

*Laporan mata kuliah Struktur Data dan Algoritma*

## **TUGAS 4 STRUKTUR DATA DAN ALGORITMA**

disusun untuk memenuhi tugas  
mata kuliah Struktur Data dan Algoritma

Oleh:

**DIAN ISLAMI  
(2308107010048)**



**PROGRAM STUDI INFORMATIKA FAKULTAS  
MATEMATIKA DAN ILMU PENGETAHUAN ALAM  
UNIVERSITAS SYIAH KUALA  
2025**

## 1. Pendahuluan

Pengurutan (sorting) merupakan salah satu proses fundamental dalam ilmu komputer yang digunakan untuk menyusun data dalam urutan tertentu, baik secara menaik (ascending) maupun menurun (descending). Berbagai algoritma telah dikembangkan untuk menangani proses pengurutan, masing-masing dengan kelebihan dan kekurangannya dalam hal efisiensi waktu dan penggunaan memori.

Pada tugas ini, dilakukan implementasi dan evaluasi terhadap enam algoritma sorting klasik, yaitu **Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan Shell Sort**. Setiap algoritma diimplementasikan dalam bahasa pemrograman C dan diuji kinerjanya dalam menangani data skala besar, baik berupa angka maupun kata.

Tujuan dari eksperimen ini adalah untuk:

1. Mengevaluasi performa algoritma sorting dalam mengolah data berukuran besar.
2. Membandingkan efisiensi waktu eksekusi dan penggunaan memori dari masing-masing algoritma.
3. Memberikan pemahaman mendalam mengenai kompleksitas waktu dan ruang (space) dari algoritma sorting melalui pengujian empiris.

Eksperimen dilakukan dengan menggunakan data acak sebanyak hingga 2.000.000 entri, baik angka maupun kata. Ukuran data bervariasi mulai dari 10.000 hingga 2.000.000, dan hasil eksperimen dicatat dalam bentuk tabel dan grafik untuk dianalisis. Hasil dari eksperimen ini diharapkan dapat memberikan wawasan praktis mengenai pemilihan algoritma sorting yang sesuai berdasarkan kebutuhan efisiensi.

## 2. Struktur Program

Program ini disusun secara modular untuk memudahkan pengelolaan kode dan memudahkan pembaca untuk memahami alur program. Struktur utama dari program terdiri atas tiga komponen utama, yaitu:

### a. Header file (sort\_algorithms.h)

File header ini berisi deklarasi semua fungsi sorting yang diimplementasikan. Tujuannya adalah agar fungsi-fungsi tersebut dapat digunakan di berbagai file sumber (source file) seperti main.c, tanpa harus menulis ulang definisi fungsinya. Dengan pendekatan ini, program menjadi lebih terstruktur dan mudah dikembangkan.

### b. File implementasi algoritma (sort\_algorithms.c)

File ini berisi seluruh implementasi dari enam algoritma sorting, baik untuk data bertipe integer maupun string. Semua fungsi ditulis secara terpisah sesuai jenis algoritma, yaitu: bubble sort, selection sort, insertion sort, merge sort, quick sort, dan shell sort. Setiap fungsi dilengkapi komentar yang menjelaskan prinsip kerja algoritma, sehingga memudahkan pemahaman dan penelusuran kode.

### c. Program utama (main.c)

File utama ini berfungsi sebagai pengendali eksekusi program. Di dalamnya, dilakukan proses pembacaan data dari file eksternal, pemanggilan fungsi-fungsi sorting, serta pengukuran waktu eksekusi dan penggunaan memori untuk keperluan eksperimen. main.c juga mengatur proses pengujian untuk berbagai ukuran data sesuai spesifikasi tugas.

### 3. Kode Program & Deskripsi Algoritma

Sebelum menguji dan menganalisis performa algoritma pengurutan, langkah pertama yang dilakukan adalah menyiapkan data uji berupa angka dan kata dalam jumlah yang ditentukan yaitu berjumlah 2.000.000 baris. Kode yang dipakai merupakan kode yang diberikan pada deskripsi tugas. Berikut penjelasan setiap bagian kode nya:

```
1 void generate_random_numbers(const char *filename, int count, int max_value) {
2     FILE *fp = fopen(filename, "w");
3     if (!fp) {
4         perror("File tidak dapat dibuka");
5         return;
6     }
7     srand(time(NULL));
8     for (int i = 0; i < count; i++) {
9         fprintf(fp, "%d\n", rand() % max_value);
10    }
11    fclose(fp);
12 }
```

Fungsi ini digunakan untuk menghasilkan sejumlah angka acak dalam rentang tertentu dan menyimpannya ke dalam file. Fungsi ini membuka file output, kemudian menggunakan rand() untuk membangkitkan angka acak yang ditulis satu per satu ke dalam file.

```
1 void random_word(char *word, int length) {
2     static const char charset[] = "abcdefghijklmnopqrstuvwxyz";
3     for (int i = 0; i < length; i++) {
4         word[i] = charset[rand() % (sizeof(charset) - 1)];
5     }
6     word[length] = '\0';
7 }
```

Fungsi ini digunakan untuk membentuk satu kata acak berdasarkan panjang yang ditentukan. Kata ini disusun dari karakter huruf kecil alfabet, dipilih secara acak, dan diakhiri dengan karakter null (\0) sebagai penutup string.

```
1 void generate_random_words(const char *filename, int count, int max_word_length) {
2     FILE *fp = fopen(filename, "w");
3     if (!fp) {
4         perror("File tidak dapat dibuka");
5         return;
6     }
7     srand(time(NULL));
8     char word[100];
9     for (int i = 0; i < count; i++) {
10        int len = (rand() % (max_word_length - 3)) + 3;
11        random_word(word, len);
12        fprintf(fp, "%s\n", word);
13    }
14    fclose(fp);
15 }
```

Fungsi ini akan menghasilkan kata acak dalam jumlah besar. Fungsi ini secara otomatis menentukan panjang setiap kata secara acak (antara 3 sampai panjang maksimum), membentuk kata menggunakan random\_word, dan menuliskannya ke dalam file.

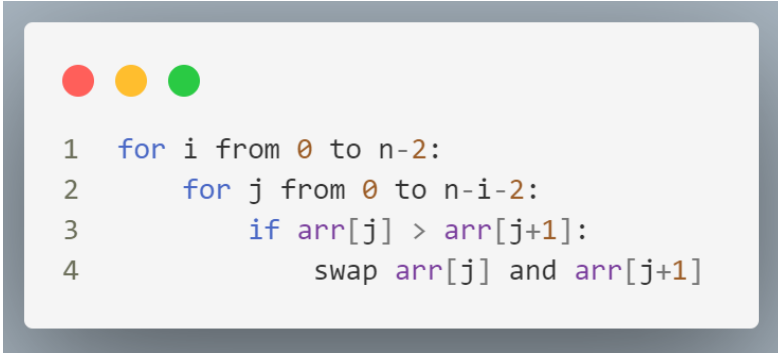
```
1 int main() {
2     generate_random_numbers("data_angka.txt", 2000000, 2000000);
3     generate_random_words("data_kata.txt", 2000000, 20);
4     return 0;
5 }
```

Fungsi main menjadi titik awal eksekusi program. Di dalamnya terdapat pemanggilan fungsi untuk membangkitkan 2 juta angka dan 2 juta kata, masing-masing disimpan ke dalam file data\_angka.txt dan data\_kata.txt.

## a. Algoritma & Implementasinya

- **Bubble sort**

Algoritma:

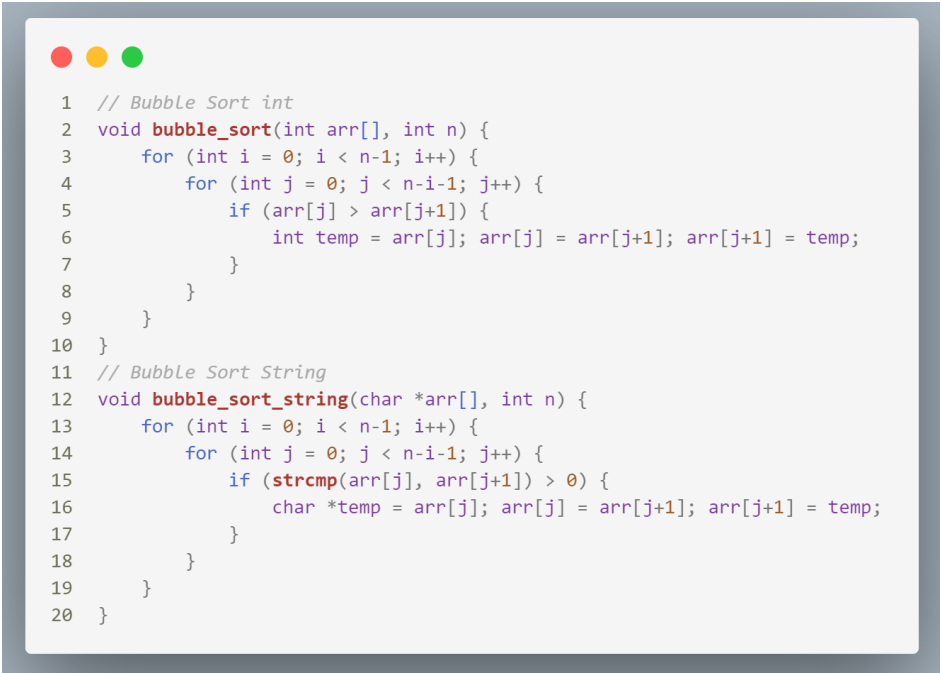


```
1 for i from 0 to n-2:
2     for j from 0 to n-i-2:
3         if arr[j] > arr[j+1]:
4             swap arr[j] and arr[j+1]
```

Bubble Sort adalah algoritma pengurutan sederhana yang bekerja dengan membandingkan pasangan elemen bersebelahan. Berikut tahapannya:

- 1) Bandingkan dua elemen bersebelahan.
- 2) Tukar jika urutannya salah (misalnya lebih besar di kiri).
- 3) Ulangi langkah di atas untuk seluruh elemen hingga tidak ada pertukaran lagi.

Implementasi kode program:



```
1 // Bubble Sort int
2 void bubble_sort(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 temp = arr[j]; arr[j] = arr[j+1]; arr[j+1] = temp;
7             }
8         }
9     }
10 }
11 // Bubble Sort String
12 void bubble_sort_string(char *arr[], int n) {
13     for (int i = 0; i < n-1; i++) {
14         for (int j = 0; j < n-i-1; j++) {
15             if (strcmp(arr[j], arr[j+1]) > 0) {
16                 char *temp = arr[j]; arr[j] = arr[j+1]; arr[j+1] = temp;
17             }
18         }
19     }
20 }
```

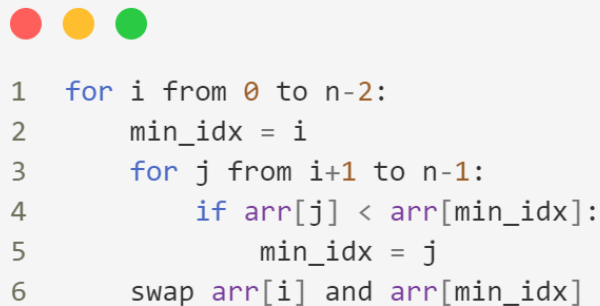
Penjelasan tambahan:

- Dua loop digunakan: loop luar untuk n-1 iterasi, dan loop dalam untuk membandingkan dan menukar elemen bersebelahan.
- Untuk versi string, digunakan strcmp() untuk membandingkan dua string.

Kode program tersebut dibedakan antara string dan int, untuk pengujian menggunakan data\_angka.txt dipakai kode yang menerima tipe data int, sedangkan pengujian dengan menggunakan data\_kata.txt akan menggunakan kode yang menerima tipe data char.

- **Selection sort**

Algoritma:

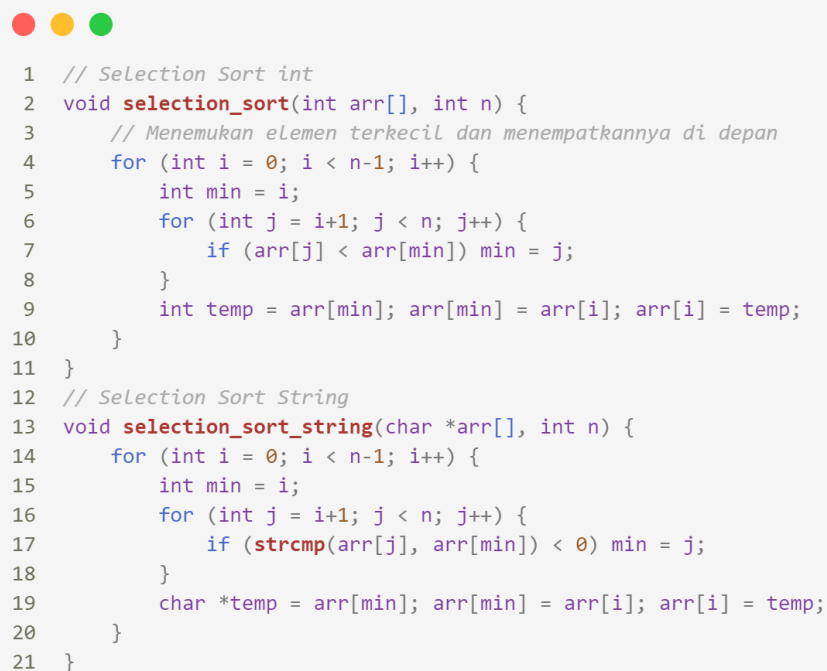


```
1 for i from 0 to n-2:
2     min_idx = i
3     for j from i+1 to n-1:
4         if arr[j] < arr[min_idx]:
5             min_idx = j
6     swap arr[i] and arr[min_idx]
```

Selection Sort mencari elemen terkecil dari bagian array yang belum terurut, lalu menukarnya ke posisi yang benar. Berikut tahapannya:

- 1) Temukan elemen terkecil dari array yang belum terurut.
- 2) Tukar elemen tersebut dengan elemen pertama dari bagian yang belum terurut.
- 3) Ulangi untuk seluruh array.

Implementasi kode program:



```
1 // Selection Sort int
2 void selection_sort(int arr[], int n) {
3     // Menemukan elemen terkecil dan menempatkannya di depan
4     for (int i = 0; i < n-1; i++) {
5         int min = i;
6         for (int j = i+1; j < n; j++) {
7             if (arr[j] < arr[min]) min = j;
8         }
9         int temp = arr[min]; arr[min] = arr[i]; arr[i] = temp;
10    }
11 }
12 // Selection Sort String
13 void selection_sort_string(char *arr[], int n) {
14     for (int i = 0; i < n-1; i++) {
15         int min = i;
16         for (int j = i+1; j < n; j++) {
17             if (strcmp(arr[j], arr[min]) < 0) min = j;
18         }
19         char *temp = arr[min]; arr[min] = arr[i]; arr[i] = temp;
20     }
21 }
```


Penjelasan tambahan:

- Loop luar menandai posisi saat ini.
- Loop dalam mencari elemen terkecil dan menyimpan di variabel indeks min.
- Setelah selesai, tukar arr[min] dan arr[i].

Sama seperti sebelumnya, kode program tersebut dibedakan antara string dan int, dipakai menyesuaikan dengan file txt yang dipakai.

- **Insertion sort**

Algoritma:




```
1  for i from 1 to n-1:
2      key = arr[i]
3      j = i - 1
4      while j >= 0 and arr[j] > key:
5          arr[j+1] = arr[j]
6          j = j - 1
7      arr[j+1] = key
```

Insertion Sort bekerja dengan menyusun array satu per satu, dimulai dari indeks pertama yang dianggap sudah terurut. Berikut tahapannya:

- 1) Setiap elemen dari indeks ke-1 hingga akhir dianggap sebagai key.
- 2) Elemen-elemen di sebelah kirinya dibandingkan dengan key.
- 3) Jika lebih besar dari key, maka elemen tersebut digeser ke kanan.
- 4) Ketika ditemukan posisi yang tepat, key disisipkan di tempat itu.

Implementasi kode program:



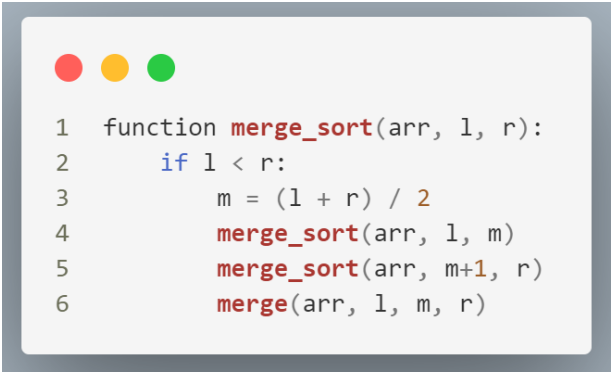
```
1  // Insertion Sort int
2  void insertion_sort(int arr[], int n) {
3      // Menyisipkan elemen pada posisi yang benar
4      for (int i = 1; i < n; i++) {
5          int key = arr[i], j = i - 1;
6          while (j >= 0 && arr[j] > key) {
7              arr[j+1] = arr[j]; j--;
8          }
9          arr[j+1] = key;
10     }
11 }
12
13 // Insertion Sort String
14 void insertion_sort_string(char *arr[], int n) {
15     for (int i = 1; i < n; i++) {
16         char *key = arr[i];
17         int j = i - 1;
18         while (j >= 0 && strcmp(arr[j], key) > 0) {
19             arr[j+1] = arr[j];
20             j--;
21         }
22         arr[j+1] = key;
23     }
24 }
```

Pada versi string, pointer ke string dibandingkan menggunakan strcmp() dan dilakukan pergeseran pointer sampai posisi yang tepat ditemukan.

- **Merge sort**

Sort ini menggunakan dua fungsi, fungsi utama dan fungsi pembantu.

Algoritma fungsi utama merge sort:



```


1 function merge_sort(arr, l, r):
2     if l < r:
3         m = (l + r) / 2
4         merge_sort(arr, l, m)
5         merge_sort(arr, m+1, r)
6         merge(arr, l, m, r)

```

Fungsi merge\_sort adalah fungsi utama yang menggunakan pendekatan rekursif dan divide-and-conquer. Berikut tahapannya:

- 1) Array dibagi dua bagian hingga elemen terkecil (subarray berisi satu elemen).
- 2) Kemudian bagian-bagian tersebut digabung kembali dalam keadaan sudah terurut melalui fungsi merge.
- 3) Proses rekursif dilakukan dengan mencari nilai tengah menggunakan  $m = l + (r - l) / 2$ .

Implementasi kode program:



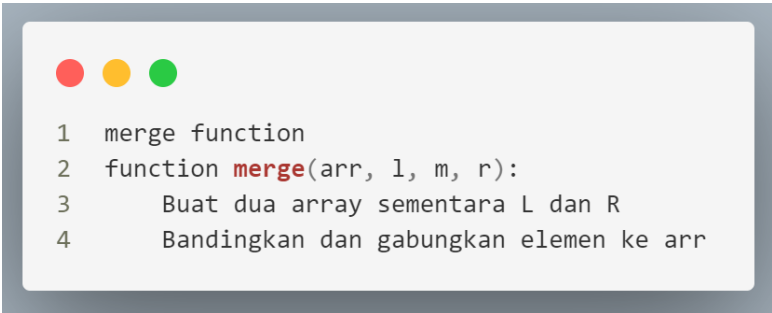
```

1 void merge_sort(int arr[], int l, int r) {
2     if (l < r) {
3         int m = l + (r - l) / 2;
4         merge_sort(arr, l, m);
5         merge_sort(arr, m + 1, r);
6         merge(arr, l, m, r);
7     }
8 }

```

Versi string (merge\_sort\_string) melakukan hal yang sama tetapi untuk array berisi pointer ke string (char \*arr[]) dan perbandingan menggunakan strcmp().

Algoritma fungsi pembantu:



```

1 merge function
2 function merge(arr, l, m, r):
3     Buat dua array sementara L dan R
4     Bandingkan dan gabungkan elemen ke arr

```

Implementasi kode program:

```
1 // Merge Sort int
2 void merge(int arr[], int l, int m, int r) {
3     int i, j, k;
4     int n1 = m - l + 1;
5     int n2 = r - m;
6
7     // Gunakan heap allocation untuk array L dan R
8     int *L = malloc(n1 * sizeof(int));
9     int *R = malloc(n2 * sizeof(int));
10
11     if (L == NULL || R == NULL) {
12         fprintf(stderr, "Gagal alokasi memori pada merge\n");
13         exit(1);
14     }
15
16     for (i = 0; i < n1; i++) L[i] = arr[l + i];
17     for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
18
19     i = 0; j = 0; k = l;
20
21     while (i < n1 && j < n2) {
22         arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
23     }
24
25     while (i < n1) arr[k++] = L[i++];
26     while (j < n2) arr[k++] = R[j++];
27
28     // Jangan lupa free heap memory
29     free(L);
30     free(R);
31 }
```

Fungsi merge adalah fungsi pembantu dari merge\_sort. Berikut tahapannya:

- 1) Fungsi merge digunakan untuk menggabungkan dua bagian array yang sudah terurut.
- 2) Elemen dari bagian kiri (L) dan kanan (R) dibandingkan satu per satu dan dimasukkan ke arr[] dalam urutan yang benar.
- 3) Alokasi dilakukan dengan malloc() untuk menghindari batas stack (heap allocation).
- 4) Setelah proses selesai, memori heap dibebaskan dengan free().
- 5) Sedangkan untuk string, hampir sama dengan fungsi merge int, namun untuk data string (char \*arr[]).
- 6) Perbandingan menggunakan strcmp() untuk mengurutkan string secara leksikografis (sesuai urutan alfabet).
- 7) Pointer string disalin dan dibandingkan, bukan isi string secara langsung.

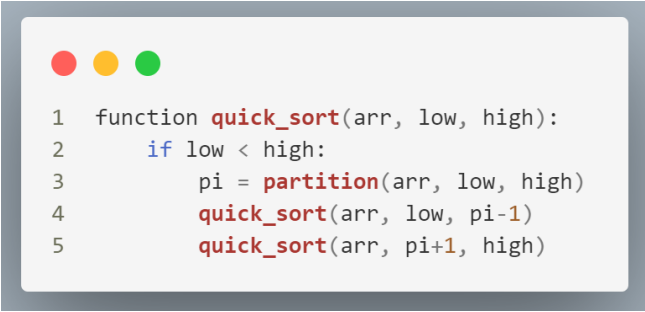
```
1 // Merge Sort String
2 void merge_string(char *arr[], int l, int m, int r) {
3     ...
4     char **L = malloc(n1 * sizeof(char *));
5     char **R = malloc(n2 * sizeof(char *));
6     if (!L || !R) {
7         ...
8     while (i < n1 && j < n2) {
9         arr[k++] = (strcmp(L[i], R[j]) <= 0) ? L[i++] : R[j++];
10    }
11    ...
12 }
```



- **Quick sort**

Sama seperti merge sort, quick sort juga menggunakan dua fungsi, fungsi utama dan fungsi pembantu.

Algoritma fungsi utama quick sort:



```
1 function quick_sort(arr, low, high):
2     if low < high:
3         pi = partition(arr, low, high)
4         quick_sort(arr, low, pi-1)
5         quick_sort(arr, pi+1, high)
```

Fungsi quick\_sort adalah fungsi utama yang menggunakan pendekatan rekursif dan divide-and-conquer. Berikut tahapannya:

- 1) Proses utama dilakukan dengan memanggil fungsi partition() untuk membagi array berdasarkan pivot.
- 2) Setelah partisi, elemen di sebelah kiri pivot akan lebih kecil, dan di sebelah kanan akan lebih besar.
- 3) Fungsi kemudian memanggil dirinya sendiri secara rekursif untuk bagian kiri dan kanan dari pivot.

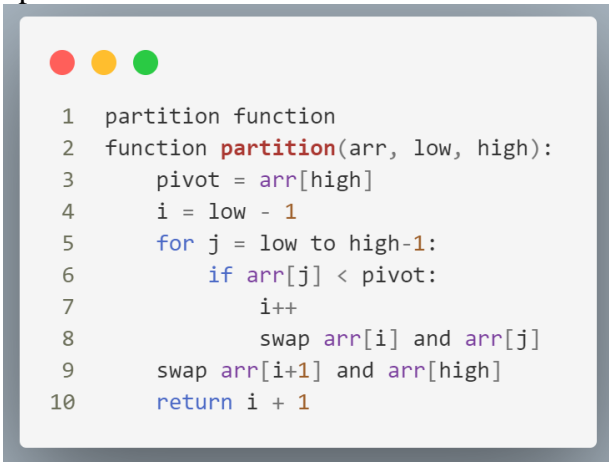
Implementasi kode program:



```
1 void quick_sort(int arr[], int low, int high) {
2     if (low < high) {
3         int pi = partition(arr, low, high);
4         quick_sort(arr, low, pi - 1);
5         quick_sort(arr, pi + 1, high);
6     }
7 }
```

Versi string (quick\_sort\_string) melakukan hal yang sama tetapi membandingkan pointer string menggunakan strcmp() dan melakukan pertukaran pointer.

Algoritma fungsi pembantu:



```
1 partition function
2 function partition(arr, low, high):
3     pivot = arr[high]
4     i = low - 1
5     for j = low to high-1:
6         if arr[j] < pivot:
7             i++
8             swap arr[i] and arr[j]
9     swap arr[i+1] and arr[high]
10    return i + 1
```

Implementasi kode program:

```
1 // Quick Sort
2 int partition(int arr[], int low, int high) {
3     int pivot = arr[high], i = (low - 1);
4     for (int j = low; j < high; j++) {
5         if (arr[j] < pivot) {
6             i++; int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
7         }
8     }
9     int temp = arr[i+1]; arr[i+1] = arr[high]; arr[high] = temp;
10    return i + 1;
11 }
```

Fungsi partition adalah fungsi pembantu dari quick\_sort. Berikut tahapannya:

- 1) Fungsi partition bertugas memindahkan semua elemen yang lebih kecil dari pivot ke kiri, dan lebih besar ke kanan.
- 2) i adalah penanda indeks akhir dari elemen-elemen yang lebih kecil dari pivot.
- 3) Setiap kali ditemukan elemen yang lebih kecil dari pivot, i akan bertambah dan elemen ditukar dengan arr[i].
- 4) Setelah seluruh elemen diproses, pivot ditukar ke posisi yang tepat (i+1).
- 5) Nilai i+1 dikembalikan sebagai indeks akhir dari elemen kecil (pivot index).
- 6) Sedangkan untuk string juga menggunakan function partition, sama seperti fungsi quick\_sort\_string dan partition\_string), penjelasannya akan sama, hanya perbandingannya menggunakan strcmp() dan pertukaran dilakukan terhadap pointer char \*.

```
1 // Partition untuk string
2 int partition_string(char *arr[], int low, int high) {
3     char *pivot = arr[high];
4     int i = low - 1;
5
6     for (int j = low; j < high; j++) {
7         if (strcmp(arr[j], pivot) < 0) {
8             i++;
9             char *temp = arr[i];
10            arr[i] = arr[j];
11            arr[j] = temp;
12        }
13    }
14
15    char *temp = arr[i + 1];
16    arr[i + 1] = arr[high];
17    arr[high] = temp;
18
19    return i + 1;
20 }
```

- **Shell sort**

Algoritma:

```
1 gap = n / 2
2 while gap > 0:
3     for i from gap to n-1:
4         temp = arr[i]
5         j = i
6         while j >= gap and arr[j - gap] > temp:
7             arr[j] = arr[j - gap]
8             j -= gap
9         arr[j] = temp
10    gap = gap / 2
```

Shell Sort adalah algoritma yang menggabungkan prinsip Insertion Sort dengan penggunaan interval (gap) untuk membandingkan dan menyisipkan elemen yang tidak bersebelahan. Berikut tahapannya:

- 1) Awalnya, elemen yang memiliki jarak gap dibandingkan dan diurutkan menggunakan prinsip insertion sort.
- 2) Nilai gap dikurangi setengah setiap iterasi hingga menjadi 1.
- 3) Ketika gap = 1, proses ini menjadi seperti Insertion Sort biasa, tetapi data sudah hampir terurut karena proses sebelumnya.

Implementasi kode program:

```
1 // Shell Sort
2 void shell_sort(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 }
```

Untuk data bertipe string:

- Perbandingan dilakukan menggunakan fungsi strcmp() karena operator relasional (>, <) tidak bisa digunakan pada string.
- Penukaran dilakukan dengan pointer char \*, bukan isi string, agar lebih efisien secara memori.

## b. Header File

File `sort_algorithms.h` berfungsi sebagai *header file* yang mendeklarasikan semua fungsi algoritma sorting yang digunakan dalam program. Berikut kode program nya:

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <time.h>
5
6  #ifndef SORT_ALGORITHMS_H
7  #define SORT_ALGORITHMS_H
8
9  void bubble_sort(int arr[], int n);
10 void selection_sort(int arr[], int n);
11 void insertion_sort(int arr[], int n);
12 void merge_sort(int arr[], int l, int r);
13 void quick_sort(int arr[], int low, int high);
14 void quick_sort_string(char *arr[], int low, int high);
15 void shell_sort(int arr[], int n);
16
17 void bubble_sort_string(char *arr[], int n);
18 void selection_sort_string(char *arr[], int n);
19 void insertion_sort_string(char *arr[], int n);
20 void merge_sort_string(char *arr[], int l, int r);
21 void quick_sort_string(char *arr[], int low, int high);
22 void shell_sort_string(char *arr[], int n);
23
24 #endif
25
```

Di bagian atas file, terdapat beberapa pustaka yang dibutuhkan pada kode program. Setelah bagian pustaka, terdapat deklarasi fungsi-fungsi sorting yang dibagi menjadi dua kelompok: untuk array bertipe `int` dan untuk array bertipe `char *` (string).

Masing-masing algoritma seperti Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan Shell Sort dideklarasikan secara terpisah baik untuk tipe data angka maupun string. Dengan pendeklarasian ini, setiap fungsi dapat digunakan di file lain (seperti `main.c`) dengan cara meng-*include* header ini, tanpa perlu menulis ulang definisinya. Hal ini menjadikan program lebih rapi dan mudah dikelola.

## c. Main function

File `main.c` merupakan file utama yang berfungsi sebagai pengendali eksekusi program. Dalam file ini dilakukan proses pembacaan data dari file eksternal, pemanggilan fungsi-fungsi sorting, serta pengukuran waktu eksekusi dan penggunaan memori untuk keperluan eksperimen. Berikut penjelasan setiap kode blok nya:

- **Header & Konstanta**

```
1  #include "sort_algorithms.h"
2  #define MAX_DATA 2000000
3  #define MAX_STRING_LEN 100
4  #define BYTE_TO_MB (1024.0 * 1024.0)
```

Bagian ini memuat header eksternal `sort_algorithms.h` yang berisi deklarasi fungsi sorting. Tiga makro didefinisikan: `MAX_DATA` sebagai batas maksimum jumlah data, `MAX_STRING_LEN` untuk panjang maksimal string, dan `BYTE_TO_MB` untuk konversi byte ke megabyte dalam penghitungan memori.

- **Fungsi `load_data`**

```
1 void load_data(const char *filename, int *arr, int count) {
2     FILE *fp = fopen(filename, "r");
3     if (!fp) {
4         perror("Gagal membuka file");
5         exit(1);
6     }
7     for (int i = 0; i < count; i++) fscanf(fp, "%d", &arr[i]);
8     fclose(fp);
9 }
10
11 void load_string_data(const char *filename, char **arr, int count) {
12     ...
13     for (int i = 0; i < count; i++) {
14         fscanf(fp, "%s", buffer);
15         arr[i] = strdup(buffer);
16     }
17     fclose(fp);
18 }
```

Fungsi `load_data` membaca file berisi data angka (`data_angka.txt`) dan menyimpannya ke dalam array `arr`. Setiap baris pada file dibaca menggunakan `fscanf` hingga mencapai jumlah `count`. Sedangkan fungsi `load_string_data` akan membaca file `data_kata.txt` dan menyimpan setiap kata ke array pointer `arr[]`. Data disalin dari buffer lokal menggunakan `strdup()` agar masing-masing string memiliki alokasi memori sendiri.

- **Fungsi Estimasi Memori**

```
1 double calculate_base_memory(int size) {
2     return (size * sizeof(int) * 2) / BYTE_TO_MB;
3 }
4 double calculate_merge_memory(int size) {
5     return (size * sizeof(int) * 4) / BYTE_TO_MB;
6 }
7 double calculate_string_memory(int size) {
8     return (size * sizeof(char *) * 2) / BYTE_TO_MB; // 2 array pointer string
9 }
```

Ketiga fungsi ini digunakan untuk menghitung estimasi penggunaan memori (dalam MB) berdasarkan jumlah data:

- ✓ `calculate_base_memory`: algoritma sorting biasa yang hanya butuh 2 array.
- ✓ `calculate_merge_memory`: untuk Merge Sort, karena butuh array tambahan.
- ✓ `calculate_string_memory`: untuk array pointer string.

- **Inisialisasi dan Pengujian di main()**

```
1  int main() {
2      int data_sizes[] = {10000, 50000, 100000, 250000, 500000,
3                          1000000, 1500000, 2000000};
4      int *data = malloc(MAX_DATA * sizeof(int));
5      int *copy = malloc(MAX_DATA * sizeof(int));
6
7      char **data_kata = malloc(MAX_DATA * sizeof(char *));
8      char **copy_kata = malloc(MAX_DATA * sizeof(char *));
9      ...
10 }
```

Fungsi main() berisi logika utama eksperimen. Program mengalokasikan memori untuk array data angka (int) dan data kata (char \*), lalu melakukan pengujian terhadap berbagai ukuran data (data\_sizes[]) secara berulang.

- **Bagian Pengujian Data Angka**

```
1  load_data("data_angka.txt", data, size);
2  double base_memory = calculate_base_memory(size);
3  double merge_memory = calculate_merge_memory(size);
4
5  printf("\n[ Angka (int) ]\n");
6  printf("Estimasi Memori Dasar (2 array int): %.2f MB\n", base_memory);
7
8  memcpy(copy, data, size * sizeof(int));
9  clock_t start = clock();
10 bubble_sort(copy, size);
11 printf("Bubble Sort      : %.2f s      | Memori: %.2f MB\n",
12        (double)(clock() - start)/CLOCKS_PER_SEC, base_memory);
13
14 // Selection Sort
15 // Insertion Sort
16 // Merge Sort
17 // Quick Sort
18 // Shell Sort
```

Untuk setiap algoritma, data angka dibaca dari file, disalin ke array copy, kemudian diurutkan menggunakan fungsi sorting yang sesuai. Waktu eksekusi diukur dengan clock() dan memori ditampilkan berdasarkan estimasi. Proses ini dilakukan untuk semua algoritma sorting.

- **Bagian Pengujian Data Angka**

Pengujian untuk data string dilakukan dengan cara serupa. Karena string adalah pointer ke karakter, setiap elemen array copy\_kata harus dialokasikan ulang (dengan strdup) sebelum digunakan untuk setiap algoritma agar tidak terjadi modifikasi langsung pada data asli. Setelah sorting selesai, semua pointer dibebaskan menggunakan free().

```

1  load_string_data("data_kata.txt", data_kata, size);
2  double str_memory = calculate_string_memory(size);
3
4  printf("\n[ Kata (string) ]\n");
5  printf("Estimasi Memori Dasar (2 array pointer): %.2f MB\n", str_memory);
6
7  for (int j = 0; j < size; j++) copy_kata[j] = strdup(data_kata[j]);
8  start = clock();
9  bubble_sort_string(copy_kata, size);
10 printf("Bubble Sort      : %.2f s      | Memori: %.2f MB\n",
11        (double)(clock() - start)/CLOCKS_PER_SEC, str_memory);
12
13 // Selection Sort
14 // Insertion Sort
15 // Merge Sort
16 // Quick Sort
17 // Shell Sort

```

- **Pembersihan Memori**

```

1  free(data);
2  free(copy);
3  free(data_kata);
4  free(copy_kata);

```

Setelah semua pengujian selesai, memori yang dialokasikan secara dinamis dibebaskan untuk mencegah kebocoran memori (memory leak).

Dengan disusunnya seluruh komponen program mulai dari header file, implementasi fungsi-fungsi sorting, hingga program utama yang menjalankan eksperimen, maka seluruh sistem telah siap digunakan untuk melakukan pengujian performa algoritma. Setelah tahap implementasi ini, langkah selanjutnya adalah mengevaluasi performa masing-masing algoritma melalui hasil eksekusi program, yang mencakup waktu proses dan estimasi penggunaan memori pada berbagai ukuran data.

#### 4. Tabel Hasil Eksperimen

##### a. Tabel Hasil Pengujian Data Angka (int)

10.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	0.20 s	0.08 mb
Selection Sort	0.07 s	0.08 mb
Insertion Sort	0.05 s	0.08 mb

Merge Sort	0.01 s	0.15 mb
Quick Sort	0.00 s	1.08 mb
Shell Sort	0.00 s	0.08 mb

50.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	7.85 s	0.38 mb
Selection Sort	2.03 s	0.38 mb
Insertion Sort	1.26 s	0.38 mb
Merge Sort	0.02 s	0.76 mb
Quick Sort	0.00 s	1.38 mb
Shell Sort	0.03 s	0.38 mb

100.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	34.70 s	0.76 mb
Selection Sort	7.72 s	0.76 mb
Insertion Sort	4.99 s	0.76 mb
Merge Sort	0.04 s	1.53 mb
Quick Sort	0.03 s	1.76 mb
Shell Sort	0.05 s	0.76 mb

250.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	210.72 s	1.91 mb
Selection Sort	48.34 s	1.91 mb
Insertion Sort	31.53 s	1.91 mb
Merge Sort	0.11 s	3.81 mb
Quick Sort	0.06 s	2.91 mb
Shell Sort	0.11 s	1.91 mb



500.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	885.32 s	3.81 mb
Selection Sort	197.86 s	3.81 mb
Insertion Sort	125.15 s	3.81 mb
Merge Sort	0.21 s	7.63 mb
Quick Sort	0.11 s	4.81 mb
Shell Sort	0.22 s	3.81 mb

1.000.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	3250.10 s	7.63 mb
Selection Sort	715.93 s	7.63 mb
Insertion Sort	472.20 s	7.63 mb
Merge Sort	0.40 s	15.26 mb
Quick Sort	0.19 s	8.63 mb
Shell Sort	0.42 s	7.63 mb

1.500.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	1369.87 s	11.44 mb
Selection Sort	1654.44 s	11.44 mb
Insertion Sort	1085.06 s	11.44 mb
Merge Sort	0.56 s	22.89 mb
Quick Sort	0.28 s	12.44 mb
Shell Sort	0.69 s	11.44 mb

2.000.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	10274.72 s	15.26 mb

Selection Sort	3322.79 s	15.26 mb
Insertion Sort	1075.26 s	15.26 mb
Merge Sort	0.39 s	30.52 mb
Quick Sort	0.21 s	16.26 mb
Shell Sort	0.46 s	15.26 mb

**b. Tabel Hasil Pengujian Data Kata (string)**

10.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	0.67 s	0.08 mb
Selection Sort	0.19 s	0.08 mb
Insertion Sort	0.11 s	0.08 mb
Merge Sort	0.00 s	0.15 mb
Quick Sort	0.01 s	1.08 mb
Shell Sort	0.00 s	0.08 mb

50.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	23.64 s	0.38 mb
Selection Sort	5.49 s	0.38 mb
Insertion Sort	2.23 s	0.38 mb
Merge Sort	0.03 s	0.76 mb
Quick Sort	0.02 s	1.38 mb
Shell Sort	0.05 s	0.38 mb

100.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	77.21 s	0.76 mb
Selection Sort	26.02 s	0.76 mb
Insertion Sort	11.15 s	0.76 mb

Merge Sort	0.07 s	1.53 mb
Quick Sort	0.05 s	1.76 mb
Shell Sort	0.10 s	0.76 mb

250.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	515.74 s	1.91 mb
Selection Sort	179.29 s	1.91 mb
Insertion Sort	82.44 s	1.91 mb
Merge Sort	0.16 s	3.81 mb
Quick Sort	0.11 s	2.91 mb
Shell Sort	0.21 s	1.91 mb

500.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	2390.55 s	3.81 mb
Selection Sort	638.22 s	3.81 mb
Insertion Sort	412.99 s	3.81 mb
Merge Sort	0.38 s	7.63 mb
Quick Sort	0.22 s	4.81 mb
Shell Sort	0.57 s	3.81 mb

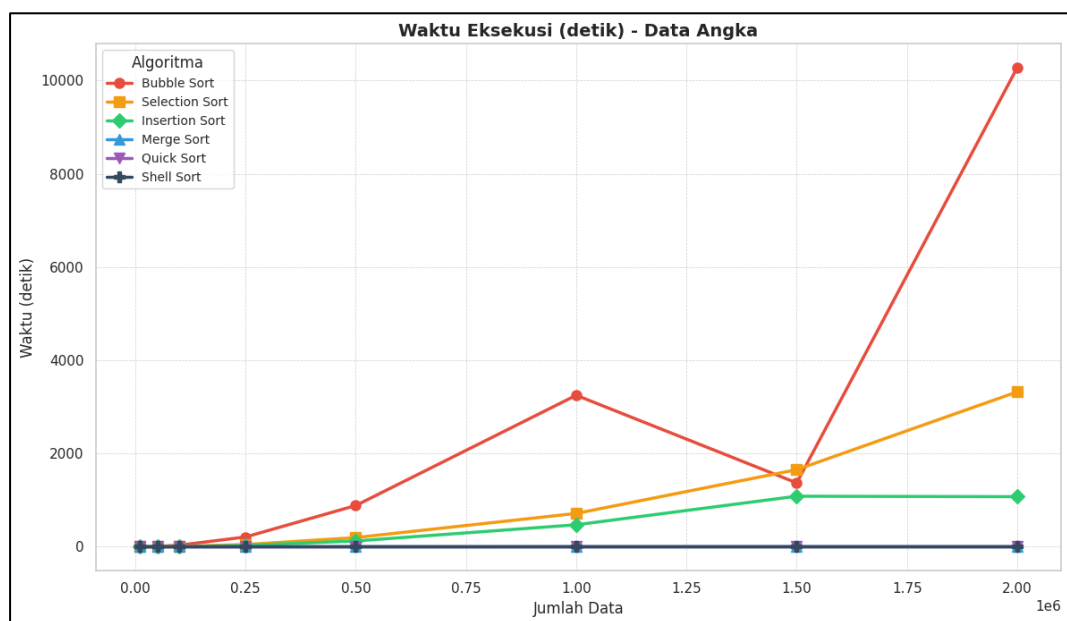
1.000.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	7839.14 s	7.63 mb
Selection Sort	3973.48 s	7.63 mb
Insertion Sort	2540.46 s	7.63 mb
Merge Sort	0.82 s	15.26 mb
Quick Sort	0.47 s	8.63 mb
Shell Sort	1.61 s	7.63 mb

1.500.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	18968.47 s	11.44 mb
Selection Sort	14200.74 s	11.44 mb
Insertion Sort	5966.74 s	11.44 mb
Merge Sort	0.70 s	22.89 mb
Quick Sort	0.45 s	12.44 mb
Shell Sort	1.55 s	11.44 mb

2.000.000 entri		
Algoritma	Waktu (s)	Memori (mb)
Bubble Sort	29835.91 s	15.26 mb
Selection Sort	19758.34 s	15.26 mb
Insertion Sort	16434.62 s	15.26 mb
Merge Sort	0.39 s	30.52 mb
Quick Sort	0.21 s	16.26 mb
Shell Sort	0.46 s	15.26 mb

## 5. Analisa Hasil

- Grafik Perbandingan Waktu Eksekusi Data Angka**



Grafik ini menunjukkan performa waktu eksekusi dari enam algoritma sorting saat mengurutkan data berupa angka (integer). Data yang diuji berjumlah mulai dari 10.000 hingga 2.000.000 entri.

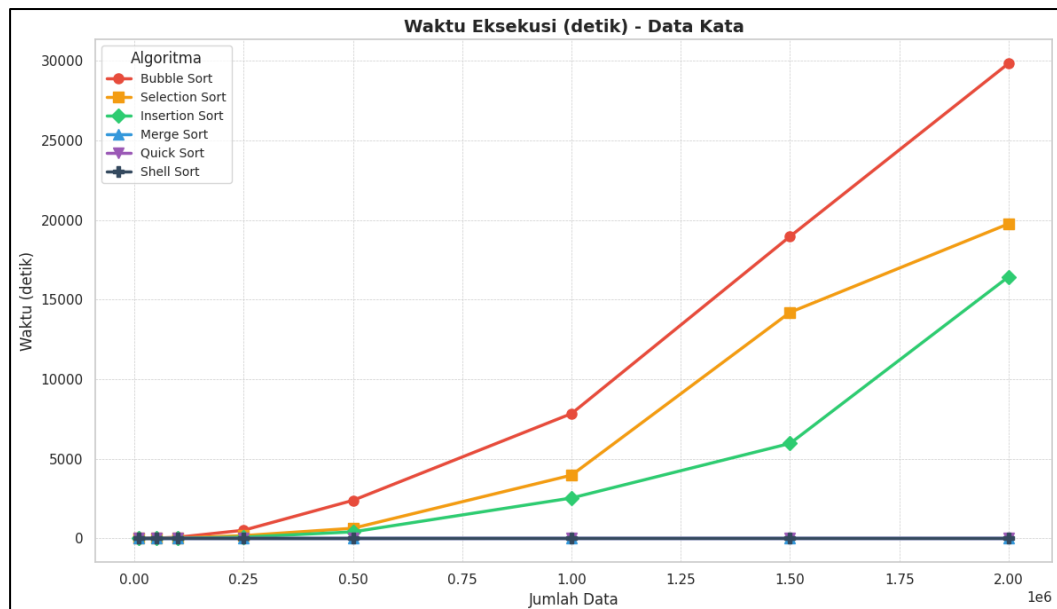
### 1. Algoritma Kuadratik ( $O(n^2)$ )

- ✓ **Bubble Sort:** Waktu eksekusi menunjukkan kenaikan tajam, mencapai lebih dari 10.000 detik pada data 2 juta angka. Kenaikan drastis menunjukkan bahwa Bubble Sort tidak efisien untuk dataset besar. Bahkan sempat terjadi fluktuasi turun di 1,5 juta data. Hal ini terjadi disebabkan oleh device yang hanya fokus melakukan sorting tanpa mengerjakan task lain, sedangkan pengujian lain dilakukan bersamaan dengan task-task lain.
- ✓ **Selection Sort:** Meskipun sedikit lebih efisien dari Bubble Sort, Selection Sort tetap tidak layak digunakan untuk data besar. Pada 2 juta data, waktu eksekusi menyentuh angka 3.000 detik.
- ✓ **Insertion Sort:** Menunjukkan performa sedikit lebih baik dari Selection Sort, stabil di angka 1.000-an detik pada 2 juta entri. Cocok untuk data kecil atau hampir terurut, tapi tidak untuk data skala besar.

### 2. Algoritma Efisien ( $O(n \log n)$ )

- ✓ **Merge Sort** dan **Quick Sort** memperlihatkan performa sangat cepat dan stabil, bahkan tidak pernah melebihi 1 detik untuk seluruh ukuran data. Quick Sort sedikit lebih cepat dari Merge Sort di hampir semua titik data.
- ✓ **Shell Sort** juga tampil efisien, dengan waktu eksekusi sangat rendah (<1 detik). Ini menjadikannya opsi yang baik jika memori terbatas.

### • Grafik Perbandingan Waktu Eksekusi Data Kata



Grafik diatas menggambarkan waktu eksekusi algoritma ketika mengurutkan data bertipe string (kata). Semua tren umum tetap terlihat, tetapi waktu eksekusinya jauh lebih tinggi dibanding pengurutan angka, karena operasi pada string (misalnya strcmp()) lebih kompleks.

## 1. Algoritma Kuadratik ( $O(n^2)$ )

- ✓ **Bubble Sort**: Performa paling buruk, mencapai lebih dari 29.000 detik pada data 2 juta kata. Waktu naik drastis mengikuti kurva eksponensial.
- ✓ **Selection Sort**: Menyentuh angka 19.000 detik, menempati posisi kedua paling lambat.
- ✓ **Insertion Sort**: Walaupun lebih cepat dibanding dua algoritma di atas, waktu eksekusinya tetap di atas 16.000 detik, tidak efisien untuk skala besar.

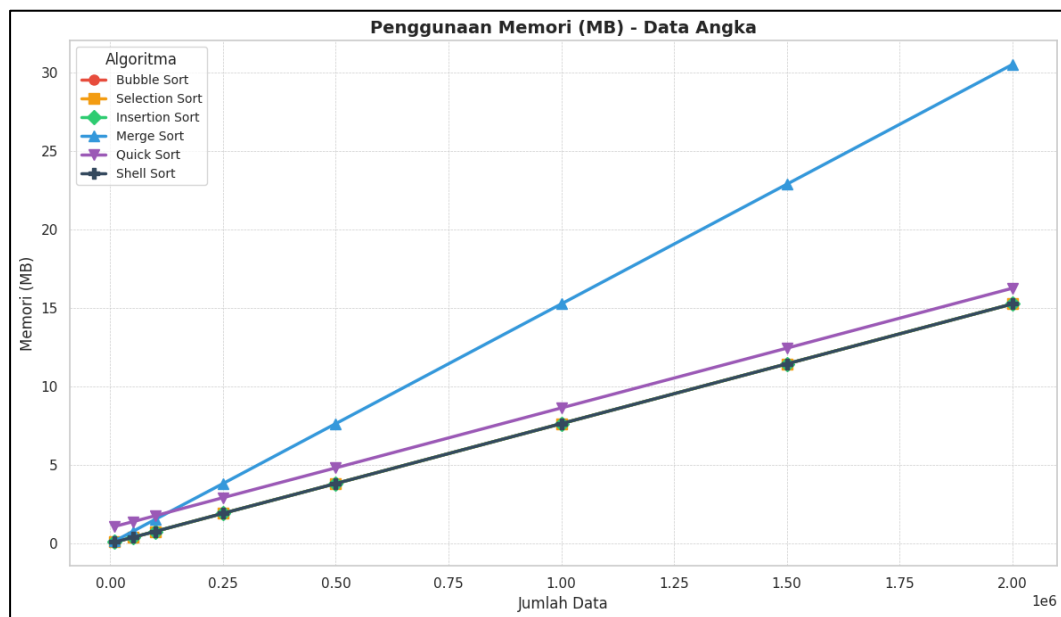
## 2. Algoritma Efisien ( $O(n \log n)$ )

- ✓ **Merge Sort** dan **Quick Sort** kembali menunjukkan dominasi. Merge Sort stabil di angka 0.39 detik untuk 2 juta kata. Quick Sort sedikit lebih cepat (sekitar 0.21 detik).
- ✓ **Shell Sort** juga tampil baik, meskipun sedikit lebih lambat dari Quick dan Merge. Waktu tertinggi hanya 0.46 detik.

### Perbedaan dengan Data Angka

Pengurutan string membutuhkan perbandingan karakter demi karakter, sehingga eksekusi jauh lebih lama. Namun, pola efisiensi tetap sama: algoritma  $O(n \log n)$  jauh lebih unggul.

#### • Grafik Perbandingan Penggunaan Memori Data Angka



Grafik ketiga menunjukkan penggunaan memori oleh masing-masing algoritma saat menangani data angka. Karena penggunaan memori untuk angka dan kata sama dalam pendekatannya (berbasis jumlah data dan struktur array), maka grafik ini juga berlaku untuk data string.

## 1. Penggunaan Memori Berdasarkan Algoritma

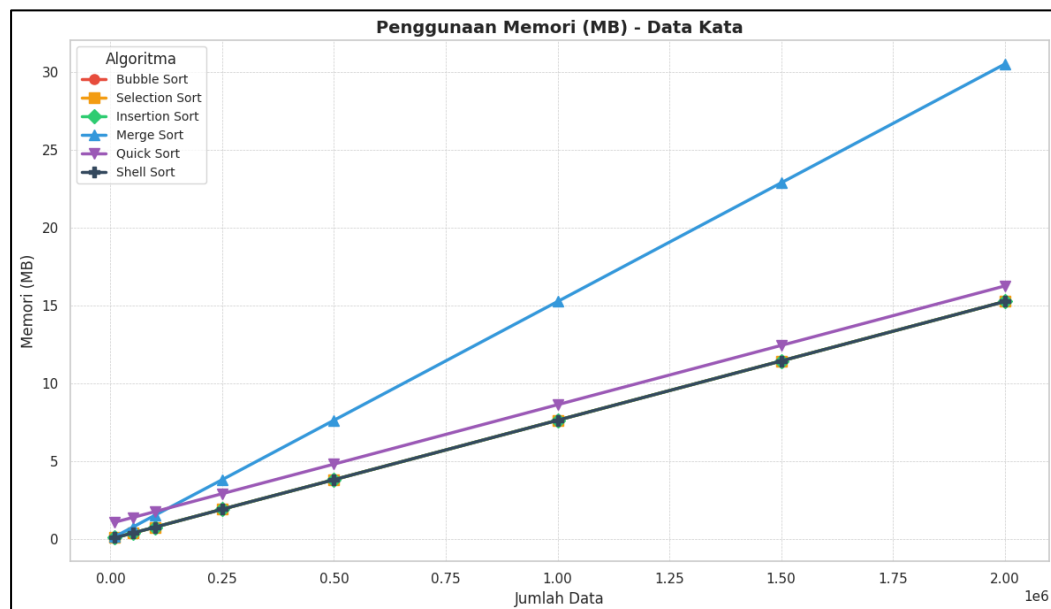
- ✓ **Merge Sort**: Paling boros memori karena menggunakan array tambahan saat proses merge. Penggunaan memori mencapai 30 MB pada 2 juta entri.

- ✓ Quick Sort: Memiliki konsumsi memori yang tinggi tapi lebih rendah dari Merge Sort (sekitar 16 MB pada 2 juta data).
- ✓ Shell Sort, Insertion Sort, Selection Sort, dan Bubble Sort: Konsumsi memori identik karena hanya menggunakan array utama. Memori maksimum hanya sekitar 15 MB, jauh lebih hemat.

## 2. Pola Pertumbuhan

- ✓ Semua algoritma memperlihatkan pertumbuhan linear terhadap jumlah data.
- ✓ Perbedaan utama terletak pada apakah algoritma menggunakan array tambahan atau tidak.
- ✓ Meskipun Merge Sort paling efisien dalam waktu, penggunaan memori tinggi bisa jadi pertimbangan pada sistem dengan sumber daya terbatas.

### • Grafik Perbandingan Penggunaan Memori Data Kata



Grafik terakhir ini menunjukkan penggunaan memori oleh masing-masing algoritma saat menangani data kata. Karena penggunaan memori untuk angka dan kata sama dalam pendekatannya (berbasis jumlah data dan struktur array), maka grafik ini dan penjelasannya sama persis dengan grafik penggunaan memori pada data angka.

Dari hasil analisis keempat grafik diatas dapat kita simpulkan bahwa algoritma dengan efisiensi tinggi cenderung memberikan performa yang stabil dan cepat, sementara algoritma sederhana kurang cocok digunakan untuk data berukuran besar karena waktu eksekusinya meningkat drastis. Selain itu, pemilihan algoritma juga perlu mempertimbangkan penggunaan memori, karena beberapa algoritma membutuhkan alokasi tambahan yang signifikan.

## 6. Kesimpulan

Berdasarkan hasil eksperimen dan analisis terhadap enam algoritma pengurutan **Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan Shell Sort** dapat disimpulkan bahwa pemilihan algoritma sorting sangat memengaruhi efisiensi proses, terutama pada data skala besar. Beberapa poin utama yang dapat disimpulkan adalah sebagai berikut:

### 1. Efisiensi Waktu Eksekusi:

- Algoritma Merge Sort dan Quick Sort terbukti paling efisien, baik untuk data bertipe angka maupun string.
- Shell Sort menawarkan waktu eksekusi yang cukup baik dan konsisten, menjadi alternatif efisien jika ingin menghindari penggunaan memori tambahan.
- Sebaliknya, algoritma Bubble Sort, Selection Sort, dan Insertion Sort menunjukkan waktu eksekusi yang sangat tinggi, terutama pada dataset besar. Bubble Sort menjadi yang paling lambat dari semua.

### 2. Performa pada Tipe Data Berbeda:

- Secara umum, sorting pada data string membutuhkan waktu lebih lama dibanding data angka karena melibatkan operasi perbandingan karakter demi karakter (menggunakan strcmp())

### 3. Penggunaan Memori:

- Merge Sort adalah yang paling boros memori karena membutuhkan array tambahan dalam proses merging.
- Quick Sort juga menggunakan lebih banyak memori dibanding algoritma lain, namun masih lebih hemat dibanding Merge Sort.
- Shell Sort, Insertion Sort, Selection Sort, dan Bubble Sort merupakan algoritma yang hemat memori karena hanya menggunakan array utama

### 4. Pola Skalabilitas:

- Hasil pengujian memperlihatkan bahwa hanya algoritma dengan kompleksitas  $O(n \log n)$  yang mampu menangani jumlah data besar (hingga jutaan entri) secara efisien baik dari sisi waktu maupun pertumbuhan memori.
- Algoritma  $O(n^2)$  tidak direkomendasikan untuk dataset besar dan hanya layak digunakan untuk data kecil atau keperluan edukasi.