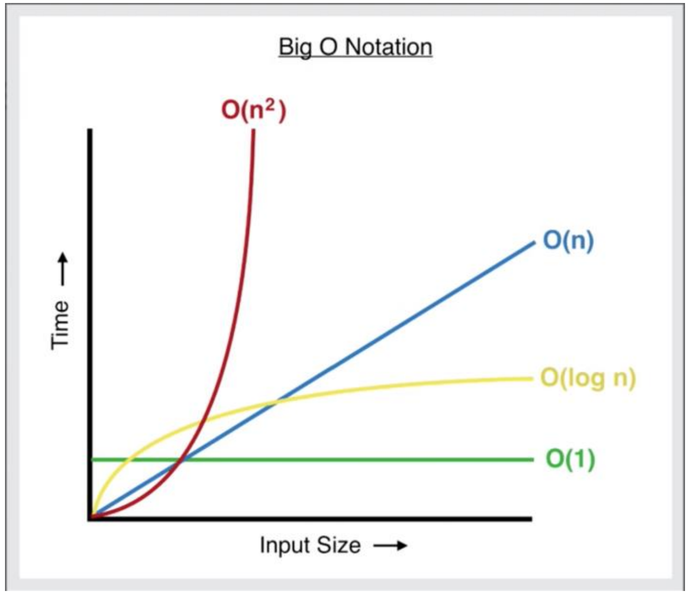


Kompleksitas Algoritma (Bagian 1)

(Update 2023)

Bahan Kuliah

IF2120 Matematika Diskrit

Oleh: Rinaldi Munir

Program Studi Teknik Informatika

STEI - ITB

```
for (i = 1; i <= n, i++)   {
    for (j =1; j <= n; j++) {
         for (k =1; k <= j; k++) {
            p = p * 20 * z;
         }
    }
}
```

Pendahuluan

- Sebuah algoritma tidak saja harus benar (sesuai spesifikasi persoalan), tetapi juga harus sangkil (*efisien*).

- Algoritma yang bagus adalah algoritma yang sangkil (*efficient*).

- Kesangkilan algoritma diukur dari waktu (*time*) yang diperlukan untuk menjalankan algoritma dan ruang (*space*) memori yang dibutuhkan oleh algoritma tersebut.

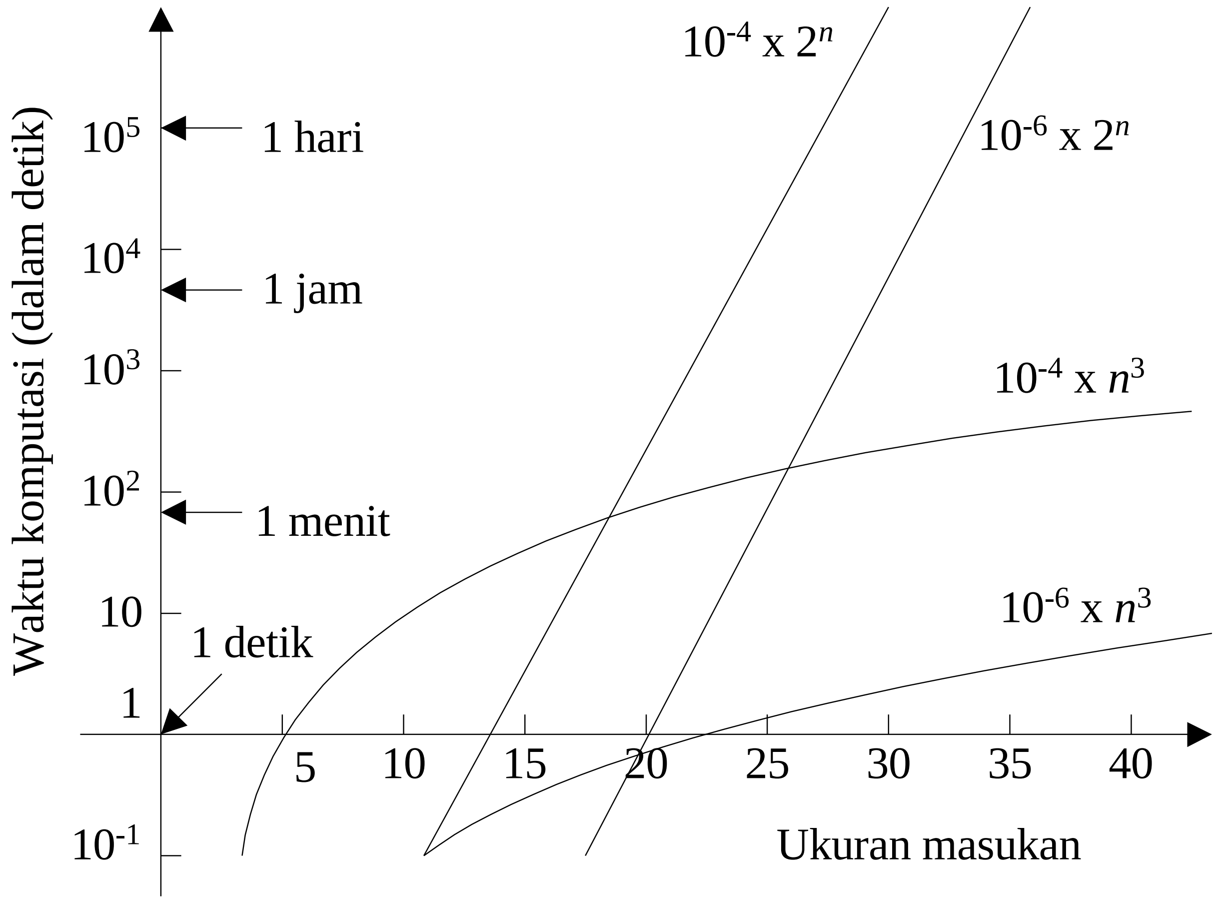
- Algoritma yang sangkil ialah algoritma yang **meminimumkan** kebutuhan waktu dan ruang memori.

- Kebutuhan waktu dan ruang memori suatu algoritma bergantung pada ukuran masukan (*n*), yang menyatakan ukuran data yang diproses oleh algoritma.

- Kesangkilan algoritma dapat digunakan untuk menilai algoritma yang bagus dari sejumlah algoritma penyelesaian persoalan.

- Sebab, sebuah persoalan dapat memiliki banyak algoritma penyelesaian. Contoh: persoalan pengurutan (*sort*), ada puluhan algoritma pengurutan (*selection sort*, *insertion sort*, *bubble sort*, dll).

- Mengapa kita memerlukan algoritma yang sangkil? Lihat grafik di bawah ini.



Model Perhitungan Kebutuhan Waktu

- Menghitung kebutuhan waktu algoritma dengan mengukur waktu eksekusi riil nya (dalam satuan detik) ketika program (yang merepresentasikan sebuah algoritma) dijalankan oleh komputer bukanlah cara yang tepat.

- Alasan:

  1. Setiap komputer dengan arsitektur berbeda memiliki bahasa mesin yang berbeda → akibatnya, waktu setiap operasi antara satu komputer dengan komputer lain tidak sama.

  2. *Compiler* bahasa pemrograman yang berbeda menghasilkan kode bahasa mesin yang berbeda → akibatnya, waktu setiap operasi antara *compiler* dengan *compiler* lain tidak sama.

- Model abstrak pengukuran waktu/ruang memori algoritma harus independen dari pertimbangan mesin (*computer*) 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** (*time complexity*) dan **kompleksitas ruang** (*space complexity*).

- Kompleksitas waktu, $T(n)$, diukur dari jumlah tahapan komputasi yang dilakukan di dalam algoritma sebagai fungsi dari ukuran masukan n.

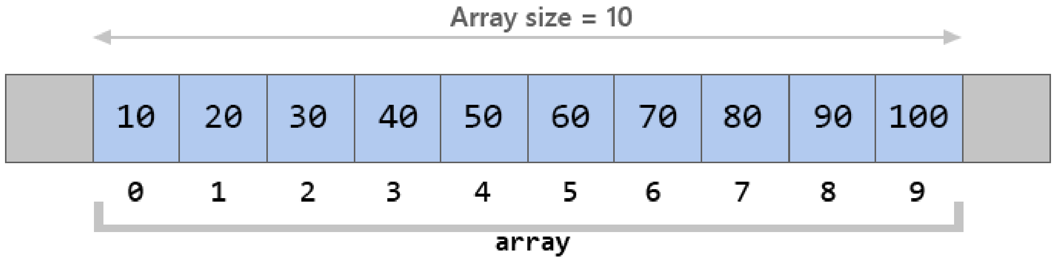
- Kompleksitas ruang, $S(n)$, diukur dari memori yang digunakan oleh struktur data yang terdapat di dalam algoritma sebagai fungsi dari ukuran masukan n.

- Dengan menggunakan besaran kompleksitas waktu/ruang algoritma, kita dapat menentukan *laju* peningkatan waktu (ruang) yang diperlukan algoritma dengan meningkatnya ukuran masukan n.

- Di dalam kuliah ini kita hanya membatasi bahasan kompleksitas waktu saja, karena dua alasan:

  1. Materi struktur data diluar lingkup mata kuliah matematika diskrit
  2. Saat ini memori komputer bukan persoalan yang kritis dibandingkan waktu

- Ukuran masukan (*n*) menyatakan banyaknya data yang diproses oleh sebuah algoritma.



Contoh:

1. algoritma pengurutan 10 elemen larik (*array*), maka $n = 10$.
2. algoritma pencarian pada 500 elemen larik, maka $n = 500$
3. algoritma *TSP* pada sebuah graf lengkap dengan 100 simpul, maka $n = 100$.
4. algoritma perkalian 2 buah matriks berukuran 50 x 50, maka $n = 50$.
5. algoritma menghitung polinom dengan derajat ≤ 100, maka $n = 100$

- Dalam perhitungan kompleksitas waktu, ukuran masukan dinyatakan sebagai variabel *n* saja (bukan instans suatu nilai).

Kompleksitas Waktu

- Pekerjaan utama di dalam kompleksitas waktu adalah menghitung (*counting*) jumlah tahapan komputasi di dalam algoritma .

- Jumlah tahapan komputasi dihitung dari berapa kali suatu operasi dilakukan sebagai fungsi ukuran masukan (*n*).

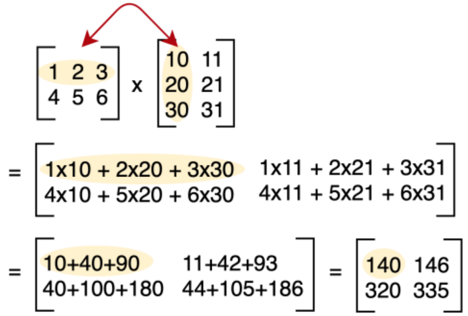
- Di dalam sebuah algoritma terdapat banyak jenis operasi:
  - Operasi baca/tulis (input a, print a)
  - Operasi aritmetika (+, -, *, /) (a + b, M * N)
  - Operasi pengisian nilai (*assignment*) (a ← 10)
  - Operasi perbandingan (a < b, k >= 10)
  - Operasi pengaksesan elemen larik, pemanggilan prosedur/fungsi, dll

- Untuk menyederhanakan perhitungan, kita tidak menghitung semua jenis operasi, tetapi kita hanya menghitung jumlah operasi khas (tipikal) yang *mendasari* suatu algoritma.

Contoh operasi khas di dalam algoritma

- Algoritma pencarian (*searching*)
  Operasi khas: operasi perbandingan elemen larik

- Algoritma pengurutan (*sorting*)
  Operasi khas: operasi perbandingan elemen dan operasi pertukaran elemen



- Algoritma perkalian dua buah matriks *AB* = *C*
  Operasi khas: operasi perkalian dan penjumlahan

- Algoritma menghitung nilai sebuah polinom $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$
  Operasi khas: operasi perkalian dan penjumlahan

Contoh 1. Tinjau algoritma menghitung rerata elemen di dalam sebuah larik (*array*).

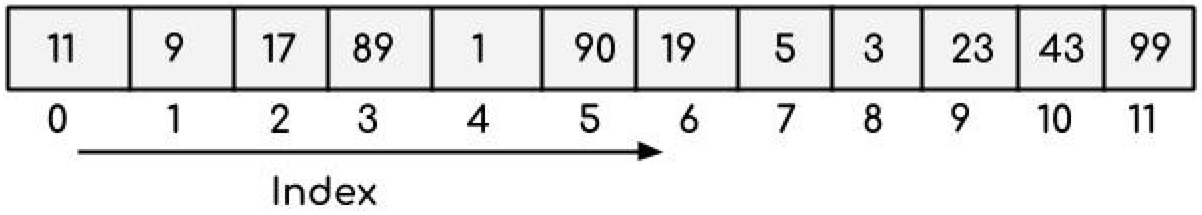
$sum \leftarrow 0$

for $i \leftarrow 1$ **to** n **do**

$\quad sum \leftarrow sum + a[i]$

endfor

$rata\_rata \leftarrow sum/n$

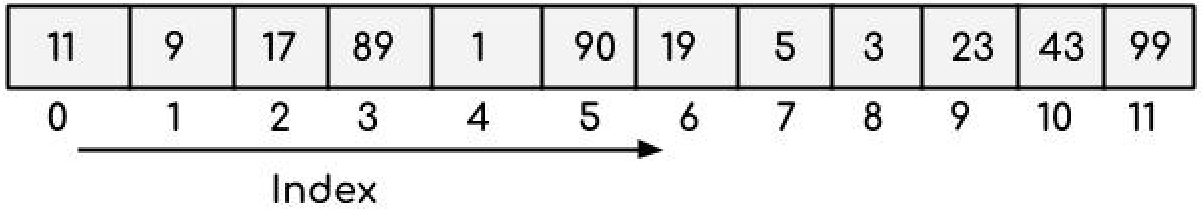


- Operasi yang mendasar pada algoritma tersebut adalah operasi penjumlahan elemen-elemen larik (yaitu $sum \leftarrow sum+a[i]$) yang dilakukan sebanyak *n* kali.

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

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

procedure *CariElemenTerbesar*(**input** $a_1, a_2, ..., a_n$: **integer**, **output** *maks* : **integer**)
{ Mencari elemen terbesar dari sekumpulan elemen larik integer $a_1, a_2, ..., a_n$.
Elemen terbesar akan disimpan di dalam maks. }
Kamus
  k : **integer**
Algoritma
  $maks \leftarrow a_1$
  $k \leftarrow 2$
  while $k \leq n$ **do**
    if $a_k > maks$ **then**
      $maks \leftarrow a_k$
    endif
    $k \leftarrow k + 1$
  endwhile



Kompleksitas waktu algoritma dihitung dari jumlah operasi perbandingan elemen larik ($a_k > maks$).
Kompleksitas waktu *CariElemenTerbesar* : $T(n) = n - 1$.

Kompleksitas waktu dibedakan atas tiga macam :

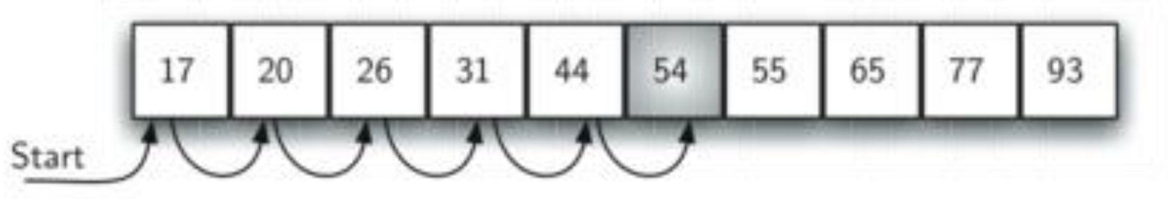
1. $T_{max}(n)$: kompleksitas waktu untuk kasus terburuk *(worst case),*
   → kebutuhan waktu maksimum.

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 3. Algoritma *sequential search* (linear search)

procedure PencarianBeruntun(**input** $a_1, a_2, ..., a_n$: **integer**, x : **integer**, **output** *idx* : *integer*)
{ Mencari elemen x di dalam larik A yang berisi n elemen. Jika x ditemukan, maka indeks elemen larik disimpan di dalam idx, idx bernilai –1 jika x tidak ditemukan
Kamus
    k : **integer**
    ketemu : **boolean**   *{ bernilai true jika x ditemukan atau false jika x tidak ditemukan }*
Algoritma:
  $k \leftarrow 1$
  ketemu ← **false**
  while ($k \le n$) **and** (**not** *ketemu*) **do**
     if $a_k = x$ **then**
         ketemu←**true**
     else
         $k \leftarrow k + 1$
     endif
  endwhile
  if *ketemu* **then**   *{ x ditemukan }*
     idx←*k*
  else
     idx← –1   *{ x tidak ditemukan }*
  endif



Jumlah operasi perbandingan elemen tabel:

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

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

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

$T_{\max}(n) = n$

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

$$T_{avg}(n) = \frac{(1+2+3+...+n)}{n} = \frac{\frac{1}{2}n(1+n)}{n} = \frac{(n+1)}{2}$$

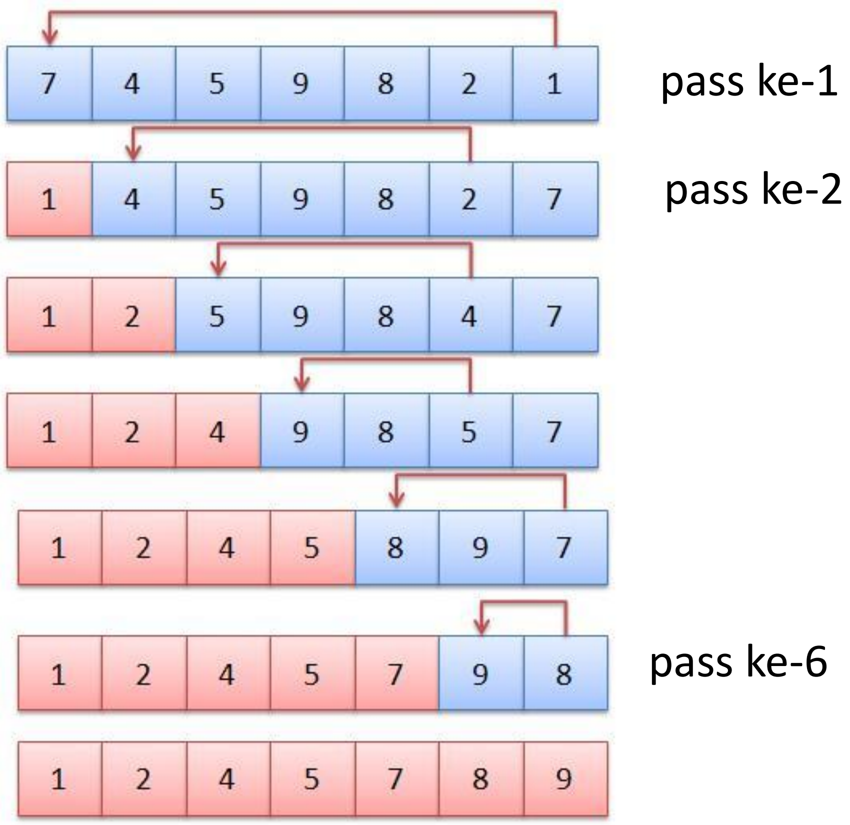
Cara lain: asumsikan bahwa $P(a_j = x) = 1/n$. Jika $a_j = x$ maka T_j yang dibutuhkan adalah $T_j = j$. Jumlah perbandingan elemen larik rata-rata:

$$T_{\text{avg}}(n) = \sum_{j=1}^{n} T_j \, P(a[j] = x) = \sum_{j=1}^{n} T_j \frac{1}{n} = \frac{1}{n} \sum_{j=1}^{n} T_j$$

$$= \frac{1}{n} \sum_{j=1}^{n} j = \frac{1}{n} \left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$$

Contoh 4: Algoritma pengurutan seleksi (*selection sort*)

procedure *SelectionSort*(**input/output** $a_1, a_2, ..., a_n$: **integer**)
{ *Mengurutkan elemen-elemen larik A yang berisi n elemen integer sehingga terurut menaik* }
Kamus
    i, j, imin, temp : **integer**
Algoritma
  for $i \leftarrow 1$ **to** $n - 1$ **do**   { *pass sebanyak n – 1 kali* }
      $imin \leftarrow i$
      for $j \leftarrow i + 1$ **to** n **do**
          if $a_j < a_{imin}$ **then**
              $imin \leftarrow j$
          endif
      endfor
      { *pertukarkan* a_{imin} *dengan* a_i }
      $temp \leftarrow a_i$
      $a_i \leftarrow a_{imin}$
      $a_{imin} \leftarrow temp$
  endfor



(i) Jumlah operasi perbandingan elemen-elemen larik ($a_j < a_{imin}$)

Untuk setiap pass ke-*i*,

i = 1 $\rightarrow$ jumlah perbandingan = n – 1

i = 2 $\rightarrow$ jumlah perbandingan = n – 2

i = 3 $\rightarrow$ jumlah perbandingan = n – 3

$\vdots$

i = n – 1 $\rightarrow$ jumlah perbandingan = 1

```
for i←1 to n – 1 do   { pass sebanyak n – 1 kali }
   imin←i
   for j←i + 1 to n do
      if a_j < a_imin then
         imin←j
      endif
   endfor
   { pertukarkan a_imin dengan a_i }
   temp←a_i
   a_i←a_imin
   a_imin←temp
endfor
```

Jumlah seluruh operasi perbandingan elemen-elemen larik adalah

$$T(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

Ini adalah kompleksitas waktu untuk kasus terbaik dan terburuk, karena algoritma *SelectionSort* tidak bergantung pada apakah data masukannya sudah terurut atau acak.

```
for i←1 to n – 1 do   { pass sebanyak n – 1 kali }
   imin←i
   for  j←i + 1 to n do
      if  a_j < a_imin then
         imin←j
      endif
   endfor
   { pertukarkan a_imin dengan a_i }
   temp←a_i
   a_i←a_imin
   a_imin←temp
endfor
```

(ii) Jumlah operasi pertukaran

Untuk setiap *i* dari 1 sampai *n* – 1, terjadi satu kali pertukaran elemen, sehingga jumlah operasi pertukaran seluruhnya adalah

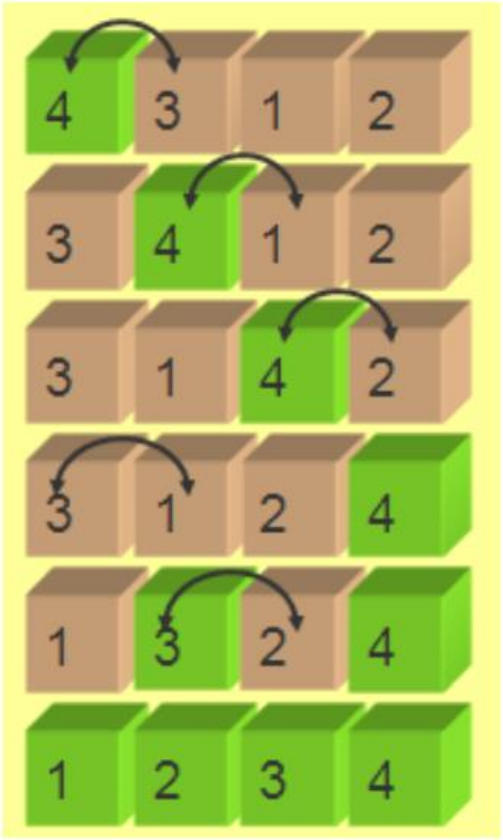
$$T(n) = n - 1.$$

Ini adalah jumlah pertukaran untuk semua kasus.

Jadi, algoritma pengurutan seleksi membutuhkan $n(n - 1)/2$ buah operasi perbandingan elemen dan $n - 1$ buah operasi pertukaran.

Contoh 5: Diberikan algoritma pengurutan *bubble-sort* seperti berikut ini. Hitung kompleksitas waktu algoritma didasarkan pada jumlah operasi perbandingan elemen-elemen larik dan jumlah operasi pertukaran.

procedure *BubbleSort*(**input/output** $a_1, a_2, ..., a_n$: **integer**)
{ Mengurut larik A yang berisi n elemen integer sehingga terrut menaik }
Kamus
    i, *j*, *temp* : **integer**
Algoritma
  for $i \leftarrow n - 1$ **downto** 1 do
    for $j \leftarrow 1$ **to** i do
      if $a_{j+1} < a_j$ **then**
        { *pertukarkan* a_j *dengan* a_{j+1} }
        $temp \leftarrow a_j$
        $a_j \leftarrow a_{j+1}$
        $a_{j+1} \leftarrow temp$
      endif
    endfor
  endfor



Sumber gambar: Prof. Amr Goneid, Department of Computer Science, AUC

(i) Jumlah operasi perbandingan elemen-elemen larik ($a_{j+1} < a_j$)

Untuk setiap *pass* ke-*i*,

$i = n - 1 \rightarrow$ jumlah perbandingan $= n - 1$

$i = n - 2 \rightarrow$ jumlah perbandingan $= n - 2$

$i = n - 3 \rightarrow$ jumlah perbandingan $= n - 3$

$\vdots$

$i = 1 \rightarrow$ jumlah perbandingan $= 1$

```
for i ← n – 1 downto 1 do
   for j ← 1 to i do
      if a_{j+1} < a_j then
         { pertukarkan a_j dengan a_{j-1} }
         temp ← a_j
         a_j ← a_{j+1}
         a_{j+1} ← temp
      endif
   endfor
endfor
```

Jumlah seluruh operasi perbandingan elemen-elemen larik adalah

$$T(n) = (n - 1) + (n - 2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

Ini adalah kompleksitas waktu untuk kasus terbaik dan terburuk, karena algoritma *BubbleSort* tidak bergantung pada apakah data masukannya sudah terurut atau acak. Jumlah operasi perbandingan sama dengan *selection sort.*

(ii) Jumlah operasi pertukaran ($temp \leftarrow a_i$; $a_i \leftarrow a_{imin}$; $a_{imin} \leftarrow temp$ **)**

Jumlah operasi pertukaran di dalam *bubble sort* hanya dapat dihitung pada kasus terbaik dan kasus terburuk. Kasus terbaik adalah tidak ada pertukaran (yaitu jika **if** $a_{j+1} < a_j$ false), yaitu semua elemen larik pada awalnya sudah terurut menaik, sehingga

$$T_{min}(n) = 0.$$

Pada kasus terburuk, (yaitu jika **if** $a_{j+1} < a_j$ bernilai true), pertukaran elemen selalu dilakukan. Jadi, jumlah operasi pertukaran elemen pada kasus terburuk sama dengan jumlah operasi perbandingan elemen-elemen larik, yaitu

$$T_{max}(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

Jadi, algoritma pengurutan *bubble sort* membutuhkan $n(n-1)/2$ buah operasi pertukaran, lebih banyak daripada algoritma *selection sort*. Ini berarti secara keseluruhan *bubble sort* lebih buruk daripada *selection sort*.

Latihan 1

Hitung kompleksitas waktu algoritma berikut berdasarkan jumlah operasi perkalian.

procedure *Kali*(**input** *x* : **integer**, *n* : **integer**, **output** *jumlah* : **integer**)
{Mengalikan x dengan i = 1, 2, …, j, yang dalam hal ini j = n, n/2, n/4, …,1. Hasil perkalian disimpan di dalam peubah jumlah. }
Kamus
    i, *j*, *k* : **integer**
Algoritma
  $j \leftarrow n$
  while $j \geq 1$ **do**
     for $i \leftarrow 1$ **to** j **do**
         $x \leftarrow x * i$
     endfor
     $j \leftarrow j$ **div** 2
  endwhile
  $jumlah \leftarrow x$

Jawaban

Untuk

$j = n$, jumlah operasi perkalian = n

$j = n/2$, jumlah operasi perkalian = $n/2$

$j = n/4$, jumlah operasi perkalian = $n/4$

…

$j = 1$, jumlah operasi perkalian = 1

Jumlah operasi perkalian seluruhnya adalah

$= n + n/2 + n/4 + \ldots + 2 + 1$ → deret geometri

$$= \frac{n(1 - 2^{{}^2\log n - 1})}{1 - \frac{1}{2}} = 2n - 1$$

```
j ← n
while j ≥ 1 do
   for i ← 1 to j do
      x ← x * i
   endfor
   j ← j div 2
endwhile
jumlah←x
```

Latihan 2

Di bawah ini adalah algoritma untuk menguji apakah dua buah matriks, *A* dan *B*, yang masing-masing berukuran $n \times n$, sama.

function *samaMatriks*(*A*, *B* : *matriks*; *n* : integer) $\rightarrow$ *boolean*
{ true jika A dan B sama; sebaliknya false jika A $\neq$ B }
Kamus
  i, *j* : **integer**
Algoritma:
  for $i \leftarrow 1$ **to** *n* **do**
    for $j \leftarrow 1$ **to** *n* **do**
      if $A_{i,j} \neq B_{i,j}$ **then**
        return false
      endif
    endfor
  endfor
  return true

(a) Apa kasus terbaik dan terburuk untuk algoritma di atas?

(b) Tentukan kompleksitas waktu terbaik dan terburuknya.

Jawaban:

(a) Kasus terbaik terjadi jika ketidaksamaan matriks ditemukan pada elemen pertama ($A_{1,1} \neq B_{1,1}$)

Kasus terburuk terjadi jika ketidaksamaan matriks ditemukan pada elemen ujung kanan bawah ($A_{n,n} \neq B_{n,n}$) atau pada kasus matriks A dan B sama, sehingga seluruh elemen matriks dibandingkan.

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

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

Latihan Mandiri

1. Diberikan matriks persegi berukuran n x n. Hitung kompleksitas waktu untuk memeriksa apakah matriks tersebut merupakan matriks simetri terhadap diagonal utama.

2. Berapa kompleksitas waktu untuk menjumlahkan matriks A dan B yang keduanya berukuran n x n?

3. Ulangi soal 2 untuk perkalian matriks A dan B.

4. Tulislah algoritma pengurutan *insertion sort* pada larik yang berukuran n elemen, hitung masing-masing kompleksitas waktu algoritma diukur dari jumlah operasi perbandingan dan jumlah operasi pertukaran elemen-elemen larik.

5. Berapa kali operasi penjumlahan pada potongan algoritma ini dilakukan?

for $i \leftarrow 1$ **to** n **do**

    for $j \leftarrow 1$ **to** n **do**

        for $k \leftarrow 1$ to j do

            $x \leftarrow x + 1$

        endfor

    endfor

endfor

6. Algoritma di bawah ini menghitung nilai polinom $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$

```
function p(input x:real)→real
{ Mengembalikan nilai p(x)}
Kamus
     j, k : integer
     jumlah, suku : real
Algoritma
    jumlah ← a0
    for  j ← 1 to n do
       { hitung ajx^j }
        suku ← aj
        for  k ← 1 to j do
              suku ← suku * x
         endfor
         jumlah ← jumlah + suku
     endfor
     return jumlah
```

Hitunglah berapa operasi perkalian dan berapa operasi penjumlahan yang dilakukan oleh algoritma tsb

Algoritma menghitung polinom yang lebih baik dapat dibuat dengan metode Horner berikut: $p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots + x(a_{n-1} + a_n x)))\ldots))$

function *p2*(**input** *x*:**real**)→**real**
{ Mengembalikan nilai p(x) dengan metode Horner}
Kamus
  k : **integer**
  $b_1, b_2, \ldots, b_n$: **real**
Algoritma
  $b_n \leftarrow a_n$
  for $k \leftarrow n - 1$ **downto** 0 **do**
    $b_k \leftarrow a_k + b_{k+1} * x$
  endfor
  return b_0

Hitunglah berapa operasi perkalian dan berapa operasi penjumlahan yang dilakukan oleh algoritma di atas? Manakah yang terbaik, algoritma *p* atau *p*2?

Bersambung ke Bagian 2