STRING (Satuan Tulisan Riset dan Inovasi Teknologi) p-ISSN: 2527 - 9661
Vol. 5 No. 2 Desember 2020 e-ISSN: 2549 - 2837

ANALISIS KOMPLEKSITAS RUANG DAN WAKTU TERHADAP LAJU PERTUMBUHAN ALGORITMA HEAP SORT, INSERTION SORT DAN MERGE DENGAN PEMROGRAMAN JAVA

Rizki Rizkyatul Basir
Program Studi Teknik Informatika, Universitas Indraprasta PGRI
rizkyatulbasir@gmail.com

Submitted March 26, 2020; Revised July 7, 2020; Accepted August 3, 2020

Abstrak

Peran algoritma dalam perangkat lunak atau pemrograman sangat penting, sehingga perlu untuk memahami konsep dasar dari algoritma. Kajian implementasi dan performa proses pengurutan menggunakan algoritma *heap sort*, *insertion sort* dan *merge sort*. Metode penelitian untuk tahap pertama, ketiga algoritma tersebut diimplementasikan untuk mengurutkan sejumlah angka yang dilakukan oleh pengguna, pada tahap kedua kode sumber untuk ketiga algoritma tersebut diubah untuk dapat mengurutkan angka yang dihasilkan secara acak dengan jumlah angka sebanyak permintaan dari pengguna. Untuk mengetahui seberapa baik performa dalam mengurutkan data, maka dalam tahap terakhir ketiga algoritma tersebut mengurutkan sejumlah angka acak dengan rentan jumlah yang sudah ditentukan dan kemudian hasilnya dibandingkan. Eksperimen yang sudah dilakukan dan berdasarkan analisis, *heap sort* merupakan salah satu metode pengurutan data yang tergolong mempunyai kecepatan tinggi, dimana kompleksitas dan kecepatan waktu pengurutan yang dibutuhkan untuk proses pengurutan menggunakan algoritma *insertion sort* dan *merge sort* yang menunjukan kurang konsisten tehadap kompleksitas ruang dan waktu. Berdasarkan hasil tabel dan data terhadap jumlah data, menunjukan bahwa algoritma *heap sort* memberikan waktu proses yang sangat konsisten dalam peningkatan lama waktu proses terhadap jumlah data.

Kata Kunci : Kompleksitas Algoritma, *Insertion Sort*, *Merge Sort*, *Heap Sort*, Pemrograman Java

Abstract

The role of the algorithms in software or programming is very important, so it is necessary to understand the basic concepts of the algorithm. Study the implementation and performance of the sorting process using heap sort, insertion sort and merge sort algorithms. The research method for the first stage, the three algorithms are implemented to sort the numbers made by the user, in the second stage the source code for the three algorithms is changed to be able to sort randomly generated numbers with as many numbers as the request from the user. To find out how well the performance in sorting the data, then in the last stage the three algorithms sort random numbers with a predetermined vulnerable number and then the results are compared. Experiments that have been carried out and based on the analysis, heap sort is one of the data sorting methods that are classified as having a high speed, where the complexity and speed of the sorting time needed for the sorting process uses insertion sort and merge sort algorithms which show less consistency in the complexity of time and space. Based on the results of tables and data on the amount of data, shows that the heap sort algorithm gives a very consistent processing time in increasing the processing time to the amount of data.

Key Words : *Algorithm Complexity, Insertion Sort, Merge Sort, Heap Sort, Java Programming*

1. PENDAHULUAN

Dalam kehidupan sehari-hari komputer sering digunakan untuk mengurutkan sekumpulan nilai. *Sorting* adalah proses pengurutan data yang sebelumnya disusun secara acak atau tidak teratur menjadi urut dan teratur menurut suatu aturan tertentu. [11]. Proses tersebut diimplementasikan dalam berbagai macam aplikasi dan pengurutan dapat berupa nilai, objek atau nama. Pada aplikasi yang berhubungan

dengan data yang dapat diurutkan, seperti Nomor Induk Mahasiswa, nilai, nomor ID sebuah barang inventaris dan lain sebagainya, untuk contoh penerapannya berupa rincian sesuai urutan tanggal dan jam pada perbankan, daftar hadir yang diurutkan berdasarkan nomor induk dan daftar pustaka yang diurutkan sesuai abjad pengarang ataupun buku disebuah perpustakaan. Fungsi-fungsi statistic seperti median dan pembuatan kuartil data (*quarter*), detil dan (*percentile*) mensyaratkan data untuk diurutkan terlebih dahulu.

Algoritma adalah deretan instruksi yang jelas untuk memecakan masalah, yaitu untuk memperoleh keluaran yang diinginkan dari suatu masukan [7]. Beberapa macam algoritma dibuat karena proses *sorting* tersebut sangat mendasar dan sering digunakan. Oleh karena itu pemahaman atas algoritma-algoritma yang ada sangatlah berguna. Banyak metode pada proses pengolahan data, salah satu diantaranya adalah metode pengurutan (*sorting*), pengurutannya secara urut naik (*ascending*) dan pengurutan secara urut turun (*descending*). Metode-metode pengurutan data pun ada berbagai jenis. Mulai dari *binary sort, insertion sort, merge sort, heap sort, buble sort* dan lain-lain.

Masing-masing jenis algoritma memiliki berbagai tingkat efektivitas dan efektivitas algoritma dapat diukur dengan berapa banyak waktu dan ruang (*space* atau algoritma memori) yang diperlukan untuk menjalankan algoritma, efektif algoritma yang dapat meminimalkan kebutuhan ruang dan waktu. Namun membutuhkan waktu dan ruang suatu algoritma bergantung pada jumlah data yang diolah dan algoritma yang digunakan. Masalah algoritma dengan waktu eksekusi yang lebih cepat tergantung dengan kompleksitas algoritma, yaitu suatu analisis algoritma tentang ukuran waktu dan ruang memori yang diperlukan oleh suatu algoritma ketika algoritma tersebut dijalankan atau dieksekusi. Sesuai dengan pengertian di atas, kompleksitas algoritma terbagi atas dua macam yaitu kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu diukur dari jumlah tahapan komputasi yang dibutuhkan untuk menjalankan program sebagai fungsi dari jumlah data n (sebagai ukuran masukan). Sedangkan kompleksitas ruang diukur dari memori yang digunakan oleh struktur data yang terdapat di dalam algoritma sebagai fungsi dari jumlah data n (sebagai ukuran masukan).

Dengan menggunakan besaran kompleksitas waktu dan ruang algoritma, akan dapat ditentukan laju pertumbuhan waktu dan ruang yang diperlukan algoritma dengan meningkatnya jumlah data (n). Kompleksitas ruang sering dikaitkan dengan struktur data yang digunakan untuk mengimplementasikan algoritma. Sedangkan struktur data yang digunakan dipengaruhi secara langsung oleh ukuran memori komputer. Dengan berkembangnya teknologi perangkat keras komputer sekarang ini masalah ukuran memori tidak lagi menjadi persoalan kritis karena komputer sekarang mempunyai ukuran memori yang besar dibandingkan dengan jaman dahulu waktu masih menggunakan *mainframe*. Untuk setiap mesin, hal ini ditemukan bahwa pilihan algoritma tergantung pada jumlah elemen yang akan diurutkan [8]. Namun demikian dari segi banyaknya langkah yang diperlukan dalam mengurutkan data tetap menjadi pertimbangan efisiensi. Selain itu faktor waktu juga menjadi pertimbangan yang tidak bisa dielakkan lagi. Dari uraian di atas, maka untuk mengetahui tingkat kecepatan dan tingkat efisiensi suaru metode pengurutan data, maka perlu dianalisis kompleksitasnya baik kompleksitas waktu maupun kompleksitas ruang yang menitikberatkan pada

banyaknya langkah yang diperlukan dalam mengurutkan data.

Waktu dan ruang yang dibutuhkan oleh algoritma bergantung pada jenis komputer dan *compiler* bahasa pemrograman. Selain bergantung pada komputer, ukuran waktu dan ruang yang dibutuhkan oleh suatu program juga ditentukan oleh *compiler* bahasa pemrograman yang digunakan. *Compiler* yang berbeda belum tentu menerjemahkan program ke dalam kode mesin (*object code*) yang sama. Sebagai implikasinya, kode mesin yang berbeda akan menggunakan waktu dan ruang memori yang berbeda pula [1]. Dengan demikian kebergantungan tersebut, maka secara teoritis model abstrak pengukuran waktu dan ruang harus bebas dari pertimbangan mesin dan *compiler* apapun. Besaran yang dipakai untuk menerangkan model abstrak pengukuran waktu dan ruang adalah kompleksitas algoritma.

Kompleksitas algoritma diukur berdasarkan kinerjanya dengan menghitung waktu eksekusi suatu algoritma. Waktu eksekusi algoritma dapat diklasifikasikan menjadi tiga kelompok besar, yaitu *best-case* (kasus terbaik), *average-case* (kasus rerata) dan *worst-case* (kasus terburuk) [3]. Pada pemrograman yang dimaksud dengan kasus terbaik, kasus terjelek dan kasus rerata suatu algoritma adalah besar kecilnya atau banyak sedikitnya sumber-sumber yang digunakan oleh suatu algoritma. Makin sedikit makin baik; makin banyak makin jelek. Biasanya sumber-sumber yang paling dipertimbangkan tak hanya waktu eksekusi tetapi bisa juga besar memori, catu-daya dan sumber-sumber lain.

Kompleksitas waktu, T(n), adalah jumlah operasi yang dilakukan untuk melaksanakan algoritma sebagai fungsi dari ukuran masukan n. [10]. Selain itu dianalisis pula besar dari kompleksitas waktu asimptotik dari suatu algoritma yang dinotasikan dengan *Big*-O. Semakin besar nilai *Big*-O, maka semakin tidak efektif lah algoritma tersebut digunakan. Pada makalah ini, akan dilakukan analisis besarnya kompleksitas waktu algoritma (T(n)) dan kompleksitas waktu asimptotik (O) terhadap beberapa algoritma pengurutan yang popular. Permasalahan pengurutan (*sorting problem*) secara formal didefinisikan sebagai berikut:

- *Input*: suatu urutan n bilangan.
- *Output*: suatu permutasi atau penyusunan kembali dari *input* sedemikian rupa sehingga pada tata urutan *ascending* (dari nilai kecil ke besar) ata pada urutan *descending* (dari nilai besar ke kecil).

Sebagai contoh jika diberikan masukan delapan bilangan acak maka keluarnya adalah sebagai berikut ini:

Input:

3	4	8	7	6	2	1	5	n=8	*Out put:*
1	2	3	4	5	6	7	8	*Ascending*	
8	7	6	5	4	3	2	1	*Descending*	

Data yang diurutkan tidak harus atau tidak hanya seperti contoh diatas, namun bisa saja *string* dengan pengurutan awalan acak di depannya ataupun jumlah karakter pada string tersebut. Dapat disimpulkan bahwa dalam pengurutan harus terdapat:

- Data yang diurutkan dalam tipe yang sama atau setidaknya memperoleh perlakuan data yang sama.
- Aturan pengurutan yang jelas.

Struktur dari algoritma *heap sort* adalah sebuah pohon biner sempurna yang memenuhi *property heap*. Node akar (*root node*) memiliki data terbesar atau terkecil yang terdapat pada pohon algortima *heap sort* termasuk algoritma *sorting* yang sulit dipahami karena banyak langkah-langkah dalam mengurutkan data.

2. METODE PENELITIAN

a Ketiga algoritma tersebut diimplementasikan untuk mengurutkan sejumlah angka yang dilakukan oleh pengguna.
b Kedua kode sumber untuk ketiga algoritma tersebut diubah untuk dapat mengurutkan angka yang dihasilkan secara acak dengan jumlah angka sebanyak permintaan dari pengguna untuk mengetahui seberapa baik performa dalam mengurutkan data.
c Ketiga algoritma tersebut mengurutkan sejumlah angka acak dengan rentan jumlah yang sudah ditentukan dan kemudian hasilnya dibandingkan.

3. HASIL DAN PEMBAHASAN

A. Perangkat Pengujian

Penelitian ini diimplementasikan pada satu buah komputer dengan spesifikasi Processor Intel(R) Core(TM) i3-2328M CPU @ 2.20GHz dengan RAM 2048MB yang bersistem operasi Windows 10 Pro 32-bit dengan perangkat lunak BlueJ 3.0.9 untuk penyusunan program komputer.

B. *Heap Sort*

Algoritma *heap sort* merupakan algoritma pengurutan yang cukup cepat [2]. Prosedur atau fungsi dasar yang digunakan adalah sebagai berikut:

- Prosedur *Heapify* yang *running* dalam O(log n) kali, merupakan kunci dalam algoritma ini.
- Prosedur *BuildHeap*, yang *running* dalam linear *time*, yang menghasilkan *heap* dari *input* array yang tidak terurutkan
- Prosedur *heap sort*, yang *running* dalam O(n log n) kali, mengurutkan array sesuai dengan posisinya.
- Sebagai contoh,struktur data *heap* ini dapat dijelaskan atau ditampilkan dalam bentuk pohon biner lengkap (*a complete binary tree*).

Algoritma dan *Pseudocode*

Binary heap digunakan sebagai struktur data dalam algoritma Heap-Sort. Sebagaimana diketahui, ketika suatu Heap dibangun maka kunci utamanya adalah: node atas selalu mengandung elemen lebih besar dari kedua node dibawahnya. Apabila elemen berikutnya ternyata lebih besar dari elemen *root*, maka harus di swap dan lakukan: proses heapify lagi. *Root* dari suatu *heap sort* mengandung elemen terbesar dari semua elemen dalam *heap*.

Proses *heap sort* dapat dijelaskan sebagai berikut:

a) Representasikan *heap* dengan n elemen dalam sebuah array A[n].
b) Elemen *root* tempatkan pada A[1].
c) Elemen A[2i] adalah node kiri dibawah A[i].
d) Elemen A[2i+1] adalah node kanan dibawah A[i].
e) Ambil nilai *root* (terbesar) A[1..n-1] dan pertukarkan dengan elemen terakhir dalam array, A[n]
f) Bentuk *Heap* dari (n-1) elemen, dari A[1] hingga A[n-1]
g) Ulangi langkah 5 dimana indeks terakhir berkurang setiap langkah

Dalam bentuk *pseudocode*, algoritma *heap sort* dapat terlihat seperti ini:

proceduresiftdown(input/output: H: *heap*)
var
    parent,*largerchild* : node
begin
    parent = *root* dari H
    largerchild = anak dari parent yang menampung elemen terbesar
while elemen di *parent* lebih kecil dari elemen di *largerchild* do
        tukar elemen di *parent* dengan elemen di *largerchild*
        parent = largerchild
        largerchild = anak dari *parent* yang menampung elemen terbesar
        end
end

Kompleksitas *Heap Sort*

Dengan menganalisa *makeheap*, maka didapat banyak pembandingan elemen

yang terjadi oleh makeheap paling banyak 2(n - 1). Selanjutnya dengan menganalisa *removekeys*, maka banyaknya pembanding elemen yang dilakukan oleh *removekeys* paling banyak adalah

$$2\sum_{j=1}^{d-1} j2^j = (d2^d - 2^{d+1} + 2) = 2n\log n - 4n + 4 \quad (1)$$

Dengan mengkombinasikan analisis dari *makeheap* dan *removekeys*, didapat banyaknya pembandingan elemen di *heap sort* ketika n merupakan eksponen dari 2 paling banyak adalah

2(n − 1) + 2n log n - 4n + 4 = 2(n log n - n + 1) ≈ 2n log n

Sehingga untuk n merupakan eksponen dari 2,

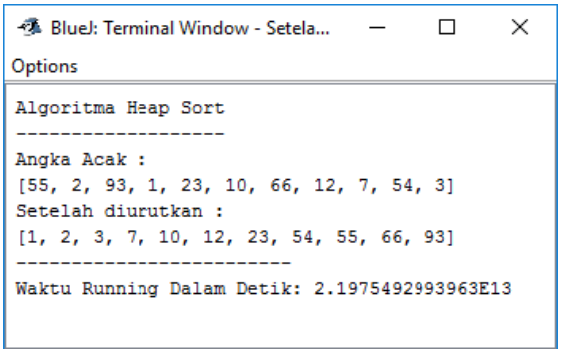
$$W(n) \approx 2n \log n \in \theta\ (n \log n) \quad (2)$$

Sulit untuk menganalisa kompleksitas kasus rata-rata *heap sort* secara analitis. Namun, studi empiris telah menunjukkan bahwa kompleksitas untuk kasus rataratanya tidak lebih baik dari kasus terburuknya. Ini menunjukkan bahwa kompleksitas untuk kasus rata-rata *heap sort* adalah

$$A(n) \approx 2n \log n \in \theta\ (n \log n) \quad (3)$$

Analisis *Heap Sort*

Kompleksitas waktu pengurutan data algoritma *heap sort* tidak terpengaruh oleh bermacam bentuk dan kondisi data awal [4]. Banyaknya perbandingan yang harus dilakukan untuk siklus pertama adalah n, perbandingan yang harus dilakukan untuk siklus yang kedua n-1, dan seterusnya. Sehingga jumlah keseluruhan perbandingan adalah 2(n-1) perbandingan. Menggunakan algoritma pengurutan seleksi, hindari pengurutan nilai dengan data pada tabel lebih besar dari 1000 buah, dan hindari mengurutkan tabel lebih dari beberapa ratus kali [9]. Berikut Dengan interval data antara 1 sampai dengan 10 elemen. Waktu eksekusi diukur dengan satuan *Second* (s).



Gambar 1. Hasil Implementasi *Heap Sort*

C. *Insertion Sort*

Cara kerja algoritma ini yaitu pengurutandengan penyisipan bekerja dengan cara menyisipkan masing-masing nilai di tempat yang sesuai di antara elemen yang lebih kecil atau sama dengan nilai tersebut. 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*. Contoh dari algoritma ini dapat kita ambil dalam kehidupan sehari-hari, misalnya mengurutkan kartu remi [6]. Anggaplah bahwa terdapat sebuah meja yang berisi setumpuk kartu. Meja ini melambangkan kondisi larik sebelum diurutkan. Langkah-langkah pengurutan adalah sebagai berikut:

a) Ambil kartu pertama dari meja, letakkan di tangan kiri.
b) Ambil kartu kedua dari meja, bandingkan dengan kartu yang berada di tangan kiri,kemudian letakkan pada urutan yang sesuai setelah perbandingan.
c) Ulangi proses hingga seluruh kartu pada meja telah diletakkan berurutan pada tangan kiri.

Kartu-kartu pada tangan kiri tersebut menunjukkan kondisi larik sesudah diurutkan.

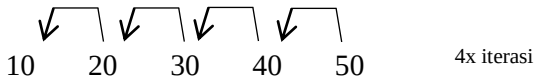
Algoritma dan *Pseudocode*

Pseudocode untuk algoritma *insertion sort* adalah sebagai berikut:

(*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}

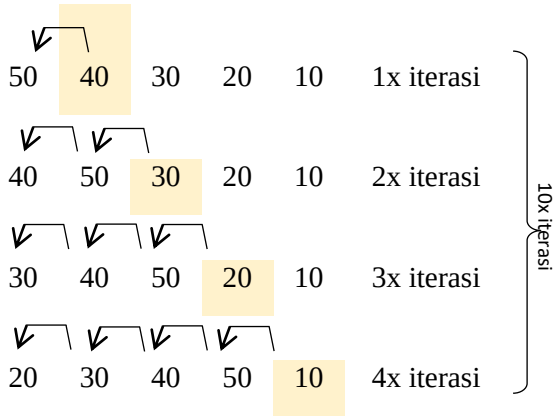
Kompleksitas Algoritma *Insertion Sort*

Kondisi terbaik tercapai jika data telah terurut, hanya satu perbadingan dilakukan untuk setiap posisi i, sehingga terdapat n-1 perbandingan, atau O(n).



Gambar 2. Kondisi *Best Case* Pada *Insertion Sort*

Kondisi terburuk (*worst case*) tercapai jika data telah urut namun dengan urutan yang terbalik. Pada kasus ini, untuk setiap i, elemen data(i) lebih kecil dari elemen data(0), ... data(i-1), masing-masing dari elemen dipindahkan satu posisi.



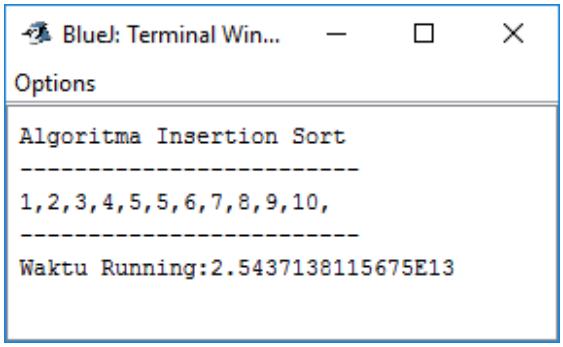
Gambar 3. Kondisi *Worst Case* Pada *Insertion Sort*

Untuk setiap iterasi i pada kalang for tertular selalu ada perbandingan i, sehingga jumlah total perbandingan untuk seluruh iterasi pada kalang ini adalah:

$$\sum_{i=1}^{n-1} i = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2) \quad (4)$$

Analisis *Insertion Sort*

Untuk kasus terbaik algoritma ini berjalan 1 kali, yaitu jika elemen dalam tabel telah terurut. Kalang (loop) while tidak pernah dijalankan. Untuk kasus terburuk algoritma ini berjalan Nmax kali. Sehingga, seperti pengurutan gelembung, pengurutan dengan penyisipan mempunyai kompleksitas algoritma O(n2). Algoritma *insertion sort* secara teknis lebih mudah diterapkan dibandingkan dengan *merge sort*, berkaitan dengan panjangnya intruksi yang diperlukan [5]. Namun, algoritma ini tetap kurang efisien untuk tabel berukuran besar (menyimpan banyak nilai). Dengan interval data antara 1 sampai dengan 10 elemen. Waktu eksekusi diukur dengan satuan *Second* (s).



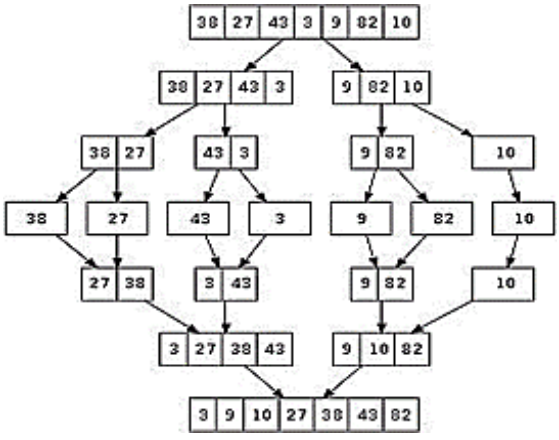
Gambar 4. Hasil Implementasi *Insertion Sort*

D. Algoritma *Merge* Sort

Secara konseptual, untuk sebuah array berukuran n, *merge sort* bekerja sebagai berikut:

a) Jika bernilai 0 atau 1, maka array sudah terurut.
   Sebaliknya:
b) Bagi array yang tidak terurut menjadi dua sub-array, masing-masing berukuran n/2.
c) Urutkan setiap sub-array. Jika sub-array tidak cukup kecil, lakukan rekursif langkah 2 terhadap sub-array.
d) Menggabungkan dua sub-array kembali menjadi satu array yang terurut.

Sebagai contoh jika terdapat data berupa 38, 27, 43, 3, 9, 82 dan 10 maka ilustrasi pengurutannya adalah sebagai berikut:



Gambar 5. Konsep *Sorting Merge Sort*

Algoritma dan *Pseudocode*

Merge sort menggabungkan dua ide utama untuk meningkatkan *runtime*-nya:

a) Array kecil akan mengambil langkah-langkah untuk menyortir lebih sedikit dari array besar.
b) Lebih sedikit langkah yang diperlukan untuk membangun sebuah array terurut dari dua buah array terurut daripada dari dua buah array tak terurut.

```
merge (arrayl, pertama, terakhir)
        tengah = (pertama + terakhir)/2;
        il = 0;
        i2 = pertama;
        i3 = tengah + 1;
        while kedua sub larik dari
        array1 memiliki elemen
        if arrayl[i2] < arrayl[i3]
                temp[il++] = arrayl[i2++];
        else
                temp[il++] = arrayl[i3++];
        masukkan ke dalam temp sisa
        elemen dari arrayl;
masukkan ke arrayl isi dari temp;
```

Kompleksitas Algoritma *Merge Sort*

Kompleksitas algoritma untuk larik dengan n elemen dan jumlah pergeseran (*T*) dihitung melalui relasi rekursif berikut ini:

$T(1) = 0$

$T(n) = 2T\left(\frac{n}{2}\right) + 2n$

Adapun M(n) dihitung lewat cara berikut:

(a) $T(n) = 2\left(2T\left(\frac{n}{4}\right) + 2\left(\frac{n}{2}\right)\right) + 2n = 4T\left(\frac{n}{4}\right) + 4n$

$= 4\left(2T\left(\frac{n}{8}\right) + 2\left(\frac{n}{4}\right)\right) + 4n = 8T\left(\frac{n}{8}\right) + 6n$

$= 2^iT\left(\frac{n}{2^1}\right) + 2in$

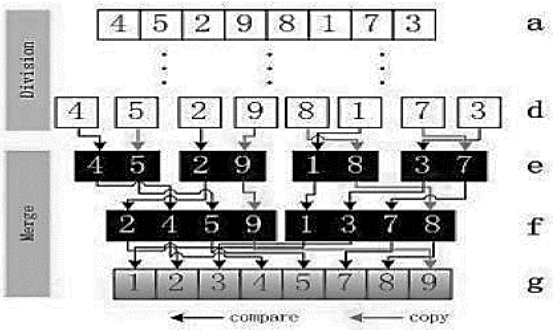
Memilih I = log n sedemikian sehingga n= 2i, maka diperoleh:

(b) $T(n) = 2^iT\left(\frac{n}{2^1}\right) + 2in = nT(1) + 2n.\log n$

$= 2n.\log n = O(n \log n)$

Kasus terburuk (*worst case*) terjadi bila selama pemanggilan fungsi setiap rekursif

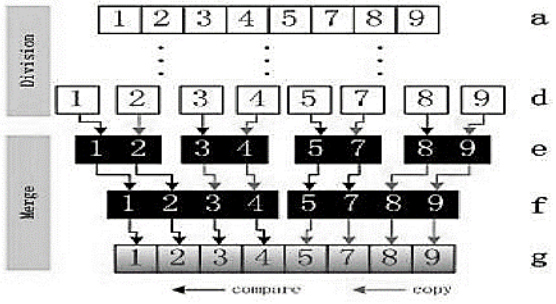
merge, nilai terbesar dari setiap elemen terletak di larik yang berbeda. Hal ini memaksa fungsi *merge* untuk melakukan pengurutan secara berpindah-pindah antar larik, seperti yang digambarkan berikut:



Gambar 6. Kondisi *Worst Case* Pada *Merge Sort*

Maka kompleksitas pada kondisi *worst case* adalah O (n log n).

Kasus terbaik (*best case*) untuk metode ini dijumpai pada kondisi dimana elemen memiliki nilai terbesar yang lebih kecil dibandingkan dengan seluruh nilai pada elemen yang lain [6], sebagaimana berikut ini:



Gambar 7. Kondisi *Best Case* Pada *Merge Sort*

Pada skenario ini hanya n/2 perbandingan dari elemen yang diperlukan. Menggunakan proses perhitungan yang sama sebagaimana dalam kasus terburuk, diperoleh [6]:

(a) $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{2}$

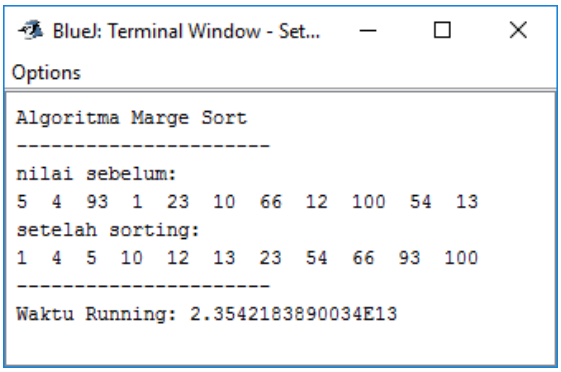
$T(n) = 2^i T\left(\frac{n}{2^i}\right) + \frac{in}{2}$

$T(n) = n * 0 + \frac{n}{2}\log n$

Dengan kata lain, diperoleh juga kompleksitas yang sama, O (n log n) [6].

Analisis *Merge Sort*

Algoritma *merge sort* melalukan pass atau traversal berkali-kali, dan setiap kali pass mengurutkan sejumlah nilai yang sama dengan ukuran set menggunakan *insertion sort*. Ukuran dari set yang harus diurutkan semakin membesar setiap kali melakukan pass pada tabel, sampai set tersebut mencakup seluruh elemen tabel. Ketika ukuran dari set semakin membesar, sejumlah nilai yang harus diurutkan semakin mengecil. Ukuran dari set yang digunakan untuk setiap kali iterasi (*iteration*) mempunyai efek besar terhadap efisiensi 2) pengurutan. Algoritma *merge sort* adalah algoritma yang relatif sederhana. Hal ini menjadikan algoritma *merge sort* adalah pilihan yang baik dan efisien untuk mengurutkan nilai dalam suatu tabel berukuran sedang. Dengan interval data antara 1 sampai dengan 10 elemen. Waktu eksekusi diukur dengan atuan *Second* (s).



Gambar 8. Hasil Implementasi *Merge Sort*

4. SIMPULAN

Berdasarkan analisis yang dilakukan untuk menemukan besarnya kompleksitas waktu algoritma dan kompleksitas waktu asimptotik tiap algoritma yang telah dilakukan sebelumnya, dapat dibentuk sebuah tabel untuk menampilkan nilai

kompleksitas waktu asimptotik untuk setiap algoritma.

Tabel 1. Nilai Kompleksitas Algoritma

Algoritma	Kompleksitas (O)		
	Best Case	***Average Case***	***Worst Case***
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Metode pengurutan algoritma *heap sort* metode pengurutan data yang tergolong mempunyai kecepatan tinggi, dimana kompleksitas dan kecepatan waktu pengurutan data yang dibutuhkan untuk proses pengurutan memiliki proses O (n log n).

Hasil pengamatan berdasarkan tabel data dan kecepatan proses pengurutan, serta grafik pertumbuhan waktu proses pengurutan data terhadap jumlah data. Pengurutan data dengan algoritma *heap sort* memberikan waktu proses yang konsisten naik dalam peningkatan lama waktu proses terhadap jumlah data.

Dengan demikian algoritma *heap sort* dalam proses pengurutan data termasuk algoritma yang relatif cepat dibanding dengan *insertion sort* dan *merge sort*, proses pengurutan data dengan algoritma *heap sort* tidak terpengaruh kondisi atau urutan data (bukan jumlah data, sedangkan jumlah data mempengaruhi lamanya waktu proses pengurutan, dimana pertumbuhan waktu proses sesuai dengan penambahan jumlah data), waktu proses pengurutan konsisten naik sesuai dengan pertambahan jumlah data.

DAFTAR PUSTAKA

[1] C. A. Akhyati, A. Johar, B. Susilo, *Perangkat Lunak Pendukung Pembelajaran Algoritma Heap Sort*. Bengkulu: Universitas Bengkulu, 2014.

[2] Ardi, A. Wijaya, dan F. N. Noris, *Aplikasi Simulasi Pengurutan Data Menggunakan Algoritma Heap Sort*. Bengkulu: Universitas Muhammadiyah Bengkulu, 2015.

[3] O. Goldreich, "Computational Complexity: A Conceptual Perspective". New York: Cambridge University Press, 2008.

[4] S. Irianto, dan H. Mustafidah, *Analisis Kompleksitas Waktu dan Ruang Terhadap Laju Pertumbuhan Algoritma Heap Sort*. Purwokerto: Universitas Muhammadiyah Purwokerto, 2006.

[5] A. H. Saptadi, dan D. W. Sari, *Analisis Algoritma Insertion Sort, Merge Sort dan Implementasinya Dalam Bahasa Pemrograman C++*. Palembang: Universitas Sriwijaya Palembang, 2012.

[6] R. Hibbler, "Merge Sort", Dept. Of Computer Science. Florida Institute of Technology. Florida, USA. 2008.

[7] Saputra, dkk. 2010. *Analisis Algoritma Rekursif Quick Sort* (http://www.mediafire.com/download/py3q5jevwp59vj/DAA+2010+IF32_01+Analisis+Algoritma+Rekursif+Quick+Sort.pdf, diakses 31 Januari 2017)

[8] V. Sharma, S. Singh, K. S. Kahlon, 2008. "Performance Study of Improved Heap Sort Algorithm and Other Sorting Algorithms on Different Platforms". India: Department of Computer science & Engineering Chitkara Institute of Engg. & Technology.

[9] Suarga. "Algoritma Pemrograman". Yogyakarta: Andi, 2012.

[10] Tjaru, dan B. N. Setia, 2010. "Kompleksitas Algoritma Pengurutan Selection Sort dan Insertion Sort". *Makalah IF2091 Strategi Algoritmik Tahun 2009* (http://informatika.stei.itb.ac.id/~rinaldi.muir/Matdis/20092010/Makalah091

0/MakalahStrukdis0910074.pdf, diakses 31 Januari 2017)

[11] Yahya, dan Y. Sofyansyah, 2014. "Analisa Perbandingan Algoritma Bubble Sort dan Selection Sort Dengan Metode Perbandingan Eksponensial". Jurnal Pelita Informatika Budi Darma, Volume 6, Nomor 3. (http://pelita-informatika.com/berkas/jurnal/28.%20Sofyansayah.pdf, diakses 31 Januari 2017)