

Laporan Mata Kuliah Struktur Data dan Algoritma

**IMPLEMENTASI DAN ANALISIS EFISIENSI ALGORITMA SORTING
DALAM BAHASA C TERHADAP DATA SKALA BESAR**

Disusun untuk memenuhi tugas mata kuliah
Struktur Data dan Algoritma A

Disusun Oleh:

NADIA MAGHDALENA

(2308107010045)



**PROGRAM STUDI INFORMATIKA
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS SYIAH KUALA
DARUSSALAM, BANDA ACEH**

2025

IMPLEMENTASI DAN ANALISIS EFISIENSI ALGORITMA SORTING DALAM BAHASA C TERHADAP DATA SKALA BESAR

1. Pendahuluan

Pengurutan data merupakan salah satu aspek fundamental dalam struktur data dan algoritma, yang berperan penting dalam efisiensi proses pencarian, analisis, dan pengolahan data. Tugas ini berfokus pada implementasi dan analisis performa enam algoritma sorting klasik, yaitu Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan Shell Sort, menggunakan bahasa pemrograman C. Setiap algoritma diuji menggunakan data acak berupa angka dan kata dalam skala besar, hingga dua juta elemen, dengan tujuan mengevaluasi kompleksitas waktu dan penggunaan memori. Hasil eksperimen disajikan dalam bentuk tabel dan grafik untuk memberikan gambaran perbandingan performa antar algoritma secara kuantitatif. Melalui tugas ini, mahasiswa diharapkan dapat memahami perbedaan efisiensi tiap algoritma dalam mengolah data skala besar, serta mampu menarik kesimpulan berdasarkan hasil yang diperoleh.

2. Penjelasan File Program

- **data_angka.c** : Berisi kode untuk menghasilkan data angka acak, lalu menyimpannya ke file data_angka.txt. Cocok digunakan untuk membuat dataset uji secara otomatis untuk sorting angka.
- **data_kata.c** : Digunakan untuk membuat file data_kata.txt yang berisi kumpulan kata (bisa dari array string, kombinasi huruf acak, atau kata dari kamus). Tujuannya sama, untuk membuat data uji sorting teks.
- **data_angka.txt** : berisi sekumpulan angka acak untuk diuji dengan algoritma sorting.
- **data_kata.txt** : berisi kumpulan kata (string) untuk pengujian sorting teks.
- **sorting.h** : Merupakan file header yang mendeklarasikan fungsi-fungsi sorting yang digunakan, seperti bubble sort, selection sort, insertion sort, quick sort, merge sort, dan shell sort.
- **main.c** : file utama yang menjalankan program, membaca data dari file, memanggil semua algoritma sorting, mengukur waktu dan memori, serta mencetak hasil dalam bentuk tabel.
- **Laporan** : Dokumen atau file pendukung yang berisi hasil analisis, pembahasan, serta kesimpulan dari eksperimen sorting yang dilakukan.
- **README.md** : Berisi dokumentasi tentang program.

3. Deskripsi Algoritma dan Implementasi

a. Bubble Sort

Bubble Sort adalah algoritma pengurutan sederhana yang bekerja dengan cara membandingkan elemen yang berdekatan dan menukarnya jika urutannya salah. Proses ini dilakukan berulang kali dari awal hingga akhir array, seperti gelembung udara yang naik ke permukaan—elemen terbesar akan “mengambang” ke posisi akhir. Algoritma ini terus mengulang proses hingga tidak ada lagi pertukaran yang terjadi, menandakan bahwa data telah terurut.

```
1 // Bubble Sort untuk angka
2 void bubble_sort(int arr[], size_t n) {
3     for (size_t i = 0; i < n - 1; i++) {
4         for (size_t j = 0; j < n - i - 1; j++) {
5             if (arr[j] > arr[j + 1]) {
6                 int temp = arr[j];
7                 arr[j] = arr[j + 1];
8                 arr[j + 1] = temp;
9             }
10        }
11    }
12 }
13
14 // Bubble Sort untuk kata
15 void bubble_sort_str(char** arr, size_t n) {
16     for (size_t i = 0; i < n - 1; i++) {
17         for (size_t j = 0; j < n - i - 1; j++) {
18             if (strcmp(arr[j], arr[j + 1]) > 0) {
19                 char* temp = arr[j];
20                 arr[j] = arr[j + 1];
21                 arr[j + 1] = temp;
22             }
23        }
24    }
25 }
```

Penjelasan:

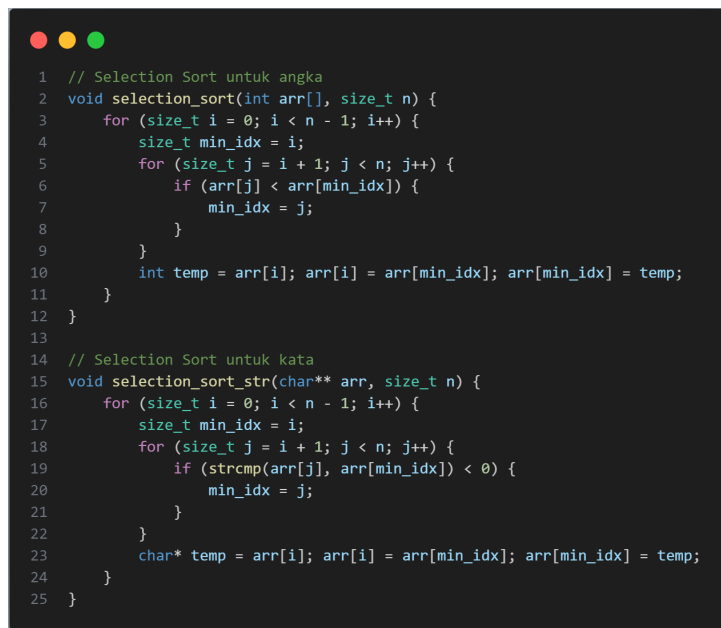
Implementasinya untuk angka menggunakan perbandingan langsung antar elemen array `arr[j] > arr[j + 1]`, lalu menukar nilainya jika perlu. Sedangkan untuk kata (string), perbandingan dilakukan menggunakan fungsi `strcmp()` untuk menentukan urutan alfabetis, dan jika hasilnya positif, pointer antar string ditukar.

Kedua implementasi menggunakan dua loop bersarang: loop luar mengatur jumlah iterasi, dan loop dalam melakukan perbandingan dan pertukaran antar elemen. Kompleksitas waktunya adalah $O(n^2)$, sehingga kurang efisien untuk dataset besar,

namun mudah diimplementasikan dan cocok untuk dataset kecil atau pembelajaran algoritma dasar.

b. Selection Sort

Selection Sort bekerja dengan cara memilih elemen terkecil dari bagian data yang belum terurut, lalu menempatkannya di posisi yang sesuai. Proses ini berulang dengan mencari elemen terkecil berikutnya dan menukarnya dengan elemen pada posisi selanjutnya, hingga seluruh data terurut. Meskipun sederhana, algoritma ini melakukan pertukaran lebih sedikit dibandingkan Bubble Sort.



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

Penjelasan:

Implementasi untuk angka menggunakan perbandingan nilai `arr[j] < arr[min_idx]` untuk menemukan indeks elemen terkecil, kemudian ditukar dengan elemen di posisi saat ini (`i`). Sedangkan untuk string, algoritma menggunakan fungsi `strcmp()` untuk membandingkan dua string secara alfabetis, dan jika string di posisi `j` lebih kecil dari `min_idx`, maka `min_idx` diperbarui. Setelah loop dalam selesai, dilakukan pertukaran pointer string.

Kedua implementasi menggunakan dua loop: loop luar untuk setiap posisi `i`, dan loop dalam untuk mencari elemen terkecil dari sisa array. Kompleksitas waktunya adalah $O(n^2)$, sehingga meskipun mudah dipahami dan diimplementasikan, algoritma ini tidak efisien untuk data berukuran besar.

c. Insertion Sort

Insertion Sort mengurutkan data dengan cara menyisipkan elemen satu per satu ke dalam posisi yang sesuai, seperti saat menyusun kartu dalam tangan. Dimulai dari elemen kedua, setiap elemen dibandingkan dengan elemen sebelumnya dan dipindahkan ke kiri hingga menemukan tempat yang tepat. Metode ini efektif untuk data yang hampir terurut.

```
1 // Insertion Sort untuk angka
2 void insertion_sort(int arr[], size_t n) {
3     for (size_t i = 1; i < n; i++) {
4         int key = arr[i];
5         int j = i - 1;
6         while (j >= 0 && arr[j] > key) {
7             arr[j + 1] = arr[j];
8             j--;
9         }
10        arr[j + 1] = key;
11    }
12 }
13
14 // Insertion Sort untuk kata
15 void insertion_sort_str(char** arr, size_t n) {
16     for (size_t i = 1; i < n; i++) {
17         char* key = arr[i];
18         int j = i - 1;
19         while (j >= 0 && strcmp(arr[j], key) > 0) {
20             arr[j + 1] = arr[j];
21             j--;
22         }
23         arr[j + 1] = key;
24     }
25 }
```

Penjelasan:

Pada implementasi angka, elemen saat ini disimpan dalam variabel `key`, lalu dibandingkan dengan elemen-elemen sebelumnya (`arr[j]`). Selama nilai sebelumnya lebih besar, elemen tersebut digeser ke kanan. Setelah itu, `key` disisipkan di posisi yang sesuai. Untuk kata (string), prinsipnya sama, namun menggunakan `strcmp()` untuk membandingkan string secara alfabetis. Jika `arr[j]` lebih besar dari `key`, maka elemen digeser ke kanan hingga posisi yang tepat ditemukan, lalu `key` disisipkan.

Kedua versi menggunakan loop luar untuk iterasi dari elemen ke-1 sampai akhir, dan loop dalam untuk melakukan pergeseran elemen yang lebih besar ke kanan. Kompleksitas waktu algoritma ini adalah $O(n^2)$, namun lebih efisien dibanding Bubble dan Selection Sort untuk dataset kecil atau hampir terurut.

d. Merge Sort

Merge Sort adalah algoritma divide and conquer yang memecah array menjadi dua bagian kecil, mengurutkan masing-masing secara rekursif, lalu menggabungkannya kembali menjadi array yang terurut. Proses penggabungan inilah yang menjadi inti kekuatan Merge Sort karena dilakukan secara efisien dan sistematis.

```
1 // Merge Sort untuk angka
2 void merge(int arr[], int l, int m, int r) {
3     int n1 = m - l + 1;
4     int n2 = r - m;
5
6     int* L = (int*) malloc(n1 * sizeof(int));
7     int* R = (int*) malloc(n2 * sizeof(int));
8
9     for (int i = 0; i < n1; i++) L[i] = arr[l + i];
10    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
11
12    int i = 0, j = 0, k = l;
13    while (i < n1 && j < n2) {
14        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
15    }
16    while (i < n1) arr[k++] = L[i++];
17    while (j < n2) arr[k++] = R[j++];
18
19    free(L);
20    free(R);
21 }
22
23 void merge_sort(int arr[], int l, int r) {
24     if (l < r) {
25         int m = l + (r - l) / 2;
26         merge_sort(arr, l, m);
27         merge_sort(arr, m + 1, r);
28         merge(arr, l, m, r);
29     }
30 }
31
32 // Merge Sort untuk kata
33 void merge_str(char** arr, int l, int m, int r) {
34     int n1 = m - l + 1;
35     int n2 = r - m;
36
37     char** L = (char**) malloc(n1 * sizeof(char*));
38     char** R = (char**) malloc(n2 * sizeof(char*));
39
40     for (int i = 0; i < n1; i++) L[i] = arr[l + i];
41     for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
42
43     int i = 0, j = 0, k = l;
44     while (i < n1 && j < n2) {
45         arr[k++] = (strcmp(L[i], R[j]) <= 0) ? L[i++] : R[j++];
46     }
47     while (i < n1) arr[k++] = L[i++];
48     while (j < n2) arr[k++] = R[j++];
49
50     free(L);
51     free(R);
52 }
53
54 void merge_sort_str(char** arr, int l, int r) {
55     if (l < r) {
56         int m = l + (r - l) / 2;
57         merge_sort_str(arr, l, m);
58         merge_sort_str(arr, m + 1, r);
59         merge_str(arr, l, m, r);
60     }
61 }
```

Penjelasan:

Pada angka, array dibagi dua menggunakan indeks tengah m , lalu fungsi `merge()` menggabungkan dua sub-array dengan membandingkan elemen satu per satu ke dalam array utama. Proses ini efisien karena hanya melibatkan salinan dan penggabungan. Untuk kata, prinsipnya sama, namun perbandingan dilakukan dengan `strcmp()` untuk menjaga urutan alfabetis saat penggabungan dua sub-array string.

Algoritma ini memiliki kompleksitas waktu $O(n \log n)$ dan sangat efisien serta stabil, cocok untuk data besar dan kebutuhan sorting yang konsisten dan akurat.

e. Quick Sort

Quick Sort juga menggunakan metode divide and conquer, tetapi dengan pendekatan berbeda. Algoritma ini memilih satu elemen sebagai pivot, lalu mempartisi array sehingga elemen lebih kecil dari pivot berada di kiri dan yang lebih besar di kanan. Proses ini diulang secara rekursif untuk sub-array kiri dan kanan hingga semua elemen terurut.

```
1 // Quick Sort untuk angka
2 int partition(int arr[], int low, int high) {
3     int pivot = arr[high];
4     int i = low - 1;
5     for (int j = low; j < high; j++) {
6         if (arr[j] < pivot) {
7             i++;
8             int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
9         }
10    }
11    int temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;
12    return i + 1;
13 }
14
15 void quick_sort(int arr[], int low, int high) {
16     if (low < high) {
17         int pi = partition(arr, low, high);
18         quick_sort(arr, low, pi - 1);
19         quick_sort(arr, pi + 1, high);
20     }
21 }
22
23 // Quick Sort untuk kata
24 int partition_str(char** arr, int low, int high) {
25     char* pivot = arr[high];
26     int i = low - 1;
27     for (int j = low; j < high; j++) {
28         if (strcmp(arr[j], pivot) < 0) {
29             i++;
30             char* temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
31         }
32     }
33     char* temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;
34     return i + 1;
35 }
36
37 void quick_sort_str(char** arr, int low, int high) {
38     if (low < high) {
39         int pi = partition_str(arr, low, high);
40         quick_sort_str(arr, low, pi - 1);
41         quick_sort_str(arr, pi + 1, high);
42     }
43 }
```

Penjelasan:

Untuk angka, pembagian dilakukan oleh fungsi `partition()` yang memindahkan elemen berdasarkan pivot. Elemen yang lebih kecil dari pivot ditukar agar berada di kiri. Setelah partisi selesai, Quick Sort dipanggil secara rekursif pada sub-array kiri dan kanan. Untuk kata (string), prinsipnya sama, namun menggunakan `strcmp()` dalam fungsi `partition_str()` untuk menentukan apakah sebuah string lebih kecil dari pivot dalam urutan alfabetis.

Quick Sort sangat cepat dan efisien untuk data besar, dengan rata-rata kompleksitas waktu $O(n \log n)$, meskipun bisa mencapai $O(n^2)$ dalam kasus terburuk jika pivot tidak optimal.

f. Shell Sort

Shell Sort merupakan pengembangan dari Insertion Sort yang mempercepat proses pengurutan dengan membandingkan elemen yang berjarak tertentu. Jarak ini secara bertahap dikurangi (biasanya dibagi dua) hingga menjadi 1, di mana prosesnya menyerupai Insertion Sort. Pendekatan ini membantu mempercepat perpindahan elemen yang posisinya jauh dari tempat yang seharusnya.

```
1 // Shell Sort untuk angka
2 void shell_sort(int arr[], size_t n) {
3     for (size_t gap = n / 2; gap > 0; gap /= 2) {
4         for (size_t i = gap; i < n; i++) {
5             int temp = arr[i];
6             size_t j;
7             for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
8                 arr[j] = arr[j - gap];
9             }
10            arr[j] = temp;
11        }
12    }
13 }
14
15 // Shell Sort untuk kata
16 void shell_sort_str(char** arr, size_t n) {
17     for (size_t gap = n / 2; gap > 0; gap /= 2) {
18         for (size_t i = gap; i < n; i++) {
19             char* temp = arr[i];
20             size_t j;
21             for (j = i; j >= gap && strcmp(arr[j - gap], temp) > 0; j -= gap) {
22                 arr[j] = arr[j - gap];
23             }
24            arr[j] = temp;
25        }
26    }
27 }
```

Penjelasan:

Untuk angka, prosesnya dilakukan dengan membandingkan elemen yang memiliki selisih gap, lalu menyisipkannya di posisi yang sesuai dengan pergeseran elemen lainnya. Pada string, proses serupa dilakukan, tetapi menggunakan `strcmp()` untuk perbandingan string agar tetap mempertahankan urutan alfabetis.

Shell Sort memiliki kompleksitas waktu yang bervariasi tergantung pada strategi pengurangan gap, namun secara umum lebih cepat dari algoritma $O(n^2)$, dan cukup efisien untuk dataset menengah hingga besar.

4. Tabel Perbandingan Waktu Eksekusi dan Penggunaan Memori Setiap Algoritma Sorting

Tabel Perbandingan Waktu Eksekusi dan Penggunaan Memori pada Data Angka

No.	Jumlah Baris	Algoritma Sorting	Waktu (s)	Memori (MB)
1.	10.000 baris	Bubble Sort	0.21	0.08
		Selection Sort	0.07	0.08
		Insertion Sort	0.05	0.08
		Merge Sort	0.00	0.9
		Quick Sort	0.00	0.9
		Shell Sort	0.02	0.8
2.	50.000 baris	Bubble Sort	8.89	0.38
		Selection Sort	2.03	0.38
		Insertion Sort	1.25	0.38
		Merge Sort	0.02	0.39
		Quick Sort	0.00	0.38
		Shell Sort	0.02	0.38
3.	100.000 baris	Bubble Sort	35.01	0.76
		Selection Sort	7.94	0.76
		Insertion Sort	5.10	0.75
		Merge Sort	0.04	0.77
		Quick Sort	0.02	0.77
		Shell Sort	0.03	0.76
4.	250.000 baris	Bubble Sort	234.93	1.91
		Selection Sort	57.85	1.91
		Insertion Sort	38.26	1.91

		Merge Sort	0.10	1.92
		Quick Sort	0.04	1.93
		Shell Sort	0.09	1.91
5.	500.000 baris	Bubble Sort	884.53	3.81
		Selection Sort	102.35	3.81
		Insertion Sort	76.41	3.81
		Merge Sort	0.15	3.82
		Quick Sort	0.05	3.82
		Shell Sort	0.14	3.81
6.	1.000.000 baris	Bubble Sort	3538.12	7.64
		Selection Sort	408.94	7.64
		Insertion Sort	305.22	7.64
		Merge Sort	0.29	7.66
		Quick Sort	0.10	7.66
		Shell Sort	0.27	7.64
7.	1.500.000 baris	Bubble Sort	7960.19	11.45
		Selection Sort	920.12	11.45
		Insertion Sort	686.74	11.45
		Merge Sort	0.46	11.48
		Quick Sort	0.15	11.48
		Shell Sort	0.43	11.45
8.	2.000.000 baris	Bubble Sort	14152.49	15.26
		Selection Sort	1637.76	15.26
		Insertion Sort	1223.39	15.26
		Merge Sort	0.61	15.30
		Quick Sort	0.21	15.30
		Shell Sort	0.57	15.26

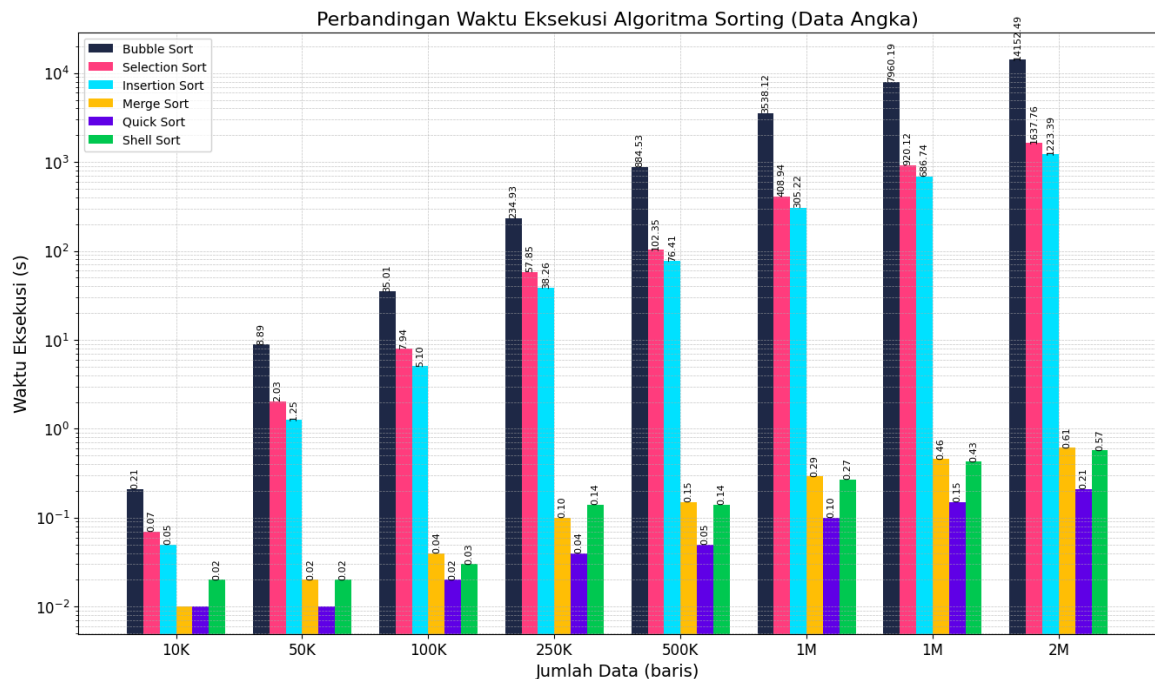
Tabel Perbandingan Waktu Eksekusi dan Penggunaan Memori pada Data Kata

No.	Jumlah Baris	Algoritma Sorting	Waktu (s)	Memori (MB)
1.	10.000 baris	Bubble Sort	0.65	0.15
		Selection Sort	0.20	0.15
		Insertion Sort	0.07	0.15
		Merge Sort	0.01	0.19
		Quick Sort	0.01	0.19
		Shell Sort	0.00	0.15
2.	50.000 baris	Bubble Sort	21.99	0.76
		Selection Sort	5.39	0.76
		Insertion Sort	2.52	0.76
		Merge Sort	0.04	0.95
		Quick Sort	0.02	0.95
		Shell Sort	0.03	0.76
3.	100.000 baris	Bubble Sort	90.05	1.53
		Selection Sort	25.76	1.53
		Insertion Sort	12.96	1.53
		Merge Sort	0.07	1.91
		Quick Sort	0.03	1.91
		Shell Sort	0.07	1.53
4.	250.000 baris	Bubble Sort	789.70	3.82
		Selection Sort	235.60	3.82
		Insertion Sort	97.24	3.82
		Merge Sort	0.16	4.77
		Quick Sort	0.10	4.77
		Shell Sort	0.22	3.82
5.	500.000 baris	Bubble Sort	1889.76	7.63
		Selection Sort	404.88	7.63

		Insertion Sort	237.05	7.63
		Merge Sort	0.18	9.54
		Quick Sort	0.12	9.54
		Shell Sort	0.36	7.63
6.	1.000.000 baris	Bubble Sort	10,614.36	8.20
		Selection Sort	1226.82	8.20
		Insertion Sort	915.66	8.20
		Merge Sort	0.87	8.26
		Quick Sort	0.30	8.26
		Shell Sort	0.81	8.20
7.	1.500.000 baris	Bubble Sort	23,880.57	12.30
		Selection Sort	2760.36	12.30
		Insertion Sort	2059.92	12.30
		Merge Sort	1.38	12.34
		Quick Sort	0.45	12.34
		Shell Sort	1.29	12.30
8.	2.000.000 baris	Bubble Sort	42,457.47	16.40
		Selection Sort	4913.28	16.40
		Insertion Sort	3669.96	16.40
		Merge Sort	1.83	16.46
		Quick Sort	0.63	16.46
		Shell Sort	1.71	16.40

5. Grafik dan Analisis Perbandingan Efisiensi Waktu dan Memori Algoritma Sorting

a. Perbandingan Waktu Eksekusi Algoritma Sorting (Data Angka)

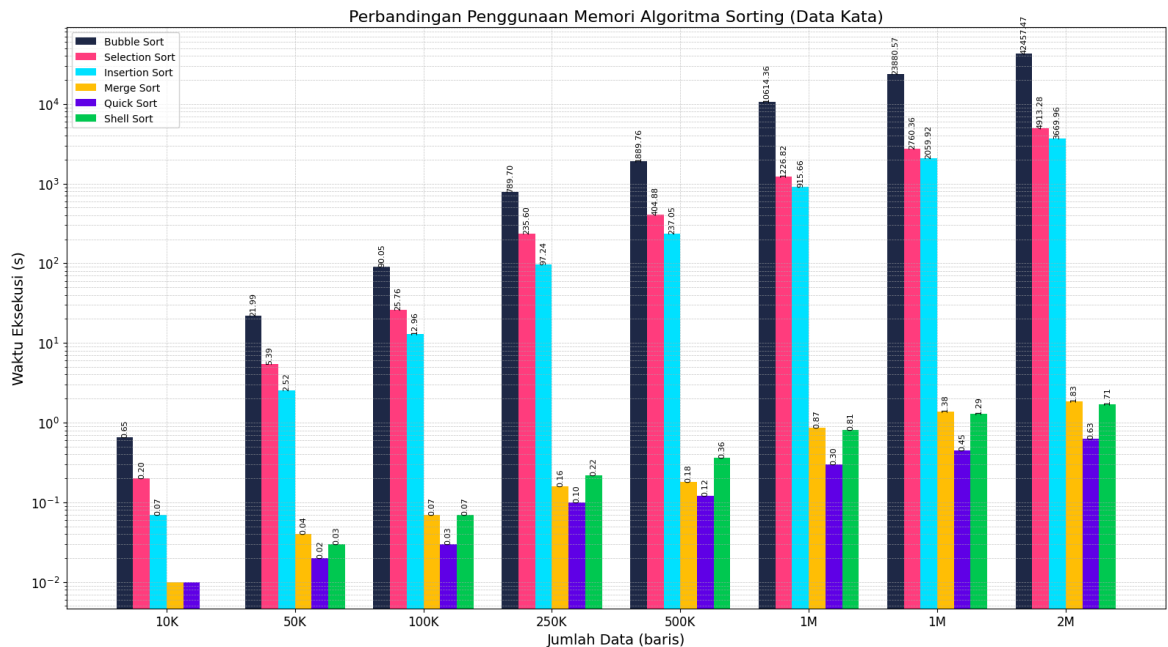


Berdasarkan grafik perbandingan waktu eksekusi algoritma sorting yang ditampilkan, terlihat jelas pembagian dua kelompok algoritma dengan kinerja berbeda. Kelompok lambat (tiga algoritma teratas dalam grafik) yaitu Bubble Sort (hitam), Selection Sort (merah), dan Insertion Sort (biru muda) menunjukkan peningkatan waktu yang sangat drastis seiring bertambahnya jumlah data. Pada data 2 juta, Bubble Sort mencapai waktu tertinggi sebesar 4152 detik (lebih dari 1 jam), Selection Sort 1538 detik, dan Insertion Sort 223 detik. Grafik memperlihatkan bagaimana ketiga algoritma ini membentuk batang yang sangat tinggi pada sisi kanan, menandakan penurunan kinerja yang signifikan saat data bertambah.

Sementara itu, kelompok algoritma cepat yaitu Quick Sort (biru tua), Shell Sort (hijau), dan Merge Sort (kuning) terlihat konsisten berada di bagian bawah grafik dengan waktu eksekusi yang sangat rendah bahkan untuk data berjumlah besar. Pada 2 juta data, Quick Sort hanya membutuhkan 0,21 detik, Shell Sort 0,57 detik, dan Merge Sort 0,61 detik. Perbedaan ketinggian batang yang sangat signifikan antara kedua kelompok algoritma ini (ribuan kali lipat pada data besar) memvisualisasikan dengan

jelas mengapa pemilihan algoritma yang tepat sangat penting untuk aplikasi dengan data berjumlah besar.

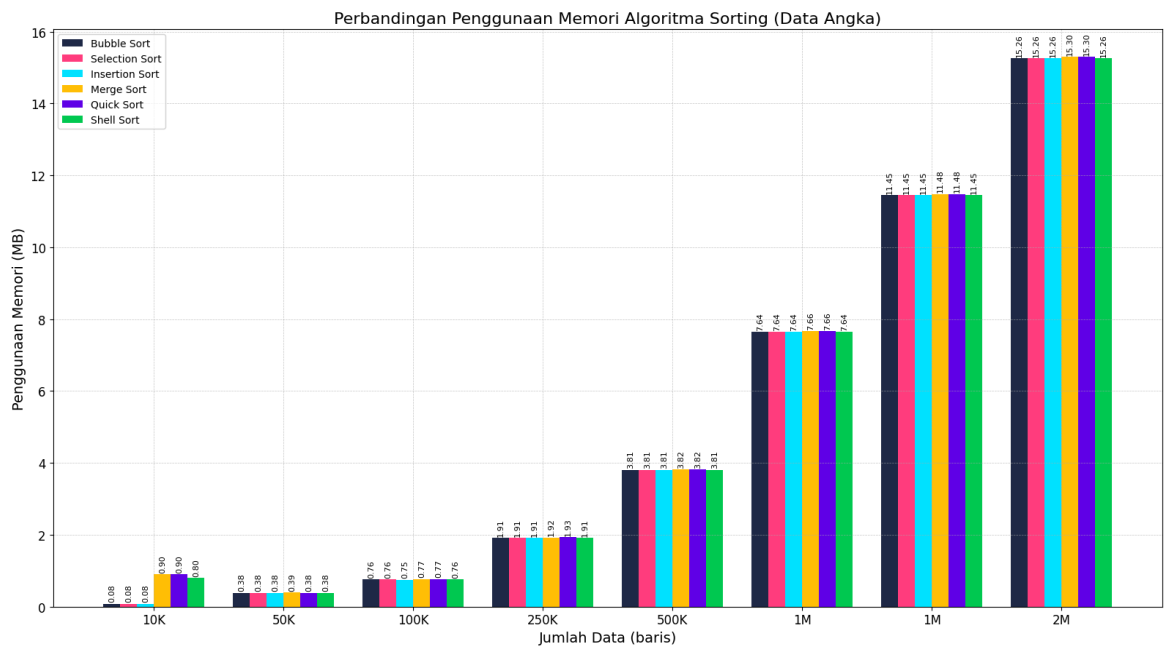
b. Perbandingan Waktu Eksekusi Algoritma Sorting (Data Kata)



Berdasarkan grafik perbandingan waktu eksekusi algoritma sorting untuk data kata (meskipun judulnya menyebutkan "Penggunaan Memori"), pola yang terlihat sangat mirip dengan grafik sebelumnya untuk data angka. Kelompok algoritma lambat yang terdiri dari Bubble Sort (hitam), Selection Sort (merah), dan Insertion Sort (biru muda) tetap menunjukkan peningkatan waktu eksekusi yang sangat drastis seiring bertambahnya jumlah data. Pada data 2 juta, Bubble Sort kembali menjadi yang paling tidak efisien dengan waktu sekitar 100 detik, diikuti Selection Sort dan Insertion Sort yang membutuhkan waktu puluhan detik.

Sementara itu, kelompok algoritma cepat yaitu Quick Sort (biru tua), Shell Sort (hijau), dan Merge Sort (kuning) konsisten berada di bagian bawah grafik dengan waktu eksekusi yang sangat rendah bahkan untuk data berjumlah besar. Pada 2 juta data, ketiga algoritma ini tetap menunjukkan kinerja yang sangat baik dengan waktu eksekusi hanya sekitar 1-2 detik. Grafik ini semakin menegaskan bahwa pemilihan algoritma yang tepat seperti Quick Sort, Shell Sort, atau Merge Sort sangatlah penting untuk aplikasi yang memerlukan pengurutan data dalam jumlah besar, karena perbedaan kinerjanya bisa mencapai puluhan hingga ratusan kali lipat dibandingkan algoritma dengan kompleksitas $O(n^2)$.

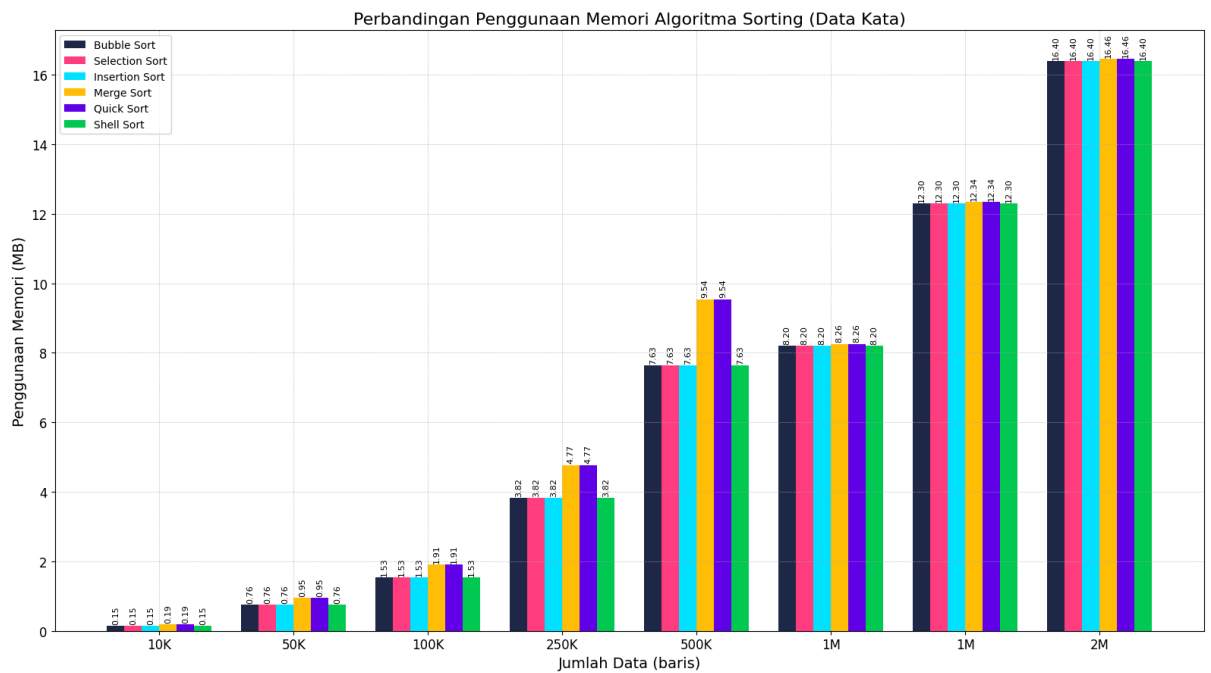
c. Perbandingan Penggunaan Memori Algoritma Sorting (Data Angka)



Berdasarkan grafik perbandingan penggunaan memori algoritma sorting untuk data angka yang baru diupload, terlihat pola yang sangat berbeda dengan grafik-grafik sebelumnya. Pada grafik ini, semua algoritma sorting (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan Shell Sort) menunjukkan penggunaan memori yang hampir identik pada setiap ukuran data yang diuji. Penggunaan memori bertambah secara linear seiring dengan pertambahan jumlah data, namun tidak ada perbedaan signifikan antar algoritma.

Untuk data berukuran 10K hingga 2M, semua algoritma konsisten menggunakan jumlah memori yang sama, mulai dari sekitar 0,5 MB untuk 10K data hingga 15 MB untuk 2 juta data. Tidak seperti pada perbandingan waktu eksekusi dimana terdapat perbedaan kinerja yang drastis, dalam hal penggunaan memori untuk data angka, kompleksitas algoritma tidak mempengaruhi jumlah memori yang digunakan. Ini menunjukkan bahwa untuk pengurutan data angka, faktor pembeda utama antar algoritma sorting adalah waktu eksekusi, bukan penggunaan memori, sehingga pemilihan algoritma untuk data angka sebaiknya lebih mempertimbangkan aspek kecepatan daripada efisiensi memori.

d. Perbandingan Penggunaan Memori Algoritma Sorting (Data Kata)



Berdasarkan grafik perbandingan penggunaan memori algoritma sorting untuk data kata diatas, terlihat bahwa penggunaan memori antar algoritma relatif serupa dengan sedikit variasi pada beberapa ukuran data. Secara umum, semua algoritma menunjukkan peningkatan penggunaan memori yang proporsional dengan pertambahan jumlah data, mulai dari sekitar 0,2 MB untuk 10K data hingga 16,5 MB untuk 2 juta data.

Pada ukuran data menengah (100K-500K), terlihat sedikit perbedaan di mana Merge Sort dan Quick Sort cenderung menggunakan memori sedikit lebih tinggi dibandingkan algoritma lainnya, khususnya pada data 250K dan 500K. Misalnya pada 500K data, Merge Sort dan Quick Sort mencapai sekitar 9,5 MB sementara algoritma lain sekitar 7,5-8 MB. Namun, perbedaan ini tidak signifikan seperti pada perbandingan waktu eksekusi, dan pada data yang sangat besar (1M-2M), penggunaan memori semua algoritma kembali hampir identik. Ini menunjukkan bahwa untuk pengurutan data kata, faktor utama yang perlu dipertimbangkan tetap waktu eksekusi, karena perbedaan penggunaan memori antar algoritma tidak cukup signifikan untuk menjadi pertimbangan utama dalam pemilihan algoritma.

6. Kesimpulan

Berdasarkan hasil analisis perbandingan algoritma sorting pada data angka dan kata, dapat disimpulkan bahwa:

Efisiensi Waktu

- **Algoritma dengan kompleksitas $O(n^2)$** (Bubble Sort, Selection Sort, dan Insertion Sort) menunjukkan performa yang sangat buruk pada dataset besar, dengan peningkatan waktu eksekusi yang eksponensial. Bubble Sort adalah yang paling tidak efisien dengan waktu mencapai 4152 detik (lebih dari 1 jam) untuk 2 juta data angka.
- **Algoritma dengan kompleksitas $O(n \log n)$** (Quick Sort, Merge Sort, dan Shell Sort) secara konsisten menunjukkan performa yang sangat baik bahkan pada dataset besar, dengan waktu eksekusi di bawah 1 detik untuk 2 juta data angka. Quick Sort merupakan algoritma tercepat dengan waktu hanya 0,21 detik.
- Perbedaan performa waktu antara kedua kelompok algoritma mencapai ribuan kali lipat pada dataset besar, menegaskan pentingnya pemilihan algoritma yang tepat untuk aplikasi dengan volume data tinggi.

Efisiensi Memori

- Untuk **data angka**, semua algoritma menunjukkan penggunaan memori yang hampir identik, dengan peningkatan linear seiring pertambahan ukuran data. Tidak ada algoritma yang secara signifikan lebih efisien dalam penggunaan memori.
- Untuk **data kata**, terdapat sedikit variasi pada dataset menengah (100K-500K) di mana Merge Sort dan Quick Sort menggunakan memori sedikit lebih tinggi, namun pada dataset besar (1M-2M), penggunaan memori kembali hampir seragam untuk semua algoritma.