**LAPORAN PRAKTIKUM 2**

**ANALISIS ALGORITMA**



Disusun oleh :

Muhammad Zulfikar Ali

140810180064

**PROGRAM STUDI S-1 TEKNIK INFORMATIKA**

**FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM**

**UNIVERSITAS PADJADJARAN**

**2020**

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 |
| Jawaban Studi Kasus 1  T(n) = 2(n – 2) + (n – 2) + 2  = 3 n - 4 |

**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 *y1, y2, …, yn*
* 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 *y65x* , maka waktu pencariannya ½ kali lebih cepat daripada *y130x*

Oleh karena itu, kompleksitas waktu dibedakan menjadi 3 macam:

1. *Tmin(n )* : kompleksitas waktu untuk kasus terbaik (***best case***) merupakan kebutuhan waktu minimum yang diperlukan algoritma sebagai fungsi dari .

1. *Tavg(n)* : 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. *Tmax(n)* : 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 *x1, x2, …, xn* yang telah terurut menaik dan tidak ada elemen ganda. Buatlah programnya dengan C++ dan hitunglah kompleksitas waktu terbaik, terburuk, dan ratarata 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 di dalam elemen *x1, x2, …, xn*. Lokasi (indeks elemen) tempat ditemukan diisi ke dalam idx. Jika 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 |

|  |
| --- |
| Jawaban Studi Kasus 2  Jawaban Studi Kasus 2   * Kompleksitas waktu terbaik (best case) : Tmin(n) = 1 atau ini terjadi apabila a1 = x. * Kompleksitas waktu terburuk (worst case) : Tmax(n) = n atau ini terjadi apabila an = x atau nilai x tidak ditemukan * Kompleksitas waktu rata2 adalah Jika x ditemukan pada posisi ke-*j*, maka operasi perbandingan   (ak = x) akan dieksekusi sebanyak *j* kali.  Tavg(n) = |

Studi Kasus 3: *Binary Search*

Diberikan larik bilangan bulan *x1, x2, …, xn* yang telah terurut menaik dan tidak ada elemen ganda. Buatlah programnya dengan C++ dan hitunglah kompleksitas waktu terbaik, terburuk, dan ratarata 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**   1.  1 2.  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*}   1.  mid + 1   else {*mencari di bagian kiri*}   1.  mid – 1   endif  endif  endwhile  {*found or i > j* }  If found then  Idx  mid  else  Idx  0  endif |
| Jawaban Studi Kasus 3   * Kasus waktu terbaik (Best case)   Tmin(n) = 1   * Kasus waktu rata2 : jika terdapat pada index pada awal atau akhir elemen. * Kasus waktu terburuk (Worst case):   Tmax (n) = 2log n |

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 *x1, x2, …, xn* : integer)  { Mengurutkan elemen-elemen *x1, x2, …, xn* dengan metode insertion sort.  Input: *x1, x2, …, xn*  OutputL *x1, x2, …, xn* (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   * Kasus waktu terbaik (Best case) : jika array yang ada sudah terurut dengan benar, jadi looping tidak akan dilakukan lagi. * Kasus waktu rata2 : jika array elemen yang ada sudah terurut setengahnya / sebagian dari seluruh elemen   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). |

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 *x1, x2, …, xn* : integer)  { Mengurutkan elemen-elemen *x1, x2, …, xn* dengan metode selection sort.  Input *x1, x2, …, xn*  OutputL *x1, x2, …, xn* (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  a. 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.  b. 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. |