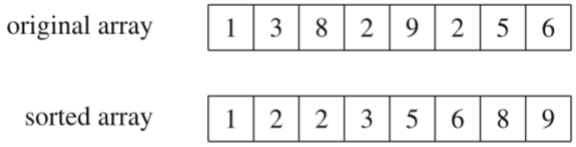
yang dapat dibuktikan dengan menggunakan rumus induksi dan rekursif. Dengan menggunakan rumus ini, kita dapat menyelesaikan masalah dalam waktu *O(1).*

BAB 4
ALGORITMA PENYORTIRAN

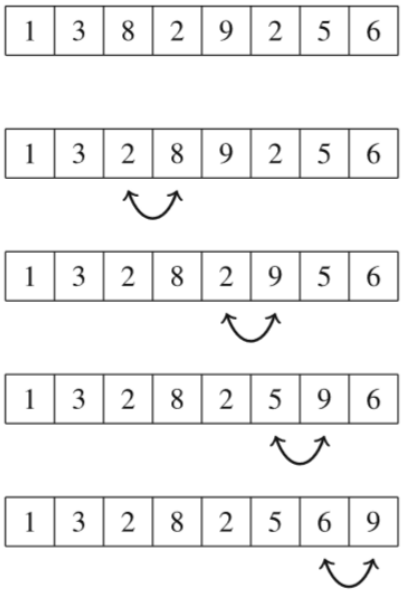
Banyak algoritme yang efisien didasarkan pada pengurutan data input, karena pengurutan sering kali membuat pemecahan masalah menjadi lebih mudah. Bab ini membahas teori dan praktik pengurutan sebagai alat desain algoritma. Bagian pertama membahas tiga algoritma pengurutan penting: bubble sort, merge sort, dan counting sort. Setelah ini, kita akan belajar bagaimana menggunakan algoritma pengurutan yang tersedia di pustaka standar C++. Bagian selanjutnya menunjukkan bagaimana pengurutan dapat digunakan sebagai subrutin untuk membuat algoritme yang efisien. Misalnya, untuk menentukan dengan cepat apakah semua elemen array adalah unik, pertama-tama kita dapat mengurutkan array dan kemudian cukup memeriksa semua pasangan elemen berurutan. Bagian akhir menyajikan algoritma pencarian biner, yang merupakan blok pembangun penting lainnya dari algoritma yang efisien.

Algoritma Penyortiran

Masalah dasar dalam pengurutan adalah sebagai berikut: Diberikan sebuah array yang berisi n elemen, urutkan elemen dalam urutan yang meningkat. Sebagai contoh, Gambar 4.1 menunjukkan sebuah array sebelum dan sesudah pengurutan. Pada bagian ini kita akan melalui beberapa algoritma penyortiran dasar dan memeriksa propertinya. Sangat mudah untuk merancang algoritma pengurutan waktu O(n2), tetapi ada juga algoritma yang lebih efisien. Setelah membahas teori pengurutan, kita akan fokus pada penggunaan pengurutan dalam praktik di C++.



Gambar 4.1 Array sebelum dan sesudah sorting



Gambar 4.2 Putaran pertama pengurutan gelembung

4.1 Sortir Gelembung

Bubble sort adalah algoritma pengurutan sederhana yang bekerja dalam waktu O(n2). Algoritma terdiri dari n putaran, dan pada setiap putaran, iterasi melalui elemen array. Setiap kali dua elemen berurutan ditemukan dalam urutan yang salah, algoritme menukarnya. Algoritma tersebut dapat diimplementasikan sebagai berikut:

```
for (int i = 0; i < n; i++) {
        for (int j = 0; j < n-1; j++) {
                if (array[j] > array[j+1]) {
                        swap(array[j],array[j+1]);
                }
        }
}
```

Setelah putaran pertama bubble sort, elemen terbesar akan berada di posisi yang benar, dan lebih umum lagi, setelah k putaran, k elemen terbesar akan berada di posisi yang benar. Jadi, setelah n putaran, seluruh array akan diurutkan. Sebagai contoh, Gambar 4.2 menunjukkan putaran pertama swap ketika bubble sort digunakan untuk mengurutkan array. Bubble sort adalah contoh algoritma pengurutan yang selalu menukar elemen berurutan dalam array. Ternyata kompleksitas waktu dari algoritma tersebut selalu paling sedikit O(n2), karena dalam kasus terburuk, O(n2) swap diperlukan untuk menyortir array.

0	1	2	3	4	5	6	7
1	2	2	6	3	5	9	8

Gambar 4.3 Array ini memiliki tiga inversi: (3,4), (3,5), dan (6,7)

Inversi

Sebuah konsep yang berguna ketika menganalisis algoritma pengurutan adalah inversi: sepasang indeks array (a,b) sedemikian rupa sehingga a < b dan array[a] >array[b], yaitu, elemen-elemen berada dalam urutan yang salah. Misalnya, array pada Gambar 4.3 memiliki tiga inversi: (3,4), (3,5), dan (6,7). Jumlah inversi menunjukkan berapa banyak pekerjaan yang diperlukan untuk mengurutkan array. Sebuah array benar-benar diurutkan ketika tidak ada inversi. Di sisi lain, jika elemen array berada dalam urutan terbalik, jumlah inversi adalah

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

yang terbesar mungkin. Mengganti sepasang elemen berurutan yang berada dalam urutan yang salah akan menghapus tepat satu inversi dari array. Oleh karena itu, jika algoritma pengurutan hanya dapat menukar elemen berurutan, setiap swap menghilangkan paling banyak satu inversi, dan kompleksitas waktu algoritma setidaknya O(n2).

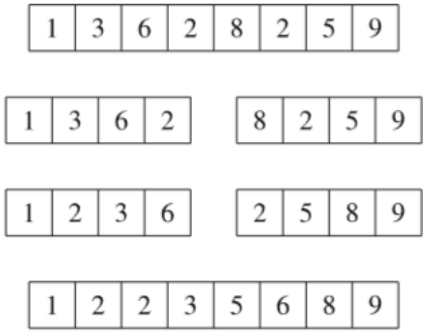
4.1.1 Gabungkan Sortir

Jika kita ingin membuat algoritme pengurutan yang efisien, kita harus dapat mengurutkan ulang elemen yang berada di bagian array yang berbeda. Ada beberapa algoritma pengurutan yang bekerja dalam waktu O(n log n). Salah satunya adalah merge sort, yang berbasis rekursi. Merge sort mengurutkan subarray array[a ...b]sebagai berikut:

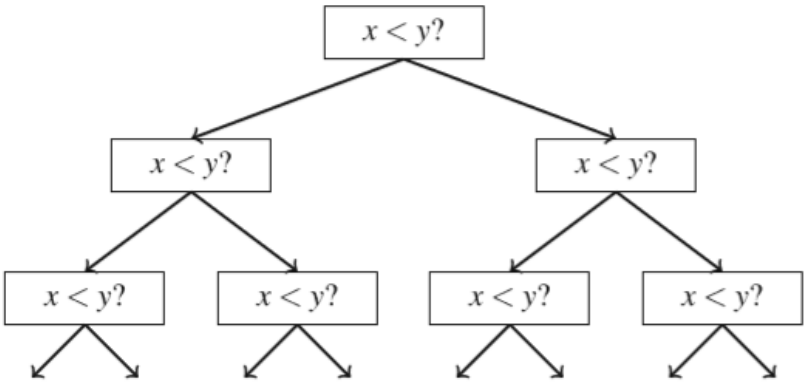
1. Jika *a =b,* jangan lakukan apa-apa, karena subarray yang hanya berisi satu elemen sudah diurutkan.
2. Hitung posisi elemen tengah*: k =[(a+b)/2]*
3. Urutkan secara rekursif larik subarray*[a ...k].*
4. Urutkan secara rekursif larik subarray*[k+1...b].*

5. Gabungkan subarray terurut *[a ...k]* dan larik *[k+1...b]* ke dalam larik subarray terurut *[a ...b]*.

Sebagai contoh, Gambar 4.4 menunjukkan bagaimana merge sort mengurutkan array delapan elemen. Pertama, algoritma membagi array menjadi dua subarray dari empat elemen. Kemudian, ia mengurutkan subarray ini secara rekursif dengan memanggil dirinya sendiri. Terakhir, ia menggabungkan subarray yang diurutkan ke dalam larik terurut yang terdiri dari delapan elemen. Merge sort adalah algoritma yang efisien, karena membagi dua ukuran subarray pada setiap langkah. Kemudian, penggabungan subarray yang diurutkan dimungkinkan dalam waktu linier, karena mereka sudah diurutkan. Karena ada level rekursif *O(log n),* dan pemrosesan setiap level membutuhkan total waktu O(n), algoritme bekerja dalam waktu *O(n log n).*



Gambar 4.4 Mengurutkan array menggunakan merge sort



Gambar 4.5 Kemajuan algoritma pengurutan yang membandingkan elemen array

4.1.2 Menyortir Batas Bawah

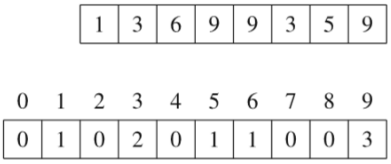
Apakah mungkin untuk mengurutkan array lebih cepat daripada dalam waktu O(n log n)? Ternyata ini tidak mungkin ketika kita membatasi diri pada algoritma pengurutan yang didasarkan pada perbandingan elemen array. Batas bawah untuk kompleksitas waktu dapat dibuktikan dengan mempertimbangkan pengurutan sebagai proses di mana setiap perbandingan dua elemen memberikan lebih banyak informasi tentang isi array. Gambar4.5 mengilustrasikan pohon yang dibuat dalam proses ini. Di sini “x < y?” berarti bahwa beberapa elemen x dan y dibandingkan. Jika x < y, proses berlanjut ke kiri, dan sebaliknya ke kanan. Hasil dari proses tersebut adalah kemungkinan cara untuk mengurutkan array, total n!cara. Untuk alasan ini, ketinggian pohon setidaknya harus

$$log_2(n!) = log_2(1) + log_2(2) + \ldots + log_2(n)$$

Kami mendapatkan batas bawah untuk jumlah ini dengan memilih n/2 elemen terakhir dan mengubah nilai setiap elemen menjadi $log_2(n/2)$. Ini menghasilkan perkiraan

$$log_2(n!) \geq \left(\frac{n}{2}\right).log_2(\frac{n}{2})$$

jadi tinggi pohon dan jumlah langkah terburuk dalam algoritma pengurutan adalah $\Omega(n \log n)$.



Gambar 4.6 Mengurutkan array dengan menggunakan counting sort

4.1.3 Praktek Penyortiran

Batas bawah $\Omega(n \log n)$ tidak berlaku untuk algoritma yang tidak membandingkan elemen array tetapi menggunakan beberapa informasi lain. Contoh dari algoritma tersebut adalah menghitung sort yang mengurutkan array dalam waktu $O(n)$ dengan asumsi bahwa setiap elemen dalam array adalah bilangan bulat antara $0...c$ dan $c = O(n)$. Algoritme membuat larik pembukuan, yang indeksnya merupakan elemen larik asli. Algoritme mengulangi melalui larik asli dan menghitung berapa kali setiap elemen muncul dalam larik.

Sebagai contoh, Gambar 4.6 menunjukkan sebuah array dan array pembukuan yang sesuai. Misalnya nilai pada posisi 3 adalah 2, karena nilai 3 muncul 2 kali pada larik aslinya. Konstruksi susunan pembukuan membutuhkan waktu $O(n)$. Setelah ini, array yang diurutkan dapat dibuat dalam waktu $O(n)$, karena jumlah kemunculan setiap elemen dapat diambil dari array pembukuan. Dengan demikian, total kompleksitas waktu pengurutan pencacahan adalah $O(n)$. Counting sort adalah algoritma yang sangat efisien tetapi hanya dapat digunakan jika konstanta c cukup kecil, sehingga elemen array dapat digunakan sebagai indeks dalam array pembukuan.

Latihan Sortir

Dalam praktiknya, hampir tidak pernah merupakan ide yang baik untuk menerapkan algoritme pengurutan buatan sendiri, karena semua bahasa pemrograman modern memiliki algoritme pengurutan yang baik di pustaka standar mereka. Ada banyak alasan untuk menggunakan fungsi perpustakaan: tentu saja benar dan efisien, dan juga mudah digunakan. Dalam C++, fungsi sort secara efisien [3] mengurutkan isi dari struktur data. Misalnya, kode berikut mengurutkan elemen vektor dalam urutan yang meningkat:

```
vector<int> v = {4,2,5,3,5,8,3};
sort(v.begin(),v.end());
```

Setelah disortir, isi vektor akan menjadi [2, 3, 3, 4, 5, 5, 8]. Urutan pengurutan default meningkat, tetapi urutan terbalik dimungkinkan sebagai berikut:

```
sort(v.rbegin(),v.rend());
```

Array biasa dapat diurutkan sebagai berikut:

```
int n = 7; // array size
```

[3] Standar C++11 mengharuskan fungsi sortir bekerja dalam waktu O(n log n); implementasi yang tepat tergantung pada kompiler.

```
int a[] = {4,2,5,3,5,8,3};
sort(a,a+n);
```

Kemudian, kode berikut mengurutkan string s:

```
string s = "monkey";
sort(s.begin(), s.end());
```

Menyortir string berarti bahwa karakter string diurutkan. Misalnya, string "monyet" menjadi "ekmnoy".

Operator Perbandingan

Fungsi pengurutan mengharuskan operator pembanding didefinisikan untuk tipe data elemen yang akan diurutkan. Saat menyortir, operator ini akan digunakan kapan pun diperlukan untuk mengetahui urutan dua elemen. Sebagian besar tipe data C++ memiliki operator perbandingan bawaan, dan elemen dari tipe tersebut dapat diurutkan secara otomatis. Angka diurutkan menurut nilainya, dan string diurutkan menurut abjad. Pasangan diurutkan terutama menurut elemen pertamanya dan yang kedua menurut elemen keduanya:

```
vector<pair<int,int>> v;
v.push_back({1,5});
v.push_back({2,3});
v.push_back({1,2});
sort(v.begin(), v.end());
// result : [(1, 2), (1, 5), (2, 3)]
```

Dengan cara yang sama, tupel diurutkan terutama oleh elemen pertama, kedua oleh elemen kedua, dll[4]:

```
vector<tuple<int,int,int>> v;
v.push_back({2,1,4});
v.push_back({1,5,3});
v.push_back({2,1,3});
sort(v.begin(), v.end());
//result:[(1,5,3),(2,1,3),(2,1,4)]
```

Struktur yang ditentukan pengguna tidak memiliki operator perbandingan secara otomatis. Operator harus didefinisikan di dalam struct sebagai operator fungsi<, yang parameternya adalah elemen lain dari tipe yang sama. Operator harus mengembalikan true jika elemen lebih kecil dari parameter, dan false sebaliknya. Misalnya, titik struct berikut berisi koordinat x dan y dari suatu titik. Operator pembanding didefinisikan sehingga titik-titik diurutkan terutama oleh koordinat x dan sekunder oleh koordinat y.

```
struct point {
        int x, y;
        bool operator<(const point &p) {
        if (x == p.x) return y < p.y;
        else return x < p.x;
        }
};
```

Fungsi Perbandingan

Dimungkinkan juga untuk memberikan fungsi perbandingan eksternal ke fungsi sortir sebagai fungsi panggilan balik. Misalnya, fungsi perbandingan berikut comp

[4] Perhatikan bahwa di beberapa kompiler lama, fungsi make_tuple harus digunakan untuk membuat tupel, bukan kurung kurawal (misalnya, make_tuple(2,1,4) alih-alih {2,1,4}).

mengurutkan string terutama menurut panjangnya dan yang kedua menurut urutan abjad:

```
bool comp(string a, string b) {
    if (a.size() == b.size()) return a < b;
    else return a.size() < b.size();
}
```

Sekarang vektor string dapat diurutkan sebagai berikut:

```
sort(v.begin(), v.end(), comp);
```

3.2 Memecahkan Masalah dengan Menyortir

Seringkali, kita dapat dengan mudah menyelesaikan masalah dalam waktu $O(n^2)$ menggunakan algoritma brute force, tetapi algoritma seperti itu terlalu lambat jika ukuran inputnya besar. Sebenarnya, tujuan yang sering dalam desain algoritma adalah untuk menemukan algoritma waktu *O(n)* atau *O(n log n)* untuk masalah yang dapat diselesaikan secara sepele dalam waktu $O(n^2)$. Penyortiran adalah salah satu cara untuk mencapai tujuan ini. Misalnya, misalkan kita ingin memeriksa elemen-elemen yang salah dalam sebuah narasi yang unik. Algoritma brute force melewati semua pasangan elemen dalam waktu $O(n^2)$:

```
bool ok = true;
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        if (array[i] == array[j]) ok = false;
    }
}
```

Namun, kita dapat menyelesaikan masalah dalam waktu *O(n log n*) dengan mengurutkan array terlebih dahulu. Kemudian, jika ada elemen yang sama, elemen-elemen tersebut bersebelahan dalam larik terurut, sehingga mudah ditemukan dalam waktu *O(n):*

```
bool ok = true;
sort(array, array+n);
for (int i = 0; i < n-1; i++) {
    if (array[i] == array[i+1]) ok = false;
}
```

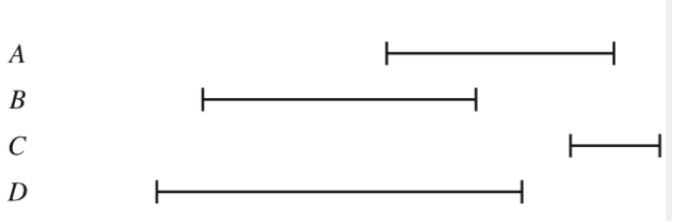
Beberapa masalah lain dapat diselesaikan dengan cara yang sama dalam waktu *O(n log n),* seperti menghitung jumlah elemen yang berbeda, menemukan elemen yang paling sering, dan menemukan dua elemen yang perbedaannya minimum.

4.2.1 Algoritma Penyapuan Garis

Algoritma garis sapuan memodelkan masalah sebagai serangkaian peristiwa yang diproses dalam urutan yang diurutkan. Sebagai contoh, anggaplah ada sebuah restoran dan kita mengetahui waktu kedatangan dan keberangkatan semua pelanggan pada hari tertentu. Tugas kita adalah mencari tahu jumlah maksimum pelanggan yang mengunjungi restoran pada waktu yang sama. Sebagai contoh, Gambar 4.7 menunjukkan contoh masalah di mana ada empat pelanggan A, B, C, dan D. Dalam hal ini, jumlah maksimum pelanggan simultan adalah tiga antara kedatangan A dan kepergian B.

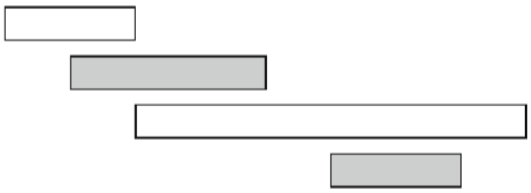
Untuk mengatasi masalah tersebut, kami membuat dua acara untuk setiap pelanggan: satu acara untuk kedatangan dan acara lainnya untuk pergi. Kemudian, kami mengurutkan peristiwa dan melewatinya sesuai dengan waktunya. Untuk menemukan jumlah maksimum pelanggan, kami memelihara konter yang nilainya meningkat saat pelanggan datang dan berkurang saat pelanggan pergi. Nilai

penghitung terbesar adalah jawaban dari masalah. Gambar4.8 menunjukkan kejadian dalam skenario contoh kita. Setiap pelanggan diberikan dua peristiwa: "+" menunjukkan pelanggan yang datang dan "−" menunjukkan pelanggan yang pergi. Algoritma yang dihasilkan bekerja dalam waktu O(n log n), karena pengurutan kejadian membutuhkan waktu O(n log n) dan bagian garis sapuan membutuhkan waktu *O(n).*

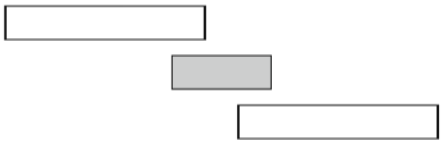


Gambar 4.7 Contoh masalah restoran

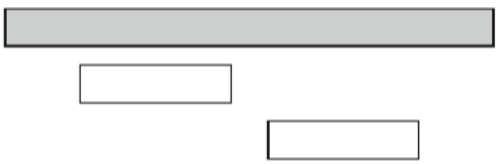
Gambar 4.8 Memecahkan masalah restoran menggunakan algoritma garis sapuan



Gambar 4.9 Contoh masalah penjadwalan dan solusi optimal dengan dua kejadian



Gambar 4.10 Jika kita memilih kejadian yang pendek, kita hanya dapat memilih satu kejadian, tetapi kita dapat memilih kedua kejadian yang panjang



Gambar 4.11 Jika kita memilih kejadian pertama, kita tidak dapat memilih kejadian lain, tetapi kita dapat memilih dua kejadian lainnya

4.2.2 Penjadwalan Acara

Banyak masalah penjadwalan dapat diselesaikan dengan menyortir data input dan kemudian menggunakan strategi serakah untuk membangun solusi. Algoritma serakah selalu membuat pilihan yang terlihat terbaik saat ini dan tidak pernah mengambil kembali pilihannya. Sebagai contoh, perhatikan masalah berikut:

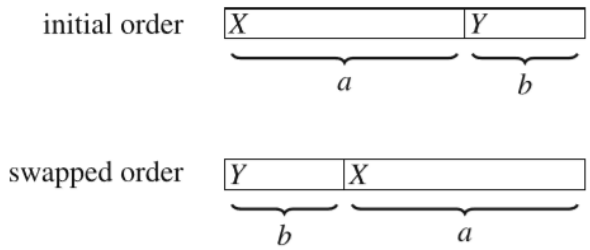
Mengingat peristiwa dengan waktu mulai dan berakhirnya, temukan jadwal yang mencakup sebanyak mungkin peristiwa. Misalnya, Gambar 4.9 menunjukkan contoh masalah di mana solusi optimal adalah memilih dua kejadian. Dalam masalah ini, ada beberapa cara bagaimana kita bisa mengurutkan data input.

Salah satu strateginya adalah mengurutkan acara menurut panjangnya dan memilih acara sesingkat mungkin. Namun, strategi ini tidak selalu berhasil, seperti yang ditunjukkan pada Gambar 4.10. Kemudian, ide lain adalah mengurutkan peristiwa menurut waktu mulainya dan selalu memilih kemungkinan peristiwa berikutnya yang dimulai sedini mungkin. Namun, kita juga dapat menemukan contoh tandingan untuk strategi ini, yang ditunjukkan pada Gambar 4.11. Ide ketiga adalah mengurutkan peristiwa menurut waktu berakhirnya dan selalu memilih kemungkinan peristiwa berikutnya yang berakhir sedini mungkin. Ternyata algoritma ini selalu menghasilkan solusi yang optimal. Untuk membenarkan hal ini, pertimbangkan apa yang terjadi jika pertama-tama kita memilih suatu peristiwa yang berakhir lebih lambat daripada peristiwa yang berakhir sedini mungkin. Sekarang, kita akan memiliki paling banyak jumlah pilihan yang sama yang tersisa bagaimana kita dapat memilih acara berikutnya. Oleh karena itu, memilih acara yang berakhir kemudian tidak akan pernah bisa menghasilkan solusi yang lebih baik, dan algoritma serakah benar.

4.2.3 Tugas dan Tenggat Waktu *(Deadline)*

Akhirnya, pertimbangkan masalah di mana kita memberikan tugas dengan durasi dan tenggat waktu dan tugas kita adalah memilih perintah untuk melakukan tugas. Untuk setiap tugas, kita memperoleh *d* - *x* poin di mana *d* adalah tenggat waktu tugas dan *x* adalah saat kita menyelesaikan tugas. Berapakah skor total terbesar yang dapat kita peroleh?

Gambar 4.12 Jadwal optimal untuk tugas



Gambar 4.13 Memperbaiki solusi dengan menukar tugas X dan Y

Sebagai contoh, anggaplah bahwa tugas-tugasnya adalah sebagai berikut:

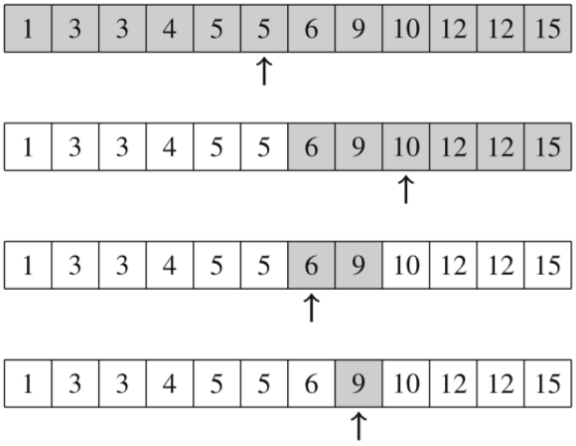
Deadline durasi tugas		
A	4	2
B	3	10
C	2	8
D	4	15

Gambar 4.12 menunjukkan jadwal optimal untuk tugas-tugas dalam skenario contoh kami. Dengan menggunakan jadwal ini, *C* menghasilkan 6 poin, *B* menghasilkan 5 poin, *A* menghasilkan −*7* poin, dan *D* menghasilkan 2 poin, sehingga skor totalnya adalah 6. Ternyata solusi optimal untuk masalah tidak bergantung pada tenggat waktu di semua, tetapi strategi serakah yang benar adalah dengan hanya melakukan tugas yang diurutkan berdasarkan durasinya dalam urutan yang meningkat.

Alasan untuk ini adalah jika kita pernah melakukan dua tugas satu demi satu sehingga tugas pertama memakan waktu lebih lama daripada tugas kedua, kita dapat memperoleh solusi yang lebih baik jika kita menukar tugas. Misalnya, pada Gambar 4.13, ada dua tugas *X* dan *Y* dengan durasi *a* dan *b.* Awalnya, *X* dijadwalkan sebelum Y. Namun, karena $a > b$, tugas harus ditukar. Sekarang *X* memberi *b* poin lebih sedikit dan *Y* memberi poin lebih banyak, sehingga skor total meningkat $a-b > 0$. Jadi, dalam solusi optimal, tugas yang lebih pendek harus selalu datang sebelum tugas yang lebih panjang, dan tugas harus diurutkan berdasarkan durasi mereka.

4.3 Pencarian Biner

Pencarian biner adalah algoritma waktu O(log n) yang dapat digunakan, misalnya, untuk memeriksa secara efisien apakah array yang diurutkan berisi elemen tertentu. Pada bagian ini, pertama-tama kita fokus pada implementasi pencarian biner, dan setelah itu, kita akan melihat bagaimana pencarian biner dapat digunakan untuk menemukan solusi optimal untuk masalah.



Gambar 4.14 Cara tradisional untuk mengimplementasikan pencarian biner.

Pada setiap langkah kami memeriksa elemen tengah dari subarray aktif dan melanjutkan ke bagian kiri atau kanan

4.3.2 Menerapkan Pencarian

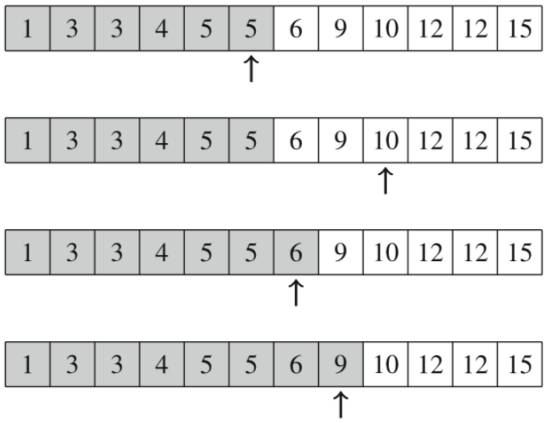
Misalkan kita diberikan array yang diurutkan dari n elemen dan kita ingin memeriksa apakah array tersebut berisi elemen dengan nilai target x. Selanjutnya kita membahas dua cara untuk mengimplementasikan algoritma pencarian biner untuk masalah ini.

Metode Pertama

Cara paling umum untuk mengimplementasikan pencarian biner mirip dengan mencari kata dalam kamus.[5] Pencarian mempertahankan subarray aktif dalam array, yang awalnya berisi semua elemen array. Kemudian, sejumlah langkah dilakukan, yang masing-masing membagi dua rentang pencarian. Pada setiap langkah, pencarian memeriksa elemen tengah dari subarray aktif. Jika elemen tengah memiliki nilai target, pencarian dihentikan. Jika tidak, pencarian secara rekursif berlanjut ke kiri atau kanan setengah dari subarray, tergantung pada nilai elemen tengah. Sebagai contoh, Gambar 4.14 menunjukkan bagaimana elemen dengan nilai 9 ditemukan dalam array. Pencarian dapat diimplementasikan sebagai berikut:

```
int a = 0, b = n-1;
while (a <= b) {
        int k = (a+b)/2;
        if (array[k] == x) {
        //xfound at index k
        }
        if (array[k] < x) a = k+1;
        else b = k-1;
}
```

Dalam implementasi ini, range dari subarray yang aktif adalah *a ...b*, dan range awalnya adalah *0...n−1.* Algoritma membagi dua ukuran subarray pada setiap langkah, sehingga kompleksitas waktu adalah *O(log n).*



Gambar 4.15 Cara alternatif untuk mengimplementasikan pencarian biner. Kami memindai array dari kiri ke kanan melompati elemen

Metode Kedua

Cara lain untuk mengimplementasikan pencarian biner adalah dengan menelusuri larik dari kiri ke kanan membuat lompatan. Panjang lompatan awal adalah n/2, dan panjang lompatan dibagi dua pada setiap putaran: pertama n/4, kemudian n/8, kemudian n/16, dst., hingga akhirnya panjangnya menjadi 1. Pada setiap putaran, kita membuat melompat sampai kita akan berakhir di luar array atau di elemen yang nilainya melebihi nilai target. Setelah lompatan, elemen yang diinginkan telah ditemukan atau kita tahu bahwa itu tidak muncul dalam array. Gambar 4.15

[5] Beberapa orang, termasuk penulis buku ini, masih menggunakan kamus cetak. Contoh lain adalah menemukan nomor telepon di buku telepon cetak, yang bahkan lebih usang.

mengilustrasikan teknik dalam skenario contoh kita. Kode berikut mengimplementasikan pencarian:

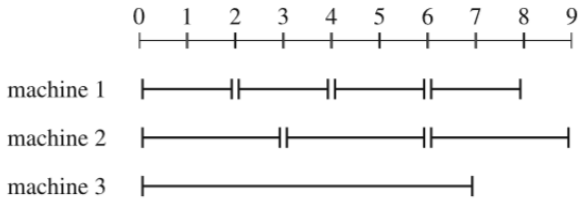
```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
        while (k+b < n && array[k+b] <= x) k += b;
}
if (array[k] == x) {
        // xfound at index k
}
```

Selama pencarian, variabel b berisi panjang lompatan saat ini. Kompleksitas waktu dari algoritma ini adalah O(log n), karena kode dalam while loop dilakukan paling banyak dua kali untuk setiap panjang lompatan.

4.3.2 Menemukan Solusi Optimal

Misalkan kita sedang memecahkan masalah dan memiliki fungsi valid(x) yang mengembalikan nilai true jika x adalah solusi yang valid dan salah jika sebaliknya. Selain itu, kita tahu bahwa valid(x) salah jika $x < k$ dan benar jika $x \geq k$. Dalam situasi ini, kita dapat menggunakan pencarian biner untuk menemukan nilai k secara efisien. Idenya adalah untuk mencari biner untuk nilai x terbesar yang valid(x) salah. Jadi, nilai berikutnya $k = x + 1$ adalah nilai terkecil yang memungkinkan untuk valid(k) benar. Pencarian dapat diimplementasikan sebagai berikut:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
        while (!valid(x+b)) x += b;
}
int k = x+1;
```



Gambar 4.16 Jadwal pemrosesan yang optimal: mesin 1 memproses empat pekerjaan, mesin 2 memproses tiga pekerjaan, dan mesin 3 memproses satu pekerjaan

Panjang lompatan awal z harus menjadi batas atas untuk jawabannya, yaitu, nilai apa pun yang kita pasti tahu bahwa valid(z) adalah benar. Algoritme memanggil fungsi yang valid *O(log z)* kali, sehingga waktu berjalan tergantung pada fungsi yang valid. Misalnya, jika fungsi bekerja dalam waktu *O(n),* waktu berjalannya adalah *O(n log z).*

Contoh Pertimbangkan masalah di mana tugas kita adalah memproses *k* pekerjaan menggunakan *n* mesin. Setiap mesin i diberi bilangan bulat *pi:* waktu untuk memproses satu pekerjaan. Berapa waktu minimum untuk memproses semua pekerjaan? Sebagai contoh, misalkan k =8, n =3 dan waktu pemrosesan adalah p1 =2,p 2 =3, dan p3 =7. Dalam hal ini, total waktu pemrosesan minimum adalah 9, dengan mengikuti jadwal pada Gambar 4.16. Biarkan valid(x) menjadi fungsi yang mencari tahu

apakah mungkin untuk memproses semua pekerjaan menggunakan paling banyak x unit waktu.

Dalam skenario contoh kita, jelas valid(9) adalah benar, karena kita dapat mengikuti jadwal pada Gambar 4.16. Di sisi lain, valid(8) harus salah, karena waktu pemrosesan minimum adalah 9. Menghitung nilai valid(x) itu mudah, karena setiap mesin i dapat memproses paling banyak pekerjaan x/pi dalam x unit waktu. Jadi, jika jumlah semua nilai x/pi adalah k atau lebih, x adalah solusi yang valid. Kemudian, kita dapat menggunakan pencarian biner untuk menemukan nilai minimum x yang valid(x) adalah benar. Seberapa efisien algoritma yang dihasilkan? Fungsi yang valid membutuhkan waktu *O(n),* sehingga algoritma bekerja dalam waktu *O(n log z),* di mana *z* adalah batas atas untuk jawabannya. Satu nilai yang mungkin untuk *z* adalah *kp1* yang sesuai dengan solusi di mana hanya mesin pertama yang digunakan untuk memproses semua pekerjaan. Ini pasti batas atas yang valid.

BAB 5
STRUKTUR DATA

Bab ini memperkenalkan struktur data terpenting dari pustaka standar C++. Dalam pemrograman kompetitif, sangat penting untuk mengetahui struktur data mana yang tersedia di perpustakaan standar dan bagaimana menggunakannya. Ini sering menghemat banyak waktu ketika mengimplementasikan suatu algoritma. Bagian pertama menjelaskan struktur vektor yang merupakan array dinamis yang efisien. Setelah ini, kita akan fokus menggunakan iterator dan rentang dengan struktur data, dan secara singkat membahas deque, stack, dan antrian. Bagian selanjutnya membahas set, Maps dan antrian prioritas. Struktur data tersebut sering digunakan sebagai blok penyusun algoritma yang efisien, karena memungkinkan kita untuk memelihara struktur dinamis yang mendukung pencarian dan pembaruan yang efisien. Bagian terakhir menunjukkan beberapa hasil tentang efisiensi struktur data dalam praktik. Seperti yang akan kita lihat, ada perbedaan kinerja penting yang tidak dapat dideteksi hanya dengan melihat kompleksitas waktu.

5.1 Array Dinamis

Dalam C++, array biasa adalah struktur berukuran tetap, dan tidak mungkin mengubah ukuran array setelah membuatnya. Misalnya, kode berikut membuat array yang berisi n nilai integer:

```
int array[n];
```

Larik dinamis adalah larik yang ukurannya dapat diubah selama eksekusi program. Pustaka standar C++ menyediakan beberapa larik dinamis, yang paling berguna adalah struktur vektornya.

5.1.1 Vektor

Vektor adalah larik dinamis yang memungkinkan kita menambahkan dan menghapus elemen di akhir struktur secara efisien. Misalnya, kode berikut membuat vektor kosong dan menambahkan tiga elemen ke dalamnya:

```
vector<int> v; v.push_back(3);
//[3]
v.push_back(2);
//[3,2]
v.push_back(5);
//[3,2,5]
```

Kemudian, elemen dapat diakses seperti dalam array biasa:

```
cout << v[0] << "\n";
//3
cout << v[1] << "\n";
//2
cout << v[2] << "\n";
//5
```

Cara lain untuk membuat vektor adalah dengan memberikan daftar elemennya:

```
vector<int> v = {2,4,2,5,1};
```

Kami juga dapat memberikan jumlah elemen dan nilai awalnya:

```
vector<int> a(8); //size 8, initial value 0
vector<int> b(8,2); //size 8, initial value 2
```

Ukuran fungsi mengembalikan jumlah elemen dalam vektor. Misalnya, kode berikut mengiterasi melalui vektor dan mencetak elemen-elemennya:

```
for (int i = 0; i < v.size(); i++) {
        cout << v[i] << "\n";
}
```

Cara yang lebih singkat untuk melakukan iterasi melalui vektor adalah sebagai berikut:

```
for (auto x : v) {
        cout << x << "\n";
}
```

Fungsi kembali mengembalikan elemen terakhir dari vektor, dan fungsi pop_back menghapus elemen terakhir:

```
vector<int> v = {2,4,2,5,1};
cout << v.back() <<"\n";//1
v.pop_back();
cout << v.back() <<"\n";//5
```

Vektor diimplementasikan sehingga operasi push_back dan pop_back bekerja rata-rata dalam waktu *O(1).* Dalam prakteknya, menggunakan vektor hampir secepat menggunakan array biasa.

5.1.2 Iterator dan Rentang

Iterator adalah variabel yang menunjuk ke elemen struktur data. Iterator start menunjuk ke elemen pertama dari struktur data, dan iterator end menunjuk ke posisi setelah elemen terakhir. Misalnya, situasinya dapat terlihat sebagai berikut dalam vektor v yang terdiri dari delapan elemen:

```
[ 5, 2, 3, 1, 2, 5, 7, 1 ]
  ↑                    ↑
  v.begin()        v.end()
```

Perhatikan asimetri dalam iterator: begin() menunjuk ke elemen dalam struktur data, sedangkan end() menunjuk di luar struktur data. Rentang adalah urutan elemen berurutan dalam struktur data. Cara biasa untuk menentukan rentang adalah dengan memberikan iterator ke elemen pertamanya dan posisi setelah elemen terakhirnya. Secara khusus, iterator begin() dan end() mendefinisikan rentang yang berisi semua elemen dalam struktur data. Fungsi pustaka standar C++ biasanya beroperasi dengan rentang. Sebagai contoh, kode berikut pertama-tama mengurutkan sebuah vektor, kemudian membalik urutan elemen-elemennya, dan akhirnya mengocok elemen-elemennya.

```
sort(v.begin(),v.end());
reverse(v.begin(),v.end()); random_shuffle(v.begin(),v.end());
```

Elemen yang ditunjuk oleh iterator dapat diakses menggunakan sintaks *. Misalnya, kode berikut mencetak elemen pertama dari sebuah vektor:

```
cout << *v.begin() << "\n";
```

Untuk memberikan contoh yang lebih berguna, lower_bound memberikan iterator ke elemen pertama dalam rentang terurut yang nilainya paling sedikit x, dan upper_bound memberikan iterator ke elemen pertama yang nilainya lebih besar dari x:

```
vector<int> v = {2,3,3,5,7,8,8,8};
auto a = lower_bound(v.begin(),v.end(),5);
auto b = upper_bound(v.begin(),v.end(),5);
cout << *a <<" "<<*b<<"\n";//57
```

Perhatikan bahwa fungsi di atas hanya berfungsi dengan benar ketika rentang yang diberikan diurutkan. Fungsi menggunakan pencarian biner dan menemukan elemen yang diminta dalam waktu logaritmik.

Jika tidak ada elemen seperti itu, fungsi mengembalikan iterator ke elemen setelah elemen terakhir dalam rentang. Pustaka standar C++ berisi sejumlah besar fungsi berguna yang perlu ditelusuri. Misalnya, kode berikut membuat vektor yang berisi elemen unik dari vektor asli dalam urutan yang diurutkan:

```
sort(v.begin(),v.end());
v.erase(unique(v.begin(),v.end()),v.end());
```

5.1.3 Struktur Lainnya

Deque adalah larik dinamis yang dapat dimanipulasi secara efisien di kedua ujung struktur. Seperti vektor, deque menyediakan fungsi push_back dan pop_back, tetapi juga menyediakan fungsi push_front dan pop_front yang tidak tersedia dalam vektor. Sebuah deque dapat digunakan sebagai berikut:

```
deque<int> d;
d.push_back(5);//[5]
d.push_back(2);//[5,2]
d.push_front(3);//[3,5,2]
d.pop_back();//[3,5]
d.pop_front();//[5]
```

Operasi deque juga bekerja dalam waktu rata-rata O(1). Namun, deques memiliki faktor konstan yang lebih besar daripada vektor, jadi deques harus digunakan hanya jika ada kebutuhan untuk memanipulasi kedua ujung array. C++ juga menyediakan dua struktur data khusus yang, secara default, berdasarkan deque. Tumpukan memiliki fungsi push dan pop untuk memasukkan dan menghapus elemen di akhir struktur dan fungsi atas yang mengambil elemen terakhir:

```
stack<int> s;
s.push(2); //[2]
s.push(5); //[2,5]
cout << s.top() << "\n";//5
s.pop();//[2]
cout << s.top() << "\n";//2
```

Kemudian, dalam antrian, elemen disisipkan di ujung struktur dan dikeluarkan dari bagian depan struktur. Kedua fungsi depan dan belakang disediakan untuk mengakses elemen pertama dan terakhir.

```
queue<int> q;
q.push(2); //[2]
q.push(5); //[2,5]
cout << q.front() << "\n";//2
q.pop();//[5]
cout << q.back() << "\n";//5
```

5.2 Tetapkan Struktur

Aset adalah struktur data yang memelihara kumpulan elemen. Operasi dasar himpunan adalah penyisipan elemen, pencarian, dan penghapusan. Set diimplementasikan sehingga semua operasi di atas menjadi efisien, yang sering memungkinkan kita untuk meningkatkan waktu berjalan dari algoritma menggunakan set.

5.2.1 *Sets and Multiset*

Pustaka standar C++ berisi dua set struktur:

- *set* didasarkan pada pohon pencarian biner seimbang dan operasinya bekerja dalam waktu *O(log n).*
- unordered_setis berdasarkan tabel *hash* dan operasinya bekerja, rata-rata, [6] dalam waktu O(1).

Kedua struktur tersebut efisien, dan seringkali salah satunya dapat digunakan. Karena mereka digunakan dengan cara yang sama, kami fokus pada struktur himpunan dalam contoh berikut. Kode berikut membuat himpunan yang berisi bilangan bulat dan menunjukkan beberapa operasinya. Sisipan fungsi menambahkan elemen ke himpunan, hitungan fungsi mengembalikan jumlah kemunculan elemen dalam himpunan, dan fungsi menghapus menghapus elemen dari himpunan.

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << "\n";//1
cout << s.count(4) << "\n";//0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n";//0
cout << s.count(4) << "\n";//1
```

Sifat penting dari himpunan adalah bahwa semua elemennya berbeda. Jadi, hitungan fungsi selalu mengembalikan 0 (elemen tidak ada dalam himpunan) atau 1 (elemen ada dalam himpunan), dan sisipan fungsi tidak pernah menambahkan elemen ke himpunan jika sudah ada. Kode berikut menggambarkan hal ini:

```
set<int> s;
s.insert(3);
s.insert(3);
s.insert(3);
cout << s.count(3)<<"\n";//1
```

Himpunan dapat digunakan sebagian besar seperti vektor, tetapi tidak mungkin untuk mengakses elemen menggunakan notasi []. Kode berikut mencetak jumlah elemen dalam satu set dan kemudian iterasi melalui elemen:

```
cout << s.size() << "\n";
for (auto x : s) {
cout << x << "\n";
}
```

Fungsi find(x) mengembalikan iterator yang menunjuk ke elemen yang nilainya x. Namun, jika set tidak berisi x, iterator akan menjadi end().

```
auto it = s.find(x);
if (it == s.end()) {
        //x is not found
}
```

[6] Kompleksitas waktu kasus terburuk dari operasi adalah O(n), tetapi ini sangat tidak mungkin terjadi.

Set yang dipesan

Perbedaan utama antara dua struktur himpunan C++ adalah bahwa himpunan diurutkan, sedangkan unordered_set tidak. Jadi, jika kita ingin mempertahankan urutan elemen, kita harus menggunakan struktur himpunan. Misalnya, pertimbangkan masalah menemukan nilai terkecil dan terbesar dalam suatu himpunan. Untuk melakukan ini secara efisien, kita perlu menggunakan struktur himpunan. Karena elemen diurutkan, kita dapat menemukan nilai terkecil dan terbesar sebagai berikut:

```
auto first = s.begin();
auto last = s.end(); last--;
cout <<*first<<" "<<*last<<"\n";
```

Perhatikan bahwa karena end() menunjuk ke elemen setelah elemen terakhir, kita harus mengurangi iterator satu per satu. Struktur himpunan juga menyediakan fungsi lower_bound(x) dan upper_bound(x) yang mengembalikan iterator ke elemen terkecil dalam himpunan yang nilainya paling sedikit atau lebih besar dari x, masing-masing. Dalam kedua fungsi tersebut, jika elemen yang diminta tidak ada, nilai kembaliannya adalah end().

```
cout << *s.lower_bound(x) << "\n";
cout << *s.upper_bound(x) << "\n";
```

Multiset

Multiset adalah himpunan yang dapat memiliki beberapa salinan dengan nilai yang sama. C++ memiliki struktur multiset dan unordered_multiset yang menyerupai set dan unordered_set. Misalnya, kode berikut menambahkan tiga salinan nilai 5 ke multiset.

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n";// 3
```

Fungsi erase menghapus semua salinan nilai dari multiset:

```
s.erase(5);
cout << s.count(5) << "\n"; / / 0
```

Seringkali, hanya satu nilai yang harus dihapus, yang dapat dilakukan sebagai berikut:

```
s.erase(s.find(5));
cout << s.count(5) << "\n"; //2
```

Perhatikan bahwa fungsi count and erase memiliki faktor O(k) tambahan di mana k adalah jumlah elemen yang dihitung/dihapus. Secara khusus, tidak efisien untuk menghitung jumlah salinan suatu nilai dalam multiset menggunakan fungsi count.

5.2.2 Peta/Maps

Maps adalah himpunan yang terdiri dari pasangan nilai kunci. Peta juga dapat dilihat sebagai larik umum. Sementara kunci dalam array biasa selalu bilangan bulat berturut-turut *0,1,...,n−1,* di mana *n* adalah ukuran array, kunci dalam Maps dapat berupa tipe data apa pun dan tidak harus nilai berturut-turut.

Pustaka standar C++ berisi dua struktur Maps yang sesuai dengan struktur yang ditetapkan: Maps didasarkan pada pohon pencarian biner seimbang dan mengakses elemen membutuhkan waktu *O(log n),* sedangkan unordered_map menggunakan hashing dan mengakses elemen membutuhkan waktu rata-rata *O(1)* . Kode berikut membuat Maps yang kuncinya adalah string dan nilainya adalah bilangan bulat:

```
map<string,int> m;
m["monkey"] = 4; m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n";//3
```

Jika nilai kunci diminta tetapi Maps tidak memuatnya, kunci tersebut secara otomatis ditambahkan ke Maps dengan nilai default. Misalnya, dalam kode berikut, kunci "aybabtu" dengan nilai 0 ditambahkan ke Maps.

```
map<string,int> m;
cout << m["aybabtu"] << "\n"; / / 0
```

Hitungan fungsi memeriksa apakah ada kunci di Maps:

```
if (m.count("aybabtu")) {
//key exists
}
```

Kemudian, kode berikut mencetak semua kunci dan nilai dalam Maps:

```
for (auto x : m) {
    cout << x.first << " " << x.second << "\n";
}
```

5.2.3 Antrian Prioritas

Antrian prioritas adalah multiset yang mendukung penyisipan elemen dan, tergantung pada jenis antrian, pengambilan dan pemindahan elemen minimum atau maksimum. Penyisipan dan penghapusan membutuhkan waktu O(log n), dan pengambilan membutuhkan waktu O(1). Antrian prioritas biasanya didasarkan pada struktur heap, yang merupakan pohon biner khusus. Sementara multiset menyediakan semua operasi antrian prioritas dan lebih banyak lagi, keuntungan menggunakan antrian prioritas adalah memiliki faktor konstan yang lebih kecil. Jadi, jika kita hanya perlu menemukan elemen minimum atau maksimum secara efisien, adalah ide yang baik untuk menggunakan antrian prioritas daripada set atau multiset.

Secara default, elemen dalam antrian prioritas C++ diurutkan dalam urutan menurun, dan dimungkinkan untuk menemukan dan menghapus elemen terbesar dalam antrian. Kode berikut menggambarkan hal ini:

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n";//7
q.pop();
cout << q.top() << "\n";//5
q.pop();
q.push(6);
cout << q.top() << "\n";//6
q.pop();
```

Jika kita ingin membuat antrian prioritas yang mendukung pencarian dan penghapusan elemen terkecil, kita dapat melakukannya sebagai berikut:

```
priority_queue<int,vector<int>,greater<int>> q;
```

5.2.4 Perangkat Berbasis Kebijakan

Kompiler g++ juga menyediakan beberapa struktur data yang bukan bagian dari pustaka standar C++. Struktur seperti ini disebut struktur berbasis kebijakan. Untuk menggunakan struktur ini, baris berikut harus ditambahkan ke kode:

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
```

Setelah ini, kita dapat mendefinisikan struktur data indexed_set yang seperti set tetapi dapat diindeks seperti array. Definisi nilai int adalah sebagai berikut:

```
typedef tree<int,null_type,less<int>,rb_tree_tag,
        tree_order_statistics_node_update> indexed_set;
```

Kemudian, kita dapat membuat satu set sebagai berikut:

```
indexed_set s;
s.insert(2);
s.insert(3);
s.insert(7);
s.insert(9);
```

Keistimewaan set ini adalah kita memiliki akses ke indeks yang dimiliki elemen dalam array yang diurutkan. Fungsi find_by_order mengembalikan iterator ke elemen pada posisi tertentu:

```
auto x = s.find_by_order(2);
cout << *x << "\n"; //7
```

Kemudian, fungsi order_of_key mengembalikan posisi elemen yang diberikan:

```
cout << s.order_of_key(7) << "\n";//2
```

Jika elemen tidak muncul di himpunan, kita mendapatkan posisi yang akan dimiliki elemen di himpunan:

```
cout << s.order_of_key(6) << "\n";//2
cout << s.order_of_key(8) << "\n";//3
```

Kedua fungsi tersebut bekerja dalam waktu logaritmik.

5.3 Eksperimen

Pada bagian ini, kami menyajikan beberapa hasil mengenai efisiensi praktis dari struktur data yang disajikan dalam bab ini. Sementara kompleksitas waktu adalah alat yang hebat, mereka tidak selalu mengatakan seluruh kebenaran tentang efisiensi, jadi ada baiknya juga melakukan eksperimen dengan implementasi nyata dan kumpulan data.

5.3.1 Set Versus Penyortiran

Banyak masalah dapat diselesaikan dengan menggunakan set atau pengurutan. Penting untuk disadari bahwa algoritma yang menggunakan pengurutan biasanya jauh lebih cepat, bahkan jika ini tidak terbukti hanya dengan melihat kompleksitas waktu. Sebagai contoh, pertimbangkan masalah menghitung jumlah elemen unik dalam vektor. Salah satu cara untuk memecahkan masalah adalah dengan menambahkan semua elemen ke himpunan dan mengembalikan ukuran himpunan. Karena tidak diperlukan untuk mempertahankan urutan elemen, kita dapat menggunakan set atau unordered_set. Kemudian, cara lain untuk menyelesaikan masalah adalah dengan mengurutkan vektor terlebih dahulu dan kemudian menelusuri elemen-elemennya. Sangat mudah untuk menghitung jumlah elemen unik setelah mengurutkan vektor. Tabel 5.1 menunjukkan hasil percobaan di mana algoritma di atas diuji menggunakan vektor acak dari nilai int. Ternyata algoritma unordered_set sekitar dua kali lebih cepat dari algoritma set, dan algoritma pengurutan lebih dari sepuluh kali lebih cepat

daripada algoritma set. Perhatikan bahwa baik algoritma set dan algoritma pengurutan bekerja dalam waktu *O(n log n*); masih yang terakhir jauh lebih cepat. Alasan untuk ini adalah bahwa pengurutan adalah operasi sederhana, sedangkan pohon pencarian biner seimbang yang digunakan dalam set adalah struktur data yang kompleks.

Tabel 5.1 Hasil percobaan di mana jumlah elemen unik dalam vektor dihitung.

Ukuran input *n*	Set (s)	unordered_map (s)	Sorting
10^6	0.65	0.34	0.11
2 x 10^6	1.50	0.76	0.18
4 x10^6	3.38	1.63	0.33
8 x 10^6	7.57	3.45	0.68
16x10^6	17.35	.18	1.38

Dua algoritma pertama menyisipkan elemen ke struktur yang ditetapkan, sedangkan algoritma terakhir mengurutkan vektor dan memeriksa elemen berurutan

Tabel 5.2 Hasil percobaan di mana nilai yang paling sering dalam suatu vektor ditentukan.

Ukuran input *n*	Maps (s)	unordered_map (s)	Array (s)
10^6	0.55	0.23	0.01
2 x 10^6	1.13	0.39	0.03
4 x10^6	2.34	0.73	0.03
8 x 10^6	4.68	1.46	0.06
16x10^6	9.57	2.83	0.11

Dua algoritma pertama menggunakan struktur Maps, dan algoritma terakhir menggunakan array biasa

5.3.2 Peta (Map) Versus Array

Maps adalah struktur yang nyaman dibandingkan dengan array, karena indeks apa pun dapat digunakan, tetapi mereka juga memiliki faktor konstan yang besar. Dalam percobaan kami berikutnya, kami membuat vektor bilangan bulat acak antara 1 dan 10^6 dan kemudian menentukan nilai yang paling sering dengan menghitung jumlah setiap elemen. Pertama kami menggunakan Maps, tetapi karena batas atas 10^6 cukup kecil, kami juga dapat menggunakan array. Tabel 5.2 menunjukkan hasil percobaan. Sementara unordered_map sekitar tiga kali lebih cepat dari Maps, array hampir seratus kali lebih cepat. Dengan demikian, array harus digunakan bila memungkinkan, bukan Maps. Khususnya, perhatikan bahwa sementara unordered_map menyediakan operasi waktu O(1), ada faktor konstan besar yang tersembunyi dalam struktur data.

5.3.3 Antrian Prioritas Versus Multiset

Apakah antrian prioritas benar-benar lebih cepat daripada multiset? Untuk mengetahui hal ini, kami melakukan percobaan lain di mana kami membuat dua vektor dari n bilangan int acak. Pertama, kami menambahkan semua elemen dari vektor pertama ke dalam struktur data. Kemudian, kami melewati vektor kedua dan berulang kali menghapus elemen terkecil dari struktur data dan menambahkan elemen baru ke dalamnya. Tabel 5.3 menunjukkan hasil percobaan. Ternyata dalam masalah ini antrian prioritas sekitar lima kali lebih cepat daripada multiset.

Tabel 5.3 Hasil percobaan di mana elemen ditambahkan dan dihapus menggunakan multiset dan antrian prioritas

Ukuran input *n*	**Multiset (s)**	**Priority_queue (s)**
10^6	1.17	0.19
2×10^6	2.77	0.41
4×10^6	6.10	1.05
8×10^6	13.39	2.52
16×10^6	30.39	5.95

BAB 6
PEMROGRAMAN DINAMIS

Pemrograman dinamis adalah teknik desain algoritma yang dapat digunakan untuk menemukan solusi optimal untuk masalah dan menghitung jumlah solusi. Bab ini adalah pengantar pemrograman dinamis, dan teknik ini akan digunakan berkali-kali nanti dalam buku ini ketika merancang algoritma. Bagian pertama membahas elemen dasar pemrograman dinamis dalam konteks masalah penukaran koin. Dalam masalah ini kita diberikan satu set nilai koin dan tugas kita adalah membuat sejumlah uang dengan menggunakan koin sesedikit mungkin. Ada algoritma serakah sederhana untuk masalah ini, tetapi seperti yang akan kita lihat, itu tidak selalu menghasilkan solusi yang optimal. Namun, dengan menggunakan pemrograman dinamis, kita dapat membuat algoritma yang efisien yang selalu menemukan solusi optimal. Bagian selanjutnyamenyajikan pilihan masalah yang menunjukkan beberapa kemungkinan pemrograman dinamis. Soal-soal tersebut meliputi menentukan barisan kenaikan terpanjang dalam suatu larik, menemukan jalur optimal dalam kisi dua dimensi, dan menghasilkan semua jumlah bobot yang mungkin dalam masalah knapsack.

6.1 Konsep Dasar

Pada bagian ini, kita membahas konsep dasar pemrograman dinamis dalam konteks masalah perubahan koin. Pertama-tama kita menyajikan algoritma yang disepakati untuk masalah tersebut, yang tidak selalu menghasilkan solusi optimal. Setelah ini, kami menunjukkan bagaimana masalah dapat diselesaikan secara efisien menggunakan pemrograman dinamis.

6.1.1 Ketika Keserakahan Gagal

Misalkan kita diberi satu set nilai koin coin ={ c_1,c_2,...,c_k} dan jumlah uang n yang ditargetkan, dan kita diminta untuk membuat jumlah n menggunakan koin sesedikit mungkin. Tidak ada batasan berapa kali kita dapat menggunakan setiap nilai koin. Misalnya, jika koin={ 1,2,5}dan n =12, solusi optimalnya adalah 5+5+2=12, yang membutuhkan tiga koin. Ada algoritma serakah alami untuk memecahkan masalah: selalu pilih koin terbesar yang mungkin sehingga jumlah nilai koin tidak melebihi jumlah target. Misalnya, jika n = 12, pertama-tama kita memilih dua koin bernilai 5, dan kemudian satu koin bernilai 2, yang melengkapi penyelesaian. Ini terlihat seperti strategi yang masuk akal, tetapi apakah selalu optimal?

Ternyata strategi ini tidak selalu berhasil. Misalnya, jika koin = {1,3,4} dan n = 6, solusi optimal hanya memiliki dua koin (3+3 = 6) tetapi strategi serakah menghasilkan solusi dengan tiga koin (4+1+1 = 6). Contoh tandingan sederhana ini menunjukkan bahwa algoritma serakah tidak benar.[7] Bagaimana kita bisa menyelesaikan masalah? Tentu saja, kita dapat mencoba menemukan algoritma serakah lain, tetapi tidak ada strategi jelas lain yang dapat kita pertimbangkan. Kemungkinan lain adalah membuat algoritma brute force yang melewati semua cara yang mungkin untuk memilih koin. Algoritma seperti itu pasti akan memberikan hasil yang benar, tetapi akan sangat lambat pada input yang besar. Namun, dengan menggunakan pemrograman dinamis, kita dapat membuat algoritma yang hampir mirip dengan algoritma brute force tetapi

[7] Ini adalah pertanyaan yang menarik kapan tepatnya algoritma serakah bekerja. Pearson [24] menjelaskan algoritma yang efisien untuk menguji ini.

juga efisien. Dengan demikian, kita berdua dapat yakin bahwa algoritme itu benar dan menggunakannya untuk memproses input yang besar. Selanjutnya, kita dapat menggunakan teknik yang sama untuk memecahkan sejumlah besar masalah lain.

6.1.2 Menemukan Solusi Optimal

Untuk menggunakan pemrograman dinamis, kita harus merumuskan masalah secara rekursif sehingga solusi masalah dapat dihitung dari solusi ke sub masalah yang lebih kecil. Dalam masalah koin, masalah rekursif alami adalah menghitung nilai suatu fungsi. solve (x): berapa jumlah minimum koin yang diperlukan untuk membentuk jumlah x? Jelas, nilai fungsi bergantung pada nilai koin. Misalnya, If coin = {1,3,4}, nilai pertama dari fungsi tersebut adalah sebagai berikut:

solve(0) = 0
solve(1) = 1
solve(2) = 2
solve(3) = 1
solve(4) = 1
solve(5) = 2
solve(6) = 2
solve(7) = 2
solve(8) = 2
solve(9) = 3
solve(10) = 3

Misalnya, solve(10) = 3, karena paling sedikit diperlukan 3 koin untuk membentuk jumlah 10. Solusi optimalnya adalah 3+3+4=10. Sifat esensial dari solve adalah bahwa nilainya dapat dihitung secara rekursif dari nilai yang lebih kecil. Idenya adalah untuk fokus pada koin pertama yang kita pilih untuk dijumlahkan. Sebagai contoh, dalam skenario di atas, koin pertama dapat berupa 1, 3 atau 4. Jika pertama kita memilih koin 1, tugas yang tersisa adalah membentuk jumlah 9 menggunakan jumlah koin minimum, yang merupakan submasalah dari koin asli. masalah. Tentu saja, hal yang sama berlaku untuk koin 3 dan 4. Jadi, kita dapat menggunakan rumus rekursif berikut untuk menghitung jumlah minimum koin:

solve(x) = min(solve(x −1)+1,
solve(x −3)+1,
solve(x −4)+1).

Kasus dasar dari rekursi adalah solve(0) = 0, karena tidak diperlukan koin untuk membentuk jumlah kosong. Sebagai contoh,

solve(10) =solve(7)+1=solve(4)+2=solve(0)+3=3.

Sekarang kita siap untuk memberikan fungsi rekursif umum yang menghitung jumlah minimum koin yang diperlukan untuk membentuk jumlah x:

$$solve(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ minc \in coins\ solve(x\ -c) + 1\ x\ >\ 0 \end{cases}$$

Pertama, jika x < 0, nilainya tak berhingga, karena tidak mungkin membentuk jumlah uang negatif. Kemudian, jika x =0,nilainya nol, karena uang logam diperlukan untuk membentuk jumlah kosong. Akhirnya, jika x > 0, variabel c melewati semua kemungkinan bagaimana memilih koin pertama dari jumlah tersebut. Setelah fungsi rekursif yang memecahkan masalah telah ditemukan, kita dapat langsung mengimplementasikan solusi dalam C++ (konstanta INF menunjukkan tak terhingga):

```
int solve(int x) {
        if (x < 0)return INF;
        if (x == 0) return 0;
        int best = INF;
        for (auto c : coins) {
        best = min(best, solve(x-c)+1);
        }
        return best;
}
```

Namun, fungsi ini tidak efisien, karena mungkin ada banyak cara untuk membuat jumlah dan fungsi memeriksa semuanya. Untungnya, ternyata ada cara sederhana untuk membuat fungsi tersebut menjadi efisien.

Memoisasi

Ide kunci dalam pemrograman dinamis adalah memoisasi, yang berarti bahwa kita menyimpan setiap nilai fungsi dalam array secara langsung setelah menghitungnya. Kemudian, ketika nilai tersebut dibutuhkan lagi, nilai tersebut dapat diambil dari array tanpa panggilan rekursif. Untuk melakukan ini, kami membuat array

```
bool ready[N];
int value[N];
```

di mana ready[x] menunjukkan apakah nilai solve(x) telah dihitung, dan jika ya, nilai[x] berisi nilai ini. Konstanta N telah dipilih sehingga semua nilai yang dibutuhkan sesuai dengan array. Setelah ini, fungsi tersebut dapat diimplementasikan secara efisien sebagai berikut:

```
int solve(int x) {
if (x < 0)return INF;
if (x == 0) return 0;
if (ready[x]) return value[x];
int best = INF;
for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    ready[x] = true;
    value[x] = best;
    return best;
}
```

Fungsi menangani kasus dasar x < 0 dan x = 0 seperti sebelumnya. Kemudian memeriksa dari ready[x]jika solve(x) telah disimpan dalam nilai[x], dan jika ya, fungsi langsung mengembalikannya. Jika tidak, fungsi menghitung nilai solve(x) secara rekursif dan menyimpannya dalam nilai[x]. Fungsi ini bekerja dengan efisien, karena jawaban untuk setiap parameter x dihitung secara rekursif hanya satu kali. Setelah nilai penyelesaian (x) telah disimpan dalam nilai [x], fungsi tersebut dapat diambil secara efisien setiap kali fungsi dipanggil kembali dengan parameter x. Kompleksitas waktu dari algoritma adalah *O(nk),* di mana n adalah jumlah target dan k adalah jumlah koin.

Implementasi Berulang

Perhatikan bahwa kita juga dapat membangun nilai array secara iteratif menggunakan loop sebagai berikut:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
        value[x] = INF;
```

```
        for (auto c : coins) {
        if (x-c >= 0) {
        value[x] = min(value[x], value[x-c]+1);
        }
        }
}
```

Faktanya, sebagian besar programmer kompetitif lebih menyukai implementasi ini, karena lebih pendek dan memiliki faktor konstan yang lebih kecil. Mulai sekarang, kami juga menggunakan implementasi berulang dalam contoh kami. Namun, seringkali lebih mudah untuk memikirkan solusi pemrograman dinamis dalam hal fungsi rekursif.

Membangun Solusi

Kadang-kadang kita diminta untuk menemukan nilai solusi optimal dan memberikan contoh bagaimana solusi tersebut dapat dibangun. Untuk membangun solusi optimal dalam masalah koin, kita dapat mendeklarasikan array baru yang menunjukkan untuk setiap jumlah uang koin pertama dalam solusi optimal:

```
int first[N];
```

Kemudian, kita dapat memodifikasi algoritma sebagai berikut:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
        value[x] = INF;
        for (auto c : coins) {
        if (x-c >= 0 && value[x-c]+1 <
        value[x]) { value[x] = value[x-c]+1;
        first[x] = c;
        }
        }
}
```

Setelah ini, kode berikut mencetak koin yang muncul dalam solusi optimal untuk jumlah n:

6.1.3 Menghitung Solusi

Sekarang mari kita pertimbangkan varian lain dari masalah koin di mana tugas kita adalah menghitung jumlah total cara untuk menghasilkan jumlah x menggunakan koin. Misalnya, jika coin={ 1,3,4}dan x =5, ada total 6 cara:

- 1+1+1+1+1
- 1+1+3
- 1+3+1
- 3+1+1
- 1+4
- 4+1

Sekali lagi, kita dapat menyelesaikan masalah secara rekursif. Biarkan Memecahkan(x) menunjukkan jumlah cara kita dapat membentuk jumlah x. Misalnya, jika coin={ 1,3,4}, maka solve(5) =6 dan rumus rekursifnya adalah

solve(x) =solve(x −1)+ solve(x −3)+ solve(x −4).

Maka, fungsi rekursif umum adalah sebagai berikut:

$$solve(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum \in c\ oins\ solve(x\ - c) + 1\ x\ >\ 0 \end{cases}$$

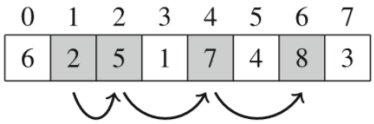
Jika x < 0, nilainya nol, karena tidak ada solusi. Jika x = 0, nilainya satu, karena hanya ada satu cara untuk membentuk jumlah kosong. Jika tidak, kita menghitung jumlah semua nilai dari bentuk solve(x c) di mana c dalam koin. Kode berikut menyusun jumlah larik sedemikian rupa sehingga count[x] sama dengan nilai solve(x) untuk 0≤ x n:

```
count[0] = 1;
for (int x = 1; x <= n; x++) {
        for (auto c : coins) {
        if (x-c >= 0) {
        count[x] += count[x-c];
        }
        }
}
```

Seringkali jumlah solusi sangat besar sehingga tidak diperlukan untuk menghitung jumlah pastinya tetapi cukup untuk memberikan jawaban modulo m dimana, misalnya, m = 10^9 +7. Hal ini dapat dilakukan dengan mengubah kode sehingga semua perhitungan dilakukan modulo m. Pada kode di atas, cukup tambahkan baris
count[x] %= m;
setelah garis
count[x] += count[x-c];

6.2 Contoh Lebih Lanjut

Setelah membahas konsep dasar pemrograman dinamis, sekarang kita siap untuk membahas serangkaian masalah yang dapat diselesaikan secara efisien menggunakan pemrograman dinamis. Seperti yang akan kita lihat, pemrograman dinamis adalah teknik serbaguna yang memiliki banyak aplikasi dalam desain algoritma.



Gambar 6.1 Urutan kenaikan terpanjang dari larik ini adalah [2,5,7,8]

6.2.1 Urutan Peningkatan Terpanjang

Barisan kenaikan terpanjang dalam larik n elemen adalah barisan panjang maksimum elemen larik yang bergerak dari kiri ke kanan, dan setiap elemen dalam barisan lebih besar dari elemen sebelumnya. Misalnya, Gambar 6.1 menunjukkan barisan naik terpanjang dalam larik delapan elemen. Kita dapat secara efisien menemukan peningkatan suburutan terpanjang dalam sebuah array menggunakan pemrograman dinamis. Misalkan length(k) menyatakan panjang dari barisan kenaikan terpanjang yang berakhir pada posisi k. Kemudian, jika kita menghitung semua nilai length (k) di mana 0 ≤ k ≤ n−1, kita akan menemukan panjang dari barisan yang bertambah panjang. Nilai fungsi untuk array contoh kita adalah sebagai berikut:

length(0) = 1
length(1) = 1
length(2) = 2
length(3) = 1
length(4) = 3
length(5) = 2
length(6) = 4

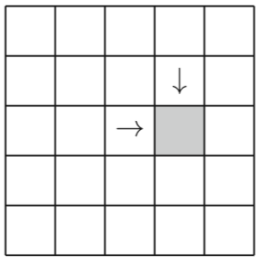
length(7) = 2

Misalnya, panjang(6) = 4, karena barisan kenaikan terpanjang yang berakhir pada posisi 6 terdiri dari 4 elemen. Untuk menghitung nilai length (k), kita harus mencari posisi i < k dimana array[i] < array[k]dan panjang(i) sebesar mungkin. Kemudian kita mengetahui bahwa length (k) = length (i)+1, karena ini adalah cara optimal untuk menambahkan sinar[k] ke suatu barisan. Namun, jika tidak ada posisi i seperti itu, maka length (k) = 1, yang berarti bahwa turunannya hanya berisi array[k]. Karena semua nilai fungsi dapat dihitung dari nilai yang lebih kecil, kita dapat menggunakan pemrograman dinamis untuk menghitung nilainya. Dalam kode berikut, nilai fungsi akan disimpan dalam length array.

```
for (int k = 0; k < n; k++) {
length[k] = 1;
        for (int i = 0; i < k; i++) {
        if (array[i] < array[k]) {
        length[k] = max(length[k],length[i]+1);
        }
        }
}
```

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

Gambar 6.2 Jalur optimal dari sudut kiri atas ke sudut kanan bawah



Gambar 6.3 Dua cara yang mungkin untuk mencapai persegi di jalan

Algoritma yang dihasilkan jelas bekerja dalam waktu O(n2).[8]

6.2.2 Jalur dalam Grid

Masalah kita selanjutnya adalah menemukan jalur dari sudut kiri atas ke sudut kanan bawah dari kisi nxn, dengan batasan bahwa kita hanya dapat bergerak ke bawah dan ke kanan. Setiap kotak berisi bilangan bulat, dan jalurnya harus dibuat sedemikian rupa sehingga jumlah nilai sepanjang jalur tersebut sebesar mungkin. Sebagai contoh, Gambar 6.2 menunjukkan jalur optimal dalam kisi 5x5. Jumlah nilai pada lintasan adalah 67, dan ini adalah jumlah terbesar yang mungkin pada lintasan dari sudut kiri atas ke sudut kanan bawah. Asumsikan bahwa baris dan kolom kisi diberi nomor dari

[8] Dalam masalah ini, juga dimungkinkan untuk menghitung nilai pemrograman dinamis secara lebih efisien dalam waktu O(n log n). Dapatkah Anda menemukan cara untuk melakukan ini?

1 hingga n, dan value[y][x]sama dengan nilai kuadrat (y,x). Misalkan sum(y,x) menunjukkan jumlah maksimum pada lintasan dari sudut kiri atas ke persegi (y,x). Kemudian, sum(n,n) memberi tahu kita jumlah maksimum dari sudut kiri atas ke sudut kanan bawah. Misalnya, pada kisi di atas, jumlah(5,5) =67. Sekarang kita bisa menggunakan rumus

sum(y,x) =max(sum(y,x −1),sum(y−1,x))+value[y][x],

yang berdasarkan pengamatan bahwa lintasan yang berakhir pada bujur sangkar (y,x) dapat berasal dari square (y,x −1) atau dari square(y−1,x) (Gambar 6.3). Jadi, kami memilih arah yang memaksimalkan jumlah. Kami berasumsi bahwa sum(y,x) = 0 ify = 0 atau x =0, sehingga rumus rekursif juga berfungsi untuk kuadrat paling kiri dan paling atas. Karena jumlah fungsi memiliki dua parameter, array pemrograman dinamis juga memiliki dua dimensi. Misalnya, kita dapat menggunakan array

```
int sum[N][N];
```

dan hitung jumlahnya sebagai berikut:

```
for (int y = 1; y <= n; y++) {
        for (int x = 1; x <= n; x++) {
        sum[y][x] = max(sum[y][x-1],sum[y-1][x])+value[y][x];
        }
}
```

Kompleksitas waktu dari algoritma adalah O(n^2).

6.2.3 Masalah Ransel

Istilah knapsack mengacu pada masalah di mana satu set objek diberikan, dan himpunan bagian dengan beberapa properti harus ditemukan. Masalah knapsack seringkali dapat diselesaikan dengan menggunakan pemrograman dinamis. Pada bagian ini, kita fokus pada masalah berikut: Diberikan daftar bobot *[w_1,w_2,...,w_n],* tentukan semua jumlah yang dapat dibangun dengan menggunakan bobot. Misalnya, Gambar 6.4 menunjukkan jumlah yang mungkin untuk bobot [1,3,3,5]. Dalam hal ini, semua jumlah antara 0...12 dimungkinkan, kecuali 2 dan 10. Misalnya, jumlah 7 dimungkinkan karena kita dapat memilih bobot[1,3,3]. Untuk memecahkan masalah, kami fokus pada submasalah di mana kami hanya menggunakan k bobot pertama untuk membangun jumlah. Misalkan possible(x,k) =benar jika kita dapat membuat jumlah x menggunakan k bobot pertama, dan jika tidak, possible(x,k)=false. Nilai fungsi dapat dihitung secara rekursif menggunakan rumus

possible(x,k) =possible(x −wk,k−1) or possible(x,k−1),

yang didasarkan pada fakta bahwa kita dapat menggunakan atau tidak menggunakan bobot wk dalam penjumlahan. Jika kita menggunakan wk, tugas yang tersisa adalah membentuk jumlah x−wk menggunakan bobot k−1 pertama, dan jika kita tidak menggunakan wk, tugas yang tersisa adalah membentuk jumlah x menggunakan bobot k−1 pertama. Kasus dasarnya adalah

$$\text{possible(x0)} = \begin{cases} true & x = 0 \\ false & x \neq 0 \end{cases}$$

karena jika tidak ada bobot yang digunakan, kita hanya dapat membentuk penjumlahan0. Akhirnya, possible(x,n) memberi tahu kita apakah kita dapat membuat jumlah x menggunakan semua bobot.

0	1	2	3	4	5	6	7	8	9	10	11	12
✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓

Gambar 6.4 Membuat penjumlahan menggunakan bobot[1,3,3,5]

	0	1	2	3	4	5	6	7	8	9	10	11	12
$k=0$	✓												
$k=1$	✓	✓											
$k=2$	✓	✓		✓	✓								
$k=3$	✓	✓		✓	✓		✓	✓					
$k=4$	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓

Gambar 6.5 Memecahkan masalah knapsack untuk bobot [1,3,3,5]menggunakan pemrograman dinamis

Gambar 6.5 menunjukkan semua nilai fungsi untuk bobot [1, 3, 3, 5] (simbol " √" menunjukkan nilai sebenarnya). Misalnya, baris k = 2 memberi tahu kita bahwa kita dapat membuat sum[0,1,3,4] menggunakan bobot[1,3]. Biarkan m menunjukkan jumlah total bobot. Solusi pemrograman dinamis waktu O(nm) berikut sesuai dengan fungsi rekursif:

```
possible[0][0] = true;
  for (int k = 1; k <= n; k++) {
            for (int x = 0; x <= m; x++) {
                    if (x-w[k] >= 0) { possible[x][k] |=
            possible[x-w[k]][k-1];
                    }
            possible[x][k] |= possible[x][k-1];
                    }
   }
```

Ternyata ada juga cara yang lebih ringkas untuk mengimplementasikan perhitungan pemrograman dinamis, dengan hanya menggunakan array satu dimensi yang memungkinkan[x] yang menunjukkan apakah kita dapat membuat subset dengan jumlah x. Triknya adalah memperbarui array dari kanan ke kiri untuk setiap bobot baru:

```
possible[0] = true;
        for (int k = 1; k <= n; k++) {
                for (int x = m-w[k]; x >= 0; x--) {
                possible[x+w[k]] |= possible[x];
                }
        }
```

Perhatikan bahwa ide pemrograman dinamis umum yang disajikan di bagian ini juga dapat digunakan dalam masalah knapsack lainnya, seperti dalam situasi di mana objek memiliki bobot dan nilai dan kita harus menemukan subset nilai maksimum yang bobotnya tidak melebihi batas yang diberikan.

6.2.4 Dari Permutasi ke Subset

Menggunakan pemrograman dinamis, seringkali dimungkinkan untuk mengubah sebuah iterasi pada permutasi menjadi sebuah iterasi pada himpunan bagian. Manfaatnya adalah bahwa n!, jumlah permutasi, jauh lebih besar dari 2^n, jumlah himpunan bagian. Sebagai contoh, jika n =20, $n! \approx 2.4 \times 10^{18}$ dan $2^n \approx 10^6$. Jadi, untuk nilai n tertentu, kita dapat melewati himpunan bagian secara efisien tetapi tidak melalui

permutasi. Sebagai contoh, perhatikan masalah berikut: Ada lift dengan berat maksimum x, dan n orang yang ingin naik dari lantai dasar ke lantai atas. Orang-orang tersebut diberi nomor 0,1,...,n−1, dan berat orang i adalah weight[i]. Berapa jumlah minimum perjalanan yang diperlukan untuk membawa semua orang ke lantai atas? Misalnya, misalkan x =12, n =5, dan bobotnya adalah sebagai berikut:

- weight[0]=2
- weight[1]=3
- weight[2]=4
- weight[3]=5
- weight[4]=9

Dalam skenario ini, jumlah minimum perjalanan adalah dua. Salah satu solusi optimal adalah sebagai berikut: pertama, orang 0, 2, dan 3 naik lift (berat total 11), dan kemudian, orang 1 dan 4 naik lift (total berat 12). Soal dapat dengan mudah diselesaikan dalam waktu *O(n!n)* dengan menguji semua kemungkinan permutasi dari *n* orang. Namun, kita dapat menggunakan pemrograman dinamis untuk membuat algoritma waktu $O(2^n n)$ yang lebih efisien. Idenya adalah untuk menghitung untuk setiap subset orang dua nilai: jumlah minimum perjalanan yang dibutuhkan dan berat minimum orang yang naik dalam kelompok terakhir. Biarkan rides(S) menunjukkan jumlah minimum perjalanan untuk subset S, dan biarkan last(S) menunjukkan berat minimum perjalanan terakhir dalam solusi di mana jumlah perjalanan minimum. Misalnya, dalam skenario di atas

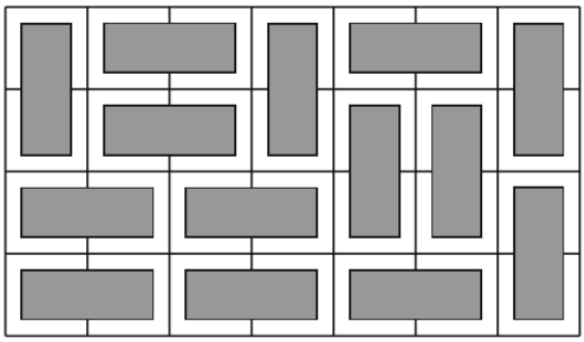
rides({3,4}) =2 and last({3,4}) =5,

karena cara optimal bagi orang 3 dan 4 untuk sampai ke lantai atas adalah mereka mengambil dua jalan terpisah dan orang 4 pergi pertama, yang meminimalkan berat perjalanan kedua. Tentu saja, tujuan akhir kita adalah menghitung nilai rides({0...n−1}). Kita dapat menghitung nilai fungsi secara rekursif dan kemudian menerapkan pemrograman dinamis. Untuk menghitung nilai subset S, kami menelusuri semua orang yang termasuk dalam S dan secara optimal memilih orang terakhir p yang memasuki elevator. Setiap pilihan seperti itu menghasilkan submasalah untuk sebagian kecil orang. Jika terakhir(S\p)+ weight[p]≤x,kita dapat menambahkan p ke jalan terakhir. Jika tidak, kita harus menyimpan wahana baru yang hanya berisi p. Cara mudah untuk mengimplementasikan perhitungan pemrograman dinamis adalah dengan menggunakan operasi bit. Pertama, kita mendeklarasikan sebuah array

pair<**int**,int> best[1<<N];

yang berisi untuk setiap subset S sepasang (naik(S),terakhir(S)). Untuk subset kosong, tidak ada perjalanan yang diperlukan:

best[0] = {0,0};



Gambar 6.6 Salah satu cara untuk mengisi kisi 4×7 menggunakan ubin 1×2 dan 2×1

Kemudian, kita dapat mengisi array sebagai berikut:

```
for (int s = 1; s < (1<<n); s++) {
        //initial value : n+1rides are needed
        best[s] = {n+1,0};
                for (int p = 0; p < n; p++) {
        if (s&(1<<p)) {
                auto option = best[s^(1<<p)];
        if (option.second+weight[p] <= x) {
        //addp to a nexist in gride
        option.second += weight[p];
} else {
        //reserve a new ride for p
        option.first++;
        option.second = weight[p];
        }
        best[s] = min(best[s], option);
        }
        }
}
```

Perhatikan bahwa loop di atas menjamin bahwa untuk setiap dua himpunan bagian S_1 dan S_2 sedemikian rupa sehingga $S_1 \subset S_2$, kita memproses S_1 sebelum S_2. Dengan demikian, nilai pemrograman dinamis dihitung dalam urutan yang benar.

6.2.5 Menghitung kotak

Terkadang keadaan solusi pemrograman dinamis lebih kompleks daripada kombinasi nilai yang tetap. Sebagai contoh, perhatikan masalah menghitung jumlah cara yang berbeda untuk mengisi kisi nxm menggunakan ubin ukuran 1x2 dan 2x1. Misalnya, ada total 781 cara untuk mengisi kisi 4x7, salah satunya adalah solusi yang ditunjukkan pada Gambar 6.6. Masalah tersebut dapat diselesaikan dengan menggunakan pemrograman dinamis dengan menelusuri grid baris demi baris. Setiap baris dalam solusi dapat direpresentasikan sebagai string yang berisi m karakter dari himpunan {⊓, ⊔, ⊏, ⊐}. Misalnya, solusi pada Gambar 6.6 terdiri dari empat baris yang sesuai dengan string berikut:

- ⊓⊏⊐⊓⊏⊐⊓
- ⊔⊏⊐⊔⊓⊓⊔
- ⊏⊐⊏⊐⊔⊔⊓
- ⊏⊐⊏⊐⊏⊐⊔

Misalkan baris grid diindeks dari 1 sampai n. Misalkan count(k,x) menyatakan banyaknya cara untuk membangun solusi untuk baris 1...k sedemikian rupa sehingga string x sesuai dengan baris k. Dimungkinkan untuk menggunakan pemrograman dinamis di sini, karena status baris hanya dibatasi oleh status baris sebelumnya. Solusi valid jika baris 1 tidak berisi karakter , baris tidak berisi karakter , dan semua baris berurutan kompatibel. Misalnya, baris ⊔⊏⊐⊔⊓⊓⊔ dan ⊏⊐⊏⊐⊔⊔⊓ kompatibel, sedangkan baris ⊓⊏⊐⊓⊏⊐⊓ dan ⊏⊐⊏⊐⊏⊐⊔ tidak kompatibel. Karena satu baris terdiri dari beberapa karakter dan ada empat pilihan untuk setiap karakter, jumlah baris yang berbeda paling banyak 4^m.

Kita dapat melalui $O(4^m)$ keadaan yang mungkin untuk setiap baris, dan untuk setiap keadaan, ada $O(4^m)$ keadaan yang mungkin untuk baris sebelumnya, sehingga kompleksitas waktu dari penyelesaiannya adalah $O(n4^{2m})$. Dalam praktiknya, ide yang

baik untuk memutar kisi-kisi sehingga sisi terpendek memiliki panjang m, karena faktor 42m mendominasi kompleksitas waktu. Dimungkinkan untuk membuat solusi lebih efisien dengan menggunakan representasi baris yang lebih ringkas. Ternyata cukup untuk mengetahui kolom mana dari baris sebelumnya yang berisi bujur sangkar atas ubin vertikal. Dengan demikian, kita dapat merepresentasikan baris hanya dengan menggunakan karakter ⊓ dan , Dimana merupakan kombinasi dari karakter ⊔, ⊏ dan ⊐ . Menggunakan representasi ini, hanya ada 2^m baris yang berbeda, dan kompleksitas waktu adalah $O(n2^{2m})$. Sebagai catatan terakhir, ada juga rumus langsung untuk menghitung jumlah ubin:

$$\prod_{a=1}^{[\frac{n}{2}]} \prod_{b=1}^{[\frac{m}{2}]} 4\,x\left(cos^2\frac{\pi a}{n+1} + cos^2\frac{\pi b}{m+1}\right)$$

Rumus ini sangat efisien, karena menghitung jumlah ubin dalam waktu O(nm), tetapi karena jawabannya adalah perkalian bilangan real, masalah saat menggunakan rumus adalah bagaimana menyimpan hasil antara secara akurat.

BAB 7
ALGORITMA GRAFIK

Banyak masalah pemrograman dapat diselesaikan dengan mempertimbangkan situasi sebagai graf dan menggunakan algoritma graf yang sesuai. Dalam bab ini, kita akan mempelajari dasar-dasar graf dan pilihan algoritma graf yang penting. Bagian pertama membahas terminologi grafik dan struktur data yang dapat digunakan untuk mewakili grafik dalam algoritma. Bagian selanjutnya memperkenalkan dua algoritma traversal graf fundamental. Pencarian Depth First adalah cara sederhana untuk mengunjungi semua node yang dapat dicapai dari node awal, dan pencarian Breadth First mengunjungi node dalam urutan jarak yang meningkat dari node awal. Bagian selanjutnya menyajikan algoritma untuk menemukan jalur terpendek dalam grafik berbobot.

Algoritma Bellman-Ford adalah algoritma sederhana yang menemukan jalur terpendek dari node awal ke semua node lainnya. Algoritma Dijkstra merupakan algoritma yang lebih efisien yang mensyaratkan bahwa bobot genap tidak negatif. Algoritma Floyd-Warshall menentukan jalur terpendek antara semua pasangan simpul dari suatu graf. Bagian tengah mengeksplorasi sifat khusus dari grafik asiklik berarah. Kita akan belajar bagaimana membangun semacam topologi dan bagaimana menggunakan pemrograman dinamis untuk secara efisien memproses grafik tersebut. Bagian selanjutnya berfokus pada grafik sukses di mana setiap node memiliki suksesor unik. Kami akan membahas cara yang efisien untuk menemukan penerus node dan algoritma Floyd untuk deteksi siklus. Bagian akhir menyajikan algoritma Kruskal dan Prim untuk membangun pohon merentang minimum. Algoritma Kruskal didasarkan pada struktur union-find yang efisien yang juga memiliki kegunaan lain dalam desain algoritma.

7.1 Dasar-dasar Grafik

Pada bagian ini, pertama-tama kita akan membahas terminologi yang digunakan ketika membahas graf dan propertinya. Setelah ini, kami fokus pada struktur data yang dapat digunakan untuk mewakili grafik dalam pemrograman algoritma.

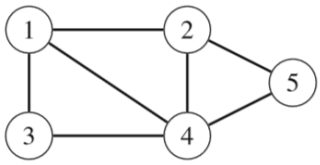
7.1.1 Terminologi Grafik

Graf terdiri dari node (juga disebut simpul) yang terhubung dengan tepi. Dalam buku ini, variabel n menunjukkan jumlah node dalam grafik, dan variabel m menunjukkan jumlah tepi. Node diberi nomor menggunakan bilangan bulat 1,2,...,n. Sebagai contoh, Gambar 7.1 menunjukkan grafik dengan 5 node dan 7 tepi. Sebuah jalur mengarah dari sebuah simpul ke simpul lain melalui tepi grafik. Panjang suatu lintasan adalah jumlah sisi di dalamnya. Sebagai contoh, Gambar 7.2 menunjukkan jalur 1→3→4→ 5panjang3darinode1tonode5. Siklus adalah jalur di mana node pertama dan terakhir adalah sama. Sebagai contoh, Gambar 7.3 menunjukkan siklus 1→3→4→1. Sebuah graf terhubung jika ada jalur antara dua node. Pada Gambar 7.4, graf kiri terhubung, tetapi graf kanan tidak terhubung, karena tidak mungkin untuk berpindah dari node 4 ke node lain.

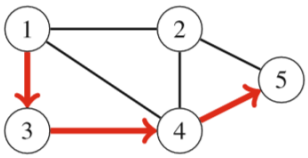
Bagian-bagian yang terhubung dari suatu graf disebut komponen-komponennya. Misalnya, grafik pada Gambar 7.5 memiliki tiga komponen:{1, 2, 3},{4, 5, 6, 7}, dan{8}. Pohon adalah graf terhubung yang tidak mengandung siklus. Gambar 7.6 menunjukkan contoh graf yang berupa pohon. Pada graf berarah, sisi-sisinya hanya

dapat dilintasi satu arah saja. Gambar 7.7 menunjukkan contoh graf berarah. Graf ini berisi lintasan 3→1→2→5 dari simpul 3 ke simpul 5, tetapi tidak ada lintasan dari simpul 5 ke simpul 3. Dalam graf berbobot, setiap sisi diberi bobot. Bobot sering diinterpretasikan sebagai panjang sisi, dan panjang lintasan adalah jumlah dari bobot sisinya. Misalnya, graf pada Gambar 7.8 berbobot, dan panjang jalur1→3→4→5 adalah 1+7+3=11. Ini adalah jalur terpendek dari node 1 ke node 5.

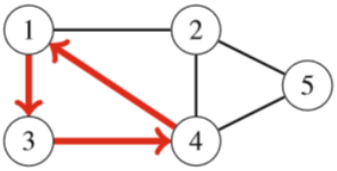
Dua simpul desa bertetangga atau berdekatan jika ada tepi di antara mereka. Derajat suatu simpul adalah jumlah tetangganya. Gambar 7.9 menunjukkan derajat setiap simpul dari suatu graf. Sebagai contoh, derajat simpul 2 adalah 3, karena tetangganya adalah 1, 4, dan 5. Jumlah derajat dalam suatu graf selalu 2m, di mana m adalah jumlah rusuk, karena setiap ruas menaikkan derajat tepat dua simpul oleh satu. Oleh karena itu, jumlah derajat adalah selalu genap. Suatu graf beraturan jika derajat setiap simpul adalah konstanta d. Suatu graf dikatakan lengkap jika derajat setiap simpul adalah n−1, yaitu, graf tersebut memuat semua kemungkinan sisi di antara simpul-simpul tersebut. Dalam graf berarah, derajat masuk suatu simpul adalah jumlah sisi yang berakhir pada simpul tersebut, dan derajat keluar suatu simpul adalah jumlah sisi yang dimulai pada simpul tersebut.



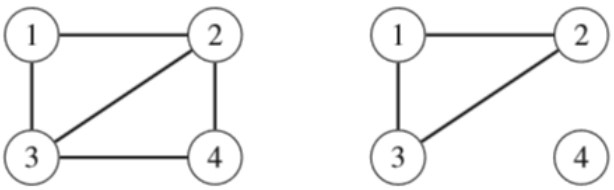
Gambar 7.1 Graf dengan 5 node dan 7 edge



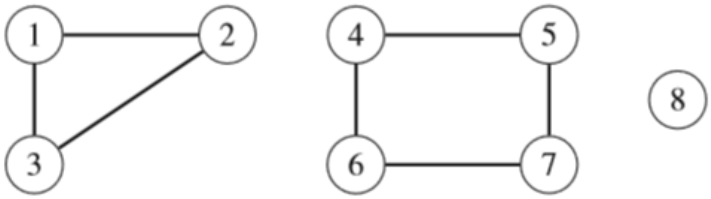
Gambar 7.2 Jalur dari node 1 ke node 5



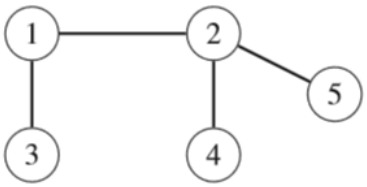
Gambar 7.3 Siklus tiga node



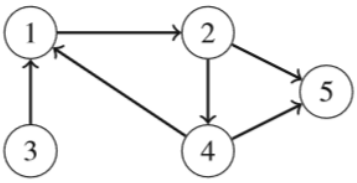
Gambar 7.4 Grafik kiri terhubung, grafik kanan tidak



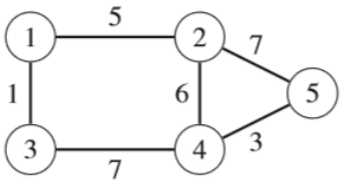
Gambar 7.5 Grafik dengan tiga komponen



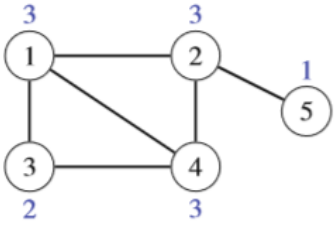
Gambar 7.6 Sebuah pohon



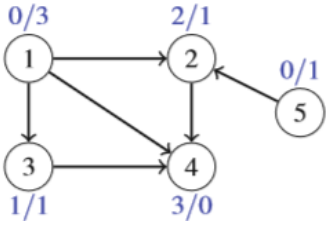
Gambar 7.7 Graf berarah



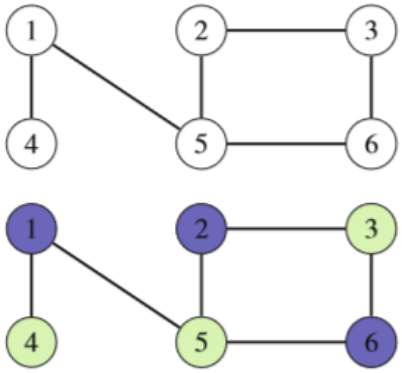
Gambar 7.8 Grafik berbobot



Gambar 7.9 Derajat node



Gambar 7.10 Indegrees dan outdegrees



Gambar 7.11 Graf bipartit dan pewarnaannya

Gambar 7.10 menunjukkan derajat masuk dan derajat keluar dari setiap simpul graf. Sebagai contoh, simpul 2 memiliki derajat masuk 2 dan derajat keluar 1. Suatu graf bipartit jika dimungkinkan untuk mewarnai simpul-simpulnya menggunakan dua warna sedemikian rupa sehingga tidak ada simpul-simpul yang bertetangga yang memiliki warna yang sama. Ternyata suatu graf adalah bipartit tepat ketika tidak memiliki siklus dengan jumlah sisi ganjil. Sebagai contoh, Gambar 7.11 menunjukkan grafik bipartit dan pewarnaannya.

7.1.2 Representasi Grafik

Ada beberapa cara untuk merepresentasikan grafik dalam algoritma. Pilihan struktur data tergantung pada ukuran grafik dan cara algoritma memprosesnya. Selanjutnya kita akan membahas tiga representasi populer.

Daftar Kedekatan

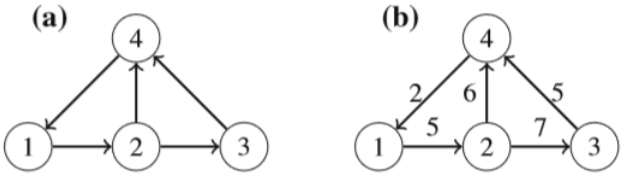
Dalam representasi daftar ketetanggaan, setiap simpul x dari graf diberi daftar ketetanggaan yang terdiri dari simpul-simpul di mana terdapat sisi dari x. Daftar ketetanggaan adalah cara paling populer untuk merepresentasikan grafik, dan sebagian besar algoritme dapat diimplementasikan secara efisien dengan menggunakan daftar tersebut. Cara mudah untuk menyimpan daftar adjacency adalah dengan mendeklarasikan array vektor sebagai berikut:

```
vector<int> adj[N];
```

Konstanta N dipilih sehingga semua daftar kedekatan dapat disimpan. Sebagai contoh, grafik pada Gambar 7.12a dapat disimpan sebagai berikut:

```
adj[1].push_back(2);
adj[2].push_back(3);
adj[2].push_back(4);
adj[3].push_back(4);
adj[4].push_back(1);
```

Jika grafik tidak berarah, dapat disimpan dengan cara yang sama, tetapi setiap tepi ditambahkan di kedua arah.



Gambar 7.12 Contoh grafik

Untuk graf berbobot, strukturnya dapat diperluas sebagai berikut:

```
vector<pair<int,int>> adj[N];
```

Dalam hal ini, daftar kedekatan simpul a berisi pasangan (b,w) selalu ketika ada tepi dari simpul tonodeb dengan bobot w. Misalnya, grafik pada Gambar. 7.12b dapat disimpan sebagai berikut:

```
adj[1].push_back({2,5});
adj[2].push_back({3,7});
adj[2].push_back({4,6});
adj[3].push_back({4,5});
adj[4].push_back({1,2});
```

Dengan menggunakan daftar adjacency, kita dapat secara efisien menemukan node yang dapat kita pindahkan dari node tertentu melalui sebuah edge. Misalnya, loop berikut melewati semua node yang dapat kita pindahkan dari node s:

```
for (auto u : adj[s]) {
//process node u
}
```

Matriks Kedekatan

Matriks ketetanggaan menunjukkan sisi-sisi yang terdapat dalam suatu graf. Kita dapat memeriksa secara efisien dari matriks ketetanggaan jika ada tepi di antara dua simpul. Matriks dapat disimpan sebagai array

```
int adj[N][N];
```

dimana setiap nilai adj[a][b]menunjukkan apakah graf tersebut mengandung tepi dari simpul a ke simpul b. Jika sisi dimasukkan ke dalam graf, maka adj[a][b]=1, dan sebaliknya adj[a][b]=0. Sebagai contoh, matriks ketetanggaan untuk graf pada Gambar 7.12a adalah

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Jika graf berbobot, representasi matriks ketetanggaan dapat diperpanjang sehingga matriks tersebut memuat bobot sisi jika sisi tersebut ada. Dengan menggunakan representasi ini, grafik pada Gambar 7.12b sesuai dengan matriks berikut:

$$\begin{bmatrix} 0 & 5 & 0 & 0 \\ 0 & 0 & 7 & 1 \\ 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 \end{bmatrix}$$

Kelemahan dari representasi matriks ketetanggaan adalah bahwa matriks ketetanggaan mengandung n^2 elemen, dan biasanya kebanyakan adalah nol. Untuk alasan ini, representasi tidak dapat digunakan jika grafiknya besar.

Daftar Tepi

Sebuah daftar tepi berisi semua tepi grafik dalam beberapa urutan. Ini adalah cara yang mudah untuk merepresentasikan graf jika algoritma memproses semua sisinya, dan tidak diperlukan untuk menemukan sisi yang dimulai pada simpul tertentu. Daftar tepi dapat disimpan dalam vektor

```
vector<pair<int,int>> edges;
```

di mana setiap pasangan (a,b) menunjukkan bahwa ada tepi dari simpul a ke simpul b. Dengan demikian, grafik pada Gambar 7.12a dapat direpresentasikan sebagai berikut:

```
edges.push_back({1,2});
edges.push_back({2,3});
```

```
edges.push_back({2,4});
edges.push_back({3,4});
edges.push_back({4,1});
```

Jika graf berbobot, struktur dapat diperpanjang sebagai berikut:

```
vector<tuple<int,int,int>> edges;
```

Setiap elemen dalam daftar ini berbentuk (a, b, w), yang berarti ada sisi dari simpul a ke simpul b dengan bobot w. Sebagai contoh, grafik pada Gambar 7.12b dapat direpresentasikan sebagai berikut[9]:

```
edges.push_back({1,2,5});
edges.push_back({2,3,7});
edges.push_back({2,4,6});
edges.p→ush_back({3,4,5});
edges.push_back({4,1,2});
```

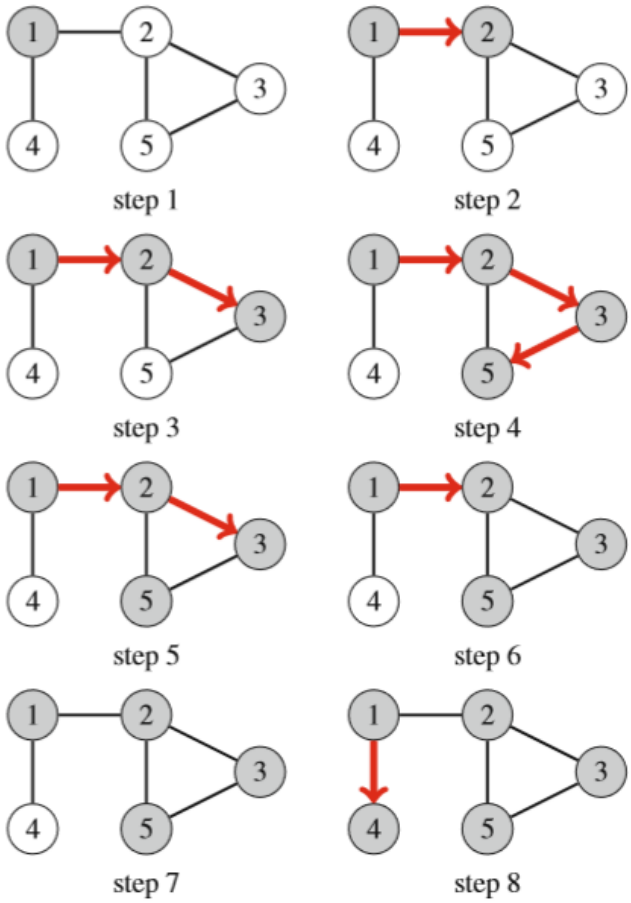
7.2 Grafik Traversal

Bagian ini membahas dua algoritma graf fundamental: depth-first search dan breadth-firstsearch. Kedua algoritma tersebut memberikan node awal dalam graf, dan mereka mengunjungi semua node yang dapat dicapai dari node awal. Perbedaan dalam algoritma adalah urutan di mana mereka mengunjungi node.

7.2.1 Pencarian Kedalaman-Pertama

Depth-first search (DFS) adalah teknik traversal grafik langsung. Algoritma dimulai pada node awal dan berlanjut ke semua node lain yang dapat dijangkau dari node awal menggunakan tepi grafik. Depth-firstsearch selalu mengikuti satu jalur dalam grafik selama menemukan node baru. Setelah ini, ia kembali ke node sebelumnya dan mulai menjelajahi bagian lain dari grafik. Algoritme melacak node yang dikunjungi, sehingga memproses setiap node hanya sekali. Gambar7.13menunjukkan depth-firstsearchprocessesagraph.Pencarian dapat dimulai pada simpul mana pun dari grafik; dalam contoh ini kita memulai pencarian pada node 1. Pertama pencarian mengeksplorasi jalur 1→2→ 3→5, kemudian kembali ke node 1 dan mengunjungi node yang tersisa 4.

[9] Di beberapa kompiler lama, fungsi make_tuple harus digunakan sebagai ganti kurung kurawal (mis., make_tuple(1,2,5) alih-alih {1,2,5}).



Gambar 7.13 *Depth-first search*

Implementasi

Depth-first search dapat dengan mudah diimplementasikan menggunakan rekursi. Fungsi berikut dfs memulai pencarian depth-first pada node yang diberikan. Fungsi mengasumsikan bahwa grafik disimpan sebagai daftar ketetanggaan dalam array

```
vector<int> adj[N];
```

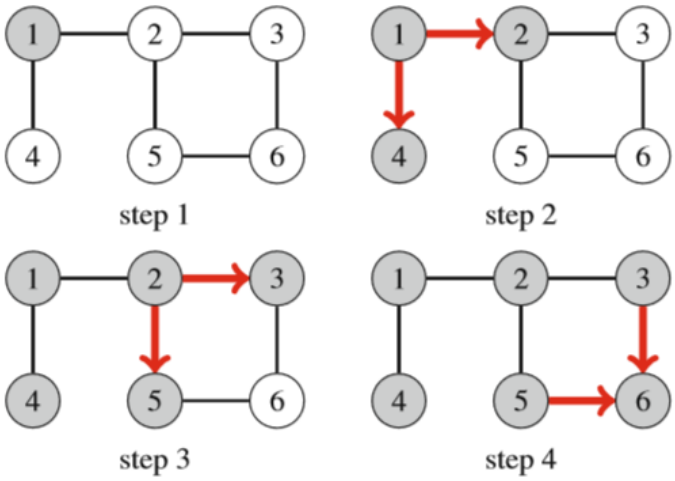
dan juga memelihara sebuah array

```
bool visited[N];
```

yang melacak node yang dikunjungi. Awalnya, setiap nilai array salah, dan ketika pencarian tiba di node, nilai yang dikunjungi menjadi benar. Fungsi tersebut dapat diimplementasikan sebagai berikut:

```
void dfs(int s) {
    if (visited[s]) return;
    visited[s] = true;
    //process nodes
    for (auto u: adj[s]) {
    dfs(u);
    }
}
```

Kompleksitas waktu depth-first search adalah O(n+m) di mana n adalah jumlah node dan m adalah jumlah edge, karena algoritma memproses setiap node dan edge satu kali.



Gambar 7.14 Breadth-first search

7.2.2 Pencarian Luas-Pertama

Breadth-first search (BFS) mengunjungi simpul-simpul dari suatu graf dalam urutan jarak yang meningkat dari simpul awal. Dengan demikian, kita dapat menghitung jarak dari simpul awal ke semua simpul lain menggunakan pencarian pertama-lebar. Namun, pencarian luas-pertama lebih sulit untuk diterapkan daripada pencarian mendalam-pertama. Breadth-firstsearch berjalan melalui node satu demi satu. Pertama, pencarian mengeksplorasi node yang jaraknya dari node awal adalah 1, kemudian node yang jaraknya 2, dan seterusnya. Proses ini berlanjut sampai semua node telah dikunjungi. Gambar 7.14 menunjukkan bagaimana pencarian luas-pertama memproses grafik. Misalkan pencarian dimulai pada node 1. Pertama pencarian mengunjungi node 2 dan 4 dengan jarak 1, kemudian node 3 dan 5 dengan jarak 2, dan terakhir node 6 dengan jarak 3.

Penerapan

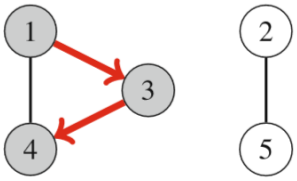
Pencarian *Breadth-first* lebih sulit untuk diterapkan daripada pencarian depth-first, karena algoritma mengunjungi node di bagian grafik yang berbeda. Implementasi tipikal didasarkan pada antrian yang berisi node. Pada setiap langkah, node berikutnya dalam antrian akan diproses. Kode berikut mengasumsikan bahwa grafik disimpan sebagai daftar kedekatan dan mempertahankan struktur data berikut:

```
queue<int> q;
bool visited[N];
int distance[N];
```

Antrian q berisi node yang akan diproses dalam urutan peningkatan jaraknya. Node baru selalu ditambahkan ke akhir antrian, dan node di awal antrian adalah node berikutnya yang akan diproses. Larik yang dikunjungi menunjukkan simpul mana yang telah dikunjungi pencarian, dan jarak larik akan berisi jarak dari simpul awal ke semua simpul grafik. Pencarian dapat diimplementasikan sebagai berikut, mulai dari node x:

```
visited[x] = true;
distance[x] = 0;
q.push(x);
while (!q.empty()) {
        int s = q.front(); q.pop();
        //process node s
        for (auto u : adj[s]) {
```

```
            if (visited[u]) continue;
            visited[u] = true;
            distance[u] = distance[s]+1;
            q.push(u);
            }
}
```



Gambar 7.15 Memeriksa konektivitas grafik

Seperti pencarian mendalam-pertama, kompleksitas waktu pencarian pertama adalah O(n+m), di mana n adalah jumlah simpul dan m adalah jumlah sisi.

7.2.3 Aplikasi

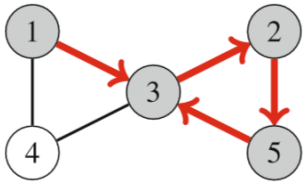
Dengan menggunakan algoritma traversal graf, kita dapat memeriksa banyak properti graf. Biasanya, baik pencarian mendalam-pertama dan pencarian luas-pertama dapat digunakan, tetapi dalam praktiknya, pencarian mendalam-pertama adalah pilihan yang lebih baik, karena lebih mudah untuk diterapkan. Dalam aplikasi yang dijelaskan di bawah ini, kami akan menganggap bahwa grafik tidak berarah.

Pemeriksaan Konektivitas

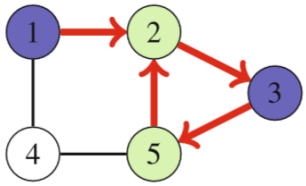
Suatu graf terhubung jika terdapat lintasan antara dua simpul pada graf tersebut. Dengan demikian, kita dapat memeriksa apakah suatu graf terhubung dengan memulai pada simpul sembarang dan mencari tahu apakah kita dapat menjangkau semua simpul lainnya. Sebagai contoh, pada Gambar 7.15, karena pencarian depth-first dari node 1 tidak mengunjungi semua node, kita dapat menyimpulkan bahwa graf tidak terhubung. Dengan cara yang sama, kita juga dapat menemukan semua komponen graf yang terhubung dengan melakukan iterasi melalui node dan selalu memulai pencarian kedalaman-pertama baru jika node saat ini belum termasuk dalam komponen apa pun.

Deteksi Siklus

Sebuah graf berisi sebuah siklus jika selama traversal graf, kita menemukan sebuah simpul yang tetangganya (selain simpul sebelumnya pada jalur saat ini) telah dikunjungi. Sebagai contoh, pada Gambar 7.16, pencarian depth-first dari node 1 mengungkapkan bahwa grafik tersebut berisi sebuah siklus. Setelah berpindah dari node 2 ke node 5 kita melihat bahwa tetangga 3 dari node 5 telah dikunjungi. Dengan demikian, grafik tersebut berisi siklus yang melewati simpul 3, misalnya 3→2→5→3.



Gambar 7.16 Menemukan sebuah siklus dalam sebuah graf



Gambar 7.17 Konik saat memeriksa bipartit

Cara lain untuk menentukan apakah suatu graf berisi siklus adalah dengan menghitung jumlah simpul dan sisi di setiap komponen. Jika suatu komponen mengandung c node dan tidak ada siklus, komponen tersebut harus mengandung tepat *c−1 edge* (jadi harus berupa pohon). Jika ada c atau lebih tepi, komponen tersebut pasti mengandung sebuah siklus.

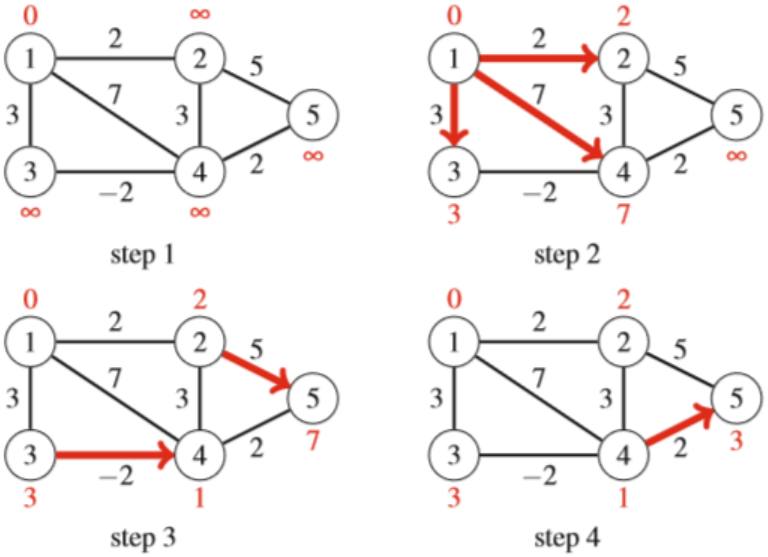
Pemeriksaan Bipartit

Suatu graf dikatakan bipartit jika simpul-simpulnya dapat diwarnai dengan menggunakan dua warna sehingga tidak ada simpul-simpul yang bertetangga dengan warna yang sama. Sangat mudah untuk memeriksa apakah suatu graf bipartit menggunakan algoritma traversal graf. Idenya adalah memilih dua warna X dan Y, mewarnai simpul awal X, semua tetangganya Y, semua tetangganya X, dan seterusnya. Jika pada beberapa titik pencarian kita melihat bahwa dua node yang berdekatan memiliki warna yang sama, ini berarti graf tersebut tidak bipartit. Jika tidak, grafiknya bipartit dan satu pewarnaan telah ditemukan.

Sebagai contoh, pada Gambar 7.17, pencarian depth-first dari node1 menunjukkan bahwa grafik tersebut bukan bipartit, karena kita perhatikan bahwa kedua node 2 dan 5 harus memiliki warna yang sama, sedangkan node tersebut berdekatan dalam grafik. Algoritma ini selalu bekerja, karena ketika hanya ada dua warna yang tersedia, warna node awal pada komponen menentukan warna jatuh node lain dalam komponen. Tidak ada bedanya apa warnanya. Perhatikan bahwa pada kasus umum sulit untuk mengetahui apakah odesinagraf dapat diwarnai dengan menggunakan k warna sehingga tidak ada node yang berdekatan yang memiliki warna yang sama. Masalahnya sudah NP-keras untuk *k =3*.

7.3 Jalur terpendek

Menemukan jalur terpendek antara dua simpul dari suatu graf merupakan masalah penting yang memiliki banyak aplikasi praktis. Misalnya, masalah alami yang terkait dengan jaringan jalan adalah menghitung kemungkinan panjang terpendek dari rute antara dua kota, mengingat panjang jalan. Dalam graf tak berbobot, panjang jalur sama dengan jumlah sisinya, dan kita dapat menggunakan pencarian pertama untuk menemukan jalur terpendek. Namun, pada bagian ini kita fokus pada graf berbobot di mana algoritma yang lebih canggih diperlukan untuk menemukan jalur terpendek.



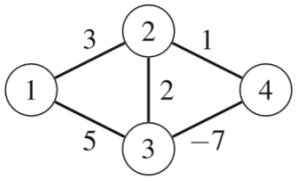
Gambar 7.18 Algoritma Bellman-Ford

7.3.1 Algoritma Bellman–Ford

Algoritma Bellman-Ford menemukan jalur terpendek dari node awal ke semua node grafik. Algoritme dapat memproses semua jenis graf, asalkan graf tersebut tidak mengandung siklus dengan panjang negatif. Jika grafik berisi siklus negatif, algoritma dapat mendeteksi ini. Algoritma melacak jarak dari node awal ke semua node grafik. Awalnya, jarak ke node awal adalah 0 dan jarak ke node lain tidak terbatas. Algoritma kemudian mengurangi jarak dengan menemukan tepi yang memperpendek jalur sampai tidak mungkin untuk mengurangi jarak. Gambar 7.18 menunjukkan bagaimana algoritma Bellman-Ford memproses grafik. Pertama, algoritme mengurangi jarak menggunakan tepi 1→2, 1→3 dan 1→4, kemudian menggunakan tepi 2→5 dan 3→4, dan terakhir menggunakan tepi 4→5. Setelah ini, tidak ada tepi yang dapat digunakan untuk mengurangi jarak, yang berarti bahwa jarak adalah final.

Penerapan

Implementasi dari algoritma Bellman-Ford di bawah ini menentukan jarak terpendek dari node *x* ke semua node dari grafik. Kode tersebut mengasumsikan bahwa graf disimpan sebagai edge list edge yang terdiri dari tupel-tupel berbentuk *(a,b,w),* artinya terdapat edge dari node *a* ke node *b* dengan bobot *w.* Algoritma terdiri dari n 1 putaran, dan pada setiap putaran algoritma melewati semua tepi grafik dan mencoba untuk mengurangi jarak. Algoritma membangun jarak array yang akan berisi jarak dari node x ke semua node. Konstanta INF menunjukkan jarak tak berhingga.

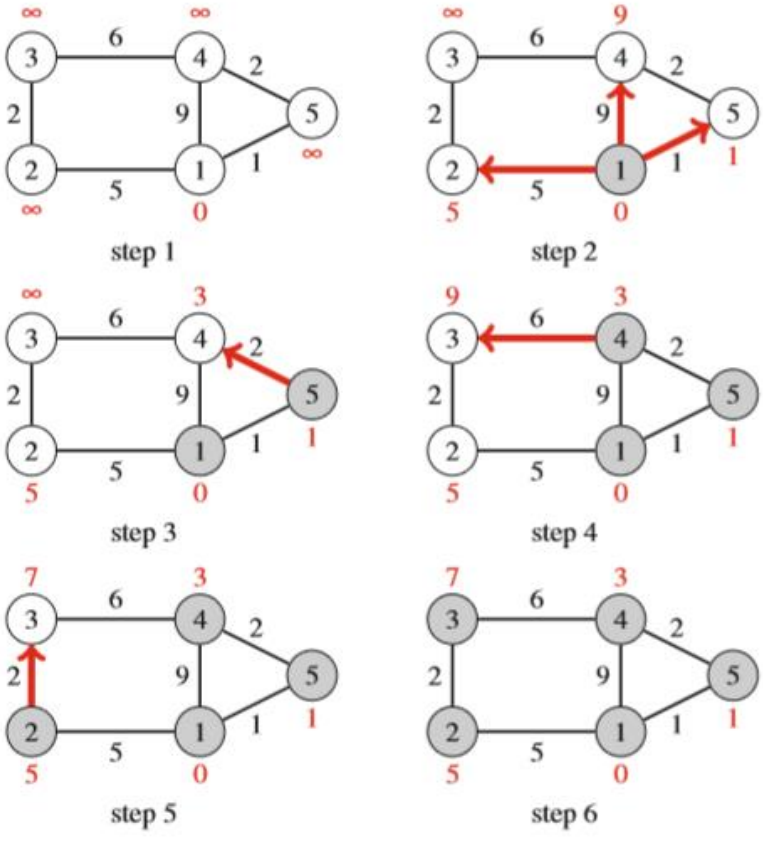


Gambar 7.19 Grafik dengan siklus negatif

```
for (int i = 1; i <= n; i++) {
        distance[i] = INF;
}
distance[x] = 0;
```

```
for (int i = 1; i <= n-1; i++) {
        for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
        }
}
```

Kompleksitas waktu dari algoritma adalah O(nm), karena algoritma terdiri dari n−1 putaran dan mengitari semua sisi selama keliling. Jika ada siklus negatif dalam graf, semua jarak adalah final setelah n−1 putaran, karena setiap jalur terpendek dapat berisi paling banyak n−1 sisi. Ada beberapa cara untuk mengoptimalkan algoritma dalam praktek. Pertama, jarak akhir biasanya dapat ditemukan lebih awal daripada setelah 1 putaran, jadi kita dapat menghentikan algoritma jika tidak ada jarak yang dapat dikurangi selama putaran. Varian yang lebih maju adalah algoritma SPFA ("Algoritma Jalur Terpendek Lebih Cepat") yang mempertahankan antrian node yang mungkin digunakan untuk mengurangi jarak. Hanya node dalam antrian yang akan diproses, yang seringkali menghasilkan pencarian yang lebih efisien.



Gambar 7.20 Algoritma Dijkstra

Siklus Negatif

Algoritma Bellman–Fordal juga dapat digunakan untuk memeriksa apakah grafik berisi siklus dengan panjang negatif. Dalam hal ini, jalur apa pun yang berisi siklus dapat dipersingkat berkali-kali hingga tak terbatas, sehingga konsep jalur tes pendek tidak berarti. Misalnya, grafik pada Gambar 7.19 berisi siklus negatif 2→3→4→2 dengan panjang −4. Siklus negatif dapat dideteksi menggunakan algoritma Bellman-Ford dengan menjalankan algoritma untuk putaran. Jika putaran terakhir mengurangi jarak,

grafik berisi siklus negatif. Perhatikan bahwa algoritma ini dapat digunakan untuk mencari siklus negatif di seluruh grafik terlepas dari simpul awal.

7.3.2 Algoritma Dijkstra

Algoritma Dijkstra mencari jalur terpendek dari node awal ke semua node pada graf, seperti algoritma Bellman-Fordal. Keuntungan dari algoritma Dijkstra adalah lebih efisien dan dapat digunakan untuk memproses graf besar. Namun, algoritma ini mensyaratkan bahwa tidak ada sisi berbobot negatif dalam graf. Seperti algoritma Bellman–Fordal, algoritma Dijkstra mempertahankan jarak ke node dan menguranginya selama pencarian. Pada setiap langkah, algoritma Dijkstra memilih node yang belum diproses dan yang jaraknya mungkin sekecil mungkin. Kemudian, algoritma melewati semua sisi yang mulai menggunakan node tersebut dan mengurangi jaraknya.

Algoritma Dijkstra efisien, karena hanya memproses setiap sisi dalam graf satu kali, menggunakan fakta bahwa tidak ada sisi negatif. Gambar 7.20 menunjukkan bagaimana algoritma Dijkstra memproses grafik. Seperti pada algoritma Bellman-Ford, jarak awal ke semua node, kecuali node awal, tidak berhingga. Algoritme memproses node dalam urutan 1, 5, 4, 2, 3, dan pada setiap node mengurangi jarak menggunakan tepi yang mulai di node. Perhatikan bahwa jarak ke node tidak pernah berubah setelah memproses node.

Penerapan

Implementasi yang efisien dari algoritma Dijkstra mengharuskan kita dapat secara efisien menemukan node dengan jarak minimum yang belum diproses. Struktur data yang sesuai untuk antrian prioritas ini yang berisi node utama yang diurutkan berdasarkan jaraknya. Dengan menggunakan antrian prioritas, node berikutnya yang akan diproses dapat diambil dalam waktu logaritmik. Implementasi buku teks yang khas dari algoritma Dijkstra menggunakan antrian prioritas yang memiliki operasi untuk memodifikasi nilai dalam antrian. Ini memungkinkan kita untuk memiliki satu instance dari setiap node dalam antrian dan memperbarui jaraknya bila diperlukan. Namun, antrian prioritas perpustakaan standar tidak menyediakan operasi seperti itu, dan implementasi yang agak berbeda biasanya digunakan dalam pemrograman kompetitif.

Idenya adalah untuk selalu menambahkan instance baru dari sebuah node ke antrian prioritas ketika jaraknya berubah. Implementasi algoritma Dijkstra kami menghitung jarak minimum dari node *x* ke semua node lain dari grafik. Graf tersebut disimpan sebagai adjacency list sehingga *adj[a]* selalu mengandung pasangan (*b*,*w)* jika terdapat edge dari node *a* ke node *b* dengan bobot *w*. Antrian prioritas

```
priority_queue<pair<int,int>> q;
```

berisi pasangan bentuk *(−d,x)*, artinya jarak arus ke simpul x adalah d. Jarak array berisi jarak ke setiap node, dan array yang diproses menunjukkan apakah sebuah node telah diproses. Perhatikan bahwa antrian prioritas berisi jarak negatif ke node. Alasan untuk ini adalah bahwa versi default dari antrian prioritas C++ menemukan elemen maksimum, sementara kita ingin menemukan elemen minimum. Dengan mengeksploitasi jarak negatif, kita dapat langsung menggunakan antrian prioritas default.[10] Perhatikan juga bahwa meskipun mungkin ada beberapa contoh dari sebuah

[10] Tentu saja, kita juga dapat mendeklarasikan antrian prioritas pada Bagian 5.2.3 dan menggunakan jarak positif, tetapi implementasinya akan lebih lama.

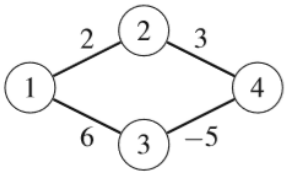
node dalam antrian prioritas, hanya contoh dengan jarak minimum yang akan diproses. Implementasinya adalah sebagai berikut:

```
for (int i = 1; i <= n; i++) {
        distance[i] = INF;
}
distance[x] = 0;
q.push({0,x});
while (!q.empty()) {
        int a = q.top().second; q.pop();
                if ( processed[a]) continue;
                        processed[a] = true;
                for (auto u : adj[a]) {
                int b = u.first, w = u.second;
         if (distance[a]+w < distance[b]) {
                distance[b] = distance[a]+w;
                q.push({-distance[b],b});
                }
        }
}
```

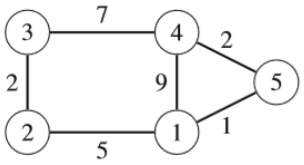
Kompleksitas waktu dari implementasi di atas adalah O(n +m logm), karena algoritma melewati semua node grafik dan menambahkan untuk setiap edge paling banyak satu jarak ke antrian prioritas.

Tepi Negatif

Efisiensi algoritma Dijkstra didasarkan pada kenyataan bahwa graf tersebut tidak memiliki sisi negatif. Namun, jika graf memiliki sisi negatif, algoritme mungkin memberikan hasil yang salah. Sebagai contoh, perhatikan graf pada Gambar 7.21. Jalur terpendek dari node 1 ke node 4 adalah 1→3→4 dan panjangnya adalah 1. Namun, algoritma Dijkstra salah menemukan jalur 1→2→4 dengan mengikuti tepi bobot minimum.



Gambar 7.21 Grafik di mana algoritma Dijkstra gagal



Gambar 7.22 Input untuk algoritma Floyd–Warshall

7.3.3 Algoritma Floyd–Warshall

Algoritma Floyd–Warshall menyediakan cara alternatif untuk mengatasi masalah dalam menemukan jalur terpendek. Tidak seperti algoritma lain dalam bab ini, algoritma ini menemukan jalur terpendek antara semua pasangan simpul dari graf

dalam sekali jalan. Algoritma mempertahankan matriks yang berisi jarak antara node. Matriks awal secara langsung dibangun berdasarkan matriks ketetanggaan dari grafik. Kemudian, algoritme terdiri dari putaran yang berurutan, dan pada setiap putaran, ia memilih node baru yang dapat bertindak sebagai node perantara di jalur mulai sekarang, dan mengurangi jarak menggunakan node ini. Mari kita simulasikan algoritma Floyd–Warshall untuk grafik pada Gambar 7.22. Dalam hal ini, matriks awal adalah sebagai berikut:

$$\begin{matrix} 0 & 5 & \infty & 9 & 1 \\ 5 & 0 & 2 & \infty & \infty \\ \infty & 2 & 0 & 7 & \infty \\ 9 & \infty & 7 & 0 & 2 \\ 1 & \infty & \infty & 2 & 0 \end{matrix}$$

Pada putaran pertama, node 1 adalah node perantara baru. Terdapat jalur baru antara node 2 dan 4 dengan panjang 14, karena node 1 menghubungkannya. Ada juga jalur baru antara node 2 dan 5 dengan panjang 6.

$$\begin{matrix} 0 & 5 & \infty & 9 & 1 \\ 5 & 0 & 2 & \mathbf{14} & \mathbf{6} \\ \infty & 2 & 0 & 7 & \infty \\ 9 & \mathbf{14} & 7 & 0 & 2 \\ 1 & \mathbf{6} & \infty & 2 & 0 \end{matrix}$$

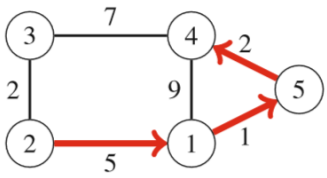
Pada putaran kedua, node2 adalah node perantara baru. Ini menciptakan jalur baru antara node 1 dan 3 dan antara node 3 dan 5:

$$\begin{matrix} 0 & 5 & 7 & 9 & 1 \\ 5 & 0 & 2 & 14 & 6 \\ \mathbf{7} & 2 & 0 & 7 & \mathbf{8} \\ 9 & 14 & 7 & 0 & 2 \\ 1 & 6 & \mathbf{8} & 2 & 0 \end{matrix}$$

Algoritma terus seperti ini, sampai semua node telah ditunjuk node menengah. Setelah algoritma selesai, matriks berisi jarak minimum antara dua node:

$$\begin{matrix} 0 & 5 & 7 & 3 & 1 \\ 5 & 0 & 2 & 8 & 6 \\ 7 & 2 & 0 & 7 & 8 \\ 3 & 8 & 7 & 0 & 2 \\ 1 & 6 & 8 & 2 & 0 \end{matrix}$$

Sebagai contoh, matriks memberitahu kita bahwa jarak terpendek antara node 2 dan 4 adalah 8. Ini sesuai dengan jalur pada Gambar 7.23.



Gambar 7.23 Jalur terpendek dari node 2 ke node 4

Penerapan

Algoritma Floyd–Warshall sangat mudah diimplementasikan. Implementasi di bawah ini membangun matriks jarak di mana dist[a][b]menunjukkan jarak terpendek antara node a dan b. Pertama, algoritma menginisialisasi dist menggunakan adjacency matrix adj dari grafik:

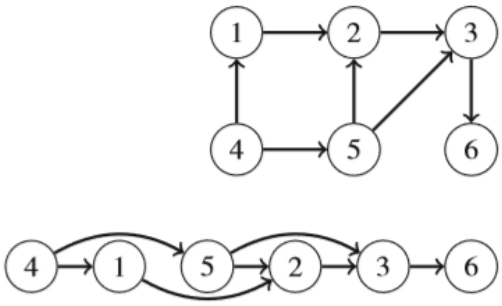
```
for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
```

```
            if (i == j) dist[i][j] = 0;
            else if (adj[i][j]) dist[i][j] = adj[i][j];
            else dist[i][j] = INF;
            }
    }
```

Setelah ini, jarak terpendek dapat ditemukan sebagai berikut:

```
    for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
        dist[i][j]= min(dist[i][j],dist[i][k]+dist[k][j]);
            }
            }
    }
```

Kompleksitas waktu dari algoritme adalah O(n3), karena mengandung tiga loop bersarang yang melewati simpul-simpul graf. Karena implementasi dari algoritma Floyd-Warshall tidak sederhana, algoritma dapat menjadi pilihan yang baik bahkan jika hanya diperlukan untuk menemukan satu jalur terpendek dalam grafik. Namun, algoritme hanya dapat digunakan jika graf sangat kecil sehingga kompleksitas waktu kubik cukup cepat.



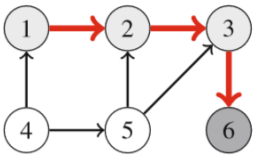
Gambar 7.24 Grafik dan pengurutan topologi

7.4 Grafik Acyclic Berarah

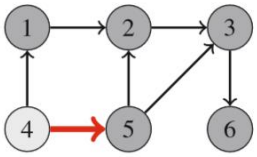
Kelas penting dari graf adalah *graf asiklik terarah*, juga disebut DAG. Grafik seperti itu tidak mengandung siklus, dan banyak masalah yang lebih mudah dipecahkan jika kita berasumsi bahwa ini masalahnya. Secara khusus, kita selalu dapat membangun semacam topologi untuk grafik dan kemudian menerapkan pemrograman dinamis.

7.4.1 Penyortiran Topologis

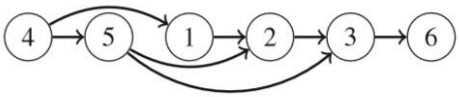
Pengurutan topologi adalah pengurutan simpul-simpul dari suatu graf berarah sedemikian rupa sehingga jika terdapat lintasan dari simpul a ke simpul b, maka simpul a muncul sebelum simpul b dalam pengurutan tersebut. Sebagai contoh, pada Gambar 7.24, satu jenis topologi yang mungkin adalah *[4,1,5,2,3,6].* Graf berarah memiliki jenis topologi yang tepat ketika asiklik. Jika graf berisi siklus, tidak mungkin untuk membentuk semacam topologi, karena tidak ada simpul dari siklus yang dapat muncul sebelum simpul lain dari siklus dalam pengurutan. Ternyata pencarian mendalam-pertama dapat digunakan untuk memeriksa apakah graf berarah berisi siklus dan, jika tidak, untuk membangun semacam topologi.



Gambar 7.25 Pencarian pertama menambahkan node 6, 3, 2, dan 1 ke dalam daftar



Gambar 7.26 Pencarian kedua menambahkan node 5 dan 4 ke dalam daftar



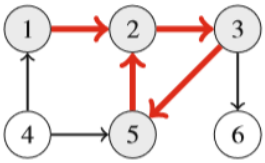
Gambar 7.27 Pengurutan topologi akhir

Idenya adalah menelusuri simpul-simpul grafik dan selalu memulai pencarian mendalam-pertama pada simpul saat ini jika belum diproses. Selama pencarian, node memiliki tiga kemungkinan status:

- state 0: node belum diproses (putih)
- state 1: node sedang diproses (abu-abu muda)
- state 2: node telah diproses (abu-abu tua)

Awalnya, keadaan setiap simpul adalah 0. Ketika pencarian mencapai simpul untuk pertama kalinya, keadaannya menjadi 1. Akhirnya, setelah penjumlahan dari simpul diproses, keadaannya menjadi 2. Jika graf berisi siklus, kita akan menemukan ini selama pencarian, karena cepat atau lambat kita akan sampai dioda yang keadaannya 1. Dalam hal ini, tidak mungkin untuk membuat pengurutan topologi. Jika graf tidak mengandung siklus, kita dapat membuat pengurutan topologi dengan menambahkan setiap nodetoalis ketika keadaannya menjadi 2. Akhirnya, kita membalik daftar dan mendapatkan pengurutan topologi untuk graf.

Sekarang kita siap untuk membangun semacam topologi untuk grafik contoh kita. Pencarian pertama (Gambar 7.25) dilanjutkan dari node 1 ke node 6, dan menambahkan node 6, 3, 2, dan 1 ke daftar. Kemudian, pencarian kedua (Gambar 7.26) dimulai dari node4tonode5dan menambahkan node5dan4ke daftar.Daftar terbalik terakhir adalah [4, 5, 1, 2 ,3, 6], yang sesuai dengan jenis topologi (Gambar 7.27). Perhatikan bahwa jenis topologi tidak unik; dapat ada beberapa macam topologi untuk grafik. Gambar 7.28 menunjukkan graf yang tidak memiliki sortasi topologi. Selama pencarian, terdapat simpul2 yang state-nya 1, yang berarti graf tersebut berisi siklus. Memang ada siklus 2→3→5→2.



Gambar 7.28 Graf ini tidak memiliki pengurutan topologi, karena mengandung siklus

7.4.2 Pemrograman Dinamis

Dengan menggunakan pemrograman dinamis, kita dapat secara efisien menjawab banyak pertanyaan tentang jalur dalam grafik asiklik berarah. Contoh pertanyaan seperti itu adalah:

- Apa jalur terpendek/terpanjang dari simpul a ke simpul b?
- Ada berapa banyak jalan yang berbeda?
- Berapa jumlah tepi minimum/maksimum dalam suatu jalur?
- Node mana yang muncul di setiap jalur yang mungkin?

Perhatikan bahwa banyak dari masalah di atas sulit dipecahkan atau tidak terdefinisi dengan baik untuk graf umum. Sebagai contoh, pertimbangkan masalah menghitung jumlah jalur dari node *a* ke node *b*. Misalkan path (x) menyatakan jumlah path dari node *a* ke node *x*. Sebagai kasus dasar, path(a) =1. Kemudian, untuk menghitung nilai lain dari path (x), kita dapat menggunakan rumus rekursif

paths(x) =paths(s1)+paths(s2)+⋯+paths(sk),

dimana s1,s2,...,sk adalah node dari mana ada tepi ke x. Karena grafiknya asiklik, nilai jalur dapat dihitung dalam urutan topologi. Gambar7.29 menunjukkan nilai jalur dalam skenario contoh di mana kita ingin menghitung jumlah jalur dari node 1 ke node 6. Misalnya,

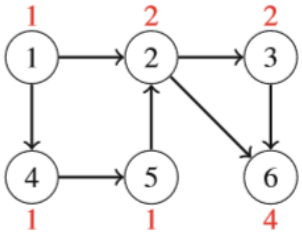
paths(6) =paths(2)+paths(3),

karena edge yang berakhir pada node 6 adalah 2→6 dan 3→ 6. Karena path(2) = 2 dan path(3) =2, kita simpulkan bahwa path(6) =4. Jalan-jalannya adalah sebagai berikut:

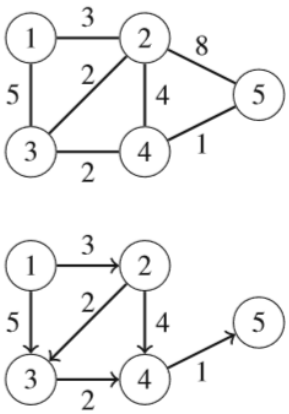
- 1→2→3→6
- 1→2→6
- 1→4→5→2→3→6
- 1→4→5→2→6

Memproses Jalur Terpendek

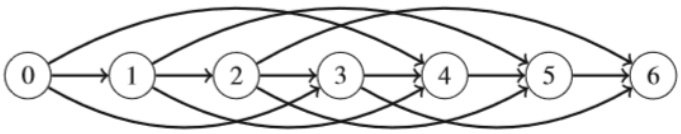
Pemrograman dinamis juga dapat digunakan untuk menjawab pertanyaan tentang jalur terpendek pada grafik umum (tidak harus asiklik). Yaitu, jika kita mengetahui jarak minimum dari node awal ke node lain (misalnya, setelah menggunakan algoritma Dijkstra), kita dapat dengan mudah membuat grafik jalur terpendek asiklik berarah yang menunjukkan untuk setiap node cara yang mungkin untuk mencapai node menggunakan jalur terpendek dari node awal. Misalnya, Gambar 7.30 menunjukkan grafik dan grafik jalur terpendek yang sesuai.



Gambar 7.29 Menghitung jumlah jalur dari node 1 ke node 6



Gambar 7.30 Sebuah graf dan graf jalur terpendeknya



Gambar 7.31 Masalah koin sebagai grafik asiklik berarah

Masalah Koin Ditinjau Kembali

Faktanya, setiap masalah pemrograman dinamis dapat direpresentasikan sebagai grafik asiklik terarah di mana setiap node sesuai dengan keadaan pemrograman dinamis dan tepinya menunjukkan bagaimana keadaan bergantung satu sama lain. Misalnya, pertimbangkan masalah pembentukan jumlah uang n menggunakan koin $\{c_1,c_2,...,c_k\}$ Dalam skenario ini, kita dapat membuat grafik di mana setiap simpul sesuai dengan jumlah uang, dan ujung-ujungnya menunjukkan bagaimana koin dapat dipilih. Sebagai contoh, Gambar 7.31 menunjukkan grafik untuk koin {1,3,4} dan n = 6. Dengan menggunakan representasi ini, jalur terpendek dari node 0 ke node n sesuai dengan solusi dengan jumlah koin minimum, dan jumlah total jalur dari simpul 0 ke simpul n sama dengan jumlah total solusi.

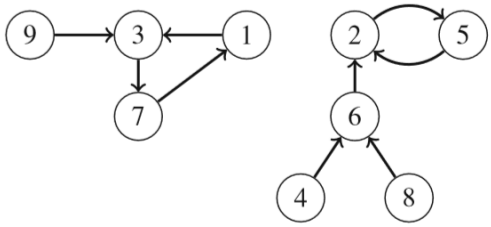
7.5 Grafik Pengganti

Kelas khusus lain dari graf berarah adalah graf penerus/pengganti. Dalam grafik tersebut, derajat keluar setiap simpul adalah 1, yaitu, setiap simpul memiliki penerus yang unik. Graf penerus terdiri dari satu atau lebih komponen, yang masing-masing berisi satu siklus dan beberapa jalur yang mengarah ke sana. Graf penerus kadang-kadang disebut graf

fungsional, karena setiap graf penerus berkorespondensi dengan fungsi succ(x) yang mendefinisikan sisi-sisi dari graf. Parameter x adalah simpul dari grafik, dan fungsinya memberikan penerus dari simpul tersebut. Misalnya, fungsi

$$\begin{array}{l|l} x & 123456789 \\ \hline succ(x) & 357622163 \end{array}$$

mendefinisikan grafik pada Gambar. 7.32.



Gambar 7.32 Grafik penerus

Gambar 7.33 Berjalan dalam graf penerus

7.5.1 Menemukan Pengganti

Karena setiap simpul dari graf penerus memiliki penerus yang unik, kita juga dapat mendefinisikan fungsi succ(x,k) yang memberikan simpul yang akan kita capai jika web mulai pada simpul x dan berjalan k langkah ke depan. Misalnya, dalam contoh grafik succ(4,6) =2, karena kita akan mencapai simpul 2 dengan berjalan 6 langkah dari simpul 4 (Gambar 7.33). Jalan lurus ke depan untuk menghitung nilai succ(x,k) adalah untuk memulai pada simpul x dan berjalan k langkah ke depan, yang membutuhkan waktu O(k). Namun, dengan menggunakan preprocessing, setiap nilai succ(x,k) dapat dihitung hanya dalam waktu O(logk). Biarkan u menunjukkan jumlah langkah maksimum yang akan kita jalani. Idenya adalah untuk menghitung semua nilai succ(x,k) di mana k adalah pangkat dua dan paling banyak u. Ini dapat dilakukan secara efisien, karena kita dapat menggunakan pengulangan berikut:

$$succ(x,k) = \begin{cases} succ(x) & k = 1 \\ succ(x, k/2), k/2) & k > 1 \end{cases}$$

Idenya adalah bahwa jalur dengan panjang k yang dimulai pada simpul x dapat dibagi menjadi dua jalur dengan panjang k/2. Menghitung semua nilai succ(x,k) di mana k adalah pangkat dua dan paling banyak u membutuhkan waktu O(n logu), karena nilai O(logu) dihitung untuk setiap node. Dalam grafik contoh kami, nilai pertama adalah sebagai berikut:

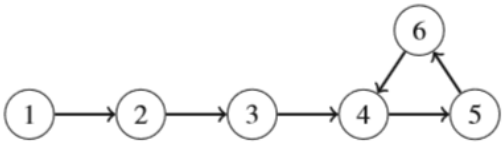
x	1 2 3 4 5 6 7 8 9
succ(x,1)	3 5 7 6 2 2 1 6 3
succ(x,2)	7 2 1 2 5 5 3 2 7
succ(x,4)	3 2 7 2 5 5 1 2 3
succ(x,8)	7 2 1 2 5 5 3 2 7
...	

Setelah praperhitungan, setiap nilai succ(x,k) dapat dihitung dengan menyajikan k sebagai jumlah dari kekuatan lunak dua. Penyajian seperti itu selalu terdiri dari O(logk) bagian, jadi menghitung nilai succ(x,k) membutuhkan waktu O(logk). Misalnya, jika kita ingin menghitung nilai succ(x,11), kita menggunakan rumus

succ(x,11) =succ(succ(succ(x,8),2),1).

Dalam grafik contoh,

succ(4,11) =succ(succ(succ(4,8),2),1) =5.



Gambar 7.34 Siklus dalam graf penerus

7.5.2 Deteksi Siklus

Pertimbangkan graf penerus yang hanya berisi jalur yang berakhir dalam satu siklus. Kita dapat mengajukan pertanyaan berikut: jika kita memulai perjalanan kita di node awal, apa node pertama dalam siklus dan berapa banyak node yang terkandung dalam siklus? Sebagai contoh, pada Gambar 7.34, kita mulai berjalan di node 1, node pertama yang termasuk dalam siklus adalah node 4, dan siklus terdiri dari tiga node (4, 5, dan 6). Cara sederhana untuk mendeteksi siklus adalah berjalan di dalam graf dan melacak titik jatuh yang telah dikunjungi. Setelah sebuah node dikunjungi untuk kedua kalinya, kita dapat menyimpulkan bahwa node tersebut adalah node pertama dalam siklus.

Metode ini bekerja dalam waktu O(n) dan juga menggunakan memori O(n). Namun, ada algoritma yang lebih baik untuk deteksi siklus. Kompleksitas waktu dari algoritma tersebut masih O(n), tetapi mereka hanya menggunakan memori O(1), yang mungkin merupakan peningkatan penting jika n besar. Salah satu algoritma tersebut adalah algoritma Floyd, yang berjalan di grafik menggunakan dua pointer a dan b. Kedua pointer dimulai pada node awal x. Kemudian, pada setiap belokan, penunjuk a berjalan satu langkah ke depan dan penunjuk b berjalan dua langkah ke depan. Proses berlanjut sampai pointer bertemu satu sama lain:

```
a = succ(x);
b = succ(succ(x));
while (a != b) {
        a = succ(a);
        b = succ(succ(b));
}
```

Pada titik ini, penunjuk a telah berjalan k langkah dan penunjuk b telah berjalan 2k langkah, sehingga panjang siklus membagi k. Dengan demikian, simpul pertama yang termasuk dalam siklus dapat ditemukan dengan memindahkan penunjuk a ke simpul x dan memajukan penunjuk selangkah demi selangkah hingga bertemu kembali.

```
a = x;
while (a != b) {
        a = succ(a);
        b = succ(b);
}
first = a;
```

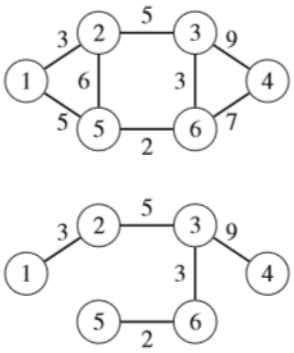
Setelah ini, panjang siklus dapat dihitung sebagai berikut:

```
b = succ(a);
length = 1;
while (a != b) {
        b = succ(b);
```

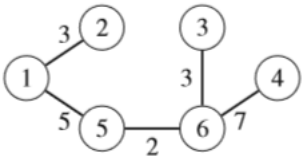
```
            length++;
        }
```

7.6 Pohon Rentang Minimum

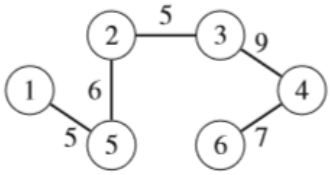
Sebuah pohon merentang berisi semua simpul dari suatu graf dan beberapa sisinya sehingga ada jalur antara dua simpul. Seperti pohon pada umumnya, spanning tree saling berhubungan dan asiklik. Bobot spanning tree adalah jumlah bobot edge-nya. Sebagai contoh, Gambar 7.35 menunjukkan grafik dan salah satu pohon merentangnya. Bobot pohon merentang ini adalah 3+5+9+3+2=22. Pohon merentang minimum adalah pohon merentang yang bobotnya sekecil mungkin. Gambar 7.36 menunjukkan pohon merentang minimum untuk grafik contoh kita dengan bobot 20. Dengan cara yang sama, pohon merentang maksimum adalah pohon merentang yang bobotnya sebesar mungkin. Gambar 7.37 menunjukkan pohon merentang maksimum untuk graf contoh kita dengan bobot 32. Perhatikan bahwa suatu graf mungkin memiliki beberapa pohon merentang minimum dan maksimum, sehingga pohonnya tidak unik.



Gambar 7.35 Graf dan pohon merentang



Gambar 7.36 Pohon merentang minimum dengan bobot 20



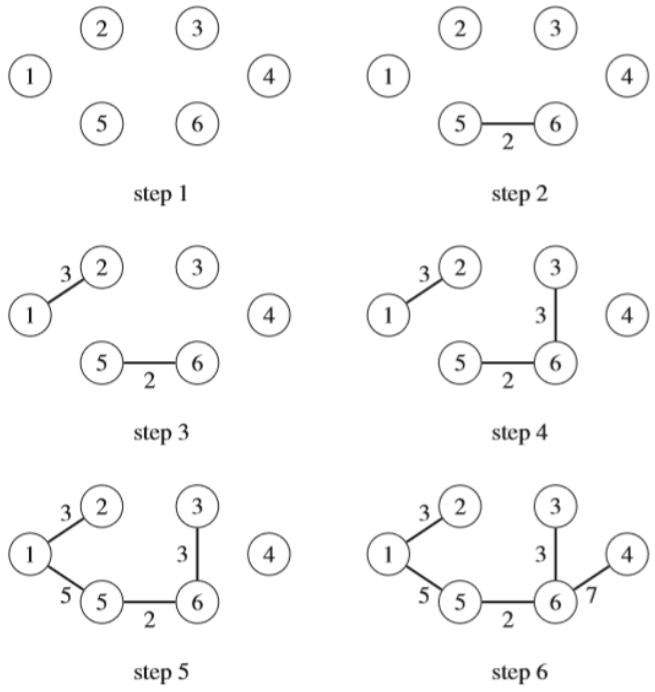
Gambar 7.37 Pohon merentang maksimum dengan bobot 32

Ternyata beberapa metode serakah dapat digunakan untuk membangun pohon merentang minimum dan maksimum. Bagian ini membahas dua algoritme yang memproses tepi-tepi graf yang diurutkan berdasarkan bobotnya. Kami berfokus pada menemukan pohon merentang minimum, tetapi algoritme yang sama juga dapat menemukan pohon merentang maksimum dengan memproses tepi dalam urutan terbalik.

7.6.1 Algoritma Kruskal

Algoritma Kruskal membangun pohon merentang minimum dengan menambahkan tepi ke grafik dengan rakus. Pohon merentang awal hanya berisi simpul-simpul dari graf dan tidak mengandung sisi apapun. Kemudian algoritma melewati tepi yang diurutkan berdasarkan bobotnya dan selalu menambahkan tepi ke grafik jika tidak membuat siklus. Algoritma mempertahankan komponen grafik. Awalnya, setiap simpul dari grafik milik komponen yang terpisah. Selalu ketika tepi ditambahkan ke grafik, dua komponen bergabung. Akhirnya, semua node termasuk dalam komponen yang sama, dan pohon merentang minimum telah ditemukan. Sebagai contoh, mari kita buat pohon merentang minimum untuk grafik contoh kita (Gambar 7.35). Langkah pertama adalah mengurutkan tepi dalam urutan kenaikan bobotnya:

Berat	edge
5-6	2
1-2	3
3-6	3
1-5	5
2-3	5
2-5	6
4-6	7
3-4	9



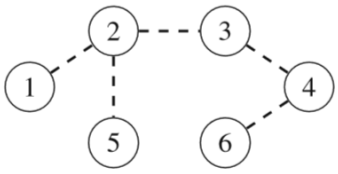
Gambar 7.38 Algoritma Kruskal

Kemudian, kita menelusuri daftar dan menambahkan setiap sisi ke grafik jika bergabung dengan dua komponen yang terpisah. Gambar 7.38 menunjukkan langkah-langkah dari algoritma. Awalnya, setiap node milik komponennya sendiri. Kemudian, tepi pertama pada daftar (5–6, 1–2, 3–6, dan 1-5) ditambahkan ke grafik. Setelah ini, tepi berikutnya akan menjadi 2-3, tetapi tepi ini tidak ditambahkan, karena akan

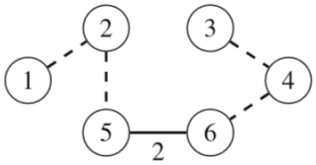
membuat siklus. Hal yang sama berlaku untuk tepi 2–5. Akhirnya, tepi 4–6 ditambahkan, dan pohon merentang minimum sudah siap.

Mengapa Ini Bekerja?

Ini adalah pertanyaan bagus mengapa algoritma Kruskal bekerja. Mengapa strategi serakah menjamin bahwa kita akan menemukan pohon merentang minimum? Mari kita lihat apa yang menguntungkantepi bobot minimum dari grafik yang tidak termasuk dalam pohon merentang. Sebagai contoh, anggaplah pohon merentang minimum dari grafik contoh kita tidak akan berisi tepi bobot minimum 5–6. Kita tidak tahu struktur pasti dari pohon merentang seperti itu, tetapi bagaimanapun juga pohon itu harus mengandung beberapa sisi. Asumsikan bahwa pohon akan terlihat seperti pohon pada Gambar 7.39. Namun, tidak mungkin bahwa pohon pada Gambar 7.39 akan menjadi pohon merentang minimum, karena kita dapat menghilangkan tepi dari pohon dan menggantinya dengan tepi dengan bobot minimum 5–6. Ini menghasilkan pohon merentang yang bobotnya lebih kecil, ditunjukkan pada Gambar 7.40. Untuk alasan ini, selalu optimal untuk memasukkan tepi bobot minimum di pohon untuk menghasilkan pohon merentang minimum. Dengan menggunakan argumen serupa, kita dapat menunjukkan bahwa menambahkan tepi berikutnya dalam urutan bobot ke pohon juga optimal, dan seterusnya. Oleh karena itu, algoritma Kruskal selalu menghasilkan pohon merentang minimum.



Gambar 7.39 Sebuah pohon merentang minimum hipotetis



Gambar 7.40 Menyertakan tepi 5–6 mengurangi bobot pohon merentang

Penerapan

Saat menerapkan algoritma Kruskal, akan lebih mudah untuk menggunakan representasi daftar tepi dari grafik. Fase pertama dari algoritma mengurutkan tepi dalam daftar dalam waktu O(m log m). Setelah ini, tahap kedua dari algoritma membangun pohon merentang minimum sebagai berikut:

```
for (...) {
if (!same(a,b)) unite(a,b);
}
```

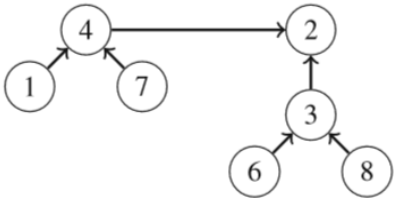
Loop melewati tepi dalam daftar dan selalu memproses tepi (a,b) di mana a dan b adalah dua node. Dibutuhkan dua fungsi: fungsi yang sama menentukan apakah a dan b berada dalam komponen yang sama, dan fungsi bersatu menggabungkan komponen yang mengandung a dan b. Masalahnya adalah bagaimana secara efisien mengimplementasikan fungsi-fungsi yang sama dan bersatu. Salah satu kemungkinan adalah untuk mengimplementasikan fungsi yang sama sebagai traversal grafik dan

memeriksa apakah kita bisa mendapatkan dari node a ke node b. Namun, kompleksitas waktu dari fungsi tersebut akan menjadi O(n+m) dan algoritma yang dihasilkan akan lebih rendah, karena fungsi yang sama akan dipanggil untuk setiap sisi dalam grafik. Kami akan memecahkan masalah menggunakan struktur union-find yang mengimplementasikan kedua fungsi dalam waktu O(log n). Dengan demikian, kompleksitas waktu dari algoritma Kruskal akan menjadi O(m log n) setelah mengurutkan daftar tepi.

7.6.2 Struktur Temukan Serikat

Struktur union-find memelihara kumpulan set. Himpunan itu saling lepas, jadi tidak ada elemen yang termasuk lebih dari satu himpunan. Dua operasi waktu O(log n) didukung: operasi bersatu menggabungkan dua himpunan, dan operasi menemukan menemukan perwakilan dari himpunan yang berisi elemen tertentu. Dalam struktur union-find, satu elemen di setiap himpunan adalah perwakilan dari himpunan, dan ada jalur dari elemen lain dari himpunan ke perwakilan. Misalnya, asumsikan bahwa himpunannya adalah {1,4,7}, {5} dan {2,3,6,8}. Gambar 7.41 menunjukkan satu cara untuk merepresentasikan himpunan ini. Dalam hal ini perwakilan dari himpunan adalah 4,5, dan 2. Kita dapat menemukan perwakilan dari elemen apa pun dengan mengikuti jalur yang dimulai pada elemen tersebut. Misalnya, elemen 2 adalah perwakilan untuk elemen 6, karena kita mengikuti jalur 6→3→2.Dua elemen termasuk dalam kelompok yang sama persis ketika perwakilannya sama.

Gambar 7.41 Struktur union-find dengan tiga set



Gambar 7.42 Menggabungkan dua set menjadi satu set

Untuk menggabungkan dua set, perwakilan dari satu set terhubung ke perwakilan dari set lainnya. Misalnya, Gambar 7.42 menunjukkan cara yang mungkin untuk menggabungkan himpunan{1,4,7} dan {2,3,6,8}. Dari sini, elemen 2 adalah perwakilan untuk seluruh himpunan dan perwakilan lama 4 menunjuk ke elemen 2. Efisiensi struktur temukan serikat tergantung pada bagaimana himpunan digabungkan. Ternyata kita dapat mengikuti strategi sederhana: selalu hubungkan perwakilan dari himpunan yang lebih kecil ke perwakilan dari himpunan yang lebih besar (atau jika himpunan memiliki ukuran yang sama, kita dapat membuat pilihan yang sewenang-wenang). Dengan menggunakan strategi ini, panjang jalur apa pun akan menjadi O(log n), sehingga kita dapat menemukan perwakilan elemen apa pun secara efisien dengan mengikuti jalur yang sesuai.

Penerapan

Struktur union-find dapat dengan mudah diimplementasikan menggunakan array. Dalam implementasi berikut, tautan larik menunjukkan untuk setiap elemen elemen berikutnya di jalur, atau elemen itu sendiri jika itu adalah perwakilan, dan ukuran larik menunjukkan untuk setiap perwakilan ukuran set yang sesuai. Awalnya, setiap elemen milik set terpisah:

```
for (int i = 1; i <= n; i++) link[i] = i;
for (int i = 1; i <= n; i++) size[i] = 1;
```

Fungsi find mengembalikan perwakilan untuk elemen x. Perwakilan dapat ditemukan dengan mengikuti jalan yang dimulai di x.

```
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}
```

Fungsi yang sama memeriksa apakah elemen a dan b termasuk dalam himpunan yang sama. Ini dapat dengan mudah dilakukan dengan menggunakan fungsi find:

```
bool same(int a, int b) {
return find(a) == find(b);
}
```

Fungsi bersatu bergabung dengan himpunan yang berisi elemen a dan b (elemen harus dalam himpunan yang berbeda). Fungsi tersebut pertama-tama menemukan perwakilan dari himpunan dan kemudian menghubungkan himpunan yang lebih kecil ke himpunan yang lebih besar.

```
void unite(int a, int b) {
    a = find(a);
    b = find(b);
    if (size[a] < size[b]) swap(a,b);
    size[a] += size[b];
    link[b] = a;
}
```

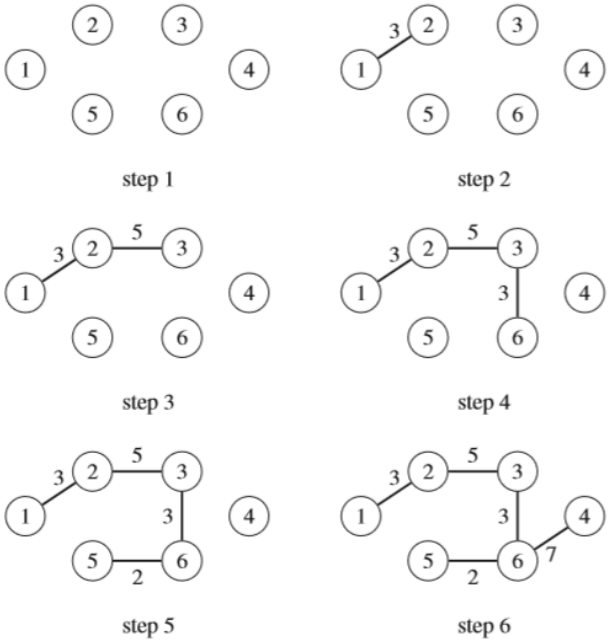
Kompleksitas waktu dari fungsi find adalah O(log n) dengan asumsi bahwa panjang setiap jalur adalah O(log n). Dalam hal ini, fungsi-fungsi yang sama dan bersatu juga bekerja dalam waktu O(log n). Fungsi bersatu memastikan bahwa panjang setiap jalur adalah O(log n) dengan menghubungkan himpunan yang lebih kecil ke himpunan yang lebih besar.

Kompresi Jalur

Berikut adalah cara alternatif untuk mengimplementasikan operasi find:

```
int find(int x) {
    if (x == link[x])
    return x; return link[x] = find(link[x]);
}
```

Fungsi ini menggunakan kompresi jalur: setiap elemen di jalur akan langsung menunjuk ke perwakilannya setelah operasi. Dapat ditunjukkan bahwa dengan menggunakan fungsi ini, operasi union-find bekerja dalam waktu O(α(n)) diamortisasi, di mana α(n) adalah invers fungsi Ackermann yang tumbuh sangat lambat (hampir konstan). Namun, kompresi jalur tidak dapat digunakan dalam beberapa aplikasi struktur union-find, seperti dalam algoritma konektivitas dinamis.



Gambar 7.43 Algoritma Prim

7.6.3 Algoritma Prim

Algoritma Prim merupakan metode alternatif untuk membangun pohon merentang minimum. Algoritme pertama-tama menambahkan simpul arbitrer ke pohon, dan kemudian selalu memilih tepi bobot minimum yang menambahkan simpul baru ke pohon. Akhirnya, semua node telah ditambahkan dan pohon merentang minimum telah ditemukan. Algoritma Prim mirip dengan algoritma Dijkstra. Perbedaannya adalah bahwa algoritma Dijkstra selalu memilih node yang jaraknya dari node awal minimum, tetapi algoritma Prim hanya memilih node yang dapat ditambahkan ke pohon menggunakan tepi bobot minimum. Sebagai contoh, Gambar 7.43 menunjukkan bagaimana algoritma Prim membangun pohon merentang minimum untuk grafik contoh kita, dengan asumsi bahwa node awal adalah node 1. Seperti algoritma Dijkstra, algoritma Prim dapat diimplementasikan secara efisien menggunakan antrian prioritas. Antrian prioritas harus berisi semua node yang dapat dihubungkan ke komponen saat ini menggunakan tepi tunggal, dalam urutan bobot tepi yang sesuai. Kompleksitas waktu algoritma Prim adalah O(n+m logm) yang sama dengan kompleksitas waktu algoritma Dijkstra. Dalam praktiknya, algoritme Prim dan Kruskal keduanya efisien, dan pilihan algoritme adalah masalah selera. Namun, sebagian besar programmer kompetitif menggunakan algoritma Kruskal.

BAB 8
TOPIK DESAIN ALGORITMA

8.1 Algoritma Bit-Paralel

Algoritma bit-paralel didasarkan pada fakta bahwa bit angka individu dapat dimanipulasi secara paralel menggunakan operasi bit. Dengan demikian, cara untuk merancang algoritma yang efisien adalah dengan merepresentasikan langkah-langkah algoritma sehingga dapat diimplementasikan secara efisien menggunakan operasi bit.

8.1.1 Jarak Hamming

Jarak Hamming Hamming (a,b) antara dua senar a dan b dengan panjang yang sama adalah jumlah posisi di mana senar berbeda. Sebagai contoh,

hamming(01101,11001) =2.

Pertimbangkan masalah berikut: Diberikan n bit string, masing-masing panjang k, hitung jarak Hamming minimum antara dua string. Misalnya, jawaban untuk [00111,01101,11110]adalah 2, karena

- hamming(00111,01101) =2,
- hamming(00111,11110) =3,
- hamming(01101,11110) =3.

Cara langsung untuk menyelesaikan masalah ini adalah melalui semua pasangan string dan menghitung jarak Hamming mereka, yang menghasilkan algoritma waktu $O(n^2k)$.Fungsi berikut menghitung jarak antara string a dan b:

```
int hamming(string a, string b) {
        int d = 0;
        for (int i = 0; i < k; i++) {
        if (a[i] != b[i]) d++;
        }
        return d;
}
```

Namun, karena string terdiri dari bit, kita dapat mengoptimalkan solusi dengan menyimpan string sebagai bilangan bulat dan menghitung jarak menggunakan operasi bit. Secara khusus, jika k ≤ 32, kita dapat menyimpan string sebagai nilai int dan menggunakan fungsi berikut untuk menghitung jarak:

```
int hamming(int a, int b) {
return __builtin_popcount(a^b);
}
```

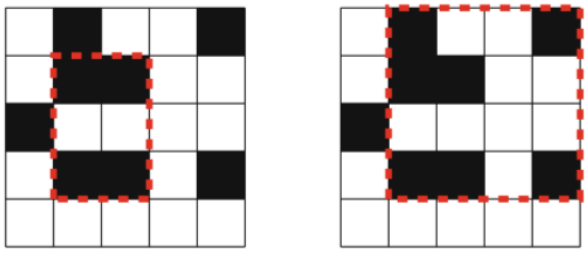
Pada fungsi di atas, operasi xor membangun sebuah string yang memiliki satu bit pada posisi di mana a dan b berbeda. Kemudian, jumlah satu bit dihitung menggunakan fungsi __builtin_popcount. Tabel 8.1 menunjukkan perbandingan waktu berjalan dari algoritma asli dan algoritma bit-paralel pada komputer modern. Dalam masalah ini, algoritma bit-paralel sekitar 20 kali lebih cepat dari algoritma aslinya.

8.1.2 Menghitung Subgrid

Sebagai contoh lain, perhatikan masalah berikut: Diberikan sebuah grid n × n yang setiap kotaknya berwarna hitam (1) atau putih (0), hitung jumlah subgrid yang semua sudutnya berwarna hitam. Sebagai contoh, Gambar 8.1 menunjukkan dua subgrid tersebut dalam sebuah grid.

Tabel 8.1 Waktu berjalan dari algoritma saat menghitung jarak Hamming minimum dari n bit string dengan panjang k =30

Ukuran *n*	**Algiritma asli (s)**	**Algorima bit-paralel (s)**
5000	0.84	0.06
10000	3.24	0.`8
15000	7.23	0.37
20000	12.79	0.63
25000	19.99	0.97



Gambar 8.1 Grid ini berisi dua subgrid dengan sudut hitam

Ada algoritma waktu $O(n^3)$ untuk menyelesaikan masalah: lihat semua pasangan baris $O(n^2)$, dan untuk setiap pasangan(a,b) hitung, dalam waktu O(n), jumlah kolom yang berisi a kotak hitam di kedua baris a dan b. Kode berikut mengasumsikan bahwa warna[y][x]menunjukkan warna pada baris y dan kolom x:

```
int count = 0;
for (int i = 0; i < n; i++) {
        if (color[a][i] == 1 && color[b][i] == 1) {
        count++;
        }
}
```

Kemudian, setelah mengetahui bahwa ada kolom hitung yang kedua kotaknya berwarna hitam, kita dapat menggunakan rumus count(count−1)/2 untuk menghitung jumlah subgrid yang baris pertamanya a dan baris terakhir b. Untuk membuat algoritma bit-paralel, kami merepresentasikan setiap baris(row) k sebagai bit set n-bit row[k] di mana satu bit menunjukkan kotak hitam. Kemudian, kita dapat menghitung jumlah kolom di mana baris a dan b keduanya memiliki kotak hitam menggunakan operasi an dan dan menghitung jumlah satu bit. Ini dapat dengan mudah dilakukan sebagai berikut menggunakan struktur bitset:

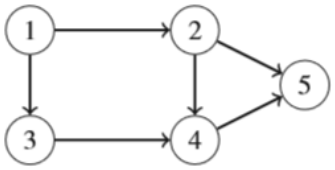
```
int count = (row[a]&row[b]).count();
```

Tabel 8.2 menunjukkan perbandingan algoritma asli dan algoritma bit-paralel untuk ukuran grid yang berbeda. Perbandingan menunjukkan bahwa algoritma bit-paralel bisa sampai 30 kali lebih cepat dari algoritma aslinya.

Tabel 8.2 Waktu berjalan dari algoritma untuk menghitung subgrid

Ukuran Grid *n*	Algiritma asli (s)	Algorima bit-paralel (s)
1000	0.65	0.05
1500	2.17	0.14
2000	5.51	0.30

2500	12.67	0.52
300	26.36	0.87



Gambar 8.2 Grafik dan nilai jangkauannya. Misalnya, reach(2) =3, karena node 2, 4, dan 5 dapat dicapai dari node 2

8.1.3 Keterjangkauan dalam Grafik

Diberikan graf asiklik terarah dari n simpul, pertimbangkan masalah penghitungan untuk setiap simpul x suatu nilai reach(x): jumlah simpul yang dapat dicapai dari simpul x. Misalnya, Gambar 8.2 menunjukkan grafik dan nilai jangkauannya. Masalah tersebut dapat diselesaikan dengan menggunakan pemrograman dinamis dalam waktu $O(n^2)$ dengan membuat untuk setiap node sebuah daftar node yang dapat dicapai darinya. Kemudian, untuk membuat algoritma bit-paralel, kami merepresentasikan setiap daftar sebagai bitset dari n bit. Hal ini memungkinkan kita untuk secara efisien menghitung gabungan dari dua daftar tersebut menggunakan operasi atau. Dengan asumsi bahwa jangkauan adalah array dari struktur bitset dan grafik disimpan sebagai daftar adjacency di adj, perhitungan untuk node x dapat dilakukan sebagai berikut:

```
reach[x][x] = 1;
for (auto u : adj[x]) {
reach[x] |= reach[u];
}
```

Tabel 8.3 menunjukkan beberapa waktu berjalan untuk algoritma bit-paralel. Dalam setiap pengujian, graf memiliki n simpul dan 2n sisi acak a → b dimana a < b. Perhatikan bahwa algoritme menggunakan sejumlah besar memori untuk nilai n yang besar. Dalam banyak kontes, batas memori mungkin 512MB atau lebih rendah.

Tabel 8.3 Waktu berjalan dari algoritme saat menghitung node yang dapat dijangkau dalam grafik

Ukuran Grid *n*	Running Time (s)	Memori yang digunakan (MB)
2×10^4	0.06	50
4×10^4	0.17	200
6×10^4	0.32	450
8×10^4	0.51	800
10^5	0.78	1250

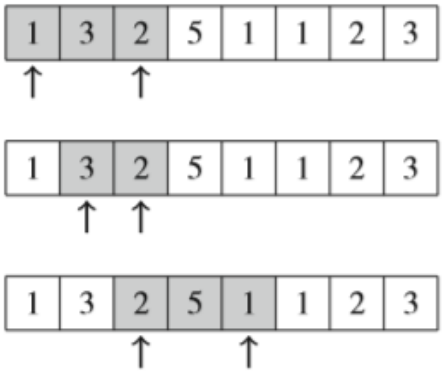
8.2 Analisis Diamortisasi

Struktur algoritma sepuluh secara langsung memberi tahu kita kompleksitas waktunya, tetapi terkadang analisis langsung tidak memberikan gambaran efisiensi yang sebenarnya. Analisis diamortisasi dapat digunakan untuk menganalisis urutan operasi yang kompleksitas waktunya bervariasi. Idenya adalah untuk memperkirakan total waktu yang digunakan untuk semua operasi tersebut selama algoritma, alih-alih berfokus pada operasi individu.

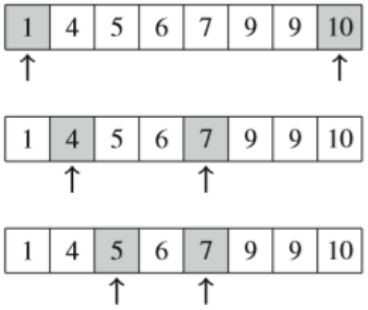
8.2.1 Metode Dua Pointer

Dalam metode dua pointer, dua pointer berjalan melalui array. Kedua pointer bergerak ke satu arah saja, yang memastikan bahwa algoritma bekerja secara efisien. Sebagai contoh pertama tentang bagaimana menerapkan teknik ini, pertimbangkan sebuah masalah di mana kita diberikan sebuah array dari n bilangan bulat positif dan jumlah target x, dan kita ingin menemukan sebuah subarray yang jumlahnya adalah x atau melaporkan bahwa tidak ada subarray tersebut. Soal tersebut dapat diselesaikan dalam waktu O(n) dengan menggunakan metode two pointers. Idenya tetap memiliki pointer yang menunjuk ke nilai terakhir dari subarray. Setiap belokan, pointer kiri bergerak satu langkah ke kanan, dan pointer kanan bergerak ke kanan selama jumlah subarray yang dihasilkan paling banyak x. Jika jumlahnya menjadi tepat x, solusi telah ditemukan.

Sebagai contoh, Gambar 8.3 menunjukkan bagaimana algoritma memproses array ketika jumlah target adalah x = 8. Subarray awal berisi nilai 1, 3, dan 2, yang jumlahnya adalah 6. Kemudian, pointer kiri bergerak satu langkah ke kanan, dan penunjuk kanan tidak bergerak, karena jika tidak, jumlahnya akan melebihi x. Akhirnya, penunjuk kiri bergerak satu langkah ke kanan, dan penunjuk kanan bergerak dua langkah ke kanan. Jumlah subarray adalah 2+5+1=8, jadi subarray yang diinginkan telah ditemukan. Waktu berjalan dari algoritma tergantung pada jumlah langkah pointer kanan bergerak. Meskipun tidak ada batas atas yang berguna tentang berapa banyak langkah yang dapat digerakkan oleh penunjuk pada satu putaran, kita tahu bahwa penunjuk menggerakkan total O(n) langkah selama algoritme, karena hanya bergerak ke kanan. Karena penunjuk kiri dan kanan menggerakkan O(n) langkah, algoritme bekerja dalam waktu O(n).



Gambar 8.3 Menemukan subarray dengan jumlah 8 menggunakan metode dua pointer



Gambar 8.4 Memecahkan masalah 2SUM menggunakan metode dua pointer

Masalah 2SUM

Masalah lain yang dapat diselesaikan dengan menggunakan metode dua pointer adalah *masalah 2SUM*: diberikan array n angka dan jumlah target x, temukan dua nilai array sedemikian rupa sehingga jumlahnya adalah x, atau laporkan bahwa tidak ada nilai seperti itu. Untuk memecahkan masalah, pertama-tama kita mengurutkan nilai array dalam urutan yang meningkat. Setelah itu, kita melakukan iterasi melalui array menggunakan dua pointer. Penunjuk kiri dimulai dari nilai pertama dan bergerak satu langkah ke kanan pada setiap belokan. Penunjuk kanan dimulai dari nilai terakhir dan selalu bergerak ke kiri sampai jumlah nilai kiri dan kanan paling banyak x. Jika jumlahnya tepat x, solusi telah ditemukan. Sebagai contoh, Gambar 8.4 menunjukkan bagaimana algoritma memproses sebuah array ketika jumlah target adalah x =12. Pada posisi awal, jumlah nilai adalah 1+10=11 yang lebih kecil dari x. Kemudian penunjuk kiri bergerak satu langkah ke kanan, dan penunjuk kanan bergerak tiga langkah ke kiri, dan jumlahnya menjadi 4+7=11. Setelah ini, penunjuk kiri bergerak satu langkah ke kanan lagi. Penunjuk kanan tidak bergerak, dan solusi 5+7=12 telah ditemukan.

Waktu berjalan dari algoritma adalah *O(n log n)*, karena pertama-tama mengurutkan array dalam waktu *O(n log n),* dan kemudian kedua pointer memindahkan *O(n)* langkah. Perhatikan bahwa pemecahan masalah juga dimungkinkan dengan cara lain dalam waktu *O(n log n)* menggunakan pencarian biner. Dalam solusi seperti itu, pertama-tama kita mengurutkan array dan kemudian melakukan iterasi melalui nilai-nilai array dan untuk setiap nilai biner mencari nilai lain yang menghasilkan jumlah *x*.

Faktanya, banyak masalah yang dapat diselesaikan dengan menggunakan metode dua pointer juga dapat diselesaikan dengan menggunakan struktur pengurutan atau himpunan, terkadang dengan faktor logaritma tambahan. Masalah *kSUM* yang lebih umum juga menarik. Dalam soal ini kita harus mencari k elemen sedemikian rupa sehingga jumlah mereka adalah *x*. Ternyata kita bisa menyelesaikan masalah 3SUM dalam waktu $O(n^2)$ dengan memperluas algoritma 2SUM di atas. Dapatkah Anda melihat bagaimana kita bisa melakukannya? Untuk waktu yang lama, sebenarnya dipikirkan bahwa $O(n^2)$ akan menjadi kompleksitas waktu terbaik untuk masalah 3SUM. Namun, pada tahun 2014, Grønlund dan Pettie menunjukkan bahwa ini tidak terjadi.

8.2.2 Elemen Kecil Terdekat

Analisis diamortisasi sering digunakan untuk memperkirakan jumlah operasi yang dilakukan pada struktur data. Operasi mungkin terdistribusi tidak merata sehingga sebagian besar operasi terjadi selama fase tertentu dari algoritma, tetapi jumlah total operasi terbatas. Sebagai contoh, misalkan kita ingin mencari untuk setiap elemen larik elemen terkecil terdekat, yaitu, elemen kecil pertama yang mendahului elemen dalam array. Ada kemungkinan bahwa tidak ada elemen seperti itu, dalam hal ini algoritme harus melaporkan ini.

Selanjutnya kita akan memecahkan masalah secara efisien menggunakan struktur tumpukan. Telusuri larik dari kiri ke kanan dan pertahankan setumpuk elemen larik. Pada setiap posisi larik, kami menghapus elemen dari tumpukan hingga elemen teratas lebih kecil dari elemen saat ini, atau tumpukan kosong. Kemudian, kami melaporkan bahwa elemen teratas adalah elemen terkecil terdekat dari elemen saat ini, atau jika tumpukan kosong, tidak ada elemen seperti itu. Akhirnya, kami menambahkan elemen saat ini ke tumpukan.

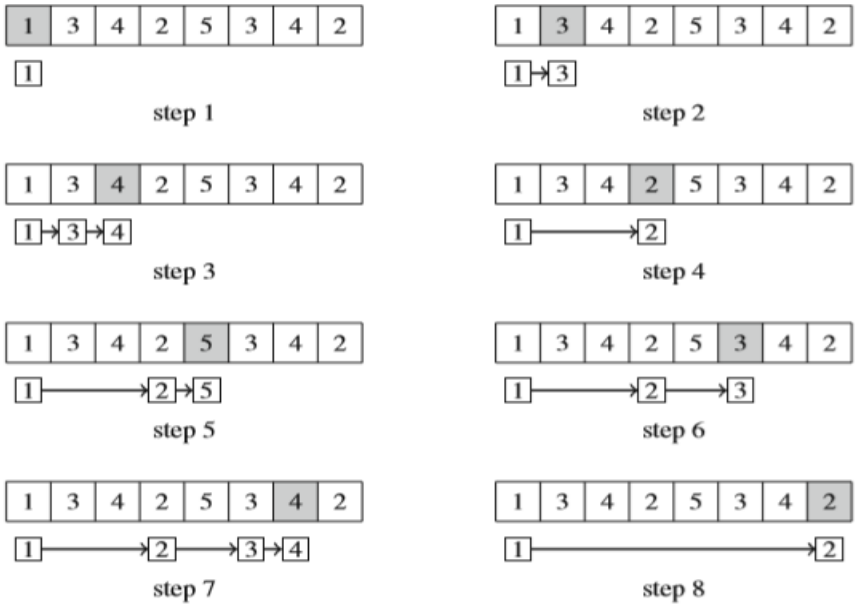
Gambar 8.5 menunjukkan bagaimana algoritma memproses sebuah array. Pertama, elemen 1 ditambahkan ke tumpukan. Karena merupakan elemen pertama dalam array,

jelas tidak memiliki elemen terdekat yang lebih kecil. Setelah ini, elemen 3 dan 4 ditambahkan ke tumpukan. Elemen terkecil dari 4 adalah 3, dan elemen terkecil dari 3 adalah 1. Kemudian, elemen 2 berikutnya lebih kecil dari dua elemen teratas dalam tumpukan, sehingga elemen 3 dan 4 dikeluarkan dari tumpukan. Jadi, elemen terkecil dari 2 yang terdekat adalah 1. Setelah ini, elemen 2 ditambahkan ke tumpukan. Algoritme berlanjut seperti ini, hingga seluruh array telah diproses.

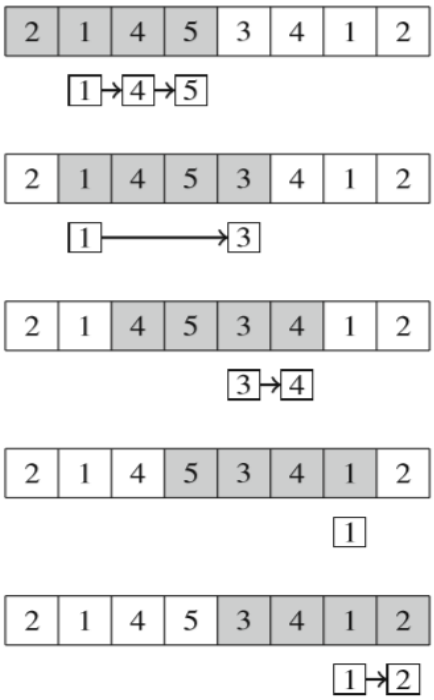
Efisiensi algoritma tergantung pada jumlah total operasi stack. Jika elemen saat ini lebih besar dari elemen teratas dalam tumpukan, maka elemen tersebut akan langsung ditambahkan ke tumpukan, yang mana cukup efisien. Namun, terkadang tumpukan dapat berisi beberapa elemen yang lebih besar dan membutuhkan waktu untuk menghapusnya. Namun, setiap elemen ditambahkan tepat satu kali ke tumpukan dan dihapus paling banyak satu kali dari tumpukan. Jadi, setiap elemen menyebabkan operasi tumpukan O(1), dan algoritma bekerja dalam waktu O(n).

8.2.3 Jendela Geser Minimum

Jendela geser adalah subarray berukuran konstan yang bergerak dari kiri ke kanan melalui array. Sebuah posisi jendela pengajaran, kami ingin menghitung beberapa informasi tentang elemen-elemen di dalam jendela. Selanjutnya kita akan fokus pada masalah mempertahankan jendela geser minimum, yang berarti bahwa kita ingin melaporkan nilai terkecil di dalam setiap jendela.



Gambar 8.5 Menemukan elemen terkecil terdekat dalam waktu linier memakai tumpukan



Gambar 8.6 Menemukan minimum jendela geser dalam waktu linier

Minima jendela geser dapat dihitung menggunakan ide serupa yang kami gunakan untuk menghitung elemen terkecil terdekat. Kali ini kita mempertahankan antrian di mana setiap elemen lebih besar dari elemen sebelumnya, dan elemen pertama selalu sesuai dengan elemen minimum di dalam jendela. Setelah setiap jendela bergerak, kami menghapus elemen dari akhir antrian sampai elemen antrian terakhir lebih kecil dari elemen jendela baru, atau antrian menjadi kosong. Kami juga menghapus elemen antrian pertama jika tidak berada di dalam jendela lagi. Akhirnya, kami menambahkan elemen jendela baru ke antrian.

Gambar 8.6 menunjukkan bagaimana algoritma memproses array ketika ukuran jendela geser adalah 4. Pada posisi jendela pertama, nilai terkecil adalah 1. Kemudian jendela bergerak satu langkah ke kanan. Elemen baru 3 lebih kecil dari elemen 4 dan 5 dalam antrian, sehingga elemen 4 dan 5 dikeluarkan dari antrian dan elemen 3 ditambahkan ke antrian. Nilai terkecil masih 1. Setelah ini, jendela bergerak lagi, dan elemen terkecil 1 bukan milik jendela lagi. Dengan demikian, ia dihapus dari antrian, dan nilai terkecil sekarang adalah 3. Juga elemen baru 4 ditambahkan ke antrian. Elemen baru berikutnya 1 lebih kecil dari semua elemen dalam antrian, sehingga semua elemen dihapus dari antrian, dan hanya berisi elemen 1. Akhirnya, jendela mencapai posisi terakhirnya. Elemen 2 ditambahkan ke antrian, tetapi nilai terkecil di dalam jendela tetap 1.

Karena setiap elemen array ditambahkan ke antrian tepat satu kali dan dihapus dari antrian paling banyak satu kali, algoritma bekerja dalam waktu O(n).

8.3 Menemukan Nilai Minimum

Misalkan ada fungsi f(x) yang pertama hanya berkurang, kemudian mencapai nilai minimumnya, dan kemudian hanya meningkat. Misalnya, Gambar 8.7 menunjukkan fungsi

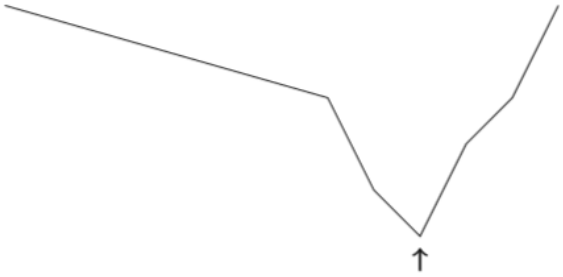
yang nilai minimumnya ditandai dengan panah. Jika kita mengetahui bahwa fungsi kita memiliki sifat ini, kita dapat secara efisien menemukan nilai minimumnya.

8.3.1 Pencarian Terner

Pencarian terner menyediakan cara yang efisien untuk menemukan nilai minimum dari suatu fungsi yang mula-mula berkurang dan kemudian meningkat. Asumsikan bahwa kita mengetahui bahwa nilai x yang meminimalkan f (x) berada dalam suatu rentang [xL,xR]. Idenya adalah untuk membagi rentang menjadi tiga bagian berukuran sama[xL,a],[a,b], dan[b,xR]dengan memilih

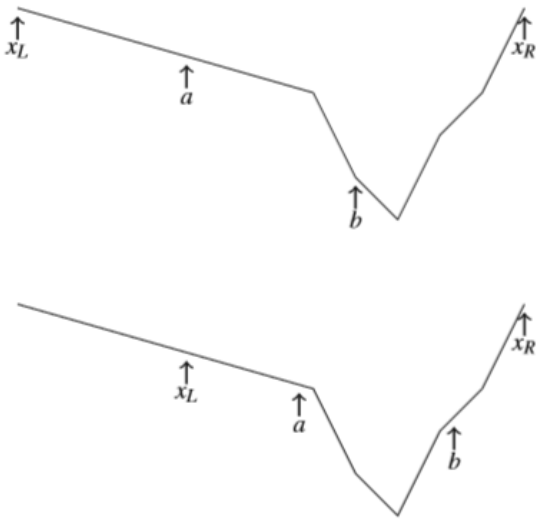
$$a = \frac{2xL + xR}{3} \; dan \; b = \frac{xL + 2xR}{3}$$

Kemudian, jika f (a)< f (b), kita simpulkan bahwa minimum harus dalam range[xL,b], dan sebaliknya harus dalam range[a,xR]. Setelah ini, kami melanjutkan pencarian secara rekursif, hingga ukuran rentang aktif cukup kecil. Sebagai contoh, Gambar 8.8 menunjukkan langkah pertama pencarian ternary dalam skenario contoh kita. Karena f (a)> f (b), rentang baru menjadi [a,xR].

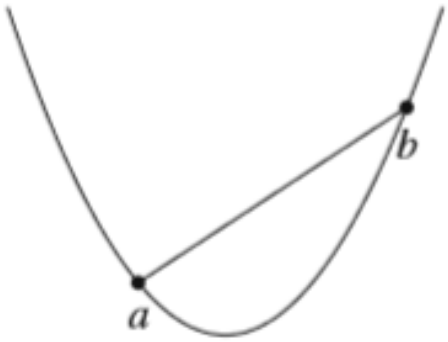


Gambar 8.7 Fungsi dan nilai minimumnya

Dalam praktiknya, kita sering mempertimbangkan fungsi yang parameternya adalah bilangan bulat, dan pencarian dihentikan ketika rentang hanya berisi satu elemen. Karena ukuran rentang baru selalu 2/3 dari rentang sebelumnya, algoritma bekerja dalam waktu O(log n), di mana n adalah jumlah elemen dalam rentang asli. Perhatikan bahwa ketika bekerja dengan parameter bilangan bulat, kita juga dapat menggunakan pencarian biner daripada pencarian terner, karena cukup untuk menemukan posisi pertama x dimana f (x) f (x +1).



Gambar 8.8 Mencari minimum menggunakan pencarian ternary



Gambar 8.9 Contoh fungsi cembung: f (x) = x2

8.3.2 Fungsi Cembung

Suatu fungsi dikatakan cembung jika ruas garis antara dua titik pada grafik fungsi selalu terletak di atas atau pada grafik. Sebagai contoh, Gambar 8.9 menunjukkan grafik f(x) = x^2, yang merupakan fungsi cembung. Memang, segmen garis antara titik a dan b terletak di atas grafik. Jika kita mengetahui bahwa nilai minimum dari suatu fungsi cembung berada dalam rentang [x_L,x_R], kita dapat menggunakan pencarian terner untuk menemukannya. Namun, perhatikan bahwa beberapa titik fungsi cembung mungkin memiliki nilai minimum. Misalnya, f (x) = 0 adalah cembung dan nilai minimumnya adalah 0. Fungsi cembung memiliki beberapa sifat yang berguna : jika f(x) dan g(x) adalah fungsi cembung, maka juga f (x)+g(x) dan maks(f (x),g(x))adalah fungsi cembung. Misalnya, jika kita memiliki n fungsi cembung f_1, f_2,..., f_n, kita segera mengetahui bahwa juga fungsi f_1 + f_2 +...+ f_n harus cembung dan kita dapat menggunakan pencarian terner untuk menemukan nilai minimumnya .

8.3.3 Meminimalkan Jumlah

Diberikan n bilangan a_1, a_2,...,a_n, pertimbangkan masalah mencari nilai x yang meminimalkan jumlah

$$|a_1 - x| + |a_2 - x| + \cdots + |a_n - x|$$

Misalnya, jika angkanya [1, 2, 9, 2, 6], solusi optimalnya adalah memilih x =2, yang menghasilkan jumlah

$$|1 - 2| + |2 - 2| + |9 - 2| + |6 - 2| = 12$$

Karena setiap fungsi |ak x| cembung, jumlahnya juga cembung, jadi kita bisa menggunakan pencarian terner untuk menemukan nilai optimal x. Namun, ada juga solusi yang lebih mudah. Ternyata pilihan optimal untuk x selalu merupakan median angka, yaitu elemen tengah setelah diurutkan. Misalnya daftar [1, 2, 9, 2, 6] menjadi[1, 2, 2, 6, 9]setelah diurutkan, jadi mediannya adalah 2.

Median selalu optimal, karena jika x lebih kecil dari median, jumlah menjadi lebih kecil dengan meningkatkan x, dan jika x lebih besar dari median, jumlah menjadi lebih kecil dengan mengurangi x. Jika genap dan ada dua median, kedua median dan semua nilai di antara keduanya adalah pilihan yang optimal. Kemudian, pertimbangkan masalah meminimalkan fungsi

$$(a_1 - x)^2 + (a_2 - x)^2 + \cdots + (a_n - x)^2$$

Misalnya, jika angkanya [1, 2, 9, 2, 6], solusi terbaik adalah memilih x =4, yang menghasilkan jumlah

$$(1 - 4)^2 + (2 - 4)^2 + (9 - 4)^2 + (2 - 4)^2 + (6 - 4)^2 = 46$$

Sekali lagi, fungsi ini cembung dan kita dapat menggunakan pencarian terner untuk menyelesaikan masalah, tetapi ada juga solusi sederhana: pilihan optimal untuk x adalah rata-rata angka. Dalam contoh rata-ratanya adalah (1+2+9+2+6)/5=4. Hal ini dapat dibuktikan dengan menyajikan penjumlahan sebagai berikut:

$$nx^2 - 2x(a_1 + a_2 + \cdots + a_n) + (a_1^2 + a_2^2 + \cdots + a_n^2$$

Bagian terakhir tidak bergantung pada x, jadi kita bisa mengabaikannya. Bagian yang tersisa membentuk fungsi $nx^2 - 2xs$ di mana $s = a_1 + a_2 + \cdots + a_n$. Ini adalah parabola membuka ke atas dengan akar $x = 0$ dan $x = 2s/n$, dan nilai minimum adalah rata-rata akar $x = s/n$, yaitu rata-rata angka $a_1, a_2, \ldots, a_n$.

BAB 9
KUERI RENTANG

9.1 Pertanyaan tentang Array Statis

Di bagian ini, kami fokus pada situasi di mana array statis, yaitu, nilai array tidak pernah diperbarui di antara kueri. Dalam hal ini, cukup untuk melakukan praproses array sehingga kita dapat menjawab pertanyaan jangkauan secara efisien. Pertama kita akan membahas cara sederhana untuk memproses kueri jumlah menggunakan array jumlah awalan, yang juga dapat digeneralisasi ke dimensi yang lebih tinggi. Setelah ini, kita akan mempelajari algoritma tabel sparse untuk memproses kueri minimum, yang lebih sulit. Perhatikan bahwa sementara kami fokus pada pemrosesan kueri minimum, kami juga selalu dapat memproses kueri maksimum menggunakan metode serupa.

9.1.1 Jumlah Kueri

Misalkan sum_q(a,b)(“range sum query”) menunjukkan jumlah nilai array dalam suatu *range[a,b].* Kita dapat secara efisien memproses kueri jumlah apa pun dengan terlebih dahulu membuat larik jumlah awalan. Setiap nilai dalam prefiks sumarray sama dengan jumlah nilai dalam array asli hingga posisi yang sesuai, yaitu, nilai pada posisi k adalah sum_q(0,k). Sebagai contoh, Gambar 9.1 menunjukkan sebuah array dan array jumlah prefiksnya. Prefiks sumarray dapat dikonstruksi dalam waktu O(n). Kemudian, karena array prefixsum berisi semua nilai penjumlahan(0,k), kita dapat menghitung nilai sum_q(a,b) dalam waktu O(1) menggunakan rumus

$$sum_q(a,b) = sum_q(0,b) - sum_q(o,a-1)$$

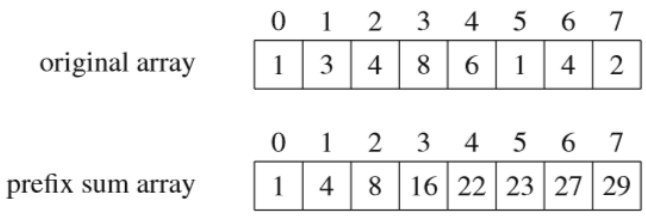
Dengan mendefinisikan sumq(0,−1) =0, rumus di atas juga berlaku ketika a =0. Sebagai contoh, Gambar 9.2 menunjukkan bagaimana menghitung jumlah nilai dalam rentang [3,6]menggunakan array jumlah awalan. Kita dapat melihat dalam larik asli bahwa sum_q(3,6) = 8+6+1+4=19. Dengan menggunakan array penjumlahan awalan, kita hanya perlu memeriksa dua nilai:

sum_q(3,6) =$sumq$(0,6)−sum_q(0,2) = 27−8=19.

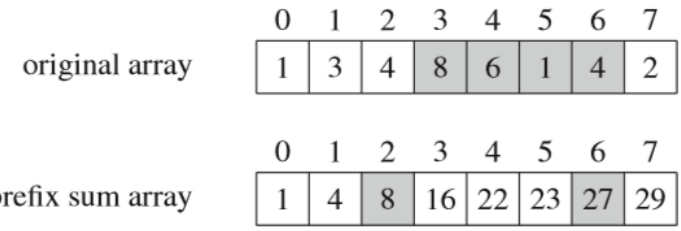
Dimensi Lebih Tinggi Hal ini juga memungkinkan untuk menggeneralisasi ide ini ke dimensi yang lebih tinggi. Sebagai contoh, Gambar 9.3 menunjukkan array jumlah prefiks dua dimensi yang dapat digunakan untuk menghitung jumlah dari setiap subarray persegi panjang dalam waktu O(1). Setiap jumlah dalam larik ini sesuai dengan subarray yang dimulai di sudut kiri atas larik. Jumlah subarray abu-abu dapat dihitung menggunakan rumus

S(A) − S(B) − S(C) + S(D),

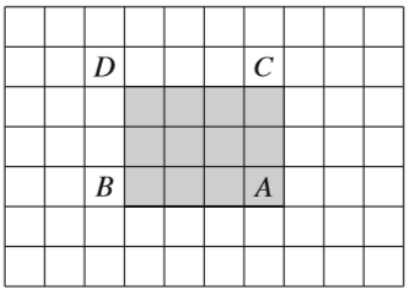
di mana S(X) menunjukkan jumlah nilai dalam subarray persegi panjang dari sudut kiri atas ke posisi X.



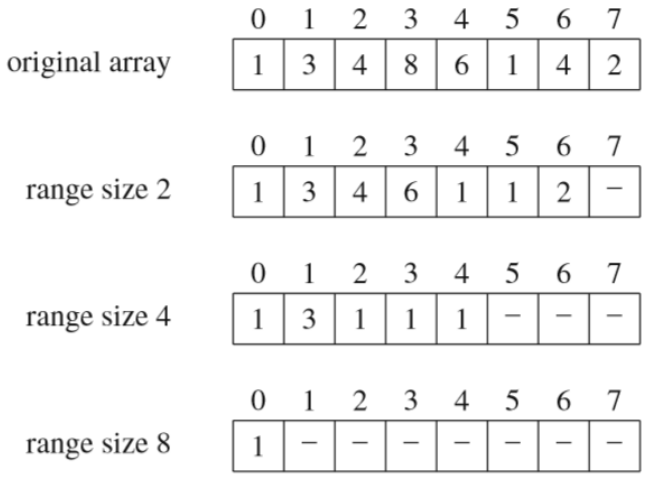
Gambar 9.1 Array dan prefiks sum arraynya



Gambar 9.2 Menghitung jumlah rentang menggunakan array jumlah awalan



Gambar 9.3 Menghitung jumlah jangkauan dua dimensi



Gambar 9.4 Prapemrosesan untuk kueri minimum

9.1.2 Queri Minimum

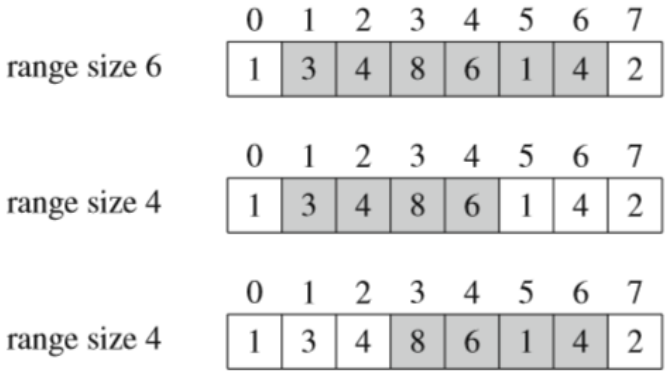
Misalkan $\min_q$(a,b) ("rentang kueri minimum") menunjukkan nilai larik minimum dalam rentang [a,b]. Selanjutnya kita akan membahas teknik yang dapat digunakan untuk memproses kueri minimum apa pun dalam waktu O(1) setelah pemrosesan awal waktu *O(n log n)*. Metode ini disebabkan oleh Bender dan Farach-Colton dan sering disebut algoritma tabel sparse. Idenya adalah untuk menghitung semua nilai $\min_q$(a,b) di mana b−a+1 (panjang rentang) adalah pangkat dua. Sebagai contoh, Gambar 9.4 menunjukkan nilai yang telah dihitung sebelumnya untuk larik delapan elemen. Jumlah nilai yang dihitung sebelumnya adalah *O(n log n),* karena ada panjang rentang *O(log n)* yang merupakan pangkat dua. Nilai dapat dihitung secara efisien menggunakan rumus rekursif

$$\min_{\mathrm{q}}(a,b) = \min(\min_{\mathrm{q}}\left(a, a+w-1\right), \min_{\mathrm{q}}(ab,b))$$

di mana b−a+1 adalah pangkat dua dan w= (b−a+1)/2. Menghitung semua nilai tersebut membutuhkan waktu *O(n log n).* Setelah ini, setiap nilai $\min_q$(a,b) dapat

dihitung dalam waktu O(1) sebagai minimal dua nilai yang telah dihitung sebelumnya. Misalkan k adalah pangkat dua terbesar yang tidak melebihi b−a+1. Kita dapat menghitung nilai $min_q(a,b)$ menggunakan rumus

Dalam rumus di atas, rentang [a, b] direpresentasikan sebagai gabungan dari rentang [a, a + k−1] dan [b−k+1, b], keduanya memiliki panjang k. Sebagai contoh, perhatikan range[1,6] pada Gambar 9.5. Panjang rentang adalah 6, dan pangkat dua terbesar yang tidak melebihi 6 adalah 4. Jadi rentang[1,6] adalah gabungan dari rentang[1,4]dan[3,6]. Karena $min_q(1,4)$ =3 dan $min_q(3,6)$ =1, kita simpulkan bahwa $min_q(1,6)$ =1. Perhatikan bahwa ada juga teknik canggih yang dapat digunakan untuk memproses kueri rentang minimum dalam waktu O(1) setelah pemrosesan awal waktu O(n) saja (lihat, misalnya, Fischer dan Heun), tetapi teknik ini berada di luar cakupan buku ini.



Gambar 9.5 Menghitung rentang minimum menggunakan dua rentang yang tumpang tindih

9.2 Struktur Pohon

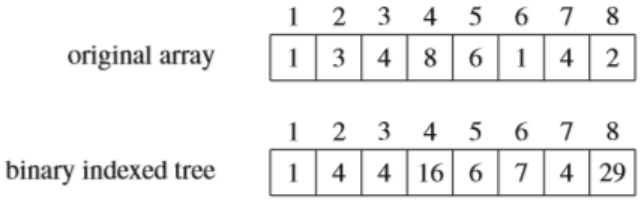
Bagian ini menyajikan dua struktur pohon, yang dengannya kita dapat memproses kueri rentang dan memperbarui nilai larik dalam waktu logaritmik. Pertama, kami membahas pohon berindeks biner yang mendukung kueri jumlah, dan setelah itu, kami fokus pada pohon segmen yang juga mendukung beberapa kueri lainnya.

9.2.1 Pohon Terindeks Biner

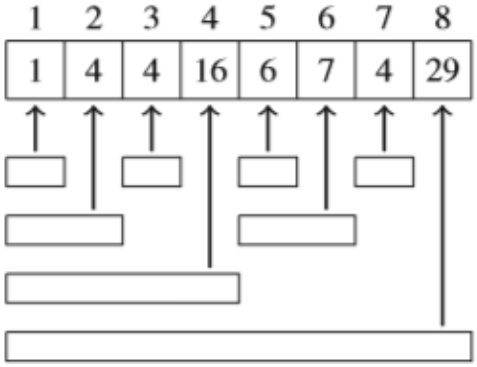
Pohon berindeks biner (atau pohon Fenwick) dapat dilihat sebagai varian dinamis dari array jumlah awalan. Ini menyediakan dua operasi waktu *O(log n)*: memproses kueri jumlah rentang dan memperbarui nilai. Bahkan jika nama strukturnya adalah pohon berindeks biner, struktur tersebut biasanya direpresentasikan sebagai array. Saat membahas pohon berindeks biner, kami berasumsi bahwa semua array diindeks satu, karena membuat implementasi struktur lebih mudah. Misalkan p(k) menyatakan pangkat terbesar dari dua yang membagi k. Kami menyimpan pohon berindeks biner sebagai tree array sedemikian rupa sehingga

$$tree[k]=sum^q(k- p(k)+1,k),$$

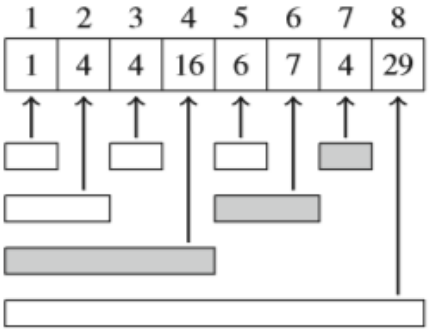
yaitu, setiap posisi k berisi jumlah nilai dalam rentang larik asli yang panjangnya p(k) dan berakhir di posisi k. Misalnya, karena p(6) =2, pohon[6] berisi nilai sumq(5,6). Gambar 9.6 menunjukkan sebuah larik dan pohon berindeks biner yang sesuai. Gambar 9.7 menunjukkan dengan lebih jelas bagaimana setiap nilai dalam pohon berindeks biner sesuai dengan rentang dalam larik asli.



Gambar 9.6 Array dan pohon berindeks binernya



Gambar 9.7 Rentang dalam pohon berindeks biner



Gambar 9.8 Memproses kueri penjumlahan rentang menggunakan pohon berindeks biner

Menggunakan pohon berindeks biner, setiap nilai sumq(1, k) dapat dihitung dalam waktu O(log n), karena suatu rentang[1, k] selalu dapat dibagi menjadi O(log n) subrentang yang jumlahnya telah disimpan di pohon . Misalnya, untuk menghitung nilai sumq(1,7), kita membagi range [1,7] menjadi tiga subrange [1, 4], [5, 6], dan [7, 7] (Gambar 9.8) . Karena jumlah dari sub-rentang tersebut tersedia di pohon, kita dapat menghitung jumlah seluruh rentang menggunakan rumus:

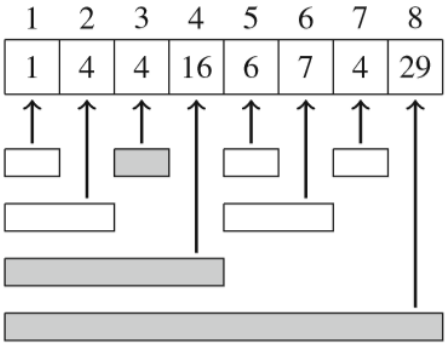
$$sum_q(1,7) = sum_q(1,4)+sum_q(5,6)+sum_q(7,7) = 16+7+4=27.$$

Kemudian, untuk menghitung nilai $sum_q(a, b)$ di mana $a > 1$, kita dapat menggunakan trik yang sama dengan yang kita gunakan dengan array jumlah awalan:

$$sum_q(a,b) = sum_q(1,b) - sum_q(1,a-1)$$

Kita dapat menghitung sumq(1, b) dan sumq(1, a−1) dalam waktu O(log n), jadi total kompleksitas waktu adalah O(log n). Setelah memperbarui nilai array, beberapa nilai dalam pohon berindeks biner harus diperbarui. Misalnya, ketika nilai pada posisi 3 berubah, kita harus memperbarui subrange[3,3],[1,4], dan [1,8] (Gambar 9.9). Karena

setiap elemen array termasuk dalam subrange O(log n), maka cukup untuk memperbarui nilai pohon O(log n).



Gambar 9.9 Memperbarui nilai dalam pohon berindeks biner

Penerapan

Operasi pohon berindeks biner dapat diimplementasikan secara efisien menggunakan operasi bit. Fakta kunci yang diperlukan adalah kita dapat dengan mudah menghitung nilai p(k) apa pun menggunakan rumus bit

$$p(k) = k \,\&\, -k,$$

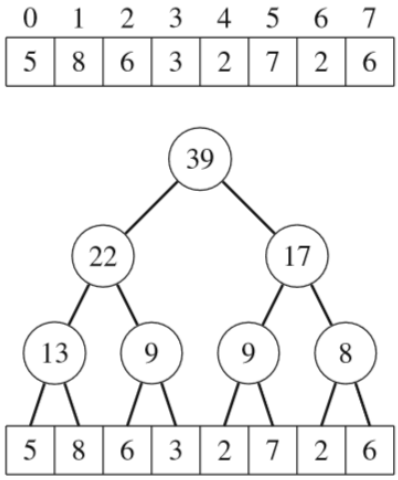
yang mengisolasi satu bit k yang paling tidak signifikan. Pertama, fungsi berikut menghitung nilai $sum_q(1,k)$:

```
int sum(int k) {
        int s = 0;
                while (k >= 1) {
        s += tree[k];
        k -= k&-k;
        }
        return s;
}
```

Kemudian, fungsi berikut meningkatkan nilai array pada posisi k sebesar x (x bisa positif atau negatif):

```
void add(int k, int x) {
        while (k <= n) {
        tree[k] += x;
        k += k&-k;
        }
}
```

Kompleksitas waktu dari kedua fungsi tersebut adalah O(log n), karena fungsi mengakses nilai O(log n) di pohon berindeks biner, dan setiap perpindahan ke posisi berikutnya membutuhkan waktu O(1).



Gambar 9.10 Array dan pohon segmen yang sesuai untuk jumlah kueri

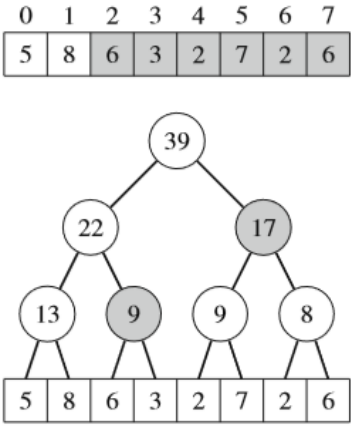
9.2.2 Pohon Segmen

Pohon segmen adalah struktur data yang menyediakan dua operasi waktu O(log n): memproses kueri rentang dan memperbarui nilai larik. Pohon segmen mendukung kueri jumlah, kueri minimum, dan banyak kueri lainnya. Pohon segmen berasal dari algoritma geometris dan implementasi bottom-up yang elegan yang disajikan di bagian ini mengikuti buku teks oleh Sta´nczyk. Pohon segmen adalah pohon biner yang simpul tingkat bawahnya sesuai dengan elemen larik, dan simpul lainnya berisi informasi yang diperlukan untuk memproses kueri rentang. Saat membahas pohon segmen, kami berasumsi bahwa ukuran larik adalah pangkat dua, dan pengindeksan berbasis nol digunakan, karena lebih mudah untuk membangun pohon segmen untuk larik seperti itu. Jika ukuran larik bukan pangkat dua, kita selalu dapat menambahkan elemen tambahan ke dalamnya.

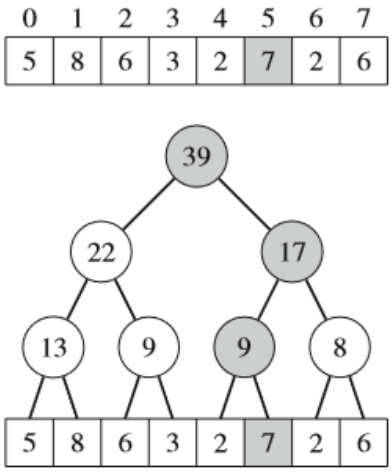
Pertama-tama kita akan membahas pohon segmen yang mendukung jumlah kueri. Sebagai contoh, Gambar 9.10 menunjukkan array dan pohon segmen yang sesuai untuk kueri jumlah. Setiap simpul pohon internal sesuai dengan rentang array yang ukurannya pangkat dua. Ketika pohon segmen mendukung kueri jumlah, nilai setiap simpul internal adalah jumlah dari nilai larik yang sesuai, dan dapat dihitung sebagai jumlah nilai simpul anak kiri dan kanannya.

Ternyata sembarang range[a,b] dapat dibagi menjadi subrange O(log n) yang nilainya disimpan dalam simpul pohon. Misalnya, Gambar 9.11 menunjukkan rentang[2,7]dalam larik asli dan di pohon segmen. Dalam hal ini, dua simpul pohon sesuai dengan rentang, dan $sum_q(2,7)$ =9+17=26. Ketika jumlah dihitung menggunakan node yang terletak setinggi mungkin di pohon, paling banyak dua node pada setiap tingkat pohon diperlukan.

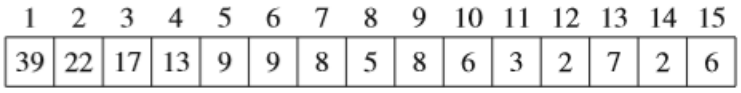
Oleh karena itu, jumlah total node adalah O(log n). Setelah pembaruan array, kita harus memperbarui semua node yang nilainya bergantung pada nilai yang diperbarui. Ini dapat dilakukan dengan melintasi jalur dari elemen larik yang diperbarui ke simpul teratas dan memperbarui simpul di sepanjang jalur. Sebagai contoh, Gambar 9.12 menunjukkan node yang berubah ketika nilai pada posisi 5 berubah. Jalur dari bawah ke atas selalu terdiri dari O(log n) node, sehingga setiap pembaruan mengubah O(log n) node di pohon.



Gambar 9.11 Memproses kueri penjumlahan rentang menggunakan pohon segmen



Gambar 9.12 Memperbarui nilai array di pohon segmen



Gambar 9.13 Isi pohon segmen dalam array

Penerapan

Cara mudah untuk menyimpan konten pohon segmen adalah dengan menggunakan larik elemen 2n di mana n adalah ukuran larik asli. Node pohon disimpan dari atas ke bawah: tree [1] adalah node teratas, tree[2] dan tree[3] adalah anak-anaknya, dan seterusnya. Akhirnya, nilai dari tree [n] ke tree[2n−1] sesuai dengan tingkat bawah pohon, yang berisi nilai-nilai larik asli. Perhatikan bahwa elemen tree[0]tidak digunakan. Misalnya, Gambar 9.13 menunjukkan bagaimana pohon contoh kita disimpan. Perhatikan bahwa induk dari tree[k] adalah tree[k/2], anak kirinya adalah tree[2k], dan anak kanannya adalah tree[2k+1]. Selain itu, posisi simpul (selain simpul atas) adalah genap jika anak kiri dan ganjil jika anak kanan.

Fungsi berikut menghitung nilai $sum_q(a,b)$:

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
```

```
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
        }
        return s;
}
```

Fungsi mempertahankan rentang dalam larik pohon segmen. Awalnya, rentangnya adalah [a+n, b+n]. Pada setiap langkah, rentang dipindahkan satu tingkat lebih tinggi di pohon, dan nilai-nilai simpul yang tidak termasuk dalam rentang yang lebih tinggi ditambahkan ke penjumlahan. Fungsi berikut meningkatkan nilai array pada posisi k sebesar x:

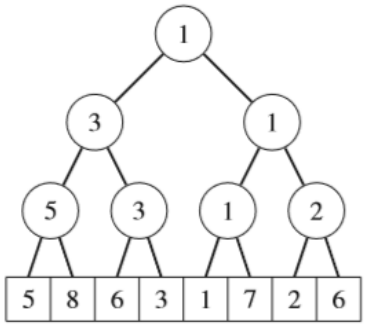
```
void add(int k, int x) {
        k += n;
        tree[k] += x;
        for (k /= 2; k >= 1; k /= 2) {
        tree[k] = tree[2*k]+tree[2*k+1];
        }
}
```

Pertama nilai di tingkat bawah pohon diperbarui. Setelah ini, nilai semua simpul pohon internal diperbarui, hingga simpul teratas pohon tercapai. Kedua fungsi di atas bekerja dalam waktu O(log n), karena pohon segmen dari n elemen terdiri dari level O(log n) dan fungsi bergerak satu tingkat lebih tinggi di pohon pada setiap langkah.

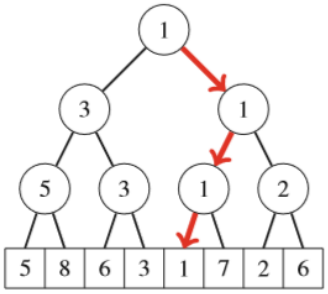
Kueri lainnya

Pohon segmen dapat mendukung kueri rentang apa pun di mana kita dapat membagi rentang menjadi dua bagian, menghitung jawaban secara terpisah untuk kedua bagian, dan kemudian menggabungkan jawaban secara efisien. Contoh kueri tersebut adalah minimum dan maksimum, pembagi umum terbesar, dan operasi bit dan, atau, dan xor. Misalnya, pohon segmen pada Gambar 9.14 mendukung kueri minimum. Dalam pohon ini, setiap simpul berisi nilai terkecil dalam rentang larik yang sesuai.

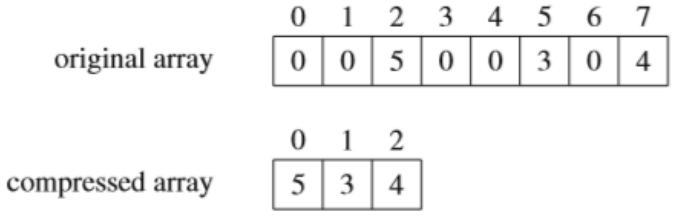
Node teratas dari pohon berisi nilai terkecil di seluruh array. Operasi dapat diimplementasikan seperti sebelumnya, tetapi alih-alih jumlah, minima dihitung. Struktur pohon segmen juga memungkinkan kita untuk menggunakan metode gaya pencarian biner untuk menemukan elemen array. Misalnya, jika pohon mendukung kueri minimum, kita dapat menemukan posisi elemen dengan nilai terkecil dalam waktu O(log n). Sebagai contoh, Gambar 9.15 menunjukkan bagaimana elemen dengan nilai terkecil 1 dapat ditemukan dengan melintasi jalur ke bawah dari simpul teratas.



Gambar 9.14 Pohon segmen untuk memproses kueri minimum rentang



Gambar 9.15 Menggunakan pencarian biner untuk menemukan elemen minimum



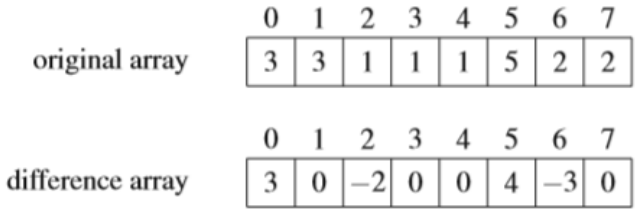
Gambar 9.16 Mengompresi array menggunakan kompresi indeks

9.2.3 Teknik Tambahan

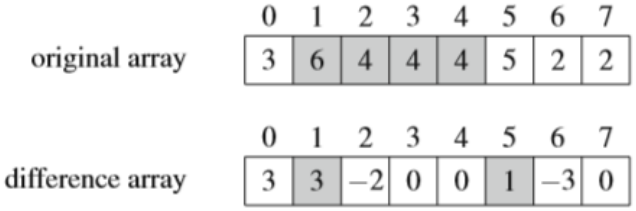
Kompresi Indeks

Keterbatasan dalam struktur data yang dibangun di atas array adalah bahwa elemen diindeks menggunakan bilangan bulat berurutan. Kesulitan muncul ketika indeks besar diperlukan. Misalnya, jika kita ingin menggunakan indeks 109, array harus berisi 109 elemen yang akan membutuhkan terlalu banyak memori. Namun, jika kita mengetahui semua indeks yang diperlukan selama algoritma sebelumnya, kita dapat melewati batasan ini dengan menggunakan kompresi indeks.

Idenya adalah untuk mengganti indeks asli dengan bilangan bulat berturut-turut 0,1,2, dan seterusnya. Untuk melakukan ini, kami mendefinisikan fungsi c yang memampatkan indeks. Fungsi tersebut memberikan setiap indeks asli i indeks terkompresi c(i) sedemikian rupa sehingga jika a dan b adalah dua indeks dan a < b, maka c(a)<c(b). Setelah mengompresi indeks, kita dapat dengan mudah melakukan kueri menggunakan indeks tersebut. Gambar 9.16 menunjukkan contoh sederhana dari kompresi indeks. Di sini hanya indeks 2,5, dan 7 yang benar-benar digunakan, dan semua nilai larik lainnya adalah nol. Indeks terkompresi adalah c(2) =0, c(5) =1, dan c(7) =2, yang memungkinkan kita membuat larik terkompresi yang hanya berisi tiga elemen.



Gambar 9.17 Array dan perbedaannya array



Gambar 9.18 Memperbarui rentang array menggunakan array perbedaan

Pembaruan Rentang

Sejauh ini, kami telah menerapkan struktur data yang mendukung kueri rentang dan pembaruan nilai tunggal. Sekarang mari kita pertimbangkan situasi yang berlawanan, di mana kita harus memperbarui rentang dan mengambil nilai tunggal. Kami fokus pada operasi yang meningkatkan semua elemen dalam rentang[a,b]oleh x. Ternyata kita dapat menggunakan struktur data yang disajikan dalam bab ini juga dalam situasi ini. Untuk melakukan ini, kami membangun array perbedaan yang nilainya menunjukkan perbedaan antara nilai berurutan dalam array asli.

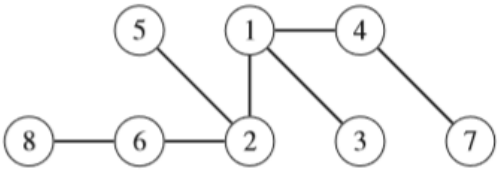
Array asli adalah array jumlah awalan dari array perbedaan. Gambar 9.17 menunjukkan sebuah larik dan larik perbedaannya. Misalnya, nilai 2 pada posisi 6 dalam larik asli sesuai dengan jumlah 3−2+4−3=2 dalam larik selisih. Keuntungan dari array perbedaan adalah bahwa kita dapat memperbarui rentang dalam array asli dengan mengubah hanya dua elemen dalam array perbedaan. Lebih tepatnya, untuk meningkatkan nilai dalam rentang [a,b] sebesar x, kita meningkatkan nilai pada posisi a sebanyak x dan menurunkan nilai pada posisi b+1 sebanyak x.

Misalnya, untuk meningkatkan nilai larik asli antara posisi 1 dan 4 sebanyak 3, kami meningkatkan nilai larik selisih pada posisi 1 sebanyak 3 dan menurunkan nilai pada posisi 5 sebanyak 3 (Gambar 9.18). Jadi, kami hanya memperbarui nilai tunggal dan memproses jumlah kueri dalam larik perbedaan, sehingga kami dapat menggunakan pohon berindeks biner atau pohon segmen. Tugas yang lebih sulit adalah membuat struktur data yang mendukung kueri rentang dan pembaruan rentang. Dalam Sect.15.2.1, kita akan melihat bahwa ini juga dimungkinkan dengan menggunakan pohon segmen malas.

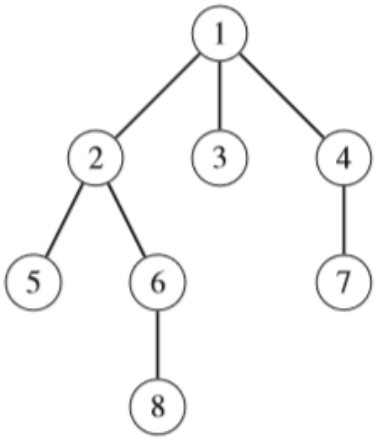
BAB 10
POHON ALGORITMA

Teknik Dasar Pohon adalah graf siklik terhubung yang terdiri dari n simpul dan n−1 sisi. Menghapus tepi apa pun dari pohon akan membaginya menjadi dua komponen, dan menambahkan tepi apa pun akan membuat siklus. Selalu ada jalur unik antara dua simpul pohon. Daun pohon adalah simpul dengan hanya satu tetangga. Sebagai contoh, perhatikan pohon pada Gambar 10.1. Pohon ini terdiri dari 8 simpul dan 7 tepi, dan daunnya adalah simpul 3, 5, 7, dan 8. Pada pohon berakar, salah satu simpul ditunjuk sebagai akar pohon, dan semua simpul lainnya ditempatkan di bawah akar.

Tetangga bawah dari suatu simpul disebut anak-anaknya, dan tetangga atas suatu simpul disebut induknya. Setiap node memiliki tepat satu parent, kecuali root yang tidak memiliki parent. Struktur pohon berakar adalah rekursif: setiap simpul pohon bertindak sebagai akar dari subpohon yang berisi simpul itu sendiri dan semua simpul yang ada di dalam subpohon turunannya. Misalnya, Gambar 10.2 menunjukkan pohon berakar di mana simpul 1 adalah akar dari pohon. Anak dari node 2 adalah node 5 dan 6, dan parent dari node 2 adalah node 1. Subtree dari node 2 terdiri dari node 2, 5, 6, dan 8.



Gambar 10.1 Sebuah pohon yang terdiri dari 8 node dan 7 edge



Gambar 10.2 Pohon berakar di mana simpul 1 adalah simpul akar

10.1 Teknik Dasar

10.1.1 Lintasan Pohon

Algoritma traversal grafik umum dapat digunakan untuk melintasi node dari sebuah pohon. Namun, traversal pohon lebih mudah diterapkan daripada graf umum, karena tidak ada siklus di pohon, dan tidak mungkin mencapai simpul dari lebih dari satu arah. Cara khas untuk melintasi pohon adalah memulai pencarian kedalaman-pertama pada simpul arbitrer. Fungsi rekursif berikut dapat digunakan:

```
void dfs(int s, int e) {
    //process node s
```

```
            for (auto u : adj[s]) {
            if (u != e) dfs(u, s);
            }
    }
```

Fungsi tersebut diberikan dua parameter: node saat ini dan node sebelumnya e. Maksud dari parameter e adalah untuk memastikan bahwa pencarian hanya berpindah ke node yang belum dikunjungi.

```
            dfs(x, 0);
```

Pemanggilan fungsi berikut memulai pencarian di node x:

```
    void dfs(int s, int e) {
            count[s] = 1;
            for (auto u : adj[s]) {
            if (u == e) continue;
            dfs(u, s);
            count[s] += count[u];
            }
    }
```

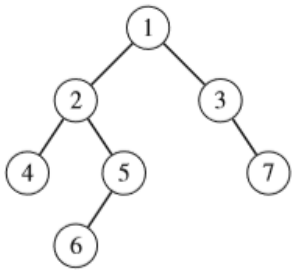
Pada pemanggilan pertama e =0, karena tidak ada simpul sebelumnya, dan diperbolehkan untuk melanjutkan ke sembarang arah di pohon.

Traversal Pohon Biner

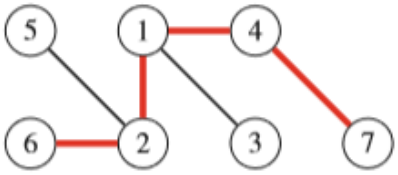
Dalam pohon biner, setiap simpul memiliki subpohon kiri dan kanan (yang mungkin kosong), dan ada tiga urutan traversal pohon yang populer:

- *pre-order*: pertama-tama proses root node, lalu traverse subtree kiri, lalu traverse subtree kanan
- *in-order*: pertama telusuri subpohon kiri, lalu proses simpul akar, lalu telusuri subpohon kanan
- *post-order:* pertama traverse subtree kiri, lalu traverse subtree kanan, lalu proses root node

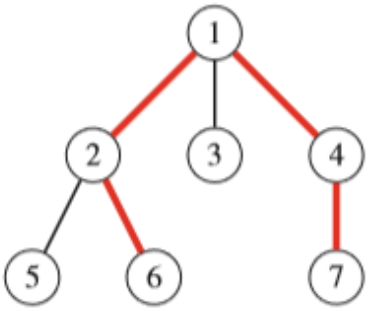
Sebagai contoh, pada Gambar 10.3, pre-order adalah [1, 2, 4, 5, 6, 3, 7], *in-order* adalah [4,2,6,5,1,3,7], dan post-ordernya adalah [4, 6, 5, 2, 7, 3, 1]. Jika kita mengetahui pre-order dan in-order sebuah pohon, kita dapat merekonstruksi struktur persisnya. Sebagai contoh, satu-satunya pohon yang mungkin dengan *pre-order*[1, 2, 4, 5, 6, 3, 7] dan *in-order* [4, 2, 6, 5, 1, 3, 7] ditunjukkan pada Gambar. 10.3. *Post-order* dan *in-order* juga secara unik menentukan struktur pohon. Namun, jika kita hanya mengetahui *pre-order* dan *post-order*, mungkin ada lebih dari satu pohon yang cocok dengan pemesanan.



Gambar 10.3 Pohon biner



Gambar 10.4 Pohon yang diameternya 4



Gambar 10.5 Node 1 adalah titik tertinggi pada jalur diameter

10.1.2 Menghitung Diameter

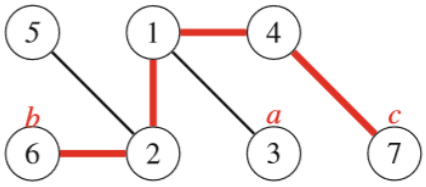
Diameter pohon adalah panjang maksimum jalur antara dua node. Sebagai contoh, Gambar.10.4 menunjukkan pohon yang diameternya 4 yang sesuai dengan jalur dengan panjang 4 antara node 6 dan 7. Perhatikan bahwa pohon juga memiliki jalur lain dengan panjang 4 antara node 5 dan 7. Selanjutnya kita akan membahas dua O (n) algoritma waktu untuk menghitung diameter pohon. Algoritma pertama didasarkan pada pemrograman dinamis, dan algoritma kedua menggunakan pencarian mendalam-pertama.

Algoritma Pertama

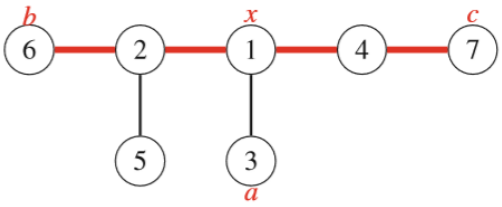
Cara umum untuk mendekati masalah pohon adalah pertama-tama melakukan root pada pohon secara sewenang-wenang dan kemudian menyelesaikan masalah secara terpisah untuk setiap subpohon. Algoritma pertama kami untuk menghitung diameter didasarkan pada ide ini. Pengamatan penting adalah bahwa setiap jalur dalam pohon berakar memiliki titik tertinggi: simpul tertinggi yang dimiliki jalur tersebut. Dengan demikian, kita dapat menghitung untuk setiap simpul x panjang jalur terpanjang yang titik tertingginya adalah x. Salah satu jalur tersebut sesuai dengan diameter pohon. Misalnya, pada Gambar.10.5, simpul 1 adalah titik tertinggi pada jalur yang sesuai dengan diameter. Kami menghitung untuk setiap node x dua nilai:

- toLeaf(x): panjang maksimum jalur dari x ke daun apa pun
- maxLength(x): panjang maksimum lintasan yang titik tertingginya adalah x

Misalnya, pada Gambar 10.5, toLeaf (1) =2, karena ada jalur 1→2→6, dan maxLength(1) =4, karena ada jalur 6→2→1→4→7. Dalam hal ini, maxLength(1) sama dengan diameter.



Gambar 10.6 Node a, b, dan c saat menghitung diameter



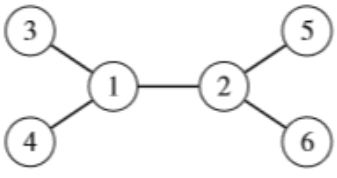
Gambar 10.7 Mengapa algoritma bekerja?

Algoritma Kedua

Cara lain yang efisien untuk menghitung diameter pohon didasarkan pada dua pencarian *depth-first.* Pertama, kita memilih simpul sembarang a di pohon dan mencari simpul terjauh b dari a. Kemudian, kami menemukan simpul terjauh c dari b. Diameter pohon adalah jarak antara b dan c. Sebagai contoh, Gambar.10.6 menunjukkan cara yang mungkin untuk memilih node a, b, dan c saat menghitung diameter untuk pohon contoh kita. Ini adalah metode yang elegan, tetapi mengapa itu berhasil? Ini membantu untuk menggambar pohon sehingga jalur yang sesuai dengan diameter adalah horizontal dan semua simpul lainnya menggantung darinya (Gambar 10.7). Node x menunjukkan tempat jalur dari node a bergabung dengan jalur yang sesuai dengan diameter. Simpul terjauh dari a adalah simpul b, simpul c, atau beberapa simpul lain yang jaraknya paling sedikit dari simpul x. Dengan demikian, simpul ini selalu merupakan pilihan yang valid untuk titik akhir jalur yang sesuai dengan diameter.

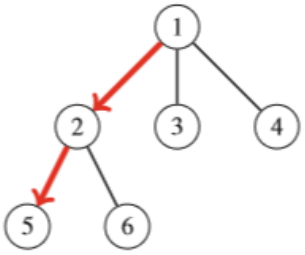
10.1.3 Semua Jalur Terpanjang

Masalah kita selanjutnya adalah menghitung untuk setiap simpul pohon x sebuah nilai maxLength(x): panjang maksimum dari sebuah jalur yang dimulai pada simpul x. Misalnya, Gambar 10.8 menunjukkan pohon dan nilai maxLength-nya. Hal ini dapat dilihat sebagai generalisasi dari masalah diameter pohon, karena yang terbesar dari panjang tersebut sama dengan diameter pohon. Juga, masalah ini dapat diselesaikan dalam waktu O(n). Sekali lagi, titik awal yang baik adalah melakukan root pada pohon secara sewenang-wenang. Bagian pertama dari masalah ini adalah menghitung untuk setiap simpul x panjang maksimum jalur yang turun melalui anak x. Misalnya, jalur terpanjang dari simpul 1 melewati anaknya 2 (Gambar 10.9). Bagian ini mudah diselesaikan dalam waktu O(n), karena kita dapat menggunakan pemrograman dinamis seperti yang telah kita lakukan sebelumnya.

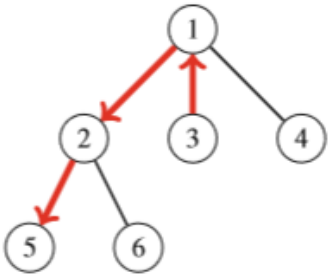


$$\begin{aligned}
\texttt{maxLength}(1) &= 2\\
\texttt{maxLength}(2) &= 2\\
\texttt{maxLength}(3) &= 3\\
\texttt{maxLength}(4) &= 3\\
\texttt{maxLength}(5) &= 3\\
\texttt{maxLength}(6) &= 3
\end{aligned}$$

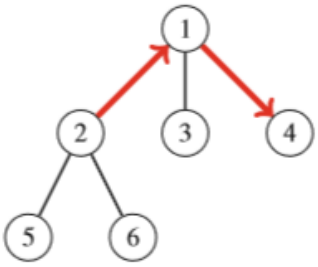
Gambar 10.8 Menghitung panjang jalur maksimum



Gambar 10.9 Jalur terpanjang yang dimulai pada node 1



Gambar 10.10 Jalur terpanjang dari simpul 3 melewati induknya



Gambar 10.11 Dalam hal ini, jalur terpanjang kedua dari orang tua harus dipilih

Kemudian, bagian kedua dari masalah ini adalah menghitung untuk setiap simpul x panjang maksimum jalur ke atas melalui induknya p. Misalnya, jalur terpanjang dari simpul 3 melewati induknya 1 (Gambar 10.10). Sepintas, tampaknya kita harus terlebih dahulu pindah ke p dan kemudian memilih jalur terpanjang (atas atau bawah) dari p. Namun, ini tidak selalu berhasil, karena jalur seperti itu dapat melalui x (Gambar 10.11). Namun, kita dapat menyelesaikan bagian kedua dalam waktu O(n) dengan menyimpan panjang maksimum dua jalur untuk setiap simpul x:

- $\text{maxLength}_1(x)$: panjang maksimum jalur dari x ke daun
- $\text{maxLength}_2(x)$ panjang maksimum jalur dari x ke daun, dalam arah lain dari jalur pertama

Misalnya, pada Gambar 10.11,$\text{maxLength1}(1)$ =2 menggunakan jalur1→2→5, dan $\text{maxLength}_2(1)$ =1 menggunakan jalur 1→3. Akhirnya, untuk menentukan jalur panjang maksimum dari simpul x ke atas melalui induknya p, kami mempertimbangkan dua kasus: jika jalur yang sesuai dengan $\text{maxLength}_1(p)$ melewati x, panjang maksimum adalah $\text{maxLength}_2(p)+1$ dan sebaliknya panjang maksimum adalah $\text{maxLength}_1(p)+1$.

10.2 Kueri Pohon

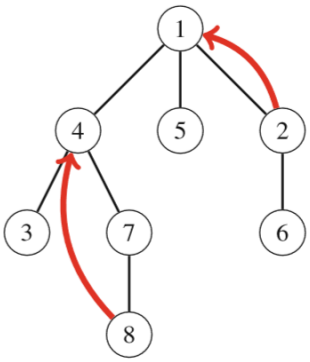
Di bagian ini kami fokus pada pemrosesan kueri pada pohon yang di-rooting. Kueri semacam itu biasanya terkait dengan subpohon dan jalur pohon, dan dapat diproses dalam waktu konstan atau logaritmik.

10.2.1 Menemukan Ancestor

Leluhur ke-k dari nodex dalam pohon berakar adalah simpul yang akan kita capai jika kita menaikkan tingkat k dari x. Misalkan ancestor(x,k) menunjukkan ancestor ke-k dari sebuah simpul x (atau 0 jika tidak ada ancestor seperti itu). Misalnya, pada Gambar 10.12, ancestor(2,1) =1 dan ancestor (8,2) =4. Cara mudah menghitung nilai ancestor(x,k) adalah dengan melakukan barisan k bergerak di pohon. Namun, kompleksitas waktu dari metode ini adalah O(k), yang mungkin lambat, karena sebuah pohon dari n node mungkin memiliki jalur n node. Untungnya, kita dapat menghitung secara efisien nilai ancestor(x,k) dalam O(log k) waktu setelah preprocessing. Idenya

adalah terlebih dahulu menghitung semua nilai ancestor(x,k) di mana k adalah pangkat dua. Sebagai contoh, nilai untuk pohon pada Gambar 10.12 adalah sebagai berikut:

x	1 2 3 4 5 6 7 8
ancestor(x,1)	0 1 4 1 1 2 4 7
ancestor(x,2)	0 0 1 0 0 1 1 4
ancestor(x,4)	0 0 0 0 0 0 0 0
...	



Gambar 10.12 Menemukan nenek moyang node

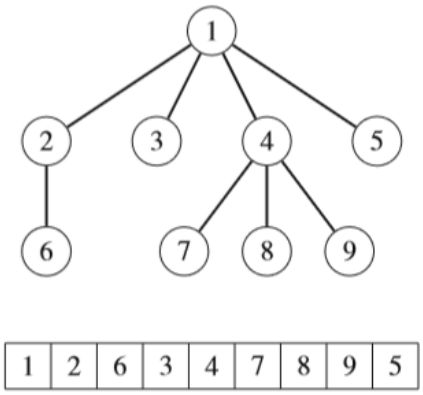
Karena kita tahu bahwa sebuah node selalu memiliki kurang dari n ancestor, cukup untuk menghitung nilai O(log n) untuk setiap node dan preprocessing membutuhkan waktu O(n log n). Setelah ini, setiap nilai ancestor (x,k) dapat dihitung dalam waktu O(logk) dengan menyatakan k sebagai jumlah di mana setiap suku adalah pangkat dua.

10.2.2 Jalur dan Subtree

Sebuah array traversal pohon berisi node dari pohon berakar dalam urutan di mana pencarian kedalaman-pertama dari node akar mengunjungi mereka. Misalnya, Gambar 10.13 menunjukkan pohon dan larik traversal pohon yang sesuai. Sebuah properti penting dari array traversal pohon adalah bahwa setiap subpohon dari pohon sesuai dengan subarray dalam array traversal pohon sehingga elemen pertama dari subarray adalah simpul akar. Sebagai contoh, Gambar 10.14 menunjukkan subarray yang sesuai dengan subtree dari node 4.

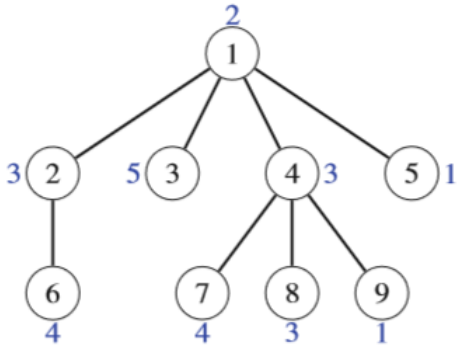
Subtree Query

Misalkan setiap simpul di pohon diberi nilai dan tugas kita adalah memproses dua jenis kueri: memperbarui nilai simpul dan menghitung jumlah nilai di subpohon simpul. Untuk memecahkan masalah, kita membangun sebuah array traversal pohon yang berisi tiga nilai untuk setiap node: pengidentifikasi node, ukuran subtree, dan nilai node. Misalnya, Gambar 10.15 menunjukkan pohon dan larik yang sesuai.

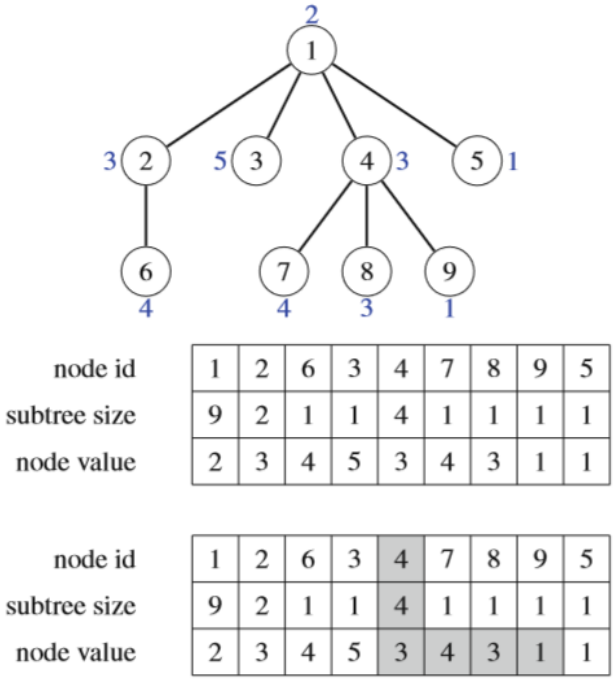


Gambar 10.13 Sebuah pohon dan susunan traversal pohonnya

Gambar 10.14 Subtree dari node 4 dalam array traversal tree



Gambar 10.15 Array traversal pohon untuk menghitung jumlah subpohon

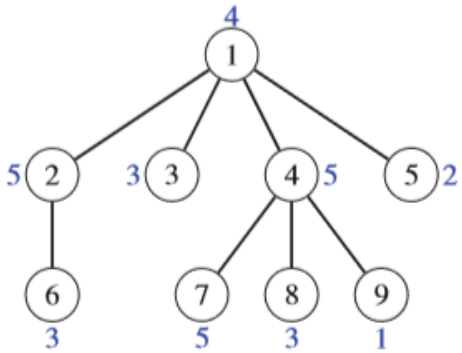


Gambar 10.16 Menghitung jumlah nilai pada subtree dari node 4

Dengan menggunakan larik ini, kita dapat menghitung jumlah nilai dalam setiap subpohon dengan terlebih dahulu menentukan ukuran subpohon dan kemudian menjumlahkan nilai dari node yang bersesuaian. Sebagai contoh, Gambar 10.16 menunjukkan nilai yang kita akses saat menghitung jumlah nilai dalam subpohon dari node 4. baris terakhir dari array memberitahu kita bahwa jumlah nilai adalah 3+4+3+1=11. Untuk menjawab pertanyaan secara efisien, cukup untuk menyimpan baris terakhir dari larik dalam pohon berindeks biner atau pohon segmen. Setelah ini, kita dapat memperbarui nilai dan menghitung jumlah nilai dalam waktu O(log n).

Kueri Jalur/Parth

Dengan menggunakan larik traversal pohon, kita juga dapat menghitung jumlah nilai secara efisien pada jalur dari simpul akar ke simpul pohon mana pun. Sebagai contoh, pertimbangkan masalah di mana tugas kita adalah memproses dua jenis kueri: memperbarui nilai sebuah simpul dan menghitung jumlah nilai pada jalur dari akar ke simpul. Untuk memecahkan masalah, kita membangun sebuah array traversal pohon yang berisi untuk setiap node identifier, ukuran subtree, dan jumlah nilai pada jalur dari root ke node (Gambar 10.17). Ketika nilai sebuah simpul bertambah x, jumlah semua simpul dalam subpohonnya bertambah x. Sebagai contoh, Gambar 10.18 menunjukkan array setelah meningkatkan nilai node 4 sebesar 1. Untuk mendukung kedua operasi, kita harus dapat meningkatkan semua nilai dalam suatu rentang dan mengambil satu nilai. Ini dapat dilakukan dalam waktu O(log n) menggunakan indeks biner atau pohon segmen dan array perbedaan (lihat Bagian 9.2.3).

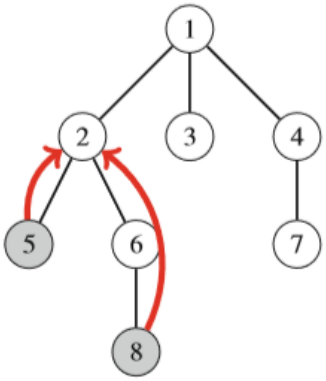


Gambar 10.17 Array traversal pohon untuk menghitung jumlah jalur

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
path sum	4	9	12	7	9	14	12	10	6

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
path sum	4	9	12	7	10	15	13	11	6

Gambar 10.18 Meningkatkan nilai node 4 sebesar 1



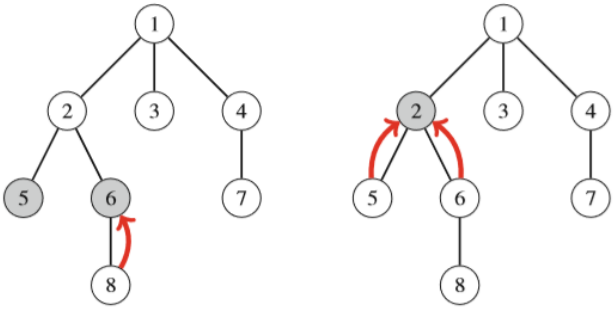
Gambar 10.19 Nenek moyang terendah dari node 5 dan 8 adalah node 2

10.2.3 Ancestor Umum Terendah

Nenek moyang terendah dari dua node dari pohon berakar adalah node terendah yang subtree berisi kedua node. Sebagai contoh, pada Gambar 10.19, nenek moyang terendah dari node 5 dan 8 adalah node 2. Masalah yang khas adalah untuk secara efisien memproses query yang mengharuskan kita untuk menemukan nenek moyang terendah dari dua node. Selanjutnya kita akan membahas dua teknik yang efisien untuk memproses query tersebut.

Metode Pertama

Karena kita dapat secara efisien menemukan ancestor ke-k dari setiap simpul di pohon, kita dapat menggunakan fakta ini untuk membagi masalah menjadi dua bagian. Kami menggunakan dua pointer yang awalnya menunjuk ke dua node yang nenek moyangnya paling rendah harus kita temukan. Pertama, kami memastikan bahwa pointer menunjuk ke node pada level yang sama di pohon. Jika hal ini tidak terjadi pada awalnya, kami memindahkan salah satu pointer ke atas. Setelah ini, kami menentukan jumlah langkah minimum yang diperlukan untuk memindahkan kedua pointer ke atas sehingga mereka akan menunjuk ke node yang sama. Node yang ditunjuk oleh pointer setelah ini adalah leluhur bersama terendah. Karena kedua bagian dari algoritme dapat dilakukan dalam waktu O(log n) menggunakan informasi yang telah dihitung sebelumnya, kita dapat menemukan leluhur bersama terendah dari dua node mana pun dalam waktu O(log n). Gambar 10.20 menunjukkan bagaimana kita dapat menemukan leluhur bersama terendah dari node 5 dan 8 dalam skenario contoh kita. Pertama, kita pindahkan pointer kedua satu tingkat ke atas sehingga menunjuk ke node 6 yang sejajar dengan node 5. Kemudian, kita pindahkan kedua pointer satu langkah ke atas ke node 2 yang merupakan common ancestor terendah.



Gambar 10.20 Dua langkah menemukan nenek moyang bersama terendah dari node 5 dan 8

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
node id	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
depth	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Gambar 10.21 Array traversal pohon yang diperluas untuk memproses kueri leluhur bersama terendah

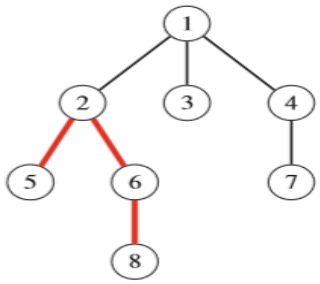
Metode Kedua

Cara lain untuk memecahkan masalah, diusulkan oleh Bender dan Farach Colton, didasarkan pada array traversal pohon diperpanjang, kadang-kadang disebut tourtree Euler. Untuk membangun array, kita menelusuri node pohon menggunakan pencarian depth-first dan menambahkan setiap node ke array selalu ketika pencarian depth-first berjalan melalui node (tidak hanya pada kunjungan pertama). Oleh karena itu, sebuah simpul yang memiliki k anak muncul k+1 kali dalam larik, dan ada total 2n−1 simpul dalam larik. Kami menyimpan dua nilai dalam array: pengidentifikasi simpul dan kedalaman simpul di pohon.

Gambar 10.21 menunjukkan larik yang dihasilkan dalam skenario contoh kita. Sekarang kita dapat menemukan leluhur bersama terendah dari node a dan b dengan mencari node dengan kedalaman minimum antara node a dan b dalam array. Sebagai contoh, Gambar 10.22 menunjukkan bagaimana menemukan leluhur bersama terendah dari simpul 5 dan 8. Kedalaman minimum simpul di antara mereka adalah simpul2 yang kedalamannya 2, jadi nenek moyang bersama terendah dari simpul 5 dan 8 adalah simpul 2. Perhatikan bahwa sejak sebuah node dapat muncul beberapa kali dalam array, mungkin ada beberapa cara untuk memilih posisi node a dan b. Namun, pilihan apa pun dengan benar menentukan leluhur bersama terendah dari node.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
node id	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
depth	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Gambar 10.22 Menemukan leluhur bersama terendah dari node 5 dan 8



Gambar 10.23 Menghitung jarak antara node 5 dan 8

Dengan menggunakan teknik ini, untuk menemukan leluhur bersama terendah dari dua node, cukup untuk memproses kueri rentang minimum. Cara yang biasa adalah dengan menggunakan pohon segmen untuk memproses kueri seperti itu dalam waktu O(log n). Namun, karena lariknya statis, kami juga dapat memproses kueri dalam waktu O(1) setelah pemrosesan awal waktu O(n log n).

Menghitung Jarak

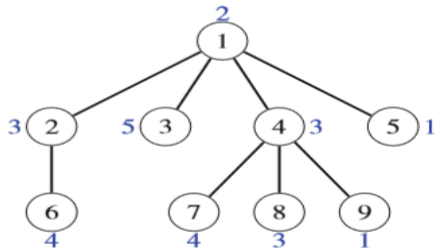
Akhirnya, pertimbangkan masalah pemrosesan kueri di mana kita perlu menghitung jarak antara node a dan b (yaitu, panjang jalur antara a dan b). Ternyata masalah ini direduksi menjadi menemukan leluhur bersama terendah dari node. Pertama, kita membasmi pohon secara sewenang-wenang. Setelah ini, jarak node a dan b dapat dihitung menggunakan rumus

$$\text{depth}(a)+\text{depth}(b)-2\cdot\text{depth}(c),$$

di mana c adalah leluhur bersama terendah dari a dan b. Sebagai contoh, untuk menghitung jarak antara node 5 dan 8 pada Gambar 10.23, pertama-tama kita tentukan bahwa nenek moyang terendah dari node adalah node 2. Kemudian, karena depthsofthenodesa adalah depth(5) =3, depth(8) =4, dan kedalaman(2) =2, kita simpulkan bahwa jarak antara simpul 5 dan 8 adalah 3+4−2·2=3.

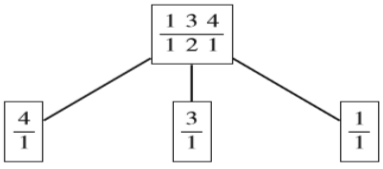
10.2.4 Menggabungkan Struktur Data

Sejauh ini, kita telah membahas algoritme online untuk kueri pohon. Algoritme tersebut dapat memproses kueri satu demi satu sedemikian rupa sehingga setiap kueri dijawab sebelum menerima kueri berikutnya. Namun, dalam banyak masalah, properti online tidak diperlukan, dan kami mungkin menggunakan algoritme offline untuk menyelesaikannya. Algoritma tersebut diberikan satu set lengkap pertanyaan yang dapat dijawab dalam urutan apapun. Algoritme offline seringkali lebih mudah dirancang daripada algoritme online.



Gambar 10.24 Subtree dari node 4 berisi dua node yang nilainya 3

Salah satu metode untuk membangun sebuah algoritma offline adalah dengan melakukan traversal pohon kedalaman-pertama dan memelihara struktur data dalam node. Pada setiap node s, kami membuat struktur data d[s] yang didasarkan pada struktur data anak-anak dari s. Kemudian, dengan menggunakan struktur data ini, semua kueri yang terkait dengan s diproses. Sebagai contoh, pertimbangkan masalah berikut: Kami diberikan pohon berakar di mana setiap node memiliki beberapa nilai. Tugas kita adalah memproses query yang meminta untuk menghitung jumlah node dengan nilai x pada subtree dari node s. Sebagai contoh, pada Gambar 10.24, subtree dari node 4 berisi dua node yang nilainya 3.



Gambar 10.25 Memproses query menggunakan struktur Maps

Gambar 10.26 Menggabungkan struktur Maps pada sebuah simpul

Dalam masalah ini, kita dapat menggunakan struktur Maps untuk menjawab pertanyaan. Sebagai contoh, Gambar 10.25 menunjukkan Maps untuk node 4 dan anak-anaknya. Jika kita membuat struktur data seperti itu untuk setiap node, kita dapat dengan mudah memproses semua kueri yang diberikan, karena kita dapat menangani semua kueri yang terkait dengan sebuah node segera setelah membuat struktur datanya. Namun, akan terlalu lambat untuk membuat semua struktur data dari awal.

Sebagai gantinya, pada setiap node, kami membuat struktur data awal d[s] yang hanya berisi nilai s. Setelah ini, kita melewati anak-anak dari s dan menggabungkan d[s]dan semua struktur data d[u] di mana u adalah anak dari s. Sebagai contoh, pada pohon di atas, Maps untuk simpul 4 dibuat dengan menggabungkan Maps pada Gambar 10.26. Di sini Maps pertama adalah struktur data awal untuk node 4, dan tiga Maps lainnya sesuai dengan node 7, 8, dan 9. Penggabungan pada node s dapat dilakukan sebagai berikut: Kami melewati anak-anak dari s dan pada setiap anak u menggabungkan d[s]dan d[u]. Kami selalu menyalin konten dari d[u]ke d[s]. Namun, sebelum ini, kita menukar isi d[s]dan d[u]jika d[s]lebih kecil dari d[u]. Dengan melakukan ini, setiap nilai hanya disalin O(log n) kali selama traversal pohon, yang memastikan bahwa algoritmanya efisien. Untuk menukar isi dua struktur data a dan b secara efisien, kita cukup menggunakan kode berikut:

```
swap(a,b);
```

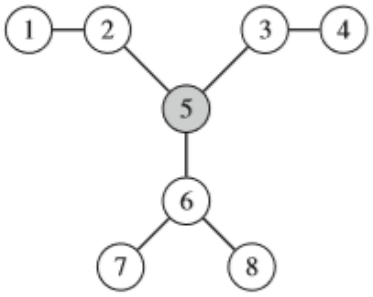
Ini dijamin bahwa kode di atas bekerja dalam waktu yang konstan ketika a dan b adalah struktur data perpustakaan standar C++.

10.3 Teknik Lanjutan

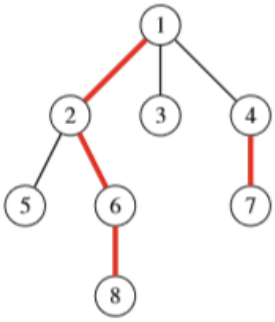
Pada bagian ini, kita membahas dua teknik pemrosesan pohon tingkat lanjut. Dekomposisi Centroid membagi pohon menjadi subpohon yang lebih kecil dan memprosesnya secara rekursif. Dekomposisi heavylight mewakili pohon sebagai kumpulan jalur khusus, yang memungkinkan kita untuk memproses kueri jalur secara efisien.

10.3.1 Dekomposisi Centroid

Sebuah centroid ofatreeofn node adalah node yang penghapusannya membagi pohon menjadi subtree yang masing-masing berisi paling banyak [n/2]node. Setiap pohon memiliki centroid, dan dapat ditemukan dengan rooting pohon secara sewenang-wenang dan selalu berpindah ke subtree yang memiliki jumlah node maksimum, hingga node saat ini adalah centroid. Dalam teknik dekomposisi centroid, pertama-tama kita mencari centroid dari pohon dan memproses semua jalur yang melalui centroid. Setelah ini, kami menghapus centroid dari pohon dan memproses subpohon yang tersisa secara rekursif. Karena menghapus centroid selalu menciptakan subpohon yang ukurannya paling banyak setengah dari ukuran pohon asli, kompleksitas waktu dari algoritma tersebut adalah O(n log n), asalkan kita dapat memproses setiap subpohon dalam waktu linier. Sebagai contoh, Gambar 10.27 menunjukkan langkah pertama dari algoritma dekomposisi centroid. Pada pohon ini, node 5 adalah satu-satunya centroid, jadi pertama-tama kita proses semua jalur yang melalui node 5. Setelah ini, node 5 dikeluarkan dari pohon, dan kita proses ketiga subtree {1, 2}, {3, 4 }, dan {6, 7, 8} secara rekursif.



Gambar 10.27 Dekomposisi centroid



Gambar 10.28 Dekomposisi berat-ringan

Dengan menggunakan dekomposisi centroid, kita dapat, misalnya, menghitung secara efisien jumlah jalur dengan panjang x dalam sebuah pohon. Saat memproses sebuah pohon, pertama-tama kita mencari pusat massa dan menghitung jumlah jalur yang melaluinya, yang dapat dilakukan dalam waktu linier. Setelah ini, kami menghapus centroid dan memproses pohon yang lebih kecil secara rekursif. Algoritma yang dihasilkan bekerja dalam waktu O (n log n).

10.3.2 Dekomposisi Berat-Ringan

Dekomposisi berat-ringan[11] membagi node pohon menjadi satu set jalur yang disebut jalur berat. Jalur berat dibuat sehingga jalur antara dua simpul pohon dapat direpresentasikan sebagai sub jalur O(log n) dari jalur berat. Dengan menggunakan teknik ini, kita dapat memanipulasi node pada jalur antara node pohon hampir seperti elemen dalam array, dengan hanya faktor O(log n) tambahan.

Untuk membangun jalur berat, pertama-tama kita melakukan root pada pohon secara sewenang-wenang. Kemudian, kita memulai jalur berat pertama di akar pohon dan selalu pindah ke simpul yang memiliki ukuran subpohon maksimum. Setelah ini, kami memproses subpohon yang tersisa secara rekursif. Misalnya, pada Gambar 10.28, ada empat jalur berat: 1–2–6–8, 3, 4–7, dan 5 (perhatikan bahwa dua jalur hanya memiliki satu simpul). Sekarang, pertimbangkan jalur mana pun antara dua simpul di pohon. Karena kami selalu memilih subpohon ukuran maksimum saat membuat jalur berat, ini menjamin bahwa kami dapat membagi jalur menjadi O(log n) subjalur sehingga masing-masing dari mereka adalah subjalur dari satu jalur berat. Sebagai contoh, pada Gambar 10.28, jalur antara node 7 dan 8 dapat dibagi menjadi dua subpath berat: pertama 7-4, kemudian 1-2-6-8.

Manfaat dekomposisi berat-ringan adalah bahwa setiap jalur berat dapat diperlakukan seperti array node. Misalnya, kita dapat menetapkan pohon segmen untuk setiap jalur

[11] Sleator dan Tarjan memperkenalkan ide tersebut dalam konteks struktur data link/cut tree mereka.

berat dan mendukung kueri jalur yang canggih, seperti menghitung nilai simpul minimum dalam suatu jalur atau meningkatkan nilai setiap simpul dalam suatu jalur. Kueri tersebut dapat diproses dalam waktu $O(\log^2 n)$[12], karena setiap jalur terdiri dari O(log n) jalur berat dan setiap jalur berat dapat diproses dalam waktu O(log n). Meskipun banyak masalah dapat diselesaikan dengan menggunakan dekomposisi berat-ringan, ada baiknya untuk diingat bahwa seringkali ada solusi lain yang lebih mudah untuk diterapkan.

[12] Notasi logk n sesuai dengan (log n)k.

BAB 11
MATEMATIKA

11.1 Teori Bilangan

Teori bilangan adalah cabang matematika yang mempelajari bilangan bulat. Pada bagian ini, kita akan membahas pilihan topik dan algoritma teori bilangan, seperti menemukan bilangan prima dan faktor, dan memecahkan persamaan bilangan bulat.

11.1.1 Faktor dan Bilangan Prima

Suatu bilangan bulat a disebut faktor atau pembagi bilangan bulat b jika a membagi b. Jikaa adalah faktor dari b, kita tulis a | b, dan sebaliknya kita tulis a ł b. Misalnya, faktor dari 24 adalah 1, 2, 3, 4, 6, 8, 12, dan 24. Suatu bilangan bulat n > 1 adalah prima jika hanya faktor positifnya adalah 1 dan n. Misalnya, 7, 19, dan 41 adalah bilangan prima, tetapi 35 bukanlah bilangan prima, karena 5x7=35. Untuk setiap bilangan bulat n > 1, ada faktorisasi prima yang unik

$$n = p_1^{\alpha 1} p_2^{\alpha 2} \dots p_k^{\alpha k}$$

di mana p_1, p_2,...,p_k adalah bilangan prima yang berbeda dan α_1, α_2,...,α_k adalah bilangan bulat positif. Misalnya, faktorisasi prima untuk 84 adalah

$$84 = 2^2 x 3^1 x 7^1$$

Misalkan τ(n) menyatakan banyaknya faktor dari suatu bilangan bulat n. Misal τ(12) = 6, karena faktor dari 12 adalah 1, 2, 3, 4, 6, dan 12. Untuk menghitung nilai τ (n), kita dapat menggunakan rumus

$$\tau(n) = \prod_{i=1}^{k} (\alpha_i + 1)$$

karena untuk setiap pi prima, ada α i+1 cara untuk memilih berapa kali muncul dalam faktor. Misalnya, karena $12=2^2 \cdot 3$, τ (12) =3x2=6. Kemudian, misalkan σ(n) menyatakan jumlah faktor dari suatu bilangan bulat. Misalnya, σ(12) =28, karena 1+2+3+4+6+12 = 28. Untuk menghitung nilai σ(n), kita dapat menggunakan rumus

$$\sigma(n) = \prod_{i=1}^{k} \left(1 + p_i + \cdots + p_i^{\alpha i} = \prod_{i=1}^{k} \left(\frac{p_i^{\alpha i+1} - 1}{p_i - 1} \right) \right.$$

di mana bentuk yang terakhir didasarkan pada rumus deret geometri. Misalnya, σ(12) = $(2^3\ 1)/(2-1)\cdot(3^2\ 1)/(3-1)$ =28.

Algoritma Dasar

Jika suatu bilangan bulat n bukan prima, dapat direpresentasikan sebagai produk a x b, di mana a ≤ √n atau b ≤ √n, sehingga pasti memiliki faktor antara 2 dan [√n] . Dengan menggunakan pengamatan ini, kita dapat menguji apakah suatu bilangan bulat adalah bilangan prima dan menemukan faktorisasi primanya dalam waktu O(√n). Fungsi prima berikut memeriksa apakah bilangan bulat n yang diberikan adalah prima. Fungsi tersebut mencoba membagi dengan semua bilangan bulat antara 2 dan [√n] , dan jika tidak ada yang membagi n, maka n adalah prima.

```
bool prime(int n) {
        if (n < 2)return false;
        for (int x = 2; x*x <= n; x++) {
        if (n%x == 0)
        return false;
        }
```

```
        return true;
    }
```

Kemudian, faktor-faktor fungsi berikut ini membangun sebuah vektor yang berisi faktorisasi prima dari n. Fungsi membagi n dengan faktor primanya dan menambahkannya ke vektor.Proses berakhir ketika ada bilangan yang tidak memiliki faktor antara 2 dan [√n]. Jika n > 1, itu adalah prima dan faktor terakhir

```
vector<int> factors(int n) {
        vector<int> f;
        for (int x = 2; x*x <= n; x++) {
        while (n%x == 0) {
        f.push_back(x);
        n /= x;
        }
        }
        if (n > 1) f.push_back(n);
        return f;
}
```

Perhatikan bahwa setiap faktor prima muncul dalam vektor sebanyak ia membagi bilangan tersebut. Misalnya, $12=2^2 \times 3$, maka hasil dari fungsi tersebut adalah [2, 2, 3].

Sifat-sifat bilangan prima

Sangat mudah untuk menunjukkan bahwa ada bilangan prima yang tak berhingga. Jika jumlah bilangan prima akan terbatas, kita dapat membuat himpunan P ={p_1, p_2,...,p_n} yang akan memuat semua bilangan prima. Misalnya, p_1 =2, p_2 =3, p_3 =5, dan seterusnya. Namun, dengan menggunakan himpunan P seperti itu, kita dapat membentuk bilangan prima baru

$$p_1 p_2 \cdots p_n + 1$$

yang akan lebih besar dari semua elemen di P. Ini adalah kontradiksi, dan jumlah bilangan prima harus tak terhingga. *Fungsi penghitungan prima $\pi(n)$* memberikan jumlah bilangan prima. Misalnya, π(10) =4, karena bilangan prima hingga 10 adalah 2, 3, 5, dan 7. Dapat ditunjukkan bahwa

$$\pi(n) \approx \frac{n}{1n\,n}$$

yang berarti bahwa bilangan prima cukup sering. Misalnya, pendekatan untuk $\pi(10^6)$ adalah $10^6/\ln 10^6 \approx 72382$, dan nilai eksaknya adalah 78498.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	1	0	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1

Gambar 11.1 Hasil saringan Eratosthenes untuk n =20

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	3	2	5	2	7	2	3	2	11	2	13	2	3	2	17	2	19	2

Gambar 11.2 Sebuah saringan Eratosthenes yang diperluas yang berisi faktor prima terkecil dari setiap bilangan

11.1.2 Saringan Eratosthenes

Saringan Eratosthenes adalah algoritme prapemrosesan yang membangun saringan array dari mana kita dapat memeriksa secara efisien apakah ada bilangan bulat x

antara 2...n yang prima. Jika x prima, maka sieve[x]=0, dan sebaliknya sieve[x]=1. Sebagai contoh, Gambar.11.1 menunjukkan isi saringan untuk n =20. Untuk membangun array, algoritma iterasi melalui bilangan bulat 2...n satu per satu. Selalu ketika bilangan prima x baru ditemukan, algoritme mencatat bahwa bilangan 2x,3x,4x, dll., bukan bilangan prima. Algoritma dapat diimplementasikan sebagai berikut, dengan asumsi bahwa setiap elemen saringan awalnya nol:

```
for (int x = 2; x <= n; x++) {
        if (sieve[x]) continue;
                for (int u = 2*x; u <= n; u += x) {
        sieve[u] = 1;
        }
}
```

Loop dalam dari algoritma dieksekusi [n/x] kali untuk setiap nilai x. Jadi, batas atas untuk waktu berjalan dari algoritma adalah jumlah harmonik

$$\sum_{x=2}^{n} \left[\frac{n}{x}\right] = \left[\frac{n}{2}\right] + \left[\frac{n}{3}\right] + \left[\frac{n}{4}\right] + \cdots = O(n \log n)$$

Faktanya, algoritma ini lebih efisien, karena loop dalam hanya akan dieksekusi jika bilangan x adalah bilangan prima. Dapat ditunjukkan bahwa waktu berjalan dari algoritma hanya O(n log log n), kompleksitas yang sangat dekat dengan O(n). Dalam praktiknya, saringan Eratosthenes sangat efisien; Tabel 11.1 menunjukkan beberapa waktu berjalan yang sebenarnya. Ada beberapa cara untuk memperpanjang saringan Eratosthenes. Sebagai contoh, kita dapat menghitung untuk setiap bilangan k faktor prima terkecilnya (Gambar 11.2). Setelah ini, kita dapat memfaktorkan secara efisien bilangan apa pun antara 2...n dengan menggunakan saringan.(Perhatikan bahwa bilangan n memiliki O(log n) faktor prima.)

Tabel 11.1 Waktu berjalan dari saringan Eratosthenes

Batas ata *n*	Running time (s)
10^6	0.01
2 x 10^6	0.03
4 x 10^6	0.07
8 x 10^6	0.14
16 x 10^6	0.28
32 x 10^6	0.57
64 x 10^6	1.16
128 x 10^6	2.35

11.1.3 Algoritma Euclid

Pembagi persekutuan terbesar dari bilangan bulat a dan b, dilambangkan dengan gcd(a,b), adalah bilangan bulat terbesar yang membagi a dan b. Misalnya, gcd(30,12) = 6. Konsep terkait adalah kelipatan persekutuan terkecil, dilambangkan lcm(a,b), yang merupakan bilangan bulat terkecil yang habis dibagi a dan b. Rumusnya

$$lcm(a,b) = \frac{ab}{\gcd(a,b)}$$

dapat digunakan untuk menghitung kelipatan persekutuan terkecil. Misalnya, lcm(30,12) = 360/gcd(30,12) =60. Salah satu cara untuk mencari gcd(a, b) adalah

dengan membagi a dan b menjadi faktor prima, dan kemudian memilih untuk setiap bilangan prima pangkat terbesar yang muncul pada kedua faktorisasi. Misalnya, untuk menghitung gcd(30,12), kita dapat membuat faktorisasi 30 = 2 · 3 · 5 dan 12=2^2 · 3, dan menyimpulkan bahwa gcd(30,12) =2 x 3=6. Namun, teknik ini tidak efisien jika a dan b adalah bilangan besar. Algoritma Euclid menyediakan cara yang efisien untuk menghitung nilai gcd(a,b). Algoritma ini didasarkan pada rumus

$$g(a,b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & b \neq 0 \end{cases}$$

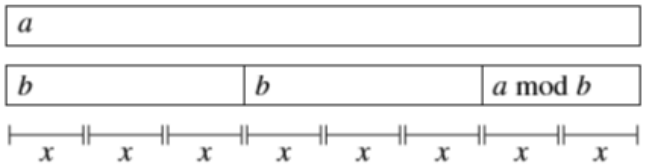
Sebagai contoh,

gcd(30,12) =gcd(12,6) =gcd(6,0) =6.

Algoritma tersebut dapat diimplementasikan sebagai berikut:

```
int gcd(int a, int b) {
        if (b == 0) return a;
                return gcd(b, a%b);
}
```

Mengapa algoritma bekerja? Untuk memahami hal ini, perhatikan Gambar.11.3. dimana x = gcd(a, b). Karena x membagi a dan b, ia juga harus membagi a mod b, yang menunjukkan mengapa rumus rekursif berlaku. Dapat dibuktikan bahwa algoritma Euclid bekerja dalam waktu O(log n), dimana n =min (a, b).



Gambar 11.3 Mengapa algoritma Euclid bekerja?

Algoritma Euclid yang Diperluas

Algoritma Euclid juga dapat diperluas sehingga memberikan bilangan bulat x dan y yang

ax + by = gcd(a, b).

Misalnya, ketika a =30 dan b =12,

30 x 1 + 12 x (−2) = 6.

Kita juga dapat menyelesaikan masalah ini dengan menggunakan rumus gcd(a,b) =gcd(b, a mod b). Misalkan kita telah memecahkan masalah untuk gcd(b, a mod b), dan kita mengetahui nilai x' dan y' yang

bx' + (a mod b)y' = gcd(a, b).

Kemudian, karena a mod b =a−[a/b] x b,

bx' + (a−[a/b] x b)y' = gcd(a, b),

yang sama dengan

ay' + (a-[a/b] x b)y' = gcd(a, b)

Jadi, kita dapat memilih x = y' dan y = x' [a/b] x y'. Dengan menggunakan ide ini, fungsi berikut mengembalikan sebuah tupel (x, y, gcd(a, b)) yang memenuhi persamaan.

```
tuple<int,int,int> gcd(int a, int b) {
        if (b == 0) {
                return {1,0,a};
                        } else {
                int x,y,g;
                tie(x,y,g) = gcd(b,a%b);
```

```
        return {y,x-(a/b)*y,g};
        }
}
```

Kita dapat menggunakan fungsi sebagai berikut:

```
int x,y,g;
tie(x,y,g) = gcd(30,12);
cout << x << " " << y << " " << g << "\n"; //1-2 6
```

11.1.4 Modular Eksponensial

Seringkali ada kebutuhan untuk menghitung nilai x^n mod m secara efisien. Ini dapat dilakukan dalam waktu O(log n) menggunakan rumus rekursif berikut:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{\frac{n}{2}} . x^{\frac{n}{2}} & n \; is \; even \\ x^{n-1} . x & n \; is \; old \end{cases}$$

Misalnya, untuk menghitung nilai x^{100}, pertama-tama kita hitung nilai x^{50}, lalu gunakan rumus $x^{100} = x^{50} \cdot x^{50}$. Kemudian, untuk menghitung nilai x^{50}, pertama-tama kita menghitung nilai x^{25} dan seterusnya. Karena n selalu dibelah dua ketika genap, perhitungannya hanya membutuhkan waktu O(log n). Algoritma tersebut dapat diimplementasikan sebagai berikut:

```
int modpow(int x, int n, int m) {
        if (n == 0) return 1%m;
        long long u = modpow(x,n/2,m);
        u = (u*u)%m;
        if (n%2 == 1) u = (u*x)%m;
        return u;
}
```

11.1.5 Teorema Euler

Dua bilangan bulat a dan b disebut koprima jika gcd(a,b) = 1. *Fungsi total Euler $\varphi(n)$*memberikan jumlah bilangan bulat antara 1...n yang koprima dengan n. Misalnya, φ (10) =4, karena 1, 3, 7, dan 9 adalah bilangan prima sampai 10. Nilai φ (n) berapa pun dapat dihitung dari faktorisasi prima n menggunakan rumus

$$\varphi(n) = \prod_{i=1}^{k} P_i^{\alpha i-1}(p_i - 1)$$

Misalnya, karena 10=2·5, φ (10) =$2^0 \cdot (2-1) \cdot 5^0 \cdot (5-1)$ =4.

Teorema Euler menyatakan bahwa

$$x^{\varphi(m)} \bmod m = 1$$

untuk semua bilangan bulat koprima positif x dan m. Misalnya, teorema Euler memberi tahu kita bahwa $7^4 \bmod 10=1$, karena 7 dan 10 adalah koprima dan φ (10) =4. Jika m prima, φ (m) =m−1, maka rumusnya menjadi

$$x^{m-1} \bmod m = 1,$$

yang dikenal sebagai teorema kecil Fermat. Ini juga menyiratkan bahwa

$$x^n \bmod m = x^{n \bmod (m-1)} \bmod m,$$

yang dapat digunakan untuk menghitung nilai xn jika n sangat besar.

Kebalikan Perkalian Modular

Invers perkalian modular dari x terhadap m adalah nilai $inv_m(x)$ sedemikian sehingga

$$x \cdot inv_m(x) \bmod m = 1.$$

Misalnya, $inv_{17}(6) = 3$, karena $6 \cdot 3 \bmod 17 = 1$. Dengan menggunakan invers perkalian modular, kita dapat membagi bilangan modulo m, karena pembagian dengan x sama dengan perkalian dengan $inv_m(x)$. Misalnya, karena kita tahu bahwa $inv_{17}(6) = 3$, kita dapat menghitung nilai 36/6 mod 17 dengan cara lain menggunakan rumus $36 \cdot 3 \bmod 17$. Invers perkalian modular ada tepat ketika x dan m koprima. Dalam hal ini, dapat dihitung menggunakan rumus

$$inv_m(x) = x^{\phi(m)-1},$$

yang didasarkan pada teorema Euler. Khususnya, jika m prima, $\phi(m) = m-1$ dan rumusnya menjadi

$$inv_m(x) = x^{m-2}.$$

Sebagai contoh,

$$inv_{17}(6) \bmod 17 = 6^{17-2} \bmod 17 = 3.$$

Rumus di atas memungkinkan kita untuk menghitung invers perkalian modular secara efisien menggunakan algoritme eksponensial modular.

11.1.6 Menyelesaikan Persamaan

Persamaan Diophantine

Persamaan Diophantine adalah persamaan bentuk

$$ax+by=c,$$

di mana a, b, dan c adalah konstanta dan nilai x dan y harus dicari. Setiap angka dalam persamaan harus bilangan bulat. Misalnya, satu solusi untuk persamaan

$$5x + 2y = 11$$

adalah $x = 3$ dan $y = -2$. Kita dapat menyelesaikan persamaan Diophantine secara efisien dengan menggunakan algoritma Euclid yang diperluas yang memberikan bilangan bulat x dan y yang memenuhi persamaan

$$ax+by=gcd(a,b).$$

Persamaan Diophantine dapat diselesaikan dengan tepat jika c habis dibagi oleh gcd(a,b). Sebagai contoh, mari kita cari bilangan bulat x dan y yang memenuhi persamaan

$$39x + 15y = 12.$$

Persamaan dapat diselesaikan, karena $gcd(39,15) = 3$ dan $3 \mid 12$. Algoritma Euclid yang diperluas memberi kita

$$39 \cdot 2 + 15 \cdot (-5) = 3,$$

dan dengan mengalikannya dengan 4, persamaannya menjadi

$$39 \cdot 8 + 15 \cdot (-20) = 12,$$

jadi solusi persamaannya adalah $x = 8$ dan $y = -20$. Solusi untuk persamaan Diophantine tidak unik, karena kita dapat membentuk sejumlah solusi tak berhingga jika kita mengetahui satu solusi. Jika pasangan (x,y) adalah solusi, maka semua pasangan juga

$$\left(x + \frac{kb}{gcd(a,b)}, y - \frac{ka}{gcd(a,b)}\right)$$

adalah solusi, di mana k adalah sembarang bilangan bulat.

Teorema Sisa Cina

Teorema sisa Cina memecahkan sekelompok persamaan bentuk

$$x = a1 \bmod m1$$
$$x = a2 \bmod m2$$
$$\ldots$$
$$x = an \bmod mn$$

dimana semua pasangan m_1, m_2,..., m_n adalah koprima. Ternyata solusi dari persamaan tersebut adalah

$$x = a_1X_1inv_{m1}(X_1)+a_2X_2inv_{m2}(X_2)+\cdots+a_nX_ninv_{mn}(X_n),$$

di mana

$$X_k = \frac{m_1 m_2 \ldots m_n}{m_k}$$

Dalam solusi ini, untuk setiap k =1, 2,..., n,

$$a_kX_kinv_{mk}(X_k) \bmod m_k = a_k,$$

karena

$$X_kinv_{mk}(X_k) \bmod m_k = 1.$$

Karena semua suku lain dalam jumlah habis dibagi mk, mereka tidak berpengaruh pada sisa dan x mod $m_k = a_k$. Misalnya, solusi untuk

$$x = 3 \bmod 5$$
$$x = 4 \bmod 7$$
$$x = 2 \bmod 3$$

adalah

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263.$$

Setelah kita menemukan solusi x, kita dapat membuat sejumlah tak hingga dari solusi lain, karena semua bilangan berbentuk

$$x + m_1m_2\cdots m_n$$

adalah solusi.

11.2 Kombinatorik

Combinatorics mempelajari metode untuk menghitung kombinasi objek. Biasanya, tujuannya adalah menemukan cara untuk menghitung kombinasi secara efisien tanpa menghasilkan setiap kombinasi secara terpisah. Pada bagian ini, kita membahas pilihan teknik kombinatorial yang dapat diterapkan pada sejumlah besar masalah.

11.2.1 Koefisien Binomial

Koefisien binomial (n/k) memberikan banyak cara untuk memilih suatu subset dari k elemen dari n elemen. Misalnya, (5/3) = 10, karena himpunan {1, 2, 3, 4, 5}memiliki 10 himpunan bagian dari 3 elemen:

{1,2,3},{1,2,4},{1,2,5},{1,3,4},{1,3,5},
{1,4,5},{2,3,4},{2,3,5},{2,4,5},{3,4,5}

Koefisien binomial dapat dihitung secara rekursif menggunakan rumus

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

dengan kasus dasar

$$\binom{n}{0} = \binom{n}{n} = 1$$

Untuk melihat mengapa rumus ini bekerja, pertimbangkan elemen arbitrer x dalam himpunan. Jika kita memutuskan untuk memasukkan x dalam subset kita, tugas yang tersisa adalah memilih k 1 elemen dari n−1 elemen. Kemudian, jika kita tidak memasukkan x dalam himpunan bagian kita, kita harus memilih k elemen dari n−1 elemen. Cara lain untuk menghitung koefisien binomial adalah dengan menggunakan rumus

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

yang didasarkan pada alasan berikut: Ada n! permutasi dari n elemen. Kami melewati semua permutasi dan selalu memasukkan k elemen pertama dari permutasi dalam himpunan bagian. Karena urutan elemen dalam himpunan bagian dan di luar himpunan bagian tidak penting, hasilnya dibagi dengan k!dan (n−k)! Untuk koefisien binomial,

$$\binom{n}{k} = \binom{n}{n-k}$$

karena kita sebenarnya membagi himpunan n elemen menjadi dua himpunan bagian: yang pertama berisi k elemen dan yang kedua berisi n−k elemen. Jumlah koefisien binomial adalah

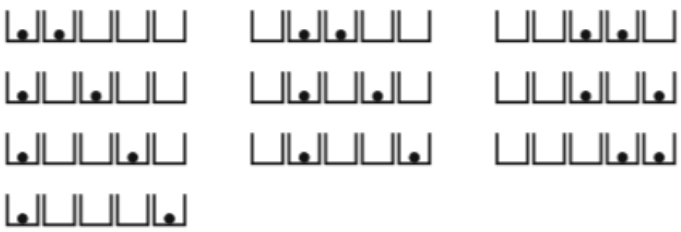
$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = 2^n$$

Alasan untuk nama "koefisien binomial" dapat dilihat ketika binomial (a+b) dipangkatkan ke n:

$$(a+b)^n = \binom{n}{0} a^n b^0 + \binom{n}{1} a^{n-1} b^1 + \cdots \binom{n}{n-1} a^1 b^{n-1} + \binom{n}{n} a^0 b^n$$

Koefisien binomial juga muncul dalam segitiga Pascal (Gambar 11.4) di mana setiap nilai sama dengan jumlah dua nilai di atas.

Gambar 11.4 5 baris pertama segitiga Pascal



Gambar 11.5 Skenario 1: Setiap kotak berisi paling banyak satu bola

Koefisien Multinomial

$$\binom{n}{k_1, k_2, \ldots, k_m} = \frac{n!}{k_1!\ k_2! \ldots k_m!}$$

memberikan jumlah cara suatu himpunan n elemen dapat dibagi menjadi himpunan bagian dengan ukuran k_1, k_2,..., k_m, di mana $k_1 + k_2 + \cdots + k_m = n$. Koefisien multinomial dapat dilihat sebagai generalisasi dari koefisien binomial; jika m =2, rumus di atas sesuai dengan rumus koefisien binomial.

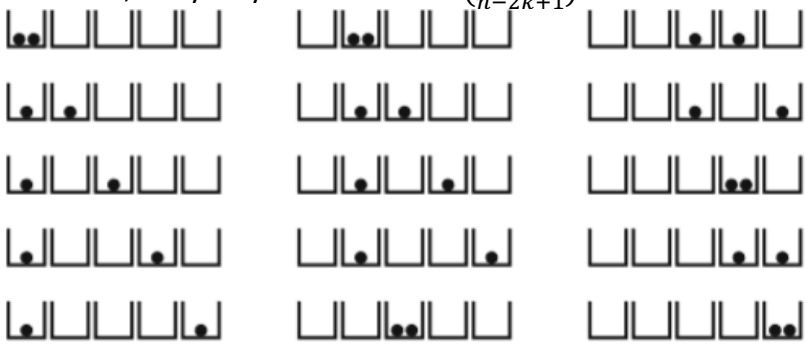
Kotak dan Bola

"Kotak dan bola" adalah model yang berguna, di mana kita menghitung cara untuk menempatkan k bola dalam n kotak. Mari kita pertimbangkan tiga skenario:

Skenario 1: Setiap kotak dapat berisi paling banyak satu bola. Misalnya, ketika n = 5 dan k = 2, ada 10 kombinasi (Gambar 11.5). Dalam skenario ini, jumlah kombinasi secara langsung adalah koefisien binomial $\binom{n}{k}$

Skenario 2: Sebuah kotak dapat berisi beberapa bola. Misalnya, ketika n = 5 dan k =2, ada 15 kombinasi (Gambar 11.6). Dalam skenario ini, proses penempatan bola di dalam kotak dapat direpresentasikan sebagai string yang terdiri dari simbol "o" dan "→." Awalnya, asumsikan bahwa kita berdiri di kotak paling kiri. Simbol "o" berarti kita menempatkan bola di kotak saat ini, dan simbol "→" berarti kita pindah ke kotak berikutnya di sebelah kanan. Sekarang setiap solusi adalah string dengan panjang *k+n−1* yang berisi k simbol "o" dan n−1 simbol "→." Misalnya, solusi kanan atas pada Gambar.11.6 sesuai dengan string "→→o→ o →." Dengan demikian, kita dapat menyimpulkan bahwa jumlah kombinasi adalah $\binom{k+n-1}{k}$

Skenario 3: Setiap kotak dapat berisi bola atmostone, dan tambahan, tidak dua kotak yang berdekatan dapat berisi bola. Misalnya, ketika n = 5 dan k = 2, ada 6 kombinasi (Gambar 11.7). Dalam skenario ini, kita dapat mengasumsikan bahwa k bola awalnya ditempatkan di dalam kotak dan ada kotak kosong di antara setiap dua kotak yang berdekatan. Tugas yang tersisa adalah memilih posisi untuk kotak kosong yang tersisa. Ada n 2k +1 kotak seperti itu dan k +1 posisi untuk mereka. Jadi, dengan menggunakan rumus Skenario 2, banyaknya solusi adalah $\binom{n-k+1}{n-2k+1}$



Gambar 11.6 Skenario 2: Sebuah kotak dapat berisi beberapa bola

Gambar 11.7 Skenario 3: Setiap kotak berisi paling banyak satu bola dan tidak ada dua kotak yang berdekatan yang berisi bola

11.2.2 Nomor Catalan

Nomor Catalan C_n memberikan jumlah ekspresi kurung yang valid yang terdiri dari n kurung kiri dan n kurung kanan. Misalnya, C_3 =5, karena kita dapat membuat total lima ekspresi kurung menggunakan tiga kurung kiri dan tiga kurung kanan:

- ()()()
- (())()
- ()(())
- ((()))
- (()())

Apa sebenarnya ekspresi kurung yang valid? Aturan berikut secara tepat mendefinisikan semua ekspresi kurung yang valid:

- Ekspresi kurung kosong valid.
- Jika ekspresi A valid, maka ekspresi (A) juga valid.

- Jika ekspresi A dan B valid, maka ekspresi AB juga valid.

Cara lain untuk mengkarakterisasi ekspresi tanda kurung yang valid adalah bahwa jika kita memilih awalan dari ekspresi seperti itu, ekspresi tersebut harus mengandung setidaknya kurung kiri sebanyak kurung kanan, dan ekspresi lengkap harus berisi jumlah kurung kiri dan kanan yang sama.

Nomor Catalan dapat dihitung menggunakan rumus

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$$

di mana kami mempertimbangkan cara untuk membagi ekspresi kurung menjadi dua bagian yang keduanya ekspresi kurung yang valid, dan bagian pertama mungkin sesingkat mungkin tetapi tidak kosong. Untuk setiap i, bagian pertama berisi i +1 pasang kurung dan jumlah ekspresi yang valid adalah produk dari nilai-nilai berikut:

- C_i:jumlah cara untuk membuat ekspresi kurung menggunakan tanda kurung bagian pertama, tidak termasuk tanda kurung terluar
- C_{n-i-1}:jumlah cara untuk membuat ekspresi kurung menggunakan tanda kurung bagian kedua.

Kasus dasarnya adalah C_0=1, karena kita dapat membuat ekspresi kurung kosong menggunakan nol pasang kurung. Nomor Catalan juga dapat dihitung menggunakan rumus

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

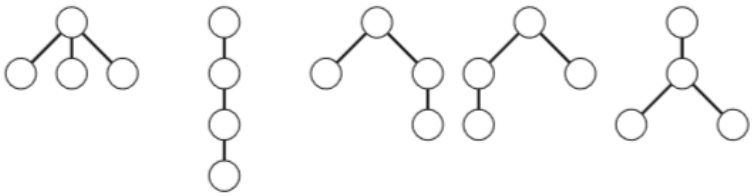
yang dapat dijelaskan sebagai berikut: Ada total $\binom{2n}{n}$ cara untuk membangun ekspresi kurung (belum tentu valid) yang berisi n kurung kiri dan n kurung kanan. Mari kita hitung jumlah ekspresi seperti itu yang tidak valid. Jika ekspresi kurung tidak valid, harus berisi awalan di mana jumlah kurung kanan melebihi jumlah kurung kiri. Idenya adalah untuk memilih awalan terpendek dan membalikkan setiap tanda kurung di awalan. Misalnya, ekspresi ())()(memiliki awalan ()), dan setelah membalik tanda kurung, ekspresi menjadi)((()(. Ekspresi yang dihasilkan terdiri dari n +1 kiri dan n 1 kurung kanan. Sebenarnya, ada cara unik untuk menghasilkan ekspresi n+1 kiri dan n−1 kurung kanan dengan cara di atas.Jumlah ekspresi tersebut adalah$\binom{2n}{n+1}$, yang sama dengan jumlah ekspresi kurung yang tidak valid. Jadi, jumlah ekspresi kurung yang valid dapat dihitung dengan menggunakan rumus

$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1}\binom{2n}{n} = \frac{1}{n+1}\binom{2n}{n}$$

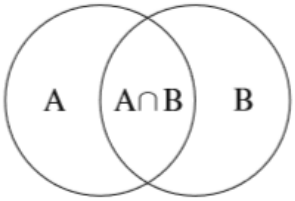
Menghitung Pohon

Kita juga dapat menghitung struktur pohon tertentu menggunakan bilangan Catalan. Pertama, C_n sama dengan jumlah pohon biner dari n node, dengan asumsi bahwa anak kiri dan anak kanan dibedakan. Misalnya, karena C_3 =5, ada 5 pohon biner dari 3 node (Gambar 11.8). Kemudian, Cn juga sama dengan jumlah pohon berakar umum dari n+1 node. Misalnya, ada 5 pohon berakar dari 4 simpul (Gambar 11.9).

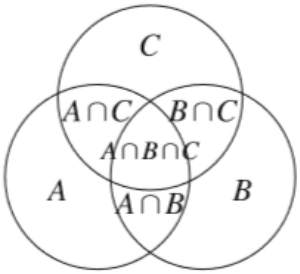
Gambar 11.8 Ada 5 pohon biner dari 3 node



Gambar 11.9 Ada 5 pohon berakar dari 4 node



Gambar 11.10 Prinsip inklusi-eksklusi untuk dua set



Gambar 11.11 Prinsip inklusi-eksklusi untuk tiga set

11.2.3 Inklusi-eksklusi

Inklusi-eksklusi adalah teknik yang dapat digunakan untuk menghitung ukuran suatu himpunan jika ukuran persimpangan diketahui, dan sebaliknya. Contoh sederhana dari teknik ini adalah rumusnya

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

di mana A dan B adalah himpunan dan |X| menunjukkan ukuran X. Gambar 11.10 mengilustrasikan rumus. Dalam hal ini, kami ingin menghitung ukuran persatuan A∪B yang sesuai dengan luas daerah yang dimiliki oleh setidaknya satu lingkaran pada Gambar 11.10. Kita dapat menghitung luas A ∪ B dengan terlebih dahulu menjumlahkan luas A dan B dan kemudian mengurangkan luas A ∩ B dari hasilnya. Ide yang sama dapat diterapkan ketika jumlah set lebih besar. Ketika ada tiga set, rumus inklusi-eksklusi adalah

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|,$$

yang sesuai dengan Gambar 11.11. Dalam kasus umum, ukuran persatuan X_1 X_2 ⋯ X_n dapat dihitung dengan melalui semua persimpangan yang mungkin yang berisi beberapa himpunan X_1, X_2,..., X_n.

Jika sebuah persimpangan berisi jumlah himpunan ganjil, ukurannya ditambahkan ke jawaban, dan jika tidak, ukurannya dikurangi dari jawaban. Perhatikan bahwa ada rumus serupa untuk menghitung ukuran persimpangan dari ukuran serikat pekerja. Sebagai contoh,

$$|A \cap B| = |A| + |B| - |A \cup B|$$

dan

$$|A \cap B \cap C| = |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C| + |A \cup B \cup C|.$$

Menghitung Kekacauan

Sebagai contoh, mari kita hitung jumlah derangements dari {1,2,...,n}, yaitu permutasi di mana tidak ada elemen yang tersisa di tempat asalnya. Misalnya, ketika n =3, ada dua derangements: (2,3,1) dan (3,1,2). Salah satu pendekatan untuk memecahkan masalah adalah dengan menggunakan inklusi-eksklusi. Misalkan X_k adalah himpunan permutasi yang memuat elemen k pada posisi k. Misalnya, ketika n = 3, himpunannya adalah sebagai berikut:

$$X1 = \{(1,2,3),(1,3,2)\}$$
$$X2 = \{(1,2,3),(3,2,1)\}$$
$$X3 = \{(1,2,3),(2,1,3)\}$$

Jumlah derangements sama dengan

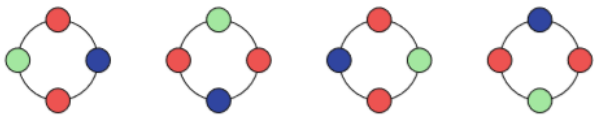
$$n! - |X_1 \cup X_2 \cup \cdots \cup X_n|,$$

sehingga cukup untuk menghitung $|X_1 X_2 \cdots X_n|$. Menggunakan inklusi-pengecualian, ini mengurangi untuk menghitung ukuran persimpangan. Selain itu, perpotongan dari c himpunan berbeda X_k memiliki (n−c)! elemen, karena perpotongan seperti itu terdiri dari semua permutasi yang mengandung elemen c di tempat asalnya. Dengan demikian, kita dapat menghitung ukuran persimpangan secara efisien. Misalnya, ketika n = 3,

$$|X_1 \cup X_2 \cup X_3| = |X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3|$$
$$= 2+2+2-1-1-1+1$$
$$= 4,$$

jadi banyaknya kekacauan adalah 3!−4=2. Ternyata masalah juga bisa diselesaikan tanpa menggunakan inklusi-eksklusi. Misalkan f (n) menyatakan banyaknya derangements untuk {1, 2,..., n}. Kita dapat menggunakan rumus rekursif berikut:

$$f(n) \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ (n-1)\big(f(n-2) + f(n-1)\big) & n > 2 \end{cases}$$



Gambar 11.12 Empat kalung simetris

Rumus dapat dibuktikan dengan mempertimbangkan kemungkinan bagaimana elemen 1 berubah dalam kekacauan. Ada n 1 cara untuk memilih elemen x yang menggantikan elemen 1. Dalam setiap pilihan tersebut, ada dua opsi*: Opsi1*:Kita juga mengganti elemenx dengan elemen 1. Setelah ini, tugas utamanya adalah membuat derangement dari n−2 elemen. *Opsi 2*: Kita mengganti elemen x dengan beberapa elemen selain 1. Sekarang kita harus membuat derangement dari n 1 elemen, karena kita tidak dapat mengganti elemen x dengan elemen 1, dan semua elemen lainnya harus diubah.

11.2.4 Lemma Burnside

Lemma Burnside dapat digunakan untuk menghitung jumlah kombinasi yang berbeda sehingga kombinasi simetris dihitung hanya sekali. Lemma Burnside menyatakan bahwa banyaknya kombinasi adalah

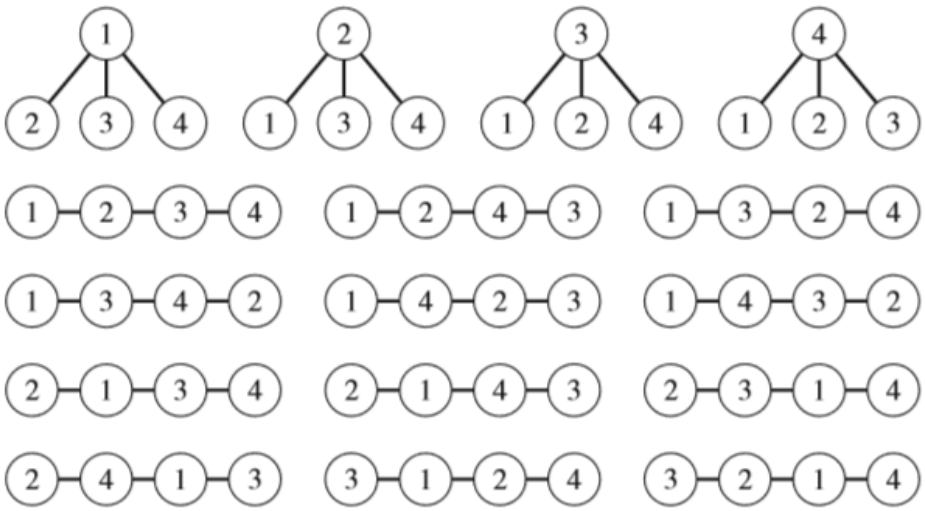
$$\frac{1}{n}\sum_{k=1}^{n} c(k)$$

di mana ada n cara untuk mengubah posisi kombinasi, dan ada kombinasi c(k) yang tetap tidak berubah ketika cara ke-k diterapkan. Sebagai contoh, mari kita hitung jumlah kalung dari n mutiara, di mana setiap mutiara memiliki m kemungkinan warna. Dua kalung dikatakan simetris jika sebangun setelah diputar. Misalnya, Gambar 11.12 menunjukkan empat kalung simetris, yang harus dihitung sebagai kombinasi tunggal.
Ada n cara untuk mengubah posisi kalung, karena dapat diputar k =0,1,...,n−1langkah searah jarum jam.Misalnya, ifk =0, semua kalung m^n tetap sama, dan jika k = 1, hanya m kalung dimana setiap mutiara memiliki warna yang sama tetap sama. Dalam kasus umum, total kalung $m^{gcd(k,n)}$ tetap sama, karena balok mutiara berukuran gcd(k, n) akan saling menggantikan. Jadi, menurut lemma Burnside, jumlah kalung yang berbeda adalah

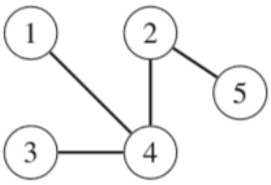
$$\frac{1}{n}\sum_{k=0}^{n-1} m^{\gcd(k,n)}$$

Misal, banyaknya kalung berbeda yang terdiri dari 4 mutiara dan 3 warna adalah

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24$$



Gambar 11.13 Ada 16 pohon berlabel berbeda dari 4 node



Gambar 11.14 Kode Prüfer dari pohon ini adalah [4,4,2]

11.2.5 Rumus Cayley

Rumus Cayley menyatakan bahwa ada total n^{n-2} pohon berlabel berbeda dari n simpul. Node diberi label 1, 2,..., n, dan dua pohon dianggap berbeda jika struktur atau pelabelannya berbeda. Misalnya, ketika n = 4, ada 4^{4-2} = 16 pohon berlabel, ditunjukkan pada Gambar 11.13. Rumus Cayley dapat dibuktikan dengan

menggunakan kode Prüfer. Kode Prüfer adalah urutan angka n−2 yang menggambarkan pohon berlabel. Kode dibangun dengan mengikuti proses yang menghilangkan n 2 daun dari pohon.

Pada setiap langkah, daun dengan label terkecil dihapus, dan label tetangga satu-satunya ditambahkan ke kode. Sebagai contoh, kode Prüfer dari pohon pada Gambar 11.14 adalah [4, 4, 2], karena telah memindahkan daun 1, 3, dan 4. Kita dapat membuat kode Prüfer untuk setiap pohon, dan yang lebih penting, pohon aslinya dapat direkonstruksi dari kode Prüfer. Oleh karena itu, jumlah pohon berlabel dari n node sama dengan n^{n-2}, jumlah kode Prüfer dengan panjang n.

11.3 Matriks

Matriks adalah konsep matematika yang sesuai dengan array dua dimensi dalam pemrograman. Sebagai contoh,

$$A = \begin{bmatrix} 6 & 13 & 7 & 4 \\ 7 & 0 & 8 & 2 \\ 9 & 5 & 4 & 18 \end{bmatrix}$$

adalah matriks berukuran 3x4, yaitu memiliki 3 baris dan 4 kolom. Notasi[i,j]mengacu pada elemen pada baris i dan kolom j dalam sebuah matriks. Misalnya, pada matriks di atas, A[2,3]=8 dan A[3,1]=9. Kasus khusus matriks adalah vektor yang merupakan matriks satu dimensi berukuran nx1. Sebagai contoh,

$$A^T = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

adalah vektor yang mengandung tiga elemen. Transpos A^T dari matriks A diperoleh ketika baris dan kolom A ditukar, yaitu, A^T[i, j]=A[j,i]:

$$A^T = \begin{bmatrix} 6 & 7 & 9 \\ 13 & 0 & 5 \\ 7 & 8 & 4 \\ 4 & 2 & 18 \end{bmatrix}$$

Suatu matriks adalah matriks persegi jika memiliki jumlah baris dan kolom yang sama. Misalnya, matriks berikut adalah matriks persegi:

$$A = \begin{bmatrix} 3 & 12 & 4 \\ 5 & 9 & 15 \\ 0 & 2 & 4 \end{bmatrix}$$

11.3.1 Operasi Matriks

Jumlah A+B matriks A dan B didefinisikan jika matriks-matriks tersebut berukuran sama. Dihasilkan trix di mana setiap elemen memiliki jumlah elemen yang sesuai di A dan B. Misalnya,

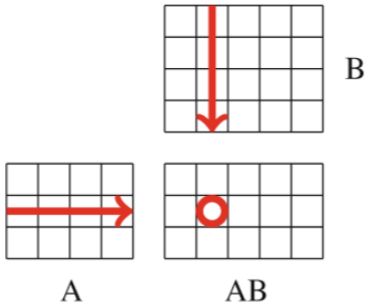
$$\begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{bmatrix} \begin{bmatrix} 10 & 10 & 7 \\ 6 & 18 & 4 \end{bmatrix}$$

Mengalikan matriks A dengan nilai x berarti setiap elemen A dikalikan dengan x. Sebagai contoh,

$$2\,x\,\begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2\,x\,6 & 2\,x\,1 & 2\,x\,4 \\ 2\,x\,3 & 2\,x\,9 & 2\,x\,2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}$$

Perkalian AB dari matriks A dan B didefinisikan jika A berukuran a×n dan B berukuran n×b, yaitu lebar A sama dengan tinggi B. Hasilnya adalah matriks berukuran a ×b yang elemen-elemennya dihitung menggunakan rumus

$$AB[i,j]| = \sum_{k=1}^{n} (A|i,k]x\ B[k,j])$$



Gambar 11.15 Intuisi dibalik rumus perkalian matriks

Idenya adalah bahwa setiap elemen AB adalah jumlah produk dari elemen A dan B sesuai dengan Gambar 11.15. Sebagai contoh,

$$\begin{bmatrix}4 & 4\\7 & 9\\5 & 6\end{bmatrix} x \begin{bmatrix}1 & 4\\2 & 9\end{bmatrix} = \begin{bmatrix}1\,x\,1+4\,x\,2 & 1\;\;x\,6+4\,x\,9\\3\,x\,1+9x2 & 3\,x\,6+9\,x\,9\\8\,x\,1+6\,x\,2 & 8\,x\,6+6\,x\,9\end{bmatrix} = \begin{bmatrix}9 & 42\\21 & 99\\20 & 102\end{bmatrix}$$

Kita dapat langsung menggunakan rumus di atas untuk menghitung hasil kali C dari dua matriks n ×n A dan B dalam waktu $O(n^3)$[13]:

```
for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
                for (int k = 1; k <= n; k++) {
                        C[i][j] += A[i][k]*B[k][j];
                }
        }
}
```

Perkalian matriks bersifat asosiatif, jadi A(BC) = (AB)C berlaku, tetapi tidak komutatif, jadi biasanya AB ≠ BA . Matriks identitas adalah matriks bujur sangkar di mana setiap elemen pada diagonalnya adalah 1 dan semua elemen lainnya adalah 0. Sebagai contoh, matriks berikut adalah matriks identitas 3x3:

$$I = \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$$

Mengalikan matriks dengan matriks identitas tidak mengubahnya. Sebagai contoh,

$$\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} x \begin{bmatrix}1 & 4\\3 & 9\\8 & 6\end{bmatrix} = \begin{bmatrix}1 & 4\\3 & 9\\8 & 6\end{bmatrix} dan \begin{bmatrix}1 & 4\\3 & 9\\8 & 6\end{bmatrix} x \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} = \begin{bmatrix}1 & 4\\3 & 9\\8 & 6\end{bmatrix}$$

Kekuatan A^k dari matriks A didefinisikan jika A adalah matriks persegi. Definisi ini didasarkan pada perkalian matriks:

$$A^k = \underbrace{A \cdot A \cdot A \cdots A}_{k \text{ times}}$$

Sebagai contoh,

$$\begin{bmatrix}2 & 5\\1 & 4\end{bmatrix}^3 = \begin{bmatrix}2 & 5\\1 & 4\end{bmatrix} x \begin{bmatrix}2 & 5\\1 & 4\end{bmatrix} x \begin{bmatrix}2 & 5\\1 & 4\end{bmatrix} = \begin{bmatrix}48 & 165\\33 & 11\,4\end{bmatrix}$$

[13] Sementara algoritma waktu lurus ke depan $O(n^3)$ salah dalam pemrograman kompetitif, secara teoritis ada algoritma yang lebih efisien. Pada tahun 1969, Strassen [31] menemukan algoritma yang pertama, sekarang disebut algoritma Strassen, yang kompleksitas waktunya adalah $O(n^{2.81})$. Algoritma terbaik saat ini, diusulkan oleh Le Gall pada tahun 2014, bekerja dalam waktu $O(n^{2.37})$.

Selain itu, A0 adalah matriks identitas. Sebagai contoh,

$$\begin{bmatrix}2 & 5\\1 & 4\end{bmatrix}^0 = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$$

Matriks A^k dapat dihitung secara efisien dalam waktu O(n3 logk) menggunakan algoritma pada Bagian 11.1.4. Sebagai contoh,

$$\begin{bmatrix}2 & 5\\1 & 4\end{bmatrix}^8 = \begin{bmatrix}2 & 5\\1 & 4\end{bmatrix}^4 x \begin{bmatrix}2 & 5\\1 & 4\end{bmatrix}^4$$

11.3.2 Perulangan Linier

Perulangan linier adalah fungsi f (n) yang nilai awalnya adalah f (0), f (1),..., f (k− 1) dan nilai yang lebih besar dihitung secara rekursif menggunakan rumus

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \cdots + c_k f(n-k),$$

di mana c_1, c_2,..., c_k adalah koefisien konstan. Pemrograman dinamis dapat digunakan untuk menghitung setiap nilai f (n) dalam waktu O(kn) dengan menghitung semua nilai f (0), f (1),..., f (n) satu demi satu. Namun, seperti yang akan kita lihat selanjutnya, kita juga dapat menghitung nilai f (n) dalam waktu $O(k^3 \log n)$ menggunakan operasi matriks. Ini merupakan peningkatan penting jika k kecil dan n besar.

Bilangan Fibonacci

Contoh sederhana dari perulangan linier adalah fungsi berikut yang mendefinisikan bilangan Fibonacci:

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2)$$

Dalam hal ini, k =2 dan $c_1 = c_2 = 1$.

Untuk menghitung angka Fibonacci secara efisien, kami menyatakan rumus Fibonacci sebagai matriks persegi X berukuran 2×2, yang berlaku sebagai berikut:

$$X\, x \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

Jadi, nilai f (i) dan f (i +1) diberikan sebagai "input" untuk X, dan X menghitung nilai f (i +1) dan f (i +2) dari keduanya. Ternyata matriks seperti itu adalah

$$X = \begin{bmatrix}1 & 1\\0 & 1\end{bmatrix}$$

Sebagai contoh,

$$\begin{bmatrix}1 & 1\\0 & 1\end{bmatrix} x \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix}1 & 1\\0 & 1\end{bmatrix} x \begin{bmatrix}5\\8\end{bmatrix} = \begin{bmatrix}8\\13\end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}$$

Dengan demikian, kita dapat menghitung f (n) menggunakan rumus

$$\begin{bmatrix} f(n) \\ f(n) \end{bmatrix} = X^n\, x \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix}0 & 1\\1 & 1\end{bmatrix}^n x \begin{bmatrix}0\\1\end{bmatrix}$$

Nilai X^n dapat dihitung dalam waktu O(log n), sehingga nilai f (n) juga dapat dihitung dalam waktu O(log n).

Kasus Umum

Mari kita perhatikan kasus umum di mana f (n) adalah sembarang perulangan linier. Sekali lagi, tujuan kami adalah untuk membangun matriks X yang

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}$$

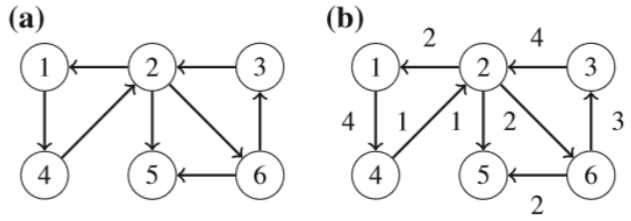
Matriks seperti itu adalah

$$X = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & \cdots & c_1 \end{bmatrix}.$$

Pada k 1 baris pertama, setiap elemen adalah 0 kecuali bahwa satu elemen adalah 1. Baris-baris ini menggantikan f (i) dengan f (i +1), f (i +1) dengan f (i +2), dan seterusnya pada. Kemudian, baris terakhir berisi koefisien perulangan untuk menghitung nilai baru f (i +k).

$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

Sekarang, f (n) dapat dihitung dalam waktu O(k3 log n) menggunakan rumus



Gambar 11.16 Contoh grafik untuk operasi matriks

11.3.3 Matriks dan Grafik

Kekuatan matriks kedekatan grafik memiliki sifat yang menarik. Ketika M adalah matriks ketetanggaan dari graf tak berbobot, matriks M^n memberikan untuk setiap pasangan simpul (a,b) jumlah lintasan yang dimulai pada simpul a, berakhir pada simpul b, dan mengandung tepat n sisi. Diperbolehkan bahwa sebuah node muncul di jalur beberapa kali. Sebagai contoh, perhatikan grafik pada Gambar 11.16a. Matriks ketetanggaan dari graf ini adalah

$$M = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Maka matriks

$$M^4 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

memberikan jumlah jalur yang berisi tepat 4 tepi. Misalnya, $M^4[2,5]=2$, karena ada dua jalur 4 sisi dari node 2 ke node 5: 2→1→4→2→5 dan 2→6→3→2→5. Dengan menggunakan ide yang sama dalam graf berbobot, kita dapat menghitung untuk setiap pasangan simpul (a,b) panjang terpendek dari suatu lintasan dari a ke b dan mengandung tepat n sisi. Untuk menghitung ini, kami mendefinisikan perkalian

matriks dengan cara baru, sehingga kami tidak menghitung jumlah jalur tetapi meminimalkan panjang jalur. Sebagai contoh, perhatikan grafik pada Gambar 11.16b. Mari kita buat matriks ketetanggaan di mana berarti bahwa suatu sisi tidak ada, dan nilai-nilai lain sesuai dengan bobot sisi. matriksnya adalah

$$M = \begin{bmatrix} \infty & \infty & \infty & 4 & \infty & \infty \\ 2 & \infty & \infty & \infty & 1 & 2 \\ \infty & 4 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & \infty & 2 & \infty \end{bmatrix}$$

Alih-alih rumus

$$AB[i,j] = \sum_{k=1}^{n} (A[i,k]\ x\ B[k,j])$$

kita sekarang menggunakan rumus

$$AB[i,j] = \min_{k=1}^{n)} A[i,k]\ +\ B[k,j])$$

untuk perkalian matriks, jadi kami menghitung minima bukan jumlah, dan jumlah elemen bukan produk. Setelah modifikasi ini, kekuatan matriks meminimalkan panjang jalur dalam grafik. Misalnya, sebagai

$$M^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix}$$

kita dapat menyimpulkan bahwa panjang minimum dari sebuah lintasan dengan 4 sisi dari simpul 2 ke simpul 5 adalah 8. Lintasan tersebut adalah 2→1→4→2→5.

11.3.4 Eliminasi Gauss

Eliminasi Gauss adalah cara sistematis untuk menyelesaikan sekelompok persamaan linier. Idenya adalah untuk mewakili persamaan sebagai matriks dan kemudian menerapkan urutan operasi baris matriks sederhana yang menyimpan informasi persamaan dan menentukan nilai untuk setiap variabel.

Misalkan kita diberikan sekelompok n persamaan linier, yang masing-masing berisi n variabel:

$$a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1$$
$$a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = b_2$$
$$\ldots$$
$$a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n = b_n$$

Kami mewakili persamaan sebagai matriks sebagai berikut:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & b_n \end{bmatrix}$$

Untuk menyelesaikan persamaan, kita ingin mengubah matriks menjadi

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & c_1 \\ 0 & 1 & \cdots & 0 & c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & c_n \end{bmatrix}$$

yang menyatakan bahwa penyelesaiannya adalah x1 =c_1,x_2 =c_2,...,x_n =c_n. Untuk melakukan ini, kami menggunakan tiga jenis operasi baris matriks:

1. Tukar nilai dari dua baris.
2. Kalikan setiap nilai dalam satu baris dengan konstanta nonnegatif.
3. Tambahkan baris, dikalikan dengan konstanta, ke baris lain.

Setiap operasi di atas menyimpan informasi persamaan, yang menjamin bahwa solusi akhir sesuai dengan persamaan awal. Kita dapat memproses setiap kolom matriks secara sistematis sehingga algoritma yang dihasilkan bekerja dalam waktu $O(n^3)$. Sebagai contoh, perhatikan kelompok persamaan berikut:

$$2x_1 + 4x_2 + x_3 = 16$$
$$x_1 + 2x_2 + 5x_3 = 17$$
$$3\,x_1 + x_2 + x_3 = 8$$

Dalam hal ini matriksnya adalah sebagai berikut:

$$\begin{bmatrix} 2 & 4 & 1 & 16 \\ 1 & 2 & 5 & 17 \\ 3 & 1 & 1 & 8 \end{bmatrix}$$

Kami memproses kolom matriks dengan kolom. Pada setiap langkah, kami memastikan bahwa kolom saat ini memiliki satu di posisi yang benar dan semua nilai lainnya adalah nol. Untuk memproses kolom pertama, pertama kalikan baris pertama dengan 1/2:

$$\begin{bmatrix} 1 & 2 & \frac{1}{2} & 8 \\ 1 & 2 & 5 & 17 \\ 3 & 1 & 1 & 8 \end{bmatrix}$$

Kemudian kita tambahkan baris pertama ke baris kedua (dikalikan dengan 1) dan baris pertama ke baris ketiga (dikalikan dengan −3):

$$\begin{bmatrix} 1 & 2 & \frac{1}{2} & 8 \\ 0 & 0 & \frac{9}{2} & 9 \\ 0 & -5 & -\frac{1}{2} & -16 \end{bmatrix}$$

Setelah ini, kami memproses kolom kedua. Karena nilai kedua pada baris kedua adalah nol, pertama-tama kita menukar baris kedua dan ketiga:

$$\begin{bmatrix} 1 & 2 & \frac{1}{2} & 8 \\ 0 & -5 & -\frac{1}{2} & -16 \\ 0 & 0 & \frac{9}{2} & 9 \end{bmatrix}$$

Kemudian kita kalikan baris kedua dengan $-\frac{1}{5}$ dan tambahkan ke baris pertama (dikalikan dengan 2):

$$\begin{bmatrix} 1 & 2 & \frac{3}{10} & \frac{8}{5} \\ 0 & 1 & \frac{1}{10} & \frac{16}{5} \\ 0 & 0 & \frac{9}{2} & 9 \end{bmatrix}$$

Akhirnya, kami memproses kolom ketiga dengan terlebih dahulu mengalikannya dengan $\frac{2}{9}$ dan kemudian menambahkannya ke baris pertama (dikalikan dengan $-\frac{3}{10}$) dan ke baris kedua (dikalikan dengan $-\frac{1}{10}$):

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 2 \end{bmatrix}$$

Sekarang kolom terakhir dari matriks memberitahu kita bahwa solusi untuk kelompok persamaan asli adalah $x_1 = 1$, $x_2 = 3$, $x_3 = 2$. Perhatikan bahwa eliminasi Gaussian hanya berfungsi jika kelompok persamaan memiliki solusi unik. Misalnya grup

$$x_1 + x_2 = 2$$
$$2\,x_1 + 2x_2 = 4$$

memiliki jumlah solusi tak berhingga, karena kedua persamaan mengandung informasi yang sama. Di sisi lain, kelompok

$$x_1 + x_2 = 5$$
$$x_1 + x_2 = 7$$

tidak dapat diselesaikan, karena persamaannya kontradiktif. Jika tidak ada solusi unik, kami akan melihat ini selama algoritme, karena pada titik tertentu kami tidak akan berhasil memproses kolom.

11.4 Probabilitas

Probabilitas adalah bilangan real antara 0 dan 1 yang menunjukkan seberapa besar kemungkinan suatu kejadian. Jika suatu peristiwa pasti terjadi, peluangnya adalah 1, dan jika suatu peristiwa tidak mungkin terjadi, peluangnya adalah 0. Peluang suatu peristiwa dilambangkan dengan P(⋯) di mana tiga titik menggambarkan peristiwa tersebut. Misalnya, ketika melempar dadu, ada enam kemungkinan hasil 1,2,...,6, dan P("hasilnya genap")=1/2. Untuk menghitung probabilitas suatu peristiwa, kita dapat menggunakan kombinatorik atau mensimulasikan proses yang menghasilkan peristiwa tersebut. Sebagai contoh, pertimbangkan sebuah eksperimen di mana kita mengambil tiga kartu teratas dari tumpukan kartu yang dikocok[14] Berapa probabilitas bahwa setiap kartu memiliki nilai yang sama (misalnya, ♠8, ♣8, dan ♦8)? Salah satu cara untuk menghitung peluang adalah dengan menggunakan rumus

$$\frac{jumlah\ hasil\ yang\ diinginkan}{Jumlah\ total}$$

Dalam contoh kita, hasil yang diinginkan adalah yang nilai setiap kartunya sama. Ada $13\binom{4}{3}$ hasil seperti itu, karena ada 13 kemungkinan nilai kartu dan $\binom{4}{3}$ cara untuk memilih 3 suit dari

[14] Setumpuk kartu terdiri dari 52 kartu. Setiap kartu memiliki suit (sekop ♠, berlian♦, klub♣, atau hati) dan nilai (bilangan bulat antara 1 dan 13).

4 kemungkinan suit. Lalu, ada total $\binom{52}{3}$ hasil, karena kita memilih 3 kartu dari 52 kartu. Jadi, peluang kejadian tersebut adalah .

$$\frac{13\binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}$$

Cara lain untuk menghitung probabilitas adalah dengan mensimulasikan proses yang menghasilkan kejadian tersebut. Dalam contoh kita, kita menggambar tiga kartu, sehingga prosesnya terdiri dari tiga langkah. Kami mensyaratkan bahwa setiap langkah proses berhasil. Pengambilan kartu pertama pasti berhasil, karena kartu apa pun boleh. Langkah kedua berhasil dengan probabilitas 3/51, karena tersisa 51 kartu dan 3 di antaranya bernilai sama dengan kartu pertama. Dengan cara yang sama, langkah ketiga berhasil dengan probabilitas 2/50. Jadi, peluang bahwa seluruh proses berhasil adalah

$$1\ x\ \frac{3}{51}\ x\ \frac{2}{50} = \frac{1}{425}$$

11.4.1 Bekerja dengan Acara (Events)

Cara mudah untuk merepresentasikan event adalah dengan menggunakan set. Misalnya, hasil yang mungkin terjadi saat melempar dadu adalah {1, 2, 3, 4, 5, 6}, dan setiap himpunan bagian dari himpunan ini adalah kejadian. Acara "hasilnya genap" sesuai dengan himpunan{2, 4, 6}. Setiap hasil *x* diberi probabilitas *p(x),* dan probabilitas *P(X)* dari suatu kejadian *X* dapat dihitung dengan menggunakan rumus

$$P(X) = \sum_{x \in X} p(x)$$

Misalnya, ketika sebuah dadu dilempar, *p(x)* =1/6 untuk setiap hasil *x*, maka peluang kejadian “hasilnya genap” adalah

p(2)+ p(4)+ p(6) =1/2.

Karena kejadian direpresentasikan sebagai himpunan, kita dapat memanipulasinya menggunakan operasi himpunan standar:

- Komplemen A berarti “A tidak terjadi”. Misalnya, ketika melempar sebuah dadu, komplemen dari A ={ 2, 4, 6}adalah A ={ 1, 3, 5}.
- Persatuan A∪B berarti "A atau B terjadi." Misalnya, gabungan dari A ={ 2,5}dan B ={ 4,5,6}adalah A∪ B ={ 2,4,5,6}.
- Perpotongan A∩ B berarti “A dan B terjadi”. Misalnya, perpotongan A ={ 2,5}dan B ={ 4,5,6}adalah A∩ B ={ 5}.

Komplemen Probabilitas A dihitung dengan menggunakan rumus

P(A)=1-P(A)

Terkadang, kita dapat menyelesaikan masalah dengan mudah menggunakan pelengkap dengan menyelesaikan masalah yang berlawanan. Misalnya, peluang mendapatkan setidaknya satu enam saat melempar dadu sepuluh kali adalah

$1-(5/6)^{10}$

Di sini 5/6 adalah probabilitas bahwa hasil satu lemparan bukan enam, dan $(5/6)^{10}$ adalah probabilitas bahwa tidak satu pun dari sepuluh lemparan adalah enam. Pelengkap ini adalah jawaban dari masalah.

Union

Probabilitas A∪ B dihitung menggunakan rumus

P(A ∪ B) = P(A)+ P(B)− P(A∩ B).

Sebagai contoh, perhatikan kejadian A = “hasilnya genap” dan B = “hasilnya kurang dari 4” ketika melempar sebuah dadu. Dalam hal ini, kejadian A∪B berarti “hasilnya genap atau kurang dari 4”, dan peluangnya adalah

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) = 1/2 + 1/2 - 1/6 = 5/6.$$

Jika kejadian A dan B saling lepas, yaitu A∩B kosong, maka peluang kejadian A B adalah $P(A \cup B) = P(A) + P(B)$.

Persimpangan

Probabilitas A∩ B dapat dihitung dengan menggunakan rumus

$$P(A \cap B) = P(A)P(B|A),$$

di mana P(B|A) adalah probabilitas bersyarat bahwa B terjadi dengan asumsi bahwa kita tahu bahwa A terjadi. Misalnya, dengan menggunakan kejadian dari contoh kita sebelumnya, P(B|A) = 1/3, karena kita tahu bahwa hasilnya termasuk dalam himpunan{2,4,6}, dan salah satu hasilnya kurang dari 4. Dengan demikian,

$$P(A \cap B) = P(A)P(B|A) = 1/2 \cdot 1/3 = 1/6.$$

Kejadian A dan B saling bebas jika

$$P(A|B) = P(A) \text{ and } P(B|A) = P(B),$$

yang berarti fakta bahwa B terjadi tidak mengubah probabilitas A, dan sebaliknya. Dalam hal ini, peluang terjadinya perpotongan adalah

$$P(A \cap B) = P(A)P(B).$$

11.4.2 Variabel Acak

Variabel acak adalah nilai yang dihasilkan oleh proses acak. Misalnya, ketika melempar dua dadu, variabel acak yang mungkin adalah

$$X = \text{“jumlah hasil”}.$$

Sebagai contoh, jika hasilnya adalah[4,6](artinya pertama kita melempar empat dan kemudian enam), maka nilai X adalah 10. Kami menyatakan dengan P(X = x) probabilitas bahwa nilai a variabel acak Xis x. Misalnya, ketika melempar dua dadu, P(X = 10) = 3/36, karena jumlah total hasil adalah 36 dan ada tiga cara yang mungkin untuk mendapatkan jumlah 10: [4,6],[5,5] , dan[6,4].

Gambar 11.17 Kemungkinan cara untuk menempatkan dua bola dalam empat kotak

Nilai yang Diharapkan

Nilai yang diharapkan E[X]menunjukkan nilai rata-rata dari variabel acak X. Nilai yang diharapkan dapat dihitung sebagai jumlah

$$\sum_{x} P(X) = x)x$$

di mana x melewati semua kemungkinan nilai X. Misalnya, ketika melempar dadu, hasil yang diharapkan adalah

$$1/6 \times 1 + 1/6 \times 2 + 1/6 \times 3 + 1/6 \times 4 + 1/6 \times 5 + 1/6 \times 6 = 7/2.$$

Properti yang berguna dari nilai yang diharapkan adalah linearitas. Artinya jumlah $E[X_1 + X_2 + \cdots + X_n]$selalu sama dengan jumlah $E[X_1]+E[X_2]+\cdots+E[X_n]$. Ini berlaku bahkan jika variabel acak bergantung satu sama lain. Misalnya, ketika melempar dua dadu, jumlah yang diharapkan dari nilai-nilai mereka adalah

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

Mari kita pertimbangkan masalah di mana bola secara acak ditempatkan di kotak, dan tugas kita adalah menghitung jumlah kotak kosong yang diharapkan. Setiap bola memiliki peluang yang sama untuk ditempatkan di salah satu kotak. Misalnya, Gambar 11.17 menunjukkan kemungkinan ketika n = 2. Dalam hal ini, jumlah kotak kosong yang diharapkan adalah

$$\frac{0+0+1+1}{4} = \frac{1}{2}$$

Kemudian, dalam kasus umum, probabilitas bahwa satu kotak kosong adalah

$$\left(\frac{n-1}{n}\right)^n$$

karena tidak ada bola yang harus ditempatkan di dalamnya. Oleh karena itu, dengan menggunakan linearitas, jumlah kotak kosong yang diharapkan adalah

$$n\,x\,\left(\frac{n-1}{n}\right)^n$$

Distribusi

Distribusi variabel acak X menunjukkan probabilitas setiap nilai yang mungkin dimiliki X. Distribusi terdiri dari nilai-nilai P(X = x). Misalnya, ketika melempar dua dadu, distribusi jumlah mereka adalah:

x	2	3	4	5	6	7	8	9	10	11	12
P(X=x)	1/36	2/36	3/36	4/36	5/36	6/36	5/36	4/36	3/36	2/36	1/36

Dalam distribusi seragam, variabel acak X memiliki n nilai yang mungkin a,a +1 ,...,b dan probabilitas setiap nilai adalah 1/n. Misalnya, ketika melempar dadu, a =1, b =6, dan P(X = x) =1/6 untuk setiap nilai x. Nilai yang diharapkan dari X dalam distribusi seragam adalah

$$E[X] = \frac{a+b}{2}$$

Dalam distribusi binomial, n upaya dilakukan dan probabilitas bahwa satu upaya berhasil adalah p. Variabel acak X menghitung jumlah upaya yang berhasil, dan probabilitas nilai x adalah

$$P(X = x) = p^x(1-p)^{n-x}\binom{n}{x}$$

di mana p^x dan $(1-p)^{n-x}$ sesuai dengan upaya yang berhasil dan tidak berhasil, dan $\binom{n}{x}$ adalah jumlah cara kita dapat memilih urutan upaya. Misalnya, ketika melempar dadu sepuluh kali, peluang melempar enam tepat tiga kali adalah $(1/6)^3\,(5/6)^7\,10^3$. Nilai harapan dari X dalam distribusi binomial adalah

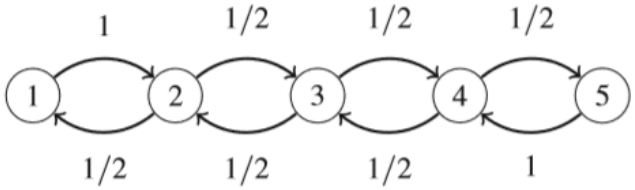
$$E[X]=pn.$$

Dalam distribusi geometrik, probabilitas bahwa suatu upaya berhasil adalah p, dan kami melanjutkan sampai keberhasilan pertama terjadi. Variabel acak X menghitung jumlah upaya yang diperlukan, dan probabilitas nilai x adalah

$$P(X = x) = (1-p)^{x-1}p,$$

di mana $(1-p)^{x-1}$ sesuai dengan upaya yang gagal dan p sesuai dengan upaya pertama yang berhasil. Misalnya, jika kita melakukan lemparan sampai kita mendapatkan enam, peluang banyaknya lemparan tepat 4 adalah $(5/6)^3$ 1/6. Nilai yang diharapkan dari X dalam distribusi geometrik adalah

$$E[X] = \frac{1}{p}$$



Gambar 11.18 Rantai Markov untuk bangunan yang terdiri dari lima lantai

11.4.3 Rantai Markov

Rantai Markov adalah proses acak yang terdiri dari keadaan dan transisi di antara keadaan tersebut. Untuk setiap keadaan, kita mengetahui kemungkinan perpindahan ke keadaan lain. Rantai Markov dapat direpresentasikan sebagai graf yang simpulnya sesuai dengan keadaan dan tepinya menggambarkan transisi. Sebagai contoh, pertimbangkan masalah di mana memakai di lantai 1 di lantai dan gedung. Pada setiap langkah, kami secara acak berjalan satu lantai ke atas atau satu lantai ke bawah, kecuali bahwa kami selalu berjalan satu lantai dari lantai 1 dan satu lantai turun dari lantai n. Berapa peluang berada di lantai m setelah k langkah? Dalam masalah ini, setiap lantai bangunan sesuai dengan keadaan dalam rantai Markov. Misalnya, Gambar 11.18 menunjukkan rantai ketika n =5. Distribusi probabilitas dari rantai Markov adalah vektor[p_1, p_2, ..., p_n], di mana p_k adalah probabilitas bahwa keadaan saat ini adalah k. Rumus $p_1 + p_2 + \cdots + p_n = 1$ selalu berlaku.

Dalam skenario di atas, distribusi awalnya adalah [1,0,0,0,0], karena kita selalu memulai di lantai 1. Distribusi berikutnya adalah [0,1,0,0,0], karena kita hanya bisa bergerak dari floor 1 to floor 2. Setelah ini, kita dapat memindahkan satuflooruporonefloordown, sehingga distribusi berikutnya adalah [1/2,0,1/2,0,0], dan seterusnya. Cara yang efisien untuk mensimulasikan perjalanan dalam rantai Markov adalah dengan menggunakan pemrograman dinamis. Idenya adalah untuk menjaga distribusi probabilitas, dan pada setiap langkah melalui semua kemungkinan bagaimana kita bisa bergerak. Dengan menggunakan metode ini, kita dapat mensimulasikan perjalanan m langkah dalam waktu $O(n^2m)$. Transisi rantai Markov juga dapat direpresentasikan sebagai matriks sama yang memperbarui distribusi probabilitas. Dalam skenario di atas, matriksnya adalah

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix}$$

Ketika kita mengalikan distribusi probabilitas dengan matriks ini, kita mendapatkan distribusi baru setelah bergerak satu langkah. Sebagai contoh, kita dapat berpindah dari distribusi [1,0,0,0,0]ke distribusi[0,1,0,0,0]sebagai berikut:

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Dengan menghitung daya matriks secara efisien, kita dapat menghitung distribusi setelah langkah-langkah dalam waktu $O(n^3 \log m)$.

11.4.4 Algoritma Acak

Kadang-kadang kita dapat menggunakan keacakan untuk memecahkan masalah, bahkan jika masalahnya tidak terkait dengan probabilitas. Algoritma acak adalah algoritma yang didasarkan pada keacakan. Ada dua jenis algoritma acak yang populer:

- *Algoritma Monte Carlo* adalah algoritma yang terkadang memberikan jawaban yang salah. Agar algoritma seperti itu berguna, kemungkinan jawaban yang salah harus kecil.
- *Algoritma Las Vegas* adalah algoritma yang selalu memberikan jawaban yang benar, tetapi waktu berjalannya bervariasi secara acak. Tujuannya adalah untuk merancang algoritma yang efisien dengan probabilitas tinggi.

Selanjutnya kita akan membahas tiga contoh masalah yang dapat diselesaikan dengan menggunakan algoritma tersebut.

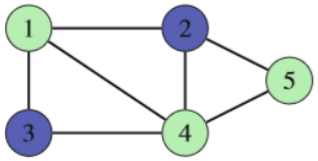
Statistik Pesanan

Statistik orde ke-k dari suatu array adalah elemen pada posisi k setelah mengurutkan array dalam urutan yang meningkat. Sangat mudah untuk menghitung statistik urutan apa pun dalam waktu O(n log n) dengan terlebih dahulu mengurutkan array, tetapi apakah benar-benar diperlukan untuk mengurutkan seluruh array hanya untuk menemukan satu elemen? Ternyata kita dapat menemukan statistik pesanan menggunakan algoritma Las Vegas, yang waktu berjalan yang diharapkan adalah O(n). Algoritme memilih elemen acak x dari array dan memindahkan elemen yang lebih kecil dari x ke bagian kiri array, dan semua elemen lainnya ke bagian kanan array. Ini membutuhkan waktu O(n) ketika ada n elemen.

Asumsikan bahwa bagian kiri berisi elemen dan bagian kanan berisi elemen. Ifa =k,elemen x adalah statistik orde kek. Jika tidak, jika > k, secara rekursif cari statistik orde ke-k untuk bagian kiri, dan jika a < k, kita cari statistik orde ketiga secara rekursif untuk bagian kanan di mana r = k −a −1. Pencarian berlanjut dengan cara yang sama, sampai elemen yang diinginkan telah ditemukan. Ketika setiap elemen x dipilih secara acak, ukuran array menjadi setengahnya pada setiap langkah, sehingga kompleksitas waktu untuk menemukan statistik orde ke-k adalah sekitar

$$n+n/2+n/4+n/8+\cdots=O(n).$$

Perhatikan bahwa kasus terburuk dari algoritma membutuhkan waktu $O(n^2)$, karena kemungkinan x selalu dipilih sedemikian rupa sehingga merupakan salah satu dari elemen terkecil atau terbesar dalam larik dan langkah O(n) diperlukan. Namun, kemungkinannya sangat kecil sehingga kita dapat berasumsi bahwa ini tidak pernah terjadi dalam praktik.



Gambar 11.19 Pewarnaan graf yang valid

Memverifikasi Perkalian Matriks

Diberikan matriks A, B, dan C, masing-masing berukuran n × n, masalah kita selanjutnya adalah memverifikasi apakah AB=C memenuhi. Tentu saja, kita dapat memecahkan

masalah hanya dengan menghitung waktu produk ABin $O(n^3)$, tetapi kita dapat berharap bahwa memverifikasi jawabannya akan lebih mudah daripada menghitungnya dari awal. Ternyata kita dapat menyelesaikan masalah tersebut dengan menggunakan algoritma Monte Carlo yang kompleksitas waktunya hanya $O(n^2)$.Idenya sederhana:kita memilih vektordominan X dari n elemen dan menghitung matriks ABX dan CX. Jika ABX=CX, kami melaporkan bahwa AB=C, dan sebaliknya kami melaporkan bahwa AB ≠ C.

Kompleksitas waktu dari algoritma ini adalah $O(n^2)$, karena kita dapat menghitung matriks ABX dan CX dalam waktu $O(n^2)$. Kita dapat menghitung matriks ABX secara efisien dengan menggunakan representasi A(BX), jadi hanya dua perkalian n×n dan n×1 matriks ukuran diperlukan. Kelemahan dari algoritma ini adalah kecilnya kemungkinan algoritma membuat kesalahan ketika melaporkan bahwa AB=C. Sebagai contoh,

$$\begin{bmatrix}6 & 8\\ 1 & 3\end{bmatrix} \neq \begin{bmatrix}8 & 7\\ 3 & 2\end{bmatrix}$$

tetapi

$$\begin{bmatrix}6 & 8\\ 1 & 3\end{bmatrix}\begin{bmatrix}3\\ 6\end{bmatrix} = \begin{bmatrix}8 & 7\\ 3 & 2\end{bmatrix}\begin{bmatrix}3\\ 6\end{bmatrix}$$

Namun, dalam praktiknya, probabilitas bahwa algoritme membuat kesalahan kecil, dan kita dapat mengurangi probabilitas dengan memverifikasi hasilnya menggunakan beberapa vektor acak X sebelum melaporkan bahwa AB=C.

Pewarnaan Grafik

Diberikan sebuah graf yang berisi n simpul dan m sisi, masalah terakhir kita adalah menemukan cara untuk mewarnai simpul menggunakan dua warna sehingga untuk paling sedikit m/2 sisi, titik akhir memiliki warna yang berbeda. Misalnya, Gambar 11.19 menunjukkan pewarnaan graf yang valid. Dalam hal ini graf berisi tujuh sisi, dan titik-titik ujung dari lima sisi tersebut memiliki warna yang berbeda dalam pewarnaannya. Masalah tersebut dapat diselesaikan dengan menggunakan algoritma Las Vegas yang menghasilkan pewarnaan acak sampai pewarnaan yang valid ditemukan. Dalam pewarnaan acak, warna setiap simpul dipilih secara bebas sehingga peluang kedua warna adalah 1/2. Oleh karena itu, jumlah sisi yang diharapkan yang titik ujungnya memiliki warna berbeda adalah m/2. Karena pewarnaan acak diharapkan valid, kita akan segera menemukan pewarnaan yang valid dalam praktik.

11.5 Teori Permainan

Di bagian ini, kami fokus pada permainan dua pemain di mana para pemain bergerak secara bergantian dan memiliki set gerakan yang sama, dan tidak ada elemen acak. Tujuan kami adalah menemukan strategi yang dapat kami ikuti untuk memenangkan permainan tidak peduli apa yang dilakukan lawan, jika strategi seperti itu ada. Ternyata ada strategi umum untuk permainan seperti itu, dan kami dapat menganalisis permainan menggunakan teori nim. Pertama, kami akan menganalisis game sederhana di mana pemain mengeluarkan tongkat dari tumpukan, dan setelah ini, kami akan menggeneralisasi strategi yang digunakan dalam game tersebut ke game lain.

11.5.1 Definisi Permainan

Mari kita pertimbangkan permainan yang dimulai dengan setumpuk n tongkat. Dua pemain bergerak secara bergantian, dan pada setiap gerakan, pemain harus mengeluarkan 1, 2, atau 3 batang dari tumpukan. Akhirnya, pemain yang melepaskan tongkat terakhir memenangkan permainan. Misalnya, jika n = 10, permainan dapat dilanjutkan sebagai berikut:

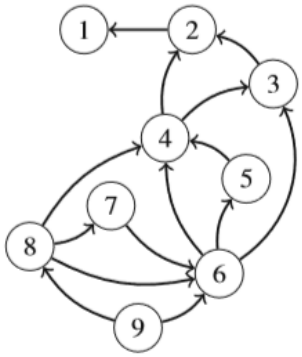
- Pemain A mengeluarkan 2 tongkat (8 tongkat tersisa).
- Pemain B mengeluarkan 3 tongkat (5 tongkat tersisa).
- Pemain A mengeluarkan 1 tongkat (4 tongkat tersisa).
- Pemain B mengeluarkan 2 tongkat (2 tongkat tersisa).
- Pemain A menghapus 2 tongkat dan menang.

Game ini terdiri dari state 0,1,2,...,n, dimana jumlah state sesuai dengan jumlah stick yang tersisa. Keadaan menang adalah keadaan dimana pemain akan memenangkan permainan jika mereka bermain secara optimal, dan keadaan kalah adalah keadaan dimana pemain akan kalah dalam permainan jika lawan bermain secara optimal. Ternyata kita bisa mengelompokkan semua state dari sebuah game sehingga setiap state adalah state yang menang atau state yang kalah. Pada game di atas, state0 jelas-jelas kehilangan state, karena pemain tidak dapat melakukan gerakan apapun.

Negara bagian 1, 2, dan 3 adalah status pemenang, karena pemain dapat menghapus 1, 2, atau 3 tongkat dan memenangkan permainan. Negara 4, pada gilirannya, adalah negara yang kalah, karena setiap langkah mengarah ke negara yang merupakan negara pemenang bagi lawan. Lebih umum, jika ada pergerakan yang mengarah dari keadaan saat ini ke keadaan kalah, itu adalah keadaan menang, dan jika tidak, keadaan kalah. Dengan menggunakan pengamatan ini, kita dapat mengklasifikasikan semua keadaan permainan dimulai dengan keadaan kalah di mana tidak ada kemungkinan perpindahan. Gambar 11.20 menunjukkan klasifikasi keadaan 0...15 (W menunjukkan keadaan menang dan L menunjukkan keadaan kalah).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	*W*	*W*	*W*	*L*	*W*	*W*	*W*	*L*	*W*	*W*	*W*	*L*	*W*	*W*	*W*

Gambar 11.20 Klasifikasi keadaan 0...15 dalam permainan tongkat



Gambar 11.21 Grafik keadaan dari permainan pembagian

1	2	3	4	5	6	7	8	9
L	*W*	*L*	*W*	*L*	*W*	*L*	*W*	*L*

Gambar 11.22 Klasifikasi keadaan 1.9 dalam permainan pembagian

Sangat mudah untuk menganalisis permainan ini: keadaan k adalah keadaan kalah jika k habis dibagi 4, dan sebaliknya adalah keadaan menang. Cara optimal untuk memainkan permainan adalah selalu memilih langkah setelah jumlah tongkat di tumpukan habis dibagi 4. Akhirnya, tidak ada tongkat yang tersisa dan lawannya kalah. Tentu saja, strategi ini mengharuskan jumlah tongkat tidak habis dibagi 4 saat kita bergerak. Jika ya, tidak ada yang optimal.

Mari kita pertimbangkan permainan tongkat lain, di mana di setiap keadaan k, diperbolehkan untuk menghapus sejumlah x tongkat sedemikian rupa sehingga x lebih kecil dari k dan membagi k. Sebagai contoh, dalam keadaan 8 kita dapat menghapus 1, 2, atau 4 batang, tetapi dalam keadaan 7 satu-satunya gerakan yang diperbolehkan adalah memindahkan 1 batang. Gambar 11.21 menunjukkan keadaan1...9 dari grafik keadaan permainan, yang simpulnya adalah keadaan dan tepinya adalah pergerakan di antara keduanya: Keadaan akhir dalam permainan ini adalah selalu keadaan 1 yang kehilangan state, karena tidak ada langkah yang valid. Gambar 11.22 menunjukkan klasifikasi state1...9.Ternyata dalam permainan ini, semua state bernomor genap adalah state yang menang, dan state yang bernomor genap adalah state yang kalah.

11.5.2 Permainan Nim

Permainan nim merupakan permainan sederhana yang memiliki peranan penting dalam teori permainan, karena banyak permainan lain yang dapat dimainkan dengan menggunakan strategi yang sama. Pertama, kami fokus pada nim, dan setelah ini, kami menggeneralisasi strategi ke game lain. Ada n tumpukan di nim, dan setiap tumpukan berisi sejumlah tongkat. Para pemain bergerak secara bergantian, dan setiap belokan, pemain memilih sebuah tumpukan yang masih berisi tongkat dan mengeluarkan sejumlah tongkat darinya. Pemenangnya adalah pemain yang melepaskan tongkat terakhir.

Keadaan dalam nim berbentuk[x_1, x_2, ..., x_n], di mana xi menyatakan jumlah batang di tumpukan i. Misalnya, [10.12,5] adalah keadaan di mana ada tiga tumpukan dengan 10, 12, dan 5 batang. Keadaan [0,0,...,0] adalah keadaan kalah, karena tidak mungkin menghilangkan tongkat apapun, dan ini selalu merupakan keadaan akhir.

Analisis

Ternyata kita dapat dengan mudah mengklasifikasikan keadaan nim apa pun dengan menghitung nim sum s = x1 x2 $\cdots\bigoplus x_n$, di mana $\bigoplus$ menunjukkan operasi xor. Negara bagian yang nim sumnya 0 adalah negara bagian yang kalah, dan semua negara bagian lainnya adalah negara bagian yang menang. Misalnya, nim sum dari[10,12,5]adalah $10\bigoplus 12\bigoplus 5=3$, jadi negara bagian tersebut adalah negara bagian yang menang. Tapi bagaimana nim sum terkait dengan game nim? Kita dapat menjelaskan ini dengan melihat bagaimana nim sum berubah ketika keadaan nim berubah.

Keadaan yang hilang: Keadaan akhir[0, 0,..., 0]keadaan hilang, dan nimsumis0, seperti yang diharapkan. Di negara bagian lain yang kalah, setiap langkah mengarah ke keadaan menang, karena ketika satu nilai xi berubah, nim sum juga berubah, sehingga nim sum berbeda dari 0 setelah bergerak.

Status menang: Kita dapat pindah ke status kalah jika ada tumpukan i yang xi s < xi. Dalam hal ini, kita dapat menghapus tongkat dari tumpukan i sehingga akan berisi tongkat xi s, yang akan menyebabkan keadaan kalah. Selalu ada tumpukan seperti itu, di mana xi memiliki satu bit pada posisi paling kiri satu bit s.

Contoh

Sebagai contoh, perhatikan keadaan[10. 12, 5].Keadaan ini merupakan keadaan menang, karena nim sum-nya adalah 3. Dengan demikian, harus ada langkah yang mengarah pada keadaan yang kalah. Selanjutnya kita akan menemukan langkah seperti itu. Jumlah negara bagian adalah sebagai berikut:

10	1010
12	1100
5	0101

3	0011

Dalam hal ini, heap dengan 10 stick adalah satu-satunya heap yang memiliki satu bit pada posisi paling kiri satu bit dari nim sum:

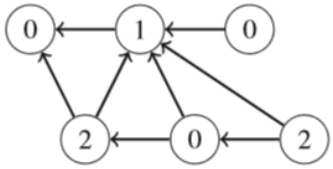
10	1010
12	1100
5	0101
3	0011

Ukuran heap yang baru harus 10⊕3=9, jadi kami akan menghapus satu batang saja. Setelah ini, negara akan menjadi [9,12,5], yang merupakan negara yang kalah:

9	1010
12	1100
5	0101
0	0000

Permainan Misre

Dalam permainan misère nim, tujuan permainannya berlawanan, jadi pemain yang melepaskan tongkat terakhir akan kalah dalam permainan. Ternyata game misère nim bisa dimainkan secara optimal hampir seperti game nim standar.



Gambar 11.23 Nomor Grundy dari status permainan

Idenya adalah pertama-tama mainkan game misère seperti game standar, tetapi ubah strategi di akhir game. Strategi baru akan diperkenalkan dalam situasi di mana setiap tumpukan akan berisi tongkat batu setelah gerakan berikutnya. Dalam permainan standar, kita harus memilih langkah yang setelah itu ada jumlah tumpukan genap dengan satu tongkat. Namun, dalam permainan misère, kami memilih langkah sehingga ada tumpukan ganjil dengan satu tongkat. Strategi ini bekerja karena keadaan dimana perubahan strategi selalu muncul dalam permainan, dan keadaan ini adalah keadaan menang, karena mengandung tepat satu tumpukan yang memiliki lebih dari satu tongkat sehingga nim sum tidak 0.

11.5.3 Dalil *Sprague–Grundy*

Teorema Sprague-Grundy menggeneralisasi strategi yang digunakan dalam nim untuk semua game yang memenuhi persyaratan berikut:

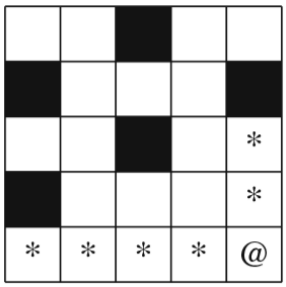
- Ada dua pemain yang bergerak bergantian.
- Gim ini terdiri dari status, dan kemungkinan gerakan dalam status tidak bergantung pada giliran siapa.
- Permainan berakhir ketika seorang pemain tidak dapat bergerak.
- Permainan pasti akan berakhir cepat atau lambat.
- Para pemain memiliki informasi lengkap tentang status dan gerakan yang diizinkan, dan tidak ada keacakan dalam permainan.

Nomor Grundy

Idenya adalah untuk menghitung untuk setiap permainan menyatakan nomor Grundy yang sesuai dengan jumlah tongkat di tumpukan nim. Ketika kita mengetahui angka Grundy dari semua negara bagian, kita bisa memainkan game seperti game nim. Jumlah Grundy dari status permainan dihitung menggunakan rumus

$$\text{mex}(\{g_1,g_2,...,g_n\}),$$

di mana g_1, g_2,..., g_n adalah bilangan Grundy dari keadaan dimana kita dapat berpindah dari keadaan tersebut, dan fungsi mex memberikan bilangan nonnegatif terkecil yang tidak ada dalam himpunan. Misalnya, mex({0,1,3}) =2. Jika suatu keadaan tidak memiliki kemungkinan bergerak, bilangan Grundy-nya adalah 0, karena mex(∅) =0. Misalnya, Gambar 11.23menunjukkan grafik status permainan di mana setiap negara bagian diberi angka Grundy. Status angka Grundy yang kalah adalah 0, dan angka Grundy dari negara yang menang adalah angka positif.



Gambar 11.24 Kemungkinan gerakan pada belokan pertama

0	1	■	0	1
■	0	1	2	■
0	2	■	1	0
■	3	0	4	1
0	4	1	3	2

Gambar 11.25 Nomor Grundy dari status permainan

Pertimbangkan keadaan yang bilangan Grundynya adalah x. Kita dapat berpikir bahwa itu sesuai dengan tumpukan nim yang memiliki x tongkat. Khususnya, jika x > 0, kita dapat pindah ke keadaan yang bilangan Grundy-nya 0,1,...,x 1, yang mensimulasikan pemindahan tongkat dari tumpukan nim. Ada satu perbedaan, meskipun dimungkinkan untuk pindah ke keadaan yang nomor Grundy lebih besar dari x dan "tambah" menempel pada tumpukan. Namun, lawan selalu dapat membatalkan langkah seperti itu, jadi ini tidak mengubah strategi.

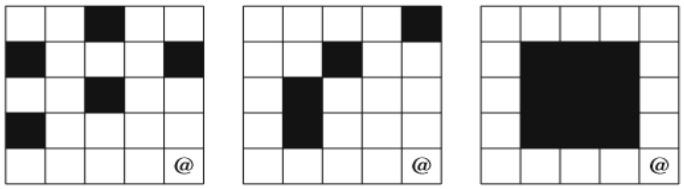
Sebagai contoh, pertimbangkan permainan di mana para pemain memindahkan sebuah nama nama. Setiap kotak labirin adalah lantai atau dinding. Pada setiap giliran, pemain harus menggerakkan gambar beberapa langkah ke kiri atau ke atas. Pemenang permainan adalah pemain yang melakukan gerakan terakhir. Gambar 11.24 menunjukkan kemungkinan konfigurasi awal permainan, di mana @ menunjukkan angka dan * menunjukkan kotak tempat ia dapat bergerak. Status permainan adalah semua petak lantai labirin. Gambar 11.25 menunjukkan nomor Grundy dari negara bagian dalam konfigurasi ini.

Menurut teorema Sprague–Grundy, setiap keadaan dari permainan labirin berhubungan dengan banyak permainan kemudian.Misalnya,bilangan Grundy dari kotak kanan bawah adalah 2, jadi ini adalah keadaan menang. Kita bisa mencapai keadaan kalah dan memenangkan permainan dengan menggerakkan empat langkah ke kiri atau dua langkah ke atas.

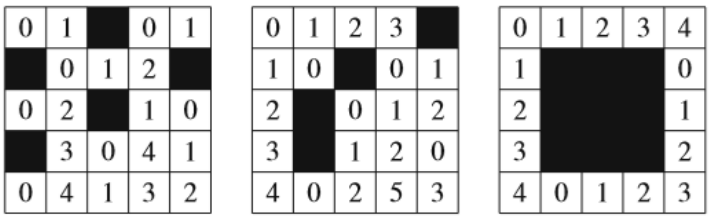
Subgame

Asumsikan bahwa permainan kita terdiri dari subpermainan, dan pada setiap giliran, pemain pertama-tama memilih subpermainan dan kemudian bergerak dalam subpermainan tersebut. Permainan berakhir ketika tidak mungkin untuk melakukan gerakan apa pun di sub-permainan apa pun. Dalam hal ini, angka Grundy dari sebuah game sama dengan jumlah nim dari angka Grundy dari subgame. Game ini kemudian dapat dimainkan seperti game nim dengan menghitung semua angka Grundy untuk subgame dan kemudian nim sum mereka.

Sebagai contoh, pertimbangkan permainan yang terdiri dari tiga labirin. Pada setiap giliran, pemain memilih salah satu labirin dan kemudian memindahkan sosok di labirin tersebut. Gambar 11.26 menunjukkan konfigurasi awal permainan, dan Gambar 11.27 menunjukkan nomor Grundy yang sesuai. Dalam konfigurasi ini, jumlah nim dari bilangan Grundy adalah 2⊕ 3⊕3 = 2, sehingga pemain pertama yang dapat memenangkan permainan. Satu langkah optimal adalah naik dua langkah di labirin pertama, yang menghasilkan nim sum 0⊕3⊕3=0.



Gambar 11.26 Game yang terdiri dari tiga subgame



Gambar 11.27 Angka grundy di subgame

Game Grundy

Terkadang gerakan dalam game membagi game menjadi subgame yang independen satu sama lain. Dalam hal ini, nomor Grundy dari status permainan adalah

$$\text{mex}(\{g_1, g_2, ..., g_n\}),$$

di mana ada n kemungkinan gerakan dan

$$g_k = a_{k,1} \oplus a_{k,2} \oplus ... \oplus a_{k,m},$$

artinya gerakan k membagi permainan menjadi m subgame yang bilangan Grundynya adalah $a_{k,1}$, $a_{k,2}$, ..., $a_{k,m}$. Contoh game semacam itu adalah game Grundy. Awalnya, ada satu tumpukan yang memiliki n batang. Pada setiap giliran, pemain memilih tumpukan dan membaginya menjadi dua tumpukan kosong sehingga tumpukan memiliki ukuran yang berbeda. Pemain yang membuat langkah terakhir memenangkan permainan.

Misalkan g(n) menyatakan bilangan Grundy dari tumpukan berukuran n. Nomor Grundy dapat dihitung dengan melalui semua cara untuk membagi tumpukan menjadi dua tumpukan. Misalnya, ketika n =8, kemungkinannya adalah 1+7, 2+6, dan 3+5, jadi

$$g(8) = \text{mex}(\{g(1) \oplus g(7), g(2) \oplus g(6), g(3) \oplus g(5)\}).$$

Dalam permainan ini, nilai g(n) didasarkan pada nilai g(1),...,g(n−1). Kasus dasarnya adalah g(1) = g(2) = 0, karena tidak mungkin membagi tumpukan 1 dan 2 batang menjadi tumpukan yang lebih kecil. Angka Grundy pertama adalah:

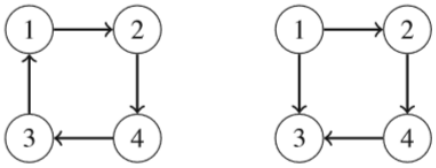
g(1) = 0
g(2) = 0
g(3) = 1
g(4) = 0
g(5) = 2
g(6) = 1
g(7) = 0
g(8) = 2

Angka Grundy untuk n = 8 adalah 2, jadi dimungkinkan untuk memenangkan permainan. Langkah yang menang adalah membuat tumpukan 1+7, karena $g(1) \oplus g(7) = 0$.

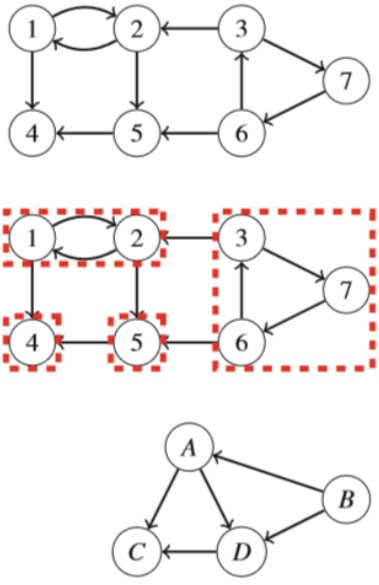
BAB 12
ALGORITMA GRAFIK TINGKAT LANJUT

12.1 Konektivitas yang Kuat

Grafik berarah disebut terhubung kuat jika terdapat lintasan dari sembarang simpul ke semua simpul lain dalam graf tersebut. Sebagai contoh, graf kiri pada Gambar 12.1 terhubung kuat sedangkan grafik kanan tidak. Grafik kanan tidak terhubung kuat, karena, misalnya, tidak ada jalur dari node 2 ke node 1. Graf berarah selalu dapat dibagi menjadi komponen terhubung kuat. Setiap komponen tersebut berisi satu set maksimal node sedemikian rupa sehingga ada jalur dari setiap node ke semua node lainnya, dan komponen membentuk grafik komponen asiklik yang mewakili struktur dalam dari grafik asli. Misalnya, Gambar.12.2 menunjukkan grafik, komponen yang terhubung kuat dan grafik komponen yang sesuai. Komponen-komponen tersebut adalah A ={ 1,2}, B ={ 3,6,7}, C ={ 4}, dan D ={ 5}.



Gambar 12.1 Grafik kiri terhubung kuat, grafik kanan tidak



Gambar 12.2 Sebuah graf, komponen-komponennya yang terhubung kuat dan graf komponennya

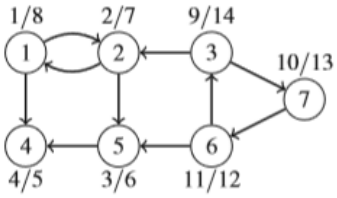
Grafik komponen merupakan graf asiklik berarah, sehingga lebih mudah diproses dibandingkan graf aslinya. Karena grafik tidak mengandung siklus, kita selalu dapat membuat semacam topologi dan menggunakan pemrograman dinamis untuk memprosesnya.

12.1.1 Algoritma Kosaraju

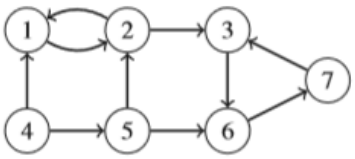
Algoritma Kosaraju adalah metode yang efisien untuk menemukan komponen graf yang terhubung kuat. Algoritme melakukan dua pencarian mendalam-pertama: pencarian pertama membangun daftar node sesuai dengan struktur grafik, dan pencarian kedua membentuk komponen yang terhubung kuat. Fase pertama dari algoritma Kosaraju membangun daftar node dalam urutan di mana pencarian mendalam pertama memprosesnya. Algoritme menelusuri node dan memulai

pencarian depth-first pada setiap node yang diproses. Setiap node akan ditambahkan ke daftar setelah diproses.

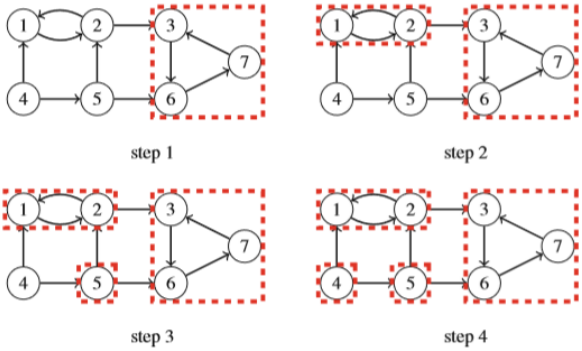
Sebagai contoh, Gambar 12.3 menunjukkan urutan pemrosesan node dalam grafik contoh kita. Notasi x/y berarti pemrosesan kemudian dimulai pada waktu x dan selesai pada waktu y. Daftar yang dihasilkan adalah[4, 5, 2, 1, 6, 7, 3] Fase kedua dari algoritma Kosaraju membentuk komponen yang terhubung kuat. Pertama, algoritma membalikkan setiap tepi grafik. Ini menjamin bahwa selama pencarian kedua, kita akan selalu menemukan komponen yang terhubung kuat dan valid. Gambar 12.4 menunjukkan grafik dalam contoh kita setelah membalikkan tepi.



Gambar 12.3 Urutan pemrosesan node



Gambar 12.4 Graf dengan sisi terbalik



Gambar 12.5 Membangun komponen yang terhubung kuat

Setelah ini, algoritma akan menelusuri daftar node yang dibuat oleh pencarian pertama, dalam urutan terbalik. Jika sebuah node tidak termasuk dalam komponen, algoritma akan membuat komponen baru dengan memulai pencarian depth-first yang menambahkan semua node baru yang ditemukan selama pencarian ke komponen baru. Perhatikan bahwa karena semua buku dibalik, komponen tidak "bocor" ke bagian lain dari grafik.

Gambar 12.5 menunjukkan bagaimana algoritma memproses grafik contoh kita. Urutan pemrosesan node adalah [3,7,6,1,2,5,4]. Pertama, node 3 menghasilkan komponen {3,6,7}. Kemudian, node 7 dan 6 dilewati, karena sudah menjadi bagian dari sebuah komponen. Setelah ini, node 1 menghasilkan komponen {1,2}, dan node 2 dilewati. Terakhir, node 5 dan 4 menghasilkan komponen{5}dan{4}. Kompleksitas waktu dari algoritma mis O(n+m), karena algoritma melakukan dua pencarian kedalaman-pertama.

12.1.2 Masalah 2SAT

Dalam masalah 2SAT, kami diberikan rumus logis

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m),$$

di mana setiap ai dan bi adalah variabel logika (x_1, x_2, ..., x_n) atau negasi dari variabel logika ($\neg x_1$, x_2, ..., x_n). Simbol "∧" dan "∨" menunjukkan operator logika "dan" dan "atau." Tugas kita adalah memberi nilai pada setiap variabel sehingga rumusnya benar, atau menyatakan bahwa ini tidak mungkin. Misalnya rumus

$$L1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

benar ketika variabel ditetapkan sebagai berikut:

$$\begin{cases} x_1 = salah \\ x_2 = salah \\ x_3 = salah \\ x_4 = salah \end{cases}$$

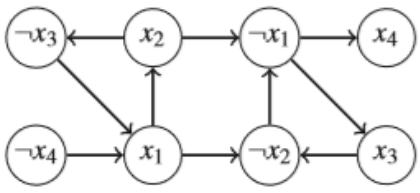
Namun, rumus

$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

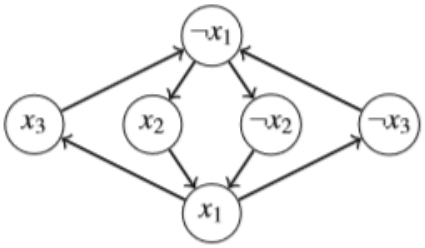
selalu salah, terlepas dari bagaimana kita menetapkan nilai. Alasan untuk ini adalah bahwa kita tidak dapat memilih nilai untuk x_1 tanpa menciptakan kontradiksi. Jika x_1 salah, maka x_2 dan x_2 harus benar yang tidak mungkin, dan jika x_1 benar, baik x_3 dan $\neg x_3$ harus benar yang juga tidak mungkin. Sebuah instance dari masalah 2SAT dapat direpresentasikan sebagai grafik implikasi yang nodenya sesuai dengan variabel x_i dan negasi $\neg x_i$, dan edge menentukan hubungan antar variabel. Setiap pasangan ($a_i \vee b_i$) menghasilkan dua sisi: $\neg a_i \rightarrow b_i$ dan $\neg b_i \rightarrow a_i$.

Artinya jika ai tidak tahan, bi harus tahan, dan sebaliknya. Sebagai contoh, Gambar.12.6 menunjukkan grafik implikasi dari L_1, dan Gambar.12.7 menunjukkan grafik implikasi dari L2. Struktur grafik implikasi memberi tahu kita apakah mungkin untuk menetapkan nilai variabel sehingga rumusnya benar. Hal ini dapat dilakukan dengan tepat ketika tidak ada simpul xi dan xi sedemikian rupa sehingga kedua simpul tersebut termasuk dalam komponen terhubung kuat yang sama.

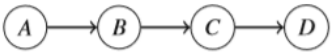
Jika terdapat simpul seperti itu, graf tersebut berisi lintasan dari x_i ke $\neg x_i$ dan juga lintasan dari x_i ke x_i, jadi x_i dan ¬xi harus benar yang tidak mungkin. Sebagai contoh, graf implikasi dari L_1 tidak memiliki simpul x_i dan $\neg x_i$ sehingga kedua simpul tersebut merupakan bagian dari komponen yang terhubung kuat, sehingga terdapat solusi. Kemudian, pada grafik implikasi L_2 semua node termasuk dalam komponen terhubung kuat yang sama, sehingga tidak ada solusi.



Gambar 12.6 Grafik implikasi dari L_1



Gambar 12.7 Grafik implikasi dari L_2



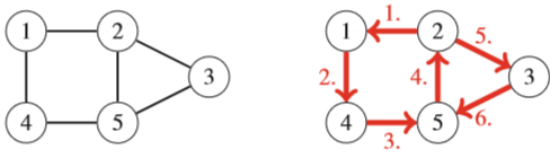
Gambar 12.8 Grafik komponen L_1

Jika ada solusi, nilai untuk variabel dapat ditemukan dengan menelusuri simpul dari grafik komponen dalam urutan topologi terbalik. Pada setiap langkah, kami memproses komponen yang tidak mengandung tepi yang mengarah ke komponen yang tidak diproses. Jika variabel dalam komponen belum diberi nilai, nilainya akan ditentukan sesuai dengan nilai dalam komponen, dan jika sudah memiliki nilai, nilainya tetap tidak berubah. Proses berlanjut sampai setiap variabel telah diberi nilai.

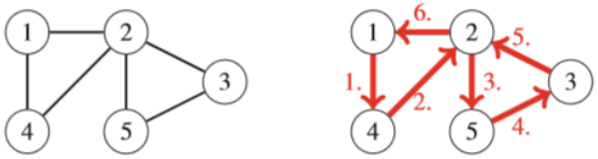
Gambar 12.8 menunjukkan grafik komponen L_1. Komponen-komponen tersebut adalah A ={¬ x4},B ={ x_1,x_2,¬x_3}, C = {¬ x_1,¬x_2,x_3}, dan D ={ x_4}. Saat membangun solusi, pertama-tama kita memproses komponen D di mana x4 menjadi benar. Setelah ini, kami memproses komponen C di mana x_1 dan x_2 menjadi salah dan x_3 menjadi benar. Semua variabel telah diberi nilai, sehingga komponen A dan B yang tersisa tidak mengubah nilai variabel. Perhatikan bahwa metode ini bekerja, karena grafik aplikasi memiliki struktur khusus: jika ada jalur dari simpul xi ke simpul xj dan dari simpul x_j ke simpul x_j, maka simpul xi tidak pernah menjadi benar. Alasan untuk ini adalah bahwa ada juga jalur dari node¬x_j ke node¬x_i, dan baik x_i maupun xj menjadi salah. Masalah yang lebih sulit adalah masalah 3SAT, di mana setiap bagian dari rumus berbentuk (a_i V b_i V c_i). Masalah ini adalah NP-hard, jadi tidak ada algoritma yang efisien untuk memecahkan masalah yang diketahui.

12.2 Jalur Lengkap

Pada bagian ini kita membahas dua jenis lintasan khusus dalam graf: lintasan Euler adalah lintasan yang melalui setiap sisi tepat satu kali, dan lintasan Hamilton adalah lintasan yang mengunjungi setiap simpul tepat satu kali. Walaupun lintasan tersebut pada pandangan pertama terlihat cukup mirip, masalah komputasi yang terkait dengannya sangat berbeda.



Gambar 12.9 Graf dan lintasan Euler



Gambar 12.10 Graf dan sirkuit Euler

12.2.1 Jalur Eulerian

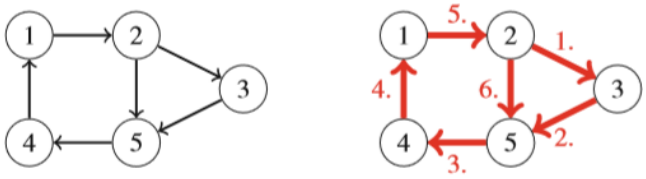
Lintasan Euler adalah lintasan yang melalui setiap sisi graf tepat satu kali. Selanjutnya, jika jalur seperti itu dimulai dan berakhir pada simpul yang sama, itu disebut sirkuit Euler. Gambar 12.9 menunjukkan jalur Euler dari node 2 ke node 5, dan Gambar 12.10 menunjukkan sirkuit Euler yang dimulai dan berakhir pada node 1. Keberadaan jalur dan sirkuit Euler tergantung pada derajat node. Pertama, graf tak berarah memiliki lintasan Euler dengan tepat jika semua sisinya termasuk dalam komponen terhubung yang sama dan

- derajat setiap simpul genap, atau
- derajat tepat dua simpul adalah ganjil, dan derajat semua simpul lainnya genap.

Dalam kasus pertama, setiap jalur Euler juga merupakan sirkuit Euler. Dalam kasus kedua, simpul derajat ganjil adalah titik akhir dari jalur Euler, yang bukan merupakan sirkuit Euler. Pada Gambar 12.9, simpul 1, 3, dan 4 berderajat 2, dan simpul 2 dan 5 berderajat 3. Tepatnya dua simpul memiliki derajat ganjil, jadi ada jalur Euler antara simpul 2 dan 5, tetapi graf tidak memiliki sirkuit Euler. Pada Gambar 12.10, semua simpul berderajat genap, sehingga graf tersebut memiliki sirkuit Euler. Untuk menentukan apakah graf berarah memiliki jalur Euler, kita fokus pada derajat masuk dan derajat keluar dari simpul. Graf berarah berisi jalur Euler dengan tepat jika semua sisinya termasuk dalam komponen terhubung kuat yang sama dan

- di setiap simpul, derajat masuk sama dengan derajat keluar, atau
- di satu simpul, derajat masuk satu lebih besar dari derajat keluar, di simpul lain, derajat keluar satu lebih besar dari derajat masuk, dan di semua simpul lainnya, derajat masuk sama dengan derajat keluar.

Dalam kasus pertama, setiap jalur Euler juga merupakan sirkuit Euler, dan dalam kasus kedua, grafhasan jalur Euler yang dimulai pada titik yang derajat keluarnya lebih besar dan berakhir pada simpul yang derajat masuknya lebih besar. Sebagai contoh, pada Gambar 12.11, simpul 1, 3, dan 4 memiliki derajat masuk 1 dan derajat keluar 1, simpul 2 memiliki derajat masuk 1 dan derajat keluar 2, dan simpul 5 memiliki derajat 2 dan derajat keluar 1. Oleh karena itu, graf tersebut berisi jalur Euler dari simpul 2 ke simpul 5.



Gambar 12.11 Graf berarah dan lintasan Euler

Konstruksi

Algoritma Hierholzer adalah metode yang efisien untuk membangun sirkuit Euler untuk graf. Algoritma terdiri dari beberapa putaran, yang masing-masing menambahkan tepi baru ke sirkuit. Tentu saja, kita berasumsi bahwa grafik tersebut berisi sirkuit Euler; jika tidak, algoritma Hierholzer tidak dapat menemukannya. Algoritme dimulai dengan sirkuit kosong yang hanya berisi satu node dan kemudian melanjutkan sirkuit langkah demi langkah dengan menambahkan sub sirkuit ke dalamnya. Proses berlanjut hingga semua tepi ditambahkan ke sirkuit. Sirkuit diperpanjang dengan mencari simpul x yang termasuk dalam rangkaian tetapi getat keluar yang tidak termasuk dalam rangkaian. Kemudian, jalur baru dari simpul x yang

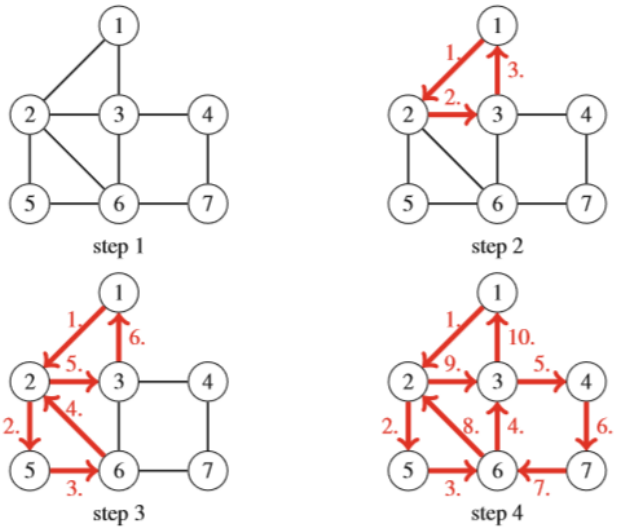
hanya berisi tepi yang tidak ada dalam rangkaian dibangun. Cepat atau lambat, jalur akan kembali ke simpul x, yang membuat subsirkuit.

Jika suatu graf tidak memiliki sirkuit Euler tetapi memiliki lintasan Euler, kita masih dapat menggunakan algoritma Hierholzer untuk mencari lintasan dengan menambahkan sisi tambahan pada graf dan menghilangkan sisi tersebut setelah rangkaian dibuat. Misalnya, dalam graf tak berarah, kita menambahkan sisi ekstra di antara dua simpul berderajat ganjil. Sebagai contoh, Gambar 12.12 menunjukkan bagaimana algoritma Hierholzer membangun sirkuit Euler dalam graf tak berarah. Pertama, algoritme menambahkan subsirkuit 1→2→3→1, kemudian subsirkuit 2→5→6→2, dan akhirnya subsirkuit 6→3→4→7→6. sirkuit.

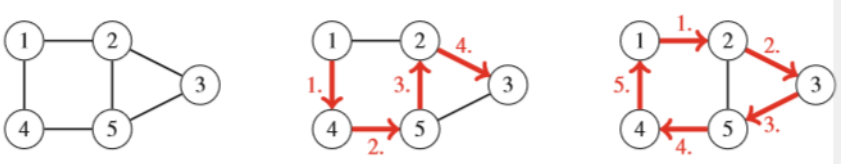
12.2.2 Jalur Hamilton

Lintasan Hamilton adalah lintasan yang mengunjungi setiap simpul graf tepat satu kali. Selanjutnya, jika jalur seperti itu dimulai dan berakhir pada simpul yang sama, itu disebut sirkuit Hamilton. Misalnya, Gambar 12.13 menunjukkan grafik yang memiliki jalur Hamilton dan sirkuit Hamilton. Masalah yang terkait dengan jalur Hamilton adalah NP-hard: tidak ada yang tahu cara umum untuk memeriksa secara efisien apakah suatu graf memiliki jalur atau sirkuit Hamilton. Tentu saja, dalam beberapa kasus khusus kita dapat yakin bahwa suatu graf berisi lintasan Hamilton. Misalnya, jika grafnya lengkap, yaitu ada sisi di antara semua pasangan simpul, pasti mengandung jalur Hamiltonian.

Cara sederhana untuk mencari jalur Hamiltonian adalah dengan menggunakan algoritma *backtracking* yang melewati semua cara yang mungkin untuk membangun jalur. Kompleksitas waktu dari algoritma tersebut setidaknya O(n!), karena ada n! cara yang berbeda untuk memilih urutan n node. Kemudian, dengan menggunakan pemrograman dinamis, kita dapat membuat solusi waktu $O(2^n n^2)$ yang lebih efisien, yang menentukan untuk setiap subset node S dan setiap node x ∈ S jika ada jalur yang mengunjungi semua node S tepat satu kali dan berakhir pada simpul x.



Gambar 12.12 Algoritma Hierholzer



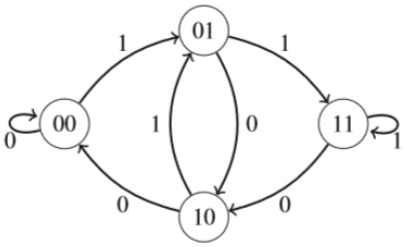
Gambar 12.13 Graf, lintasan Hamilton, dan sirkuit Hamilton

12.2.3 Aplikasi

Barisan De Bruijn Barisan De Bruijn adalah string yang berisi setiap string dengan panjang n tepat satu kali sebagai substring, untuk alfabet tetap dengan k karakter. Panjang string tersebut adalah k^n +n−1 karakter. Misalnya, ketika n = 3 dan k = 2, contoh deret De Bruijn adalah

0001011100.

Substring dari string ini adalah kombinasi dari tiga bit: 000, 001, 010, 011, 100, 101, 110, dan 111.



Gambar 12.14 Membangun barisan De Bruijn dari lintasan Euler

1	4	11	16	25
12	17	2	5	10
3	20	7	24	15
18	13	22	9	6
21	8	19	14	23

Gambar 12.15 Tur ksatria terbuka di papan 5x5

Barisan De Bruijn selalu berkorespondensi dengan lintasan Euler dalam graf di mana setiap simpul berisi string n 1 karakter, dan setiap sisi menambahkan satu karakter ke string. Misalnya, graf pada Gambar 12.14 sesuai dengan skenario di mana n =3 dan k =2. Untuk membuat barisan De Bruijn, kita mulai dari simpul sembarang dan mengikuti jalur Euler yang mengunjungi setiap tepi tepat satu kali. Ketika karakter di simpul awal dan di tepi dibaca bersama-sama, menghasilkan string yang memiliki karakter skn+n−1 dan merupakan barisan De Bruijn yang valid.

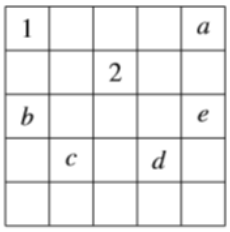
Tur Ksatria

Tur ksatria adalah urutan gerakan seorang ksatria di papan catur n × n mengikuti aturan catur sehingga ksatria mengunjungi setiap kotak tepat satu kali. Tur malam disebut tertutup jika ksatria akhirnya kembali ke kotak awal dan sebaliknya disebut terbuka. Misalnya, Gambar 12.15 menunjukkan tur ksatria terbuka di papan 5x5.

Tur seorang ksatria sesuai dengan jalur Hamiltonian dalam grafik yang simpulnya mewakili kuadrat papan, dan dua simpul terhubung dengan tepi jika seorang ksatria dapat bergerak di antara kotak sesuai dengan aturan catur lainnya. Cara alami untuk membangun backtracking wisata malam hari. Karena ada banyak kemungkinan gerakan, pencarian dapat dibuat lebih efisien dengan menggunakan heuristik yang mencoba untuk membimbing ksatria sehingga tur lengkap akan ditemukan dengan cepat.

Aturan Warnsdorf adalah heuristik sederhana dan efektif untuk menemukan tur ksatria. Menggunakan aturan itu, memungkinkan untuk secara efisien membangun tur

bahkan pada papan besar. Idenya adalah untuk selalu menggerakkan ksatria sehingga berakhir di sebuah kotak di mana jumlah gerakan lanjutan yang mungkin mungkin sekecil mungkin. Misalnya, pada Gambar.12.16, ada lima kotak yang mungkin di mana ksatria dapat bergerak (persegi ... e). Dalam situasi ini, Warnsdorf bergerak. ksatria ke kotak a, karena setelah pilihan ini, hanya ada satu kemungkinan langkah. Pilihan lain akan memindahkan ksatria ke kotak di mana akan ada tiga gerakan yang tersedia.



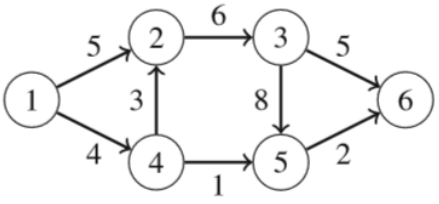
Gambar 12.16 Menggunakan aturan Warndorf untuk membangun tur ksatria

12.3 Arus Maksimum

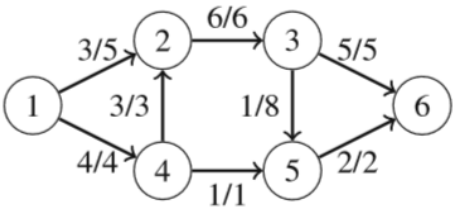
Dalam masalah aliran maksimum, kita diberikan graf berbobot berarah yang berisi dua simpul khusus: sumber adalah simpul tanpa tepi masuk, dan sink adalah simpul tanpa tepi keluar. Tugas kita adalah mengirim banyak kemungkinan dari sumber ke sink. Setiap sisi memiliki kapasitas yang membatasi aliran yang dapat melalui sisi tersebut, dan di setiap node perantara, aliran masuk dan keluar harus seimbang.

Sebagai contoh, perhatikan grafik pada Gambar 12.17, di mana node 1 adalah sumber dan node 6 adalah sink. Aliran maksimum dalam grafik ini adalah 7, ditunjukkan pada Gambar.12.18. Notasi v/k berarti bahwa aliran v unit diarahkan melalui tepi yang kapasitasnya k unit. Ukuran aliran adalah 7, karena sumber mengirimkan 3+4 unit aliran dan bak cuci menerima 5+2 unit aliran. Sangat mudah untuk melihat bahwa aliran ini maksimum, karena kapasitas total tepi yang menuju ke bak cuci adalah 7.

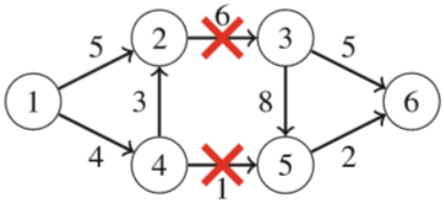
Ternyata masalah aliran maksimum salah terhubung ke masalah graf lain, masalah pemotongan minimum, di mana penyimpanan tugas kita memindahkan satu set tepi dari graf sedemikian rupa sehingga tidak akan ada jalur dari sumber ke bak cuci setelah pemindahan dan berat total tepi yang dilepas adalah minimum. Sebagai contoh, perhatikan kembali grafik pada Gambar 12.17. Ukuran potongan minimum adalah 7, karena cukup untuk menghilangkan tepi 2 → 3 dan 4 → 5, seperti ditunjukkan pada Gambar 12.19. Setelah ujung-ujungnya dilepas, akan ada jalur dari sumber ke bak cuci. Ukuran potongan adalah 6+1=7, dan potongannya minimum, karena tidak ada potongan yang sah yang bobotnya kurang dari 7.



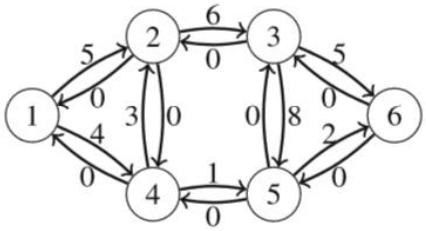
Gambar 12.17 Grafik dengan sumber 1 dan sink 6



Gambar 12.18 Aliran maksimum grafik adalah 7



Gambar 12.19 Potongan minimum grafik adalah 7



Gambar 12.20 Representasi grafik dalam algoritma Ford–Fulkerson

Bukan suatu kebetulan bahwa aliran maksimum dan pemotongan minimum sama dalam contoh grafik kita. Sebaliknya, ternyata mereka selalu sama, jadi konsepnya adalah dua sisi dari mata uang yang sama. Selanjutnya kita akan membahas algoritma Ford-Fulkerson yang dapat digunakan untuk mencari aliran maksimum dan potongan minimum dari suatu graf. Algoritme juga membantu kita memahami mengapa mereka sama.

12.1.1 Algoritma Ford–Fulkerson

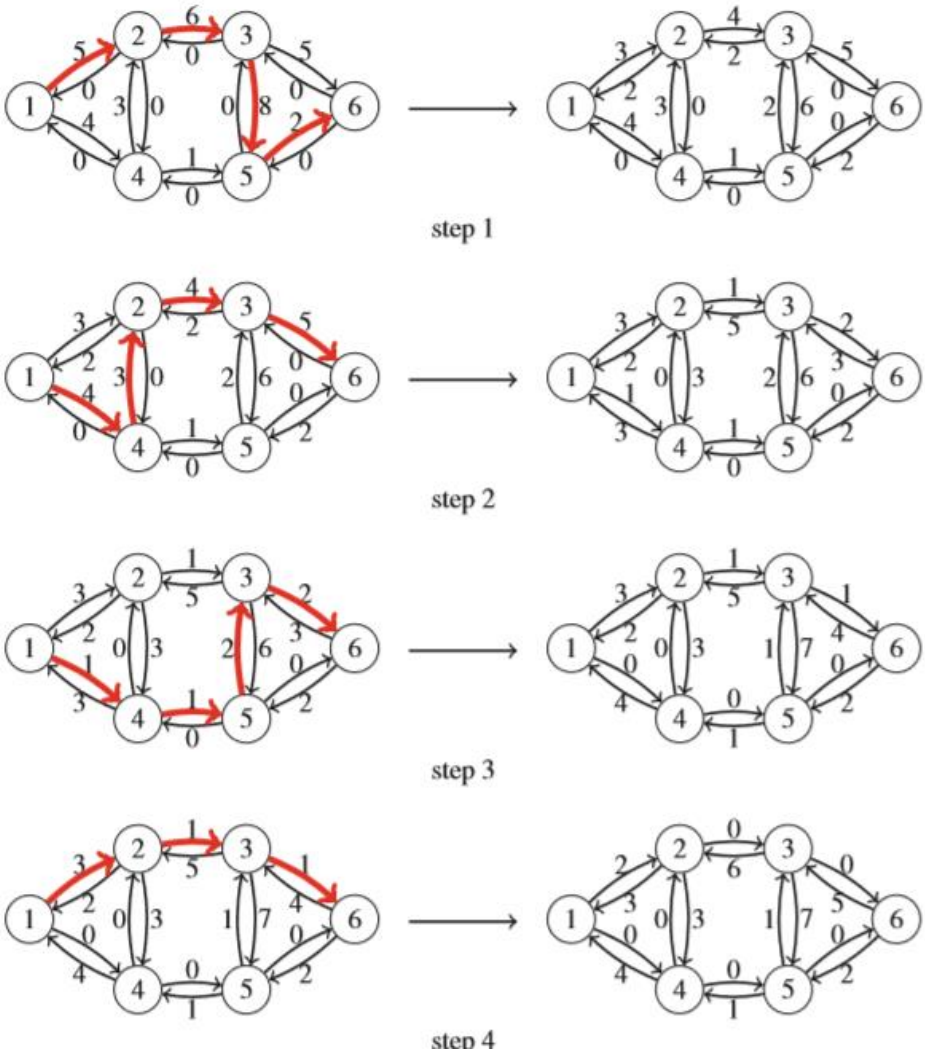
Algoritma Ford-Fulkerson menemukan aliran maksimum dalam grafik. Algoritme dimulai dengan aliran kosong, dan pada setiap langkah menemukan jalur dari sumber ke sink yang menghasilkan lebih banyak aliran. Akhirnya, ketika algoritma tidak dapat meningkatkan aliran lagi, aliran maksimum telah ditemukan.

Algoritma menggunakan representasi grafik khusus di mana setiap tepi asli memiliki tepi terbalik di arah lain. Bobot masing-masing tepi menunjukkan seberapa banyak lagi aliran yang dapat kita rutekan melaluinya. Pada awal algoritma, bobot setiap sisi asli sama dengan kapasitas sisi, dan bobot setiap sisi sebaliknya adalah nol. Gambar12.20 menunjukkan representasi baru untuk grafik contoh kita. Algoritma Ford-Fulkerson terdiri dari beberapa putaran. Pada setiap putaran, algoritma menemukan jalur dari sumber ke sink sedemikian rupa sehingga setiap sisi pada jalur memiliki bobot positif. Jika ada lebih dari satu kemungkinan jalur yang tersedia, salah satu dari mereka dapat dipilih. Setelah memilih jalur, aliran meningkat sebanyak x unit, di mana x adalah bobot tepi terkecil pada jalur. Selain itu, bobot setiap sisi pada lintasan berkurang x, dan bobot setiap sisi sebaliknya bertambah x.

Idenya adalah bahwa meningkatkan aliran mengurangi jumlah aliran yang dapat melewati tepi di masa depan. Di sisi lain, adalah mungkin untuk membatalkan aliran

nanti dengan menggunakan tepi sebaliknya jika ternyata akan bermanfaat untuk mengarahkan aliran dengan cara lain. Algoritme meningkatkan aliran selama ada jalur dari sumber ke sink melalui tepi berbobot positif. Kemudian, jika tidak ada jalur seperti itu, algoritma berakhir dan aliran maksimum telah ditemukan.

Gambar 12.21 menunjukkan bagaimana algoritma Ford-Fulkerson menemukan aliran maksimum untuk grafik contoh kita. Dalam hal ini, ada empat putaran. Pada putaran pertama, algoritme memilih jalur 1 → 2 → 3 → 5 → 6. Bobot tepi minimum pada jalur ini adalah 2, sehingga aliran bertambah 2 satuan. Kemudian, algoritma memilih tiga jalur lain yang meningkatkan aliran sebesar 3,1, dan 1 satuan. Setelah ini, tidak ada jalur dengan tepi berbobot positif, sehingga aliran maksimumnya adalah 2+3+1+1=7.

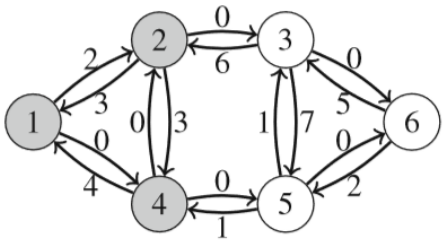


Gambar 12.21 Algoritma Ford–Fulkerson

Menemukan Jalan

Algoritma Ford-Fulkerson tidak menentukan bagaimana kita harus memilih jalur yang meningkatkan aliran. Bagaimanapun, algoritme akan berhenti cepat atau lambat dan dengan benar menemukan aliran maksimum. Namun, efisiensi algoritma tergantung pada bagaimana jalur dipilih. Cara sederhana untuk menemukan jalur adalah dengan menggunakan pencarian mendalam-pertama. Biasanya ini bekerja dengan baik, tetapi dalam kasus terburuk, setiap jalur hanya meningkatkan aliran satu unit, dan algoritmanya lambat. Untungnya, kita dapat menghindari situasi ini dengan

menggunakan salah satu teknik berikut: Algoritme Edmonds-Karp memilih setiap jalur sehingga jumlah tepi pada jalur menjadi sekecil mungkin. Hal ini dapat dilakukan dengan menggunakan pencarian luas-pertama alih-alih pencarian kedalaman-pertama untuk menemukan jalur. Dapat dibuktikan bahwa ini menjamin bahwa aliran meningkat dengan cepat, dan kompleksitas waktu dari algoritma adalah $O(m^2n)$.



Gambar 12.22 Node 1, 2, dan 4 termasuk dalam himpunan A

Algoritme penskalaan kapasitas[15] menggunakan pencarian mendalam-pertama untuk menemukan jalur di mana setiap bobot tepi setidaknya merupakan nilai ambang bilangan bulat. Awalnya, nilai ambang adalah beberapa angka besar, misalnya, jumlah semua bobot tepi dari grafik. Selalu ketika jalan tidak dapat ditemukan, nilai ambang dibagi dengan 2. Algoritma berakhir ketika nilai ambang menjadi 0. Kompleksitas waktu dari algoritma adalah $O(m^2 \log c)$, di mana c adalah nilai ambang awal. Dalam praktiknya, algoritme penskalaan kapasitas lebih mudah diimplementasikan, karena pencarian kedalaman pertama dapat digunakan untuk menemukan jalur. Kedua algoritme tersebut cukup efisien untuk masalah yang biasanya muncul dalam kontes pemrograman.

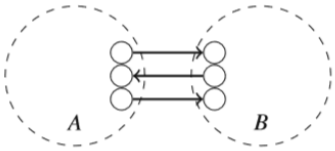
Potongan Minimum

Ternyata setelah algoritma Ford–Fulkerson telah menemukan aliran maksimum, itu juga menentukan cut minimum. Perhatikan grafik yang dihasilkan oleh algoritma, dan biarkan A menjadi himpunan node yang dapat dicapai dari sumber menggunakan tepi berbobot positif. Sekarang potongan minimum terdiri dari tepi-tepi graf asli yang dimulai pada beberapa simpul di A, berakhir di beberapa simpul di luar A, dan yang kapasitasnya digunakan sepenuhnya pada aliran maksimum. Misalnya, pada Gambar 12.22, A terdiri dari simpul 1, 2, dan 4, dan minimum tepi potongnya adalah 2→3 dan 4→5, yang beratnya 6+1=7.

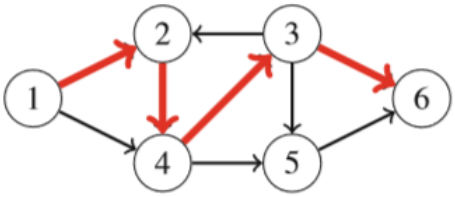
Mengapa aliran yang dihasilkan oleh algoritma maksimum dan mengapa pemotongan minimum? Alasannya adalah bahwa graf tidak dapat memuat aliran yang ukurannya lebih besar dari berat potongan graf tersebut. Oleh karena itu, selalu ketika aliran dan potongan sama, mereka adalah aliran maksimum dan potongan minimum. Untuk melihat mengapa di atas berlaku, pertimbangkan potongan grafik sedemikian rupa sehingga sumbernya milik A, wastafel milik B, dan ada beberapa tepi di antara set (Gambar 12.23). Ukuran potongan adalah jumlah bobot sisi yang bergerak dari A ke B. Ini adalah batas atas untuk aliran dalam grafik, karena aliran harus mengalir dari A ke B. Jadi, ukuran a aliran maksimum lebih kecil dari atau sama dengan ukuran setiap potongan dalam grafik. Di sisi lain, algoritma Ford-Fulkerson menghasilkan aliran yang

[15] Algoritma elegan ini tidak terlalu terkenal; deskripsi rinci dapat ditemukan di buku teks oleh Ahuja, Magnanti, dan Orlin

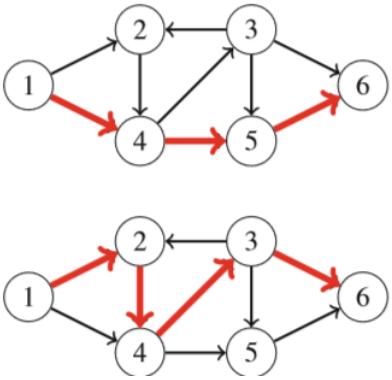
ukurannya sama persis dengan ukuran potongan grafik. Dengan demikian, aliran harus menjadi aliran maksimum, dan pemotongan harus minimum.



Gambar 12.23 Merutekan aliran dari A ke B



Gambar 12.24 Dua jalur edge-disjoint dari node 1 ke node 6



Gambar 12.25 Jalur node-disjoint dari node 1 ke node 6

12.3.2 Jalur Terputus

Banyak masalah graf dapat diselesaikan dengan mereduksinya menjadi masalah aliran maksimum. Contoh pertama kita dari masalah tersebut adalah sebagai berikut: kita diberikan grafik berarah dengan sumber dan sink, dan tugas kita adalah menemukan jumlah maksimum jalur terputus dari sumber ke sink.

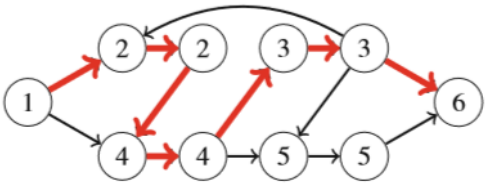
Jalur Tepi-Terputus

Kami pertama fokus pada masalah menemukan jumlah maksimum jalur tepi-disjoint dari sumber ke wastafel. Ini berarti bahwa setiap sisi dapat muncul di paling banyak satu jalur. Sebagai contoh, pada Gambar 12.24, jumlah maksimum jalur edge-disjoint adalah 2 (1→2→4→3→6 dan 1→4→5→6). Ternyata jumlah maksimum jalur tepi-disjoint selalu sama dengan aliran maksimum grafik di mana kapasitas setiap tepi adalah satu. Setelah aliran maksimum telah dibangun, jalur tepi-disjoint dapat ditemukan dengan rakus dengan mengikuti jalur dari sumber ke wastafel.

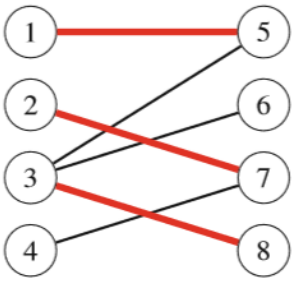
Jalur Node-Disjoint

Kemudian, pertimbangkan masalah menemukan jumlah maksimum jalur simpul-disjoint dari sumber ke sink. Dalam hal ini, setiap node, kecuali source dan sink, mungkin muncul paling banyak di satu jalur, yang dapat mengurangi jumlah maksimum jalur terputus-putus. Memang, dalam grafik contoh kami, jumlah maksimum jalur simpul-disjoint adalah 1 (Gambar 12.25). Kita juga dapat mengurangi masalah ini

menjadi masalah aliran maksimum. Karena setiap node dapat muncul di jalur paling batu, kita harus membatasi aliran yang melewati node. Konstruksi standar untuk ini adalah membagi setiap node menjadi dua node sedemikian rupa sehingga node pertama memiliki tepi masuk dari node asli, node kedua memiliki tepi keluar dari node asli, dan ada tepi baru dari node pertama ke simpul kedua. Gambar12.26 menunjukkan grafik yang dihasilkan dan aliran maksimumnya dalam contoh kita.



Gambar 12.26 Sebuah konstruksi yang membatasi aliran melalui node



Gambar 12.27 Pencocokan maksimum

12.3.3 Kecocokan Maksimum

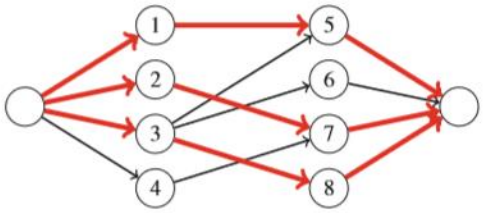
Pencocokan maksimum dari suatu graf adalah himpunan pasangan simpul dengan ukuran maksimum di mana setiap pasangan terhubung dengan sisi dan setiap simpul memiliki paling banyak satu pasangan. Sementara memecahkan masalah pencocokan maksimum dalam graf umum membutuhkan algoritma yang rumit, masalahnya jauh lebih mudah untuk dipecahkan jika kita mengasumsikan bahwa graf tersebut bipartit. Dalam hal ini kita dapat mengurangi masalah ke masalah aliran maksimum. Simpul-simpul graf abipartit selalu dapat dibagi menjadi dua kelompok sehingga semua sisi graf bergerak dari grup kiri ke grup kanan. Sebagai contoh, Gambar 12.27 menunjukkan kecocokan maksimum dari graf bipartit yang grup kirinya {1,2,3,4}dan grup kanannya{5,6,7,8}.

Kita dapat mengurangi masalah pencocokan maksimum bipartit menjadi masalah aliran maksimum dengan menambahkan dua node baru ke grafik: sumber dan sink. Kami juga menambahkan tepi dari sumber ke setiap simpul kiri dan dari setiap simpul kanan ke wastafel. Setelah ini, ukuran aliran maksimum pada graf yang dihasilkan sama dengan ukuran pencocokan maksimum pada graf asli. Misalnya, Gambar 12.28 menunjukkan reduksi dan aliran maksimum untuk grafik contoh kita.

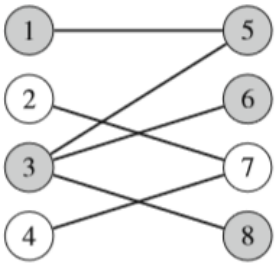
Teorema Hall

Teorema Hall dapat digunakan untuk mengetahui apakah graf partit memiliki kecocokan yang memuat semua node kiri atau kanan. Jika jumlah node kiri dan kanan adalah sama, teorema Hall memberitahu kita apakah mungkin untuk membangun pencocokan sempurna yang berisi semua node dari grafik. Asumsikan bahwa kita ingin menemukan kecocokan yang berisi semua node kiri. Misalkan X adalah himpunan simpul kiri dan misalkan f (X) himpunan tetangganya. Menurut teorema Hall,

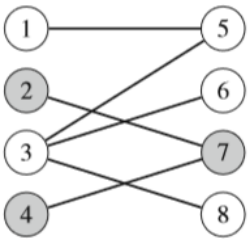
kecocokan yang berisi semua simpul kiri ada tepat ketika untuk setiap himpunan X yang mungkin, kondisi|X|≤|f (X)| berlaku.



Gambar 12.28 Pencocokan maksimum sebagai aliran maksimum



Gambar 12.29 X ={ 1,3}danf (X) ={ 5,6,8}

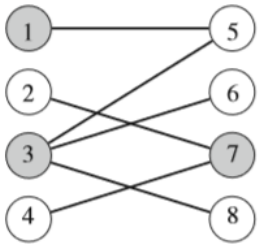


Gambar 12.30 X ={ 2,4}danf (X) ={ 7}

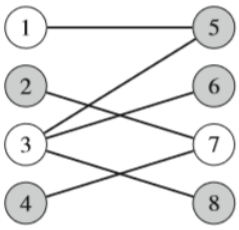
Mari kita pelajari teorema Hall dalam contoh grafik kita. Pertama, misalkan X ={ 1,3} yang menghasilkan f (X) ={ 5,6,8}(Gambar 12.29).Kondisi teorema Hall berlaku, karena |X|=2dan|f (X)|=3. Maka, misalkan X ={ 2,4}hasil mana f (X) ={ 7}(Gambar 12.30). Dalam hal ini, |X|=2 dan|f (X)|=1, maka kondisi teorema Hall tidak terus.Ini berarti bahwa tidak mungkin untuk membentuk pencocokan sempurna untuk grafik.Hasil ini tidak mengejutkan, karena kita sudah tahu bahwa pencocokan maksimum dari grafik adalah 3 dan bukan 4. Jika kondisi teorema Hall tidak berlaku, himpunan X menjelaskan mengapa kita tidak dapat membentuk pencocokan tersebut. Karena X berisi lebih banyak node daripada f (X) untuk semua node, pada Gambar.12 dan Untuk semua node, ada beberapa node daripada f (X). 4 harus terhubung dengan node 7, yang tidak mungkin.

Teorema K˝onig

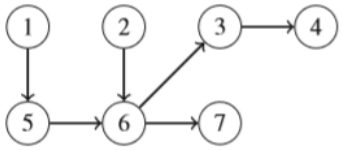
Penutup simpul minimum dari suatu graf adalah himpunan simpul minimum sedemikian rupa sehingga setiap tepi graf memiliki setidaknya satu titik akhir dalam himpunan tersebut. Dalam grafik umum, menemukan penutup simpul minimum adalah masalah NP-hard. Namun, jika grafiknya bipartit, teorema Ko˝nig memberi tahu kita bahwa ukuran penutup simpul minimum selalu sama dengan ukuran pencocokan maksimum. Dengan demikian, kita dapat menghitung ukuran penutup simpul minimum menggunakan algoritma aliran maksimum.



Gambar 12.31 Penutup node minimum



Gambar 12.32 Satu set independen maksimum



Gambar 12.33 Contoh graf untuk membuat penutup jalur

Misalnya, karena kecocokan maksimum dari grafik contoh kita adalah 3, teorema K˝onig memberi tahu kita bahwa ukuran penutup simpul minimum juga 3. Gambar 12.31 menunjukkan bagaimana penutup semacam itu dapat dibangun. Node yang tidak termasuk ke dalam penutup node minimum membentuk himpunan independen maksimum. Ini adalah himpunan node terbesar yang mungkin sehingga tidak ada node di dalam himpunan yang terhubung dengan sebuah sisi. Sekali lagi, menemukan himpunan bebas maksimum dalam graf umum adalah masalah NP-hard, tetapi dalam graf bipartit kita dapat menggunakan teorema K˝onig untuk menyelesaikan masalah secara efisien. Gambar12.32 menunjukkan himpunan bebas maksimum untuk grafik contoh kita.

12.3.4 Penutup jalan

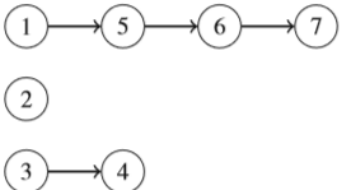
Path cover adalah sekumpulan jalur dalam graf sedemikian rupa sehingga setiap simpul dari graf tersebut memiliki setidaknya satu jalur. Ternyata dalam graf asiklik berarah, kita dapat mereduksi masalah menemukan tutupan jalur minimum ke masalah menemukan aliran maksimum di graf lain.

Penutup Jalur Node-Disjoint

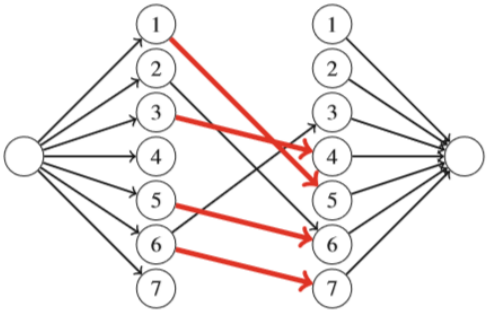
Dalam penutup jalur node-disjoint, setiap node milik tepat satu jalur. Sebagai contoh, perhatikan grafik pada Gambar 12.33. Penutup jalur node-disjoint minimum dari grafik ini terdiri dari tiga jalur (Gambar 12.34). Kita dapat menemukan minimum node-disjointpathcover dengan membuat graf yang cocok di mana setiap node dari graf asli diwakili oleh dua node: node kiri dan node kanan. Ada tepi dari simpul kiri ke simpul kanan jika ada tepi seperti itu pada graf aslinya.

Selain itu, grafik yang cocok berisi sumber dan wastafel, dan ada tepi dari sumber ke semua node kiri dan dari semua node kanan ke wastafel. Setiap tepi dalam pencocokan

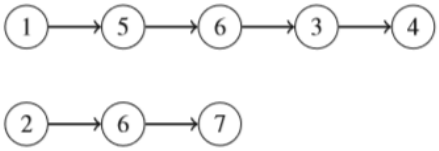
maksimum dari grafik yang cocok sesuai dengan tepi dalam penutup jalur simpul-disjoint minimum dari grafik asli. Dengan demikian, ukuran penutup jalur simpul-disjoint minimum adalah n−c, di mana n adalah jumlah simpul dalam graf asli, dan c adalah ukuran pencocokan maksimum. Misalnya, Gambar 12.35 menunjukkan grafik yang cocok untuk grafik di Gambar 12.33. Pencocokan maksimum adalah 4, sehingga penutup jalur simpul-disjoint minimum terdiri dari 7−4=3 jalur.



Gambar 12.34 Penutup jalur simpul-disjoint minimum



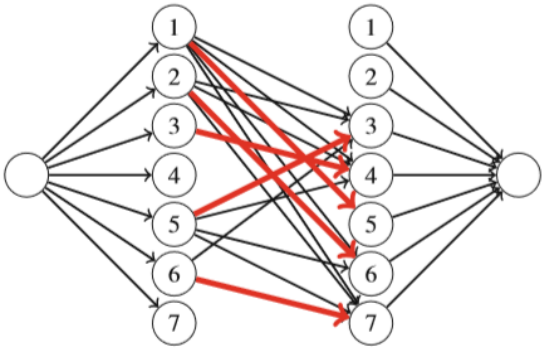
Gambar 12.35 Grafik yang cocok untuk menemukan penutup jalur simpul-disjoint minimum



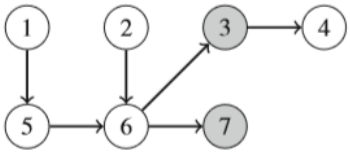
Gambar 12.36 Penutup jalur umum minimum

12.3.4 Penutup Jalan

Penutup jalur umum adalah penutup jalur di mana sebuah simpul dapat dimiliki oleh lebih dari satu jalur. Penutup jalur umum minimum mungkin lebih kecil dari penutup jalur simpul-disjoint minimum, karena sebuah simpul dapat digunakan beberapa kali dalam jalur. Perhatikan kembali graf pada Gambar 12.33.Penutup jalur umum minimum dari graf ini terdiri dari dua jalur (Gambar 12.36). Penutup jalur umum minimum dapat ditemukan hampir seperti penutup jalur simpul-putus-putus minimum. Cukuplah dengan menambahkan beberapa sisi baru pada graf yang cocok sehingga selalu ada sisi a → b ketika ada lintasan dari a ke b pada graf asli (mungkin melalui beberapa simpul). Gambar12.37 menunjukkan grafik pencocokan yang dihasilkan untuk grafik contoh kita.



Gambar 12.37 Grafik yang cocok untuk menemukan penutup jalur umum minimum



Gambar 12.38 Node 3 dan 7 membentuk antirantai maksimum

Teorema Dilworth

Antichain adalah himpunan simpul dalam graf sedemikian rupa sehingga tidak ada jalur dari simpul mana pun ke simpul lain menggunakan tepi graf. Teorema Dilworth menyatakan bahwa dalam graf asiklik berarah, ukuran penutup jalur umum minimum sama dengan ukuran antirantai maksimum. Misalnya, pada Gambar 12.38, node 3 dan 7 membentuk antichain dari dua node. Ini adalah antirantai maksimum, karena penutup jalur umum minimum dari grafik ini memiliki dua jalur (Gambar 12.36).

12.4 Pohon Pencarian Kedalaman Pertama

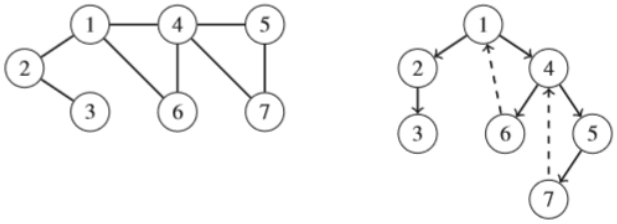
Ketika *depth-first search* memproses grafik terhubung, ia juga membuat pohon merentang berarah yang berakar yang dapat disebut sebagai pohon telusur pertama mendalam. Kemudian, grafik tepi dapat diklasifikasikan menurut perannya selama pencarian. Dalam graf tak berarah, akan ada dua jenis tepi: tepi pohon yang termasuk dalam pohon pencarian kedalaman-pertama dan tepi belakang yang menunjuk ke simpul yang sudah dikunjungi. Perhatikan bahwa tepi belakang selalu menunjuk ke nenek moyang dari sebuah simpul. Sebagai contoh, Gambar 12.39 menunjukkan grafik dan pohon pencarian depth-first. Tepi padat adalah tepi pohon, dan tepi putus-putus adalah tepi belakang. Pada bagian ini, kita akan membahas beberapa aplikasi untuk pohon pencarian kedalaman-pertama dalam pemrosesan grafik.

12.4.1 Bikonektivitas

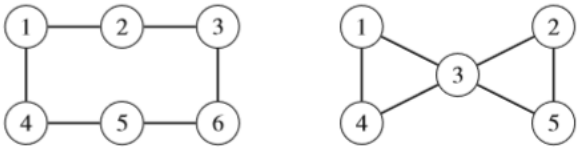
Grafik terhubung disebut bikoneksi jika tetap terhubung setelah menghapus simpul tunggal (dan tepinya) dari grafik. Sebagai contoh, pada Gambar 12.40, graf kiri terhubung dua, tetapi graf kanan tidak. Grafik kanan tidak bikoneksi, karena menghilangkan simpul 3 dari graf memutuskan graf dengan membaginya menjadi dua komponen{1,4}dan{2,5}.

Sebuah node disebut titik artikulasi jika menghapus node dari grafik memutuskan hubungan grafik. Dengan demikian, graf bikoneksi tidak memiliki titik artikulasi. Dengan cara yang sama, sebuah edge disebut bridge jika menghilangkan edge dari graf akan memutuskan graf tersebut. Sebagai contoh, pada Gambar 12.41, node 4, 5, dan

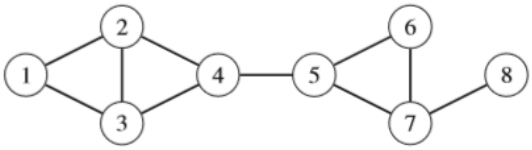
7 adalah titik artikulasi, dan edge 4-5 dan 7-8 adalah bridge. Kita dapat menggunakan depth-firstsearch untuk secara efisien menemukan titik dan jembatan artikulasi dalam sebuah graf. Pertama, untuk menemukan jembatan, kita memulai pencarian kedalaman-pertama pada simpul arbitrer, yang membangun pohon pencarian kedalaman-pertama. Sebagai contoh, Gambar 12.42 menunjukkan pohon pencarian depth-first untuk grafik contoh kita.



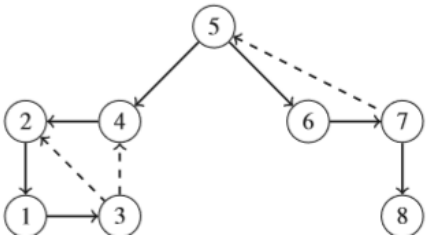
Gambar 12.39 Sebuah grafik dan pohon pencarian kedalaman-pertamanya



Gambar 12.40 Grafik kiri bikoneksi, grafik kanan tidak



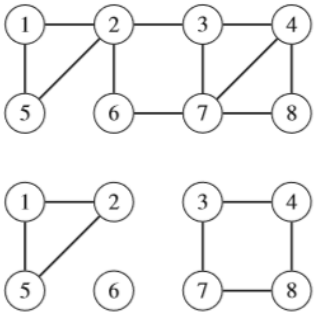
Gambar 12.41 Grafik dengan tiga titik artikulasi dan dua jembatan



Gambar 12.42 Menemukan jembatan dan titik artikulasi menggunakan pencarian kedalaman-pertama

Sebuah edge a → b berhubungan dengan sebuah bridge tepat ketika itu adalah sebuah edge pohon, dan tidak ada back edge dari subtree b ke a atau ancestor dari a. Sebagai contoh, pada Gambar 12.42, edge 5 → 4 merupakan bridge, karena tidak ada back edge dari node {1,2,3,4}ke node 5. Namun, edge 6→7 bukanlah bridge, karena ada adalah tepi belakang 7→5, dan simpul 5 adalah nenek moyang dari simpul 6. Menemukan titik artikulasi sedikit lebih sulit, tetapi kita dapat kembali menggunakan pohon pencarian depthfirst. Pertama, jika simpul x adalah akar dari pohon, itu adalah titik artikulasi tepat ketika ia memiliki dua atau lebih anak. Kemudian, jika x bukan akar, itu adalah titik artikulasi tepat ketika ia memiliki anak yang subpohonnya tidak mengandung tepi belakang dari leluhur x.

Sebagai contoh, pada Gambar 12.42, simpul 5 adalah titik artikulasi, karena merupakan akar dan memiliki dua anak, dan simpul 7 adalah titik artikulasi, karena subpohon dari anak 8 tidak mengandung tepi belakang ke nenek moyang dari 7. Namun simpul 2 bukan merupakan titik artikulasi, karena terdapat sisi belakang 3 → 4, dan simpul 8 bukan merupakan titik artikulasi, karena tidak memiliki anak.



Gambar 12.43 Graf dan subgraf Euler

12.4.2 Subgraf Euler

Suatu subgraf Euler dari suatu graf berisi simpul-simpul dari graf tersebut dan suatu himpunan bagian dari sisi-sisinya sedemikian rupa sehingga derajat setiap simpulnya genap. Sebagai contoh, Gambar 12.43 menunjukkan graf dan subgraf Eulernya. Pertimbangkan masalah menghitung jumlah subgraf Euler untuk graf terhubung. Ternyata ada rumus sederhana untuk ini: selalu ada 2^k subgraf Euler di mana k adalah jumlah tepi belakang di pohon pencarian kedalaman-pertama dari grafik. Perhatikan bahwa k =m−(n−1) di mana n adalah jumlah node dan m adalah jumlah edge.

Pohon pencarian kedalaman-pertama membantu untuk memahami mengapa rumus ini berlaku. Pertimbangkan setiap subset tetap dari tepi belakang di pohon pencarian kedalaman-pertama. Untuk membuat subgraf Euler yang berisi sisi-sisi ini, kita perlu memilih subset dari sisi-sisi pohon sehingga setiap simpul memiliki derajat genap. Untuk melakukan ini, kami memproses pohon dari bawah ke atas dan selalu menyertakan tepi pohon di subgraf tepat ketika itu menunjuk ke simpul yang derajatnya genap dengan tepi. Kemudian, karena jumlah derajat genap, derajat simpul akar juga genap.

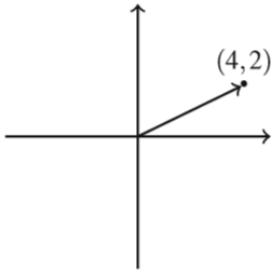
BAB 13
GEOMETRI

13.1 Teknik Geometris

Tantangan ketika memecahkan masalah geometris adalah bagaimana mendekati masalah sehingga jumlah kasus khusus sekecil mungkin dan ada cara mudah untuk mengimplementasikan solusi. Di bagian ini, kita akan membahas seperangkat alat yang membuat pemecahan masalah geometris lebih mudah.

13.1.1 Bilangan Kompleks

Bilangan kompleks adalah bilangan berbentuk x +yi, di mana i =√−1 adalah satuan imajiner. Interpretasi geometris dari bilangan kompleks adalah bahwa bilangan itu mewakili titik dua dimensi (x, y) atau vektor dari titik asal ke titik (x, y). Sebagai contoh, Gambar.13.1 mengilustrasikan bilangan kompleks 4+2i.



Gambar 13.1 Bilangan kompleks 4+2i diinterpretasikan sebagai titik dan vektor

Kompleks kelas bilangan kompleks C++ berguna ketika memecahkan masalah geometris. Dengan menggunakan kelas, kita dapat merepresentasikan titik dan vektor sebagai bilangan kompleks, dan menggunakan fitur kelas untuk memanipulasinya. Untuk melakukan ini, pertama-tama mari kita definisikan tipe koordinat C. Tergantung pada situasinya, tipe yang cocok adalah long long atau long double. Sebagai aturan umum, adalah baik untuk menggunakan koordinat bilangan bulat bila memungkinkan, karena perhitungan dengan bilangan bulat adalah tepat. Berikut adalah kemungkinan definisi tipe koordinat:

```
typedef long long C;
```

dan

```
typedef long double C;
```

Setelah ini, kita dapat mendefinisikan tipe kompleks P yang merepresentasikan sebuah titik atau vektor:

```
typedef complex<C> P;
```

Akhirnya, makro berikut mengacu pada koordinat x dan y:

```
#define X real()
#define Y imag()
```

Misalnya, kode berikut membuat titik p = (4,2) dan mencetak koordinat x dan y:

```
P p = {4,2};
cout << p.X << "" << p.Y << "\n"; / / 4 2
```

Kemudian, kode berikut membuat vektor v= (3,1) dan u = (2,2), dan setelah itu menghitung jumlah s =v+u.

```
P v = {3,1};
P u = {2,2};
```

```
P s = v+u;
cout << s.X << "" << s.Y << "\n"; / / 5 3
```

Fungsi

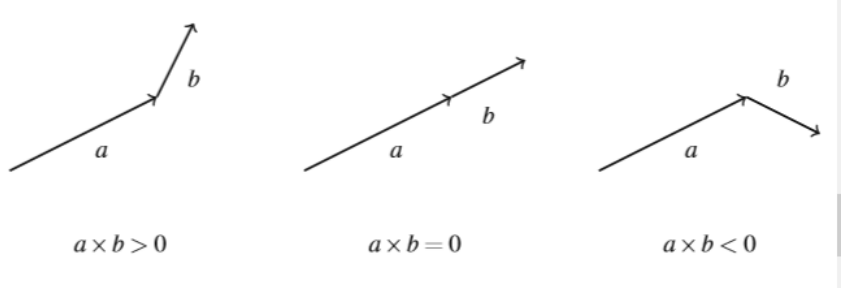
Kelas kompleks juga memiliki fungsi yang berguna dalam masalah geometri. Fungsi-fungsi berikut ini hanya boleh digunakan bila jenis koordinatnya adalah ganda panjang (atau jenis titik mengambang lainnya). Fungsi abs(v) menghitung panjang |v| dari vektor v= (x,y) menggunakan rumus $\sqrt{x^2+y^2}$. Fungsi tersebut juga dapat digunakan untuk menghitung jarak antara titik (x_1, y_1) dan (x_2, y_2), karena jarak tersebut sama dengan panjang vektor (x_2 x_1, y_2 y_1). Sebagai contoh, kode berikut menghitung jarak antara titik (4,2) dan (3,−1)

```
P a = {4,2};
P b = {3,-1};
cout << abs(b-a) << "\n"; / / 3 . 1 6 2 2 8
```

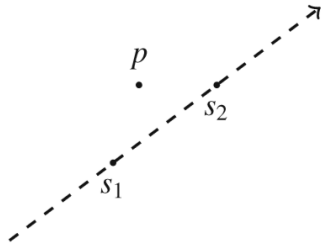
Fungsi arg(v) menghitung sudut vektor v= (x, y) terhadap sumbu x. Fungsi memberikan sudut dalam radian, di mana r radian sama dengan 180r/π derajat. Besar sudut vektor yang menunjuk ke kanan adalah 0, dan sudut berkurang searah jarum jam dan bertambah berlawanan arah jarum jam. Fungsi polar(s,a) membangun sebuah vektor yang panjangnya s dan yang menunjuk ke sudut a, diberikan dalam radian. Sebuah vektor dapat diputar dengan sudut a dengan mengalikannya dengan vektor dengan panjang 1 dan sudut a. Kode berikut menghitung sudut vektor (4,2), memutar 1/2 radian berlawanan arah jarum jam, lalu menghitung sudut lagi:

```
P v = {4,2};
cout << arg(v) << "\n"; / / 0 . 4 6 3 6 4 8
v *= polar(1.0,0.5);
cout << arg(v) << "\n"; / / 0 . 9 6 3 6 4 8
```

Titik dan Garis Hasil kali silang a×b dari vektor a = (x1, y1) dan b = (x2, y2) didefinisikan sebagai x1y2 x2y1. Ini memberitahu kita arah yang b ternyata ketika ditempatkan langsung setelah a. Ada tiga kasus yang diilustrasikan pada Gambar.13.2:



Gambar 13.2 Interpretasi produk silang



Gambar 13.3 Menguji lokasi suatu titik

- a×b > 0: b belok kiri
- a×b =0: b tidak berputar (atau berputar 180 derajat)

- a×b < 0: b belok kanan

Sebagai contoh, perkalian silang dari vektor a = (4,2) dan b = (1,2) adalah 4·2−2·1 =6, yang sesuai dengan skenario pertama Gambar.13.2. Produk silang dapat dihitung menggunakan kode berikut:

```
P a = {4,2};
P b = {1,2};
C p = (conj(a)*b).Y; / / 6
```

Kode di atas berfungsi, karena fungsi konj meniadakan koordinat y dari suatu vektor, dan ketika vektor (x_1,$-y_1$) dan (x_2, y_2) dikalikan, koordinat y hasilnya adalah $x_1y_2 - x_2y_1$. Selanjutnya kita akan membahas beberapa aplikasi produk silang.

Lokasi Titik Pengujian

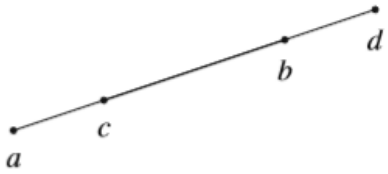
Produk silang dapat digunakan untuk menguji apakah suatu titik terletak di sisi kiri atau kanan suatu garis. Asumsikan bahwa garis melalui titik s_1 dan s_2, kita mencari dari s_1 ke s_2 dan titiknya adalah p. Misalnya, pada Gambar.13.3, p terletak di sisi kiri garis. Perkalian silang $(p-s_1)\times(p-s_2)$ menunjukkan lokasi titik p. Jika hasil kali silangnya positif, p terletak di ruas kiri, dan jika hasil kali silangnya negatif, p terletak di ruas kanan. Akhirnya, jika hasil kali silangnya nol, titik-titik s_1, s_2, dan p berada pada garis yang sama.

Perpotongan Segmen Garis

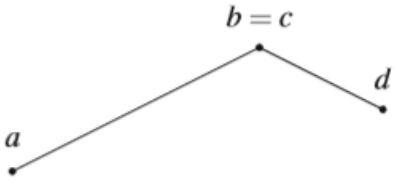
Selanjutnya, perhatikan masalah pengujian apakah dua ruas garis ab dan cd berpotongan. Ternyata jika segmen garis berpotongan, ada tiga kemungkinan kasus:

Kasus 1: Segmen garis berada pada garis yang sama dan saling tumpang tindih. Dalam hal ini, terdapat tak hingga banyaknya titik potong. Sebagai contoh, pada Gambar 13.4, semua titik antara c dan b adalah titik potong. Untuk mendeteksi kasus ini, kita dapat menggunakan produk silang untuk menguji apakah semua titik berada pada garis yang sama. Jika ya, kita dapat mengurutkannya dan memeriksa apakah segmen garis saling tumpang tindih.

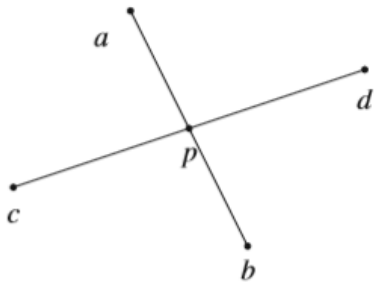
Kasus 2: Segmen garis memiliki simpul umum yang merupakan satu-satunya titik persimpangan. Sebagai contoh, pada Gambar.13.5 titik potongnya adalah b =c. Kasus ini mudah diperiksa, karena hanya ada empat kemungkinan titik potong: a =c, a =d, b =c, dan b =d.



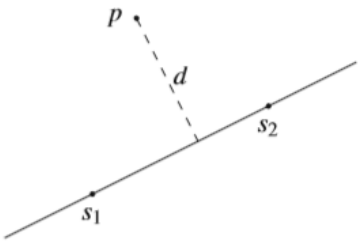
Gambar 13.4 Kasus 1: segmen garis berada pada garis yang sama dan saling tumpang tindih



Gambar 13.5 Kasus 2: segmen garis memiliki simpul yang sama



Gambar 13.6 Kasus 3: segmen garis memiliki titik potong yang bukan simpul



Gambar 13.7 Menghitung jarak dari p ke garis

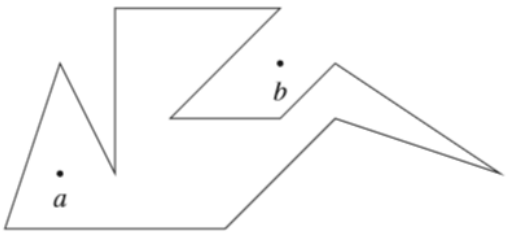
Kasus 3: Ada tepat satu titik perpotongan yang bukan merupakan simpul dari setiap ruas garis. Pada Gambar.13.6, titik p adalah titik potong. Dalam hal ini, ruas-ruas garis berpotongan tepat ketika kedua titik c dan d berada pada sisi yang berbeda dari sebuah garis yang melalui a dan b, dan titik a dan b berada pada sisi yang berbeda dari sebuah garis yang melalui c dan d. Kita dapat menggunakan produk silang untuk memeriksa ini.

13.1.2 Titik dan Garis

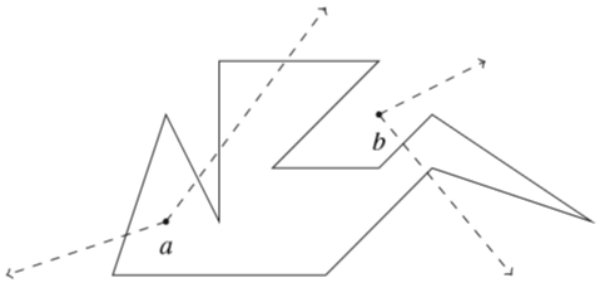
Sifat lain dari perkalian silang adalah luas segitiga dapat dihitung dengan menggunakan rumus

$$\frac{[(a-c)x(b-c)]}{2}$$

di mana a, b, dan c adalah titik sudut segitiga. Dengan menggunakan fakta ini, kita dapat memperoleh rumus untuk menghitung jarak terpendek antara titik dan garis. Misalnya, pada Gambar 13.7, d adalah jarak terpendek antara titik p dan garis yang ditentukan oleh titik s_1 dan s_2.



Gambar 13.8 Titik a di dalam dan titik b di luar poligon



Gambar 13.9 Mengirim sinar dari titik a dan b

Luas segitiga yang simpulnya adalah s_1, s_2, dan p dapat dihitung dengan dua cara:inilah1 2|s_2 s_1|d (rumus standar yang diajarkan di sekolah)dan 1/2((s_1 - p)× (s_2 - p)) (rumus perkalian silang). Jadi, jarak terpendek adalah

$$d = \frac{(s_1 - p)\, x\, (s_2 - p)}{[s_2 - s_1]}$$

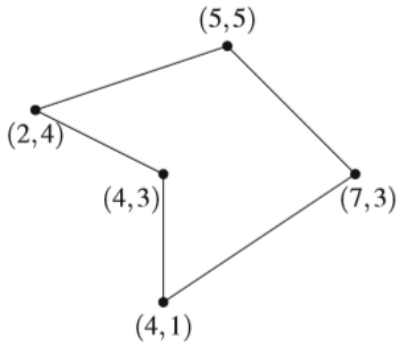
Titik dalam Poligon

Akhirnya, pertimbangkan masalah pengujian apakah titik terletak di dalam atau di luar poligon. Misalnya, pada Gambar.13.8, titik a berada di dalam poligon dan titik b berada di luar poligon. Cara yang mudah untuk menyelesaikan masalah ini adalah dengan mengirim sinar dari titik ke arah yang berubah-ubah dan menghitung berapa kali sinar itu menyentuh batas poligon. Jika bilangan ganjil, titik berada di dalam poligon, dan jika bilangan genap, titik berada di luar poligon. Misalnya, pada Gambar 13.9, sinar dari sentuhan 1 dan 3 kali batas poligon, jadi a berada di dalam poligon. Dengan cara yang sama, sinar dari b menyentuh 0 dan 2 kali batas poligon, jadi b berada di luar poligon.

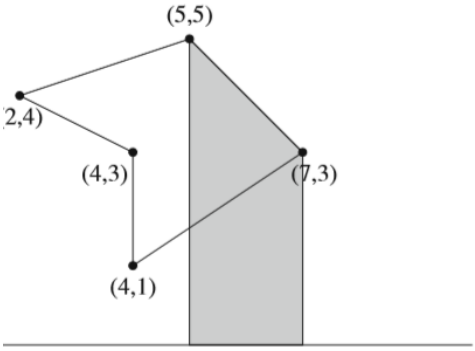
13.1.3 Daerah Poligon

Rumus umum untuk menghitung luas poligon, kadang-kadang disebut rumus tali sepatu, adalah sebagai berikut:

$$\frac{1}{2}|\sum_{i=1}^{n-1}(p_i x p_{i+1}| = \frac{1}{2}|\sum_{i=1}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)$$



Gambar 13.10 Sebuah poligon yang luasnya 17/2



Gambar 13.11 Menghitung luas poligon menggunakan trapesium

Di sini simpul-simpulnya adalah $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$,...,$p_n = (x_n, y_n)$ sedemikian rupa sehingga p_i dan p_i+1 adalah simpul-simpul yang berdekatan pada batas poligon , dan simpul pertama dan terakhir adalah sama, yaitu $p_1 = p_n$. Misalnya, luas poligon pada Gambar 13.10 adalah

$$\frac{|(2 \cdot 5 - 5 \cdot 4) + (5 \cdot 3 - 7 \cdot 5) + (7 \cdot 1 - 4 \cdot 3) + (4 \cdot 3 - 4 \cdot 1) + (4 \cdot 4 - 2 \cdot 3)|}{2} = \frac{7}{2}$$

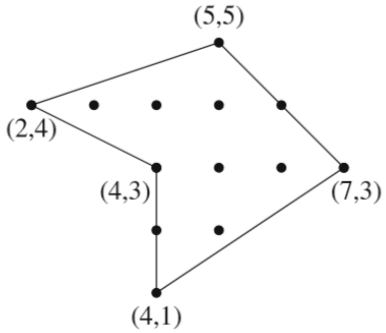
Gagasan di balik rumus tersebut adalah melalui trapesium yang satu sisinya merupakan sisi poligon, dan sisi lainnya terletak pada garis horizontal y =0. Misalnya, Gambar 13.11 menunjukkan salah satu trapesium tersebut. Luas masing-masing trapesium adalah

$$(x_{i-1} - x_i) \frac{y_i + y_{i+1}}{2}$$

di mana simpul poligon adalah p_i dan p_{i+1}. Jika $x_{i+1} > x_i$, luasnya positif, dan jika $x_{i+1} < x_i$, luasnya negatif. Kemudian, luas poligon adalah jumlah luas semua trapesium tersebut, yang menghasilkan rumus

$$\left[\sum_{i=1}^{n-1} (x_{i-1} - x_i) \frac{y_i + y_{i+1}}{2}\right] = \frac{1}{2} \left[\sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)\right]$$

Perhatikan bahwa nilai mutlak dari penjumlahan yang diambil, karena nilai dari jumlah tersebut mungkin positif atau negatif, tergantung pada apakah kita berjalan searah jarum jam atau berlawanan arah jarum jam di sepanjang batas poligon.



Gambar 13.12 Menghitung luas poligon menggunakan teorema Pick

Teorema Pick

Teorema Pick menyediakan cara lain untuk menghitung luas poligon, dengan asumsi bahwa semua simpul poligon memiliki koordinat bilangan bulat. Teorema Pick memberi tahu kita bahwa luas poligon adalah

$$a+b/2-1,$$

di mana a adalah jumlah titik bilangan bulat di dalam poligon dan b adalah jumlah titik bilangan bulat pada batas poligon. Misalnya, luas poligon pada Gambar 13.12 adalah

$$6+7/2-1=17/2.$$

13.1.4 Fungsi Jarak

Fungsi jarak mendefinisikan jarak antara dua titik. Fungsi jarak yang biasa digunakan adalah jarak Euclidean dimana jarak antara titik (x_1, y_1) dan (x_2, y_2) adalah

$$\sqrt{(x_2 - x_1)^2} + y_2 - y_1)^2$$

Fungsi jarak alternatif adalah jarak Manhattan dimana jarak antara titik (x_1, y_1) dan (x_2, y_2) adalah

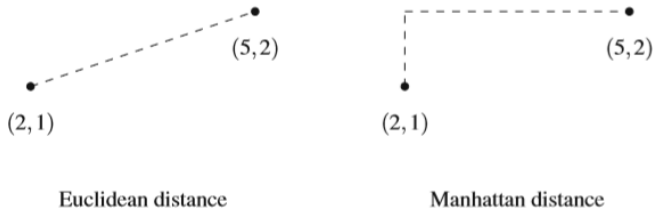
$$|x_1 - x_2| + |y_1 - y_2|$$

Sebagai contoh, pada Gambar 13.13, jarak Euclidean antara titik-titik adalah

$$\sqrt{(5-2)^2} + (2 - 1)^2 = \sqrt{10}$$

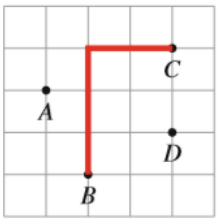
dan jarak Manhattan adalah

$$|5-2|+|2-1|=4.$$

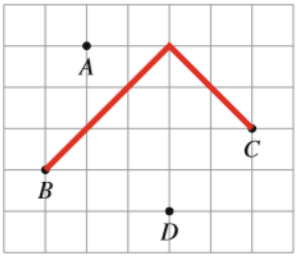


Gambar 13.13 Dua fungsi jarak

Gambar 13.14 Daerah dalam jarak 1



Gambar 13.15 Titik B dan C memiliki jarak Manhattan maksimum



Gambar 13.16 Jarak maksimum Manhattan setelah mengubah koordinat

Gambar 13.14 menunjukkan daerah yang berada dalam jarak 1 dari titik pusat, menggunakan jarak Euclidean dan Manhattan. Beberapa masalah lebih mudah dipecahkan jika jarak Manhattan digunakan daripada jarak Euclidean. Sebagai contoh, diberikan satu set titik dalam bidang dua dimensi, pertimbangkan masalah menemukan dua titik yang jarak Manhattan adalah maksimum. Misalnya, pada Gambar 13.15, kita harus memilih titik Pita C untuk mendapatkan jarak Manhattan maksimum 5. Teknik yang berguna terkait dengan jarak Manhattan adalah mengubah koordinat sehingga menjadi titik (x, y) , y−x).Ini akan memutar set titik 45◦ dan menskalakannya. Misalnya, Gambar 13.16 menunjukkan hasil transformasi dalam skenario contoh kita.

Kemudian, perhatikan dua titik $p_1 = (x_1, y_1)$ dan $p_2 = (x_2, y_2)$ yang koordinat transformasinya adalah $p'_1 = (x'_1, y'_1)$ dan $p'_2 = (x'_2, y'_2)$ .Sekarang ada dua cara untuk menyatakan jarak Manhattan antara p_1 dan p_2:

$$|x_1 - x_2| + |y_1 - y_2| = \max(|x'_1 - x'_2|, |y'_1 - y'_2|)$$

Misalnya, jika $p_1 = (1,0)$ dan $p_2 = (3,3)$, koordinat yang diubah adalah $p'_1 = (1,-1)$ dan $p'_2 = (6,0)$ dan jarak Manhattan adalah

$$|1-3| + |0-3| = \max(|1-6|, |-1-0|) = 5.$$

Koordinat yang diubah menyediakan cara sederhana untuk beroperasi dengan jarak Manhattan, karena kita dapat mempertimbangkan koordinat x dan y secara terpisah. Khususnya, untuk memaksimalkan jarak Manhattan, kita harus menemukan dua titik yang koordinat transformasinya memaksimalkan nilai

$$\max(|x'_1 - x'_2|, |y'_1 - y'_2|)$$

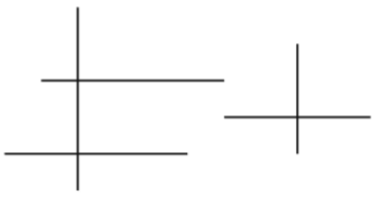
Ini mudah, karena perbedaan horizontal atau vertikal dari koordinat yang ditransformasikan harus maksimum.

13.2 Algoritma Garis Sapu

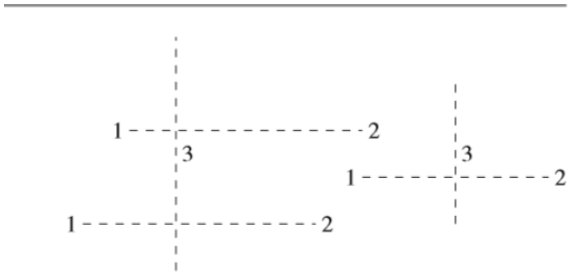
Banyak masalah geometris dapat diselesaikan dengan menggunakan algoritma garis sapuan. Ide dalam algoritma tersebut adalah untuk mewakili sebuah contoh dari masalah sebagai satu set peristiwa yang sesuai dengan titik-titik di pesawat. Kemudian, kejadian diproses dalam urutan yang meningkat sesuai dengan koordinat x atau y mereka.

13.2.1 Titik Persimpangan

Diberikan himpunan n ruas garis, masing-masing ruas horizontal atau vertikal, pertimbangkan masalah menghitung jumlah total titik potong. Misalnya, pada Gambar 13.17, ada lima ruas garis dan tiga titik potong.



Gambar 13.17 Lima ruas garis dengan tiga titik potong



Gambar 13.18 Acara yang sesuai dengan segmen garis

Sangat mudah untuk menyelesaikan masalah dalam waktu O(n2), karena kita dapat menelusuri semua pasangan segmen garis yang mungkin dan memeriksa apakah mereka berpotongan. Namun, kita dapat memecahkan masalah dengan lebih efisien dalam waktu O(n log n) menggunakan algoritma garis sapuan dan struktur data kueri rentang. Idenya adalah untuk memproses titik akhir segmen garis dari kiri ke kanan dan fokus pada tiga jenis peristiwa:

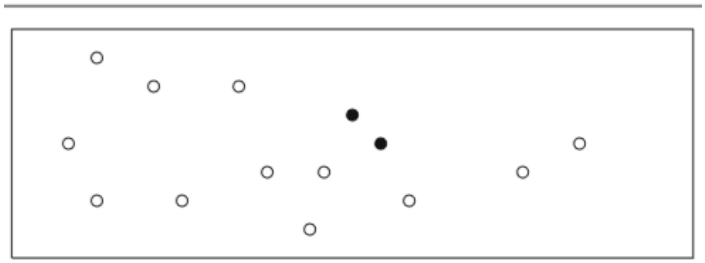
1) segmen horizontal dimulai
2) segmen horizontal berakhir
3) segmen vertikal

Gambar 13.18 menunjukkan kejadian dalam skenario contoh kita. Setelah membuat kejadian, kita menelusurinya dari kiri ke kanan dan menggunakan struktur data yang mempertahankan koordinat segmen horizontal aktif. Kejadian 1, tambahkan koordinat segmen ke struktur, dan tanggal 2, hilangkan koordinat y dari struktur. Titik potong dihitung pada kejadian 3: saat memproses segmen vertikal antara titik y_1 dan y_2, kami menghitung jumlah segmen horizontal aktif yang koordinat y berada di antara y_1 dan y_2, dan menambahkan angka ini ke jumlah total titik persimpangan. Untuk menyimpan koordinat y dari segmen horizontal, kita dapat menggunakan indeks biner atau pohon segmen, mungkin dalam kompresi indeks. Pemrosesan setiap kejadian membutuhkan waktu O(log n), sehingga algoritma bekerja dalam waktu O(n log n).

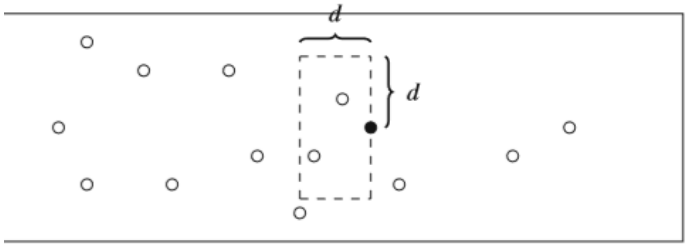
13.2.2 Masalah Pasangan Terdekat

Diberikan satu set n titik, masalah kita berikutnya adalah mencari dua titik yang jarak Euclidean-nya minimum. Misalnya, Gambar 13.19 menunjukkan sekumpulan titik, di mana pasangan terdekat dicat hitam. Ini adalah contoh lain dari masalah yang dapat diselesaikan dalam waktu O(n log n) menggunakan algoritma garis sapuan.[16] Kami menelusuri titik-titik dari kiri ke kanan dan mempertahankan nilai d: jarak minimum antara dua titik yang terlihat sejauh ini. Pada setiap titik, kita menemukan titik terdekatnya ke kiri. Jika jaraknya kurang dari d, itu adalah jarak minimum baru dan kami memperbarui nilai d.

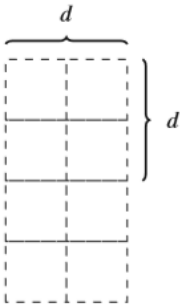
[16] Membuat algoritma yang efisien untuk masalah pasangan terdekat adalah penting untuk membuka masalah geometri komputasi. Akhirnya, Shamos dan Hoey menemukan algoritma pembagian dan penaklukan yang bekerja dalam waktu O (n log n). Algoritma garis sapuan yang disajikan di sini memiliki elemen-elemen umum dengan algoritmanya, tetapi lebih mudah untuk diimplementasikan.



Gambar 13.19 Contoh masalah pasangan terdekat

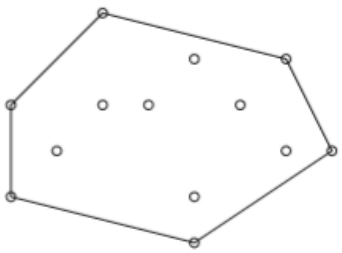


Gambar 13.20 Wilayah di mana titik terdekat harus terletak

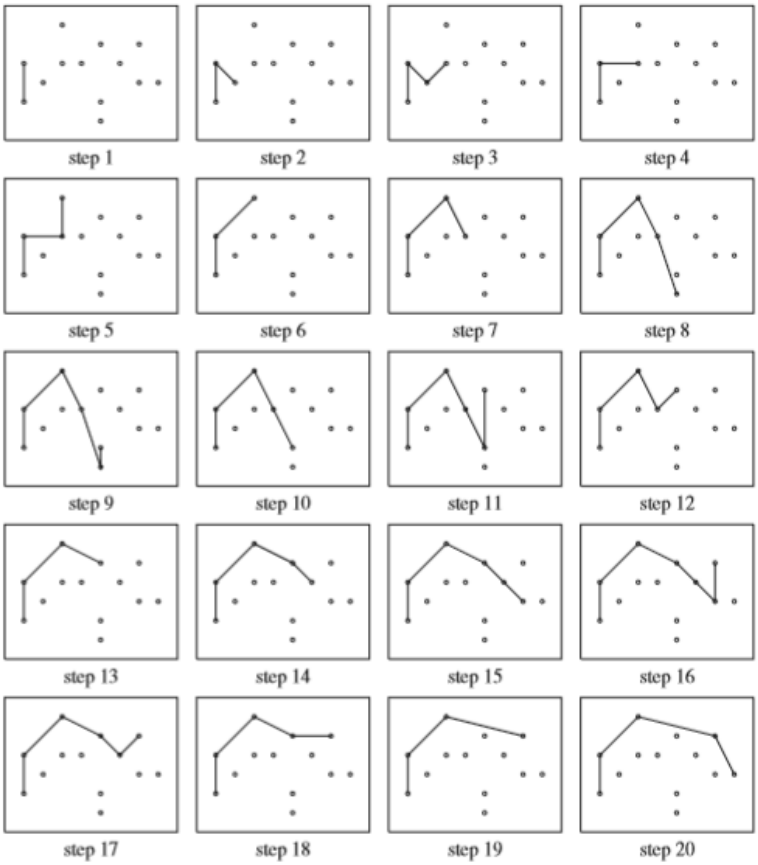


Gambar 13.21 Daerah titik terdekat mengandung O(1) titik

nilai d: jarak minimum antara dua titik yang terlihat sejauh ini. Pada setiap titik, kita menemukan titik terdekatnya ke kiri. Jika jaraknya kurang dari d, itu adalah jarak minimum baru dan kami memperbarui nilai d. Jika titik saat ini adalah (x, y) dan ada titik di sebelah kiri dalam jarak kurang dari d, koordinat x dari titik tersebut harus antara [x d,x]dan koordinat y harus antara[y−d, y+d]. Dengan demikian, cukup untuk hanya mempertimbangkan titik-titik yang terletak dalam rentang tersebut, yang membuat algoritma menjadi efisien. Misalnya, pada Gambar 13.20, wilayah yang ditandai dengan garis putus-putus berisi titik-titik yang dapat berada dalam jarak d dari titik aktif. Efisiensi algoritma salah berdasarkan fakta bahwa daerah selalu hanya berisi O(1) poin. Untuk melihat mengapa hal ini berlaku, pertimbangkan Gambar 13.21. Karena jarak minimum saat ini antara dua titik adalah d, setiap persegi d/2×d/2 dapat berisi paling banyak satu titik. Dengan demikian, paling banyak ada delapan titik di wilayah tersebut.



Gambar 13.22 Lambung cembung dari kumpulan titik



Gambar 13.23 Membangun bagian atas lambung cembung menggunakan algoritma Andrew

Kita dapat menelusuri titik-titik pada daerah dalam waktu O(log n) dengan mempertahankan himpunan titik-titik yang koordinat x-nya berada di antara [x d,x] sehingga titik-titik tersebut diurutkan dalam urutan yang meningkat menurut koordinat y-nya. Kompleksitas waktu dari algoritma adalah O(n log n), karena kita melalui n titik dan menentukan untuk setiap titik titik terdekatnya ke kiri dalam waktu O(log n).

13.2.3 Masalah Kulit Cembung

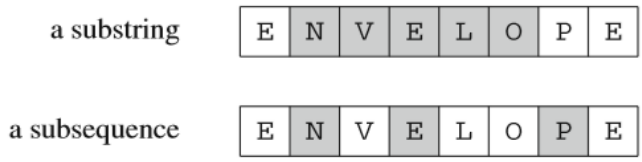
Convex hull adalah poligon cembung terkecil yang berisi semua titik dari himpunan titik tertentu. Di sini kecembungan berarti bahwa segmen garis antara dua simpul dari poligon sepenuhnya berada di dalam poligon. Misalnya, Gambar 13.22 menunjukkan lambung cembung dari himpunan titik. Ada banyak algoritma yang efisien untuk membangun lambung cembung. Mungkin yang paling sederhana di antara mereka adalah algoritma Andrew [2], yang akan kami jelaskan selanjutnya. Algoritma pertama-tama menentukan titik paling kiri dan paling kanan dalam himpunan, dan kemudian membangun lambung cembung dalam dua bagian: pertama lambung atas dan kemudian lambung bawah. Kedua bagian itu serupa, jadi kita bisa fokus membangun lambung bagian atas. Pertama, kami mengurutkan titik-titik terutama menurut koordinat x dan yang kedua menurut koordinat y. Setelah ini, kita melewati titik-titik dan menambahkan setiap titik ke lambung kapal. Selalu setelah menambahkan titik ke lambung, kami memastikan bahwa segmen garis terakhir di lambung tidak berbelok ke

kiri. Selama berbelok ke kiri, kami berulang kali menghapus titik terakhir kedua dari lambung. Gambar 13.23 menunjukkan bagaimana algoritma Andrew menciptakan lambung atas untuk kumpulan titik contoh kita.

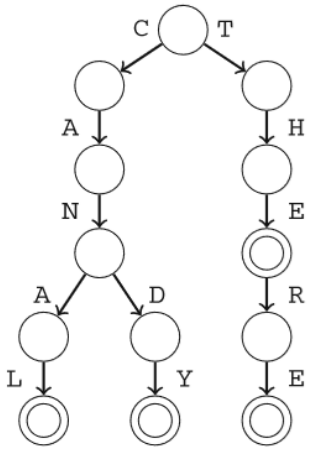
BAB 14
ALGORITMA STRING

14.1 Topik Dasar

Sepanjang bab ini, kami berasumsi bahwa semua string diindeks nol. Misalnya, string s dengan panjang n terdiri dari karakter s[0],s[1],...,s[n−1]. Substring adalah urutan karakter berurutan dalam string. Kami menggunakan notasi s[a ...b]untuk merujuk ke substring dari s yang dimulai pada posisi a dan berakhir pada posisi b. Prefiks adalah substring yang berisi karakter pertama dari sebuah string, dan suffix adalah substring yang berisi karakter terakhir dari sebuah string. Barisan berikutnya adalah barisan karakter dalam barisan aslinya. Semua substring adalah barisan, tetapi kebalikannya tidak benar (Gambar 14.1).



Gambar 14.1 NVELO adalah substring, NEP adalah subsequence



Gambar 14.2 Sebuah trie yang berisi string CANAL, CANDY, THE, dan THERE

14.1.1 Struktur Trie

Atrie adalah pohon berakar yang mempertahankan satu set string. Setiap string dalam rangkaian tersebut disimpan sebagai rantai karakter yang dimulai pada simpul akar. Jika dua string memiliki awalan umum, mereka juga memiliki rantai umum di dalam pohon. Sebagai contoh, trie pada Gambar 14.2 sesuai dengan himpunan {CANAL,CANDY,THE,THERE}. Lingkaran dalam simpul berarti bahwa string dalam himpunan berakhir di simpul. Setelah membangun sebuah trie, kita dapat dengan mudah memeriksa apakah itu berisi string yang diberikan dengan mengikuti rantai yang dimulai pada simpul akar. Kita juga dapat menambahkan string baru ke dalam percobaan dengan terlebih dahulu mengikuti rantai dan kemudian menambahkan node baru jika diperlukan. Kedua operasi tersebut bekerja dalam waktu O(n) di mana n adalah panjang string. Trie dapat disimpan dalam array

```
int trie[N][A];
```

di mana N adalah jumlah maksimum node (panjang total maksimum string dalam set) dan A adalah ukuran alfabet. Node trie diberi nomor 0,1,2,... sedemikian rupa sehingga

jumlah root adalah 0, dan trie[s][c] menentukan node berikutnya dalam rantai ketika kita berpindah dari node s menggunakan karakter C. Selalu ada yang menunjukkan bahwa kita dapat memperluas struktur trie. Misalnya, kita diberi kueri yang mengharuskan kita menghitung jumlah string dalam himpunan yang memiliki awalan tertentu. Kita dapat melakukan ini secara efisien dengan menyimpan untuk setiap node trie jumlah string yang rantainya melewati node.

	O	P	E	R	A
T	0	0	0	0	0
O	1	1	1	1	1
U	1	1	1	1	1
R	1	1	1	2	2

Gambar 14.3 Nilai fungsi lcs untuk menentukan turunan umum terpanjang dari TOUR dan OPERA

14.1.2 Pemrograman Dinamis

Pemrograman dinamis dapat digunakan untuk memecahkan banyak masalah string. Selanjutnya kita akan membahas dua contoh masalah tersebut.

Barisan Umum Terpanjang

Urutan umum terpanjang dari dua string adalah string terpanjang yang muncul sebagai urutan di kedua string. Misalnya, urutan umum terpanjang dari TOUR dan OPERA adalah OR. Dengan menggunakan pemrograman dinamis, kita dapat menentukan barisan umum terpanjang dari dua string x dan dalam waktu O (nm), di mana dan m menunjukkan panjang string. Untuk melakukan ini, kita tentukan fungsi lcs(i,j) yang memberikan panjang dari barisan umum terpanjang dari awalan x[0...i]andy[0...j]. Kemudian, kita dapat menggunakan

$$lcs(i,j) = \begin{cases} lsc(i, j\ -\ 1)\ +1 & x[i] = y[j] \\ max(lcs(i, j\ -\ 1), lcs(i\ -\ 1, j) & lainnya \end{cases}$$

Idenya adalah jika karakter x[i]dan y[j]sama, kita mencocokkannya dan menambah panjang barisan persekutuan terpanjang sebanyak satu. Jika tidak, kami menghapus karakter terakhir dari x atau y, tergantung pada pilihan mana yang optimal. Misalnya, Gambar 14.3 menunjukkan nilai fungsi lcs dalam skenario contoh kita.

Edit Jarak Jarak edit (atau jarak Levenshtein) antara dua string menunjukkan jumlah minimum operasi pengeditan yang mengubah string pertama menjadi string kedua. Operasi pengeditan yang diizinkan adalah sebagai berikut:

- menyisipkan karakter (mis., ABC →ABCA)
- menghapus karakter (misalnya, ABC →AC)
- memodifikasi karakter (misalnya, ABC → ADC)

Misalnya, jarak edit antara LOVE dan MOVIE adalah 2, karena pertama kita dapat melakukan operasi LOVE→MOVE (modifikasi) dan kemudian operasi MOVE →MOVIE (insert). Kita dapat menghitung jarak edit antara dua string x dan y dalam waktu O(nm), di mana danm adalah panjang dari string. Biarkan edit(i, j) menunjukkan jarak edit antara awalan x[0...i] dan y[0...j] . Nilai fungsi dapat dihitung menggunakan perulangan

		M	O	V	I	E
L	1	2	3	4	5
O	2	1	2	3	4
V	3	2	1	2	3
E	4	3	2	2	2

Gambar 14.4 Nilai fungsi edit untuk menentukan jarak edit antara LOVE dan MOVIE

$$\begin{aligned} edit(a,b) = &\min(edit(a,b-1)+1,\\ &edit(a-1,b)+1,\\ &edit(a-1,b-1)+cost(a,b)),\end{aligned}$$

$$where cost(a,b) = 0 if x[a]=y[b], and otherwise cost(a,b) = 1.$$

Rumus mempertimbangkan tiga cara untuk mengedit string x: masukkan karakter di akhir x, hapus karakter terakhir dari x, atau mencocokkan/memodifikasi karakter terakhir dari x. Dalam kasus terakhir, jika x[a]=y[b], kita dapat mencocokkan karakter terakhir tanpa mengedit. Misalnya, Gambar 14.4 menunjukkan nilai fungsi edit dalam skenario contoh kita.

14.2 String Hashing

Dengan menggunakan hashing string, kita dapat secara efisien memeriksa apakah dua string sama dengan membandingkan nilai hashnya. Nilai hash adalah bilangan bulat yang dihitung dari karakter string. Jika dua string sama, nilai hashnya juga sama, yang memungkinkan untuk membandingkan string berdasarkan nilai hashnya.

14.2.1 Hashing Polinomial

Cara biasa untuk mengimplementasikan hashing string adalah hashing polinomial, yang berarti bahwa nilai hash dari string s dengan panjang n adalah

$$(s[0]A^{n-1} + s[1]A^{n-2} + \cdots + s[n-1]A^0) \bmod B,$$

di mana s[0],s[1],...,s[n−1]diinterpretasikan sebagai kode karakter, dan A dan B adalah konstanta yang dipilih sebelumnya. Sebagai contoh, mari kita hitung nilai hash dari string ABACB. Kode karakter A, B, dan C adalah 65, 66, dan 67. Kemudian, kita perlu memperbaiki konstanta; misalkan A =3 dan B =97. Jadi, nilai hashnya adalah

$$(65 \cdot 3^4 + 66 \cdot 3^3 + 65 \cdot 32 + 66 \cdot 3^1 + 67 \cdot 3^0) \bmod 97 = 40.$$

Ketika hashing polinomial digunakan, kita dapat menghitung nilai hash dari setiap substring dari string s dalam waktu O(1) setelah pemrosesan awal waktu O(n). Idenya adalah untuk membangun sebuah array h sedemikian rupa sehingga h[k]berisi nilai hash dari awalan s[0...k]. Nilai array dapat dihitung secara rekursif sebagai berikut:

$$h[0]=s[0]$$
$$h[k]=(h[k-1]A+s[k]) \bmod B$$

Selain itu, kami membuat array p di mana $p[k]=A^k \bmod B$:

$$p[0]=1$$
$$p[k]=(p[k-1]A) \bmod B.$$

Membangun array di atas membutuhkan waktu O(n). Setelah ini, nilai hash dari setiap substring s[a ...b]dapat dihitung dalam waktu O(1) menggunakan rumus

$$(h[b]-h[a-1]p[b-a+1]) \bmod B$$

dengan asumsi bahwa a > 0. Jika a =0, nilai hashnya adalah h[b].

14.2.2 Aplikasi

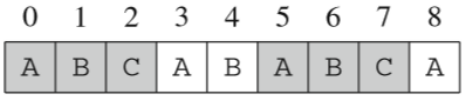
Kita dapat secara efisien menyelesaikan banyak masalah string menggunakan hashing, karena ini memungkinkan kita untuk membandingkan substring string yang berubah-ubah dalam waktu O(1). Faktanya, kita sering dapat mengambil algoritma brute force dan membuatnya efisien dengan menggunakan hashing.

Pencocokan Pola

Masalah string yang mendasar adalah masalah pencocokan pola: diberikan sebuah string s dan dapattern p, carilah posisi di mana p terjadi pada s. Misalnya, pola ABC terjadi pada posisi 0 dan 5 dalam string ABCABABCA (Gambar 14.5). Kita dapat menyelesaikan masalah pencocokan pola dalam waktu $O(n^2)$ menggunakan algoritma brute force yang melewati semua posisi di mana p dapat muncul dalam s dan membandingkan string karakter demi karakter. Kemudian, kita dapat membuat algoritma bruteforce efisien menggunakan hashing, karena setiap perbandingan string hanya membutuhkan O(1) waktu. Ini menghasilkan algoritma waktu O(n).

Substring Berbeda

Pertimbangkan masalah menghitung jumlah substring berbeda dengan panjang k dalam sebuah string. Misalnya, string ABABAB memiliki dua substring yang berbeda dengan panjang 3: ABA dan BAB. Menggunakan hashing, kita dapat menghitung nilai hash dari setiap substring dan mengurangi masalah untuk menghitung jumlah bilangan bulat yang berbeda dalam daftar, yang dapat dilakukan dalam waktu O(n log n).



Gambar 14.5 Pola ABC muncul dua kali dalam string ABCABABCA

Rotasi Minimal

Rotasi string dapat dibuat dengan berulang kali memindahkan karakter pertama string ke ujung string. Misalnya, rotasi ATLAS adalah ATLAS, TLASA, LASAT, ASATL, dan SATLA. Selanjutnya kita akan mempertimbangkan masalah menemukan rotasi string yang minimal secara leksikografis. Misalnya, rotasi minimal ATLAS adalah ASATL. Kita dapat memecahkan masalah secara efisien dengan menggabungkan hashing string dan pencarian biner. Ide utamanya adalah bahwa kita dapat menemukan urutan leksikografis dari dua string dalam waktu logaritmik.

Pertama, kami menghitung panjang awalan umum dari string menggunakan pencarian biner. Di sini hashing memungkinkan kita untuk memeriksa waktu O(1) apakah dua prefiks dengan panjang tertentu cocok. Setelah ini, kami memeriksa karakter berikutnya setelah awalan umum, yang menentukan urutan string. Kemudian, untuk memecahkan masalah, kami membuat string yang berisi dua salinan dari string asli (misalnya, ATLASATLAS) dan melalui substring dengan panjang n mempertahankan substring minimal. Karena setiap perbandingan dapat dilakukan dalam waktu O(log n), algoritma bekerja dalam waktu O(n log n).

14.2.3 Tabrakan dan Parameter

Risiko nyata ketika membandingkan nilai hash adalah tabrakan, yang berarti bahwa dua string memiliki konten yang berbeda tetapi nilai hash yang sama. Dalam kasus ini, algoritma yang bergantung pada nilai hash menyimpulkan bahwa string adalah sama,

tetapi kenyataannya tidak, dan algoritma dapat memberikan hasil yang salah. Tabrakan selalu mungkin, karena jumlah string yang berbeda lebih besar dari jumlah nilai hash yang berbeda. Namun, peluang tumbukan kecil jika konstanta A dan B dipilih dengan cermat. Cara yang biasa adalah memilih konstanta acak yang mendekati 109, misalnya, sebagai berikut:

$$A = 911382323$$
$$B = 972663749$$

Dengan menggunakan konstanta seperti itu, tipe long long dapat digunakan saat menghitung nilai hash, karena produk AB dan BB akan cocok dengan long long. Tetapi apakah cukup memiliki sekitar 109 nilai hash yang berbeda? Mari kita pertimbangkan tiga skenario di mana hashing dapat digunakan:

Skenario 1: String x dan y dibandingkan satu sama lain. Probabilitas tabrakan adalah 1/B dengan asumsi bahwa semua nilai hash memiliki kemungkinan yang sama.

Skenario 2: Sebuah string x dibandingkan dengan string y_1, y_2,...,y_n. Peluang terjadinya satu atau lebih tumbukan adalah

$$1-(1-1/B)^n.$$

Skenario 3: Semua pasangan string x_1,x_2,...,x_n dibandingkan satu sama lain. Peluang terjadinya satu atau lebih tumbukan adalah

$$1 - \frac{B \,.\, (B-1) \,.\, (B-2) \ldots (B-n+1)}{B^n}$$

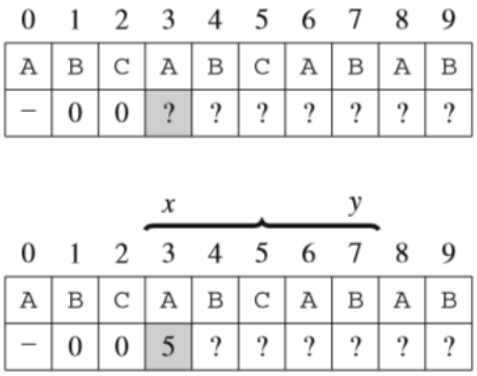
Tabel 14.1 Probabilitas tabrakan dalam skenario hashing ketika n =10^6

Konstan B	Skenario 1	Skenario 2	Skenario 3
10^3	0.00	1.00	1.00
10^6	0.00	0.63	1.00
10^9	0.00	0.00	1.00
10^{12}	0.00	0.00	0.39
10^{15}	0.00	0.00	0.00
10^{18}	0.00	0.00	0.00

Tabel 14.1 menunjukkan peluang tumbukan untuk nilai B yang berbeda bila n =10 6. Tabel menunjukkan bahwa dalam Skenario 1 dan 2, peluang tumbukan diabaikan bila B≈10^9. Namun, dalam Skenario 3 situasinya sangat berbeda: tabrakan hampir selalu terjadi ketika B≈10^9. Fenomena dalam Skenario 3 dikenal sebagai paradoks ulang tahun: jika ada n orang di sbuah ruangan, kemungkinan dua orang memiliki hari ulang tahun yang sama besar meskipun n cukup kecil. Dalam hashing, dengan demikian, ketika semua nilai hash dibandingkan satu sama lain, probabilitas bahwa beberapa dua nilai hash adalah sama besar.

0	1	2	3	4	5	6	7	8	9
A	B	C	A	B	C	A	B	A	B
–	0	0	5	0	0	2	0	2	0

Gambar 14.6 Z-array ABCABCABAB



Gambar 14.7 Skenario 1: Menghitung nilai z[3]

Kita dapat membuat probabilitas tabrakan lebih kecil dengan menghitung beberapa nilai hash menggunakan parameter yang berbeda. Tabrakan kecil kemungkinannya tidak akan terjadi di semua nilai hash pada waktu yang bersamaan. Misalnya, dua nilai hash dengan parameter B 109 berkorespondensi dengan satu nilai hash dengan parameter $B \approx 10^{18}$, yang membuat kemungkinan tumbukan menjadi sangat kecil. Beberapa orang menggunakan konstanta $B = 2^{32}$ dan $B = 2^{64}$, yang lebih mudah, karena operasi dengan bilangan bulat 32-dan 64-bit dihitung modulo 2^{32} dan 2^{64}. Namun, ini bukan pilihan yang baik, karena memungkinkan untuk membuat input yang selalu menghasilkan tabrakan ketika konstanta bentuk 2^x digunakan.

14.3 Z-Algoritma

Array-Z z dari string s dengan panjang n berisi untuk setiap k =0,1,...,n−1 panjang substring terpanjang dari s yang dimulai pada posisi k dan merupakan awalan dari s. Jadi, z[k]=p memberitahu kita bahwa s[0...p−1]sama dengan s[k ...k+ p−1], tetapi[p]dan s[k+ p]adalah karakter yang berbeda (atau panjang tali adalah k+p). Sebagai contoh, Gambar.14.6 menunjukkan Z-array ABCABCABAB. Dalam larik, misalnya, z[3]=5, karena substring ABCAB dengan panjang5 adalah awalan dari s, tetapi substring ABCABA dengan panjang 6 bukan merupakan awalan dari s.

14.3.1 Membangun Z-Array

Selanjutnya kami menjelaskan sebuah algoritma, yang disebut algoritma-Z yang secara efisien membangun array-Z dalam waktu O(n). [17] Algoritma menghitung nilai-nilai array-Z dari kiri ke kanan dengan menggunakan informasi yang sudah disimpan dalam array dan dengan membandingkan substring karakter demi karakter. Untuk menghitung nilai array Z secara efisien, algoritme mempertahankan rentang[x, y] sedemikian rupa sehingga s[x ...y]adalah awalan dari s, nilai z[x]telah ditentukan, dan y adalah sebagai besar mungkin. Karena kita tahu bahwa s[0...y−x]dan s[x ...y]sama, kita dapat menggunakan informasi ini saat menghitung nilai array berikutnya. Misalkan kita telah menghitung nilai z[0],z[1],...,z[k−1]dan kita ingin menghitung nilai z[k]. Ada tiga skenario yang mungkin:

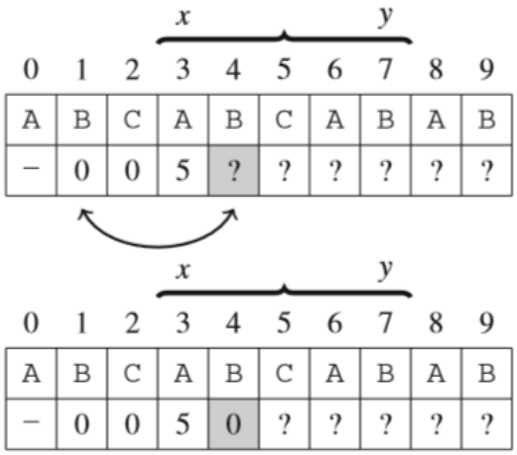
Skenario 1: y < k. Dalam hal ini, kami tidak memiliki informasi tentang posisi k, jadi kami menghitung nilai z[k] dengan membandingkan substring karakter demi karakter.

[17] Gusfield menyajikan metode Z-algorithma yang paling sederhana yang diketahui untuk pencocokan pola waktu-linier dan mengatribusikan ide asli ke Main dan Lorentz

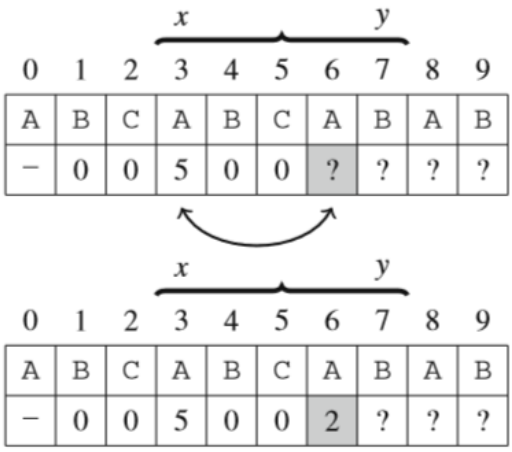
Misalnya, pada Gambar.14.7, belum ada rentang [x, y], jadi kami membandingkan substring yang dimulai dari posisi 0 dan 3 karakter demi karakter. Karena z[3]=5, rentang[x, y] baru menjadi[3,7].

Skenario2: y k andk+z[k−x]≤y.Dalam hal ini kita tahubahwa[k]=z[k−x], karena s[0...y−x] dan s[x ...y] sama dan kita tetap berada di dalam kisaran [x, y]. Sebagai contoh, pada Gambar.14.8, kami menyimpulkan bahwa z[4]=z[1]=0.

Skenario 3: y k dan k+z[k−x] > y. Dalam hal ini kita tahu bahwa z[k]≥y−k +1. Namun, karena kita tidak memiliki informasi setelah posisi y, kita harus membandingkan substring karakter dengan karakter awal pada posisi y−k+1 dan y+1. Misalnya, pada Gambar.14.9, kita mengetahui bahwa z[6]≥2. Kemudian, karena s[2] =s[8], ternyata sebenarnya z[6]=2.



Gambar 14.8 Skenario 2: Menghitung nilai z[4]



Gambar 14.9 Skenario 3: Menghitung nilai z[6]

Algoritma yang dihasilkan bekerja dalam waktu O(n), karena selalu ketika dua karakter cocok ketika membandingkan substring karakter dengan karakter, nilai y meningkat. Jadi, total kerja yang dibutuhkan untuk membandingkan substring hanya O(n).

14.3.2 Aplikasi

Algoritma Z menyediakan cara alternatif untuk menyelesaikan banyak masalah string yang juga dapat diselesaikan menggunakan hashing. Namun, tidak seperti hashing, algoritma Z selalu berfungsi dan tidak ada risiko tabrakan. Dalam praktiknya, sering kali muncul rasa apakah akan menggunakan hashing atau algoritma Z.

Pencocokan Pola

Pertimbangkan lagi masalah pencocokan pola, di mana tugas kita adalah menemukan kemunculan pola p dalam string s. Kami telah memecahkan masalah menggunakan hashing, tetapi sekarang kita akan melihat bagaimana algoritma Z menangani masalah tersebut. Pemrosesan string adalah ide saat ini untuk membangun string yang terdiri dari beberapa bagian individu yang dipisahkan oleh karakter khusus. Dalam masalah ini, kita dapat membuat string p#s, di mana p dan s dipisahkan oleh karakter khusus # yang tidak muncul dalam string. Kemudian, Z-array dari p#s memberitahu kita posisi di mana p terjadi di s, karena posisi tersebut mengandung panjang p.

0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C	#	A	B	C	A	B	A	B	C	A
–	0	0	0	3	0	0	2	0	3	0	0	1

Gambar 14.10 Pencocokan pola menggunakan algoritma-Z

0	1	2	3	4	5	6	7	8	9	10
A	B	A	C	A	B	A	C	A	B	A
–	0	1	0	7	0	1	0	3	0	1

Gambar 14.11 Menemukan perbatasan menggunakan algoritma-Z

0	1	2	3	4	5	6	7
2	6	0	3	7	1	5	4

Gambar 14.12 Array sufiks dari string ABAACBAB

0	2	AACBAB
1	6	AB
2	0	ABAACBAB
3	3	ACBAB
4	7	B
5	1	BAACBAB
6	5	BAB
7	4	CBAB

Gambar 14.13 Cara lain untuk merepresentasikan array sufiks

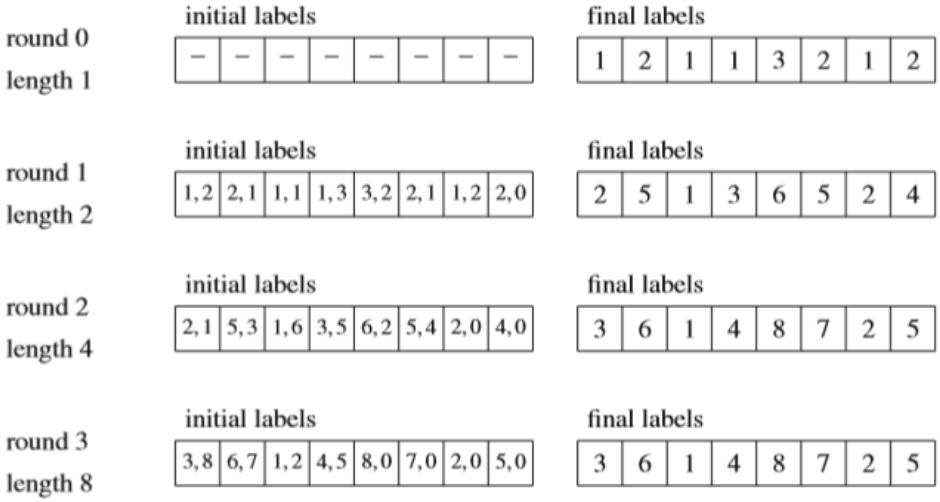
Gambar 14.10 menunjukkan Z-array untuk s =ABCABABCA dan p =ABC. Posisi 4 dan 9 mengandung nilai 3, yang berarti p terjadi pada posisi 0 dan 5 dalam s.

Finding Borders

Border adalah string yang merupakan awalan dan akhiran, tetapi bukan seluruh string. Misalnya, perbatasan ABACABACABA adalah A, ABA, dan ABACABA. Semua batas dari sebuah string dapat ditemukan secara efisien dengan menggunakan algoritma Z, karena sufiks pada posisi k adalah sebuah perbatasan tepat ketika k+z[k]=n di mana n adalah panjang string. Misalnya, pada Gambar 14.11,4+z[4]=11, yang berarti bahwa ABACABA adalah batas dari string.

14.4 Himpunan Akhir *(Array Suffix)*

Array sufiks dari sebuah string menggambarkan urutan leksikografis dari sufiksnya. Setiap nilai dalam array sufiks adalah posisi awal sufiks. Sebagai contoh, Gambar 14.12 menunjukkan array sufiks dari string ABAACBAB. Seringkali lebih mudah untuk merepresentasikan array sufiks secara vertikal dan juga menunjukkan sufiks yang sesuai (Gambar 14.13). Namun, perhatikan bahwa array sufiks itu sendiri hanya berisi posisi awal sufiks dan bukan karakternya.



Gambar 14.14 Membuat label untuk string ABAACBAB

14.4.1 Metode Penggandaan Awalan

Cara sederhana dan efisien untuk membuat larik sufiks dari sebuah string adalah dengan menggunakan konstruksi penggandaan awalan, yang bekerja dalam waktu O(n $\log^2$ n) atau O(n log n), tergantung pada implementasinya.[18] Algoritma ini terdiri dari ronde bernomor 0, 1, ..., log2 n , dan putaran i melewati substring yang panjangnya 2^i. Selama suatu putaran, setiap substring x dengan panjang 2*i* diberi label bilangan bulat(x) sedemikian sehinggal(a) =l(b) tepat ketika a =b dan l(a)<l(b) tepat ketika a < b. Pada putaran 0, setiap substring hanya terdiri dari satu karakter, dan kita dapat, misalnya, menggunakan label A = 1, B = 2,dansoon. Kemudian, pada putaran, di mana i > 0, kita menggunakan label untuk substring dengan panjang2i−1 untuk membuat label untuk substring dengan panjang2i. Untuk memberikan label panjangl(x)untuk substring x dari panjang 2_i,dibagi menjadi dua substring x panjang 2_i adalah l(a) dan l(b). (Jika paruh kedua dimulai di luar string, kita asumsikan bahwa labelnya adalah 0.) Pertama, kita memberi x label awal yang merupakan pasangan (l(a), l(b)).

Kemudian, setelah semua substring dengan panjang 2_i diberi label awal, kita sortir label awal pemberian label adalah agar setelah putaran terakhir, setiap substring memiliki label unik, dan label tersebut menunjukkan urutan leksikografis dari substring tersebut. Kemudian, kita dapat dengan mudah membangun array sufiks berdasarkan label. Gambar 14.14 menunjukkan konstruksi label untuk ABAACBAB. Misalnya, setelah putaran 1, kita mengetahui bahwa l(AB) =2 dan l(AA) =1. Kemudian, pada putaran 2, label awal untuk ABAA adalah (2,1). Karena ada dua label awal yang lebih kecil ((1,6) dan (2,0)), label akhir adalah l(ABAA) =3. Perhatikan bahwa dalam contoh

[18] Gagasan penggandaan awalan adalah karena Karp, Miller, dan Rosenberg. Ada juga algoritma waktu O(n) yang lebih maju untuk membangun array sufiks; Kärkkäinen and Sanders menyediakan algoritma yang cukup sederhana.

ini, setiap label sudah unik setelah putaran2, karena empat karakter pertama dari substring benar-benar menentukan urutan leksikografisnya.

Gambar 14.15 Menemukan kemunculan BA dalam ABAACBAB menggunakan array sufiks

Algoritma yang dihasilkan bekerja dalam waktu O(n $\log^2$ n), karena ada O(log n) putaran dan kami mengurutkan daftar n pasangan pada setiap putaran. Bahkan, implementasi O(n log n) juga dimungkinkan, karena kita dapat menggunakan algoritma pengurutan waktu-linear untuk mengurutkan pasangan. Namun, implementasi waktu O(n $\log^2$ n) langsung hanya dengan menggunakan fungsi sortir C++ biasanya cukup efisien.

14.2.2 Menemukan Pola

Setelah membangun array sufiks, kita dapat secara efisien menemukan kemunculan setiap pola yang diberikan dalam string. Ini dapat dilakukan dalam waktu O(k log n), di mana adalah panjang string dan k adalah panjang pola. Idenya adalah untuk memproses karakter pola demi karakter dan mempertahankan rentang dalam array sufiks yang sesuai dengan awalan pola yang diproses sejauh ini. Dengan menggunakan pencarian biner, kita dapat secara efisien memperbarui rentang setelah setiap karakter baru.

Sebagai contoh, pertimbangkan untuk menemukan kemunculan pola BA dalam string ABAACBAB (Gambar 14.15). Pertama, rentang pencarian kami adalah [0,7], yang mencakup seluruh array akhiran. Kemudian, setelah memproses karakter B, rentang menjadi [4,6]. Akhirnya, setelah memproses karakter A, rentang menjadi [5,6]. Dengan demikian, kami menyimpulkan bahwa BA memiliki dua kemunculan di ABAACBAB di posisi 1 dan 5. Dibandingkan dengan hashing string dan algoritma Z yang dibahas sebelumnya, keuntungan dari array sufiks adalah bahwa kami dapat memproses beberapa kueri yang terkait dengan pola yang berbeda secara efisien, dan tidak perlu mengetahui pola terlebih dahulu saat membangun array sufiks.

14.4.3 Himpunan LCP

Larik LCP dari sebuah string memberikan nilai LCP untuk setiap sufiksnya: panjang dari awalan umum terpanjang dari sufiks dan akhiran berikutnya dalam larik akhiran. Gambar 14.16 menunjukkan larik LCP untuk string ABAACBAB. Misalnya, nilai LCP dari akhiran BAACBAB adalah 2, karena awalan umum terpanjang dari BAACBAB dan BAB adalah BA. Perhatikan bahwa sufiks terakhir dalam larik sufiks tidak memiliki nilai LCP.

0	1	AACBAB
1	2	AB
2	1	ABAACBAB
3	0	ACBAB
4	1	B
5	2	BAACBAB
6	0	BAB
7	–	CBAB

Gambar 14.16 Array LCP dari string ABAACBAB

Selanjutnya kami menyajikan algoritma yang efisien, karena Kasai et al, untuk membangun larik LCP dari sebuah string, asalkan kita telah membangun larik sufiksnya. Algoritma ini didasarkan pada pengamatan berikut: Pertimbangkan sufiks yang nilai LCPnya adalah x. Jika kita menghilangkan karakter pertama dari sufiks dan mendapatkan sufiks lain, kita segera mengetahui bahwa nilai LCP-nya harus paling sedikit x 1. Sebagai contoh, pada Gambar 14.16, nilai LCP dari sufiks BAACBAB adalah 2, jadi kita tahu bahwa nilai LCP dari sufiks AACBAB harus paling sedikit 1. Faktanya, itu terjadi tepat 1.

Kita dapat menggunakan pengamatan di atas untuk secara efisien membangun LCP dengan menghitung nilai LCP dalam urutan penurunan panjang sufiks. Pada setiap sufiks, kita menghitung nilai LCPnya dengan membandingkan sufiks dan sufiks berikutnya dalam larik sufiks karakter demi karakter. Sekarang kita dapat menggunakan fakta bahwa kita mengetahui nilai LCP dari sufiks yang memiliki satu karakter lagi. Jadi, nilai LCP saat ini harus paling sedikit x 1, di mana x adalah nilai LCP sebelumnya, dan kita tidak perlu membandingkan karakter x 1 pertama dari sufiks. Ada algoritma yang bekerja dalam waktu O(n), karena hanya perbandingan O(n). dilakukan selama algoritma.

Dengan menggunakan larik LCP, kita dapat menyelesaikan beberapa masalah string tingkat lanjut secara efisien. Misalnya, untuk menghitung jumlah substring yang berbeda dalam string, kita cukup mengurangi jumlah semua nilai dalam larik LCP dari jumlah total substring, yaitu, jawaban untuk masalah ini adalah

$$\frac{n(n+1)}{2}c$$

di mana n adalah panjang string dan c adalah jumlah semua nilai dalam larik LCP. Misalnya, string ABAACBAB memiliki substring yang berbeda.

$$\frac{8\,.\,9}{2} - 7 = 29$$

BAB 15
TOPIK TAMBAHAN

15.1 Teknik Akar Kuadrat

Akar kuadrat dapat dilihat sebagai "logaritma orang miskin": kompleksitas O(√n) lebih baik daripada O(n) tetapi lebih buruk daripada O(log n). Bagaimanapun, banyak struktur data dan algoritma yang melibatkan akar kuadrat yang cepat dan dapat digunakan dalam praktik. Bagian ini menunjukkan beberapa contoh bagaimana akar kuadrat dapat digunakan dalam desain algoritma.

15.1.1 Struktur data

Kadang-kadang kita dapat membuat struktur data yang efisien dengan membagi array menjadi blok-blok berukuran n dan memelihara informasi tentang nilai array di dalam setiap blok. Misalnya, anggaplah kita harus memproses dua jenis kueri: memodifikasi nilai array dan menemukan nilai minimum dalam rentang. Sebelumnya kita telah melihat bahwa pohon segmen dapat mendukung kedua operasi dalam waktu O(log n), tetapi selanjutnya kita akan menyelesaikan masalah dengan cara lain yang lebih sederhana di mana operasi membutuhkan **waktu O(√n).**

Kami membagi array menjadi blok-blok dari n elemen, dan mempertahankan untuk setiap blok nilai minimum di dalamnya. Sebagai contoh, Gambar 15.1 menunjukkan larik 16 elemen yang dibagi menjadi blok 4 elemen. Ketika nilai array berubah, blok yang sesuai perlu diperbarui. Ini dapat dilakukan dalam waktu O(√n) dengan menelusuri nilai-nilai di dalam blok, seperti yang ditunjukkan pada Gambar 15.2. Kemudian, untuk menghitung nilai minimum dalam suatu rentang, kita membagi rentang menjadi tiga bagian sehingga rentang terdiri dari nilai tunggal dan blok di antaranya. Gambar 15.3 menunjukkan contoh pembagian seperti itu. Jawaban atas kueri adalah satu nilai atau nilai minimum di dalam blok. Karena jumlah elemen tunggal adalah O(√n) dan jumlah blok juga O(√n), kueri membutuhkan waktu O(√n).

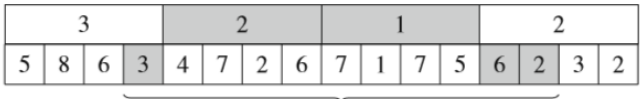
Seberapa efisien struktur yang dihasilkan dalam praktik? Untuk mengetahui hal ini, kami melakukan eksperimen di mana kami membuat danarray dari nilai-nilai acak dan kemudian memproses n kueri minimum acak. Kami menerapkan tiga struktur data: pohon segmen dengan kueri waktu O(log n), struktur akar kuadrat yang dijelaskan di atas dengan kueri waktu O(√n), dan array biasa dengan kueri waktu O(n). Tabel15.1 menunjukkan hasil percobaan. Ternyata pada soal ini, struktur akar kuadrat cukup efisien hingga n =2^{18}; namun, setelah ini, jelas membutuhkan lebih banyak waktu daripada pohon segmen.

3				2				1				2			
5	8	6	3	4	7	2	6	7	1	7	5	6	2	3	2

Gambar 15.1 Struktur akar kuadrat untuk menemukan nilai minimum dalam rentang

3				4				1				2			
5	8	6	3	4	7	5	6	7	1	7	5	6	2	3	2

Gambar 15.2 Ketika nilai array diperbarui, nilai di blok yang sesuai juga harus diperbarui



Gambar 15.3 Untuk menentukan nilai minimum dalam suatu rentang, rentang dibagi menjadi nilai tunggal dan blok

Tabel 15.1 Waktu berjalan dari tiga struktur data untuk kueri minimum rentang: pohon segmen (O(log n)), struktur akar kuadrat (O(√n)), dan array biasa (O(n))

Ukuran input *n*	O(log n) kueri (s)	O(√n) Kueri (s)	O(n) Kueri (s)
2^{16}	0.02	0.05	1.50
2^{17}	0.03	0.16	6.05
2^{18}	0.07	0.28	24.82
2^{19}	0.14	1.14	>60
2^{20}	0.31	2.11	>60
2^{21}	0.66	9.27	>60

A	C	E	A
B	D	F	D
E	A	B	C
C	F	E	A

Gambar 15.4 Contoh masalah jarak huruf

15.1.2 Sub-Algoritma

Selanjutnya kita membahas dua masalah yang dapat diselesaikan secara efisien dengan membuat dua subalgoritma yang terspesialisasi untuk berbagai jenis situasi selama algoritma. Sementara salah satu dari subalgoritma dapat digunakan untuk menyelesaikan masalah tanpa yang lain, kami mendapatkan algoritme yang efisien dengan menggabungkannya.

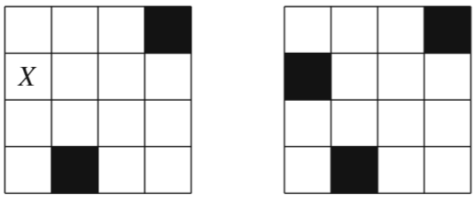
Jarak Surat

Masalah pertama kita adalah sebagai berikut: Kita diberikan sebuah kisi nxn yang setiap perseginya diberi sebuah huruf. Berapa jarak minimum Manhattan antara dua kotak yang berhuruf sama? Misalnya, pada Gambar 15.4 jarak minimum adalah 2 antara dua kotak dengan huruf "D". Untuk mengatasi masalah tersebut, kita dapat menelusuri semua huruf yang muncul dalam kisi, dan untuk setiap huruf c, tentukan jarak minimum antara dua kotak dengan huruf c. Pertimbangkan dua algoritma untuk memproses huruf tetap c:

*Algoritma1:*Melalui semua pasangan kuadrat yang berisi huruf dan menentukan pasangan jarak minimum di antara mereka. Algoritma ini bekerja dalam waktu $O(k^2)$, di mana k adalah jumlah kuadrat dengan huruf c.

Algoritma 2: Lakukan pencarian luas pertama yang secara bersamaan dimulai pada setiap kotak dengan huruf c. Pencarian membutuhkan waktu $O(n^2)$. Kedua algoritma memiliki situasi kasus terburuk tertentu. Kasus terburuk untuk Algoritma 1 adalah grid dimana setiap persegi memiliki warna yang sama, dalam hal ini k = n2 dan algoritma membutuhkan waktu $O(n^4)$. Kemudian, kasus terburuk untuk Algoritma 2 adalah grid

dimana setiap persegi memiliki warna yang berbeda. Dalam hal ini, algoritma tersebut salah melakukan $O(n^2)$ kali, yang membutuhkan waktu $O(n^4)$.



Gambar 15.5 Sebuah giliran dalam permainan kotak hitam.
Jarak minimum dari X ke kotak hitam adalah 3

Namun, kita dapat menggabungkan algoritma sehingga berfungsi sebagai subalgoritma dari satu algoritma. Idenya adalah untuk memutuskan setiap warna c secara terpisah algoritma mana yang akan digunakan. Jelas, Algoritma 1 bekerja dengan baik jika k kecil, dan Algoritma 2 paling cocok untuk kasus di mana k besar. Jadi, kita dapat memperbaiki konstanta x dan menggunakan Algoritma 1 jika k paling banyak x, dan sebaliknya menggunakan Algoritma 2. Khususnya, dengan memilih x $=\sqrt{n^2}$ =n,algoritma wegetyang bekerja dalam waktu $O(n^3)$. n^3) waktu. Kemudian, karena terdapat paling banyak n warna yang muncul di lebih dari n kotak, Algoritma 2 dijalankan paling banyak n kali, dan total waktu berjalannya juga $O(n^3)$.

Kotak Hitam

Sebagai contoh lain, perhatikan permainan berikut: Kita diberikan sebuah kotak n×n di mana tepat satu persegi berwarna hitam dan semua persegi lainnya berwarna putih. Setiap belokan, satu kotak putih dipilih, dan kita harus menghitung jarak Manhattan minimum antara kotak ini dan kotak hitam. Setelah ini, kotak putih dicat hitam. Proses ini berlanjut selama n2 1 putaran, setelah itu semua kotak dicat hitam. Misalnya, Gambar 15.5 menunjukkan satu-satunya permainan. Jarak minimum dari kotak X yang dipilih ke kotak hitam adalah 3 (dengan dua langkah ke bawah dan melakukan langkah ke kanan). Setelah ini, kotak dicat hitam.

Kita dapat memecahkan masalah dengan memproses putaran dalam kumpulan k putaran. Sebelum setiap batch, kami menghitung untuk setiap kotak kotak jarak minimum ke kotak hitam. Ini dapat dilakukan dalam waktu $O(n^2)$ dengan menggunakan pencarian pertama yang luas. Kemudian, saat memproses batch, kami menyimpan daftar semua kotak yang telah dicat hitam selama batch saat ini. Jadi, jarak minimum ke kotak hitam adalah jarak yang telah dihitung sebelumnya atau jarak ke salah satu kotak dalam daftar. Karena daftar berisi paling banyak nilai k, dibutuhkan waktu O(k) untuk menelusuri daftar.

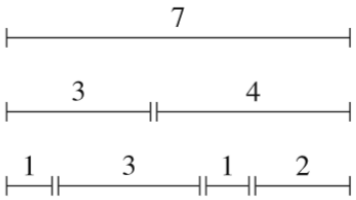
Kemudian, dengan memilih k $=\sqrt{n2}$ =n, kita mendapatkan algoritma yang bekerja dalam waktu $O(n^3)$. Pertama, ada O(n)batch, sehingga total waktu yang digunakan untuk penelitian pertama keluasan $O(n^3)$. Kemudian, daftar kotak dalam batch berisi nilai O(n), jadi menghitung jarak minimum untuk kotak $O(n^2)$ juga membutuhkan waktu $O(n^3)$.

Parameter Penyetelan

Dalam praktiknya, tidak perlu menggunakan nilai akar kuadrat yang tepat sebagai parameter, tetapi kita dapat menyempurnakan kinerja suatu algoritma dengan bereksperimen dengan parameter yang berbeda dan memilih parameter yang paling sesuai. Tentu saja, parameter optimal tergantung pada algoritme dan juga pada properti dari data uji.

Tabelv15.2 Optimalisasi nilai parameter k pada algoritma kotak hitam

Parameter *k*	Running time (s)
200	5.74
500	2.41
1000	1.32
2000	1.02
5000	1.28
10000	2.13
20000	3.97



Gambar 15.6 Beberapa partisi bilangan bulat dari tongkat dengan panjang 7

Tabel 15.2 menunjukkan hasil dari eksperimen di mana algoritma waktu $O(n^3)$untuk permainan kotak hitam dilakukan untuk nilai k yang berbeda ketika n =500. Urutan kotak yang dicat hitam dipilih secara acak. Dalam hal ini, parameter optimal tampaknya sekitar k =2000.

15.1.3 Partisi Bilangan Bulat

Misalkan ada sebuah tongkat yang panjangnya n, dan dibagi menjadi beberapa bagian yang panjangnya adalah bilangan bulat. Sebagai contoh, Gambar.15.6 menunjukkan beberapa kemungkinan partisi untuk n =7. Berapa jumlah maksimum panjang yang berbeda dalam partisi seperti itu? Ternyata ada paling banyak O(√n) panjang yang berbeda. Yaitu cara yang optimal untuk menghasilkan banyak panjang yang berbeda yang mungkin adalah dengan memasukkan panjang 1, 2, ..., k. Kemudian, sejak

$$1 + 2 + \ldots + k = \frac{k(k+1)}{2}$$

kita dapat menyimpulkan bahwa k dapat mengalahkan sebagian besar O(√n). Selanjutnya, kita akan melihat bagaimana pengamatan ini dapat digunakan saat merancang algoritma.

Masalah Ransel

Pertimbangkan masalah ransel di mana kita diberikan daftar bobot bilangan bulat [w1,w2,...,wk] sedemikian rupa sehingga $w_1 + w_2 + \cdots + w_k = n$, dan tugas kita adalah menentukan semua kemungkinan jumlah bobot yang dapat dibuat. Misalnya, Gambar.15.7 menunjukkan jumlah yang mungkin dengan menggunakan bobot[3,3,4].

0	1	2	3	4	5	6	7	8	9	10
✓			✓	✓		✓	✓			✓

Gambar 15.7 Jumlah yang mungkin menggunakan bobot[3,3,4]

Dengan menggunakan algoritma knapsack standar kita dapat menyelesaikan masalah dalam waktu $O(n_k)$, jadi jika k = O(n), kompleksitas waktu menjadi $O(n_2)$. Namun,

karena terdapat paling banyak O(√n) bobot yang berbeda, kita sebenarnya dapat menyelesaikan masalah dengan lebih efisien dengan memproses semua bobot secara bersamaan dengan nilai tertentu. Misalnya, jika bobotnya adalah [3,3,4], pertama-tama kita memproses dua bobot dari nilai 3 dan kemudian bobot nilai 4. Tidak sulit untuk memodifikasi algoritma knapsack standar sehingga memproses setiap kelompok dengan bobot yang sama hanya membutuhkan waktu O(n), yang menghasilkan algoritma waktu O(n√n).

Konstruksi String

Sebagai contoh lain, misalkan kita diberikan string dengan panjang n dan kamus kata-kata yang panjang totalnya adalah m. Tugas kita adalah menghitung jumlah cara kita dapat membuat string menggunakan kata-kata. Misalnya, ada empat cara untuk membuat string ABAB menggunakan kata-kata{A,B,AB}:

- A+B+A+B
- AB+A+B
- A+B+AB
- AB+AB

Dengan menggunakan pemrograman dinamis, kita dapat menghitung untuk setiap k =0,1,...,n jumlah cara untuk membuat awalan dengan panjang k dari string. Salah satu cara untuk melakukannya adalah dengan menggunakan trie yang berisi kebalikan dari semua kata dalam kamus, yang menghasilkan algoritma waktu $O(n_2 +m)$. Namun, pendekatan lain adalah dengan menggunakan hashing string dan fakta bahwa ada paling banyak O(√m) panjang kata yang berbeda. Dengan demikian, kita dapat membatasi diri pada panjang kata yang benar-benar ada. Ini dapat dilakukan dengan membuat set yang berisi semua nilai hash dari kata, yang menghasilkan algoritma yang waktu berjalannya adalah O(n√m+m) (menggunakan unordered_set).

15.1.4 Algoritma Mo

Algoritme Mo memproses kumpulan kueri rentang pada array statis (yaitu, nilai-nilai array tidak berubah di antara kueri). Setiap kueri mengharuskan kita menghitung sesuatu berdasarkan nilai larik dalam rentang[a,b]. Karena array statis, kueri dapat diproses dalam urutan apa pun, dan trik dalam algoritma Mo adalah menggunakan urutan khusus yang menjamin bahwa algoritma bekerja secara efisien. Algoritme mempertahankan rentang aktif dalam larik, dan jawaban atas pertanyaan tentang rentang aktif diketahui setiap saat. Algoritme memproses kueri satu per satu dan selalu memindahkan titik akhir dari rentang aktif dengan memasukkan dan menghapus elemen. Array dibagi menjadi blok elemen k = O(√n), dan kueri$[a_1,b_1]$selalu diproses sebelum kueri $[a_2, b_2]$ jika

- $[a_1/k]<[a_2/k]$ atau
- $[a_1/k]=[a_2/k]$ dan $b_1<b_2$

Jadi, semua pertanyaan yang titik akhir kirinya berada dalam blok tertentu diproses satu demi satu diurutkan menurut titik akhir kanannya. Dengan menggunakan urutan ini, algoritma hanya melakukan operasi O(n√n), karena titik akhir kiri bergerak O(n) kali O(√n) langkah, dan titik akhir kanan bergerak O(√n) kali O(n) langkah . Jadi, kedua titik akhir memindahkan total langkah O(n√n) selama algoritma.

Contoh

Pertimbangkan masalah di mana kita diberikan satu set rentang array, dan kita diminta untuk menghitung jumlah nilai yang berbeda di setiap rentang. Dalam algoritma Mo,

kueri selalu diurutkan dengan cara yang sama, tetapi cara jawaban kueri dipertahankan tergantung pada masalahnya.

Untuk memecahkan masalah, kami mempertahankan jumlah array di mana count[x] menunjukkan berapa kali elemen x muncul dalam rentang aktif. Saat kita berpindah dari satu kueri ke kueri lain, rentang aktif berubah. Ketika kita berpindah dari rentang pertama ke rentang kedua, akan ada tiga langkah: titik akhir kiri bergerak satu langkah ke kanan, dan titik akhir kanan bergerak dua langkah ke kanan. Setelah setiap langkah, jumlah larik perlu diperbarui. Setelah menambahkan elemen x, kami meningkatkan nilai count[x] sebesar 1, dan jika count[x]=1 setelah ini, kami juga meningkatkan jawaban kueri sebesar 1. Demikian pula, setelah menghapus elemen x, kami mengurangi nilai count[x] sebesar 1, dan jika count[x]=0 setelah ini, kami juga mengurangi jawaban kueri sebesar 1. Karena setiap langkah memerlukan waktu O(1), algoritma bekerja dalam O(n√ n) waktu.

15.2 Tinjauan Ulang Pohon Segmen

Pohon segmen adalah struktur data serbaguna yang dapat digunakan untuk memecahkan sejumlah besar masalah. Namun, sejauh ini kita hanya melihat sebagian kecil dari kemungkinan pohon segmen. Sekarang saatnya untuk membahas beberapa varian lanjutan dari pohon segmen yang memungkinkan kita untuk memecahkan masalah yang lebih lanjut.

Sampai saat ini, kami telah menerapkan operasi pohon segmen dengan berjalan dari bawah ke atas di pohon. Misalnya, kami telah menggunakan fungsi berikut untuk menghitung jumlah nilai dalam rentang[a,b]:

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

Namun, pada pohon segmen lanjutan, seringkali perlu untuk menerapkan operasi dari atas ke bawah sebagai berikut:

```
int sum(int a, int b, int k, int x, int y) {
    if (b < x || a > y)return 0;
    if (a <= x && y <= b)return tree[k];
    int d = (x+y)/2;
    return sum(a,b,2*k,x,d) + sum(a,b,2*k+1,d+1,y);
}
```

Dengan menggunakan fungsi ini, kita dapat menghitung jumlah dalam rentang [a,b] sebagai berikut:

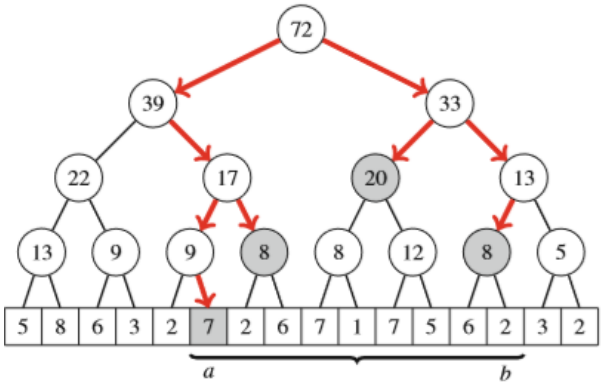
```
int s = sum(a,b,1,0,n-1);
```

Parameter k menunjukkan posisi saat ini di pohon. Awalnya k sama dengan 1, karena kita mulai dari akar pohon. Rentang [x, y] sesuai dengan k dan awalnya [0,n−1]. Saat menghitung jumlah, jika[x, y]di luar[a,b], jumlahnya adalah 0, dan jika[x,y]berada di dalam[a,b], jumlah dapat ditemukan di pohon. Jika[x,y] sebagian berada di dalam[a,b], pencarian dilanjutkan

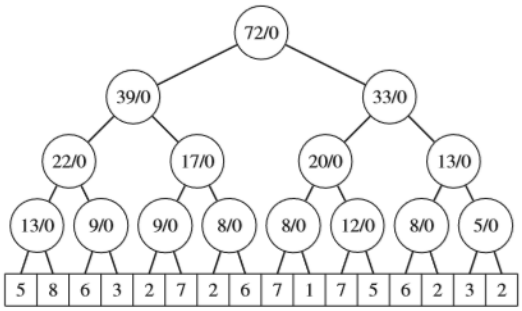
secara rekursif ke bagian kiri dan kanan [x,y]. Separuh kiri adalah[x,d], dan separuh kanan adalah [d+1, y], di mana $d = \frac{x+y}{2}$. Gambar 15.9 menunjukkan bagaimana pencarian berlangsung saat menghitung nilai sum_q(a,b). Node abu-abu menunjukkan node di mana rekursi berhenti dan jumlahnya dapat ditemukan di pohon. Juga dalam implementasi ini, operasi membutuhkan waktu O(log n), karena jumlah total node yang dikunjungi adalah O(log n).

15.2.1 Perambatan lambat

Dengan menggunakan perambatan lambat, kita dapat membangun pohon segmen yang mendukung pembaruan rentang dan kueri rentang dalam waktu O(log n). Ide untuk memperbarui peforma dan kueri dari atas ke bawah dan melakukan pembaruan dengan malas sehingga disebarkan ke bawah pohon hanya jika diperlukan.



Gambar 15.9 Melintasi pohon segmen dari atas ke bawah

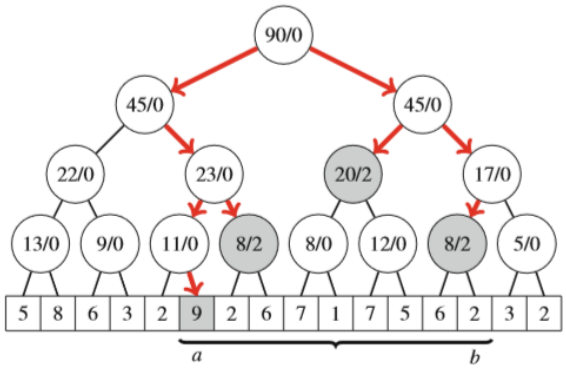


Gambar 15.10 Pohon segmen malas untuk pembaruan rentang dan kueri

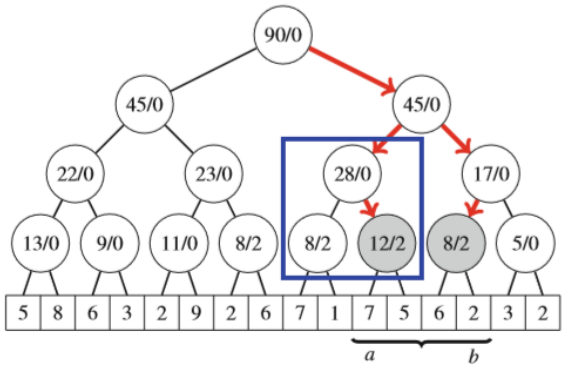
Node dari pohon segmen malas berisi dua jenis informasi. Seperti di pohon segmen biasa, setiap node berisi jumlah, nilai minimum, atau nilai lain yang terkait dengan subbarray yang sesuai. Selain itu, sebuah ode mungkin berisi informasi tentang pembaruan lambat yang belum disebarkan ke anak-anaknya. Pohon segmen malas dapat mendukung dua jenis pembaruan rentang: setiap nilai larik dalam rentang ditingkatkan dengan beberapa nilai atau diberi beberapa nilai. Kedua operasi tersebut dapat diimplementasikan menggunakan ide yang sama, dan bahkan dimungkinkan untuk membangun sebuah pohon yang mendukung kedua operasi tersebut pada saat yang bersamaan.

Mari kita perhatikan contoh di mana tujuan kita adalah membuat pohon segmen yang mendukung dua operasi: meningkatkan setiap nilai dalam[a,b]dengan konstan dan menghitung jumlah nilai dalam[a,b]. Untuk mencapai tujuan ini, kita membuat pohon di mana setiap simpul memiliki dua nilai/z:s menunjukkan jumlah nilai dalam rentang, dan z menunjukkan nilai pembaruan yang lambat, yang berarti bahwa semua nilai

dalam rentang harus ditingkatkan sebesar z. Gambar 15.10 menunjukkan contoh pohon seperti itu, di mana z =0 di semua node, artinya tidak ada pembaruan lambat yang sedang berlangsung.



Gambar 15.11 Meningkatkan nilai dalam rentang[a,b]dengan 2



Gambar 15.12 Menghitung jumlah nilai dalam rentang[a,b]

Kami menerapkan operasi pohon dari atas ke bawah. Untuk meningkatkan nilai dalam suatu range[a,b]oleh u, kita memodifikasi node sebagai berikut: Jika range[x, y]dari sebuah node sepenuhnya berada di dalam[a,b], kita meningkatkan nilai z dari node tersebut olehmu dan berhenti. Kemudian, jika [x, y]sebagian milik [a,b], kami melanjutkan perjalanan kami secara rekursif di pohon, dan setelah ini menghitung nilai s baru untuk node. Sebagai contoh, Gambar 15.11 menunjukkan pohon kita setelah meningkatkan rentang[a,b] sebanyak 2. Dalam pembaruan dan kueri, pembaruan malas disebarkan ke bawah saat kita bergerak di dalam pohon. Selalu sebelum mengakses sebuah node, kami memeriksa apakah node tersebut memiliki pembaruan lambat yang sedang berlangsung. Jika ya, kami memperbarui nilai s-nya, menyebarkan pembaruan ke turunannya, dan kemudian menghapus nilai z-nya. Sebagai contoh, Gambar 15.12 menunjukkan bagaimana pohon kita berubah ketika kita menghitung nilai suma(a,b). Persegi panjang berisi node yang nilainya berubah ketika pembaruan lambat disebarkan ke bawah.

Pembaruan Polinomial

Kita dapat menggeneralisasi pohon segmen di atas sehingga memungkinkan untuk memperbarui rentang menggunakan polinomial dari bentuk

$$p(u) = t_k u^k + t_{k-1} u^{k-1} + \cdots + t_0.$$

Dalam hal ini, pembaruan untuk nilai pada positioni di[a,b]adalah p(i a). Misalnya, menambahkan polinomial p(u) =u+1 ke[a,b] berarti nilai pada posisi a bertambah 1, nilai pada posisi a+1 bertambah 2, dan seterusnya. Untuk mendukung pembaruan

polinomial, setiap simpul diberi nilai k+2, di mana k sama dengan derajat polinomial. Nilai s adalah jumlah elemen dalam rentang, dan nilai $z_0,z_1,...,z_k$ adalah koefisien polinomial koefisien yang sesuai dengan pembaruan lambat. Sekarang, jumlah nilai dalam rentang[x, y]sama

$$s + \sum_{u=0}^{y-x} (z_k u^k + z_{k-1} u^{k-1} + \ldots + z_1 u + z_0$$

dan nilai jumlah tersebut dapat dihitung secara efisien menggunakan rumus jumlah. Misalnya, suku z_0 sesuai dengan jumlah $z_0(y-x+1)$, dan suku z_{iu} sesuai dengan jumlah

$$z_1(0 + 1 + \ldots + y) = z_1 \frac{(y-x)(y-x+1)}{2}$$

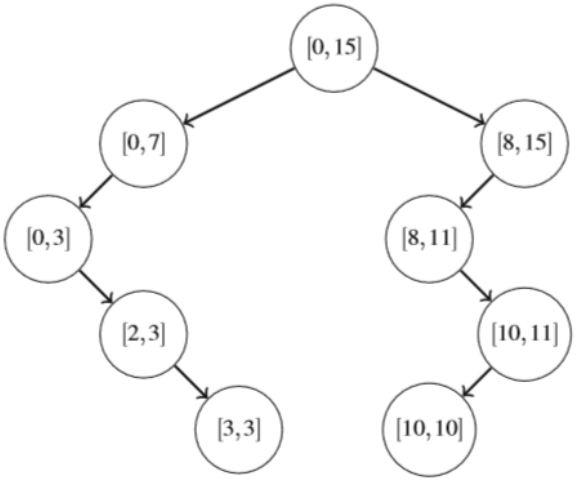
Saat menyebarkan pembaruan di pohon, indeks p(u) berubah, karena di setiap rentang [x, y], nilainya dihitung untuk u =0,1,...,y−x. Namun, kita dapat dengan mudah menangani ini, karena p (u) = p(u+h) adalah polinomialofequaldegreeas p(u). Misalnya, jika $p(u) = t_2u2 + t_1u + t_0$, maka

$$P'(u) = t_2(u+h)^2 + t_1(u+h) + t_0 = t_2u^2 + (2ht_2 + t_1)u + t_2h^2 + t_1h + t_0.$$

15.2.2 Pohon Dinamis

Pohon segmen biasa bersifat statis, yang berarti bahwa setiap simpul memiliki posisi tetap dalam larik pohon segmen dan strukturnya memerlukan jumlah memori yang tetap. Dalam pohon segmen dinamis, memori dialokasikan hanya untuk node yang benar-benar diakses selama algoritma, yang dapat menghemat sejumlah besar memori. Node dari pohon dinamis dapat direpresentasikan sebagai struct:

```
struct node {
int value;
int x, y;
node *left, *right;
node(int v, int x, int y) : value(v), x(x), y(y) {}
};
```



Gambar 15.13 Pohon segmen jarang dimana elemen pada posisi 3 dan 10 telah dimodifikasi

Di sini nilai adalah nilai simpul,[x,y] adalah rentang yang sesuai, dan titik kiri dan kanan ke subpohon kiri dan kanan. Node dapat dibuat sebagai berikut:

```
// create a node with value 2 and range [0,7]
node *x = new node(2,0,7);
```

```
// change value
x->value = 5;
```

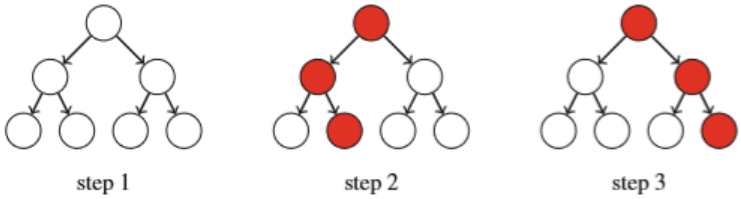
Pohon Segmen Jarang

Pohon segmen dinamis adalah struktur yang berguna ketika larik yang mendasarinya jarang, yaitu, rentang [0,n−1] dari indeks yang diizinkan besar, tetapi sebagian besar nilai larik adalah nol. Sementara pohon segmen biasa akan menggunakan memori O(n), pohon segmen dinamis hanya menggunakan memori O(k log n), di mana k adalah jumlah operasi yang dilakukan.

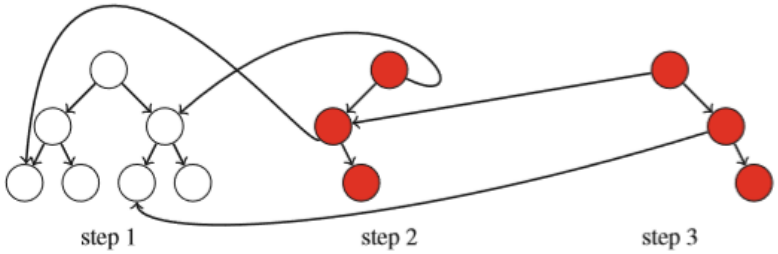
Pohon segmen jarang awalnya hanya memiliki satu simpul[0,n−1]yang nilainya nol, yang berarti bahwa setiap nilai larik adalah nol. Setelah pembaruan, node baru ditambahkan secara dinamis ke pohon. Setiap jalur dari simpul akar ke daun berisi simpul O(log n), jadi setiap operasi pohon segmen menambahkan paling banyak O(log n) simpul baru ke pohon. Jadi, setelah k operasi, pohon berisi simpul O(k log n). Sebagai contoh, Gambar 15.13 menunjukkan pohon segmen jarang di mana n =16, dan elemen pada posisi 3 dan 10 telah dimodifikasi. Perhatikan bahwa jika kita mengetahui semua elemen yang akan diperbarui selama algoritme sebelumnya, pohon segmen dinamis tidak diperlukan, karena kita dapat menggunakan pohon segmen biasa dengan kompresi indeks. Namun, ini tidak mungkin bila indeks dihasilkan selama algoritma.

Pohon Segmen Persisten

Menggunakan implementasi dinamis, kita juga dapat membuat pohon segmen persisten yang menyimpan riwayat modifikasi pohon. Dalam implementasi seperti itu, kita dapat secara efisien mengakses semua versi pohon yang telah ada selama algoritma. Ketika riwayat modifikasi tersedia, kita dapat melakukan kueri di pohon sebelumnya seperti di pohon segmen biasa, karena struktur lengkap setiap pohon disimpan. Kita juga dapat membuat pohon baru berdasarkan pohon sebelumnya dan memodifikasinya secara mandiri.



Gambar 15.14 Riwayat modifikasi pohon segmen: pohon awal dan dua pembaruan



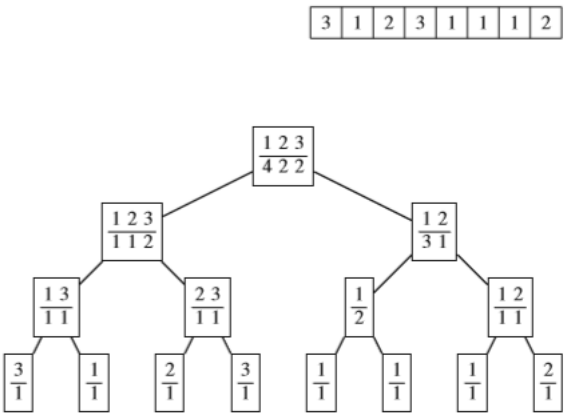
Gambar 15.15 Cara ringkas untuk menyimpan riwayat modifikasi

Pertimbangkan urutan pembaruan pada Gambar 15.14, di mana node yang ditandai berubah dan node lainnya tetap sama. Setelah setiap pembaruan, sebagian besar simpul pohon tetap sama, sehingga cara ringkas untuk menyimpan sejarah modifikasimenyimpan setiap pohon historis sebagai kombinasi simpul baru dan subpohon dari pohon sebelumnya. Gambar 15.15 menunjukkan bagaimana riwayat modifikasi dapat disimpan. Struktur setiap pohon sebelumnya dapat direkonstruksi

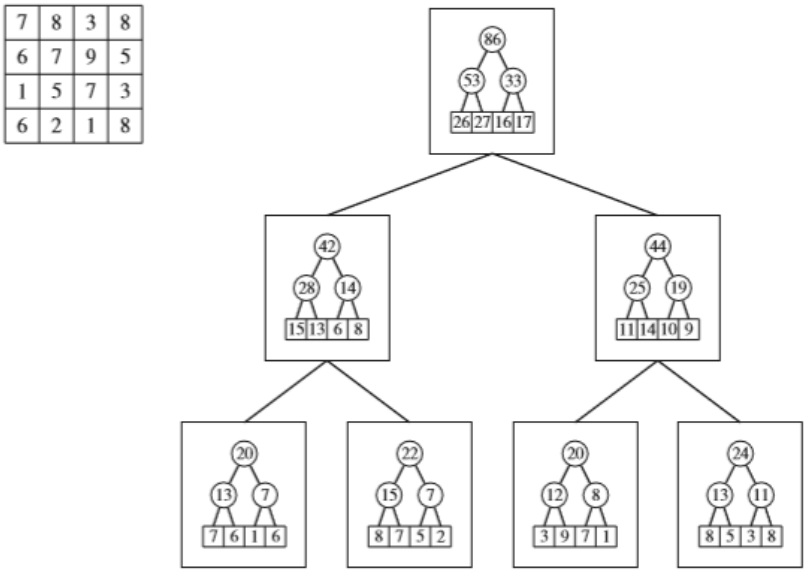
dengan mengikuti petunjuk yang dimulai dari simpul akar yang sesuai. Karena setiap operasi hanya menambahkan O(log n) node baru ke pohon, memungkinkan untuk menyimpan riwayat modifikasi penuh dari pohon.

15.2.3 Struktur Data pada Simpul (Node)

Alih-alih nilai tunggal, node dari pohon segmen juga dapat berisi struktur data yang memelihara informasi tentang rentang yang sesuai. Sebagai contoh, misalkan kita dapat menghitung jumlah kemunculan elemen x secara efisien dalam suatu rentang [a,b]. Untuk melakukan ini, kita dapat membuat pohon segmen di mana setiap node diberi struktur data yang dapat ditanyakan berapa kali elemen x muncul dalam rentang yang sesuai. Setelah ini, jawaban atas kueri dapat dihitung dengan menggabungkan hasil dari node yang termasuk dalam rentang. Tugas yang tersisa adalah memilih struktur data yang sesuai untuk masalah tersebut. Pilihan yang baik adalah struktur Maps yang kuncinya adalah elemen larik dan nilainya menunjukkan berapa kali setiap elemen muncul dalam suatu rentang. Gambar 15.16 menunjukkan larik dan pohon segmen yang sesuai. Misalnya, simpul akar pohon memberi tahu kita bahwa elemen 1 muncul 4 kali dalam larik.



Gambar 15.16 Pohon segmen untuk menghitung jumlah kemunculan elemen dalam rentang array



Gambar 15.17 Array dua dimensi dan pohon segmen yang sesuai untuk menghitung jumlah subarray persegi panjang

Setiap query pada pohon segmen di atas bekerja dalam waktu O($\log^2$ n), karena setiap node memiliki struktur Maps yang operasinya membutuhkan waktu O(log n). Pohon menggunakan memori O(n log n), karena memiliki level O(log n), dan setiap level berisi n elemen yang telah didistribusikan dalam struktur Maps.

15.2.4 Pohon Dua-Dimensi

Pohon segmen dua dimensi memungkinkan kita untuk memproses kueri yang terkait dengan subarray persegi panjang pada array dua dimensi. Idenya adalah membuat pohon segmen yang sesuai dengan kolom larik dan kemudian menetapkan setiap simpul struktur ini pohon segmen yang sesuai dengan baris larik. Misalnya, Gambar 15.17 menunjukkan pohon segmen dua dimensi yang mendukung dua kueri: menghitung jumlah nilai dalam subbarray dan memperbarui nilai array tunggal. Kedua query membutuhkan waktu O($\log^2$ n), karena node O(log n) di pohon segmen utama diakses, dan memproses setiap node membutuhkan waktu O(log n). Struktur ini menggunakan total memori O(n^2), karena pohon segmen utama memiliki O(n) node, dan setiap node memiliki pohon segmen dari O(n) node.

15.3 Makanan (Treaps)

Treap adalah pohon biner yang dapat menyimpan konten array sedemikian rupa sehingga kita dapat membagi array menjadi dua array secara efisien dan menggabungkan dua array ke dalam array. Setiap node dalam treap memiliki dua nilai: bobot dan nilai. Bobot masing-masing node lebih kecil atau sama dengan bobot anak-anaknya, dan kemudian node ditempatkan di dalam sinar setelah semua node di subtree kirinya dan sebelum semua node di subtree kanannya. Gambar15.18 menunjukkan contoh array dan treap yang sesuai. Misalnya, simpul akar memiliki bobot 1 dan nilai D. Karena subpohon kirinya berisi tiga simpul, ini berarti elemen larik pada posisi 3 memiliki nilai D.

15.3.1 Pemisahan dan Penggabungan

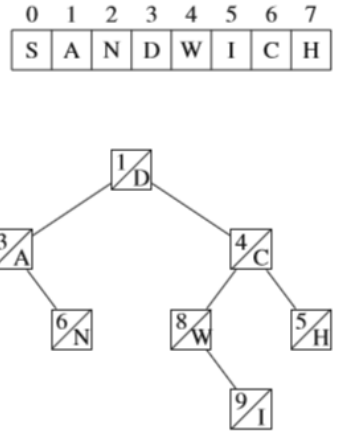
Ketika sebuah node baru ditambahkan ke dalam treap, ia diberi bobot acak. Ini menjamin bahwa pohon itu seimbang (tingginya adalah O(log n)) dengan probabilitas tinggi, dan operasinya dapat dilakukan secara efisien.

Pemisahan

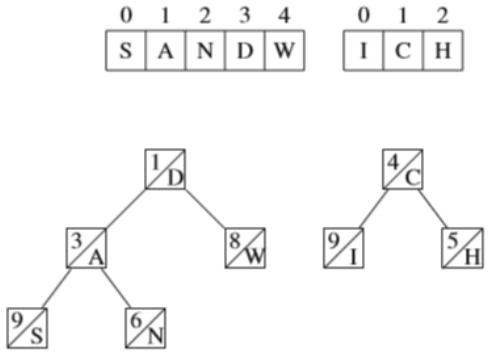
Operasi pemisahan atreap membuat duatreaps yang membagi array menjadi dua array sehingga k elemen pertama termasuk dalam array pertama dan elemen lainnya termasuk dalam array kedua. Untuk melakukan ini, kami membuat dua treap baru yang awalnya kosong dan melintasi treap asli mulai dari node root. Pada setiap langkah, jika node saat ini milik treap kiri, node dan subtree kirinya ditambahkan ke treap kiri dan kami memproses subtree kanannya secara rekursif.

Demikian pula, jika node saat ini termasuk ke dalam treap kanan, maka node dan subtree kanannya dibacakan ke treap kanan dan kita memproses subtree kirinya secara rekursif. Karena ketinggian treap adalah O(log n), operasi ini bekerja dalam waktu O(log n). Sebagai contoh, Gambar 15.19 menunjukkan untuk membuat video contoh array menjadi dua array sehingga sinar bintang pertama berisi lima elemen pertama dari array asli dan array kedua berisi tiga elemen terakhir. Pertama, simpul D termasuk dalam treap kiri, jadi kami menambahkan simpul D dan subpohon kirinya ke treap kiri. Kemudian, simpul C milik treap kanan, dan tambahkan simpul Kanan ke

subpohon kanan ke treap kanan. Terakhir, tambahkan simpul W ke treap kiri dan simpul I ke treap kanan.



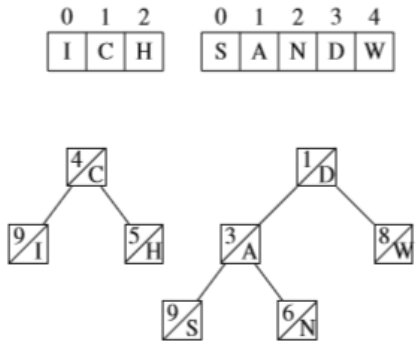
Gambar 15.18 Sebuah array dan treap yang sesuai



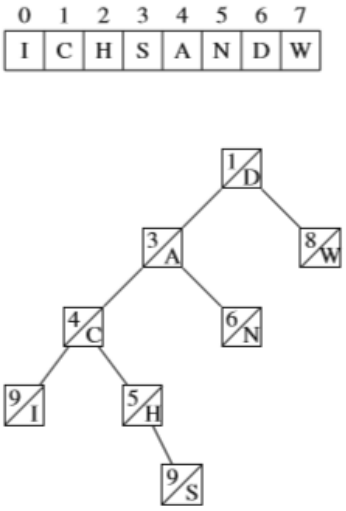
Gambar 15.19 Memisahkan sebuah array menjadi dua array

Penggabungan

Operasi penggabungan dua treap membuat treap tunggal yang menggabungkan array. Kedua treap diproses secara bersamaan, dan pada setiap langkah, treap yang akarnya memiliki bobot terkecil dipilih. Jika akar dari treap kiri memiliki bobot terkecil, akar dan subpohon kirinya dipindahkan ke treap baru dan subpohon kanannya menjadi akar baru dari treap kiri. Demikian pula, jika akar dari treap kanan memiliki bobot paling kecil, akar dan subpohon kanannya dipindahkan ketreap baru dan subpohon kirinya menjadi akar baru dari treap kiri. perangkap yang tepat. Karena ketinggian treap adalah O(log n), operasi ini bekerja dalam waktu O(log n). Misalnya, sekarang kita dapat menukar urutan dua array dalam skenario contoh kita dan kemudian menggabungkan array lagi. Gambar 15.20 menunjukkan sinar sebelum digabungkan, dan Gambar 15.21 menunjukkan hasil akhir. Pertama, node D dan subtree kanannya ditambahkan ke treap baru. Kemudian, simpul A dan subpohon kanannya menjadi subpohon kiri dari simpul D. Setelah ini, simpul C dan subpohon kirinya menjadi subpohon kiri dari simpul A. Akhirnya, simpul H dan simpul S ditambahkan ke treap baru.



Gambar 15.20 Menggabungkan dua array menjadi sebuah array, sebelum digabungkan



Gambar 15.21 Menggabungkan dua array menjadi sebuah array, setelah digabungkan

15.3.2 Implementasi

Selanjutnya kita akan mempelajari cara mudah untuk mengimplementasikan sebuah treap. Pertama, berikut adalah struct yang menyimpan simpul treap:

```
struct node {
        node *left, *right;
        int weight, size, value;
        node(int v) {
left = right = NULL;
        weight = rand();
        size = 1;
        value = v;
        }
};
```

Ukuran bidang berisi ukuran subpohon dari simpul. Karena sebuah node dapat berupa NULL, fungsi berikut berguna:

```
int size(node *treap) {
        if (treap == NULL)
        return 0; return treap->size;
}
```

Pemisahan fungsi berikut mengimplementasikan operasi pemisahan. Fungsi tersebut secara rekursif membagi treap treap ke kiri dan kanan sehingga treap kiri berisi k node pertama dan treap kanan berisi node yang tersisa.

```
void split(node *treap, node *&left,node *&right, int k) {
        if (treap == NULL) {
                left = right = NULL;
                        } else {
        if (size(treap->left) < k) {
                split(treap->right, treap->right, right,
                k-size(treap->left)-1);
                left = treap;
                        } else {
                split(treap->left, left, treap->left, k);
                right = treap;
        }
                treap->size = size(treap->left)+size(treap->right)+1;
        }
}
```

Kemudian, penggabungan fungsi berikut mengimplementasikan operasi penggabungan. Fungsi ini membuat treap treap yang pertama berisi node dari treap kiri dan kemudian node dari treap kanan.

```
void merge(node *&treap, node *left, node *right) {
        if (left == NULL) treap = right;
        else if(right == NULL) treap = left; else {
                if (left->weight < right->weight) {
                        merge(left->right, left->right, right);
                        treap = left;
                } else {
                        merge(right->left, left, right->left);
                        treap = right;
                }
                treap->size = size(treap->left)+size(treap->right)+1;
        }
}
```

Misalnya, kode berikut membuat treap yang sesuai dengan array [1,2,3,4]. Kemudian membaginya menjadi dua treapsofsize2dan menukar urutannyauntuk membuat treap baru yang sesuai dengan array[3,4,1,2].

```
node *treap = NULL;
merge(treap, treap, new node(1));
merge(treap, treap, new node(2));
merge(treap, treap, new node(3));
merge(treap, treap, new node(4));
node *left, *right;
split(treap, left, right, 2);
merge(treap, right, left);
```

15.3.3 Teknik Tambahan

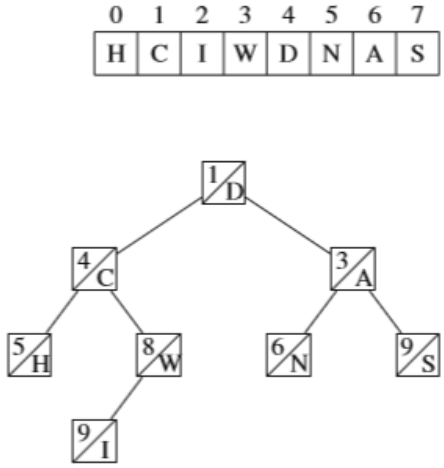
Operasi pemisahan dan penggabungan dari treap sangat kuat, karena kita dapat dengan bebas "memotong dan menempel" array dalam waktu logaritmik menggunakannya. Treap juga dapat diperpanjang sehingga bekerja hampir seperti pohon segmen. Misalnya, selain mempertahankan ukuran setiap subpohon, kita juga dapat mempertahankan jumlah nilainya, nilai minimum, dan sebagainya. Salah satu trik khusus yang terkait dengan treap adalah bahwa kita dapat membalikkan array secara efisien. Ini dapat dilakukan dengan menukar anak kiri dan kanan dari setiap node dalam treap. Sebagai contoh, Gambar 15.22 menunjukkan hasil setelah membalik array pada Gambar 15.18. Untuk melakukan ini secara efisien, kita dapat memperkenalkan bidang yang menunjukkan apakah kita harus membalikkan subpohon dari node, dan memproses operasi swapping dengan malas.

15.4 Optimasi Pemrograman Dinamis

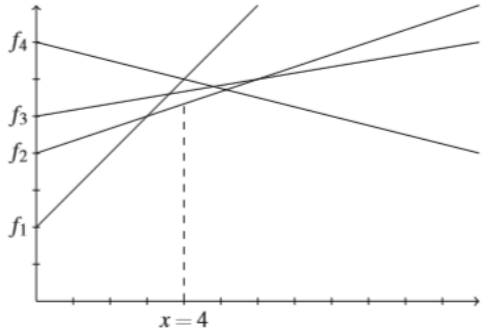
Bagian ini membahas teknik untuk mengoptimalkan solusi pemrograman dinamis. Pertama, kita fokus pada convex hulltrick, yang dapat digunakan secara efisien untuk menemukan nilai minimum dari fungsi linear. Setelah ini, kita membahas dua teknik lain yang didasarkan pada sifat-sifat fungsi biaya.

15.4.1 Trik Lambung Cembung

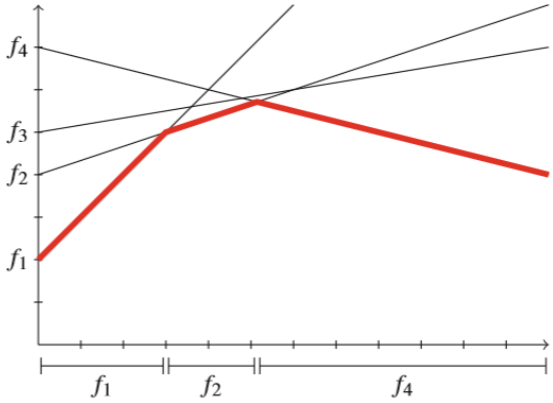
Trik convex hull memungkinkan kita untuk secara efisien menemukan nilai fungsi minimum pada suatu titik tertentux di antara asetofnfungsilinierbentuk f (x) =ax+b.Misalnya,Gambar 15.23menunjukkanfungsi $f_1(x) = x + 2$, $f_2(x) = x/3 + 4$, $f_3(x) = x/6 + 5$, dan $f_4(x) = -x/4 + 7$. Nilai minimum pada titik x =4 isf2(4) =16/3. Idenya adalah untuk membagi sumbu x menjadi rentang di mana fungsi tertentu memiliki nilai minimum. Ternyata setiap fungsi akan memiliki paling banyak satu rentang, dan kami dapat menyimpan rentang dalam daftar terurut yang akan berisi paling banyak n rentang. Misalnya, Gambar 15.24 menunjukkan rentang dalam skenario contoh kita. Pertama, f1 memiliki nilai minimum, kemudian f2 memiliki nilai minimum, dan terakhir f4 memiliki nilai minimum. Perhatikan bahwa f3 tidak pernah memiliki nilai minimum.



Gambar 15.22 Membalikkan array menggunakan treap



Gambar 15.23 Nilai fungsi minimum pada titik x =4 adalah $f_2(4)$ =16/3



Gambar 15.24 Rentang di mana f1, f2, dan f4 memiliki nilai minimum

Diberikan daftar rentang, kita dapat menemukan nilai fungsi minimum pada titik x dalam waktu O(log n) menggunakan pencarian biner. Sebagai contoh, karena titik x =4 termasuk dalam jangkauan f_2 pada Gambar 15.24, kita segera mengetahui bahwa nilai fungsi minimum pada titik x =4 adalah $f_2(4)$ =16/3. Dengan demikian, kita dapat memproses satu set k kueri dalam waktu O(k log n). Selain itu, jika kueri diberikan dalam urutan yang meningkat, kita dapat memprosesnya dalam waktu O(k) hanya dengan mengulangi rentang dari kiri ke kanan. Lalu, bagaimana cara menentukan rentangnya? Jika fungsi diberikan dalam penurunan urutan kemiringannya, kita dapat dengan mudah menemukan rentangnya, karena kita dapat mempertahankan tumpukan yang berisi rentang tersebut, dan biaya diamortisasi untuk memproses setiap fungsi adalah O(1). Jika fungsi diberikan dalam urutan arbitrer, kita perlu menggunakan struktur himpunan yang lebih canggih dan pemrosesan setiap fungsi membutuhkan waktu O(log n).

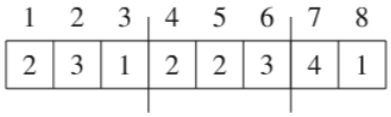
Contoh

Misalkan ada n konser berturut-turut. Tiket konser i berharga p_i, dan jika kita menghadiri konser, kita mendapatkan kupon diskon yang nilainya di ($0 < d_i < 1$). Nanti kita bisa menggunakan kupon tersebut untuk membeli tiket seharga di p euro dimana p adalah harga aslinya. Diketahui juga bahwa di d_i+1 untuk semua konser berturut-turut i dani +1.

Kami pasti ingin menghadiri konser terakhir, dan kami juga dapat menghadiri konser lainnya. Berapa total harga minimum untuk ini? Kita dapat dengan mudah memecahkan masalah menggunakan pemrograman dinamis dengan menghitung untuk setiap konser i nilai u_i: harga minimum untuk menghadiri konser i dan mungkin beberapa konser sebelumnya. Cara sederhana untuk menemukan pilihan optimal

untuk konser sebelumnya adalah melalui semua konser sebelumnya dalam waktu O(n), yang menghasilkan algoritma waktu O(n^2). Namun, kita dapat menggunakan trik lambung cembung untuk menemukan pilihan optimal dalam waktu O(log n) dan mendapatkan algoritma waktu O(n log n).

Idenya adalah untuk mempertahankan satu set fungsi linier, yang awalnya hanya berisi fungsi f (x) = x, yang berarti bahwa kita tidak memiliki kupon diskon. Untuk menghitung nilai ui untuk sebuah konser, kita menemukan fungsi f dalam himpunan kita yang meminimalkan nilai f(pi), yang dapat dilakukan dalam waktu O (log n) menggunakan fungsi trik lambung cembung. Kemudian, tambahkan sebuah fungsi f (x) =d_ix +u_i ke set kita, dan kita bisa menggunakannya untuk menghadiri konser lain nanti. Algoritma yang dihasilkan bekerja dalam waktu O(n log n).



Gambar 15.25 Cara optimal untuk membagi urutan menjadi tiga blok

Perhatikan bahwa jika diketahui tambahan bahwa pi pi+1 untuk semua konser berturut-turut dan i +1, kita dapat menyelesaikan masalah lebih efisien dalam waktu O(n), karena kita dapat memproses rentang dari kiri ke kanan dan menemukan masing-masing optimal pilihan dalam waktu konstan diamortisasi daripada menggunakan pencarian biner.

15.4.2 Pembagian dan Optimasi yang Mengatasi *(Conquer)*

Optimasi bagi dan taklukkan dapat diterapkan pada masalah pemrograman dinamis tertentu di mana barisan s_1, s_2,..., s_n dari n elemen harus dibagi menjadi k turunan dari elemen berurutan. Sebuah biaya fungsi biaya (a,b) diberikan, yang menentukan biaya untuk menciptakan sebuah urutan s_a, s_a+1, ..., s_b. Biaya total sebuah divisi adalah jumlah dari biaya individu untuk urutan berikutnya, dan tugas kita untuk menemukan divisi yang meminimalkan biaya total. Sebagai contoh, misalkan kita memiliki barisan bilangan bulat positif dan biaya(a,b) = (s_a +s_a+1 +⋯+s_b)^2.

Gambar 15.25 menunjukkan cara optimal untuk membagi barisan menjadi tiga barisan dengan menggunakan fungsi biaya ini. Total biaya pembagian adalah $(2+3+1)^2$ +$(2+2+3)^2$ +$(4+1)^2$ =110. Kita dapat memecahkan masalah dengan mendefinisikan sebuah fungsi yang dipecahkan (i, j) yang memberikan total biaya minimum untuk membagi elemen pertama, s2,...,si ke dalam j turunan. Jelas, solve(n,k) sama dengan jawaban untuk masalah tersebut. Untuk menghitung nilai penyelesaian (i, j), kita harus mencari posisi 1≤ p ≤i yang meminimalkan nilai

$$\text{solve}(p-1, j-1)+\text{cost}(p,i).$$

Misalnya, pada Gambar 15.25, pilihan optimal untuk penyelesaian(8,3) adalah p =7. Cara sederhana untuk menemukan posisi optimal adalah dengan memeriksa semua posisi 1,2,...,i, yang membutuhkan waktu O(n). Dengan menghitung semua nilai solve(i, j) seperti ini, kita mendapatkan algoritma pemrograman dinamis yang bekerja dalam waktu O(n^2k). Namun, dengan menggunakan optimasi bagi dan taklukkan, kita dapat meningkatkan kompleksitas waktu menjadi O(nk log n). Optimasi pembagian dan taklukkan dapat digunakan jika fungsi biaya memenuhi pertidaksamaan segi empat

$$\text{cost}(a,c)+\text{cost}(b,d) \leq \text{cost}(a,d)+\text{cost}(b,c)$$

untuk semua a b c d. Misalkan pos(i, j) menunjukkan posisi terkecil p yang meminimalkan biaya pembagian untuk penyelesaian(i, j). Jika pertidaksamaan di atas berlaku, dijamin bahwa pos(i, j) pos(i +1, j)untuk semua nilai sofi dan j, yang memungkinkan kita menghitung nilai penyelesaian(i, j) dengan lebih efisien.

Idenya adalah untuk membuat sebuah fungsi calc(j,a,b,x, y) yang menghitung semua nilai dari solve(i,j)untuk i band j menggunakan informasi bahwa x pos(i, j) y. Fungsi tersebut terlebih dahulu menghitung nilai dari solve(z, j) dimana z = (a+b)/2 . Kemudian melakukan pemanggilan rekursif calc(j,a,z−1,x, p)andcalc(j,z+1,b , p, y)di mana p =pos(z, j). Di sini fakta bahwa pos(i, j) pos(i +1, j) digunakan untuk membatasi rentang pencarian. Untuk menghitung semua nilai penyelesaian(i,j), kita melakukan pemanggilan fungsi calc(j,1,n,1,n) untuk setiap j = 1,2,...,k. Karena setiap pemanggilan fungsi tersebut membutuhkan waktu O(n log n), algoritme yang dihasilkan bekerja dalam waktu O(nklog n). Akhirnya, mari kita buktikan bahwa fungsi jumlah kuadrat dalam contoh kita memenuhi pertidaksamaan segi empat. Misalkan sum(a,b) menyatakan jumlah nilai dalam range[a,b], dan misalkan x =sum(b,c), y =sum(a,c)−sum(b,c), dan z = sum(b,d)−sum(b,c). Dengan menggunakan notasi ini, pertidaksamaan segi empat menjadi

$$(x+y)^2+(x+z)^2 \le (x+y+z)^2+x^2,$$

yang sama dengan

$$0 \le 2yz.$$

Karena y dan z adalah nilai non-negatif, ini melengkapi pembuktian.

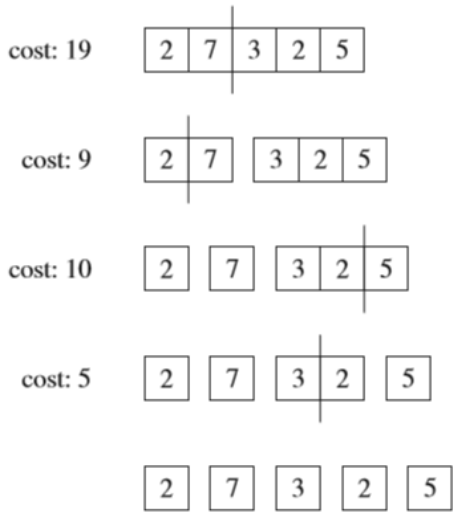
15.4.3 Optimasi Knuth

Optimasi Knuth[19] dapat digunakan pada masalah pemrograman dinamis tertentu dimana kita diminta untuk membagi barisan $s_1,s_2,...,sn$ dari n elemen menjadi elemen tunggal dengan menggunakan operasi pemisahan.Biaya fungsi(a,b) memberikan biaya pemrosesan sebuah barisan sa,sa+1,...,sb, dan Tugasnya adalah menemukan solusi yang meminimalkan jumlah total biaya pemisahan. Sebagai contoh, misalkan biaya(a,b) $=s_a+s_a+1+\cdots+s_b$. Gambar15.26 menunjukkan cara optimal untuk memproses urutan dalam kasus ini. Total biaya solusi ini adalah 19+9+10+5=43. Kita dapat menyelesaikan masalah dengan mendefinisikan sebuah fungsi yang dipecahkan (i, j) yang memberikan biaya minimum untuk membagi barisan s_i, $s_i+1,...,s_j$ menjadi elemen-elemen tunggal. Kemudian, selesaikan (1,n) memberikan jawaban untuk masalah tersebut. Untuk menentukan nilai penyelesaian (i, j), kita harus menemukan posisi i p < j yang meminimalkan nilai

$$\text{cost}(i, j)+\text{solve}(i, p)+\text{solve}(p+1, j).$$

Jika kita memeriksa semua posisi antara i dan j, kita mendapatkan algoritma pemrograman dinamis yang bekerja dalam waktu $O(n^3)$. Namun, dengan menggunakan optimasi Knuth, kita dapat menghitung nilai-nilai penyelesaian(i,j) secara lebih efisien dalam waktu $O(n^2)$.

[19] Knuth menggunakan optimasinya untuk membangun pohon pencarian biner yang optimal; kemudian, Yao menggeneralisasi optimasi ke masalah serupa lainnya.



Gambar 15.26 Cara optimal untuk membagi array menjadi elemen tunggal

Pengoptimalan Knuth dapat diterapkan jika

$$cost(b,c) \leq cost(a,d)$$

dan

$$cost(a,c)+cost(b,d) \leq cost(a,d)+cost(b,c)$$

untuk semua nilai a b c d. Perhatikan bahwa pertidaksamaan yang terakhir adalah pertidaksamaan segi empat yang juga digunakan dalam optimasi bagi dan taklukkan. Misalkan pos(i, j) menunjukkan posisi terkecil p yang meminimalkan biaya untuk penyelesaian (i, j). Jika pertidaksamaan di atas berlaku, kita tahu bahwa

$$pos(i, j-1) \leq pos(i, j) \leq pos(i+1, j).$$

Sekarang kita dapat melakukan n putaran1,2,...,n,dan pada putaran k menghitung nilai dari penyelesaian(i,j) di mana j i +1=k, yaitu, kita memproses barisan dalam urutan panjang yang bertambah. Karena kita tahu bahwa pos(i, j) harus berada di antara pos(i, j 1) dan pos(i +1, j), kita dapat melakukan setiap putaran dalam waktu O(n), dan total kompleksitas waktu dari algoritma menjadi $O(n^2)$.

15.5 Aneka ragam

Bagian ini menyajikan pilihan teknik desain algoritma lain-lain. Kami membahas teknik pertemuan di tengah, algoritma pemrograman dinamis untuk menghitung himpunan bagian, teknik pencarian biner paralel, dan solusi offline untuk masalah konektivitas dinamis.

15.5.1 Bertemu di Tengah

Teknik bertemu di tengah membagi ruang pencarian menjadi dua bagian dengan ukuran yang kira-kira sama, melakukan pencarian terpisah untuk kedua bagian, dan akhirnya menggabungkan hasil pencarian. Bertemu di tengah memungkinkan kita untuk mempercepat algoritma waktu $O(^2n)$ tertentu sehingga mereka bekerja hanya dalam waktu $O(^{2n/2})$. Perhatikan bahwa $O(2^{n/2})$ jauh lebih cepat daripada $O(^2n)$, karena $2^{n/2} = \sqrt{2^n}$. Menggunakan algoritma $O(2^n)$ kita dapat memproses input di mana n 20, tetapi menggunakan algoritma $O(^{2n/2})$ batasnya adalah n 40.

Misalkan memakai himpunan himpunan dari n bilangan bulat dan tugas kita untuk menentukan apakah himpunan tersebut memiliki himpunan bagian dengan jumlah x. Sebagai contoh, diberikan himpunan {2, 4, 5, 9} dan x =15, kita dapat memilih himpunan bagian{2,4,9}, karena 2+4+9=15. Kita dapat dengan mudah menyelesaikan

masalah dalam waktu $O(2^n)$ dengan melewati setiap kemungkinan subset, tetapi selanjutnya kita akan menyelesaikan masalah dengan lebih efisien dalam waktu $O(2^{n/2})$ menggunakan meet di tengah.

Idenya adalah untuk membagi himpunan kita menjadi dua himpunan A dan B sedemikian rupa sehingga kedua himpunan berisi sekitar setengah dari angka. Kami melakukan dua pencarian: pencarian pertama menghasilkan semua himpunan bagian dari A dan menyimpan jumlah mereka ke daftar SA, dan pencarian kedua membuat daftar SB yang serupa untuk B. Setelah ini, cukup untuk memeriksa apakah kita dapat memilih satu elemen dari SA dan elemen lain dari SB sedemikian rupa sehingga jumlah mereka adalah x, yang mungkin tepat ketika himpunan asli berisi subset dengan jumlah x.

Misalnya, mari kita lihat bagaimana himpunan{2,4,5,9}diproses. Pertama, kita membagi himpunan menjadi himpunan A ={ 2,4}dan B ={ 5,9}. Setelah ini, kita buat daftar SA =[0,2,4,6]dan SB =[0,5,9,14]. Karena SA berisi jumlah 6 dan SB berisi jumlah 9, kami menyimpulkan bahwa himpunan asli memiliki himpunan bagian dengan jumlah 6+9=15. Dengan implementasi yang baik, kita dapat membuat daftar SA dan SB dalam waktu $O(2^{n/2})$ sedemikian rupa sehingga daftar tersebut diurutkan. Setelah ini, kita dapat menggunakan dua algoritma pointer untuk memeriksa waktu $O(2^{n/2})$ jika jumlah x dapat dibuat dari SA dan SB. Dengan demikian, kompleksitas waktu total dari algoritma adalah $O(2^{n/2})$.

15.5.2 Menghitung Subset

Misal X ={ 0...n−1},dan setiap subset $S \subset X$ ditetapkan dan nilai integer[S]. Tugas kita adalah menghitung untuk setiap S

$$sum(S) = \sum_{A \subset S} value[A]$$

yaitu, jumlah nilai himpunan bagian dari S. Sebagai contoh, misalkan n =3 dan nilainya adalah sebagai berikut:

- value[Ø]=3
- value[{0}]=1
- value[{1}]=4
- value[{0,1}]=5
- value[{2}]=5
- value[{0,2}]=1
- value[{1,2}]=3
- value[{0,1,2}]=3

Dalam hal ini, misalnya,

sum({0,2}) =value[Ø]+value[{0}]+value[{2}]+value[{0,2}] =3+1+5+1=10.

Selanjutnya kita akan melihat bagaimana menyelesaikan masalah dalam waktu $O(2^n n)$ menggunakan pemrograman dinamis dan operasi bit.Idenya adalah untuk mempertimbangkan submasalah di mana elemen mana yang dapat dihilangkan dari S terbatas. Misalkan parsial(S,k) menyatakan jumlah nilai himpunan bagian dari S dengan batasan bahwa hanya elemen 0...k dapat dihapus dari S. Misalnya,

partial({0,2},1) =value[{2}]+value[{0,2}],

karena kami hanya dapat menghapus elemen 0...1. Perhatikan bahwa kita dapat menghitung nilai jumlah(S) apa pun menggunakan parsial, karena

sum(S) =partial(S,n−1).

Untuk menggunakan pemrograman dinamis, kita harus menemukan pengulangan untuk parsial. Pertama, kasus dasarnya adalah

partial(S,−1) =value[S],

karena tidak ada elemen yang dapat dihilangkan dari S. Kemudian, dalam kasus umum kita dapat menghitung nilainya sebagai berikut:

$$partial(S,k)\begin{cases} partial(S,k-1) & k \notin S \\ partial(S,k-1)+partial(S\backslash k) & k \in S \end{cases}$$

Di sini kita fokus pada elemen k. Jika k S, ada dua pilihan: kita dapat menyimpan k di subset atau menghapusnya dari subset.

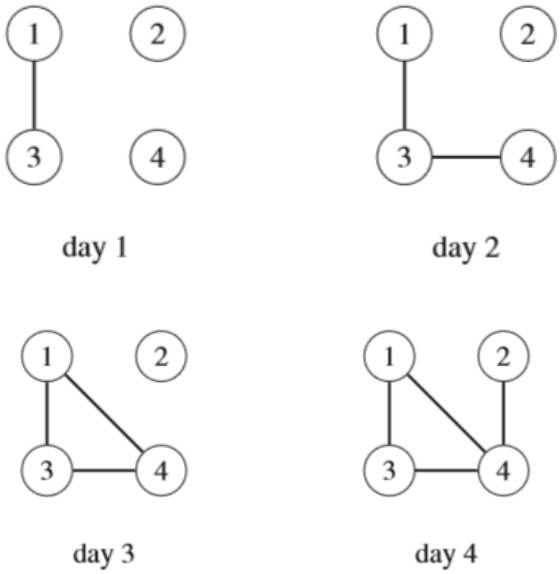
Penerapan

Ada cara yang sangat cerdas untuk mengimplementasikan solusi pemrograman dinamis menggunakan operasi bit. Yaitu kita dapat mendeklarasikan array

```
int sum[1<<N];
```

yang akan berisi jumlah setiap subset. Array diinisialisasi sebagai berikut:

```
for (int s = 0; s < (1<<n); s++) {
sum[s] = value[s];
}
```



Gambar 15.27 Contoh masalah pembangunan jalan

Kemudian, kita dapat mengisi array sebagai berikut:

```
for (int k = 0; k < n; k++) {
        for (int s = 0; s < (1<<n); s++) {
                if (s&(1<<k)) sum[s] += sum[s^(1<<k)];
        }
}
```

Kode ini menghitung nilai parsial(S, k) untuk k =0...n−1tothearraysum . Karena parsial(S,k) selalu didasarkan pada parsial(S,k−1), kita dapat menggunakan kembali jumlah array, yang menghasilkan implementasi yang sangat efisien.

15.5.3 Pencarian Biner Paralel

Pencarian biner paralel adalah teknik yang memungkinkan kita untuk membuat beberapa algoritma berbasis pencarian biner lebih efisien. Ide umumnya adalah

melakukan beberapa pencarian biner secara bersamaan, daripada melakukan pencarian secara terpisah. Sebagai contoh, perhatikan masalah berikut: Ada n kota bernomor 1, 2,..., n. Awalnya tidak ada jalan antar kota. Kemudian, selama m hari, setiap hari dibangun jalan baru antara dua kota. Akhirnya, kita diberikan k kueri dari bentuk (a,b), dan tugas kita adalah menentukan untuk setiap kueri momen paling awal ketika kota a dan b terhubung. Kita dapat mengasumsikan bahwa semua pasangan kota yang diminta terhubung setelah m hari. Gambar 15.27 menunjukkan contoh skenario dimana terdapat empat kota. Misalkan kuerinya adalah q_1 = (1,4) dan q_2 = (2,3). Jawaban untuk q1 adalah 2, karena kota 1 dan 4 terhubung setelah hari ke-2, dan jawaban untuk q_2 adalah 4, karena kota 2 dan 3 terhubung setelah hari ke-4.

Mari kita pertimbangkan masalah yang lebih mudah di mana kita hanya memiliki satu query (a,b). Dalam hal ini, kita dapat menggunakan struktur union-find untuk mensimulasikan proses penambahan jalan ke jaringan. Setelah setiap jalan baru, kami memeriksa apakah kota a dan b terhubung dan menghentikan pencarian jika terhubung. Menambahkan jalan dan memeriksa apakah kota terhubung membutuhkan waktu O(log n), sehingga algoritme bekerja dalam waktu O(m log n). Bagaimana kita bisa menggeneralisasi solusi ini ke k kueri? Tentu saja kami dapat memproses setiap kueri secara terpisah, tetapi algoritme seperti itu akan memakan waktu O(km log n), yang akan lambat jika k dan m keduanya besar. Selanjutnya kita akan melihat bagaimana kita dapat menyelesaikan masalah dengan lebih efisien menggunakan pencarian biner paralel.

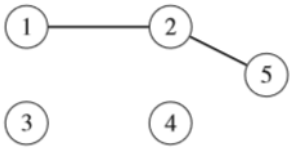
Idenya adalah untuk menetapkan setiap kueri rentang [x, y] yang berarti bahwa kota-kota terhubung untuk pertama kalinya tidak lebih awal dari setelah x hari dan tidak lebih dari setelah y hari. Awalnya, setiap rentang adalah [1, m]. Kemudian, kami mensimulasikan log m kali proses penambahan semua jalan ke jaringan menggunakan struktur temukan gabungan. Untuk setiap kueri, kami memeriksa pada saat u = (x +y)/2 jika kota-kota terhubung. Jika ya, rentang baru menjadi [x, u], dan sebaliknya rentang menjadi [u+1, y]. Setelah putaran logm, setiap rentang hanya berisi satu momen yang merupakan jawaban atas kueri. Selama setiap putaran, kami menambahkan m jalan ke jaringan dalam waktu O(m log n) dan memeriksa apakah pasangan kota terhubung dalam waktu O(k log n).Jadi, karena ada putaran log, algoritme yang dihasilkan bekerja dalam waktu O((m+k)log n log m).

15.5.4 Konektivitas Dinamis

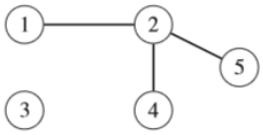
Misalkan terdapat graf n simpul dan m sisi. Kemudian, kita diberikan kueri q, yang masing-masing adalah "tambahkan tepi antara node a dan b" atau "hapus tepi antara node a dan b." Tugas kita adalah melaporkan secara efisien jumlah komponen yang terhubung dalam grafik setelah setiap kueri. Gambar15.28 menunjukkan contoh proses. Awalnya, grafik memiliki tiga komponen. Kemudian, tepi 2–4 ditambahkan, yang menggabungkan dua komponen. Setelah ini, tepi 4-5 ditambahkan dan tepi 2-5 dihilangkan, tetapi jumlah komponen tetap sama. Kemudian, tepi 1–3 ditambahkan, yang menghubungkan dua komponen, dan akhirnya, tepi 2–4 dihilangkan, yang membagi komponen menjadi dua komponen. Jika tepi hanya akan ditambahkan ke grafik, masalah akan mudah diselesaikan dengan menggunakan struktur data pencarian gabungan, tetapi operasi pemindahan membuat masalah menjadi lebih sulit. Selanjutnya kita akan membahas algoritma bagi dan taklukkan untuk memecahkan masalah versi offline di mana semua kueri diketahui sebelumnya, dan kita diizinkan

untuk melaporkan hasilnya dalam urutan apa pun. Algoritma yang disajikan di sini didasarkan pada karya Kopeliovich.

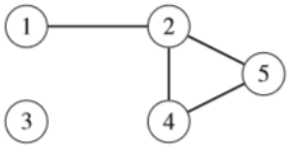
Idenya adalah untuk membuat garis waktu di mana setiap tepi diwakili oleh interval yang menunjukkan waktu penyisipan dan penghapusan tepi. Garis waktu mencakup rentang [0,q +1], dan tepi yang ditambahkan pada langkah a dan dihapus pada langkah b diwakili oleh interval[a,b]. Jika sebuah sisi termasuk dalam graf awal, a =0, dan jika sebuah sisi tidak pernah dihilangkan,b =q +1.Gambar 15.29 menunjukkan garis waktu dalam skenario contoh kami. Untuk mengolah interval, kita membuat graf yang memiliki n node dan tanpa edge, dan menggunakan fungsi rekursif yang disebut dengan range[0,q +1]. Fungsi tersebut bekerja sebagai berikut untuk range[a,b]: Pertama, jika[a,b] benar-benar berada di dalam interval sisi, dan sisinya bukan milik graf, maka ditambahkan ke graf. Kemudian, jika ukuran [a,b] adalah 1, kami melaporkan jumlah komponen yang terhubung, dan jika tidak, kami memproses secara rekursif rentang[a,k]dan[k,b]di mana k = (a+b)/2 . Terakhir, kami menghapus semua tepi yang ditambahkan pada awal pemrosesan rentang[a,b].



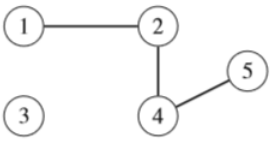
Gambar a jumlah grafik awal komponen: 3



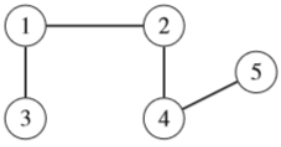
Gambar b langkah 1: tambahkan tepi 2–4 jumlah komponen: 2



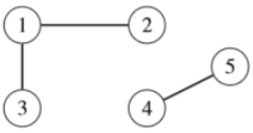
Gambar c langkah 2: tambahkan edge 4-5 jumlah komponen: 2



Gambar d langkah 3: hapus tepi 2–5 jumlah komponen: 2

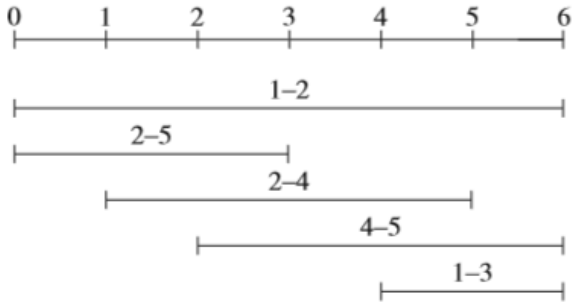


Gambar e langkah 4: tambahkan tepi 1-3 jumlah komponen: 1



Gambar f langkah 5: hapus tepi 2–4 jumlah komponen: 2

Gambar 15.28 (a-f) Masalah konektivitas dinamis



Gambar 15.29 Garis waktu penyisipan dan pelepasan tepi

Selalu ketika tepi ditambahkan atau dihapus, kami juga memperbarui jumlah komponen. Ini dapat dilakukan dengan menggunakan struktur data union-find, karena kami selalu menghapus tepi yang ditambahkan terakhir. Jadi, cukup untuk mengimplementasikan operasi undo untuk struktur union-find, yang dimungkinkan dengan menyimpan informasi tentang operasi dalam stack. Karena setiap edge ditambahkan dan dihapus paling banyak O(log q) kali dan setiap operasi bekerja dalam waktu O(log n), total waktu berjalan dari algoritma adalah O((m+q)log q log n). Perhatikan bahwa selain menghitung jumlah komponen, kami dapat mempertahankan informasi apa pun yang dapat digabungkan dengan struktur data union-find. Sebagai contoh, kita dapat mempertahankan jumlah node dalam komponen terbesar atau bipartiteness dari setiap komponen. Teknik ini juga dapat digeneralisasi ke struktur data lain yang mendukung operasi penyisipan dan pembatalan.

DAFTAR PUSTAKA

R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network Flows: Theory, Algorithms, and Applications, Pearson, 1993.

A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. Information Processing Letters, 9(5):216–219, 1979.

M. A. Bender and M. Farach-Colton. The LCA problem revisited. Latin American Symposium on Theoretical Informatics, 88–94, 2000.

J. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. IEEE Transactions on Computers, C-29(7):571–577, 1980.

Codeforces: On ”Mo’s algorithm”, http://codeforces.com/blog/entry/20032

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms, MIT Press, 2009 (3rd edition).

K. Diksetal. Looking for a Challenge? The Ultimate Problem Set from the University of War saw Programming Competitions, University of Warsaw, 2012

D. Fanding. A faster algorithm for shortest-path – SPFA. Journal of Southwest Jiaotong University, 2, 1994

P. M. Fenwick. A new data structure for cumulative frequency tables. Software: Practice and Experience, 24(3):327–336, 1994.

J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applicationsto LCA and LCE. Annual Symposium on Combinatorial Pattern Matching, 36–48, 2006.

F. Le Gall. Powers of tensors and fast matrix multiplication. International Symposium on Symbolic and Algebraic Computation, 296–303, 2014.

A.Grønlundand S. Pettie. Three somes, degenerates, and love triangles. Annual Symposium on Foundations of Computer Science, 621–630, 2014.

D. Gusfield. Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.

S. Halim and F. Halim. Competitive Programming 3: The New Lower Bound of Programming Contests, 2013.

The International Olympiad in Informatics Syllabus, https://people.ksp.sk/~misof/ioi-syllabus/

J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. International Colloquium on Automata, Languages, and Programming, 943–955, 2003.

R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. Annual ACM Symposium on Theory of Computing, 125–135, 1972.

T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. Annual Symposium on Combinatorial Pattern Matching, 181–192, 2001.

J. Kleinberg and É. Tardos. Algorithm Design, Pearson, 2005. 20. D. E. Knuth. Optimum binary search trees. Acta Informatica 1(1):14–25, 1971.

S. Kopeliovich. Offline solution of connectivity and 2-edge-connectivity problems for fully dynamic graphs. MSc thesis, Saint Petersburg State University, 2012.

M. G. Main and R. J. Lorentz. An O(n log n) algorithm for finding all repetitions in a string. Journal of Algorithms, 5(3):422–432, 1984

J. Pachocki and J. Radoszewski. Where to use and how not to use polynomial string hashing. Olympiads in Informatics, 7(1):90–100, 2013.

D. Pearson. A polynomial-time algorithm for the change-making problem. Operations Research Letters, 33(3):231–234, 2005.

27-Queens Puzzle:Massively Parallel Enumeration and Solution Counting. https://github.com/preusser/q27

M. I. Shamos and D. Hoey. Closest-point problems. Annual Symposium on Foundations of Computer Science, 151–162, 1975.

S. S. Skiena. The Algorithm Design Manual, Springer, 2008 (2nd edition).

S. S. Skiena and M. A. Revilla. Programming Challenges: The Programming Contest Training Manual, Springer, 2003.

D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. Journal of Computer and System Sciences, 26(3):362–391, 1983.

P. Sta´nczyk. Algorytmika praktyczna w konkursach Informatycznych. MSc thesis, University of Warsaw, 2006.

V. Strassen. Gaussian elimination is not optimal. Numerische Mathematik, 13(4):354–356, 1969.

F. F. Yao. Efficient dynamic programming using quadrangle inequalities. Annual ACM Symposium on Theory of Computing, 429–435, 1980.