KOMPLEKSITAS ALGORITMA

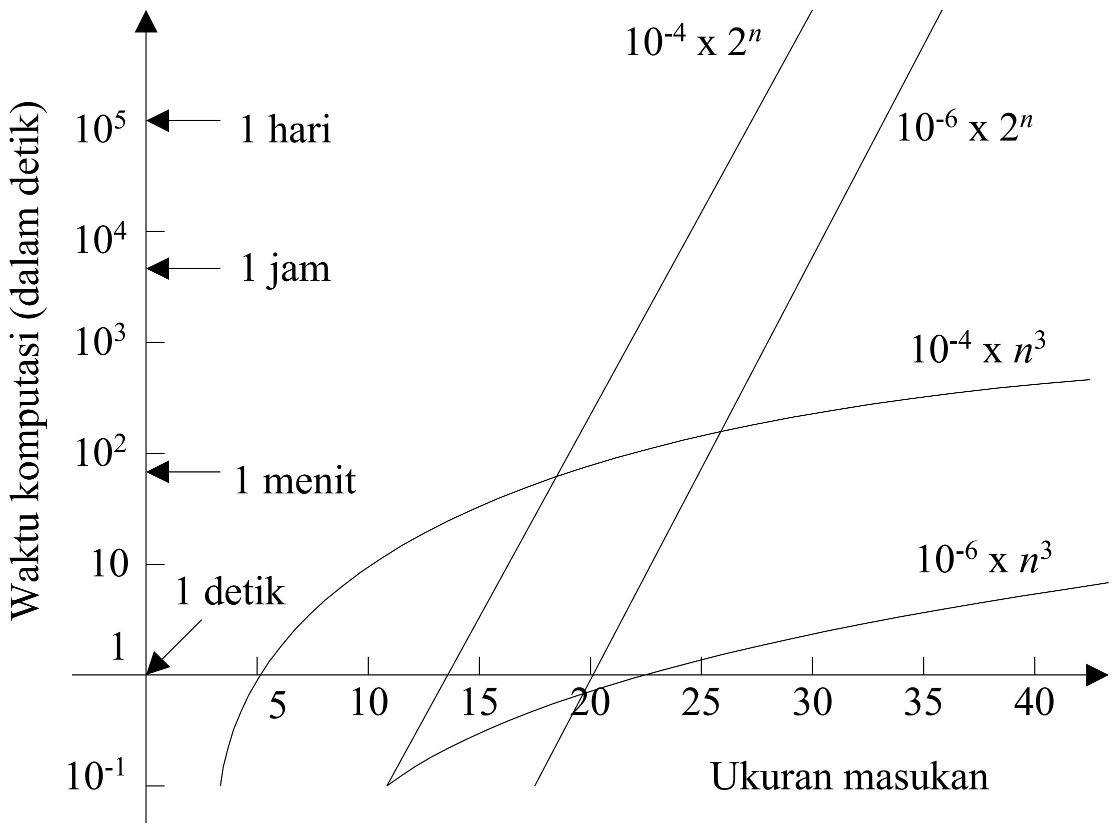
ANALISIS ALGORITMA DAN KOMPLEKSITAS

EFISIENSI ALGORITMA

- **Sebuah masalah dapat mempunyai banyak algoritma penyelesaian → algoritma tidak tunggal**
  - Contoh: masalah pengurutan (*sorting*), ada puluhan algoritma pengurutan
- **Ukuran input bisa sangat besar.**
- **Membandingkan algoritma:**
  - Dari definisi domain: input apa yang legal
  - Dari *correctness*: apakah dapat menyelesaikan semua input yang legal
  - Dari efisiensi: waktu komputasi, memori yang diperlukan, resource lainnya
- **Sebuah algoritma tidak saja harus benar, tetapi juga harus *efisien*.**
  - Algoritma yang bagus adalah algoritma yang efisien.
    - meminimumkan kebutuhan waktu dan ruang.

- **Efisiensi algoritma pada umumnya diukur dari waktu (*time*) eksekusi algoritma dan kebutuhan ruang (*space*) memori.**
- **Kebutuhan waktu dan ruang suatu algoritma bergantung pada ukuran masukan (*n*), yang menyatakan jumlah data yang diproses.**
- **Dari perbandingan efisiensi algoritma, dapat dipilih algoritma yang terbaik.**

- **Mengapa kita memerlukan algoritma yang efisien?**



MODEL PERHITUNGAN KEBUTUHAN WAKTU

- **Ada 3 macam: empiris, teoritis, hybrid.**
- **Menghitung kebutuhan waktu algoritma dengan mengukur waktu sesungguhnya ketika algoritma dieksekusi oleh komputer (disebut cara *empiris*) 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 pengukuran waktu/ruang harus independen dari pertimbangan mesin dan *compiler* apapun (cara teoritis).**
- **Cara gabungan: hybrid**

- **Cara teoritis:**
  - Menentukan secara matematis jumlah sumber daya yang diperlukan untuk setiap algoritma.
  - Jumlah diekspresikan sebagai fungsi dari ukuran *instances*.
  - Resources: waktu komputasi, *space* penyimpanan, bandwidth komunikasi, hardware komputer.
- **Keuntungan cara teoritis:**
  - Independen terhadap jenis komputer, jenis bahasa pemrograman, kemampuan pemrogram.
  - Menghemat waktu yang dipakai untuk mengimplementasikan algoritma yang tidak efisien dan waktu untuk mengujinya.
  - Memungkinkan menganalisis algoritma ybs dengan instances sangat besar.
- **Prinsip invariance: efisiensi dari dua implementasi berbeda suatu algoritma tidak akan berselisih lebih dari suatu multiplikatif konstan.**
  - $T_1(n) \leq c\ T_2(n)$ dan $T_2(n) \leq d\ T_1(n)$

KOMPLEKSITAS ALGORITMA

- **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 (INPUT)

- **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.**
  - Algoritma tentang graf: n adalah jumlah verteks atau jumlah sisi
  - Algoritma tentang image processing: n adalah jumlah pixel (2D image) atau voxel (3D image)
  - Text processing: n adalah jumlah karakter (panjang string yang diinputkan)

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 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

CONTOH

- **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 CariElemenMaks(input a1, a2, ..., an : integer,
output maks : integer)
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.

TIPE KOMPLEKSITAS WAKTU

Kompleksitas waktu dibedakan atas tiga macam :

1. *T_{max}(n)* : kompleksitas waktu untuk kasus terburuk *(worst case)*,

→ kebutuhan waktu maksimum.

2. *T_{min}(n)* : kompleksitas waktu untuk kasus terbaik *(best case)*,

→ kebutuhan waktu minimum.

3. *T_{avg}(n)*: kompleksitas waktu untuk kasus rata-rata *(average case)*

→ kebutuhan waktu secara rata-rata

Contoh: 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
```

Analisis operasi perbandingan elemen tabel:

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

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

3. ***Kasus rata-rata***: **Jika *x* ditemukan pada posisi ke-*j*, maka operasi perbandingan (a_k = x) akan dieksekusi sebanyak *j* kali.**

ANALISIS AVERAGE-CASE VS WORST-CASE

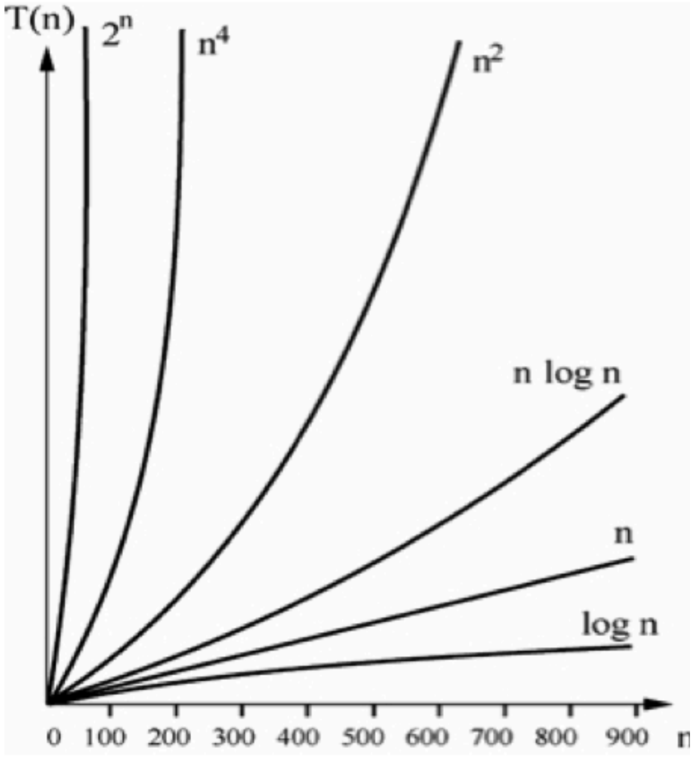
- **Analisis average case sesuai untuk algoritma yang sering digunakan dengan instances yang berbeda.**
- **Meskipun demikian, biasanya analisis difokuskan pada worst case.**
  - Memberikan batas atas dari running time untuk sembarang instance.
  - Untuk beberapa algoritma, worst-case terjadi sangat sering. Contoh: pencarian pada basis data.
  - Sangat penting untuk algoritma di mana response time diutamakan.
  - Average case sering kali sama buruknya dengan worst case.

ORDER

- **Running time: jumlah operasi khas atau langkah yang harus dilakukan.**
- **Suatu algoritma memerlukan waktu dalam order T(n) jika terdapat konstanta positif c dan implementasi algoritma mampu untuk menyelesaikan setiap *instance* berukuran n dalam waktu tidak lebih dari c T(n) satuan waktu.**
- **Implementasi yang lain dari algoritma tersebut juga dalam order T(n) → akibat dari prinsip invariance.**
- **Macam-macam order algoritma:**
  - Algoritma logaritmik : log n
  - Algoritma linier : n
  - Algoritma linieritmik: n log n
  - Algoritma kuadratik: n^2
  - Algoritma kubik : n^3
  - Algoritma polinomial: n^k, k konstanta
  - Algoritma eksponensial: c^n, c konstanta

KURVA KOMPLEKSITAS

- **Pebandingan kurva dengan berbagai macam kompleksitas**



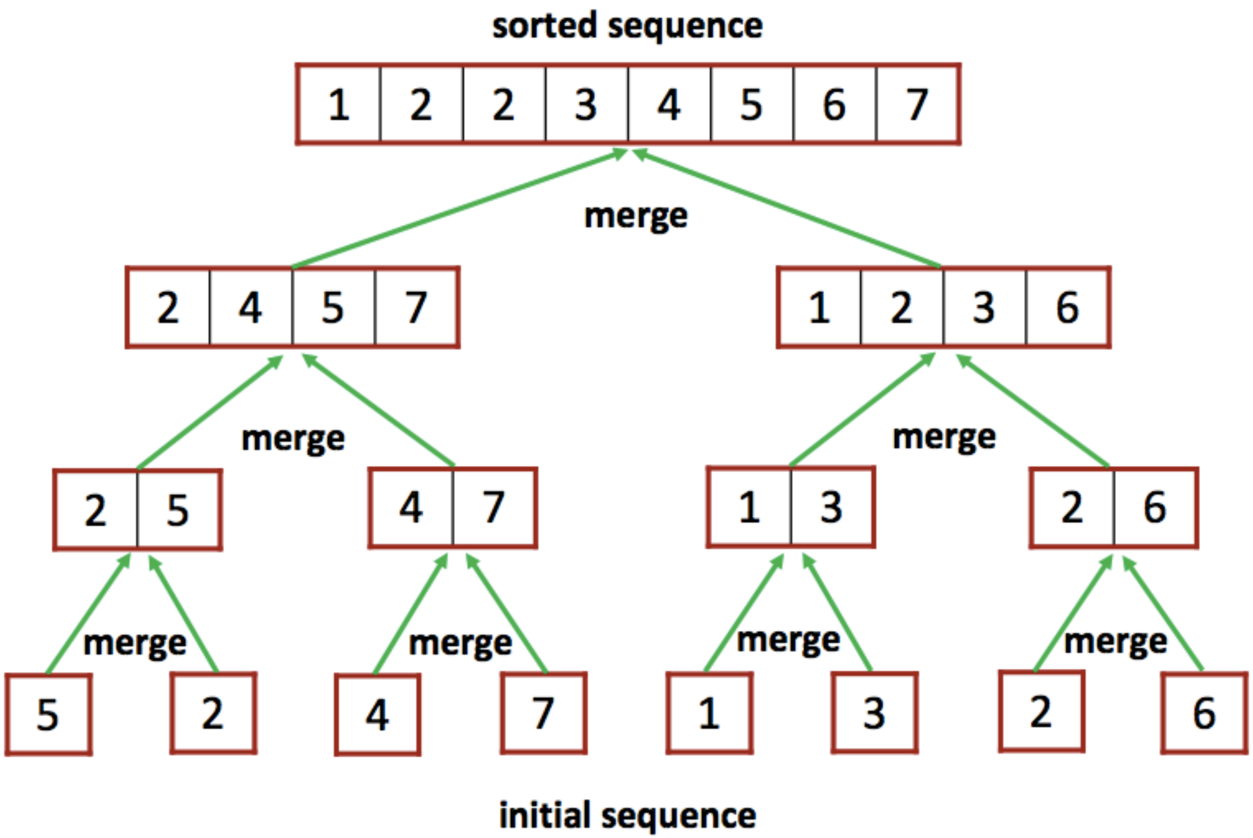
MENDISAIN ALGORITMA

- **Beberapa cara mendisain algoritma:**
  - Incremental:
    - Contoh: insertion sort, setelah mengurutkan subarray $A[1 \ldots j-1]$, kemudian akan menghasilkan array yang sudah terurut $A[1 \ldots j]$.
  - Divide and conquer
    - Divide: Membagi masalah menjadi sejumlah sub-masalah
    - Conquer: Menyelesaikan beberapa sub-masalah tersebut secara rekursif.
    - Jika ukuran sub-masalah kecil, selesaikan dengan cara yang straightforward.
    - Combine: Gabungkan penyelesaian-penyelesaian dari sub-masalah untuk menyelesaikan masalah awal.

CONTOH: MERGE SORT

- **Divide: Membagi masalah dengan membagi array menjadi 2 sub-array: *A*[*p*...*q*] dan *A*[*q*+1...*r*], dengan *q* adalah tengah-tengah dari *A*[*p*...*r*].**
- **Conquer: menyelesaikan pengurutan 2 sub-array *A*[*p*...*q*] dan *A*[*q*+1...*r*] secara rekursif.**
- **Combine: dengan merging 2 sub-rray yang terurut untuk menghasilkan jawaban yang terurut.**
- **Algoritma:**
  - MERGE-SORT (*A*, *p*, *r*)
    1. if $p < r$ //Ceck untuk base case
    2. then $q \leftarrow (p+r)/2 \lfloor\rfloor$ //Divide
    3. MERGE-SORT(*A*, *p*, *q*) //Conquer
    4. MERGE-SORT(*A*, *q* + 1, *r*) //Conquer
    5. MERGE(*A*, *p*, *q*, *r*) //Combine
  - Pemanggilan awal: MERGE-SORT(*A*, 1, *n*)

- **Ilustrasi merge sort:**



LATIHAN

- **Tentukan operasi khas untuk permasalahan:**
  - Algoritma perkalian 2 buah matriks
  - Algoritma kruskal (mendapatkan minimum spanning tree)
- **Tentukan kasus terbaik dan kasus terburuk untuk algoritma insertion sort seperti berikut ini:**
  - INSERTION-SORT(A)

```
for j ← 2 to length[A] do
    key ← A[j]
    /* sisipkan A[j] ke sekuens terurut A[1... j − 1]*/
    i ← j −1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i −1
    A[i + 1] ← key
```