INFORMATIKA, Vol.3 September 2016, pp. 208~218
ISSN: 2355-6579
E-ISSN**:** 2528-2247

ANALISA PERBANDINGAN KOMPLEKSITAS ALGORITMA *SELECTIONSORT* DAN *INSERTIONSORT*

Reza Maulana

Program Studi Magister Ilmu Komputer, STMIK Nusa Mandiri Jakarta
Jl. Salemba Raya No.5, Jakarta Pusat, Indonesia
e-mail: reza.rza@bsi.ac.id

Abstract

Sorting of data is one of the critical operations in data processing. There are various sorting algorithms. However, only some of which are used as the basis for the introduction and understanding of the learning process in an institution. Among the selection sort and insertion sort. Both sorting algorithm that is the focus of discussion on this scientific work. Is the domain on this occasion is a time to try to analyze the speed of the algorithm selection sort and insertion sort, with all the data input in the form of integers in a list or array of data at random intervals between 1,000 and 100,000 elements. Sorting algorithms selection algorithm has a complexity of O (n^2), the same as the insertion sorting algorithms. Each of these types of algorithms have different levels of effectiveness. The effectiveness of an algorithm can be measured by how the amount of time it takes to run the algorithm. This article will only discuss and analyze the complexity of time for each type of algorithm. The algorithm is written in this paper is an algorithm that is implemented in C ++ programming language. Based on the complexity of the test of time, it can be concluded that the insertionsort algorithm is faster, than the selectionsort algorithm.

Keywords*: Algorithm Complexity, Sorting, Selection Sort, Insertion Sort*

1. Pendahuluan

Istilah algoritma, mungkin bukan sesuatu yang asing bagi kita. Ditinjau dari asal-usul katanya, kata 'Algoritma' mempunyai sejarah yang agak aneh. Orang hanya menemukan kata Algorism yang berarti proses menghitung dengan angka Arab. Seseorang dikatakan 'Algorist' jika menghitung menggunakan angka Arab. Para ahli bahasa berusaha menemukan asal kata ini namun hasilnya kurang memuaskan. Akhirnya para ahli sejarah matematika menemukan asal kata tersebut yang berasal dari nama penulis buku Arab terkenal, yaitu Abu Abdullah Muhammad Ibnu Musa Al-Khuwarizmidibaca orang barat menjadi Algorism.

Algortima merupakan kumpulan perintah atau instruksi yang dibuat secara jelas dalam penyelesaian suatu masalah. Kumpulan perintah atau instruksi yang dimaksud dibangun dari suatu bahasa pemrograman yang dapat dimengerti oleh komputer. Kumpulan perintah atau instruksi tersebut berupa langkah-langkah atau tahapan-tahapan yang diberikan untuk menyelesaikan suatu pekerjaan atau suatu masalah secara cepat, lebih mudah, dan efisien.

Algortima dapat kita asumsikan kurang lebih sama dengan suatu prosedur atau kegiatan yang sering kita lakukan dalam kehidupan kita sehari-hari. Misalnya dalam membuat atau menyeduh secangkir kopi, langkah-langkahnya sebagai berikut: masak air hingga mendidih, masukkan kopi dalam secangkir gelas kurang lebih sebanyak empat sendok teh, kemudian tuang air yang sudah kita masak kedalam cangkir tersebut, aduk hingga serbuk kopi tidak mengapung lagi, setelah itu masukkan gula sebanyak dua sendok teh, aduk hingga gula larut dalam air kopi, dan akhirnya kopi siap untuk di minum.

Salah satu pemanfaatan algortima dalam dunia pemrograman yaitu algortima pengurutan atau *sorting algorithm,* yang sering digunakan dalam proses pengurutan suatu data.

Pengurutan data atau yang lebih dikenal istilah *sorting* merupakan salah satu hal yang amat sangat penting bagi dunia

Diterima Agustus 5, 2016; Revisi Agustus 15, 2016; Disetujui Agustus 30, 2016

keinformatikaan. Terutama dalam hal pengelolaan data. Karena dengan pengurutan, proses pengelolaan data dalam dilakukan secara cepat, lebih mudah, dan efisien.

Algoritma pengurutan atau *sorting algorithm* adalah algoritma yang menyimpan suatu list pada suatu urutan tertentu, biasanya membesar atau mengecil, biasanya digunakan untuk mengurutkan angka ataupun huruf. Efisiensi pada pengurutan ini diperlukan untuk mengoptimalkan kecepatan pemrosesan. Semakin efisien suatu algoritma, maka pada saat dieksekusi dan dijalankan akan menghabiskan waktu yang lebih cepat dan bisa menerima lebih banyak masukan dari user.

Ada berbagai macam algoritma pengurutan. Namun, hanya beberapa yang dipakai sebagai dasar perkenalan dan pemahaman terhadap proses belajar dalam suatu institusi. Diantaranya yaitu s*election* s*ort* dan *insertion* s*ort*. Kedua algortima pengurutan inilah yang menjadi fokus pembahasan pada karya ilmiah ini. Yang menjadi domain pada kesempatan ini adalah mencoba untuk menganalisis kecepatan waktu dari algoritma *selection sort* dan *insertion sort*, dengan sekali inputan data berupa bilangan bulat (*integer*) dalam suatu list atau array secara acak (*random*) dengan interval data antara 1.000 sampai dengan 100.000 elemen. Dari kedua algoritma ini yang manakah lebih cepat dan efisien dalam menyelesaikan pengurutan dengan interval data tersebut. Waktu eksekusi diukur dengan satuan *milisecond* (ms).

Kompleksitas waktu untuk algoritma-algoritma yang dibahas akan dinyatakan dengan notasi O besar *(Big-O notation)*. Definisi dari notasi O besar adalah, jika sebuah algoritma mempunyai waktu asimptotik *O(f(n)),* maka jika n dibuat semakin besar, waktu yang dibutuhkan tidak akan pernah melebihi suatu konstanta C dikali dengan *f(n).* Jadi *f(n)* adalah adalah batas atas *(upper bound)* dari *T(n)* untuk n yang besar. *O(n)* dihitung berdasarkan banyaknya jumlah operasi perbandingan yang dilakukan dalam algoritma tersebut.

Ada beberapa penelitian tentang perbandingan algoritma pengurutan dengan menggunakan beberapa metode algoritma yang mendukung penelitian ini. Yahya (2014) telah melakukan penelitian untuk perbandingan algoritma *bubble sort* dan *selection sort* dengan metode perbandingan eksponensial. Penelitian ini dilakukan dengan membandingkan algortima *bubble sort* dan *selection sort* dengan menggunakan metode pembanding yaitu metode perbandingan eksponensial. Hasil dari penelitian ini menyatakan dari perbandingan eksponensial yang dilakukan terbukti bahwa algoritma *selection sort* lebih unggul dan lebih cepat dalam mengurutkan data dibandingkan dengan algortima *bubble sort.*

Penelitian lain yang dilakukan oleh Suryani (2013) dalam perbandingan metode *bubble sort* dan *insertion sort* terhadap efisiensi memori. Penelitian ini dilakukan dengan membandingkan algortima *bubble sort* dan *insertion sort* dengan menggunakan bahasa pemrograman C++ terhadap 100 sampai 1500 data dengan kelipatan 100 data yang masing-masing bertipe *integer.* Hasil dari penelitian ini menyatakan bahwa algoritma *bubble sort* tetap menggunakan memori yang lebih efisien dibandingkan dengan metode *insertion sort*.

Untuk mengetahui kecepatan waktu tiap algoritma, maka digunakan perangkat lunak penghitung kecepatan algoritma dalam bahasa pemrograman C++. Alasan mengapa menggunakan bahasa pemrograman C++ pada penelitian ini, karena bahasa pemrograman C++ merupakan bahasa yang berkekuatan tinggi dan fleksibel, yang telah banyak digunakan oleh para programmer professional untuk mengembangkan program-program yang sangat bervariasi dalam berbagai bidang.

1.1. Landasan Teori

A. Algortima

Algoritma pada dasarnya adalah alur pikiran dalam menyelesaikan masalah suatu pekerjaan, yang dituangkan dalam bentuk tertulis yang dapat dimengerti orang lain (Yahya, 2014). Algoritma adalah langkah-langkah yang logis dan terstruktur untuk menyelesaikan suatu permasalahan. (Wahyudi, 2007). Jadi, berdasarkan kedua pendapat tersebut dapat disimpulkan bahwa Algortima adalah suatu kumpulan perintah yang logis dan terstruktur untuk menyelesaikan suatu pekerjaan atau tugas tertentu.

B. Pengurutan Data

Pengurutan atau *sorting* adalah proses mengatur sekumpulan objek menurut urutan

atau susunan tertentu (Munir, 2011). Biasanya pengurutan terbagi menjadi dua yaitu *Ascending* (pengurutan dari karakter/angka kecil ke karakter/angka besar dan *Descending*(pengurutan dari karakter/angka besar ke karakter/angka kecil).

Terdapat banyak cara yang dapat dilakukan untuk melakukan proses pengurutan dari paling tinggi ke paling rendah atau sebaliknya. Untuk masalah pengurutan pada Array kita tidak bisa langsung menukar isi dari variable yang ada, tetapi menggunakan metode penukaran (Swap).

Contoh :

Misalkan bahwa score [1] = 10 dan score [2] = 8 dan ingin ditukar nilai – nilainya sehingga score [1] = 8 dan score [2] = 10. Hal ini tidak bisa langsung dilakukan dengan menukar isi dari dua variable tersebut, seperti cara ini :

score [1] = score [2];
score [2] = score [1];

Cara yang tepat adalah :

temp = score [1]
score [1] = score [2];
score [2] = temp;

C. Kompleksitas Waktu

Kompleksitas Waktu, T(n), adalah jumlah operasi yang dilakukan untuk melaksanakan algoritma sebagai fungsi dari ukuran masukan (n). Maka, dalam mengukur kompleksitas waktu dihitunglah banyaknya operasi yang dilakukan oleh algoritma. Idealnya, kita memang harus menghitung semua operasi yang ada dalam suatu algoritma. Namun, untuk alasan praktis, cukup menghitung jumlah operasi abstrak yang mendasari suatu algoritma. Operasi abstrak ini disebut Operasi Dasar. (Tjaru, 2010)

Biasanya kompleksitas algoritma dinyatakan dengan notasi O besar *(Big-O notation)*. Notasi O besar adalah, jika sebuah algoritma mempunyai waktu asimptotik *O(f(n)),* maka jika n dibuat semakin besar, waktu yang dibutuhkan tidak akan pernah melebihi suatu konstanta C dikali dengan *f(n).* Jadi *f(n)* adalah adalah batas atas *(upper bound)* dari *T(n)* untuk n yang besar. *O(n)* dihitung berdasarkan banyaknya jumlah operasi perbandingan yang dilakukan dalam algoritma tersebut (Atrinawati, 2007).

Jika kompleksitas waktu untuk menjalankan suatu algoritma dinyatakan dengan *T(n)*, dan memenuhi

$$T(n) \leq C(f(n))$$

untuk$n \geq n0,$
maka kompleksitas dapat dinyatakan dengan

$$T(n) = O(f(n)).$$

Terdapat 2 jenis penggunaan notasi Big O (Tjaru, 2010), yaitu:

1. *Infinite asymptotics*
2. *Infinitesimal asymptotics*

Perbedaan kedua jenis penggunaan notasi ini hanya pada aplikasi. Sebagai contoh, pada
infinite asymptotics dengan persamaan

$$T(n) = 2n2 - 2n + 2$$

Untuk n yang besar, pertumbuhan T(n) akan sebanding dengan n2 dan dengan mengabaikan suku yang tidak mendominasi kita, maka kita tuliskan

$$T(n) = O(n2)$$

Pada *infinitesimal asymptotics*, Big O digunakan untuk menjelaskan kesalahan dalam aproksimasi untuk sebuah fungsi matematika, sebagai contoh

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2} + O(x^3), \qquad x \to 0$$

Kesalahannya memiliki selisih

$$e^x - (1 + \frac{x}{1} + \frac{x^2}{2})$$

D. Bahasa Pemrograman C++

Bahasa C merupakan bahasa pemrograman tingkat tinggi, itu dikarenakan bahasa C dapat dimengerti dan dipelajari dengan mudah karena kedekatanya dengan bahasa manusia. Tapi banyak orang juga mengatakan bahwa bahasa C adalah *medium level progamming language* karena bahasa C juga dapat digunakan untuk memasukkan program ke mesin (Joni dan Raharjo, 2011).

2. Metode Penelitian

2.1. *Selection Sort*

Menurut Munir (2011), *Selection sort* adalah suatu metode pengurutan yang memilih elemen maksimum/minimum dari larik, lalu menempatkan elemen maksimum/minimum itu pada awal atau akhir larik (elemen terujung).

Konsep proses *selection sort* adalah mencari (memilih) nilai terkecil dan menukarnya dengan elemen paling awal

(paling kiri) pada setiap tahap. Proses Sort dilakukan tahap per tahap. Untuk n = 7 maka akan dilakukan (n – 1) = 6 tahap (Yahya, 2014).

2.2. *Insertion Sort*

Menurut Munir (2011) menjelaskan bahwa *Insertion Sort* adalah metode pengurutan dengan cara menyisipkan elemen larik pada posisi yang tepat.

Cara kerja algoritma ini adalah dengan mengambil elemen list satu-per-satu dan memasukkannya di posisi yang benar seperti namanya. Pada array, list yang baru dan elemen sisanya dapat berbagi tempat di array, meskipun cukup rumit. Untuk menghemat memori, implementasinya menggunakan pengurutan di tempat yang membandingkan elemen saat itu dengan elemen sebelumnya yang sudah diurut, lalu menukarnya terus sampai posisinya tepat. Hal ini terus dilakukan sampai tidak ada elemen tersisa di input (Tjaru, 2010).

3. Hasil dan Pembahasan

Walaupun masing-masing algoritma pengurutan mempunyai perbedaan metode dan sudut pandang dalam penyelesaian masalah, namun kompleksitas waktu yang dibutuhkan tetap menjadi fokus utama yang harus menjadi perbandingan untuk menentukan algortima mana yang lebih baik dan tepat untuk digunakan. Untuk mengetahui kecepatan waktu tiap algoritma, maka digunakan perangkat lunak penghitung kecepatan algoritma dalam bahasa pemrograman C++. Hal ini digunakan untuk mendukung analisis kecepatan pengurutan *(sorting)* data.

Dengan mengatur perangkat lunak agar banyaknya jumlah n sesuai yang diinput secara acak, sehingga kemunculan data setiap kali program dijalankan tidak sama dan tidak dapat diprediksi dan interval data antara 1.000 sampai dengan 100.000 elemen. Pengujian dilakukan disebuah laptop dengan spesifikasi, sebagai berikut: platform Windows 10 Pro, processor AMD E-450 APU with Radeon(tm) HD Graphics 1,65 GHz, dan 2 GB DDR3 RAM.

3.1. Konsep *Selection Sort*

Algoritma pengurutan sederhana salah satunya adalah *Selection Sort*. Ide dasarnya adalah melakukan beberapa kali *pass* untuk melakukan penyeleksian elemen struktur data. Untuk *sorting ascending* (menaik), elemen yang paling kecil di antara elemen-elemen yang belum urut, disimpan indeksnya, kemudian dilakukan pertukaran nilai elemen dengan indeks yang disimpan tersebut dengan elemen yang paling depan yang belum urut. Sebaliknya, untuk *sorting descending* (menurun), elemen yang paling besar yang disimpan indeksnya kemudian ditukar.

Selection Sort diakui karena kesederhanaan algoritmanya dan performanya lebih bagus daripada algoritma lain yang lebih rumit dalam situasi tertentu.

Algoritma ini bekerja sebagai berikut:

1. Mencari nilai minimum (jika *ascending*) atau maksimum (jika *descending*) dalam sebuah list
2. Menukarkan nilai ini dengan elemen pertama list
3. Mengulangi langkah di atas untuk sisa list dengan dimulai pada posisi kedua.

3.2. Simulasi *Selection Sort*

Untuk lebih jelasnya, perhatikanlah simulasi *selection sort,* sebagai berikut:

	5	3	42	23	7	12	32
Pass							
0	5	3	42	23	7	12	32
1	3	3	42	23	7	12	32
2	3	5	42	23	7	12	32
3	3	5	7	23	42	12	32
4	3	5	7	12	42	23	32
5	3	5	7	12	23	43	32
6	3	5	7	12	23	32	43

Gambar 1. Simulasi algoritma *selection sort*

Dalam satu kali *pass,* ditentukan elemen yang paling kecil di dalam bagian list yang belum urut. Elemen yang paling kecil ini, diwarnai biru . Untuk bagian larik yang telah diurutkan diberi warna merah.

3.3. Notasi Algoritmik *Selection Sort*

Procedure *SelectionSort*
(Input/Output T: TabInt, Input N:integer)
{mengurut tabel integer [1 .. N] dengan Selection Sort secara ascending}

Kamus:
i: integer
Pass: integer
min: integer
Temp: integer

Algoritma:
Pass traversal [1..N-1]
Min ← Pass
i traversal [Pass+1..N]
if (Ti < Tmin) then
min ← i
Temp ← TPass
TPass ← Tmin
Tmin ← Temp
{T[1..Pass] terurut}

3.4. Kompleksitas *Selection Sort*

Algoritma pada s*election 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*. *Pass* sendiri makin berkurang hingga nilainya menjadi semakin kecil. Berdasarkan operasi perbandingan elemennya:

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

Berarti kompleksitasnya secara simptotik adalah O(n^2).

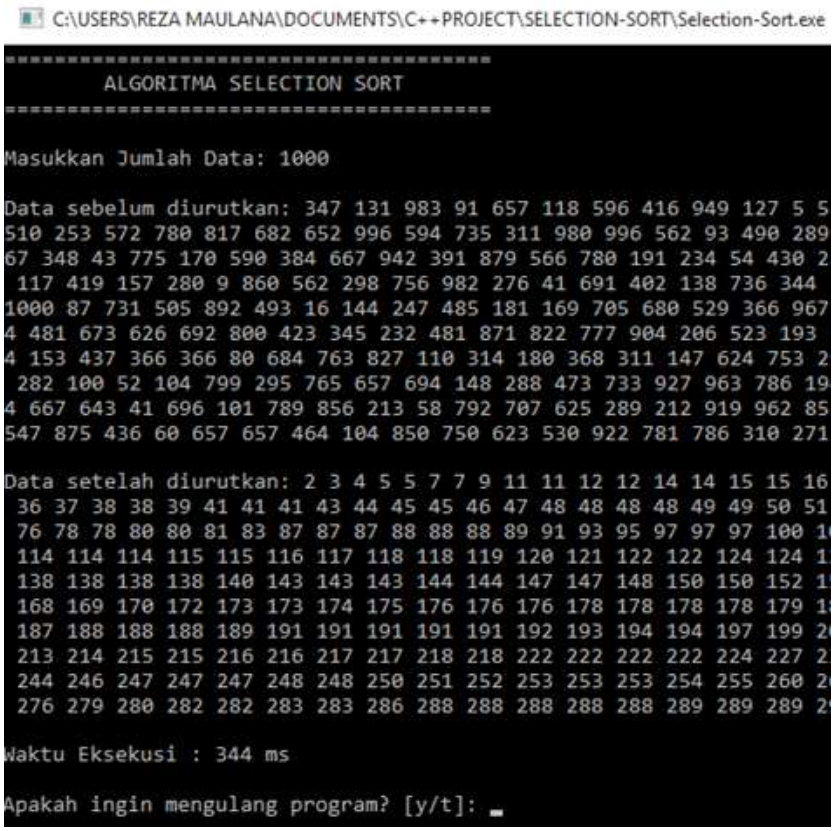
3.5. Implementasi *Selection Sort*

Penerapan algoritma *selection sort* menggunakan antar muka bahasa pemrograman yaitu C++, dengan kumpulan perintah sebagai berikut:

```
#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>

int main()
{
int n, data[100000], i, j, temp;
char ulang;
do
 {
clrscr();
cout<<"============================
========== \n";
cout<<"\tALGORITMA SELECTION SORT
\n" ;
cout<<"============================
========== \n\n";

cout<<"Masukkan Jumlah Data: ";
cin>>n;
cout<<endl;
printf("Data sebelum diurutkan: ");
for (i = 0; i < n; i++)
 {
data[i]=rand() % n + 1;
cout<<data[i]<<" ";
 }
cout<<endl<<endl;

double start = clock();
srand(time(NULL));
          for(i=0; i<(n-1); i++){
for(j=i+1; j<n; j++){
if(data[i]>data[j]){
temp=data[i];
data[i]=data[j];
data[j]=temp;
          }
       }
   }

printf("Data setelah diurutkan: ");
for (i = 0; i <= n - 1; i++) {
cout<<data[i]<<" ";
 }
double stop = clock();
cout<<endl<<endl;
 cout<<"Waktu Eksekusi : "<<(stop-start)<<"
ms";
cout<<endl<<endl;

cout<<"Apakah ingin mengulang program?
[y/t]: ";
cin>>ulang;
 }
```

```
while(ulang=='y');
return 0;
}
```

Gambar berikut adalah hasil dari *compile* program algoritma *selection sort* pada Borland C++ dengan jumlah data 1.000 elemen.



Gambar 2. Hasil program algoritma *selection sort*

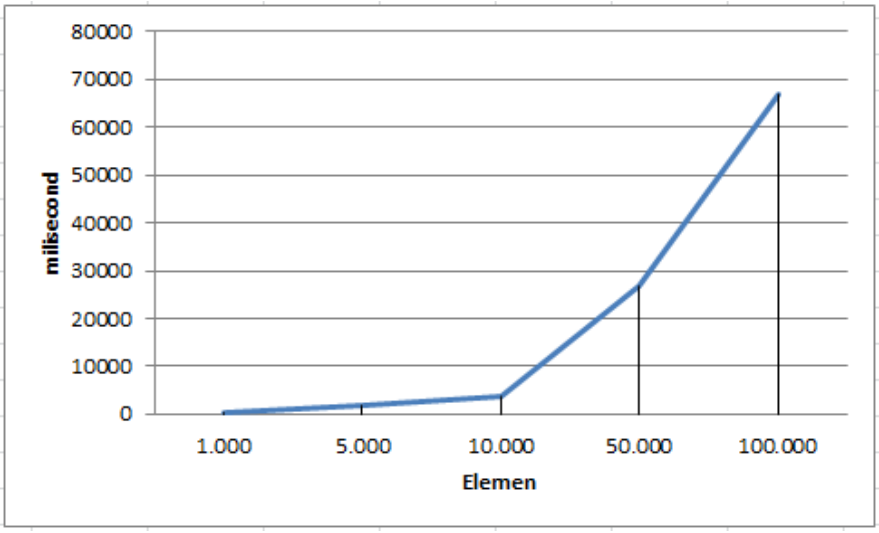
3.6. Analisa *Selection Sort*

Pada setiap elemen nilai sampai n perlu membandingkan dan memeriksa apakah ada nilai yang lebih kecil dari kebutuhan perbandingan (n-1). Dengan demikian dapat dengan mudah dilihat bahwa kompleksitas waktu algortima *selection sort* adalah $O(n^2)$. Tabel berikut menjelaskan waktu yang dibutuhkan algortima *selection sort* dalam proses mengurutkan data. Dengan interval data antara 1.000 sampai dengan 100.000 elemen. Waktu eksekusi diukur dengan satuan *milisecond* (ms).

Tabel 1. Waktu eksekusi algoritma *selection sort*

Jumlah Elemen Array	Waktu (ms)
1.000	344
5.000	1.828
10.000	3.844
50.000	26.797
100.000	66.953

Kompleksitas algortima *selection sort* dalam proses mengurutkan data, dapat dilihat pula pada grafik berikut ini:



Gambar 3. Grafik kompleksitas algoritma *selection sort*

3.7. *Insertion Sort*

3.7.1. Konsep *Insertion Sort*

Algoritma *insertion sort* adalah sebuah algoritma sederhana yang cukup efisien untuk mengurutkan sebuah list yang hampir terurut. Algorima ini juga bisa digunakan sebagai bagian dari algoritma yang lebih canggih.

Pengurutan dengan penyisipan bekerja dengan cara menyisipkan masing-masing nilai di tempat yang sesuai (di antara elemen yang lebih kecil atau sama dengan nilai tersebut dengan elemen yang lebih besar atau sama dengan nilai tersebut).

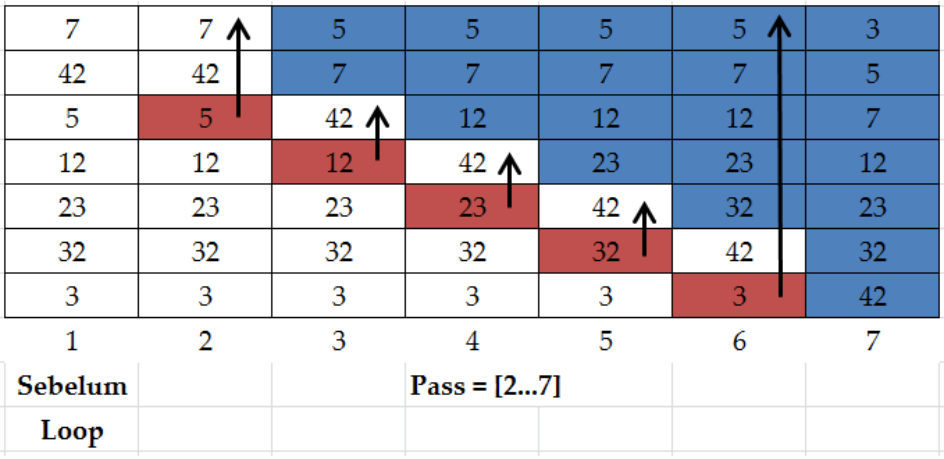
Salah satu implementasinya pada kehidupan sehari-hari adalah saat kita mengurutkan kartu remi. Kita ambil kartu satu per-satu lalu membandingkan dengan kartu sebelumnya untuk mencari posisi yang tepat.

Variasi pada umunya yang dilakukan terhadap array pada insertion sort adalah sebagai berikut :

1. Elemen awal di masukkan sembarang, lalu elemen berikutnya dimasukkan di bagian paling akhir. Elemen tersebut dibandingkan dengan elemen ke (x-1). Bila belum terurut posisi elemen sebelumnya digeser sekali ke kanan terus sampai elemen yang sedang diproses menemukan posisi yang tepat atau sampai elemen pertama.
2. Setiap pergeseran akan mengganti nilai elemen berikutnya, namun hal ini tidak menjadi persoalan sebab elemen berikutnya sudah diproses lebih dahulu.

3.7.2. Simulasi *Insertion Sort*

Untuk lebih jelasnya, perhatikanlah simulasi *insertion sort,* sebagai berikut:



Gambar 4. Simulasi algoritma *insertion sort*

Setiap satu kali *Pass* akan ada satu nilai yang disisipkan. Kemudian pada *Pass* n-1 data akan terurut. Data yang telah terurut diberi warna biru. Panah menunjukkan perubahan posisi nilai yang akan disisipkan. Data yang akan dipindah diberi warna merah.

3.8. Notasi Algoritmik *Insertion Sort*

Procedure InsertionSort
(Input/Output T: TabInt, Input N: integer)
{mengurut tabel integer [1 .. N] dengan Insertion Sort secara ascending}

Kamus:
i: integer
Pass: integer
Temp: integer

Algoritma:
Pass traversal [2..N]
Temp ←TPass
i ← pass - 1
while (j ≥ 1) and (Ti > Temp) do
Ti+1 ←Ti
i ← i-1
depend on (T, i, Temp)
Temp ≥ Ti : Ti+1 ← Temp
Temp < Ti : Ti+1 ← Ti
Ti ← Temp
{T[1..Pass-1] terurut}

3.8.1. Kompleksitas *Insertion Sort*

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.

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

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

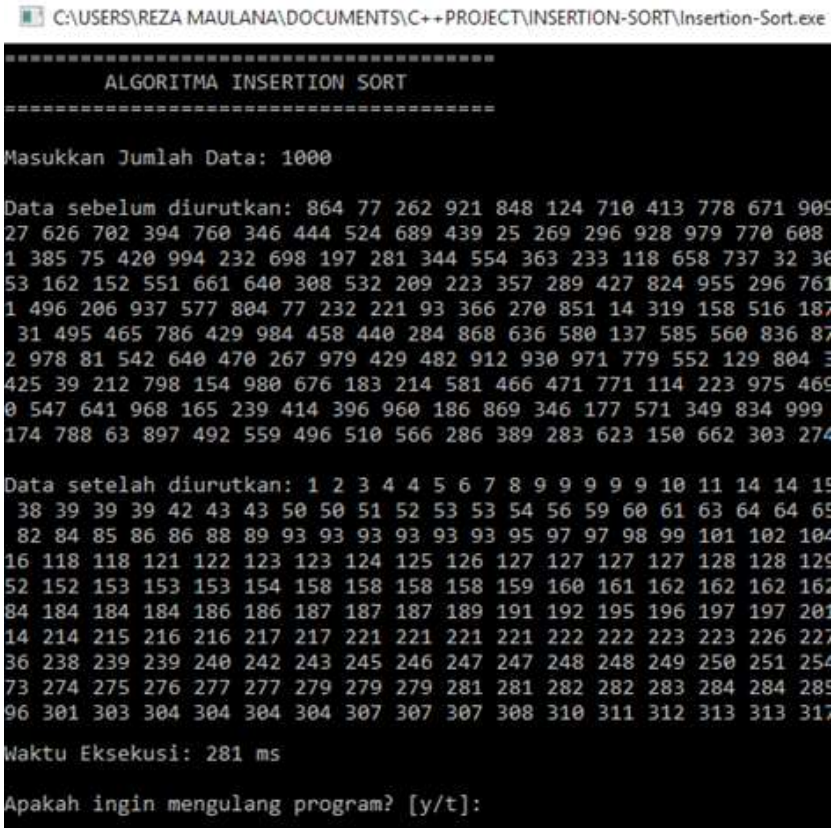
3.9. Implementasi *Insertion Sort*

Penerapan algoritma *insertion sort* menggunakan antar muka bahasa pemrograman yaitu C++, dengan kumpulan perintah sebagai berikut:

```
#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>

int main()
{
int n, data[100000], i, j, temp;
char ulang;
do
 {
clrscr();
cout<<"============================
========== \n";
cout<<"\tALGORITMA INSERTION SORT
\n" ;
cout<<"============================
========== \n\n";

cout<<"Masukkan Jumlah Data: ";
```

```
cin>>n;
cout<<endl;
printf("Data sebelum diurutkan: ");
for (i = 0; i < n; i++)
  {
data[i]=rand() % n + 1;
cout<<data[i]<<" ";
  }
cout<<endl<<endl;

double start = clock();
srand(time(NULL));
          for(i=1; i<=n-1; i++){
temp=data[i];
int j=i;
while(j>=0 && data[j-1]>temp){
data[j]=data[j-1];
j--;
     }
data[j]=temp;
   }
printf("Data setelah diurutkan: ");
for (i = 0; i <= n - 1; i++) {
cout<<data[i]<<" ";
  }
double stop = clock();
cout<<endl<<endl;

  cout<<"Waktu Eksekusi: "<<(stop-start)<<"
ms";
cout<<endl<<endl;

cout<<"Apakah ingin mengulang program?
[y/t]: ";
cin>>ulang;
  }
while(ulang=='y');
return 0;
}
```

Gambar berikut adalah hasil dari *compile* program algoritma *insertion sort* pada Borland C++ dengan jumlah data 1.000 elemen.



Gambar 5. Hasil program algoritma *insertion sort*

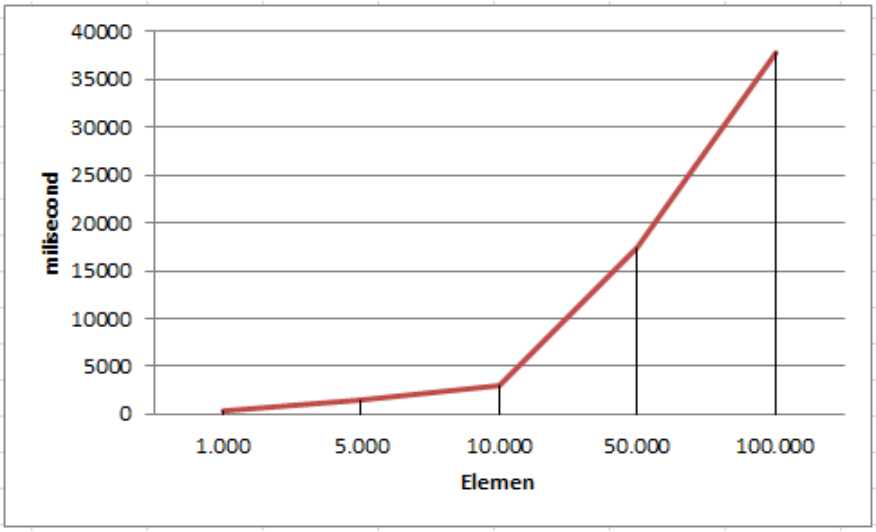
3.10. Analisa *Insertion Sort*

Pada setiap elemen nilai dibandingkan dengan elemen ke (n-1). Bila belum terurut posisi elemen sebelumnya digeser sekali ke kanan terus sampai elemen yang sedang diproses menemukan posisi yang tepat atau sampai elemen pertama. Tabel berikut menjelaskan waktu yang dibutuhkan algortima *insertion sort* dalam proses mengurutkan data. Dengan interval data antara 1.000 sampai dengan 100.000 elemen. Waktu eksekusi diukur dengan satuan *milisecond* (ms).

Tabel 2. Waktu eksekusi algoritma *insertion sort*

Jumlah Elemen Array	Waktu (ms)
1.000	281
5.000	1516
10.000	3078
50.000	17406
100.000	37703

Kompleksitas algortima *selection sort* dalam proses mengurutkan data, dapat dilihat pula pada grafik berikut ini:



Gambar 6. Grafik kompleksitas algoritma *insertion sort*

3.11. Perbandingan Kompleksitas Waktu

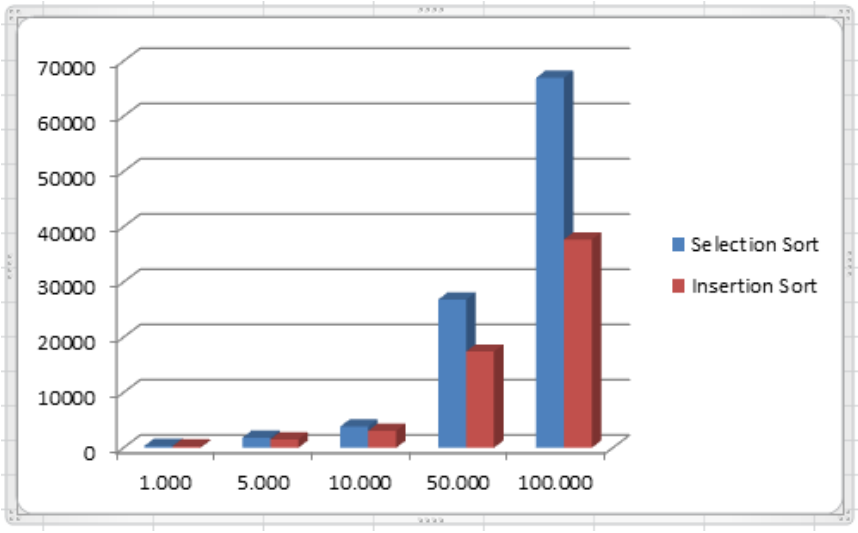
Proses pengurutan dari algortima *selection sort* dan *insertion sort* menggunakan dua cara yang berbeda. Satu dengan cara memilih elemen terkecil dari elemen sekarang maka dicatat posisisnya dan langsung ditukar. Lainnya menggunakan pengurutan di tempat yang membandingkan elemen saat itu dengan elemen sebelumnya yang sudah diurut, lalu menukarnya terus sampai posisinya tepat.

Perbandingan kompleksitas algoritma *selection sort* dan *insertion sort*, dapat dilihat pada table berikut:

Tabel 3. Perbandingan kompleksitas algoritma

Jumlah Elemen Array	Waktu (ms)	
	Selection Sort	Insertion Sort
1.000	344	281
5.000	1828	1516
10.000	3844	3078
50.000	26797	17406
100.000	66953	37703

Grafik berikut menjelaskan perbandingan algoritma *selection sort* dan *insertion sort*, dimana dapat disimpulkan bahwa *insertion sort* lebih mangkus dibandingkan *selection sort.* Walaupun sama-sama berkompleksitas $O(n^2)$.



Gambar 7. Grafik perbandingan kompleksitas algoritma

4. Kesimpulan

Berdasarkan uji kompleksitas waktu logika proses pengurutan data menggunakan algoritma *selection sort* dan *insertion sort* dengan interval data antara 1.000 sampai dengan 100.000 elemen dan waktu eksekusi diukur dengan satuan *milisecond* (ms), maka dapat disimpulkan bahwa algoritma *insertion sort* lebih cepat jika kita melihat tabel dan grafik pada pembahasan, dibandingkan dengan algortima *selection sort.*

Saran

Dalam pengembangan penelitian ini penulis memberikan beberapa saran, yaitu:

1. Pengurutan data yang dilakukan dalam penelitian ini menggunakan data numerik yang diurutkan secara *ascending*. Disarankan dapat menggunakan tipe data string sehingga penerapannya akan menjadi lebih beragam.
2. Diharapkan agar dapat menjadi bahan pembelajaran untuk melakukan proses pengurutan data.
3. Hasil ini dapat dilanjutkan dengan membandingkan antara metode-metode *sorting* lainnya.

Referensi

Atrinawati, L. H. (2007). Analisis Kompleksitas Algoritma Untuk Berbagai Macam Metode Pencarian Nilai *(Searching)* Dan Pengurutan Nilai *(Sorting)* Pada Tabel. *Program Studi Teknik Informatika.* Institut Teknologi Bandung.

Joni, I. M. dan Raharjo, B. (2011). *Pemrograman C dan Implementasinya*. Bandung: Informatika.

Munir, R. (2011). *Algoritma dan Pemrograman*. Bandung; Informatika.

Suarga. (2012). *Algoritma Pemrograman*. Yogyakarta: Andi.

Suryani, D. (2013). Perbandingan Metode Bubble Sort dan Insertion Sort Terhadap Efisiensi Memori. *Jurnal Teknologi Informasi & Pendidikan*. Vol. 6, No. 1.

Tjaru, S. N. B. (2010). Kompleksitas Algoritma Pengurutan Selection Sort dan Insertion Sort. *Program Studi Teknik Informatika*. Institut Teknologi Bandung.

Yahya, S. Y. (2014). Analisa Perbandingan Algoritma *Bubble Sort* dan *Selection Sort* Dengan Metode Perbandingan Eksponensial. *Jurnal Pelita Informatika Budi Darma.* Vol : VI, No : 3.

Wahyudi, E. N. (2007). Algoritma dan Pemrograman Format Laporan dengan Pascal. *Jurnal Teknologi Informasi DINAMIK.* Volume XII, No. 1.