Laporan Tugas

Pengurutan Data dengan Algoritma *Divide & Conquer*

Fadhil Imam Kurnia - 13515146

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

A. Permasalahan

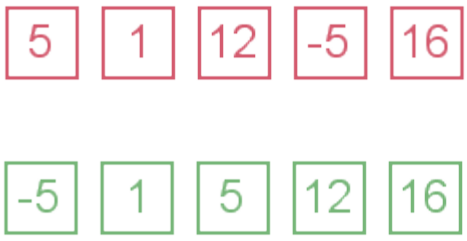
Pengurutan data merupakan salah satu masalah yang krusial dalam bidang ilmu komputer khususnya untuk pemrosesan data. Saat ini banyak permasalahan yang membutuhkan pengurutan, misalnya untuk mengurutkan data barang berdasarkan harga, atau mengurutkan data mahasiswa berdasarkan NIM. Data yang telah terurut akan lebih mudah diolah dibandingkan dengan data yang belum terurut[1].

Pengurutan (*sorting*) merupakan proses pengaturan sederetan angka ke dalam suatu urutan atau susunan tertentu. Urutan bilangan dapat dibuat terurut menaik atau terurut menurun. Data yang dapat diurutkan dapat berupa data bilangan, data karakter maupun data string. Ilustrasi kumpulan bilangaan yang belum terurut kemudian menjadi terurut dapat dilihat pada Gambar 1.

Terdapat banyak algoritma sorting saat ini, tiap algoritma memiliki ciri khasnya masing-masing. Pada laporan ini akan dibahas algoritma pengurutan menggunakan *divide* & *conquer*. Algoritma divide & conquer berusaha membagi permasalahan menjadi lebih kecil agar mudah diselesaikan. Terdapat 2 jenis algoritma *divide* & *conquer*[2], yaitu:

1. Mudah dibagi sulit digabung
2. Sulit dibagi mudah digabung

Algoritma yang termasuk jenis pertama diantaranya adalah *merge sort* dan *insertion sort*, sedangakan algoritma yang termasuk jenis kedua adalah *merge sort* dan *insertion sort*. Dalam laporan ini akan diberikan *source code* dari algoritma pengurutan tersebut.



Gambar 1 Ilustrasi data yang belum terurut dan yang sudah terurut

B. Source Code

Untuk menunjukan algoritma sorting pada tugas ini akan digunakan paradigma pemrograman berbasis objek. Terdapat kelas TabelSorter yang akan digunakan untuk menampung sejumlah bilangan integer, kelas tersebut juga berisi beberapa *method* yang dapat digunakan untuk mengurutkan data menggunakan berbagai algoritma *divide* & *conquer*. Hasil *source code* yang telah dikerjakan adalah sebagai berikut.

TabelSorter.h

```
// File  : TabelSorter.h
// Name  : Fadhil Imam Kurnia - 13515146
// Header file for TabelSorter class

#include <iostream>
using namespace std;

class TabelSorter{
public:
  // Konstruktor TabelSorter
  // Membuat tabel dengan elemen sebanyak size dan diisi dengan bilangan acak
  TabelSorter(int);
  // CopyConstructor
  // Memastikan tidak terjadi bitwise copy pada data
  TabelSorter(const TabelSorter&);
  // Destruktor TabelSorter
  // Melepas memori data yang telah dialokasi pada konstruktor
  ~TabelSorter();
  // Operator Overloading
  // Memastikan tidak terjadi bitwise copy pada data
  TabelSorter& operator=(const TabelSorter&);
  // Prosedur untuk mencetak isi TabelSorter
  void print();
  // Prosedur untuk mencetak isi TabelSorter
  friend ostream& operator<<(ostream&, const TabelSorter&);
  // Prosedur untuk mengurutkan data menggunakan mergesort
  void mergeSort(bool);
  // Prosedur untuk mengurutkan data menggunakan quicksort
  void quickSort(bool);
  // Prosedur untuk mengurutkan data menggunakan selectionsort
  void selectionSort(bool);
  // Prosedur untuk mengurutkan data menggunakan insertionsort
  void insertionSort(bool);

private:
  // Prosedur rekursif untuk mengurutkan data menggunakan mergeSort
  void mergeSortProccess(long, long, bool);
  // Prosedur rekursif untuk mengurutkan data menggunakan quickSort
  void quickSortProcsess(long, long, bool);

  const int size;
  int* data;
};
```

TabelSorter.cpp

```
// File  : TabelSorter.cpp
// Name  : Fadhil Imam Kurnia - 13515146
// Implementation code for TabelSorter class

#include "TabelSorter.h"
#include <cstdlib>

TabelSorter::TabelSorter(int size):size(size){
  data = new int[size];
  for(long i=0; i<size; i++)
    data[i] = rand();
}

TabelSorter::~TabelSorter(){
  delete[] data;
}

TabelSorter::TabelSorter(const TabelSorter& t):size(t.size){
  data = new int[size];
  for(long i=0; i<size; i++)
    data[i] = t.data[i];
}

TabelSorter& TabelSorter::operator=(const TabelSorter& t){
  if(this != &t){
    for(long i=0; i<size; i++)
      data[i] = t.data[i];
  }else{
      return *this;
  }
}

void TabelSorter::print(){
  for(long i=0; i<size; i++)
    cout << data[i] << " ";
```

```
}

ostream& operator<<(ostream& os, const TabelSorter& t){
  os << "";
  for(long i=0; i<t.size; i++)
    os << t.data[i] << endl;
  return os;
}

void TabelSorter::mergeSort(bool isIncrease){
  mergeSortProccess(0,size-1,isIncrease);
}

void TabelSorter::quickSort(bool isIncrease){
  quickSortProcsess(0,size-1,isIncrease);
}

void TabelSorter::mergeSortProccess(long start, long end, bool isIncrease){
  if (start < end){
    long mid = ((end - start) / 2) + start;
    this->mergeSortProccess(start,mid, isIncrease);
    this->mergeSortProccess(mid+1,end, isIncrease);
    // gabungin
    int* temp = new int[end-start+1];
    long i = start, j = mid+1, k = 0;
    while(i <= mid && j <=end){
      if(isIncrease){
        if(data[i] < data[j]){
          temp[k] = data[i];
          i++;
        }else{
          temp[k] = data[j];
          j++;
        }
      }else{
        if(data[i] > data[j]){
          temp[k] = data[i];
          i++;
        }else{
          temp[k] = data[j];
          j++;
        }
      }
      k++;
    }
    while(i<=mid){
      temp[k] = data[i];
      i++;
      k++;
    }
    while(j<=end){
      temp[k] = data[j];
      k++;
      j++;
    }

    i = start; k = 0;
    while(i <= end){
      data[i] = temp[k++];
      i++;
    }
    delete[] temp;
  }
}

void TabelSorter::quickSortProcsess(long start, long end, bool isIncrease){
  if (start < end){
    long mid = ((end - start) / 2) + start;
    int pivot = data[mid];
    // pisahin sesuai pivot
    long i = start, j = end;
    do {
      if(isIncrease){
        while (data[i] < pivot){
          i++;
        }
        while (data[j] > pivot){
          j--;
        }
      }else{
        while (data[i] > pivot){
          i++;
        }
        while (data[j] < pivot){
          j--;
        }
      }
      if(i<=j){
        int temp = data[i];
        data[i] = data[j];
        data[j] = temp;

        if (i+1 <= end)
          i++;
        if ( j-1 >= start)
          j--;
      }

    } while(i <= j);
    quickSortProcsess(start, j, isIncrease);
    quickSortProcsess(j+1, end, isIncrease);
```

```
  }
}

void TabelSorter::insertionSort(bool isIncrease){
  long j = 0;
  for (long i=0 ; i<size; i++){
    int idxExtreme = i;
    j = i;
    while(j < size){
      if(isIncrease){
        if (data[j] < data[idxExtreme]){
          idxExtreme = j;
        }
      }else{
        if (data[j] > data[idxExtreme]){
          idxExtreme = j;
        }
      }
      j++;
    }
    int temp = data[idxExtreme];
    data[idxExtreme] = data[i];
    data[i] = temp;
  }
}

void TabelSorter::selectionSort(bool isIncrease){
  long j = 1;
  for(long i=1; i < size; i++){
    j = i;
    while(j > 0){
      if (isIncrease){
        if (data[j-1] > data[j]){
          int temp = data[j];
          data[j] = data[j-1];
          data[j-1] = temp;
        }
      }else{
        if (data[j-1] < data[j]){
          int temp = data[j];
          data[j] = data[j-1];
          data[j-1] = temp;
        }
      }
      j--;
    }
  }
}
```

main.cpp

```
// File  : main.cpp
// Name  : Fadhil Imam Kurnia - 13515146
// Main file for sorting algorithm

#include <iostream>
#include <ctime>
#include "TabelSorter.h"
using namespace std;

int main(){
  long size;
  cout << "Please input number of data to be sorted : ";
  cin >> size;
  TabelSorter reference(size);
  TabelSorter tabel1 = reference;
  TabelSorter tabel2 = reference;
  TabelSorter tabel3 = reference;
  TabelSorter tabel4 = reference;
  clock_t t_start;
  clock_t t_end;
  double t_duration;


  // BEGIN testing mergeSort ------------------------------------------
  t_start = clock();
  tabel1.mergeSort(true);
  t_end = clock();

  t_duration = (double) ((double)t_end-(double)t_start) / CLOCKS_PER_SEC * 1000.0;
  cout  << endl
        << "Merge Sort duration: "
        << t_duration
        << " ms"
        << endl;
  // END testing mergeSort --------------------------------------------

  // BEGIN testing insertionSort --------------------------------------
  t_start = clock();
  tabel3.insertionSort(true);
  t_end = clock();

  t_duration = (double) (t_end-t_start) / CLOCKS_PER_SEC * 1000.0;
  cout  << endl
  << "Insertion Sort duration: "
  << t_duration
  << " ms"
  << endl;
  // END testing insertionSort ----------------------------------------

  // BEGIN testing quickSort ------------------------------------------
  t_start = clock();
  tabel2.quickSort(true);
  t_end = clock();

  t_duration = (double) (t_end-t_start) / CLOCKS_PER_SEC * 1000.0;
  cout  << endl
        << "Quick Sort duration: "
        << t_duration
        << " ms"
        << endl;
  // END testing quickSort --------------------------------------------

  // BEGIN testing selectionSort --------------------------------------
  t_start = clock();
  tabel4.selectionSort(true);
  t_end = clock();

  t_duration = (double) (t_end-t_start) / CLOCKS_PER_SEC * 1000.0;
  cout  << endl
        << "Selection Sort duration: "
        << t_duration
        << " ms"
        << endl;
  // END testing selectionSort ----------------------------------------

  return 0;
}
```

Pada *sorce code* tersebut kelas TabelSorter digunakan di main.cpp untuk menghasilkan data *random*. Data *random* yang sama akan digunakan untuk membandingkan 4 algoritma pengurutan yang ada dalam kelas TabelSorter. Program tersebut dapat dilihat pada https://github.com/fadhilimamk/Sorter .

Contoh Input Output

Jika dijalankan dan menambahkan pemanggilan *method* print() untuk mengecek data sebelum dan sesudah proses pengurutan maka dapat dihasilkan keluaran sebagai berikut.

Kasus Uji 1 menggunakan Merge Sort
`D:\Proyek\Sorter>main.exe`
`Please input number of data to be sorted : 20`
`41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91 95 42 27 36`
`0 5 24 27 27 34 36 41 42 45 58 61 62 64 67 69 78 81 91 95`
`Merge Sort duration: 0 ms`
	Kasus Uji 2 menggunakan Merge Sort
	`D:\Proyek\Sorter>main.exe`
	`Please input number of data to be sorted : 15`
	`41 67 34 0 69 24 78 58 62 64 5 45 81 27 61`
	`81 78 69 67 64 62 61 58 45 41 34 27 24 5 0`
	`Merge Sort duration: 0 ms`

Kasus Uji 3 menggunakan Insertion Sort
`D:\Proyek\Sorter>main.exe`
`Please input number of data to be sorted : 15`
`41 67 34 0 69 24 78 58 62 64 5 45 81 27 61`
`0 5 24 27 34 41 45 58 61 62 64 67 69 78 81`
`Insertion Sort duration: 0 ms`
	Kasus Uji 4 menggunakan Insertion Sort
	`D:\Proyek\Sorter>main.exe`
	`Please input number of data to be sorted : 9`
	`41 67 34 0 69 24 78 58 62`
	`78 69 67 62 58 41 34 24 0`
	`Insertion Sort duration: 0 ms`

Kasus Uji 5 menggunakan Quick Sort
`D:\Proyek\Sorter>main.exe`
`Please input number of data to be sorted : 13`
`41 67 34 0 69 24 78 58 62 64 5 45 81`
`0 5 24 34 41 45 58 62 64 67 69 78 81`
`Quick Sort duration: 0 ms`
	Kasus Uji 6 menggunakan Quick Sort
	`D:\Proyek\Sorter>main.exe`
	`Please input number of data to be sorted : 11`
	`41 67 34 0 69 24 78 58 62 64 5`
	`78 69 67 64 62 58 41 34 24 5 0`
	`Quick Sort duration: 0 ms`

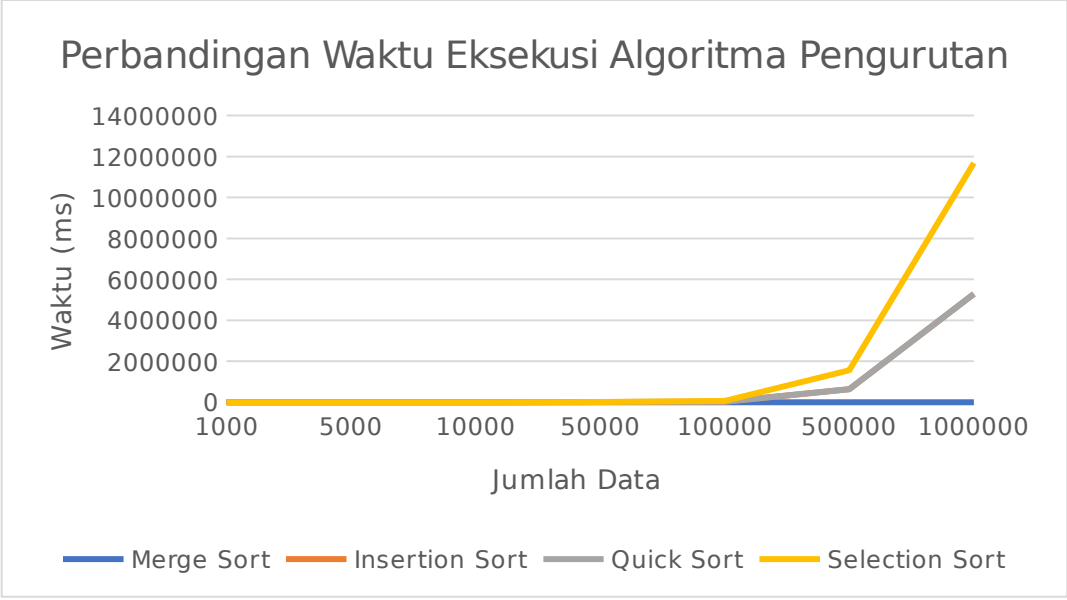
Kasus Uji 7 menggunakan Insertion Sort
`D:\Proyek\Sorter>main.exe`
`Please input number of data to be sorted : 7`
`41 67 34 0 69 24 78`
`0 24 34 41 67 69 78`
`Selection Sort duration: 0 ms`
	Kasus Uji 8 menggunakan Insertion Sort
	`D:\Proyek\Sorter>main.exe`
	`Please input number of data to be sorted : 11`
	`41 67 34 0 69 24 78 58 62 64 5`
	`78 69 67 64 62 58 41 34 24 5 0`
	`Selection Sort duration: 0 ms`

C. Perbandingan Kecepatan Algoritma

Menggunakan program tersebut kita dapat membandingkan beberapa algoritma pengurutan secara sekaligus. Data *random* yang digunakan untuk masing-masing algoritma harus sama agar hasil perbandingannya lebih terukur. Untuk membandingkan beberapa algoritma pengurutan digunakan data *random* yang jumlahnya sebanyak 1000, 5000, 10000, 50000, 100000, 500000, hingga 1000000. Hasil perbandingan yang telah dilakukan dapat dilihat pada Tabel 1 dan Gambar x.

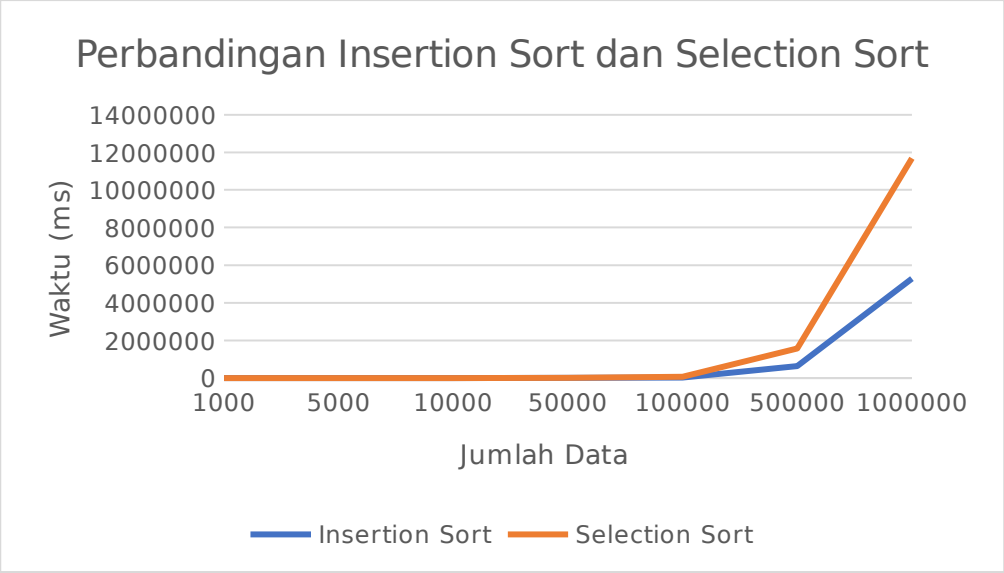
Banyak Data	Waktu Eksekusi (ms)			
	Merge Sort	Insertion Sort	Quick Sort	Selection Sort
1000	1	2	0	3
5000	3	66	1	129
10000	5	236	2	415
50000	23	6137	12	9754
100000	50	23481	24	36616
500000	247	640115	183	925541
1000000	485	5290145	265	6393110

Tabel 1 Perbandingan masing-masing algoritma pengurutan dengan data random

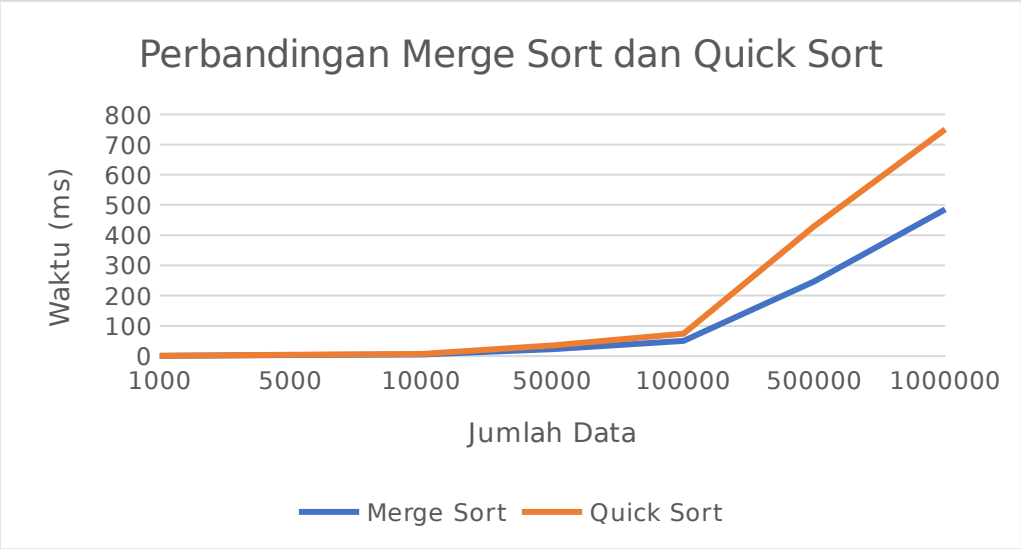


Gambar 2 Perbandingan algoritma merge sort, insertion sort, quick sort, dan selection sort

Dari data hasil uji tersebut kita juga bisa membandingkan waktu eksekusi yang dibutuhkan antara algoritma pengurutan insertion sort dan selection sort. Perbandingan kedua algoritma tersebut dapat dilihat pada Gambar x. Pada perbandingan tersebut kita dapat melihat lebih detail perbedaan waktu eksekusi antara kedua algoritma tersebut. Selain perbandingan antara insertion sort dan selection sort, perbandingan antara merge sort dan quick sort juga dipaparkan dalam Gambar x.



Gambar 3 Perbandingan algoritma merge sort, insertion sort, quick sort, dan selection sort



Gambar 4 Perbandingan algoritma merge sort, insertion sort, quick sort, dan selection sort

D. Analisis Algoritma

Berdasarkan perbandingan hasil uji pada bagian C dapat dilihat bahwa algoritma quick sort dan merge sort memiliki waktu eksekusi yang relatif singkat untuk data yang besar. Namun jika kita membandingkan keempat algoritma tersebut pada data yang tidak terlalu banyak, misalkan 1000, kita tidak dapat melihat perbedaan waktu yang sangat mencolok.

Jika kita perhatikan lebih detail antara algoritma pengurutan insertion sort dan selection sort keduanya memiliki kompleksitas yang sama yaitu $O(n^2)$. Namun jika kita lihat pada Tabel 1, algoritma insertion sort memiliki waktu eksekusi yang lebih cepat dibandingkan dengan selection sort. Hal tersebut karena pada algoritma

insertion sort kita hanya perlu menyisipkan sebuah angka kedalam bagian tabel yang sudah terurut. Sehingga proses tersebut tidak setiap saat harus mengecek semua bilangan pada bagian tabel yang sudah terurut. Pengecekan semua bilangan pada bagan tabel yang suadh terurut dapat terjadi jika bilangan yang akan disisipkan adalah bilangan ekstrem (maksimum/minimum). Berbeda dengan selection sort yang harus mencari bilangan ekstrem (maksimum/minimum) pada bagian tabel yang belum terurut. Proses pencarian bilangan ekstrem tersebut akan selalu mengecek semua bilangan pada bagian tabel yang belum terurut. Maka kompleksitas dari selection sort selalu $\frac{n(n-1)}{2}$, sedangkan kompleksitas dari insertion sort juga $\frac{n(n-1)}{2}$, namun itu terjadi pada kasus terburuk saja. Oleh karena itu pengurutan menggunakan algoritma selection sort membutuhkan waktu lebih jika dibandingkan dengan insertion sort.

Merge sort dan quick sort merupakan algoritma pengiritan yang mangkus jika kita bandingkan dengan insertion sort dan selection sort. Hal itu ditunjukan dengan perbedaan waktu eksekusi yang besar pada Tabel 1. Jika kita bandingkan lebih detail antara merge sort dan quick sort, berdasarkan hasil uji yang sudah dilakukan, waktu eksekusi merge sort relatif lebih lama dibandingkan dengan merge sort. Ada beberapa hal yang dapat mengakibatkan ini, salah satunya adalah mengenai alokasi memori tambahan yang dibutuhkan pada merge sort. Secara umum untuk menggabungkan 2 tabel yang sudah terurut pada merge sort diperlukan tabel sementara yang digunakan untuk menampung hasil penggabungan. Proses pengalokasian tabel sementara tentu saja memerlukan waktu tambahan. Oleh karena itu proses quick sort dapat lebih cepat dibandingkan dengan merge sort.

E. Ceklist Pengerjaan Tugas

Poin	Ya	Tidak
1. Program berhasil dikompilasi	√	
2. Program berhasil *running*	√	
3. Program dapat membaca koleksi data random dan menuliskan koleksi data terurut.	√	
4. Laporan berisi hasil perbandingan kecepatan eksekusi dan analisisnya	√	

Daftar Pustaka

[1] Sitorus lamhot. 2015. *Algoritma dan Pemrograman*. Penerbit Andi: Yogyakarta.
[2] Rinaldi Munir, Diktat Kuliah IF2251 Strategi Algoritmik, STEI, 2006.