



# Analisis Kompleksitas

Tim Olimpiade Komputer Indonesia

# Pendahuluan

Melalui dokumen ini, kalian akan:

- Memahami konsep analisis kompleksitas.
- Mampu menganalisis kompleksitas untuk memperkirakan *runtime* eksekusi program.



Bagian 1

## Perkenalan Analisis Algoritma



# Analisis Algoritma

- Diberikan dua algoritma untuk menyelesaikan permasalahan yang sama. Algoritma mana yang lebih cepat?
- Pengukuran seberapa cepatnya suatu algoritma biasa dinyatakan dalam kompleksitas waktu.
- Kompleksitas waktu: banyaknya komputasi yang perlu dilakukan dari awal eksekusi sampai berakhirnya algoritma.



## Contoh Soal: Membajak Sawah

Deskripsi:

- Pak Dengklek memiliki  $N$  bibit tanaman yang akan ia semai di sawahnya.
- Untuk itu, ia akan membajak sawahnya supaya sawahnya bisa memuat  $N$  tanaman.
- Sawah yang akan dibajak harus memiliki bentuk persegi panjang, tersusun atas  $R$  baris dan  $C$  kolom petak-petak. Setiap petak bisa memuat maksimal sebuah tanaman.
- Tentukan nilai  $R$  dan  $C$  supaya semua petak yang ada ditanami tanaman!
- Jika ada lebih dari satu kemungkinan jawaban, minimalkan selisih  $R$  dengan  $C$ .
- Jika masih ada lebih dari satu kemungkinan jawaban, cetak yang mana saja.



## Contoh Soal: Membajak Sawah (lanj.)

Batasan:

- $1 \leq N \leq 10^9$ .

Format Masukan:

- Sebuah baris berisi bilangan bulat, yaitu  $N$ .

Format Keluaran:

- Sebuah baris berisi dua bilangan bulat, yaitu  $R$  dan  $C$ .



## Contoh Soal: Membajak Sawah (lanj.)

Contoh Masukan

35

Contoh Keluaran

7 5



## Solusi 1: Coba Semua Kemungkinan

- Untuk setiap  $R$  dan  $C$  yang mungkin, coba hitung apakah  $R \times C$  sama dengan  $N$ .
- Jika ya, cari yang selisih  $|R - C|$  minimal.
- Cukup mencoba untuk  $1 \leq R \leq N$  dan  $1 \leq C \leq N$ .



## Solusi 1: Coba Semua Kemungkinan (lanj.)

Berikut implementasinya:

---

```
#include <cstdio>
#include <cmath>

using namespace std;

int main() {
    int N, R, C;
    scanf("%d", &N);
    R = 1;
    C = N;
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            if (i*j == N) {
                if (abs(R-C) > abs(i-j)) {
                    R = i;
                    C = j;
                }
            }
        }
    }
    printf("%d %d\n", R, C);
}
```



## Solusi 1: Coba Semua Kemungkinan (lanj.)

- Fungsi `abs` adalah fungsi yang disediakan STL `cmath` untuk mengambil harga mutlak.
- Misalkan untuk  $N = 100$ , secara kasar diperlukan  $100 \times 100$  komputasi untuk mencari nilai  $R$  dan  $C$  yang tepat.
- Jadi secara umum bisa diperkirakan bahwa untuk suatu nilai  $N$ , diperlukan  $N^2$  komputasi.
- Solusi ini dikatakan memiliki kompleksitas waktu sebesar  $O(N^2)$  (dibaca "O-N-kuadrat").
- Pertanyaan: apakah solusi ini cukup cepat? Bagaimana jika  $N = 10^9$ ?



## Cerita Sampingan: Meramal Waktu Eksekusi

- Terdapat sebuah perkiraan kasar bahwa komputer mampu melakukan 100 juta ( $10^8$ ) komputasi dalam 1 detik.
- Tentu saja, perkiraan ini masih sangat kasar. Waktu untuk melakukan  $10^8$  operasi penjumlahan tidak sama dengan waktu untuk melakukan  $10^8$  operasi modulo.
- Jenis bahasa pemrograman juga mempengaruhi waktu eksekusi algoritma, misalnya bahasa C cenderung lebih cepat daripada Java.
- Bagaimanapun juga, konvensi ini umum digunakan pada dunia pemrograman kompetitif dan sesuai untuk bahasa Pascal, C, dan C++.



## Solusi 1: Terlalu lambat!

- Untuk  $N$  yang bisa mencapai  $10^9$ , diperlukan sekitar  $10^{18}/10^8 = 10^{10}$  detik.
- Waktu tersebut setara dengan sekitar 317 tahun!
- Adakah solusi lebih efisien?



## Solusi 2: Coba Semua Kemungkinan $R$

- Tidak perlu memeriksa semua  $R$  dan  $C$ , cukup coba saja semua kemungkinan  $R$  untuk  $1 \leq R \leq N$ .
- Jika untuk suatu nilai  $R$ , diketahui  $N$  habis dibagi  $R$ , maka  $C$  dipastikan ada, yaitu  $N/R$ .



## Solusi 2: Coba Semua Kemungkinan $R$ (lanj.)

Bagian implementasi:

---

```
scanf("%d", &N);
R = 1;
C = N;
for (int i = 1; i <= N; i++) {
    if (N % i == 0) {
        int j = N / i;
        if (abs(R-C) > abs(i-j)) {
            R = i;
            C = j;
        }
    }
}
```

---



## Solusi 2: Coba Semua Kemungkinan $R$ (lanj.)

- Solusi ini bekerja dengan lebih cepat.
- Untuk suatu nilai  $N$ , kasarnya cukup dilakukan  $N$  komputasi untuk mencari nilai  $R$  dan  $C$  yang tepat.
- Solusi ini dikatakan memiliki kompleksitas waktu sebesar  $O(N)$ .
- Untuk  $N = 10^9$ , diperlukan sekitar 10 detik eksekusi algoritma.



## Solusi 3: Batasi $R$ sampai $\sqrt{N}$

- Persoalan ini sebenarnya meminta kita memfaktorkan  $N$ , supaya dua bilangan hasil faktorisasi sedekat mungkin.
- Untuk memeriksa seluruh faktor bilangan, cukup batasi sampai  $\sqrt{N}$  saja.
- Contoh: untuk  $N = 100$ , faktorisasi yang mungkin adalah:
  - $1 \times 100$
  - $2 \times 50$
  - $4 \times 25$
  - $5 \times 20$
  - $10 \times 10$
  - $20 \times 5$
  - $25 \times 4$
  - ... (faktorisasi selanjutnya hanya mengulang yang sudah ada)



## Solusi 3: Batasi $R$ sampai $\sqrt{N}$ (lanj.)

Bagian implementasi:

---

```
scanf("%d", &N);
R = 1;
C = N;

int i = 1;
while (i*i <= N) {
    if (N % i == 0) {
        int j = N / i;
        if (abs(R-C) > abs(i-j)) {
            R = i;
            C = j;
        }
    }
    i++;
}
printf("%d %d\n", R, C);
```

---



## Solusi 3: Batasi $R$ sampai $\sqrt{N}$ (lanj.)

- Kompleksitas solusi menjadi hanya  $O(\sqrt{N})$ .
- Untuk  $N = 10^9$ , hanya diperlukan sekitar 32.000 komputasi, jauh di bawah 100 juta.
- Solusi ini bekerja dengan cepat bahkan untuk  $N$  yang besar.



## Ulasan Contoh Soal

- Untuk menyelesaikan suatu permasalahan, bisa jadi ada beberapa solusi, masing-masing dengan kompleksitasnya tersendiri.
- Dari ketiga solusi yang telah dijelaskan, solusi ketiga sudah pasti paling diharapkan untuk bisa menyelesaikan permasalahan.
- Untuk mengukur seberapa efisien suatu algoritma, bisa digunakan notasi Big-Oh untuk kompleksitas waktu.



# Notasi Big-Oh

- Biasa digunakan pada ilmu komputer untuk menyatakan pertumbuhan nilai suatu fungsi terhadap ukuran masukan yang diberikan.
- Dalam kasus ini, fungsi yang dimaksud adalah fungsi banyaknya komputasi yang diperlukan jika diberikan suatu ukuran masukan.
- Kita tidak akan menggali terlalu dalam tentang hal-hal matematis di balik notasi Big-Oh ini, hanya kulit luarnya saja.



# Aturan Sederhana Notasi Big-Oh

1. Konstanta bisa diabaikan.

Contoh:  $O(3N^2)$  bisa ditulis  $O(N^2)$  saja.

Alasan: kita hanya tertarik dengan **pertumbuhan fungsinya**, bukan nilai fungsi sebenarnya.

2. Cukup ambil suku yang mendominasi.

Contoh:  $O(N^3 + N^2)$  bisa ditulis  $O(N^3)$  saja.

Alasan: untuk  $N$  yang besar, suku  $N^3$  akan jauh lebih besar daripada suku  $N^2$ , sehingga  $N^2$  menjadi tidak signifikan.



# Kelompok Kompleksitas

Biasanya kompleksitas dikelompokkan menurut kelasnya sebagai berikut:

- *Constant*:  $O(1)$   
Komputasi yang dilakukan tidak bergantung pada besarnya input. Contoh: program untuk mencari nilai harga mutlak suatu angka.
- *Logarithmic*:  $O(\log N)$   
Komputasi yang dilakukan proporsional terhadap nilai logaritma dari input.

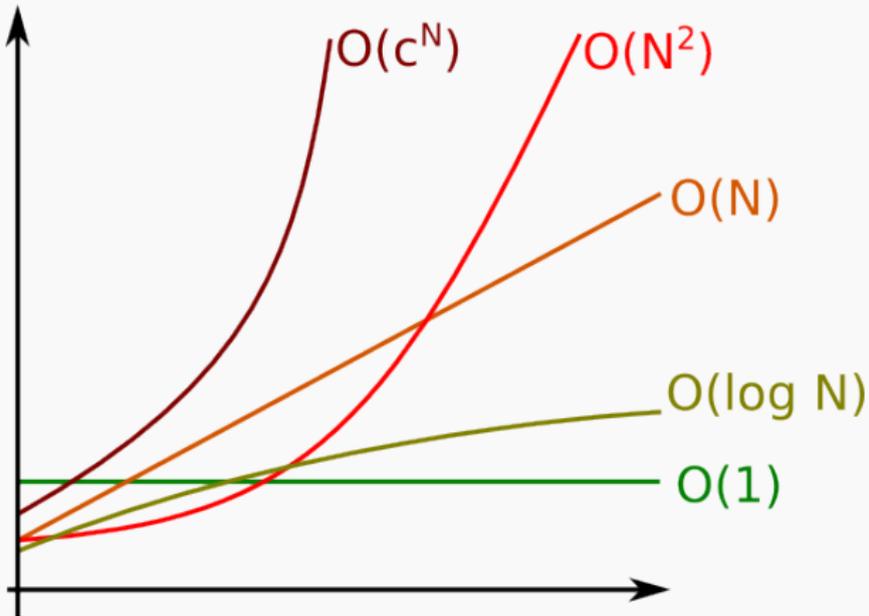


## Kelompok Kompleksitas (lanj.)

- *Linear*:  $O(N)$   
Komputasi yang dilakukan proporsional secara linier terhadap input.
- *Polynomial*:  $O(\sqrt{N})$ ,  $O(N^2)$ ,  $O(N^3)$ , ...  
Komputasi yang dilakukan proporsional secara polinomial terhadap input.
- *Exponential*:  $O(N!)$ ,  $O(2^N)$ ,  $O(N^N)$ , ...  
Komputasi yang dilakukan proporsional secara eksponensial terhadap input. Biasanya dihindari karena terlalu lambat.



## Kelompok Kompleksitas (lanj.)



## Bagian 2

# Menghitung Kompleksitas



# Menghitung Kompleksitas

- Wajib dilakukan sebelum mengimplementasikan suatu algoritma.
- Tujuannya untuk memperkirakan apakah solusi ini cukup efisien untuk menyelesaikan persoalan yang ada.
- Dengan sedikit latihan, Anda dapat menghitung kompleksitas dari algoritma sederhana.



## Contoh 1: Soal

Hitung kompleksitas waktu potongan program berikut:

---

```
total = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        total++;
    }
}
```

---



## Contoh 1: Jawaban

- Sederhana, jawabannya adalah  $O(N^2)$ .



## Contoh 2: Soal

Hitung kompleksitas waktu potongan program berikut:

---

```
total = 0;
for (int i = 1; i <= N; i++) {
    for (int j = i; j <= N; j++) { // j dimulai dari i
        total++;
    }
}
```

---



## Contoh 2: Jawaban

- Banyaknya operasi "total := total + 1" yang dilakukan adalah  $N + (N - 1) + (N - 2) + \dots + 2 + 1 = \frac{N(N+1)}{2}$ .
- Kompleksitasnya  $O\left(\frac{N(N+1)}{2}\right)$ , tetapi cukup ditulis  $O(N^2)$  saja.



## Contoh 3: Soal

Hitung kompleksitas waktu potongan program berikut:

---

```
total = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
        total++;
    }
}
```

---



## Contoh 3: Jawaban

- Kali ini terdapat dua variabel pada input, yaitu  $N$  dan  $M$ .
- Kompleksitasnya adalah  $O(NM)$ .



## Contoh 4: Soal

Hitung kompleksitas waktu potongan program berikut:

---

```
val = N;
while (val > 0) {
    val /= 3; // Setara "val = val / 3"
}
```

---



## Contoh 4: Jawaban

- Banyaknya operasi yang dilaksanakan setara dengan panjang dari barisan  $\frac{N}{3}, \frac{N}{9}, \frac{N}{27}, \dots, 1$ .
- Panjang dari barisan tersebut sebenarnya adalah logaritma basis 3 dari  $N$ , atau bisa dituliskan kompleksitasnya  $O(\log_3 N)$ .
- Namun sebenarnya  $\log_3 N = \frac{\log N}{\log 3} = \frac{1}{\log 3} \log N$ .
- Berhubung  $\frac{1}{\log 3}$  adalah konstanta, jadi cukup ditulis  $O(\log N)$  saja.



## Contoh 5: Soal

Hitung kompleksitas waktu potongan program berikut:

---

```
counter = 1;
while (counter*counter < N) {
    counter++;
}
```

---



## Contoh 5: Jawaban

- Nilai variabel *counter* akan terus bertambah, hingga kuadratnya lebih dari  $N$ .
- Misalkan jika  $N = 81$ , maka *counter* akan berhenti setelah nilainya melebihi 9.
- Kompleksitas sebenarnya adalah  $O(\sqrt{N})$ .



# Penutup

- Pelajari lebih lanjut tentang perhitungan kompleksitas melalui latihan yang diberikan.
- Terdapat notasi lainnya yang tidak kita bahas di sini, seperti Big-Theta ( $\Theta$ ), Big-Omega ( $\Omega$ ), Little-Oh ( $o$ ), dan sebagainya. Silakan Anda pelajari jika tertarik untuk mengetahui lebih lanjut.

