

PEMROGRAMAN KOMPETITIF DASAR

Panduan Memulai OSN Informatika,
ACM-ICPC, dan Sederajat



Ikatan Alumni
Tim Olimpiade
Komputer Indonesia

William Gozali & Alham Fikri Aji

Abstrak

Buku persiapan OSN informatika, atau disebut juga OSN komputer, yang ditulis oleh Ikatan Alumni Tim Olimpiade Komputer Indonesia (TOKI). Buku elektronik ini dapat diunduh (download) dan bersifat gratis. Seluruh materi disesuaikan dengan kurikulum OSN sehingga seluruh siswa calon peserta OSN dapat belajar dengan lebih mudah. Buku ini juga dapat digunakan bagi pelajar yang hendak memulai partisipasi pada ACM-ICPC.

PEMROGRAMAN KOMPETITIF DASAR, VERSI 1.9

Dipublikasikan oleh CV. Nulisbuku Jendela Dunia

Penulis	Alham Fikri Aji (IA-TOKI), William Gozali (IA-TOKI)
Kontributor	Agus Sentosa Hermawan (NUS), Ali Jaya Meilio Lie (Université de Grenoble Alpes), Arianto Wibowo (IA-TOKI), Ashar Fuadi (IA-TOKI), Cakra Wishnu Wardhana (UI), Jonathan Irvin Gunawan (Google), Maximilianus Maria Kolbe Lie (BINUS), Muhammad Ayaz Dzulfikar (UI), Muhammad Fairuzi Teguh (UI), Reynaldo Wijaya Hendry (UI)
Penyunting	Ilham Winata Kurnia (Google), Suhendry Effendy (NUS)
Desain dan tata letak	Alham Fikri Aji, Ali Jaya Meilio Lie, Pusaka Kaleb Setyabudi (Google), William Gozali

Versi elektronik buku ini dapat diakses secara gratis di URL berikut: <https://toki.id/buku-pemrograman-kompetitif-dasar>

Karya ini dilisensikan di bawah lisensi **Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0)**

Hal ini berarti Anda bebas untuk menggunakan dan mendistribusikan buku ini, dengan ketentuan:

- **Attribution:** Apabila Anda menggunakan materi-materi pada buku ini, Anda harus memberikan kredit kepada tim penulis dan Ikatan Alumni TOKI.
- **NonCommercial:** Anda tidak boleh menggunakan buku ini untuk keperluan komersial, seperti menjual ulang buku ini.
- **NoDerivatives:** Anda tidak boleh mengubah konten buku ini dalam bentuk apapun.

Masukan dan pertanyaan dapat disampaikan kepada penulis di info@toki.id.

ISBN 978-602-6598-89-9

Daftar Isi

Kata Pengantar	v
Ucapan Terima Kasih	vi
1 Perkenalan Pemrograman Kompetitif	1
Kompetensi Dasar	1
Tentang Pemrograman Kompetitif	1
Contoh Soal Pemrograman Kompetitif	2
Solusi Sederhana	3
Solusi yang Lebih Baik	4
Solusi yang Lebih Baik Lagi!	5
Menenal Kompetisi Pemrograman	7
Olimpiade Sains Nasional dan <i>International Olympiad in Informatics</i>	7
ACM-ICPC	8
Penulisan Kode	9
Perkenalan <i>Pseudocode</i>	9
2 Matematika Diskret Dasar	11
Aritmetika Modular	11
Bilangan Prima	12
Uji Keprimaan (<i>Primality Testing</i>)	12
Pembangkitan Bilangan Prima (<i>Prime Generation</i>)	12
FPB dan KPK	14
<i>Pigeonhole Principle</i> (PHP)	16
Kombinatorika	17
Aturan Perkalian dan Aturan Penjumlahan	17
Permutasi	19
Kombinasi	22
Segitiga Pascal	24
3 Pencarian dan Pengurutan	26
Pencarian	26
<i>Sequential Search</i>	26
<i>Binary Search</i>	26
Rangkuman	30
Pengurutan	30
<i>Bubble Sort</i>	31
<i>Selection Sort</i>	32
<i>Insertion Sort</i>	33
<i>Counting Sort</i>	35
Rangkuman	38
4 Brute Force	40
Konsep <i>Brute Force</i>	40
Penyelesaian dengan Teknik <i>Brute Force</i>	40

Optimisasi Teknik <i>Brute Force</i>	41
5 Divide and Conquer	45
Konsep <i>Divide and Conquer</i>	45
Studi Kasus 1: <i>Merge Sort</i>	45
Contoh Eksekusi <i>Merge Sort</i>	46
Menggabungkan Dua <i>Array</i> yang Terurut	47
Implementasi <i>Merge Sort</i>	49
Studi Kasus 2: <i>Quicksort</i>	52
Partisi <i>Quicksort</i>	52
Implementasi <i>Quicksort</i>	56
Pemilihan <i>Pivot</i>	58
Studi Kasus 3: Mencari Nilai Terbesar	58
6 Greedy	60
Konsep <i>Greedy</i>	60
Menemukan Subpersoalan	60
<i>Greedy Choice</i>	61
Penyelesaian dengan Teknik <i>Greedy</i>	62
Permasalahan pada Algoritma <i>Greedy</i>	65
Pembuktian <i>Greedy</i>	66
7 Dynamic Programming	67
Konsep <i>Dynamic Programming</i>	67
<i>Top-down</i>	67
<i>Bottom-up</i>	70
Contoh-contoh DP Sederhana	73
<i>Knapsack</i>	74
<i>Coin Change</i>	77
<i>Longest Common Subsequence</i>	78
Memotong Kayu	80
8 Struktur Data Dasar	86
<i>Dynamic Array</i>	86
Aplikasi <i>Dynamic Array</i>	88
Implementasi <i>Dynamic Array</i>	88
<i>Stack</i>	89
Aplikasi <i>Stack</i>	89
Implementasi <i>Stack</i>	91
<i>Queue</i>	92
Aplikasi <i>Queue</i>	93
Implementasi <i>Queue</i>	93
9 Perkenalan Graf	95
Konsep Graf	95
Jenis Graf	95
<i>Tree</i>	96
<i>Directed Acyclic Graph</i>	99
Representasi Graf pada Pemrograman	99

<i>Adjacency Matrix</i>	99
<i>Adjacency List</i>	100
<i>Edge List</i>	101
Perbandingan Representasi Graf	102
Penjelajahan Graf	103
DFS: <i>Depth-First Search</i>	103
BFS: <i>Breadth-First Search</i>	104
Analisis Kompleksitas DFS dan BFS	106
Contoh Permasalahan Graf	106
10 Struktur Data NonLinear	110
<i>Disjoint Set</i>	110
Konsep <i>Disjoint Set</i>	111
Analisis Kompleksitas	114
<i>Heap</i>	114
Konsep <i>Heap</i>	116
Struktur <i>Binary Heap</i>	116
Operasi <i>Push</i>	117
Operasi <i>Pop</i>	117
Operasi <i>Top</i>	120
Implementasi <i>Binary Heap</i>	120
Pembangunan <i>Heap</i> Secara Efisien	122
Catatan Implementasi <i>Heap</i>	126
Heapsort	126
11 Algoritma Graf	128
<i>Shortest Path</i>	128
Dijkstra	128
Bellman—Ford	134
Floyd—Warshall	135
<i>Minimum Spanning Tree</i>	137
Prim	139
Kruskal	142
12 Dasar—Dasar Geometri	148
Titik	148
Jarak Euclidean	148
Jarak Manhattan	149
Garis	149
Garis Vertikal	150
Segmen Garis	151
Segitiga	151
Teorema Pythagoras	152
Luas Segitiga	152
Sudut Segitiga	153
Lingkaran	153
Presisi Bilangan Riil	154
13 Persiapan Kompetisi	157

Pengenalan Medan	157
Persiapan Sebelum Kompetisi	157
Kumpulan Tips Berkompetisi	158

Kata Pengantar

Indonesia merupakan negara yang kemampuan pemrograman kompetitifnya masih berkembang. Hal ini terlihat dari perolehan tim Indonesia pada ajang International Olympiad in Informatics (IOI) yang didominasi oleh medali perak dan perunggu, serta belum rutusnya perwakilan tim dari Indonesia terkualifikasi pada ajang World Final ACM-ICPC.

Kami percaya bahwa Indonesia memiliki potensi yang besar dalam pemrograman kompetitif. Namun banyak kendala dan tantangan yang perlu dihadapi dalam perkembangan ini. Sebagai contoh, tingkat kemajuan fasilitas pembelajaran setiap daerah di Indonesia yang masih sangat beragam. Hal ini berakibat pada kemampuan pelajar antar daerah yang kurang merata.

Pada tahun 2014, Ikatan Alumni Tim Olimpiade Komputer (IA-TOKI) berinisiatif untuk mendirikan situs generasi ke-2 untuk pembelajaran pemrograman kompetitif. Situs ini dapat diakses secara gratis. Motivasi dari inisiatif ini adalah untuk pengembangan kemampuan pelajar Indonesia lewat belajar mandiri. Setiap pelajar di Indonesia kini memiliki kesempatan yang sama untuk belajar.

Berdasarkan masukan yang kami terima, solusi tahap selanjutnya untuk menyokong tujuan ini adalah dengan penerbitan buku berisikan materi pembelajaran. Dengan adanya buku pembelajaran, pembahasan materi dapat disajikan secara lebih jelas. Kelebihan lain dari buku adalah kemampuannya untuk didistribusikan secara elektronik maupun fisik. Keberadaan buku secara fisik juga mengatasi masalah fasilitas internet di Indonesia yang belum merata.

Dengan kehadiran buku ini, kami berharap perkembangan Indonesia dalam hal pemrograman kompetitif dan informatika dapat lebih terdukung. Dengan demikian, Indonesia dapat lebih berprestasi dalam bidang teknologi dan informatika.

Alham dan William

Ucapan Terima Kasih

Buku ini terbit dengan bantuan banyak orang. Cikal bakal buku ini adalah materi pembelajaran yang dimuat pada TLX Training Gate, yang mana melibatkan banyak orang pula. Seluruh kontributor didaftarkan pada halaman lainnya, tetapi beberapa dari kontributor tersebut ada yang berperan besar. Kami sebagai penulis mengucapkan terima kasih kepada:

Ashar Fuadi, yang telah menginisiasi rencana pembuatan materi pemrograman pada TLX, memberikan bimbingan terhadap tata bahasa dan sistematika penulisan.

Suhendry Effendy, sebagai penyunting buku ini yang telah memberikan berbagai macam masukan.

Ali Jaya Meilio Lie, yang memberi masukan dalam hal tata letak dan tampilan buku.

Para penulis materi pemrograman kompetitif dasar yang dimuat di TLX:

- Maximilianus Maria Kolbe Lie, yang menulis materi matematika diskret bagian kombinatorika.
- Muhammad Ayaz Dzulfikar, yang menulis materi matematika diskret bagian aritmetika modular, bilangan prima, FPB dan KPK, serta *pigeonhole principle*.
- Jonathan Irvin Gunawan, yang menulis materi *dynamic programming*.
- Arianto Wibowo, yang menulis materi graf.
- Agus Sentosa Hermawan, yang menulis materi pengenalan pemrograman kompetitif dan *brute force*.

1 Perkenalan Pemrograman Kompetitif

Untuk memulai pembahasan pada bab-bab selanjutnya, mari kita mulai dengan membahas tentang apakah sebenarnya pemrograman kompetitif. Bab perkenalan ini juga memuat informasi tentang kompetensi dasar yang Anda butuhkan dan konvensi yang digunakan pada penulisan kode.

Kompetensi Dasar

Buku ini dirancang bagi pembaca yang sudah pernah belajar pemrograman sebelumnya, namun masih awam mengenai dunia pemrograman kompetitif. Buku ini bukanlah buku untuk belajar pemrograman. Terdapat sejumlah kompetensi dasar yang perlu Anda kuasai untuk dapat memahami materi pada buku ini secara maksimal. Anda diharapkan untuk:

1. Memahami konsep pemrograman dan mampu menulis program secara umum.
2. Mengetahui jenis-jenis tipe data primitif seperti bilangan bulat, bilangan riil, *boolean*, dan karakter.
3. Mengetahui konsep tipe data komposit seperti *record* (Pascal), *struct* (C), atau *class* (C++ dan Java).
4. Memahami konsep dan mampu menulis struktur percabangan *if* pada program.
5. Memahami konsep dan mampu menulis struktur perulangan menggunakan *for* dan *while* pada program.
6. Memahami konsep *array*, baik yang berdimensi satu maupun lebih.
7. Memahami konsep subprogram berupa fungsi dan prosedur.
8. Memahami konsep rekursi, mampu mengidentifikasi basis dan hubungan rekursif dari suatu permasalahan, dan mampu menuliskannya pada program.
9. Mampu menghitung kompleksitas waktu dan memori dari suatu algoritma dan menyatakannya dalam notasi *big-Oh*.

Apabila Anda belum menguasai seluruh kompetensi dasar tersebut, kami menyarankan Anda untuk mempelajarinya terlebih dahulu. Sumber pembelajaran yang kami anjurkan adalah TLX Training Gate (<https://tlx.toki.id/training>). Seluruh kompetensi dasar yang diperlukan dibahas pada kursus berjudul "Pemrograman Dasar".

Tentang Pemrograman Kompetitif

Pemrograman kompetitif adalah kegiatan penyelesaian soal yang terdefinisi dengan jelas dengan menulis program komputer dalam batasan-batasan tertentu (memori dan waktu).¹ Terdapat tiga aspek pada definisi tersebut, yaitu:

1. Soal terdefinisi dengan jelas. Artinya diketahui dengan jelas apa maksud soal dan tidak ada keambiguan. Seluruh variabel pada soal diberikan batasan nilainya.
2. Diselesaikan dengan menulis program komputer. Program komputer yang dimaksud adalah program sederhana yang berjalan pada *command line*, tanpa perlu adanya *user interface* seperti halaman *website* atau GUI (*Graphic User Interface*). Program hanya perlu membaca masukan dan menghasilkan keluaran.
3. Memenuhi batasan yang diberikan. Umumnya batasan ini berupa batas waktu dan memori. Program yang ditulis harus mampu bekerja mulai dari membaca masukan

sampai dengan mencetak keluaran tidak lebih dari batas waktu dan memori yang diberikan.

Kita akan memahami lebih dalam tentang pemrograman kompetitif melalui contoh soal, yang akan dijelaskan pada bagian berikutnya.

Contoh Soal Pemrograman Kompetitif

Mari perhatikan contoh soal berikut.

Contoh Soal 1.1: Lampu dan Tombol

Batas waktu: 1 detik
Batas memori: 64 MB

Terdapat N tombol yang dinomori dari 1 hingga N dan sebuah lampu dalam keadaan mati. Apabila tombol ke- i ditekan, keadaan lampu akan berubah (dari mati menjadi menyala, atau sebaliknya) apabila N habis dibagi oleh i . Apabila masing-masing tombol ditekan tepat sekali, bagaimana keadaan lampu pada akhirnya?

Format Masukan

Sebuah baris berisi sebuah bilangan, yaitu N .

Format Keluaran

Sebuah baris berisi:

- "lampu mati", apabila keadaan akhir lampu adalah mati.
- "lampu menyala", apabila keadaan akhir lampu adalah menyala.

Contoh Masukan 1

5

Contoh Keluaran 1

lampu mati

Contoh Masukan 2

4

Contoh Keluaran 2

lampu menyala

Penjelasan

Pada contoh pertama, tombol yang mempengaruhi keadaan lampu adalah tombol 1 dan tombol 5. Penekanan tombol 1 mengakibatkan lampu menjadi menyala, dan penekanan tombol 5 mengembalikannya ke keadaan mati.

Pada contoh kedua, tombol yang mempengaruhi keadaan lampu adalah tombol 1, tombol 2, dan tombol 4. Penekanan tombol 1 mengakibatkan lampu menjadi menyala, penekanan tombol 2 mengembalikannya ke keadaan mati, dan penekanan tombol 4 menjadikan lampu kembali menyala.

Batasan

$$\bullet 1 \leq N \leq 10^{18}$$

Seperti yang tertera pada contoh soal Lampu dan Tombol, soal pemrograman kompetitif memiliki batasan-batasan yang jelas. Pada soal tersebut, tertera bahwa batasan waktu adalah 1 detik. Artinya, program Anda harus memiliki *running time* tidak melebihi 1 detik. Batasan memori 64 MB menyatakan bahwa program Anda juga tidak boleh memakan memori lebih dari batasan tersebut. Terakhir, terdapat batasan masukan, yakni berupa $1 \leq N \leq 10^{18}$. Artinya program Anda diharuskan dapat menyelesaikan soal tersebut untuk nilai N yang diberikan. Batasan ini juga menjamin bahwa program Anda tidak akan diuji dengan nilai-nilai di luar batasan.

Solusi Sederhana

Strategi yang paling sederhana adalah dengan menyimulasikan skenario pada deskripsi soal:

- Mulai dari tombol ke-1, dipastikan keadaan lampu akan berubah (N habis dibagi 1).
- Lanjut ke tombol ke-2, periksa apakah 2 habis membagi N . Bila ya, ubah keadaan lampu.
- Lanjut ke tombol ke-3, periksa apakah 3 habis membagi N . Bila ya, ubah keadaan lampu.
- ... dan seterusnya hingga tombol ke- N .

Setelah selesai disimulasikan, periksa keadaan lampu ruangan ke- N dan cetak jawabannya.

Berikut adalah implementasi solusi sederhana ini dalam C++:

```
#include <iostream>

using namespace std;

int main() {
    long long N;
    cin >> N;

    int divisorCount = 0;
    for (long long i = 1; i <= N; i++) {
        if (N % i == 0) {
            divisorCount++;
        }
    }

    if (divisorCount % 2 == 0) {
        cout << "lampu_mati" << endl;
    } else {
        cout << "lampu_menyala" << endl;
    }
}
```

Untuk masukan N , program sederhana ini melakukan N buah pengecekan. Artinya pada kasus terburuk, program tersebut akan melakukan $N = 10^{18}$ buah pengecekan! Jika kita asumsikan komputer sekarang dapat melakukan 10^8 operasi per detik, program ini memerlukan waktu 10 tahun untuk menyelesaikan hitungan! Program ini hanya akan bekerja untuk nilai N yang kecil. Untuk N yang lebih besar, misalnya $N = 10^9$, kemungkinan besar diperlukan waktu lebih dari 1 detik. Solusi ini tidak akan mendapatkan nilai penuh, atau bahkan 0, tergantung skema penilaian yang digunakan.

Solusi yang Lebih Baik

Dengan sedikit observasi, yang sebenarnya perlu dilakukan adalah menghitung banyaknya pembagi dari N . Apabila banyaknya pembagi ganjil, berarti pada akhirnya lampu di ruangan ke- N akan menyala. Demikian pula sebaliknya, apabila genap, berarti lampu di ruangan ke- N akan mati.

Diperlukan suatu cara untuk menghitung banyaknya pembagi dari N dengan efisien. Salah satu caranya adalah dengan melakukan faktorisasi prima terlebih dahulu. Misalkan untuk $N = 12$, maka faktorisasi primanya adalah $2^2 \times 3$. Berdasarkan faktorisasi prima ini, suatu bilangan merupakan pembagi dari 12 apabila dipenuhi:

- Memiliki faktor 2 maksimal sebanyak 2.
- Memiliki faktor 3 maksimal sebanyak 1.
- Tidak boleh memiliki faktor lainnya.

Sebagai contoh, berikut daftar seluruh pembagi dari 12:

- $1 = 2^0 \times 3^0$
- $2 = 2^1 \times 3^0$
- $3 = 2^0 \times 3^1$
- $4 = 2^2 \times 3^0$
- $6 = 2^1 \times 3^1$
- $12 = 2^2 \times 3^1$

Banyaknya pembagi dari 12 sebenarnya sama saja dengan banyaknya kombinasi yang bisa dipilih dari $\{2^0, 2^1, 2^2\}$ dan $\{3^0, 3^1\}$. Banyaknya kombinasi sama dengan mengalikan banyaknya elemen pada tiap-tiap himpunan. Sehingga, banyaknya cara adalah $3 \times 2 = 6$ cara. Dengan demikian, banyaknya pembagi dari 12 adalah 6. Cara ini juga berlaku untuk nilai N yang lain. Misalnya $N = 172.872 = 2^3 \times 3^2 \times 7^4$. Berarti banyak pembaginya adalah $4 \times 3 \times 5 = 60$.

Secara umum, banyaknya pembagi dari:

$$N = a_1^{p_1} \times a_2^{p_2} \times a_3^{p_3} \times \dots \times a_k^{p_k}$$

adalah:

$$(1 + p_1) \times (1 + p_2) \times (1 + p_3) \times \dots \times (1 + p_k)$$

Jadi pencarian banyaknya pembagi dapat dilakukan dengan melakukan faktorisasi prima pada N , lalu hitung banyak pembaginya dengan rumus tersebut. Anda cukup melakukan pengecekan sampai \sqrt{N} saja untuk melakukan faktorisasi bilangan N .

Berikut adalah implementasi solusi ini dalam C++:

```
#include <iostream>

using namespace std;

int main() {
    long long N;
    cin >> N;

    long long num = N;
    int divisorCount = 1;
    for (long long i = 2; i*i <= num; i++) {
        int factorCount = 0;
        while (num % i == 0) {
            factorCount++;
            num /= i;
        }

        divisorCount *= (1 + factorCount);
    }
    if (num > 1) { // Sisa faktor
        divisorCount *= 2;
    }

    if (divisorCount % 2 == 0) {
        cout << "lampu_mati" << endl;
    } else {
        cout << "lampu_menyala" << endl;
    }
}
```

Karena pengecekan hanya dilakukan hingga \sqrt{N} saja untuk melakukan faktorisasi bilangan N , maka perhitungan dapat dilakukan di bawah 1 detik untuk N sampai 10^{16} . Namun, untuk 10^{18} , kemungkinan program masih memerlukan waktu di atas 1 detik.

Solusi yang Lebih Baik Lagi!

Ternyata, masih ada solusi yang lebih baik dibandingkan dengan solusi menggunakan faktorisasi prima. Perhatikan bahwa kita tidak perlu tahu berapa banyak pembagi bilangan N . Kita hanya perlu tahu apakah N memiliki pembagi sebanyak ganjil atau genap.

Jika suatu bilangan, sebut saja x membagi habis N , maka otomatis $\frac{N}{x}$ juga akan membagi habis N . Perhatikan contoh berikut:

Jika $N = 12$, maka:

- Karena 1 membagi habis 12, maka $\frac{12}{1} = 12$ juga membagi habis 12
- Karena 2 membagi habis 12, maka $\frac{12}{2} = 6$ juga membagi habis 12

- Karena 3 membagi habis 12, maka $\frac{12}{3} = 4$ juga membagi habis 12

Dengan demikian, secara umum pembagi bilangan selalu memiliki pasangan, kecuali jika N merupakan bilangan kuadrat, misalnya 4, 9, 16 dan seterusnya. Mengapa demikian? Jika N merupakan bilangan kuadrat, artinya N dapat dibagi habis oleh \sqrt{N} . Khusus pembagi akar ini tidak memiliki pasangan, karena $\frac{N}{\sqrt{N}} = \sqrt{N}$. Perhatikan contoh berikut:

Jika $N = 16$, maka:

- Karena 1 membagi habis 16, maka $\frac{16}{1} = 16$ juga membagi habis 16
- Karena 2 membagi habis 16, maka $\frac{16}{2} = 8$ juga membagi habis 16
- 4 membagi habis 16, tetapi karena $\frac{16}{4}$ juga 4, maka pembagi ini tidak memiliki pasangan.

Maka kita bisa simpulkan bahwa lampu menyala jika dan hanya jika N merupakan bilangan kuadrat. Untuk menentukan apakah N merupakan bilangan kuadrat, kita dapat menghitung pembulatan dari \sqrt{N} , sebut saja s . Kemudian menghitung $s \times s$ lagi. Jika kita mendapatkan kembali N , maka N adalah bilangan kuadrat.

Sebagai contoh:

- $N = 8$, maka $\sqrt{8} \approx 2.828427$, dibulatkan menjadi 3. Hasil dari 3×3 bukan 8, maka 8 bukan bilangan kuadrat.
- $N = 9$, maka $\sqrt{9} = 3$, dibulatkan menjadi 3. Hasil dari 3×3 adalah 9, maka 9 adalah bilangan kuadrat.
- $N = 10$, maka $\sqrt{10} \approx 3.162277$, dibulatkan menjadi 3. Hasil dari 3×3 bukan 10, maka 10 bukan bilangan kuadrat.

Implementasinya sangat sederhana:

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    long long N;
    cin >> N;

    long long s = round(sqrt(N));

    if (s * s != N) {
        cout << "lampu_mati" << endl;
    } else {
        cout << "lampu_menyala" << endl;
    }
}
```

Program ini tidak memiliki perulangan sama sekali, dan hanya memanggil fungsi bawaan C++ untuk mengakarkan bilangan. Program di atas dapat menyelesaikan permasalahan hingga kasus terbesar $N = 10^{18}$ hanya dalam hitungan milisekon saja!

Melalui penyelesaian soal ini, kita mempelajari bahwa solusi yang naif belum tentu cukup untuk menyelesaikan soal. Bahkan, solusi yang lebih rumit belum tentu mendapatkan penilaian sempurna. Dibutuhkan analisis yang mendalam hingga didapatkan strategi penyelesaian yang efisien.

Mengenal Kompetisi Pemrograman

Ada banyak kompetisi pemrograman dengan tingkatan yang beragam. Pada subbab ini, kita akan fokus pada 2 format kompetisi yang cukup besar, yaitu format IOI dan format ACM-ICPC.

Olimpiade Sains Nasional dan *International Olympiad in Informatics*

Olimpiade Sains Nasional (OSN) merupakan kompetisi tahunan yang diselenggarakan oleh Kementerian Pendidikan dan Kebudayaan Indonesia. OSN merupakan kompetisi nasional untuk siswa SMA dan setingkat. Terdapat beberapa bidang pada OSN, termasuk salah satunya bidang informatika. Untuk dapat berpartisipasi pada OSN, Anda harus terlebih dahulu mengikuti dan lolos pada Olimpiade Sains Kabupaten dan Olimpiade Sains Provinsi pada bidang yang bersangkutan.

International Olympiad of Informatics (IOI) merupakan kompetisi pemrograman internasional untuk SMA dan sederajat, yang mana masing-masing negara mengirimkan paling banyak 4 perwakilan. Khusus Indonesia, 4 perwakilan akan dipilih setelah melalui 3 tahap seleksi pada Pelatihan Nasional (Pelatnas). Mulanya, 30 siswa-siswi terbaik OSN akan diikutkan Pelatnas 1. Kemudian disaring 16 besar untuk diikutkan ke Pelatnas 2. Kemudian akan disaring lagi menjadi 8 besar untuk mengikuti Pelatnas 3. Terakhir, akan disaring menjadi 4 besar sebagai perwakilan dalam ajang IOI.

Format OSN dan IOI

OSN Informatika maupun IOI diselenggarakan dalam 3 hari. Hari pertama merupakan hari uji coba, yang mana Anda akan diberikan soal-soal pemrograman kompetitif yang relatif mudah. Poin yang didapatkan pada hari pertama ini tidak dihitung sama sekali untuk penentuan hasil akhir. Kompetisi sesungguhnya ada pada hari kedua dan ketiga. Pada masing-masing hari, Anda akan diberikan 3-4 soal pemrograman yang harus diselesaikan selama 5 jam. Peringkat akhir akan dilihat dari total poin yang didapatkan setiap peserta. Waktu, durasi, dan urutan pengerjaan soal tidak mempengaruhi penilaian akhir.²

Bentuk Soal

Soal OSN dan IOI akan dibagi menjadi beberapa **subsoal**. Setiap subsoal memiliki batasan-batasan masukan khusus dan poin masing-masing. Untuk mendapatkan nilai sempurna pada suatu soal, program Anda harus bisa menyelesaikan masalah untuk setiap subsoal. Sebagai contoh, perhatikan soal berikut.

Contoh Soal 1.2: Lampu dan Tombol versi 2

Terdapat N tombol yang dinomori dari 1 hingga N dan sebuah lampu dalam keadaan mati. Apabila tombol ke- i ditekan, keadaan lampu akan berubah (dari mati menjadi menyala, atau sebaliknya) apabila N habis dibagi oleh i . Apabila masing-masing tombol ditekan tepat sekali, bagaimana keadaan lampu pada akhirnya?

Batasan

Subsoal 1 (30 Poin)

$$1 \leq N \leq 10^6$$

Subsoal 2 (40 Poin)

$$1 \leq N \leq 10^{14}$$

Subsoal 3 (30 Poin)

$$1 \leq N \leq 10^{18}$$

Soal ini sama dengan soal Lampu dan Tombol. Kali ini, terdapat 3 subsoal berbeda. Dengan mengimplementasikan solusi naif, Anda hanya akan mendapatkan 30 dari total 100 poin.

Selain itu, terdapat 3 jenis soal yang umum digunakan pada kompetisi setipe OSN/IOI:

1. *Batch*, yang merupakan jenis soal yang paling umum. Anda diminta untuk membuat program yang membaca masukan dan mencetak keluaran. Kasus uji untuk masukan dan keluaran setiap soal telah disediakan oleh tim juri, dan bersifat rahasia. Program Anda mendapatkan nilai apabila keluaran yang dihasilkan sesuai dengan keluaran dari tim juri. Soal Lampu dan Tombol merupakan contoh soal bertipe *batch*.
2. *Interaktif*, yang mana Anda diminta membuat program untuk berinteraksi dengan program dari tim juri. Anda mendapatkan nilai apabila suatu tujuan dari interaksi tercapai.
3. *Output only*, yang mirip dengan jenis soal *batch*. Perbedaannya adalah Anda diberikan masukan untuk seluruh kasus uji, dan Anda hanya perlu mengumpulkan keluaran dari masing-masing kasus uji tersebut. Dengan demikian, waktu eksekusi dan memori yang digunakan hanya terbatas pada waktu kompetisi dan kapasitas mesin yang Anda gunakan.

Kurikulum OSN

Secara umum, kurikulum OSN mengikuti kurikulum dari IOI. Buku ini kami rancang berdasarkan kurikulum OSN. Setiap materi-materi yang dapat diujikan pada OSN tercakup dalam buku ini.

ACM-ICPC

ACM-ICPC merupakan kompetisi pemrograman tingkat internasional yang ditujukan untuk mahasiswa. Kompetisi ini dilakukan secara tim yang terdiri dari 3 orang. ACM-ICPC terdiri dari ACM-ICPC Regionals dan ACM-ICPC World Finals. Anda harus mengikuti kontes regional terlebih dahulu. Jika Anda mendapatkan peringkat atas pada regional, Anda akan diundang untuk mengikuti ACM-ICPC World Finals.

Format ACM-ICPC

ACM-ICPC diselenggarakan dalam 1 hari. Waktu kontes ACM-ICPC adalah 5 jam, sama seperti OSN. Namun bedanya, jumlah soal yang diberikan jauh lebih banyak, biasanya berkisar antara 7 sampai 12 soal. Selain itu, ACM-ICPC tidak mengenal penilaian parsial. Dengan kata lain, tidak terdapat subsoal pada soal-soal ACM-ICPC.

Penilaian akhir dihitung dari banyaknya soal yang berhasil diselesaikan (mendapatkan *accepted*). Jika ada 2 tim dengan jumlah *accepted* yang sama, maka urutan ditentukan berdasarkan penalti waktu. Nilai penalti dihitung dari waktu saat soal terselesaikan, ditambah dengan beberapa poin untuk setiap pengumpulan solusi yang tidak *accepted*.³

Kurikulum ACM-ICPC

Kurikulum ACM-ICPC sangat luas. Buku ini mencakup materi-materi dasar dan cocok jika Anda baru belajar pemrograman kompetitif. Untuk mempersiapkan diri lebih dalam, disarankan untuk mencari referensi-referensi tambahan seperti buku *Competitive Programming 3*⁴ maupun forum-forum diskusi pemrograman kompetitif, juga diimbangi dengan latihan.

Penulisan Kode

Terdapat perbedaan antara kode program yang ditulis untuk kompetisi dan industri. Perbedaan ini disebabkan karena lainnya karakter program untuk kedua kebutuhan tersebut.

Program yang ditulis pada pemrograman kompetitif biasanya berumur pendek. Artinya, program yang ditulis tidak perlu diurus dan tidak akan digunakan untuk kebutuhan pada masa mendatang. Lain halnya dengan program yang ditujukan untuk industri, yang mana program tetap digunakan dan perlu dimodifikasi seiring dengan berjalannya waktu. Program pada dunia industri juga harus memperhatikan integrasi dengan komponen program lainnya.

Oleh sebab itu, kode program untuk kompetisi tidak perlu menomorsatukan faktor-faktor yang mempermudah pemeliharaan kode seperti kemudahan dibaca (*readability*) dan struktur yang memaksimalkan daya daur ulang (*reusability*). Sebagai tambahan, mengurangi perhatian pada faktor-faktor tersebut akan membantu penulisan kode pada kompetisi dengan lebih cepat. Namun perlu diperhatikan bahwa kode sebaiknya ditulis dengan rapi agar mudah dibaca, dan menghindari adanya kesalahan program (*bug*).

Kode program yang ditulis pada buku ini akan menggunakan gaya kompetitif, yang biasanya bersifat prosedural. Kode akan ditulis sebagaimana adanya dengan fokus untuk menunjukkan alur algoritma.

Perkenalan *Pseudocode*

Pada buku ini, kode akan dituliskan dengan *pseudocode*. *Pseudocode* merupakan **bahasa informal** yang mirip dengan bahasa pemrograman untuk mendeskripsikan program. Bahasa ini biasa digunakan pada materi pembelajaran algoritma, sehingga pembaca tidak perlu mempelajari suatu bahasa pemrograman tertentu. *Pseudocode* sendiri bukanlah bahasa pemrograman sungguhan.

Algoritma 1 Contoh *pseudocode* algoritma *insertion sort* untuk mengurutkan array A yang memiliki panjang N .

```

1: procedure INSERTIONSORT( $A, N$ )
2:   for  $i \leftarrow 2, N$  do
3:      $j \leftarrow i$ 
4:     while  $(j > 1) \wedge (A[j] < A[j-1])$  do      ▷ Selama  $A[j]$  masih dapat dipindah ke
        depan
5:       SWAP( $A[j], A[j-1]$ )
6:        $j \leftarrow j-1$ 
7:     end while
8:   end for
9: end procedure

```

Pada Algoritma 1, Anda dapat memperhatikan bahwa *pseudocode* cukup ekspresif dalam menyatakan alur algoritma. Sebagai konvensi, simbol-simbol beserta makna untuk komponen pada *pseudocode* dapat Anda lihat pada Tabel 1.1.

Tabel 1.1: Simbol dan kata kunci beserta makna pada *pseudocode*.

Simbol/kata kunci	Makna
\leftarrow	Operator <i>assignment</i> , artinya memberikan nilai pada variabel di ruas kiri dengan nilai di ruas kanan.
$=$	Operator <i>boolean</i> untuk memeriksa kesamaan. Nilai <i>true</i> dikembalikan bila kedua ruas sama, atau <i>false</i> jika tidak.
\wedge	Operator logika <i>and</i> .
\vee	Operator logika <i>or</i> .
$\lceil x \rceil$	Pembulatan ke atas untuk suatu bilangan riil x .
$\lfloor x \rfloor$	Pembulatan ke bawah untuk suatu bilangan riil x .
$ X $	Banyaknya anggota atau ukuran dari X , yang mana X merupakan sebuah himpunan atau daftar.
new integer [10]	Inisialisasi <i>array</i> bertipe <i>integer</i> dengan 10 elemen, yang terdefinisi dari elemen ke-0 sampai dengan 9.
for $i \leftarrow 1, N$ do	Perulangan <i>for</i> dengan i sebagai pencacah, dimulai dari angka 1 sampai dengan N .
return	Perintah untuk pengembalian pada fungsi dan prosedur, lalu keluar dari fungsi atau prosedur tersebut. Pada fungsi, nilai yang dituliskan sesudah return akan dikembalikan ke pemanggilnya.

2 Matematika Diskret Dasar

Matematika diskret merupakan cabang matematika yang mempelajari tentang sifat bilangan bulat, logika, kombinatorika, dan graf. Ilmu pada jenis matematika ini digunakan sebagai dasar ilmu komputer secara umum. Pada bagian ini, kita akan mencakup dasar ilmu matematika diskret yang umum digunakan dalam pemrograman kompetitif.

Aritmetika Modular

Modulo merupakan suatu operator matematika untuk mendapatkan sisa bagi suatu bilangan terhadap suatu bilangan lainnya. Operasi modulo bisa dilambangkan dengan `mod` pada bahasa Pascal atau `%` pada bahasa C/C++ atau Java. Operasi $a \bmod m$ biasa dibaca " a modulo m ", dan memberikan sisa hasil bagi a oleh m . Sebagai contoh:

- $5 \bmod 3 = 2$
- $10 \bmod 2 = 0$
- $21 \bmod 6 = 3$

Sifat-sifat dasar dari operasi modulo adalah:

- $(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$
- $(a - b) \bmod m = ((a \bmod m) - (b \bmod m)) \bmod m$
- $(a \times b) \bmod m = ((a \bmod m) \times (b \bmod m)) \bmod m$
- $a^b \bmod m = (a \bmod m)^b \bmod m$
- $(-a) \bmod m = (-(a \bmod m) + m) \bmod m$

Sifat-sifat tersebut dapat dimanfaatkan untuk mempermudah perhitungan modulo. Sebagai contoh, Anda diberikan bilangan n dan k , lalu diminta menghitung hasil $n! \bmod k$. Pada contoh ini, $n! = n \times (n-1) \times (n-2) \times \dots \times 1$. Seandainya kita menghitung $n!$ terlebih dahulu, kemudian baru dimodulo k , kemungkinan besar kita akan mendapatkan *integer overflow*. Ingat bahwa $n!$ dapat bernilai sangat besar dan tidak dapat direpresentasikan dengan tipe data primitif *integer*. Solusinya adalah melakukan modulo pada setiap fase perkalian. Contoh implementasi dapat dilihat pada Algoritma 2. Perhatikan pada baris ke-4, hasil perkalian selalu dimodulo untuk mencegah *integer overflow*.

Algoritma 2 Fungsi faktorial dengan memanfaatkan modulo.

```
1: function MODULARFACTORIAL( $n, k$ )
2:   result  $\leftarrow$  1
3:   for  $i \leftarrow 1, n$  do
4:     result  $\leftarrow$  (result  $\times$   $i$ ) mod  $k$ 
5:   end for
6:   return result
7: end function
```

Anda perlu berhati-hati pada kasus pembagian, karena aritmetika modular tidak serta-merta bekerja. Diketahui sifat berikut:

$$\frac{a}{b} \bmod m \neq \left(\frac{a \bmod m}{b \bmod m} \right) \bmod m.$$

Contoh:

$$\frac{12}{4} \bmod 6 \neq \left(\frac{12 \bmod 6}{4 \bmod 6} \right) \bmod 6.$$

Pada aritmetika modular, $\frac{a}{b} \bmod m$ biasa ditulis sebagai:

$$(a \times b^{-1}) \bmod m$$

dengan b^{-1} adalah **modular multiplicative inverse** dari b . Perhitungan *Modular multiplicative inverse* tidak dibahas pada buku ini. Namun, jika Anda penasaran, *modular multiplicative inverse* dapat dihitung menggunakan algoritma *extended euclid*⁵ atau memanfaatkan teorema Euler.⁶

Bilangan Prima

Bilangan prima merupakan bilangan bulat positif yang tepat memiliki dua faktor (pembagi), yaitu 1 dan dirinya sendiri. Beberapa contoh bilangan prima adalah 2, 3, 5, 13, 97.

Lawan dari bilangan prima adalah **bilangan komposit**. Bilangan komposit adalah bilangan yang memiliki lebih dari dua faktor. Contoh dari bilangan komposit adalah 6, 14, 20, 25.

Uji Keprimaan (*Primality Testing*)

Aktivitas untuk mengecek apakah suatu bilangan bulat N adalah bilangan prima disebut dengan **uji keprimaan** (*primality testing*). Solusi yang mudah untuk mengecek apakah N merupakan bilangan prima dapat dilakukan dengan mengecek apakah ada bilangan selain 1 dan N yang habis membagi N . Kita dapat melakukan iterasi dari 2 hingga $N - 1$ untuk mengetahui apakah ada bilangan selain 1 dan N yang habis membagi N . Kompleksitas waktu untuk solusi ini adalah $O(N)$. Solusi ini dapat dilihat pada Algoritma 3.

Terdapat solusi uji keprimaan yang lebih cepat daripada $O(N)$. Manfaatkan observasi bahwa jika $N = a \times b$, dan $a \leq b$, maka $a \leq \sqrt{N}$ dan $b \geq \sqrt{N}$. Kita tidak perlu memeriksa b ; seandainya N habis dibagi b , tentu N habis dibagi a . Jadi kita hanya perlu memeriksa hingga \sqrt{N} . Jadi kompleksitas waktunya adalah $O(\sqrt{N})$. Solusi ini dapat dilihat pada Algoritma 4.

Pembangkitan Bilangan Prima (*Prime Generation*)

Aktivitas untuk mendapatkan bilangan-bilangan prima disebut dengan pembangkitan bilangan prima. Misalnya, kita ingin mengetahui himpunan bilangan prima yang tidak lebih daripada N .

Algoritma 3 Uji keprimaan dengan melakukan pengecekan dari 2 sampai $N-1$.

```

1: function ISPRIMEAIVE( $N$ )
2:   if  $N \leq 1$  then
3:     return false
4:   end if
5:   for  $i \leftarrow 2, N-1$  do
6:     if  $N \bmod i = 0$  then
7:       return false
8:     end if
9:   end for
10:  return true
11: end function

```

Algoritma 4 Uji keprimaan dengan melakukan pengecekan dari 2 sampai \sqrt{N} .

```

1: function ISPRIMESQRT( $N$ )
2:   if  $N \leq 1$  then
3:     return false
4:   end if
5:    $i \leftarrow 2$ 
6:   while  $i \times i \leq N$  do           ▷ Pertidaksamaan ini ekuivalen dengan  $i \leq \sqrt{N}$ 
7:     if  $N \bmod i = 0$  then
8:       return false
9:     end if
10:     $i \leftarrow i + 1$ 
11:  end while
12:  return true
13: end function

```

Algoritma 5 Pembangkitan Bilangan Prima dengan iterasi dan uji keprimaan.

```

1: function SIMPLEPRIMEGENERATION( $N$ )
2:   primeList  $\leftarrow \{\}$ 
3:   for  $i \leftarrow 2, N$  do
4:     if ISPRIMESQRT( $i$ ) then
5:       primeList  $\leftarrow$  primeList  $\cup \{i\}$ 
6:     end if
7:   end for
8:   return primeList
9: end function

```

Pembangkitan Prima Menggunakan Iterasi

Solusi sederhana dari pembangkitan bilangan bilangan prima adalah dengan iterasi dan uji keprimaan, seperti yang tertera pada Algoritma 5.

Pada metode pembangkitan bilangan prima menggunakan iterasi, kita akan menguji keprimaan setiap angka dari 2 hingga N . Uji keprimaan suatu bilangan X yang mana $X \leq N$ memiliki kompleksitas $O(\sqrt{X})$. Kita tahu bahwa $\sqrt{X} \leq \sqrt{N}$, dan kita akan melakukan pengujian prima sebanyak N bilangan. Maka kompleksitas akhirnya adalah $O(N\sqrt{N})$.

Sieve of Eratosthenes

Terdapat solusi yang lebih cepat untuk membangkitkan bilangan prima, yaitu saringan Eratosthenes, atau **Sieve of Eratosthenes**. Ide utama utama dari algoritma ini adalah mengeliminasi bilangan-bilangan dari calon bilangan prima, yaitu bilangan komposit.

Kita tahu bahwa suatu bilangan komposit c dapat dinyatakan sebagai $c = p \times q$, dengan p suatu bilangan prima. Seandainya kita mengetahui suatu bilangan prima, kita dapat mengeliminasi kelipatan-kelipatan bilangan tersebut dari calon bilangan prima.

Contoh: jika diketahui 7 adalah bilangan prima, maka 14, 21, 28, ... dieliminasi dari calon bilangan prima.

Berikut prosedur dari Sieve of Eratosthenes:

1. Awalnya seluruh bilangan bulat dari 2 hingga N belum dieliminasi.
2. Lakukan iterasi dari 2 hingga N :
 - a) Jika bilangan ini belum dieliminasi, artinya bilangan ini merupakan bilangan prima.
 - b) Lakukan iterasi untuk mengeliminasi kelipatan bilangan tersebut.

Algoritma 6 menunjukkan implementasi Sieve of Eratosthenes. Untuk mengimplementasikannya, kita dapat menggunakan *array boolean* untuk menyimpan informasi apakah suatu bilangan telah tereliminasi. Untuk mencari bilangan prima yang $\leq N$, diperlukan memori sebesar $O(N)$. Untuk kompleksitas waktu, perhitungan matematis⁷ membuktikan bahwa solusi bekerja dalam $O(N \log(\log N))$.

Perhatikan baris ke-8 yang berisi $j = i \times i$. Di sini, j menyatakan kelipatan i yang akan dieliminasi. Perhatikan bahwa j dimulai dari $i \times i$, bukan $2i$. Alasannya adalah $2i, 3i, 4i, \dots, (i-1)i$ pasti sudah tereliminasi pada iterasi-iterasi sebelumnya.

FPB dan KPK

Ketika masih SD, kita pernah belajar memfaktorkan bilangan dengan pohon faktor. Melalui faktorisasi prima, kita dapat menyatakan suatu bilangan sebagai hasil perkalian faktor primanya. Contoh: $7875 = 3^2 \times 5^3 \times 7$.

Faktor Persekutuan Terbesar (FPB) dan Kelipatan Persekutuan Terkecil (KPK) dapat dicari melalui faktorisasi prima. Untuk setiap bilangan prima, kita menggunakan pangkat terkecil untuk FPB dan pangkat terbesar untuk KPK.

Sebagai contoh, diketahui:

- $4725 = 3^3 \times 5^2 \times 7$

Algoritma 6 Sieve of Eratosthenes

```

1: function SIEVEOFERATOSTHENES(N)
2:   eliminated ← new boolean[N + 1]           ▷ Siapkan array boolean eliminated
3:   FILLARRAY(eliminated, false)             ▷ Isi seluruh elemen eliminated dengan false
4:   primeList ← {}
5:   for i ← 2, N do
6:     if not eliminated[i] then
7:       primeList ← primeList ∪ {i}
8:       j ← i × i
9:       while j ≤ n do
10:        eliminated[j] ← true
11:        j ← j + i
12:       end while
13:     end if
14:   end for
15:   return primeList
16: end function

```

- $7875 = 3^2 \times 5^3 \times 7$

Maka:

- $FPB(4725, 7875) = 3^2 \times 5^2 \times 7 = 1525$
- $KPK(4725, 7875) = 3^3 \times 5^3 \times 7 = 23625$

Sebagai catatan, terdapat pula sifat:

$$KPK(a, b) = \frac{a \times b}{FPB(a, b)}$$

Pencarian FPB suatu bilangan menggunakan pohon faktor cukup merepotkan. Kita perlu mencari faktor prima bilangan tersebut, dan jika faktor primanya besar, tentu akan menghabiskan banyak waktu. Terdapat algoritma yang dapat mencari $FPB(a, b)$ dalam $O(\log(\min(a, b)))$, yaitu **Algoritma Euclid**.⁵

Algoritma ini sangat pendek, dan dapat diimplementasikan dengan mudah seperti pada Algoritma 7.

Algoritma 7 Algoritma Euclid untuk mencari FPB secara rekursif.

```

1: function EUCLID(a, b)
2:   if b = 0 then
3:     return a
4:   else
5:     return EUCLID(b, a mod b)
6:   end if
7: end function

```

Pigeonhole Principle (PHP)

Konsep PHP berbunyi:

Jika ada N ekor burung dan M sangkar, yang memenuhi $N > M$, maka pasti ada sangkar yang berisi setidaknya 2 ekor burung.

Secara matematis, jika ada N ekor burung dan M sangkar, maka pasti ada sangkar yang berisi setidaknya $\lceil \frac{N}{M} \rceil$ ekor burung.

Contoh Soal 2.1: Subhimpunan Terbagi

Anda diberikan sebuah *array* A berisi N bilangan bulat non-negatif. Anda ditantang untuk memilih angka-angka dari *array*-nya yang jika dijumlahkan habis dibagi N . Angka di suatu indeks *array* tidak boleh dipilih lebih dari sekali.

Apabila mungkin, cetak indeks angka-angka yang Anda ambil. Sementara apabila tidak mungkin, cetak "tidak mungkin".

Batasan

- $1 \leq N \leq 10^5$
- *Array* A hanya berisi bilangan bulat non-negatif.

Inti soal ini adalah mencari apakah pada *array* berukuran N , terdapat subhimpunan tidak kosong yang jumlah elemen-elemennya habis dibagi N .

Mari kita coba mengerjakan versi lebih mudah dari soal ini: bagaimana jika yang diminta subbarisan, bukan subhimpunan?

Anggap *array* A dimulai dari indeks 1 (*one-based*). Misalkan kita memiliki fungsi:

$$sum(k) = \sum_{i=1}^k A[i]$$

Untuk $sum(0)$, sesuai definisi nilainya adalah 0.

Perhatikan bahwa kita dapat mendefinisikan jumlah dari subbarisan $A[\ell..r]$ menggunakan sum :

$$\sum_{i=\ell}^r A[i] = sum(r) - sum(\ell - 1).$$

Jika subbarisan $A[\ell..r]$ habis dibagi N , maka $(sum(r) - sum(\ell - 1)) \bmod N = 0$. Kita dapat menuliskan ini dengan $sum(r) \bmod N = sum(\ell - 1) \bmod N$.

Observasi 1: Ada N kemungkinan nilai $(sum(x) \bmod N)$, yaitu $[0..N - 1]$.

Observasi 2: Ada $N + 1$ nilai x untuk $(sum(x) \bmod N)$, yaitu untuk $x \in [0..N]$.

Ingat bahwa $sum(0)$ ada agar jumlah subbarisan $A[1..k]$ untuk tiap k dapat kita nyatakan dalam bentuk: $(sum(k) - sum(0))$.

Observasi 3: Jika ada $N + 1$ kemungkinan nilai x dan ada N kemungkinan nilai $\text{sum}(x) \bmod N$, maka pasti ada a dan b , sehingga $\text{sum}(b) \bmod N = \text{sum}(a) \bmod N$.

Subbarisan yang menjadi solusi adalah $A[a + 1..b]$. Dengan observasi ini, kita menyelesaikan versi mudah dari soal awal kita.

Dengan menyelesaikan versi mudah, ternyata kita justru dapat menyelesaikan soal tersebut. Suatu subbarisan dari A pasti juga merupakan subhimpunan dari A . Ternyata, *selalu* ada cara untuk menjawab pertanyaan Pak Dengklek. Yakni, keluaran "tidak mungkin" tidak akan pernah terjadi. Implementasinya cukup sederhana: lihat Algoritma 8.

Algoritma 8 Solusi persoalan Subhimpunan Terbagi.

```

1: function FINDDIVISIBLESUBSEQUENCE( $A, N$ )
2:    $\text{sum} \leftarrow$  new integer $[N + 1]$                                 ▷ Inisialisasi array  $\text{sum}[0..N]$ 
3:    $\text{sum}[0] \leftarrow 0$ 
4:   for  $i \leftarrow 1, N$  do
5:      $\text{sum}[i] \leftarrow \text{sum}[i - 1] + A[i]$ 
6:   end for

7:    $\text{seenInIndex} \leftarrow$  new integer $[N]$                             ▷ Inisialisasi array  $\text{seenInIndex}[0..N - 1]$ 
8:   FILLARRAY( $\text{seenInIndex}, -1$ )                                     ▷ Isi  $\text{seenInIndex}$  dengan  $-1$ 

9:   for  $i \leftarrow 0, N$  do
10:    if  $\text{seenInIndex}[\text{sum}[i] \bmod N] = -1$  then
11:       $\text{seenInIndex}[\text{sum}[i] \bmod N] \leftarrow i$ 
12:    else
13:       $a \leftarrow \text{seenInIndex}[\text{sum}[i] \bmod N]$ 
14:       $b \leftarrow i$ 
15:      return  $[a + 1, a + 2, \dots, b]$                                ▷ Kembalikan indeks-indeks solusinya
16:    end if
17:  end for
18: end function

```

Kombinatorika

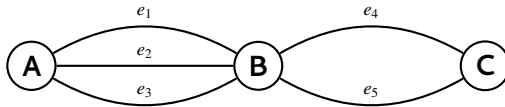
Kombinatorika adalah ilmu yang mempelajari penyusunan dari objek-objek,⁹ seperti menghitung banyaknya kemungkinan dari susunan objek menurut aturan tertentu. Mampu memahami sifat-sifat dari penyusunan objek dapat membantu pemecahan masalah dengan algoritma. Terdapat beberapa dasar teknik menghitung banyaknya penyusunan objek, yang akan dibahas pada bagian ini.

Aturan Perkalian dan Aturan Penjumlahan

Contoh Soal 2.2: Kota 1

Terdapat 3 buah kota yaitu A, B, dan C. Kota A dan kota B terhubung oleh 3 jalur berbeda yaitu e_1 , e_2 , dan e_3 . Sedangkan kota B dan kota C terhubung oleh 2 jalur

berbeda yaitu e_4 dan e_5 . Berapa banyak cara berbeda untuk menuju kota C dari kota A?



Apabila kita hitung satu per satu, maka cara yang berbeda untuk menuju kota C dari kota A adalah sebagai berikut:

- Melalui jalur e_1 kemudian jalur e_4 .
- Melalui jalur e_1 kemudian jalur e_5 .
- Melalui jalur e_2 kemudian jalur e_4 .
- Melalui jalur e_2 kemudian jalur e_5 .
- Melalui jalur e_3 kemudian jalur e_4 .
- Melalui jalur e_3 kemudian jalur e_5 .

Dengan kata lain, terdapat 6 cara berbeda untuk menuju kota C dari kota A. Namun, apabila jumlah kota dan jalur yang ada sangatlah banyak, kita tidak mungkin menulis satu per satu cara yang berbeda. Karena itulah kita dapat menggunakan **aturan perkalian**.

Misalkan suatu proses dapat dibagi menjadi N subproses independen yang mana terdapat a_i cara untuk menyelesaikan subproses ke- i . Menurut aturan perkalian, banyak cara yang berbeda untuk menyelesaikan proses tersebut adalah $a_1 \times a_2 \times a_3 \times \dots \times a_N$.

Kita akan mencoba mencari banyaknya cara menuju kota C dari kota A menggunakan aturan perkalian. Anggaplah bahwa:

- Perjalanan dari kota A menuju kota B merupakan subproses pertama, yang mana terdapat 3 cara untuk menyelesaikan subproses tersebut.
- Perjalanan dari kota B menuju kota C merupakan subproses kedua, yang mana terdapat 2 cara untuk menyelesaikan subproses tersebut.

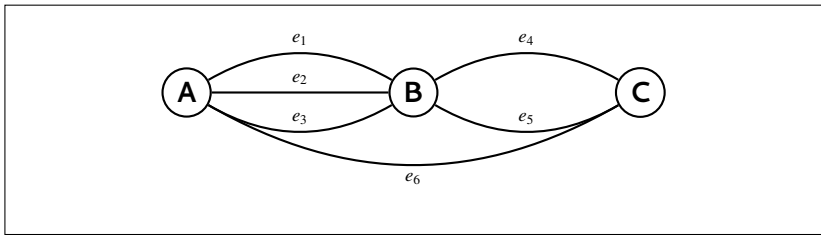
Karena perjalanan dari kota A menuju kota B dan dari kota B menuju kota C merupakan 2 subproses yang berbeda, maka kita dapat menggunakan aturan perkalian. Banyak cara berbeda dari kota A menuju kota C adalah $3 \times 2 = 6$.

Contoh Soal 2.3: Kota 2

Contoh soal ini merupakan lanjutan dari Kota 1.

Deskripsi soal, jumlah kota dan jalur serta susunan jalur yang ada sama persis dengan soal tersebut.

Tambahkan 1 jalur lagi, yaitu e_6 yang menghubungkan kota A dan C. Berapa banyak cara berbeda untuk menuju kota C dari kota A?



Dengan mencoba satu per satu setiap cara, maka terdapat 7 cara yang berbeda, yaitu 6 cara sesuai dengan soal sebelumnya, ditambah dengan menggunakan jalur e_6 .

Apabila kita menggunakan aturan perkalian, maka didapatkan banyak cara yang berbeda adalah $3 \times 2 \times 1 = 6$ yang mana jawaban tersebut tidaklah tepat. Kita tidak dapat menggunakan aturan perkalian dalam permasalahan ini, karena antara perjalanan dari kota A menuju kota C melalui kota B dengan tanpa melalui kota B merupakan 2 proses yang berbeda. Oleh karena itu, kita dapat menggunakan **aturan penjumlahan**.

Misalkan suatu proses dapat dibagi menjadi N himpunan proses berbeda yaitu $H_1, H_2, H_3, \dots, H_N$ dengan setiap himpunannya saling lepas (tidak beririsan). Menurut aturan penjumlahan, banyak cara yang berbeda untuk menyelesaikan proses tersebut adalah $|H_1| + |H_2| + |H_3| + \dots + |H_N|$ dengan $|H_i|$ merupakan banyaknya cara berbeda untuk menyelesaikan proses ke- i .

Kita akan mencoba mencari banyaknya cara menuju kota C dari kota A menggunakan aturan perkalian. Proses perjalanan dari kota A menuju kota C dapat kita bagi menjadi 2 himpunan proses yang berbeda, yaitu:

- Dari kota A menuju kota C melalui kota B, yang dapat kita dapatkan dengan aturan perkalian seperti yang dibahas pada permasalahan sebelumnya, yaitu 6 cara berbeda, dan
- Dari kota A langsung menuju kota C, yang mana terdapat 1 cara, yaitu melalui jalur e_6 .

Dengan aturan penjumlahan, banyak cara berbeda dari kota A menuju kota C adalah $6 + 1 = 7$ cara berbeda.

Untuk aturan penjumlahan, hati-hati apabila terdapat irisan dari himpunan proses tersebut. Ketika terdapat irisan, maka solusi yang kita dapatkan dengan aturan penjumlahan menjadi tidak tepat, karena ada solusi yang terhitung lebih dari sekali. Agar solusi tersebut menjadi tepat, gunakan **Prinsip Inklusi-Eksklusi** pada teori himpunan.

Permutasi

Permutasi adalah pemilihan urutan beberapa elemen dari suatu himpunan. Untuk menyelesaikan soal-soal permutasi, dibutuhkan pemahaman konsep faktorial.

Sebagai pengingat, faktorial dari N , atau dinotasikan sebagai $N!$, merupakan hasil perkalian dari semua bilangan asli kurang dari atau sama dengan N . Fungsi faktorial dirumuskan sebagai berikut:

$$N! = N \times (N-1) \times (N-2) \times \dots \times 3 \times 2 \times 1$$

Dengan $0! = 1$.

Contoh: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

Sebelum masuk ke permutasi, kita perlu mengetahui tentang **aturan pembagian** yang terkadang disebut juga sebagai redundansi. Menurut aturan pembagian, apabila terdapat K susunan cara berbeda yang kita anggap merupakan 1 cara yang sama, maka kita dapat membagi total keseluruhan cara dengan K , sehingga K cara tersebut dianggap sama sebagai 1 cara.

Sebagai contoh, banyak kata berbeda yang disusun dari huruf-huruf penyusun "TOKI" adalah $4!$ (menggunakan aturan perkalian). Apabila kita ganti soal tersebut, yaitu kata berbeda yang disusun dari huruf-huruf penyusun "BACA", solusi $4!$ merupakan solusi yang salah. Sebab, terdapat 2 buah huruf 'A'. Sebagai contoh: BA_1CA_2 dan BA_2CA_1 pada dasarnya merupakan kata yang sama. Terdapat $2!$ cara berbeda tetapi yang kita anggap sama, yaitu penggunaan A_1A_2 dan A_2A_1 . Sehingga menurut aturan pembagian, banyak kata berbeda yang dapat kita bentuk dari huruf-huruf penyusun kata "BACA" adalah:

$$\frac{4!}{2!} = \frac{24}{2} = 12$$

Contoh Soal 2.4: Kompetisi

Terdapat 5 anak (sebut saja A, B, C, D, dan E) yang sedang mengikuti sebuah kompetisi. Dalam kompetisi tersebut akan diambil 3 peserta sebagai pemenang. Berapa banyak susunan pemenang yang berbeda dari kelima orang tersebut?

Anggap bahwa kita mengambil semua anak sebagai pemenang, sehingga terdapat $5! = 120$ susunan pemenang yang berbeda (ABCDE, ABCED, ABDCE, ..., EDCBA). Apabila kita hanya mengambil 3 peserta saja, perhatikan bahwa terdapat 2 cara berbeda yang kita anggap sama. Contoh: (ABC)DE dan (ABC)ED merupakan cara yang sama, karena 3 peserta yang menang adalah A, B, dan C. Dengan menggunakan aturan pembagian, maka banyak susunan pemenang yang berbeda adalah:

$$\frac{5!}{2!} = \frac{120}{2} = 60$$

Secara umum, apabila terdapat N anak dan kita mengambil semua anak sebagai pemenang, maka terdapat $N!$ susunan cara berbeda. Tetapi apabila kita hanya mengambil R anak saja, maka akan terdapat $(N-R)!$ susunan berbeda yang kita anggap sama. Dengan aturan pembagian, banyak susunan berbeda adalah:

$$\frac{N!}{(N-R)!}$$

Nilah yang kita kenal dengan istilah permutasi. Secara formal, misalkan terdapat n objek dan kita akan mengambil r objek dari n objek tersebut yang mana $r < n$ dan urutan pengambilan diperhitungkan. Banyak cara pengambilan yang berbeda adalah permutasi r terhadap n :

$$P(n, r) = {}_n P_r = P_r^n = \frac{n!}{(n-r)!}.$$

Permutasi Elemen Berulang

Contoh Soal 2.5: Megagiga

Berapa banyak kata berbeda yang disusun dari huruf-huruf penyusun kata "MEGAGIGA"?

Observasi:

- Terdapat 8 huruf, sehingga banyak kata yang dapat disusun adalah $8!$.
- Terdapat 3 huruf 'G' sehingga terdapat 6 kata berbeda yang kita anggap sama ($G_1 G_2 G_3, G_1 G_3 G_2, \dots, G_3 G_2 G_1$).

Dengan aturan pembagian, maka banyak kata yang dapat disusun mengingat kesamaan kata pada huruf G adalah $\frac{8!}{3!}$. Perlu kita perhatikan pula bahwa terdapat 2 huruf A, sehingga dengan cara yang sama akan didapatkan banyak kata yang berbeda adalah:

$$\frac{8!}{3! \times 2!}$$

Secara umum, jika terdapat N huruf, banyak kata yang dapat kita susun adalah $N!$. Apabila terdapat K huruf dengan setiap hurufnya memiliki R_i huruf yang sama, maka dengan aturan pembagian banyak kata berbeda yang dapat disusun adalah:

$$\frac{N!}{R_1! \times R_2! \times R_3! \times \dots \times R_K!}$$

Inilah yang kita kenal dengan permutasi elemen berulang. Secara formal, misalkan terdapat n objek dan terdapat k objek yang mana setiap objeknya memiliki r_i elemen yang berulang, maka banyaknya cara berbeda dalam menyusun objek tersebut adalah:

$$P_{r_1, r_2, r_3, \dots, r_k}^n = \frac{n!}{r_1! \times r_2! \times r_3! \times \dots \times r_k!}$$

Permutasi Melingkar

Contoh Soal 2.6: Duduk Melingkar

Terdapat 4 anak, sebut saja A, B, C, dan D. Berapa banyak susunan posisi duduk yang berbeda apabila mereka duduk melingkar?

Banyak susunan posisi duduk yang berbeda apabila mereka duduk seperti biasa (tidak melingkar) adalah $4! = 24$. Perhatikan bahwa posisi duduk ABCD, BCDA, CDAB, dan

DABC merupakan susunan yang sama apabila mereka duduk melingkar, karena susunan tersebut merupakan rotasi dari susunan yang lainnya. Dengan kata lain terdapat 4 cara berbeda yang kita anggap sama. Dengan aturan pembagian, banyak susunan posisi duduk yang berbeda adalah:

$$\frac{24}{4} = 6$$

Secara umum, banyaknya susunan posisi duduk yang berbeda apabila N anak duduk seperti biasa (tidak melingkar) adalah $N!$. Untuk setiap konfigurasi duduk yang ada, kita dapat melakukan rotasi beberapa kali terhadap konfigurasi tersebut dan menghasilkan posisi duduk yang sama. Akan ada tepat N buah rotasi yang mungkin, misalnya untuk konfigurasi $A_1, A_2, A_3, \dots, A_{N-1}, A_N$:

- Rotasi 0 kali: $A_1, A_2, A_3, \dots, A_{N-1}, A_N$
- Rotasi 1 kali: $A_2, A_3, A_4, \dots, A_N, A_1$
- Rotasi 2 kali: $A_3, A_4, A_5, \dots, A_1, A_2$
- ...
- Rotasi $N - 1$ kali: $A_N, A_1, A_2, \dots, A_{N-2}, A_{N-1}$

Maka dengan aturan pembagian, banyak susunan posisi duduk yang berbeda adalah:

$$\frac{N!}{N} = (N - 1)!$$

Ini lah yang kita kenal dengan istilah permutasi melingkar, atau permutasi yang disusun melingkar. Banyaknya susunan yang berbeda dari permutasi melingkar terhadap n objek adalah:

$$P_{(melingkar)}^n = (n - 1)!$$

Kombinasi

Contoh Soal 2.7: Seleksi

Terdapat 5 anak (sebut saja A, B, C, D, dan E) yang mana akan dipilih 3 anak untuk mengikuti kompetisi. Berapa banyak susunan tim berbeda yang dapat dibentuk?

Soal ini berbeda dengan permutasi, karena susunan ABC dan BAC merupakan susunan yang sama, yaitu 1 tim terdiri dari A, B, dan C. Apabila kita anggap bahwa mereka merupakan susunan yang berbeda, maka banyaknya susunan tim adalah $P_2^5 = \frac{120}{2} = 60$. Untuk setiap susunan yang terdiri dari anggota yang sama akan terhitung 6 susunan berbeda yang mana seharusnya hanya dihitung sebagai 1 susunan yang sama, contohnya: ABC, ACB, BAC, BCA, CAB, CBA merupakan 1 susunan yang sama. Oleh karena itu dengan aturan pembagian, banyaknya susunan tim yang berbeda adalah $\frac{60}{6} = 10$ susunan berbeda.

Untuk secara umumnya, misalkan terdapat N anak dan akan kita ambil R anak untuk dibentuk sebagai 1 tim. Apabila urutan susunan diperhitungkan, maka banyaknya susunan

tim adalah P_R^N . Setiap susunan yang terdiri dari anggota yang sama akan terhitung $R!$ susunan berbeda yang mana seharusnya hanya dihitung sebagai 1 susunan yang sama. Dengan aturan pembagian, banyaknya susunan tim yang berbeda adalah:

$$\frac{P_R^N}{R!} = \frac{N!}{(N-R)! \times R!}$$

Nilai yang kita kenal dengan istilah **kombinasi**. Secara formal, jika terdapat n objek dan kita akan mengambil r objek dari n objek tersebut dengan $r < n$ dan urutan pengambilan tidak diperhitungkan, maka banyaknya cara pengambilan yang berbeda adalah kombinasi r terhadap n :

$$C(n, r) = {}_n C_r = C_r^n = \frac{n!}{(n-r)! \times r!}$$

Kombinasi dengan Perulangan

Contoh Soal 2.8: Beli Kue

Pak Dengklek ingin membeli kue pada toko kue yang menjual 3 jenis kue, yaitu rasa coklat, stroberi, dan kopi. Apabila Pak Dengklek ingin membeli 4 buah kue, maka berapa banyak kombinasi kue berbeda yang Pak Dengklek dapat beli?

Perhatikan bahwa membeli coklat-stroberi dengan stroberi-coklat akan menghasilkan kombinasi yang sama. Kita dapat membeli suatu jenis kue beberapa kali atau bahkan tidak membeli sama sekali. Contoh soal ini dapat dimodelkan secara matematis menjadi mencari banyaknya kemungkinan nilai A , B , dan C yang memenuhi $A + B + C = 4$ dan $A, B, C \geq 0$.

Untuk penyelesaiannya, kita dapat membagi 4 kue tersebut menjadi 3 bagian. Untuk mempermudah ilustrasi tersebut, kita gunakan lambang o yang berarti kue, dan $|$ yang berarti pembatas. Bagian kiri merupakan kue A , bagian tengah merupakan kue B , dan bagian kanan merupakan kue C . Sebagai contoh:

- $(o|o|oo)$ menyatakan 1 kue A , 1 kue B , dan 2 kue C .
- $(oo|oo|)$ menyatakan 2 kue A , 2 kue B , dan 0 kue C .

Dengan kata lain, semua susunan yang mungkin adalah:

- $(oooo|)$
- $(ooo|o|)$
- $(oo|oo|)$
- ...
- $(|oooo)$

yang tidak lain merupakan $C_2^6 = \frac{6!}{4! \times 2!} = 15$ susunan berbeda.

Secara umum, kita ingin mencari banyaknya susunan nilai berbeda dari $X_1, X_2, X_3, \dots, X_r$ yang mana $X_1 + X_2 + X_3 + \dots + X_r = n$. Untuk membagi n objek tersebut menjadi r bagian, maka akan dibutuhkan $r - 1$ buah pembatas, sehingga akan terdapat $n + r - 1$ buah objek, yang mana kita akan memilih $r - 1$ objek untuk menjadi simbol $|$. Dengan kata lain,

banyaknya susunan nilai yang berbeda adalah C_{r-1}^{n+r-1} . Inilah yang kita kenal dengan istilah **kombinasi dengan pengulangan**.

Secara formal, jika terdapat r jenis objek dan kita akan mengambil n objek, dengan tiap jenisnya dapat diambil 0 atau beberapa kali, maka banyaknya cara berbeda yang memenuhi syarat tersebut adalah sebagai berikut:

$$C_n^{n+r-1} = C_{r-1}^{n+r-1} = \frac{(n+r-1)!}{n! \times (r-1)!}$$

Segitiga Pascal

Segitiga Pascal merupakan susunan dari koefisien-koefisien binomial dalam bentuk segitiga. Nilai dari baris ke- n suku ke- r adalah C_r^n .

Contoh Segitiga Pascal:

- Baris ke-1: 1
- Baris ke-2: 1 1
- Baris ke-3: 1 2 1
- Baris ke-4: 1 3 3 1
- Baris ke-5: 1 4 6 4 1

Untuk memahami lebih dalam tentang Segitiga Pascal, simak contoh kasus berikut.

Contoh Soal 2.9: Kombinasi Subhimpunan

Diberikan suatu himpunan $S = \{X_1, X_2, \dots, X_n\}$. Berapa banyak cara untuk memilih r objek dari S ?

Terdapat 2 kasus:

- Kasus 1: X_n dipilih.
Artinya, $r-1$ objek harus dipilih dari himpunan $\{X_1, X_2, X_3, \dots, X_{n-1}\}$. Banyaknya cara berbeda dari kasus ini adalah C_{r-1}^{n-1} .
- Kasus 2: X_n tidak dipilih.
Artinya, r objek harus dipilih dari himpunan $\{X_1, X_2, X_3, \dots, X_n\}$. Banyaknya cara berbeda dari kasus ini adalah C_r^{n-1} .

Dengan aturan penjumlahan dari kasus 1 dan kasus 2, kita dapatkan $C_r^n = C_{r-1}^{n-1} + C_r^{n-1}$. Persamaan itulah yang sering kita gunakan dalam membuat Segitiga Pascal, karena C_r^n juga menyatakan baris ke- n dan kolom ke- r pada Segitiga Pascal.

Dalam dunia pemrograman, kadang kala dibutuhkan perhitungan seluruh nilai C_r^n yang memenuhi $n \leq N$, untuk suatu N tertentu. Pencarian nilai dari C_r^n dengan menghitung faktorial memiliki kompleksitas $O(n)$. Apabila seluruh nilai kombinasi dicari dengan cara tersebut, kompleksitas akhirnya adalah $O(N^3)$. Namun, dengan menggunakan persamaan $C_r^n = C_{r-1}^{n-1} + C_r^{n-1}$, maka secara keseluruhan kompleksitasnya dapat menjadi $O(N^2)$.

Algoritma 9 merupakan implementasi untuk pencarian seluruh nilai kombinasi C_r^n yang memenuhi $n \leq N$, untuk suatu N .

Algoritma 9 Penggunaan Segitiga Pascal untuk mencari kombinasi.

```
1: procedure PRECOMPUTECOMBINATION( $N$ )
2:    $C \leftarrow$  new integer $[N + 1][N + 1]$       ▷ Sediakan  $C$  berukuran  $(N + 1) \times (N + 1)$ 
3:   for  $i \leftarrow 0, N$  do
4:      $C[i][0] \leftarrow 1$ 
5:     for  $j \leftarrow 1, i - 1$  do
6:        $C[i][j] = C[i - 1][j - 1] + C[i - 1][j]$ 
7:     end for
8:      $C[i][i] \leftarrow 1$ 
9:   end for
10: end procedure
```

3 Pencarian dan Pengurutan

Pencarian dan pengurutan merupakan hal sederhana dan sering digunakan dalam pemrograman. Terdapat berbagai macam cara untuk melakukan pencarian maupun pengurutan. Membahas beberapa cara tersebut dapat memberikan gambaran bagaimana sebuah persoalan diselesaikan dengan berbagai algoritma.

Pencarian

Contoh Soal 3.1: Sepatu untuk Bebek

Kwek, salah satu bebek Pak Dengklek akan segera merayakan ulang tahunnya. Pak Dengklek akan memberikan Kwek hadiah ulang tahun berupa sepatu. Terdapat N sepatu di toko. Sepatu ke- i memiliki ukuran sebesar h_i . Pak Dengklek tahu bahwa ukuran kaki Kwek adalah sebuah bilangan bulat X . Karena N bisa jadi sangat besar, Pak Dengklek meminta bantuan kalian untuk mencari sepatu keberapa yang cocok dengan ukuran kaki Kwek. Bantulah dia!

Batasan

- $1 \leq N \leq 100.000$
- $1 \leq X \leq 100.000$
- $1 \leq h_i \leq 100.000$, untuk $1 \leq i \leq N$

Kita dihadapkan pada persoalan pencarian: diberikan N angka. Cari apakah X ada di antara angka-angka tersebut. Jika ditemukan, cetak di urutan keberapa angka tersebut berada.

Sequential Search

Salah satu ide yang muncul adalah dengan melakukan pengecekan kepada seluruh elemen yang ada. Dengan kata lain, kita akan melakukan:

- Periksa satu per satu dari sepatu pertama, kedua, ketiga, dan seterusnya.
- Jika ditemukan, langsung laporkan.
- Jika sampai akhir belum juga ditemukan, artinya angka yang dicari tidak ada pada daftar.

Implementasi ide tersebut dapat dilihat di Algoritma 10.

Algoritma pencarian dengan membandingkan elemen satu per satu semacam ini disebut dengan *sequential search* atau *linear search*. Jika ada N elemen pada daftar yang perlu dicari, maka paling banyak diperlukan N operasi perbandingan. Dalam kompleksitas waktu, performa algoritma *sequential search* bisa dinyatakan dalam $O(N)$.

Binary Search

Apakah ada algoritma pencarian yang lebih baik? Misalkan Anda diberikan Kamus Besar Bahasa Indonesia (KBBI) yang mengandung 90.000 kata. Anda kemudian ingin

Algoritma 10 Contoh Kode Pencarian Sederhana.

```

1: procedure SEARCH( $h, X$ )
2:   hasil  $\leftarrow 0$  ▷ Artinya belum ditemukan
3:   for  $i \leftarrow 1, N$  do
4:     if ( $h[i] = X$ ) then
5:       hasil  $\leftarrow i$ 
6:       break
7:     end if
8:   end for
9:   if ( $hasil = 0$ ) then
10:    print "beri hadiah lain"
11:  else
12:    print hasil
13:  end if
14: end procedure

```

mencari definisi suatu kata pada KBBI. Dengan *sequential search* perlu dilakukan paling banyak 90.000 operasi. Apakah kita sebagai manusia melakukan perbandingan sampai 90.000 kata?

Ternyata, kita tidak perlu mencari kata satu per satu, halaman demi halaman, sampai 90.000 kata. Bahkan, kita dapat mencari arti suatu kata cukup dalam beberapa perbandingan, yang jauh lebih sedikit dari 90.000. Bagaimana hal ini bisa terjadi?

Sifat Khusus: Terurut

KBBI memiliki sebuah sifat khusus, yaitu **terurut berdasarkan abjad**. Dengan sifat ini, kita bisa membuka halaman tengah dari KBBI, lalu periksa apakah kata yang kita cari ada pada halaman tersebut.

Misalkan kata yang kita cari berawalan 'W'. Jika halaman tengah hanya menunjukkan daftar kata dengan awalan 'G'. Jelas bahwa kata yang kita cari tidak mungkin berada di **separuh pertama** kamus. Ulangi hal serupa dengan separuh belakang kamus, buka bagian tengahnya dan bandingkan. Dengan cara ini, setiap perbandingan akan mengeliminasi separuh rentang pencarian. Pencarian seperti ini disebut dengan *binary search*.

Analisis Binary Search

Misalkan terdapat sebuah daftar berisi N elemen, dan kita perlu mencari salah satu elemen. Banyaknya operasi maksimal yang dibutuhkan sampai suatu elemen bisa dipastikan keberadaannya sama dengan panjang dari barisan $[N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \dots, 2, 1]$ yakni sebesar $\lceil \log_2 N \rceil$, sehingga kompleksitasnya $O(\log N)$.

Perhatikan Tabel 3.1 untuk membandingkan banyaknya operasi yang dibutuhkan untuk nilai N tertentu.

Seperti yang terlihat, *binary search* jauh lebih cepat dari *linear search*. Bahkan untuk $N = 10^6$, *binary search* hanya membutuhkan maksimal 20 operasi.

Tabel 3.1: Perbandingan *sequential search* dan *binary search*.

N	<i>Sequential</i>	<i>Binary</i>
50	50	6
100	100	7
150	150	8
200	200	8
250	250	8
300	300	9

Implementasi *Binary Search*

Terdapat bermacam cara implementasi *binary search*. Salah satunya seperti yang dicontohkan pada Algoritma 11. Pada algoritma ini, kita akan mencari X pada *array* h . Tentu kita asumsikan bahwa h sudah terurut.

Algoritma 11 Algoritma *binary search*.

```

1: procedure BINARYSEARCH( $h, X$ )
2:    $hasil \leftarrow 0$ 
3:    $kiri \leftarrow 1$ 
4:    $kanan \leftarrow N$ 
5:   while  $((kiri \leq kanan) \wedge (hasil = 0))$  do
6:      $tengah \leftarrow (kiri + kanan) \text{ div } 2$ 
7:     if  $(X < h[tengah])$  then
8:        $kanan \leftarrow tengah - 1$ 
9:     else if  $(X > h[tengah])$  then
10:       $kiri \leftarrow tengah + 1$ 
11:    else
12:       $hasil \leftarrow tengah$ 
13:    end if
14:  end while
15:  if  $(hasil = 0)$  then
16:    print "beri hadiah lain"
17:  else
18:    print  $hasil$ 
19:  end if
20: end procedure

```

▷ Artinya belum ditemukan

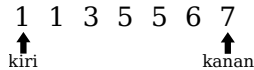
Berikut adalah cara kerja dari Algoritma 11:

- Variabel **kiri** dan **kanan** menyatakan bahwa rentang pencarian kita ada di antara [**kiri**, **kanan**]. Nilai X *mungkin* ada di dalam sana.
- Kita mengambil nilai tengah dari kiri dan kanan, lalu periksa apakah X sama dengan $h[tengah]$.
- Jika ternyata X kurang dari $h[tengah]$, artinya X tidak mungkin berada pada rentang [**tengah**, **kanan**].
- Dengan demikian, rentang pencarian kita yang baru adalah [**kiri**, **tengah-1**].

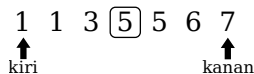
- Hal yang serupa juga terjadi ketika X lebih dari $h[\text{tengah}]$, rentang pencarian kita menjadi $[\text{tengah}+1, \text{kanan}]$.
- Tentu saja jika X sama dengan $h[\text{tengah}]$, catat posisinya dan pencarian berakhir.
- Untuk kasus X tidak ditemukan, pencarian akan terus dilakukan sampai $\text{kiri} > \text{kanan}$, yang artinya rentang pencarian sudah habis.

Ilustrasi Eksekusi *Binary Search*

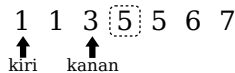
Sebagai contoh, misalkan Anda ingin mencari angka 3 dari suatu *array* $[1, 1, 3, 5, 5, 6, 7]$. Gambar 3.1 sampai Gambar 3.6 mengilustrasikan proses *binary search* dari awal sampai akhir.



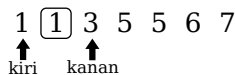
Gambar 3.1: Tahap 1: Mulai dengan menginisiasi variabel *kiri* dan *kanan* pada kedua ujung *array*.



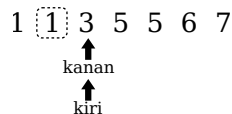
Gambar 3.2: Tahap 2: Temukan indeks *tengah* yaitu nilai tengah dari *kiri* dan *kanan*. Kemudian lakukan perbandingan nilai *tengah* dengan nilai yang ingin dicari.



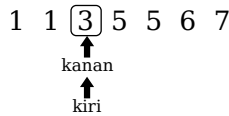
Gambar 3.3: Tahap 3: Karena $5 > 3$, maka geser *kanan* menjadi *tengah* - 1.



Gambar 3.4: Tahap 4: Temukan lagi indeks *tengah* yaitu nilai tengah dari *kiri* dan *kanan* dan bandingkan nilainya dengan nilai yang dicari.



Gambar 3.5: Tahap 5: Karena $1 < 3$, maka geser *kiri* menjadi *tengah* + 1.



Gambar 3.6: Tahap 6: Temukan lagi indeks *tengah* yaitu nilai tengah dari *kiri* dan *kanan* dan bandingkan nilainya dengan nilai yang dicari. Ternyata nilai pada indeks *tengah* adalah 3. Dengan demikian pencarian berhasil.

Rangkuman

Secara umum, kedua algoritma pencarian dapat dirangkumkan sebagai berikut.

Sequential search

- Data untuk pencarian tidak perlu terurut.
- Kompleksitasnya $O(N)$, dengan N adalah ukuran data.
- Baik diimplementasikan jika pencarian hanya dilakukan sesekali.

Binary search

- Data untuk pencarian harus terurut.
- Kompleksitasnya $O(\log N)$, dengan N adalah ukuran data.
- Baik diimplementasikan jika pencarian perlu dilakukan berkali-kali.

Bagaimana jika kita memiliki data yang sangat banyak, tidak terurut, dan butuh pencarian yang efisien? Salah satu solusinya adalah dengan **mengurutkan** data tersebut, lalu mengaplikasikan *binary search* untuk setiap pencarian.

Pengurutan

Pengurutan sering digunakan dalam pemrograman untuk membantu membuat data lebih mudah diolah. Terdapat berbagai macam cara untuk melakukan pengurutan, masing-masing dengan keuntungan dan kekurangannya.

Contoh Soal 3.2: Bebek Berbaris

Sebelum masuk ke dalam kandang, para bebek akan berbaris terlebih dahulu. Seiring dengan berjalannya waktu, bebek-bebek tumbuh tinggi. Pertumbuhan ini berbeda-beda; ada bebek yang lebih tinggi dari bebek lainnya. Terdapat N ekor bebek, bebek ke- i memiliki tinggi sebesar h_i . Perbedaan tinggi ini menyebabkan barisan terlihat kurang rapi, sehingga Pak Dengklek ingin bebek-bebek berbaris dari yang paling pendek ke paling tinggi. Bantulah para bebek untuk mengurutkan barisan mereka!

Batasan

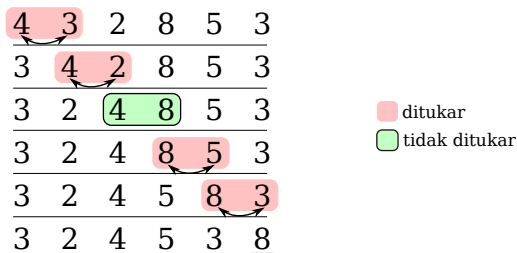
- $1 \leq N \leq 1.000$
- $1 \leq h_i \leq 100.000$, untuk $1 \leq i \leq N$

Persoalan ini meminta kita melakukan pengurutan N bilangan dengan rentang datanya

antara 1 sampai 100.000. Terdapat sejumlah algoritma pengurutan, yang akan dibahas pada bagian berikutnya.

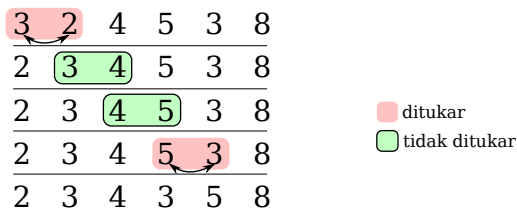
Bubble Sort

Kita dapat melakukan *bubble sort* dengan memproses setiap pasang elemen yang bersebelahan satu per satu. Mulai dari elemen pertama, cek apakah elemen sesudahnya (yaitu elemen kedua) lebih kecil. Bila ya, artinya elemen pertama ini harus terletak sesudah elemen kedua. Untuk itu, lakukan penukaran. Bila tidak, tidak perlu lakukan penukaran. Lanjut periksa elemen kedua, ketiga, dan seterusnya. Proses ini mengakibatkan elemen dengan nilai terbesar pasti digiring ke posisi terakhir seperti pada Gambar 3.7.



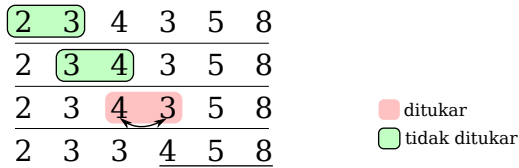
Gambar 3.7: Ilustrasi penggiringan elemen terbesar ke posisi terakhir.

Bila proses ini dilakukan lagi, maka elemen kedua terbesar akan terletak di posisi kedua dari terakhir. Kali ini pemeriksaan cukup dilakukan sampai 1 elemen sebelum posisi terakhir, sebab elemen terakhir sudah pasti tidak akan berubah posisi.



Gambar 3.8: Ilustrasi penggiringan elemen kedua terbesar ke posisi kedua dari akhir.

Demikian pula untuk eksekusi yang ketiga kalinya, yang kebetulan data sudah menjadi terurut seperti pada Gambar 3.9.



Gambar 3.9: Ilustrasi penggiringan elemen ketiga terbesar ke posisi ketiga dari akhir.

Implementasi *Bubble Sort*

Implementasi *Bubble Sort* dapat ditemukan pada Algoritma 12.

Algoritma 12 Algoritma *Bubble Sort*.

```

1: procedure BUBBLESORT( $h$ )
2:   for  $i \leftarrow 1, N - 1$  do
3:     for  $j \leftarrow 1, N - i$  do
4:       if ( $h[j] > h[j + 1]$ ) then
5:         SWAP( $h[j], h[j + 1]$ )           ▷ Tukar  $h[j]$  dengan  $h[j + 1]$ 
6:       end if
7:     end for
8:   end for
9: end procedure

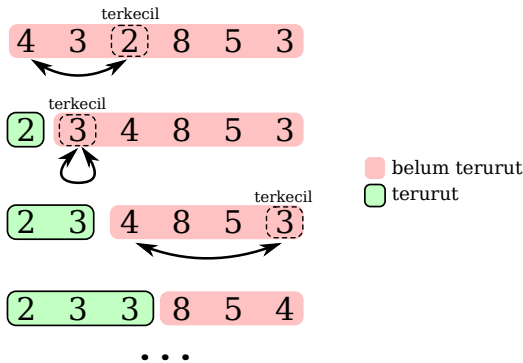
```

Jika eksekusi ke- i mengakibatkan i elemen terbesar terletak di i posisi terakhir, maka dibutuhkan N kali eksekusi hingga seluruh data terurut. Dalam sekali eksekusi, dilakukan iterasi dari elemen pertama sampai elemen terakhir, yang kompleksitasnya berkisar antara $O(1)$ sampai $O(N)$, tergantung eksekusi ke berapa. Secara rata-rata, kompleksitasnya setiap eksekusi adalah $O(N/2)$, yang bisa ditulis $O(N)$. Total kompleksitas *bubble sort* adalah $O(N^2)$.

Selection Sort

Berikut adalah langkah-langkah untuk melakukan *selection sort*:

1. Pilih elemen terkecil dari data, lalu pindahkan ke elemen pertama.
2. Pilih elemen terkecil dari data yang tersisa, lalu pindahkan ke elemen kedua.
3. Pilih elemen terkecil dari data yang tersisa, lalu pindahkan ke elemen ketiga.
4. ... dan seterusnya sampai seluruh elemen terurut.

Gambar 3.10: Ilustrasi jalannya *selection sort*.

Implementasi *Selection Sort*

Implementasi *selection sort* dapat ditemukan pada Algoritma 13.

Algoritma 13 Algoritma *selection sort*.

```

1: procedure SELECTIONSORT( $h$ )
2:   for  $i \leftarrow 1, N$  do
3:     ▷ Pencarian indeks terkecil
4:      $minIndex \leftarrow i$ 
5:     for  $j \leftarrow i + 1, N$  do
6:       if ( $h[j] < h[minIndex]$ ) then
7:          $minIndex \leftarrow j$ 
8:       end if
9:     end for
10:    SWAP( $h[i], h[minIndex]$ )
11:  end for
12: end procedure

```

Pencarian elemen terkecil dapat dilakukan dengan *linear search*. Berhubung perlu dilakukan N kali *linear search*, maka kompleksitas *selection sort* adalah $O(N^2)$.

Pengurutan Parsial

Cara kerja *selection sort* memungkinkan kita untuk melakukan *partial sort*. Jika kita hanya tertarik dengan K elemen terkecil, kita bisa melakukan proses seleksi dan menukar pada *selection sort* K kali. Dengan demikian, pencarian K elemen terkecil dapat dilakukan dalam $O(KN)$, cukup baik apabila K jauh lebih kecil dari N .

Insertion Sort

Berikut adalah langkah-langkah untuk melakukan *insertion sort*:

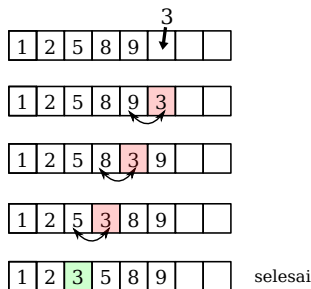
1. Siapkan *array* kosong. Secara definisi, *array* yang kosong merupakan *array* terurut.
2. Secara bertahap, sisipkan elemen baru ke dalam *array* yang sudah terurut tersebut.
3. Penyisipan ini harus dilakukan sedemikian sehingga *array* tetap terurut.
4. Misalnya saat ini data yang sudah terurut adalah [1, 2, 3, 8], lalu elemen yang akan disisipkan adalah 5, maka dihasilkan [1, 2, 3, 5, 8].

Prosesnya dapat digambarkan seperti pada Tabel 3.2

Tabel 3.2: Proses Pemindahan Data menjadi Terurut.

Data Asal	Data Terurut
[4, 3, 2, 8, 5, 3]	[]
[3, 2, 8, 5, 3]	[4]
[2, 8, 5, 3]	[3, 4]
[8, 5, 3]	[2, 3, 4]
[5, 3]	[2, 3, 4, 8]
[3]	[2, 3, 4, 5, 8]
[]	[2, 3, 3, 4, 5, 8]

Strategi yang dapat digunakan untuk menyisipkan angka adalah meletakkan angka yang hendak disisipkan pada bagian paling belakang, lalu digiring mundur sampai posisinya tepat. Misalnya pada kasus menyisipkan angka 3 pada data [1, 2, 5, 8, 9] seperti pada Gambar 3.11



Gambar 3.11: Ilustrasi penyisipan angka 3 pada data [1, 2, 5, 8, 9].

Implementasi *Insertion Sort*

Implementasi *insertion sort* dapat ditemukan pada Algoritma 14.

Untuk mengurutkan data, diperlukan N kali penyisipan. Setiap menyisipkan, dilakukan pengiringan yang kompleksitasnya:

- Pada kasus terbaik $O(1)$, ketika angka yang dimasukkan merupakan angka terbesar pada data saat ini.

Algoritma 14 Algoritma *insertion sort*.

```

1: procedure INSERTIONSORT( $h$ )
2:   for  $i \leftarrow 1, N$  do
3:      $pos \leftarrow i$ 
4:     while  $((pos > 1) \wedge (h[pos] < h[pos - 1]))$  do    ▷ Selama belum tepat, giring ke
depan
5:       SWAP( $h[pos]$ ,  $h[pos - 1]$ )
6:        $pos \leftarrow pos - 1$ 
7:     end while
8:   end for
9: end procedure

```

- Pada kasus terburuk $O(N)$, yaitu ketika angka yang dimasukkan merupakan angka terkecil pada data saat ini.
- Pada kasus rata-rata, kompleksitasnya $O(N/2)$, atau bisa ditulis $O(N)$.

Berdasarkan observasi tersebut, *insertion sort* dapat bekerja sangat cepat ketika datanya sudah hampir terurut. Pada kasus terbaik, *insertion sort* bekerja dalam $O(N)$, yaitu ketika data sudah terurut. Pada kasus terburuk, kompleksitasnya $O(N^2)$. Secara rata-rata, kompleksitasnya adalah $O(N^2)$.

Strategi *insertion* pada algoritma ini dapat digunakan untuk menambahkan sebuah elemen pada data yang sudah terurut. Ketimbang mengurutkan kembali seluruh elemen, cukup lakukan strategi *insertion* yang secara rata-rata bekerja dalam $O(N)$.

Counting Sort

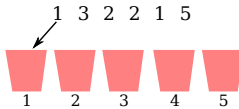
Berikut adalah langkah-langkah untuk melakukan *counting sort*:

1. Siapkan M ember yang mana M merupakan banyaknya kemungkinan nilai elemen yang muncul pada *array*.
2. Setiap ember dinomori dengan sebuah angka yang merupakan seluruh nilai elemen yang mungkin ada.
3. Untuk setiap elemen yang mau diurutkan, masukkan ke ember yang sesuai dengan nilai elemen tersebut.
4. Setelah seluruh elemen dimasukkan ke ember yang bersesuaian, keluarkan isi ember-ember mulai dari ember 1 sampai M secara berurutan.

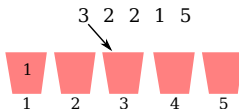
Sebagai ilustrasi, kita ingin mengurutkan *array* $[1, 3, 2, 2, 1, 5]$ secara menaik. Jika diketahui nilai elemen yang muncul hanyalah dari 1 sampai 5, maka kita cukup menyiapkan 5 ember. Ember-ember tersebut dinomori 1 sampai dengan 5. Maka, tahapan-tahapan *counting sort* dari awal hingga selesai dalam mengurutkan *array* tersebut dapat dilihat pada Gambar 3.12 sampai Gambar 3.21.



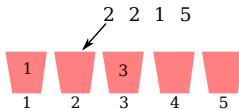
Gambar 3.12: Tahap 1: Siapkan 5 ember kosong.



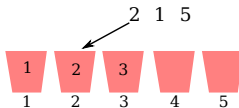
Gambar 3.13: Tahap 2: Masukkan elemen pertama ke dalam ember.



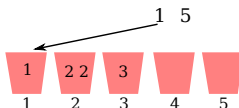
Gambar 3.14: Tahap 3: Masukkan elemen kedua ke dalam ember.



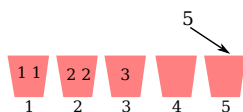
Gambar 3.15: Tahap 4: Masukkan elemen ketiga ke dalam ember.



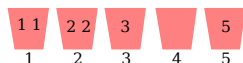
Gambar 3.16: Tahap 5: Masukkan elemen keempat ke dalam ember.



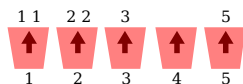
Gambar 3.17: Tahap 6: Masukkan elemen kelima ke dalam ember.



Gambar 3.18: Tahap 7: Masukkan elemen keenam ke dalam ember.



Gambar 3.19: Tahap 8: Kondisi ember setelah semua elemen dimasukkan.



Gambar 3.20: Tahap 9: Keluarkan isi dari setiap ember, mulai dari ember paling kiri.

1 1 2 2 3 5

Gambar 3.21: Tahap 10: Pengurutan selesai.

Implementasi *Counting Sort*

Ide ini dapat diwujudkan dengan menyiapkan tabel frekuensi yang berperan sebagai “ember”. Untuk setiap nilai elemen yang mungkin, catat frekuensi kemunculannya. Terakhir, iterasi tabel frekuensi dari elemen terkecil sampai elemen terbesar. Tabel frekuensi dapat diimplementasikan dengan *array* sederhana. Contoh implementasi Counting Sort dapat dilihat pada Algoritma 15

Contoh Kode

Dapat diperhatikan bahwa kompleksitas *counting sort* adalah $O(N + M)$, dengan M adalah rentang nilai data. Jika M tidak terlalu besar, maka *counting sort* dapat bekerja dengan sangat cepat. Lebih tepatnya, *counting sort* merupakan opsi yang sangat baik jika datanya memiliki rentang yang kecil, misalnya data tentang usia penduduk yang rentangnya hanya $[0, 125]$. Bandingkan dengan algoritma pengurutan lain yang kompleksitasnya $O(N^2)$!

Algoritma 15 Algoritma *counting sort*.

```

1: procedure COUNTINGSORT( $h$ )
2:   ▷ Catat frekuensinya
3:   for  $i \leftarrow 1, N$  do
4:      $x \leftarrow h[i]$ 
5:      $f_{table}[x] \leftarrow f_{table}[x] + 1$ 
6:   end for

7:   ▷ Tuang kembali ke  $h[]$ 
8:    $index \leftarrow 1$ 
9:   for  $i \leftarrow 1, 100000$  do
10:    for  $j \leftarrow 1, f_{table}[i]$  do
11:       $h[index] \leftarrow i$ 
12:       $index \leftarrow index + 1$ 
13:    end for
14:  end for
15: end procedure

```

Karena perlu membuat tabel frekuensi, maka *counting sort* hanya dapat digunakan ketika rentang nilai datanya kecil, misalnya $\leq 10^7$. Selain itu, algoritma ini hanya dapat mengurutkan data diskret. Data seperti bilangan pecahan tidak dapat diurutkan secara tepat. Alternatifnya, untuk menangani data dengan rentang yang besar atau data yang tidak diskret, kita bisa menggunakan *radix sort* yang memiliki konsep yang mirip dengan *counting sort*.

Rangkuman

Tabel 3.3: Perbandingan beberapa Algoritma Pengurutan.

Algoritma	Kompleksitas	Keterangan
<i>Bubble sort</i>	$O(N^2)$	-
<i>Selection sort</i>	$O(N^2)$	Dapat digunakan untuk <i>partial sort</i> dalam $O(KN)$
<i>Insertion sort</i>	$O(N^2)$	Sangat cepat jika data hampir terurut, kasus terbaiknya $O(N)$
<i>Counting sort</i>	$O(N + M)$	Cepat hanya untuk data dengan rentang yang kecil

Terdapat algoritma pengurutan yang lebih efisien, misalnya *quicksort* dan *merge sort*. Algoritma pengurutan tersebut menggunakan konsep *divide and conquer* yang akan dipelajari pada Bab 5.

▶▶ Sekilas Info ◀◀

Bahasa pemrograman tertentu seperti Java atau C++ memiliki fungsi pengurutan bawaan, sehingga pada umumnya Anda tidak perlu mengimplementasikan fungsi pengurutan sendiri. Namun, Anda harus paham konsep pengurutan jika sewaktu-waktu Anda harus memodifikasi algoritma pengurutan untuk menyelesaikan soal yang diberikan.

4 Brute Force

Brute force adalah salah satu strategi penyelesaian masalah. *Brute force* merupakan strategi yang sederhana, serta pasti menemukan solusi yang kita harapkan, walaupun pada umumnya memerlukan waktu yang cukup lama.

►► Sekilas Info ◀◀

Pada kompetisi dengan sistem penilaian parsial seperti OSN atau IOI, umumnya subsoal-subsoal awal dapat diselesaikan dengan *brute force*.

Konsep *Brute Force*

Brute force bukan merupakan suatu algoritma khusus, melainkan suatu strategi penyelesaian masalah. Sebutan lain dari *brute force* adalah *complete search* dan *exhaustive search*. Prinsip dari strategi ini hanya satu, yaitu **mencoba semua kemungkinan**.

Brute force **menjamin** solusi pasti benar karena menelusuri seluruh kemungkinan. Akibatnya, secara umum *brute force* bekerja dengan lambat, terutama ketika terdapat banyak kemungkinan solusi yang perlu dicoba. Sebagai contoh, salah satu penggunaan strategi *brute force* adalah dalam mengerjakan permasalahan *Subset Sum*.

Contoh Soal 4.1: *Subset Sum*

Diberikan N buah bilangan $\{a_1, a_2, \dots, a_N\}$ dan sebuah bilangan K . Apakah terdapat sebuah subhimpunan sedemikian sehingga jumlahan dari elemen-elemennya sama dengan K ? Bila ya, maka keluarkan "YA". Selain itu, keluarkan "TIDAK".

Batasan

- $1 \leq N \leq 15$
- $1 \leq K \leq 10^5$
- $1 \leq a_i \leq 10^5$

Penyelesaian dengan Teknik *Brute Force*

Ingat bahwa prinsip dari strategi *brute force* adalah menelusuri semua kemungkinan. Dengan kata lain, kita mencoba membangkitkan seluruh kemungkinan, lalu mencari jawaban dari permasalahan kita melalui seluruh kemungkinan tersebut. Dalam membangkitkan seluruh kemungkinan, kita dapat melakukannya secara iteratif maupun rekursif.

Sebagai contoh, kita kembali ke permasalahan *Subset Sum*. Untuk setiap elemen, terdapat dua kemungkinan, yaitu memilih bilangan tersebut untuk dimasukkan ke subhimpunan atau tidak. Selanjutnya, kita akan menelusuri semua kemungkinan pilihan yang ada. Jika terdapat pemilihan yang jumlahan elemen-elemennya sama dengan K , maka terdapat solusi. Hal ini dapat dengan mudah diimplementasikan secara rekursif.

Selanjutnya, mari kita analisis performa strategi *brute force* untuk permasalahan *Subset Sum*. Untuk setiap elemen, terdapat 2 pilihan, yaitu dipilih atau tidak. Karena terdapat N elemen, maka terdapat 2^N kemungkinan. Maka, kompleksitas solusi *brute force* adalah $O(2^N)$. Untuk nilai N terbesar, $2^N = 2^{15} = 32.768$, yang masih cukup dalam operasi komputer per detik pada umumnya (100 juta).

Algoritma untuk menyelesaikan permasalahan *Subset Sum* dapat ditemukan pada Algoritma 16 serta Algoritma 17.

Algoritma 16 Solusi permasalahan *Subset Sum*.

```

1: function SOLVE( $i, sum$ )
2:   if  $i > N$  then
3:     if  $sum = K$  then
4:       return true
5:     else
6:       return false
7:     end if
8:   end if
9:   option1  $\leftarrow$  SOLVE( $i + 1, sum + a_i$ )           ▷ Pilih elemen  $a_i$ 
10:  option2  $\leftarrow$  SOLVE( $i + 1, sum$ )                ▷ Tidak pilih elemen  $a_i$ 
11:  return option1  $\vee$  option2
12: end function

```

Algoritma 17 Fungsi pemanggil solusi.

```

1: function SOLVESUBSETSUM()
2:   return SOLVE(1,0)
3: end function

```

Optimisasi Teknik *Brute Force*

Bisakah solusi pada Algoritma 16 dibuat menjadi lebih cepat? Perhatikan kasus ketika sum telah melebihi K . Karena semua a_i bernilai positif, maka sum tidak akan mengecil. Karena itu, bila sum sudah melebihi K , dipastikan tidak akan tercapai sebuah solusi.

Dari sini, kita bisa mengoptimisasi fungsi *SOLVE* menjadi fungsi *OPTIMIZEDSOLVE* yang akan menghentikan suatu penelusuran kemungkinan apabila jumlahan elemen yang dipilih sudah melebihi K . Algoritma penyelesaian *Subset Sum* dengan optimasi tersebut dapat dilihat di Algoritma 18.

Optimisasi yang dilakukan oleh *OPTIMIZEDSOLVE* pada Algoritma 18 biasa disebut sebagai **pruning** (pemangkasan). *Pruning* merupakan optimisasi dengan mengurangi ruang pencarian dengan cara menghindari pencarian yang sudah pasti salah.

Meskipun mengurangi ruang pencarian, *pruning* umumnya tidak mengurangi kompleksitas solusi. Sebab, biasanya terdapat kasus yang mana *pruning* tidak mengurangi ruang pencarian secara signifikan. Pada kasus ini, solusi dapat dianggap tetap bekerja dalam $O(2^N)$.

Algoritma 18 Solusi permasalahan *Subset Sum* yang dioptimasi.

```

1: function OPTIMIZEDSOLVE( $i, sum$ )
2:   if  $i > N$  then
3:     if  $sum = K$  then
4:       return true
5:     else
6:       return false
7:     end if
8:   end if
9:   if  $sum > K$  then
10:    return false
11:  end if
12:   $option1 \leftarrow$  OPTIMIZEDSOLVE( $i + 1, sum + a_i$ )           ▷ Pilih elemen  $a_i$ 
13:   $option2 \leftarrow$  OPTIMIZEDSOLVE( $i + 1, sum$ )             ▷ Tidak pilih elemen  $a_i$ 
14:  return  $option1 \vee option2$ 
15: end function

```

Selanjutnya, mari kita lihat contoh yang mana optimisasi dapat mempercepat solusi secara signifikan.

Contoh Soal 4.2: Mengatur Persamaan

Diberikan sebuah persamaan: $p + q + r = 0$. Masing-masing dari p , q , dan r harus merupakan anggota dari $\{a_1, a_2, \dots, a_N\}$. Diketahui pula bahwa semua nilai $\{a_1, a_2, \dots, a_N\}$ unik. Berapa banyak triplet $\langle p, q, r \rangle$ berbeda yang memenuhi persamaan tersebut?

Batasan

- $1 \leq N \leq 2.000$
- $-10^5 \leq a_i \leq 10^5$

Salah satu solusi untuk menyelesaikan permasalahan tersebut adalah dengan melakukan *brute force* seluruh nilai p , q , dan r yang mungkin, seperti yang tertera pada Algoritma 19. Karena setiap variabel memiliki N kemungkinan nilai, maka kompleksitas waktu solusi tersebut adalah $O(N^3)$. Tentunya terlalu lambat untuk nilai N yang mencapai 2.000.

Apakah ada solusi yang lebih baik? Sebenarnya, kita bisa mengambil suatu observasi sederhana yang cukup berguna untuk mempercepat solusi kita.

Observasi: Jika kita sudah menentukan nilai p dan q , maka nilai r haruslah $-(p + q)$.

Berdasarkan observasi tersebut, kita dapat mencoba semua kemungkinan p dan q , lalu memeriksa apakah $-(p + q)$ terdapat di dalam array. Untuk mempercepat, pencarian $-(p + q)$ bisa menggunakan *binary search*. Implementasi solusi tersebut terdapat pada Algoritma 20.

Pada Algoritma 20, fungsi $EXIST(r)$ adalah fungsi untuk memeriksa keberadaan r di himpunan $\{a_1, a_2, \dots, a_N\}$ dengan menggunakan *binary search* (tentunya setelah diurutkan). Kompleksitas dari COUNTTRIPLETSFAST sendiri adalah $O(N^2 \log N)$.

Algoritma 19 Solusi permasalahan Mengatur Persamaan.

```

1: function COUNTTRIPLETS()
2:   count  $\leftarrow$  0
3:   for  $i \leftarrow 1, N$  do
4:     for  $j \leftarrow 1, N$  do
5:       for  $k \leftarrow 1, N$  do
6:          $p \leftarrow a[i]$ 
7:          $q \leftarrow a[j]$ 
8:          $r \leftarrow a[k]$ 
9:         if  $(p + q + r) = 0$  then
10:          count  $\leftarrow$  count + 1
11:        end if
12:      end for
13:    end for
14:  end for
15:  return count
16: end function

```

Algoritma 20 Solusi permasalahan Mengatur Persamaan yang dioptimasi.

```

1: function COUNTTRIPLETSFAST()
2:   count  $\leftarrow$  0
3:   for  $i \leftarrow 1, N$  do
4:     for  $j \leftarrow 1, N$  do
5:        $p \leftarrow a[i]$ 
6:        $q \leftarrow a[j]$ 
7:        $r \leftarrow -(p + q)$ 
8:       if EXIST( $r$ ) then
9:         count  $\leftarrow$  count + 1
10:      end if
11:    end for
12:  end for
13:  return count
14: end function

```

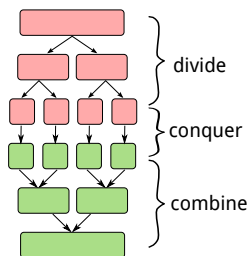
Kompleksitas $O(N^2 \log N)$ sudah cukup untuk N yang mencapai 2.000. Dari sini kita belajar bahwa optimisasi pada pencarian kadang diperlukan, meskipun ide dasarnya adalah *brute force*.

5 Divide and Conquer

Divide and conquer (*D&C*) adalah metode penyelesaian masalah dengan cara membagi masalah menjadi beberapa submasalah yang lebih kecil dan sederhana, kemudian menyelesaikan masing-masing submasalah tersebut, lalu menggabungkannya kembali menjadi sebuah solusi yang utuh. Prinsip *divide and conquer* banyak digunakan pada struktur data lanjutan dan geometri komputasional.

Konsep *Divide and Conquer*

Jika suatu masalah dapat dibagi menjadi beberapa submasalah serupa yang lebih kecil dan independen, kemudian solusi dari masing-masing submasalah dapat digabungkan, maka *divide and conquer* dapat digunakan.



Gambar 5.1: Ilustrasi proses *divide and conquer*.

Gambar 5.1 mengilustrasikan proses *divide and conquer*. Proses tersebut terdiri dari tiga tahap yaitu:

1. *Divide*: membagi masalah menjadi masalah-masalah yang lebih kecil. Biasanya menjadi setengahnya atau mendekati setengahnya.
2. *Conquer*: ketika sebuah masalah sudah cukup kecil untuk diselesaikan, langsung selesaikan masalah tersebut.
3. *Combine*: menggabungkan solusi dari masalah-masalah yang lebih kecil menjadi solusi untuk masalah yang lebih besar.

Studi Kasus 1: *Merge Sort*

Merge sort adalah algoritma pengurutan yang memiliki kompleksitas $O(N \log N)$. Algoritma *merge sort* terus membagi *array* yang ingin diurutkan menjadi dua sampai tersisa satu elemen, kemudian menggabungkannya kembali sambil mengurutkannya. Inti pengurutan *merge sort* ada pada tahap *combine*.

Cara kerja algoritma ini adalah:

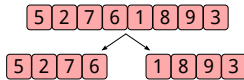
1. *Divide*: jika *array* yang akan diurutkan lebih dari 1 elemen, bagi *array* tersebut menjadi dua *array* sama besar (atau mendekati sama besar jika panjang *array* tersebut ganjil). Kemudian lakukan *merge sort* pada masing-masing *subarray* tersebut.
2. *Conquer*: ketika *array* berisi hanya 1 elemen, tidak perlu lakukan apapun. *Array* yang berisi 1 elemen sudah pasti terurut.
3. *Combine*: saat kita memiliki dua *array* yang telah terurut, kita bisa menggabungkan (*merge*) keduanya menjadi sebuah *array* yang terurut.

Mengapa *array* yang berukuran 1 bisa kita katakan sudah terurut? Jika ada sebuah *array* berukuran 1 (contohnya [42]), maka elemen tersebut tidak memiliki elemen lain yang bisa dibandingkan dengan elemen tersebut. Jadi, *array* tersebut sudah terurut.

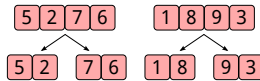
Contoh Eksekusi Merge Sort

Diberikan sebuah *array* [5,2,7,6,1,8,9,3]. Kita ingin mengurutkan *array* tersebut menggunakan *merge sort*. Gambar 5.2 hingga Gambar 5.8 mengilustrasikan proses yang terjadi dari awal hingga *array* tersebut terurut.

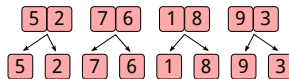
Kita memulai proses pengurutan dari tahap *divide*. Kita akan membagi *array* menjadi beberapa *subarray* hingga masing-masing memiliki 1 elemen.



Gambar 5.2: Langkah 1: Karena ukuran *array* lebih dari 1 elemen, kita bagi *array* yang kita miliki menjadi dua bagian.



Gambar 5.3: Langkah 2: *Array* yang kita miliki masih lebih dari 1 elemen, bagi lagi masing-masing menjadi dua.



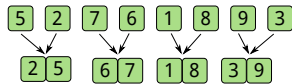
Gambar 5.4: Langkah 3: *Array* yang kita miliki masih lebih dari 1 elemen, bagi lagi masing-masing menjadi dua.

Kini kita memiliki kumpulan *array* yang dengan panjang satu. Secara definisi, masing-masing *array* tersebut telah terurut. Selanjutnya, kita masuk ke tahap *combine* untuk

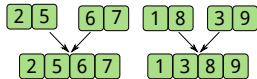
menyatukan kembali masing-masing *array* menjadi satu *array* utuh yang terurut. Hal ini dilakukan dengan menggabungkan 2 *array* yang bersebelahan dan memastikan *array* gabungannya tetap terurut.



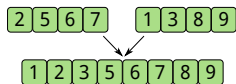
Gambar 5.5: Langkah 4: Kumpulan *array* dengan panjang satu yang masing-masing telah terurut.



Gambar 5.6: Langkah 5: Gabungkan 2 *array* yang bersebelahan. Pastikan *array* gabungan tetap terurut.



Gambar 5.7: Langkah 6: Gabungkan lagi 2 *array* yang bersebelahan. Pastikan *array* gabungan tetap terurut.

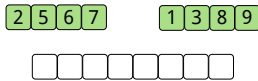


Gambar 5.8: Langkah 7: Gabungkan lagi dan akhirnya didapatkan *array* awal yang telah terurut.

Menggabungkan Dua *Array* yang Terurut

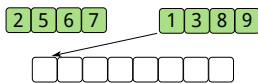
Bagaimana cara menggabungkan dua *array* yang telah terurut menjadi sebuah *array* yang terurut?

Observasi: elemen terkecil (terkiri) dari *array* gabungan pasti merupakan salah satu dari elemen terkecil (terkiri) di antara kedua *array*. Dengan kata lain, kita selalu mengambil nilai terkecil dari kedua *array* yang belum dimasukkan ke *array* gabungan sebagai elemen selanjutnya di *array* gabungan.



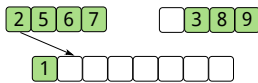
Gambar 5.9: Diberikan 2 buah *array* yang sudah terurut, kita ingin menggabungkannya menjadi satu *array* yang terurut.

Sebagai contoh, diberikan 2 buah *array* terurut [2, 5, 6, 7] dan [1, 3, 8, 9]. Untuk menggabungkan keduanya, kita akan membandingkan elemen terkecil yang belum dipilih dari masing-masing *array* dan memilih yang terkecil. Gambar 5.10 hingga Gambar 5.17 mengilustrasikan proses penggabungan dari awal sampai akhir.

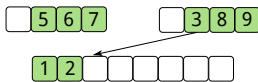


Gambar 5.10: Langkah 1: Bandingkan 1 dan 2. Karena $1 \leq 2$, kita ambil 1 sebagai elemen pertama di *array* gabungan.

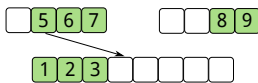
Ulangi hal serupa sampai salah satu atau kedua *array* habis.



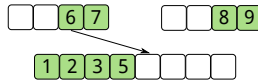
Gambar 5.11: Langkah 2: Bandingkan 2 dengan 3. Ambil 2 sebagai elemen selanjutnya karena $2 \leq 3$



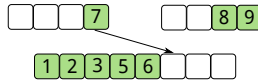
Gambar 5.12: Langkah 3: Bandingkan 5 dengan 3. Ambil 3 sebagai elemen selanjutnya karena $3 \leq 5$



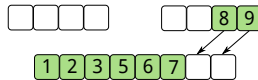
Gambar 5.13: Langkah 4: Bandingkan 5 dengan 8. Ambil 5 sebagai elemen selanjutnya karena $5 \leq 8$



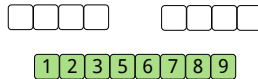
Gambar 5.14: Langkah 5: Bandingkan 6 dengan 8. Ambil 6 sebagai elemen selanjutnya karena $6 \leq 8$



Gambar 5.15: Langkah 6: Bandingkan 7 dengan 8. Ambil 7 sebagai elemen selanjutnya karena $7 \leq 8$



Gambar 5.16: Langkah 7: Ketika salah satu *array* telah habis, *array* yang masih bersisa tinggal ditempelkan di akhir *array* gabungan.



Gambar 5.17: Proses penggabungan telah selesai. *Array* telah terurut.

Berapa kompleksitas proses ini? Misalkan kedua *array* terurut yang akan digabung adalah A dan B . Pada setiap langkah, salah satu elemen dari A atau B dipindahkan ke dalam *array* gabungan. Total terdapat $|A| + |B|$ elemen yang dipindahkan, sehingga kompleksitasnya $O(|A| + |B|)$.

Implementasi Merge Sort

Algoritma 21 adalah implementasi dari prosedur utama *merge sort*. Prosedur ini akan mengurutkan *array* dari indeks *left* hingga *right*. Prosedur ini secara rekursif akan melakukan proses *divide*. Jika kita memiliki *array* dengan panjang N , maka awalnya kita akan memanggil `MERGESORT(arr, 1, N)`.

Perhatikan bahwa prosedur `MERGESORT` pada Algoritma 21 memanggil prosedur `MERGE`. Prosedur ini akan menggabungkan dua *array* yang terurut. Namun secara implementasinya, kita hanya menggunakan satu buah *array* saja. kedua *subarray* kita bedakan

Algoritma 21 Algoritma *merge sort*.

```

1: procedure MERGESORT(arr, left, right)
2:   if left = right then
3:     return                                     ▷ Tinggal 1 elemen, sudah pasti terurut
4:   else
5:     mid ← (left + right) div 2
6:     MERGESORT(arr, left, mid)
7:     MERGESORT(arr, mid + 1, right)
8:     MERGE(arr, left, mid, mid + 1, right)
9:   end if
10: end procedure

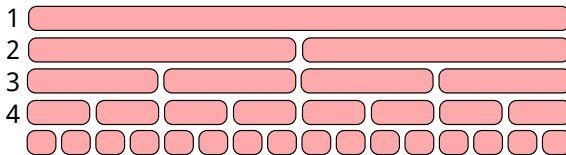
```

berdasarkan indeksnya, yang mana *array* pertama memiliki indeks $arr[aLeft..aRight]$ sementara *array* kedua memiliki indeks $arr[bLeft..bRight]$. Prosedur ini akan membuat *array* $arr[aLeft..bRight]$ menjadi terurut. Prosedur ini dapat dilihat pada Algoritma 22.

Analisis Algoritma Merge Sort

Misalkan N adalah ukuran dari *array*. Dengan proses membagi *array* menjadi dua *array* sama besar secara terus-menerus, banyak kedalaman rekursif dari *merge sort* adalah $O(\log N)$. Untuk setiap kedalaman, dilakukan *divide* dan *combine*.

Proses *divide* dan *conquer* bekerja dalam $O(1)$. Sedangkan proses *combine* melakukan operasi sebanyak elemen hasil penggabungan. Total elemen yang diproses di tiap kedalaman adalah $O(N)$.



Gambar 5.18: Visualisasi hasil pembagian di masing-masing kedalaman.

- Pada kedalaman 1, proses *combine* bekerja dalam $O(2 \times N/2)$. Total N elemen diproses.
- Pada kedalaman 2, proses *combine* bekerja dalam $O(4 \times N/4)$. Total N elemen diproses.
- Pada kedalaman 3, proses *combine* bekerja dalam $O(8 \times N/8)$. Total N elemen diproses.
- ... dan seterusnya

Dengan $O(\log N)$ menyatakan banyaknya kedalaman dan dibutuhkan operasi sebanyak $O(N)$ pada tiap kedalaman, total kompleksitas dari *merge sort* adalah $O(N \log N)$.

Bisa dibandingkan bahwa algoritma pengurutan menggunakan *merge sort* jauh lebih cepat dari algoritma pengurutan $O(N^2)$ seperti *bubble sort*. Karena itu, *merge sort*

Algoritma 22 Algoritma untuk menggabungkan 2 *array* yang terurut.

```

1: procedure MERGE(arr, aLeft, aRight, bLeft, bRight)
2:   size  $\leftarrow$  bRight - aLeft + 1           ▷ Hitung banyaknya elemen hasil gabungan
3:   temp  $\leftarrow$  new integer[size]         ▷ Buat array sementara
4:   tIndex  $\leftarrow$  0
5:   aIndex  $\leftarrow$  aLeft
6:   bIndex  $\leftarrow$  bLeft

7:   ▷ Selama kedua subarray masih ada isinya, ambil yang terkecil dan isi ke temp
8:   while (aIndex  $\leq$  aRight)  $\wedge$  (bIndex  $\leq$  bRight) do
9:     if arr[aIndex]  $\leq$  arr[bIndex] then
10:      temp[tIndex]  $\leftarrow$  arr[aIndex]
11:      aIndex  $\leftarrow$  aIndex + 1
12:    else
13:      temp[tIndex]  $\leftarrow$  arr[bIndex]
14:      bIndex  $\leftarrow$  bIndex + 1
15:    end if
16:    tIndex  $\leftarrow$  tIndex + 1
17:  end while
18:  ▷ Masukkan subarray yang masih bersisa ke temp
19:  ▷ Hanya salah satu dari kedua while ini yang akan dieksekusi
20:  while (aIndex  $\leq$  aRight) do
21:    temp[tIndex]  $\leftarrow$  arr[aIndex]
22:    aIndex  $\leftarrow$  aIndex + 1
23:    tIndex  $\leftarrow$  tIndex + 1
24:  end while
25:  while (bIndex  $\leq$  bRight) do
26:    temp[tIndex]  $\leftarrow$  arr[bIndex]
27:    bIndex  $\leftarrow$  bIndex + 1
28:    tIndex  $\leftarrow$  tIndex + 1
29:  end while

30:  ▷ Salin isi array temp ke arr
31:  for i  $\leftarrow$  0, size - 1 do
32:    arr[aLeft + i]  $\leftarrow$  temp[i]
33:  end for
34: end procedure           ▷ Selesai, kini arr[aLeft..bRight] telah terurut

```

mampu mengurutkan *array* dengan ratusan ribu elemen dalam waktu singkat.

Merge sort memiliki sifat **stable**. Artinya jika ada dua elemen a_1 dan a_2 yang bernilai sama ($a_1 = a_2$) dan pada awalnya a_1 terletak sebelum a_2 , maka setelah diurutkan dijamin a_1 akan tetap terletak sebelum a_2 .

Studi Kasus 2: *Quicksort*

Terdapat algoritma pengurutan selain *merge sort* yang memiliki kompleksitas $O(N \log N)$, salah satunya adalah ***quicksort***. *Quicksort* menggunakan prinsip *divide and conquer* dalam pengurutannya. Tahapan dari *quicksort* adalah sebagai berikut:

1. *Divide*: pilih suatu elemen yang kita sebut *pivot*, kemudian kita bagi *array* menjadi dua sehingga salah satunya selalu \leq *pivot* dan yang sisanya selalu $>$ *pivot*. Kemudian, lakukan *quicksort* pada masing-masing subarray tersebut.
2. *Conquer*: ketika *array* hanya memiliki satu elemen, *array* tersebut sudah terurut.
3. *Combine*: gabungkan *subarray* dengan menempelkan hasil *quicksort* bagian kiri dan kanan.

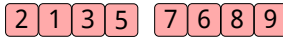
Kita ingin mengurutkan *array* [5, 2, 7, 6, 1, 8, 9, 3] secara menaik. Pilih salah satu elemen, misalnya 5. Kita sebut elemen ini sebagai ***pivot*** (pijakan).

Kemudian pindahkan seluruh elemen yang lebih kecil sama dengan *pivot* (≤ 5) ke sebelah kiri dari *pivot*, dan seluruh elemen yang lebih besar dari *pivot* (> 5) ke sebelah kanan dari *pivot*. Urutan elemen-elemen di satu bagian (sebelah kiri *pivot* atau kanan *pivot*) setelah pemindahan tidaklah penting (boleh acak). Gambar 5.20 merupakan contoh hasil dari proses ini.



5 2 7 6 1 8 9 3

Gambar 5.19: Awal mula *array*.



2 1 3 5 7 6 8 9

Gambar 5.20: Hasil *array* setelah proses *divide* dengan *pivot* 5. Semua elemen yang berada di sisi kiri memiliki nilai ≤ 5 , sementara elemen pada sisi kanan memiliki nilai > 5 .

Setelah didapatkan kedua *array* tersebut, lakukan *quicksort* serupa untuk bagian kiri dan kanan secara rekursif. *Base case* dari rekursi ini adalah ketika *array* yang hendak diurutkan hanya memiliki satu elemen. Dengan langkah tersebut, suatu ketika seluruh *array* akan menjadi terurut.

Partisi *Quicksort*

Bagian utama dari *quicksort* adalah proses partisi (bagian *divide*). Sebelum melakukan partisi, pilih satu elemen yang akan dijadikan *pivot*. Kemudian, lakukan partisi supaya

seluruh elemen yang $\leq pivot$ berada di sebelah kiri dari $pivot$, dan yang $> pivot$ di sebelah kanannya. Pemilihan elemen $pivot$ dapat dilakukan secara bebas. Bahkan pemilihan $pivot$ dengan cara tertentu dapat mempengaruhi performa algoritma *quicksort*. Pilihan yang cukup sering dipakai adalah menggunakan elemen di tengah *array* sebagai $pivot$.

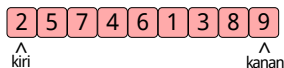
Setelah elemen $pivot$ ditentukan, bagaimana cara mempartisi *array*? Kita bisa saja menggunakan sebuah perulangan sebanyak $O(N)$ dan menampung hasil partisi di suatu *array* sementara, kemudian menempatkan kembali hasil partisi ke *array* sebenarnya. Namun cara ini agak merepotkan karena kita perlu membuat *array* sementara dan memindahkan isi dari *array*.

Partisi Hoare

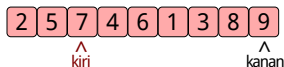
Ada beberapa algoritma untuk melakukan partisi secara *in place* atau tanpa *array* sementara. Kita akan membahas salah satu algoritma partisi yaitu algoritma partisi **Hoare**. Cara kerja algoritma Hoare adalah sebagai berikut:

1. Pilih sebuah $pivot$.
2. Siapkan dua variabel penunjuk, *kiri* dan *kanan* dengan masing-masing variabel menunjuk ujung kiri dan ujung kanan dari *array*.
3. Gerakkan variabel *kiri* ke arah kanan, sampai elemen yang ditunjuk $> pivot$.
4. Gerakkan variabel *kanan* ke arah kiri, sampai elemen yang ditunjuk $\leq pivot$.
5. Jika $kiri \leq kanan$, tukar elemen yang ditunjuk *kiri* dan *kanan*, kemudian gerakkan *kiri* ke kanan satu langkah dan *kanan* ke kiri satu langkah.
6. Jika $kiri \leq kanan$, maka kembali ke tahap 3. Jika tidak, selesai.

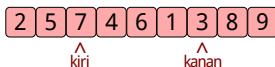
Mari kita terapkan algoritma tersebut. Asumsikan kita akan melakukan partisi pada suatu *array* [2,5,7,4,6,1,3,8,9]. Misalkan kita memilih $pivot = 5$. Simak Gambar 5.21 hingga Gambar 5.30 yang mengilustrasikan algoritma Hoare dari awal hingga akhir.



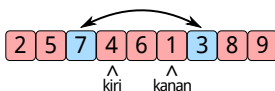
Gambar 5.21: Langkah 1: Kita mulai dengan membuat dua variabel penunjuk, yaitu *kiri* dan *kanan* dengan masing-masing variabel menunjuk ujung kiri dan ujung kanan dari *array*.



Gambar 5.22: Gerakkan variabel *kiri* ke arah kanan, sampai elemen yang ditunjuk $> pivot$.



Gambar 5.23: Gerakkan variabel *kanan* ke arah kiri, sampai elemen yang ditunjuk \leq *pivot*.



Gambar 5.24: Karena $kiri \leq kanan$, tukar elemen yang ditunjuk *kiri* dan *kanan*, lalu gerakkan *kiri* ke kanan satu langkah serta *kanan* ke kiri satu langkah.



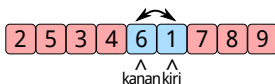
Gambar 5.25: Karena $kiri \leq kanan$, artinya partisi belum selesai.



Gambar 5.26: Gerakkan kembali variabel *kiri* ke arah kanan, sampai elemen yang ditunjuk $>$ *pivot*.



Gambar 5.27: Gerakkan variabel *kanan*. Kebetulan, elemen yang ditunjuk sudah \leq *pivot*.



Gambar 5.28: Karena $kiri \leq kanan$, tukar dan gerakkan variabel *kiri* dan *kanan* satu langkah.



Gambar 5.29: Kini sudah tidak $kiri \leq kanan$, artinya partisi selesai.



Gambar 5.30: Hasil akhir partisi. Perhatikan bahwa seluruh elemen yang $\leq pivot$ berada di kiri, dan sisanya di kanan.

Implementasi Partisi Hoare

Implementasi partisi Hoare terdapat di Algoritma 23.

Algoritma 23 Implementasi partisi Hoare.

```

1: procedure PARTITION(arr, left, right, pivot)
2:   pLeft  $\leftarrow$  left
3:   pRight  $\leftarrow$  right
4:   while pLeft  $\leq$  pRight do
5:     while arr[pLeft]  $\leq$  pivot do
6:       pLeft  $\leftarrow$  pLeft + 1
7:     end while
8:     while arr[pRight]  $>$  pivot do
9:       pRight  $\leftarrow$  pRight - 1
10:    end while
11:    if pLeft  $\leq$  pRight then
12:      SWAP(arr[pLeft], arr[pRight])
13:      pLeft  $\leftarrow$  pLeft + 1
14:      pRight  $\leftarrow$  pRight - 1
15:    end if
16:  end while
17: end procedure
  
```

Analisis Algoritma Partisi Hoare

Terdapat dua variabel penunjuk, yang setiap langkahnya selalu bergerak ke satu arah tanpa pernah mundur. Algoritma berhenti ketika variabel *kiri* dan *kanan* bertemu. Artinya, setiap elemen *array* dikunjungi tepat satu kali. Kompleksitasnya adalah $O(N)$.

Implementasi Quicksort

Setelah kita mengimplementasikan algoritma partisi, mengintegrasikannya ke *quicksort* cukup mudah. Implementasinya terdapat di Algoritma 24.

Algoritma 24 Implementasi *quicksort*.

```

1: procedure QUICKSORT(arr, left, right)
2:   if left ≥ right then
3:     return                                     ▷ Tidak ada elemen yang perlu diurutkan
4:   else
5:     pivot ← arr[(left + right) div 2]

6:     pLeft ← left
7:     pRight ← right
8:     while pLeft ≤ pRight do
9:       while arr[pLeft] ≤ pivot do
10:        pLeft ← pLeft + 1
11:      end while
12:      while arr[pRight] > pivot do
13:        pRight ← pRight - 1
14:      end while
15:      if pLeft ≤ pRight then
16:        SWAP(arr[pLeft], arr[pRight])
17:        pLeft ← pLeft + 1
18:        pRight ← pRight - 1
19:      end if
20:    end while

21:    ▷ Sampai saat ini, dipastikan pRight < pLeft
22:    QUICKSORT(left, pRight)
23:    QUICKSORT(pLeft, right)
24:  end if
25: end procedure

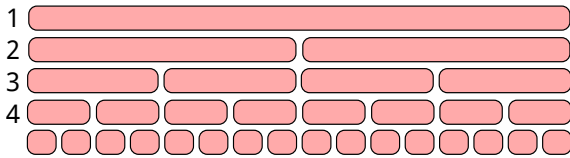
```

Analisis Algoritma Quicksort

Pada setiap kedalaman rekursi, *array* hasil partisi belum tentu memiliki ukuran yang sama. Hasil partisi bergantung pada nilai *pivot* yang kita pilih. Karena hal ini, kita dapat menganalisis kompleksitas *quicksort* dalam 3 kasus:

Kasus Terbaik

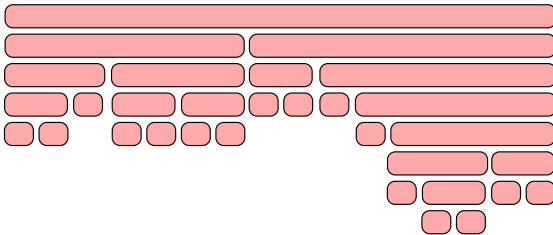
Pembelahan menjadi dua *subarray* sama besar menjamin kedalaman rekursif **sedang-kal mungkin**. Sehingga untuk kasus terbaik, jalannya algoritma menjadi seperti *merge sort* dan bekerja dalam $O(N \log N)$.



Gambar 5.31: Contoh hasil partisi untuk kasus terbaik.

Kasus Rata-Rata

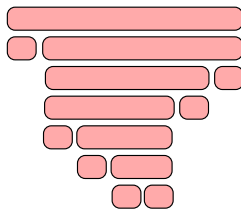
Pada kebanyakan kasus, ukuran hasil partisi berbeda-beda. Secara rata-rata kompleksitasnya masih dapat dianggap $O(N \log N)$.¹⁰



Gambar 5.32: Contoh hasil partisi untuk kasus rata-rata.

Kasus Terburuk

Kasus paling buruk: ukuran hasil partisi sangat timpang. Hal ini terjadi ketika partisi membagi *array* berukuran N menjadi dua *subarray* yang masing-masing berukuran $N - 1$ dan 1. Dalam kasus ini, kedalaman rekursif menjadi mendekati N yang mengakibatkan kompleksitas total *quicksort* menjadi $O(N^2)$.



Gambar 5.33: Contoh hasil partisi untuk kasus terburuk.

Namun tidak perlu khawatir karena dalam kondisi normal, peluang terjadinya kasus

terburuk sangat kecil. Artinya, pada sebagian besar kasus, *quicksort* berjalan dengan sangat cepat.

Pemilihan Pivot

Terdapat beberapa strategi pemilihan *pivot* untuk mencegah hasil partisi yang terlalu timpang:

- Pilih salah satu elemen secara acak, ketimbang selalu memilih elemen di tengah.
- Pilih median dari elemen paling depan, tengah, dan paling belakang.

Quicksort memiliki sifat **tidak stable**. Artinya jika dua elemen a_1 dan a_2 memenuhi:

- memiliki yang nilai sama, dan
- sebelum diurutkan a_1 terletak sebelum a_2 ,

maka setelah diurutkan, **tidak** dijamin a_1 akan tetap terletak sebelum a_2 .

Studi Kasus 3: Mencari Nilai Terbesar

Perhatikan contoh soal Nilai Terbesar berikut.

Contoh Soal 5.1: Nilai Terbesar

Diberikan sebuah *array* A yang memiliki N bilangan, carilah nilai terbesar yang ada pada A !

Masalah ini dapat diselesaikan dengan mudah menggunakan perulangan biasa. Namun, sekarang mari kita coba selesaikan dengan *divide and conquer*.

Pertama kita definisikan tahap-tahapnya:

- *Divide*: jika *array* berukuran besar (lebih dari satu elemen), bagi menjadi dua *subarray*. Kemudian, lakukan pencarian secara rekursif pada masing-masing *subarray* tersebut.
- *Conquer*: ketika *array* hanya berisi satu elemen, nilai terbesarnya pasti adalah elemen tersebut.
- *Combine*: nilai terbesar dari *array* adalah maksimum dari nilai terbesar di *subarray* pertama dan nilai terbesar di *subarray* kedua.

Implementasi Pencarian Nilai Terbesar

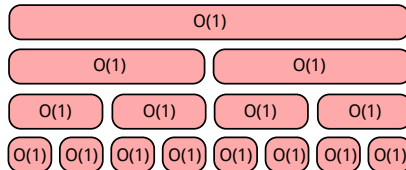
Ide pencarian nilai terbesar secara *divide and conquer* terdapat di Algoritma 25.

Algoritma 25 Pencarian nilai terbesar secara *divide and conquer*.

```

1: function FINDMAX(arr, left, right)
2:   if left = right then
3:     return arr[left]
4:   else
5:     mid ← (left + right) div 2
6:     leftMax ← FINDMAX(arr, left, mid)
7:     rightMax ← FINDMAX(arr, mid + 1, right)
8:     return max(leftMax, rightMax)
9:   end if
10: end function

```

Analisis Algoritma Mencari Nilai TerbesarGambar 5.34: Pembagian dan kompleksitas untuk menyelesaikan tiap *subarray*.

Setiap operasi *divide*, *conquer*, dan *combine* bekerja dalam $O(1)$. Pada tiap kedalaman, kompleksitas yang dibutuhkan adalah sebanyak *subarray* pada kedalaman tersebut. Kedalaman ke- i (dimulai dari 1) memiliki banyak *subarray* sebanyak 2^{i-1} . Sehingga kedalaman ke-1 memiliki $2^0 = 1$ *subarray*, kedalaman ke-2 memiliki $2^1 = 2$ *subarray*, kedalaman ke-3 memiliki $2^2 = 4$ *subarray*, dan seterusnya.

Maka operasi *conquer* dilaksanakan sebanyak $1 + 2 + 4 + 8 + \dots + 2^L$ kali, dengan L adalah banyak kedalaman rekursif dari pembagian *array*, yaitu $L = \log N$. Dengan $2^L \approx N$, keseluruhan total operasi adalah $1 + 2 + 4 + 8 + \dots + 2^L = 2^{L+1} - 1 = 2N$ operasi. Kompleksitas akhirnya menjadi $O(N)$.

Ternyata, strategi ini tidak lebih baik dari mencari nilai maksimum satu per satu menggunakan perulangan. Kesimpulannya, *divide and conquer* **tidak selalu** dapat mengurangi kompleksitas solusi naif.

6 Greedy

Greedy merupakan sebuah teknik dalam strategi penyelesaian masalah, bukan suatu algoritma khusus. Teknik *greedy* biasanya memiliki waktu eksekusi yang cepat dan biasanya mudah untuk diimplementasikan, namun terkadang sulit dibuktikan kebenarannya. Pada bab ini kita akan mempelajari cara *greedy* bekerja melalui pembahasan permasalahan *greedy* klasik: *Activity Selection*.

Konsep Greedy

Suatu persoalan dapat diselesaikan dengan teknik *greedy* jika persoalan tersebut memiliki memiliki sifat berikut:

- Solusi optimal dari persoalan dapat ditentukan dari solusi optimal subpersoalan tersebut.
- Pada setiap subpersoalan, ada suatu langkah yang bisa dilakukan yang mana langkah tersebut menghasilkan solusi optimal pada subpersoalan tersebut. Langkah ini disebut juga ***greedy choice***.

Untuk menguasai teknik *greedy*, diperlukan banyak latihan. Sebagai awalan, kita akan mempelajari salah satu permasalahan *greedy* klasik yang biasa disebut *Activity Selection*.

Contoh Soal 6.1: *Activity Selection*

Anda diberikan N buah aktivitas. Aktivitas ke- i dinyatakan dalam $\langle a_i.start, a_i.end \rangle$. Artinya, aktivitas ini dimulai pada waktu $a_i.start$ dan berakhir pada waktu $a_i.end$. Pada setiap satuan waktu, Anda dapat mengikuti paling banyak satu aktivitas. Dengan kata lain, Anda dapat mengikuti dua aktivitas i dan j jika $a_i.end \leq a_j.start$ atau $a_j.end \leq a_i.start$. Anda ingin mengatur jadwal sedemikian sehingga Anda bisa mengikuti aktivitas sebanyak mungkin.

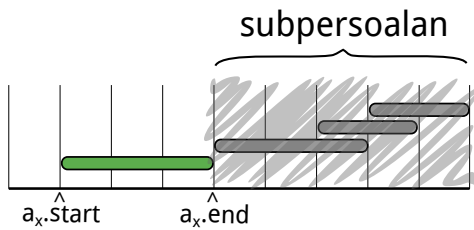
Contoh

Aktivitas: $[\langle 1, 3 \rangle, \langle 2, 6 \rangle, \langle 5, 7 \rangle, \langle 8, 9 \rangle]$

Solusi: Anda dapat hadir di 3 aktivitas berbeda yang tidak saling tumpang tindih, yaitu $\langle 1, 3 \rangle$, $\langle 5, 7 \rangle$, dan $\langle 8, 9 \rangle$

Menemukan Subpersoalan

Langkah awal dalam menyelesaikan permasalahan dengan teknik *greedy* adalah memastikan bahwa soal dapat dipecah menjadi subpersoalan yang lebih kecil.



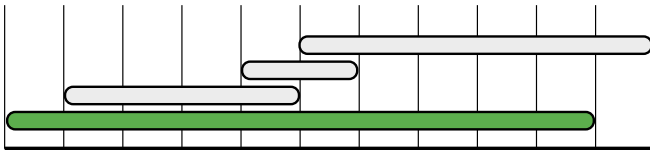
Gambar 6.1: Subpersoalan pada *activity selection*.

Bagaimana caranya menemukan subpersoalan dari *activity selection*? Diberikan daftar kegiatan yang dinomori 1 sampai N . Kita asumsikan daftar ini terurut berdasarkan nilai $a_i.start$. Asumsikan kegiatan pertama yang kita ikuti adalah kegiatan ke- x . Maka, kegiatan selanjutnya yang diikuti haruslah memiliki waktu awal $\geq a_x.end$. Jika j adalah nilai terkecil yang memenuhi $a_j.start \geq a_x.end$, maka kita bisa mendapatkan subpersoalan *activity selection* dengan kegiatan dari j sampai N , seperti yang diilustrasikan pada Gambar 6.1.

Greedy Choice

Persoalan yang dapat dicari subpersoalannya belum tentu dapat diselesaikan dengan teknik *greedy*. Pada setiap subpersoalan, harus terdapat langkah yang bisa dilakukan yang mana dapat menghasilkan solusi optimal pada subpersoalan tersebut, atau biasa disebut *greedy choice*. Pada persoalan *activity selection*, ada beberapa langkah yang dapat dilakukan, diantaranya memilih aktivitas dengan waktu mulai paling awal, aktivitas dengan durasi tersingkat, ataupun aktivitas dengan waktu akhir paling awal. Namun, tidak semua pilihan tersebut merupakan pilihan yang optimal.

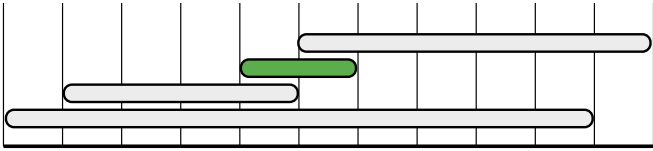
- Memilih aktivitas dengan waktu mulai paling awal.



Gambar 6.2: Kasus yang mana memilih aktivitas dengan waktu mulai paling awal tidak optimal.

Sekilas, pemilihan ini merupakan langkah yang optimal. Namun, bisa jadi ada aktivitas yang mulai lebih awal, tetapi memiliki durasi yang sangat panjang sehingga menyita waktu, seperti yang diilustrasikan pada Gambar 6.2. Dengan kata lain, pilihan ini **tidak selalu** menghasilkan solusi optimal.

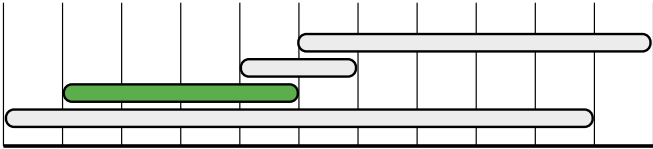
- Memilih aktivitas dengan durasi paling singkat.



Gambar 6.3: Kasus yang mana memilih aktivitas dengan durasi tersingkat tidak optimal.

Serupa dengan pemilihan sebelumnya, ternyata pemilihan ini tidak menjamin solusi yang optimal. Seperti yang terlihat pada Gambar 6.3, pemilihan ini dapat memotong dua aktivitas lain yang sebenarnya dapat kita ikuti.

- Memilih aktivitas dengan waktu akhir paling awal.

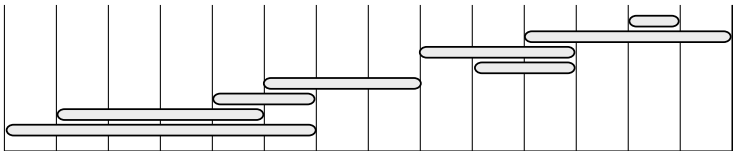


Gambar 6.4: Contoh pemilihan aktivitas dengan waktu akhir paling awal.

Dengan memilih aktivitas yang selesai lebih awal, kita mempunyai sisa waktu lebih banyak untuk aktivitas lainnya. Tanpa peduli kapan aktivitas ini mulai atau berapa durasinya, memilih yang selesai lebih awal **pasti menguntungkan**. Pilihan ini adalah merupakan *greedy choice*, yang **selalu** menghasilkan solusi optimal.

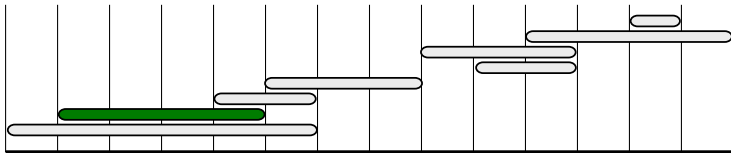
Penyelesaian dengan Teknik *Greedy*

Secara umum, persoalan dapat diselesaikan dengan teknik *greedy* dengan melakukan *greedy choice* untuk mendapatkan subpersoalan, kemudian secara rekursif menyelesaikan subpersoalan tersebut. Pada kasus *activity selection*, kita dapat menentukan aktivitas yang akan diikuti pertama kali. Selanjutnya kita mendapatkan subpersoalan, yang ternyata dapat diselesaikan dengan cara serupa!

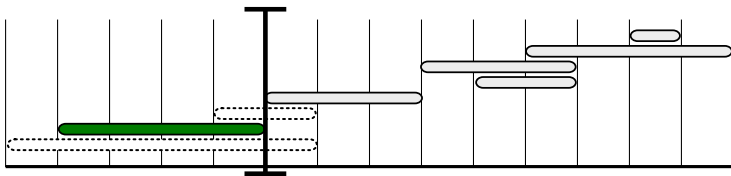


Gambar 6.5: Konfigurasi masukan awal.

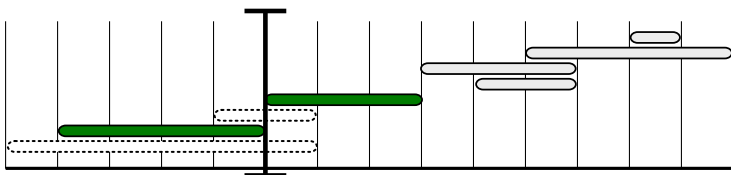
Berikut simulasi penyelesaian *activity selection* menggunakan *greedy*. Misalkan, Anda diberikan interval-interval aktivitas $\langle 1,7 \rangle$, $\langle 2,6 \rangle$, $\langle 5,7 \rangle$, $\langle 6,9 \rangle$, $\langle 9,12 \rangle$, $\langle 10,12 \rangle$, $\langle 11,15 \rangle$, $\langle 13,14 \rangle$ yang divisualisasikan di Gambar 6.5. Kita akan melakukan serangkaian *greedy choice* yang disimulasikan pada Gambar 6.6 sampai Gambar 6.13.



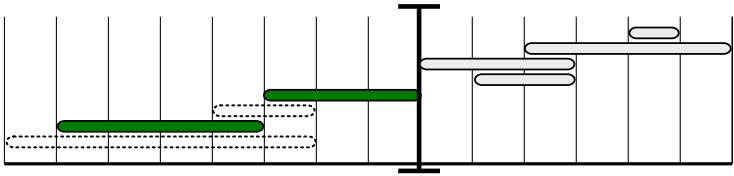
Gambar 6.6: Langkah 1: Pilihan yang ada: $\langle 1,7 \rangle$, $\langle 2,6 \rangle$, $\langle 5,7 \rangle$, $\langle 6,9 \rangle$, $\langle 9,12 \rangle$, $\langle 10,12 \rangle$, $\langle 11,15 \rangle$, $\langle 13,14 \rangle$. Kita pilih interval $\langle 2,6 \rangle$ karena interval tersebut memiliki waktu selesai paling awal.



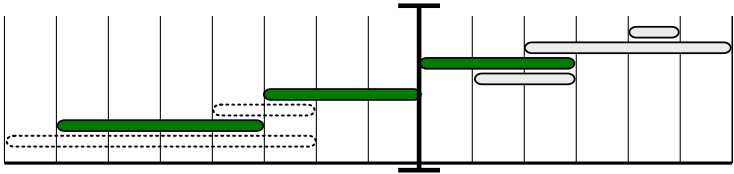
Gambar 6.7: Langkah 2: Setelah memilih $\langle 2,6 \rangle$, kita dapatkan subpersoalan dengan pilihan yang tersedia berupa: $\langle 6,9 \rangle$, $\langle 9,12 \rangle$, $\langle 10,12 \rangle$, $\langle 11,15 \rangle$, $\langle 13,14 \rangle$.



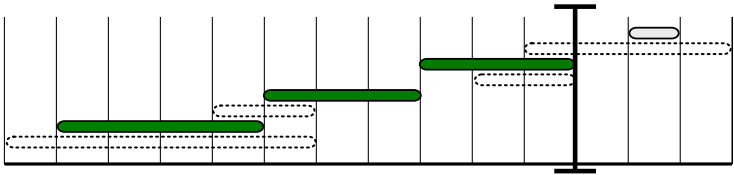
Gambar 6.8: Langkah 3: Memilih interval $\langle 6,9 \rangle$ karena interval tersebut memiliki waktu selesai paling awal.



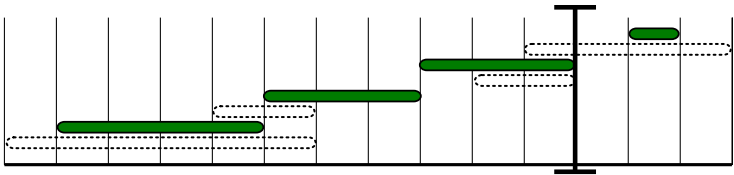
Gambar 6.9: Langkah 4: Mendapatkan subpersoalan dengan pilihan tersisa: $\langle 9, 12 \rangle$, $\langle 10, 12 \rangle$, $\langle 11, 15 \rangle$, $\langle 13, 14 \rangle$.



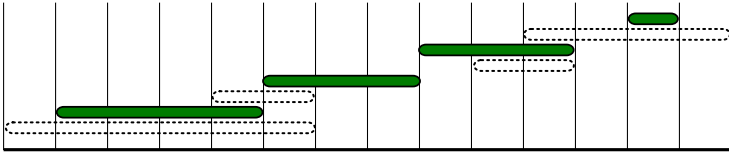
Gambar 6.10: Langkah 5: Memilih interval $\langle 9, 12 \rangle$ karena interval tersebut memiliki waktu selesai paling awal.



Gambar 6.11: Langkah 6: Mendapatkan subpersoalan dengan pilihan tersisa: $\langle 13, 14 \rangle$.



Gambar 6.12: Langkah 7: Memilih interval $\langle 13, 14 \rangle$.



Gambar 6.13: Langkah 8: Sudah tidak ada pilihan tersedia. Langkah selesai.

Dengan melakukan proses tersebut, kita bisa mendapatkan 4 interval. Tidak ada cara lain yang memberikan hasil lebih optimal. Implementasi dari algoritma tersebut dapat dilihat pada Algoritma 26:

Algoritma 26 Penyelesaian *activity selection*.

```

1: function SOLVEACTIVITYSELECTION( $a, N$ )
2:   SORTBYENDTIME( $a, N$ )           ▷ Urutkan  $a$  secara menaik berdasarkan  $a[i].end$ 
3:    $selectedCount \leftarrow 0$ 
4:    $startTime \leftarrow 0$ 
5:   for  $i \leftarrow 1, N$  do
6:     if ( $a[i].start \geq startTime$ ) then
7:        $selectedCount \leftarrow selectedCount + 1$ 
8:        $startTime \leftarrow a[i].end$ 
9:     end if
10:  end for
11:  return  $selectedCount$ 
12: end function

```

Mari kita analisis kompleksitas dari algoritma tersebut. Mengurutkan aktivitas berdasarkan waktu berakhirnya dapat dilakukan dalam $O(N \log N)$. Setelah diurutkan, pemilihan aktivitas dapat dilakukan dalam $O(N)$. Maka, kompleksitas akhirnya adalah $O(N \log N)$.

Permasalahan pada Algoritma Greedy

Greedy choice memungkinkan kita untuk memilih suatu keputusan yang dijamin akan menghasilkan solusi optimal, tanpa peduli ke depannya seperti apa. Hal ini memberi kesan "rakus", yaitu hanya mementingkan masalah yang sedang dihadapi dan selalu mengambil keputusan terbaik saat ini. Inilah sebabnya teknik ini dinamakan *greedy*.

Karena sifatnya yang "rakus", ada persoalan-persoalan yang tidak dapat diselesaikan dengan *greedy*. Perhatikan contoh soal berikut:

Contoh Soal 6.2: Penukaran Uang

Anda ingin menukar uang Rp12.000 dengan lembaran uang kertas Rp5.000, Rp2.000, dan Rp1.000. Anda ingin menukar dengan jumlah lembaran sesedikit mungkin.

Pada soal tersebut, *greedy choice* yang terpikirkan adalah dengan memilih lembaran dengan nominal terbesar yang mungkin untuk tiap subpersoalan. Pertama kita pilih lembaran Rp5.000, sehingga tersisa Rp7.000 lagi yang harus dipecah. Selanjutnya kita pilih Rp5.000 lagi dan menyisakan Rp2.000 untuk dipecah. Akhirnya, kita pilih Rp2.000 sebagai pecahan terakhir. Solusi dari kasus ini adalah dengan menggunakan 3 lembaran.

Dengan soal yang sama, bagaimana jika lembaran rupiah yang beredar bernilai Rp5.000, **Rp4.000**, dan Rp1.000? Dengan algoritma *greedy*, kita akan menukar Rp12.000 dengan lembaran Rp5.000, Rp5.000, Rp1.000, dan Rp1.000. Padahal ada solusi yang lebih baik, yaitu menggunakan 3 lembaran Rp4.000. Pada kasus tersebut, *greedy choice* yang tidak selalu dapat menghasilkan solusi optimal. Permasalahan ini tidak dapat diselesaikan oleh algoritma *greedy*. Permasalahan ini bisa diselesaikan dengan algoritma *dynamic programming*, yang akan dibahas pada Bab 7.

Pembuktian Greedy

Biasanya akan ada beberapa pilihan *greedy choice* yang ada, yang mana tidak semuanya bisa menghasilkan solusi optimal. Ketika menemukan suatu *greedy choice*, sangat dianjurkan untuk menguji kebenaran dari pilihan tersebut sebelum diimplementasikan. Namun, **pembuktian kebenaran algoritma *greedy* tidaklah mudah.**

Pengujian yang dapat dilakukan adalah dengan mencoba membuat contoh kasus yang dapat menggagalkan *greedy choice* tersebut. Teknik ini biasa disebut **proof by counterexample**. Jika ditemukan satu saja contoh kasus yang mana *greedy choice* yang diajukan tidak menghasilkan solusi optimal, maka *greedy choice* tersebut dinyatakan salah. Namun, bila Anda tidak bisa menemukan *counterexample*, **belum tentu** algoritma Anda benar.

Algoritma *greedy* terkadang mudah untuk dipikirkan dan mudah untuk diimplementasikan, namun sulit untuk dibuktikan kebenarannya. Pembuktian kebenaran algoritma *greedy* bisa jadi membutuhkan pembuktian matematis yang kompleks dan memakan waktu. Oleh karena itu, pada suasana kompetisi, intuisi dan pengalaman sangat membantu untuk menyelesaikan soal bertipe *greedy*. Berhati-hati dan teliti saat mengerjakan soal bertipe *greedy*. Perhatikan setiap detail yang ada, karena bisa berakibat fatal.

7 Dynamic Programming

Dynamic programming (DP) sejatinya adalah *brute force* yang dioptimasi. Seperti *greedy*, persoalan yang dapat diselesaikan dengan DP jika persoalan tersebut memiliki subpersoalan. Perbedaannya, DP tidak melakukan *greedy choice*, melainkan mencoba seluruh kemungkinan yang ada. DP ini sendiri merupakan topik yang sangat luas dan memerlukan latihan yang mendalam agar dapat menguasai topik ini.

Konsep *Dynamic Programming*

DP merupakan metode penyelesaian persoalan yang melibatkan pengambilan keputusan dengan memanfaatkan informasi dari penyelesaian subpersoalan yang sama namun lebih kecil. Solusi subpersoalan tersebut hanya dihitung satu kali dan disimpan di memori. Sama seperti *greedy*, solusi DP dapat bekerja hanya jika solusi optimal dari persoalan dapat ditentukan dari solusi optimal subpersoalan tersebut.

Terdapat dua cara mengimplementasikan DP:

- **Top-down**: diimplementasikan secara rekursif sambil mencatat nilai yang sudah ditemukan (memoisasi).
- **Bottom-up**: diimplementasikan secara iteratif dengan menghitung mulai dari kasus yang kecil ke besar.

Kita akan menggunakan contoh soal *Coin Change* berikut sebagai sarana mempelajari teknik DP.

Contoh Soal 7.1: *Coin Change*

Diberikan M jenis koin, masing-masing jenis bernilai a_1, a_2, \dots, a_M rupiah. Asumsikan banyaknya koin untuk setiap nominal yang ada tak terbatas. Tentukan banyaknya koin paling sedikit untuk membayar **tepat** sebesar N rupiah!

Contoh soal tersebut sudah dibahas pada bab sebelumnya dan dapat dibuktikan bahwa soal tersebut tidak dapat diselesaikan dengan *greedy*. Mari kita selesaikan soal tersebut menggunakan DP.

Top-down

Pendekatan *top-down* biasa juga disebut **memoisasi**. Kata memoisasi berasal dari "memo", yang artinya catatan. Pada *top-down*, penyelesaian masalah dimulai dari kasus yang besar. Untuk menyelesaikan kasus yang besar, dibutuhkan solusi dari kasus yang lebih kecil. Karena solusi kasus yang lebih kecil belum ada, maka kita akan mencarinya terlebih dahulu, lalu mencatat hasilnya. Hal ini dilakukan secara rekursif.

Pada soal *Coin Change*, anggaplah kita membayar N rupiah dengan koin-koin **satu per satu**. Pastilah terdapat **koin pertama** yang kita bayarkan. Jika nilai koin itu adalah a_k , maka **sisanya** yang perlu kita bayar adalah $N - a_k$. Dalam kasus ini, terdapat M **pilihan** koin untuk a_k . Perhatikan bahwa penukaran $N - a_k$ merupakan suatu subpersoalan

yang serupa dengan persoalan awalnya. Artinya, cara yang sama untuk menyelesaikan subpersoalan dapat digunakan sehingga strategi penyelesaian secara rekursif dapat digunakan.

Solusi Rekursif

Definisikan sebuah fungsi $f(x)$ sebagai banyaknya koin minimum yang dibutuhkan untuk membayar tepat x rupiah. Kita dapat mencoba-coba satu koin yang ingin kita gunakan. Jika suatu koin a_k digunakan, maka kita membutuhkan $f(x - a_k)$ koin ditambah satu koin a_k , atau dapat ditulis $f(x) = f(x - a_k) + 1$. Pencarian nilai $f(x - a_k)$ dilakukan secara rekursif, kita kembali mencoba-coba koin yang ingin digunakan. Pilihan yang terbaik akan memberikan nilai $f(x - a_k) + 1$ sekecil mungkin, sehingga kita cukup mencoba semua kemungkinan a_k , dan ambil yang hasil $f(x - a_k) + 1$ terkecil.

Dalam solusi rekursif, kita harus menentukan **base case** dari $f(x)$. Kasus terkecilnya adalah $f(0)$, yang artinya kita hendak membayar 0 rupiah. Membayar 0 rupiah tidak membutuhkan satu pun koin, sehingga $f(0) = 0$. Secara matematis, hubungan rekursif ini dituliskan:

$$f(x) = \begin{cases} 0, & x = 0 \\ \min_{\substack{1 \leq k \leq M \\ a_k \leq x}} f(x - a_k) + 1, & x > 0 \end{cases}$$

Kita implementasikan $f(x)$ sebagai fungsi $\text{SOLVE}(x)$ pada Algoritma 27.

Algoritma 27 Penyelesaian *Coin Change* secara rekursif.

```

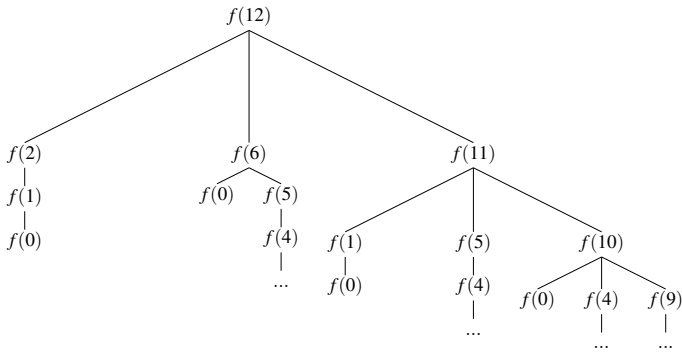
1: function SOLVE(x)
2:   if (x = 0) then
3:     return 0
4:   end if

5:   best ← ∞
6:   for k ← 1, M do
7:     if (ak ≤ x) then
8:       best ← min(best, SOLVE(x - ak) + 1)
9:     end if
10:  end for
11:  return best
12: end function

```

►► Sekilas Info ◀◀

Pada penulisan program, nilai ∞ dapat diwujudkan dengan nilai yang mendekati batas atas tipe data bilangan. Misalnya pada `integer` 32 bit, nilai yang dapat digunakan adalah 2.000.000.000. Nilai ini dapat disesuaikan menjadi nilai yang lebih besar atau kecil, bergantung pada soal dan mungkin-tidaknya nilai ini dioperasikan untuk menghindari *integer overflow*.



Gambar 7.1: Pohon rekursi setelah pemanggilan $f(12)$.

Jawaban akhirnya ada pada $SOLVE(N)$.

Mari kita lihat pohon rekursi yang dihasilkan oleh fungsi f . Pohon rekursi untuk $f(12)$ dengan nominal koin 1, 6, dan 10 rupiah dapat dilihat pada Gambar 7.1.

Jika diperhatikan pada pohon rekursi, terdapat $O(M)$ cabang untuk setiap pemanggilan f . Untuk menghitung nilai $f(N)$, kita akan memiliki pohon rekursi yang kira-kira sedalam $O(N)$. Berarti kira-kira dilakukan $O(M^N)$ pemanggilan fungsi. Karena itu, solusi ini membutuhkan $O(M^N)$ operasi, yang mana banyaknya operasi ini eksponensial.

Biasanya solusi eksponensial berjalan sangat lambat. Cobalah Anda hitung nilai $O(M^N)$ dengan $M = 3$ dan $N = 100$, untuk menyadari betapa lambatnya solusi ini! Tentu kita tidak ingin memiliki solusi eksponensial pada pemrograman kompetitif, kecuali pada soal-soal tertentu yang tidak memiliki solusi polinomial. Karena itu, kita harus melakukan sebuah optimisasi.

Optimisasi dengan Memoisasi

Jika diperhatikan, ternyata banyak $f(x)$ yang dihitung berkali-kali. Sebagai contoh, $f(5)$ dan $f(4)$ pada pohon rekursi di Gambar 7.1 muncul berkali-kali. Kita dapat melakukan memoisasi agar nilai $f(x)$ hanya dihitung tepat satu kali saja.

Perhatikan bahwa hanya ada $N + 1$ kemungkinan x untuk $f(x)$, yaitu 0 sampai N . Kita dapat mencatat hasil perhitungan $f(x)$ setelah menghitungnya. Jika suatu ketika kita kembali memerlukan nilai $f(x)$, kita tidak perlu menghitungnya kembali.

Algoritma 28 merupakan implementasi penyelesaian *Coin Change* dengan memoisasi. Asumsikan untuk menghitung suatu nilai $f(x)$, kita membutuhkan $O(M)$ iterasi. Maka, untuk menghitung nilai $f(x)$ untuk seluruh x , kita membutuhkan $O(NM)$ operasi. Banyaknya operasi ini polinomial terhadap N dan M , dan **jauh lebih cepat** daripada solusi rekursif sebelumnya.

Algoritma 28 Penyelesaian *Coin Change* secara *top-down*.

```

1: function SOLVE( $x$ )
2:   if ( $x = 0$ ) then
3:     return 0
4:   end if
5:   if computed[ $x$ ] then
6:     return memo[ $x$ ]
7:   end if
8:    $best \leftarrow \infty$ 
9:   for  $k \leftarrow 1, M$  do
10:    if ( $a_k \leq x$ ) then
11:       $best \leftarrow \min(best, SOLVE(x - a_k) + 1)$ 
12:    end if
13:  end for
14:  computed[ $x$ ]  $\leftarrow true$ 
15:  memo[ $x$ ]  $\leftarrow best$ 
16:  return  $best$ 
17: end function

```

▷ Sudah pernah dihitung, langsung kembalikan

▷ Tandai bahwa sudah pernah dihitung

▷ Simpan hasil perhitungan

Analisis Top-down

Top-down merupakan transformasi alami dari rumus rekursif, sehingga biasanya mudah diimplementasikan. Urutan pengisian tabel tidak penting dan hanya menghitung nilai dari fungsi jika hanya diperlukan. Ketika seluruh tabel memo pada akhirnya terisi, bisa saja lebih lambat karena adanya *overhead* pemanggilan fungsi.

Bottom-up

Pada *bottom-up*, penyelesaian masalah dimulai dari kasus yang kecil. Ketika merumuskan rumus rekursif, kita mengetahui jawaban kasus yang paling kecil, yaitu *base case*. Informasi ini digunakan untuk menyelesaikan kasus yang lebih besar. *Bottom-up* biasanya dianalogikan dengan pengisian "tabel DP". Berbeda dengan *top-down*, *bottom-up* ini dilakukan secara iteratif.

Kali ini kita akan mencoba menyelesaikan *Coin Change* secara *bottom-up*. Pada *bottom-up*, kita hitung semua nilai $f(x)$ untuk semua nilai x dari 0 sampai N secara menaik. Nilai dari $f(x)$ disimpan dalam *array* $f[x]$.

Dengan mudah Anda dapat memperhatikan bahwa kompleksitas solusi dengan *bottom-up* pada Algoritma 29 adalah $O(NM)$. Kompleksitas ini sama seperti dengan cara *top-down*. Kenyataannya, sebenarnya keduanya merupakan algoritma yang sama, hanya berbeda di arah pencarian jawaban.

Pengisian Tabel

Cara *bottom-up* yang dijelaskan sebelumnya terkesan seperti "mengisi tabel". Pada bagian ini, kita akan coba simulasikan *bottom-up* untuk menghitung $f(12)$ dengan nominal

Algoritma 29 Penyelesaian *Coin Change* secara *bottom-up*.

```

1: function SOLVE()
2:    $f[0] \leftarrow 0$ 

3:   for  $x \leftarrow 1, N$  do
4:      $best \leftarrow \infty$ 
5:     for  $k \leftarrow 1, M$  do
6:       if  $(a_k \leq x)$  then
7:          $best \leftarrow \min(best, f[x - a_k] + 1)$ 
8:       end if
9:     end for
10:     $f[x] \leftarrow best$ 
11:  end for

12:  return  $f[N]$ 
13: end function

```

Tabel 7.1: Tahap 1: Siapkan tabel untuk menyimpan nilai f . Mula-mula tabel tersebut kosong.

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$													

koin 1, 6, dan 10 rupiah. Mula-mula, kita memiliki tabel $f(x)$ untuk nilai x dari 0 sampai 12, seperti pada Tabel 7.1.

Berikutnya, kita akan mengisi $f(1)$. Satu-satunya pilihan adalah menukarkan dengan koin 1, karena kita tidak bisa menggunakan koin 6 atau 10. Maka $f(1) = f(1 - 1) + 1 = f(0) + 1$. Dengan demikian, nilai tabel akan menjadi seperti pada Tabel 7.3.

Hal serupa terjadi ketika kita mengisi $f(2), f(3), f(4)$, dan $f(5)$. Satu-satunya pilihan adalah menukarkan dengan koin 1, karena kita tidak bisa menggunakan koin 6 atau 10. Sehingga, $f(2) = f(1) + 1, f(3) = f(2) + 1$, dan seterusnya. Dengan demikian, nilai tabel akan menjadi seperti pada Tabel 7.4.

Langkah berikutnya adalah mengisi $f(6)$. Kali ini, terdapat pilihan untuk menggunakan koin 1 atau 6 terlebih dahulu, sehingga:

Tabel 7.2: Tahap 2: Isi *base case* yaitu $f(0) = 0$.

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$	0												

Tabel 7.3: Tahap 3: Isi tabel setelah pengisian $f(1)$.

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$	0	1											

Tabel 7.4: Tahap 4: Isi tabel setelah pengisian $f(2)$ sampai $f(5)$.

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$	0	1	2	3	4	5							

$$\begin{aligned}
 f(6) &= \min(f(6-1)+1, f(6-6)+1) \\
 &= \min(f(5)+1, f(0)+1) \\
 &= \min(5+1, 0+1) \\
 &= \min(6, 1) \\
 &= 1
 \end{aligned}$$

Memang benar, untuk membayar 6 kita hanya membutuhkan 1 koin. Dengan demikian, nilai tabel menjadi seperti pada Tabel 7.5.

Lakukan hal serupa untuk $x = 7$ sampai $x = 9$. Saat pengisian $f(10)$, terdapat pilihan untuk menggunakan koin 1, 6, atau 10 terlebih dahulu, sehingga:

$$\begin{aligned}
 f(10) &= \min(f(10-1)+1, f(10-6)+1, f(10-10)+1) \\
 &= \min(f(9)+1, f(4)+1, f(0)+1) \\
 &= \min(4+1, 4+1, 0+1) \\
 &= \min(5, 5, 1) \\
 &= 1
 \end{aligned}$$

Kembali, dapat dicek bahwa untuk membayar 10 kita hanya membutuhkan 1 koin, yaitu langsung menggunakan koin 10 (tanpa 1 dan 6). Kita dapat lakukan hal serupa untuk pengisian $f(11)$ dan $f(12)$ sebagai berikut:

Tabel 7.5: Tahap 5: Isi tabel setelah pengisian $f(6)$.

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$	0	1	2	3	4	5	1						

Tabel 7.6: Tahap 6: Isi akhir tabel.

x	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(x)$	0	1	2	3	4	5	1	2	3	4	1	2	2

$$\begin{aligned}
 f(11) &= \min(f(11-1)+1, f(11-6)+1, f(11-10)+1) \\
 &= \min(f(10)+1, f(5)+1, f(1)+1) \\
 &= \min(1+1, 5+1, 1+1) \\
 &= \min(2, 6, 2) \\
 &= 2
 \end{aligned}$$

$$\begin{aligned}
 f(12) &= \min(f(12-1)+1, f(12-6)+1, f(12-10)+1) \\
 &= \min(f(11)+1, f(6)+1, f(2)+1) \\
 &= \min(2+1, 1+1, 2+1) \\
 &= \min(3, 2, 3) \\
 &= 2
 \end{aligned}$$

Maka, tabel akhirnya dapat dilihat pada Tabel 7.6.

Analisis *Bottom-Up*

Bottom-up tidak mengalami perlambatan dari *overhead* pemanggilan fungsi. *Bottom-up* juga memungkinkan penggunaan teknik DP lanjutan seperti *flying table*, kombinasi dengan struktur data *tree*, atau teknik-teknik lanjutan lainnya. Namun, Anda harus memikirkan urutan pengisian nilai tabel. Selain itu, semua tabel harus diisi nilainya walaupun tidak dibutuhkan akhirnya.

►► Sekilas Info ◀◀

Karena sifatnya yang memanfaatkan subpersoalan, soal yang dapat diselesaikan secara *greedy* biasanya dapat juga diselesaikan secara DP, meski biasanya lebih lambat dari segi kompleksitas waktu. Jika Anda tidak dapat memformulasikan solusi *greedy*, cobalah selesaikan secara DP! Selain itu, kebenaran solusi DP lebih mudah untuk dibuktikan. Sehingga solusi DP dapat digunakan untuk menguji kebenaran solusi *greedy* Anda.

Contoh-contoh DP Sederhana

Untuk membiasakan diri berpikir secara DP, kita akan selesaikan beberapa contoh persoalan DP sederhana.

Knapsack

Contoh Soal 7.2: Knapsack

Diberikan N buah barang, dinomori dari 1 sampai N . Barang ke- i memiliki harga v_i rupiah dan berat w_i gram. Kita memiliki tas yang berkapasitas G gram. Kita ingin memasukkan beberapa barang ke dalam tas, sedemikian sehingga jumlah berat dari barang-barang yang kita masukan tidak lebih dari kapasitas tas dan jumlah harganya sebanyak mungkin.

Contoh

Jika tersedia 5 buah barang sebagai berikut:

Nomor Barang	Harga	Berat
1	6	5
2	5	4
3	4	3
4	6	7
5	4	4

Jika kita memiliki kapasitas sebesar 14, solusi terbaik adalah mengambil barang ke 1, 2, dan 5. Total harga yang diperoleh adalah $6 + 5 + 4 = 15$.

Perhatikan bahwa untuk setiap barang, terdapat dua pilihan yang dapat kita lakukan: ambil atau tinggalkan. Jika barang tersebut diambil, kapasitas tas kita berkurang dan harga barang yang kita dapatkan bertambah. Jika barang tersebut tidak diambil, tidak terjadi perubahan. Dengan demikian, kita bisa memformulasikan sebuah fungsi $dp(i, c)$ sebagai jumlah harga maksimum yang mungkin diperoleh, jika kita hanya mempunyai barang ke-1 sampai ke- i dan sisa kapasitas tas kita adalah c gram. Untuk menghitung fungsi $dp(i, c)$, kita akan mencoba apakah kita akan memasukkan barang ke- i ke tas atau tidak.

Jika $i = 0$, maka berarti tidak ada barang yang tersedia. Ini berarti $dp(i, c) = 0$. Kasus ini menjadi *base case*. Selain kasus *base case*, untuk setiap i dan c , kita memiliki paling banyak dua pilihan:

- Kasus 1: Mengambil barang ke- i

Jika kita memasukkan barang ke- i ke tas, maka kita akan menyisakan barang ke-1 sampai ke- $(i - 1)$ dan sisa kapasitas tas menjadi $c - w_i$. Harga barang yang didapatkan pada kasus ini adalah $dp(i - 1, c - w_i)$ ditambah dengan harga yang kita peroleh pada barang ke- i , atau dapat dituliskan $dp(i, c) = dp(i - 1, c - w_i) + v_i$. Karena kita memiliki kapasitas tas c , kasus ini hanya boleh dipertimbangkan jika $c \geq w_i$.

- Kasus 2: Tidak mengambil barang ke- i

Jika kita tidak memasukkan barang ke- i ke tas, maka kita akan menyisakan barang ke-1 sampai ke- $(i - 1)$ dan sisa kapasitas tas masih tetap c . Harga barang didapatkan pada kasus ini adalah $dp(i - 1, c)$, tanpa tambahan apapun (kita tidak mengambil barang ke- i), atau dapat dituliskan $dp(i, c) = dp(i - 1, c)$.

Dari kedua pilihan keputusan tersebut, kita tertarik dengan pilihan yang menghasilkan

nilai terbesar. Dengan demikian $dp(i, c)$ dapat dirumuskan sebagai berikut:

$$dp(i, c) = \begin{cases} 0, & i = 0 \\ dp(i-1, c), & i > 0 \wedge c < w_i \\ \max(dp(i-1, c - w_i) + v_i, dp(i-1, c)), & i > 0 \wedge c \geq w_i \end{cases}$$

Analisis Kompleksitas

Terdapat $O(N)$ nilai berbeda untuk nilai i dan $O(G)$ nilai berbeda untuk nilai c pada $dp(i, c)$. Dibutuhkan $O(1)$ untuk menghitung $dp(i, c)$. Sehingga untuk menghitung seluruh nilai $dp(i, c)$ untuk seluruh i dan c dibutuhkan waktu $O(NG)$.

Kita implementasikan $dp(i, c)$ sebagai fungsi $SOLVE(i, c)$ pada Algoritma 30. Jawaban akhirinya ada pada $SOLVE(N, G)$.

Pengisian tabel DP secara *bottom-up* dapat dilihat pada Algoritma 31.

Algoritma 30 *Knapsack* secara *top-down*.

```

1: function SOLVE(i, c)
2:   if (i = 0) then
3:     return 0
4:   end if
5:   if computed[i][c] then
6:     return memo[i][c]
7:   end if

8:   best ← SOLVE(i - 1, c)
9:   if (c ≥ w[i]) then
10:    best ← max(best, SOLVE(i - 1, c - w[i]) + v[i])
11:   end if
12:   computed[i][c] ← true
13:   memo[i][c] ← best
14:   return best
15: end function

```

Optimisasi Memori pada *Bottom-Up*

Terdapat sifat khusus pada perhitungan DP *Knapsack* secara *bottom-up*. Perhatikan bahwa untuk menghitung nilai $dp(i, -)$, hanya dibutuhkan informasi $dp(i-1, -)$. Informasi $dp(0, -)$, $dp(1, -)$, $dp(2, -)$, ..., $dp(i-2, -)$ tidak lagi dibutuhkan. Dengan demikian, kita dapat menghemat memori dengan tidak lagi menyimpan informasi yang sudah tidak dibutuhkan.

Untuk mewujudkannya, kita dapat membuat tabel DP yang hanya berukuran $2 \times G$. Pada awalnya, baris ke-0 digunakan untuk menyimpan $dp(0, -)$, dan baris ke-1 untuk $dp(1, -)$. Untuk perhitungan $dp(2, -)$, kita tidak lagi membutuhkan $dp(0, -)$, sehingga baris menyimpan $dp(0, -)$ dapat ditimpa untuk menyimpan $dp(2, -)$. Demikian pula untuk $dp(3, -)$, yang dapat menimpa baris yang menyimpan $dp(1, -)$. Secara umum, $dp(x, -)$ akan disimpan

Algoritma 31 *Knapsack secara bottom-up.*

```
1: function SOLVE()
2:   for  $c \leftarrow 0, G$  do ▷ Base case
3:      $dp[0][c] \leftarrow 0$ 
4:   end for

5:   for  $i \leftarrow 1, N$  do ▷ Isi "tabel" dari kasus yang kecil ke besar
6:     for  $c \leftarrow 0, G$  do
7:        $best \leftarrow dp[i-1][c]$ 
8:       if  $(c \geq w[i])$  then
9:          $best \leftarrow \max(best, dp[i-1][c-w[i]] + v[i])$ 
10:      end if
11:       $dp[i][c] \leftarrow best$ 
12:    end for
13:  end for
14:  return  $dp[N][G]$ 
15: end function
```

Algoritma 32 *Knapsack secara bottom-up dengan optimisasi flying table.*

```
1: function SOLVE()
2:    $dp \leftarrow \text{new integer}[2][G+1]$ 
3:   for  $c \leftarrow 0, G$  do
4:      $dp[0][c] \leftarrow 0$ 
5:   end for

6:   for  $i \leftarrow 1, N$  do
7:      $iNow \leftarrow i \bmod 2$ 
8:      $iPrev \leftarrow 1 - iNow$  ▷ Bernilai 0 atau 1 yang merupakan kebalikan dari  $iNow$ 
9:     for  $c \leftarrow 0, G$  do
10:       $best \leftarrow dp[iPrev][c]$ 
11:      if  $(c \geq w[i])$  then
12:         $best \leftarrow \max(best, dp[iPrev][c-w[i]] + v[i])$ 
13:      end if
14:       $dp[iNow][c] \leftarrow best$ 
15:    end for
16:  end for
17:  return  $dp[N \bmod 2][G]$ 
18: end function
```

pada baris ke- $(x \bmod 2)$. Optimisasi seperti ini disebut dengan *flying table* atau *rolling array*, dan ditunjukkan pada Algoritma 32.

Dengan optimisasi ini, kebutuhan memorinya berkurang dari $O(NG)$ menjadi $O(G)$. Perlu diperhatikan bahwa *flying table* hanya mengurangi kompleksitas memori. Kompleksitas waktu untuk perhitungan DP tetaplah sama.

Lebih jauh lagi, kita dapat menghilangkan dimensi i pada DP secara sepenuhnya, sehingga cukup digunakan tabel 1 dimensi berukuran G . Perhatikan bahwa pengisian $dp(i, c)$ hanya bergantung pada dua nilai, yaitu $dp(i-1, c)$ dan $dp(i-1, c-w[i])$, atau dengan kata lain, nilai yang tepat berada "di atas" dan "di atas kiri" sel sebuah tabel DP. Observasi ini memungkinkan kita untuk mengisi suatu $dp(i, c)$ dengan c yang menurun dari G sampai dengan $w[i]$, dan langsung menimpa sel tabel $dp(i-1, c)$. Pengisian perlu dilakukan dengan c yang menurun (atau bila dilihat pada tabel DP, dari kanan ke kiri) supaya nilai yang disimpan pada kolom $c-w[i]$ masih merupakan nilai $dp(i-1, c-w[i])$, bukan $dp(i, c-w[i])$. Optimisasi lanjutan ini ditunjukkan pada Algoritma 33.

Mengoptimisasi memori DP dari $2 \times G$ menjadi G saja mungkin tidak signifikan. Penjelasan mengenai teknik *flying table* lebih lanjut ini hanya bertujuan untuk menunjukkan bahwa terdapat hal-hal menarik yang dapat dilakukan ketika DP diimplementasikan secara *bottom-up*.

Algoritma 33 *Knapsack* secara *bottom-up* dengan optimisasi *flying table* lebih jauh.

```

1: function SOLVE()
2:   dp ← new integer[G+1]
3:   for c ← 0, G do
4:     dp[c] ← 0
5:   end for

6:   for i ← 1, N do
7:     for c ← G down to w[i] do
8:       dp[c] ← max(dp[c], dp[c-w[i]] + v[i])
9:     end for
10:  end for
11:  return dp[G]
12: end function

```

Coin Change

Coin Change merupakan salah satu persoalan DP klasik. Persoalan DP *Coin Change* memiliki 2 variasi. Variasi pertama adalah dengan jumlah koin tak hingga. Kita sudah mempelajari variasi ini serta cara penyelesaiannya di subbab sebelumnya. Variasi kedua adalah jika tiap koin hanya berjumlah satu buah, seperti yang tertera pada deskripsi soal berikut.

Contoh Soal 7.3: *Coin Change*

Diberikan M jenis koin, masing-masing jenis bernilai a_1, a_2, \dots, a_M rupiah. Asumsikan

banyaknya koin untuk setiap nominal adalah **satu buah**. Tentukan banyaknya koin paling sedikit untuk membayar **tepat** sebesar N rupiah!

Contoh

Diberikan koin-koin $[1, 1, 1, 5, 5, 5, 7]$. Jika kita ingin menukar 15 rupiah, solusinya adalah dengan menggunakan tiga keping 5. Namun jika kita ingin menukar 20 rupiah, maka solusinya adalah dengan menggunakan koin-koin 1, 1, 1, 5, 5 dan 7.

Kita tidak dapat menerapkan solusi dengan asumsi jumlah koin tak berhingga yang sudah dijelaskan di subbab sebelumnya. Pada kasus ini, kita dapat menerapkan intuisi yang serupa dengan persoalan *Knapsack*. Definisikan fungsi $dp(i, c)$ sebagai koin minimum yang diperlukan untuk menukar c rupiah, jika kita hanya mempunyai koin ke-1 sampai ke- i . Jika $c = 0$, artinya kita ingin menukar 0 rupiah, sehingga jumlah koin minimum adalah 0. Sementara jika $i = 0$ dan $c > 0$, artinya tidak ada koin yang tersedia untuk menukar c , sehingga $dp(0, c) = \infty$.

Selain *base case* tersebut, kita memiliki paling banyak dua pilihan, mengambil koin ke- i , atau tidak mengambil koin ke- i . Jika kita mengambil koin ke- i , maka kita mendapatkan subpersoalan $dp(i - 1, c - a_i)$, sementara jika kita tidak mengambil koin ke- i , kita mendapatkan subpersoalan $dp(i - 1, c)$. Dari kedua kemungkinan tersebut, pilih yang memberikan nilai terkecil. Dengan demikian, $dp(i, c)$ dapat dirumuskan sebagai berikut:

$$dp(i, c) = \begin{cases} 0, & c = 0 \\ \infty, & c > 0 \wedge i = 0 \\ dp(i - 1, c), & i > 0 \wedge c < a_i \\ \min(dp(i - 1, c - a_i) + 1, dp(i - 1, c)), & i > 0 \wedge c \geq a_i \end{cases}$$

Analisis Kompleksitas

Terdapat $O(N)$ nilai berbeda untuk nilai i dan $O(G)$ nilai berbeda untuk nilai c pada $dp(i, c)$. Dibutuhkan $O(1)$ untuk menghitung $dp(i, c)$. Sehingga untuk menghitung seluruh nilai $dp(i, c)$ untuk seluruh i dan c dibutuhkan waktu $O(NG)$.

Kita implementasikan $dp(i, c)$ sebagai fungsi $SOLVE(i, c)$. Implementasi dapat dilihat pada Algoritma 34. Jawaban akhirnya ada pada $SOLVE(N, G)$.

Implementasinya sebagai DP *bottom-up* dapat diperhatikan di Algoritma 35.

Longest Common Subsequence

Contoh Soal 7.4: Longest Common Subsequence (LCS)

Diberikan 2 buah string A dan B . Panjang kedua string tidak harus sama. Berapa panjang string terpanjang yang merupakan *subsequence* dari A dan B ?

Subsequence dari sebuah string adalah string lain yang dapat dibentuk dengan menghapus beberapa (boleh nol) karakter dari string asli tanpa mengubah urutan-

Algoritma 34 *Coin Change* secara *top-down*.

```

1: function SOLVE( $i, c$ )
2:   if ( $c = 0$ ) then
3:     return 0
4:   end if
5:   if ( $i = 0$ ) then
6:     return  $\infty$ 
7:   end if
8:   if computed[ $i$ ][ $c$ ] then
9:     return memo[ $i$ ][ $c$ ]
10:  end if
11:   $best \leftarrow \text{SOLVE}(i - 1, c)$ 
12:  if ( $c \geq a[i]$ ) then
13:     $best \leftarrow \min(best, \text{SOLVE}(i - 1, c - a[i]) + 1)$ 
14:  end if
15:  computed[ $i$ ][ $c$ ]  $\leftarrow true$ 
16:  memo[ $i$ ][ $c$ ]  $\leftarrow best$ 
17:  return  $best$ 
18: end function

```

Algoritma 35 *Coin Change* secara *bottom-up*.

```

1: function SOLVE()
2:   for  $c \leftarrow 1, G$  do ▷ Base case 1
3:      $dp[0][c] \leftarrow \infty$ 
4:   end for
5:   for  $i \leftarrow 0, N$  do ▷ Base case 2
6:      $dp[i][0] \leftarrow 0$ 
7:   end for

8:   for  $i \leftarrow 1, N$  do ▷ Isi "tabel" dari kasus yang kecil ke besar
9:     for  $c \leftarrow 0, G$  do
10:       $best \leftarrow dp[i - 1][c]$ 
11:      if ( $c \geq a[i]$ ) then
12:         $best \leftarrow \min(best, dp[i - 1][c - a[i]] + 1)$ 
13:      end if
14:       $dp[i][c] \leftarrow best$ 
15:    end for
16:  end for

17:  return  $dp[N][G]$ 
18: end function

```

nya. Sebagai contoh, "gak", "oak", "gerak", dan "gerobak" adalah *subsequence* dari "gerobak", sementara "garuk", "ombak", "gerebek", dan "georbak" bukan.

Contoh

A = "ajaib"

B = "badai"

LCS dari A dan B adalah "aai", dengan panjang 3.

Kita bisa mendefinisikan $dp(i, j)$ sebagai panjang *subsequence* terpanjang dari i karakter pertama dari A dan j karakter pertama dari B. Jika karakter ke- i dari A, atau dinotasikan dengan A_i sama dengan B_j , maka kita bisa memasangkan karakter tersebut sebagai kandidat *subsequence*. Jika $A_i \neq B_j$, maka A_i tidak bisa dipasangkan dengan B_j . Dalam kasus ini, kita perlu mengecek subpersoalan untuk mencari LCS dari i karakter pertama pada A dan $j - 1$ karakter pertama pada B, atau $i - 1$ karakter pertama pada A dan j karakter pertama pada B, lalu dipilih yang menghasilkan LCS terpanjang. Dari sini kita juga bisa lihat bahwa jika i atau j adalah 0, maka $dp(i, j) = 0$. Ini adalah *base case* dari persoalan LCS.

Dengan demikian $dp(i, j)$ dapat dirumuskan sebagai berikut:

$$dp(i, j) = \begin{cases} 0, & i = 0 \vee j = 0 \\ dp(i-1, j-1) + 1, & A_i = B_j \\ \max(dp(i-1, j), dp(i, j-1)), & A_i \neq B_j \end{cases}$$

Analisis Kompleksitas

Jika panjang string A adalah M dan panjang string B adalah N, maka terdapat $O(M)$ nilai berbeda untuk nilai i dan $O(N)$ nilai berbeda untuk nilai j pada $dp(i, j)$. Dibutuhkan $O(1)$ untuk menghitung $dp(i, j)$. Sehingga untuk menghitung seluruh nilai $dp(i, j)$ untuk seluruh i dan j dibutuhkan waktu $O(NM)$.

Kita implementasikan $dp(i, j)$ sebagai fungsi $SOLVE(i, j)$ pada Algoritma 36. Jawaban akhirnya ada pada $SOLVE(M, N)$. Implementasinya secara *bottom-up* dapat diperhatikan di Algoritma 37.

Memotong Kayu

Contoh Soal 7.5: Memotong kayu (Diadopsi dari UVa 10003 - Cutting Sticks¹¹)

Kita akan memotong sebuah batang kayu dengan panjang M meter pada N titik menjadi $N + 1$ bagian. Titik ke- i berada di L_i meter dari ujung kiri, dengan $1 \leq i \leq N$. Untuk memotong sebatang kayu menjadi dua, kita memerlukan usaha **sebesar panjang kayu yang sedang kita potong**. Cari urutan pemotongan sedemikian sehingga total usaha yang dibutuhkan minimum!

Contoh

Algoritma 36 LCS secara *top-down*.

```

1: function SOLVE( $i, j$ )
2:   if ( $i = 0 \vee j = 0$ ) then
3:     return 0
4:   end if
5:   if computed[ $i$ ][ $j$ ] then
6:     return memo[ $i$ ][ $j$ ]
7:   end if

8:   if ( $A[i] = B[j]$ ) then
9:      $best \leftarrow \text{SOLVE}(i - 1, j - 1) + 1$ 
10:  else
11:     $best \leftarrow \max(\text{SOLVE}(i - 1, j), \text{SOLVE}(i, j - 1))$ 
12:  end if
13:  computed[ $i$ ][ $j$ ]  $\leftarrow$  true
14:  memo[ $i$ ][ $j$ ]  $\leftarrow$   $best$ 
15:  return  $best$ 
16: end function

```

Algoritma 37 LCS secara *bottom-up*.

```

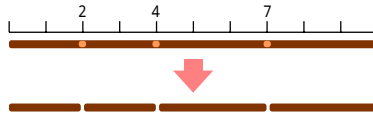
1: function SOLVE()
2:   for  $i \leftarrow 0, M$  do
3:      $dp[i][0] \leftarrow 0$ 
4:   end for
5:   for  $j \leftarrow 0, N$  do
6:      $dp[0][j] \leftarrow 0$ 
7:   end for
8:   for  $i \leftarrow 1, M$  do
9:     for  $j \leftarrow 1, N$  do
10:      if ( $A[i] = B[j]$ ) then
11:         $dp[i][j] \leftarrow dp[i - 1][j - 1] + 1$ 
12:      else
13:         $dp[i][j] \leftarrow \max(dp[i - 1][j], dp[i][j - 1])$ 
14:      end if
15:    end for
16:  end for
17:  return  $dp[M][N]$ 
18: end function

```

▷ Base case

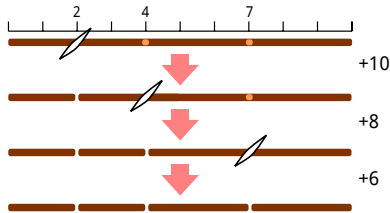
▷ Isi "tabel" dari kasus yang kecil ke besar

Contoh, terdapat sebuah kayu dengan panjang 10 meter dan terdapat 3 titik potongan pada 2 meter, 4 meter, dan 7 meter dari ujung kiri.



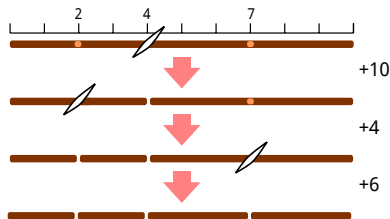
Gambar 7.2: Ilustrasi pemotongan kayu.

Kita bisa memotong pada titik 2, titik 4, lalu titik 7 dan memerlukan usaha $10 + 8 + 6 = 24$.



Gambar 7.3: Langkah pemotongan kayu.

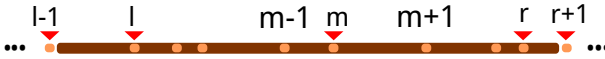
Cara lain adalah memotong pada titik 4, titik 2, lalu titik 7 dan memerlukan usaha $10 + 4 + 6 = 20$.



Gambar 7.4: Langkah pemotongan kayu yang lebih optimal.

Perhatikan bahwa untuk pemotongan pertama, terdapat N pilihan lokasi pemotongan. Jika kita memotong di posisi L_m , maka didapatkan dua batang, yang mana batang pertama perlu dipotong di titik L_1, L_2, \dots, L_{m-1} dan batang kedua di $L_{m+1}, L_{m+2}, \dots, L_N$. Dari sini kita mendapatkan subpersoalan yang serupa. Pemotongan bisa dilanjutkan secara rekursif,

dan kita pilih posisi pemotongan yang ke depannya membutuhkan usaha terkecil.



Gambar 7.5: Penomoran posisi pemotongan kayu.

Definisikan sebuah fungsi $dp(\ell, r)$ sebagai jumlah usaha minimum yang mungkin diperoleh, jika kita hanya perlu memotong di $L_\ell, L_{\ell+1}, \dots, L_r$. Untuk kemudahan, asumsikan pula $L_0 = 0$ dan $L_{N+1} = M$. Untuk menghitung $dp(\ell, r)$ kita dapat mencoba titik mana yang kita potong pertama kali. Jika kita memotong di L_m ($\ell \leq m \leq r$), maka kita akan mendapatkan dua potongan.

Dengan demikian, total usaha yang dibutuhkan jika kita melakukan pemotongan di L_m adalah jumlah dari:

- Total usaha minimum dari potongan pertama, yaitu $dp(\ell, m-1)$.
- Total usaha minimum dari potongan kedua, yaitu $dp(m+1, r)$.
- Usaha untuk pemotongan ini, yaitu $L_{r+1} - L_{\ell-1}$.

Ketika $\ell > r$, artinya sudah tidak ada pemotongan yang perlu dilakukan. Usaha yang dibutuhkan untuk kasus ini adalah 0, atau $dp(\ell, r) = 0$.

Dengan demikian, kita dapat merumuskan nilai DP sebagai berikut:

$$dp(\ell, r) = \begin{cases} 0, & \ell > r \\ \min_{\ell \leq m \leq r} dp(\ell, m-1) + dp(m+1, r) + (L_{r+1} - L_{\ell-1}), & \ell \leq r \end{cases}$$

Analisis Kompleksitas

Terdapat $O(N)$ nilai berbeda untuk nilai ℓ dan $O(N)$ nilai berbeda untuk nilai r pada $dp(\ell, r)$. Dibutuhkan iterasi sebanyak $O(N)$ untuk menghitung $dp(\ell, r)$. Sehingga untuk menghitung seluruh nilai $dp(\ell, r)$ untuk seluruh ℓ dan r dibutuhkan waktu $O(N^3)$.

Kita implementasikan $dp(\ell, r)$ sebagai fungsi $\text{SOLVE}(\ell, r)$ di Algoritma 38. Jawaban akhirnya ada pada $\text{SOLVE}(1, N)$. Implementasi secara *bottom-up* dapat diperhatikan di Algoritma 39.

Perhatikan bahwa implementasi metode *bottom-up* lebih rumit dibandingkan dengan implementasi *top-down*. Hal ini dikarenakan pengisian "tabel" dilakukan secara "tidak biasa". Pada kasus ini, urutan pengisian adalah

- $dp[1][1], dp[2][2], \dots, dp[N][N]$,
- lalu $dp[1][2], dp[2][3], \dots, dp[N-1][N]$,
- lalu $dp[1][3], dp[2][4], \dots, dp[N-2][N]$,
- lalu $dp[1][4], dp[2][5], \dots, dp[N-3][N]$,
- dan seterusnya sampai $dp[1][N]$.

Ingat bahwa pengisian "tabel" harus dilakukan dari kasus yang kecil ke besar. Definisi "kasus kecil" pada masalah ini adalah kayu dengan titik-titik pemotongan yang lebih sedikit. Dari contoh ini kita mempelajari bahwa urutan pengisian "tabel" pada DP *bottom-up* tidak

Algoritma 38 Solusi Memotong Kayu menggunakan *top-down*.

```
1: function SOLVE( $\ell, r$ )
2:   if ( $\ell > r$ ) then
3:     return 0
4:   end if
5:   if computed[ $\ell$ ][ $r$ ] then
6:     return memo[ $\ell$ ][ $r$ ]
7:   end if
8:    $best \leftarrow \infty$ 
9:    $cost \leftarrow L[r+1] - L[\ell-1]$ 
10:  for  $m \leftarrow \ell, r$  do
11:     $best \leftarrow \min(best, SOLVE(\ell, m-1) + SOLVE(m+1, r) + cost)$ 
12:  end for
13:  computed[ $\ell$ ][ $r$ ]  $\leftarrow true$ 
14:  memo[ $\ell$ ][ $r$ ]  $\leftarrow best$ 
15:  return  $best$ 
16: end function
```

Algoritma 39 Solusi Memotong Kayu menggunakan *bottom-up*.

```
1: function SOLVE()
2:    $\triangleright$  Base case
3:   for  $\ell \leftarrow 0, N+1$  do
4:     for  $r \leftarrow 0, \ell-1$  do
5:        $dp[\ell][r] \leftarrow 0$ 
6:     end for
7:   end for

8:    $\triangleright$  Isi "tabel" mulai dari kasus yang kecil
9:   for  $gap \leftarrow 0, N-1$  do
10:    for  $\ell \leftarrow 1, N-gap$  do
11:       $r \leftarrow \ell + gap$ 
12:       $best \leftarrow \infty$ 
13:       $cost \leftarrow L[r+1] - L[\ell-1]$ 
14:      for  $m \leftarrow \ell, r$  do
15:         $best \leftarrow \min(best, dp[\ell][m-1] + dp[m+1][r] + cost)$ 
16:      end for
17:       $dp[\ell][r] \leftarrow best$ 
18:    end for
19:  end for

20:  return  $dp[1][N]$ 
21: end function
```

selalu biasa. Jika urutan pengisiannya salah, maka hasil akhir yang didapatkan juga bisa jadi salah. Hal ini terjadi ketika kita hendak menyelesaikan kasus yang besar, tetapi hasil untuk kasus-kasus yang lebih kecil belum tersedia. Untuk mengetahui urutan pengisian "tabel", Anda perlu mengamati apa definisi "kasus kecil" pada masalah yang dihadapi.

8 Struktur Data Dasar

Struktur data merupakan tata cara untuk merepresentasikan dan menyimpan data, sehingga mendukung operasi terhadap data tersebut secara efisien. Struktur data paling sederhana yang telah kita pelajari adalah *array*. Pada bab ini, kita akan mempelajari struktur data *dynamic array*, *stack*, dan *queue*. Kita akan menggunakan struktur data ini pada pembelajaran Bab 9 mengenai graf.

Dynamic Array

Dynamic array merupakan *array* yang ukurannya dapat bertambah atau berkurang mengikuti banyaknya data yang sedang disimpan. Kemampuan menyesuaikan ukuran ini memberi keuntungan apabila kita tidak tahu berapa banyak elemen yang akan disimpan pada *array*, dan kita tidak mau membuat *array* yang ukurannya sebanyak elemen maksimum yang akan disimpan. Kebutuhan memori yang diperlukan kini disesuaikan dengan banyaknya elemen. Kita akan mempelajari *dynamic array* yang ukurannya hanya dapat bertambah.

Sama seperti *array*, kita dapat mengakses atau mengubah nilai pada suatu indeks yang ada pada *dynamic array* dalam $O(1)$. Pada *dynamic array*, terdapat operasi tambahan berupa pengisian suatu elemen sebagai elemen terakhir. Operasi ini biasa disebut dengan *push*.

Untuk membuat *dynamic array*, kita perlu membuat *array* biasa dengan ukuran *maxSize*. Misalnya *array* ini bernama *A*, dan indeksnya dimulai dari 0. Nilai *maxSize* dapat diinisialisasi dengan suatu nilai yang kecil, misalnya 1. Kita juga memerlukan sebuah variabel yang menyatakan banyaknya elemen yang telah disimpan, misalnya bernama *size*. Nilai awal dari *size* adalah 0.

Ketika dilakukan operasi *push*, terdapat dua kemungkinan kasus:

1. Dipenuhi $size < maxSize$
Pada kasus ini, simpan elemen terbaru pada $A[size]$, dan tambahkan nilai *size* dengan 1. Seluruh operasi ini memerlukan kompleksitas $O(1)$.
2. Dipenuhi $size = maxSize$
Kini *array A* tidak dapat menyimpan elemen yang baru. Untuk mengatasi hal ini, buat sebuah *array* baru dengan ukuran dua kali *maxSize*, lalu salin seluruh isi *A* ke *array* baru ini. Hapus *array A*, lalu ubah referensi *A* ke *array* yang baru. Kini *maxSize* menjadi berlipat ganda, dan kita dapat menyimpan elemen baru dengan melakukan langkah seperti kasus sebelumnya. Karena diperlukan penyalinan elemen ke *array* yang baru, kompleksitasnya kasus ini $O(K)$ jika terdapat *K* elemen yang telah disimpan.

►► Sekilas Info ◀◀

Pada implementasinya, *array A* perlu dibuat menggunakan konsep *pointer* dan alokasi memori.

Dengan cara ini, *dynamic array* dapat memperbesar ukurannya sesuai dengan ba-

nyaknya elemen yang disimpan. Memori yang dibutuhkan paling banyak sebesar dua kali banyaknya elemen yang disimpan.

Perhatikan bahwa operasi *push* kadang-kadang memerlukan proses yang tidak $O(1)$. Hal ini tidak seburuk kelihatannya. Sebab ketika dirata-rata, kompleksitas operasi *push* dapat dianggap $O(1)$.

Untuk lebih memahaminya, misalkan kita akan menyimpan N data pada sebuah *dynamic array* kosong dengan melakukan N kali *push*. Dengan nilai awal $maxSize = 1$, diperlukan lipat ganda ukuran *array* pada operasi *push* yang ke-2, 4, 8, 16, dan seterusnya sampai lebih besar atau sama dengan N . Secara matematis, banyaknya operasi lipat ganda yang diperlukan adalah M , dengan $M = \lfloor \log_2 N \rfloor$. Operasi lipat ganda ini dilakukan pada *push* yang ke- $2^1, 2^2, 2^3, \dots$, dan 2^M . Untuk setiap lipat ganda pada *push* yang ke- x , diperlukan penyalinan data sebanyak $x/2$ elemen.

Banyaknya elemen yang disalin pada seluruh lipat ganda dapat dihitung dengan menjumlahkan:

$$\frac{2^1}{2} + \frac{2^2}{2} + \frac{2^3}{2} + \dots + \frac{2^M}{2}$$

Yang mana jumlahan ini dapat disederhanakan menjadi:

$$2^0 + 2^1 + 2^2 + \dots + 2^{M-1}$$

Dengan rumus deret aritmetika, hasil jumlahan dari deret tersebut adalah $2^M - 1$.

Perhatikan pula bahwa dipenuhi:

$$2^{\lfloor \log_2 N \rfloor} \leq N$$

Mengingat $M = \lfloor \log_2 N \rfloor$, dipastikan $2^M - 1$ tidak lebih dari $N - 1$.

Kini kita mengetahui bahwa total elemen yang disalin secara keseluruhan untuk *push* sebanyak N kali tidak lebih dari $N - 1$. Selain penyalinan elemen, dilakukan pula penyimpanan data pada setiap *push*. Misalkan setiap penyalinan elemen atau penyimpanan data pada *push* dianggap membutuhkan 1 operasi. Artinya, banyaknya operasi maksimum untuk penyalinan elemen ditambah penyimpanan data adalah $N + N - 1$ operasi. Ketika diambil rata-ratanya, setiap *push* membutuhkan $\frac{N+N-1}{N} \approx 2$ operasi. Dengan demikian, secara rata-rata *push* dapat dianggap $O(1)$.

►► Sekilas Info ◀◀

Analisis kompleksitas untuk menghitung kompleksitas *push* disebut dengan *amortized analysis*. Pada *amortized analysis*, kompleksitas suatu algoritma tidak semata-mata dinyatakan menurut kasus terburuk yang mungkin, tetapi diperhatikan keseluruhan operasi yang dilakukan.¹²

Kompleksitas dari operasi *push dynamic array* lebih tepat untuk dikatakan *amortized* $O(1)$.

Aplikasi *Dynamic Array*

Struktur data ini cocok digunakan ketika kita tidak tahu banyaknya elemen yang akan disimpan. Sebagai contoh, misalnya kita perlu menyimpan nama N orang yang dikelompokkan menurut berat badan dalam kilogram (kg). Asumsikan pula berat badan N orang ini berupa bilangan bulat pada rentang 1 kg sampai dengan K kg, dan belum diketahui berapa paling banyak orang yang memiliki berat sekian kg.

Untuk menyimpan data tersebut, kita dapat membuat *array nama* dengan K elemen, yang masing-masing elemennya berupa *dynamic array*. Untuk orang dengan berat x kg, simpan datanya pada *dynamic array nama*[x]. Dengan cara ini, kebutuhan memorinya adalah $O(N)$.

Bandingkan apabila kita membuat *array nama* dengan K elemen, yang masing-masing elemennya berupa *array*. Berhubung untuk kasus paling buruk terdapat N orang yang memiliki berat yang sama, maka kita perlu membuat masing-masing *array* sebesar N . Kebutuhan memorinya adalah $O(KN)$, yang jelas tidak efisien apabila K cukup besar.

Contoh ini merupakan salah satu penggunaan dari *dynamic array*. Seiring dengan pembelajaran dan latihan, Anda akan menemukan kasus-kasus ketika *dynamic array* diperlukan.

Implementasi *Dynamic Array*

Implementasi *dynamic array* dapat diwujudkan dengan sebuah *array*, variabel *size*, dan *maxSize*. Operasi untuk mengubah atau mengakses elemen dari suatu indeks dapat langsung dilakukan pada A . Tentu saja indeks yang perlu diakses harus kurang dari *maxSize*. Perhatikan Algoritma 40 untuk memahami operasi *push* pada *dynamic array*.

Algoritma 40 Implementasi *dynamic array*.

```

1: procedure INITIALIZEDYNAMICARRAY() ▷ Seluruh variabel yang diinisialisasi bersifat
   global
2:   size ← 0
3:   maxSize ← 1
4:    $A \leftarrow \text{new integer}[\text{maxSize}]$ 
5: end procedure
6: procedure PUSH(item)
7:   if size = maxSize then
8:     tempA ← new integer[ $2 \times \text{maxSize}$ ] ▷ Ciptakan array baru yang lebih besar
9:     for  $i \leftarrow 0, \text{maxSize} - 1$  do
10:       $\text{tempA}[i] \leftarrow A[i]$  ▷ Salin isi array A
11:   end for
12:   delete  $A$  ▷ Hapus array A dari alokasi memori
13:    $A \leftarrow \text{tempA}$  ▷ Ubah referensi  $A$  ke array yang baru
14:   maxSize ←  $2 \times \text{maxSize}$ 
15: end if
16:    $A[\text{size}] \leftarrow \text{item}$ 
17:   size ← size + 1
18: end procedure

```

Apabila Anda menggunakan bahasa C++, Anda dapat langsung menggunakan *library dynamic array* yang tersedia dengan nama **vector**. Pada Java, *dynamic array* disediakan berupa **class ArrayList**.

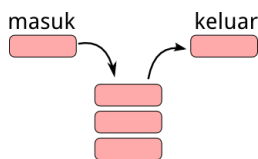
►► Sekilas Info ◀◀

Pada implementasinya, perbesaran ukuran *array* tidak harus menjadi 2x lipat. Misalnya, kita dapat menggunakan faktor perbesaran 1.5x lipat. Ternyata dengan faktor perbesaran 1.5x lipat, *dynamic array* bisa menjadi lebih efisien karena tidak banyak memori yang terbuang. Selain itu, ukuran *array* mula-mula juga tidak harus 1.

Stack

Stack dapat dimisalkan seperti tumpukan piring pada umumnya. Jika terdapat piring baru yang ingin dimasukkan, maka piring tersebut masuk dari paling atas. Jika sebuah piring akan diambil dari tumpukan, maka yang diambil juga piring yang paling atas.

Struktur data *stack* menyimpan informasi dalam bentuk seperti tumpukan. Informasi terbaru akan dimasukkan ke paling atas tumpukan. Hanya informasi paling atas yang bisa diakses/dihapus pada setiap waktunya. Oleh karena itu struktur data *stack* disebut memiliki sifat LIFO (*Last In First Out*).



Gambar 8.1: Ilustrasi penambahan dan penghapusan data pada *stack*

Stack memiliki operasi sebagai berikut:

- *push*, yaitu memasukkan elemen baru ke bagian atas tumpukan.
- *pop*, yaitu membuang elemen paling atas tumpukan.
- *top*, yaitu mengakses elemen paling atas tumpukan.
- *isEmpty*, yaitu memeriksa apakah tumpukan saat ini kosong.

Aplikasi *Stack*

Stack digunakan dalam berbagai kasus, salah satunya eksekusi fungsi pada sejumlah bahasa pemrograman. Biasanya, eksekusi fungsi terutama fungsi rekursif menggunakan struktur data *stack*. Pemanggilan fungsi rekursif yang menambah kedalaman berarti melakukan "*push*" pada *stack* eksekusi fungsi. Fungsi yang dieksekusi adalah fungsi di paling atas *stack*. Setelah fungsi di paling atas selesai, dilakukan "*pop*" dan eksekusi fungsi dilanjutkan ke fungsi yang ada di paling atas *stack* berikutnya. Oleh sebab itu, ketika pemanggilan fungsi terlalu dalam, terjadi **stack overflow**.

Selain digunakan pada fungsi rekursi, *stack* juga digunakan pada kalkulator ekspresi matematika dalam notasi *postfix*. Notasi *postfix* adalah notasi penulisan ekspresi matematika dengan urutan operand, operand, dan operator. Contoh:

- "1 2 +" bermakna "1 + 2"
- "1 2 3 + -" bermakna "1 - (2 + 3)"
- "1 2 3 + - 4 ×" bermakna "(1 - (2 + 3)) × 4"

Notasi yang biasa kita gunakan adalah notasi *infix*, yaitu dengan urutan operand, operator, dan operand.

Jika kita diberikan sebuah ekspresi dalam notasi *postfix*, nilai akhirnya dapat dicari dengan skema kerja *stack* sebagai berikut:

- Pada awalnya, inisialisasi sebuah *stack* kosong.
- Proses ekspresi dari kiri ke kanan:
 1. Jika ditemukan operand, *push* ke dalam *stack*.
 2. Jika ditemukan operator, *pop* dua kali untuk mendapat dua operand teratas *stack*, hitung, lalu *push* kembali ke dalam *stack*.
- Satu-satunya nilai terakhir di dalam *stack* adalah hasil ekspresinya.

Contoh Soal 8.1: Menghitung ekspresi *postfix*

Anda diberikan sebagai berikut:

- 1 2 3 + - 4 ×
- 1 2 × 4 × 3 5 × -
- 5 6 + 1 2 + 4 2 × - ×

Carilah hasil dari setiap ekspresi yang diberikan!

Mari kita coba selesaikan ekspresi pertama dengan menggunakan *stack*.

1. Pada mulanya, kita memiliki *stack* kosong : [].
2. *Push* angka 1 ke dalam *stack*. Isi *stack*: [1].
3. *Push* angka 2 ke dalam *stack*. Isi *stack*: [1, 2].
4. *Push* angka 3 ke dalam *stack*. Isi *stack*: [1, 2, 3].
5. Ditemukan operator +:
 - *Pop* dua kali, didapat nilai 3 dan 2. Isi *stack*: [1].
 - Operasikan 2 + 3 = 5, lalu *push* hasilnya. Isi *stack*: [1, 5].
6. Ditemukan operator -:
 - *Pop* dua kali, didapat nilai 5 dan 1. Isi *stack*: [].
 - Operasikan 1 - 5 = -4, dan *push* hasilnya. Isi *stack*: [-4].
7. *Push* angka 4 ke dalam *stack*. Isi *stack*: [-4, 4].
8. Ditemukan operator ×:
 - *Pop* dua kali, didapat nilai 4 dan -4. Isi *stack*: [].
 - Operasikan -4 × 4, dan *push* hasilnya. Isi *stack*: [-16].
9. Karena rangkaian ekspresi *postfix* sudah selesai, kita lihat angka yang ada pada *stack*, yaitu -16. Dengan demikian, 1 2 3 + - 4 × = -16.

Anda dapat mencoba menyelesaikan ekspresi-ekspresi lainnya sebagai sarana berlatih.

Aplikasi lainnya dari *stack* adalah sebagai struktur data yang menampung data saat melakukan *depth-first search*. Kita akan mempelajari *depth-first search* pada Bab 9.

Implementasi *Stack*

Anda dapat mengimplementasikan *stack* menggunakan sebuah *array* dan variabel penunjuk. Variabel penunjuk ini menyatakan indeks *array* yang menjadi elemen paling atas *stack*, dan bergerak maju/mundur sesuai dengan perintah *push/pop*. Seluruh operasi dapat dilakukan dalam $O(1)$. Algoritma 41 ini merupakan *pseudocode* implementasi *stack* menggunakan *array*.

Algoritma 41 Implementasi *stack* menggunakan *array*.

```

1: stack ← new integer[maxSize]           ▷ Buat array global stack berukuran maxSize
2: procedure INITIALIZESTACK()
3:   topOfStack ← 0                          ▷ topOfStack juga merupakan variabel global
4: end procedure
5: procedure PUSH(item)
6:   topOfStack ← topOfStack + 1
7:   stack[topOfStack] ← item
8: end procedure

9: procedure POP()
10:  topOfStack ← topOfStack - 1
11: end procedure

12: function TOP()
13:  return stack[topOfStack]
14: end function

15: function ISEMPY()
16:  if topOfStack = 0 then
17:    return true
18:  else
19:    return false
20:  end if
21: end function

```

Saat mengimplementasikan kode di atas, pastikan nilai *maxSize* sama dengan maksimal operasi *push* yang mungkin dilakukan. Dalam pemrograman kompetitif, biasanya banyaknya operasi *push* yang mungkin dapat diperkirakan dari soal yang diberikan. Perhatikan juga bahwa pada operasi *pop*, kita tidak benar-benar menghapus elemennya, melainkan hanya variabel penunjuknya yang "dimundurkan".

Contoh Soal 8.2: Pencarian string

Anda diberikan sebuah string, misalnya `acaabcbcd`. Cari string `abc` dalam string tersebut. Jika ditemukan maka hapus string `abc` tersebut, lalu ulangi pencarian. Pencarian berakhir ketika tidak terdapat string `abc` lagi. Anda diminta untuk menentukan total penghapusan yang berhasil dilakukan.

Contoh

Pada string `acaabcbcd` terdapat sebuah string `abc`, dan hapus string tersebut menjadi `acabcd`. Lalu, ditemukan lagi string `abc` dan hapus menjadi `acd`. Karena tidak ditemukan lagi string `abc`, maka jawabannya adalah 2.

Soal tersebut dapat diselesaikan dengan cara berikut:

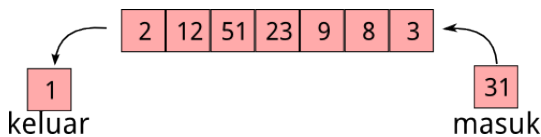
- Lakukan iterasi setiap karakter pada string tersebut.
- Untuk setiap karakter, *push* ke dalam *stack*.
- Cek 3 karakter teratas pada *stack*.
- Jika 3 karakter teratas merupakan `abc`, artinya terdapat 1 penghapusan. Lalu *pop* ketiga huruf tersebut dari *stack*.

Pada soal ini, Anda harus dapat memodifikasi struktur data *stack* agar Anda dapat melakukan operasi *top* pada 3 elemen teratas. Kompleksitas total adalah $O(N)$, dengan N merupakan panjang string.

Queue

Apakah anda pernah melihat antrian pembelian? Struktur data *queue* mirip dengan analogi antrian tersebut. Saat seorang ingin masuk ke antrian, maka orang tersebut harus mengantre dari belakang. Sementara itu, orang yang dilayani terlebih dahulu adalah orang yang paling depan.

Struktur data *queue* menyimpan informasi dalam bentuk antrian. Informasi yang baru dimasukkan ke paling belakang antrian. Hanya informasi paling depan yang bisa diakses/dihapus pada setiap waktunya. Oleh karena itu struktur data *queue* disebut memiliki sifat FIFO (*First In First Out*).



Gambar 8.2: Ilustrasi penambahan dan penghapusan data pada *queue*.

Queue memiliki beberapa operasi yang dapat dilakukan:

- *push*, yaitu memasukkan elemen baru ke bagian akhir antrian.
- *pop*, yaitu mengeluarkan elemen paling depan antrian.
- *front*, yaitu mengakses elemen yang paling depan antrian.

- *isEmpty*, yaitu memeriksa apakah antrian saat ini kosong.

Aplikasi *Queue*

Pada komputer *queue* digunakan untuk berbagai hal yang memerlukan antrian. Misalnya antrian berkas-berkas yang akan diunduh dari internet untuk ditampilkan pada *browser* Anda. *Queue* akan sering kita gunakan ketika sudah memasuki materi graf, tepatnya ketika melakukan *breadth-first search*.

Implementasi *Queue*

Anda dapat mengimplementasikan *queue* menggunakan sebuah *array* dan dua variabel penunjuk. Variabel penunjuk ini menyatakan indeks *array* yang menjadi elemen paling depan dan belakang *queue*. Kedua variabel penunjuk ini selalu bergerak maju. Seluruh operasi dapat dilakukan dalam $O(1)$. Gambar 42 ini merupakan *pseudocode* implementasi *queue* menggunakan *array*.

Algoritma 42 Implementasi *queue* menggunakan *array*.

```

1: queue ← new integer[maxSize]           ▷ Buat array global queue berukuran maxSize
2: procedure INITIALIZEQUEUE()
3:   head ← 0
4:   tail ← -1                               ▷ head dan tail juga merupakan variabel global
5: end procedure

6: procedure PUSH(item)
7:   tail ← tail + 1
8:   queue[tail] ← item
9: end procedure

10: procedure POP()
11:  head ← head + 1
12: end procedure

13: function FRONT()
14:  return queue[head]
15: end function

16: function ISEMPTY()
17:  if head > tail then
18:    return true
19:  else
20:    return false
21:  end if
22: end function

```

Kelemahan dari implementasi ini adalah beberapa elemen di bagian awal *array* tidak digunakan kembali. Misalnya telah dilakukan 15 kali *push* dan 11 kali *pop*, maka sebanyak

11 elemen pertama pada *array* tidak akan digunakan kembali. Ini adalah pemborosan, karena aslinya hanya terdapat 4 elemen di dalam *queue*. Meskipun demikian, dalam dunia kompetisi hal ini masih dapat diterima. Pastikan nilai *maxSize* sama dengan maksimal operasi *push* yang mungkin dilakukan.

9 Perkenalan Graf

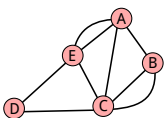
Graf merupakan konsep matematika diskret untuk merepresentasikan objek-objek yang saling berhubungan. Objek ini bisa saja berupa kota-kota yang terhubung dengan ruas jalan, individu-individu manusia yang saling mengenal, atau hubungan mangsa-pemangsa antar hewan di alam. Melalui pemahaman tentang konsep graf, kita dapat mendekati permasalahan yang melibatkan hubungan antar objek dengan lebih baik. Kita juga akan mempelajari representasi graf efisien dalam program sesuai dengan permasalahan yang ada.

Konsep Graf

Graf adalah struktur yang terdiri dari:

- **Node/vertex**, yaitu objek yang merepresentasikan suatu konsep.
- **Edge**, yaitu penghubung antar dua *node*.

Sebagai contoh, misalkan kita ingin memodelkan sejumlah kota yang dihubungkan oleh beberapa ruas jalan dengan graf. Kita dapat menganggap setiap kota sebagai *node*, dan ruas jalan sebagai *edge*.



Gambar 9.1: *Node* direpresentasikan dengan bentuk lingkaran dan *edge* direpresentasikan dengan bentuk garis.

Gambar 9.1 menggambarkan:

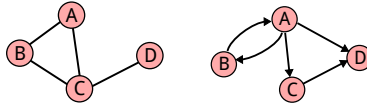
- Terdapat lima kota, yaitu A, B, C, D, dan E.
- Terdapat sembilan ruas jalan, yaitu:
 1. Antara kota A dengan kota B.
 2. Antara kota A dengan kota C.
 3. Antara kota A dengan kota E, sebanyak dua ruas.
 4. Antara kota B dengan kota C, sebanyak dua ruas.
 5. Antara kota C dengan kota D.
 6. Antara kota C dengan kota E.
 7. Antara kota D dengan kota E.

Perhatikan bahwa boleh saja terdapat beberapa *edge* yang menghubungkan dua *node* yang sama.

Jenis Graf

Berdasarkan hubungan antar *node*, graf dapat dibedakan menjadi:

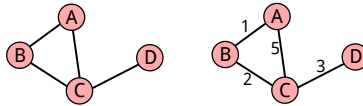
- **Graf tak berarah:** *edge* dari A ke B dapat ditelusuri dari A ke B dan B ke A.
- **Graf berarah:** *edge* dari A ke B hanya dapat ditelusuri dari A ke B.



Gambar 9.2: Graf tak berarah dan graf berarah.

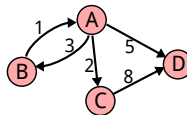
Sementara berdasarkan bobot dari *edge*, graf dapat dibedakan menjadi:

- **Graf tak berbobot,** yaitu graf dengan *edge* yang bobotnya seragam dan hanya bermakna terdapat hubungan antar *node*.
- **Graf berbobot,** yaitu graf dengan *edge* yang dapat memiliki bobot berbeda-beda. Bobot pada *edge* ini bisa jadi berupa biaya, jarak, atau waktu yang harus ditempuh jika menggunakan *edge* tersebut.



Gambar 9.3: Graf tak berbobot dan graf berbobot.

Tentu saja, suatu graf dapat memiliki kombinasi dari sifat-sifat tersebut. Misalnya graf yang berbobot dan berarah.

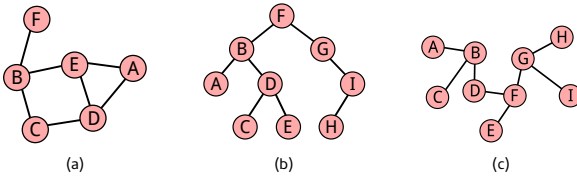


Gambar 9.4: Graf yang berbobot dan berarah.

Beberapa graf juga memiliki suatu karakteristik khusus. Contohnya adalah **tree**, **directed acyclic graph**, atau **bipartite graph**. Kali ini kita akan menyinggung *tree* dan *directed acyclic graph*.

Tree

Tree merupakan bentuk khusus dari graf. Seluruh *node* pada *tree* terhubung (tidak ada *node* yang tidak dapat dikunjungi dari *node* lain) dan tidak terdapat **cycle**. *Cycle* merupakan sekumpulan K *node* unik $[n_0, n_1, \dots, n_{K-1}]$, yang mana $K > 1$, dan setiap elemen n_i memiliki tetangga ke $n_{(i+1) \bmod K}$. Banyaknya *edge* dalam sebuah *tree* pasti $V - 1$, dengan V adalah banyaknya *node*.

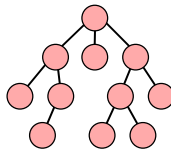


Gambar 9.5: Gambar (a) bukan *tree* karena memiliki *cycle*, salah satunya $[E,A,D]$, sedangkan gambar (b) dan (c) merupakan *tree*.

Tree sendiri bisa dikategorikan menjadi beberapa jenis, tergantung karakteristiknya.

Rooted Tree

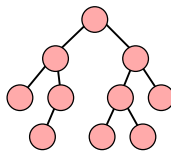
Suatu *tree* yang memiliki hierarki dan memiliki sebuah akar disebut sebagai *rooted tree*. Setiap *node* pada *rooted tree* memiliki kedalaman yang menyatakan jarak *node* tersebut ke *node root* (*node root* memiliki kedalaman 0).



Gambar 9.6: Contoh *rooted tree*. Bagian *root* ditunjukkan pada *node* yang berada di paling atas.

Binary Tree

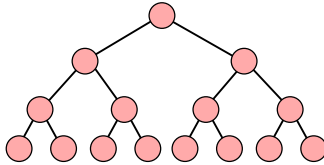
Suatu *rooted tree* yang setiap *node*-nya memiliki 0, 1, atau 2 anak disebut dengan *binary tree*.



Gambar 9.7: Contoh *binary tree*.

Perfect Binary Tree

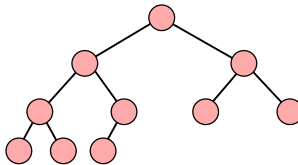
Suatu *binary tree* yang seluruh *node*-nya memiliki 2 anak, kecuali tingkat paling bawah yang tidak memiliki anak, disebut dengan *perfect binary Tree*. Bila banyaknya *node* adalah N , maka ketinggian dari pohon tersebut adalah $O(\log N)$.



Gambar 9.8: Contoh *perfect binary tree*.

Complete Binary Tree

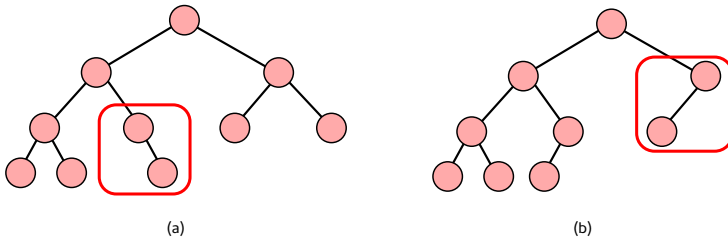
Complete binary tree adalah *binary tree* yang mana semua posisi *node* di setiap kedalamannya terisi, kecuali mungkin pada kedalaman terbawah, dan posisi *node*-nya terisi dengan urutan kiri ke kanan. Sama seperti *perfect binary Tree*, bila banyaknya *node* adalah N , maka ketinggiannya adalah $O(\log N)$. Contoh dari *complete binary tree* dapat dilihat pada gambar 9.9, sementara kedua *tree* pada gambar 9.10 bukanlah termasuk *complete binary tree*.



Gambar 9.9: Contoh *complete binary tree*.

►► Sekilas Info ◀◀

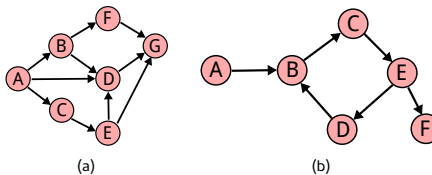
Complete binary tree dimanfaatkan pada struktur data *binary heap*. Struktur data ini akan kita pelajari pada bab Struktur Data NonLinear.



Gambar 9.10: Contoh *tree* yang bukan *complete binary tree*. *Tree* (a) bukanlah *complete binary tree* sebab elemen di tingkat paling bawah tidak berisi dari kiri ke kanan (terdapat lubang). *Tree* (b) juga bukan *complete binary tree*, sebab terdapat *node* tanpa 2 anak pada tingkat bukan paling bawah.

Directed Acyclic Graph

Directed acyclic graph (DAG) merupakan bentuk khusus dari *directed graf*, yang tepatnya bersifat tidak memiliki **cycle**. Berbeda dengan *tree* yang mana setiap *node* harus dapat dikunjungi dari *node* lainnya, sifat tersebut tidak berlaku pada DAG.



Gambar 9.11: Gambar (a) merupakan DAG, sedangkan gambar (b) bukan karena memiliki *cycle*, yaitu $[B, C, E, D]$.

Representasi Graf pada Pemrograman

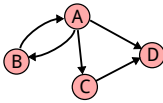
Dalam pemrograman, dibutuhkan sebuah struktur agar data mengenai graf dapat disimpan dan diolah secara efisien. Representasi yang akan kita pelajari adalah *adjacency matrix*, *adjacency list*, dan *edge list*. Masing-masing representasi ini memiliki keuntungan dan kerugiannya. Pemilihan representasi mana yang perlu digunakan bergantung dengan masalah yang dihadapi.

Adjacency Matrix

Kita akan menggunakan matriks dengan ukuran $V \times V$ dengan V merupakan banyaknya *node*. Misalnya matriks ini bernama *matrix*.

Pada graf tidak berbobot:

- Jika terdapat *edge* dari A ke B, maka $matrix[A][B] = 1$.
- Jika tidak ada, maka $matrix[A][B] = 0$.

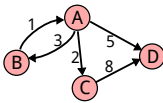


	A	B	C	D
A	0	1	1	1
B	1	0	0	0
C	0	0	0	1
D	0	0	0	0

Gambar 9.12: Graph tidak berbobot beserta *adjacency matrix*-nya.

Untuk graf berbobot, terdapat sedikit perbedaan:

- Jika terdapat *edge* dari A ke B dengan bobot w , maka $matrix[A][B] = w$.
- Jika tidak ada, maka dapat ditulis $matrix[A][B] = \infty$.



	A	B	C	D
A	∞	3	2	5
B	1	∞	∞	∞
C	∞	∞	∞	8
D	∞	∞	∞	∞

Gambar 9.13: Graph berbobot beserta *adjacency matrix*-nya.

Sifat-sifat *adjacency matrix* adalah:

- Pada graf tak berarah, *adjacency matrix* selalu simetris terhadap diagonalnya.
- Menambah atau menghapus *edge* dapat dilakukan dalam $O(1)$.
- Memeriksa apakah dua *node* terhubung dapat dilakukan dalam $O(1)$.
- Mendapatkan daftar tetangga dari suatu *node* dapat dilakukan iterasi $O(V)$, dengan V adalah banyaknya *node*.

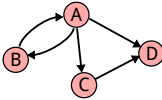
Meskipun mudah diimplementasikan, representasi ini cenderung boros dalam penggunaan memori. Memori yang dibutuhkan selalu $O(V^2)$, sehingga tidak dapat digunakan untuk graf dengan *node* mencapai ratusan ribu. Jika banyaknya *edge* jauh lebih sedikit dari $O(V^2)$, maka banyak memori yang terbuang.

Adjacency List

Alternatif lain untuk representasi graf adalah *adjacency list*.

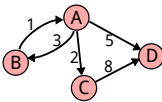
Untuk setiap *node*, buat sebuah penampung yang berisi keterangan mengenai tetangganya. Penampung ini berupa struktur data yang dapat mendukung operasi simpan, cari, dan hapus data. Pada pembahasan ini, kita akan menggunakan struktur data linier yang telah dipelajari sebelumnya seperti *array* atau *dynamic array*. Untuk graf tidak berbobot,

kita cukup menyimpan *node-node* tetangga untuk setiap *node*. Sementara untuk graf berbobot, kita dapat menyimpan *node-node* tetangga beserta bobotnya.



A	[B, C, D]
B	[A]
C	[D]
D	[]

Gambar 9.14: Graf tidak berbobot beserta *adjacency list*-nya.



A	[(B, 3), (C, 2), (D, 5)]
B	[(A, 1)]
C	[(D, 8)]
D	[]

Gambar 9.15: Graf berbobot beserta *adjacency list*-nya. Informasi yang disimpan untuk suatu tetangga adalah pasangan data yang menyatakan *node* dan bobotnya.

Untuk implementasi *adjacency list*, kita dapat menggunakan struktur data *array of dynamic array*. Tiap *dynamic array* berisi keterangan mengenai tetangga suatu *node*. Ukuran dari *array* merupakan V , yang mana V merupakan banyaknya *node*. Dengan menggunakan *dynamic array*, banyaknya memori yang digunakan untuk setiap *node* hanya sebatas banyak tetangganya. Secara keseluruhan jika graf memiliki E *edge*, maka total memori yang dibutuhkan adalah $O(E)$.

Jika diimplementasikan dengan *array of dynamic array*, maka sifat-sifat *adjacency list* adalah:

- Kompleksitas menambah *edge* adalah $O(1)$, dan menghapus adalah $O(K)$ dengan K adalah banyaknya tetangga dari *node* yang *edge*-nya dihapus.
- Memeriksa apakah dua *node* terhubung oleh *edge* juga dilakukan dalam $O(K)$.
- Demikian juga untuk mendapatkan daftar tetangga dari *node*, kompleksitasnya adalah $O(K)$. Perhatikan bahwa pencarian daftar tetangga ini adalah cara paling efisien yang mungkin.

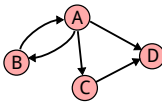
Edge List

Pada *edge list*, seluruh informasi *edge* disimpan pada sebuah penampung. Penampung ini berupa struktur data yang juga dapat mendukung operasi simpan, cari, dan hapus data. Untuk pembahasan kali ini, kita akan menggunakan struktur data *array* sebagai penampungnya.

Dengan menggunakan *array*, simpan informasi pasangan *node* yang terhubung oleh suatu *edge*. Pada graf yang berbobot, kita juga perlu menyimpan bobot dari *edge* yang bersangkutan. Jelas bahwa memori yang dibutuhkan adalah $O(E)$, dengan E adalah banyaknya *edge* pada keseluruhan graf.

Tabel 9.1: Tabel perbandingan operasi dan kompleksitas berbagai representasi graf, dengan catatan *adjacency list* diimplementasikan dengan *array of dynamic array* dan *edge list* diimplementasikan dengan *array*.

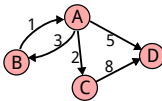
	<i>Adj.Matrix</i>	<i>Adj.List</i>	<i>Edge List</i>
Tambah <i>edge</i>	$O(1)$	$O(1)$	$O(1)$
Hapus <i>edge</i>	$O(1)$	$O(K)$	$O(E)$
Cek keterhubungan	$O(1)$	$O(K)$	$O(E)$
Daftar tetangga	$O(V)$	$O(K)$	$O(E)$
Kebutuhan memori	$O(V^2)$	$O(E)$	$O(E)$



$\langle A, B \rangle,$
 $\langle A, C \rangle,$
 $\langle A, D \rangle,$
 $\langle B, A \rangle,$
 $\langle C, D \rangle$

Gambar 9.16: Graf tidak berbobot beserta *edge list*-ya

Untuk graf berbobot, kita juga menyimpan bobot dari setiap *edge*.



$\langle A, B, 3 \rangle,$
 $\langle A, C, 2 \rangle,$
 $\langle A, D, 5 \rangle,$
 $\langle B, A, 1 \rangle,$
 $\langle C, D, 8 \rangle$

Gambar 9.17: Graf berbobot beserta *edge list*-ya.

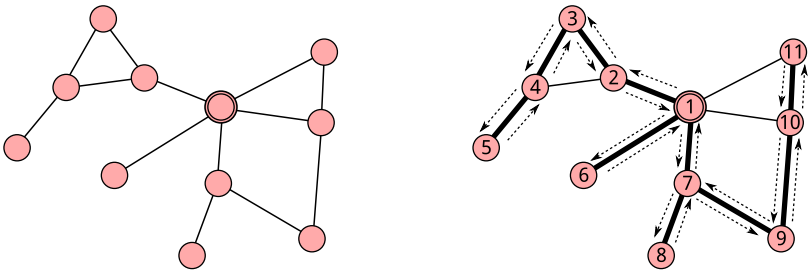
Jika diimplementasikan dengan *array*, maka sifat-sifat *edge list* adalah:

- Kompleksitas menambah *edge* adalah $O(1)$.
- Kompleksitas menghapus *edge* dan memeriksa keterhubungan sepasang *node* adalah $O(E)$.
- Demikian juga untuk mendapatkan daftar tetangga dari *node*, kompleksitasnya adalah $O(E)$.

Perbandingan Representasi Graf

Tabel 9.1 menunjukkan ringkasan operasi dan kompleksitas pada graf dengan V *node* dan E *edge* untuk berbagai representasi, dengan K adalah banyaknya *node* yang bertetangga dengan *node* yang sedang kita periksa.

Kompleksitas operasi-operasi *adjacency list* dan *edge list* yang ditunjukkan Tabel 9.1 bisa jadi lebih efisien, apabila kita menggunakan struktur data yang lebih efisien



Gambar 9.18: Contoh graf dan urutan penelusurannya dengan DFS. Angka pada *node* menunjukkan urutan *node* tersebut dikunjungi. *Node* 1 dikunjungi pertama, *node* 2 dikunjungi kedua, dan seterusnya). Anak panah menunjukkan pergerakan pengunjung *node*. *Edge* yang dicetak tebal menyatakan *edge* yang dilalui oleh penelusuran secara DFS.

dalam penampungan datanya. Sebagai contoh, struktur data *balanced binary search tree* yang digunakan sebagai penampungan dapat melakukan operasi simpan, cari, dan hapus dalam kompleksitas logaritmik. Struktur data lanjutan seperti *balanced binary search tree* tidak dibahas pada buku ini. Pembaca yang tertarik dapat mempelajarinya pada buku Introduction to Algorithm.¹³

Penjelajahan Graf

Penjelajahan graf merupakan penelusuran *node-node* pada suatu graf menurut suatu aturan tertentu. Terdapat 2 metode yang dapat digunakan, yaitu **DFS** dan **BFS**.

DFS: *Depth-First Search*

Penelusuran dilakukan dengan mengutamakan *node* yang lebih dalam terlebih dahulu (*depth-first*). Misalkan penelusuran secara DFS dimulai dari suatu *node*. Selama masih terdapat tetangga yang belum dikunjungi, kunjungi tetangga tersebut. Pada *node* tetangga ini, lakukan pula hal yang serupa. Ketika kita mencapai suatu *node* yang mana seluruh tetangganya sudah pernah dikunjungi, kembali ke *node* terakhir yang dikunjungi tepat sebelum *node* ini dikunjungi. Sebagai contoh, perhatikan Gambar 9.18. Penelusuran akan dilakukan menuju *node* yang paling dalam, lalu keluar, dan ke *node* lain yang paling dalam, dan seterusnya.

Dalam pemrograman, DFS biasa dilakukan dengan rekursi atau struktur data *stack*. Untuk keperluan implementasi, misalkan:

- Terdapat V *node* dan E *edge*.
- Setiap *node* dinomori dari 1 sampai dengan V .
- $adj(x)$ menyatakan himpunan tetangga dari *node* x .
- Terdapat *array visited*, yang mana $visited[x]$ bernilai *true* hanya jika x telah dikunjungi.

Contoh implementasi untuk DFS secara rekursif dapat diperhatikan pada Algoritma 4.3. Implementasi secara iteratif menggunakan struktur data *stack* ditunjukkan pada Algoritma

44.

Algoritma 43 Penelusuran graf dengan DFS secara rekursif.

```

1: procedure DFS(curNode)
2:   print "mengunjungi curNode"
3:   visited[curNode]  $\leftarrow$  true
4:   for adjNode  $\in$  adj(curNode) do
5:     if not visited[adjNode] then
6:       DFS(adjNode)
7:     end if
8:   end for
9: end procedure

```

Algoritma 44 Penelusuran graf dengan DFS secara iteratif menggunakan *stack*.

```

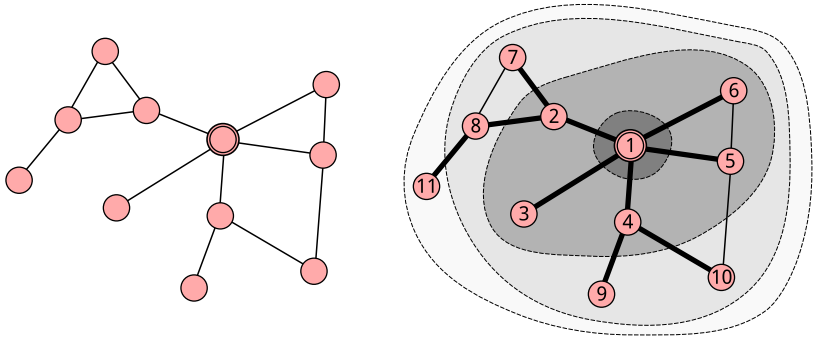
1: procedure DFS(initialNode)
2:   INITIALIZESTACK() ▷ Inisialisasi sebuah stack kosong.
3:   PUSH(initialNode)
4:   visited[initialNode]  $\leftarrow$  true
5:   while not ISEMPTYSTACK() do ▷ ISEMPTYSTACK: bernilai true jika stack kosong.
6:     curNode  $\leftarrow$  TOP()
7:     POP()
8:     print "mengunjungi curNode"
9:     for adjNode  $\in$  adj(curNode) do
10:      if not visited[adjNode] then
11:        PUSH(adjNode)
12:        visited[adjNode]  $\leftarrow$  true
13:      end if
14:    end for
15:   end while
16: end procedure

```

Perhatikan baris ke-12 Algoritma 44. Perubahan nilai *visited* ini akan menjamin suatu *node* masuk ke dalam *stack* paling banyak satu kali.

BFS: Breadth-First Search

Pada BFS, penelusuran *node* pada graf dilakukan lapis demi lapis. Semakin dekat suatu *node* dengan *node* awal, *node* tersebut akan dikunjungi terlebih dahulu.



Gambar 9.19: Sebuah graf dan urutan pengunjungan *node*-nya secara BFS. Angka pada *node* menunjukkan urutan *node* tersebut dikunjungi. *Node* 1 dikunjungi pertama, *node* 2 dikunjungi kedua, dan seterusnya). Daerah yang dibatasi garis putus-putus menyatakan daerah yang mana setiap *node* memiliki jarak yang sama kepada *node* yang dikunjungi pertama. *Edge* yang dicetak tebal menyatakan *edge* yang dilalui oleh penelusuran secara BFS.

Dalam pemrograman, BFS biasa diimplementasikan dengan bantuan struktur data *queue*. *Queue* ini menyimpan daftar *node* yang akan dikunjungi. Untuk setiap fase, kunjungi *node* yang terdapat di paling depan *queue*, dan daftarkan seluruh tetangganya yang belum dikunjungi pada bagian akhir *queue*. Lakukan hal ini sampai seluruh *node* terkunjungi. Algoritma 45 menunjukkan implementasi dari BFS.

Algoritma 45 Penelusuran graf dengan BFS.

```

1: procedure BFS(initialNode)
2:   INITIALIZEQUEUE() ▷ Inisialisasi sebuah queue kosong.
3:   PUSH(initialNode)
4:   visited[initialNode] ← true
5:   while not ISEMPYQUEUE() do ▷ ISEMPYQUEUE: bernilai true jika queue kosong.
6:     curNode ← FRONT()
7:     POP()
8:     print "mengunjungi curNode"
9:     for adjNode ∈ adj(curNode) do
10:      if not visited[adjNode] then
11:        PUSH(adjNode)
12:        visited[adjNode] ← true
13:      end if
14:    end for
15:  end while
16: end procedure

```

Analisis Kompleksitas DFS dan BFS

Baik DFS maupun BFS sama-sama mengunjungi setiap *node* tepat satu kali, dengan memanfaatkan seluruh *edge*. Kompleksitas dari kedua metode adalah:

- $O(V^2)$, jika digunakan *adjacency matrix*.
- $O(V + E)$, jika digunakan *adjacency list*.

Contoh Permasalahan Graf

Contoh 1

Contoh Soal 9.1: Perjalanan Tersingkat

Di suatu provinsi, terdapat N buah kota. Kota-kota dinomori dari 1 sampai dengan N . Terdapat E ruas jalan yang masing-masing menghubungkan dua kota berbeda.

Pak Dengklek tinggal di kota A. Suatu hari, beliau ingin pergi ke kota B. Karena sudah tua, Pak Dengklek ingin melewati sesedikit mungkin ruas jalan untuk sampai ke kota B. Tentukan berapa banyak ruas jalan tersedikit yang perlu beliau lewati untuk pergi dari kota A ke kota B!

Permasalahan ini dapat diselesaikan dengan BFS. Karena sifatnya yang menelusuri *node* lapis demi lapis, jika suatu *node* dikunjungi, maka jarak yang ditempuh dari awal sampai *node* tersebut pasti jarak terpendek. Hal ini selalu benar untuk segala graf tak berbobot; BFS selalu menemukan *shortest path* dari suatu *node* ke seluruh *node* lainnya. Algoritma 46 menunjukkan implementasi solusinya.

Contoh 2

Contoh Soal 9.2: Persebaran Informasi

Bebek-bebek Pak Dengklek merupakan hewan sosial. Apabila seekor bebek mengetahui suatu informasi, dia akan langsung memberitahu seluruh teman-temannya. Diketahui terdapat N ekor bebek di peternakan Pak Dengklek, yang dinomori dari 1 sampai dengan N .

Pak Dengklek mengetahui M hubungan pertemanan di antara bebek-bebeknya. Masing-masing hubungan ini dinyatakan dengan a_i dan b_i , yang artinya bebek a_i berteman dengan bebek b_i . Tentu saja, pertemanan ini berlaku secara dua arah.

Kini Pak Dengklek memiliki sebuah informasi yang ingin ia sampaikan kepada seluruh bebeknya. Berapa banyak bebek paling sedikit yang Pak Dengklek perlu beritahu tentang informasi ini, sedemikian sehingga seluruh bebek pada akhirnya akan mengetahui informasi tersebut?

Pada soal ini, kita dapat menyatakan hubungan pertemanan antar bebek sebagai graf tidak berarah dan tidak berbobot dengan N *node*, dan M *edge*. Bebek ke- i akan menjadi *node* i , dan hubungan pertemanan menghubungkan *node-node* tersebut sebagai *edge*.

Algoritma 46 Solusi Perjalanan Tersingkat menggunakan BFS, sambil mencatat kedalaman pengunjungan suatu *node* pada array *visitTime*.

```

1: procedure SHORTESTPATH(A,B)
2:   INITIALIZEQUEUE().
3:   visitTime  $\leftarrow$  new integer[V + 1]
4:   FILLARRAY(visitTime, -1)            $\triangleright$  Inisialisasi array visitTime dengan -1.

5:   PUSH(A)
6:   visitTime[A]  $\leftarrow$  0
7:   while not ISEMPYQUEUE() do
8:     curNode  $\leftarrow$  FRONT()
9:     POP()
10:    for adjNode  $\in$  adj(curNode) do
11:       $\triangleright$  Jika adjNode belum pernah dikunjungi...
12:      if visitTime[adjNode] = -1 then
13:        PUSH(adjNode)
14:        visitTime[adjNode]  $\leftarrow$  visitTime[curNode] + 1
15:      end if
16:    end for
17:  end while
18:  return visitTime[B]
19: end procedure

```

Apabila Pak Dengklek memberi tahu bebek ke-*i* suatu informasi, banyaknya bebek yang akhirnya akan mengetahui informasi tersebut sama saja dengan *node-node* yang dapat dikunjungi dari *node i*.

Kita dapat menyelesaikan soal ini dengan cara berikut:

1. Awalnya, seluruh *node* dianggap belum pernah dikunjungi.
2. Pilih salah satu *node* yang belum pernah dikunjungi, lalu lakukan penelusuran mulai dari *node* tersebut. Seluruh *node* yang dilalui pada penelusuran ini dianggap telah dikunjungi.
3. Apabila masih ada *node* yang belum pernah dikunjungi, lakukan langkah ke-2. Apabila seluruh *node* telah dikunjungi, laporkan banyaknya penelusuran graf yang dilakukan sebagai jawaban dari persoalan ini.

Urutan pengunjungan *node* tidaklah penting, sehingga penelusuran graf yang digunakan dapat berupa DFS atau BFS. Algoritma 47 menggunakan DFS yang diimplementasikan secara rekursif sebagai penelusuran grafnya.

Permasalahan semacam ini biasa disebut dengan menghitung banyaknya komponen. Sebuah komponen pada graf tidak berbobot merupakan sekumpulan *node* yang dapat saling mengunjungi satu sama lain, tanpa ada *node* lain pada graf yang dapat dimasukkan pada kumpulan *node* ini. Dalam sebuah graf, mungkin saja terdapat beberapa komponen.

Algoritma 47 Solusi Persebaran Informasi menggunakan DFS.

```

1: visited ← new boolean[MAX_N]
2: procedure DFS(curNode)
3:   visited[curNode] ← true
4:   for adjNode ∈ adj(curNode) do
5:     if not visited[adjNode] then
6:       DFS(adjNode)
7:     end if
8:   end for
9: end procedure

10: function SOLVE()
11:   FILLARRAY(visited, false)
12:   result ← 0
13:   for i ← 1, N do
14:     if not visited[i] then
15:       DFS(i)
16:       result ← result + 1
17:     end if
18:   end for
19:   return result
20: end function

```

▷ Fungsi utama yang mengembalikan jawaban soal

▶▶ Sekilas Info ◀◀

Algoritma penelusuran graf untuk mengunjungi seluruh *node* pada suatu komponen umumnya disebut dengan *flood fill*. Secara harfiah, *flood fill* berarti "pengisian banjir". Nama ini sesuai dengan jalannya algoritma yang seperti air banjir mengalir dari suatu *node* ke *node* lainnya yang bertetangga. Pada Algoritma 47, *flood fill* ditunjukkan pada baris ke-2 sampai dengan baris ke-9.

Contoh 3

Contoh Soal 9.3: Kado Iseng

Sebentar lagi Pak Ganesh berulang tahun. Sebagai teman sepermainan, Pak Dengklek hendak memberikan kado yang berupa tiket konser musisi kesukaannya. Namun, Pak Dengklek tidak ingin semata-mata memberikan tiket ini. Ia berencana untuk membungkusnya dalam kotak, yang akan dimasukkan ke dalam kotak lebih besar, dan dimasukkan lagi ke kotak yang lebih besar, dan seterusnya.

Di gudang, Pak Dengklek memiliki N kotak yang dapat menampung tiket konsernya, dinomori dari 1 sampai dengan N . Kotak ke- i memiliki ukuran panjang P_i , lebar L_i , dan tinggi T_i . Suatu kotak a dapat dimuat ke kotak b apabila dipenuhi $P_a < P_b$, $L_a < L_b$, dan $T_a < T_b$. Untuk alasan estetika, kotak tidak dapat dirotasi.

Pak Dengklek ingin agar hadiahnya dibungkus oleh kotak sebanyak mungkin. Bantulah Pak Dengklek menentukan banyaknya lapisan kotak terbesar yang mungkin!

Pada soal ini, tidak terdapat graf yang terlihat secara eksplisit. Namun, kita dapat memodelkan hubungan antar kotak sebagai graf. Misalkan setiap kotak merepresentasikan sebuah *node*. Apabila kotak a dapat dimuat di kotak b , berarti terdapat *edge* dari a ke b . Struktur graf ini akan membentuk sebuah *directed acyclic graph*.

Untuk penyelesaian masalahnya, kita dapat mulai dengan ide yang sederhana: pilih suatu kotak, masukkan ke salah satu kotak yang lebih besar, dan ulangi secara rekursif untuk kotak yang lebih besar. Kompleksitas solusi sederhana ini eksponensial terhadap banyaknya kotak, dan tidak efisien.

Solusi sederhana sebelumnya dapat dibuat lebih baik dengan prinsip *dynamic programming*. Definisikan fungsi $dp(a)$ yang menyatakan banyaknya lapisan kotak maksimum yang dapat dicapai untuk membungkus kotak a . Untuk mencari $dp(a)$, coba semua kemungkinan kotak b yang dapat membungkus a (atau dengan kata lain, seluruh tetangga dari a). Pilih yang memberikan nilai $dp(b)$ terbesar, lalu tambahkan dengan 1 karena banyaknya lapisan bertambah. *Base case* terjadi ketika tidak ada kotak yang dapat membungkus a , atau himpunan tetangga dari a berupa himpunan kosong. Untuk kasus tersebut $dp(a) = 0$.

$$dp(a) = \begin{cases} 0, & adj(a) = \{\} \\ 1 + \max_{b \in adj(a)} dp(b), & adj(a) \neq \{\} \end{cases}$$

Jawaban dari soal ini adalah $1 + dp(x)$, dengan x merupakan salah satu kotak yang memberikan nilai dp terbesar. Kompleksitas dari solusi ini adalah $O(E)$.

10 Struktur Data NonLinear

Struktur data tidak harus selalu menyimpan data pada sebuah rantai lurus. Terdapat pula struktur data yang menyimpan datanya pada struktur bercabang yang memungkinkan pelaksanaan suatu operasi secara efisien. Pada bagian ini, kita akan mempelajari struktur data *disjoint set* dan *heap*.

Disjoint Set

Disjoint set merupakan struktur data yang efisien untuk mengelompokkan elemen-elemen secara bertahap. *Disjoint set* mendukung operasi *join*, yaitu menggabungkan kelompok dari sepasang elemen. Operasi lain yang didukung adalah *check*, yaitu memeriksa apakah sepasang elemen berada di kelompok yang sama. Sebagai motivasi, perhatikan contoh soal berikut:

Contoh Soal 10.1: Membangun Jalan Antar Kandang

Pak Dengklek memiliki N kandang bebek yang tersebar di peternakannya, dinomori dari 1 sampai dengan N . Setiap pagi, Pak Dengklek akan mengunjungi satu per satu kandang dan memberi makan bebek-bebeknya.

Berhubung musim hujan telah tiba, tanah di peternakan Pak Dengklek menjadi becek dan perjalanan antar kandang ke kandang menjadi tidak nyaman. Untuk mengatasi masalah ini, Pak Dengklek berencana membangun jalan setapak yang menghubungkan kandang-kandangnya.

Dalam merencanakan pembangunan jalan ini, Pak Dengklek meminta bantuan Anda untuk melaksanakan sejumlah operasi. Setiap operasi dapat berbentuk salah satu dari:

- $join(a, b)$, artinya Pak Dengklek akan membangun jalan yang menghubungkan kandang a dengan kandang b . Kini seseorang dapat berpindah dari kandang a ke kandang b (dan sebaliknya) dengan jalan ini.
- $check(a, b)$, artinya Pak Dengklek ingin mengetahui apakah kandang a dan kandang b terhubung?

Pak Dengklek mendefinisikan dua kandang dikatakan terhubung apabila seseorang yang berada di kandang a dapat berpindah ke kandang b dengan menggunakan serangkaian jalan setapak (boleh nol).

Bantulah Pak Dengklek dalam melaksanakan operasi-operasi tersebut!

Kita dapat merepresentasikan kandang-kandang yang saling terhubung oleh serangkaian jalan dengan himpunan. Pada awalnya, setiap kandang terhubung hanya dengan dirinya sendiri, jadi setiap kandang membentuk himpunannya masing-masing. Ketika terdapat pembangunan jalan dari kandang a ke kandang b , seluruh kandang yang terhubung dengan kandang a kini terhubung dengan seluruh kandang yang terhubung dengan kandang b . Operasi ini dapat dipandang sebagai operasi penggabungan himpunan. Sementara itu,

pemeriksaan apakah sepasang kandang terhubung dapat dilakukan dengan memeriksa apakah kedua kandang berada pada himpunan yang sama.

Sebagai contoh, misalkan $N = 5$. Kondisi himpunan pada awalnya adalah $\{1\}, \{2\}, \{3\}, \{4\}, \{5\}$. Berikut contoh operasi-operasi secara berurutan dan perilaku yang diharapkan:

1. $\text{join}(1, 4)$, kini himpunan yang ada adalah $\{1, 4\}, \{2\}, \{3\}, \{5\}$.
2. $\text{check}(1, 2)$: laporkan bahwa 1 dan 2 berada di kelompok berbeda.
3. $\text{join}(1, 2)$, kini kelompok yang ada adalah $\{1, 2, 4\}, \{3\}, \{5\}$.
4. $\text{check}(1, 2)$: laporkan bahwa 1 dan 2 berada di kelompok yang sama.
5. $\text{join}(3, 5)$, kini kelompok yang ada adalah $\{1, 2, 4\}, \{3, 5\}$.
6. $\text{join}(2, 3)$, kini kelompok yang ada adalah $\{1, 2, 3, 4, 5\}$.
7. $\text{check}(1, 5)$: laporkan bahwa 1 dan 5 berada di kelompok yang sama.

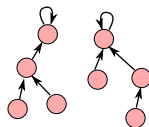
Solusi yang dapat digunakan adalah dengan merepresentasikan setiap kandang sebagai *node* seperti dalam konsep graf. Operasi *join* dapat dianggap sebagai penambahan *edge*. Sedangkan untuk setiap operasi *check*, diperlukan penjelajahan graf untuk memeriksa apakah kedua *node* terhubung. Representasi graf yang paling efisien dalam permasalahan ini adalah *adjacency list*.

Kompleksitas *check* adalah $O(N^2)$ apabila penjelajahan graf dilakukan secara naif dan seluruh kemungkinan *edge* dilalui. Solusi sederhana ini tidak efisien ketika banyak dilakukan operasi $\text{check}(a, b)$. Oleh karena itu diperlukan sebuah struktur data efisien yang dapat menyelesaikan permasalahan ini.

Konsep Disjoint Set

Ide dasar untuk menyelesaikan permasalahan tersebut adalah: untuk setiap kelompok yang ada, pilih suatu elemen sebagai perwakilan kelompok. Perlu diperhatikan untuk setiap elemen perlu mengetahui siapa perwakilan kelompoknya. Untuk memeriksa apakah dua elemen berada pada kelompok yang sama, periksa apakah perwakilan kelompok mereka sama. Untuk menggabungkan kelompok dari dua elemen, salah satu perwakilan kelompok elemen perlu diwakilkan oleh perwakilan kelompok elemen lainnya.

Setiap elemen perlu menyimpan *pointer* ke elemen yang merupakan perwakilannya. *Pointer* yang ditunjuk oleh suatu perwakilan kelompok adalah dirinya sendiri. Karena pada awalnya setiap elemen membentuk kelompoknya sendiri, maka awalnya setiap *pointer* ini menunjuk pada dirinya sendiri. Untuk mempermudah, mari kita sebut *pointer* ini sebagai *parent*.



Gambar 10.1: Ilustrasi dua kelompok elemen berikut *parent* dari masing-masing elemen pada *disjoint set*.

Inisialisasi *Disjoint Set*

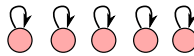
Inisialisasi *disjoint set* untuk N elemen dapat dilakukan dengan menyimpan *parent* dari masing-masing elemen berupa diri sendiri. Kita dapat menggunakan *array* untuk menyimpan *parent* setiap elemen. Pada Algoritma 48, *array par* menyimpan indeks elemen yang ditunjuk sebagai *parent* dari suatu elemen.

Algoritma 48 Inisialisasi dari *disjoint set*.

```

1: procedure INITIALIZE()
2:   for  $i \leftarrow 0, N - 1$  do                                ▷ Indeks elemen dimulai dari 0 (zero-based)
3:      $par[i] \leftarrow i$ 
4:   end for
5: end procedure

```

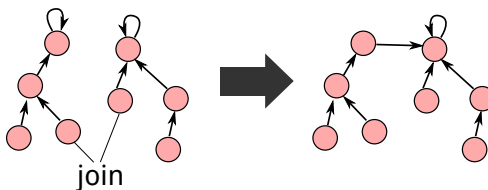


Gambar 10.2: Hasil dari inisialisasi *disjoint set*. Setiap elemen menunjuk ke dirinya sendiri.

Operasi *Join*

Ketika kelompok dua elemen perlu digabungkan, ubah *parent* dari salah satu perwakilan kelompok ke perwakilan kelompok lainnya.

Perhatikan bahwa yang perlu diubah adalah *parent* dari perwakilan kelompok suatu elemen, bukan *parent* elemen itu sendiri. Hal ini menjamin seluruh elemen pada kelompok tersebut kini menunjuk pada perwakilan dari kelompok lainnya, yang berarti seluruh elemen pada kedua kelompok kini menunjuk pada perwakilan kelompok yang sama.



Gambar 10.3: Contoh operasi *join* pada dua elemen dan hasilnya.

Secara sederhana, operasi *join* dapat dituliskan dalam Algoritma 49.

Operasi *findRepresentative*

Fungsi $FINDREPRESENTATIVE(x)$ mengembalikan elemen perwakilan dari kelompok tempat elemen x berada. Fungsi $FINDREPRESENTATIVE(x)$ dapat diimplementasikan secara

Algoritma 49 Operasi penggabungan kelompok pada *disjoint set*.

```

1: procedure JOIN( $a, b$ )
2:    $repA \leftarrow$  FINDREPRESENTATIVE( $a$ )
3:    $repB \leftarrow$  FINDREPRESENTATIVE( $b$ )
4:    $par[repA] = repB$ 
5: end procedure

```

rekursif, yaitu dengan menelusuri *parent* dari suatu elemen sampai ditemukan elemen yang memiliki *parent* berupa dirinya sendiri, seperti yang ditunjukkan pada Algoritma 50.

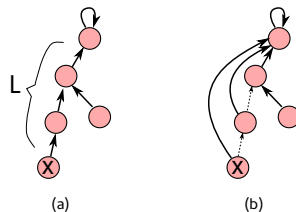
Algoritma 50 Pencarian perwakilan kelompok dari suatu elemen x .

```

1: function FINDREPRESENTATIVE( $x$ )
2:   if  $par[x] = x$  then
3:     return  $x$ 
4:   else
5:     return FINDREPRESENTATIVE( $par[x]$ )
6:   end if
7: end function

```

Fungsi FINDREPRESENTATIVE memiliki kekurangan yaitu kompleksitasnya sebesar $O(L)$, dengan L adalah panjangnya jalur dari elemen x sampai elemen perwakilan kelompoknya. Ketika L mendekati N , fungsi ini tidak efisien bila dipanggil berkali-kali. Oleh karena itu kita dapat menerapkan teknik *path compression*, yaitu mengubah nilai *parent* dari setiap elemen yang dilalui **langsung** ke elemen perwakilan kelompok. Hal ini menjamin untuk pemanggilan FINDREPRESENTATIVE berikutnya pada elemen yang bersangkutan bekerja secara lebih efisien.



Gambar 10.4: Gambar (a) menunjukkan contoh pemanggilan FINDREPRESENTATIVE pada elemen x , yang membutuhkan kompleksitas sebesar $O(L)$. Gambar (b) menunjukkan hasil *path compression* sesudah pemanggilan FINDREPRESENTATIVE.

Implementasi dari *path compression* cukup dilakukan dengan menambahkan pencatatan elemen perwakilan kelompok untuk setiap elemen yang dilalui.

Algoritma 51 Pencarian perwakilan kelompok dari suatu elemen x yang efisien dengan *path compression*.

```

1: function FINDREPRESENTATIVE( $x$ )
2:   if  $par[x] = x$  then
3:     return  $x$ 
4:   else
5:      $par[x] \leftarrow$  FINDREPRESENTATIVE( $par[x]$ )    ▷ Catat elemen representatifnya
6:     return  $par[x]$ 
7:   end if
8: end function

```

Operasi *check*

Operasi untuk memeriksa apakah a dan b berada pada kelompok yang sama dapat dilakukan dengan memeriksa apakah keduanya memiliki perwakilan kelompok yang sama. Hal ini ditunjukkan pada Algoritma 52.

Algoritma 52 Implementasi pemeriksaan apakah a dan b berada pada kelompok yang sama.

```

1: function CHECK( $a, b$ )
2:   return FINDREPRESENTATIVE( $a$ ) = FINDREPRESENTATIVE( $b$ )
3: end function

```

Analisis Kompleksitas

Apabila seluruh *parent* elemen sudah dikenakan *path compression*, maka setiap elemen langsung menunjuk ke elemen perwakilan kelompoknya. Artinya, kini fungsi FINDREPRESENTATIVE bekerja dalam $O(1)$. Kompleksitas satu kali pemanggilan FINDREPRESENTATIVE tidak dapat didefinisikan secara pasti. Perhitungan secara matematis membuktikan bahwa dengan *path compression*, kompleksitas untuk M operasi FINDREPRESENTATIVE dan $N - 1$ operasi JOIN, dengan $M \geq N$, bekerja dalam $O(M \log N)$.¹⁴

Heap

Heap merupakan struktur data yang umum dikenal pada ilmu komputer. Nama *heap* sendiri berasal dari Bahasa Inggris, yang berarti "gundukan". *Heap* merupakan struktur data yang mampu mencari nilai terbesar secara efisien dari sekumpulan elemen yang terus bertambah. Untuk lebih jelasnya, perhatikan contoh berikut.

Contoh Soal 10.2: Hobi Batu Akik

Di belakang peternakan Pak Dengklek, terdapat sebuah gua yang kaya akan batu akik. Setiap kali Pak Dengklek melewatinya, ia akan mengambil sebuah batu akik untuk dibawa pulang. Karena pernah belajar ilmu kebumian, Pak Dengklek dapat menilai seberapa berharganya batu akik yang ia ambil. Untuk kemudahan, Pak Dengklek memberikan nilai harga tersebut berupa suatu bilangan bulat.

Pak Blangkon, sahabat karib Pak Dengklek, sering mengunjungi Pak Dengklek untuk membeli batu akik. Sebagai penggemar berat batu akik, ia hanya tertarik dengan batu akik paling berharga. Pak Blangkon akan meminta Pak Dengklek menunjukkan harga tertinggi yang ada, dan apabila disukai, batu tersebut akan dibeli. Pak Dengklek kadang-kadang kewalahan, karena batu akik yang ia miliki bisa jadi sangat banyak sehingga pencarian batu paling berharga menjadi repot. Ia meminta bantuan Anda untuk melakukan pencatatan atas batu yang dimiliki dan melayani Pak Blangkon secara efisien.

Bantulah Pak Dengklek!

Contoh

Misalkan:

- `simpan(x)` menyatakan Pak Dengklek membawa pulang batu akik seharga x .
- `lihat()` menyatakan Pak Blangkon ingin tahu harga tertinggi untuk batu akik yang dimiliki Pak Dengklek pada saat ini.
- `jual()` menyatakan Pak Dengklek menjual batu akik dengan harga tertinggi kepada Pak Blangkon.

Berikut contoh operasinya dan perilaku yang diharapkan:

- `simpan(5)`, harga-harga batu akik yang disimpan: [5].
- `simpan(7)`, harga-harga batu akik yang disimpan: [5, 7].
- `simpan(3)`, harga-harga batu akik yang disimpan: [5, 7, 3].
- `lihat()`, laporkan bahwa 7 merupakan harga tertinggi dari batu akik yang disimpan.
- `jual()`, harga-harga batu akik yang tersisa: [5, 3].
- `lihat()`, laporkan bahwa 5 merupakan harga tertinggi dari batu akik yang disimpan.

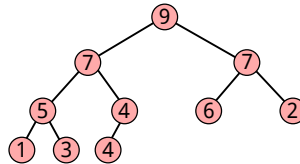
Mari kita mulai dengan solusi sederhana. Solusi paling mudah adalah membuat sebuah *array* dan variabel yang menunjukkan posisi terakhir elemen pada *array*. Untuk setiap operasi `simpan(x)`, isikan nilai x pada elemen *array* yang ditunjuk, geser variabel penunjuk ke belakang, lalu urutkan data secara menaik. Operasi `lihat()` dapat dilayani dengan mengembalikan elemen terbesar yang dipastikan berada di paling belakang *array*. Operasi `jual()` dapat dilayani dengan menggeser variabel penunjuk ke depan, sehingga elemen terbesar kini tidak dianggap berada di dalam *array*.

Misalkan N menyatakan banyaknya elemen yang terisi pada *array*. Jika pengurutan yang digunakan adalah *quicksort*, maka operasi `simpan(x)` berlangsung dalam $O(N \log N)$. Operasi `lihat()` dan `jual()` berlangsung dalam $O(1)$. Perhatikan bahwa pengurutan akan lebih efisien jika digunakan *insertion sort*, sehingga kompleksitas `simpan(x)` menjadi $O(N)$. Kompleksitas dari solusi sederhana ini ditunjukkan pada Tabel 10.1.

Solusi sederhana ini tidak efisien ketika banyak dilakukan operasi `add(x)`. Kita akan mempelajari bagaimana *heap* mengatasi masalah ini secara efisien.

Tabel 10.1: Kompleksitas solusi Hobi Batu Akik dengan solusi sederhana.

Operasi	Dengan <i>insertion sort</i>
simpan(x)	$O(N)$
lihat()	$O(1)$
jual()	$O(1)$

Gambar 10.5: Contoh *binary heap*.

Konsep *Heap*

Secara umum, *heap* mendukung operasi-operasi sebagai berikut:

- *push*, yaitu memasukkan elemen baru ke penyimpanan.
- *pop*, yaitu membuang elemen **terbesar** dari penyimpanan.
- *top*, yaitu mengakses elemen **terbesar** dari penyimpanan.

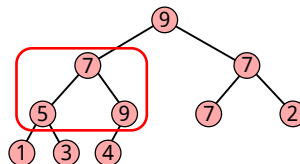
Operasi *heap* tersebut dapat diimplementasikan dengan berbagai cara. Kita akan mempelajari salah satunya, yaitu *binary heap*.

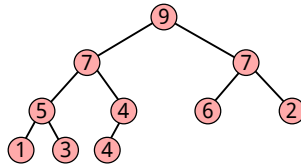
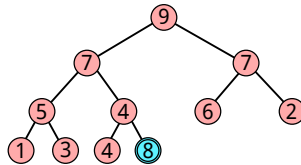
Struktur *Binary Heap*

Struktur data *Binary Heap* memiliki sifat:

- Berstruktur *complete binary tree*.
- Setiap *node* merepresentasikan elemen yang disimpan pada *heap*.
- Setiap *node* memiliki nilai yang **lebih besar** daripada *node* anak-anaknya.

Pada gambar 10.6, pohon tersebut bukanlah sebuah *binary heap*. Perhatikan *node* yang ditandai. *Node* tersebut memiliki nilai 7, sementara salah satu anaknya memiliki

Gambar 10.6: Contoh bukan *binary heap*.

Gambar 10.7: Tahap 1: Bentuk awal *heap*.Gambar 10.8: Tahap 2: Kondisi *heap* setelah ditambahkan *node* baru pada bagian akhir *complete binary tree*.

nilai 9. Ini melanggar aturan setiap *node* memiliki nilai yang **lebih besar** daripada *node* anak-anaknya.

Binary heap perlu memiliki struktur seperti ini untuk menjamin operasi-operasinya dapat dilakukan secara efisien. Misalkan N adalah banyaknya elemen yang sedang disimpan. Operasi *push* dan *pop* bekerja dalam $O(\log N)$, sementara *top* bekerja dalam $O(1)$. Kita akan melihat satu per satu bagaimana operasi tersebut dilaksanakan.

Operasi *Push*

Operasi *push* pada *binary heap* dilakukan dengan 2 tahap:

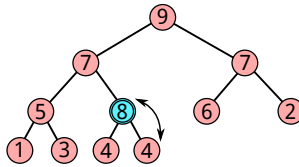
1. Tambahkan *node* baru di posisi yang memenuhi aturan *complete binary tree*.
2. Selama elemen *node* yang merupakan orang tua langsung dari elemen ini memiliki nilai yang lebih kecil, tukar nilai elemen kedua *node* tersebut.

Sebagai contoh, misalkan hendak ditambahkan elemen bernilai 8 ke suatu *binary heap* yang ditunjukkan pada Gambar 10.7. Untuk selanjutnya, Gambar 10.8 sampai dengan Gambar 10.11 menyimulasikan secara bertahap untuk operasi penambahan elemen.

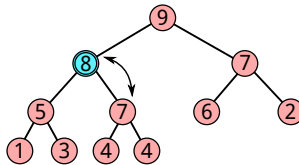
Kasus terburuk terjadi ketika pertukaran yang terjadi paling banyak. Hal ini terjadi ketika elemen yang dimasukkan merupakan nilai yang paling besar pada *heap*. Banyaknya pertukaran yang terjadi sebanding dengan kedalaman dari *complete binary tree*, yaitu $O(\log N)$. Dengan demikian, kompleksitas untuk operasi *push* adalah $O(\log N)$.

Operasi *Pop*

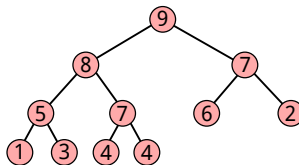
Operasi *pop* pada *binary heap* dilakukan dengan 3 tahap:



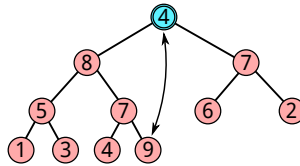
Gambar 10.9: Tahap 3: Penukaran nilai pada *node* baru dengan *parent*-nya karena nilainya lebih kecil.



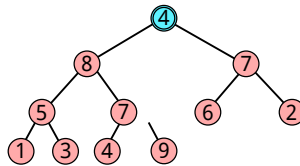
Gambar 10.10: Tahap 4: Penukaran lanjutan nilai pada *node* baru dengan *parent*-nya karena nilainya juga lebih kecil.



Gambar 10.11: Tahap 5: *Node* baru telah memenuhi kondisi yang diperlukan pada *heap*, operasi *push* selesai



Gambar 10.12: Tahap 1: Struktur *heap* setelah elemen *root* dengan elemen terakhir ditukar.

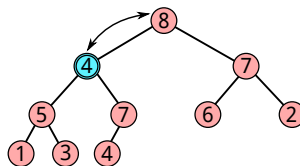


Gambar 10.13: Tahap 2: Hapus elemen terakhir pada *heap*.

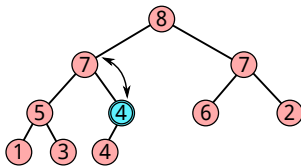
1. Tukar posisi elemen pada *root* dengan elemen terakhir mengikuti aturan *complete binary tree*.
2. Buang elemen terakhir *binary heap*, yang telah berisi elemen dari *root*.
3. Selama elemen yang ditukar ke posisi *root* memiliki anak langsung yang berelemen lebih besar, tukar elemen tersebut dengan salah anaknya yang memiliki elemen **terbesar**.

Misalkan akan dilakukan *pop* pada *heap* pada Gambar 10.11. Untuk selanjutnya, Gambar 10.12 sampai dengan Gambar 10.16 akan menggambarkan simulasi secara bertahap untuk operasi *pop*.

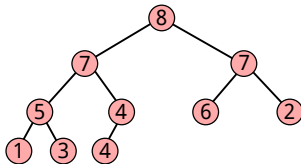
Kasus terburuk juga terjadi ketika pertukaran yang terjadi paling banyak. Hal ini terjadi ketika elemen yang ditempatkan di *root* cukup kecil, sehingga perlu ditukar sampai ke tingkat paling bawah. Banyaknya pertukaran yang terjadi sebanding dengan kedalaman



Gambar 10.14: Tahap 3: Perbaiki struktur *heap* dengan menukar elemen pada *root* dengan anaknya yang bernilai terbesar.



Gambar 10.15: Tahap 4: Lanjutkan penukaran terhadap anaknya yang bernilai terbesar.



Gambar 10.16: Tahap 5: Kini sudah tidak ada anak yang bernilai lebih besar, operasi *pop* selesai.

dari *complete binary tree*. Dengan demikian, kompleksitas untuk operasi *pop* adalah $O(\log N)$.

Operasi *Top*

Operasi *top* adalah operasi untuk mengambil nilai terbesar pada *heap*. Karena setiap *node* memiliki nilai yang **lebih besar** daripada *node* anak-anaknya, maka nilai terbesar dari seluruh *heap* selalu terletak di *root*. Oleh karena itu, operasi ini cukup mengembalikan nilai dari *root*. Kompleksitas operasi ini adalah $O(1)$.

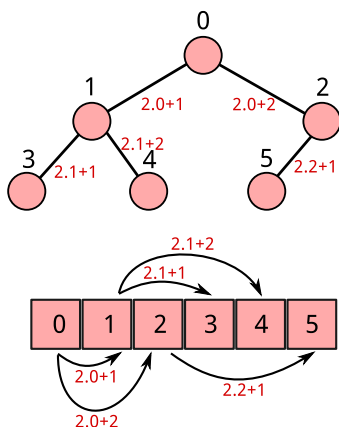
Dengan demikian, berkat penerapan *heap*, seluruh operasi pada persoalan Hobi Batu Akik dapat dilakukan dengan efisien. Tabel 10.2 menunjukkan kompleksitas masing-masing operasi untuk solusi dengan *insertion sort* dan dengan *heap*.

Tabel 10.2: Perbandingan kompleksitas solusi Hobi Batu Akik.

Operasi	Dengan <i>insertion sort</i>	Dengan <i>heap</i>
simpan(x)	$O(N)$	$O(\log N)$
lihat()	$O(1)$	$O(1)$
jual()	$O(1)$	$O(\log N)$

Implementasi *Binary Heap*

Untuk mengimplementasikan *binary heap*, dibutuhkan implementasi dari *tree*. Implementasi representasi *tree* dapat menggunakan teknik representasi graf yang telah



Gambar 10.17: Contoh penomoran indeks *complete binary tree* pada *array*.

dipelajari sebelumnya. Namun, untuk *tree* dengan kondisi tertentu, kita dapat menggunakan representasi yang lebih sederhana. Terutama pada kasus ini, yang mana *tree* yang diperlukan adalah *complete binary tree*.

Complete binary tree dapat direpresentasikan dengan sebuah *array*. Misalkan *array* ini bersifat *zero-based*, yaitu dimulai dari indeks 0. Elemen pada indeks ke- i menyatakan elemen pada *node* ke- i . Anak kiri dari *node* ke- i adalah *node* ke- $(2i + 1)$. Anak kanan dari *node* ke- i adalah *node* ke- $(2i + 2)$.

Dengan logika yang serupa, orang tua dari *node* ke- i adalah *node* ke- $\lfloor \frac{i-1}{2} \rfloor$. Apabila Anda memutuskan untuk menggunakan *array one-based*, maka rumusnya menjadi:

- Anak kiri: $2i$.
- Anak kanan: $2i + 1$.
- Orang tua: $\lfloor i/2 \rfloor$

Karena panjang *array* dapat bertambah atau berkurang, diperlukan *array* yang ukurannya dinamis. Namun, pada contoh ini, kita akan menggunakan *array* berukuran statis dan sebuah variabel yang menyatakan ukuran *array* saat ini. Ukuran *array* statis ini bisa dibuat sebesar banyaknya operasi *push* paling banyak yang mungkin dilakukan, yang misalnya dinyatakan sebagai *maxSize*. Algoritma 53 menunjukkan prosedur untuk melakukan inisialisasi pada *heap*.

Algoritma 53 Inisialisasi pada *heap*, *arrHeap* dan *size* merupakan variabel global.

- 1: $arrHeap \leftarrow \text{new integer}[maxSize]$ ▷ Buat *array arrHeap* berukuran *maxSize*
 - 2: **procedure** INITIALIZEHEAP()
 - 3: $size \leftarrow 0$ ▷ Variabel yang menunjukkan ukuran *array* saat ini.
 - 4: **end procedure**
-

Untuk memudahkan penulisan kode, mari definisikan fungsi pembantu yang ditunjukkan pada Algoritma 54. Secara berturut-turut, fungsi tersebut adalah fungsi untuk mencari indeks orang tua, anak kiri, maupun anak kanan dari suatu *node*. Algoritma 55 sampai dengan Algoritma 57 menunjukkan implementasi untuk operasi *push*, *pop*, dan *top*.

Algoritma 54 Fungsi-fungsi pembantu yang memudahkan implementasi *heap*.

```

1: function GETPARENT( $x$ )
2:   return  $(x - 1) \text{ div } 2$ 
3: end function

4: function GETLEFT( $x$ )
5:   return  $2x + 1$ 
6: end function

7: function GETRIGHT( $x$ )
8:   return  $2x + 2$ 
9: end function

```

Algoritma 55 Prosedur untuk melakukan *push* pada *heap*.

```

1: procedure PUSH( $val$ )
2:    $i \leftarrow size$ 
3:    $arrHeap[i] \leftarrow val$ 
4:   while  $(i > 0) \wedge (arrHeap[i] > arrHeap[GETPARENT(i)])$  do
5:     SWAP( $arrHeap[i]$ ,  $arrHeap[GETPARENT(i)]$ )
6:      $i \leftarrow GETPARENT(i)$ 
7:   end while
8:    $size \leftarrow size + 1$ 
9: end procedure

```

Pembangunan *Heap* Secara Efisien

Ketika Anda memiliki data N elemen, dan hendak dimasukkan ke dalam *heap*, Anda dapat:

1. Membuat *heap* kosong, lalu melakukan *push* satu per satu hingga seluruh data dimuat *heap*. Kompleksitasnya $O(N \log N)$, atau
2. Membuat *array* dengan N elemen, lalu *array* ini dibuat menjadi *heap* dalam $O(N)$ menggunakan cara yang lebih efisien.

Untuk membuat *heap* dari N elemen secara efisien, kita dapat membuat *array* dengan N elemen, lalu isi *array* ini dengan elemen-elemen yang akan dimasukkan pada *heap*. Kemudian lakukan langkah berikut untuk masing-masing *node*, mulai dari tingkat kedua dari paling bawah sampai ke tingkat paling atas:

1. Misalkan *node* ini adalah *node* x .
2. Jadikan *subtree* yang bermula pada *node* x ini agar membentuk *heap* dengan suatu operasi baru, yaitu *heapify*.

Algoritma 56 Prosedur untuk melakukan *pop* pada *heap*.

```

1: procedure POP()
2:   SWAP(arrHeap[0], arrHeap[size - 1])
3:   size ← size - 1                                ▷ Lakukan pembuangan pada elemen terakhir
4:   i ← 0
5:   swapped ← true
6:   while swapped do                               ▷ Perulangan untuk perbaikan struktur heap
7:     maxIdx ← i
8:     if (GETLEFT(i) < size) ∧ (arrHeap[maxIdx] < arrHeap[GETLEFT(i)]) then
9:       maxIdx ← GETLEFT(i)
10:    end if
11:    if (GETRIGHT(i) < size) ∧ (arrHeap[maxIdx] < arrHeap[GETRIGHT(i)]) then
12:      maxIdx ← GETRIGHT(i)
13:    end if
14:    SWAP(arrHeap[i], arrHeap[maxIdx])
15:    swapped ← (maxIdx ≠ i)                        ▷ true bila terjadi pertukaran
16:    i ← maxIdx
17:  end while
18: end procedure

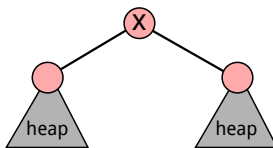
```

Algoritma 57 Fungsi untuk melakukan *top* pada *heap*

```

1: function TOP()
2:   return arrHeap[0]
3: end function

```



Gambar 10.18: Operasi *heapify* hanya dapat bekerja apabila *subtree* kiri dan *subtree* kanan dari *root* telah membentuk *heap*.

Heapify adalah operasi untuk membentuk sebuah *heap* yang bermula pada suatu *node*, dengan asumsi anak kiri dan anak kanan *node* tersebut sudah membentuk *heap*.

Berhubung kedua anak telah membentuk *heap*, kita cukup memindahkan elemen *root* ke posisi yang tepat. Jika elemen *root* sudah lebih besar daripada elemen anaknya, tidak ada yang perlu dilakukan. Sementara bila elemen *root* lebih kecil daripada salah satu elemen anaknya, tukar elemennya dengan elemen salah satu anaknya yang **paling besar**. Kegiatan ini sebenarnya sangat mirip dengan yang kita lakukan pada operasi *pop*. *Heapify* dapat diimplementasikan seperti yang ditunjukkan pada Algoritma 58.

Algoritma 58 Prosedur untuk melakukan *heapify* pada *heap*.

```

1: procedure HEAPIFY(rootIdx)
2:   i ← rootIdx
3:   swapped ← true
4:   while swapped do
5:     maxIdx ← i
6:     if (GETLEFT(i) < size) ∧ (arrHeap[maxIdx] < arrHeap[GETLEFT(i)]) then
7:       maxIdx ← GETLEFT(i)
8:     end if
9:     if (GETRIGHT(i) < size) ∧ (arrHeap[maxIdx] < arrHeap[GETRIGHT(i)]) then
10:      maxIdx ← GETRIGHT(i)
11:    end if
12:    SWAP(arrHeap[i], arrHeap[maxIdx])
13:    swapped ← (maxIdx ≠ i)                                ▷ true bila terjadi pertukaran
14:    i ← maxIdx
15:  end while
16: end procedure

```

Dengan adanya operasi *heapify*, operasi *pop* sendiri dapat kita sederhanakan seperti yang ditunjukkan pada Algoritma 59.

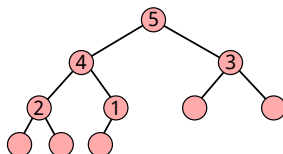
Sekali melakukan *heapify*, kasus terburuknya adalah elemen *root* dipindahkan hingga ke paling bawah *tree*. Jadi kompleksitasnya adalah $O(H)$, dengan H adalah ketinggian dari *complete binary tree*. Berhubung $H = O(\log N)$, dengan N adalah banyaknya elemen pada *heap*, kompleksitas *heapify* adalah $O(\log N)$.

Algoritma 59 Operasi *pop* yang disederhanakan dengan *heapify*.

```

1: procedure POP()
2:   SWAP(arrHeap[0], arrHeap[size - 1])
3:   size ← size - 1           ▷ Lakukan pembuangan pada elemen terakhir
4:   HEAPIFY(0)
5: end procedure

```



Gambar 10.19: Ilustrasi urutan pelaksanaan *heapify*. Angka pada *node* menyatakan urutan pelaksanaan *heapify*.

Implementasi Pembangunan *Heap* Secara Efisien

Setelah memahami *heapify*, kita dapat menulis prosedur pembangunan *heap* dari array A berisi N elemen secara efisien. Prosedur ini ditunjukkan pada Algoritma 60. Elemen terakhir yang berada pada tingkat kedua paling bawah dapat ditemukan dengan mudah, yaitu elemen dengan indeks $N/2$ dibulatkan ke bawah dan dikurangi 1.

Algoritma 60 Prosedur untuk membuat *heap* secara efisien dari array A .

```

1: procedure MAKEHEAP( $A, N$ )
2:   INITIALIZEHEAP( $N$ )
3:   for  $i \leftarrow 0, N - 1$  do
4:     arrHeap[size] ←  $A[i]$            ▷ Salin array  $A$  ke array heap
5:     size ← size + 1
6:   end for
7:   for  $i \leftarrow \lfloor N/2 \rfloor - 1$  down to 0 do           ▷ Lakukan proses heapify
8:     HEAPIFY( $i$ )
9:   end for
10: end procedure

```

Analisis Pembangunan *Heap* Secara Efisien

Proses *heapify* melakukan $N/2$ kali operasi yang masing-masing memiliki kompleksitas waktu $O(\log N)$, sehingga kompleksitas akhirnya terkesan $O(N \log N)$. Namun pada kasus ini, sebenarnya setiap *heapify* tidak benar-benar $O(\log N)$. Kenyataannya, banyak operasi *heapify* yang dilakukan pada tingkat bawah, yang relatif lebih cepat dari *heapify* pada tingkat di atas. Perhitungan secara matematis¹⁵ membuktikan bahwa kompleksitas keseluruhan MAKEHEAP adalah $O(N)$.

Catatan Implementasi *Heap*

Tentu saja, Anda dapat membuat *heap* dengan urutan yang terbalik, yaitu elemen terkecilnya di atas. Dengan demikian, operasi yang didukung adalah mencari atau menghapus elemen terkecil. Biasanya *heap* dengan sifat ini disebut dengan **min-heap**, sementara *heap* dengan elemen terbesar di atas disebut dengan **max-heap**. Agar lebih rapi, Anda dapat menggunakan **struct** (C) atau **class** (C++) pada implementasi *heap*. Bagi pengguna C++, struktur data **priority_queue** dari *header queue* merupakan struktur data *heap*.

Pada ilmu komputer, *heap* dapat digunakan sebagai *priority queue*, yaitu antrean yang terurut menurut suatu kriteria. Sifat *heap* juga dapat digunakan untuk optimisasi suatu algoritma. Contoh paling nyatanya adalah untuk mempercepat algoritma Dijkstra, yang akan kita pelajari pada Bab 11. Berbagai solusi persoalan *greedy* juga dapat diimplementasikan secara efisien dengan *heap*. *Heap* dapat pula digunakan untuk pengurutan yang efisien, dan akan dibahas pada bagian berikutnya.

Dengan mempelajari *heap*, Anda memperdalam pemahaman tentang bagaimana penggunaan struktur data yang tepat dapat membantu menyelesaikan persoalan tertentu secara efisien.

Heapsort

Apakah Anda masih ingat dengan *selection sort*? Berikut adalah ringkasan langkah-langkahnya:

1. Pilih elemen terkecil, lalu tempatkan di paling awal data.
2. Ulangi hingga seluruh data terurut menaik.

Pada *selection sort*, langkah pencarian elemen terkecil dilakukan dengan *linear search*. Langkah ini bisa kita optimisasikan menggunakan struktur data *heap*, yang bisa mencari elemen terkecil dengan kompleksitas logaritmik. Algoritma pengurutan menggunakan *heap* ini dinamakan **heapsort**. Implementasinya ditunjukkan pada Algoritma 61.

Algoritma 61 Prosedur *heapsort* untuk mengurutkan *array A* dengan ukuran N secara menaik.

```

1: procedure HEAPSORT(A, N)
2:   MAKEHEAP(A, N)
3:   for  $i \leftarrow 0, N - 1$  do
4:      $A[i] \leftarrow \text{TOP}()$ 
5:     POP()
6:   end for
7: end procedure

```

▷ Heap yang dibuat adalah *min-heap*

▷ Simpan elemen terkecil ke- i pada posisi ke- i

▷ Hapus elemen terkecil ke- i dari *heap*

Mari kita analisis kompleksitas dari algoritma *heapsort*. Sebagaimana dijelaskan pada bagian sebelumnya, operasi MAKEHEAP memiliki kompleksitas waktu $O(N)$. Kemudian, kita melakukan N kali operasi POP, yang masing-masing memiliki kompleksitas waktu $O(\log N)$. Maka, kompleksitas waktu dari algoritma *heapsort* adalah $O(N + N \log N) = O(N \log N)$.

Heapsort memiliki kompleksitas waktu yang sama dengan *quicksort* maupun *merge sort*, yakni $O(N \log N)$. Namun, *heapsort* memiliki keuntungan yang tidak dimiliki keduanya, yaitu bisa melakukan *partial sort* (pengurutan parsial).

Pengurutan parsial adalah aktivitas mengurutkan K elemen terkecil (atau terbesar) saja dari suatu *array*. *Selection sort* juga bisa melakukan pengurutan parsial, namun membutuhkan waktu $O(NK)$. Dengan *heapsort*, untuk melakukan pengurutan parsial kita cukup melakukan operasi POP sebanyak K kali saja. Kompleksitas waktunya menjadi $O(N + K \log N)$, yang lebih baik dari $O(NK)$.

11 Algoritma Graf

Setelah Anda memahami cara merepresentasikan dan menyelesaikan persoalan graf sederhana, kini waktunya untuk mempelajari algoritma graf yang lebih rumit. Algoritma yang akan dipelajari pada bab ini adalah penyelesaian untuk permasalahan yang klasik pada graf, yaitu pencarian *shortest path* dan *Minimum Spanning Tree* (MST).

Untuk memulai pembahasan tentang algoritma graf, asumsikan:

- V menyatakan banyaknya *node*.
- E menyatakan banyaknya *edge*.
- $w[a][b]$ menyatakan bobot *edge* yang menghubungkan *node* a dengan *node* b .
- *Node* pada graf dinomori dari 1 sampai dengan V .
- $adj(x)$ menyatakan himpunan *node* yang merupakan tetangga dari *node* x .

Shortest Path

Salah satu persoalan umum yang dihadapi pada permasalahan graf adalah pencarian *shortest path* (jalur terpendek). Definisi masalah yang diberikan adalah: diberikan graf, cari serangkaian *edge* yang perlu dilalui mulai dari suatu *node* menuju suatu *node* lainnya, sedemikian sehingga jumlah bobot dari *edge* yang dilalui sekecil mungkin. Pada kasus graf tidak berbobot, kita dapat mengasumsikan bobot setiap *edge* adalah 1.

Definisi bobot dan jalur terpendek bergantung pada masalah yang dihadapi. Misalnya bobot *edge* pada suatu graf yang merepresentasikan kota dan jalan menyatakan waktu tempuh, berarti *shortest path* menyatakan total waktu tempuh paling sedikit yang diperlukan untuk berpindah dari suatu kota ke kota lainnya.

Terdapat beberapa algoritma yang menyelesaikan masalah *shortest path*. Apabila kita diberikan graf yang tidak berbobot, pencarian *shortest path* dapat menggunakan penelusuran graf secara BFS. Untuk graf berbobot, diperlukan algoritma yang lebih kompleks. Pada bab ini, kita akan mempelajari algoritma Dijkstra, Bellman—Ford, dan Floyd—Warshall. Nama ketiga algoritma tersebut berasal dari nama pencipta-penciptanya.

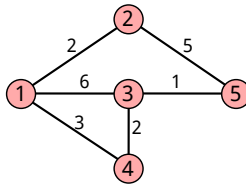
Dijkstra

Konsep

Algoritma Dijkstra menggunakan konsep *greedy* untuk menemukan *shortest path* dari suatu *node*, misalnya s , ke semua *node* lainnya. Untuk setiap tahapan, diperlukan pencatatan informasi berikut untuk seluruh *node*:

- $dist[x]$: jarak terpendek yang sejauh ini ditemukan untuk mencapai *node* x dari *node* s .
- $visited[x]$: bernilai *true* apabila suatu *node* x telah dikunjungi.

Sebagai tambahan, kita juga dapat mencatat *node* mana yang dikunjungi sebelum mencapai suatu *node* x pada *shortest path*. Kita nyatakan *node* tersebut sebagai $pred[x]$. Informasi ini diperlukan apabila Anda membutuhkan *edge-edge* mana yang perlu digunakan pada *shortest path*.



Gambar 11.1: Graf berbobot dan tidak berarah.

►► Sekilas Info ◀◀

Apabila soal yang dihadapi tidak membutuhkan informasi *edge-edge* yang perlu digunakan pada *shortest path*, Anda dapat menghemat waktu dan pengetikan kode dengan mengabaikan $pred[x]$.

Pada awalnya, seluruh *node* x memiliki nilai $dist[x] = \infty$, dan $visited[x] = false$. Khusus untuk *node* s , kita mengetahui bahwa jarak terpendek untuk mencapai s dari s sendiri adalah 0. Oleh sebab itu, $dist[s] = 0$. Seluruh nilai $pred[x]$ bisa kita isi -1 , sebab tidak diketahui *node* mana yang dikunjungi sebelum x .

Setelah inisialisasi selesai dilakukan, jalankan langkah-langkah berikut:

1. Pilih sebuah *node* yang belum dikunjungi dan memiliki $dist$ terkecil. Sebut saja *node* ini sebagai u .
2. Apabila tidak ada lagi *node* yang belum dikunjungi, atau $dist[u]$ bernilai ∞ , artinya tidak ada lagi *node* yang dapat dikunjungi. Algoritma Dijkstra berakhir.
3. Karena u dikunjungi, maka set $visited[u] = true$. Kini dipastikan *shortest path* dari s menuju u adalah $dist[u]$. Diketahui pula *edge* $(pred[u], u)$ merupakan *edge* yang perlu digunakan pada *shortest path* ke u .
4. Untuk setiap *node* v yang merupakan tetangga dari u , lakukan sebuah proses yang disebut "*relax*". Proses *relax* untuk *node* u dan v adalah proses untuk memperbaharui jarak terpendek mencapai v , apabila ternyata mencapai v melalui u membutuhkan jarak yang lebih kecil. Dengan kata lain, dilakukan:

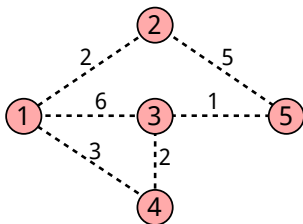
$$dist[v] = \min(dist[v], dist[u] + w[u][v])$$

Setiap kali $dist[v]$ mengalami perubahan, ubah nilai $pred[v]$ menjadi u . Sebab sejauh ini diketahui bahwa untuk mencapai v pada *shortest path*, *node* sebelumnya adalah u .

5. Kembali ke langkah 1.

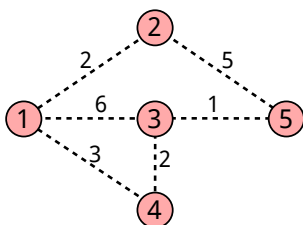
Mari kita simulasikan Dijkstra pada graf berbobot dan tidak berarah yang ditunjukkan pada Gambar 11.1. Misalnya *node* permulaan adalah 1, sehingga kita hendak mencari *shortest path* dari 1 ke semua *node* lainnya. Gambar 11.2 sampai Gambar 11.7 menunjukkan simulasi Dijkstra pada graf tersebut.

Gambar 11.7 menunjukkan *shortest path* mulai dari *node* 1 ke seluruh *node* lainnya. Anda dapat melihat kebenaran hasil algoritma Dijkstra dengan membandingkannya dengan



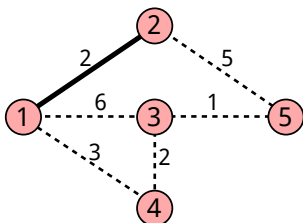
	1	2	3	4	5
<i>dist</i>	0	∞	∞	∞	∞
<i>visited</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>
<i>pred</i>	-1	-1	-1	-1	-1

Gambar 11.2: Tahap 1: Lakukan inisialisasi untuk seluruh data *node*. Perhatikan perbedaan nilai inisialisasi untuk *node* permulaan, yaitu *node* 1.



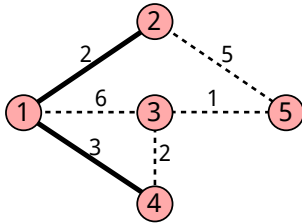
	1	2	3	4	5
<i>dist</i>	0	2	6	3	∞
<i>visited</i>	T	T	T	T	F
<i>pred</i>	-1	<u>1</u>	<u>1</u>	<u>1</u>	-

Gambar 11.3: Tahap 2: *Node* yang belum dikunjungi dan memiliki nilai *dist* terkecil adalah 1. Tandainya dengan $visited[1] = true$, dan *relax* seluruh tetangganya.



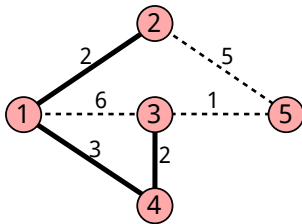
	1	2	3	4	5
<i>dist</i>	0	2	6	3	7
<i>visited</i>	<i>T</i>	T	<i>F</i>	<i>F</i>	<i>F</i>
<i>pred</i>	-1	1	1	1	<u>2</u>

Gambar 11.4: Tahap 3: *Node* yang belum dikunjungi dan memiliki nilai *dist* terkecil adalah 2. Tandainya dengan $visited[2] = true$, dan *relax* seluruh tetangganya. Hanya *node* 5 yang mengalami perubahan *dist*. Kini diketahui pula bahwa *edge* $(1,2)$ merupakan bagian dari *shortest path* ke 2.



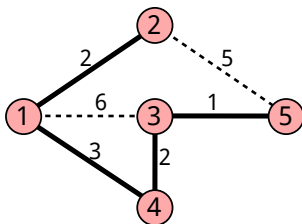
	1	2	3	4	5
<i>dist</i>	0	2	<u>5</u>	3	7
<i>visited</i>	<i>T</i>	<i>T</i>	<i>F</i>	T	<i>F</i>
<i>pred</i>	-1	1	<u>4</u>	1	2

Gambar 11.5: Tahap 4: *Node* yang belum dikunjungi dan memiliki nilai *dist* terkecil adalah 4. Ubah nilai *visited* dan *relax* seluruh tetangganya. *Node* 3 mengalami perubahan *dist*.



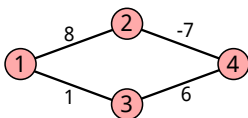
	1	2	3	4	5
<i>dist</i>	0	2	5	3	<u>6</u>
<i>visited</i>	<i>T</i>	<i>T</i>	T	<i>T</i>	<i>F</i>
<i>pred</i>	-1	1	4	1	<u>3</u>

Gambar 11.6: Tahap 5: *Node* yang belum dikunjungi dan memiliki nilai *dist* terkecil adalah 3. Ubah nilai *visited* dan *relax* seluruh tetangganya. Hanya *node* 5 yang mengalami perubahan *dist*.



	1	2	3	4	5
<i>dist</i>	0	2	5	3	6
<i>visited</i>	<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>	T
<i>pred</i>	-1	1	4	1	3

Gambar 11.7: Tahap 6: Satu-satunya *node* yang belum dikunjungi adalah 5. Ubah nilai *visited* dan *relax* seluruh tetangganya. Algoritma selesai.



Gambar 11.8: Contoh graf yang mengakibatkan algoritma Dijkstra gagal. *Shortest path* dari 1 ke 4 yang dihasilkan algoritma Dijkstra adalah melalui 1, 3, lalu 4 dengan jarak total 7. Padahal, melalui 1, 2, lalu 4 memiliki jarak total 1.

pencarian *shortest path* secara manual.

Sifat algoritma Dijkstra yang pada setiap tahapnya selalu memilih *node* berjarak tempuh terkecil merupakan suatu bentuk *greedy*. Diasumsikan bahwa pengutamaan perjalanan yang jarak tempuhnya terkecil saat itu akan menghasilkan *shortest path* yang benar. Hal ini akan selalu benar untuk graf yang tidak memiliki bobot negatif. Untuk kasus graf yang memiliki bobot negatif, asumsi ini tidak lagi memberikan *shortest path* yang optimal. Sebab mungkin saja terdapat suatu jalur yang jarak tempuhnya sangat besar, tetapi pada akhirnya akan melewati *edge* dengan bobot negatif yang besar pula. Perhatikan Gambar 11.8 untuk lebih jelasnya.

Implementasi

Algoritma 62 menunjukkan implementasi untuk algoritma Dijkstra. Representasi graf yang cocok untuk digunakan adalah *adjacency list*, sehingga pencacahan seluruh tetangga dari suatu *node* efisien.

Untuk mencetak *edge-edge* yang perlu dilalui untuk mencapai suatu *node* x dari *node* permulaan (yaitu s), cukup lakukan penelusuran dari x ke belakang menggunakan informasi $pred[x]$ sampai mencapai s . Algoritma pencariannya ditunjukkan pada Algoritma 63. Waspada bahwa belum tentu semua *node* x dapat dikunjungi dari s . Ketika hal ini terjadi, $pred[x]$ masih tetap bernilai -1 .

Untuk kasus terburuk, seluruh *node* dapat dicapai dari s sehingga terdapat V iterasi. Pada setiap iterasi, dilakukan pencarian *node* untuk di-*relax* dan proses *relax*. Algoritma 62 menunjukkan pencarian *node* untuk di-*relax* yang diimplementasikan dengan *linear search*, sehingga kompleksitas sekali pencariannya adalah $O(V)$. Untuk proses *relax* sendiri, total kompleksitasnya ketika seluruh *node* di-*relax* adalah $O(E)$. Kompleksitas total untuk implementasi ini adalah $O(V^2 + E)$. Kompleksitas ini juga dapat dituliskan sebagai $O(V^2)$ saja, sebab banyaknya *edge* dibatasi oleh V^2 .

Optimisasi dengan *heap*

Pencarian *node* untuk di-*relax* sebenarnya merupakan pencarian nilai terkecil dari suatu kumpulan. Kita dapat menggunakan struktur data *heap* untuk membuat pencarian ini lebih efisien, yaitu $O(\log V)$ per pencarian.

Heap yang kita perlukan adalah *min-heap*, yang menyimpan daftar *node* yang terurut berdasarkan jarak tempuh sejauh ini. Untuk mempermudah implementasi, kita dapat membuat *heap* yang dapat menyimpan elemen berupa pasangan data (*jarak, node*). Setiap kali jarak tempuh suatu *node* diperbaharui, masukkan datanya ke dalam *heap*. Hal ini

Algoritma 62 Implementasi algoritma Dijkstra.

```

1: function DIJKSTRA(s)
2:   dist ← new integer[V + 1]
3:   visited ← new boolean[V + 1]
4:   pred ← new integer[V + 1]
5:   FILLARRAY(dist, ∞)
6:   FILLARRAY(visited, false)
7:   FILLARRAY(pred, -1)

8:   dist[s] ← 0
9:   while true do                                     ▷ Perulangan ini akan diakhiri dengan break
10:    u ← -1
11:    minDist ← ∞
12:    for i ← 1, V do                                 ▷ Cari node yang belum dikunjungi dan memiliki dist terkecil
13:      if (not visited[i] ∧ (dist[i] < minDist)) then
14:        u ← i
15:        minDist ← dist[i]
16:      end if
17:    end for
18:    if (u = -1) ∨ ISINFINITE(dist[u]) then
19:      break                                         ▷ Akhiri perulangan while
20:    end if

21:    visited[u] ← true
22:    for v ∈ adj(u) do                               ▷ Lakukan relax untuk semua tetangga u
23:      if dist[v] > dist[u] + w[u][v] then
24:        dist[v] = dist[u] + w[u][v]
25:        pred[v] = u
26:      end if
27:    end for
28:  end while

29:  return dist                                     ▷ Kembalikan tabel shortest path yang bermula dari s
30: end function

```

Algoritma 63 Mencetak *edge-edge* yang dilalui untuk mencapai *node x*.

```

1: procedure PRINTPATH(x)
2:   if (pred[x] = -1) ∧ (x ≠ s) then
3:     print "Tidak ada jalan"
4:   else if pred[x] ≠ -1 then
5:     PRINTPATH(pred[x])
6:     print pred[x], x
7:   end if
8: end procedure

```

memungkinkan *heap* menyimpan beberapa *node* yang sama dengan jarak yang berbeda. Tidak menjadi masalah, sebab ketika hasil *pop heap* adalah *node* yang sudah dikunjungi, kita cukup mengabaikannya. Algoritma 64 menunjukkan implementasi yang dioptimisasi dengan *heap*.

Algoritma 64 Implementasi algoritma Dijkstra dengan optimisasi *heap*.

```

1: function DIJKSTRA(s)
2:   dist ← new integer[V + 1]
3:   visited ← new boolean[V + 1]
4:   pred ← new integer[V + 1]
5:   FILLARRAY(dist, ∞)
6:   FILLARRAY(visited, false)
7:   FILLARRAY(pred, -1)

8:   INITIALIZEHEAP()                                ▷ Buat sebuah min-heap
9:   PUSH((0, s))                                    ▷ Masukkan node s dengan jarak tempuh 0 ke dalam heap
10:  dist[s] ← 0
11:  while not ISEMPYHEAP() do
12:    (curDist, u) ← POP()                          ▷ Dapatkan node yang jarak tempuhnya terkecil

13:    if not visited[u] then                          ▷ Hanya proses apabila u belum dikunjungi
14:      visited[u] ← true
15:      for v ∈ adj(u) do                            ▷ Lakukan relax untuk semua tetangga u
16:        if dist[v] > dist[u] + w[u][v] then
17:          dist[v] = dist[u] + w[u][v]
18:          pred[v] = u
19:          PUSH((dist[v], v))                        ▷ Masukkan jarak dan node yang diperbaharui ke
             heap
20:        end if
21:      end for
22:    end if
23:  end while
24:  return dist
25: end function

```

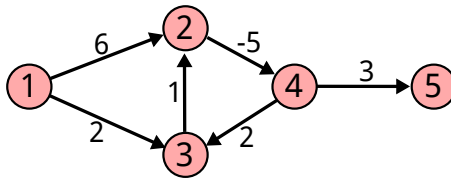
Optimisasi ini membuat kompleksitas algoritma Dijkstra berjalan dalam $O((V + E) \log V)$.

Bellman—Ford

Konsep

Untuk mengatasi kekurangan algoritma Dijkstra pada graf dengan bobot negatif, kita dapat menggunakan algoritma Bellman—Ford. Algoritma ini menerima input berupa graf dan sebuah *node* awal, misalnya *s*, dan mengembalikan *shortest path* dari *s* ke seluruh *node* lainnya. Algoritma ini juga memberikan kemampuan tambahan berupa pendeteksian *negative cycle*.

Negative cycle merupakan *cycle* pada graf berbobot yang jumlah bobotnya negatif. Ketika *negative cycle* dapat dicapai dari suatu *node* awal dan dapat menuju ke suatu *node*



Gambar 11.9: Contoh graf yang memiliki *negative cycle*, yaitu $[2,3,4]$. *Shortest path* dari 1 ke 5 tidak terdefinisi karena adanya *negative cycle*.

akhir, maka *shortest path* dari kedua *node* tersebut menjadi tidak terdefinisi. Sebab, perjalanan dari *node* awal ke *node* akhir dapat berputar-putar di *negative cycle* sambil terus mengurangi bobot perjalanannya. Gambar 11.9 menunjukkan contoh graf dengan *negative cycle*.

Ide dari algoritma Bellman–Ford adalah terus menerus melakukan proses *relax*, sampai pada akhirnya tidak ada perubahan bobot perjalanan dan *shortest path* yang benar ditemukan. Observasi yang dimanfaatkan algoritma ini adalah, *shortest path* dari suatu *node* ke *node* lainnya dipastikan tidak melibatkan lebih dari $V - 1$ *edge*. Artinya apabila sebanyak $V - 1$ kali seluruh *edge* di-*relax*, maka seluruh *shortest path* akan ditemukan. Dengan demikian, algoritma ini dapat disimpulkan menjadi: sebanyak $V - 1$ kali, *relax* seluruh *edge*.

Untuk mendeteksi keberadaan *negative cycle*, kita dapat melakukan *relax* tambahan sebanyak satu kali. Jika masih terdapat bobot perjalanan yang berubah, dipastikan perubahan itu disebabkan oleh *negative cycle*.

Implementasi

Representasi graf yang cocok digunakan adalah *edge list* atau *adjacency list*, untuk mendapatkan informasi seluruh *edge* secara efisien. Algoritma 65 menunjukkan implementasinya, sementara Algoritma 66 menunjukkan fungsi tambahan untuk mendeteksi keberadaan *negative cycle*.

Terdapat $V - 1$ tahapan relaksasi *edge*, yang mana setiap relaksasi bekerja dalam $O(E)$. Secara jelas, kita dapat memperhatikan bahwa kompleksitas algoritma Bellman–Ford adalah $O(VE)$. Meskipun lebih lambat daripada Dijkstra, algoritma ini bersifat lebih umum.

Floyd–Warshall

Konsep

Kadang-kadang, kita membutuhkan informasi *shortest path* dari setiap *node* ke setiap *node* lainnya pada suatu graf. Selain dengan menjalankan Dijkstra atau Bellman–Ford satu per satu untuk setiap *node*, kita juga dapat menggunakan algoritma Floyd–Warshall.

Konsep dari algoritma ini adalah *dynamic programming*. Definisikan sebuah fungsi $f(i, j, k)$ yang menyatakan jarak terpendek untuk berpindah dari *node* i ke *node* j , dengan

Algoritma 65 Implementasi algoritma Bellman–Ford.

```
1: function BELLMAN-FORD( $s$ )
2:    $dist \leftarrow$  new integer[ $V + 1$ ]

3:   FILLARRAY( $dist, \infty$ )
4:    $dist[s] \leftarrow 0$ 
5:   for  $i \leftarrow 1, V - 1$  do
6:     for  $\langle u, v \rangle \in edgeList$  do
7:       if  $dist[v] > dist[u] + w[u][v]$  then
8:          $dist[v] = dist[u] + w[u][v]$ 
9:       end if
10:    end for
11:  end for

12:  return  $dist$ 
13: end function
```

Algoritma 66 Pendeteksian *negative cycle* dengan memanfaatkan hasil pencarian *shortest path* menggunakan Bellman–Ford.

```
1: function HASNEGATIVECYCLE( $s$ )
2:    $dist \leftarrow$  BELLMAN-FORD( $s$ )

3:   for  $\langle u, v \rangle \in edgeList$  do
4:     if  $dist[v] > dist[u] + w[u][v]$  then
5:       return true
6:     end if
7:   end for
8:   return false
9: end function
```

hanya menggunakan *node-node* perantara $\{1, 2, 3, \dots, k\}$. Nilai dari $f(i, j, k)$ merupakan salah satu yang terkecil dari:

- Jarak terpendek dari i ke j , apabila tidak menggunakan k sebagai perantaranya. Nilai ini sama dengan $f(i, j, k - 1)$.
- Jarak terpendek dari i ke j , apabila k digunakan sebagai perantaranya. Jaraknya adalah jarak i ke k , ditambah k ke j , dengan masing-masing hanya boleh menggunakan $\{1, 2, 3, \dots, k - 1\}$ sebagai perantaranya. Nilai ini sama dengan $f(i, k, k - 1) + f(k, j, k - 1)$.

Kasus ketika perpindahan i ke j tidak boleh menggunakan perantara menjadi *base case*. Kasus ini terjadi ketika $k = 0$, yang artinya *node* perantara yang boleh digunakan berupa himpunan kosong. Pada kasus ini, $f(i, j, 0) = w[i][j]$.

Dengan demikian, dapat dirumuskan DP berikut:

$$f(i, j, k) = \begin{cases} w[i][j], & k = 0 \\ \min(f(i, j, k - 1), f(i, k, k - 1) + f(k, j, k - 1)), & k > 0 \end{cases}$$

Shortest path dari *node* s ke *node* t dinyatakan pada $f(s, t, V)$.

Implementasi

Seperti DP umumnya, kita dapat menggunakan *top-down* atau *bottom-up*. Implementasi yang biasa digunakan adalah versi *bottom-up* menggunakan optimisasi *flying table*, karena pendek dan mudah ditulis. Optimisasi *flying table* dapat dilakukan sebab perhitungan DP $f(-, -, k)$ hanya membutuhkan informasi $f(-, -, k - 1)$. Representasi graf yang mudah digunakan adalah *adjacency matrix*. Algoritma 67 menunjukkan implementasinya.

Terlihat jelas bahwa kompleksitas waktunya adalah $O(V^3)$. Meskipun lebih lambat daripada menggunakan algoritma Dijkstra atau Bellman–Ford satu per satu, Floyd–Warshall memiliki kelebihan dalam kemudahan implementasinya. Cukup dengan tiga lapis *for ... loop*, *shortest path* antar setiap *node* ditemukan. Algoritma ini cocok digunakan apabila hanya terdapat sedikit *node*, dan Anda hendak menghemat waktu pengetikan kode.

Minimum Spanning Tree

Persoalan lainnya yang cukup klasik pada permasalahan graf adalah mencari **Minimum Spanning Tree** pada graf berbobot tidak berarah. **Spanning Tree** dari suatu graf adalah subgraf yang mana seluruh *node* pada graf tersebut terhubung dan membentuk *tree*.

Perhatikan contoh pada Gambar 11.10. Kita diberikan sebuah graf seperti pada gambar (a). Untuk membentuk *spanning tree*, kita akan memilih beberapa *edge* dari graf asli sedemikian sehingga *edge* yang diambil membentuk *tree* yang menghubungkan semua *node*. Gambar (b) merupakan contoh *spanning tree* yang didapat dengan mengambil *edge* $\{(1, 4), (2, 4), (3, 4), (4, 5)\}$. Sementara itu, gambar (c) bukanlah merupakan *spanning tree* dari (a) karena subgraf yang dihasilkan tidak membentuk *tree*. Gambar (d) juga bukanlah *spanning tree* dari (a) karena ada salah satu *node* yang tidak terhubung, yaitu *node* 1.

Algoritma 67 Implementasi algoritma Floyd–Warshall dengan DP *bottom-up*.

```

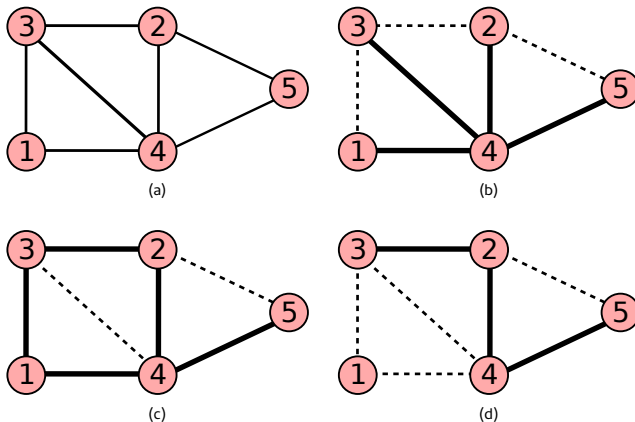
1: function FLOYD-WARSHALL()
2:    $dist \leftarrow \text{new integer}[V + 1][V + 1]$ 
3:   FILLARRAY( $dist, \infty$ )
4:   for  $i \leftarrow 1, V$  do
5:     for  $j \leftarrow 1, V$  do
6:        $dist[i][j] \leftarrow w[i][j]$  ▷ Base case
7:     end for
8:   end for

9:   for  $k \leftarrow 1, V$  do ▷ Pengisian tabel
10:    for  $i \leftarrow 1, V$  do
11:      for  $j \leftarrow 1, V$  do
12:         $dist[i][j] \leftarrow \min(dist[i][j], dist[i][k] + dist[k][j])$ 
13:      end for
14:    end for
15:  end for

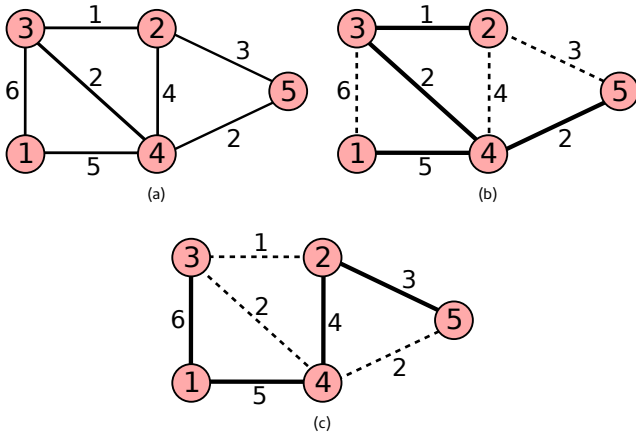
16:  return  $dist$ 
17: end function

```

▷ Kini $dist[i][j]$ berisi *shortest path* dari i ke j
 ▷ Kembalikan tabel DP



Gambar 11.10: Diberikan suatu graf seperti pada gambar (a). Gambar (b) merupakan salah satu *spanning tree* dari (a), namun (c) dan (d) bukanlah *spanning tree* dari (a).



Gambar 11.11: Diberikan suatu graf berbobot seperti pada gambar (a). Gambar (b) merupakan contoh MST dari (a), namun (b) bukanlah MST dari (a).

Minimum Spanning Tree (MST) adalah *spanning tree* pada graf berbobot yang mana total bobot dari seluruh *edge* pada *spanning tree* tersebut minimum. Perhatikan ilustrasi pada Gambar 11.11. Misalkan diberikan graf berbobot seperti pada gambar (a). Gambar (b) dan (c) merupakan *spanning tree* dari (a), namun total bobot dari *spanning tree* (b) merupakan yang minimum dari seluruh kemungkinan *spanning tree*. Maka (b) adalah MST dari (a).

Kita akan mempelajari 2 algoritma untuk mencari MST pada suatu graf berbobot, yaitu algoritma Prim dan algoritma Kruskal. Kedua algoritma ini bekerja dengan memanfaatkan konsep *greedy*.

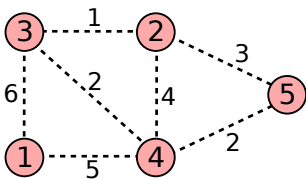
Prim

Cara kerja Prim serupa dengan cara kerja Dijkstra. Mula-mula, kita asumsikan seluruh *node* tidak terhubung satu sama lain. Setelah itu, kita akan memilih sembarang *node* sebagai *node* pertama pada *tree* yang akan kita bentuk. Kemudian, kita akan melakukan ekspansi pada *tree* solusi dengan cara memilih *node* satu per satu untuk digabungkan ke *tree* solusi, hingga seluruh *node* masuk ke dalam *tree*.

Untuk mengimplementasikan algoritma Prim, kita definisikan variabel berikut:

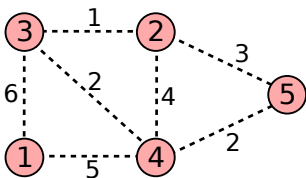
- $dist[x]$: bobot terkecil yang dapat dipakai untuk menghubungkan *node* x ke *tree*.
- $visited[x]$: bernilai *true* jika *node* x sudah masuk ke dalam *tree* solusi, dan *false* jika belum.

Umumnya, soal MST hanya meminta Anda mencari total bobot dari MST saja. Namun, jika Anda juga memerlukan *edge* solusi, maka Anda harus mendefinisikan variabel baru $pred[x]$ yang menyatakan tetangga dari *node* x yang menghubungkan x dengan *tree* solusi.



	1	2	3	4	5
<i>dist</i>	0	∞	∞	∞	∞
<i>visited</i>	<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>
<i>pred</i>	-1	-1	-1	-1	-1

Gambar 11.12: Tahap 1: Bentuk awal graf beserta isi tabel *dist* dan *visited*. Kita memulai dari *node* 1, maka tandai *visited*[1] = *true* serta *dist*[1] = 0.



	1	2	3	4	5
<i>dist</i>	0	∞	6	5	∞
<i>visited</i>	T	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>
<i>pred</i>	-1	-1	<u>1</u>	<u>1</u>	-1

Gambar 11.13: Tahap 2: Lakukan proses *relax* terhadap *node* 1. Selanjutnya memilih *node* yang belum dikunjungi dengan nilai *dist* terendah untuk ekspansi, yaitu *node* 4. Catatan: perubahan nilai pada tabel karena proses *relax* dicetak dengan warna biru dan digaris bawah.

Awalnya kita set nilai ini dengan -1 karena masih belum diketahui *node* mana yang menghubungkan x .

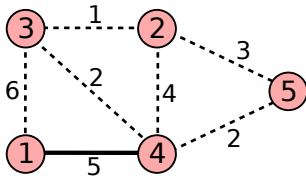
Sesuai definisi, pada mulanya seluruh *visited*[i] bernilai *false*. Selain itu, seluruh *node* x memiliki nilai *dist*[x] = ∞ . Kita mulai dengan mengambil sembarang *node* untuk dijadikan *node* pertama dari *tree* solusi. Untuk memudahkan, Anda dapat memilih *node* berindeks 1. Khusus untuk *node* pertama ini, kita juga set *dist*-nya menjadi 0. Untuk setiap *node* u yang mana u baru saja dimasukkan ke dalam *tree* solusi, maka kita akan melakukan proses *relax* sebagai berikut:

$$dist[v] = \min(dist[v], w[u][v]), \text{ untuk setiap } v \in adj(u) \text{ dan } visited[v] = false$$

Jika kita mengubah nilai *dist*[v], maka kita perlu mengubah nilai *pred*[v] = u , karena *edge* dengan bobot terkecil saat ini untuk menghubungkan v adalah dengan melalui u .

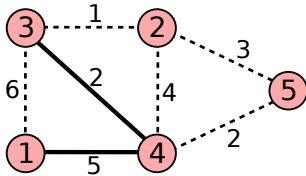
Kemudian, proses pemilihan ekspansi *tree* solusi dilakukan dengan memilih suatu *node* x yang mana *visited*[x] = *false* dan *dist*[x] minimum. Proses *relax* akan diulangi lagi hingga seluruh *node* masuk ke dalam solusi. Gambar 11.12 sampai Gambar 11.17 mengilustrasikan cara kerja Prim.

Setelah menjalankan algoritma Prim, kita dapat mengetahui total bobot dari MST dengan menghitung total nilai pada tabel *dist*. Pada kasus ini, total bobotnya adalah 10. Untuk mendapatkan *edge-edge* solusi MST, cukup lihat tabel *pred*. Untuk setiap *node* u , jika nilai *pred*[u] tidaklah -1 , maka *edge* $\langle u, pred[u] \rangle$ masuk dalam solusi MST. Pada kasus



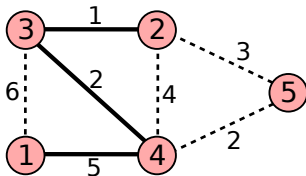
	1	2	3	4	5
<i>dist</i>	0	4	<u>2</u>	5	<u>2</u>
<i>visited</i>	T	F	F	T	F
<i>pred</i>	-1	4	4	1	4

Gambar 11.14: Tahap 3: Lakukan proses *relax* terhadap *node* 4. Selanjutnya memilih *node* yang belum dikunjungi dengan nilai *dist* terendah untuk ekspansi. Ada 2 kandidat, kita bebas memilih yang mana saja. Misal kita pilih *node* 3.



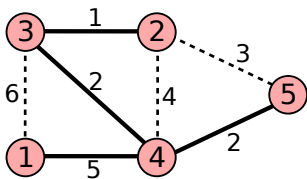
	1	2	3	4	5
<i>dist</i>	0	<u>1</u>	2	5	2
<i>visited</i>	T	F	T	T	F
<i>pred</i>	-1	3	4	1	4

Gambar 11.15: Tahap 4: Lakukan proses *relax* terhadap *node* 3. Selanjutnya memilih *node* yang belum dikunjungi dengan nilai *dist* terendah untuk ekspansi, yaitu *node* 2.



	1	2	3	4	5
<i>dist</i>	0	1	2	5	2
<i>visited</i>	T	T	T	T	F
<i>pred</i>	-1	3	4	1	4

Gambar 11.16: Tahap 5: Lakukan proses *relax* terhadap *node* 2. Selanjutnya memilih *node* yang belum dikunjungi dengan nilai *dist* terendah untuk ekspansi, yaitu *node* 5.



	1	2	3	4	5
<i>dist</i>	0	1	2	5	2
<i>visited</i>	<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>	T
<i>pred</i>	-1	3	4	1	4

Gambar 11.17: Tahap 6: Lakukan proses *relax* terhadap *node* 5. Karena semua nilai pada *visited* sudah *true*, maka algoritma selesai.

ini, *edge* yang masuk dalam solusi MST adalah $\{(1,3), (2,4), (5,1), (2,4)\}$.

Kompleksitas Prim

Mari kita analisa kompleksitas algoritma Prim. Pada mulanya, semua *node* kecuali *node* awal tidak termasuk di dalam *tree*. Pada setiap iterasi pemilihan *node*, tepat satu buah *node* akan masuk ke dalam *tree*. Dengan demikian, akan ada tepat $O(V)$ pemilihan *node*. Jika pemilihan diimplementasikan secara naif dengan mencari *node* tersebut secara linear, maka kompleksitas total pemilihan *node* adalah $O(V^2)$. Pada setiap pemilihan *node*, terjadi proses *relax*. Secara total, setiap *edge* akan mengalami *relax* sebanyak 2 kali, sehingga kompleksitas akhir algoritma Prim adalah $O(V^2 + E)$.

Serupa dengan Dijkstra, kita dapat melakukan optimasi pemilihan *edge* dengan memanfaatkan struktur data *heap*, sehingga setiap operasi pemilihan *edge* memiliki kompleksitas $O(\log V)$. Dengan demikian, kompleksitas total algoritma Prim adalah $O((V + E) \log V)$.

Implementasi

Secara implementasi, Algoritma Prim sangat serupa dengan algoritma Dijkstra. Perbedaan mendasarnya adalah pada setiap pengunjungan *node*, kita tidak menghitung jarak total dari *node* awal, melainkan hanya bobot pada *edge*-nya saja. Algoritma 69 merupakan prosedur untuk mencetak *edge-edge* solusi.

►► Sekilas Info ◀◀

Dengan menguasai algoritma Dijkstra terlebih dahulu, Anda akan dapat mempelajari algoritma Prim dengan relatif mudah.

Kruskal

Algoritma Kruskal merupakan algoritma alternatif untuk mencari MST. Kruskal memulai dengan masing-masing *node* membentuk *tree* tersendiri, kemudian menggabungkan *tree-tree* tersebut dengan terus menambahkan *edge* berbobot terkecil, sampai seluruh *node* pada graf terhubung dengan sebuah *tree*.

Algoritma 68 Implementasi algoritma Prim dengan optimisasi *heap*.

```

1: function PRIM(s)
2:   cost ← 0
3:   dist ← new integer[V + 1]
4:   visited ← new boolean[V + 1]
5:   pred ← new integer[V + 1]
6:   FILLARRAY(dist, ∞)
7:   FILLARRAY(visited, false)
8:   FILLARRAY(pred, -1)

9:   INITIALIZEHEAP()                                ▷ Buat sebuah min-heap
10:  PUSH((0, s))                                     ▷ Masukkan node s dengan jarak tempuh 0 ke dalam heap
11:  dist[s] ← 0
12:  while not ISEMPTYHEAP() do
13:    (curDist, u) ← POP()                            ▷ Dapatkan node yang dapat digabungkan ke
                                                    tree solusi dengan bobot edge terkecil
14:    if not visited[u] then                          ▷ Hanya proses apabila u belum termasuk pada tree
solusi
15:      visited[u] ← true
16:      cost ← cost + dist[u]                         ▷ Tambahkan bobot edge yang menghubungkan
                                                    tree solusi terhadap u ke dalam solusi
17:      for v ∈ adj(u) do                             ▷ Lakukan relax untuk semua tetangga u
18:        if (dist[v] > w[u][v]) ∧ (not visited[v]) then
19:          dist[v] = w[u][v]
20:          pred[v] = u
21:          PUSH((dist[v], v))                       ▷ Masukkan bobot edge dan node yang
                                                    diperbaharui ke heap
22:        end if
23:      end for
24:    end if
25:  end while
26:  return cost
27: end function

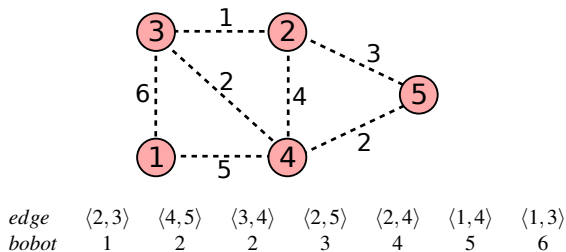
```

Algoritma 69 Mencetak *edge-edge* yang termasuk dalam solusi MST.

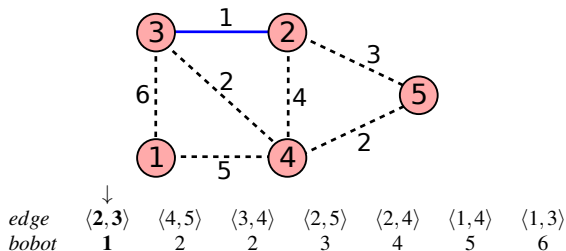
```

1: procedure PRINTPATH
2:   for i ← 1, V do
3:     if not pred[i] = -1 then
4:       print pred[i], i
5:     end if
6:   end for
7: end procedure

```



Gambar 11.18: Tahap 1: Bentuk awal graf. Kita urutkan *edge* berdasarkan bobotnya secara menaik seperti yang tertera pada tabel.



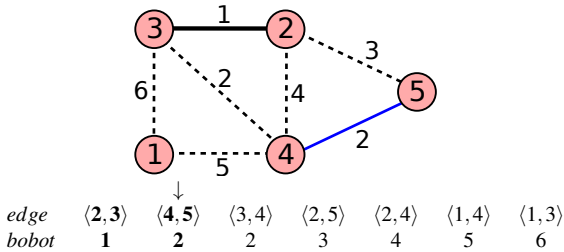
Gambar 11.19: Tahap 2: Kita coba *edge* $\langle 2,3 \rangle$. Karena menambahkan *edge* tersebut tidak membentuk siklus, kita ambil sebagai solusi. Untuk kemudahan, solusi ditandai dengan *edge* yang dicetak tebal.

Diberikan graf dengan V *node* dan E *edge*. Mula-mula, urutkan seluruh *edge* tersebut berdasarkan bobotnya secara menaik. Kita akan memilih *edge-edge* untuk ditambahkan sebagai solusi dari MST, yang pada awalnya masih kosong. Untuk setiap *edge* e , mulai dari yang memiliki bobot terkecil: Jika penambahan *edge* e ke dalam solusi MST tidak membentuk siklus, tambahkan *edge* tersebut sebagai solusi MST. Untuk lebih jelasnya, perhatikan contoh pada Gambar 11.18 sampai Gambar 11.26.

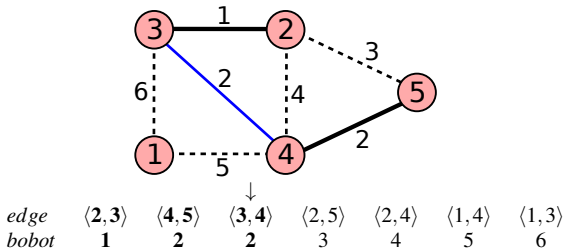
Kompleksitas Kruskal

Bagaimana caranya mengecek apakah dengan menambah suatu *edge*, akan menciptakan siklus? Solusi paling sederhana adalah dengan melakukan DFS atau BFS. Karena ada E buah *edge* yang akan dicek, dan setiap pengecekan memerlukan waktu $O(E)$, maka kompleksitasnya adalah $O(E^2)$.

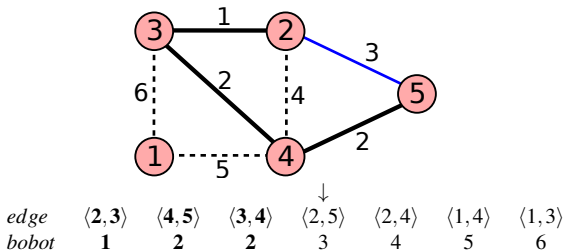
Agar lebih efisien, kita dapat memanfaatkan struktur data *disjoint set*. Mula-mula, kita siapkan N buah set yang masing-masing berisi satu *node* pada graf. Suatu *edge* yang menghubungkan *node* x dan y dapat diambil jika x dan y berada pada set yang berbeda. Jika kita mengambil *edge* tersebut, kita gabungkan kedua set yang mengandung x dan y . Jika x dan y berada pada set yang sama, artinya x dan y sudah tergabung pada *tree* yang sama. Dengan demikian, jika kita mengambil *edge* baru yang menghubungkan x dan y , maka



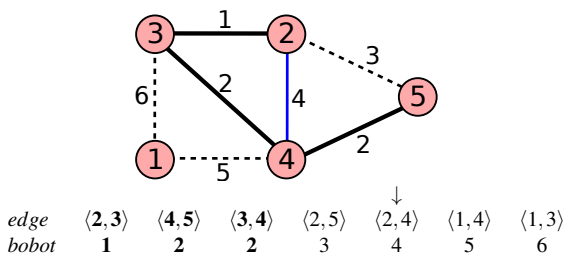
Gambar 11.20: Tahap 3: Berikutnya kita coba *edge* $\langle 4,5 \rangle$. Karena menambahkan *edge* tersebut tidak membentuk siklus, kita ambil sebagai solusi.



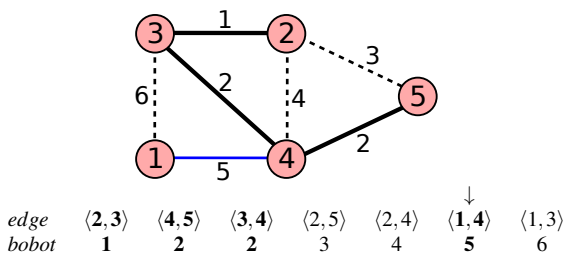
Gambar 11.21: Tahap 4: Berikutnya kita coba *edge* $\langle 3,4 \rangle$. Karena menambahkan *edge* tersebut tidak membentuk siklus, kita ambil sebagai solusi.



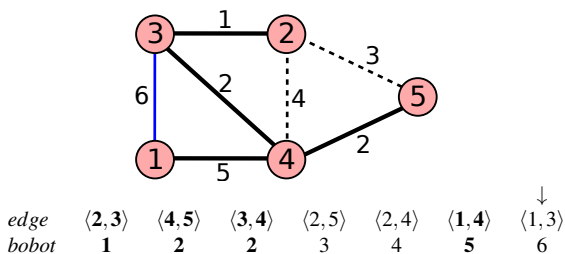
Gambar 11.22: Tahap 5: Berikutnya kita coba *edge* $\langle 2,5 \rangle$. Ternyata menambahkan *edge* tersebut akan membentuk siklus. *Edge* tersebut tidak dijadikan solusi.



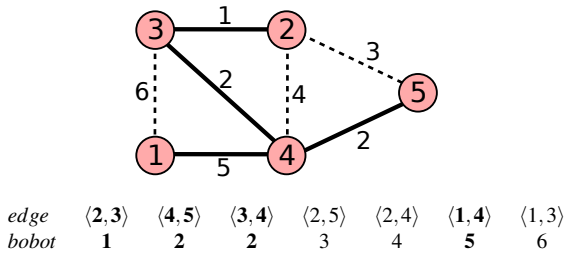
Gambar 11.23: Tahap 6: Berikutnya kita coba *edge* $\langle 2,4 \rangle$. Ternyata menambahkan *edge* tersebut akan membentuk siklus. *Edge* tersebut tidak dijadikan solusi.



Gambar 11.24: Tahap 7: Berikutnya kita coba *edge* $\langle 1,4 \rangle$. Karena menambahkan *edge* tersebut tidak membentuk siklus, kita ambil sebagai solusi.



Gambar 11.25: Tahap 8: Berikutnya kita coba *edge* $\langle 1,3 \rangle$. Ternyata menambahkan *edge* tersebut akan membentuk siklus. *Edge* tersebut tidak dijadikan solusi.



Gambar 11.26: Tahap 9: Semua *edge* sudah diperiksa, algoritma selesai.

dijamin akan menghasilkan siklus. Maka, kompleksitas akhir Kruskal adalah $O(E \log E)$.

implementasi

Algoritma 70 Implementasi algoritma Kruskal dengan optimisasi *disjoint set*.

```

1: function KRUSKAL(edgeList)
2:   INITIALIZEDISJOINTSET()
3:   SORT(edgeList)
4:   for  $\langle u, v \rangle \in \textit{edgeList}$  do
5:     if not CHECK(u, v) then
6:        $\textit{cost} \leftarrow \textit{cost} + w[u][v]$ 
7:       JOIN(u, v)
8:     end if
9:   end for
10:  return cost
11: end function

```

▷ Urutkan berdasarkan bobotnya

Implementasi algoritma Kruskal cukup sederhana, dan dapat dilihat pada Algoritma 70.

12 Dasar-Dasar Geometri

Geometri merupakan materi yang dapat yang diujikan pada pemrograman kompetitif. Untuk menyelesaikan soal yang memiliki unsur geometri, diperlukan kemampuan untuk memodelkan bidang spasial dan menghitung sifat-sifat objek spasial seperti jarak, luas, volume, bentuk, dan sebagainya. Bab ini mengulas dasar-dasar geometri dan cara memodelkannya pada pemrograman.

Titik

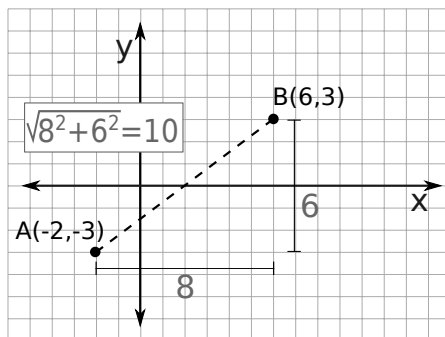
Titik merupakan elemen mendasar pada dunia geometri. Titik dapat didefinisikan pada 2 dimensi, 3 dimensi, atau lebih.

Pada bidang 2 dimensi, titik dapat dianggap berada pada suatu koordinat (x, y) . Representasi titik pada program dapat diwujudkan dengan tipe data komposit yang menyimpan nilai x dan y . Tipe data ini misalnya **record** (Pascal), **struct** (C) atau **class** (C++/Java). Alternatif lainnya adalah menggunakan tipe data untuk merepresentasikan *tuple*, apabila bahasa pemrograman yang Anda gunakan mendukungnya.

Jarak Euclidean

Salah satu persoalan yang muncul pada geometri komputasional adalah menghitung jarak antara 2 titik. **Jarak Euclidean** adalah definisi jarak yang umum digunakan pada bidang 2 dimensi atau ruang 3 dimensi. Jarak pada bidang 2 dimensi dari dua titik $A(x, y)$ dan $B(x, y)$ adalah:

$$\text{dist}(A, B) = \sqrt{(A.x - B.x)^2 + (A.y - B.y)^2}$$



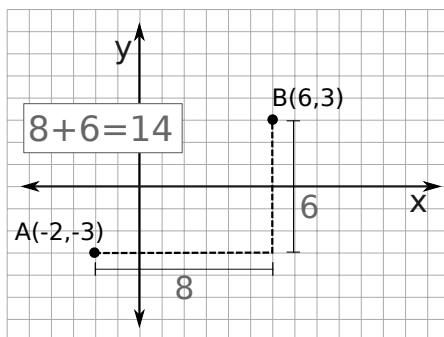
Gambar 12.1: Jarak Euclidean dari titik A dan B.

Sebagai contoh, jika diberikan titik $(-2, -3)$ dan $(6, 3)$, jarak euclidean dari kedua titik tersebut adalah 10.

Jarak Manhattan

Jarak Manhattan adalah definisi jarak lain yang juga biasa dipakai. Jarak Manhattan dihitung dari selisih antara jarak masing-masing sumbu secara independen. Pada bidang 2 dimensi, jarak Manhattan 2 titik adalah:

$$\text{dist}(A, B) = |A.x - B.x| + |A.y - B.y|$$



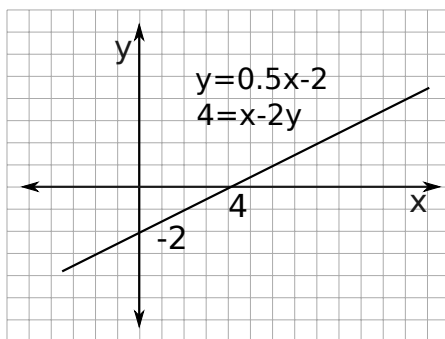
Gambar 12.2: Jarak Manhattan dari titik A dan B.

Sebagai contoh, jika diberikan titik $(-2, -3)$ dan $(6, 3)$, jarak manhattan dari kedua titik tersebut adalah 14.

Garis

Garis merupakan himpunan seluruh titik (x, y) , yang memenuhi suatu persamaan $Ax + By = C$, dengan A , B , dan C merupakan suatu bilangan riil. Bentuk lain dari persamaan garis yang umum adalah $y = mx + c$, dengan m dan c suatu bilangan riil. Pada representasi ini, m disebut juga dengan **gradien**, sementara c disebut konstanta. Perlu diperhatikan bahwa garis memiliki panjang yang tidak terbatas.

Untuk kemudahan, mari kita sebut representasi $Ax + By = C$ sebagai representasi (A, B, C) dan representasi $y = mx + c$ sebagai (m, c) .



Gambar 12.3: Contoh garis pada bidang 2 dimensi.

Untuk merepresentasikan garis, kita dapat membuat kelompok data yang menyimpan nilai $\langle A, B, C \rangle$, atau $\langle m, c \rangle$, bergantung pada persamaan yang Anda gunakan. Apabila ditelusuri, kedua persamaan ini sebenarnya berkaitan:

$$\begin{aligned} Ax + By &= C \\ By &= C - Ax \\ y &= \frac{C}{B} - \frac{A}{B}x \\ y &= \left(-\frac{A}{B}\right)x + \frac{C}{B} \end{aligned}$$

Jadi $m = -\frac{A}{B}$ dan $c = \frac{C}{B}$.

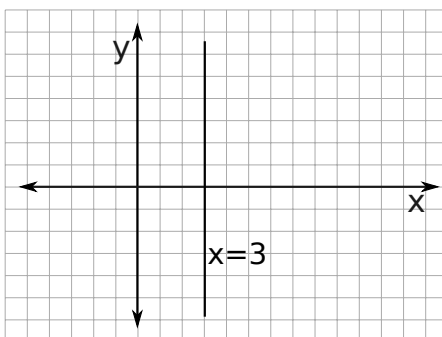
Garis Vertikal

Hati-hati saat merepresentasikan garis vertikal, misalnya $x = 3$ seperti pada gambar 12.4. Representasi $\langle A, B, C \rangle$ dapat merepresentasikannya, yaitu dengan $A = 1, B = 0, C = 3$.

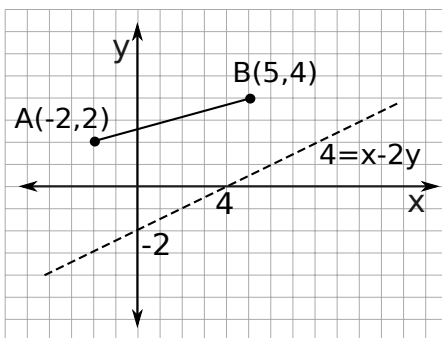
Sementara representasi $\langle m, c \rangle$ memiliki kesulitan, karena nilai m yang tidak terdefinisi:

$$\begin{aligned} m &= -\frac{A}{B} \\ &= -\frac{1}{0} \end{aligned}$$

Untuk kasus yang mungkin terdapat garis vertikal, representasi $\langle A, B, C \rangle$ lebih disarankan.



Gambar 12.4: Contoh garis vertikal



Gambar 12.5: Garis tegas menyatakan segmen garis, dan garis putus-putus menyatakan garis.

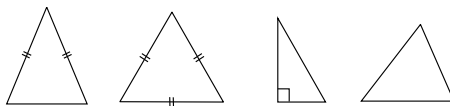
Segmen Garis

Segmen garis merupakan garis yang terdefinisi dari suatu titik (x_1, y_1) ke titik (x_2, y_2) . Berbeda dengan garis, segmen garis memiliki panjang yang berhingga, yaitu terbatas di ujung-ujungnya saja. Segmen garis dapat direpresentasikan dengan dua titik, yaitu kedua ujungnya.

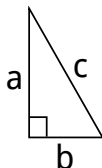
Anda dapat menggunakan rumus jarak euclidean antara dua titik ujung segmen garis untuk mengetahui panjang dari segmen garis tersebut.

Segitiga

Segitiga merupakan bangun 2 dimensi yang paling mendasar. Segitiga terdiri dari 3 titik sudut, yang mana antar titik sudut terhubung oleh segmen garis. Berdasarkan



Gambar 12.6: Contoh segitiga berturut-turut dari kiri ke kanan: segitiga sama kaki, sama sisi, siku-siku, dan sembarang.



Gambar 12.7: Segitiga siku-siku dengan panjang sisi tegak lurus a dan b , serta panjang sisi miring c .

panjang segmen garisnya, segitiga dapat berupa segitiga sama kaki, sama sisi, siku-siku, atau sembarang.

Teorema Pythagoras

Pada segitiga siku-siku, kita dapat mengetahui panjang sisi miringnya menggunakan rumus Teorema Pythagoras.

Misalkan kedua sisi yang tegak lurus dari suatu segitiga siku-siku memiliki panjang a dan b , sementara panjang sisi miringnya adalah c . Menurut Teorema Pythagoras, berlaku hubungan:

$$a^2 + b^2 = c^2$$

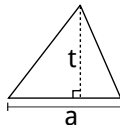
Rumus tersebut dapat digunakan untuk menghitung panjang suatu sisi segitiga siku-siku jika diketahui panjang kedua sisi lainnya.

Luas Segitiga

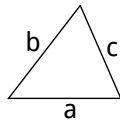
Rumus klasik dari luas segitiga dengan panjang alas a dan tinggi t adalah:

$$L = \frac{a \times t}{2}$$

Ada kalanya kita tidak tahu tinggi dari segitiga, sehingga rumus sebelumnya tidak dapat digunakan. Alternatif lain adalah menggunakan **rumus Heron**, yaitu:



Gambar 12.8: Tinggi dan alas dari segitiga yang dapat digunakan untuk pencarian luas.



Gambar 12.9: Segitiga sembarang dengan panjang sisi a , b , dan c .

$$L = \sqrt{s(s-a)(s-b)(s-c)}$$

$$s = \frac{a+b+c}{2}$$

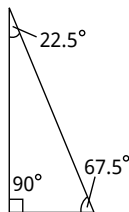
Sudut Segitiga

Jumlah sudut-sudut pada suatu segitiga adalah 180° .

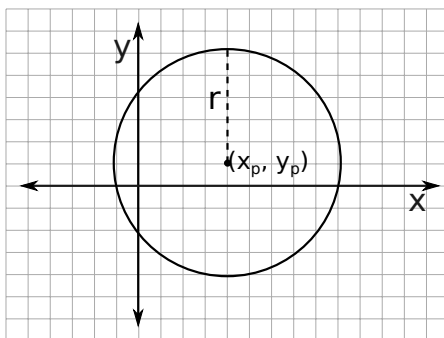
Sifat yang mirip juga ditemukan pada segi- N . Untuk segi- N , jumlah sudut-sudutnya adalah $180(N-2)^\circ$. Misalnya untuk segi-4, jumlah sudut-sudutnya adalah $180(4-2)^\circ = 360^\circ$

Lingkaran

Lingkaran dengan titik pusat (x_p, y_p) dan jari-jari r merupakan himpunan seluruh titik (x, y) yang memenuhi $(x-x_p)^2 + (y-y_p)^2 = r^2$. Jari-jari lingkaran biasa juga disebut dengan **radius**.



Gambar 12.10: Jumlah sudut segitiga.



Gambar 12.11: Contoh lingkaran dengan titik pusat di (x_p, y_p) dan jari-jari r .

Beberapa properti lingkaran yang umum:

- Luas dari lingkaran dengan jari-jari r adalah πr^2 .
- Keliling dari lingkaran dengan jari-jari r adalah $2\pi r$.
- Diameter lingkaran merupakan dua kali jari-jari, atau $2r$.

Konstanta π (pi) merupakan konstanta dalam matematika yang dapat digunakan dalam pencarian luas atau keliling lingkaran.

►► Sekilas Info ◀◀

Hati-hati! Nilai π **tidak sama dengan** $\frac{22}{7}$. Penggunaan $\frac{22}{7}$ hanyalah aproksimasi dari nilai π yang sesungguhnya.

$$\pi = 3.14159265359..$$

$$\frac{22}{7} = 3.14285714285..$$

Untuk perhitungan pada kompetisi pemrograman, Anda dapat menggunakan nilai π yang diberikan pada soal. Jika soal tidak memberikan nilai π yang digunakan, Anda dapat mencarinya dengan fungsi *arc-cosinus*, yaitu fungsi pada trigonometri. Nilai π didapatkan dari:

- Pascal: $\arccos(-1)$
- C++: $\text{acos}(-1)$ (memerlukan include `cmath`)

Presisi Bilangan Riil

Pada kompetisi pemrograman, disarankan selalu mengutamakan penggunaan bilangan bulat daripada bilangan riil. Hal ini dikarenakan komputer memiliki keterbatasan dalam

merepresentasikan bilangan riil. Representasi bilangan riil dari angka 1 bisa jadi adalah 0.999999999999 atau 1.00000000001. Akibatnya, perhitungan yang melibatkan bilangan riil kurang akurat jika dibandingkan dengan perhitungan dengan bilangan bulat. Masalah lainnya adalah dalam hal memeriksa kesamaan dua bilangan. Nilai dari ekspresi $(1.0/3 * 3 = 3.0)$ mungkin saja mengembalikan *false*.

Sebagai ilustrasi, perhatikan contoh soal berikut.

Contoh Soal 12.1: Potongan Pizza

Pak Dengklek sedang memesan pizza pada suatu restoran. Ia menemukan dua buah menu pizza. Pizza pertama memiliki jari-jari A inci dan akan dibagi menjadi B potong sama rata. Sementara pizza kedua memiliki jari-jari C inci dan akan dibagi menjadi D potong sama rata. Pak Dengklek penasaran, apakah satu potong pada pizza pertama memiliki luas yang sama dengan satu potong pada pizza kedua. Bantulah Pak Dengklek mencari jawabannya!

Algoritma 71 Solusi Potongan Pizza versi 1.

```
1: function CHECKPIZZA(A, B, C, D)
2:   return  $(\pi \times A \times A) / B = (\pi \times C \times C) / D$ 
3: end function
```

Soal Potongan Pizza ini cukup sederhana. Solusi yang mudah terpikirkan adalah dengan menghitung luas masing-masing potongan pizza, lalu melakukan perbandingan seperti pada Algoritma 71.

Algoritma 72 Solusi Potongan Pizza versi 2.

```
1: function CHECKPIZZA(A, B, C, D)
2:   return  $(A \times A) / B = (C \times C) / D$ 
3: end function
```

Namun, π muncul pada kedua ruas di Algoritma 71. Kita dapat menghilangkan π sehingga didapatkan perbandingan seperti pada Algoritma 72.

Algoritma 73 Solusi Potongan Pizza versi 3.

```
1: function CHECKPIZZA(A, B, C, D)
2:   return  $A \times A \times D = C \times C \times B$ 
3: end function
```

Terakhir, kita dapat pindahkan B di ruas kiri serta D di ruas kanan untuk mendapatkan perbandingan pada Algoritma 73. Meskipun ketiga algoritma ini sama secara matematis, Algoritma 73 lebih aman dibandingkan 2 algoritma lainnya. Pada Algoritma 73, tidak ada pembagian sama sekali, juga tidak ada pemakaian nilai π , sehingga perhitungan dapat dilakukan pada bilangan bulat.

Jika terpaksa menggunakan bilangan riil, seperti `real` atau `double`, Anda perlu menggunakan teknik khusus untuk memeriksa kesamaan dua bilangan riil. Caranya adalah mengambil selisih mutlak dari kedua bilangan tersebut, lalu bandingkan hasilnya dengan suatu nilai toleransi yang sangat kecil seperti 10^{-9} . Nilai toleransi ini biasa disebut dengan *epsilon* (ϵ). Apabila selisih mutlaknya lebih kecil dari nilai toleransi, maka kedua bilangan tersebut dapat dianggap sama.

Algoritma 74 Memeriksa kesamaan bilangan riil a dan b dengan nilai toleransi ϵ .

```
1: function ISEQUAL( $a, b$ )
2:   return ABS( $a - b$ ) <  $\epsilon$ 
3: end function
```

Pada Algoritma 74, ABS merupakan fungsi harga mutlak. Fungsi $ABS(x)$ mengembalikan nilai $-x$ apabila x bernilai negatif, dan x pada kasus lainnya. Kita dapat menggunakan ISEQUAL untuk memeriksa apakah $a \leq b$ atau $a < b$, untuk bilangan riil a dan b , seperti yang ditunjukkan pada Algoritma 75 dan Algoritma 76.

Algoritma 75 Memeriksa apakah $a \leq b$ untuk tipe data bilangan riil.

```
1: function ISLESSEQ( $a, b$ )
2:   return ISEQUAL( $a, b$ )  $\vee$  ( $a < b$ )
3: end function
```

Algoritma 76 Memeriksa apakah $a < b$ untuk tipe data bilangan riil.

```
1: function ISLESS( $a, b$ )
2:   return (not ISEQUAL( $a, b$ ))  $\wedge$  ( $a < b$ )
3: end function
```

13 Persiapan Kompetisi

Menguasai seluruh kompetensi dasar pemrograman dan algoritma saja belum tentu cukup untuk memenangkan kompetisi. Terdapat faktor-faktor lain yang berperan saat kompetisi berlangsung, mulai dari peraturan kompetisi sampai kondisi fisik dan mental Anda. Penguasaan terhadap faktor-faktor ini akan membantu Anda dalam mempersiapkan diri lebih matang menjelang kompetisi.

Pengenalan Medan

Mengenali kondisi kompetisi yang Anda ikuti adalah hal yang penting. Kondisi ini antara lain:

- **Aturan kompetisi.**
Pastikan peraturan kontes yang akan berlangsung telah dipahami. Termasuk diantaranya perhitungan poin dan penentuan juara. Sebagai contoh, jika penilaian poin dilakukan berdasarkan aturan IOI, kita tidak perlu cepat-cepat dalam mengerjakan soal. Selain itu kita tidak perlu langsung memikirkan solusi penuh dari soal yang diberikan. Strategi ini tidak berlaku jika perhitungan poin dilakukan berdasarkan aturan ACM-ICPC, karena adanya risiko penalti waktu.
- **Waktu yang diberikan.**
Pada umumnya, durasi kompetisi berkisar antara 1 sampai 5 jam. Perlu diketahui bahwa strategi untuk berkompetisi selama 5 jam belum tentu bekerja dengan efektif untuk kompetisi berdurasi 3 jam.
- **Sumber daya yang ada.**
Anda juga perlu memperhatikan sumber daya saat berkompetisi, berupa:
 - *Compiler* yang disediakan.
 - *Text editor* yang disediakan.
 - Sistem operasi yang disediakan.
 - Kemampuan mesin.

Merupakan suatu keharusan untuk mampu mengkompilasi program dengan *compiler* yang diberikan, menggunakan *text editor* yang disediakan, dan terbiasa dengan sistem operasi yang ada. Apabila kompetisi yang Anda ikuti menyelenggarakan sesi uji coba, manfaatkan kesempatan ini untuk membiasakan diri dengan sumber daya tersebut.

Informasi kemampuan mesin dapat digunakan untuk memprediksi *running time*. Kadang-kadang, kemampuan mesin *server* untuk melakukan *grading* memiliki kecepatan yang jauh lebih besar (atau bahkan kecil!) dari mesin pada umumnya. Apabila disediakan, Anda dapat membaca informasi kecepatan CPU *server* penyelenggara lomba.

Persiapan Sebelum Kompetisi

Dalam kompetisi pemrograman, kemampuan seseorang dapat dibagi menjadi beberapa komponen, yaitu dasar ilmu yang dimiliki, kemampuan berpikir logis dan kreativitas, serta

penguasaan medan laga. Seluruh komponen tersebut perlu diasah, dan kekurangan atas suatu komponen mempengaruhi performa berkompetisi. Sebagai contoh, seseorang yang mengetahui banyak teori dan mampu mengaplikasikan logika berpikir dalam menyelesaikan masalah bisa jadi tidak menang, karena gugup saat kompetisi.

Dasar ilmu dapat diperkaya dengan cara menghadiri sesi pembelajaran, membaca buku, atau membaca artikel. Untuk mempelajari lebih dalam tentang pemrograman kompetitif, kami menyarankan untuk membaca buku *Competitive Programming*.⁴

Untuk mengasah kemampuan berpikir logis dan kreativitas, latihan secara berkala perlu dilakukan. Latihan dapat dilakukan di situs-situs seperti:

- TLX Training Gate (<https://tlx.toki.id/training>), yaitu situs yang dibuat oleh Ikatan Alumni Tim Olimpiade Komputer Indonesia (IA TOKI) dengan silabus dan materi yang disesuaikan dengan Olimpiade Sains Nasional bidang komputer atau informatika. Selain berisi materi pembelajaran, situs ini berisi arsip soal kompetisi-kompetisi tingkat nasional di Indonesia.
- TLX Competition Gate (<https://tlx.toki.id/competition>), yaitu situs yang juga dibuat oleh IA TOKI yang menyelenggarakan kontes lokal secara berkala.
- Codeforces (<https://codeforces.com/>), situs mirip seperti TLX Competition Gate dengan cakupan internasional.
- SPOJ (<https://www.spoj.com/>), situs berisi kumpulan soal dari berbagai negara.
- COCI (<http://www.hsin.hr/coci>), situs dari negara Kroasia yang berisi soal-soal bermutu tinggi dan kontes secara periodik.

Mampu menyelesaikan soal setingkat kompetisi yang Anda ikuti saja tidak cukup. Anda harus dapat menyelesaikannya dalam situasi yang serupa saat kompetisi. Sebab, mengerjakan soal pada situasi latihan dan situasi kompetisi sangat berbeda. Terdapat tekanan mental yang lebih banyak saat kompetisi berlangsung. Menyimulasikan suasana kompetisi ini membantu dalam persiapan mental untuk menguasai medan laga. Sebagai contoh, untuk kompetisi OSN, buatlah simulasi dengan memilih 3 buah soal, kemudian kerjakan sebaik-baiknya selama 5 jam tanpa bantuan.

Kumpulan Tips Berkompetisi

Berikut adalah tips sebelum berkompetisi:

- Beristirahatlah pada hari sebelum pertandingan agar pikiran Anda tenang.
- Jika Anda sudah melakukan persiapan yang matang jauh-jauh hari sebelumnya, tidak dianjurkan untuk begadang dan berusaha untuk belajar lagi. Percayakan pada pengalaman Anda selama persiapan untuk berkompetisi esok harinya.

Kemudian berikut adalah tips saat berkompetisi:

- Disarankan untuk membaca semua soal terlebih dahulu sebelum memulai mengerjakan.
- Alokasikan waktu untuk memikirkan perkiraan solusi untuk setiap soal. Jangan sampai ada penyesalan di akhir karena kehabisan waktu untuk mengerjakan soal lain, sementara ada soal lainnya yang ternyata bisa Anda kerjakan, luput dari perhatian.
- Ketika menemukan suatu algoritma untuk menyelesaikan soal, luangkan waktu beberapa menit untuk memikirkan kembali algoritma tersebut matang-matang. Hal ini untuk menghindari membuang-buang waktu karena Anda baru sadar ada kesalahan

algoritma saat Anda sudah mulai menulis program.

- Untuk kontes berdurasi 5 jam, beristirahatlah sejenak jika merasa lelah, karena ketika lelah, konsentrasi akan berkurang. Sadari bahwa 5 jam merupakan waktu yang panjang.
- Atur makanan Anda saat dan sebelum kontes. Jangan sampai terlalu lapar atau terlalu kenyang.
- Untuk kompetisi dengan gaya IOI, sangat dianjurkan untuk mengerjakan semua soal meskipun tidak mendapatkan nilai sempurna. Anda bisa mendapatkan beberapa nilai tambahan dengan mengerjakan subsoal yang diberikan.

Bibliografi

- ¹ Ashar Fuadi. What is competitive programming? <https://www.quora.com/What-is-competitive-programming-2/answer/Ashar-Fuadi>. Diakses: 2017-12-25.
- ² ioi 2017 contest rules. <http://ioi2017.org/contest/rules/>. Diakses: 2017-12-24.
- ³ World finals rules. <https://icpc.baylor.edu/worldfinals/rules>. Diakses: 2017-12-24.
- ⁴ Steven Halim and Felix Halim. *Competitive Programming*. 3rd Edition. 2013.
- ⁵ Brilliant.org. Extended euclidian algorithm. <https://brilliant.org/wiki/extended-euclidean-algorithm/>. Diakses: 2018-01-12.
- ⁶ M Perz. Thomas koshy, "elementary number theory with applications", 2002. *Smarandache Notions Journal*, 13:284–285, 2002.
- ⁷ Paul Pritchard. Linear prime-number sieves: A family tree. *Science of computer programming*, 9(1):17–35, 1987.
- ⁸ R. A. Mollin. *Fundamental Number Theory with Applications*. 2nd ed. Boca Raton: Chapman & Hall/CRC, 2008.
- ⁹ Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 6th edition, 2007.
- ¹⁰ Brian C. Dean. A simple expected running time analysis for randomized "divide and conquer" algorithms. *Discrete Applied Mathematics*, 154(1):1 – 5, 2006.
- ¹¹ M Perz. Uva 10003 - cutting sticks. https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=944. Diakses: 2017-12-25.
- ¹² Rebecca Fiebrink. Amortized analysis explained (pdf). 2007. Diakses: 2018-01-18.
- ¹³ Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
- ¹⁴ Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, March 1984.
- ¹⁵ Marek A. Suchenek. Elementary yet precise worst-case analysis of floyd's heap-construction program. *Fundam. Inf.*, 120(1):75–92, January 2012.

Tentang Penulis

William Gozali

William Gozali mendapatkan medali perak pada Olimpiade Sains Nasional Bidang Informatika tahun 2009 dan 2010. Ia kemudian menjadi perwakilan Indonesia dalam ajang International Olympiad of Informatics tahun 2011 di Thailand dan berhasil memperoleh medali perunggu. Semasa kuliah, ia mengikuti ACM-ICPC di berbagai regional dan berhasil menjadi salah satu finalis ACM-ICPC World Finals tahun 2014 dan 2015.

Dari tahun 2011 sampai 2014, William aktif menjadi anggota *scientific committee* TOKI yang bertugas merumuskan dan melaksanakan Pelatihan Nasional TOKI. Ia kemudian mengetuainya pada tahun 2015. Selepas itu, ia aktif menulis materi pembelajaran TOKI pada situs TLX.

Alham Fikri Aji

Alham Fikri Aji merupakan peroleh medali perak pada Olimpiade Sains Nasional Bidang Informatika tahun 2008. Ia kemudian berhasil menjadi perwakilan Indonesia dalam ajang International Olympiad of Informatics tahun 2010 di Kanada dan mendapatkan medali perak untuk Indonesia. Ia juga menjadi salah satu finalis di ACM-ICPC World Finals tahun 2014.

Alham aktif berkontribusi sebagai penulis soal di berbagai kompetisi. Ia menjadi tim penulis soal OSN sejak tahun 2013 hingga 2017 dan penulis soal di ACM-ICPC dari tahun 2014 sampai 2017. Selain itu, Alham juga menjadi salah satu tim juri di ACM-ICPC Regional Asia Jakarta pada tahun 2015 dan 2017.

Setelah bertahun-tahun menulis materi pembelajaran, sejumlah anggota dari Ikatan Alumni Tim Olimpiade Komputer Indonesia dan kontributor lainnya bertekad untuk membukukan materi-materi tersebut. Buku ini disusun sebagai panduan untuk memulai pembelajaran kompetisi pemrograman, terutama Olimpiade Sains Nasional (OSN) bidang komputer/informatika, dan ditulis langsung dari mereka yang pernah menjuarai kompetisi pemrograman nasional maupun internasional.

Penulisan buku ini disajikan secara sistematis sehingga dapat digunakan untuk berbagai kalangan, mulai dari siswa sekolah menengah hingga mahasiswa perguruan tinggi yang hendak mengikuti kompetisi pemrograman lokal atau regional.

Versi terkini bisa didapatkan di:

<https://toki.id/buku-pemrograman-kompetitif-dasar>

