Pengantar Big O dan Model Matematika Big O

# Pendahuluan

Algoritma merupakan langkah-langkah yang ditempuh untuk menyelesaikan masalah. Ada sebuah, atau sekumpulan, input yang diproses sehingga menghasilkan sebuah (atau sekumpulan) output ataupun kondisi yang didefinisikan sebagai solusi untuk masalah di awal. Langkah-langkah yang ditempuh, bertujuan untuk mentransfer logika manusia dalam menyelesaikan masalah komputasi, ke dalam program komputer.

Masalah komputasi tidak hanya dapat diselesaikan dengan sebuah cara. Bisa jadi, ada banyak jenis cara atau algoritma yang dapat menyelesaikan sebuah pokok permasalahan yang sama. Sebagai contoh, masalah pengurutan dapat diselesaikan dengan algoritma Bubble Sort, Merge Sort, dan berbagai algoritma yang lain.

## Kompleksitas Algoritma

Algoritma yang berbeda tentu saja memiliki langkah-langkah yang berbeda. Sehingga, ada algoritma yang lebih “cepat” dibandingkan algoritma yang lain. Dan ada pula algoritma yang lebih “hemat” dalam penggunaan memori. Algoritma yang baik adalah algoritma yang dapat menyelesaikan masalah sekaligus efisien dalam pemakaian waktu dan ruang. Istilah yang digunakan untuk mengukur hal tersebut adalah **kompleksitas algoritma**. Maka, terdapat 2 jenis kompleksitas algoritma, yakni:

* Kompleksitas waktu (T(n)), dan
* kompleksitas ruang (S(n)).

Tentu diperlukan sebuah metode untuk mengukur seberapa “cepat”-kah sebuah algoritma atau seberapa “hemat”-kah pemakaian ruang memori algoritma tersebut. Menganalisis sebuah algoritma mencakup menganalisis kompleksitas algoritma. Meskipun terdapat 2 jenis kompleksitas, dalam analisis algoritma yang lebih difokuskan adalah analisis “kecepatan” algoritma tersebut, atau kompleksitas waktunya (T(n)).

“Cepat – lambat” sebuah algoritma tidak dihitung menggunakan satuan waktu yang umum digunakan dalam kehidupan, seperti detik, menit, ataupun jam. Alasannya karena kecepatan proses sebuah algoritma yang sudah diimplementasikan ke dalam program komputer bergantung pada banyak faktor, seperti processor, sistem operasi, compiler yang digunakan, dan faktor lain. Menghitung komplksitas waktu sebuah algoritma adalah **menghitung jumlah langkah yang ditempuh, serta perubahan jumlah langkah saat ukuran input berubah.** Jadi, tidak aka nada hubungannya dengan performa komputer atau kompiler yang digunakan sebagai media implementasi algoritma tersebut.

## Notasi Asimtotik

Sebagaimana dijelaskan pada sub-bagian di atas, untuk mengukur kompleksitas waktu algoritma tidak digunakan satuan waktu yang umum. Maka, dibutuhkan sebuah notasi khusus. Notasi matematis yang digunakan disebuh **notasi asimtotik.** Notasi asimtotik digunakan untuk melihat seberapa efisien performa sebuah algoritma bila dibandingkan dengan algoritma yang lain.

Terdapat beberapa jenis notasi asimtotik dalam bidang ilmu matematika yang digunakan dalam pengukuran kompleksitas algoritma. 3 contoh jenis notasi asimtotik adalah:

1. O (Big-O)
2. Ω (Big-Omega)
3. ϴ (Big-Theta)

Dalam modul ini tidak akan dijelaskan secara terperinci bagaimana menggunakan masing-masing notasi. Namun, yang akan dijelaskan adalah bagaimana kompleksitas waktu sebuah algoritma dapat ditemukan dengan menggunakan notasi tersebut.

Notasi yang paling umum digunakan adalah O (Big-O). Alasannya adalah karena notasi tersebut digunakan untuk menggambarkan batas atas (*upper limit*) dari sebuah fungsi ketika masukan untuk fungsi tersebut meningkat (*increase*). Alhasil, Big-O sangat cocok untuk memperlihatkan *worst case* dari sebuah algoritma. Apa yang dimaksud dengan *worst case* akan dijelaskan pada salah satu sub-bagian dari modul ini.

Contoh penggunaan Big-O: Algoritma Bubble Sort. T(n) = O(n2). Artinya: algoritma Bubble Sort memiliki kompleksitas waktu n2.

# Mengukur Kompleksitas Algoritma Menggunakan Big-O

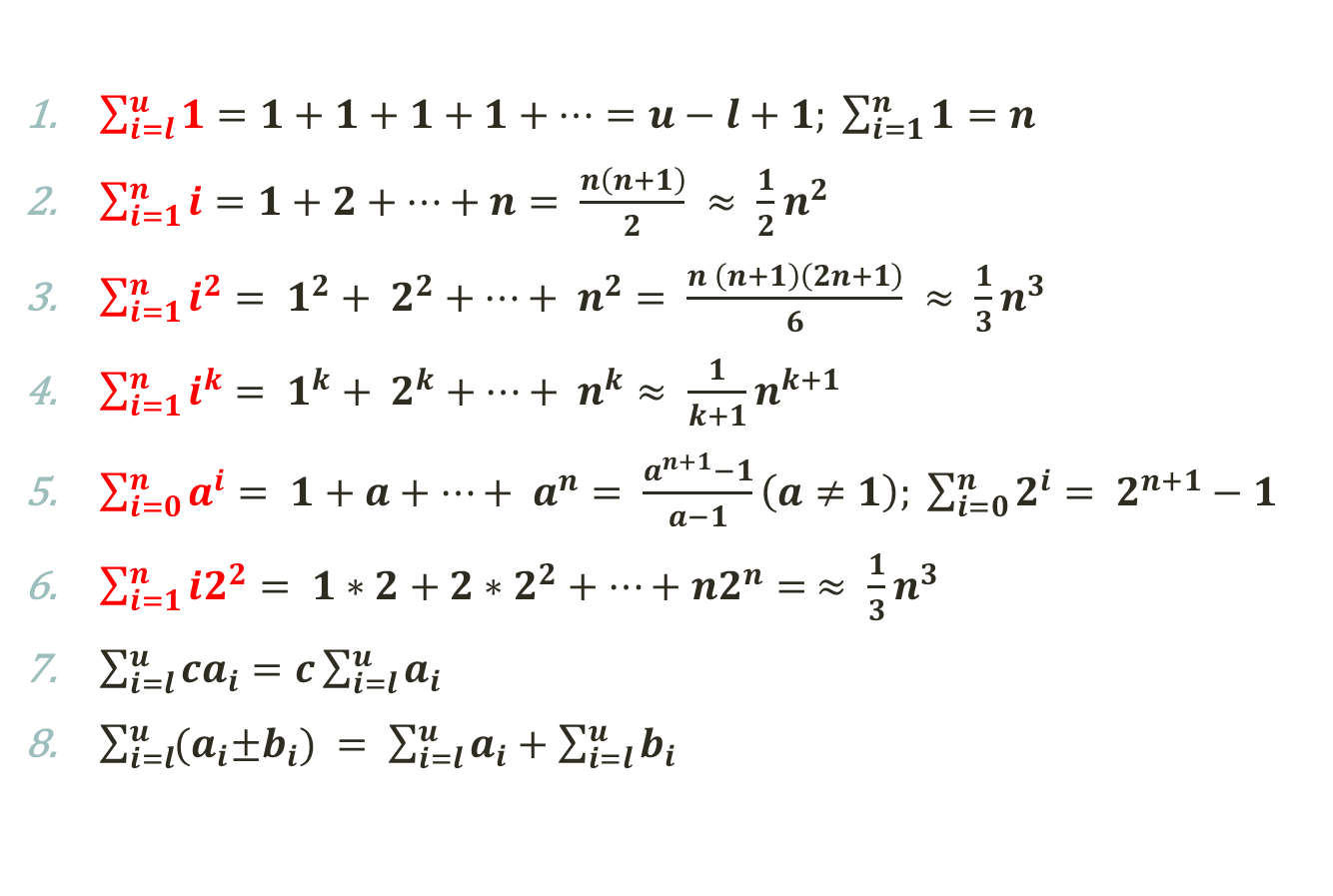
Cara menghitung kompleksitas algoritma non rekursif tidak persis dengan algoritma rekursif. Namun yang pasti adalah, bahwa penggunaan fungsi matematis dan aturan dasar penjumlahan digunakan dalam perhitungan.

Pada setiap modul, seraya berbagai teknik desain algoritma akan dibahas, juga akan diperlihatkan bagaimana menghitung kompleksitas algoritmanya.

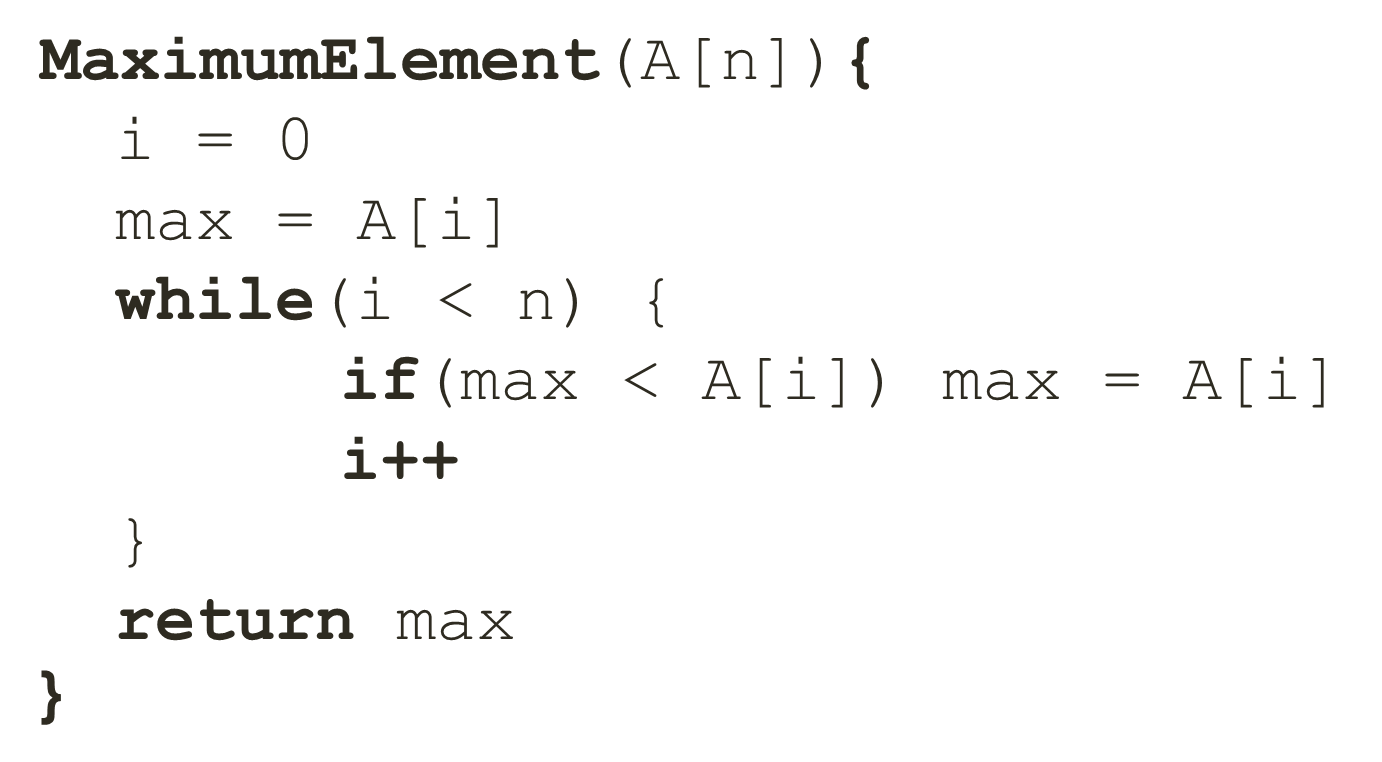
Secara umum, langkah-langkah untuk mengukur kompleksitas algoritma non rekursif (umum):

1. Mengidentifikasi operasi dasar
2. Menentukan ukuran input.
3. Memeriksa apakah pertumbuhan operasi dasar hanya bergantung pada ukuran input. Jika ya, maka tidak ada *worst case* ataupun *best case*. Namun jika terdapat faktor lain, maka perhitungan *worst case* dibedakan dengan *best case*.
4. Menghitung derajat pertumbuhan langkah operasi dasar terhadap perubahan ukuran input dengan menggunakan fungsi matematika.

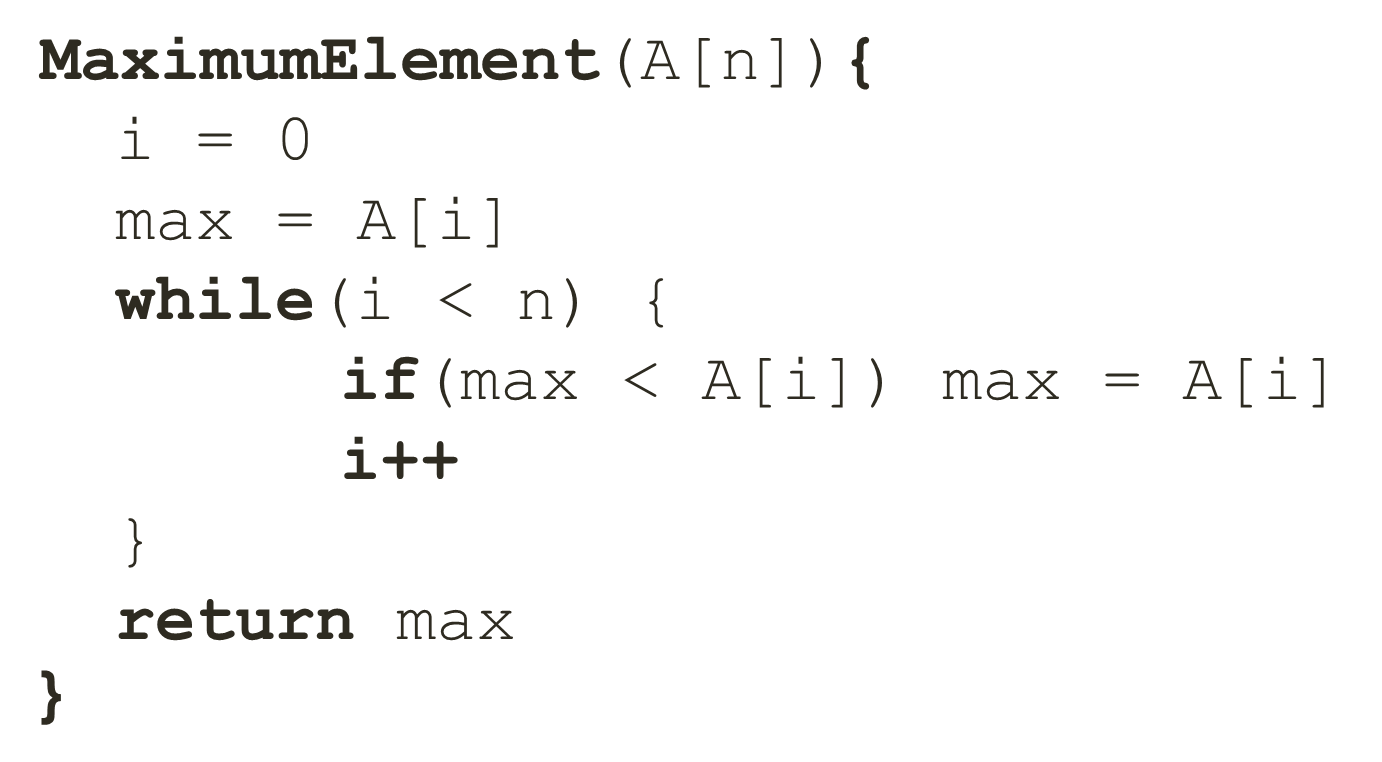
Beberapa rumus matematika yang umum digunakan:



Contoh sederhana:



1. **Mengidentifikasi operasi dasar**. Operasi dasar adalah perbandingan antara **max** dan **A[i]**, karena itulah operasi yang paling sering dilakukan yang merupakan logika dasar algoritma tersebut untuk memecahkan masalah.
2. **Menentukan ukuran input**. Ukuran input adalah **n**, yang merupakan besar / panjang dari array atau list **A**. **n** menjadi ukuran input karena berpengaruh terhadap banyaknya operasi dasar dilakukan.
3. **Memeriksa apakah pertumbuhan operasi dasar hanya bergantung pada ukuran input.** Untuk algoritma di atas, **n** merupakan satu-satunya penentu banyaknya operasi dasar dilakukan, sehingga dalam perhitungan kompleksitas tidak diikutsertakan analisis *worst case, best case,* maupun *average case*.
4. **Menghitung derajat pertumbuhan langkah operasi dasar terhadap perubahan ukuran input dengan menggunakan fungsi matematika.** Di bawah ini merupakan cara perhitungannya.



Penjelasan: Perulangan merupakan deret, sehingga perulangan yang terpola dengan baik dapat digantikan ke dalam rumus matematis di atas. Isi dari sigma berisi 1 karena: **dalam setiap perulangan terdapat 1 kali operasi dasar dilakukan**.

Penjelasan: Gunakan rumus pertama yang tertera sebelum contoh algoritma.

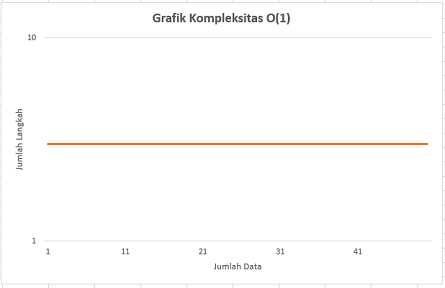
Penjelasan: Hasil akhir diubah ke dalam notasi asimtotik sesuai dengan aturan notasi.

Alhasil, algoritma di atas memiliki kompleksitas O(n), atau sering disebut dengan **kompleksitas linear**. Terdapat beberapa pengelompokan kompleksitas yang umum.

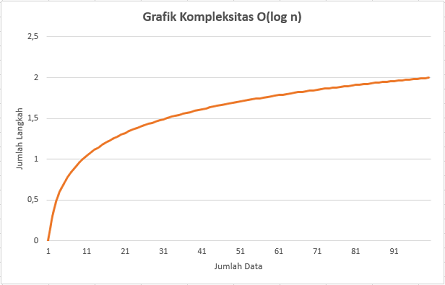
# Kriteria Efisiensi Umum

Terdapat beberapa kriteria efisiensi umum:

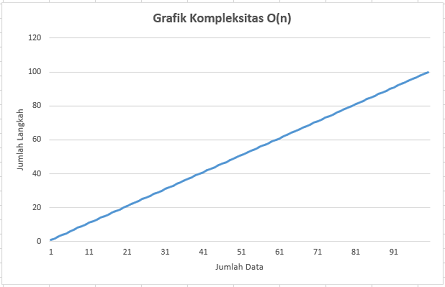
1. ***O*(1): Kompleksitas Konstan**. Merupakan kriteria di mana ukuran input sama sekali tidak berpengaruh pada jumlah langkah sebuah algoritma. Kompleksitas ini merupakan jenis yang paling efisien / cepat.



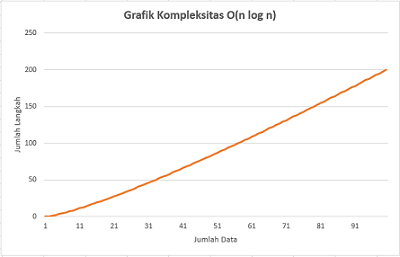
1. ***O* (log n): Kompleksitas Logaritmik**. Merupakan kompleksitas di mana perubahan / peningkatan besar dari ukuran input hanya memberikan sedikit sekali pengaruh terhadap pertumbuhan jumlah langkah. Algoritma yang memiliki kompleksitas logaritmik sering dijumpai pada algoritma yang dibuat dengan teknik *divide & conquer* yang akan dibahas pada salah satu modul.



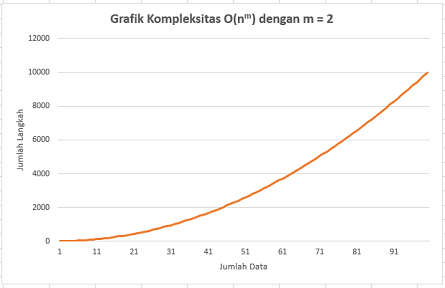
1. ***O* (n): Linear**. Merupakan kompleksitas algoritma di mana pertumbuhan ukuran input sebanding dengan pertumbuhan jumlah langkah. Algoritma dengan kompleksitas linear dipandang sebagai algoritma yang cepat / efisien, meskipun itu juga bergantung pada permasalahan apa yang diselesaikan oleh sebuah algoritma.



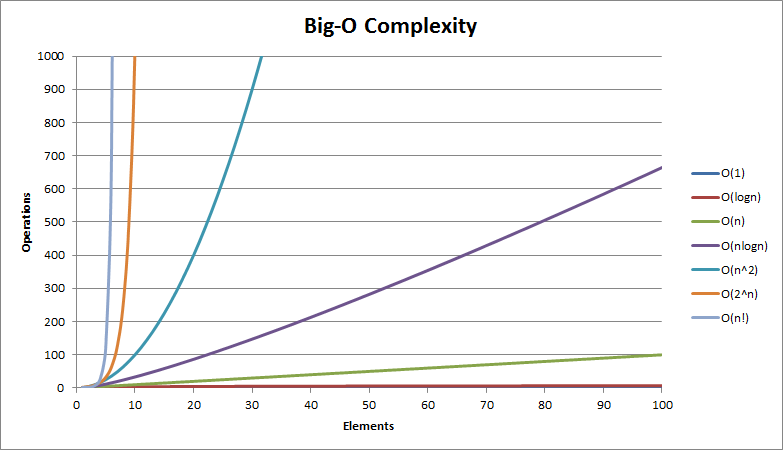
1. ***O* (n log n)**. Kompleksitas ini belum memiliki nama khusus. Jenis kompleksitas ini juga sering dijumpai pada algoritma yang berbasis teknik *divide & conquer*. Hanya saja, pada kompleksitas jenis ini, perubahan jumlah langkah sedikit lebih besar daripada kompleksitas linear.



1. ***O* (nm): Kompleksitas Polinomial**. Merupakan jenis kompleksitas yang tinggi. *O* (n2) disebut dengan **kompleksitas kuadratik**. Merupakan jenis kompleksitas di mana sedikit perubahan pada ukuran input akan berpengaruh besar pada jumlah langkah.



1. **O (n!): Kompleksitas faktorial**. Merupakan jenis kompleksitas yang sangat tinggi. Sedikit perubahan pada ukuran input akan berpengaruh sangat besar terhadap jumlah langkah. Kompleksitas ini sangat dihindari, namun untuk sebuah permasalahan komputasi yang sangat kompleks algoritma dengan kompleksitas yang tinggi juga sering digunakan.



# Worst Case, Average Case, dan Best Case

Dalam pengukuran kompleksitas algoritma, terdapat *worst case, average case,* dan *best case*.

1. *Best case* / kasus terbaik: merupakan kondisi di mana algoritma akan bekerja dengan kompleksitas paling rendah untuk ukuran input yang sama.
2. *Worst case* / kasus terburuk: merupakan kebalikan dari kasus terbaik, di mana algoritma akan bekerja dengan kompleksitas tertinggi untuk ukuran input yang sama.
3. *Average case* / kasus rata-rata: merupakan kondisi yang paling sering terjadi untuk sebuah permasalahan.