Chapter 2
Time complexity

Efisiensi algoritma sangat penting dalam competitive programming. Biasanya, mudah untuk merancang algoritma yang menyelesaikan masalah dengan lambat, tetapi tantangan sebenarnya adalah menemukan algoritma yang cepat. Jika algoritma terlalu lambat, ia hanya akan mendapatkan sebagian poin atau bahkan tidak mendapatkan poin sama sekali.

Kompleksitas waktu suatu algoritma memperkirakan seberapa banyak waktu yang akan digunakan algoritma untuk suatu input tertentu. Idenya adalah merepresentasikan efisiensi sebagai sebuah fungsi dengan parameter yang merupakan ukuran input. Dengan menghitung kompleksitas waktu, kita dapat menentukan apakah algoritma cukup cepat tanpa harus mengimplementasikannya terlebih dahulu.

Aturan Perhitungan

Kompleksitas waktu suatu algoritma dinotasikan sebagai $O(\cdots)$, di mana tiga titik tersebut mewakili suatu fungsi. Biasanya, variabel n digunakan untuk menyatakan ukuran input. Misalnya, jika input berupa array angka, maka n adalah ukuran array tersebut, dan jika input berupa string, maka n adalah panjang string tersebut.

Loops

Salah satu alasan umum mengapa suatu algoritma menjadi lambat adalah karena mengandung banyak loop yang menelusuri input. Semakin banyak loop bersarang (*nested loops*) dalam algoritma, semakin lambat algoritma tersebut. Jika terdapat k loop bersarang, maka kompleksitas waktunya adalah $O(n^k)$.

Sebagai contoh, kompleksitas waktu dari kode berikut adalah $O(n)$:

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

Sedangkan kompleksitas waktu dari kode berikut adalah $O(n^2)$:

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

Hal ini terjadi karena terdapat dua loop bersarang, sehingga jumlah total iterasi adalah $n \times n = n^2$.

Orde/Tingkat Pertumbuhan

Kompleksitas waktu tidak menunjukkan jumlah pasti eksekusi kode dalam sebuah loop, tetapi hanya menunjukkan orde pertumbuhannya. Misalnya, dalam contoh berikut, kode di dalam loop dieksekusi $3n, n + 5$, dan $\lceil n/2 \rceil$ kali, tetapi semuanya memiliki kompleksitas $O(n)$:

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

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

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

Sebagai contoh lain, kompleksitas waktu dari kode berikut adalah $O(n^2)$:

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

Loop bagian dalam berjalan dari $i + 1$ hingga n, menghasilkan total iterasi sekitar $n(n - 1)/2$, yang tetap dalam orde $O(n^2)$.

Tahapan/Fase

Jika suatu algoritma terdiri dari beberapa tahapan yang dieksekusi secara berurutan, kompleksitas waktu totalnya ditentukan oleh tahapan dengan kompleksitas terbesar. Hal ini disebabkan karena tahapan yang paling lambat biasanya menjadi **bottleneck** dalam kode.

Sebagai contoh, kode berikut terdiri dari tiga fase dengan kompleksitas $O(n)$, $O(n^2)$, dan $O(n)$. Maka, kompleksitas totalnya adalah $\boldsymbol{O(n^2)}$, karena itu adalah fase dengan orde terbesar:

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

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // kode
    }
}

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

Meskipun ada dua fase dengan kompleksitas $O(n)$, fase dengan kompleksitas $O(n^2)$ mendominasi perhitungan waktu eksekusi.

Beberapa variabel

Terkadang, kompleksitas waktu tergantung pada beberapa faktor. Dalam hal ini, rumus kompleksitas waktu akan mengandung beberapa variabel.

Sebagai contoh, kompleksitas waktu dari kode berikut adalah $O(nm)$:

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

Ini berarti bahwa jika ada dua variabel n dan m yang mempengaruhi waktu eksekusi, waktu eksekusi algoritma akan bertumbuh sebanding dengan perkalian keduanya, $n \cdot m$.

Rekursi

Kompleksitas waktu dari fungsi rekursif bergantung pada jumlah kali fungsi dipanggil dan kompleksitas waktu dari satu kali panggilan. Total kompleksitas waktu adalah hasil kali dari kedua nilai tersebut.

Sebagai contoh, pertimbangkan fungsi berikut:

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

Panggilan $f(n)$ menyebabkan n kali panggilan fungsi, dan kompleksitas waktu dari setiap panggilan adalah $O(1)$. Dengan demikian, total kompleksitas waktu adalah $O(n)$.

Sebagai contoh lain, pertimbangkan fungsi berikut:

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

Dalam hal ini, setiap panggilan fungsi menghasilkan dua panggilan lainnya, kecuali untuk $n = 1$. Mari kita lihat apa yang terjadi ketika g dipanggil dengan parameter n. Tabel berikut menunjukkan panggilan fungsi yang dihasilkan oleh satu panggilan ini:

Panggilan fungsi	Jumlah panggilan
$g(n)$	1
$g(n - 1)$	2
$g(n - 2)$	4
…	…
$g(1)$	2^{n-1}

Berdasarkan ini, kompleksitas waktu adalah

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

Kelas kompleksitas

Daftar berikut berisi kompleksitas waktu yang umum digunakan dalam algoritma:

$O(1)$ Waktu eksekusi algoritma dengan **waktu konstan** tidak bergantung pada ukuran input. Algoritma dengan waktu konstan yang tipikal adalah rumus langsung yang menghitung jawaban.

$O(\log n)$ Algoritma **logaritmik** sering membagi ukuran input menjadi dua pada setiap langkah. Waktu eksekusi algoritma seperti ini adalah logaritmik, karena $log_2 n$ adalah jumlah pembagian n dengan 2 untuk mendapatkan 1.

$O(\sqrt{n})$ Algoritma **akar kuadrat** lebih lambat daripada $O(\log n)$ tetapi lebih cepat daripada $O(n)$. Sifat khusus dari akar kuadrat adalah bahwa $\sqrt{n} = n/\sqrt{n}$, jadi akar kuadrat $\sqrt{n}$ berada di tengah input.

$O(n)$ Algoritma **linier** memproses input sejumlah tetap kali. Ini seringkali merupakan kompleksitas waktu terbaik yang mungkin, karena biasanya perlu mengakses setiap elemen input setidaknya sekali sebelum memberikan jawaban.

$O(n \log n)$ Kompleksitas waktu ini sering menunjukkan bahwa algoritma mengurutkan input, karena kompleksitas waktu algoritma pengurutan yang efisien adalah $O(n \log n)$. Kemungkinan lainnya adalah algoritma menggunakan struktur data di mana setiap operasi memakan waktu $O(\log n)$.

$O(n^2)$ Algoritma **kuadrat** sering mengandung dua loop bersarang. Kemungkinan untuk memeriksa semua pasangan elemen input dalam waktu $O(n^2)$.

$O(n^3)$ Algoritma **kubik** sering mengandung tiga loop bersarang. Kemungkinan untuk memeriksa semua triplet elemen input dalam waktu $O(n^3)$.

$O(2^n)$ Kompleksitas waktu ini sering menunjukkan bahwa algoritma mengiterasi melalui semua subset elemen input. Sebagai contoh, subset dari {1,2,3} adalah ∅, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, dan {1,2,3}.

$O(n!)$ Kompleksitas waktu ini sering menunjukkan bahwa algoritma mengiterasi melalui semua permutasi elemen input. Sebagai contoh, permutasi dari {1,2,3} adalah (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), dan (3,2,1).

Sebuah algoritma disebut **polinomial** jika kompleksitas waktunya paling banyak $O(n^k)$ di mana k adalah sebuah konstanta. Semua kompleksitas waktu di atas, kecuali $O(2^n)$ dan $O(n!)$, adalah polinomial. Dalam praktiknya, konstanta k biasanya kecil, dan oleh karena itu, kompleksitas waktu polinomial berarti bahwa algoritma tersebut **cukup efisien**.

Sebagian besar algoritma dalam buku ini adalah polinomial. Namun, masih ada banyak masalah penting di mana tidak ada algoritma polinomial yang diketahui, yaitu, belum ada yang tahu cara menyelesaikannya dengan efisien. Masalah **NP-hard** adalah kumpulan masalah penting untuk yang belum ada algoritma polinomial yang diketahui.

Estimasi efisiensi

Dengan menghitung kompleksitas waktu dari sebuah algoritma, kita dapat memeriksa, sebelum mengimplementasikan algoritma tersebut, apakah algoritma itu cukup efisien untuk menyelesaikan masalah. Titik awal untuk estimasi adalah fakta bahwa komputer modern dapat melakukan beberapa ratus juta operasi dalam satu detik.

Sebagai contoh, anggap batas waktu untuk sebuah masalah adalah satu detik dan ukuran input adalah $n = 10^5$. Jika kompleksitas waktunya adalah $O(n^2)$, algoritma akan melakukan sekitar $(10^5)^2 = 10^{10}$ operasi. Ini seharusnya memakan waktu setidaknya beberapa puluh detik, jadi algoritma tersebut sepertinya terlalu lambat untuk menyelesaikan masalah tersebut.

Di sisi lain, dengan ukuran input yang diberikan, kita dapat mencoba menebak kompleksitas waktu yang dibutuhkan oleh algoritma untuk menyelesaikan masalah tersebut. Tabel berikut berisi beberapa estimasi berguna dengan asumsi batas waktu satu detik.

Ukuran input	Kebutuhan kompleksitas waktu
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ atau $O(n)$
n sangat besar	$O(1)$ atau $O(log\ n)$

Sebagai contoh, jika ukuran input adalah $n = 10^5$, kemungkinan besar diharapkan bahwa kompleksitas waktu algoritma adalah $O(n)$ atau $O(n\ log\ n)$. Informasi ini mempermudah perancangan algoritma, karena ini mengeliminasi pendekatan-pendekatan yang akan menghasilkan algoritma dengan kompleksitas waktu yang lebih buruk.

Namun, penting untuk diingat bahwa kompleksitas waktu hanya merupakan perkiraan efisiensi, karena ia menyembunyikan *faktor konstan*. Sebagai contoh, algoritma yang berjalan dalam waktu $O(n)$ mungkin melakukan $n/2$ atau $5n$ operasi. Ini memiliki efek penting pada waktu eksekusi algoritma yang sebenarnya.

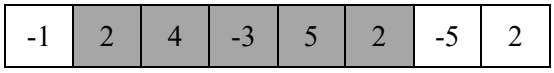
Jumlah subarray maksimum

Seringkali ada beberapa algoritma yang mungkin untuk menyelesaikan sebuah masalah dengan kompleksitas waktu yang berbeda. Bagian ini membahas sebuah masalah klasik yang memiliki solusi $O(n^3)$ yang langsung. Namun, dengan merancang algoritma yang lebih baik, masalah ini dapat diselesaikan dalam waktu $O(n^2)$ dan bahkan dalam waktu $O(n)$.

Diberikan sebuah array yang berisi n angka, tugas kita adalah menghitung jumlah subarray maksimum, yaitu jumlah terbesar yang mungkin dari urutan nilai yang berurutan dalam array. Masalah ini menarik ketika mungkin ada nilai negatif dalam array. Sebagai contoh, dalam array

-1	2	4	-3	5	2	-5	2

subarray berikut menghasilkan jumlah maksimum 10:



Kami mengasumsikan bahwa subarray kosong diperbolehkan, sehingga jumlah subarray maksimum selalu setidaknya 0.

Algoritma 1

Cara sederhana untuk menyelesaikan masalah ini adalah dengan melalui semua subarray yang mungkin, menghitung jumlah nilai di setiap subarray, dan mempertahankan jumlah maksimum. Berikut adalah kode yang mengimplementasikan algoritma ini:

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

Variabel `a` dan `b` mengatur indeks pertama dan terakhir dari subarray, dan jumlah nilai dihitung ke dalam variabel `sum`. Variabel `best` berisi jumlah maksimum yang ditemukan selama pencarian.

Waktu kompleksitas algoritma ini adalah $O(n^3)$, karena terdiri dari tiga loop bertingkat yang melalui input.

Algoritma 2

Mudah untuk membuat Algoritma 1 lebih efisien dengan menghapus satu loop darinya. Hal ini dapat dilakukan dengan menghitung jumlah pada saat yang sama ketika ujung kanan subarray bergerak. Hasilnya adalah kode berikut:

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

Setelah perubahan ini, waktu kompleksitasnya menjadi $O(n^2)$.

Algoritma 3

Mengejutkan, ternyata masalah ini dapat diselesaikan dalam waktu $O(n)$, yang berarti hanya satu loop sudah cukup. Ide dasarnya adalah untuk menghitung, untuk setiap posisi array, jumlah maksimum dari subarray yang berakhir di posisi tersebut. Setelah itu, jawaban untuk masalah ini adalah maksimum dari jumlah-jumlah tersebut.

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

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

Pada kasus kedua, karena kita ingin menemukan subarray dengan jumlah maksimum, subarray yang berakhir di posisi $k - 1$ harus memiliki jumlah maksimum juga. Dengan demikian, kita dapat menyelesaikan masalah ini dengan efisien dengan menghitung jumlah maksimum subarray untuk setiap posisi akhir dari kiri ke kanan.

Kode berikut mengimplementasikan algoritma ini:

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

Algoritma ini hanya mengandung satu loop yang melalui input, sehingga waktu kompleksitasnya adalah $O(n)$. Ini juga merupakan waktu kompleksitas terbaik yang mungkin, karena algoritma apapun untuk masalah ini harus memeriksa semua elemen array setidaknya sekali.

Perbandingan Efisiensi

Menarik untuk mempelajari seberapa efisien algoritma dalam praktik. Tabel berikut menunjukkan waktu eksekusi dari algoritma-algoritma di atas untuk berbagai nilai n pada komputer modern. Pada setiap uji coba, input dihasilkan secara acak. Waktu yang dibutuhkan untuk membaca input tidak diukur.

Ukuran n array	Algoritma 1	Algoritma 2	Algoritma 3
10^2	0.0 s	0.0 s	0.0 s
10^3	0.1 s	0.0 s	0.0 s
10^4	> 10.0 s	0.1 s	0.0 s
10^5	> 10.0 s	5.3 s	0.0 s
10^6	> 10.0 s	> 10.0 s	0.0 s
10^7	> 10.0 s	> 10.0 s	0.0 s

Perbandingan menunjukkan bahwa semua algoritma efisien ketika ukuran input kecil, tetapi input yang lebih besar menonjolkan perbedaan signifikan dalam waktu eksekusi algoritma-algoritma tersebut. Algoritma 1 menjadi lambat ketika $n = 10^4$, dan Algoritma 2 menjadi lambat ketika $n = 10^5$. Hanya Algoritma 3 yang mampu memproses bahkan input terbesar dengan cepat.