Algoritma dan Kompleksitas Algoritma

Algoritma

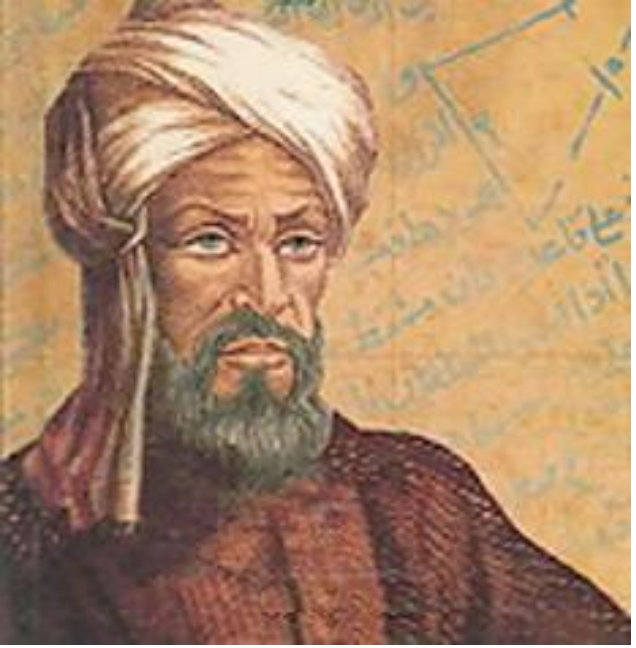
- Algoritma adalah urutan logis langkah-langkah penyelesaian masalah yang ditinjau secara sistematis.

Asal Usul Algoritma

- Kata ini tidak muncul dalam kamus Webster hingga tahun 1957
- Orang hanya menemukan kata "*algorism*" yang berarti "proses menghitung dengan angka Arab"
- Anda dikatakan *algorist* jika anda menggunakan angka Arab
- Para ahli bahasa berusaha menemukan asal kata algorism ini namun hasilnya kurang memuaskan

Asal Usul

- Akhirnya para ahli sejarah matematika menemukan asal mula kata tsb.
- Kata ‘algorism’ berasal dari nama penulis buku arab yang terkenal yaitu Abu Ja’far Muhammad Ibn Musa *al-Khuwarizmi*



Ibn Musa al-Khuwarizmi

- Seorang ahli matematika, astronomi, astrologi, dan geografi yang berasal dari Persia.
- Lahir sekitar tahun 780 di Khwārizm (sekarang Khiva, Uzbekistan) dan wafat sekitar tahun 850 di Baghdad.
- Hampir sepanjang hidupnya, ia bekerja sebagai dosen di Sekolah Kehormatan di Baghdad

- Buku pertamanya, al-Jabar, adalah buku pertama yang membahas solusi sistematik dari linear dan notasi kuadrat. Sehingga ia disebut sebagai Bapak Aljabar.
- Translasi bahasa Latin dari Aritmatika beliau, yang memperkenalkan angka India, kemudian diperkenalkan sebagai Sistem Penomoran Posisi Desimal di dunia Barat pada abad ke 12
- Karya: Aljabar, Dixit Algorismi, Rekonstruksi Planetarium, Astronomi, dll

Penulisan Algoritma

- Dapat menggunakan kalimat deskriptif
- Yang bagus untuk algoritma yang pendek
- Dapat juga dinyatakan dalam bahasa pemrograman, namun memiliki kerumitan sintaks
- Para ilmuwan lebih menyukai penulisan dalam *Pseudo-Code*

Contoh Algoritma (1)

Resep membuat Rendang Padang

1. Potong daging sapi menjadi potongan-potongan dadu atau sesuai selera.
2. Haluskan bumbu serupa bawang merah, bawang putih, cabe merah, kunyit, laos dan jahe.
3. Masukkan seluruh bumbu tadi ke santan. Tambahkan dua buah daun jeruk, satu lembar daun kunyit dan sebatang serai.
4. Masak santan di atas api sedang. Aduk terus hingga santan mendidih.
5. Masukkan daging sapi dan kecilkan api. Sekali-kali santan diaduk agar tidak pecah.
6. Jika sudah timbul minyak dan santan sudah kering, matikan api. Rendang siap dihidangkan.

Contoh Algoritma (2)

Mencari elemen terbesar pada suatu array

1. Asumsikan a_1 sebagai elemen terbesar sementara. Simpan a_1 ke dalam *maks*.
2. Bandingkan *maks* dengan elemen a_2. Jika a_2 lebih besar dari *maks*, maka nilai *maks* diganti dengan a_2.
3. Ulangi langkah 2 untuk elemen-elemen berikutnya (a_3,a_4,...,a_n).
4. Berhenti jika tidak ada lagi elemen yang dibandingkan. Dalam hal ini, *maks* berisi nilai dari elemen terbesar.

Contoh *Pseudo-code*

```
procedure CariElemenTerbesar (input a1, a2, … , an : integer,
                              output maks : integer)
{  Mencari elemen terbesar di antara elemen a1, a2, … , an.
   Elemen terbesar akan disimpan di dalam maks.
   Masukan : a1, a2, … , an
   Keluaran : maks
}
Deklarasi
   i : integer

Algoritma:
   maks a1
   for i 2 to n do
       if ai > maks then
           maks ai
       endif
   endfor
```

Kompleksitas Algoritma

Efisiensi Algoritma

- Efisiensi (Kemangkusan) Algoritma
- Algoritma yang bagus → yang mangkus
- Diukur dari berapa jumlah waktu dan ruang (*space*) memori yang dibutuhkan untuk menjalankan
- Algoritma yang mangkus adalah yang meminimumkan kebutuhan waktu dan ruang

Kebutuhan waktu dan ruang

- Kebutuhan waktu (*time*) dan ruang (*space*) bergantung pada ukuran masukan
- Biasanya adalah jumlah data yang diproses
- Ukuran masukan disimbolkan dengan n
- Contoh: untuk masalah pengurutan (*sorting*) 100 buah elemen berarti n=100

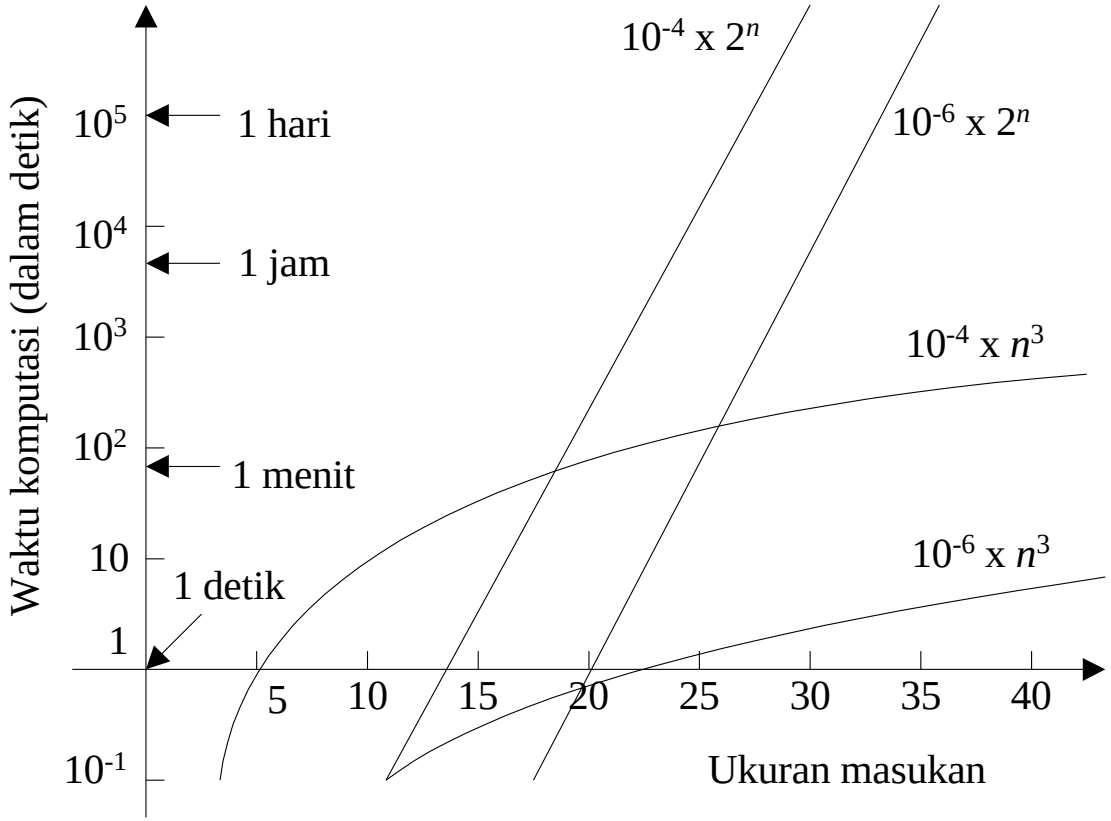
Mengapa kita perlu algoritma yang efisien?

- Misalkan, untuk menyelesaikan sebuah masalah tertentu, telah tersedia:
  - Algoritma yang waktu eksekusinya dalam orde eksponensial (2^n), dengan n adalah jumlah masukan yang diproses
  - Sebuah komputer yang mampu menjalankan program dengan masukan berukuran n dalam waktu $10^{-4} \times 2^n$ detik.
- Maka, dapat dihitung bahwa jika:
  - n=10, dibutuhkan waktu eksekusi kira-kira 1/10 detik
  - n=20, dibutuhkan waktu eksekusi kira-kira 2 menit
  - n= 30, dibutuhkan waktu eksekusi lebih dari satu hari
- Dalam setahun, hanya dapat menyelesaikan persoalan dengan masukan sebanyak 38 saja!!!

Jika kita punya algoritma yang lebih baik?

- Misalkan algoritma yang kita punya sekarang dalam waktu orde kubik (n^3)
- Masalah diselesaikan dalam $10^{-4} \times n^3$ detik
- DALAM SATU HARI SAJA:
- Kita dapat menyelesaikan lebih dari 900 masukan!!
- Dalam setahun dapat menyelesaikan 6800 lebih!!

See the difference!!



Model Perhitungan Kebutuhan Waktu

- Menghitung kebutuhan waktu algoritma dengan mengukur waktu sesungguhnya (dalam satuan detik) ketika algoritma dieksekusi oleh komputer bukan cara yang tepat.
- Alasan:
  1. Setiap komputer dengan arsitektur berbeda mempunyai bahasa mesin yang berbeda → waktu setiap operasi antara satu komputer dengan komputer lain tidak sama.
  2. *Compiler* bahasa pemrograman yang berbeda menghasilkan kode mesin yang berbeda → waktu setiap operasi antara compiler dengan compiler lain tidak sama.

Model Abstrak

- Model abstrak pengukuran waktu/ruang harus independen dari pertimbangan mesin dan compiler apapun.

- Besaran yang dipakai untuk menerangkan model abstrak pengukuran waktu/ruang ini adalah **kompleksitas algoritma.**

- Ada dua macam kompleksitas algoritma, yaitu: **kompleksitas waktu** dan **kompleksitas ruang.**

- Kompleksitas waktu, ***T*(*n*)**, diukur dari jumlah tahapan komputasi yang dibutuhkan untuk menjalankan algoritma sebagai fungsi dari ukuran masukan *n*.
- Kompleksitas ruang, ***S*(*n*)**, diukur dari memori yang digunakan oleh struktur data yang terdapat di dalam algoritma sebagai fungsi dari ukuran masukan *n*.
- Dengan menggunakan besaran kompleksitas waktu/ruang algoritma, kita dapat menentukan *laju* peningkatan waktu (ruang) yang diperlukan algoritma dengan meningkatnya ukuran masukan *n*.

Ukuran masukan (*n*): jumlah data yang diproses oleh sebuah algoritma.

- Contoh: algoritma pengurutan 1000 elemen larik, maka $n = 1000$.
- Contoh: algoritma *TSP* pada sebuah graf lengkap dengan 100 simpul, maka $n = 100$.
- Contoh: algoritma perkalian 2 buah matriks berukuran 50 x 50, maka $n = 50$.
- Dalam praktek perhitungan kompleksitas, ukuran masukan dinyatakan sebagai variabel *n* saja.

Kompleksitas Waktu

- Jumlah tahapan komputasi dihitung dari berapa kali suatu operasi dilaksanakan di dalam sebuah algoritma sebagai fungsi ukuran masukan (n).

- Di dalam sebuah algoritma terdapat bermacam jenis operasi:
  - Operasi baca/tulis
  - Operasi aritmetika (+, -, *, /)
  - Operasi pengisian nilai (*assignment*)
  - Operasi pengakasesan elemen larik
  - Operasi pemanggilan fungsi/prosedur
  - dll

- Dalam praktek, kita hanya menghitung jumlah operasi khas (tipikal) yang *mendasari* suatu algoritma.

Contoh operasi khas di dalam algoritma

- Algoritma pencarian di dalam larik

  Operasi khas: perbandingan elemen larik
- Algoritma pengurutan

  Operasi khas: perbandingan elemen, pertukaran elemen
- Algoritma penjumlahan 2 buah matriks

  Operasi khas: penjumlahan
- Algoritma perkalian 2 buah matriks

  Operasi khas: perkalian dan penjumlahan

- **Contoh 1.** Tinjau algoritma menghitung rerata sebuah larik (*array*).

```
sum ←0
for i ← 1 to n do
     sum ← sum + a[i]
endfor
rata_rata ← sum/n
```

- Operasi yang mendasar pada algoritma tersebut adalah operasi penjumlahan elemen-elemen a_i (yaitu `sum←sum+a[i]`) yang dilakukan sebanyak *n* kali.

- Kompleksitas waktu: $T(n) = n$.

Contoh 2. Algoritma untuk mencari elemen terbesar di dalam sebuah larik (*array*) yang berukuran *n* elemen.

```
procedure CariElemenTerbesar(input a1, a2, ..., an : integer, output
maks : integer)
{ Mencari elemen terbesar dari sekumpulan elemen larik integer a1, a2,
..., an.
  Elemen terbesar akan disimpan di dalam maks.
  Masukan: a1, a2, ..., an
  Keluaran: maks (nilai terbesar)
}
Deklarasi
   k : integer

Algoritma
   maks←a1
   k←2
   while k ≤ n do
     if ak > maks then
        maks←ak
     endif
     i←i+1
   endwhile
   { k > n }
```

Kompleksitas waktu algoritma dihitung berdasarkan jumlah operasi perbandingan elemen larik (`A[i] > maks`).

Kompleksitas waktu `CariElemenTerbesar` : $T(n) = n - 1$.

Kompleksitas waktu dibedakan atas tiga macam :

- $T_{max}(n)$: kompleksitas waktu untuk kasus terburuk *(worst case)*, → kebutuhan waktu maksimum.
- $T_{min}(n)$: kompleksitas waktu untuk kasus terbaik *(best case)*, → kebutuhan waktu minimum.
- $T_{avg}(n)$: kompleksitas waktu untuk kasus rata-rata *(average case)* → kebutuhan waktu secara rata-rata

Contoh 3. Algoritma *sequential search*.

```
procedure PencarianBeruntun(input a1, a2, ..., an : integer, x : integer,
                            output idx : integer)
Deklarasi
  k : integer
  ketemu : boolean    { bernilai true jika x ditemukan atau false jika x
tidak ditemukan }

Algoritma:
  k←1
  ketemu ← false
  while (k ≤ n) and (not ketemu) do
    if ak = x then
      ketemu←true
    else
      k ← k + 1
    endif
  endwhile
  { k > n or ketemu }

  if ketemu then    { x ditemukan }
     idx←k
  else
     idx← 0         { x tidak ditemukan }
  endif
```

Jumlah operasi perbandingan:

1. *Kasus terbaik*: ini terjadi bila $a_1 = x$.

$T_{min}(n) = 1$

2. *Kasus terburuk*: bila $a_n = x$ atau x tidak ditemukan.

$T_{max}(n) = n$

Contoh 4. Algoritma pencarian biner (*bynary search*).

```
procedure PencarianBiner(input a1, a2, ..., an : integer, x : integer,
                         output idx : integer)
Deklarasi
   i, j, mid : integer
   ketemu : boolean

Algoritma
   i←1
   j←n
   ketemu←false
   while (not ketemu) and ( i ≤ j) do
      mid ← (i+j) div 2
      if amid = x then
        ketemu ← true
      else
         if amid < x then     { cari di belahan kanan }
          i←mid + 1
         else                    { cari di belahan kiri }
          j←mid - 1;
         endif
      endif
   endwhile
   {ketemu or i > j }

   if ketemu then
      idx←mid
   else
      idx←0
   endif
```

1. *Kasus terbaik*

$$T_{min}(n) = 1$$

2. *Kasus terburuk*:

$$T_{max}(n) = {}^2\log n$$

Contoh 5. Algoritma pengurutan seleksi (*selection sort*).

```
procedure Urut(input/output a1, a2, ..., an : integer)
Deklarasi
   i, j, imaks, temp : integer

Algoritma
  for i←n downto 2 do    { pass sebanyak n - 1 kali }
    imaks←1
    for j←2 to i do
      if aj > aimaks then
         imaks←j
      endif
    endfor
    { pertukarkan aimaks dengan ai }
    temp←ai
    ai←aimaks
    aimaks←temp

  endfor
```

Untuk setiap *i* dari 1 sampai *n* – 1, terjadi satu kali pertukaran elemen, sehingga jumlah operasi pertukaran seluruhnya adalah $T(n) = n - 1$.

Kompleksitas $O(1)$ berarti waktu pelaksanaan algoritma adalah tetap, tidak bergantung pada ukuran masukan. Contohnya prosedur `tukar` di bawah ini:

```
procedure tukar(var a:integer; var b:integer);
var
  temp:integer;
begin
  temp:=a;
  a:=b;
  b:=temp;
end;
```

Di sini jumlah operasi penugasan (*assignment*) ada tiga buah dan tiap operasi dilakukan satu kali. Jadi, $T(n) = 3 = O(1)$.

Kompleksitas Waktu Asimptotik

- Tinjau $T(n) = 2n^2 + 6n + 1$

Perbandingan pertumbuhan $T(n)$ dengan n^2

n	$T(n) = 2n^2 + 6n + 1$	n^2
10	261	100
100	2061	1000
1000	2.006.001	1.000.000
10.000	2.000.060.001	1.000.000.000

- Untuk n yang besar, pertumbuhan $T(n)$ sebanding dengan n^2. Pada kasus ini, $T(n)$ tumbuh seperti n^2 tumbuh.

- $T(n)$ tumbuh seperti n^2 tumbuh saat n bertambah. Kita katakan bahwa $T(n)$ berorde n^2 dan kita tuliskan

$$T(n) = O(n^2)$$

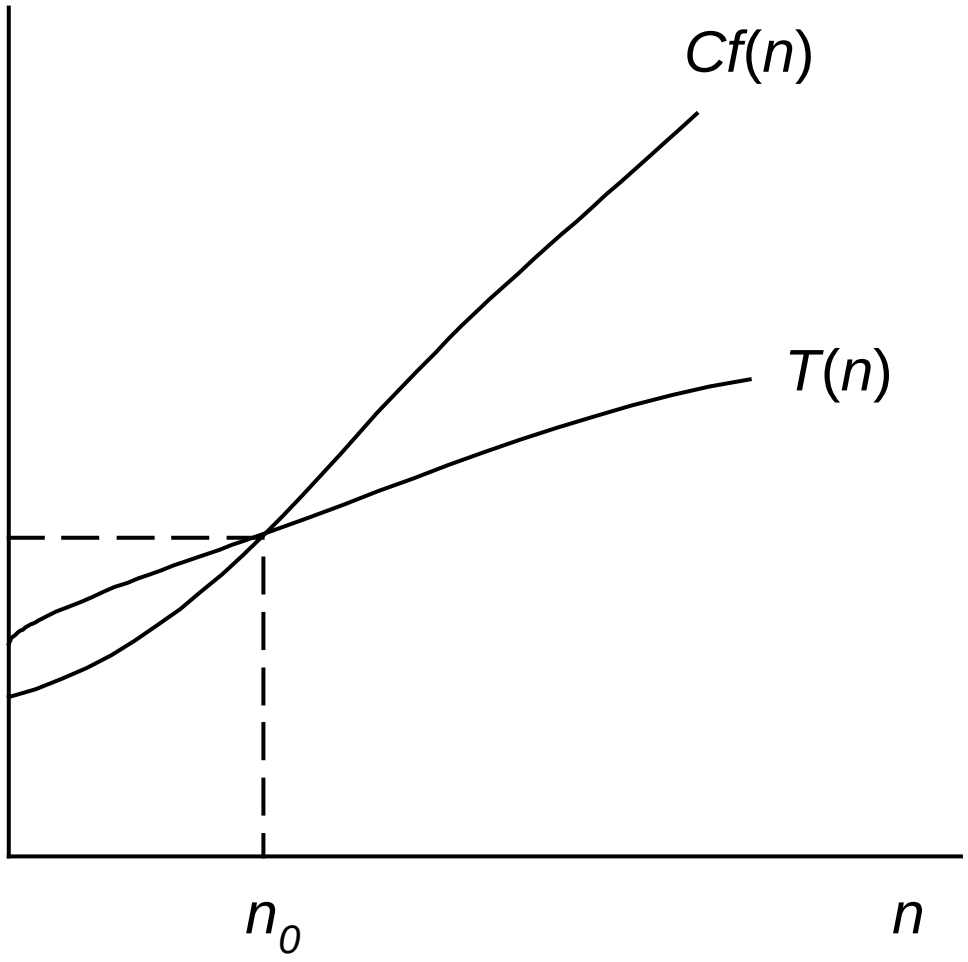
Notasi "***O***" disebut notasi "***O*-Besar**" (***Big*-*O***) yang merupakan notasi **kompleksitas waktu asimptotik**.

DEFINISI. $T(n) = O(f(n))$ (dibaca "$T(n)$ adalah $O(f(n)$" yang artinya *T(n) berorde paling besar f(n)*) bila terdapat konstanta C dan n_0 sedemikian sehingga

$$T(n) \leq C(f(n))$$

untuk $n \geq n_0$.

$f(n)$ adalah batas lebih atas (*upper bound*) dari $T(n)$ untuk n yang besar.


Cf(n)
T(n)
n0
n

- **Teorema**: Bila $T(n) = a_m\, n^m + a_{m-1}\, n^{m-1} + \ldots + a_1 n + a_0$ adalah polinom derajat m maka $T(n) = O(n^m)$.

- **Jadi, cukup melihat suku (*term*) yang mempunyai pangkat terbesar.**

- Contoh:

  $T(n) = 5 = 5n^0 = O(n^0) = O(1)$

  $T(n) = n(n-1)/2 = n^2/2 - n/2 = O(n^2)$

  $T(n) = 3n^3 + 2n^2 + 10 = O(n^3)$

Teorema tersebut digeneralisasi untuk suku dominan lainnya:

1. Eksponensial mendominasi sembarang perpangkatan (yaitu, $y^n > n^p$, $y > 1$)
2. Perpangkatan mendominasi ln n (yaitu $n^p > \ln n$)
3. Semua logaritma tumbuh pada laju yang sama (yaitu $a \log(n) = b \log(n)$)
4. $n \log n$ tumbuh lebih cepat daripada n tetapi lebih lambat daripada n^2

Contoh: $T(n) = 2^n + 2n^2 = O(2^n)$.

$T(n) = 2n \log(n) + 3n = O(n \log(n))$

$T(n) = \log(n^3) = 3 \log(n) = O(\log(n))$

$T(n) = 2n \log(n) + 3n^2 = O(n^2)$

TEOREMA. Misalkan $T_1(n) = O(f(n))$ dan $T_2(n) = O(g(n))$, maka

(a) $T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(\max(f(n), g(n))$

(b) $T_1(n)T_2(n) = O(f(n))O(g(n)) = O(f(n)g(n))$

(c) $O(cf(n)) = O(f(n))$, c adalah konstanta

(d) $f(n) = O(f(n))$

Contoh 9. Misalkan $T_1(n) = O(n)$ dan $T_2(n) = O(n^2)$, maka

(a) $T_1(n) + T_2(n) = O(\max(n, n^2)) = O(n^2)$

(b) $T_1(n)T_2(n) = O(n.n^2) = O(n^3)$

Contoh 10. $O(5n^2) = O(n^2)$

$n^2 = O(n^2)$

Pengelompokan Algoritma Berdasarkan Notasi *O*-Besar

Kelompok Algoritma	Nama
$O(1)$	konstan
$O(\log n)$	logaritmik
$O(n)$	lanjar
$O(n \log n)$	$n \log n$
$O(n^2)$	kuadratik
$O(n^3)$	kubik
$O(2^n)$	eksponensial
$O(n!)$	faktorial

Urutan spektrum kompleksitas waktu algoritma adalah :

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

$O(\log n)$ Kompleksitas waktu logaritmik berarti laju pertumbuhan waktunya berjalan lebih lambat daripada pertumbuhan n. Algoritma yang termasuk kelompok ini adalah algoritma yang memecahkan persoalan besar dengan mentransformasikannya menjadi beberapa persoalan yang lebih kecil yang berukuran sama (misalnya algoritma `pencarian_biner`). Di sini basis algoritma tidak terlalu penting sebab bila n dinaikkan dua kali semula, misalnya, log n meningkat sebesar sejumlah tetapan.

$O(n)$ Algoritma yang waktu pelaksanaannya lanjar (linear) umumnya terdapat pada kasus yang setiap elemen masukannya dikenai proses yang sama, misalnya algoritma `pencarian_beruntun`. Bila n dijadikan dua kali semula, maka waktu pelaksanaan algoritma juga dua kali semula.

O(*n* log *n*) Waktu pelaksanaan yang *n* log *n* terdapat pada algoritma yang memecahkan persoalan menjadi beberapa persoalan yang lebih kecil, menyelesaikan tiap persoalan secara independen, dan menggabung solusi masing-masing persoalan. Algoritma yang diselesaikan dengan teknik bagi dan gabung (divide and conquer) mempunyai kompleksitas asimptotik jenis ini. Bila *n* = 1000, maka *n* log *n* mungkin 20.000. Bila *n* dijadikan dua kali semual, maka *n* log *n* menjadi dua kali semula (tetapi tidak terlalu banyak)

Contoh: Mergesort, QuickSort

$O(n^2)$ Algoritma yang waktu pelaksanaannya kuadratik hanya praktis digunakan untuk persoalana yang berukuran kecil. Umumnya algoritma yang termasuk kelompok ini memproses setiap masukan dalam dua buah kalang bersarang, misalnya pada algoritma urut_maks. Bila n = 1000, maka waktu pelaksanaan algoritma adalah 1.000.000. Bila n dinaikkan menjadi dua kali semula, maka waktu pelaksanaan algoritma meningkat menjadi empat kali semula.

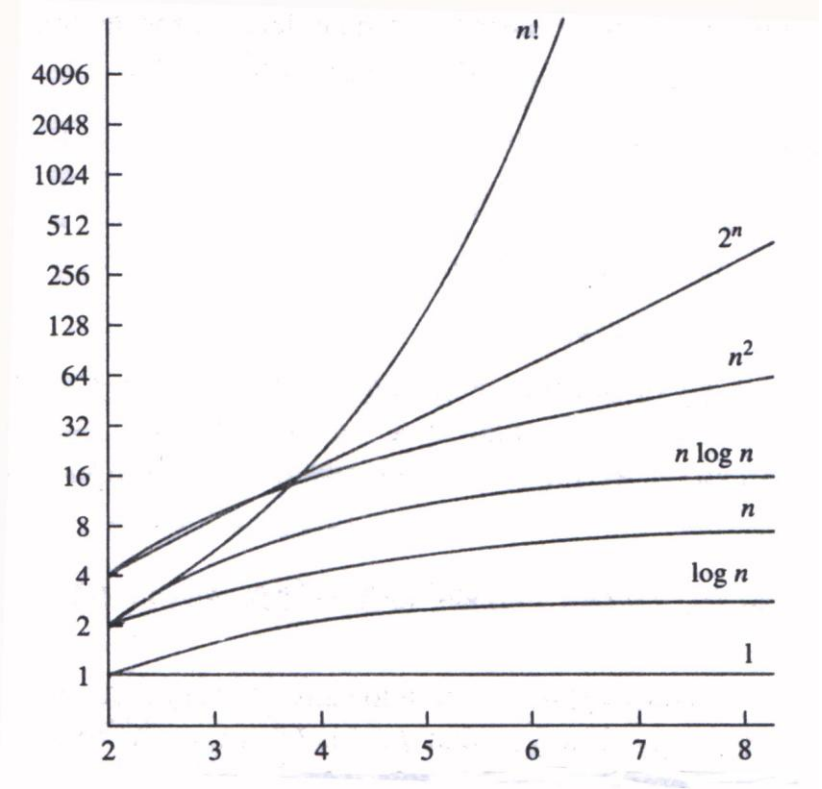
$O(n^3)$ Seperti halnya algoritma kuadratik, algoritma kubik memproses setiap masukan dalam tiga buah kalang bersarang, misalnya algoritma perkalian matriks. Bila n = 100, maka waktu pelaksanaan algoritma adalah 1.000.000. Bila n dinaikkan menjadi dua kali semula, waktu pelaksanan algoritma meningkat menjadi delapan kali semula.

$O(2^n)$ Algoritma yang tergolong kelompok ini mencari solusi persoalan secara "*brute force*", misalnya pada algoritma mencari sirkuit Hamilton (lihat Bab 9). Bila $n = 20$, waktu pelaksanaan algoritma adalah 1.000.000. Bila n dijadikan dua kali semula, waktu pelaksanaan menjadi kuadrat kali semula!

$O(n!)$ Seperti halnya pada algoritma eksponensial, algoritma jenis ini memproses setiap masukan dan menghubungkannya dengan n - 1 masukan lainnya, misalnya algoritma Persoalan Pedagang Keliling (*Travelling Salesperson Problem* - lihat bab 9). Bila $n = 5$, maka waktu pelaksanaan algoritma adalah 120. Bila n dijadikan dua kali semula, maka waktu pelaksanaan algoritma menjadi faktorial dari $2n$.

Nilai masing-masing fungsi untuk setiap bermacam-macam nilai *n*

log n	*n*	*n log n*	n^2	n^3	2^n	*n*!
0	1	0	1	1	2	1
1	2	2	4	8	4	2
2	4	8	16	64	16	24
3	9	24	64	512	256	362880
4	16	64	256	4096	65536	20922789888000
5	32	160	1024	32768	4294967296	(terlalu besar)



Kegunaan Notasi *Big-Oh*

- Notasi *Big-Oh* berguna untuk membandingkan beberapa algoritma dari untuk masalah yang sama

  → menentukan yang terbaik.

- Contoh: masalah pengurutan memiliki banyak algoritma penyelesaian,

  Selection sort, *insertion sort* → $T(n) = O(n^2)$

  Quicksort → $T(n) = O(n \log n)$

  Karena $n \log n < n^2$ untuk n yang besar, maka algoritma *quicksort* lebih cepat (lebih baik, lebih mangkus) daripada algoritma *selection sort* dan *insertion sort*.

Referensi

- Munir, R., 2005, Matematika Diskrit, Penerbit IF, Bandung
- A. Rosen, H Kenneth (2012). Discrete Mathematics and Its Applications. Mc Graw Hill.
- Siang, J.J., 2002, Matematika Diskrit dan Aplikasinya pada Ilmu Komputer