STRUKTUR DATA DAN ALGORITMA

Disusun Oleh:

Razian Sabri **2308107010050**

PROGRAM STUDI INFORMATIKA

FAKULTAS MIPA UNIVERSITAS SYIAH KUALA

DARUSSALAM, BANDA ACEH

2025

Deskripsi dan Implementasi Algoritma Sorting

1. Bubble Sort

Bubble Sort adalah algoritma pengurutan sederhana yang bekerja dengan membandingkan dan menukar elemen-elemen berdekatan secara berulang hingga tidak ada lagi pertukaran yang diperlukan, menandakan data telah terurut. Algoritma ini memiliki kompleksitas waktu $O(n^2)$ pada kasus terburuk (data terbalik) dan rata-rata, namun $O(n)$ pada kasus terbaik (data sudah terurut), dengan sifat *stable* (mempertahankan urutan elemen bernilai sama) dan *in-place* (hanya membutuhkan memori tambahan konstan).

```
void bubbleSort(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. Selection Sort

Selection Sort adalah algoritma pengurutan sederhana yang bekerja dengan cara mencari elemen terkecil (atau terbesar, tergantung pengurutan) dari bagian array yang belum terurut dan menukarnya dengan elemen di posisi saat ini, lalu mengulangi proses tersebut hingga seluruh array terurut. Algoritma ini memiliki kompleksitas waktu $O(n^2)$ untuk semua kasus (terbaik, rata-rata, dan terburuk) karena selalu melakukan perbandingan penuh meskipun data sudah terurut, bersifat *in-place* (tidak memerlukan memori tambahan signifikan), namun tidak stabil karena pertukaran elemen dapat mengubah urutan relatif nilai yang sama.

```
void selectionSort(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;
   }
}
```

3. Insertion Sort

Insertion Sort adalah algoritma pengurutan sederhana yang bekerja dengan cara membangun bagian array yang sudah terurut satu per satu, di mana setiap elemen baru disisipkan pada posisi yang tepat dalam bagian terurut tersebut. Algoritma ini memiliki kompleksitas waktu $O(n^2)$ pada kasus terburuk (data terurut

terbalik) dan rata-rata, namun O(n) pada kasus terbaik (data sudah hampir terurut), menjadikannya efisien untuk dataset kecil atau hampir terurut. Insertion Sort bersifat stabil (mempertahankan urutan elemen bernilai sama), in-place (hanya membutuhkan memori tambahan konstan), dan sering digunakan secara praktis karena implementasinya yang sederhana dan kinerja optimal untuk data kecil atau sebagai bagian dari algoritma hybrid seperti Timsort.

```
void insertionSort(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;
   }
}
```

4. Merge Sort

Merge Sort adalah algoritma pengurutan berbasis *divide and conquer* yang bekerja dengan membagi array menjadi dua bagian secara rekursif, mengurutkan masing-masing bagian, lalu menggabungkannya (merge) kembali secara terurut**.** Algoritma ini memiliki kompleksitas waktu O(n log n) untuk semua kasus (terbaik, rata-rata, dan terburuk), menjadikannya sangat efisien untuk dataset besar. Merge Sort bersifat *stabil* (urutan elemen bernilai sama tetap terjaga) dan *tidak in-place* (membutuhkan memori tambahan O(n) untuk proses penggabungan), serta cocok untuk pengurutan eksternal (data di penyimpanan eksternal) atau struktur data linked list.

```
void merge(int arr[], int l, int m, int r) {
   int n1 = m - l + 1;
   int n2 = r - m;
   int L[n1], R[n2];
   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++];
}

void mergeSort(int arr[], int l, int r) {
   if (l < r) {
      int m = (l + r)/2;
      mergeSort(arr, l, m);
      mergeSort(arr, m+1, r);
      merge(arr, l, m, r);
```

```
    }
}
```

5. Quick Sort

Quick Sort adalah algoritma pengurutan *divide and conquer* yang bekerja dengan memilih sebuah elemen pivot, lalu mengatur elemen lain sehingga yang lebih kecil dari pivot berada di kirinya dan yang lebih besar di kanannya, kemudian mengulangi proses ini secara rekursif pada sub-array kiri dan kanan pivot. Algoritma ini memiliki kompleksitas waktu O(n log n) pada kasus terbaik (pivot membagi array secara seimbang) dan rata-rata, tetapi $O(n^2)$ pada kasus terburuk (pivot terpilih adalah elemen terkecil/terbesar). Quick Sort bersifat *in-place* (umumnya hanya membutuhkan memori tambahan O(log n) untuk rekursi) dan *tidak stabil*, namun sangat efisien dalam praktik karena konstanta operasinya yang kecil, menjadikannya salah satu algoritma pengurutan tercepat untuk data acak.

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

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

6. Shell Sort

Shell Sort adalah algoritma pengurutan yang mengoptimasi Insertion Sort dengan cara mengurutkan elemen-elemen berjarak tertentu (gap) terlebih dahulu**,** lalu secara bertahap mengurangi jarak (gap) hingga mencapai 1 (di mana ia menjadi Insertion Sort biasa). Algoritma ini memiliki kompleksitas waktu bergantung pada urutan gap yang digunakan, dengan kasus terbaik mencapai O(n log n) (untuk urutan gap seperti Hibbard atau Sedgewick) dan kasus terburuk $O(n^2)$ (untuk gap sederhana seperti n/2, n/4,...), namun dalam praktik sering mendekati O(n^(3/2)). Shell Sort bersifat *in-place* dan *tidak stabil*, tetapi lebih efisien daripada Insertion Sort untuk data berukuran sedang karena mengurangi jumlah perbandingan dan pertukaran elemen yang jauh. Meskipun kurang populer dibanding Quick Sort atau Merge Sort, Shell Sort tetap berguna dalam lingkungan dengan memori terbatas karena implementasinya yang ringan.

```
void shellSort(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;
        }
    }
}
```

Tabel Hasil Eksperimen

1. Data Angka - Waktu Eksekusi (ms)

Algoritma	10.000	50.000	100.000	250.000	500.000	1.000.000	1.500.000	2.000.000
Bubble Sort	163	5.730	23.495	-	-	-	-	-
Selection Sort	70	1.760	7.205	-	-	-	-	-
Insertion Sort	47	1.205	4.809	30.049	-	-	-	-
Merge Sort	2	12	25	65	145	270	417	561
Quick Sort	1	5	11	26	58	129	211	313
Shell Sort	1	10	21	63	125	273	447	624

2. Data Kata - Waktu Eksekusi (ms)

Algoritma	10.000	50.000	100.000	250.000	500.000	1.000.000	1.500.000	2.000.000
Bubble Sort	440	14.131	61.815	-	-	-	-	-
Selection Sort	169	5.059	23.841	-	-	-	-	-
Insertion Sort	77	2.680	12.074	254.011	-	-	-	-
Merge Sort	2	16	35	108	261	543	853	1.196
Quick Sort	1	9	25	95	194	431	744	998
Shell Sort	2	23	60	316	761	1.869	3.097	4.436

3. Penggunaan RAM (KB) - Data Angka

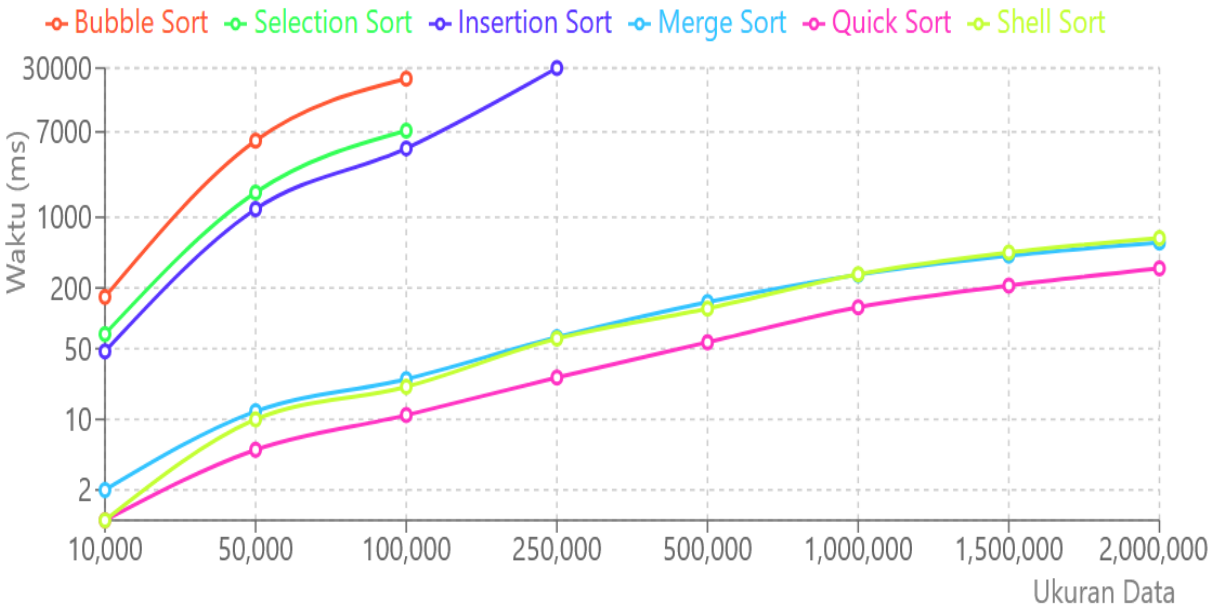
Algoritma	10.000	50.000	100.000	250.000	500.000	1.000.000	1.500.000	2.000.000
Bubble Sort	3.068	3.608	4.064	-	-	-	-	-
Selection Sort	3.088	3.608	4.064	-	-	-	-	-
Insertion Sort	3.272	4.024	4.112	5.376	-	-	-	-
Merge Sort	3.280	3.644	4.060	6.244	11.228	14.200	18.588	23.568
Quick Sort	3.280	3.644	4.060	6.244	11.228	14.200	18.588	23.568
Shell Sort	3.280	3.644	4.060	5.268	11.228	14.200	18.588	23.568

4. Penggunaan RAM (KB) - Data Kata

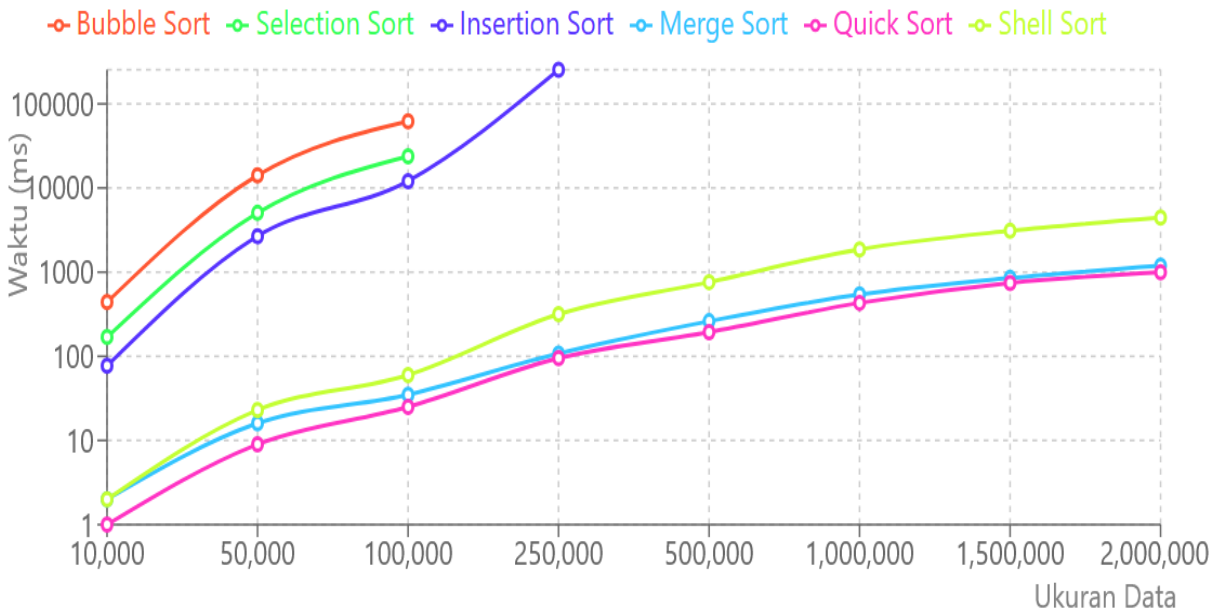
Algoritma	10.000	50.000	100.000	250.000	500.000	1.000.000	1.500.000	2.000.000
Bubble Sort	3.752	6.556	10.080	-	-	-	-	-
Selection Sort	3.780	6.572	10.116	-	-	-	-	-
Insertion Sort	3.820	6.560	10.108	20.412	-	-	-	-
Merge Sort	3.768	7.048	11.176	22.500	37.932	75.356	109.936	145.296
Quick Sort	3.768	6.588	10.136	20.476	37.896	72.820	107.800	142.760
Shell Sort	3.768	6.880	10.116	20.460	37.936	72.856	107.820	142.752

Grafik Perbandingan Waktu dan Memory

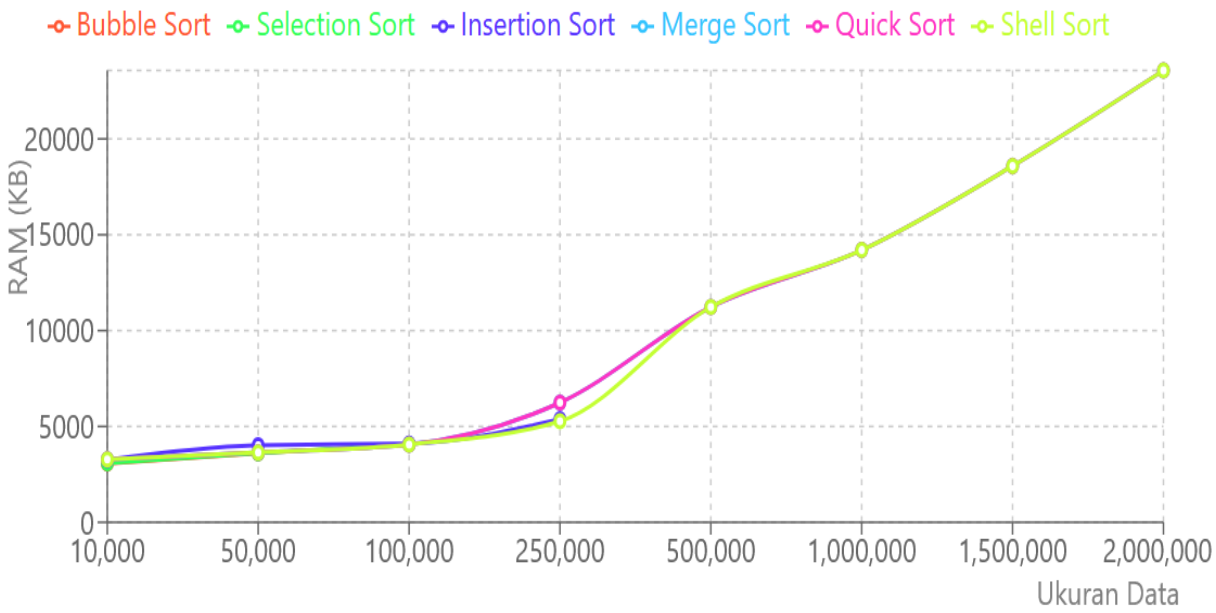
1. Data Angka - Waktu Eksekusi (ms)



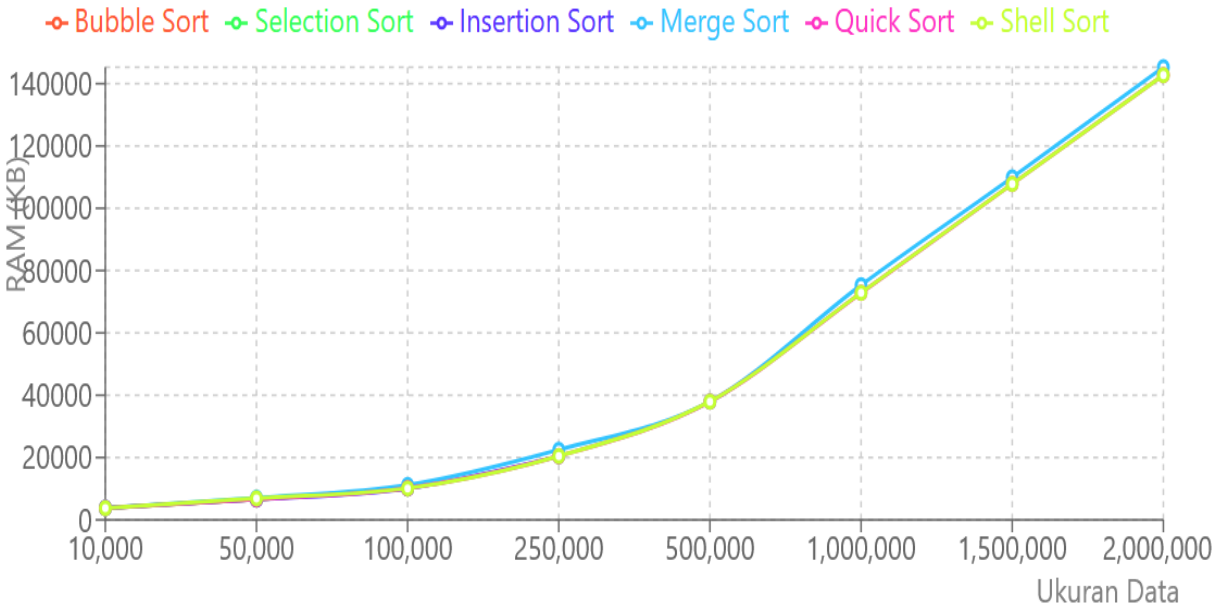
2. Data Kata - Waktu Eksekusi (ms)



3. Penggunaan RAM (KB) - Data Angka



4. Penggunaan RAM (KB) - Data Kata



Analisis dan Kesimpulan

1. Analisis Performa Waktu Eksekusi

Perbandingan antara algoritma dengan kompleksitas $O(n^2)$ dan $O(n \log n)$ menunjukkan perbedaan performa yang sangat signifikan dalam konteks pengurutan data. Algoritma $O(n^2)$ seperti Bubble Sort, Selection Sort, dan Insertion Sort mengalami peningkatan waktu eksekusi yang sangat drastis ketika ukuran data bertambah. Pada data berukuran 100.000 elemen, Bubble Sort membutuhkan 23.495 ms untuk data angka dan 61.815 ms untuk data kata. Algoritma $O(n^2)$ menjadi tidak praktis untuk dataset berukuran lebih dari 250.000 elemen karena waktu eksekusi yang tidak wajar.

Sebaliknya, algoritma $O(n \log n)$ seperti Merge Sort, Quick Sort, dan Shell Sort menunjukkan performa yang jauh lebih baik dan stabil. Quick Sort konsisten menjadi algoritma tercepat di hampir semua skenario, membutuhkan hanya 1-2 ms untuk dataset 10.000 elemen dan 11-25 ms untuk dataset 100.000 elemen. Bahkan pada dataset berukuran 2.000.000 elemen, Quick Sort hanya membutuhkan 313 ms untuk data angka dan 998 ms untuk data kata. Perbedaan performa antara kedua kategori algoritma ini bisa mencapai ribuan kali lipat pada dataset besar.

Tipe data yang diurutkan juga memberikan pengaruh besar terhadap performa algoritma. Pengurutan data kata membutuhkan waktu 2-4 kali lebih lama dibandingkan data angka untuk semua algoritma. Ini disebabkan oleh kompleksitas operasi perbandingan string yang lebih tinggi dan overhead memori yang lebih besar untuk manipulasi objek string. Dampak perbedaan ini semakin terasa pada dataset besar, dengan selisih waktu eksekusi mencapai 3-4 kali lipat.

2. Analisis Penggunaan Memori

Penggunaan RAM untuk algoritma sorting menunjukkan peningkatan linier seiring pertambahan ukuran data. Untuk dataset kecil hingga menengah (10.000-100.000 elemen), penggunaan RAM relatif serupa antar algoritma, berkisar 3.000-4.000 KB untuk data angka. Pada dataset besar, algoritma $O(n \log n)$ cenderung memerlukan RAM lebih besar karena kebutuhan ruang tambahan untuk operasi divide-and-conquer.

Perbedaan tipe data memberikan dampak dramatis pada penggunaan memori. Dataset kata memerlukan RAM 3-6 kali lebih besar dibandingkan dataset angka dengan jumlah elemen yang sama. Pada pengujian dengan 2.000.000 elemen, algoritma sorting membutuhkan sekitar 23.568 KB untuk data angka namun melonjak hingga 142.752-145.296 KB untuk data kata. Perbedaan ini disebabkan oleh karakteristik penyimpanan tipe data, di mana integer memiliki ukuran tetap dan kecil, sementara string membutuhkan alokasi dinamis dengan overhead metadata.

Penggunaan memori puncak menunjukkan pola yang relatif konsisten dengan penggunaan RAM reguler, mengindikasikan bahwa algoritma sorting tidak mengalami lonjakan memori signifikan selama eksekusi. Namun, algoritma seperti Merge Sort menunjukkan penggunaan memori puncak lebih tinggi karena kebutuhan array sementara dalam proses penggabungan.

3. Karakteristik Khusus Algoritma

Setiap algoritma sorting menunjukkan profil performa yang unik dalam berbagai konteks pengujian:

a. Quick Sort unggul di hampir semua skenario berkat strategi pivot yang efektif dan jumlah perbandingan yang lebih sedikit. Algoritma ini konsisten menjadi yang tercepat dan menawarkan keseimbangan optimal antara kecepatan dan penggunaan memori, meskipun tidak menjamin stabilitas pengurutan dan memiliki kasus terburuk $O(n^2)$.
b. Merge Sort mempunyai performa yang sangat stabil dan konsisten hampir seperti Quick Sort. Keunggulan utamanya adalah jaminan performa worst-case O(n log n) dan sifatnya yang stabil dalam mempertahankan urutan relatif elemen dengan nilai sama. Trade-off utamanya adalah kebutuhan memori tambahan O(n) untuk array sementara.
c. Shell Sort menunjukkan perilaku menarik dengan performa yang baik untuk data angka tetapi mengalami penurunan signifikan hingga 4-5 kali lebih lambat untuk data kata. Variabilitas performa ini membatasi kegunaan umumnya, terutama untuk aplikasi yang bekerja dengan berbagai jenis data.
d. Insertion Sort menunjukkan performa kompetitif untuk dataset kecil karena efisiensinya pada data yang hampir terurut dan implementasi yang sederhana. Algoritma ini seringkali menjadi pilihan yang baik untuk dataset kecil di mana overhead dari algoritma yang lebih kompleks tidak sepadan dengan peningkatan kecepatan.

4. Kesimpulan

Berdasarkan ukuran data, untuk dataset berukuran kecil (< 10.000 elemen), Insertion Sort menawarkan keseimbangan yang baik antara kesederhanaan implementasi dan performa. Algoritma ini bekerja efisien pada data yang hampir terurut dan memiliki overhead minimal. Namun, jika kecepatan adalah prioritas absolut, Quick Sort tetap menjadi pilihan optimal. Untuk dataset berukuran menengah (10.000-100.000 elemen), Quick Sort atau Merge Sort sangat direkomendasikan dikarenakan algoritma $O(n^2)$ sudah menunjukkan penurunan performa yang signifikan dan sebaiknya dihindari. Jika stabilitas pengurutan menjadi pertimbangan penting, Merge Sort lebih disukai dibandingkan Quick Sort. Untuk dataset berukuran besar (lebih dari 100.000 elemen), Quick Sort merupakan algoritma yang paling direkomendasikan untuk sebagian besar skenario. Performa superiornya pada dataset besar menjadikannya pilihan default yang solid. Merge Sort menjadi alternatif yang baik jika stabilitas pengurutan diperlukan, meskipun dengan sedikit trade-off performa.

Berdasarkan tipe data, untuk pengurutan data angka, Quick Sort, Merge Sort, dan Shell Sort semuanya menunjukkan performa yang sangat baik. Quick Sort tetap menjadi algoritma tercepat dalam pengujian dan direkomendasikan sebagai pilihan default dan untuk pengurutan data kata atau string, Quick Sort secara konsisten menunjukkan performa terbaik dengan margin yang signifikan. Merge Sort merupakan alternatif yang solid dengan performa yang stabil. Shell Sort sebaiknya dihindari untuk pengurutan data string karena penurunan performa yang drastis.

Berdasarkan kebutuhan aplikasi, Untuk aplikasi real-time yang membutuhkan responsivitas tinggi, Quick Sort adalah pilihan optimal dengan kemampuannya mengurutkan data dengan latensi minimal, untuk sistem dengan memori terbatas, Quick Sort dengan implementasi iteratif atau Insertion Sort (untuk data sangat kecil) dapat dipertimbangkan. Meskipun algoritma $O(n^2)$ memiliki kebutuhan memori yang sedikit lebih rendah, trade-off performa waktu hampir selalu membuat algoritma ini tidak praktis kecuali untuk dataset sangat kecil, Untuk database dan sistem file yang bekerja dengan dataset besar, Merge Sort lebih disukai karena kemampuannya untuk diparalelkan dan stabilitas yang penting untuk operasi database kompleks, dan untuk pengurutan data string dalam aplikasi pengolahan teks atau sistem indexing, Quick Sort jauh lebih cepat dan menjadi pilihan yang jelas.

Kesimpulan akhir algoritma Quick Sort menawarkan keseimbangan terbaik antara kecepatan dan penggunaan memori untuk sebagian besar skenario pengurutan data dan menjadi rekomendasi default

untuk implementasi algoritma sorting. Namun, pemahaman terhadap karakteristik dan trade-off masing-masing algoritma, serta konteks spesifik aplikasi, tetap penting untuk pemilihan algoritma yang optimal.