**Analisis Kompleksitas Algoritma 5 Macam Pengurutan Nilai (*Sorting*)**

**Dinda Atikah Wulandari | Najla Nur Adila**

1301190156 | 1301194001

[Dindaatikahwulandari@student.telkomuniversity.ac.id](mailto:Dindaatikahwulandari@student.telkomuniversity.ac.id) | [najlanuradila@student.telkomuniversity.ac.id](mailto:najlanuradila@student.telkomuniversity.ac.id)

### ABSTRAK

**Peran algoritma dalam pemrograman sangat penting, sehingga perlu untuk memahami konsep dasar dari algoritma. Begitu banyak logika pemrograman yang telah diciptakan, untuk kasus yang umum dan juga khusus. seiring berkembangnya kemajuan di bidang informatika dan teknologi, tuntutan untuk menemukan metode pemecahan masalah secara lebih tepat, efektif dan kuat menjadi sebuah kebutuhan. Dalam menyelesaikan sebuah masalah memerlukan algoritma penyelesaian seperti algoritma pengurutan (*Sorting*). Algoritma *Sorting* (pengurutan) memiliki banyak macam, seperti *selection sort* (pengurutan dengan memilih), *insertion sort* (pengurutan dengan memilah), *bubble sort* (pengurutan dengan membandingkan 2 item), *quick sort* (pengurutan dengan prinsip rekursif ), dan *merge sort* (pengurutan dengan *devide-and-conquer*). Masing-masing jenis algoritma memiliki berbagai tingkat efektivitas. Efektivitas algoritma dapat diukur dengan berapa banyak waktu dan ruang yang diperlukan untuk menjalankan algoritma. Namun tidak ada algoritma terbaik untuk semua keadaan, kadang kala sebuah algoritma sangat efisien ketika jumlah datanya sedikit, namun kinerjanya berkurang ketika jumlah data ditambahkan atau meningkat. Pada penelitian ini akan dianalisa perbandingan algoritma pengurutan data, yaitu *bubble sort, selection sort, insertion sort, merge sort*, dan *quick sort*.**

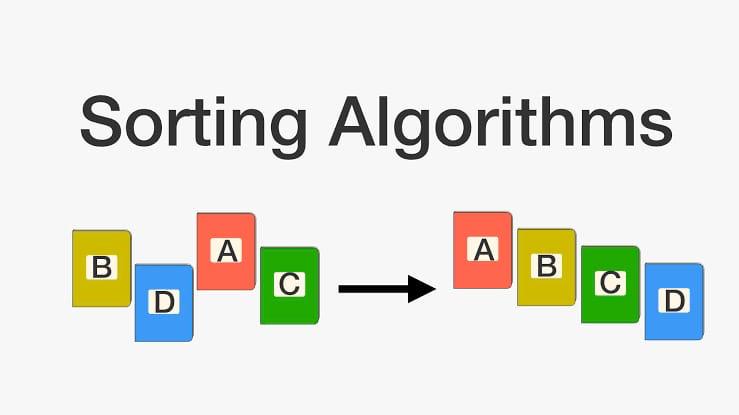
**Kata kunci:** Analisis Kompleksitas Algoritma, *Selection Sort, Insertion Sort, Bubble Sort, Quick Sort, Merge Sort.*

### 1. PENDAHULUAN

Kemajuan teknologi telah memberikan pengaruh terhadap perkembangan data. Data menjadi lebih besar dan bervariasi. Dengan meningkatnya jumlah data maka pengolahan data menjadi lebih kompleks. Sebelum dilakukan pengolahan data, ada proses-proses yang diakukan diantaranya yaitu pengurutan. Pengurutan data atau sorting merupakan salah satu jenis operasi penting dalam pengolahan data. Pengurutan data sangat penting digunakan, sehingga sampai saat ini telah banyak metode-metode pengurutan data dan mungkin akan tetap bermunculan metode-metode baru. Pengurutan data memegang peranan penting yang banyak dipertimbangkan agar keseluruhan permasalahan (terutama mengenai pengolahan data) menjadi lebih baik dan lebih cepat untuk diselesaikan, sehingga menghasilkan data yang akurat. Semakin efisien suatu algoritma, maka pada saat dieksekusi dan dijalankan akan menghabiskan waktu yang lebih cepat dan bisa menerima lebih banyak masukan dari *user*.

Pengurutan data *(sorting)* didefinisikan sebagai proses untuk menyusun kembali himpunan objek dengan menggunakan aturan tertentu. Tujuannya adalah untuk mendapatkan kemudahan dalam pencarian data dari suatu himpunan. Kelebihan suatu data yang terurut adalah mudah untuk dicek apabila ada data yang hilang. Algoritma sorting banyak digunakan untuk menyelesaikan kasus komputasi yang sederhana yang terdapat pada contoh sehari-hari. Sebagai contoh algoritma sorting digunakan ketika akan mengurutkan data masyarakat berdasarkan Nomor Induk Kependudukan (NIK).

Kompleksitas dari suatu algoritma merupakan ukuran seberapa banyak komputasi yang dibutuhkan algoritma tersebut untuk menyelesaikan masalah. Secara informal, algoritma yang dapat menyelesaikan suatu permasalahan dalam waktu yang singkat memiliki kompleksitas yang rendah, sementara algoritma yang membutuhkan waktu yang lama untuk menyelesaikan masalahnya mempunyai kompleksitas yang tinggi. Kompleksitas algoritma terdiri dari dua macam yaitu kompleksitas waktu T(*n)* dan kompleksitas ruang S(*n*).



**Gambar 1. Gambar Simulasi Sorting**

### 2. METODE SORTING

* 1. **ALGORITMA SORTING *SELECTION SORT***
     1. **Konsep *Selection Sort***

Hal pertama yang akan dilakukan *selection sort* yaitu menemukan elemen paling terkecil dan menukarnya dengan elemen di posisi pertama dalam *array*, dan algoritma terus mengulangi penukaran elemen sampai dengan elemen di posisi terakhir. Secara efisien algoritma akan membagi list menjadi 2 bagian, yaitu bagian list dengan elemen yang sudah diurutkan dan bagian list dengan elemen yang akan diurutkan.

* + 1. **Algoritma *Selection Sort* dalam Bahasa *C++***

int main(){

int a, k, c, d, g ;

int size = (dataset);

int max = (dataset);

int b[size];

std::srand(std::time(nullptr));

for (int &v : b ) v = std::rand() % max + 1;

cout<<"Selection sort"<<endl;

cout<<"mengurutkan nilai dari besar ke kecil"<<endl;

for(a=0; a<size; a++){

cout<<b[a]<<endl;

}

for(a=0; a<size-1; a++){

c=a;

for(d=a+1; d<size; d++){

if(b[c]<b[d]){

c=d;

}

}

g =b[c];

b[c] = b[a];

b[a] = g;

}

cout<<"\n setelah diurutkan akan menjadi : \n";

for(a=0; a<size; a++){

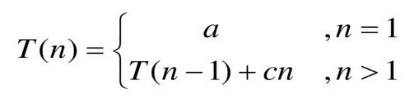
cout<<b[a]<<" \n";

}

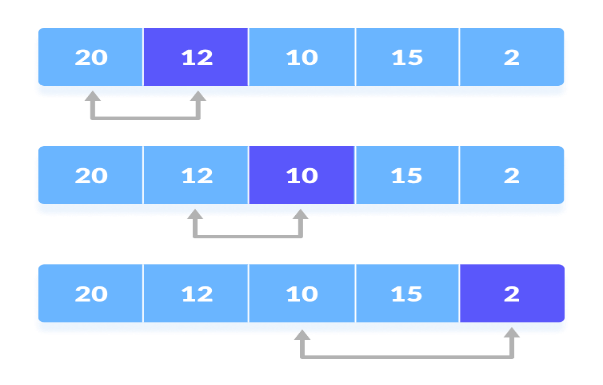
}

* + 1. **Kompleksitas *Selection Sort***

T(n) = (n - 1) + (n - 2) + (n - 3) + ..... + 2 + 1 = = n(n - 1) / 2 = O(n²)



* + 1. **Contoh penerapan**



* 1. **ALGORITMA SORTING *INSERTION SORT***
     1. **Konsep *Insertion Sort***

*Insertion sort* memiliki cara kerja algoritma yaitu pengurutan dengan menyisipkan masing-masing nilai di tempat yang sesuai diantara elemen yang lebih kecil atau sama dengan nilai tersebut.

* + 1. **Algoritma *Insertion Sort* dalam Bahasa *C++***

int main() {

int b, a;

int size = 2000000;

int max = 2000000;

int arr[size];

std::srand(std::time(nullptr));

for (int &v : arr ) v = std::rand() % max + 1;

cout<<"\tINSERTION SORT\n\n";

for(a=0; a<size; a++){

cout<<arr[a]<<endl;

}

insertion\_sort(arr, size);

return 0;

}

void insertion\_sort(int arr[], int size) {

int i, j ,tmp;

for (i = 1; i < size; i++) {

j = i;

while (j > 0 && arr[j - 1] > arr[j]) {

tmp = arr[j];

arr[j] = arr[j - 1];

arr[j - 1] = tmp;

j--;

}

}

print\_array(arr, size);

}

void print\_array(int arr[], int size){

cout<< "Pengurutan : ";

int j;

for (j=0; j<size; j++)

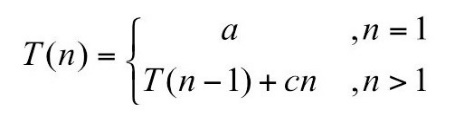
cout <<" "<< arr[j];

cout << endl;

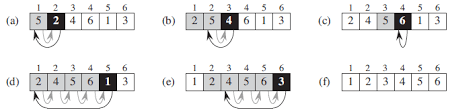
}

* + 1. **Kompleksitas *Insertion Sort***

T(n) = 1 + 2 + … + n – 1 = = n(n - 1) / 2 = O(n²)



* + 1. **Contoh penerapan**



* 1. **ALGORITMA SORTING *BUBBLE SORT***
     1. **Konsep *Bubble Sort***

*Bubble sort* merupakan sebuah teknik pengurutan data dengan cara menukar dua data yang bersebelahan jika urutan dari data tersebut salah.Algoritma ini tidak cocok untuk set data dengan jumlah besar karena kompleksitas dari algoritma ini adalah *O(n)* dimana n adalah jumlah item.

* + 1. **Algoritma *Bubble Sort* dalam Bahasa *C++***

int main(){

const int size = (dataset);

const int max\_v = (dataset);

int array[size] {};

std::srand( std::time(nullptr) );

for( int& v : array ) v = std::rand() % max\_v + 1 ;

std::cout << "unsorted array:\n\t" ;

print\_array( array, size ) ;

sort\_array( array, size ) ;

std::cout << "sorted array:\n\t" ;

print\_array( array, size ) ;

}

void print\_array( const int array[], int size ){

if( array == nullptr ) return ;

for( int i = 0 ; i < size ; ++i ) std::cout << array[i] << ' ' ;

std::cout << '\n' ;

}

void swap\_items( int array[], int a, int b ) {

const auto temp = array[a];

array[a] = array[b];

array[b] = temp;

}

bool bubble\_up( int array[], int size ){

bool swapped = false ;

for( int i = 0; i < (size-1); ++i ){

if( array[i] > array[i+1] ){

swap\_items( array, i, i+1 ) ;

swapped = true ;

}

}

return swapped ;

}

void sort\_array( int array[], int size ){

if( array == nullptr || size < 2 ) return ;

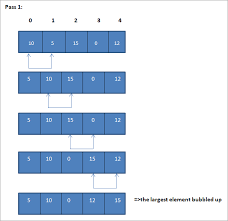
while( bubble\_up( array, size ) ) ;

}

* + 1. **Kompleksitas *Bubble Sort***

Kompleksitas Algoritma *Bubble Sort* dapat dilihat dari beberapa jenis kasus, yaitu *worst-cas*, *average-case*, dan *best-case*.

* Dalam kasus *best-case*, data yang akan disorting telah terurut sebelumnya, sehingga proses perbandingan hanya dilakukan sebanyak (n-1) kali, dengan satu kali *pass*. Persamaan *Big-O* yang diperoleh dari proses ini adalah O(n).
* Dalam kasus *worst-case*, data terkecil berada pada ujung *array*, setiap kali melakukan satu *pass*, data terkecil akan bergeser kearah awal sebanyak satu step. Sehingga jumlah proses pada kondisi *best-*case dapat dirumuskan sebagai berikut: Jumlah proses = n² + n
* Dalam kasus *avarange-case*, jumlah pass ditentukan dari elemen yang mengalami pergeseran ke kiri paling banyak. Jumlah proses perbandingan dapat dihitung sebagai berikut: Jumlah proses = x² + x
  + 1. **Contoh penerapan**



* 1. **ALGORITMA SORTING *QUICK SORT***
     1. **Konsep *Quick Sort***

*Quick Sort* merupakan suatu algoritma pengurutan data yang menggunakan teknik pemecahan data menjadi list berbeda. Algoritma ini mengambil salah satu elemen secara acak yang disebut dengan pivot lalu menyimpan semua elemen yang lebih kecil disebelah kiri pivot dan semua elemen yang lebih besar disebelah kanan pivot.

* + 1. **Algoritma *Quick sort* dalam Bahasa *C++***

int main(){

int i, n ;

int size = (dataset);

int max = (dataset);

int data[size];

std::srand(std::time(nullptr));

for (int &v : data ) v = std::rand() % max + 1;

cout<<"\nData sebelum diurutkan: "<<endl;

for(i=0;i<size;i++){

cout<<data[i]<<" ";

}

cout<<"\n";

quick\_sort(data,0,size-1);

cout<<"\nHasil pengurutan:\n";

for (i=0;i<size;i++){

cout<<data[i]<<" ";

}

}

void quick\_sort(int arr[], int left, int right){

int i = left, j = right; int tmp;

int pivot = arr[(left+right)/2];

while (i<j){

while (arr[i] < pivot)

i++;

while (arr[j] > pivot)

j--;

if (i<=j){

tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

i++;j--;

}

}

if (left < j)

quick\_sort(arr, left, j);

if (i < right)

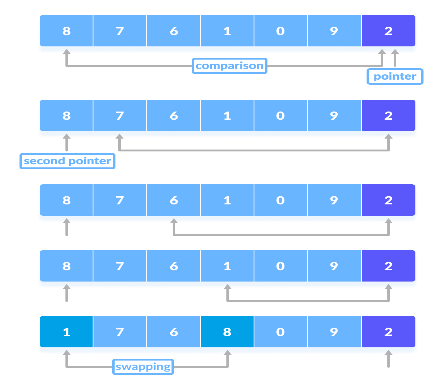
quick\_sort(arr, i, right);

}

* + 1. **Kompleksitas *Quick Sort***

Efisiensi algoritma sangat dipengaruhi elemen mana yang dipilih sebagai *pivot*. Pemilihan *pivot* secara acak disarankan jika data yang akan diurutkan tidak random. Selama *pivot* dipilih secara acak, *quick sort* memiliki kompleksitas algoritma O(n log n).

* + 1. **Contoh Penerapan**



* 1. **ALGORITMA SORTING *MERGE SORT***
     1. **Konsep *Merge Sort***

Algoritma pengurutan data *merge sort* dilakukan dengan menggunakan cara *divide and conquer* yaitu dengan memecah kemudian menyelesaikan setiap bagian kemudian menggabungkannya kembali. Algoritma dirumuskan dalam 3 langkah berpola *divide-and-conquer*. Berikut menjelaskan langkah kerja dari *Merge sort*:

* *Divide*, Memilah elemen – elemen dari rangkaian data menjadi dua bagian.
* *Conquer*, *Conquer* setiap bagian dengan memanggil prosedur *merge sort* secara rekursif
* Kombinasi, Mengkombinasikan dua bagian tersebut secara rekursif untuk mendapatkan rangkaian data berurutan
  + 1. **Algoritma *Merge Sort* dalam Bahasa *C++***

int size = (dataset);

int main(){

int i;

int max = (dataset);

int arr[size];

std::srand(std::time(nullptr));

for (int &v : arr ) v = std::rand() % max + 1;

cout<<"MERGE SORT\n\n";

merge\_sort(arr, 0, size-1);

for (int i=0; i<size; i++){

cout<<arr[i]<<" ";

}

return 0;

}

void merge\_sort(int \*arr, int low, int high){

int mid;

if (low < high){

mid=(low+high)/2;

merge\_sort(arr,low,mid);

merge\_sort(arr,mid+1,high);

merge(arr,low,high,mid);

}

}

void merge(int \*arr, int low, int high, int mid){

int i, j, k, c[size];

i = low;

k = low;

j = mid + 1;

while (i <= mid && j <= high){

if (arr[i] < arr[j]){

c[k] = arr[i];

k++;

i++;

}

else{

c[k] = arr[j];

k++;

j++;

}

}

while (i <= mid){

c[k] = arr[i];

k++;

i++;

}

while (j <= high){

c[k] = arr[j];

k++;

j++;

}

for (i = low; i < k; i++){

arr[i] = c[i];

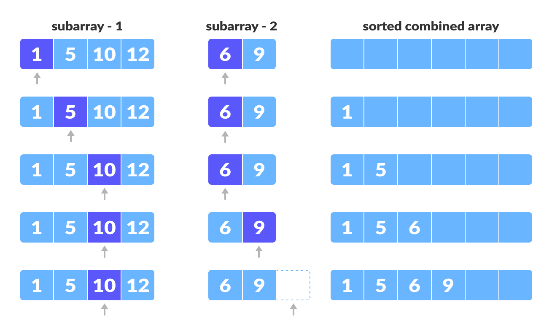
}

}

* + 1. **Kompleksitas *Merge Sort***

Untuk pengurutan n elemen, *merge sort* memiliki kompleksitas O(n log n). Jika *running time*  untuk *merge sort* terhadap struktur data dengan banyak elemen n adalah T(n), maka *recurrence*-nya T(n) = 2T(n/2) + n.

* + 1. **Contoh Penerapan**



1. **Hasil eksperimen/pengujian**
   1. **Bubble Sort**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 500 | 1000 | 10000 | 20000 | 100000 | 200000 |
| Uji 1 | 0,391 | 0,458 | 4,534 | 10,54 | 88,846 | 294,815 |
| Uji 2 | 0,253 | 0,419 | 4,112 | 9,431 | 86,48 | 285,683 |
| Uji 3 | 0,222 | 0,438 | 4,162 | 9,272 | 86,075 | 273,703 |
| Uji 4 | 0,239 | 0,434 | 4,085 | 9,161 | 88,012 | 278,393 |
| Uji 5 | 0,201 | 0,448 | 4,169 | 9,177 | 87,558 | 265,38 |

* 1. **Selection Sort**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 500 | 1000 | 10000 | 20000 | 100000 | 200000 |
| Uji 1 | 1,378 | 1,686 | 10,299 | 16,029 | 74,406 | 187,533 |
| Uji 2 | 0,964 | 1,991 | 14,183 | 15,252 | 95,925 | 180,376 |
| Uji 3 | 0,969 | 1,768 | 9,001 | 17,395 | 96,376 | 158,21 |
| Uji 4 | 1,087 | 1,591 | 10,829 | 14,514 | 83,172 | 150,043 |
| Uji 5 | 0,897 | 1,931 | 13,194 | 17,92 | 80,914 | 147,264 |

* 1. **Insertion Sort**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 500 | 1000 | 10000 | 20000 | 100000 | 200000 |
| Uji 1 | 0,832 | 1,664 | 10,344 | 14,78 | 88,667 | 172,596 |
| Uji 2 | 0,501 | 0,988 | 13,858 | 16,646 | 71,73 | 170,777 |
| Uji 3 | 0,792 | 1,351 | 14,464 | 20,31 | 71,441 | 169,3 |
| Uji 4 | 0,681 | 1,169 | 12,355 | 19,43 | 71,029 | 169,391 |
| Uji 5 | 0,718 | 1,209 | 11,052 | 15,925 | 70,243 | 170,554 |

* 1. **Merge Sort**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 500 | 1000 | 10000 | 20000 | 100000 | 200000 |
| Uji 1 | 0,249 | 0,443 | 4,175 | 8,399 | 37,666 | 67,446 |
| Uji 2 | 0,195 | 0,403 | 3,728 | 7,55 | 35,49 | 66,723 |
| Uji 3 | 0,331 | 0,435 | 3,671 | 7,205 | 36,663 | 65,883 |
| Uji 4 | 0,209 | 0,472 | 3,704 | 7,883 | 35,79 | 66,806 |
| Uji 5 | 0,202 | 0,444 | 3,68 | 7,141 | 36,643 | 66,134 |

* 1. **Quick Sort**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 500 | 1000 | 10000 | 20000 | 100000 | 200000 |
| Uji 1 | 0,263 | 0,401 | 3,901 | 9,167 | 37,993 | 67,962 |
| Uji 2 | 0,211 | 0,429 | 4,767 | 7,198 | 35,465 | 68,917 |
| Uji 3 | 0,203 | 0,499 | 3,651 | 7,117 | 41,66 | 67,232 |
| Uji 4 | 0,222 | 0,664 | 3,598 | 8,256 | 35,207 | 66,408 |
| Uji 5 | 0,211 | 0,659 | 3,577 | 7,109 | 34,885 | 66,052 |

* 1. **Perbandingan Setiap Sorting**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Bubble | Selection | Insertion | Merge | Quick |
| 500 | 0,391 | 1,378 | 0,832 | 0,249 | 0,231 |
| 1000 | 0,458 | 1,686 | 1,664 | 0,443 | 0,401 |
| 10000 | 4,534 | 10,299 | 10,344 | 4,175 | 3,901 |
| 20000 | 10,54 | 16,029 | 14,78 | 8,399 | 8,167 |
| 100000 | 88,846 | 74,406 | 88,667 | 37,666 | 36,372 |
| 200000 | 294,82 | 187,533 | 172,596 | 67,446 | 67,001 |

* 1. **Analisis Performansi dan Running Time**

Dari table serta grafik yang ada diatas, bisa kita lihat perbedaan waktu yang cukup jelas diatara algoritma-algoritma tersebut. Pengurutan data dengan metode *Quick Sort* lebih cepat jika dibandingkan dengan algoritma yang lain. Meskipun kecepatan *Merge Sort* dan *Quick Sort* tidak beda jauh bahkan hampir sama, tetapi menurut data pada tabel, *Quick Sort* merupakan pengurutan data dengan metode yang paling cepat. Hal ini dibuktikan dengan keadaan grafik waktu perbandingan setiap algoritma sorting.

1. **KESIMPULAN**

Berdasarkan hasil penelitian yang telah dilakukan maka dapat diambil kesimpulan bahwa:

1. Algoritma *Quick Sort* lebih cepat dalam melakukan pengurutan data jika dibandingkan dengan *Bubble Sort, Selection Sort, Insertion Sort, dan Merge Sort.*
2. Algoritma *Bubble Sort* mempunyai algoritma yang sederhana sehingga lebih mudah untuk dipahami.
3. *Merge Sort* dan *Quick Sort* memiliki selisih waktu yang sangat kecil
4. *Selection Sort* dan *Insertion Sort* merupakan metode pengurutan yang paling sederhana
5. *Bubble Sort* membutuhkan waktu komputasi paling lama

**LAMPIRAN**

[**https://docs.google.com/spreadsheets/d/1TlkRX47ogNF5toWpPePPyz8OdRSS7jyMc84wsjhdQ0g/edit?usp=sharing**](https://docs.google.com/spreadsheets/d/1TlkRX47ogNF5toWpPePPyz8OdRSS7jyMc84wsjhdQ0g/edit?usp=sharing)

**REFERENSI**

[1] Azizah, U. N. (n.d.). *Analisis Kompleksitas Algoritma.* Retrieved Desember 25, 2020, from http://repository.upi.edu/2878/6/S\_MTK\_0900249\_CHAPTER3.pdf

[2] Ezar, M. (2017, Agustus ). Perbandingan Kecepatan Gabungan Algoritma Utama Quick Sort dan Merge Sort dengan Algoritma Tambahan Insertion Sort, Bubble Sort, dan Selection Sort. *Teknik Informatika dan Sistem Informasi , III*, 319. Retrieved Desember 25, 2020, from https://www.researchgate.net/profile/Muhammad\_Rivan/publication/326229872\_Perbandingan\_Kecepatan\_Gabungan\_Algoritma\_Utama\_Quick\_Sort\_dan\_Merge\_Sort\_dengan\_Algoritma\_Tambahan\_Insertion\_Sort\_Bubble\_Sort\_dan\_Selection\_Sort

[3] Maulana, M. R. (2017, Oktober). Komparasi Algoritma Sorting pada Bahasa Pemrograman Java. *XII*, 2. Retrieved Desember 25, 2020, from https://ejournal.stmik-wp.ac.id/index.php/ictech/article/download/9/8

[4] Rahayuningsih, P. A. (2016). Analisis Perbandingan Kompleksitas Algoritma Pengurutan Nilai (Sorting). *Sains dan Manajemen, IV*, 1. doi: https://doi.org/10.31294/evolusi.v4i2.702

[5] Sonita, A., & Nurtaneo, F. (2015, September). ANALISIS PERBANDINGAN ALGORITMA BUBBLE SORT, MERGE SORT, DAN QUICK SORT DALAM PROSES PENGURUTAN KOMBINASI ANGKA DAN HURUF. *Jurnal Pseudocode, II*, 1-2. Retrieved Desember 25, 2020, from https://media.neliti.com/media/publications/126448-ID-analisis-perbandingan-algoritma-bubble-s.pdf

[6] Wisudawan, W. F. (t.thn.). *Kompleksitas Algoritma Sorting yang Populer Dipakai.* Diambil kembali dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2007-2008/Makalah/MakalahIF2153-0708-112.pdf