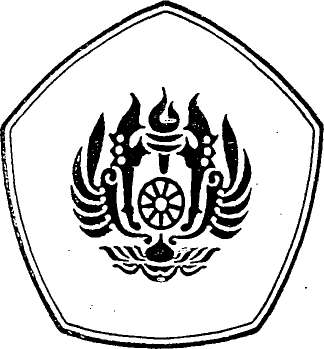
**LAPORAN PRAKTIKUM 2**

**KOMPLEKSITAS WAKTU DARI ALGORITMA**

**MATA KULIAH**

**ANALISIS ALGORITMA**



**FIRMANSYAH YANUAR - 140810170051**

**PROGRAM STUDI S-1 TEKNIK INFORMATIKA**

**DEPARTEMEN ILMU KOMPUTER**

**FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM**

**UNIVERSITAS PADJADJARAN**

**MARET 2019**

# Pendahuluan

Dalam memecahkan suatu masalah dengan komputer seringkali kita dihadapkan pada pilihan berikut:

1. Menggunakan algoritma yang waktu eksekusinya cepat dengan komputer standar
2. Menggunakan algoritma yang waktu eksekusinya tidak terlalu cepat dengan komputer yang cepat

Dikarenakan keterbatasan sumber daya, pola pemecahan masalah beralih ke pertimbangan menggunakan algoritma. Oleh karena itu diperlukan algoritma yang efektif dan efisien atau lebih tepatnya Algoritma yang mangkus.

Algoritma yang mangkus diukur dari berapa **jumlah waktu dan ruang (space) memori** yang dibutuhkan untuk menjalankannya. Algoritma yang mangkus ialah algoritma yang meminimumkan kebutuhan waktu dan ruang. Penentuan kemangkusan algoritma adakah dengan melakukan pengukuran kompleksitas algoritma.

Kompleksitas algoritma terdiri dari kompleksitas waktu dan ruang. Terminologi yang diperlukan dalam membahas kompleksitas waktu dan ruang adalah:

1. Ukuran input data untuk suatu algoritma, .

Contoh algoritma pengurutan elemen-elemen larik, adalah jumlah elemen larik. Sedangkan dalam algoritma perkalian matriks n adalah ukuran matriks .

1. Kompleksitas waktu, adalah jumlah operasi yang dilakukan untuk melaksanakan algoritma sebagai fungsi dari input .
2. Kompleksitas ruang, , adalah ruang memori yang dibutuhkan algoritma sebagai fungsi dari input .

**KOMPLEKSITAS WAKTU**

Kompleksitas waktu sebuah algoritma dapat dihitung dengan langkah-langkah sebagai berikut:

1. Menetapkan ukuran input
2. Menghitung banyaknya operasi yang dilakukan oleh algoritma.

Dalam sebuah algoritma terdapat banyak jenis operasi seperti operasi penjumlahan, pengurangan, perbandingan, pembagian, pembacaan, pemanggilan prosedur, dsb.

**CONTOH**

**Algoritma Menghitung Nilai Rata-rata**

procedure HitungRerata (input x1, x2, …, xn: integer, output r: real)

{ Menghitung nilai rata-rata dari sekumpulan elemen larik integer x1, x2, … xn.

Nilai rata-rata akan disimpan di dalam variable r.

**Input:** x1, x2, … xn

**Output**: r (nilai rata-rata)

}

**Deklarasi**

i : integer

jumlah : real

**Algoritma**

Jumlah 🡨 0

i 🡨 1

while i ≤ n do

jumlah 🡨 jumlah + ai

i 🡨 i + 1

endwhile

{i > n}

r 🡨 jumlah/n {nilai rata-rata}

**Menghitung Kompleksitas Waktu dari Algoritma Menghitung Nilai Rata-rata**

Jenis-jenis operasi yang terdapat di dalam Algoritma HitungRerata adalah:

* Operasi pengisian nilai/*assignment* (dengan operator “🡨”)
* Operasi penjumlahan (dengan operator “+”)
* Operasi pembagian (dengan operator “/”)

Cara menghitung kompleksitas waktu dari algoritma tersebut adalah sengan cra menghitung masing-masing jumlah operasi. Jika operasi tersebut berada di sebuah loop, maka jumlah operasinya bergantung berapa kali loop tersebut diulangi.

1. Operasi pengisian nilai (*assignment*)

jumlah 🡨 0, 1 kali

k 🡨 1, 1 kali

jumlah 🡨jumlah + ak n kali

k 🡨 k+1, n kali

r 🡨 jumlah/n, 1 kali

Jumlah seluruh operasi pengisian nilai (*assignment*) adalah

1. Operasi penjumlahan

Jumlah + ak, n kali

k+1, n kali

Jumlah seluruh operasi penjumlahan adalah

1. Operasi pembagian

Jumlah seluruh operasi pembagian adalah

Jumlah/n 1 kali

Dengan demikian, kompleksitas waktu algoritma dihitung berdasarkan jumlah operasi aritmatika dan operasi pengisian nilai adalah:

# Studi Kasus 1: Pencarian Nilai Maksimal

Buatlah programnya dan hitunglah kompleksitas waktu dari algoritma berikut:

**Algoritma Pencarian Nilai Maksimal**

procedure CariMaks(input x1, x2, …, xn: integer, output maks: integer)

{ Mencari elemen terbesar dari sekumpulan elemen larik integer x1, x2, …, xn. Elemen terbesar akan disimpan di dalam maks

Input: x1, x2, …, xn

Output: maks (nilai terbesar)

}

**Deklarasi**

i : integer

**Algoritma**

maks 🡨 x1

i 🡨 2

while i ≤ n do

if xi > maks then

maks 🡨 xi

endif

i 🡨 i + 1

endwhile

{i > n}

Jawaban Studi Kasus 1

**Source Code:**

#include <iostream>

using *namespace* std;

#define N 4

*int* CariMaks(*int* x[]){

*int* maks = x[0];

for(*int* i = 1; i < N; i++){

if(x[i] > maks)

maks = x[i];

}

return maks;

}

*int* main(){

*int* x[N] = {1,3,99,2};

cout << CariMaks(x);

}

**Hasil Program:**



1. Operasi pengisian nilai (*assignment*)

maks 🡨 x1 1 kali

i 🡨 2, 1 kali

maks 🡨 x1 n-1 kali (worst case)

0 kali (best case)

i 🡨 i + 1, n kali

Jumlah seluruh operasi pengisian nilai (*assignment*) adalah

1. Operasi penjumlahan

i + 1k, n kali

Jumlah seluruh operasi penjumlahan adalah

**PEMBAGIAN KOMPLEKSITAS WAKTU**

Hal lain yang harus diperhatikan dalam menghitung kompleksitas waktu suatu algoritma adalah parameter yang mencirikan ukuran input. Contoh pada algoritma pencarian, waktu yang dibutuhkan untuk melakukan pencarian tidak hanya bergantung pada ukuran larik () saja, tetapi juga bergantung pada nilai elemen () yang dicari.

Misalkan:

* Terdapat sebuah larik dengan panjang elemen 130 dimulai dari
* Asumsikan elemen-elemen larik sudah terurut. Jika , maka waktu pencariannya lebih cepat 130 kali dari pada atau tidak ada di dalam larik.
* Demikian pula, jika , maka waktu pencariannya ½ kali lebih cepat daripada

Oleh karena itu, kompleksitas waktu dibedakan menjadi 3 macam:

1. : kompleksitas waktu untuk kasus terbaik (***best case***)

merupakan kebutuhan waktu minimum yang diperlukan algoritma sebagai fungsi dari .

1. : kompleksitas waktu untuk kasus rata-rata (***average case***)

merupakan kebutuhan waktu rata-rata yang diperlukan algoritma sebagai fungsi dari . Biasanya pada kasus ini dibuat asumsi bahwa semua barisan input bersifat sama. Contoh pada kasus *searching* diandaikan data yang dicari mempunyai peluang yang sama untuk tertarik dari larik.

1. : kompleksitas waktu untuk kasus terburuk (***worst case***)

merupakan kebutuhan waktu maksimum yang diperlukan algoritma sebagai fungsi dari .

# Studi Kasus 2: *Sequential Search*

Diberikan larik bilangan bulan yang telah terurut menaik dan tidak ada elemen ganda. Buatlah programnya dengan C++ dan hitunglah kompleksitas waktu terbaik, terburuk, dan rata-rata dari algoritma pencarian beruntun (*sequential search*). Algoritma *sequential search* berikut menghasilkan indeks elemen yang bernilai sama dengan y. Jika y tidak ditemukan, indeks 0 akan dihasilkan.

procedure SequentialSearch(input : integer, y : integer, output idx : integer)

{ Mencari di dalam elemen .

Lokasi (indeks elemen) tempat ditemukan diisi ke dalam idx.

Jika tidak ditemukan, makai idx diisi dengan 0.

Input:

Output: idx

}

**Deklarasi**

i : integer

found : boolean {bernilai true jika y ditemukan atau false jika y tidak ditemukan}

**Algoritma**

i 🡨 1

found 🡨 false

while (i ≤ n) and (not found) do

if xi = y then

found 🡨 true

else

i 🡨 i + 1

endif

endwhile

{*i < n or found*}

If found then {*y ditemukan*}

idx 🡨 i

else

idx 🡨 0 {y tidak ditemukan}

endif

Jawaban Studi Kasus 2

1. Best Case: ini terjadi bila a1 = x.

Tmin(n) = 1

1. Worst Case : bila an = x atau x tidak ditemukan.

Tmax(n) = n

1. *Average Case*: Jika *x* ditemukan pada posisi ke-*j*, maka operasi perbandingan (ak = x) akan dieksekusi sebanyak *j* kali.

*T*avg(*n*) = (1+2+3+..+n)/n = (1/2n(1+n))/n = (n+1)/2

Source Code:

#include <iostream>

using *namespace* std;

#define N 4

*int* SequentialSearch(*int* \*x, *int* y){

*int* idx;

*int* i = 0;

*bool* found = false;

while( i < sizeof(x) && !found){

if(x[i] == y)

found = true;

else

i++;

}

if(found)

idx = i;

else

idx = 0;

return idx;

}

*int* main(){

*int* x[N] = {1,3,99,2};

cout << "Index key : " << SequentialSearch(x,2);

}

Hasil Program



# Studi Kasus 3: *Binary Search*

Diberikan larik bilangan bulan yang telah terurut menaik dan tidak ada elemen ganda. Buatlah programnya dengan C++ dan hitunglah kompleksitas waktu terbaik, terburuk, dan rata-rata dari algoritma pencarian bagi dua (*binary search*). Algoritma *binary search* berikut menghasilkan indeks elemen yang bernilai sama dengan y. Jika y tidak ditemukan, indeks 0 akan dihasilkan.

procedure BinarySearch(input : integer, x : integer, output : idx : integer)

{ Mencari y di dalam elemen . Lokasi (indeks elemen) tempat y ditemukan diisi ke dalam idx. Jika y tidak ditemukan makai dx diisi dengan 0.

**Input:**

**Output: idx**

}

**Deklarasi**

i, j, mid : integer

found : Boolean

**Algoritma**

i 🡨 1

j 🡨 n

found 🡨 false

while (not found) and ( i ≤ j) do

mid 🡨 (i + j) div 2

if xmid = y then

found 🡨 true

else

if xmid < y then {*mencari di bagian kanan*}

i 🡨 mid + 1

else {*mencari di bagian kiri*}

j 🡨 mid – 1

endif

endif

endwhile

{*found or i > j* }

If found then

Idx 🡨 mid

else

Idx 🡨 0

Endif

Jawaban Studi Kasus 3

1. Kasus terbaik

Tmin(n) = 1

1. Kasus terburuk

Tmax (n) = 2log n

Source Code

#include <iostream>

using *namespace* std;

#define N 5

*int* BinarySearch(*int* \*x, *int* y){

*int* i = 0,j = N,mid;

*bool* found = false;

while (!found && i<=j){

mid = (i+j)/2;

if( x[mid] == y )

found = true;

else if( x[mid] < y)

i = mid + 1;

else

j = mid - 1;

}

}

*int* main(){

*int* x[N] = {1,3,99,2,4};

cout << "Index key : " << BinarySearch(x,2);

}

Screenshot



# Studi Kasus 4: Insertion Sort

1. Buatlah program insertion sort dengan menggunakan bahasa C++
2. Hitunglah operasi perbandingan elemen larik dan operasi pertukaran pada algoritma insertion sort.
3. Tentukan kompleksitas waktu terbaik, terburuk, dan rata-rata untuk algoritma insertion sort.

procedure InsertionSort(input/output : integer)

{ Mengurutkan elemen-elemen dengan metode insertion sort.

Input:

OutputL (sudah terurut menaik)

}

**Deklarasi**

i, j, insert : integer

**Algoritma**

for i 🡨 2 to n do

insert 🡨 xi

j 🡨 i

while (j < i) and (x[j-i] > insert) do

x[j]🡨 x[j-1]

j🡨j-1

endwhile

x[j] = insert

endfor

Jawaban Studi Kasus 4

Loop sementara dijalankan hanya jika i> j dan arr [i] <arr [j]. Jumlah total iterasi loop sementara (Untuk semua nilai i) sama dengan jumlah inversi.

Kompleksitas waktu keseluruhan dari jenis penyisipan adalah O (n + f (n)) di mana f (n) adalah jumlah inversi.Jika jumlah inversi adalah O(n), maka kompleksitas waktu dari jenis penyisipan adalah O(n).

Dalam kasus terburuk, bisa ada inversi n \* (n-1) / 2. Kasus terburuk terjadi ketika array diurutkan dalam urutan terbalik. Jadi kompleksitas waktu kasus terburuk dari jenis penyisipan adalah O (n2).

Source Code

#include <iostream>

using *namespace* std;

#define N 5

*void* InsertionSort(*int* \*x){

*int* insert,j;

for(*int* i = 1; i < N; i++){

insert = x[i];

j = i-1;

while(j >= 0 && x[j] > insert){

x[j+1] = x[j];

j--;

}

x[j+1] = insert;

}

}

*void* printArray(*int* \*x){

for(*int* i = 0; i < N; i++)

{

cout << " " << x[i];

}

cout << endl;

}

*int* main(){

*int* x[N] = {1,3,99,2,4};

cout << "Sebelum sort : "; printArray(x);

InsertionSort(x);

cout << "Setelah sort : "; printArray(x);

}

Screenshot



# Studi Kasus 5: Selection Sort

1. Buatlah program selection sort dengan menggunakan bahasa C++
2. Hitunglah operasi perbandingan elemen larik dan operasi pertukaran pada algoritma selection sort.
3. Tentukan kompleksitas waktu terbaik, terburuk, dan rata-rata untuk algoritma insertion sort.

procedure SelectionSort(input/output : integer)

{ Mengurutkan elemen-elemen dengan metode selection sort.

Input:

OutputL (sudah terurut menaik)

}

**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 xj > ximaks then

imaks 🡨 j

endif

endfor

{pertukarkan ximaks dengan xi}

temp 🡨 xi

xi 🡨 ximaks

ximaks 🡨 temp

endfor

Jawaban Studi Kasus 5

1. Jumlah operasi perbandingan element. Untuk setiap *pass* ke-*i*,

*i*= 1 –>  jumlah perbandingan  = *n* – 1

*i* = 2 –>  jumlah perbandingan = *n* – 2

*i*= 3  –> jumlah perbandingan = *n* – 3

:

*i* = *k* –>  jumlah perbandingan = *n* – *k*

:

*i*= *n* – 1  –> jumlah perbandingan = 1

Jumlah seluruh operasi perbandingan elemen-elemen larik adalah    *T*(*n*) = (*n* – 1) + (*n* – 2) + … + 1

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

1. 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.

Jadi, algoritma pengurutan maksimum membutuhkan *n*(*n* – 1 )/2 buah operasi perbandingan elemen dan *n* – 1  buah operasi pertukaran.

Source Code

#include <iostream>

using *namespace* std;

#define N 5

*void* SelectionSort(*int* \*x){

*int* imaks,temp;

for(*int* i = N-1; i >= 1; i--){

imaks = 0;

for(*int* j = 1; j <= i; j++)

if(x[j] > x[imaks])

imaks = j;

temp = x[i];

x[i] = x[imaks];

x[imaks] = temp;

}

}

*void* printArray(*int* \*x){

for(*int* i = 0; i < N; i++)

{

cout << " " << x[i];

}

cout << endl;

}

*int* main(){

*int* x[N] = {1,3,99,2,4};

cout << "Sebelum sort : "; printArray(x);

SelectionSort(x);

cout << "Setelah sort : "; printArray(x);

}

Screenshot

