TUGAS SDA D

disusun untuk memenuhi tugas
mata kuliah struktur data dan algoritma

Oleh:

FIRAH MAULIDA

2308107010034

JURUSAN INFORMATIKA
FAKULTAS METEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS SYIAH KUALA
DARUSSALAM, BANDA ACEH
2024

A. Deskripsi Algoritma

1. Bubble Sort

Bubble Sort adalah algoritma sorting yang sederhana namun tidak efisien untuk data besar. Algoritma ini bekerja dengan membandingkan elemen-elemen yang berdekatan dalam array dan menukarnya jika elemen pertama lebih besar dari yang kedua. Proses ini diulang hingga tidak ada lagi pertukaran yang dilakukan, menandakan bahwa array sudah terurut.

Waktu Kompleksitas:

- Best case: O(n) jika array sudah terurut.
- Worst case dan Average case: $O(n^2)$.

```c
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;
            }
        }
    }
}

void bubble_sort_str(char arr[][MAX_WORD_LEN], 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[MAX_WORD_LEN];
                strcpy(tmp, arr[j]);
                strcpy(arr[j], arr[j+1]);
                strcpy(arr[j+1], tmp);
            }
        }
    }
}
```

2. Selection Sort

Selection Sort juga merupakan algoritma yang sederhana, di mana algoritma ini bekerja dengan mencari elemen terkecil (atau terbesar, tergantung urutan) dalam array dan menukarnya dengan elemen pertama yang belum terurut. Proses ini diulang untuk elemen berikutnya hingga seluruh array terurut.

Waktu Kompleksitas:

- Best case, Worst case, dan Average case: $O(n^2)$.

```c
// Selection Sort
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 tmp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = tmp;
    }
}
// Selection Sort untuk String
void selection_sort_str(char arr[][MAX_WORD_LEN], 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;
        }
        if (min_idx != i) {
            char tmp[MAX_WORD_LEN];
            strcpy(tmp, arr[i]);
            strcpy(arr[i], arr[min_idx]);
            strcpy(arr[min_idx], tmp);
        }
    }
}
```

3. **Insertion Sort**

   Insertion Sort bekerja dengan cara mengurutkan array secara bertahap dengan memindahkan elemen-elemen ke posisi yang sesuai di bagian array yang sudah terurut. Proses ini dimulai dari elemen kedua dan bergerak ke depan, memasukkan setiap elemen ke dalam subarray terurut dengan cara membandingkan dan memindahkannya ke posisi yang tepat.

   Waktu Kompleksitas:

   - Best case: O(n) jika array sudah terurut.
   - Worst case dan Average case: $O(n^2)$.

```c
void insertion_sort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = key;
    }
}
// Insertion Sort untuk String
void insertion_sort_str(char arr[][MAX_WORD_LEN], int n) {
    for (int i = 1; i < n; i++) {
        char key[MAX_WORD_LEN];
        strcpy(key, arr[i]);
        int j = i - 1;
        while (j >= 0 && strcmp(arr[j], key) > 0) {
            strcpy(arr[j + 1], arr[j]);
            j--;
        }
        strcpy(arr[j + 1], key);
    }
}
```

4. **Merge Sort**

   Merge Sort adalah algoritma yang menggunakan pendekatan divide and conquer. Algoritma ini membagi array menjadi dua bagian, mengurutkan kedua bagian secara rekursif, dan akhirnya menggabungkannya menjadi satu array yang terurut. Merge Sort merupakan algoritma yang sangat efisien, terutama untuk data besar.

   Waktu Kompleksitas:

   - Best case, Worst case, dan Average case: O(n log n).

```c
void merge(int arr[], int temp[], int l, int m, int r) {
    int i = l;    // index untuk subarray kiri
    int j = m + 1;    // index untuk subarray kanan
    int k = l;    // index untuk array hasil gabungan

    while ((i <= m) && (j <= r)) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }

    // Salin sisa elemen dari subarray kiri, jika ada
    while (i <= m) {
        temp[k++] = arr[i++];
    }

    // Salin sisa elemen dari subarray kanan, jika ada
    while (j <= r) {
        temp[k++] = arr[j++];
    }

    // Salin hasil kembali ke array asli
    for (i = l; i <= r; i++) {
        arr[i] = temp[i];
    }
}

void merge_sort_recursive(int arr[], int temp[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        merge_sort_recursive(arr, temp, l, m);
        merge_sort_recursive(arr, temp, m+1, r);

        merge(arr, temp, l, m, r);
    }
}

// Fungsi merge_sort yang dipanggil dari main
void merge_sort(int arr[], int n) {
    int *temp = (int*)malloc(n * sizeof(int));
    if (temp == NULL) {
        printf("Gagal alokasi memori untuk merge sort\n");
        return;
    }

    merge_sort_recursive(arr, temp, 0, n-1);

    free(temp);
}
```

5. **Quick Sort**

   Quick Sort juga menggunakan metode divide and conquer, di mana algoritma memilih elemen pivot dan membagi array menjadi dua bagian: elemen yang lebih kecil dan elemen yang lebih besar dari pivot. Kemudian, kedua bagian tersebut diurutkan secara rekursif. Quick Sort umumnya lebih cepat daripada Merge Sort dalam prakteknya meskipun memiliki kompleksitas yang sama pada kasus terburuk.

   Waktu Kompleksitas:

   - Best case dan Average case: O(n log n).
   - Worst case: $O(n^2)$ (terjadi jika pivot yang dipilih tidak efektif).

```c
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int partition_int(int arr[], int low, int high) {
    // Pilih pivot sebagai elemen tengah untuk menghindari worst case
    int mid = low + (high - low) / 2;
    swap(&arr[mid], &arr[high]);

    int pivot = arr[high];
    int i = (low - 1);

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

void quick_sort_recursive(int arr[], int low, int high) {
    if (low < high) {
        // Batasi kedalaman rekursi untuk mencegah stack overflow
        if (high - low < 16) {
            // Gunakan insertion sort untuk array kecil
            for (int i = low + 1; i <= high; i++) {
                int key = arr[i];
                int j = i - 1;
                while (j >= low && arr[j] > key) {
                    arr[j + 1] = arr[j];
                    j--;
                }
                arr[j + 1] = key;
            }
        } else {
            int pi = partition_int(arr, low, high);

            quick_sort_recursive(arr, low, pi - 1);
            quick_sort_recursive(arr, pi + 1, high);
        }
    }
}

// Fungsi quick_sort yang dipanggil dari main
void quick_sort(int arr[], int n) {
    quick_sort_recursive(arr, 0, n-1);
}
```

6. Shell Sort

Shell Sort adalah peningkatan dari Insertion Sort yang memperkenalkan konsep "gap" atau jarak antara elemen-elemen yang dibandingkan. Dengan menggunakan gap yang lebih besar pada iterasi awal, Shell Sort dapat memperbaiki efisiensi Insertion Sort untuk data yang lebih besar.

Waktu Kompleksitas:

- Best case: O(n log n) (tergantung pada urutan gap yang digunakan).
- Worst case dan Average case: O(n^(3/2)) atau lebih buruk tergantung pada implementasi gap.

```c
void shell_sort(int arr[], int n) {
    // Gunakan urutan gap yang lebih efisien (Sedgewick's sequence)
    int gaps[] = {701, 301, 132, 57, 23, 10, 4, 1};
    int num_gaps = 8;

    for (int g = 0; g < num_gaps; g++) {
        int gap = gaps[g];

        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}
```

B. Tabel Hasil Ekperimen

1. Tabel Angka

No.	Algoritma	Ukuran	Waktu (detik)	Memori (KB)
1.	Bubble Sort	10.000	0.262000 detik	39 KB
		50.000	7.742000 detik	195 KB
		100.000	33.999000 detik	391 KB
		250.000	210.799000 detik	977 KB
		500.000	866.400000 detik	1953 KB
		1.000.000	4497.204000 detik	3906 KB
		1.500.000	8636.046000 detik	5859 KB
		2.000.000	15763.723000 detik	7813 KB
2.	Selection Sort	10.000	0.130000 detik	39 KB
		50.000	3.506000 detik	195 KB
		100.000	12.908000 detik	391 KB
		250.000	82.955000 detik	977 KB
		500.000	344.391000 detik	1953 KB
		1.000.000	1468.058000 detik	3906 KB
		1.500.000	3490.729000 detik	5859 KB
		2.000.000	5905.591000 detik	7813 KB
3.	Insertion Sort	10.000	0.069000 detik	39 KB
		50.000	2.282000 detik	195 KB
		100.000	7.564000 detik	391 KB
		250.000	44.326000 detik	977 KB

		500.000	194.203000 detik	1953 KB
		1.000.000	1065.262000 detik	3906 KB
		1.500.000	2221.561000 detik	5859 KB
		2.000.000	3061.680000 detik	7813 KB
4.	Merge Sort	10.000	0.001000 detik	39 KB
		50.000	0.009000 detik	195 KB
		100.000	0.019000 detik	391 KB
		250.000	0.055000 detik	977 KB
		500.000	0.116000 detik	1953 KB
		1.000.000	0.277000 detik	3906 KB
		1.500.000	0.379000 detik	5859 KB
		2.000.000	0.410000 detik	7813 KB
5.	Quick Sort	10.000	0.001000 detik	39 KB
		50.000	0.008000 detik	195 KB
		100.000	0.018000 detik	391 KB
		250.000	0.043000 detik	977 KB
		500.000	0.091000 detik	1953 KB
		1.000.000	0.216000 detik	3906 KB
		1.500.000	0.352000 detik	5859 KB
		2.000.000	0.489000 detik	7813 KB
6.	Shell Sort	10.000	0.002000 detik	39 KB
		50.000	0.016000 detik	195 KB
		100.000	0.030000 detik	391 KB
		250.000	0.091000 detik	977 KB
		500.000	0.186000 detik	1953 KB
		1.000.000	2.715000 detik	3906 KB
		1.500.000	5.850000 detik	5859 KB
		2.000.000	7.768000 detik	7813 KB

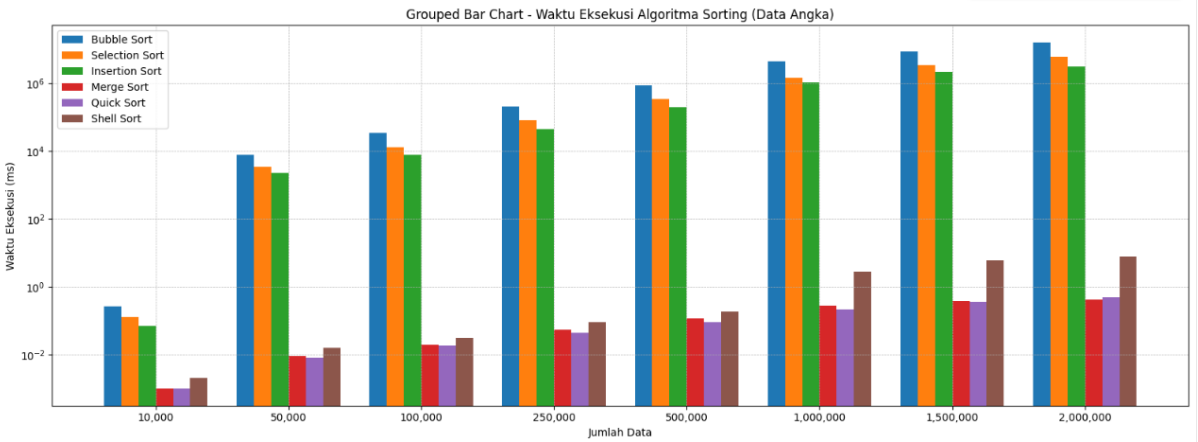
2. Tabel Kata

No.	Algoritma	Ukuran	Waktu (detik)	Memori (KB)
1.	Bubble Sort	10.000	1.786000 detik	977 KB
		50.000	47.279000 detik	4883 KB
		100.000	179.133000 detik	9766 KB
		250.000	1559.882000 detik	24414 KB
		500.000	5397.061000 detik	48828 KB
		1.000.000	22184.694000 detik	97656 KB
		1.500.000	49915.685000 detik	146484 KB
		2.000.000	88738.834000 detik	195312 KB
2.	Selection Sort	10.000	0.244000 detik	977 KB
		50.000	8.943000 detik	4883 KB
		100.000	39.470000 detik	9766 KB
		250.000	392.310000 detik	24414 KB
		500.000	1119.420000 detik	48828 KB
		1.000.000	4507.404000 detik	97656 KB
		1.500.000	10141.67000 detik	146484 KB
		2.000.000	18029.618000 detik	195312 KB
3.	Insertion Sort	10.000	0.635000 detik	977 KB

		50.000	16.608000 detik	4883 KB
		100.000	68.636000 detik	9766 KB
		250.000	625.747000 detik	24414 KB
		500.000	1824.651000 detik	48828 KB
		1.000.000	7493.934000 detik	97656 KB
		1.500.000	12489.655000 detik	146484 KB
		2.000.000	22381.639000 detik	195312 KB
4.	Merge Sort	10.000	0.009000 detik	977 KB
		50.000	0.048000 detik	4883 KB
		100.000	0.107000 detik	9766 KB
		250.000	0.413000 detik	24414 KB
		500.000	0.627000 detik	48828 KB
		1.000.000	1.305000 detik	97656 KB
		1.500.000	2.050000 detik	146484 KB
		2.000.000	2.693000 detik	195312 KB
5.	Quick Sort	10.000	0.007000 detik	977 KB
		50.000	0.043000 detik	4883 KB
		100.000	0.081000 detik	9766 KB
		250.000	0.478000 detik	24414 KB
		500.000	0.467000 detik	48828 KB
		1.000.000	1.155000 detik	97656 KB
		1.500.000	1.670000 detik	146484 KB
		2.000.000	2.319000 detik	195312 KB
6.	Shell Sort	10.000	0.011000 detik	977 KB
		50.000	0.090000 detik	4883 KB
		100.000	0.222000 detik	9766 KB
		250.000	2.634000 detik	24414 KB
		500.000	4.376000 detik	48828 KB
		1.000.000	17.642000 detik	97656 KB
		1.500.000	39.841000 detik	146484 KB
		2.000.000	65.116000detik	195312 KB

C. Grafik Perbandingan Waktu dan Memori

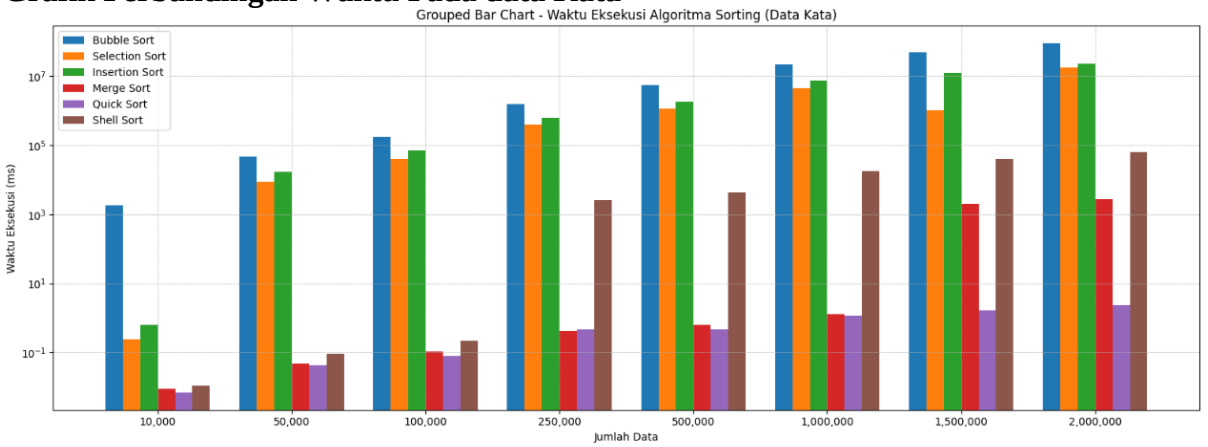
1. Grafik Perbandingan Waktu Pada data Angka



- semakin banyak jumlah data yang diolah, semakin besar pula waktu eksekusi yang dibutuhkan oleh seluruh algoritma.

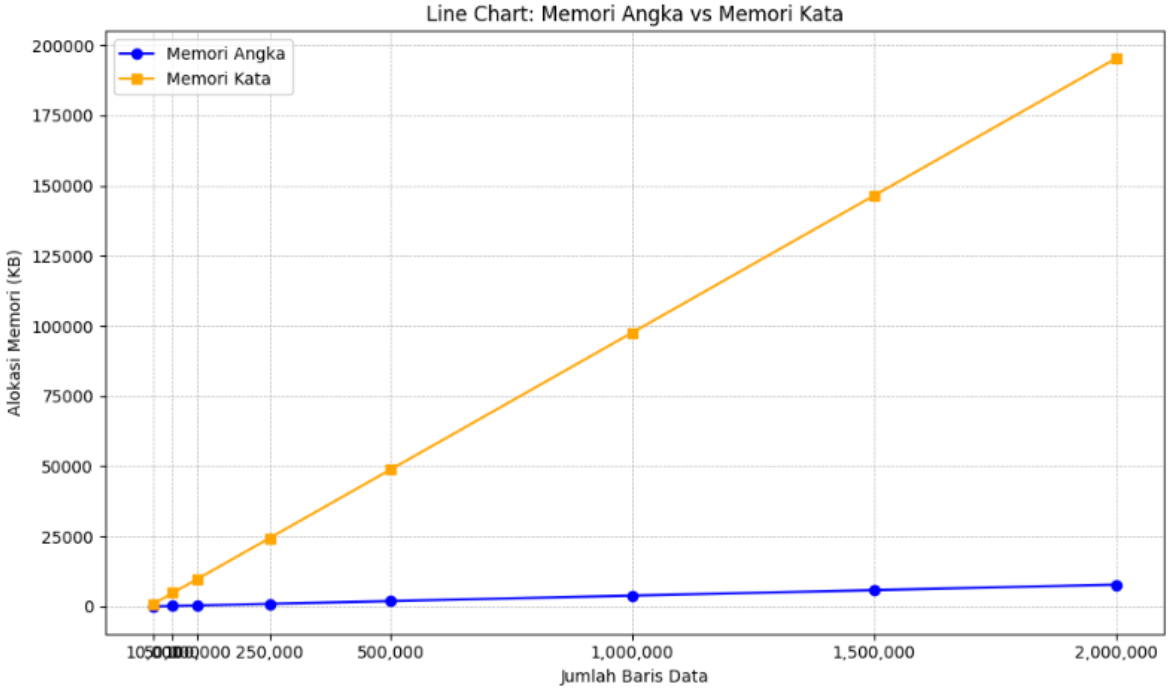
- Bubble Sort menunjukkan waktu eksekusi paling tinggi pada seluruh ukuran data. Untuk dataset berisi 2.000.000 elemen, Bubble Sort memerlukan waktu sekitar 15.763 detik, menjadikannya algoritma dengan performa terburuk.
- Selection Sort dan Insertion Sort juga memperlihatkan kinerja yang kurang efisien. Meski sedikit lebih cepat dibandingkan Bubble Sort, keduanya tetap membutuhkan waktu yang besar, masing-masing sekitar 5.905 detik dan 3.061 detik untuk data sebanyak 2.000.000 elemen.
- Merge Sort dan Quick Sort menunjukkan performa paling efisien, dengan waktu eksekusi jauh lebih rendah dibandingkan algoritma lainnya. Pada data 2.000.000, Merge Sort dapat menyelesaikan proses hanya dalam 0,410 detik dan Quick Sort dalam 0,489 detik.
- Shell Sort berada di antara kedua kelompok tersebut: lebih cepat dibandingkan algoritma sederhana, namun tidak seefisien Merge Sort atau Quick Sort. Untuk 2.000.000 data, Shell Sort membutuhkan waktu 7.768 detik.

2. Grafik Perbandingan Waktu Pada data Kata



- Semakin besar ukuran data, semakin besar pula waktu eksekusi yang dibutuhkan oleh semua algoritma. Hal ini menunjukkan adanya pertumbuhan kompleksitas yang sejalan dengan pertambahan jumlah elemen yang harus diolah.
- Bubble Sort menunjukkan performa paling lambat di antara semua algoritma. Untuk dataset sebesar 2.000.000 elemen, Bubble Sort membutuhkan waktu sekitar 88.738 detik dengan penggunaan memori sebesar 195.312 KB. Ini menjadikan Bubble Sort algoritma dengan kinerja terburuk dalam pengujian ini.
- Selection Sort dan Insertion Sort juga memperlihatkan performa yang kurang efisien. Selection Sort membutuhkan waktu 18.029 detik untuk mengolah 2.000.000 data, sementara Insertion Sort lebih lambat lagi dengan waktu 22.381 detik, keduanya menggunakan memori sebesar 195.312 KB.
- Merge Sort dan Quick Sort menunjukkan efisiensi yang jauh lebih baik dibandingkan algoritma lainnya. Pada data sebanyak 2.000.000 elemen, Merge Sort menyelesaikan proses sorting dalam waktu 2,693 detik dan Quick Sort dalam waktu 2,319 detik, dengan penggunaan memori yang sama yaitu 195.312 KB.
- Shell Sort menempati posisi tengah antara algoritma sederhana dan algoritma efisien. Shell Sort membutuhkan waktu 65,116 detik untuk menyelesaikan sorting pada dataset 2.000.000 elemen, dengan penggunaan memori yang sama, yaitu 195.312 KB.

3. **Grafik Perbandingan Memori Pada data Angka dan Kata**



Grafik "Line Chart: Memori Angka vs Memori Kata" menunjukkan perbandingan penggunaan memori antara data bertipe angka dan data bertipe kata berdasarkan jumlah baris data yang diolah. Dari grafik terlihat bahwa alokasi memori meningkat secara linear untuk kedua tipe data seiring bertambahnya jumlah baris data. Namun, penggunaan memori untuk data kata jauh lebih besar dibandingkan data angka. Sebagai contoh, pada 1.000.000 baris data, memori yang digunakan untuk angka hanya sekitar 3.906 KB, sedangkan untuk kata mencapai sekitar 97.656 KB. Hal ini menunjukkan bahwa penyimpanan data bertipe kata memerlukan ruang yang jauh lebih besar, sekitar 25 kali lipat dibandingkan data angka. Perbedaan ini kemungkinan besar disebabkan oleh representasi data kata (string) yang secara alami lebih kompleks dan bervariasi dibandingkan representasi data numerik yang lebih sederhana dan tetap. Oleh karena itu, dalam pengolahan data berskala besar, tipe data yang digunakan sangat memengaruhi efisiensi penggunaan memori, dan data bertipe angka terbukti lebih hemat dalam penggunaan sumber daya.

D. Analisis dan Kesimpulan

- **Bubble Sort merupakan algoritma dengan waktu eksekusi paling lambat.**
  Hal ini disebabkan oleh kompleksitas waktu sebesar **O(n^2)**, di mana setiap elemen harus dibandingkan dan ditukar berkali-kali hingga seluruh array terurut. Ketika jumlah data bertambah, proses ini menjadi sangat berat. Pada pengujian dengan 2 juta data, Bubble Sort memerlukan waktu lebih dari 15.000 detik, menjadikannya sangat tidak efisien untuk data besar.
- **Selection Sort dan Insertion Sort sedikit lebih baik, tapi tetap lambat untuk data besar.**
  Meskipun Selection Sort hanya melakukan satu swap per iterasi, ia tetap harus melakukan pencarian elemen terkecil di setiap langkah, menghasilkan banyak perbandingan. Insertion Sort juga memiliki kompleksitas **O(n^2)** dan hanya efisien untuk

data kecil atau hampir terurut. Keduanya masih tidak cocok digunakan dalam skala data besar.

- **Merge Sort dan Quick Sort adalah algoritma dengan performa terbaik.**
  Keduanya memiliki kompleksitas **O(n log n)** yang jauh lebih efisien dibanding $O(n^2)$. Merge Sort bekerja dengan strategi *divide and conquer*, memecah array menjadi bagian kecil lalu menggabungkannya kembali secara terurut. Quick Sort juga membagi data menggunakan pivot, dan biasanya lebih cepat dari Merge Sort di praktik karena overhead yang lebih kecil. Hasil pengujian menunjukkan keduanya tetap cepat bahkan pada data 2 juta elemen.
- **Shell Sort menawarkan performa menengah.**
  Shell Sort lebih baik dari algoritma sederhana karena menggunakan jarak (*gap*) untuk mempercepat penyisipan di awal. Meskipun tidak seefisien Merge atau Quick Sort, Shell Sort tetap layak dipertimbangkan untuk data besar jika implementasi gap-nya baik.
- **Penggunaan memori meningkat linear, tetapi data kata jauh lebih boros.**
  Hasil pengujian menunjukkan bahwa data bertipe kata (*string*) menggunakan memori jauh lebih besar daripada data angka. Contohnya, pada 1 juta elemen, data angka hanya membutuhkan sekitar 3.906 KB, sedangkan data kata bisa mencapai 97.656 KB. Ini karena string terdiri dari banyak karakter dan panjangnya bervariasi, sedangkan angka memiliki representasi yang tetap.
- **Spesifikasi laptop sangat memengaruhi hasil waktu eksekusi.**
  Laptop dengan prosesor yang cepat, RAM besar, dan penyimpanan SSD akan memproses sorting lebih cepat, terutama pada algoritma kompleks. Sebaliknya, jika pengujian dilakukan di perangkat berspesifikasi rendah, maka waktu eksekusi akan lebih lama dari seharusnya. Oleh karena itu, hasil perlu dievaluasi dengan mempertimbangkan kondisi perangkat yang digunakan.
- **Pemilihan algoritma dan tipe data sangat penting untuk efisiensi.**
  Untuk data berskala besar, Merge Sort dan Quick Sort sangat disarankan karena cepat dan stabil. Jika efisiensi memori menjadi prioritas, maka penggunaan data bertipe angka lebih menguntungkan. Pemilihan algoritma dan struktur data yang tepat akan memberikan dampak besar terhadap kinerja keseluruhan sistem.