NAMA : Afie Syahrulloh Arridlo

NPM : 140810170040

KASUS I

Source Code :

```
#include <iostream>
#include <time.h>
#include <conio.h>
#include <windows.h>

using namespace std;
int data[100];

void mergeSort(int awal, int mid, int akhir)
{
   cout<<endl;
   int temp[100], tempAwal = awal, tempMid = mid, i = 0;
   while(tempAwal < mid && tempMid < akhir)
   {
      if(data[tempAwal] < data[tempMid])
         temp[i] = data[tempAwal],tempAwal++;
      else
         temp[i] = data[tempMid],tempMid++;
      i++;
   }
   while(tempAwal < mid)  //kalau masih ada yang sisa
      temp[i] = data[tempAwal],tempAwal++,i++;
```

```
    while(tempMid < akhir)
        temp[i] = data[tempMid],tempMid++,i++;
    for(int j=0,k=awal;j<i,k<akhir;j++,k++)  //mengembalikan ke array semula, tapi
        cout<<data[k]<<' '<<temp[j]<<endl, data[k] = temp[j]; //sudah urut
}

void merge(int awal, int akhir) //membagi data secara rekursif
{
    if(akhir-awal != 1)
    {
        int mid = (awal+akhir)/2;
        merge(awal, mid);
        merge(mid, akhir);
        mergeSort(awal, mid, akhir);
    }
}

int main()
{
    int n,t1,t2;
    cout<<"Masukan banya data = ";cin>>n;
    cout<<"Masukan data yang akan di susun = ";
    t1=GetTickCount();
    for(int i=0;i<n;i++)
        cin>>data[i];
    merge(0,n);
    for(int i=0;i<n;i++)
```

```
    cout<<data[i]<<' ';
  t2=GetTickCount();
  cout << endl << "Waktu eksekusi Merge Sort dengan "<<n <<" angka acak = " << (int)(t2 - t1) << " ms";
  cout<<endl;
  return 0;
}
```

Screenshot Program :

```
Masukan banya data = 20
Masukan data yang akan di susun = 1 5 7 4 3 2 5 7 4 2 3 4 5 6 2 4 5 3 2 4
1 2 2 2 2 3 3 3 4 4 4 4 4 5 5 5 5 6 7 7
```

B)

```
Waktu eksekusi Merge Sort dengan 20 angka acak = 20391 ms

Process returned 0 (0x0)   execution time : 23.471 s
Press any key to continue.
```

KASUS II

Source Code :

```
#include <iostream>
#include <conio.h>

using namespace std;

int data[10],data2[10];
int n;

void tukar(int a, int b)
{
 int t;
```

```
 t = data[b];
 data[b] = data[a];
 data[a] = t;
}
void selection_sort()
{
 int pos,i,j;
 for(i=1;i<=n-1;i++)
 {
  pos = i;
  for(j = i+1;j<=n;j++)
  {
   if(data[j] < data[pos]) pos = j;
  }
  if(pos != i) tukar(pos,i);
 }
}

int main()
{
 cout<<"Masukkan Jumlah Data : ";
 cin>>n;
 for(int i=1;i<=n;i++)
 {
  cout<<"Masukkan data yang akan diurutkan : ";
  cin>>data[i];
  data2[i]=data[i];
```

```
}
selection_sort();
cout<<"Data Setelah di Sort : ";
for(int i=1; i<=n; i++)
{
 cout<<" "<<data[i];
}
cout<<"\n\nSorting dengan selection sort Selesai";
getch();
}
```

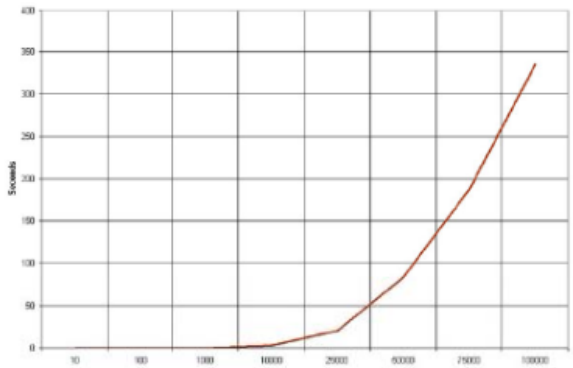
Kompleksitas Algoritma :

Algoritma di dalam Selection Sort terdiri dari kalang bersarang. Dimana kalang tingkat pertama (disebut pass) berlangsung N-1 kali. Di dalam kalang kedua, dicari elemen dengan nilai terkecil. Jika didapat, indeks yang didapat ditimpakan ke variabel min. Lalu dilakukan proses penukaran. Begitu seterusnya untuk setiap Pass. Passsendiri makin berkurang hingga nilainya menjadi semakin kecil. Berdasarkan operasi perbandingan elemennya:

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

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

Berarti kompleksitasnya secara simptotik adalah $O(n^2)$. Adapun grafik efisiensi selection sort dapat dilihat pada tabel dibawah ini:



Gambar 2. Grafik Kompleksitas Selection Sort

KASUS III

Source Code :

```
#include <cstdlib>
#include <iostream>
 #include <conio.h>
using namespace std;

//member function
void insertion_sort(int arr[], int length);
void print_array(int array[],int size);

int main() {
 cout<<"\tINSERTION SORT\n\n";
int array[5]= {5,4,3,2,1};
insertion_sort(array,5);
 getch();
 return 0;
}//end of main

void insertion_sort(int arr[], int length) {
 int i, j ,tmp;

 for (i = 1; i < length; i++) { //1. Pengecekan mulai dari data ke-1 sampai  data ke-n
 j = i;
 while (j > 0 && arr[j - 1] > arr[j]) {  //2. Bandingkan data ke-I ( I = data ke-2 s/d data ke-n )
 tmp = arr[j];
```

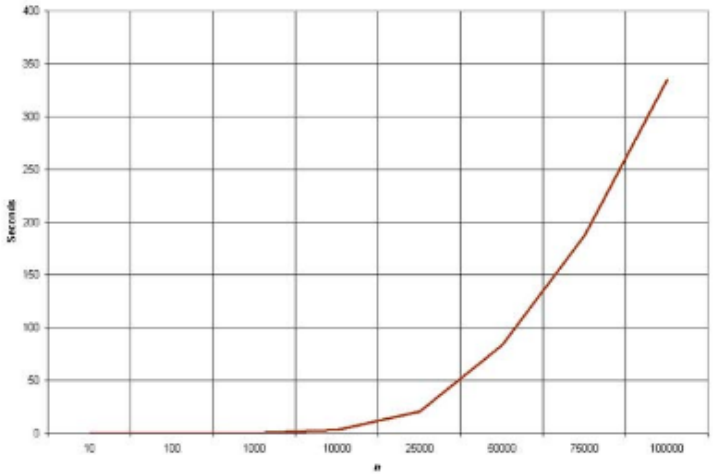
```
 arr[j] = arr[j - 1]; //3. Bandingkan data ke-I tersebut dengan data sebelumnya (I-1), Jika lebih kecil maka data tersebut dapat disisipkan ke data awal sesuai dgn posisisi yg seharusnya


 arr[j - 1] = tmp;

 j--;

 //4. Lakukan langkah 2 dan 3 untuk bilangan berikutnya ( I= I+1 ) sampai didapatkan urutan yg optimal.

 }//end of while loop

 print_array(arr,5);

 }//end of for loop

 }//end of insertion_sort.

void print_array(int array[], int size){

 cout<< "Pengurutan : ";

 int j;

 for (j=0; j<size;j++)

 for (j=0; j<size;j++)

 cout <<" "<< array[j];

 cout << endl;

 }//end of print_array
```

Kompleksitas Algoritma :

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.

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

Secara Kompleksitas, selection sort dan insertion sort mempunyai Big-Oh yang sama. Walaupun begitu, *insertion sort* sebenarnya lebih mangkus. Perhatikan gambar berikut:



Gambar 3. Grafik Kompleksitas Insertion Sort

KASUS IV

Source Code :

```
#include <cstdlib>

#include <iostream>

#include<conio.h>

using namespace std;


void print_array(int array[], int size) {

 cout<< "Proses Langkah Pengurutan Bubble Sort: ";

 int j;

 for (j=0; j<size;j++)

 cout <<" "<< array[j];

 cout << endl;
```

```
}//end of print_array


void bubble_sort(int arr[], int size) {
 bool not_sorted = true;
 int j=1,tmp;


 while (not_sorted)  {
 not_sorted = false;
 j++;
 //1. Pengecekan Mulai dari data ke-1 sampai data ke-n
 for (int i = 0; i < size - j; i++) {
 //2. Bandingkan data ke-n dengan data sebelumnya (n-1)
 if (arr[i] > arr[i + 1]) {
 tmp = arr[i];
 //3. Jika lebih kecil maka pindahkan bilangan tersebut dengan bilangan yg ada didepannya
( sebelumnya  ) satu persatu  (n-1,n-2,n-3,....dst)
 arr[i] = arr[i + 1];
 arr[i + 1] = tmp;
 not_sorted = true;
 //4. Jika lebih besar maka tidak terjadi pemindahan
    }//end of if
   print_array(arr,5);
  }//end of for loop
  //5Ulangi langkah 2 dan 3 s/d sort optimal.
 }//end of while loop
}//end of bubble_sort
```

```
int main() {
   cout<<"\tBUBBLE SORT\n\n";
 int array[5]= {5,4,3,2,1};
 print_array(array,5);
 bubble_sort(array,6);
 getch();
 return 0;
}//end of main
```

Kompleksitas Algoritma :

Kompleksitas Algoritma Bubble Sort dapat dilihat dari beberapa jenis kasus, yaitu worst-case, dan average-case.

Kondisi Best-Case

Dalam kasus ini, data yang akan disorting telah terurut sebelumnya, sehingga proses perbandingan hanya dilakukan sebanyak (n-1) kali, dengan satu kali pass. Proses perbandingan dilakukan hanya untuk memverifikasi keurutan data. Contoh Best-Case dapat dilihat pada pengurutan data “1 2 3 4” di bawah ini.

Pass Pertama

(1 2 3 4) menjadi (1 2 3 4)

(1 2 3 4) menjadi (1 2 3 4)

(1 2 3 4) menjadi (1 2 3 4)

Dari proses di atas, dapat dilihat bahwa tidak terjadi penukaran posisi satu kalipun, sehingga tidak dilakukan pass selanjutnya. Perbandingan elemen dilakukan sebanyak tiga kali.

Proses perbandingan pada kondisi ini hanya dilakukan sebanyak (n-1) kali. Persamaan Big-O yang diperoleh dari proses ini adalah O(n). Dengan kata lain, pada kondisi Best-Case algoritma Bubble Sort termasuk pada algoritma lanjar.

Kondisi Worst-Case

Dalam kasus ini, data terkecil berada pada ujung array. Contoh Worst-Case dapat dilihat pada pengurutan data “4 3 2 1” di bawah ini.

Pass Pertama

(4 3 2 1) menjadi (3 4 2 1)

(3 4 2 1) menjadi (3 2 4 1)

(3 2 4 1) menjadi (3 2 1 4)

Pass Kedua

(3 2 1 4) menjadi (2 3 1 4)

(2 3 1 4) menjadi (2 1 3 4)

(2 1 3 4) menjadi (2 1 3 4)

Pass Ketiga

(2 1 3 4) menjadi (1 2 3 4)

(1 2 3 4) menjadi (1 2 3 4)

(1 2 3 4) menjadi (1 2 3 4)

Pass Keempat

(1 2 3 4) menjadi (1 2 3 4)

(1 2 3 4) menjadi (1 2 3 4)

(1 2 3 4) menjadi (1 2 3 4)

Dari langkah pengurutan di atas, terlihat bahwa setiap kali melakukan satu pass, data terkecil akan bergeser ke arah awal sebanyak satu step. Dengan kata lain, untuk menggeser data terkecil dari urutan keempat menuju urutan pertama, dibutuhkan pass sebanyak tiga kali, ditambah satu kali pass untuk memverifikasi. Sehingga jumlah proses pada kondisi best case dapat dirumuskan sebagai berikut.

Jumlah proses = n2+n (3)

Dalam persamaan (3) di atas, n adalah jumlah elemen yang akan diurutkan. Sehingga notasi Big-O yang didapat adalah O(n2). Dengan kata lain, pada kondisi worst-case, algoritma Bubble Sort termasuk dalam kategori algoritma kuadratik.

Kondisi Average-Case

Pada kondisi average-case, jumlah pass ditentukan dari elemen mana yang mengalami penggeseran ke kiri paling banyak. Hal ini dapat ditunjukkan oleh proses pengurutan suatu array, misalkan saja (1 8 6 2). Dari (1 8 6 2), dapat dilihat bahwa yang akan mengalami proses penggeseran paling banyak adalah elemen 2, yaitu sebanyak dua kali.

Pass Pertama

(1 8 6 2) menjadi (1 8 6 2)

(1 8 6 2) menjadi (1 6 8 2)

(1 6 8 2) menjadi (1 6 2 8)

Pass Kedua

(1 6 2 8) menjadi (1 6 2 8)

(1 6 2 8) menjadi (1 2 6 8)

(1 2 6 8) menjadi (1 2 6 8)

Pass Ketiga

(1 2 6 8) menjadi (1 2 6 8)

(1 2 6 8) menjadi (1 2 6 8)

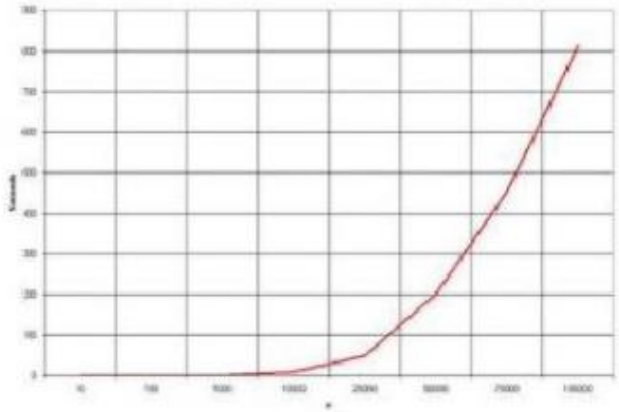
(1 2 6 8) menjadi (1 2 6 8)

Dari proses pengurutan di atas, dapat dilihat bahwa untuk mengurutkan diperlukan dua buah passing, ditambah satu buah passing untuk memverifikasi. Dengan kata lain, jumlah proses perbandingan dapat dihitung sebagai berikut.

Jumlah proses = x2+x (4)

Dalam persamaan (4) di atas, x adalah jumlah penggeseran terbanyak. Dalam hal ini, x tidak pernah lebih besar dari n, sehingga x dapat dirumuskan sebagai Dari persamaan (4) dan (5) di atas, dapat disimpulkan bahwa notasi big-O nya adalah O(n2). Dengan kata lain, pada kondisi average case algoritma Bubble Sort termasuk dalam algoritma kuadratik.

Adapun grafik efisiensi bubble sort dapat dilihat pada tabel dibawah ini:



Gambar 1. Grafik Kompleksitas Bubble Sort