PENGURUTAN DATA DENGAN MENGGUNAKAN ALGORITMA *DIVIDE AND CONQUER*

LAPORAN

Disusun untuk memenuhi persyaratan nilai mata kuliah

IF2211 - Strategi Algoritma

oleh

Felix Limanta / 13515065

PROGRAM STUDI TEKNIK INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

2017

1. *Source Code*

Program diimplementasi dengan bahasa C++ dan memanfaatkan *library* standar (STL) dari C++. Implementasi algoritma pengurutan diletakkan di berkas .h, bukan berkas .cpp, karena algortima pengurutan diimplementasi sebagai fungsi generik.

Karena keterbatasan perangkat keras, *insertion sort* dan *selection sort* diimplementasi secara iteratif. Implementasi *insertion sort* dan *selection sort* secara rekursif akan menyebabkan pemanggilan rekursi sejumlah data yang diurutkan, sehingga untuk ukuran data yang sangat besar, *stack* komputer di mana program dijalankan tidak akan kuat menampung seluruh pemanggilan rekursi.

1.1 main.cpp

```
/**     NIM:            13515065
        Nama:           Felix Limanta
        Tanggal:        24 Februari 2017
        File:           main.cpp
        Topik;          Divide and Conquer
        Deskripsi:      Driver algoritma sorting
  */

#include "merge_sort.h"
#include "insertion_sort.h"
#include "quick_sort.h"
#include "selection_sort.h"
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <iostream>
#include <random>
#include <string>

using namespace std;

#define GEN_RANDOM          0
#define MERGE_SORT          1
#define INSERTION_SORT      2
#define QUICK_SORT          3
#define SELECTION_SORT      4

/**     Function definition
  */
void generate_random (size_t size);
int* load_unsorted (size_t size);
bool is_sorted (int data[], size_t size);
void save_sorted (int data[], size_t size, char sort_type, int64_t execution_time);

int main(int argc, char* argv[]) {
        int action;
        size_t size;

        cout << "Action:    0. Generate random data\n" <<
                        "           1. Merge sort\n" <<
```

```
                        "           2. Insertion sort\n" <<
                        "           3. Quick sort\n" <<
                        "           4. Selection sort\n" <<
                        "Action:    ";
        if (argc < 2) {
                cin >> action;
        } else {
                action = atoi(argv[1]);
                cout << action << '\n';
        }

        cout << "Data size: ";
        if (argc < 3) {
                cin >> size;
        } else {
                size = atoi(argv[2]);
                cout << size << '\n';
        }

        if (action == GEN_RANDOM) {
                // Random data generation
                generate_random(size);
                cout << "Random data generated" << '\n';
        } else {
                // Sorting
                int* data = load_unsorted(size);
                int64_t start_time = clock();
                switch (action) {
                        case MERGE_SORT:
                                merge_sort(data, size); break;
                        case INSERTION_SORT:
                                insertion_sort(data,size); break;
                        case QUICK_SORT:
                                quick_sort(data,size); break;
                        case SELECTION_SORT:
                                selection_sort(data,size); break;
                }

                // Measure execution time
                int64_t end_time = clock();
                int64_t exec_time = (end_time - start_time) * 1000 / CLOCKS_PER_SEC;

                // Check integrity and output result
                if (is_sorted(data,size)) {
                        save_sorted(data,size,action,exec_time);
                        cout << "Data sorted successfully\n" <<
                                "Execution time: " << exec_time;
                } else {
                        cout << "Sorting failed" << '\n';
                }
        }

        return 0;
}

/**     @brief Random Number Generator
        Generates a text file containing randomly generated unsigned integer
        File is saved to path "../output/{size}_unsorted.txt"
        First line of file contains file name as header
        Data starts at third line
        @param size    Number of data in file
  */
void generate_random (size_t size) {
        ofstream outf;
        string fname = "../output/" + to_string(size) + "_unsorted.txt";
        outf.open(fname.c_str(), ios::out);
```

```
	if (outf.is_open()) {
		unsigned int seed = time(NULL);
		subtract_with_carry_engine<unsigned,24,10,24> generator (seed);
		outf << fname << "\n\n";
		for (size_t i = 0; i < size; ++i)
			outf << generator() << '\n';
		outf.close();
	}
}

/**	@brief Load unsorted file
	Loads file containing unsorted data as generated by function generate_random
	Loads "../output/{size}_unsorted.txt"
	@param size	Number of data to be loaded
	@return Pointer to array containing data
  */
int* load_unsorted (size_t size) {
	ifstream inf;
	string fname = "../output/" + to_string(size) + "_unsorted.txt";
	inf.open(fname.c_str(), ios::in);

	if (inf.is_open()) {
		int* data = new int[size];
		string instr;
		getline(inf, instr);
		getline(inf, instr);
		for (size_t i = 0; i < size; ++i) {
			getline(inf, instr);
			data[i] = stoi(instr);
		}
		inf.close();
		return data;
	} else
		return NULL;
}

/**	@brief Check if data is sorted
	@param data	Array to be checked
	@param size	Size of array
	@return True if data is sorted ascending
  */
bool is_sorted (int data[], size_t size) {
	bool sorted = true;
	for (size_t i = 1; sorted && i < size; ++i)
		sorted = data[i] >= data[i-1];
	return sorted;
}

/**	@brief Save sorted data to text file
	File is saved to path "../output/{size}_{algorithm}.txt"
	@param data		Array of sorted data to be saved
	@param size		Size of array
	@param sort_type	Type of sorting
	@param exec_time	Execution time of sorting algorithm
  */
void save_sorted (int data[], size_t size, char sort_type, int64_t exec_time) {
	ofstream outf;
	string fname = "../output/" + to_string(size);
	switch (sort_type) {
		case MERGE_SORT:
			fname += "_merge_sort.txt"; break;
		case SELECTION_SORT:
			fname += "_selection_sort.txt"; break;
		case QUICK_SORT:
			fname += "_quick_sort.txt"; break;
```

```
                case INSERTION_SORT:
                        fname += "_insertion_sort.txt"; break;
        }
        outf.open(fname,ios::out);

        if (outf.is_open()) {
                outf << fname << '\n' << exec_time << "\n\n";
                for (size_t i = 0; i < size; ++i)
                        outf << data[i] << '\n';
                outf.close();
        }
}
```

1.2 merge_sort.h

```
/**     NIM:            13515065
        Nama:           Felix Limanta
        Tanggal:        24 Februari 2017
        File:           merge_sort.h
        Topik;          Divide and Conquer
        Deskripsi:      Definisi dan realisasi algoritma merge sort
  */

#ifndef __MERGE_SORT__
#define __MERGE_SORT__

#include <array>
#include <cstdlib>

/**     @brief Merging process
        Consolidates data from two subarrays to one sorted array
        @param data     array of data to be sorted
        @param low              lower bound index of first subarray
        @param mid              higher bound index of first subarray
        @param high     higher bound index of second subarray
        Lower bound index of second subarray assumed to be mid + 1
  */
template <class T>
void merge (T data[], size_t low, size_t mid, size_t high) {
        size_t l1 = low, l2 = mid + 1, i, j;
        T* temp = new T[high - low + 1];

        // Consolidate both arrays
        for (i = 0; l1 <= mid && l2 <= high; ++i) {
                if (data[l1] <= data[l2])
                        temp[i] = data[l1++];
                else
                        temp[i] = data[l2++];
        }

        // Move remaining elmt of 1st array
        while (l1 <= mid)
                temp[i++] = data[l1++];

        // Move remaining elmt of 2nd array
        while (l2 <= high)
                temp[i++] = data[l2++];

        // Copy data fron temp array to actual array
        for (i = low, j = 0; i <= high; ++i, ++j)
                data[i] = temp[j];
```

```
        delete [] temp;
}

/**     @brief Merge Sort
        Sort array of data with merge sort
        @param data    data to be sorted
        @param low             lower bound index of data
        @param high    higher bound index of data
  */
template <class T>
void merge_sort (T data[], size_t low, size_t high) {
        if (low >= high) {     // Basis, 1 elmt
                // Do nothing, 1 elmt already sorted
        } else {       // Recurrence
                // Divide
                int mid = (low + high) / 2;

                // Conquer
                merge_sort(data, low , mid);
                merge_sort(data, mid + 1, high);

                // Combine
                merge(data, low, mid, high);
        }
}

/**     @brief Merge Sort
        Sort array of data with merge sort
        Wrapper for merge_sort(data,low,high)
        @param data    data to be sorted
        @param size    size of data to be sorted
  */
template <class T>
void merge_sort (T data[], size_t size) {
        merge_sort(data, 0, size-1);
}

#endif
```

1.3 insertion_sort.h

```
/**     NIM:           13515065
        Nama:          Felix Limanta
        Tanggal:       24 Februari 2017
        File:          insertion_sort.h
        Topik;         Divide and Conquer
        Deskripsi:     Definisi dan realisasi algoritma insertion sort
  */

#ifndef __INSERTION_SORT__
#define __INSERTION_SORT__

#include <cstdlib>

/**     @brief Insertion Sort
        Sort array of data with insertion sort
        @param data    data to be sorted
        @param size    size of data to be sorted
  */
template <class T>
void insertion_sort (T data[], size_t size) {
        if (size > 1) {
                for (size_t i = 1; i < size; ++i) {
```

```
                        // Divide: Store inserted data
                        int temp = data[i];

                        // Conquer: search for place to insert while shifting the rest
                        int j;
                        for (j = i - 1; j > 0 && temp < data[j]; --j)
                                data[j+1] = data[j];

                        // Combine: insert data
                        if (temp >= data[j])       // Insert
                                data[j+1] = temp;
                        else { // Insert as 1st elmt
                                data[j+1] = data[j];
                                data[j] = temp;
                        }
                }
        }
}

#endif
```

1.4 quick_sort.h

```
/**     NIM:            13515065
        Nama:           Felix Limanta
        Tanggal:        24 Februari 2017
        File:           quick_sort.h
        Topik;          Divide and Conquer
        Deskripsi:      Definisi dan realisasi algoritma quick sort
  */

#ifndef __QUICK_SORT__
#define __QUICK_SORT__

#include <cstdlib>

/**     @brief Partitioning process
        Partitions array such that every value on the left is less than every value on the
right
        @param data     array of data to be sorted
        @param low              lower bound index of array
        @param high     higher bound index of array
        @param mid              index of partitioning point
        Every data on the left of partitioning point is less than data on the right
  */
template <class T>
void partition (T data[], size_t low, size_t high, size_t& mid) {
        // Set pivot as middle element
        int pivot = data[(low + high) / 2];

        int i = low, j = high;
        do {
                // Seek element left of pivot greater than pivot
                while (data[i] < pivot) i++;

                // Seek element right of pivot less than pivot
                while (data[j] > pivot) j--;

                // Swap if iterators haven't passed by each other
                if (i <= j) {
                        int temp = data[i];
                        data[i++] = data[j];
                        data[j--] = temp;
```

```
                }
        } while (i <= j);     // Stop if iterators have passed by each other
        mid = j;
}

/**     @brief Quick Sort
        Sort array of data with quick sort
        @param data    data to be sorted
        @param low             lower bound index of data
        @param high    higher bound index of data
  */
template <class T>
void quick_sort (T data[], size_t low, size_t high) {
        if (low >= high) {    // Basis, 1 elmt
                // Do nothing, 1 elmt already sorted
        } else {       // Recurrence
                // Divide
                int mid;
                partition(data, low, high, mid);

                // Conquer
                quick_sort(data, low, mid);
                quick_sort(data, mid + 1, high);

                // Combine: do nothing
        }
}

/**     @brief Quick Sort
        Sort array of data with quick sort
        Wrapper for quick_sort(data,low,high)
        @param data    data to be sorted
        @param size    size of data to be sorted
  */
template <class T>
void quick_sort (T data[], size_t size) {
        merge_sort(data, 0, size-1);
}

#endif
```

1.5 selection_sort.h

```
/**     NIM:           13515065
        Nama:          Felix Limanta
        Tanggal:       24 Februari 2017
        File:          selection_sort.h
        Topik;         Divide and Conquer
        Deskripsi:     Definisi dan realisasi algoritma selection sort
  */

#ifndef __SELECTION_SORT__
#define __SELECTION_SORT__

#include <cstdlib>

/**     @brief Find index of minimum value
        Uses divide and conquer mixed with regular traversal to find minimum value index
        @param data    data to be searched
        @param low     lower bound index of data to be searched
        @param high    higher bound index of data to be searched
        @param delta   minimum size of division before resorting to traversal
```

```
        From experiment, best value of delta is found to be around 5000
        @return Index of minimum value in array
  */
template <class T>
size_t findmin (T data[], size_t low, size_t high, size_t delta) {
        if (high - low <= delta) {    // Basis, traverse to find minimum
                size_t min = low;
                for (size_t i = low + 1; i <= high; ++i)
                        if (data[i] < data[min])
                                min = i;
                return min;
        } else {        // Recurrence
                // Divide
                size_t mid = (low + high) / 2;

                // Conquer
                size_t min1 = findmin(data,low,mid,delta);
                size_t min2 = findmin(data,mid+1,high,delta);

                // Combine
                return (data[min1] <= data[min2]) ? min1 : min2;
        }
}

/**     @brief Selection Sort
        Sort array of data with selection sort
        @param data    data to be sorted
        @param size    size of data to be sorted
  */
template <class T>
void selection_sort (T data[], size_t size) {
        if (size > 1) {
                // Divide: isolate element by element
                for (size_t i = 0; i < size - 1; ++i) {
                        // Conquer
                        // Find min value
                        size_t i_min = findmin(data,i,size-1,5000);

                        // Switch values to sort
                        int temp = data[i];
                        data[i] = data[i_min];
                        data[i_min] = temp;
                }
                // Combine: do nothing
        }
}

#endif
```

2. Contoh Masukan dan Keluaran

Untuk keringkasan contoh, ukuran data masukan dan keluaran yang diberikan di sini dibatas hanya 20.

2.1 Masukan

Format masukan:

- Baris 1: nama berkas, berisi ukuran data

- Baris 2 kosong
- Baris 3 ke bawah berisi data yang belum terurut

```
../output/20_unsorted.txt

11007482
406519
9502511
10141960
13807186
7730448
5039178
7673118
10378431
13688111
12730918
5148462
1431151
2424464
5275143
5000750
13412747
8699912
7798121
8971844
```

2.2 Keluaran

Format keluaran:

- Baris 1: nama berkas, berisi ukuran data dan algoritma yang digunakan
- Baris 2: waktu eksekusi, dalam ms
- Baris 3 kosong
- Baris 4 ke bawah berisi data yang sudah terurut

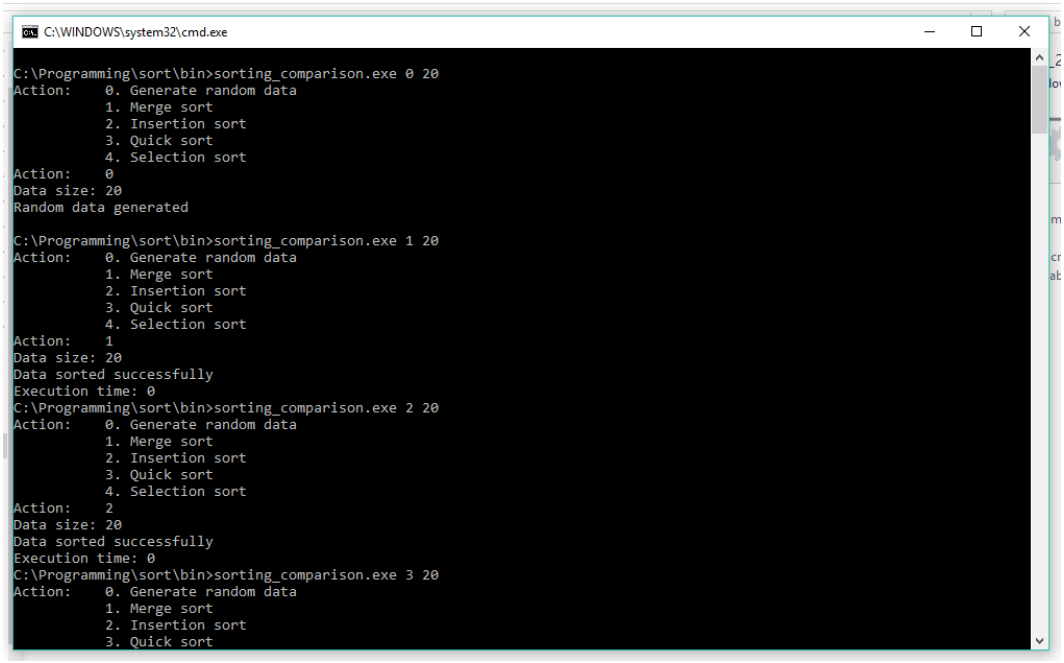
Tabel 1. Contoh berkas keluaran berdasarkan algoritma pengurutan yang dipakai

Merge Sort	***Insertion Sort***
../output/20_merge_sort.txt 0 406519 1431151 2424464 5000750 5039178 5148462 5275143 7673118 7730448 7798121 8699912 8971844	../output/20_insertion_sort.txt 0 406519 1431151 2424464 5000750 5039178 5148462 5275143 7673118 7730448 7798121 8699912 8971844

9502511 10141960 10378431 11007482 12730918 13412747 13688111 13807186	9502511 10141960 10378431 11007482 12730918 13412747 13688111 13807186

Quick Sort	***Selection Sort***
../output/20_quick_sort.txt 0 406519 1431151 2424464 5000750 5039178 5148462 5275143 7673118 7730448 7798121 8699912 8971844 9502511 10141960 10378431 11007482 12730918 13412747 13688111 13807186	../output/20_selection_sort.txt 0 406519 1431151 2424464 5000750 5039178 5148462 5275143 7673118 7730448 7798121 8699912 8971844 9502511 10141960 10378431 11007482 12730918 13412747 13688111 13807186

2.3 Tampilan di Layar



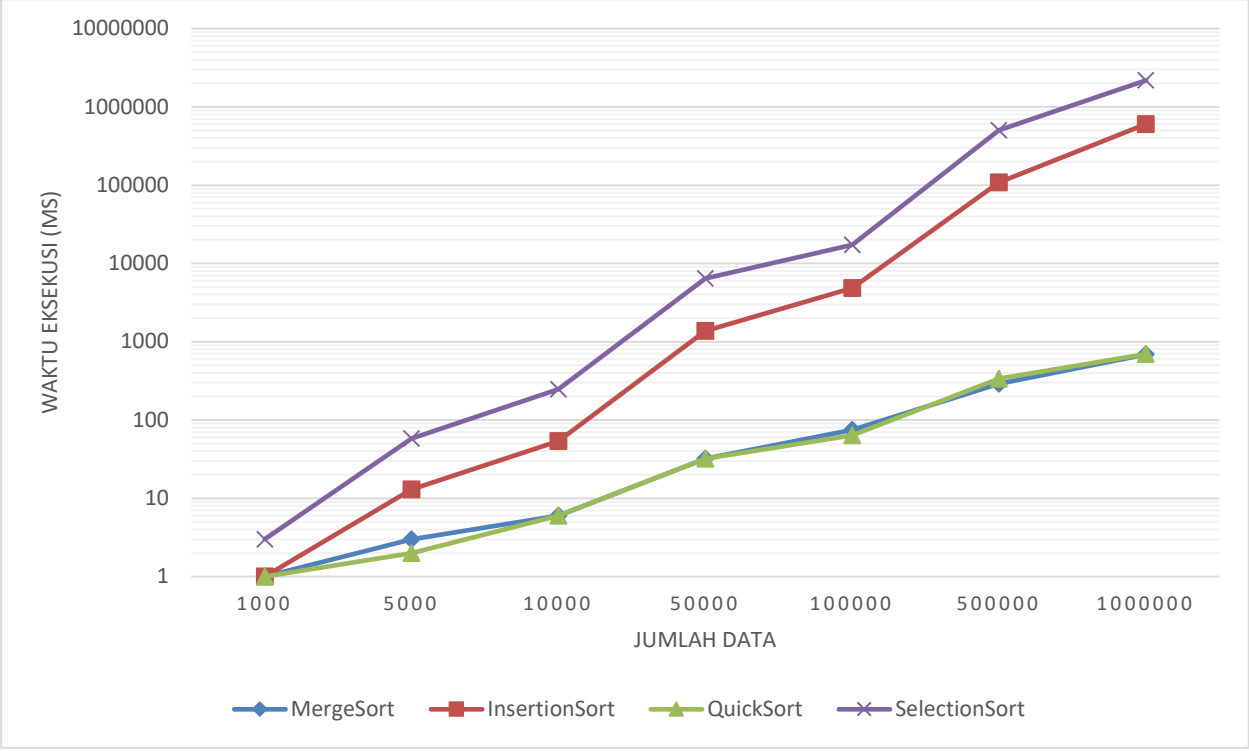
Gambar 1. Contoh tampilan di layar

3. Analisis Eksekusi Algoritma

Berikut adalah waktu eksekusi setiap algoritma pengurutan untuk data bilangan bulat acak dengan jumlah data 1000—1000000 beserta grafik untuk membandingkan waktu eksekusi algoritma-algoritma tersebut. Grafik ditampilkan dalam skala logaritmik untuk mempermudah pembacaan.

Tabel 2. Waktu eksekusi algoritma pengurutan berdasarkan jumlah data (dalam ms)

		Jumlah data						
		1000	5000	10000	50000	100000	500000	1000000
Algoritma	MergeSort	1	3	6	32	75	291	687
	InsertionSort	1	13	54	1375	4828	108801	602179
	QuickSort	1	2	6	32	64	338	695
	SelectionSort	3	58	247	6431	17345	501885	2181746



Bagan 1. Grafik perbandingan waktu eksekusi algoritma pengurutan. Kedua sumbu ditampilkan dalam skala logaritmik untuk mempermudah presentasi

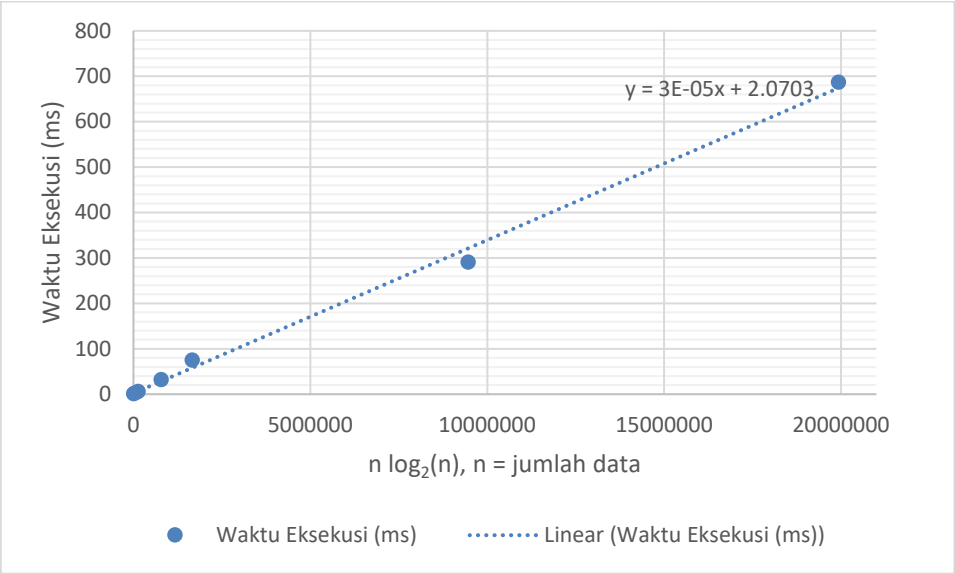
Sebagai referensi, berikut adalah kompleksitas waktu dan kebutuhan ruang masing-masing algoritma pengurutan. [1]

Tabel 3. Perbandingan kompleksitas waktu dan kebutuhan ruang masing-masing algoritma pengurutan

Algoritma	Waktu			Ruang
	Terbaik	Terburuk	Rata-rata	
Merge Sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	tergantung
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	konstan
Quick Sort	$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$	konstan
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	konstan

3.1 Merge Sort

Merge sort tidak memiliki kasus terbaik maupun terburuk karena tanpa peduli keadaan awal data yang akan diurutkan, setiap data akan dibagi dan dipilah sampai terurut.



Bagan 2. Waktu eksekusi *merge sort* beserta *trendline* regresi lanjar

Dari grafik di atas, dapat dilihat bahwa waktu eksekusi *merge sort* bertumbuh secara linear dengan n log n dari ukuran data.

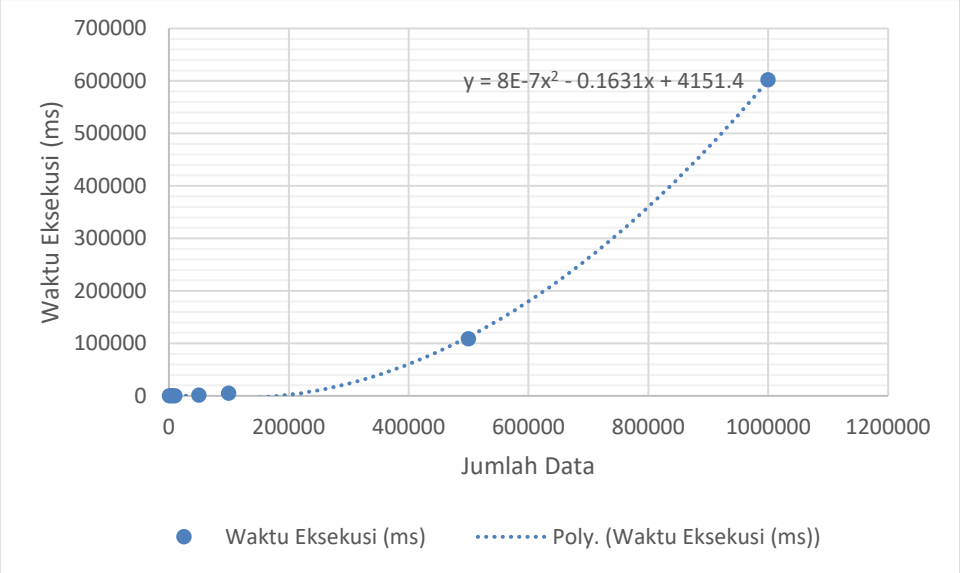
Tabel 4. Waktu eksekusi *merge sort* beserta waktu proyeksi berdasarkan regresi

Jumlah Data	n log₂(n)	Waktu Eksekusi (ms)	Proyeksi
1000	9965.784285	1	2.4063621
5000	61438.5619	3	4.1420485
10000	132877.1238	6	6.5509906
50000	780482.0237	32	28.388534
100000	1660964.047	75	58.078804
500000	9465784.285	291	321.26103
1000000	19931568.57	687	674.17223

Meskipun *merge sort* memiliki kompleksitas waktu yang kecil serta waktu eksekusi yang cepat, *merge sort* memerlukan larik kerja tambahan dalam proses penggabungan berukuran hingga jumlah data yang diurutkan, sehingga *merge sort* tidak terlalu efisien dalam penggunaan ruang. [2]

3.2 Insertion Sort

Pada *insertion sort*, untuk kasus terburuk (setiap elemen terurut terbalik dengan pengurutan yang dikehendaki), setiap elemen digeser sampai ke awal larik. Untuk kasus terbaik, jumlah operasi penggeseran elemen adalah 0. Untuk kasus rata-rata, setiap elemen digeser sebanyak $\frac{n}{2}$ kali. [3] [2]



Bagan 3. Waktu eksekusi *insertion sort* beserta *trendline* regresi kuadratik

Dari grafik di atas, dapat dilihat bahwa waktu eksekusi *insertion sort* bertumbuh kuadratik dengan ukuran data.

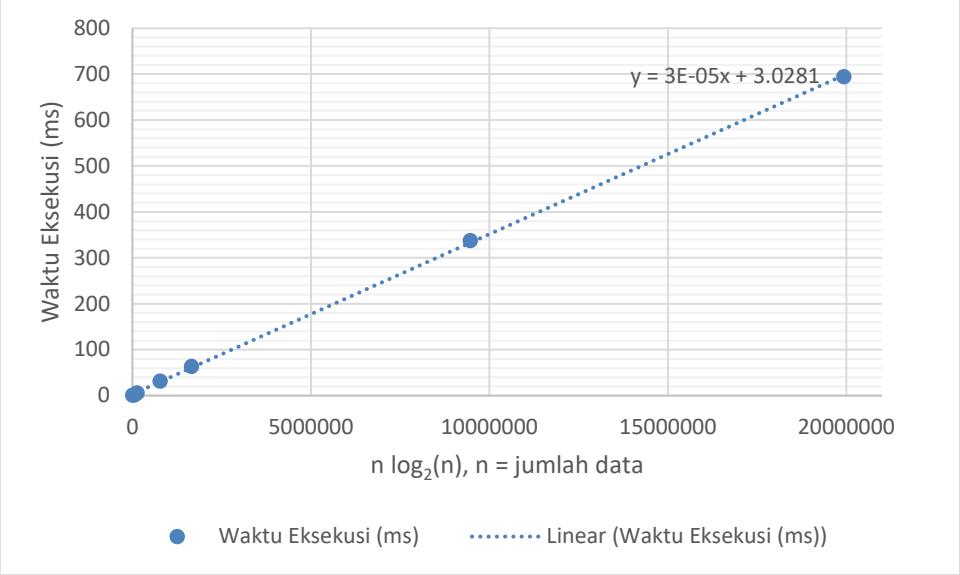
Tabel 5. Waktu eksekusi *insertion sort* beserta waktu proyeksi berdasarkan regresi

Jumlah Data	Waktu Eksekusi (ms)	Proyeksi
1000	1	3988.988
5000	13	3354.687
10000	54	2596.025
50000	1375	-2104.73

100000	4828	-4559.313
500000	108801	112658.5
1000000	602179	601316.9

3.3 Quick Sort

Pada *quick sort*, kasus terbaik terjadi ketika *pivot* yang dipilih adalah elemen median, sehingga kedua upalarik berukuran relatif sama setiap kali pempartisian. Penentuan median sebuah larik tak terurut merupakan persoalan tersendiri. Kasus terburuk terjadi ketika *pivot* yang dipilih merupakan nilai maksimum atau minimum larik, sehingga salah satu ukuran partisi adalah 1, sedangkan ukuran partisi lain adalah ukuran partisi sebelumnya dikurang 1.



Bagan 4. Waktu eksekusi *quick sort* beserta *trendline* regresi lanjar

Dari grafik di atas, dapat dilihat bahwa waktu eksekusi *quick sort* bertumbuh secara linear dengan n log n dari ukuran data.

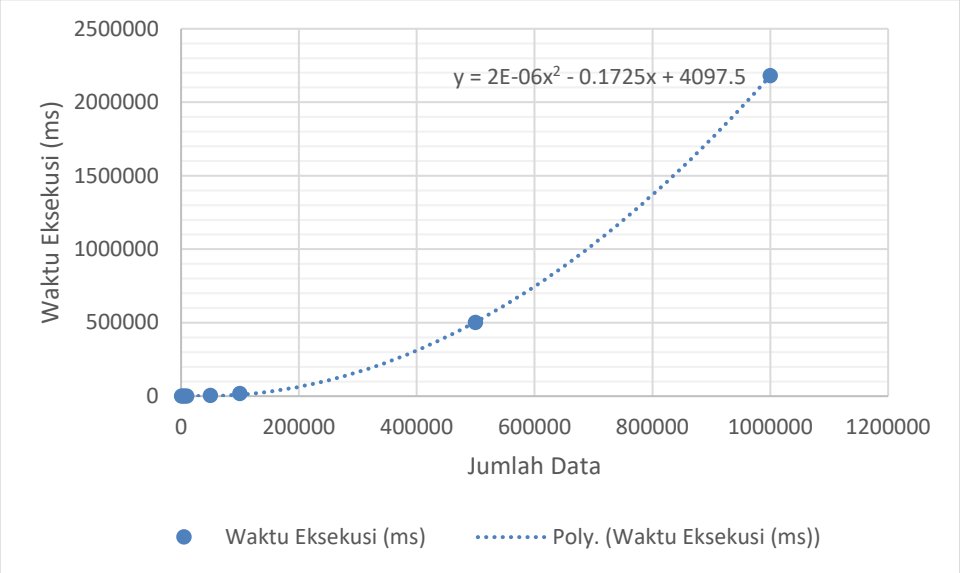
Tabel 6. Waktu eksekusi *quick sort* beserta waktu proyeksi berdasarkan regresi

Jumlah Data	n $\log_2(n)$	Waktu Eksekusi (ms)	Proyeksi
1000	9965.78428	1	3.375408
5000	61438.5619	2	5.169398
10000	132877.124	6	7.65926
50000	780482.024	32	30.23036
100000	1660964.05	64	60.91796
500000	9465784.28	338	332.9408
1000000	19931568.6	695	697.7068

Walaupun *quick sort* secara rata-rata dan menurut hasil yang didapat memiliki performa yang hampir sama dengan *merge sort*, *quick sort* tidak dapat menjamin kompleksitas $O(n \log_2 n)$ pada kasus terburuk, sedangkan kompleksitas *merge sort* stabil. Meskipun demikian, dibandingkan *merge sort*, *quick sort* tidak memerlukan larik kerja sementara, sehingga secara umum *quick sort* lebih disukai daripada *merge sort*.

3.4 Selection Sort

Kompleksitas waktu *selection sort* tidak memiliki kasus terbaik atau terburuk karena bahkan pada larik yang sudah terurut sempurna, algoritma tetap memindai seluruh larik untuk mencari nilai yang hendak ditukar. Ini dilakukan pada setiap elemen pada larik. [4]



Bagan 5. Waktu eksekusi *selection sort* beserta *trendline* regresi kuadratik

Dari grafik di atas, dapat dilihat bahwa waktu eksekusi *insertion sort* bertumbuh kuadratik dengan ukuran data.

Tabel 7. Waktu eksekusi *selection sort* beserta waktu proyeksi berdasarkan regresi

Jumlah Data	Waktu Eksekusi (ms)	Proyeksi
1000	3	3927.30763
5000	58	3293.7041
10000	247	2607.42267

50000	6431	1346.09094
100000	17345	10341.7254
500000	501885	505198.767
1000000	2181746	2180999.98

Meskipun sama-sama memiliki kompleksitas asimtotik $O(n^2)$, *selection sort* tidak lebih baik dari *insertion sort* karena pemindaian seluruh larik yang wajib. Ini dapat menjadi alasan mengapa untuk ukuran data yang sama, pengurutan dengan *selection sort* memakan waktu hingga 5 kalinya pengurutan dengan *insertion sort*.

3.5 Kesimpulan

Dari keempat algoritma pengurutan yang diuji, *merge sort* dan *quick sort* memiliki performa yang hampir sama. Meskipun demikian, *quick sort* lebih efisien secara memori karena tidak membutuhkan larik kerja tambahan dibandingkan *merge sort*.

Dibandingkan kedua algoritma tersebut, *insertion sort* dan *selection sort* kalah jauh dalam performa. Dari keduanya, *selection sort* lebih buruk dari *insertion sort* karena *insertion sort* tidak perlu menelusuri larik sampai habis, sedangkan *selection sort* perlu.

4. *Checklist* Penilaian

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	√	

Referensi

[1] A. Allain, “Sorting Algorithm Comparison,” Cprogramming.com, [Online]. Available: http://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html. [Diakses 27 Februari 2017].

[2] R. Munir, Diktat Kuliah IF2211 Strategi Algoritma, Bandung: Teknik Informatika ITB, 2009.

[3] R. bin Muhammad, “Insertion Sort,” [Online]. Available: http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/insertionSort.htm. [Diakses 26 Februari 2017].

[4] S. A. Alvarez, “Selection Sort Analysis,” 30 April 2012. [Online]. Available: http://www.cs.bc.edu/~alvarez/CS1/Notes/selectionSortAnalysis. [Diakses 26 Februari 2017].

[5] I. Liem, Draft Diktat Kuliah Dasar Pemrograman (Bagian Pemrograman Prosedural), Bandung: STEI ITB, 2007.