Tugas Praktikum

Analisis Algoritma

(*Modul 2*)

Disusun oleh :

Muhammad Islam Taufikurahman

(140810160062)

S-1 Teknik Informatika

Fakultas Matematika & Ilmu Pengetahuan Alam

Universitas Padjadjaran

Jalan Raya Bandung - Sumedang Km. 21 Jatinangor 45363

Studi Kasus 1: Pencarian Nilai Maksimal

Algoritma:

```
procedure CariMaks(input x1, x2, ..., xn: integer, output maks: integer){
    Mencari elemen terbesar dari sekumpulan elemen larik integer x1, x2, ...,
xn.
    Elemen terbesar akandisimpan 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
```

Program:

```
/**
* NilaiMaksimal
*/
public class NilaiMaksimal {

   static int cariNilaiMaks(int[] x) {
       int max = x[0];
       for (int i = 0; i < x.length; i++) {
           if (x[i] > max) {
               max = x[i];
           }
       }
       return max;
   }
```

```
    public static void main(String[] args) {
        int[] x = {1, 7, 0, 1, 5};
        System.out.print(cariNilaiMaks(x));
    }
}
```

Kompleksitas waktu:

Kompleksitas waktu algoritma dihitung berdasarkan jumlah operasi perbandingan elemen larik (A[i] > maks). Kompleksitas waktu CariNilaiMaks : T(n) = n – 1.

Studi Kasus 2: Sequential Search

Algoritma:

```
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
```

Program:

```java
/**
* SeqentialSearch
*/
public class SeqentialSearch {

   static boolean sequentialSearch(boolean found, int[] array, int key) {
       int i = 0;
       while (i < array.length && !found) {
           if (array[i] == key) {
               found = true;
           } else {
               i += 1;
           }
       }
       return found;
   }

   public static void main(String[] args) {
       int key = 5;
       boolean found = false;
       int[] array = {1, 2, 3, 4, 5, 6, 7};

       if (sequentialSearch(found, array, key)) {
           System.out.print(key + " ditemukan");
       } else {
           System.out.print("tidak ditemukan");
       }
   }
}
```

Kompleksitas waktu:

1. Kompleksitas waktu terbaik:

   Adalah ketika elemen ditemukan pada array[0].

2. Kompleksitas waktu terburuk:

   Adalah ketika elemen ditemukan pada array[n] (elemen terakhir) atau tidak ditemukan sama sekali sampai elemen terakhir.

3. Kompleksitas waktu rata-rata:

   Adalah dimana elemen ditemukan diantara elemen pertama dan terakhir.

Studi Kasus 3: Binary Search

Algoritma:

```
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
```

Program:

```java
/**
* BinarySearch
*/
public class BinarySearch {

   static int binarySearch(int arr[], int l, int r, int x)
   {
       if (r >= l) {
           int mid = l + (r - l) / 2;
            // If the element is present at the
           // middle itself
           if (arr[mid] == x)
               return mid;
            // If element is smaller than mid, then
           // it can only be present in left subarray
           if (arr[mid] > x)
               return binarySearch(arr, l, mid - 1, x);
            // Else the element can only be present
           // in right subarray
           return binarySearch(arr, mid + 1, r, x);
       }
        // We reach here when element is not present
       // in array
       return -1;
   }

   public static void main(String[] args) {
       int arr[] = {5, 7, 11, 12, 32};
       int x = 10;
       int result = binarySearch(arr, 0, arr.length - 1, x);
       if (result == -1) {
           System.out.println("Elemen ditemukan");
       } else {
           System.out.println("Elemen ditemukan pada index " + result);
       }
   }
}
```

Kompleksitas waktu:

1. Kompleksitas waktu terbaik:

   $T_{min}(n) = 1$

   Adalah ketika elemen ditemukan pada nilai tengah dari panjang array.

2. Kompleksitas waktu terburuk:

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

Adalah ketika elemen tidak ditemukan di dalam array.

Studi Kasus 4: Insertion Sort

Algoritma:

```
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]
            jj-1
      endwhile
      x[j] = insert
      endfor
```

Program:

```
/**
* InsertionSort
*/
public class InsertionSort {

   static int[] insertionSort(int[] array)
   {
       for (int i = 1; i < array.length; ++i) {
           int key = array[i];
           int j = i - 1;
            while (j >= 0 && array[j] > key) {
               array[j + 1] = array[j];
               j = j - 1;
           }
           array[j + 1] = key;
```

```
        }

        return array;
    }

    public static void main(String[] args) {
        int[] array = {2, 9, 1, 5, 1, 3};

        for (int i = 0; i < array.length; i++) {
            System.out.println(insertionSort(array)[i]);
        }
    }
}
```

Kompleksitas waktu:

J	Perbandingan	Perpindahan	Total Operasi
2	1	1	2
3	2	2	4
4	3	3	6
5	4	4	8
N	$(n-1)$	$(n-1)$	$2(n-1)$

Sehingga total kompleksitas waktu T(n) untuk *worst case* yang dibutuhkan adalah:

$$
\begin{aligned}
T(n) &= 2(1) + 2(2) + 2(3) + 2(4) + 2(5) + \cdots + 2(n-1)) \\
&= 2\big(1 + 2 + 3 + 4 + 5 + \cdots + (n-1)\big) \\
&= 2\frac{(n-1)(n)}{2} \\
&= (n-1)(n)
\end{aligned}
$$

Kompleksitas waktu terbaik:

Untuk kasus terbaik algoritma ini berjalan 1 kali, yaitu jika elemen dalam tabel telah terurut. Loop while tidak pernah dijalankan.

Kompleksitas waktu terburuk:

Untuk kasus terburuk algoritma ini berjalan Nmax kali.

Studi Kasus 5: Selection Sort

Algoritma:

```
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
```

Program:

```
/**
* SelectionSort
*/
public class SelectionSort {

   static int[] selectionSort(int[] array)
   {
       int n = array.length;
        for (int i = 0; i < n-1; i++) {
           int min_idx = i;
           for (int j = i+1; j < n; j++)
               if (array[j] < array[min_idx])
                   min_idx = j;
            int temp = array[min_idx];
           array[min_idx] = array[i];
           array[i] = temp;
       }

       return array;
   }

   public static void main(String[] args) {
       int[] array = {2, 9, 1, 5, 1, 3};

       for (int i = 0; i < array.length; i++) {
           System.out.println(selectionSort(array)[i]);
       }
   }
}
```

Kompleksitas waktu:

(i) Jumlah operasi perbandingan elemen

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

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

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

(ii) Jumlah operasi pertukaran

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

$T(n) = n - 1$.

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