Laporan Mata Kuliah Struktur Data dan Algoritma

SORTING ALGORITHM

disusun untuk memenuhi
tugas IV mata kuliah Struktur Data dan Algoritma

Oleh:

TINSARI RAUHANA
2308107010038

JURUSAN INFORMATIKA
FAKULTAS METEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS SYIAH KUALA
DARUSSALAM, BANDA ACEH
2025

1. Deskripsi Algoritma dan Cara Implementasi

1.1 Deskripsi Umum

Program ini dibangun menggunakan bahasa C dan disusun secara modular melalui pemisahan kode ke dalam dua berkas utama, yaitu:

a. **sorting.h**: File header yang berisi semua fungsi algoritma sorting yang diuji, baik untuk data angka (int) maupun data kata (string). Setiap algoritma dibuat dalam fungsi tersendiri dan sudah diberi komentar penjelasan di atasnya. Fungsi-fungsi ini bisa dipanggil dari file utama sesuai kebutuhan..
b. **main.c**: File utama program yang mengatur jalannya pengujian. Di dalamnya terdapat fungsi main, daftar ukuran data uji, serta fungsi untuk membaca data dari file (data_angka.txt dan data_kata.txt). Program juga memiliki fungsi benchmark untuk menghitung waktu eksekusi dan memori. Tersedia menu interaktif agar pengguna bisa memilih jenis data dan ukuran data yang ingin diuji.

Program dirancang untuk menerima data dari file yang telah dihasilkan sebelumnya (menggunakan fungsi pembangkit data acak). Data akan dibaca, kemudian disalin ke array baru untuk setiap algoritma sorting agar pengukuran performa tidak saling mempengaruhi.

1.2 Implementasi Algoritma

a. Bubble Sort

Algoritma ini bekerja dengan cara membandingkan dua elemen berurutan dalam array dan menukarnya apabila urutannya salah. Proses ini diulang terus hingga seluruh elemen dalam array berada pada posisi yang tepat. Kompleksitas waktu dari algoritma ini adalah **O(n²)**. Bentuk implementasi:

```
// Bubble Sort
// Prinsip: Bandingkan dua elemen yang bersebelahan dan tukar kalau salah urutan.
// Diulang terus sampai semua elemen sudah pada tempatnya.
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 tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
}
```

Gambar 1.2.1. Potongan Kode Bubble Sort data angka.

```c
// Bubble Sort (String)
// Prinsip: Bandingkan dua kata bersebelahan, tukar kalau urutan abjadnya
salah.
void bubble_sort_str(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 *tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
            }
}
```

Gambar 1.2.2. Potongan Kode Bubble Sort data kata.

b. Selection Sort

Pada setiap iterasi, algoritma ini mencari elemen terkecil dari bagian array yang belum terurut dan menukarnya dengan elemen di posisi saat ini. Meskipun jumlah perbandingan tetap tinggi, jumlah pertukaran cenderung lebih sedikit dibanding Bubble Sort. Kompleksitas waktunya **O(n²)**.

```c
// Selection Sort
// Prinsip: Cari elemen terkecil dari sisa data, lalu tukar dengan posisi
sekarang.
void selection_sort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min = i;
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[min])
                min = j;
        int tmp = arr[min];
        arr[min] = arr[i];
        arr[i] = tmp;
    }
}
```

Gambar 1.2.3. Potongan Kode Selection Sort data angka.

```c
// Selection Sort (String)
// Prinsip: Cari kata terkecil (secara abjad), tukar dengan posisi
sekarang.
void selection_sort_str(char **arr, int n) {
    for (int i = 0; i < n - 1; i++) {
        int min = i;
        for (int j = i + 1; j < n; j++)
            if (strcmp(arr[j], arr[min]) < 0)
                min = j;
        char *tmp = arr[min];
        arr[min] = arr[i];
        arr[i] = tmp;
    }
}
```

Gambar 1.2.4. Potongan Kode Selection Sort data kata.

c. Insertion Sort

Insertion Sort membangun array yang sudah terurut satu per satu. Untuk setiap elemen, algoritma akan menyisipkannya ke posisi yang sesuai pada bagian array yang sudah terurut. Efisien pada dataset kecil atau hampir terurut. Kompleksitas waktunya **O(n²)**.

```c
// Insertion Sort
// Prinsip: Ambil satu per satu data, lalu sisipkan ke posisi yang tepat
di bagian data yang sudah terurut.
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];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Gambar 1.2.5. Potongan Kode Insertion Sort data angka.

```c
// Insertion Sort (String)
// Prinsip: Ambil satu kata, lalu sisipkan ke posisi yang tepat di bagian yang sudah urut.
void insertion_sort_str(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;
    }
}
```

Gambar 1.2.6. Potongan Kode Insertion Sort data kata.

d. Merge Sort

Merge Sort menggunakan pendekatan **divide-and-conquer**, yaitu memecah array menjadi dua bagian, mengurutkan masing-masing bagian secara rekursif, lalu menggabungkannya dalam satu array terurut. Meskipun stabil dan efisien, algoritma ini membutuhkan ruang tambahan untuk proses merge. Kompleksitas waktunya **O(n log n)**.

```c
// Merge Sort
// Prinsip: Pecah array jadi dua bagian, urutkan masing-masing, lalu gabung jadi satu secara berurutan.
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++) L[i] = arr[l+i];
    for (int i = 0; i < n2; i++) R[i] = arr[m+1+i];

    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++];
}
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);
    }
}
```

Gambar 1.2.7. Potongan Kode Merge Sort data angka.

```c
// Merge Sort (String)
// Prinsip: Pecah daftar kata menjadi bagian kecil, urutkan, lalu
gabungkan lagi secara terurut.
void merge_str(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 i = 0; i < n2; i++) R[i] = arr[m + 1 + i];

    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_str(char **arr, int n) {
    for (int curr_size = 1; curr_size <= n-1; curr_size *= 2) {
        for (int left_start = 0; left_start < n-1; left_start +=
2*curr_size) {
            int mid = left_start + curr_size - 1;
            int right_end = (left_start + 2*curr_size - 1 < n-1) ?
(left_start + 2*curr_size - 1) : (n-1);
            if (mid < right_end)
                merge_str(arr, left_start, mid, right_end);
        }
    }
}
```

Gambar 1.2.8. Potongan Kode Merge Sort data kata.

e. Quick Sort

Quick Sort juga berbasis **divide-and-conquer**, namun dengan strategi berbeda: algoritma memilih satu elemen sebagai *pivot*, lalu mempartisi array ke dalam dua bagian (kurang dari pivot dan lebih dari pivot), dan mengurutkannya secara rekursif. Versi untuk string menggunakan pendekatan iteratif dengan stack. Kompleksitas waktu rata-ratanya **O(n log n)**.

```
/ Quick Sort
// Prinsip: Pilih satu elemen sebagai pivot, bagi data jadi dua (lebih
kecil dan lebih besar),
// lalu urutkan keduanya secara rekursif.
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 tmp = arr[++i];
            arr[i] = arr[j];
            arr[j] = tmp;
        }
    }
    int tmp = arr[i+1]; arr[i+1] = arr[high]; arr[high] = tmp;
    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);
    }
}
```

Gambar 1.2.9. Potongan Kode Quick Sort data angka.

```
// Quick Sort (String)
// Prinsip: Pilih kata sebagai pivot, pisahkan yang lebih kecil dan lebih
besar dari pivot, urutkan pakai stack.
void quick_sort_str(char **arr, int n) {
    int *stack = malloc(n * sizeof(int));
    int top = -1;

    stack[++top] = 0;
    stack[++top] = n - 1;

    while (top >= 0) {
        int high = stack[top--];
        int low = stack[top--];

        char *pivot = arr[high];
        int i = low - 1;
        for (int j = low; j <= high - 1; j++) {
            if (strcmp(arr[j], pivot) < 0) {
                i++;
                char *tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
            }
        }
```

```
        char *tmp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = tmp;

        int pi = i + 1;

        if (pi - 1 > low) {
            stack[++top] = low;
            stack[++top] = pi - 1;
        }
        if (pi + 1 < high) {
            stack[++top] = pi + 1;
            stack[++top] = high;
        }
    }
    free(stack);
}
```

Gambar 1.2.10. Potongan Kode Quick Sort data kata.

f. Shell Sort

Shell Sort merupakan generalisasi dari Insertion Sort yang membandingkan elemen dengan jarak tertentu (gap). Jarak ini akan dikurangi secara bertahap hingga menjadi satu. Strategi ini memungkinkan elemen bergerak lebih cepat ke posisi akhir yang sesuai. Kompleksitas waktu bervariasi, namun lebih baik dari $O(n^2)$.

```
// Shell Sort
// Prinsip: Urutkan data yang jaraknya jauh dulu, lalu kurangi jaraknya perlahan hingga jadi 1.
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;
        }
    }
}
```

Gambar 1.2.11. Potongan Kode Shell Sort data angka.

```c
// Shell Sort (String)
// Prinsip: Bandingkan kata yang jaraknya jauh dulu, lalu makin dekat
hingga jaraknya 1.
void shell_sort_str(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;
        }
    }
}
```

Gambar 1.2.11. Potongan Kode Shell Sort data kata.

1.3 Mekanisme Pengujian dan Benchmarking

Di dalam main.c, pengujian dilakukan melalui fungsi benchmark_int, benchmark_merge_quick, dan benchmark_str. Ketiganya bertanggung jawab untuk menjalankan algoritma sorting terhadap salinan data yang sesuai, mencatat waktu eksekusi menggunakan clock() dari pustaka <time.h>, dan mencetak hasil waktu serta estimasi penggunaan memori dalam KB. Untuk pengurutan data kata, pendekatan serupa digunakan, namun dengan array pointer ke string (char**) agar manipulasi data string lebih fleksibel dan efisien.

2. Hasil Eksperimen

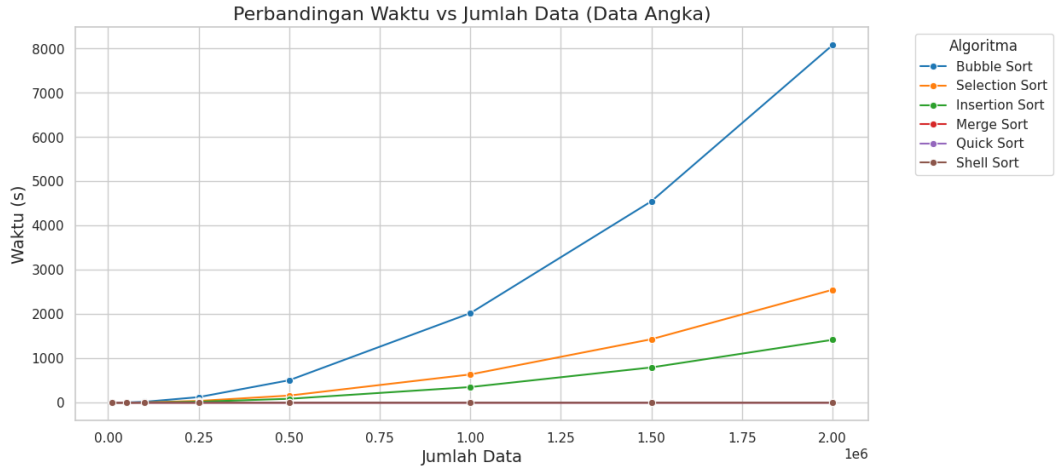
No.	Algoritma	Jenis Data	Jumlah Data	Waktu (s)	Memori(KB)
1.	Bubble Sort	Data Angka	10000	0.191834	39
			50000	4.797947	195
			100000	19.689968	390
			250000	125.607527	976
			500000	506.964956	1953
			1000000	2023.087578	3906
			1500000	4552.076913	5859
			2000000	8081.743668	7812
		Data Kata	10000	0.457874	205
			50000	11.597951	1025
			100000	45.474956	2050
			250000	286.178195	5126
			500000	1273.235425	10253
			1000000	10172.967798	20507
			1500000	18835.337014	30761
			2000000	27124.250612	40223
2.	Selection Sort	Data Angka	10000	0.063698	39
			50000	1.581190	195

			100000	6.395053	390
			250000	39.882727	976
			500000	160.606564	1953
			1000000	638.007828	3906
			1500000	1434.650723	5859
			2000000	2551.952734	7812
		Data Kata	10000	0.169993	205
			50000	4.305268	1025
			100000	17.469076	2050
			250000	110.259295	5126
			500000	490.819940	10253
			1000000	2799.191572	20507
			1500000	6550.641194	30761
			2000000	9341.431162	40223
3.	Insertion Sort	Data Angka	10000	0.034149	39
			50000	0.859013	195
			100000	3.535552	390
			250000	21.956294	976
			500000	88.112136	1953
			1000000	353.145516	3906
			1500000	796.869120	5859
			2000000	1420.613189	7812
		Data Kata	10000	0.085747	205
			50000	2.198316	1025
			100000	8.869181	2050
			250000	56.871498	5126
			500000	252.707508	10253
			1000000	3839.953465	20507
			1500000	5775.525874	30761
			2000000	7721.340927	40223
4.	Merge Sort	Data Angka	10000	0.001047	39
			50000	0.006031	195
			100000	0.012854	390
			250000	0.035004	976
			500000	0.073191	1953
			1000000	0.153989	3906
			1500000	0.236230	5859
			2000000	0.319322	7812
		Data Kata	10000	0.002737	205
			50000	0.018751	1025
			100000	0.043096	2050
			250000	0.123954	5126
			500000	0.301601	10253
			1000000	1.023772	20507
			1500000	1.531947	30761

			2000000	1.989965	40223
5.	Quick Sort	Data Angka	10000	0.000756	39
			50000	0.004474	195
			100000	0.009394	390
			250000	0.025350	976
			500000	0.052883	1953
			1000000	0.111648	3906
			1500000	0.169598	5859
			2000000	0.233667	7812
		Data Kata	10000	0.002164	205
			50000	0.012403	1025
			100000	0.026257	2050
			250000	0.070139	5126
			500000	0.150365	10253
			1000000	0.346455	20507
			1500000	0.511569	30761
			2000000	0.740633	40223
6.	Shell Sort	Data Angka	10000	0.001345	39
			50000	0.008537	195
			100000	0.018642	390
			250000	0.051875	976
			500000	0.114795	1953
			1000000	0.255911	3906
			1500000	0.404979	5859
			2000000	0.560439	7812
		Data Kata	10000	0.001667	205
			50000	0.009856	1025
			100000	0.021615	2050
			250000	0.059607	5126
			500000	0.128236	10253
			1000000	0.296560	20507
			1500000	0.441151	30761
			2000000	0.602375	40223

3. Grafik Perbandingan waktu dan memori

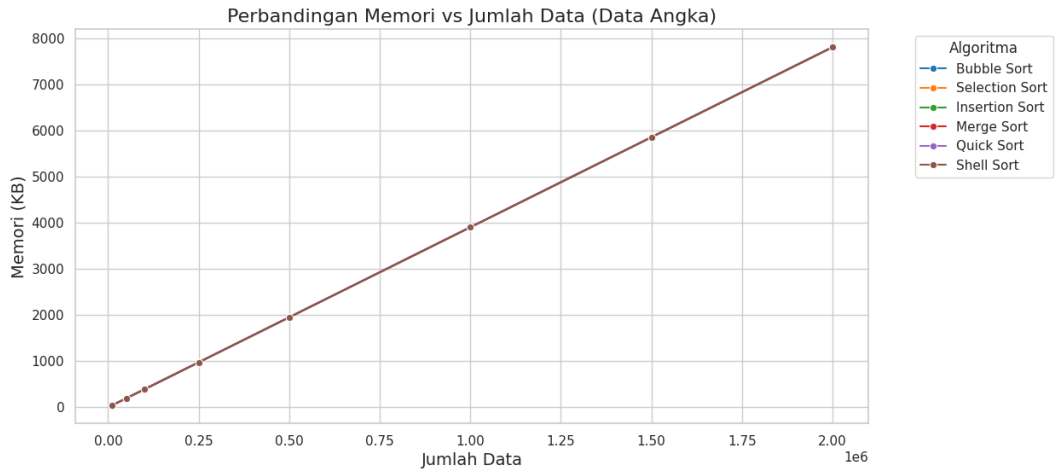
- Grafik Perbandingan Waktu dan Jumlah Data Angka



Grafik 3.1. Perbandingan Waktu dan Jumlah Data Angka

Grafik menunjukkan bahwa Bubble Sort, Selection Sort, dan Insertion Sort memiliki waktu eksekusi yang meningkat tajam seiring bertambahnya jumlah data, dengan Bubble Sort sebagai yang paling lambat. Ketiganya kurang efisien untuk data besar karena kompleksitas waktunya $O(n^2)$. Sementara itu, Shell Sort menunjukkan kinerja yang jauh lebih baik, dan dua algoritma tercepat secara konsisten adalah Merge Sort dan Quick Sort. Keduanya memiliki waktu eksekusi yang rendah dan stabil meskipun jumlah data sangat besar, menjadikannya pilihan paling efisien untuk pengurutan data angka skala besar.

- Grafik Perbandingan Memori dan Jumlah Data Angka

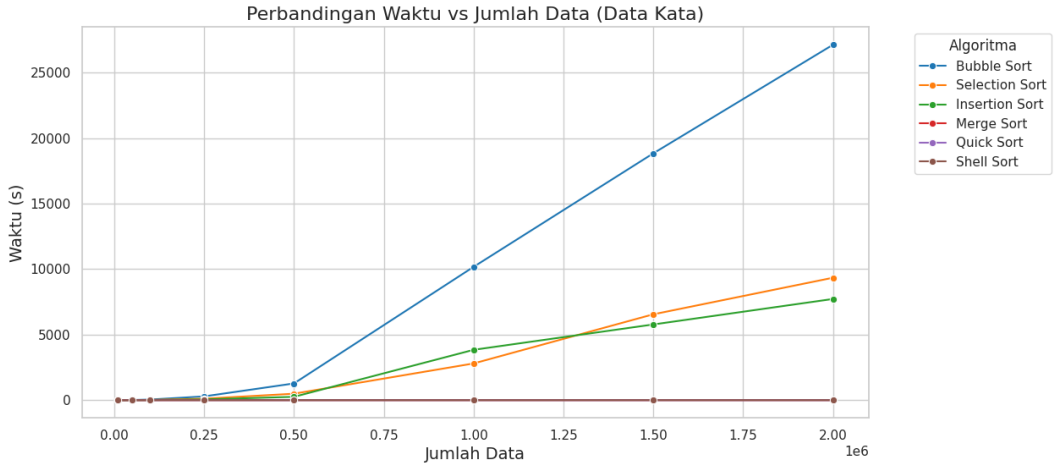


Grafik 3.2. Perbandingan Memori dan Jumlah Data Angka

Grafik menunjukkan bahwa penggunaan memori dari semua algoritma sorting pada data angka cenderung sama dan meningkat secara linear terhadap jumlah data. Hal ini terjadi karena seluruh algoritma menggunakan struktur data yang serupa (array

integer) dan tidak melakukan alokasi memori tambahan yang signifikan, kecuali pada saat salinan data dibuat untuk proses pengujian. Perbedaan memori antarmetode tidak terlihat jelas, sehingga dapat disimpulkan bahwa faktor memori bukan pembeda utama performa antar algoritma pada data angka.

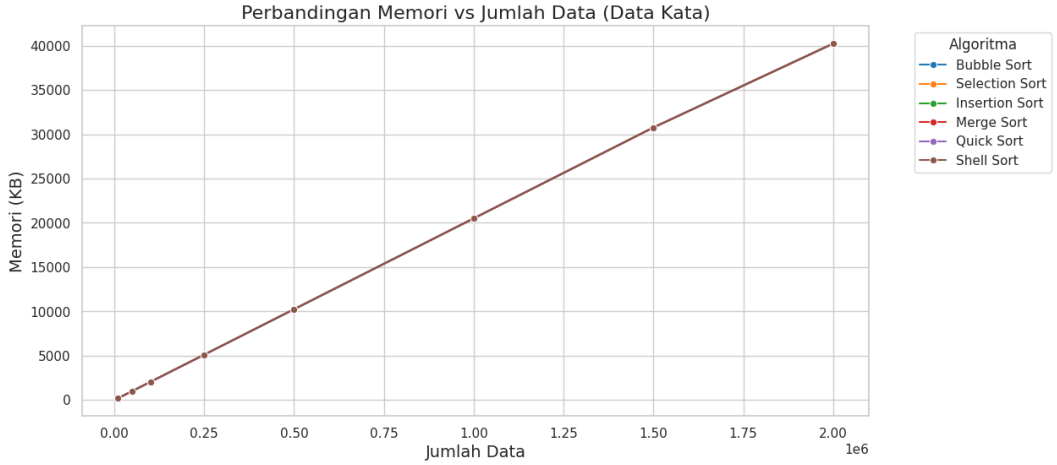
- Grafik Perbandingan Waktu dan Jumlah Data Kata



Grafik 3.3. Perbandingan Waktu dan Jumlah Data Kata

Grafik menunjukkan waktu eksekusi sorting data kata menunjukkan pola yang mirip dengan data angka, dimana algoritma Bubble Sort membutuhkan waktu paling lama dan meningkat drastis seiring bertambahnya jumlah data. Quick Sort dan Merge Sort menjadi algoritma yang paling efisien, dengan waktu yang relatif rendah dan stabil. Namun, Selection Sort yang pada beberapa ukuran data terlihat lebih cepat dari Insertion Sort yang seharusnya tidak demikian karena secara teori dan praktik, Insertion Sort umumnya lebih efisien. Perbedaan ini disebabkan oleh kondisi perangkat saat pengujian, di mana pada saat tertentu sistem sedang menjalankan banyak aplikasi lain secara bersamaan, sehingga berdampak pada beban prosesor dan menurunkan performa eksekusi algoritma tertentu.

- Grafik Perbandingan Memori dan Jumlah Data Kata



Grafik 3.4. Perbandingan Memori dan Jumlah Data Kata

Grafik menunjukkan bahwa penggunaan memori oleh semua algoritma saat mengurutkan data kata meningkat secara linear seiring bertambahnya jumlah data. Pola ini seragam dan hampir tidak terlihat perbedaan antar algoritma, karena seluruh metode menggunakan struktur data string yang sama dan alokasi memori yang sebanding. Dengan demikian, memori bukan menjadi faktor pembeda utama performa untuk kasus pengurutan data kata.

4. Analisis dan Kesimpulan

4.1 Analisis Hasil Eksperimen

Berdasarkan hasil pengujian terhadap enam algoritma sorting yaitu Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan Shell Sort dengan berbagai ukuran data hingga 2.000.000 elemen, diperoleh sejumlah hasil yang konsisten dan relevan dengan teori kompleksitas algoritma.

- **Performa Waktu Eksekusi Berdasarkan Jenis Algoritma**
  Hasil pengujian menunjukkan bahwa algoritma dengan kompleksitas waktu **$O(n^2)$,** yaitu Bubble Sort, Selection Sort, dan Insertion Sort, memiliki waktu eksekusi yang terus meningkat drastis seiring bertambahnya jumlah data. Bubble Sort menjadi algoritma dengan performa terburuk, terutama pada data skala besar, baik pada data angka maupun data kata. Sementara itu, Quick Sort dan Merge Sort menunjukkan performa yang jauh lebih efisien dan stabil karena kompleksitas waktunya **O(n log n).** Shell Sort berada di posisi menengah dan menawarkan hasil yang cukup efisien dengan peningkatan waktu yang masih tergolong rasional.
- **Perbandingan Waktu Sorting antara Data Angka dan Data Kata**
  Waktu eksekusi sorting untuk data kata secara umum lebih tinggi dibanding data angka. Hal ini disebabkan oleh proses perbandingan string yang lebih kompleks dibanding perbandingan nilai integer. Meski begitu, urutan efisiensi antar algoritma tetap konsisten di kedua jenis data. Quick Sort dan Merge Sort tetap unggul, sementara algoritma $O(n^2)$ masih menunjukkan keterbatasan performa yang sama, hanya skalanya menjadi lebih besar pada data bertipe string.
- **Ketidakkonsistenan Hasil pada Beberapa Ukuran Data**
  Dalam salah satu pengujian, Selection Sort tampak lebih cepat daripada Insertion Sort pada ukuran data tertentu, yang bertentangan dengan teori umum. Hal ini tidak disebabkan oleh kesalahan implementasi, melainkan karena kondisi lingkungan saat eksperimen berlangsung. Laptop digunakan untuk berbagai aktivitas lain yang cukup berat pada saat pengujian tersebut dilakukan, sehingga berdampak pada performa algoritma yang sensitif terhadap beban prosesor, seperti Insertion Sort. Ini menunjukkan bahwa faktor eksternal seperti kondisi sistem juga dapat memengaruhi hasil eksperimen algoritmik.

- **Penggunaan Memori yang Seragam antar Algoritma**
  Berdasarkan grafik memori, seluruh algoritma menunjukkan pola penggunaan memori yang hampir sama, yaitu meningkat secara linear terhadap jumlah data. Tidak ada perbedaan mencolok antar algoritma dalam aspek ini, baik pada data angka maupun data kata. Hal ini dikarenakan semua algoritma mengolah array dengan cara yang serupa dan tidak menggunakan struktur data tambahan secara signifikan, kecuali Merge Sort yang secara teori membutuhkan ruang tambahan, namun dampaknya tidak tampak dominan dalam eksperimen ini.

4.2 Kesimpulan

Adapun kesimpulan yang diperoleh berdasarkan hasil pengujian terhadap enam algoritma sorting, yaitu:

1. Algoritma Efisien untuk Skala Besar

Quick Sort dan **Merge Sort** terbukti sebagai algoritma paling efisien dalam pengurutan data berskala besar. Keduanya konsisten menunjukkan waktu eksekusi yang rendah dan stabil, baik pada data angka maupun kata. Ini sesuai dengan karakteristik algoritma yang memang dirancang untuk efisiensi tinggi, terutama ketika data sangat besar dan tidak terurut.

2. Algoritma yang Tidak Direkomendasikan untuk Data Besar

Bubble Sort, **Selection Sort**, dan **Insertion Sort** tidak direkomendasikan untuk digunakan pada dataset besar karena performanya yang buruk. Ketiganya mengalami peningkatan waktu eksekusi yang sangat signifikan, bahkan hingga ribuan detik pada data 2 juta baris. Penggunaannya sebaiknya dibatasi untuk data kecil atau tujuan pembelajaran algoritma dasar.

3. Shell Sort sebagai Alternatif Menengah

Shell Sort dapat dijadikan pilihan alternatif saat menginginkan keseimbangan antara kecepatan dan kesederhanaan. Meskipun tidak secepat Quick atau Merge Sort, Shell Sort masih jauh lebih baik dari algoritma $O(n^2)$ dan tidak membutuhkan memori tambahan yang besar.

4. Aspek Memori Tidak Menjadi Faktor Pembeda

Dari sisi penggunaan memori, tidak ditemukan perbedaan signifikan antar algoritma. Seluruh metode mengonsumsi memori secara linear sesuai jumlah data, sehingga pemilihan algoritma untuk efisiensi lebih baik difokuskan pada kecepatan eksekusi, bukan penggunaan memori.

5. Kondisi Sistem Dapat Mempengaruhi Hasil

Eksperimen ini juga menunjukkan bahwa kondisi perangkat saat pengujian, seperti beban kerja laptop, dapat memengaruhi hasil pengukuran waktu eksekusi. Oleh karena itu, pengujian performa algoritma sebaiknya dilakukan dalam kondisi sistem yang stabil dan minim gangguan untuk mendapatkan hasil yang lebih akurat dan konsisten.

Secara keseluruhan, **Quick Sort dan Merge Sort** merupakan algoritma yang paling direkomendasikan untuk pengurutan data skala besar karena konsistensi, efisiensi, dan skalabilitasnya. Sementara itu, Bubble Sort, Selection Sort, dan Insertion Sort hanya cocok untuk data berukuran kecil atau tujuan edukasi. Shell Sort bisa menjadi alternatif apabila dibutuhkan keseimbangan antara efisiensi waktu dan kesederhanaan implementasi.