LAPORAN TUGAS 2

Analisis Algoritma

Kompleksitas Waktu Dari Algoritma

140810160014

SACHI HONGO

KELAS B

S-1 TEKNIK INFORMATIKA

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM

UNIVERSITAS PADJADJARAN

2019

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
```

```
#include <iostream>

using namespace std;


int main()

{

   int i, n;

   float arr[100];


   cout << "Masukkan jumlah dari elemen(1 s/d 100): ";

   cin >> n;

   cout << endl;


   // Store number entered by the user

   for(i = 0; i < n; ++i)
```

```
  {
    cout << "Masukkan angka " << i + 1 << " : ";
    cin >> arr[i];
  }

  // Loop to store largest number to arr[0]
  for(i = 1;i < n; ++i)
  {
    // Change < to > if you want to find the smallest element
    if(arr[0] < arr[i])
        arr[0] = arr[i];
  }
  cout << "\nNilai maksimal = " << arr[0];

  return 0;
}
```

"E:\SACHI-140810160014\Semester 6\Analisis Algoritma\Praktikum\Praktikum-Analgo\Praktikum_2\maxvalue.exe"

```
Masukkan jumlah dari elemen(1 s/d 100): 6

Masukkan angka 1 : 8
Masukkan angka 2 : 89
Masukkan angka 3 : 777
Masukkan angka 4 : 1
Masukkan angka 5 : 65
Masukkan angka 6 : 2

Nilai maksimal = 777
Process returned 0 (0x0)   execution time : 18.220 s
Press any key to continue.
```

Penyelesaian :

Jenis-jenis operasi yang terdapat di dalam Algoritma Pencarian Nilai Maksmimal adalah:

- Operasi pengisian nilai/*assignment* (dengan operator "←")
- Operasi penjumlahan (dengan operator "+")

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

(i) Operasi pengisian nilai (*assignment*)

maks ← x_1	1 kali
i ← 2	1 kali
maks ← x_i	n kali
i ← i + 1	n kali

Jumlah seluruh operasi pengisian nilai (*assignment*) adalah

$t_1 = 1 + 1 + n + n = 2 + 2n$

(ii) Operasi penjumlahan

i + 1	n kali

Jumlah seluruh operasi penjumlahan adalah
$t_2 = n$

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

$$T(n) = t_1 + t_2 = 1 + 1 + n + n + n = 2 + 3n$$

Studi Kasus 2: *Sequential Search*

Diberikan larik bilangan bulan $x_1, x_2, \ldots\ x_n$ 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 x1, x2, ... xn : integer,  y : integer, output idx : integer)
{    Mencari y di dalam elemen x1, x2, ... xn. Lokasi (indeks elemen) tempat y ditemukan diisi ke dalam idx.
     Jika y tidak ditemukan, makai idx diisi dengan 0.
     Input: x1, x2, ... xn
     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
```

```
#include <iostream>


using namespace std;


int main()

{
```

```
	int arr[100], i, num, n, c=0, cek;
	cout<<"\nMasukkan jumlah dari elemen(1 s/d 100) : ";
	cin>>n;
	for(i=0; i<n; i++)
	{
	   cout<<"Masukkan angka : ";
		cin>>arr[i];
	}
	cout<<"\nMasukkan angka yang ingin dicari : ";
	cin>>num;


	for(i=0; i<n; i++)
	{
		if(arr[i]==num)
		{
			c=1;
			cek=i+1;
			break;
		}
	}
	if(c==0)
	{			cout<<"Angka tidak ditemukan..!!";
	}
	else
	{			cout<<num<<" ditemukan pada posisi "<<cek;
	}

}
```

```
"E:\SACHI-140810160014\Semester 6\Analisis Algoritma\Praktikum\Praktikum-Analgo\Praktikum_2\sequentials

Masukkan jumlah dari elemen(1 s/d 100) : 7
Masukkan angka : 9
Masukkan angka : 7
Masukkan angka : 5
Masukkan angka : 79
Masukkan angka : 65
Masukkan angka : 555
Masukkan angka : 4

Masukkan angka yang ingin dicari : 555
555 ditemukan pada posisi 6
Process returned 0 (0x0)   execution time : 47.517 s
Press any key to continue.
```

Penyelesaian :

Algoritma pencarian beruntun membandingkan setiap elemen larik dengan x, mulai dari elemen pertama sampai x ditemukan atau sampai elemen terakhir. jika x ditemukan, maka proses pencarian dihentikan. kita akan menghitung jumlah operasi perbandingan elemen larik yang terjadi selama pencarian ($a_i = x$). operasi perbandingan yang lain, seperti $i \leq n$ tidak akan dihitung. operasi perbandingan elemen-elemen larik adalah abstrak yang mendasari algoritma alagoritma pencarian.

1. *Kasus terbaik* : ini terjadi bila $a_i = x$.
   operasi perbandingan elemen ($a_i = x$) hanya dilakukan satu kali, maka

   $T_{min}(n) = 1$

2. Kasus terburuk: bila $a_n = x$ atau x tidak ditemukan.
   seluruh elemen larik dibandingkan, maka jumlah perbandingan elemen larik ($a_i = x$) adalah
   $T_{max}(n) = n$

3. Kasus rata-rata : Jika x ditentukan pada posisi ke-j, maka operasi perbandingan ($a_i = x$) dilakukan sebanyak j kali. Jadi, kebutuhan waktu rata-rata algoritma pencarian beruntun adalah

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

Cara lain yang dapat digunakan dalam menghitung T_{avg} diatas adalah sebagai berikut :

Asumsikan bahwa peluang x terdapat di sembarang lokasi larik adalah sama, artinya, peluang elemen ke-j = x adalah $\frac{1}{n}$, atau kita tulis P ($a_j = x$) = $\frac{1}{n}$. jika $a_j = x$ maka T_j yang dibutuhkan adalah $T_j = j$. jumlah perbandingan elemen larik secara rata-rata adalah :

$$T_{avg}(n) = \sum_{j=1}^{n} T_j P(A[\, j\,] = X) = \sum_{j=1}^{n} T_j \frac{1}{n} = \frac{1}{n} \sum_{j=1}^{n} T_j = \frac{1}{n} \sum_{j=1}^{n} j = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$$

Studi Kasus 3: *Binary Search*

Diberikan larik bilangan bulan $x_1, x_2, \ldots\ x_n$ yang <u>telah terurut</u> 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 x1, x2, ... xn : integer, x : integer, output : idx : integer)
{    Mencari y di dalam elemen x1, x2, ... xn. Lokasi (indeks elemen) tempat y ditemukan diisi ke dalam idx.
     Jika y tidak ditemukan makai dx diisi dengan 0.
     Input: x1, x2, ... xn
     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
```

```
#include <iostream>
using namespace std;

int main()
{
	int count, i, arr[100], num, first, last, middle;
	cout<<"\nMasukkan jumlah dari elemen :";
    cin>>count;

	for (i=0; i<count; i++)
	{
		cout<<"Masukkan angka : ";
	 cin>>arr[i];
	}
	cout<<"\nMasukkan angka yang ingin dicari :";
    cin>>num;
	first = 0;
	last = count-1;
	middle = (first+last)/2;
	while (first <= last)
	{
	  if(arr[middle] < num)
	  {
		first = middle + 1;

	  }
	  else if(arr[middle] == num)
	  {
		cout<<num<<" ditemukan pada posisi "<<middle+1<<"\n";
	 break;
```

```
        }
        else {
            last = middle - 1;
        }
        middle = (first + last)/2;
    }
    if(first > last)
        {
          cout<<num<<" tidak ditemukan ";
        }
        return 0;
}
```

"E:\SACHI-140810160014\Semester 6\Analisis Algoritma\Praktikum\Praktikum-Analgo\Praktikum_2\bir

```
Masukkan jumlah dari elemen :6
Masukkan angka : 5
Masukkan angka : 8
Masukkan angka : 64
Masukkan angka : 0
Masukkan angka : 4
Masukkan angka : 7

Masukkan angka yang ingin dicari :78
78 tidak ditemukan
Process returned 0 (0x0)   execution time : 14.023 s
Press any key to continue.
```

Penyelesaian :

Algoritma pencarian biner membagi larik dipertengahan menjdi dua bagian yang berukuranm sama ($\frac{n}{2}$ bagian), bagian kiri dan bagian kanan. jika elemen pertengahan tidak sama dnegan x, keputusan dibuat untuk melakukan penarian pada bagian kiri atau bagian kanan. proses bagi dua dilakukan lagi pada bagian yagn dipilih. perhatikanlah bahwa setiap kali memasuki kalang while do maka ukuran larik yang ditelusuri berkuran menjadi setengah kali ukuran semula : n, $\frac{n}{2}$, $\frac{n}{4}$,…

Kita akan menghitung jumlah operasi perbanddingan elemen dengan x yang terjadi selama pencarian (a_{mid} = x). opeasi perbandingan yang lain, seperti I≤ *j* dan a_{mid} < x tidak akan dihitung. untuk penyederhanaan, asumsikan ukuran larik adalah perangkatan dari dua (yaitu, $n=2^k$)

1. Kasus terbaik

   Kasus terbaik adalah bila x ditemukan pada elemen pertengahan (a_{mid}), dan operasi perbandingan elemen (a_{mid} = x) yang dilakukan hanya satu kali. Pada kasus ini

   $T_{min}(n) = 1$

2. Kasus terburuk

   Pada kasus terburuk, elemen x ditemukan ketika ukuran larik = 1. Pada kasus terburuk ini, ukuran larik setiap kali memasuki kalang while-do adalah:

   n, n/2, n/4, n/8,…, 1 (sebanyak ^{2}logn kali)

   artinya, kalang *while-do* dikerjakan sebanyak ^{2}log n kali.

   Contoh : n = 128 ⇒ 128 → 64 → 32 → 16 →8 → 4 → 2 →1(sebanyak ^{2}log 128 = 7 kali pembagian)

   jumlah operasi perbandingan elemen (a_{mid} = x) adalah :

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

Kompleksitas waktu rata-rata algoritma pencarian bagi dua lebih sulit ditentukan. ■Tetapi biasanya kompleksitas waktu terburuknya $T_{avg}(n) = {}^2\log n$

Kadang-kadang penentuan kasus barguna untuk menghitung operasi yang bergantung pada kondisi tertentu. misalnya pada potongan algoritma berikut kita ingin menghitung nilai rata-rata elemen larik yang ganjil:

```
k ← 1
jumlah ← 0
for k ← 1 to n do
   if ak mod 2 = 1 then (ak ganjil)
       jumlah ← jumlah + ak
   endif
endfor
```

Misalkan kompleksitas waktu algoritma dihitung berdasarkan jumlah operasi penjumlahan elemen (yaitu, `jumlah` + a_k). instruksi jumlah ← jumlah + a_k hanya dikerjakan jika konsisi a_k mod 2 = 1 benar (yaitu jika a_k ganjil). meskipun demikian, kita tetap dapat menghitung kompleksitas waktu algoritma dengan mengansumsikan bahwa pada kasus terburuk semua elemen bernilai ganjil sehingga kondisi a_k mod 2 = 1 terpenuhi dan instruksi penjumlahan jumah ← jumlah + a_k dikerjakan sebanyak n kali. jadi, kasus terburuk,

$T_{max}(n) = n$

dan pada kasus terbaik, semua elemen genap sehingga instruksi penjumlahan jumlah ← jumlah + a_k tidak pernah dikerjakan, jadi $T_{min}(n) = 0$

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.

<u>procedure</u> InsertionSort(<u>input/output</u> $x_1, x_2, \dots\ x_n$: <u>integer</u>)
{ Mengurutkan elemen-elemen $x_1, x_2, \dots\ x_n$ dengan metode insertion sort.
Input: $x_1, x_2, \dots\ x_n$
OutputL $x_1, x_2, \dots\ x_n$ (sudah terurut menaik)
}

Deklarasi

i, j, insert : <u>integer</u>

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
```

```
#include<iostream>


using namespace std;


int main()

{

   int i,j,n,temp,a[30];

   cout<<"\nMasukkan jumlah dari elemen : ";

   cin>>n;


   for(i=0;i<n;i++)

   {

      cout<<"Masukkan angka : ";

      cin>>a[i];

   }
```

```
    }

    for(i=1;i<=n-1;i++)
    {
        temp=a[i];
        j=i-1;

        while((temp<a[j])&&(j>=0))
        {
            a[j+1]=a[j];    //moves element forward
            j=j-1;
        }

        a[j+1]=temp;    //insert element in proper place
    }

    cout<<"\nHasil sorting\n";
    for(i=0;i<n;i++)
    {
        cout<<a[i]<<" ";
    }

    return 0;
}
```

```
"E:\SACHI-140810160014\Semester 6\Analisis Algoritma\Praktikum\Praktikum-Analgo\Praktikum_2\insertionsort.exe"

Masukkan jumlah dari elemen : 9
Masukkan angka : 8
Masukkan angka : 1
Masukkan angka : 29
Masukkan angka : 2
Masukkan angka : 38
Masukkan angka : 991
Masukkan angka : 5
Masukkan angka : 87
Masukkan angka : 22

Hasil sorting
1 2 5 8 22 29 38 87 991
Process returned 0 (0x0)   execution time : 31.817 s
Press any key to continue.
```

Penyelesaian :

Algoritma *Insertion Sort* juga terdiri dari 2 kalang bersarang. Dimana terjadi N-1 *Pass* (dengan N adalah banyak elemen struktur data), dengan masing-masing *Pass* terjadi *i* kali operasi perbandingan. *i* tersebut bernilai 1 untuk *Pass* pertama, bernilai 2 untuk *Pass* kedua, begitu seterusnya hingga *Pass* ke N-1.

$$T(n) = 1 + 2 + \cdots + n - 1 = \sum_{i=1}^{n-1} \frac{n(n-1)}{2} = O(n^2)$$

1. Kasus terbaik

   $T_{min}(n)$ = n

2. Kasus terburuk

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

3. Kasus rata-rata

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

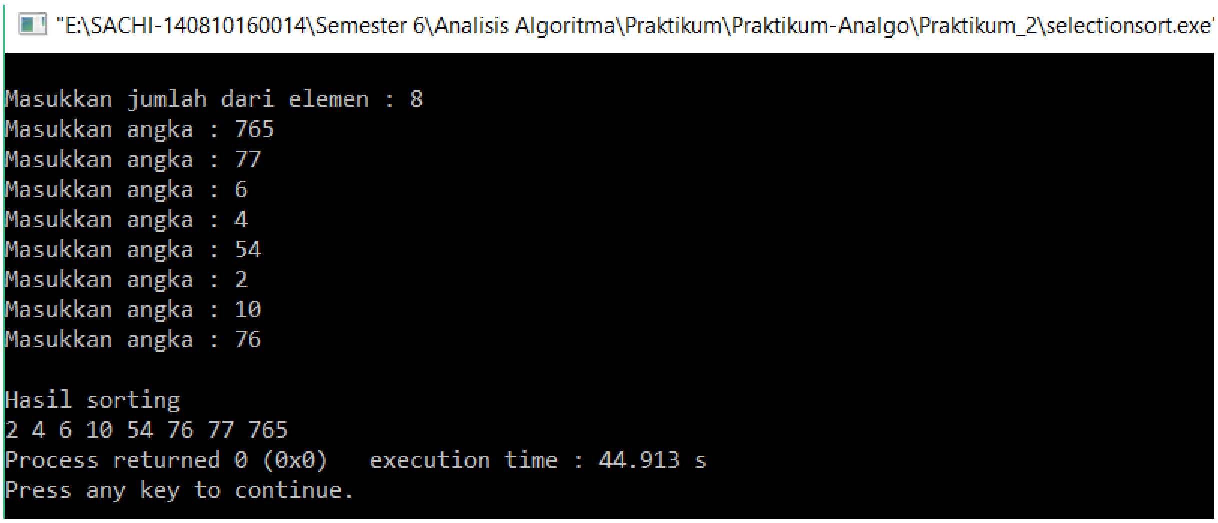
Studi Kasus 5: Selection Sort

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 selection sort.

```
procedure SelectionSort(input/output x_1, x_2, ... x_n : integer)
{   Mengurutkan elemen-elemen x_1, x_2, ... x_n dengan metode selection sort.
    Input: x_1, x_2, ... x_n
    OutputL x_1, x_2, ... x_n (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 x_j > x_imaks then
                imaks ← j
              endif
            endfor
            {pertukarkan x_imaks dengan x_i}
            temp ← x_i
            x_i ← x_imaks
            x_imaks ← temp
        endfor
```

```
#include<iostream>
using namespace std;


int main()
{
   int i,j,n,loc,temp,min,a[30];
   cout<<"\nMasukkan jumlah dari elemen : ";
   cin>>n;


   for(i=0;i<n;i++)
   {
      cout<<"Masukkan angka : ";
      cin>>a[i];
   }
```

```
    for(i=0;i<n-1;i++)
    {
        min=a[i];
        loc=i;
        for(j=i+1;j<n;j++)
        {
            if(min>a[j])
            {
                min=a[j];
                loc=j;
            }
        }

        temp=a[i];
        a[i]=a[loc];
        a[loc]=temp;
    }

    cout<<"\nHasil sorting\n";
    for(i=0;i<n;i++)
    {
        cout<<a[i]<<" ";
    }

    return 0;
}
```



Penyelesaian :

Algoritma urut seleksi terdiri dari n – 1 kali pass. pada setiap kali pass, kita mencari elemen terbesar dari elemen-elemen $a_1, a_2, \ldots, a_n$, lalu mempertukarkan elemen terbesar dengan a_n. pass berikutnya akan mencrai elemen terbesar dari sekumpulan $a_1, a_2, \ldots, a_{n-1}$, begitu seterusnya sampai larik pass terakhir sehingga tinggal satu elemen yang pasti sudah terurut. Operasi abstrak yang mendasari algoritma pengurutan adalah operasi perbandingan elemen larik ($a_j > a_{imaks}$) dan operasi pertukaran (diwakili oleh tiga buah instruksi : temp $\leftarrow a_n$, $a_n \leftarrow a_{imaks}$, $a_{imaks} \leftarrow$ temp). Kedua operasi ini kita pisahkan perhitungannya sebagai berikut :

(i) Jumlah operasi perbandingan elemen

Untuk setiap pass ke - i , i = n , n – 1, … , 2, operasi perbandingan elemen yang dilakukan adalah sebagai berikut :

$i = n$ → jumlah operasi perbandingan elemen = n – 1

$i = n - 1$ → jumlah operasi perbandingan elemen = n – 2

$i = n - 2$ → jumlah operasi perbandingan elemen = n – 3

⋮

$i = 2$ → jumlah operasi perbandingan elemen = 1

jumlah seluruh operasi perbandingan elemen-elemen larik adalah

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

Ini adalah kompleksitas waktu untuk kasus terbaik dan terburuk, karena algoritma urut seleksi tidak bergantung pada data masukan apakah sudah terurut atau acak.

(ii) Jumlah operasi pertukaran

Untuk setiap i dan n samapai 2, terjadi satu kali pertukaran elemen, sehingga jumlah operasi pertukaran seluruhnya adalah

T(n) = n – 1.

jadi, algoritma pengurutan seleksi membutuhkan $\frac{n(n-1)}{2} = O(n^2)$ buah operasi perbandingan elemen dan n – 1 buah operasi pertukaran.

1. Kasus terbaik

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

2. Kasus terburuk

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

3. Kasus rata-rata

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