

Tugas 4

Implementasi dan analisis performa algoritma sorting

disusun untuk memenuhi
Mata Kuliah SDA
kelas D

Oleh:

KHALISHA ADZRAINI ARIF
2308107010031



PROGRAM STUDI INFORMATIKA
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUANALAM
UNIVERSITAS SYIAH KUALA
DARUSSALAM, BANDA ACEH
2025

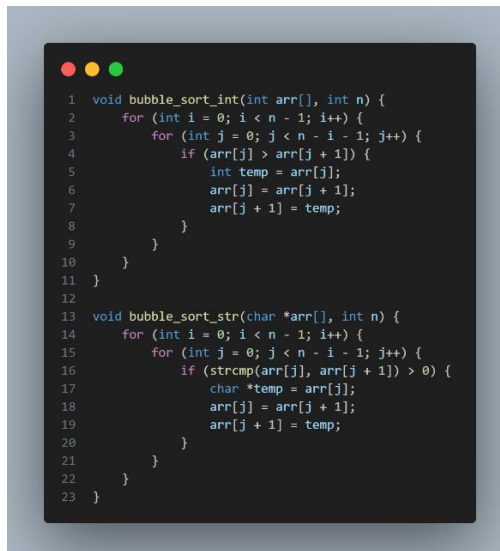
A. Deskripsi algoritma dan cara implementasi

1. Bubble Sort

Bubble Sort merupakan algoritma pengurutan yang sangat sederhana. Algoritma ini bekerja dengan cara membandingkan elemen-elemen bersebelahan dan menukarnya jika urutannya tidak sesuai. Proses ini diulang terus-menerus hingga seluruh elemen dalam array berada dalam urutan yang benar. Nama "bubble" berasal dari proses elemen terbesar yang 'mengambang' ke posisi terakhir seperti gelembung air.

Cara Implementasi:

- Lakukan perulangan dari elemen pertama hingga elemen ke-n.
- Bandingkan dua elemen yang berdekatan. Jika elemen kiri lebih besar dari elemen kanan, tukar posisinya.
- Ulangi proses untuk setiap elemen, mengurangi batas perulangan karena elemen terbesar sudah berada di posisi akhir.
- Proses berhenti jika dalam satu perulangan tidak ada pertukaran yang terjadi.



```
1 void bubble_sort_int(int arr[], int n) {
2     for (int i = 0; i < n - 1; i++) {
3         for (int j = 0; j < n - i - 1; j++) {
4             if (arr[j] > arr[j + 1]) {
5                 int temp = arr[j];
6                 arr[j] = arr[j + 1];
7                 arr[j + 1] = temp;
8             }
9         }
10    }
11 }
12
13 void bubble_sort_str(char *arr[], int n) {
14     for (int i = 0; i < n - 1; i++) {
15         for (int j = 0; j < n - i - 1; j++) {
16             if (strcmp(arr[j], arr[j + 1]) > 0) {
17                 char *temp = arr[j];
18                 arr[j] = arr[j + 1];
19                 arr[j + 1] = temp;
20             }
21         }
22     }
23 }
```

2. Selection Sort

Selection Sort bekerja dengan cara mencari elemen terkecil dari daftar yang belum diurutkan, kemudian menempatkannya ke posisi awal. Setelah itu, proses dilanjutkan ke elemen berikutnya hingga seluruh data terurut. Algoritma ini tidak memerlukan banyak pertukaran, tetapi tetap kurang efisien untuk dataset besar.

Cara Implementasi:

- Mulai dari indeks pertama, anggap elemen tersebut sebagai nilai minimum.
- Cari elemen terkecil dari sisa array.
- Jika ditemukan elemen yang lebih kecil, tukar dengan posisi saat ini.
- Lanjutkan proses ke indeks berikutnya hingga akhir array.

```

1 void selection_sort_int(int arr[], int n) {
2     for (int i = 0; i < n - 1; i++) {
3         int min_idx = i;
4         for (int j = i + 1; j < n; j++) {
5             if (arr[j] < arr[min_idx])
6                 min_idx = j;
7         }
8         int temp = arr[min_idx];
9         arr[min_idx] = arr[i];
10        arr[i] = temp;
11    }
12 }
13
14 void selection_sort_str(char *arr[], int n) {
15     for (int i = 0; i < n - 1; i++) {
16         int min_idx = i;
17         for (int j = i + 1; j < n; j++) {
18             if (strcmp(arr[j], arr[min_idx]) < 0)
19                 min_idx = j;
20         }
21         char *temp = arr[min_idx];
22         arr[min_idx] = arr[i];
23         arr[i] = temp;
24     }
25 }

```

3. Insertion Sort

Insertion Sort menyusun elemen dengan cara mengambil satu per satu data dan menempatkannya pada posisi yang sesuai dalam bagian array yang sudah terurut. Cara kerjanya menyerupai seseorang yang menyusun kartu remi di tangan, menyisipkan kartu pada posisi yang tepat.

Cara Implementasi:

- Mulai dari elemen kedua (indeks 1), anggap elemen pertama sudah terurut.
- Bandingkan elemen tersebut dengan elemen sebelumnya.
- Geser elemen yang lebih besar ke kanan untuk memberi tempat.
- Tempatkan elemen pada posisi yang sesuai.
- Ulangi proses hingga seluruh elemen masuk ke posisi yang benar.

```

1 void insertion_sort_int(int arr[], int n) {
2     for (int i = 1; i < n; i++) {
3         int key = arr[i];
4         int j = i - 1;
5         while (j >= 0 && arr[j] > key) {
6             arr[j + 1] = arr[j];
7             j--;
8         }
9         arr[j + 1] = key;
10    }
11 }
12
13 void insertion_sort_str(char *arr[], int n) {
14     for (int i = 1; i < n; i++) {
15         char *key = arr[i];
16         int j = i - 1;
17         while (j >= 0 && strcmp(arr[j], key) > 0) {
18             arr[j + 1] = arr[j];
19             j--;
20         }
21         arr[j + 1] = key;
22     }
23 }

```

4. Merge Sort

Merge Sort adalah algoritma rekursif yang menggunakan strategi divide and conquer. Data dibagi menjadi dua bagian yang lebih kecil, kemudian masing-masing bagian diurutkan, lalu digabungkan kembali. Keunggulan utama Merge

Sort adalah efisiensinya pada data besar dan kestabilannya dalam mempertahankan urutan elemen yang sama.

Cara Implementasi:

- Bagi array menjadi dua bagian secara rekursif hingga setiap bagian hanya berisi satu elemen.
- Gabungkan dua bagian tersebut dengan membandingkan elemen dari masing-masing bagian dan menyusunnya dalam array baru secara berurutan.
- Lanjutkan proses hingga seluruh bagian tergabung kembali menjadi satu array teratur.

```
1 void merge(int arr[], int l, int m, int r) {
2     int n1 = m - l + 1, n2 = r - m;
3     int l1[n1], r1[n2];
4     for (int i = 0; i < n1; i++) l1[i] = arr[l + i];
5     for (int j = 0; j < n2; j++) r1[j] = arr[m + 1 + j];
6     int i = 0, j = 0, k = l;
7     while (i < n1 && j < n2) arr[k++] = (l1[i] <= r1[j]) ? l1[i++] : r1[j++];
8     while (i < n1) arr[k++] = l1[i++];
9     while (j < n2) arr[k++] = r1[j++];
10 }
11
12 void merge_sort_int(int arr[], int l, int r) {
13     if (l < r) {
14         int m = l + (r - l) / 2;
15         merge_sort_int(arr, l, m);
16         merge_sort_int(arr, m + 1, r);
17         merge(arr, l, m, r);
18     }
19 }
20
21 void merge_sort_wrapper(int arr[], int n) {
22     merge_sort_int(arr, 0, n - 1);
23 }
24
25 void merge_str(char *arr[], int l, int m, int r) {
26     int n1 = m - l + 1, n2 = r - m;
27     char *l1[n1], *r1[n2];
28     for (int i = 0; i < n1; i++) l1[i] = arr[l + i];
29     for (int j = 0; j < n2; j++) r1[j] = arr[m + 1 + j];
30     int i = 0, j = 0, k = l;
31     while (i < n1 && j < n2) arr[k++] = (strcmp(l1[i], r1[j]) <= 0) ? l1[i++] : r1[j++];
32     while (i < n1) arr[k++] = l1[i++];
33     while (j < n2) arr[k++] = r1[j++];
34 }
35
36 void merge_sort_str(char *arr[], int l, int r) {
37     if (l < r) {
38         int m = l + (r - l) / 2;
39         merge_sort_str(arr, l, m);
40         merge_sort_str(arr, m + 1, r);
41         merge_str(arr, l, m, r);
42     }
43 }
44
45 void merge_sort_str_wrapper(char *arr[], int n) {
46     merge_sort_str(arr, 0, n - 1);
47 }
```

5. Quick Sort

Quick Sort juga menggunakan metode divide and conquer. Algoritma ini memilih satu elemen sebagai pivot, kemudian mempartisi array sehingga elemen yang lebih kecil dari pivot berada di sebelah kiri dan yang lebih besar di sebelah kanan. Proses ini dilakukan secara rekursif pada kedua bagian tersebut.

Cara Implementasi:

- Pilih satu elemen sebagai pivot (bisa elemen pertama, terakhir, atau acak).
- Pindahkan semua elemen yang lebih kecil dari pivot ke kiri, dan elemen yang lebih besar ke kanan.
- Rekursifkan proses pada sub-array kiri dan kanan secara terpisah.
- Lanjutkan hingga sub-array tersusun dengan benar.

```

1 // Quick Sort untuk array integer
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++;
7             int tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
8         }
9     }
10    int tmp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = tmp;
11    return i + 1;
12 }
13
14 void quick_sort_int(int arr[], int low, int high) {
15     if (low < high) {
16         int pi = partition(arr, low, high);
17         quick_sort_int(arr, low, pi - 1);
18         quick_sort_int(arr, pi + 1, high);
19     }
20 }
21
22 // Wrapper agar bisa dipanggil dari main.c
23 void quick_sort_wrapper(int arr[], int n) {
24     quick_sort_int(arr, 0, n - 1);
25 }
26
27 // Quick Sort untuk array string
28 int partition_str(char *arr[], int low, int high) {
29     char *pivot = arr[high];
30     int i = low - 1;
31     for (int j = low; j < high; j++) {
32         if (strcmp(arr[j], pivot) < 0) {
33             i++;
34             char *tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
35         }
36     }
37     char *tmp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = tmp;
38     return i + 1;
39 }
40
41 void quick_sort_str(char *arr[], int low, int high) {
42     if (low < high) {
43         int pi = partition_str(arr, low, high);
44         quick_sort_str(arr, low, pi - 1);
45         quick_sort_str(arr, pi + 1, high);
46     }
47 }
48
49 // Wrapper agar bisa dipanggil dari main.c
50 void quick_sort_str_wrapper(char *arr[], int n) {
51     quick_sort_str(arr, 0, n - 1);
52 }
53

```

6. Shell Sort

Shell Sort adalah pengembangan dari Insertion Sort. Perbedaannya terletak pada jarak antar elemen yang dibandingkan dan ditukar (disebut gap). Dengan membandingkan elemen yang terpisah jauh di awal, elemen-elemen dapat bergerak lebih cepat ke posisi yang benar. Gap akan mengecil secara bertahap hingga mendekati 1, dan pada akhirnya dilakukan seperti Insertion Sort.

Cara Implementasi:

- Tentukan nilai awal gap (umumnya setengah panjang array).
- Bandingkan dan urutkan elemen-elemen yang berjarak sejauh gap.
- Kurangi nilai gap, biasanya dengan membaginya dua.
- Ulangi proses sampai gap bernilai 1 dan seluruh data terurut.

```

1 void shell_sort_int(int arr[], int n) {
2     for (int gap = n / 2; gap > 0; gap /= 2) {
3         for (int i = gap; i < n; i++) {
4             int temp = arr[i];
5             for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
6                 arr[j] = arr[j - gap];
7             arr[j] = temp;
8         }
9     }
10 }
11
12 void shell_sort_str(char *arr[], int n) {
13     for (int gap = n / 2; gap > 0; gap /= 2) {
14         for (int i = gap; i < n; i++) {
15             char *temp = arr[i];
16             int j;
17             for (j = i; j >= gap && strcmp(arr[j - gap], temp) > 0; j -= gap)
18                 arr[j] = arr[j - gap];
19             arr[j] = temp;
20         }
21     }
22 }

```

B. Tabel hasil eksperimen (waktu dan memori)

Algoritma	Data Angka Waktu (s)							
	10000	50000	100000	250000	500000	1000000	1500000	2000000
Bubble Sort	0.107	4.152	18.438	119.649	485.767	4.487.956	4.282.872	9.354.738
Selection Sort	0.107	1.208	5.919	31.147	124.653	944.363	1.139.265	3.149.675
Insertion Sort	0.029	0.973	3.858	24.884	97.080	970.531	1.187.625	2.081.673
Merge Sort	0.010	0.017	0.021	0.026	0.072	0.213	0.408	0.78
Quick Sort	0.000	0.004	0.021	0.022	0.042	0.101	0.150	0.223
Shell Sort	0.001	0.002	0.012	0.047	0.092	0.204	0.351	0.601

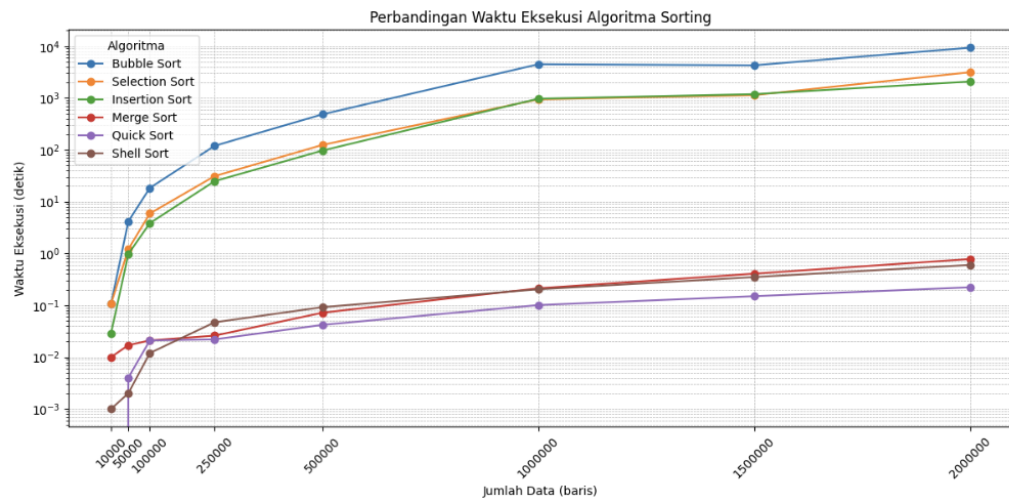
Algoritma	Data Angka Memori (KB)							
	10000	50000	100000	250000	500000	1000000	1500000	2000000
Bubble Sort	215684	215840	216036	216573	217048	219472	221480	223524
Selection Sort	215700	215856	216052	216588	217092	205912	69628	223501
Insertion Sort	215724	215856	216000	216588	217064	103856	42816	222156
Merge Sort	215724	216032	216372	217548	219000	199972	42956	218321
Quick Sort	215724	216036	216372	217552	219004	199972	42956	218321
Shell Sort	215724	216036	216376	217552	219004	199972	42956	218321

Algoritma	Data Kata Waktu (s)							
	10000	50000	100000	250000	500000	1000000	1500000	2000000
Bubble Sort	0.355	10.645	44.765	504.040	3.227.437	12.886.549	18.964.675	21.634.086
Selection Sort	0.355	3.931	16.528	302.128	1.751.859	8.238.994	12.646.235	15.467.857
Insertion Sort	0.079	2.387	10.243	210.670	1.416.591	5.970.531	13.939.741	18.252.816
Merge Sort	0.002	0.018	0.022	0.078	0.263	0.408	0.601	0.811
Quick Sort	0.001	0.014	0.028	0.055	0.190	0.296	0.351	0.498
Shell Sort	0.001	0.019	0.052	0.283	0.092	1.466	1.945	2.505

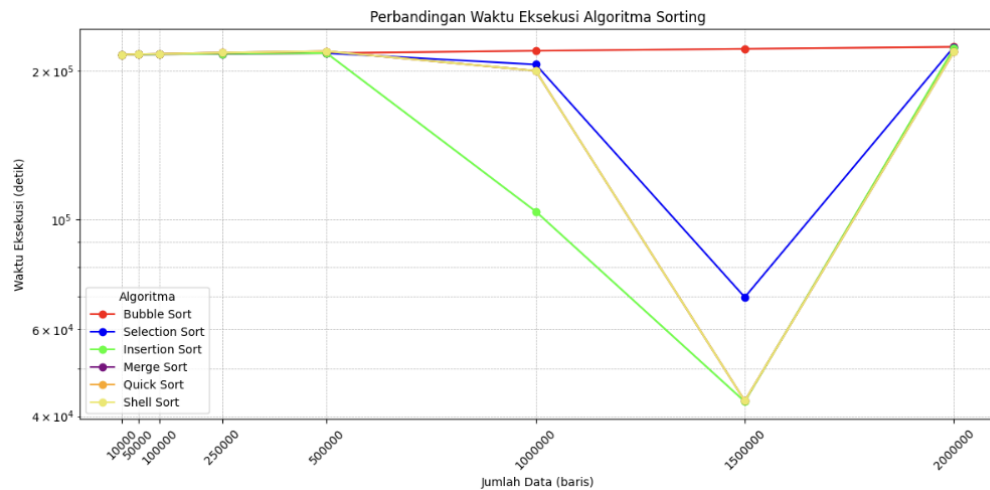
Algoritma	Data Kata Memori (KB)							
	10000	50000	100000	250000	500000	1000000	1500000	2000000
Bubble Sort	215724	216036	216376	217552	66400	103820	177328	229279
Selection Sort	215724	216036	216376	217552	55176	103844	177321	229279
Insertion Sort	215724	216036	216376	217552	53424	103856	177322	229279
Merge Sort	215724	216036	216376	217552	55364	104052	177256	229275
Quick Sort	215724	216036	216376	217552	55160	104052	177256	229275
Shell Sort	215724	216036	216376	217552	219004	104052	177256	229275

C. Grafik perbandingan waktu dan memory

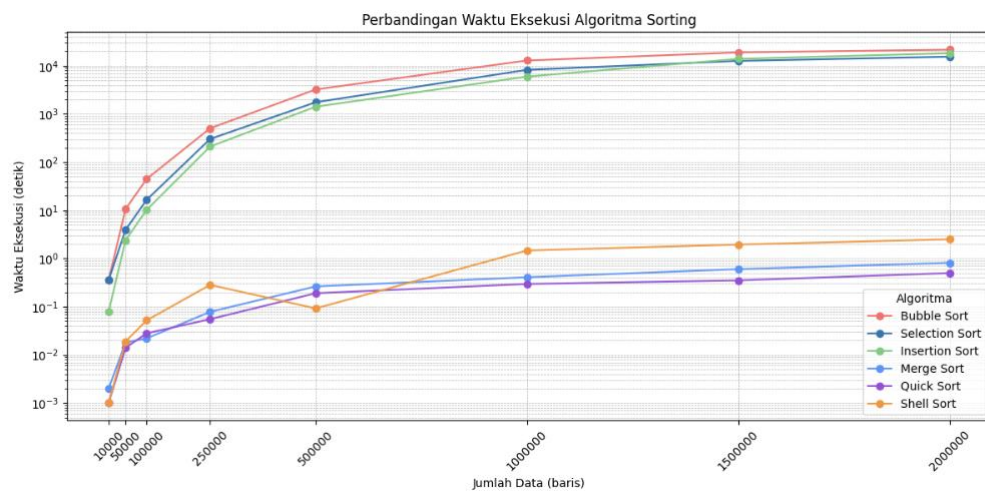
1. Perbandingan eksekusi waktu data angka



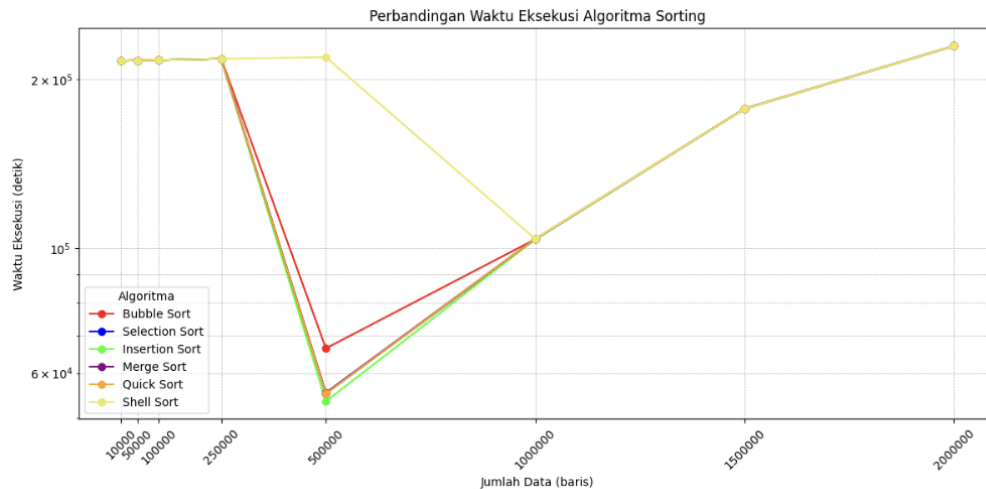
2. Perbandingan eksekusi Memori data angka



3. Perbandingan eksekusi waktu data kata



4. Perbandingan eksekusi Memori data kata



D. Analisis dan kesimpulan

Berdasarkan data yang disajikan mengenai waktu eksekusi dan penggunaan memori dari berbagai algoritma pengurutan (sorting), berikut adalah analisis yang lebih mendalam.

Dari segi waktu eksekusi, Bubble Sort menunjukkan kinerja yang paling buruk. Waktu eksekusinya meningkat tajam seiring dengan bertambahnya jumlah data, bahkan untuk ukuran data sebesar 1.000.000, waktu eksekusinya mencapai lebih dari 4 detik. Hal ini sesuai dengan teori bahwa Bubble Sort memiliki kompleksitas waktu $O(n^2)$, sehingga waktu eksekusinya akan sangat terpengaruh oleh jumlah data yang diurutkan.

Selection Sort dan Insertion Sort juga menunjukkan waktu eksekusi yang lebih tinggi dibandingkan algoritma lain, namun lebih baik dari Bubble Sort. Selection Sort memiliki kompleksitas waktu $O(n^2)$, sedangkan Insertion Sort cenderung lebih efisien pada dataset yang lebih kecil dengan kompleksitas $O(n^2)$ di kondisi terburuk, meskipun lebih cepat dibandingkan Bubble Sort pada ukuran data lebih besar.

Di sisi lain, Merge Sort, Quick Sort, dan Shell Sort menunjukkan waktu eksekusi yang jauh lebih efisien. Merge Sort dan Quick Sort, dengan kompleksitas waktu $O(n \log n)$, tidak terpengaruh secara signifikan oleh ukuran data dan memberikan kinerja yang stabil pada data besar. Quick Sort sedikit lebih unggul dibandingkan Merge Sort dalam hal waktu eksekusi. Shell Sort, meskipun sedikit lebih lambat dibandingkan Merge Sort dan Quick Sort, tetap menunjukkan performa yang baik dengan waktu eksekusi yang jauh lebih cepat dibandingkan algoritma $O(n^2)$.

Dari sisi penggunaan memori, secara teori, Bubble Sort, Selection Sort, dan Insertion Sort seharusnya memiliki penggunaan memori yang lebih rendah karena mereka memiliki kompleksitas ruang $O(1)$, yang berarti hanya membutuhkan sedikit ruang tambahan selain ruang untuk data input itu sendiri. Namun, dalam percobaan yang dilakukan, penggunaan memori terlihat relatif konsisten di semua algoritma, meskipun sedikit lebih tinggi pada Bubble Sort dan Selection Sort.

Merge Sort dan Quick Sort, yang memiliki kompleksitas ruang $O(n)$, membutuhkan lebih banyak memori karena mereka memerlukan ruang tambahan untuk menyimpan data yang dipecah-pecah (khususnya pada Merge Sort). Walaupun

demikian, penggunaan memori pada percobaan ini tidak sepenuhnya mencerminkan teori kompleksitas ruang yang diharapkan. Hal ini dapat disebabkan oleh faktor-faktor lain, seperti multitasking pada perangkat yang digunakan atau optimasi memori yang dilakukan oleh sistem operasi, yang dapat mempengaruhi alokasi memori selama pengujian.

Secara keseluruhan, algoritma Merge Sort dan Quick Sort merupakan pilihan terbaik dari segi efisiensi waktu dan penggunaan memori pada dataset besar, dengan Quick Sort sedikit lebih unggul. Shell Sort juga menunjukkan performa yang baik, meskipun lebih lambat daripada Merge Sort dan Quick Sort. Sementara itu, Bubble Sort harus dihindari pada dataset besar karena waktu eksekusinya yang sangat buruk dan penggunaan memori yang relatif tinggi. Selection Sort dan Insertion Sort lebih efisien daripada Bubble Sort, tetapi tetap kalah jauh dibandingkan algoritma-algoritma dengan kompleksitas $O(n \log n)$.

Pengujian ini dapat lebih akurat jika dilakukan dengan mengurangi faktor eksternal, seperti proses lain yang berjalan di perangkat yang sama tanpa gangguan secara terpisah, dan memastikan alokasi memori setiap algoritma dapat diekspresikan secara jelas sesuai teori kompleksitas ruang.