Kompleksitas Algoritma dalam Strategi Algoritma *Sorting*

Emilia Andari Razak/13515056
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13515056@std.itb.ac.id

Abstrak*— Kompleksitas algoritma digunakan untuk menentukan efisiensi suatu algoritma berdasarkan waktu (time) atau ruang (space). Kompleksitas algoritma adalah salah satu tolak ukur dalam memilih algoritma, seperti algoritma pengurutan. Untuk jumlah data yang sangat besar, kompleksitas algoritma sangat memengaruhi waktu eksekusi program. Pada makalah ini algoritma yang akan dibahas adalah *merge sort, insertion sort, selection sort,* dan *quick sort*. algoritma yang kompleksitas waktunya paling efisien adalah *quick sort,* diikuti dengan *merge sort, insertion sort,* lalu *selection sort.

***Kata kunci*—Algoritma, Kompleksitas, *Sorting*, Waktu**

I. PENDAHULUAN

Pada dunia pemrograman, banyak algoritma yang dapat digunakan. Untuk membandingkan algoritma-algoritma tersebut, banyak faktor yang perlu dipertimbangkan, seperti kompleksitas algoritma. Kompleksitas suatu algoritma dapat ditinjau dari dua faktor, yaitu waktu (*time*) dan ruang (*space*). Dalam dunia nyata, data yang digunakan dan dimanipulasi tidaklah sedikit. Dengan data yang besar, kompleksitas suatu algoritma akan sangat memengaruhi waktu eksekusi suatu program.

Algoritma pengurutan biasanya digunakan untuk mengoptimasi penggunaan algoritma lain (seperti *search* dan *merge*). *Sorting* berperan besar dalam pemrosesan data komersil dan komputasi sains modern. Penggunaan sorting di dunia nyata dapat ditemukan pada transaksi perbankan, optimasi kombinatorial, astrofisika, perkiraan cuaca, dan masih banyak bidang lainnya.

Dari puluhan algoritma pengurutan, terdapat masing-masing algoritma memiliki kekurangan dan kelebihan. Algoritma pengurutan biasanya dilihat dari kestabilan algoritma, kebutuhan memori, dan waktu. Pengukuran waktu juga dilihat dari kemungkinan terbaik (*best*), kemungkinan terburuk (*worst*), atau rata-rata (*average*). Namun, pada makalah ini akan lebih ditekankan waktu eksekusi yang dibutuhkan.

II. DASAR TEORI

- **Kompleksitas Algoritma**

Kebutuhan waktu dan ruang suatu algoritma bergantung pada ukuran masukan (n), yang menyatakan jumlah data yang diproses. Komputer dengan arsitektur yang berbeda memiliki waktu operasi yang berbeda. *Compiler* yang digunakan dalam eksekusi program juga memengaruhi waktu eksekusi. Oleh karena itu, untuk membandingkan kompleksitas waktu algoritma-algoritma itu sendiri dibutuhkan model yang independen dari kedua variable di atas (dan variable lainnya). Kompleksitas waktu **T(n)**, adalah fungsi yang menyatakan waktu yang dibutuhkan untuk menjalankan suatu algoritma berdasarkan jumlah tahapannya, dengan ukuran masukan **n**.

Kompleksitas waktu dibedakan menjadi tiga macam:

1. $\mathbf{T_{max}(n)}$ -> kasus terburuk (*worst-case*)
2. $\mathbf{T_{min}(n)}$ -> kasus terbaik (*best-case*)
3. $\mathbf{T_{avg}(n)}$ -> kasus rata-rata

Dalam kompleksitas algoritma, fungsi $T(n)=2n^2+6n+1$ relatif sama dengan $T(n)=n^2$. Fungsi-fungsi dengan kompleksitas waktu yang relatif mirip, memiliki orde yang sama. Pada contoh ini, kedua fungsi tersebut berorder $O(n^2)$. Kita menggunakan notasi *big-O* untuk menghitung secara asimptotik laju pertumbuhan waktu eksekusi algoritma dengan batas atas (*upper-bound*).

Pengelompokkan algoritma berdasarkan notasi-O besar (diurutkan dari yang tercepat sampai yang terlambat):

Notasi-O	Nama
O(1)	*Constant*
O(log n)	*Logarithmic*
O(n)	*Linear*
O(n log n)	*Log Linear*
$O(n^2)$	*Quadratic*
$O(n^3)$	*Cubic*
$O(2^n)$	*Exponential*

Tabel 2.1: Notasi-O

- **Teorema Master**

Misalkan,

$T(n)= aT(n/b) + f(n)$

Dimana a dan b adalah konstanta, $a \geq 1$ dan $b>1$. F(n) adalah waktu dari fungsi untuk membagi n menjadi n/b dan menggabungkan solusinya.

Jika $f(n) \in O(n^d)$, dimana $d \geq 0$, d menyatakan banyak rekurens

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Gambar 2.1: Teorema Master. Sumber: *Introduction to the Design and Analysis of Algorithms*, 3rd Ed – Levitin, halaman 171

III. PEMBAHASAN

A. Kompleksitas Algoritma Merge Sort

ALGORITHM *Mergesort*$(A[0..n-1])$
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
Mergesort$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A) //see below

Gambar 3.1: Algoritma *Merge Sort* (1). Sumber: *Introduction to the Design and Analysis of Algorithms*, 3rd Ed – Levitin, halaman 172

Algoritma *merge* sort melakukan operasi *sorting* terhadap sebuah *array* berukuran A[0..n-1], membagi *array* menjadi dua A[0..(n/2) – 1] dan A[n/2.. n-1], melakukan *sorting* secara rekursif, lalu menggabungkan hasil yang telah di *sort* menjadi sebuah *array* berukuran A[0..n-1].

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$
//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
if $B[i] \le C[j]$
$A[k] \leftarrow B[i]$; $i \leftarrow i+1$
else $A[k] \leftarrow C[j]$; $j \leftarrow j+1$
$k \leftarrow k+1$
if $i = p$
copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Gambar 3.2: Algoritma *Merge Sort* (2). Sumber: *Introduction to the Design and Analysis of Algorithms*, 3rd Ed – Levitin, halaman 172

Pada prosedur *merge*, dua buah *array* yang sudah terurut B[0..p-1] dan C[0..q-1] dibandingkan kembali kemudian disatukan menjadi sebuah *array* A[0..p+q-1]. Hal ini dilakukan di dalam rekurens.

Bagaimana tingkat efesiensi dari algoritma *merge sort*? Karena algoritma *merge sort* selalu membagi 2, maka lebih mudah jika kita memisalkan bahwa n adalah selalu pangkat dari 2. Jadi,

$$T(n) = 2T(n/2) + T_{merge}(n) \quad (1)$$

Pada $T_{merge}(n)$, dilakukan perbandingan antara elemen-elemen *array*. Setiap tahap, sebuah perbandingan dilakukan sehingga jumlah elemen yang harus dibandingkan pada setiap tahap berkurang 1. Pada kasus terburuk, *array* B dan *array* C tidak ada yang kosong terlebih dahulu kecuali tinggal satu elemen. Sejumlah n-1 perbandingan harus dilakukan. Maka dari itu, untuk kasus terburuk $T_{merge}(n)=n-1$. Menyulihkan $T_{merge}(n)$ kepada (1) maka kita mendapatkan

$$T_{worst}(n) = 2T_{worst}(n/2) + n-1 \quad (2)$$

Untuk n>1, $T_{worst}(1)=0$. Karena, pada satu elemen tidak ada perbandingan yang dilakukan. Lalu, bagaimana dengan notasi-O-nya? Menurut Teorema Master, dengan a=b=2, dan d=1 (karena $T_{merge}(n)=n-1 \in O(n^1)$). Pada kasus ini $a=b^d$. Jadi *merge sort* termasuk $O(n^d \log n)$, atau disimplifikasi menjadi O(n log n).

Kelebihan algoritma *merge sort* dibandingkan dengan algoritma sorting lainnya adalah stabilitas dari *merge sort*. Dari kasus terbaik sampai terburuk algoritma *merge sort* dapat mempertahankan kompleksitas waktu O(n log n). Namun, kekurangan untuk algoritma *merge sort* adalah *space complexity* yang cukup tinggi, yaitu sejumlah n.

B. Kompleksitas Algoritma Quick Sort

ALGORITHM *Quicksort*$(A[l..r])$
//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
$s \leftarrow$ *Partition*$(A[l..r])$ //s is a split position
Quicksort$(A[l..s-1])$
Quicksort$(A[s+1..r])$

Gambar 3.3: Algoritma *Quick Sort(1)*. Sumber: *Introduction to the Design and Analysis of Algorithms*, 3rd Ed – Levitin, halaman 176

Algoritma *Quick Sort* membagi *array* berdasarkan *value* elemennya. Untuk melakukan partisi, dibutuhkan sebuah *pivot* S. Masing-masing elemen dibandingkan dengan S, sehingga *array* di sebelah kiri mengandung elemen yang lebih kecil dari S, dan *array* di sebelah kanan mengandung elemen yang lebih besar dari S.

ALGORITHM *HoarePartition*$(A[l..r])$
//Partitions a subarray by Hoare's algorithm, using the first element
// as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: Partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
repeat $i \leftarrow i+1$ **until** $A[i] \ge p$
repeat $j \leftarrow j-1$ **until** $A[j] \le p$
swap$(A[i], A[j])$
until $i \ge j$
swap$(A[i], A[j])$ //undo last swap when $i \ge j$
swap$(A[l], A[j])$
return j

Gambar 3.4: Algoritma *Quick Sort* (2). Sumber*: Introduction to the Design and Analysis of Algorithms*, 3rd Ed – Levitin, halaman 178

Partisi dilakukan dengan dua buah *index pointer* i dan j, yang menelusuri *array* dari kiri dan dari kanan. *Index pointer* i menelusuri *array* dari kiri dan melewati elemen-elemen yang lebih kecil dari *pivot*, kemudian berhenti saat ada elemen yang lebih besar atau sama dengan *pivot*. Sebaliknya, *index pointer* j menelusuri *array* dari kanan dan melewati elemen-elemen yang lebih besar dari *pivot*, kemudian berhenti saat ada elemen yang lebih kecil atau sama dengan *pivot*.

Akan ada tiga situasi setelah penelusuran berhenti. Jika i<j, maka A[i] dan A[j] ditukar. Jika i>j, kita sudah menelusuri seluruh array. Jika i=j, maka *index pointer* i dan j menunjuk ke p.

Jadi, jumlah langkah yang dilakukan algoritma partisi adalah n+1 (jika i>j) atau n (jika i=j). Kasus terbaik untuk algoritma *quick sort* adalah ketika pemisahan yang dilakukan menghasilkan dua buah *array* yang jumlah elemennya sama rata. Sehingga,

$$T_{best}(n) = 2T_{best}(n/2) + n \quad (1)$$

Menurut Teorema Master, $T_{best}(n) \in O(n \log n)$. Hal ini dikarenakan a=b=2. Dan f(n)=n, sehingga d=1. Oleh karena itu, Tbest(n) memenuhi $a=b^d$.

Pada kasus terburuk algoritma *quick sort*, *array* yang dihasilkan akan berat sebelah. Dimana salah satu *array* kosong dan salah satu *array* berisi semua elemen pada *array* A[0..n-1]. Hal ini justru mungkin terjadi pada *array* yang telah terurut. Karena menggunakan A[0] sebagai *pivot*, maka *index pointer* i akan berhenti pada A[1], dan *index pointer* j akan berhenti pada A[0]. Sehingga menghasilkan *array* yang sama. Jadi jumlah perbandingan yang dilakukan adalah

$$T_{worst}(n) = (n+1) + n + \ldots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in O(n^2) \quad (2)$$

Algoritma *quick sort* bekerja sangat baik dalam melakukan pengurutan terhadap elemen-elemen yang *random*. *Quick sort* bekerja sedikit lebih cepat dibandingkan algoritma O(n log n) lainnya seperti *merge sort*. *Space complexity* yang dibutuhkan juga tidak terlalu banyak, yaitu O(log n). Walaupun begitu, quick sort memiliki kelemahan. Algoritma quick sort tidak stabil.

C. Kompleksitas Algoritma Selection Sort

ALGORITHM *SelectionSort*$(A[0..n-1])$
//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
  $min \leftarrow i$
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[j] < A[min]$ $min \leftarrow j$
  swap $A[i]$ and $A[min]$

Gambar 3.5: Algoritma *Selection Sort*. Sumber*: Introduction to the Design and Analysis of Algorithms*, 3rd Ed – Levitin, halaman 99

Algoritma *selection sort* adalah salah satu algoritma sorting yang paling intuitif, karena dilakukan dengan cara *brute force*. *Selection sort* menelusuri seluruh *array*, mencari elemen terkecil dan menukar elemen terkecil dengan elemen pertama. Lalu, dari elemen kedua menelusuri kembali *array* sampai elemen terakhir, mencari elemen terkecil dan menukar dengan elemen kedua. Hal ini diiterasi terus sampai seluruh *array* telah terurut, sehingga:

$$T(n) = \sum_{i=0}^{n-2}[(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2}(n-1-i) = \frac{(n-1)n}{3}$$

Oleh karena itu, algoritma *selection sort* $\in O(n^2)$.

D. Kompleksitas Algoritma Insertion Sort

ALGORITHM *SelectionSort*$(A[0..n-1])$
//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
  $min \leftarrow i$
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[j] < A[min]$ $min \leftarrow j$
  swap $A[i]$ and $A[min]$

Gambar 3.6: Algoritma *Selection Sort*. Sumber*: Introduction to the Design and Analysis of Algorithms*, 3rd Ed – Levitin, halaman 134

Ide awal dari *selection sort* adalah melakukan penelusuran kepada *array* dari elemen (n-2) ke 0 dan menyisipkan elemen A[n-1] di tempat dimana A[n-1] lebih besar atau sama dengan elemen sebelumnya (sebelah kiri). Hal ini terus dilakukan secara rekursif.

Walaupun ide awal untuk *selection sort* adalah rekursi, lebih mudah untuk mengimplementasi *selection sort* secara iteratif. Dimulai dari A[1] sampai A[n-1], A[i] disisipkan di tempat yang tepat diantara i elemen pertama yang telah terurut.

$$A[0] \le \cdots \le A[j] < A[j+1] \le \cdots \le A[i-1] \mid A[i] \cdots A[n-1]$$

smaller than or equal to $A[i]$ greater than $A[i]$

Gambar 3.7: Ilustrasi *Selection Sort*. Sumber*: Introduction to the Design and Analysis of Algorithms*, 3rd Ed – Levitin, halaman 134

Kunci dari operasi yang dilakukan oleh *selection sort* adalah perbandingan A[j] > v. Jumlah perbandingan yang dilakukan bergantung kepada *array* yang menjadi input untuk algoritma *selection sort*. Pada kasus terburuk, perbandingan A[j] > v dieksekusi pada jumlah terbanyak. Hal ini akan terjadi apabila input *array* adalah *array* kondisi terurut mengecil. Sehingga, jumlah perbandingan yang dilakukan adalah

$$T_{worst}(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

Untuk kasus terburuk, kompleksitas algoritma *selection sort* adalah $O(n^2)$.

Kasus terbaik untuk algoritma *selection sort* adalah ketika perbandingan A[j] > v hanya dilakukan sekali pada setiap iterasi kalang terluar. Hal ini dapat terjadi apabila *array* yang menjadi *input* untuk algoritma *selection sort* telah terurut membesar. Sehingga,

$$T_{best}(n) = \sum_{i=1}^{n-1} 1 = n-1$$

Untuk kasus terbaik, kompleksitas algoritma *selection sort* adalah O(n).

Algoritma	$T_{best}(n)$	$T_{worst}(n)$	$T_{avg}(n)$
Merge sort	O(n log n)	O(n log n)	O(n log n)
Quick sort	O(n log n)	$O(n^2)$	O(n log n)
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$

Tabel 3.1: Rangkuman Kompleksitas Waktu Algoritma Sorting.

IV. EKSPERIMEN

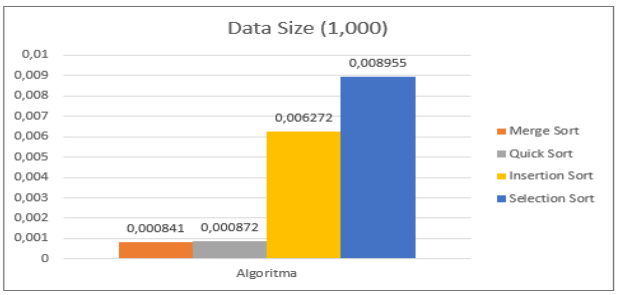
Setelah membahas dan menghitung kompleksitas algoritma berdasarkan dasar teori, maka dilakukan percobaan until menguji kebenaran kompleksitas waktu algoritma tersebut. Percobaan dilakukan dengan menguji algoritma *merge sort*, *quick sort*, *selection sort*, dan *insertion sort* menggunakan dataset yang berukuran kecil hingga besar. *Input* adalah koleksi data bertipe *integer* yang di-*generate* secara *random* berukuran 1.000, 5.000, 10.000, 50.000, 100.000, 500.000, dan 1.000.000.

Untuk mengukur kecepatan algoritma, digunakan method untuk mendapatkan waktu (currentTime) sebelum dan sesudah algoritma pengurutan dieksekusi, lalu dihitung delta antara kedua waktu. Proses *generate* koleksi data secara *random* tidak diukur kecepatannya. Sebelum mengukur kecepatan eksekusi algoritma, dilakukan terlebih dulu pengecekan kebenaran hasil pengurutan yang diberikan oleh setiap algoritma.

Berikut akan ditampilkan grafik untuk perbandingan waktu eksekusi algoritma *merge sort*, *quick sort*, *selection sort*, dan *insertion sort*. Karena eksperimen dilakukan pada komputer yang sama, makan dapat dianggap bahwa perbedaan waktu eksekusi independen dari arsitektur komputer dan *compiler*.

Angka pada kolom y adalah satuan waktu dalam detik (s), ditampilkan masing-masing algoritma dengan waktu persis eksekusinya.

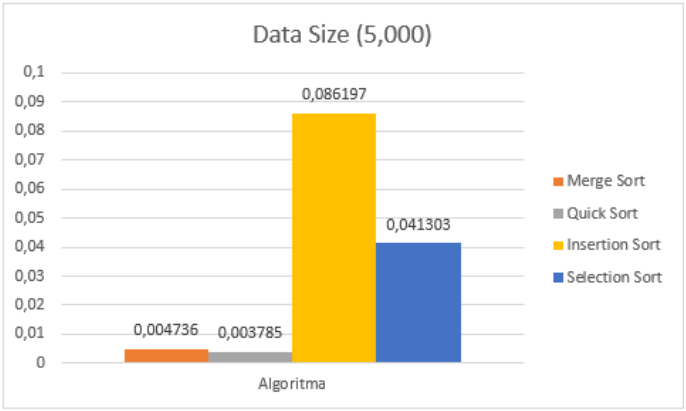
- Data Size 1.000



Grafik 4.1: Perbandingan algoritma *sorting* data *size* 1.000

Pada data *size* 1.000, semua algoritma memiliki waktu eksekusi yang < 1 detik. *Merge sort* algoritma yang paling cepat dengan waktu 0,000841 detik. Diikuti dengan algoritma *quick sort* dengan waktu 0,000872 detik. Kemudian *insertion sort* dengan waktu 0,006272. Lalu terakhir *selection sort* dengan waktu 0,008955.

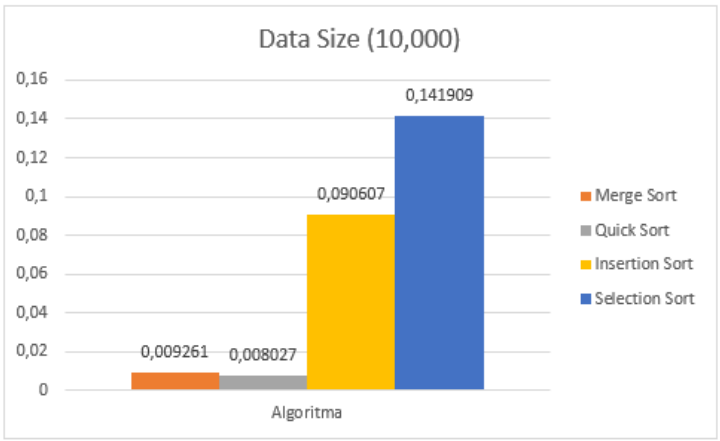
- Data Size 5.000



Grafik 4.2: Perbandingan algoritma *sorting* data size 5.000

Pada data *size* 5.000, semua algoritma masih memiliki waktu eksekusi yang < 1 detik. Kali ini yang paling cepat adalah *quick sort* dengan waktu 0,003785 detik. Diikuti dengan *merge sort* dengan waktu 0,004736 detik. Kemudian *selection sort* dengan waktu 0,041303 detik. Terakhir *insertion sort* dengan waktu 0,086197 detik.

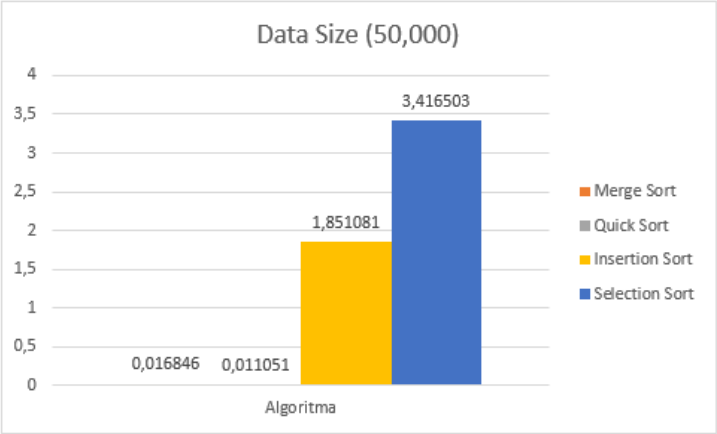
- Data Size 10.000



Grafik 4.3: Perbandingan algoritma *sorting* data size 10.000

Pada data *size* 10.000, semua algoritma masih memiliki waktu eksekusi yang < 1 detik. Kali ini yang paling cepat adalah *quick sort* dengan waktu 0,008027 detik. Diikuti dengan *merge sort* dengan waktu 0,009261 detik. Kemudian *insertion sort* dengan waktu 0,090607 detik. Terakhir *selection sort* dengan waktu 0,141909 detik.

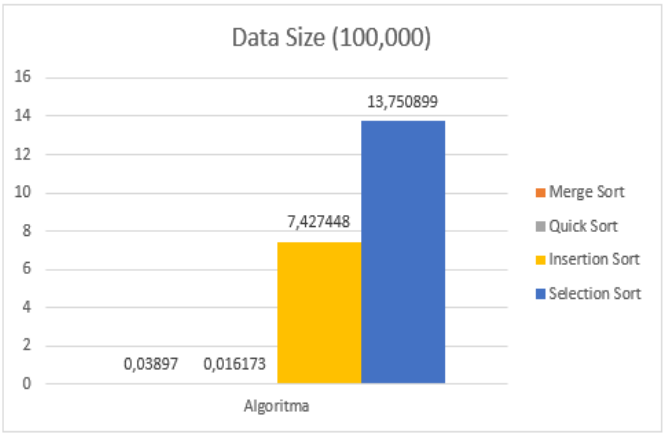
- Data Size 50.000



Grafik 4.4: Perbandingan algoritma *sorting* data size 50.000

Pada data size 50.000, perbedaan waktu eksekusi algoritma sudah mulai terlihat. Dimana algoritma *merge sort* dan *quick sort* sekitar 10^3 kali lebih cepat dari *insertion sort* dan *selection sort*. Algoritma yang paling cepat adalah *quick sort* dengan waktu 0,011051 detik. Diikuti dengan *merge sort* dengan waktu 0,016846 detik. Kemudian *insertion sort* dengan waktu 1,851081 detik. Terakhir *selection sort* dengan waktu 3,416503 detik.

- Data Size 100.000

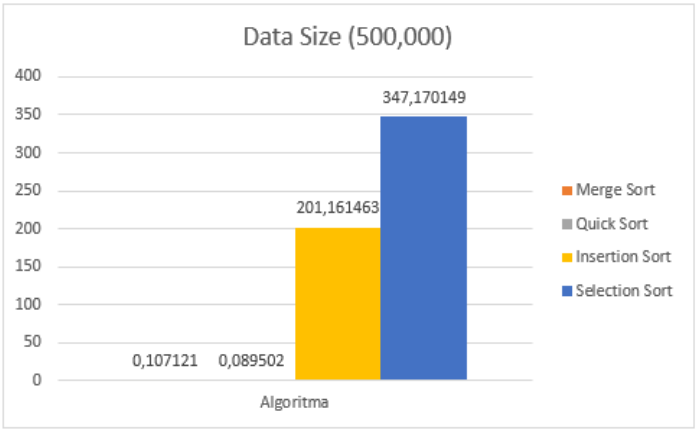


Grafik 4.5: Perbandingan algoritma *sorting* data size 100.000

Pada data size 100.000, perbedaan waktu eksekusi algoritma semakin terlihat. Algoritma yang paling cepat adalah *quick sort* dengan waktu 0,016173 detik. Diikuti dengan *merge sort* dengan waktu yang tidak jauh berbeda, yaitu 0,03897 detik. Kemudian *insertion sort* dengan waktu 7,427448 detik. Terakhir *selection sort* dengan waktu 13,750899 detik.

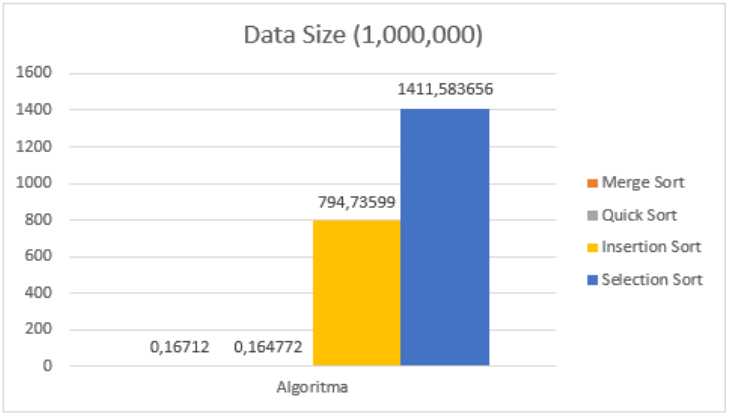
- Data Size 500.000

Pada data size 500.000, perbedaan waktu eksekusi algoritma cukup signifikan. Algoritma yang paling cepat adalah *quick sort* dengan waktu 0,089502 detik. Diikuti dengan *merge sort* dengan waktu yang tidak jauh berbeda, yaitu 0,107121 detik. Kemudian *insertion sort* dengan waktu 201,161463 detik. Terakhir *selection sort* dengan waktu 347,170149 detik.



Grafik 4.6: Perbandingan algoritma *sorting* data size 500.000

- Data Size 1.000.000



Grafik 4.7: Perbandingan algoritma *sorting* data size 1.000.000

Pada data size 1.000.000, perbedaan waktu eksekusi algoritma sudah sangat signifikan. Algoritma yang paling cepat adalah *quick sort* dengan waktu 0,164772 detik. Diikuti dengan *merge sort* dengan waktu yang tidak jauh berbeda, yaitu 0,16712 detik. Kemudian *insertion sort* dengan waktu 794,73599 detik. Terakhir *selection sort* dengan waktu 1411,583656 detik (sekitar setengah jam).

- Analisis

Secara keseluruhan, peringkat kecepatan algoritma adalah sebagai berikut:

1. Quick Sort
2. Merge Sort
3. Insertion Sort
4. Selection Sort

Akan tetapi, perbedaan *quick sort* dan *merge sort* tidak terlalu berarti, dan masih sesuai pada teori yaitu berorde O (n log n). Bahkan untuk data *size* 1.000, waktu eksekusi *merge sort* lebih cepat dari *quick sort*. Hal ini dikarenakan walaupun eksekusi dalam *loop quick sort* lebih cepat dari *merge sort*, tetapi *quick sort* tidak stabil seperti *merge sort*. Sehingga, algoritma *quick sort* sangat bergantung pada *input* data set yang diberikan (sudah terurut, random, dsb). Seiring bertumbuh koleksi data, algoritma

insertion sort dan *selection sort* memiliki waktu eksekusi yang jauh berbeda dengan *merge sort* dan *quick sort*. Hal ini menunjukkan bahwa sesuai teori, kompleksitas waktu untuk *insertion sort* dan *selection sort* adalah $O(n^2)$

Waktu eksekusi(s)	**Algoritma**			
Data	Merge Sort	Quick Sort	Insertion Sort	Selection Sort
1,000	0.000841	0.000872	0.006272	0.008955
5,000	0.004736	0.003785	0.086197	0.041303
10,000	0.009261	0.008027	0.090607	0.141909
50,000	0.016846	0.011051	1.851081	3.416503
100,000	0.03897	0.016173	7.427448	13.750899
500,000	0.107121	0.089502	201.161463	347.170149
1,000,000	0.16712	0.164772	794.73599	1411.583656

Tabel 4.1: Waktu Eksekusi Algoritma seluruh data set

V. KESIMPULAN

- Algoritma yang paling cepat adalah *quick sort*, diikuti dengan *merge sort*, *insertion sort*, lalu *selection sort*.
- Kompleksitas waktu algoritma berguna sebagai *tools* untuk membandingkan antar algoritma *sorting*.

REFERENSI

[1] Introduction to the Design and Analysis of Algorithms, 3rd Ed – Levitin
[2] Matematika Diskrit_ Ed. 3 - Rinaldi Munir
[3] https://algs4.cs.princeton.edu/20sorting/ 2 Desember 2017, 21.20
[4] https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/ 2 Desember 2017, 21.33
[5] http://interactivepython.org/runestone/static/pythonds/AlgorithmAnalysis/BigONotation.html 2 Desember 2017, 22.36
[6] http://bigocheatsheet.com/ 3 Desember 2017, 16.46

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Desember 2017

Emilia Andari Razak
13515056

Lampiran

Source Code untuk *Algoritma Sorting*

```
#include <stdlib.h>
#include <ctime>
#include <iostream>
#include <iomanip>
#include <time.h>

using namespace std;

void swap(int *x, int *y){
	int temp;

	temp=*x;
	*x=*y;
	*y=temp;
}


int Partition(int *A, int l, int r){
	//Prekondisi: 0<=l<r<=n-1
	int i, j, p;

	p=A[l];
	i=l;
	j=r+1;
	do{
		do{
			i++;
		}while(A[i]<p);
		do{
			j--;
		}while(A[j]>p);
		swap(A[i], A[j]);
	}while(i<j);
	swap(A[i], A[j]); //undo last swap
	swap(A[l], A[j]);
	return j;
}
void Merge(int *B, int *C, int *A, int p, int q){
	int i,j,k;

	i=0; j=0; k=0;

	while (i<p && j<q){
		if (B[i]<=C[j]){
			A[k]=B[i];
			i++;
		}
		else{
			A[k]=C[j];
			j++;
		}
		k++;
	}

	if (i==p){
		while(j<q){
			A[k]=C[j];
			k++;
			j++;
		}
	}
	else{
		while(i<p){
			A[k]=B[i];
			k++;
			i++;
		}
	}
}

void SelectionSort(int *A, int n){
	int i, j, min;

	for(i=0;i<(n-1);i++){
		min=i;
		for(j=i+1;j<n;j++){
			if(A[j]<A[min]){
				min=j;
			}
		}
		swap(A[i],A[min]);
	}

}

void InsertionSort(int *A, int n){
	int i,j,v;

	for(i=1;i<n;i++){
		v=A[i];
		j=i-1;
		while(j>=0 && A[j]>v){
			A[j+1]=A[j];
			j--;
		}
		A[j+1]=v;
	}
}
void QuickSort(int *A, int l, int r){
	int s;
	if (l<r){
		s=Partition(A,l,r);
		QuickSort(A,l,(s-1));
```

```
QuickSort(A,(s+1),r);
        }
}


void MergeSort(int *A, int n){
        if (n%2==0){
                int i,j,k;

                int B[n/2];
                int C[n/2];

                i=0;
                j=0;
                k=0;
                while(k<(n/2)){
                        B[i]=A[k];
                        i++;
                        k++;
                }

                while(k<n){
                        C[j]=A[k];
                        j++;
                        k++;
                }
                MergeSort(B,(n/2));
                MergeSort(C,(n/2));
                Merge(B,C,A, (n/2), (n/2));
        }
        else{
                if(n>1){
                        int i,j,k;

                        int B[(n/2)];
                        int C[(n/2)+1];

                        i=0;
                        j=0;
                        k=0;
                        while(k<(n/2)){
                                B[i]=A[k];
                                i++;
                                k++;
                        }

                        while(k<n+1){
                                C[j]=A[k];
                                j++;
                                k++;
                        }
                        MergeSort(B,(n/2));
                        MergeSort(C,(n/2)+1);
                        Merge(B,C,A, (n/2),
(n/2)+1);
                        }
        }
}
```

```
int main(){
        int i, size,l,r;
        double deltaT;
        clock_t t1, t2;

        int *array;

        cin>>size;
        l=0; r=size-1;
        array=new int[size-1];

        srand((unsigned)time(0));

   for(i=0; i<size; i++){
      array[i] = (rand()%100)+1;
        }
        t1 = clock();
        MergeSort(array, size);
        t2 = clock();
        deltaT=t2-t1;

        for(i=0; i<size; i++){
     cout<<array[i]<<" ";
        }
        cout<<endl;
        cout<<"MergeSort:
"<<setprecision(10)<<deltaT/1000000<<endl;
        cout<<endl;
        srand((unsigned)time(0));

   for(i=0; i<size; i++){
      array[i] = (rand()%100)+1;
        }

        t1 = clock();
        QuickSort(array, l,r);
        t2 = clock();
        deltaT=t2-t1;

        for(i=0; i<size; i++){
     cout<<array[i]<<" ";
        }
        cout<<endl;
        cout<<"QuickSort:
"<<setprecision(10)<<deltaT/1000000<<endl;
        cout<<endl;
        srand((unsigned)time(0));
for(i=0; i<size; i++){
      array[i] = (rand()%100)+1;
        }

        t1 = clock();
        InsertionSort(array, size);
        t2 = clock();
        deltaT=t2-t1;

        for(i=0; i<size; i++){
     cout<<array[i]<<" ";
        }
```

```
cout<<endl;
            cout<<"InsertionSort:
"<<setprecision(10)<<deltaT/1000000<<endl;
            cout<<endl;

            srand((unsigned)time(0));

   for(i=0;  i<size;  i++){
       array[i] = (rand()%100)+1;
            }

            t1 = clock();
            SelectionSort(array, size);
            t2 = clock();
            deltaT=t2-t1;

            for(i=0;  i<size;  i++){
     cout<<array[i]<<"  ";
            }
            cout<<endl;
            cout<<"SelectionSort:
"<<setprecision(10)<<deltaT/1000000<<endl;
            cout<<endl;


            delete []array;
            return 0;
}
```