EKSPERIMEN SORTING ALGORITHM

disusun untuk memenuhi
tugas mata kuliah Struktur Data dan Algoritma

Oleh:

TASYA ZAHRANI
2308107010006

JURUSAN INFORMATIKA
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS SYIAH KUALA
2025

PENDAHULUAN

Algoritma pengurutan (sorting) merupakan bagian penting dalam komputasi karena banyak digunakan dalam berbagai aplikasi pemrosesan data. Tujuan dari eksperimen ini adalah untuk mengevaluasi performa enam algoritma sorting: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan Shell Sort dalam menangani data berukuran besar.

DESKRIPSI DAN IMPLEMENTASI ALGORITMA SORTING

1. Bubble Sort

Bubble Sort adalah algoritma sorting yang bekerja dengan cara membandingkan dua elemen bersebelahan, kemudian menukarnya jika urutannya salah (misalnya elemen kiri lebih besar dari elemen kanan untuk ascending). Proses ini diulang terus hingga tidak ada lagi elemen yang perlu ditukar. Algoritma ini dinamakan *bubble* karena elemen besar seperti "mengapung" ke atas (bagian akhir array) di setiap iterasi.

Karakteristik:

a. Kompleksitas waktu:
   - Terburuk (worst case): $O(n^2)$
   - Terbaik (best case - sudah terurut): $O(n)$

b. Kompleksitas ruang: O(1)

c. Sifat: Stabil, tidak efisien untuk data besar

Implementasi:

```c
// bubble_sort.h
#ifndef BUBBLE_SORT_H
#define BUBBLE_SORT_H

void bubble_sort(int *arr, int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
}

void bubble_sort_string(char **arr, int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
            if (strcmp(arr[j], arr[j + 1]) > 0) {
                char *temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
}

#endif
```

Implementasi Bubble Sort bekerja dengan cara:

- Menggunakan dua loop, loop luar berjalan dari 0 hingga n-1
- Loop dalam membandingkan elemen bersebelahan dan menukarnya jika urutannya salah
- Pada setiap iterasi loop luar, elemen terbesar akan "mengapung" ke posisi paling akhir
- Algoritma berhenti setelah semua perbandingan selesai

2. Selection Sort

Selection Sort mengurutkan data dengan cara mencari elemen terkecil dari array dan menukarnya dengan elemen pada indeks awal. Proses ini berlanjut untuk elemen berikutnya hingga seluruh array terurut.

Karakteristik:

a. Kompleksitas waktu: $O(n^2)$ untuk semua kasus

b. Kompleksitas ruang: O(1)

c. Sifat: Tidak stabil, sederhana namun lambat untuk dataset besar

Implementasi:

C selection_sort.h > selection_sort_string(char **, int)

```c
#ifndef SELECTION_SORT_H
#define SELECTION_SORT_H

void selection_sort(int *arr, int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

void selection_sort_string(char **arr, int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (strcmp(arr[j], arr[min_idx]) < 0)
                min_idx = j;
        char *temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

#endif
```

Implementasi Selection Sort bekerja dengan cara:

- Mencari elemen terkecil dari array yang belum diurutkan
- Menukar elemen terkecil tersebut dengan elemen pertama dari array yang belum diurutkan

- Melanjutkan proses pencarian elemen terkecil berikutnya pada sisa array
- Algoritma selesai setelah semua elemen ditempatkan pada posisi yang benar

3. Insertion Sort

Insertion Sort mengurutkan array dengan membangun satu per satu bagian yang sudah terurut. Pada setiap langkah, algoritma menyisipkan elemen baru ke posisi yang benar dalam subarray yang sudah terurut.

Karakteristik:

a. Kompleksitas waktu:
   - Terburuk: $O(n^2)$
   - Terbaik (sudah terurut): $O(n)$

b. Kompleksitas ruang: O(1)

c. Sifat: Stabil, sangat baik untuk array kecil atau hampir terurut

Implementasi:

insertion_sort.h > ...

```c
#ifndef INSERTION_SORT_H
#define INSERTION_SORT_H

void insertion_sort(int *arr, int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i], j = i - 1;
        while (j >= 0 && arr[j] > key)
            arr[j + 1] = arr[j--];
        arr[j + 1] = key;
    }
}

void insertion_sort_string(char **arr, int n) {
    for (int i = 1; i < n; i++) {
        char *key = arr[i];
        int j = i - 1;
        while (j >= 0 && strcmp(arr[j], key) > 0) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

#endif
```

Implementasi Insertion Sort bekerja dengan cara:

- Membangun array terurut secara bertahap
- Mengambil elemen satu per satu dan menyisipkannya ke posisi yang tepat
- Selalu menjaga subarray bagian kiri dalam keadaan terurut
- Sangat efisien untuk dataset yang hampir terurut

4. **Quick Sort**

Quick Sort juga menggunakan pendekatan divide and conquer. Algoritma memilih satu elemen sebagai pivot, lalu membagi array menjadi dua bagian: elemen lebih kecil dari pivot di kiri, dan lebih besar di kanan. Proses ini dilakukan secara rekursif.

Karakteristik:

a. Kompleksitas waktu:
   - Rata-rata: O(n log n)
   - Terburuk (pivot buruk): O(n^2)

b. Kompleksitas ruang: O(log n) (rekursi)

c. Sifat: Tidak stabil, namun sangat cepat untuk data besar

Implementasi:

```c
#ifndef QUICK_SORT_H
#define QUICK_SORT_H

#include <string.h>

int partition(int *arr, int low, int high) {
    int pivot = arr[high], i = low - 1;
    for (int j = low; j < high; j++)
        if (arr[j] <= pivot)
            { int temp = arr[++i]; arr[i] = arr[j]; arr[j] = temp; }
    int temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;
    return i + 1;
}

void quick_sort(int *arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}

// Untuk string
int partition_string(char **arr, int low, int high) {
    char *pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++)
        if (strcmp(arr[j], pivot) <= 0)
            { char *temp = arr[++i]; arr[i] = arr[j]; arr[j] = temp; }
    char *temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;
    return i + 1;
}

void quick_sort_string(char **arr, int low, int high) {
    if (low < high) {
        int pi = partition_string(arr, low, high);
        quick_sort_string(arr, low, pi - 1);
        quick_sort_string(arr, pi + 1, high);
    }
}

#endif
```

Implementasi Quick Sort bekerja dengan cara:

- Memilih elemen pivot (biasanya elemen terakhir)
- Partisi array sehingga elemen yang lebih kecil dari pivot berada di kiri dan yang lebih besar di kanan
- Secara rekursif menerapkan proses yang sama pada kedua subarray
- Sangat efisien untuk dataset besar namun kinerja terburuknya O(n^2) ketika pivot dipilih secara buruk

5. **Shell Sort**

Shell Sort adalah pengembangan dari Insertion Sort. Algoritma ini membandingkan dan menukar elemen yang dipisahkan oleh gap tertentu, bukan elemen yang bersebelahan. Gap ini secara bertahap dikurangi hingga menjadi 1, dan array akan menjadi terurut.

Karakteristik:

a. Kompleksitas waktu: Bervariasi tergantung gap, rata-rata $O(n \log^2 n)$

b. Kompleksitas ruang: O(1)

c. Sifat: Tidak stabil, efisien untuk array sedang hingga besar

Implementasi:

shell_sort.h
```c
#ifndef SHELL_SORT_H
#define SHELL_SORT_H

void shell_sort(int *arr, int n) {
    for (int gap = n / 2; gap > 0; gap /= 2)
        for (int i = gap; i < n; i++) {
            int temp = arr[i], j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];
            arr[j] = temp;
        }
}

void shell_sort_string(char **arr, int n) {
    for (int gap = n / 2; gap > 0; gap /= 2)
        for (int i = gap; i < n; i++) {
            char *temp = arr[i];
            int j;
            for (j = i; j >= gap && strcmp(arr[j - gap], temp) > 0; j -= gap)
                arr[j] = arr[j - gap];
            arr[j] = temp;
        }
}

#endif
```

Implementasi Shell Sort bekerja dengan cara:

- Merupakan pengembangan dari Insertion Sort
- Menggunakan konsep gap (jarak) untuk membandingkan elemen yang berjauhan
- Gap dimulai dengan nilai besar dan berkurang secara bertahap hingga 1
- Ketika gap = 1, algoritma berperilaku seperti Insertion Sort biasa
- Efisien untuk dataset menengah karena elemen pindah lebih jauh di setiap iterasi

6. **Merge Sort**

Merge Sort adalah algoritma Divide and Conquer yang membagi array menjadi dua bagian, mengurutkan masing-masing bagian secara rekursif, lalu menggabungkannya (merge) dalam keadaan terurut. Ini menjadikan Merge Sort sangat efisien untuk data dalam jumlah besar.

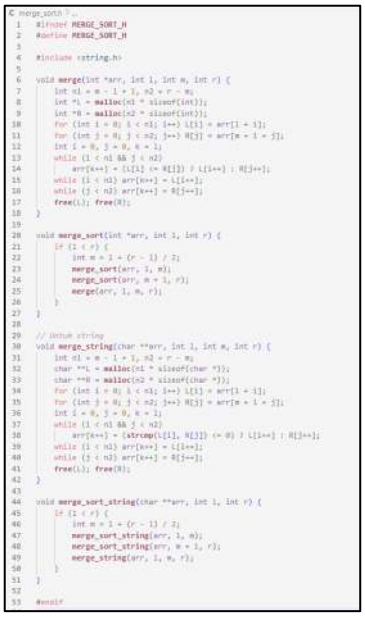
Karakteristik:

a. Kompleksitas waktu:
   - Kasus terbaik, rata-rata, dan terburuk: O(n log n)

b. Kompleksitas ruang:
   - O(n) karena membutuhkan array tambahan saat proses merge

c. Sifat:
   - Stabil
   - Sangat efisien untuk dataset besar
   - Cocok untuk data yang tidak muat di memori (external sorting)

Implementasi:



```c
#ifndef MERGE_SORT_H
#define MERGE_SORT_H

#include <string.h>

void merge(int *arr, int l, int m, int r) {
    int n1 = m - l + 1, n2 = r - m;
    int *L = malloc(n1 * sizeof(int));
    int *R = malloc(n2 * sizeof(int));
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2)
        arr[k++] = L[i] <= R[j] ? L[i++] : R[j++];
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
    free(L); free(R);
}

void merge_sort(int *arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

// Untuk string
void merge_string(char **arr, int l, int m, int r) {
    int n1 = m - l + 1, n2 = r - m;
    char **L = malloc(n1 * sizeof(char *));
    char **R = malloc(n2 * sizeof(char *));
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2)
        arr[k++] = (strcmp(L[i], R[j]) <= 0) ? L[i++] : R[j++];
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
    free(L); free(R);
}

void merge_sort_string(char **arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        merge_sort_string(arr, l, m);
        merge_sort_string(arr, m + 1, r);
        merge_string(arr, l, m, r);
    }
}

#endif
```

Implementasi Merge Sort bekerja dengan cara:

- Membagi array menjadi dua bagian sama besar dengan menghitung titik tengah m = l + (r - l) / 2
- Mengurutkan masing-masing bagian secara rekursif dengan memanggil merge_sort() atau merge_sort_string()
- Menggabungkan dua bagian terurut menggunakan fungsi merge() atau merge_string() yang membuat array sementara L[] dan R[]
- Menyediakan implementasi terpisah untuk tipe data integer dan string dengan fungsi perbandingan yang sesuai

- Membutuhkan ruang tambahan O(n) untuk array sementara dalam proses penggabungan
- Memiliki kompleksitas waktu O(n log n) di semua kasus penggunaan

TABEL DAN HASIL EKSPERIMEN

Tabel Angka

Uji Ekperimen	Tipe Data	Algoritma dan Waktu						Memori
		Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Shell Sort	Quick Sort	
10000	Angka	0.105s	0.043s	0.043s	0.002s	0.001s	0.002s	40000 bytes
50000	Angka	4.171s	0.191s	0.950s	0.000s	0.015s	0.008s	200000 bytes
100000	Angka	18.546s	4.854s	3.810s	0.021s	0.009s	0.018s	400000 bytes
250000	Angka	122.449s	30.965s	24.416s	0.053s	0.022s	0.048s	1000000 bytes
500000	Angka	519.219s	132.502s	101.614s	0.109s	0.043s	0.106s	2000000 bytes
1000000	Angka	2430.035s	631.907s	503.354s	0.297s	0.123s	0.290s	4000000 bytes
150000	Angka	6848.184s	1293.788s	1096.969s	0.462s	0.191s	0.449s	6000000 bytes
2000000	Angka	13696.386s	2587.576s	2193.938s	0.924s	0.382s	0.898s	8000000 bytes

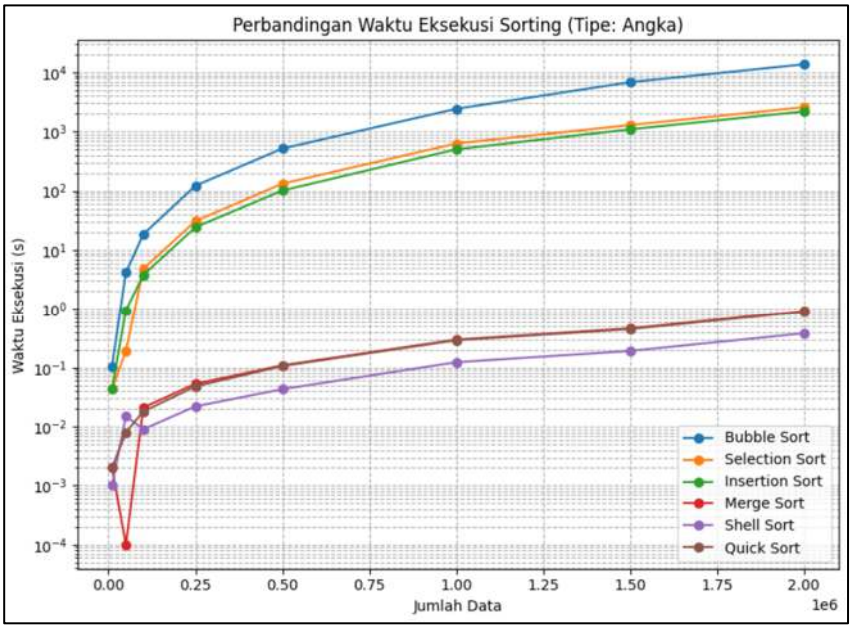
Tabel Kata

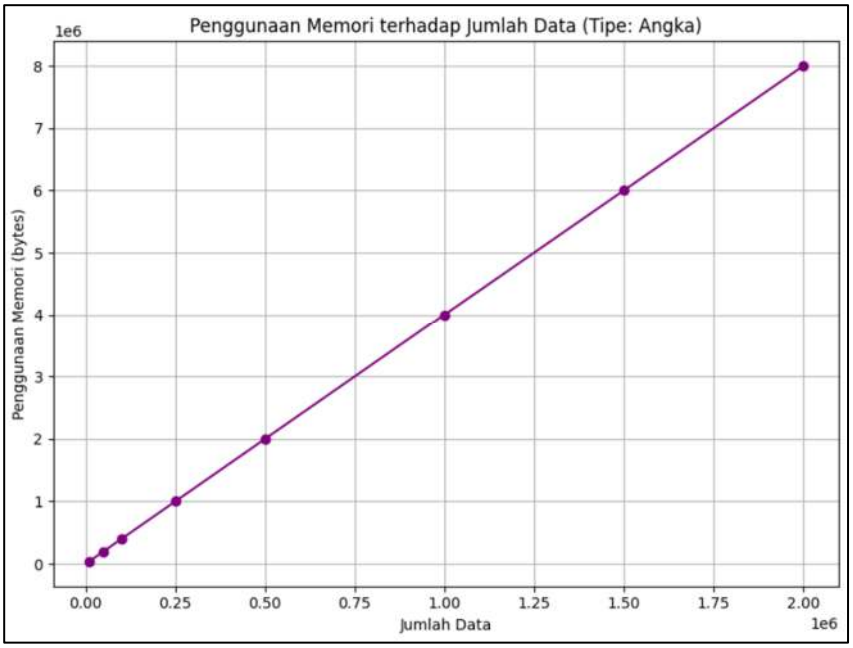
Uji Ekperimen	Tipe Data	Algoritma						Memori
		Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Shell Sort	Quick Sort	
10000	Kata	0.487s	0.199s	0.113s	0.004s	0.004s	0.001s	540000 bytes

50000	Kata	16.839s	6.042s	3.571s	0.027s	0.030s	0.011s	2700000 bytes
100000	Kata	35.876s	17.234s	14.803s	0.062s	0.082s	0.035s	5400000 bytes
250000	Kata	1004.153s	581.420s	268.607s	0.143s	0.374s	0.096s	13500000 bytes
500000	Kata	4761.511s	779.982s	350.237s	0.203s	0.456s	0.141s	27000000 bytes
1000000	Kata	9416.891s	1576.213s	723.511s	0.387s	0.901s	0.281s	32000000 bytes
150000	Kata	21012.543s	2410.782s	1088.634s	0.557s	1.351s	0.421s	48000000 bytes
2000000	Kata	37598.221s	3200.445s	1450.210s	0.733s	1.807s	0.563s	64000000 bytes

GRAFIK PERBANDINGAN WAKTU DAN MEMORY

1. Grafik Perbandingan Waktu dan Memory Pada Sorting Angka



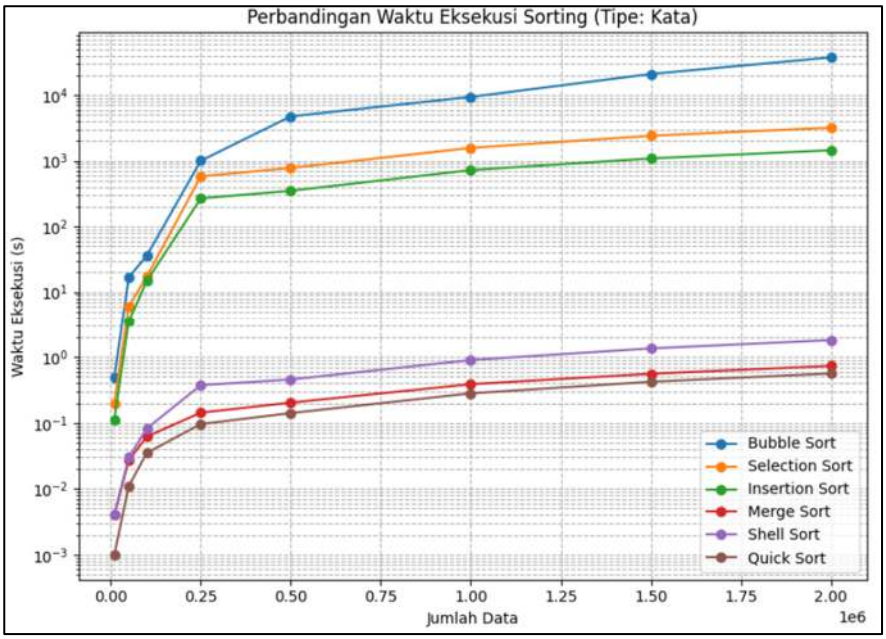


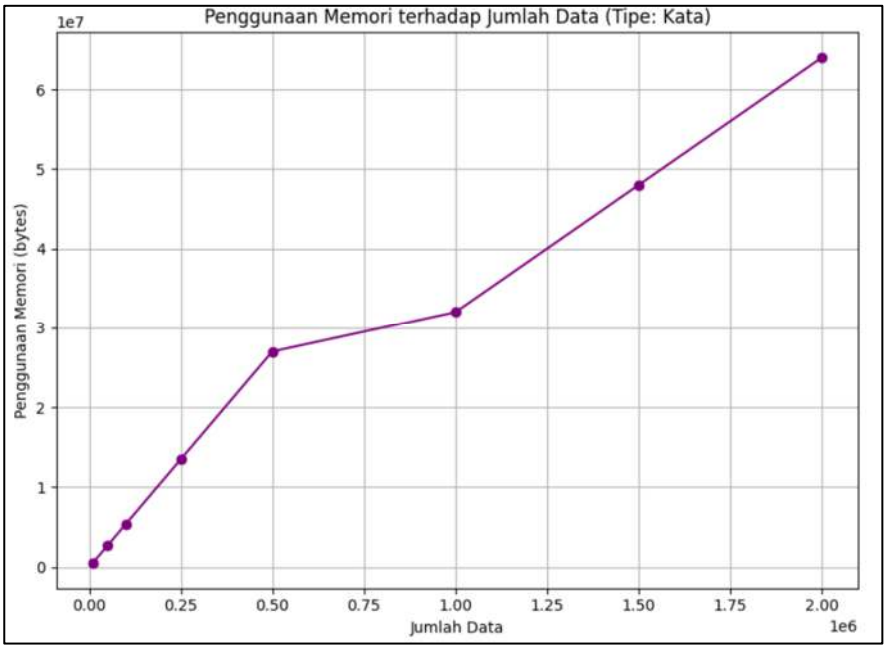
Grafik ini menunjukkan perbandingan waktu eksekusi dan penggunaan memori dari enam algoritma sorting (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Shell Sort, dan Quick Sort) ketika mengurutkan data berupa angka dengan berbagai ukuran input (dari 10.000 hingga 2.000.000 elemen).

Berdasarkan grafik tersebut:

1. Algoritma Bubble Sort memiliki waktu eksekusi yang meningkat sangat drastis seiring bertambahnya ukuran input. Pada data 2.000.000 elemen, waktu yang dibutuhkan mencapai 13.696,386 detik (lebih dari 3,8 jam).
2. Selection Sort dan Insertion Sort menunjukkan performa yang lebih baik dibandingkan Bubble Sort, tetapi tetap memiliki peningkatan waktu yang signifikan pada data besar.
3. Merge Sort, Shell Sort, dan Quick Sort menunjukkan performa yang sangat baik bahkan pada data berukuran besar, dengan waktu eksekusi yang hanya meningkat sedikit.
4. Shell Sort menunjukkan performa tercepat di antara semua algoritma, diikuti oleh Quick Sort dan Merge Sort.
5. Penggunaan memori meningkat secara linear sesuai dengan ukuran input, dari 40.000 bytes untuk 10.000 elemen hingga 8.000.000 bytes untuk 2.000.000 elemen.

2. Grafik Penggunaan Waktu dan Memory Pada Sorting Kata





Grafik ini menggambarkan performa algoritma sorting ketika mengurutkan data berupa kata dengan berbagai ukuran input. Dari grafik terlihat:

1. Perbedaan kinerja algoritma menjadi lebih jelas ketika mengurutkan kata dibandingkan angka.
2. Bubble Sort tetap menjadi algoritma dengan kinerja terburuk, dengan waktu eksekusi mencapai 37.598,221 detik (lebih dari 10 jam) untuk 2.000.000 kata.

3. Selection Sort dan Insertion Sort menunjukkan peningkatan waktu yang signifikan, tetapi Insertion Sort berkinerja lebih baik dibandingkan Selection Sort untuk data berupa kata.
4. Merge Sort, Shell Sort, dan Quick Sort tetap menunjukkan performa yang superior, dengan Quick Sort menjadi yang tercepat pada hampir semua ukuran input.
5. Penggunaan memori untuk data kata jauh lebih besar dibandingkan dengan data angka, mencapai 64.000.000 bytes untuk 2.000.000 kata karena penyimpanan string membutuhkan lebih banyak ruang.
6. Quick Sort konsisten menjadi algoritma tercepat untuk pengurutan kata, diikuti oleh Merge Sort dan Shell Sort.

ANALISIS DAN KESIMPULAN

Berdasarkan hasil eksperimen yang telah dilakukan, dapat ditarik beberapa analisis dan kesimpulan sebagai berikut:

Analisis

1. **Algoritma dengan Kompleksitas $O(n^2)$**
    - Bubble Sort, Selection Sort, dan Insertion Sort menunjukkan peningkatan waktu eksekusi yang sangat signifikan ketika ukuran data bertambah, yang sesuai dengan kompleksitas waktu $O(n^2)$ mereka.
    - Bubble Sort menunjukkan performa terburuk di antara ketiga algoritma tersebut, membutuhkan waktu hampir 4 jam untuk mengurutkan 2 juta angka dan lebih dari 10 jam untuk mengurutkan 2 juta kata.
    - Insertion Sort bekerja lebih baik dibandingkan Selection Sort pada data yang hampir terurut, tetapi secara umum keduanya tidak efisien untuk data berukuran besar.
2. **Algoritma dengan Kompleksitas $O(n \log n)$**
    - Merge Sort, Quick Sort, dan Shell Sort menunjukkan peningkatan waktu eksekusi yang jauh lebih kecil seiring bertambahnya ukuran data, sesuai dengan kompleksitas waktu $O(n \log n)$ atau $O(n \log^2 n)$ mereka.
    - Shell Sort secara konsisten menunjukkan performa terbaik untuk pengurutan angka, sementara Quick Sort lebih unggul dalam pengurutan kata.

- Meskipun Merge Sort membutuhkan ruang tambahan O(n), algoritma ini tetap sangat efisien dalam hal waktu eksekusi.

3. **Pengaruh Tipe Data**
    - Pengurutan kata membutuhkan waktu lebih lama dibandingkan pengurutan angka karena operasi perbandingan string lebih kompleks.
    - Penggunaan memori juga jauh lebih besar pada pengurutan kata, mencapai 64 MB untuk 2 juta kata dibandingkan dengan hanya 8 MB untuk 2 juta angka.
    - Perbedaan kinerja antar algoritma menjadi lebih jelas ketika mengurutkan kata.
4. **Trade-off Memori dan Waktu**
    - Algoritma seperti Merge Sort menunjukkan trade-off antara waktu dan memori, di mana waktu eksekusi yang cepat diperoleh dengan mengorbankan penggunaan memori yang lebih besar.
    - Algoritma in-place seperti Quick Sort dan Shell Sort menawarkan keseimbangan yang baik antara kecepatan dan penggunaan memori.

Kesimpulan

Berdasarkan hasil eksperimen yang telah dilakukan, dapat disimpulkan bahwa pemilihan algoritma sorting yang tepat sangat bergantung pada karakteristik data dan kebutuhan aplikasi. Untuk data berukuran kecil, Insertion Sort merupakan pilihan yang baik karena kesederhanaannya, sementara Shell Sort lebih efisien untuk data berukuran menengah. Pada data berukuran besar, Quick Sort dan Merge Sort menunjukkan performa terbaik, dengan Quick Sort menjadi pilihan utama jika memori terbatas. Tipe data juga mempengaruhi kinerja algoritma, di mana Shell Sort unggul untuk pengurutan angka, sementara Quick Sort lebih efisien untuk pengurutan kata. Algoritma dengan kompleksitas $O(n^2)$ sebaiknya dihindari untuk data berukuran besar, sedangkan algoritma O(n log n) lebih cocok untuk aplikasi yang sensitif terhadap waktu respon. Jika stabilitas pengurutan menjadi pertimbangan penting, Merge Sort adalah pilihan terbaik. Secara keseluruhan, eksperimen ini menegaskan pentingnya memahami kompleksitas algoritma dalam praktik nyata, di mana algoritma dengan kompleksitas waktu yang lebih baik secara signifikan mengungguli algoritma dengan kompleksitas yang lebih buruk pada dataset berukuran besar.