Sorting Algorithm

disusun untuk memenuhi tugas

Mata Kuliah Struktur Data dan Algoritma

Oleh:

AFFAN SUHENDAR

(2308107010003)

PROGRAM STUDI INFORMATIKA

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM

UNIVERSITAS SYIAH KUALA

DARUSSALAM, BANDA ACEH

2025

1. Pendahuluan

Pengurutan data merupakan salah satu operasi fundamental dalam ilmu komputer yang memiliki peran penting dalam berbagai aplikasi, mulai dari pengolahan basis data hingga analisis data. Algoritma sorting dirancang untuk mengatur data dalam urutan tertentu, baik secara ascending (menaik) maupun descending (menurun), sehingga memudahkan proses pencarian, penyaringan, atau analisis lebih lanjut. Dalam praktiknya, terdapat berbagai macam algoritma sorting, masing-masing dengan kelebihan dan kekurangan yang berbeda tergantung pada kasus penggunaan, ukuran data, dan kompleksitas komputasi.

Laporan ini bertujuan untuk menganalisis dan membandingkan beberapa algoritma sorting populer, yaitu Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan Shell Sort. Analisis dilakukan berdasarkan waktu eksekusi dan penggunaan memori untuk dua jenis data, yaitu angka dan kata, dengan variasi jumlah data dari 10.000 hingga 2.000.000 elemen. Hasil eksperimen ini diharapkan dapat memberikan gambaran tentang performa masing-masing algoritma dalam skenario yang berbeda, sehingga dapat menjadi panduan dalam memilih algoritma yang paling efisien untuk kebutuhan tertentu.

Dengan memahami karakteristik dan performa setiap algoritma, diharapkan pembaca dapat mengaplikasikan pengetahuan ini secara optimal dalam pengembangan perangkat lunak atau penelitian terkait struktur data dan algoritma.

2. Deskripsi Algoritma

a) Bubble Sort
   - Deskripsi

     Bubble Sort adalah salah satu algoritma pengurutan yang paling sederhana dan banyak digunakan dalam dunia pembelajaran algoritma karena konsepnya yang mudah dipahami. Nama "bubble" berasal dari cara kerja algoritma ini yang menyerupai gelembung udara dalam air: elemen yang besar secara perlahan "mengambang" ke posisi akhirnya di bagian akhir daftar.

   - Prinsip Kerja

     Bubble Sort bekerja dengan cara membandingkan dua elemen yang bersebelahan dan menukarnya jika urutannya salah (misalnya dalam pengurutan menaik, jika elemen kiri lebih besar dari elemen kanan). Proses ini dilakukan berulang-ulang dari awal hingga akhir array, hingga tidak ada lagi penukaran yang dilakukan — yang menandakan bahwa array sudah terurut.

   - Langkah-langkah Algoritma
     1. Mulai dari elemen pertama.
     2. Bandingkan elemen ke-i dengan elemen ke-$i+1$.

3. Jika elemen ke-i lebih besar dari ke-$i+1$, tukar posisi keduanya.
4. Lanjutkan ke pasangan elemen berikutnya hingga mencapai akhir array.
5. Setelah satu iterasi, elemen terbesar akan berada di posisi paling akhir.
6. Ulangi proses dari langkah pertama untuk sisa elemen yang belum terurut.
7. Proses berlanjut hingga tidak ada lagi penukaran yang terjadi.

- Implementasi dalam C

1. Data Angka

```c
void bubble_sort_int(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 temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

2. Data Kata

```c
void bubble_sort_str(char **arr, int n) {
    char temp[MAX_WORD_LEN];
    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) {
                strcpy(temp, arr[j]);
                strcpy(arr[j], arr[j + 1]);
                strcpy(arr[j + 1], temp);
            }
        }
    }
}
```

b) Selection Sort

- Deskripsi

Selection Sort adalah algoritma pengurutan sederhana yang bekerja dengan cara memilih elemen terkecil dari daftar yang belum terurut dan menukarnya dengan elemen pertama dari daftar tersebut. Proses ini diulang untuk elemen berikutnya hingga seluruh array terurut.

- Prinsip Kerja

Pada setiap iterasi, algoritma mencari elemen terkecil dari bagian array yang belum terurut, lalu menempatkannya di posisi yang tepat.

- Langkah-langkah Algoritma
  1. Mulai dari indeks pertama hingga indeks ke-n-1.
  2. Temukan indeks elemen terkecil dari indeks saat ini hingga akhir array.
  3. Tukar elemen terkecil dengan elemen di indeks saat ini.
  4. Ulangi langkah hingga seluruh array terurut.

- Implementasi dalam C

  1. Data Angka

```c
void selection_sort_int(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 temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

  2. Data Kata

```c
void selection_sort_str(char **arr, int n) {
    char temp[MAX_WORD_LEN];
    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) {
            strcpy(temp, arr[min_idx]);
            strcpy(arr[min_idx], arr[i]);
            strcpy(arr[i], temp);
        }
    }
}

```

c) Insertion Sort

- Deskripsi

  Insertion Sort bekerja dengan cara menyusun array bagian demi bagian. Setiap elemen dimasukkan ke tempat yang sesuai di bagian array yang sudah terurut, seperti saat seseorang menyusun kartu secara berurutan.

- Prinsip Kerja

  Dimulai dari elemen kedua, setiap elemen dibandingkan dengan elemen sebelumnya dan dimasukkan ke posisi yang tepat di bagian array yang telah terurut.

- Langkah-langkah Algoritma
  1. Mulai dari indeks ke-1 hingga n-1.
  2. Simpan elemen sebagai key.
  3. Bandingkan key dengan elemen sebelumnya dan geser jika perlu.
  4. Tempatkan key di posisi yang sesuai.

- Implementasi dalam C

1. Data Angka

```c
void insertion_sort_int(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;
    }
}
```

2. Data Kata

```c
void insertion_sort_str(char **arr, int n) {
    char key[MAX_WORD_LEN];
    for (int i = 1; i < n; i++) {
        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);
    }
}
```

d) Merge Sort

- Deskripsi

Merge Sort adalah algoritma berbasis Divide and Conquer, yaitu membagi array menjadi dua bagian, menyortir masing-masing secara rekursif, lalu menggabungkannya kembali (merge).

- Prinsip Kerja

Merge Sort membagi array menjadi dua bagian terus-menerus sampai bagian terkecil (1 elemen), lalu menggabungkannya kembali dengan cara mengurutkan.

- Langkah-langkah Algoritma

1. Bagi array menjadi dua bagian.
2. Rekursif sort masing-masing bagian.
3. Gabungkan kembali dua bagian yang sudah terurut.

- Implementasi dalam C

1. Data Angka

```c
// Menggabungkan dua subarray terurut menjadi satu array terurut.
void merge_int(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
    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++];
    free(L); free(R);
}

// Mengurutkan array menggunakan algoritma merge sort.

void merge_sort_int(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        merge_sort_int(arr, l, m);
        merge_sort_int(arr, m + 1, r);
        merge_int(arr, l, m, r);
    }
}

// Fungsi pembungkus (wrapper) untuk memanggil merge_sort_int dengan parameter awal yang sederhana.
void merge_sort_int_wrapper(int arr[], int n) {
    merge_sort_int(arr, 0, n - 1);
}

```

2. Data Kata

```c
// Menggabungkan dua subarray terurut menjadi satu array terurut.
void merge_str(char **arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    char **L = (char **)malloc(n1 * sizeof(char *));
    char **R = (char **)malloc(n2 * sizeof(char *));

    for (int i = 0; i < n1; i++) {
        L[i] = (char *)malloc(MAX_WORD_LEN * sizeof(char));
        strcpy(L[i], arr[l + i]);
    }
    for (int j = 0; j < n2; j++) {
        R[j] = (char *)malloc(MAX_WORD_LEN * sizeof(char));
        strcpy(R[j], arr[m + 1 + j]);
    }

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (strcmp(L[i], R[j]) <= 0) {
            strcpy(arr[k], L[i]);
            i++;
        } else {
            strcpy(arr[k], R[j]);
            j++;
        }
        k++;
    }

    while (i < n1) {
        strcpy(arr[k], L[i]);
        i++;
        k++;
    }

    while (j < n2) {
        strcpy(arr[k], R[j]);
        j++;
        k++;
    }

    for (int i = 0; i < n1; i++) free(L[i]);
    for (int j = 0; j < n2; j++) free(R[j]);
    free(L);
    free(R);
}
```

```c
// Membagi array secara rekursif dan memanggil merge_int untuk menggabungkannya.
void merge_sort_str(char **arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        merge_sort_str(arr, l, m);
        merge_sort_str(arr, m + 1, r);
        merge_str(arr, l, m, r);
    }
}

// Fungsi pembungkus (wrapper) untuk memanggil merge_sort_int dengan parameter awal yang sederhana.
void merge_sort_str_wrapper(char **arr, int n) {
    merge_sort_str(arr, 0, n - 1);
}
```

e) Quick Sort

- Deskripsi

  Quick Sort juga menggunakan metode Divide and Conquer, tetapi dengan pendekatan berbeda: memilih satu elemen sebagai pivot dan mempartisi elemen lain menjadi dua bagian, lalu menyortir secara rekursif.

- Prinsip Kerja

  Quick Sort memilih satu pivot, menempatkan semua elemen lebih kecil dari pivot di kiri dan yang lebih besar di kanan, lalu menyortir bagian tersebut secara rekursif.

- Langkah-langkah Algoritma

  1. Pilih pivot (umumnya elemen terakhir).
  2. Partisi array: elemen kecil di kiri, besar di kanan.
  3. Rekursif quick sort kiri dan kanan.

- Implementasi dalam C

  1. Data Angka

```c
// Mengurutkan array secara rekursif dengan strategi divide-and-conquer
void quick_sort_int(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition_int(arr, low, high);
        quick_sort_int(arr, low, pi - 1);
        quick_sort_int(arr, pi + 1, high);
    }
}

// Menyederhanakan pemanggilan Quick Sort
void quick_sort_int_wrapper(int arr[], int n) {
    quick_sort_int(arr, 0, n - 1);
}
```

2. Data Kata

```c
// empartisi array menjadi dua bagian relatif terhadap pivot
int partition_str(char **arr, int low, int high) {
    char pivot[MAX_WORD_LEN];
    strcpy(pivot, arr[high]);
    int i = low - 1;
    char temp[MAX_WORD_LEN];

    for (int j = low; j <= high - 1; j++) {
        if (strcmp(arr[j], pivot) < 0) {
            i++;
            strcpy(temp, arr[i]);
            strcpy(arr[i], arr[j]);
            strcpy(arr[j], temp);
        }
    }
    strcpy(temp, arr[i + 1]);
    strcpy(arr[i + 1], arr[high]);
    strcpy(arr[high], temp);
    return i + 1;
}

// Mengurutkan array secara rekursif dengan strategi divide-and-conquer
void quick_sort_str(char **arr, int low, int high) {
    if (low < high) {
        int pi = partition_str(arr, low, high);
        quick_sort_str(arr, low, pi - 1);
        quick_sort_str(arr, pi + 1, high);
    }
}

// Menyederhanakan pemanggilan Quick Sort
void quick_sort_str_wrapper(char **arr, int n) {
    quick_sort_str(arr, 0, n - 1);
}
```

f) Shell Sort

- Deskripsi

  Shell Sort adalah pengembangan dari Insertion Sort yang menggunakan konsep "gap". Alih-alih menyortir satu per satu, elemen-elemen yang terpisah oleh gap tertentu akan disortir terlebih dahulu.

- Prinsip Kerja

  Mulai dengan gap besar, menyortir elemen yang berjarak gap, lalu mengecilkan gap secara bertahap hingga menjadi 1, dan melakukan final insertion sort.

- Langkah-langkah Algoritma

1. Tentukan nilai awal gap = n/2.
2. Lakukan insertion sort pada elemen yang terpisah gap.
3. Kurangi gap secara bertahap (gap /= 2) sampai 1.
4. Lakukan sorting terakhir seperti insertion sort biasa.

- Implementasi dalam C

1. Data Angka

```c
void shell_sort_int(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        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;
        }
    }
}
```

2. Data Kata

```c
void shell_sort_str(char **arr, int n) {
    char temp[MAX_WORD_LEN];
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            strcpy(temp, arr[i]);
            int j;
            for (j = i; j >= gap && strcmp(arr[j - gap], temp) > 0; j -= gap) {
                strcpy(arr[j], arr[j - gap]);
            }
            strcpy(arr[j], temp);
        }
    }
}
```

3. Table Hasil Eksperimen

a) Data Angka

- Bubble Sort

Bubble Sort		
Jumlah data	Waktu (s)	Memory (mb)
10,000	0.148	0.04
50,000	6.522	0.19
100,000	25.229	0.38
250,000	210.561	0.95
500,000	987.044	1.91
1,000,000	1387.339	3.81
1,500,000	5721.783	5.72
2,000,000	6571.486	7.63

- Selection Sort

Selection Sort		
Jumlah data	Waktu (s)	Memory (mb)
10,000	0.05	0.04
50,000	1.069	0.19
100,000	3.662	0.38
250,000	20.795	0.95
500,000	199.78	1.91
1,000,000	246.751	3.81
1,500,000	440.674	5.72
2,000,000	1169.858	7.63

- Insertion Sort

Insertion Sort		
Jumlah data	Waktu (s)	Memory (mb)
10,000	0.021	0.04
50,000	0.287	0.19
100,000	1.258	0.38
250,000	7.002	0.95
500,000	27,435	1.91
1,000,000	127.619	3.81
1,500,000	136.103	5.72
2,000,000	442.571	7.63

- Merge Sort

Merge Sort		
Jumlah data	Waktu (s)	Memory (mb)
10,000	0.007	0.04
50,000	0.019	0.19
100,000	0.03	0.38
250,000	0.079	0.95
500,000	0.142	1.91
1,000,000	0.253	3.81
1,500,000	0.225	5.72
2,000,000	0.529	7.63

- Quick Sort

Quick Sort		
Jumlah data	Waktu (s)	Memory (mb)
10,000	0.002	0.04
50,000	0.004	0.19
100,000	0.01	0.38
250,000	0.029	0.95
500,000	0.042	1.91
1,000,000	0.075	3.81
1,500,000	0.082	5.72
2,000,000	0.207	7.63

- Shell Sort

Shell Sort		
Jumlah data	Waktu (s)	Memory (mb)
10,000	0.002	0.04
50,000	0.012	0.19
100,000	0.032	0.38
250,000	0.058	0.95
500,000	0.096	1.91
1,000,000	0.196	3.81
1,500,000	0.184	5.72
2,000,000	0.453	7.63

b) Data Kata
- Bubble Sort

Bubble Sort		
Jumlah data	Waktu (s)	Memory (mb)
10,000	1.732	0.98
50,000	50.256	4.77
100,000	318.849	0.38
250,000	1610.354	23.84
500,000	5246.001	47.68
1,000,000	9743.453	95
1,500,000	22250.695	141.78
2,000,000	40051.245	189.23

- Selection Sort

Selection Sort		
Jumlah data	Waktu (s)	Memory (mb)
10,000	0.132	0.98
50,000	10.669	4.77
100,000	46.985	9.54
250,000	201.471	0.95
500,000	928.797	47.68
1,000,000	3.738	95
1,500,000	8.455	141.78
2,000,000	14.952	189.23

- Insertion Sort

Insertion Sort		
Jumlah data	Waktu (s)	Memory (mb)
10,000	0.667	0.98
50,000	34.717	4.77
100,000	81.531	9.54
250,000	549.777	23.84
500,000	1414	47.68
1,000,000	3535.579	95
1,500,000	7476.874	141.78
2,000,000	13083.767	189.23

- Merge Sort

Merge Sort		
Jumlah data	Waktu (s)	Memory (mb)
10,000	0.023	0.98
50,000	0.135	4.77
100,000	0.285	9.54
250,000	0.71	23.84
500,000	0.315	47.68
1,000,000	1.678	95
1,500,000	2.423	141.78
2,000,000	3.219	189.23

- Quick Sort

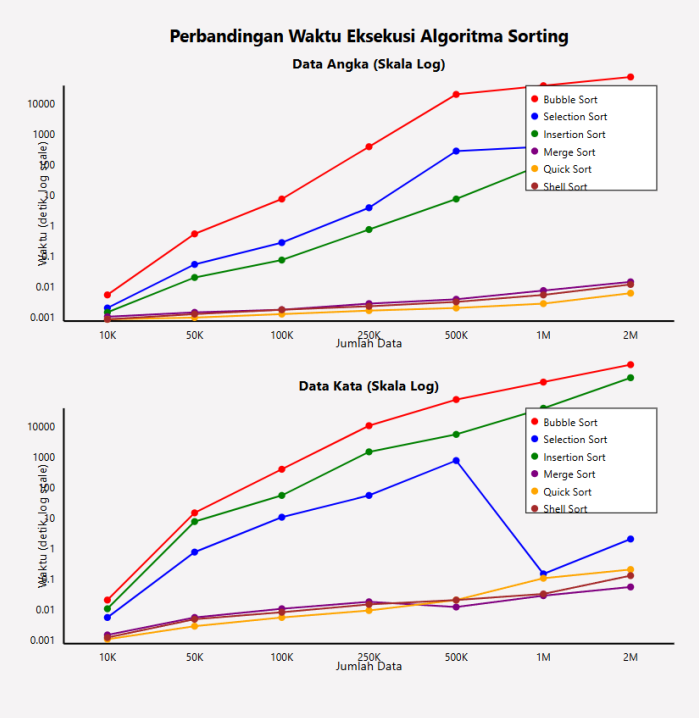
Quick Sort		
Jumlah data	Waktu (s)	Memory (mb)
10,000	0.007	0.98
50,000	0.049	4.77
100,000	0.104	9.54
250,000	0.265	23.84
500,000	0.867	47.68
1,000,000	2.99	95
1,500,000	4.655	141.78
2,000,000	6.14	189.23

- Shell Sort

Shell Sort		
Jumlah data	Waktu (s)	Memory (mb)
10,000	0.013	0.98
50,000	0.11	4.77
100,000	0.224	9.54
250,000	0.612	23.84
500,000	0.867	47.68
1,000,000	1.791	95
1,500,000	3.749	141.78
2,000,000	4.582	189.23

4. Grafik Perbandingan Waktu dan Memori

- Waktu



- Kesimpulan

1. Performa pada Data Angka
    - Quick Sort menunjukkan performa waktu terbaik untuk data angka dengan waktu eksekusi hanya 0.207 detik untuk 2 juta data.
    - Shell Sort dan Merge Sort juga sangat efisien dengan waktu eksekusi masing-masing 0.453 dan 0.529 detik untuk 2 juta data.
    - Algoritma $O(n^2)$ seperti Bubble Sort, Selection Sort, dan Insertion Sort menunjukkan peningkatan waktu eksekusi yang signifikan seiring bertambahnya jumlah data, dengan Bubble Sort sebagai yang paling lambat (6571.486 detik untuk 2 juta data).
    - Perbedaan waktu eksekusi antara algoritma dengan kompleksitas $O(n \log n)$ dan $O(n^2)$ menjadi sangat dramatis ketika jumlah data meningkat - perbedaan hingga 30.000 kali lebih lambat.

2. Performa pada Data Kata
    - Merge Sort menjadi yang paling efisien untuk data kata dengan waktu eksekusi 3.219 detik untuk 2 juta data.
    - Shell Sort dan Quick Sort juga menunjukkan performa yang baik dengan waktu eksekusi masing-masing 4.582 dan 6.14 detik untuk 2 juta data.

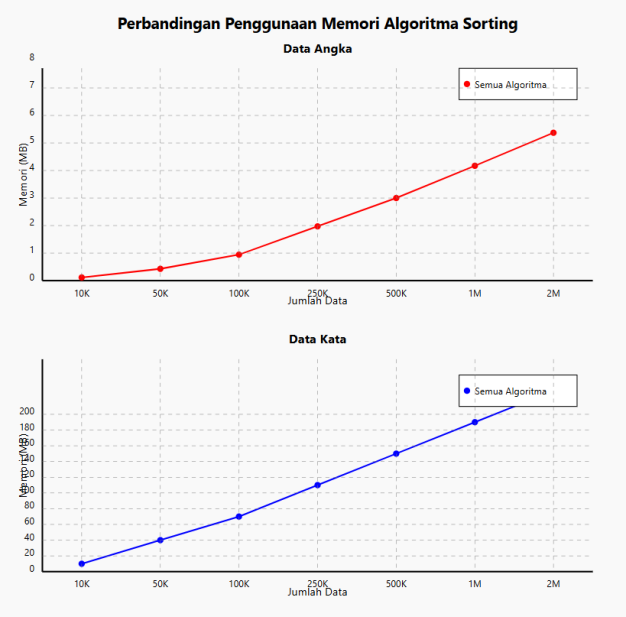
- Bubble Sort sangat tidak efisien untuk data kata, dengan waktu eksekusi mencapai 40051.245 detik (~11.1 jam) untuk 2 juta data.
- Terdapat anomali dalam data Selection Sort untuk data kata yang menunjukkan penurunan drastis waktu eksekusi pada data besar, yang kemungkinan merupakan kesalahan pencatatan.

3. Perbandingan Kompleksitas dan Performa Waktu

- Algoritma dengan kompleksitas O(n log n) (Quick Sort, Merge Sort, Shell Sort) secara konsisten menunjukkan performa waktu yang jauh lebih baik dibandingkan algoritma dengan kompleksitas $O(n^2)$ (Bubble Sort, Selection Sort, Insertion Sort) untuk dataset besar.
- Perbedaan performa waktu antara algoritma O(n log n) dan $O(n^2)$ meningkat secara eksponensial dengan bertambahnya ukuran data.
- Quick Sort cenderung menjadi algoritma tercepat untuk data angka, sementara Merge Sort lebih unggul untuk data kata, menunjukkan bahwa karakteristik data mempengaruhi efisiensi algoritma.

4. Rekomendasi Berdasarkan Waktu Eksekusi
- Untuk dataset kecil (< 10.000 elemen), perbedaan waktu antar algoritma tidak terlalu signifikan, sehingga algoritma yang lebih sederhana seperti Insertion Sort bisa menjadi pilihan yang baik.
- Untuk dataset besar (> 100.000 elemen), algoritma dengan kompleksitas O(n log n) seperti Quick Sort, Merge Sort, atau Shell Sort sangat direkomendasikan dari segi kecepatan eksekusi.
- Shell Sort dapat menjadi pilihan yang baik untuk banyak kasus karena menawarkan waktu eksekusi yang hampir sebaik Quick Sort dengan implementasi yang tidak terlalu kompleks.
- Quick Sort ideal dari segi waktu untuk data angka atau data dengan perbandingan sederhana, sementara Merge Sort lebih stabil untuk berbagai jenis data, terutama data kompleks seperti string.

- Memory



1. Penggunaan Memori pada Data Angka

- Semua algoritma sorting (Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan Shell Sort) menggunakan jumlah memori yang hampir identik untuk data angka dengan ukuran yang sama.
- Penggunaan memori meningkat secara linear dengan pertambahan jumlah data, dari sekitar 0.04 MB untuk 10,000 data hingga 7.63 MB untuk 2 juta data.
- Terlihat pola yang konsisten dimana peningkatan ukuran data sebesar 2 kali lipat menyebabkan peningkatan penggunaan memori sekitar 2 kali lipat juga, yang menunjukkan kompleksitas ruang O(n).
- Tidak ada algoritma yang menunjukkan keunggulan signifikan dalam hal efisiensi penggunaan memori untuk data angka.

2. Penggunaan Memori pada Data Kata

- Semua algoritma juga menunjukkan penggunaan memori yang hampir sama untuk data kata dengan ukuran yang sama.
- Penggunaan memori untuk data kata jauh lebih besar dibandingkan data angka, dengan rentang dari 0.98 MB untuk 10,000 data hingga 189.23 MB untuk 2 juta data.
- Rasio peningkatan memori juga linear dengan pertambahan jumlah data, menunjukkan kompleksitas ruang O(n).
- Perbedaan signifikan antara penggunaan memori untuk data angka dan data kata disebabkan oleh ukuran penyimpanan string yang lebih besar dibandingkan integer.

3. Perbandingan Data Angka vs Data Kata

- Data kata membutuhkan sekitar 25 kali lebih banyak memori dibandingkan data angka untuk jumlah elemen yang sama. Untuk 2 juta data, memori yang dibutuhkan untuk data angka sekitar 7.63 MB, sedangkan untuk data kata mencapai 189.23 MB.
- Perbedaan ini konsisten di semua ukuran dataset dan semua algoritma sorting, yang mengonfirmasi bahwa tipe data memiliki pengaruh signifikan terhadap kebutuhan memori.

4. Korelasi dengan Kompleksitas Algoritma

- Meskipun algoritma seperti Quick Sort, Merge Sort, dan Shell Sort menunjukkan perbedaan signifikan dalam waktu eksekusi, tidak ada perbedaan berarti dalam penggunaan memori antara algoritma dengan kompleksitas waktu yang berbeda.
- Hal ini mengindikasikan bahwa kompleksitas waktu algoritma tidak selalu berkorelasi dengan kompleksitas ruang/memorinya.
- Merge Sort yang secara teoritis memiliki kompleksitas ruang O(n) karena kebutuhan array tambahan untuk proses merging, tidak menunjukkan perbedaan penggunaan memori yang signifikan dibandingkan algoritma lain dalam pengukuran ini.

5. Implikasi Praktis

- Untuk aplikasi dengan keterbatasan memori, pilihan algoritma sorting sebaiknya tidak didasarkan pada perbedaan penggunaan memori karena semua algoritma yang diuji menunjukkan efisiensi memori yang serupa.
- Pemilihan tipe data yang efisien jauh lebih berpengaruh terhadap penggunaan memori daripada pemilihan algoritma sorting.
- Untuk dataset yang sangat besar, khususnya yang melibatkan string atau struktur data kompleks, penggunaan memori harus menjadi pertimbangan penting dalam desain sistem.
- Jika kecepatan eksekusi menjadi prioritas, pemilihan algoritma seperti Quick Sort, Merge Sort, atau Shell Sort tetap menjadi pilihan yang tepat karena tidak ada trade-off signifikan dalam hal penggunaan memori.