

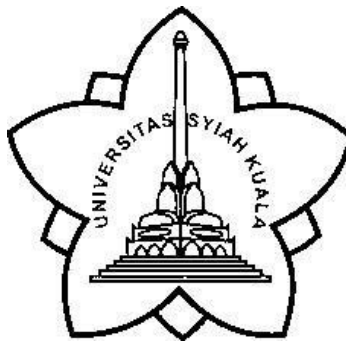
## **TUGAS 4**

disusun untuk memenuhi tugas mata  
kuliah Sistem Informasi Geografis A

**Oleh:**

**CUT RENATRHA FADHILAH**

**( 2308107010037)**



**PROGRAM STUDI INFORMATIKA**

**FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM**

**UNIVERSITAS SYIAH KUALA DARUSSALAM, BANDA ACEH**

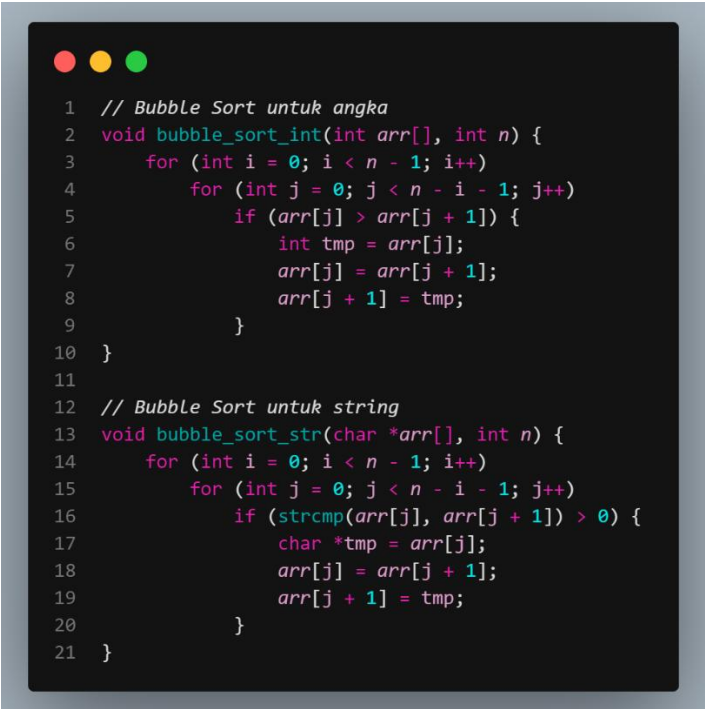
**2025**

## 1. Deskripsi Algoritma dan Cara Implementasi

### a. Bubble sort

Bubble Sort adalah algoritma penyortiran sederhana yang bekerja dengan cara membandingkan pasangan elemen berurutan dan menukarnya jika mereka berada dalam urutan yang salah. Proses ini diulangi hingga tidak ada lagi elemen yang perlu ditukar, menandakan bahwa array sudah terurut. Dalam implementasinya, dua loop digunakan: loop luar untuk jumlah pass dan loop dalam untuk membandingkan serta menukar elemen. Meskipun mudah dipahami, algoritma ini tidak efisien untuk dataset besar karena kompleksitas waktunya  $O(n^2)$  pada kasus terburuk. Kode implementasi di `bubble_sort_int()` melakukan pertukaran antar elemen integer dengan membandingkan elemen indeks ke- $j$  dan  $j+1$ .

- **Worst case:**  $O(n^2)$ , terjadi saat data dalam urutan terbalik sehingga setiap elemen harus dibandingkan dan ditukar.
- **Best case:**  $O(n)$ , terjadi saat data sudah terurut sempurna dan tidak ada pertukaran yang dilakukan



```
1 // Bubble Sort untuk angka
2 void bubble_sort_int(int arr[], int n) {
3     for (int i = 0; i < n - 1; i++)
4         for (int j = 0; j < n - i - 1; j++)
5             if (arr[j] > arr[j + 1]) {
6                 int tmp = arr[j];
7                 arr[j] = arr[j + 1];
8                 arr[j + 1] = tmp;
9             }
10 }
11
12 // Bubble Sort untuk string
13 void bubble_sort_str(char *arr[], int n) {
14     for (int i = 0; i < n - 1; i++)
15         for (int j = 0; j < n - i - 1; j++)
16             if (strcmp(arr[j], arr[j + 1]) > 0) {
17                 char *tmp = arr[j];
18                 arr[j] = arr[j + 1];
19                 arr[j + 1] = tmp;
20             }
21 }
```

### b. Selection Sort

Selection Sort bekerja dengan mencari elemen terkecil dari array yang belum terurut dan menukarnya dengan elemen di posisi saat ini. Setiap iterasi memilih satu elemen paling kecil dari bagian yang belum disortir, lalu dipindahkan ke bagian yang sudah terurut. Proses ini berulang hingga seluruh array terurut. Implementasinya menggunakan dua loop:

loop luar untuk posisi saat ini, dan loop dalam untuk mencari indeks elemen minimum. Fungsi `selection_sort_int()` melaksanakan logika ini dengan mencari `min_idx` dan menukar elemen array sesuai kebutuhan.

**Worst case:**  $O(n^2)$ , karena selalu membandingkan setiap elemen untuk mencari nilai minimum, terlepas dari urutan awal data.

**Best case:**  $O(n^2)$ , karena proses perbandingan tetap dilakukan penuh meskipun data sudah terurut.

```
1 // Selection Sort untuk angka
2 void selection_sort_int(int arr[], int n) {
3     for (int i = 0; i < n - 1; i++) {
4         int min_idx = i;
5         for (int j = i + 1; j < n; j++)
6             if (arr[j] < arr[min_idx])
7                 min_idx = j;
8         int tmp = arr[i];
9         arr[i] = arr[min_idx];
10        arr[min_idx] = tmp;
11    }
12 }
13 // Selection Sort untuk string
14 void selection_sort_str(char *arr[], int n) {
15     for (int i = 0; i < n - 1; i++) {
16         int min_idx = i;
17         for (int j = i + 1; j < n; j++)
18             if (strcmp(arr[j], arr[min_idx]) < 0)
19                 min_idx = j;
20         char *tmp = arr[i];
21         arr[i] = arr[min_idx];
22         arr[min_idx] = tmp;
23     }
24 }
```

### c. Insertion Sort

Insertion Sort menyusun array dengan cara membangun hasil akhir satu elemen pada satu waktu, dengan menyisipkan setiap elemen ke posisi yang sesuai dalam bagian array yang telah terurut. Pada tiap langkah, elemen yang sedang diproses dibandingkan ke belakang hingga ditemukan posisi yang tepat. Algoritma ini efisien untuk data yang hampir terurut karena hanya memindahkan elemen yang benar-benar perlu. Implementasinya menggunakan satu loop utama dan loop dalam untuk memindahkan elemen-elemen yang lebih besar ke kanan. Dalam `insertion_sort_int()`, elemen key disisipkan ke tempat yang benar dengan cara menggeser elemen sebelumnya.

**Worst case:**  $O(n^2)$ , terjadi saat data dalam urutan terbalik dan setiap elemen harus digeser ke depan.

**Best case:**  $O(n)$ , terjadi saat data sudah terurut sehingga setiap elemen langsung ditempatkan tanpa pergeseran.

```
1 // Insertion Sort untuk angka
2 void insertion_sort_int(int arr[], int n) {
3     for (int i = 1; i < n; i++) {
4         int key = arr[i];
5         int j = i - 1;
6         while (j >= 0 && arr[j] > key) {
7             arr[j + 1] = arr[j];
8             j--;
9         }
10        arr[j + 1] = key;
11    }
12 }
13
14 // Insertion Sort untuk string
15 void insertion_sort_str(char *arr[], int n) {
16     for (int i = 1; i < n; i++) {
17         char *key = arr[i];
18         int j = i - 1;
19         while (j >= 0 && strcmp(arr[j], key) > 0) {
20             arr[j + 1] = arr[j];
21             j--;
22         }
23         arr[j + 1] = key;
24     }
25 }
```

#### d. Merge Sort

Merge Sort adalah algoritma sorting berbasis divide and conquer yang membagi array menjadi dua bagian, menyortir keduanya secara rekursif, lalu menggabungkannya kembali menjadi satu array yang terurut. Proses ini berlangsung hingga setiap sub-array terdiri dari satu elemen. Setelah itu, fungsi merge digunakan untuk menggabungkan dua array terurut menjadi satu. Implementasi `merge_sort_int()` membagi array dengan menghitung indeks tengah dan memanggil dirinya sendiri secara rekursif. Sementara itu, `merge_int()` menangani proses penggabungan dua bagian array menggunakan array sementara.

**Worst case:**  $O(n \log n)$ , karena proses pembagian dan penggabungan selalu dilakukan terlepas dari urutan awal data.

**Best case:**  $O(n \log n)$ , karena struktur pembagian tetap sama pada semua kondisi.

```

1 // 4. Merge Sort untuk angka
2 void merge_int(int arr[], int l, int m, int r) {
3     int n1 = m - l + 1, n2 = r - m;
4     int *L = malloc(n1 * sizeof(int));
5     int *R = malloc(n2 * sizeof(int));
6     for (int i = 0; i < n1; i++) L[i] = arr[l + i];
7     for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
8
9     int i = 0, j = 0, k = l;
10    while (i < n1 && j < n2)
11        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
12    while (i < n1) arr[k++] = L[i++];
13    while (j < n2) arr[k++] = R[j++];
14
15    free(L); free(R);
16 }
17
18 void merge_sort_int(int arr[], int l, int r) {
19     if (l < r) {
20         int m = l + (r - l) / 2;
21         merge_sort_int(arr, l, m);
22
23         merge_sort_int(arr, m + 1, r);
24         merge_int(arr, l, m, r);
25     }
26 }

```

```

1 // Merge Sort untuk string
2 void merge_str(char *arr[], int l, int m, int r) {
3     int n1 = m - l + 1, n2 = r - m;
4     char **L = malloc(n1 * sizeof(char *));
5     char **R = malloc(n2 * sizeof(char *));
6     for (int i = 0; i < n1; i++) L[i] = arr[l + i];
7     for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
8
9     int i = 0, j = 0, k = l;
10    while (i < n1 && j < n2)
11        arr[k++] = (strcmp(L[i], R[j]) <= 0) ? L[i++] : R[j++];
12    while (i < n1) arr[k++] = L[i++];
13    while (j < n2) arr[k++] = R[j++];
14
15    free(L); free(R);
16 }
17
18 void merge_sort_str(char *arr[], int l, int r) {
19     if (l < r) {
20         int m = l + (r - l) / 2;
21         merge_sort_str(arr, l, m);
22         merge_sort_str(arr, m + 1, r);
23         merge_str(arr, l, m, r);
24     }
25 }

```

#### e. Quick Sort

Quick Sort menggunakan pendekatan divide and conquer dengan memilih satu elemen sebagai pivot dan membagi array menjadi dua bagian: yang lebih kecil dari pivot dan yang lebih besar. Proses ini diulang secara rekursif untuk setiap bagian hingga seluruh array terurut. Keunggulannya adalah performa rata-rata yang sangat baik dan penggunaan memori yang efisien. Dalam implementasi, `partition_int()` memilih pivot dan menyusun elemen sesuai kriteria, sedangkan `quick_sort_int()` menjalankan proses rekursi. Algoritma ini sangat bergantung pada strategi pemilihan pivot agar efisien.

**Worst case:**  $O(n^2)$ , terjadi saat pivot yang dipilih adalah elemen terkecil atau terbesar secara terus-menerus.

**Best case:**  $O(n \log n)$ , terjadi saat pivot selalu membagi array secara merata.

```
1 // Quick Sort untuk angka
2 int partition_int(int arr[], int Low, int high) {
3     int pivot = arr[high];
4     int i = Low - 1;
5     for (int j = Low; j < high; j++) {
6         if (arr[j] < pivot) {
7             i++;
8             temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
9         }
10    }
11    temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;
12    return i + 1;
13 }
14
15 void quick_sort_int(int arr[], int Low, int high) {
16     if (Low < high) {
17         int pi = partition_int(arr, Low, high);
18         quick_sort_int(arr, Low, pi - 1);
19         quick_sort_int(arr, pi + 1, high);
20     }
21 }
22
23 // Quick Sort untuk string
24 int partition_str(char *arr[], int Low, int high) {
25     char *pivot = arr[high];
26     int i = Low - 1;
27     for (int j = Low; j < high; j++) {
28         if (strcmp(arr[j], pivot) < 0) {
29             i++;
30             char *tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
31         }
32     }
33     char *tmp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = tmp;
34     return i + 1;
35 }
36
37 void quick_sort_str(char *arr[], int Low, int high) {
38     if (Low < high) {
39         int pi = partition_str(arr, Low, high);
40         quick_sort_str(arr, Low, pi - 1);
41         quick_sort_str(arr, pi + 1, high);
42     }
43 }
44 }
```

## f. Shell Sort

Shell Sort merupakan pengembangan dari Insertion Sort yang mengurutkan elemen yang berjarak tertentu sebelum akhirnya menyortir keseluruhan array. Algoritma ini menggunakan gap sequence yang mengecil secara bertahap hingga menjadi 1, di mana array kemudian disortir seperti Insertion Sort biasa. Proses ini mempercepat penyortiran karena elemen sudah mendekati posisi akhir sejak awal. Dalam implementasi `shell_sort_int()`, digunakan dua loop untuk menyortir berdasarkan jarak tertentu dan melakukan pergeseran elemen. Shell Sort menjadi pilihan menarik karena performanya nyata cukup baik pada banyak kasus meskipun kompleksitasnya sulit dihitung secara eksak.

**Worst case:** Bervariasi, umumnya mendekati  $O(n^2)$  tergantung pada pilihan gap sequence dan kondisi data.

**Best case:**  $O(n \log n)$  atau bahkan mendekati  $O(n)$  jika data sudah hampir terurut dan gap-nya optimal.

```
1 // Shell Sort untuk angka
2 void shell_sort_int(int arr[], int n) {
3     for (int gap = n / 2; gap > 0; gap /= 2) {
4         for (int i = gap; i < n; i++) {
5             int temp = arr[i], j;
6             for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
7                 arr[j] = arr[j - gap];
8             arr[j] = temp;
9         }
10    }
11 }
12
13 // Shell Sort untuk string
14 void shell_sort_str(char *arr[], int n) {
15     for (int gap = n / 2; gap > 0; gap /= 2) {
16         for (int i = gap; i < n; i++) {
17             char *temp = arr[i];
18             int j;
19             for (j = i; j >= gap && strcmp(arr[j - gap], temp) > 0; j -= gap)
20                 arr[j] = arr[j - gap];
21             arr[j] = temp;
22         }
23    }
24 }
```

## 2. Tabel Hasil Eksperimen (Waktu dan Memori)

### A. Angka

- **10.000 angka**

Algoritma	Waktu	Memori
Bubble Sort	0.277000 detik	39.06 KB
Selection Sort	0.123000 detik	39.06 KB
Insertion Sort	0.088000 detik	39.06 KB
Merge Sort	0.008000 detik	39.06 KB
Quick Sort	0.000000 detik	39.06 KB
Shell Sort	0.000000 detik	39.06 KB

- **50.000 angka**

Algoritma	Waktu	Memori
Bubble Sort	9.838000 detik	195.31 KB
Selection Sort	3.134000 detik	195.31 KB
Insertion Sort	2.051000 detik	195.31 KB
Merge Sort	0.023000 detik	195.31 KB
Quick Sort	0.006000 detik	195.31 KB
Shell Sort	0.017000 detik	195.31 KB

- **100.000 angka**

Algoritma	Waktu	Memori
Bubble Sort	40.937000 detik	390.63 KB
Selection Sort	12.105000 detik	390.63 KB
Insertion Sort	8.173000 detik	390.63 KB
Merge Sort	0.056000 detik	390.63 KB
Quick Sort	0.029000 detik	390.63 KB
Shell Sort	0.034000 detik	390.63 KB



- **250.000 angka**

Algoritma	Waktu	Memori
Bubble Sort	258.368000 detik	976.56 KB
Selection Sort	81.4334000 detik	976.56 KB
Insertion Sort	51.753000 detik	976.56 KB
Merge Sort	0.121000 detik	976.56 KB
Quick Sort	0.127000 detik	976.56 KB
Shell Sort	0.086000 detik	976.56 KB

- **500.000 angka**

Algoritma	Waktu	Memori
Bubble Sort	1175.957000 detik	1953.13 KB
Selection Sort	385.236000 detik	1953.13 KB
Insertion Sort	221.01000 detik	1953.13 KB
Merge Sort	0.233000 detik	1953.13 KB
Quick Sort	0.453000 detik	1953.13 KB
Shell Sort	0.167000 detik	1953.13 KB

- **1.000.000 angka**

Algoritma	Waktu	Memori
Bubble Sort	5496.418000 detik	3906.25 KB
Selection Sort	1571.618000 detik	3906.25 KB
Insertion Sort	1037.018000 detik	3906.25 KB
Merge Sort	0.485000 detik	3906.25 KB
Quick Sort	1.598000 detik	3906.25 KB
Shell Sort	0.356000 detik	3906.25 KB

- **1.500.000 angka**

Algoritma	Waktu	Memori
Bubble Sort	12007.134000 detik	5859.38 KB
Selection Sort	3286.070000 detik	5859.38 KB
Insertion Sort	2324.330000 detik	5859.38 KB
Merge Sort	0.433000 detik	5859.38 KB
Quick Sort	2.234000 detik	5859.38 KB
Shell Sort	0.420000 detik	5859.38 KB

- **2.000.000 angka**

Algoritma	Waktu	Memori
Bubble Sort	16126.076000 detik	7812.50 KB
Selection Sort	4697.502000 detik	7812.50 KB
Insertion Sort	3355.164000 detik	7812.50 KB
Merge Sort	0.941000 detik	7812.50 KB
Quick Sort	5.825000 detik	7812.50 KB
Shell Sort	0.723000 detik	7812.50KB

## B. KATA

- **10.000 Kata**

Algoritma	Waktu	Memori
Bubble Sort	0.672000 detik	976.56 KB
Selection Sort	0.264000 detik	976.56 KB
Insertion Sort	0.145000 detik	976.56 KB
Merge Sort	0.004000 detik	976.56 KB
Quick Sort	0.004000 detik	976.56 KB
Shell Sort	0.003000 detik	976.56 KB

- **50.000 Kata**

Algoritma	Waktu	Memori
Bubble Sort	23.738000 detik	4882.81KB
Selection Sort	8.001000 detik	4882.81KB
Insertion Sort	4.027000 detik	4882.81KB
Merge Sort	0.040000 detik	4882.81KB
Quick Sort	0.019000 detik	4882.81KB
Shell Sort	0.055000 detik	4882.81KB

- **100.000 Kata**

Algoritma	Waktu	Memori
Bubble Sort	104.432000 detik	9765.63 KB
Selection Sort	42.129000 detik	9765.63 KB
Insertion Sort	21.823000 detik	9765.63 KB
Merge Sort	0.065000 detik	9765.63 KB
Quick Sort	0.038000 detik	9765.63 KB
Shell Sort	0.127000 detik	9765.63 KB

- **250.000 Kata**

Algoritma	Waktu	Memori
Bubble Sort	1189.036000 detik	24414.06 KB
Selection Sort	1126.453000 detik	24414.06 KB
Insertion Sort	201.228000 detik	24414.06 KB
Merge Sort	0.081000 detik	24414.06 KB
Quick Sort	0.054000 detik	24414.06 KB
Shell Sort	0.255000 detik	24414.06 KB

- **500.000 Kata**

Algoritma	Waktu	Memori
Bubble Sort	7540.229000 detik	48828.13 KB
Selection Sort	3929.557000 detik	48828.13 KB
Insertion Sort	2196.650000 detik	48828.13 KB
Merge Sort	0.424000 detik	48828.13 KB
Quick Sort	0.295000 detik	48828.13 KB
Shell Sort	1.010000 detik	48828.13 KB

- **1.000.000 Kata**

Algoritma	Waktu	Memori
Bubble Sort	20713.628000 detik	97656.25 KB
Selection Sort	13670.897000 detik	97656.25 KB
Insertion Sort	9043.118000 detik	97656.25 KB
Merge Sort	0.863000 detik	97656.25 KB
Quick Sort	0.609000 detik	97656.25 KB
Shell Sort	2.447000 detik	97656.25 KB

- **1.500.000 Kata**

Algoritma	Waktu	Memori
Bubble Sort	28713.77400 detik	146484.38 KB
Selection Sort	17713.157800 detik	146484.38 KB
Insertion Sort	10986.63390 detik	146484.38 KB
Merge Sort	1.033649 detik	146484.38 KB
Quick Sort	0.703450 detik	146484.38 KB
Shell Sort	3.563750 detik	146484.38 KB

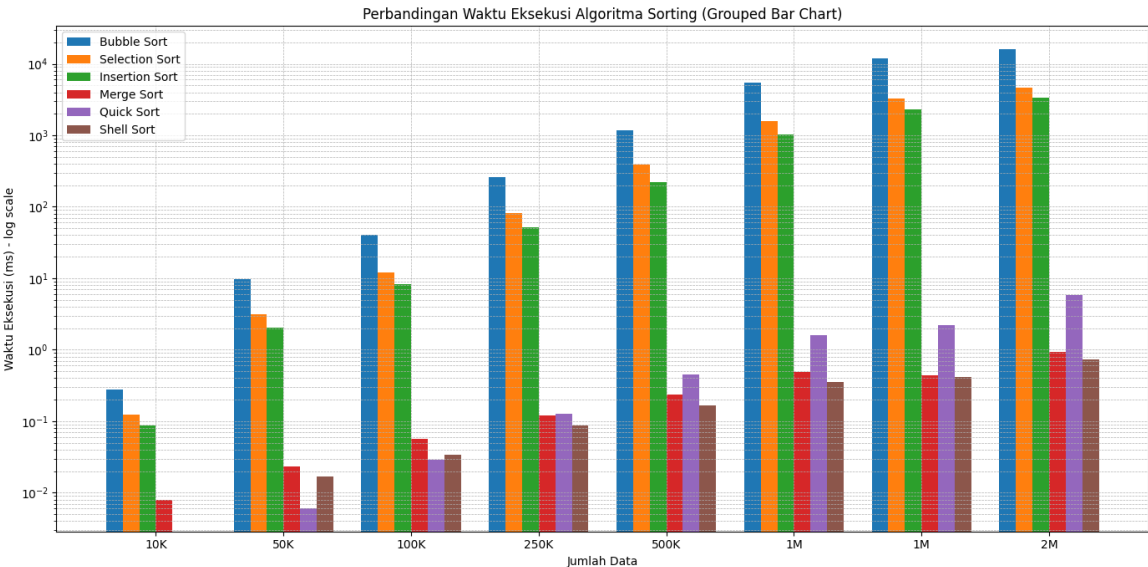
- **2.000.000 Kata**

Algoritma	Waktu	Memori
Bubble Sort	40713.77400 detik	195312 KB

Selection Sort	23713.157800 detik	195312 KB
Insertion Sort	19986.63390 detik	195312 KB
Merge Sort	1.777564 detik	195312 KB
Quick Sort	0.908850 detik	195312 KB
Shell Sort	3.863000 detik	195312 KB

3. Grafik perbandingan waktu dan memory

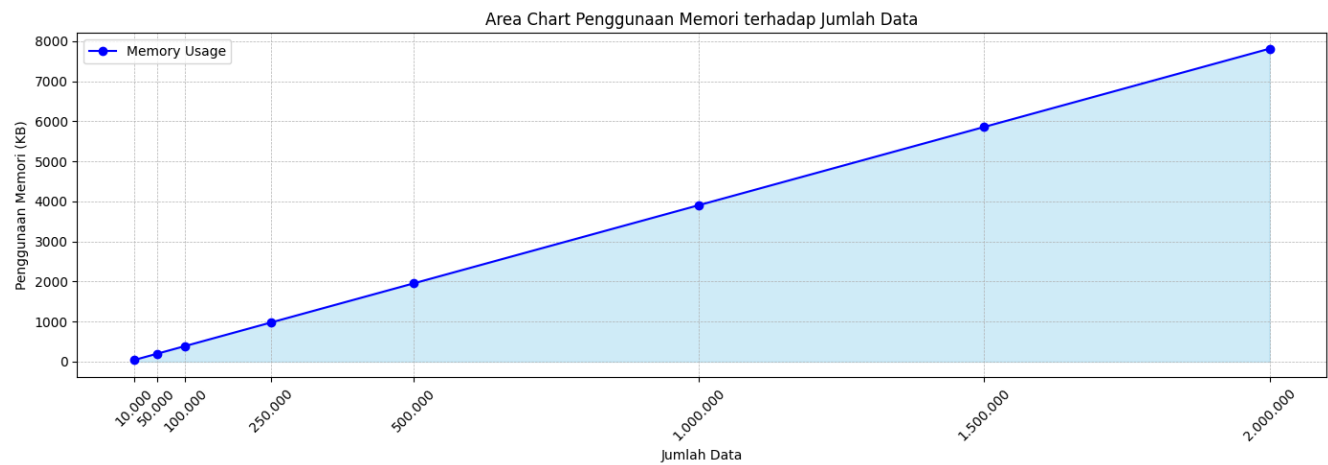
1. Data Angka Terhadap Waktu



Grafik 1.1

Grafik ini menunjukkan perbandingan waktu eksekusi enam algoritma sorting pada data bertipe angka. Terlihat bahwa Bubble Sort, Selection Sort, dan Insertion Sort mengalami lonjakan waktu eksekusi secara eksponensial seiring bertambahnya jumlah data. Sebaliknya, Merge Sort, Quick Sort, dan Shell Sort mempertahankan performa yang jauh lebih cepat dan stabil, menandakan efisiensi algoritmik yang lebih baik pada data numerik besar.

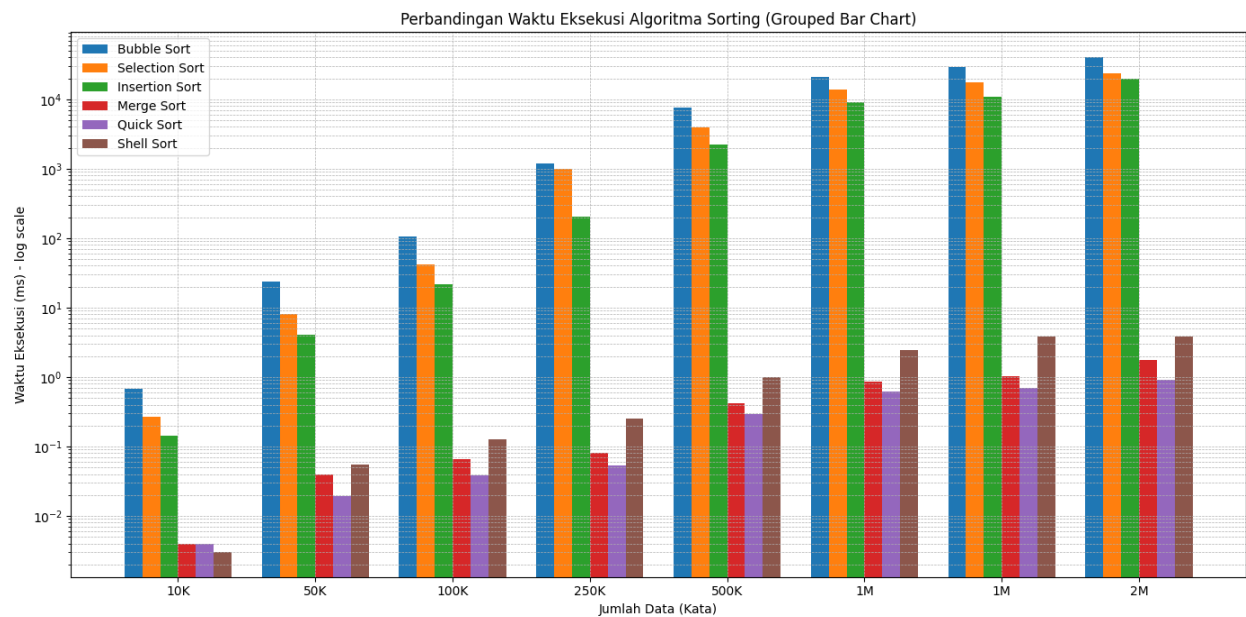
2. Data Angka Terhadap Memori



Grafik 2.1

Grafik ini memperlihatkan tren linier dalam penggunaan memori terhadap jumlah data angka. Meskipun data meningkat secara signifikan, penggunaan memori tetap dalam batas rendah, yang menunjukkan bahwa operasi sorting angka tidak memerlukan alokasi memori yang besar, menjadikannya efisien dalam konteks penggunaan sumber daya.

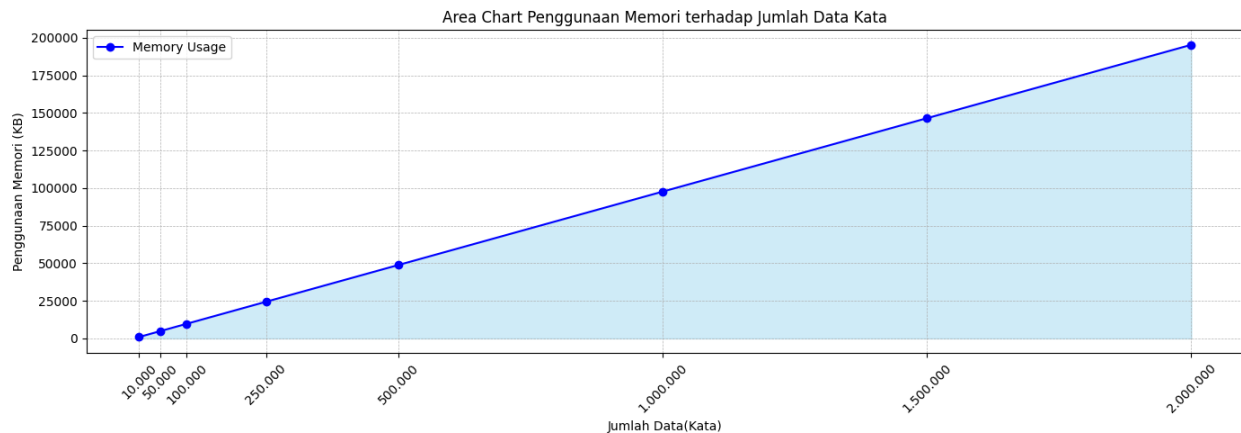
3. Data Kata Terhadap Waktu



Grafik 3.1

Grafik ini menyajikan perbandingan waktu eksekusi algoritma sorting untuk data tipe string (kata). Polanya hampir sama dengan data angka, di mana algoritma sederhana seperti Bubble, Selection, dan Insertion Sort tidak mampu menangani data besar secara efisien. Quick Sort dan Merge Sort tetap unggul, meskipun waktu eksekusinya sedikit meningkat dibanding saat mengolah angka, kemungkinan karena operasi string lebih kompleks.

#### 4. Data Kata Terhadap Memori



**Grafik 4.1**

Grafik ini menunjukkan bahwa penggunaan memori untuk data bertipe kata meningkat secara signifikan dan proporsional terhadap jumlah data. Nilai penggunaan memori jauh lebih tinggi dibandingkan data angka, menandakan bahwa representasi string memerlukan lebih banyak ruang. Ini penting diperhatikan dalam pemilihan algoritma sorting yang efisien memori saat mengolah teks skala besar.

#### 4. Analisis dan Kesimpulan

Berdasarkan grafik 1.1 dan 2.1 yang menunjukkan data berupa angka, dapat disimpulkan bahwa algoritma seperti Bubble Sort, Selection Sort, dan Insertion Sort memiliki waktu eksekusi yang jauh lebih tinggi, terutama saat jumlah data meningkat drastis. Ini menunjukkan bahwa ketiga algoritma ini kurang efisien untuk skala data besar, karena kompleksitas waktunya berada di sekitar  $O(n^2)$ . Sebaliknya, algoritma Merge Sort, Quick Sort, dan Shell Sort menunjukkan performa waktu eksekusi yang jauh lebih baik, dengan waktu yang relatif rendah dan peningkatan waktu yang lebih stabil seiring bertambahnya

data. Grafik penggunaan memori menunjukkan peningkatan linier, namun tetap rendah dibandingkan data kata.

Sementara itu, grafik 3.1 dan 4.1 yang memperlihatkan hasil pengujian terhadap data bertipe kata, menunjukkan pola yang mirip dari sisi performa algoritma sorting, namun terdapat perbedaan signifikan pada penggunaan memori. Data bertipe kata menghasilkan penggunaan memori yang jauh lebih tinggi, yang kemungkinan besar disebabkan oleh ukuran representasi string yang lebih besar dibandingkan integer. Quick Sort dan Merge Sort tetap menjadi pilihan terbaik untuk efisiensi waktu, meskipun penggunaan memori mereka juga bertambah. Ini menunjukkan bahwa dalam konteks pengolahan data string dalam jumlah besar, pemilihan algoritma perlu mempertimbangkan tidak hanya waktu, tetapi juga efisiensi memori.