ISSN 2085-4552

Analisis Perbandingan Efisiensi Algoritma *Brute Force* dan *Divide and Conquer* dalam Proses Pengurutan Angka

Fenina Adline Twince Tobing[1], James Ronald Tambunan[2]
[1] Program Studi Informatika, Fakultas Teknik dan Informatika, Universitas Multimedia Nusantara, Tangerang, Indonesia
fenina.tobing@umn.ac.id
[2] Program Studi Manajemen Informatika, AMIK Widya Loka, Medan, Indonesia
jamesronaldtambunan@gmail.com

Diterima 30 April 2020
Disetujui 17 Juni 2020

***Abstract*—Perbandingan algoritma dibutuhkan untuk mengetahui tingkat efisiensi suatu algoritma. Penelitian ini membandingkan efisiensi dari dua strategi algoritma *sort* yang sudah ada yaitu *Brute Force* dan *Divide and Conquer*. Algoritma *Brute Force* yang akan diuji adalah *Bubble Sort* dan *Selection Sort*. Algoritma *Divide and Conquer* yang akan diuji adalah *Quick Sort* dan *Merge Sort*. Cara yang dilakukan dalam penelitian ini adalah melakukan tes dengan data sebanyak 50 sampai 100.000 untuk setiap algoritma. Tes dilakukan dengan menggunakan bahasa pemrograman *JavaScript*. Hasil dari penelitian ini adalah algoritma *Quick Sort* dengan strategi *Divide and Conquer* memiliki efisiensi yang baik serta *running time* yang cepat dan algoritma *Bubble Sort* dengan strategi *Brute* Force memiliki efisiensi yang buruk serta *running time* yang lama.**

***Index Terms*—Algoritma, Bubble Sort, Brute Force, Divide and Conquer, Efisiensi, Merge Sort, Quick Sort, Selection Sort**

I. PENDAHULUAN

Algoritma merupakan hal yang penting dalam menyelesaikan sebuah permasalahan. Dibutuhkan sebuah metode yang tepat karena dapat mempengaruhi hasil yang diinginkan. Sebaik apapun algoritma, jika menghasilkan output yang salah, maka algoritma tersebut bukanlah algoritma yang baik. Sebuah algoritma yang baik adalah algoritma yang efektif, efisien, tepat sasaran dan terstruktur. Hal ini dapat diukur dari waktu eksekusi algoritma (*running time*) dan kebutuhan ruang memori (*memory space*) yang digunakan. Namun, kebutuhan waktu dan ruang dari suatu algoritma bergantung pada jumlah data (*input*) yang ingin diproses dan algoritma yang digunakan.

Algoritma *sorting* adalah salah satu alogitma dasar yang sering digunakan untuk menyelesaikan masalah pengurutan. Pengurutan data atau *sorting* merupakan salah satu jenis operasi penting dalam pengolahan data. Data yang sudah terurut memiliki beberapa keuntungan. Selain mempercepat waktu pencarian, dari data yang terurut dapat langsung diperoleh nilai maksimum dan nilai minimum.

Dalam penyelesaian algoritma *sorting* atau algoritma pengurutan, ada beberapa metode yang dapat digunakan yaitu: *bubble sort*, *selection sort*, *insertion sort*, *merge sort*, *quick sort*, *heap sort*, *counting sort*, *radix sort*, dan *bucket sort* [1]. Banyaknya metode yang dapat digunakan dalam algoritma pengurutan ini belum tentu merupakan hal yang baik, dikarenakan tidak semua jenis metode memiliki hasil *running time* yang baik.

Pada penelitian sebelumnya [2], mengenai perbandingan efisiensi strategi *Brute Force* dengan algoritma *Bubble Sort* dan strategi *Divide and Conquer* dengan algoritma *Quick Sort*, didapatkan bahwa algoritma *Quick Sort* lebih efektif dan efisien dalam menangani masalah pengurutan. Namun, algoritma yang menggunakan strategi *Brute Force* dan juga *Divide and Conquer* tidak hanya *Bubble Sort* dan *Quick Sort* tetapi ada juga *Selection Sort* yang masuk dalam strategi *Brute Force* dan *Merge Sort* yang masuk dalam strategi *Divide and Conquer*.

Berdasarkan hal tersebut, akan dianalisa strategi algoritma mana yang efektif dan efisien dalam menyelesaikan masalah pengurutan angka dengan menggunakan strategi algoritma *Brute Force* dan *Divide and Conquer*.

II. LANDASAN TEORI

A. *Algoritma Sorting*

Algoritma *Sorting* adalah kumpulan langkah-langkah dalam menyelesaikan masalah dengan suatu metode tertentu. *Sorting* didefinisikan sebagai proses pengurutan sejumlah data yang disusun secara acak menjadi terurut dan teratur. Pengurutan ini terbagi menjadi dua yaitu *ascending* dan *descending*.

B. *Kompleksitas Algoritma*

Kompleksitas suatu algoritma merupakan ukuran seberapa banyak komputasi yang dibutuhkan algoritma tersebut untuk mendapatkan hasil yang diinginkan. Hal-hal yang mempengaruhi kompleksitas waktu [3]:

1. Jumlah masukan data untuk suatu algoritma (n).
2. Waktu yang dibutuhkan unutk menjalankan algoritma tersebut.
3. Ruang memori yang dibutuhkan untuk menjalankan algoritma.

Kompleksitas mempengaruhi performa atau kinerja dari suatu algoritma. Kompleksitas dibagi menjadi 3 jenis [4] yaitu:

- Kompleksitas kasus terburuk (*worst case*) dari algoritma adalah fungsi yang ditentukan oleh jumlah maksimum langkah-langkah yang diambil pada setiap *instance* dari ukuran *n*. Hal ini mewakili kurva melewati titik tertinggi dari setiap kolom.
- Kompleksitas kasus terbaik (*best case*) dari algoritma adalah fungsi yang didefinisikan oleh jumlah minimum langkah yang diambil pada setiap *instance* ukuran *n*. Hal ini mewakili kurva melewati titik terendah dari setiap kolom.
- Kompleksitas kasus rata-rata (*average* case) dari algoritma adalah fungsi yang didefinisikan oleh jumlah rata-rata langkah yang diambil pada setiap *instance* dari ukuran *n*.

Masing-masing jenis kompleksitas ini menunjukkan kecepatan atau waktu yang dibutuhkan algoritma untuk mengeksekusi sejumlah kode.

Ada 2 macam kompleksitas algoritma [5], yaitu kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu disimbolkan dengan *T(n)* dan kompleksitas ruang *S(n)*. Kompleksitas waktu, *T(n)*, diukur dari jumlah tahapan komputasi yang dibutuhkan untuk menjalankkan algoritma sebagai fungsi dari ukuran masukan *n*. Komplesitas ruang, *S(n)*, diukur dari memori yang digunakan oleh struktur data yang terdapat di dalam algoritma sebagai fungsi dari ukuran masukan *n*.

C. *Kompleksitas Waktu*

Kompleksitas waktu [6] adalah konsep dalam ilmu komputer yang berkaitan dengan kuantifikasi jumlah waktu yang diambil oleh seperangkat kode atau algoritma untuk memproses atau menjalankan fungsi dari jumlah input.

Analisis kompleksitas waktu algoritma [7] adalah membandingkan waktu yang dibutuhkan algoritma dalam menyelesaikan perintah. Pada penelitian Estrada,A.H [8] menyebutkan bahwa dengan menganalisis kompleksitas waktu, dapat disimpulkan bahwa banyaknya jumlah data-*n* berpengaruh terhadap kebutuhan waktu yang diperlukan.

Pada algoritma pengurutan terutama pada pengurutan dengan perbandingan, operasi dasarnya adalah operasi-operasi perbandingan elemen-elemen suatu larik dan operasi pertukaran elemen. Kedua hal itu dihitung terpisah karena jumlah keduanya tidaklah sama. Kompleksitas algoritma dinyatakan secara *asimptotik* dengan notasi *big-O*. Jika kompleksitas waktu untuk menjalankan suatu algoritma dinyatakan dengan *T(n)* dan memenuhi $T(n) \leq C(f(n))$ untuk $n \geq n0$, maka kompleksitas dapat dinyatakan dengan $T(n) = O(f(n))$ [9].

D. *Brute Force*

Brute Force adalah teknik paling sederhana untuk menyelesaikan permasalahan komputasi pada umumnya. Secara konseptual, *Brute Force* bekerja sebagai berikut [10]:

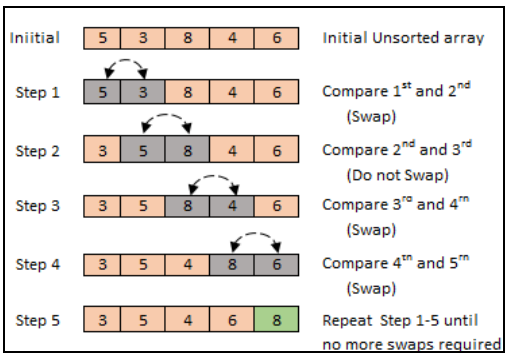
1. Mula-mula *pattern* dicocokkan pada awal teks
2. Dengan bergerak dari kiri ke kanan, bandingkan setiap karakter di dalam *pattern* dengan karakter yang bersesuaian di dalam teks sampai:
   - Semua karakter yang dibandingkan cocok atau sama (pencarian berhasil), atau
   - Dijumpai sebuah ketidakcocokan karakter (pencarian belum berhasil)
3. Bila *pattern* belum ditemukan kecocokannya dan teks belum habis, geser *pattern* satu karakter ke kanan dan ulangi langkah 2.

Algoritma yang menggunakan strategi *Brute Force* adalah algoritma *Bubble Sort* dan *Selection Sort*.

D.1 *Bubble Sort*

D.2.1 Konsep *Bubble Sort*

Bubble sort [11] adalah metode pengurutan yang membandingkan elemen yang sekarang dengan elemen berikunya, jika elemen sekarang > elemen berikutnya maka posisinya ditukar, kalau tidak, tidak perlu ditukar, misalnya untuk n = 7 maka akan dilakukan (n – 1) = 6 tahap (mulai dari 0 sampai dengan n - 2).



Gambar 1. Contoh simulasi *bubble sort*

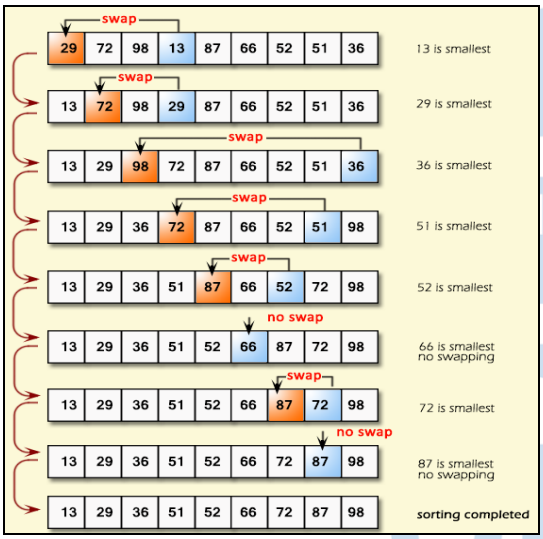
D.2.2 Kompleksitas *Bubble Sort*

Kompleksitas waktu rata-rata dari *Bubble Sort* adalah $O(n^2)$ [12]. Untuk *worst case*, algoritma ini memiliki kompleksitas $O(n^2)$ [13] saat data yang akan diurutkan diurutkan terbalik terlebih dahulu. Untuk *best case*, algoritma ini memiliki kompleksitas *O(n)* [13] disaat data sudah diurutkan terlebih dahulu.

D.2 *Selection Sort*

D.2.1 Konsep *Selection Sort*

Selection Sort [11] adalah metode pengurutan yang membandingkan elemen yang sekarang dengan elemen berikutnya sampai ke elemen yang terakhir. Jika ditemukan elemen lain yang lebih kecil dari elemen sekarang maka posisinya dicatat dan langsung ditukar.



Gambar 2. Contoh simulasi *selection sort*

D.2.2 Kompleksitas *Selection Sort*

Selection Sort memiliki kompleksitas waktu rata-rata, *worst case*, dan *best case* yang sama, yaitu $O(n^2)$ [14] dikarenakan algoritma *sort* ini memilki dua *nested loop*.

E. Divide and Conquer

Divide and Conquer adalah metode penyelesaian masalah dengan membagi masalah utama menjadi masalah yang lebih kecil. Pada strategi ini, terdapat 3 bagian utama dalam menyelesaikan suatu masalah yaitu *divide*, *conquer*, dan *combine*.

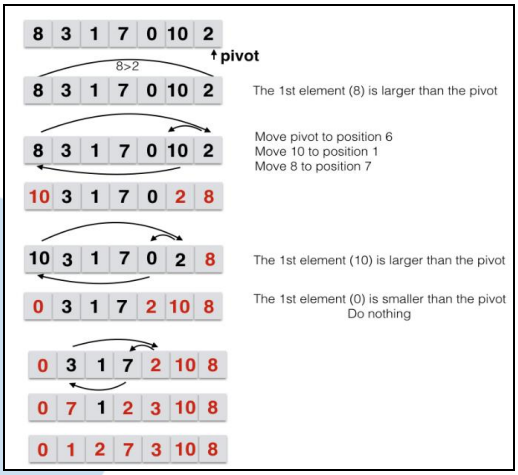
Algoritma yang menggunakan strategi *Divide and Conquer* adalah algoritma *Quick Sort* dan *Merge Sort*.

E.1 *Quick Sort*

E.1.1 Konsep *Quick Sort*

Quick Sort [15] adalah sebuah algoritma sorting dari model *Divide and Conquer* yaitu dengan cara mereduksi tahap demi tahap sehingga menjadi 2 bagian yang lebih kecil.

Algoritma *Quick Sort* melakukan *sorting* dengan membagi masalah menjadi sub masalah dan sub masalah dibagi lagi menjadi sub-sub masalah sehingga *sorting* tersebut menjadi lebih cepat walaupun memakan ruang memori yang besar.



Gambar 3. Contoh simulasi *quick sort*

E.1.2 Kompleksitas *Quick Sort*

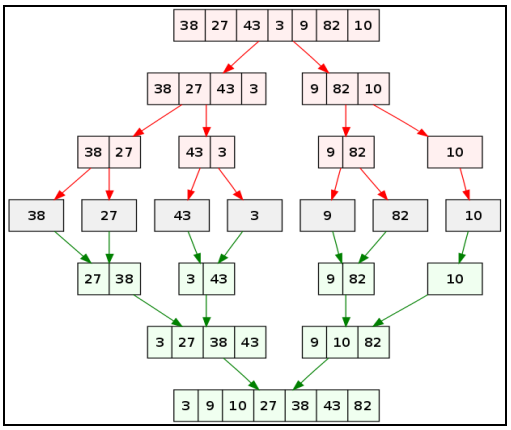
Algoritma pengurutan *Quick Sort* memiliki kompleksitas waktu rata-rata dan *best case O(n log n)* [16]. Untuk *worst case*, kompleksitas waktu dari *Quick Sort* ini adalah $O(n^2)$ yang dimana terjadi apabila data sudah terurut dan *pivot* yang dipilih adalah data yang pertama [16].

E.2 *Merge Sort*

E.2.1 Konsep *Merge Sort*

Secara konseptual, sebuah *array* berukuran *n*, *Merge Sort* bekerja sebagai berikut [17] :

1. Jika bernilai 0 atau 1, maka *array* sudah terurut.
2. Bagi *array* yang tidak terurut menjadi dua *subarray*, masing-masing berukuran n/2.
3. Urutkan setiap sub-array. Jika sub-array tidak cukup kecil, lakukan rekursif langkah 2 terhadap *sub-array*.
4. Menggabungkan dua *sub-array* kembali menjadi satu *array* yang terurut.



Gambar 4. Contoh simulasi *merge sort*

E.2.2 Kompleksitas *Merge Sort*

Algoritma ini memiliki kompleksitas waktu rata-rata, *worst case*, dan *best case* yang sama yaitu *O(n log n)* [14].

III. HASIL DAN PEMBAHASAN

Pada penelitian ini, digunakan beberapa jumlah data yaitu 50, 100, 500, 1.000, 2.500, 5.000, 10.000, 25.000, 50.000, 100.000 dengan *range* 1.000.000 dan diukur menggunakan besaran *millisecond* dengan 3 angka dibelakangan koma agar terlihat perbedaannya. Data yang digunakan berlaku untuk semua algoritma *sort* yang dibahas, dan bahasa pemograman yang digunakan adalah *JavaScript*. Hasil *running time* tiap algoritma bergantung pada kondisi dan jenis laptop atau komputer yang digunakan.

A. *Bubble Sort*

Gambar 5 merupakan potongan *source code JavaScript* dari algoritma *Bubble Sort*:

```
let bubbleSort = (data) => {
  let flag;
  let n = data.length - 1;
  let x = data;
  do {
    flag = false;
    for (var i = 0; i < n; i++) {
      if (x[i] > x[i + 1]) {
        var temp = x[i];
        x[i] = x[i + 1];
        x[i + 1] = temp;
        flag = true;
      }
    }
    n--;
  } while (flag);
  return x;
};
```

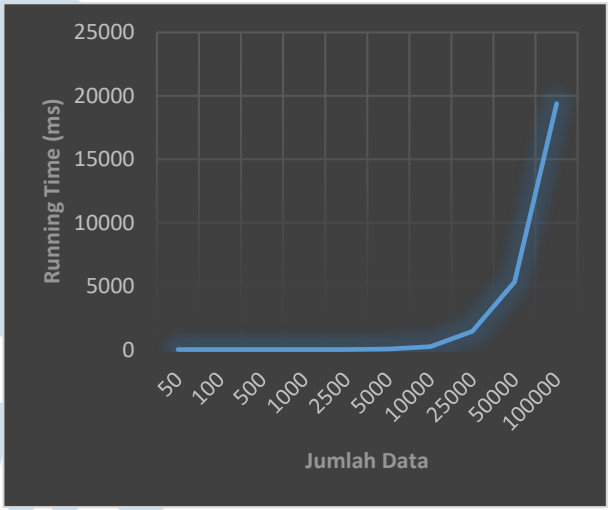
Gambar 5. Potongan *source code Bubble Sort*

Hasil waktu eksekusi dari algoritma *Bubble Sort* dapat dilihat pada Tabel 1.

Tabel 1. Waktu eksekusi algoritma *Bubble Sort*

Data	*Running Time* (*ms*)
50	0.24
100	0.425
500	1.98
1.000	2.825
2.500	9.46
5.000	42.4
10.000	232.175
25.000	1443.36
50.000	5330.995
100.000	19379.27

Kompleksitas algoritma *Bubble Sort* dapat digambarkan seperti grafik berikut.



Gambar 6. Grafik kompleksitas algoritma *Bubble Sort*

B. *Selection Sort*

Gambar 7 merupakan potongan *source code JavaScript* dari algoritma *Selection Sort*:

```
let selectionSort = (data) => {
  let len = data.length;
  for (let i = 0; i < len; i++) {
    let min = i;
    for (let j = i + 1; j < len; j++) {
      if (data[min] > data[j]) {
        min = j;
      }
    }
    if (min !== i) {
      let tmp = data[i];
      data[i] = data[min];
      data[min] = tmp;
    }
  }
  return data;
};
```

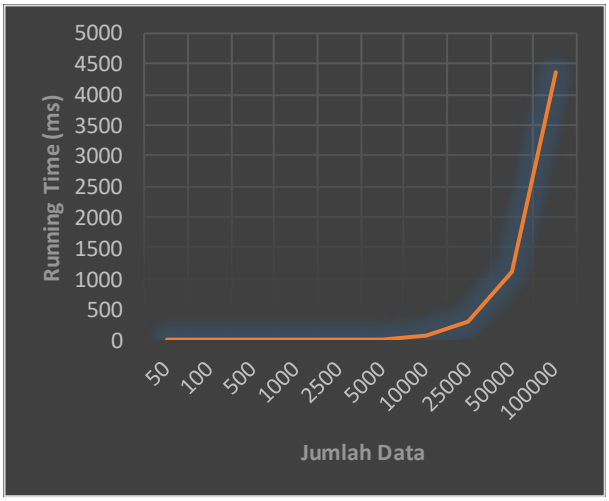
Gambar 7. Potongan *source code Selection Sort*

Hasil waktu eksekusi dari algoritma *Selection Sort* dapat dilihat pada Tabel 2.

Tabel 2. Waktu eksekusi algoritma *Selection Sort*

Data	*Running Time* (*ms*)
50	0.105
100	0.23
500	1.36
1.000	1.19
2.500	2.36
5.000	10.025
10.000	49.48
25.000	284.925
50.000	1109.13
100.000	4354.09

Kompleksitas algoritma *Selection Sort* dapat digambarkan seperti grafik berikut.



Gambar 5. Grafik kompleksitas algoritma *Selection Sort*

C. *Quick Sort*

Gambar 9 merupakan potongan *source code JavaScript* dari algoritma *Quick Sort*:

```
let quickSort = (items, left, right) => {
  let index;
  if (items.length > 1) {
    index = partition(items, left, right);
    if (left < index - 1) {
      quickSort(items, left, index - 1);
    }
    if (index < right) {
      quickSort(items, index, right);
    }
  }
  return items;
};
```

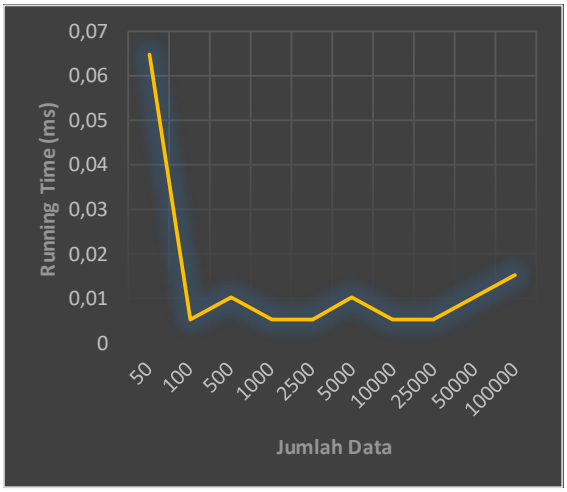
Gambar 9. Potongan *source code Quick Sort*

Hasil waktu eksekusi dari algoritma *Quick Sort* dapat dilihat pada Tabel 3.

Tabel 1. Waktu eksekusi algoritma *Quick Sort*

Data	*Running Time* (*ms*)
50	0.065
100	0.005
500	0.01
1.000	0.005
2.500	0.005
5.000	0.01
10.000	0.005
25.000	0.005
50.000	0.01
100.000	0.015

Kompleksitas algoritma *Quick Sort* dapat digambarkan seperti grafik berikut.



Gambar 10. Grafik kompleksitas algoritma *Quick Sort*

D. *Merge Sort*

Gambar 11 merupakan potongan *source code JavaScript* dari algoritma *Merge Sort*:

```
let mergeSort = (unsortedArray) => {
  if (unsortedArray.length <= 1) {
    return unsortedArray;
  }
  const middle = Math.floor(unsortedArray.length / 2);

  const left = unsortedArray.slice(0, middle);
  const right = unsortedArray.slice(middle);

  return merge(mergeSort(left), mergeSort(right));
};
```

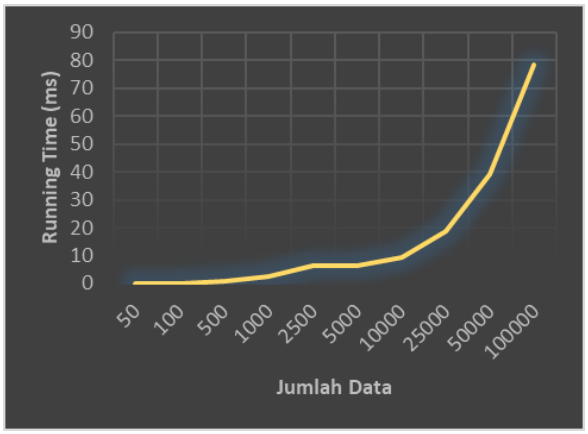
Gambar 11. Potongan *source code Quick Sort*

Hasil waktu eksekusi dari algoritma *Merge Sort* dapat dilihat pada Tabel 4.

Tabel 4. Waktu eksekusi algoritma *Merge Sort*

Data	*Running Time* (ms)
50	0.15
100	0.18
500	0.81
1.000	2.665
2.500	6.265
5.000	6.3
10.000	9.18
25.000	18.815
50.000	39.265
100.000	78.525

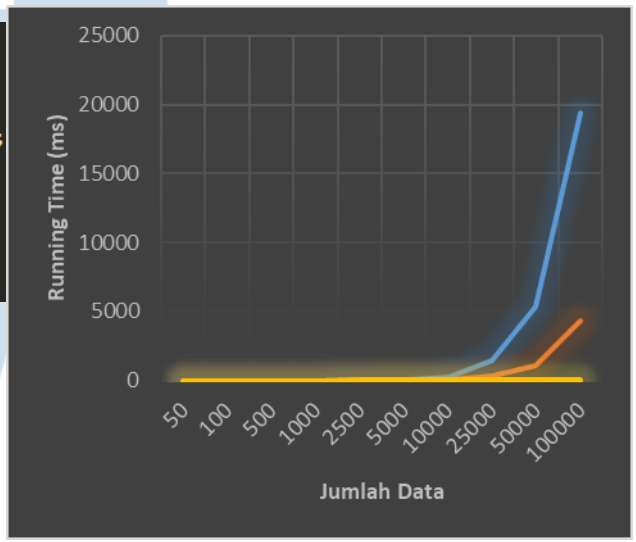
Kompleksitas algoritma *Merge Sort* dapat digambarkan seperti grafik berikut.



Gambar 12. Grafik kompleksitas algoritma *Merge Sort*

E. *Perbandingan Algoritma Bubble Sort, Selection Sort, Quick Sort dan Merge Sort*

Grafik dibawah ini merupakan hasil perbandingan efisiensi strategi *Brute Force* dan *Divide and Conquer*.



Gambar 13. Grafik perbandingan kompleksitas algoritma

IV. SIMPULAN

Berdasarkan dari hasil algoritma pengurutan yang dilakukan menggunakan strategi *Brute Force* dan *Divide and Conquer*, dapat disimpulkan bahwa strategi *Divide and Conquer* lebih efektif dan efisien dalam menangani masalah pengurutan. Algoritma yang mudah dalam hal implementasi adalah *Bubble Sort* dan *Selection Sort*, keduanya memiliki kompleksitas $O(n^2)$. Algoritma yang lebih efisien adalah algoritma *Quick Sort* dan *Merge Sort* dengan

kompleksitasnya adalah *O(n log n)*. Dari keempat algoritma ini, yang memiliki efisiensi paling bagus adalah algoritma *Quick Sort* dengan *running time* rata-rata sebesar 0,0135 *ms* dari data yang diuji. Sedangkan *bubble sort* merupakan algoritma yang memiliki efisiensi paling buruk dari keempat algoritma yang diuji dengan *running time* rata-rata sebesar 2644,313 *ms* dari data yang diuji.

DAFTAR PUSTAKA

[1] "Sorting Algorithms," [Online]. Available: https://guide.freecodecamp.org/algorithms/sorting-algorithms/. [Accessed 20 Mei 2020].

[2] Amran. "Perbandingan Strategi Brute Force dan Divide and Conquer Pada Algoritma Pengurutan". https://www.academia.edu/39080158/Perbandingan_Strategi_Brute_Force_and_Divide_an_Conquer_Pada_Algoritma_Pengurutan. [Accessed 20 Mei 2020].

[3] Wikipedia, http://en.wikipedia.org/wiki/ Big_O_notation/ ". [Accessed 20 Mei 2020].

[4] "Best, Worst, and Average-Case Complexity," [Online]. Available: https://www8.cs.umu.se/kurser/TDBA77/VT06/algorithms/BOOK/BOOK/NODE13.HTM. [Accessed 22 Mei 2020].

[5] "Algorithm Efficiency," [Online]. Available: http://www.cs.kent.edu/~durand/CS2/Notes/03_Algs/ds_alg_efficiency.html. [Accessed 22 Mei 2020].

[6] "Time Complexity," [Online]. Available: https://www.techopedia.com/definition/22573/time-complexity. [Accessed 22 Mei 2020].

[7] D. W. Nugraha, "Penerapan Kompleksitas Waktu Algoritma Prim Untuk Menghitung Kemampuan Komputer Dalam Melaksanakan Perintah," vol. 2, 2012.

[8] A. H. Estrada S, "Telaah Waktu Eksekusi Program Terhadap Kompleksitas Waktu Algoritma Brute Force Dan Divide And Conquer Dalam Penyelesaian Operasi List," vol. 3, 2003.

[9] S. N. B. Tjaru, "Kompleksitas Algoritma Pengurutan Selection Sort dan Insertion Sort". MAKALAH IF2091 STRATEGI ALGORITMIK, 2009.

[10] P. Triono, "Analisis Perbandingan Algoritma Sorting dan Searching". https://www.academia.edu/6526802/Analisis-perbandingan-algoritma, 2010. [Accessed 22 Mei 2020].

[11] S. Y. Yahya, "Analisa Perbandingan Algoritma Bubble Sort dan Selection Sort Dengan Metode Perbandingan Eksponensial," *Pelita Informatika Budi Darma*, vol. VI, 2014.

[12] K. Ali, "A Comparative Study of Well Known Sorting Algorithms," *International Journal of Advanced Research in Computer Science*, vol. 8, no. 1, pp. 277-280, 2017.

[13] M. Usman, M. Afzal and Z. Bajwa, "Performance Analysis of Sorting Algorithms with C#," *International Journal for Research in Applied Science & Engineering*, vol. 3, no. 1, pp. 201-204, 2015.

[14] D. Rajagopal and K. Thilakavalli, "Different Sorting Algorithm's Comparison based Upon the," *International Journal of u- and e- Service, Science and Technology*, vol. 9, no. 8, pp. 287-296, 2016.

[15] N. D. S. L. S. P. Parag Bhalchandra, "A Comprehensive Note on Complexity Issues in Sorting Algorithms," *Advances in Computational Research*, vol. 1, no. 2, pp. 1-9, 2009.

[16] M. E. Wira Putra, "Perbandingan Algoritma Pengurutan Merge Sort, Quick Sort dan Heap Sort Dilihat dari Kompleksitasnya".https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2009-2010/Makalah0910/MakalahStrukdis0910-023.pdf [Accessed 22 Mei 2020].