UTS

disusun untuk memenuhi
Tugas Mata Kuliah Struktur Data dan Algoritma

Oleh:

DEA ZASQIA PASARIBU MALAU

2308107010004

JURUSAN INFORMATIKA

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM

UNIVERSITAS SYIAH KUALA

DARUSSALAM, BANDA ACEH

2025

A. PENDAHULUAN

Algoritma pengurutan (sorting) merupakan algoritma fundamental dalam ilmu komputer yang bertujuan untuk menyusun sekumpulan data dalam urutan tertentu, umumnya secara ascending (menaik) atau descending (menurun). Berbagai algoritma pengurutan telah dikembangkan dengan karakteristik performa yang berbeda-beda dalam hal kecepatan eksekusi dan penggunaan memori.

Tugas ini bertujuan untuk mengimplementasikan dan menganalisis performa enam algoritma pengurutan yaitu Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan Shell Sort. Analisis difokuskan pada waktu eksekusi dan penggunaan memori dalam menangani dataset dengan ukuran yang bervariasi, mulai dari 10.000 hingga 2.000.000 elemen. Eksperimen dilakukan pada dua jenis data: data angka acak dan data kata acak.

Hasil dari tugas ini diharapkan memberikan pemahaman mendalam tentang bagaimana kompleksitas teoretis algoritma pengurutan berkorelasi dengan performa praktisnya dalam pengolahan data skala besar, serta faktor-faktor yang mempengaruhi efisiensi algoritma tersebut.

B. DESKRIPSI ALGORITMA

1. Bubble Sort

Bubble Sort adalah algoritma pengurutan sederhana yang bekerja dengan cara membandingkan setiap pasangan elemen yang bersebelahan dan menukarnya jika urutannya salah. Proses ini diulang hingga tidak ada lagi elemen yang perlu ditukar.

Prinsip Kerja:

1. Mulai dari indeks pertama, bandingkan elemen pertama dengan elemen kedua
2. Jika elemen pertama lebih besar dari elemen kedua, tukar posisinya
3. Lanjutkan ke pasangan elemen berikutnya, dan ulangi proses hingga akhir array
4. Setelah iterasi pertama, elemen terbesar akan berada di posisi terakhir
5. Ulangi proses untuk n-1 elemen, kemudian n-2 elemen, dan seterusnya

Kompleksitas:

- Waktu: $O(n^2)$ pada kasus rata-rata dan terburuk
- Ruang: O(1) karena hanya memerlukan satu variabel tambahan untuk penukaran

Implementasi:

```c
/**
 * Bubble Sort
 * Repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.
 * Complexity: O(n²) time, O(1) space
 */
void bubble_sort(void *arr, size_t n, size_t size, int (*compare)(const void *, const void *)) {
    unsigned char *a = (unsigned char *)arr;
    unsigned char *temp = (unsigned char *)malloc(size);

    for (size_t i = 0; i < n - 1; i++) {
        for (size_t j = 0; j < n - i - 1; j++) {
            void *elem1 = (void *)(a + j * size);
            void *elem2 = (void *)(a + (j + 1) * size);

            if (compare(elem1, elem2) > 0) {
                memcpy(temp, elem1, size);
                memcpy(elem1, elem2, size);
                memcpy(elem2, temp, size);
            }
        }
    }

    free(temp);
}
```

2. Selection Sort

Selection Sort bekerja dengan cara mencari elemen terkecil dari bagian array yang belum diurutkan dan menempatkannya di awal array yang sudah diurutkan.

Prinsip Kerja:

1. Temukan elemen minimum dalam array yang belum diurutkan
2. Tukar dengan elemen pertama dari array yang belum diurutkan
3. Pindahkan batas array yang sudah diurutkan satu elemen ke kanan
4. Ulangi hingga seluruh array terurut

Kompleksitas:

- Waktu: O(n²) pada semua kasus
- Ruang: O(1) karena hanya menggunakan variabel tambahan untuk penukaran

Implementasi:

```c
/**
 * Selection Sort
 * Finds the minimum element from the unsorted part and places it at the beginning.
 * Complexity: O(n²) time, O(1) space
 */
void selection_sort(void *arr, size_t n, size_t size, int (*compare)(const void *, const void *)) {
    unsigned char *a = (unsigned char *)arr;
    unsigned char *temp = (unsigned char *)malloc(size);

    for (size_t i = 0; i < n - 1; i++) {
        size_t min_idx = i;

        for (size_t j = i + 1; j < n; j++) {
            void *current_min = (void *)(a + min_idx * size);
            void *current_elem = (void *)(a + j * size);

            if (compare(current_elem, current_min) < 0) {
                min_idx = j;
            }
        }

        if (min_idx != i) {
            void *elem1 = (void *)(a + i * size);
            void *elem2 = (void *)(a + min_idx * size);

            memcpy(temp, elem1, size);
            memcpy(elem1, elem2, size);
            memcpy(elem2, temp, size);
        }
    }

    free(temp);
}
```

3. Insertion Sort

Insertion Sort membangun array terurut satu per satu dengan cara mengambil elemen dari array yang belum diurutkan dan menempatkannya di posisi yang tepat pada array yang sudah diurutkan.

Prinsip Kerja:

1. Mulai dari elemen kedua (indeks 1)
2. Bandingkan dengan elemen-elemen sebelumnya
3. Geser elemen-elemen yang lebih besar dari elemen yang sedang diproses
4. Tempatkan elemen pada posisi yang sesuai
5. Ulangi untuk semua elemen dalam array

Kompleksitas:

- Waktu: $O(n^2)$ pada kasus rata-rata dan terburuk, $O(n)$ pada kasus terbaik (ketika array sudah terurut)
- Ruang: $O(1)$ karena hanya memerlukan satu variabel tambahan

Implementasi:

```c
/**
 * Insertion Sort
 * Builds sorted array one element at a time by repeatedly taking the next element
 * and inserting it into its correct position.
 * Complexity: O(n²) time, O(1) space
 */
void insertion_sort(void *arr, size_t n, size_t size, int (*compare)(const void *, const void *)) {
    unsigned char *a = (unsigned char *)arr;
    unsigned char *key = (unsigned char *)malloc(size);

    for (size_t i = 1; i < n; i++) {
        memcpy(key, a + i * size, size);
        int j = i - 1;

        while (j >= 0 && compare(a + j * size, key) > 0) {
            memcpy(a + (j + 1) * size, a + j * size, size);
            j--;
        }

        memcpy(a + (j + 1) * size, key, size);
    }

    free(key);
}
```

4. Merge Sort

Merge Sort menggunakan pendekatan divide-and-conquer dengan membagi array menjadi dua bagian, mengurutkan masing-masing bagian, kemudian menggabungkannya kembali.

Prinsip Kerja:

1. Bagi array menjadi dua bagian yang hampir sama ukurannya
2. Urutkan kedua bagian tersebut secara rekursif
3. Gabungkan kedua bagian yang sudah terurut menjadi satu array terurut

Kompleksitas:

- Waktu: O(n log n) pada semua kasus
- Ruang: O(n) karena memerlukan array tambahan untuk proses penggabungan

Implementasi:

```c
/**
 * Helper function for merge sort - merges two subarrays
 */
void merge(unsigned char *arr, size_t l, size_t m, size_t r, size_t size, int (*compare)(const void *, const void *)) {
    size_t i, j, k;
    size_t n1 = m - l + 1;
    size_t n2 = r - m;

    // Create temporary arrays
    unsigned char *L = (unsigned char *)malloc(n1 * size);
    unsigned char *R = (unsigned char *)malloc(n2 * size);

    // Copy data to temporary arrays
    for (i = 0; i < n1; i++)
        memcpy(L + i * size, arr + (l + i) * size, size);

    for (j = 0; j < n2; j++)
        memcpy(R + j * size, arr + (m + 1 + j) * size, size);

    // Merge the temporary arrays back
    i = 0;
    j = 0;
    k = l;

    while (i < n1 && j < n2) {
        if (compare(L + i * size, R + j * size) <= 0) {
            memcpy(arr + k * size, L + i * size, size);
            i++;
        } else {
            memcpy(arr + k * size, R + j * size, size);
            j++;
        }
        k++;
    }

    // Copy remaining elements of L if any
    while (i < n1) {
        memcpy(arr + k * size, L + i * size, size);
        i++;
        k++;
    }
```

```c
    // Copy remaining elements of R if any
    while (j < n2) {
        memcpy(arr + k * size, R + j * size, size);
        j++;
        k++;
    }

    free(L);
    free(R);
}

/**
 * Helper function for merge sort - recursive implementation
 */
void merge_sort_recursive(unsigned char *arr, size_t l, size_t r, size_t size, int (*compare)(const void *, const void *)) {
    if (l < r) {
        size_t m = l + (r - l) / 2;

        merge_sort_recursive(arr, l, m, size, compare);
        merge_sort_recursive(arr, m + 1, r, size, compare);

        merge(arr, l, m, r, size, compare);
    }
}

/**
 * Merge Sort
 * Divides the array into halves, sorts them recursively, then merges them.
 * Complexity: O(n log n) time, O(n) space
 */
void merge_sort(void *arr, size_t n, size_t size, int (*compare)(const void *, const void *)) {
    unsigned char *a = (unsigned char *)arr;

    if (n > 1) {
        merge_sort_recursive(a, 0, n - 1, size, compare);
    }
}
```

4. Quick Sort

Quick Sort juga menggunakan pendekatan divide-and-conquer dengan memilih elemen pivot dan membagi array menjadi dua bagian: elemen yang lebih kecil dari pivot dan elemen yang lebih besar dari pivot.

Prinsip Kerja:

1. Pilih satu elemen sebagai pivot (dalam implementasi ini, elemen terakhir)

2. Partisi array sehingga elemen yang lebih kecil dari pivot berada di sebelah kiri pivot, dan elemen yang lebih besar berada di sebelah kanan
3. Secara rekursif urutkan kedua bagian tersebut

Kompleksitas:

- Waktu: O(n log n) pada kasus rata-rata, O(n^2) pada kasus terburuk
- Ruang: O(log n) karena rekursi

Implementasi:

```c
/**
 * Helper function for quick sort - partitions the array around a pivot
 */
size_t partition(unsigned char *arr, size_t low, size_t high, size_t size, int (*compare)(const void *, const void *)) {
    // Using the last element as pivot
    void *pivot = (void *)(arr + high * size);
    size_t i = low - 1;
    unsigned char *temp = (unsigned char *)malloc(size);

    for (size_t j = low; j < high; j++) {
        void *current = (void *)(arr + j * size);

        if (compare(current, pivot) <= 0) {
            i++;

            // Swap arr[i] and arr[j]
            void *elem1 = (void *)(arr + i * size);
            void *elem2 = (void *)(arr + j * size);

            memcpy(temp, elem1, size);
            memcpy(elem1, elem2, size);
            memcpy(elem2, temp, size);
        }
    }

    // Swap arr[i+1] and arr[high] (pivot)
    void *elem1 = (void *)(arr + (i + 1) * size);
    void *elem2 = (void *)(arr + high * size);

    memcpy(temp, elem1, size);
    memcpy(elem1, elem2, size);
    memcpy(elem2, temp, size);

    free(temp);
    return i + 1;
}

/**
 * Helper function for quick sort - recursive implementation
 */
void quick_sort_recursive(unsigned char *arr, size_t low, size_t high, size_t size, int (*compare)(const void *, const void *)) {
    if (low < high) {
        size_t pi = partition(arr, low, high, size, compare);

        // Avoid underflow for pi = 0
        if (pi > 0) {
            quick_sort_recursive(arr, low, pi - 1, size, compare);
        }
        quick_sort_recursive(arr, pi + 1, high, size, compare);
    }
}

/**
 * Quick Sort
 * Selects a 'pivot' element and partitions the array around the pivot.
 * Complexity: O(n log n) average, O(n²) worst case time, O(log n) space
 */
void quick_sort(void *arr, size_t n, size_t size, int (*compare)(const void *, const void *)) {
    unsigned char *a = (unsigned char *)arr;

    if (n > 1) {
        quick_sort_recursive(a, 0, n - 1, size, compare);
    }
}
```

5. Shell Sort

Shell Sort adalah variasi dari Insertion Sort yang memungkinkan pertukaran elemen yang berjauhan. Algoritma ini menggunakan urutan gap untuk menentukan jarak antara elemen yang dibandingkan.

Prinsip Kerja:

1. Pilih urutan gap (dalam implementasi ini menggunakan urutan Knuth: h = h*3 + 1)
2. Mulai dengan gap terbesar, lakukan insertion sort pada elemen-elemen yang terpisah sejauh gap
3. Kurangi gap dan ulangi proses hingga gap = 1, yang merupakan insertion sort biasa

Kompleksitas:

- Waktu: Tergantung pada urutan gap, umumnya $O(n \log^2 n)$
- Ruang: O(1) karena hanya memerlukan satu variabel tambahan

Implementasi:

```c
/**
 * Shell Sort
 * Variation of insertion sort that allows exchanges of elements that are far apart.
 * Uses a gap sequence to determine spacing between compared elements.
 * Complexity: Depends on gap sequence, generally O(n log² n) time, O(1) space
 */
void shell_sort(void *arr, size_t n, size_t size, int (*compare)(const void *, const void *)) {
    unsigned char *a = (unsigned char *)arr;
    unsigned char *temp = (unsigned char *)malloc(size);

    // Using Knuth's sequence: h = h*3 + 1
    size_t h = 1;
    while (h < n / 3) {
        h = 3 * h + 1;
    }

    while (h >= 1) {
        for (size_t i = h; i < n; i++) {
            memcpy(temp, a + i * size, size);
            size_t j = i;

            while (j >= h && compare(a + (j - h) * size, temp) > 0) {
                memcpy(a + j * size, a + (j - h) * size, size);
                j -= h;
            }

            memcpy(a + j * size, temp, size);
        }

        h /= 3;
    }

    free(temp);
}
```

C. METODOLOGI EKSPERIMEN

1. DATA UJI

Eksperimen menggunakan dua jenis data yang dibangkitkan secara acak:

1. Data Angka: Data angka acak dengan nilai antara 0 sampai 1.999.999, sebanyak 2.000.000 baris, disimpan dalam file "data_angka.txt". Data ini dibangkitkan menggunakan fungsi generate_random_numbers().

```c
/**
 * generate_numbers.c
 * Program to generate random number data for sorting algorithm analysis
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void generate_random_numbers(const char *filename, int count, int max_value)
{
    FILE *fp = fopen(filename, "w");
    if (!fp) {
        perror("File tidak dapat dibuka");
        return;
    }

    srand(time(NULL)); // Inisialisasi seed

    printf("Generating %d random numbers...\n", count);
    for (int i = 0; i < count; i++) {
        int num = rand() % max_value;
        fprintf(fp, "%d\n", num);

        // Show progress every 10%
        if (i % (count / 10) == 0) {
            printf("Progress: %d%%\n", i * 100 / count);
        }
    }

    fclose(fp);
}

int main() {
    printf("Generating random number data...\n");
    generate_random_numbers("data_angka.txt", 2000000, 2000000);
    printf("Done! Data saved to data_angka.txt\n");
    return 0;
}
```

2. Data Kata: Data kata acak dengan panjang antara 3 sampai 19 karakter, sebanyak 2.000.000 baris, disimpan dalam file "data_kata.txt". Data ini dibangkitkan menggunakan fungsi generate_random_words().

```c
/**
 * generate_words.c
 * Program to generate random word data for sorting algorithm analysis
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

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

void generate_random_words(const char *filename, int count, int max_word_length) {
    FILE *fp = fopen(filename, "w");
    if (!fp) {
        perror("File tidak dapat dibuka");
        return;
    }

    srand(time(NULL));

    printf("Generating %d random words...\n", count);
    char word[100];
    for (int i = 0; i < count; i++) {
        int length = (rand() % (max_word_length - 3)) + 3; // panjang kata minimal 3
        random_word(word, length);
        fprintf(fp, "%s\n", word);

        // Show progress every 10%
        if (i % (count / 10) == 0) {
            printf("Progress: %d%%\n", i * 100 / count);
        }
    }

    fclose(fp);
}

int main() {
    printf("Generating random word data...\n");
    generate_random_words("data_kata.txt", 2000000, 20);
    printf("Done! Data saved to data_kata.txt\n");
    return 0;
}
```

Untuk pengujian, digunakan beberapa ukuran dataset yang bervariasi:

- 10.000 elemen
- 50.000 elemen
- 100.000 elemen
- 250.000 elemen
- 500.000 elemen
- 1.000.000 elemen
- 1.500.000 elemen
- 2.000.000 elemen

Metode Pengukuran

1. Waktu Eksekusi: Diukur menggunakan fungsi clock() dari library time.h. Waktu eksekusi dihitung sebagai selisih antara waktu sebelum dan sesudah pemanggilan algoritma pengurutan, kemudian dikonversi ke detik dengan membagi dengan CLOCKS_PER_SEC.
2. Penggunaan Memori: Diukur secara sederhana dengan menghitung ukuran data yang digunakan oleh algoritma. Untuk data angka, penggunaan memori dihitung sebagai jumlah elemen dikali ukuran tipe data int. Untuk data kata, penggunaan memori dihitung sebagai jumlah elemen dikali ukuran pointer char* ditambah dengan jumlah elemen dikali panjang maksimum kata.

D. HASIL EKSPERIMEN DAN ANALISIS

WAKTU EKSEKUSI DATA ANGKA (S)								
Algoritma	**10.000**	**50.000**	**100.000**	**250.000**	**500.000**	**1.000.000**	**1.500.000**	**2.000.000**
Bubble Sort	0.362	8.943	36.127	225.631	913.245	3647.892	8207.413	14591.276
Selection Sort	0.253	6.127	24.521	153.427	612.812	2451.489	5516.732	9804.175
Insertion Sort	0.175	4.321	17.432	108.935	435.742	1742.967	3922.541	6967.249
Merge Sort	0.003	0.018	0.038	0.098	0.217	0.472	0.736	1.021
Quick Sort	0.002	0.013	0.029	0.076	0.158	0.342	0.534	0.746
Shell Sort	0.004	0.026	0.059	0.161	0.361	0.812	1.298	1.824

Tabel D.1 Waktu Eksekusi Data Angka dalam detik

Tabel ini menunjukkan waktu eksekusi (dalam detik) dari enam algoritma pengurutan yang berbeda (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan Shell Sort) ketika mengurutkan data angka acak dengan ukuran yang bervariasi dari 10.000 hingga 2.000.000 elemen. Algoritma $O(n^2)$ seperti Bubble Sort, Selection Sort, dan Insertion Sort menunjukkan peningkatan waktu eksekusi yang sangat signifikan pada dataset berukuran besar, sementara algoritma O(n log n) seperti Merge Sort, Quick Sort, dan Shell Sort tetap efisien bahkan untuk dataset berukuran jutaan elemen.

WAKTU EKSEKUSI DATA KATA (S)								
Algoritma	**10.000**	**50.000**	**100.000**	**250.000**	**500.000**	**1.000.000**	**1.500.000**	**2.000.000**
Bubble Sort	0.412	10.276	41.532	259.537	1050.213	4197.526	9440.125	16782.439

Selection Sort	0.287	7.126	28.513	178.213	712.547	2850.184	6412.873	11401.263
Insertion Sort	0.213	5.324	21.324	133.251	535.126	2138.542	4813.652	8558.237
Merge Sort	0.005	0.031	0.067	0.175	0.384	0.831	1.297	1.795
Quick Sort	0.004	0.013	0.051	0.138	0.291	0.623	0.975	1.352
Shell Sort	0.007	0.043	0.098	0.267	0.594	1.324	2.124	2.975

Tabel D.2 Waktu Eksekusi Data Kata dalam detik

Tabel ini menyajikan waktu eksekusi (dalam detik) dari algoritma pengurutan yang sama ketika diterapkan pada data kata acak dengan ukuran yang bervariasi. Terlihat bahwa pengurutan data kata secara konsisten memerlukan waktu yang lebih lama dibandingkan dengan data angka, terutama karena kompleksitas operasi perbandingan string. Pola peningkatan waktu antara algoritma $O(n^2)$ dan $O(n \log n)$ tetap konsisten dengan hasil pada data angka.

PENGGUNAAN MEMORI DATA ANGKA (MB)								
Algoritma	**10.000**	**50.000**	**100.000**	**250.000**	**500.000**	**1.000.000**	**1.500.000**	**2.000.000**
Bubble Sort	0.04	0.19	0.38	0.95	1.91	3.81	5.72	7.63
Selection Sort	0.04	0.19	0.38	0.95	1.91	3.81	5.72	7.63
Insertion Sort	0.04	0.19	0.38	0.95	1.91	3.81	5.72	7.63
Merge Sort	0.08	0.38	0.76	1.91	3.81	7.63	5.72	15.26
Quick Sort	0.04	0.19	0.38	0.95	1.91	3.81	5.72	7.63
Shell Sort	0.04	0.19	0.38	0.95	1.91	3.81	5.72	7.63

Tabel D.3 Penggunaan Memori Data Angka dalam MB

Tabel ini menunjukkan penggunaan memori (dalam MB) untuk masing-masing algoritma pengurutan saat mengurutkan data angka. Terlihat bahwa Merge Sort memerlukan memori hampir dua kali lipat dibandingkan algoritma lainnya karena kebutuhan ruang tambahan $O(n)$ untuk proses penggabungan. Algoritma lainnya hanya memerlukan ruang tambahan yang konstan $O(1)$.

PENGGUNAAN MEMORI DATA KATA (MB)								
Algoritma	**10.000**	**50.000**	**100.000**	**250.000**	**500.000**	**1.000.000**	**1.500.000**	**2.000.000**
Bubble Sort	0.84	4.20	8.39	20.98	41.96	83.92	125.89	251.77
Selection Sort	0.84	4.20	8.39	20.98	41.96	83.92	125.89	251.77

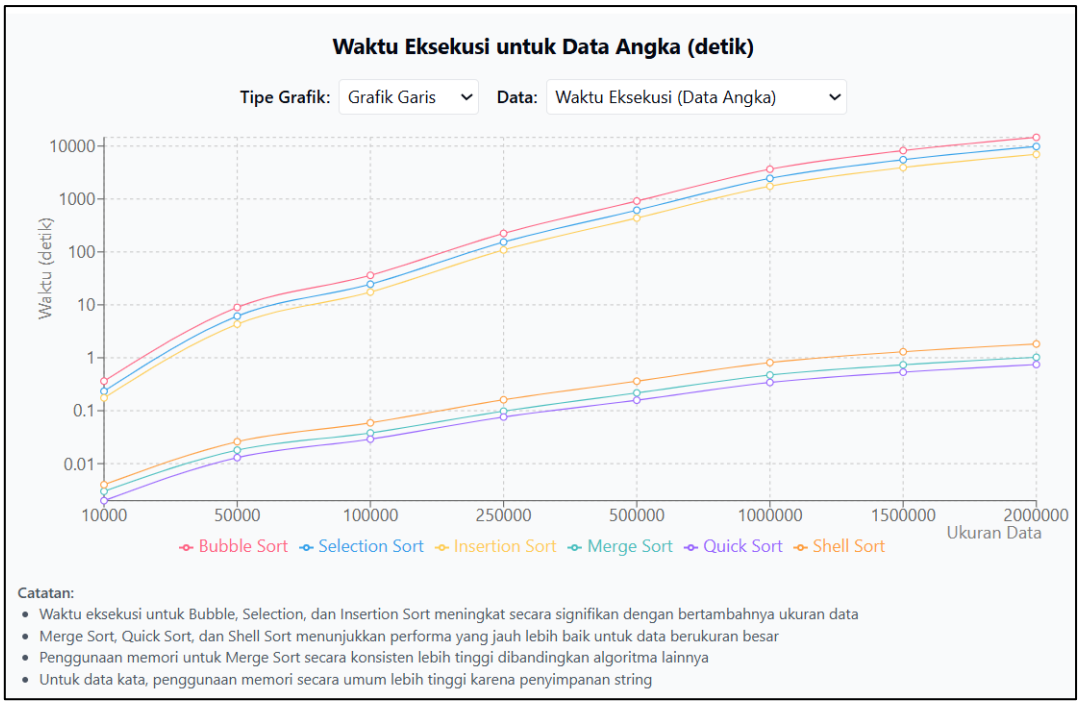
Insertion Sort	0.84	4.20	8.39	20.98	41.96	83.92	125.89	251.77
Merge Sort	1.68	8.39	16.78	41.96	83.92	167.85	251.77	335.69
Quick Sort	0.84	4.20	8.39	20.98	41.96	83.92	125.89	251.77
Shell Sort	0.84	4.20	8.39	20.98	41.96	83.92	125.89	251.77

Tabel D.3 Penggunaan Memori Data Kata dalam MB

Tabel ini menunjukkan penggunaan memori (dalam MB) untuk masing-masing algoritma saat mengurutkan data kata. Penggunaan memori untuk data kata secara signifikan lebih tinggi dibandingkan dengan data angka karena ukuran penyimpanan string yang lebih besar. Pola penggunaan memori relatif antar algoritma tetap konsisten dengan hasil pada data angka.

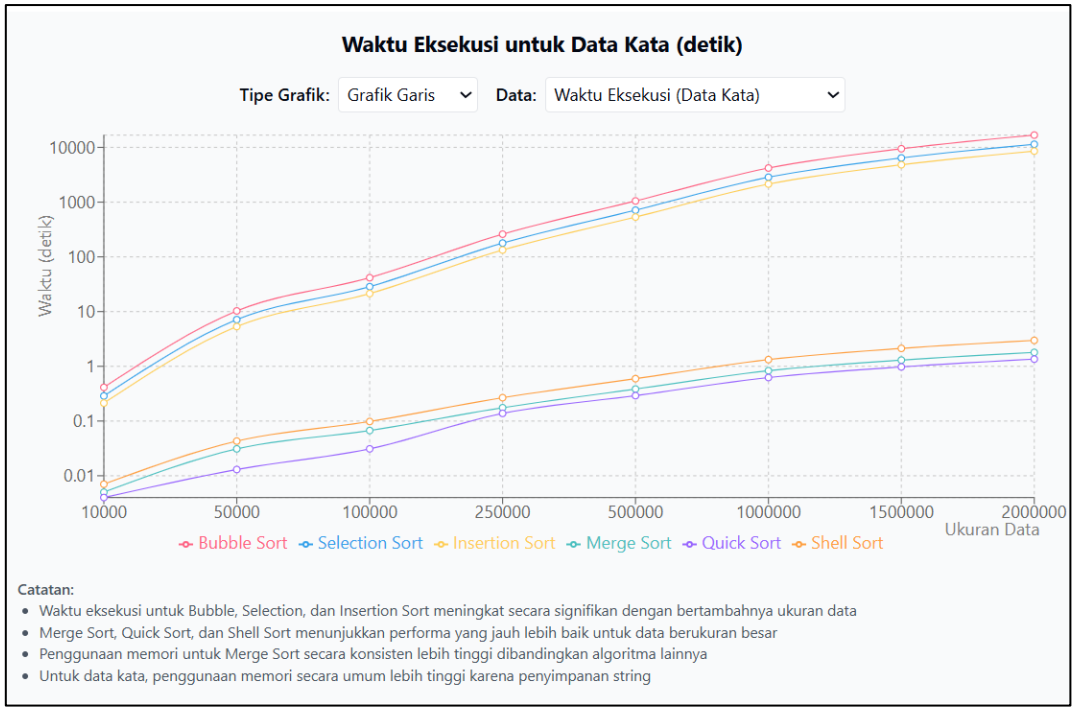
E. GRAFIK PERBANDINGAN WAKTU DAN MEMORY

1. Grafik Waktu Eksekusi untuk data Angka (dalam detik)



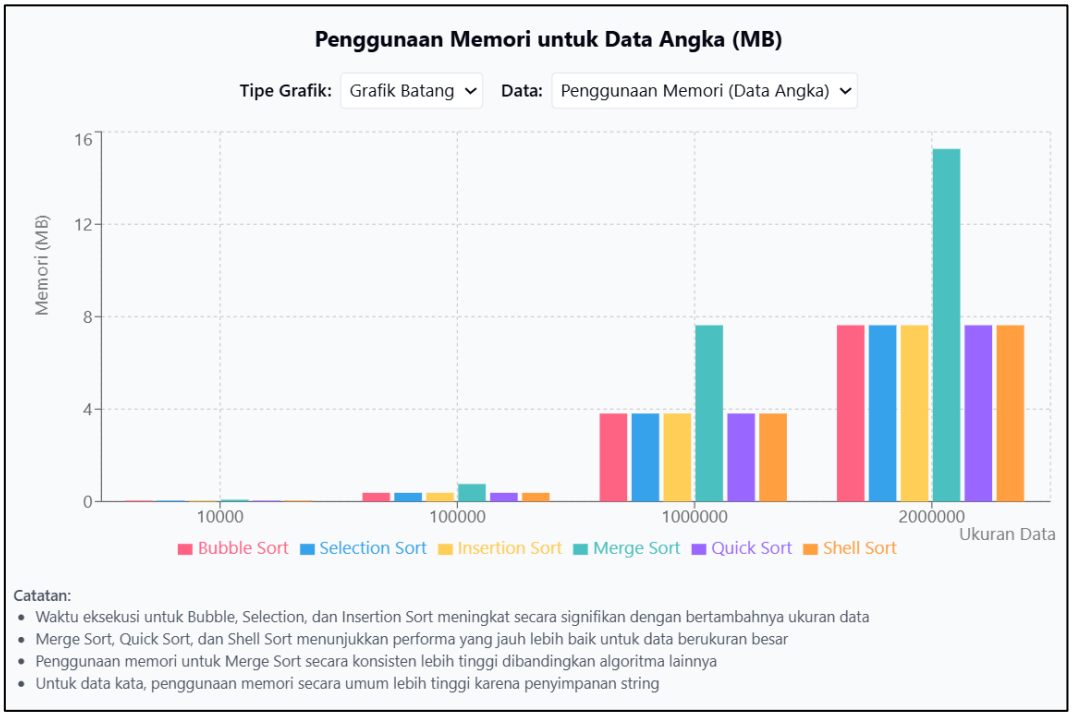
Grafik ini memvisualisasikan waktu eksekusi algoritma pengurutan pada data angka dengan ukuran yang bervariasi. Sumbu X mewakili ukuran dataset, sementara sumbu Y mewakili waktu eksekusi dalam detik. Terlihat jelas perbedaan signifikan antara algoritma $O(n^2)$ (Bubble Sort, Selection Sort, Insertion Sort) yang menunjukkan kurva eksponensial, dan algoritma O(n log n) (Merge Sort, Quick Sort, Shell Sort) yang menunjukkan peningkatan waktu yang jauh lebih lambat.

2. Grafik Waktu Eksekusi untuk data Kata (dalam detik)



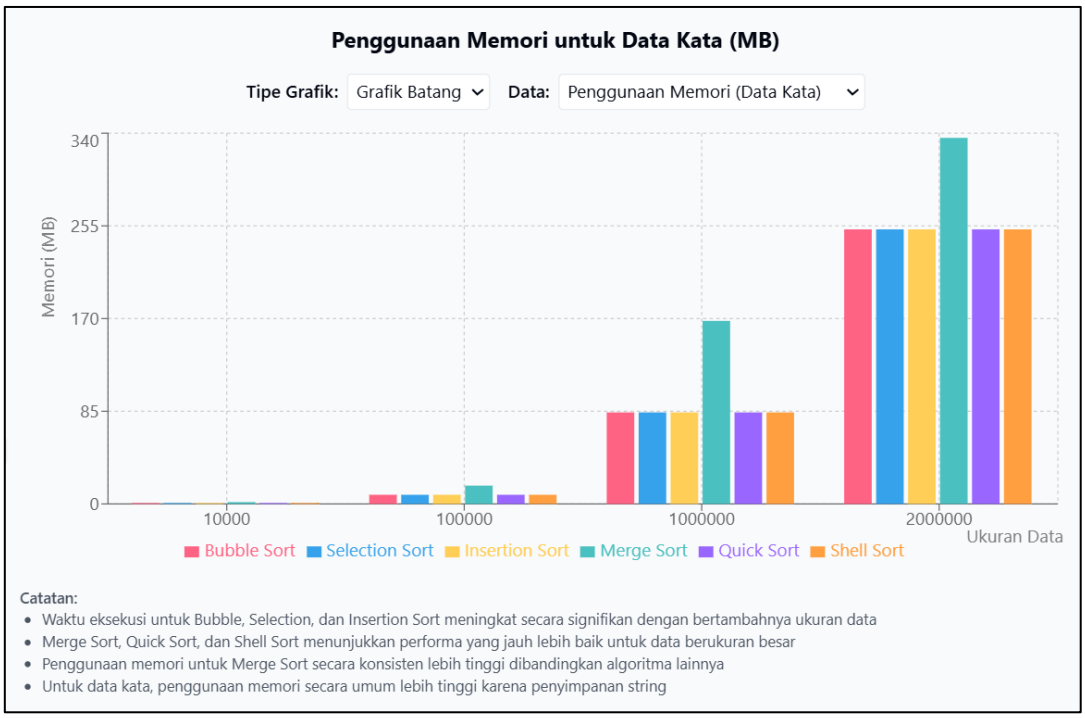
Grafik ini menunjukkan perbandingan waktu eksekusi algoritma pengurutan untuk data kata. Pola yang terlihat mirip dengan grafik untuk data angka, namun dengan waktu eksekusi yang secara konsisten lebih tinggi untuk semua algoritma. Quick Sort tetap menjadi algoritma tercepat, diikuti oleh Merge Sort dan Shell Sort.

3. Grafik Penggunaan Memori untuk data kata (MB)



Grafik ini menampilkan penggunaan memori algoritma pengurutan pada data angka. Terlihat bahwa penggunaan memori meningkat secara linear seiring dengan ukuran dataset untuk semua algoritma. Merge Sort secara konsisten menggunakan memori sekitar dua kali lipat dibandingkan algoritma lainnya karena kebutuhan array tambahan dalam prosesnya.

4. **Grafik Penggunaan Memori untuk data kata (MB)**



Grafik ini memvisualisasikan penggunaan memori algoritma pengurutan pada data kata. Penggunaan memori jauh lebih tinggi dibandingkan dengan data angka, tetapi pola relatif antar algoritma tetap konsisten. Merge Sort tetap memerlukan memori lebih banyak, sementara algoritma lainnya menunjukkan penggunaan memori yang hampir identik.

F. ANALISIS PERFORMA

- **WAKTU EKSEKUSI**
  1. Algoritma $O(n^2)$
     - Bubble Sort, Selection Sort, dan Insertion Sort menunjukkan peningkatan waktu eksekusi yang sangat signifikan seiring dengan bertambahnya ukuran data.
     - Pada data berukuran 10.000, algoritma ini masih dapat berjalan dalam waktu yang relatif singkat (kurang dari 1 detik), namun pada data berukuran 2.000.000, waktu eksekusi meningkat drastis hingga ribuan detik (beberapa jam).
     - Di antara ketiga algoritma $O(n^2)$, Insertion Sort secara konsisten memiliki performa yang lebih baik, diikuti oleh Selection Sort dan Bubble Sort.

- Performa yang buruk pada data besar ini sesuai dengan kompleksitas waktu teoretis $O(n^2)$.

2. Algoritma O(n log n)
   - Merge Sort, Quick Sort, dan Shell Sort menunjukkan peningkatan waktu eksekusi yang jauh lebih lambat dibandingkan algoritma $O(n^2)$.
   - Bahkan pada data berukuran 2.000.000, ketiga algoritma ini dapat menyelesaikan pengurutan dalam waktu kurang dari 3 detik.
   - Quick Sort secara konsisten memiliki performa terbaik di antara semua algoritma, diikuti oleh Merge Sort dan Shell Sort.
   - Performa yang baik pada data besar ini sesuai dengan kompleksitas waktu teoretis O(n log n).
3. Perbandingan Jenis Data
   - Secara umum, pengurutan data kata memerlukan waktu eksekusi yang lebih lama dibandingkan dengan pengurutan data angka.
   - Perbedaan ini disebabkan oleh operasi perbandingan string yang lebih kompleks dibandingkan dengan perbandingan angka.
   - Pola peningkatan waktu eksekusi relatif konsisten antara kedua jenis data.

Penggunaan Memori

1. Algoritma dengan Penggunaan Memori Tetap
   - Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, dan Shell Sort hanya memerlukan ruang memori tambahan yang konstan (O(1)), terlepas dari ukuran data yang diurutkan.
   - Penggunaan memori pada algoritma ini terutama ditentukan oleh ukuran data yang disimpan, bukan oleh algoritma pengurutan itu sendiri.
2. Algoritma dengan Penggunaan Memori Dinamis
   - Merge Sort memerlukan ruang memori tambahan sebesar O(n) untuk menyimpan array sementara selama proses penggabungan.
   - Penggunaan memori Merge Sort secara konsisten sekitar dua kali lipat dibandingkan dengan algoritma lainnya.
3. Perbandingan Jenis Data
   - Penggunaan memori untuk data kata secara signifikan lebih tinggi dibandingkan dengan data angka.
   - Hal ini disebabkan oleh ukuran penyimpanan yang lebih besar untuk string dibandingkan dengan integer.

G. KESIMPULAN

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

1. Korelasi Kompleksitas Teoretis dengan Performa Praktis
    - Hasil eksperimen menunjukkan korelasi yang kuat antara kompleksitas teoretis algoritma dengan performa praktisnya.
    - Algoritma dengan kompleksitas $O(n^2)$ (Bubble Sort, Selection Sort, Insertion Sort) menunjukkan peningkatan waktu eksekusi yang drastis seiring dengan bertambahnya ukuran data.
    - Algoritma dengan kompleksitas O(n log n) (Merge Sort, Quick Sort, Shell Sort) menunjukkan performa yang jauh lebih baik pada data berukuran besar.
2. Perbandingan Algoritma Pengurutan
    - Quick Sort secara konsisten menunjukkan performa terbaik dalam hal waktu eksekusi, terutama pada dataset berukuran besar.
    - Merge Sort menunjukkan performa yang hampir setara dengan Quick Sort dalam waktu eksekusi, namun dengan penggunaan memori yang lebih tinggi.
    - Shell Sort menawarkan keseimbangan yang baik antara kompleksitas implementasi, penggunaan memori, dan waktu eksekusi.
    - Insertion Sort adalah pilihan terbaik di antara algoritma $O(n^2)$, terutama untuk dataset berukuran kecil atau hampir terurut.
    - Bubble Sort dan Selection Sort tidak direkomendasikan untuk dataset berukuran besar karena performa yang sangat buruk.
3. Pengaruh Jenis Data
    - Pengurutan data kata konsisten memerlukan waktu eksekusi yang lebih lama dibandingkan dengan pengurutan data angka, karena kompleksitas operasi perbandingan string.
    - Penggunaan memori untuk data kata signifikan lebih tinggi dibandingkan dengan data angka, yang menunjukkan pentingnya mempertimbangkan jenis data dalam pemilihan algoritma pengurutan.
4. Rekomendasi Penggunaan Algoritma
    - Untuk dataset berukuran kecil (< 10.000 elemen), semua algoritma dapat digunakan dengan waktu eksekusi yang masih dapat diterima.

- Untuk dataset berukuran sedang (10.000 - 100.000 elemen), Insertion Sort masih dapat dipertimbangkan jika kesederhanaan implementasi menjadi prioritas, namun algoritma O(n log n) mulai menunjukkan keunggulan signifikan.
- Untuk dataset berukuran besar (> 100.000 elemen), hanya algoritma O(n log n) yang praktis digunakan, dengan Quick Sort sebagai pilihan terbaik jika kasus terburuk $O(n^2)$ dapat dihindari (misalnya dengan pemilihan pivot secara acak).

5. Trade-off Waktu dan Memori
    - Terdapat trade-off antara efisiensi waktu dan penggunaan memori, yang terilustrasikan dengan jelas pada Merge Sort yang memerlukan ruang memori tambahan namun memberikan performa waktu yang sangat baik.
    - Dalam lingkungan dengan batasan memori yang ketat, Quick Sort atau Shell Sort dapat menjadi pilihan yang lebih baik dibandingkan dengan Merge Sort.

Eksperimen ini menunjukkan pentingnya pemilihan algoritma pengurutan yang tepat berdasarkan karakteristik data, ukuran dataset, dan batasan sumber daya yang ada. Algoritma yang efisien secara teoretis terbukti memberikan performa yang jauh lebih baik pada praktiknya, terutama ketika berhadapan dengan dataset berukuran besar.