LogicLink: Journal of Artificial Intelligence and Multimedia in Informatics
E-ISSN: 3062-9098 | P-ISSN: 3063-4504 | Vol. 2 No. 1, June 2025, pp. 1 – 17

Analisis Komparatif Efisiensi Memori dan Waktu Komputasi pada 8 Algoritma *Sorting* menggunakan C++

Muhammad Irfan Ali*[1], Rangga Dzikri Fardiarsyah[2], Lukman Shodik[3], Fadilah Zahra Dwi Kinanti[4], Imam Prayogo Pujiono[5]
[1,2,3,4,5]Universitas Islam Negeri K.H. Abdurrahman Wahid Pekalongan, Pekalongan, Indonesia
*E-mail : muhammad.irfan.ali24058@mhs.uingusdur.ac.id**
**Penulis Korespondensi*

Received: 4 May 2025; Revised: 18 May 2025; Accepted: 28 May 2025

Abstrak - Penelitian ini bertujuan menganalisis efisiensi waktu komputasi dan alokasi memori delapan algoritma pengurutan (*Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Quick Sort*, *Merge Sort*, *Heap Sort*, *Counting Sort*, dan *Radix Sort*) yang diimplementasikan dalam bahasa pemrograman C++. Dataset uji terdiri dari tiga kategori ukuran: 100, 1.000, dan 10.000 elemen, dihasilkan secara acak dengan nilai antara 1 hingga 99. Rentang ini dipilih agar pengujian dilakukan dalam kondisi nilai yang terbatas dan mengandung banyak duplikasi, guna mendukung evaluasi efisiensi secara konsisten. Metode penelitian melibatkan pembuatan dataset menggunakan fungsi *random array*, pengukuran waktu eksekusi dalam satuan nanodetik, dan pemantauan penggunaan memori melalui metrik WorkingSetSize. Setiap algoritma diuji tiga kali pada setiap kategori data untuk memastikan konsistensi hasil. Hasil penelitian menunjukkan bahwa *Heap Sort* mencapai waktu eksekusi tercepat pada data kecil (101.266 nanodetik untuk 100 elemen), *Counting Sort* dan *Radix Sort* unggul pada data sedang, sedangkan *Counting Sort* memberikan performa terbaik pada data besar (1.483.166 nanodetik untuk 10.000 elemen). *Counting Sort* juga menunjukkan efisiensi memori yang stabil dibandingkan algoritma lain, sementara *Bubble Sort* konsisten menjadi algoritma dengan performa terendah di semua skala. Simpulan penelitian merekomendasikan *Heap Sort* untuk skala data kecil, *Counting Sort* dan *Radix Sort* untuk skala data sedang, serta *Counting Sort* untuk data besar. Temuan ini menjadi panduan praktis bagi pengembang dalam memilih algoritma sesuai skala data dan ketersediaan sumber daya.
Kata Kunci: Algoritma Pengurutan, Optimasi Memori, Analisis Waktu Eksekusi, C++, Dataset

Abstract - *This study aims to analyze the efficiency of computation time and memory allocation of eight sorting algorithms (Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Counting Sort, and Radix Sort) implemented in C++ programming language. The test dataset consists of three size categories: 100, 1,000, and 10,000 elements, randomly generated with values between 1 and 99. This range was chosen so that the tests are conducted under conditions of limited value and contain a lot of duplication, in order to support consistent efficiency evaluation. The research method involved generating datasets using the random array function, measuring execution time in nanoseconds, and monitoring memory usage through the WorkingSetSize metric. Each algorithm was tested three times on each category of data to ensure consistency of results. The results showed that Heap Sort achieved the fastest execution time on small data (101,266 nanoseconds for 100 elements), Counting Sort and Radix Sort excelled on medium data, while Counting Sort delivered the best performance on large data (1,483,166 nanoseconds for 10,000 elements). Counting Sort also demonstrated stable memory efficiency compared to the other algorithms, whereas Bubble Sort consistently exhibited the poorest performance across all scales. The research conclusion recommends Heap Sort for small-scale data, Counting Sort and Radix Sort for medium-scale data, and Counting Sort for large-*

scale data. The findings provide practical guidance for developers in selecting algorithms according to data scale and resource availability.
Keywords: *Sorting Algorithm, Memory Optimization, Execution Time Analysis, C++, Dataset*

1. PENDAHULUAN

Perkembangan teknologi informasi telah mengubah paradigma penyimpanan data, dari metode konvensional ke sistem digital berbasis komputasi awan (Pujiono et al., 2024). Namun, pertumbuhan volume data yang masif menuntut teknik pengurutan yang efisien, termasuk struktur himpunan-terpisah/union-find (Frühwirth, 2010) dan penerapan *Selection Sort* pada PHP (Sandria et al., 2022) untuk memudahkan pencarian dan analisis (Knuth, 1998; Syafnidawaty, 2020; Yagci & Mishra, 2016). Persoalan pengurutan merupakan persoalan yang sangat menarik perhatian para ilmuwan sains komputer, karena untuk persoalan yang sama terdapat puluhan algoritma pengurutan yang sudah ditemukan orang (Knuth, 1998; Munir & Lidya, 2016; Ullah, 2016). Pada dasarnya riset tentang algoritma pengurutan bertujuan untuk memperoleh algoritma pengurutan yang lebih mangkus (Arifin & Setiyadi, 2020; Poetra, 2022; Syafnidawaty, 2020).

Pada penelitian pertama dilakukan Arifin dan Setiyadi (2020), melakukan pengujian terhadap efisiensi algoritma *Bubble Sort* dan *Quick Sort*. Implementasi pengujian dilakukan menggunakan bahasa pemrograman C++. Hasil penelitian tersebut menunjukkan bahwa algoritma *Quick Sort* memiliki kinerja yang lebih optimal daripada *Bubble Sort*. Analisis menyatakan bahwa keunggulan *Quick Sort* disebabkan oleh konsumsi memori yang lebih rendah selama proses pengurutan.

Penelitian kedua dilakukan oleh Rahayuningsih (2016), yang menguji performa dari beberapa algoritma sorting yang mana terdapat algoritma *Bubble Sort* dan juga *Quick Sort*, penelitian ini diimplementasikan pada aplikasi desktop yang dibangun menggunakan bahasa pemrograman Visual Basic 6.0. Hasil dari penelitian ini menjelaskan secara rinci bahwa algoritma *Bubble Sort* membutuhkan waktu 6.84 detik untuk mengurutkan 250 data, sedangkan algoritma *Quick Sort* hanya membutuhkan waktu 0.11 detik untuk mengurutkan 250 data.

Penelitian ketiga dilakukan oleh Saputro dan Khasanah (2018) serta Pujiono et al. (2025) yang menguji performa dari algoritma *Bubble Sort* yang diimplementasikan pada Bahasa C++. Hasil dari penelitian ini menjelaskan bahwa algoritma *Bubble Sort* membutuhkan waktu 0.094 detik untuk mengurutkan 100 data.

Penelitian keempat selanjutnya dilakukan oleh Saptadi dan Sari (2012) serta Pratama et al. (2018), melakukan evaluasi terhadap kinerja beberapa algoritma pengurutan, termasuk *Quick Sort*. Proses pengujian diimplementasikan menggunakan bahasa pemrograman C++. Hasil penelitian tersebut membuktikan bahwa *Quick Sort* memerlukan waktu 0,03807 detik untuk mengurutkan 100 data.

Pada penelitian ini, dilakukan perbandingan efisiensi memori dan waktu komputasi antara 8 algoritma sorting, yaitu *Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Quick Sort*, *Merge Sort*, *Heap Sort*, *Counting Sort*, & *Radix Sort* menggunakan bahasa pemrograman C++.

2. METODE PENELITIAN

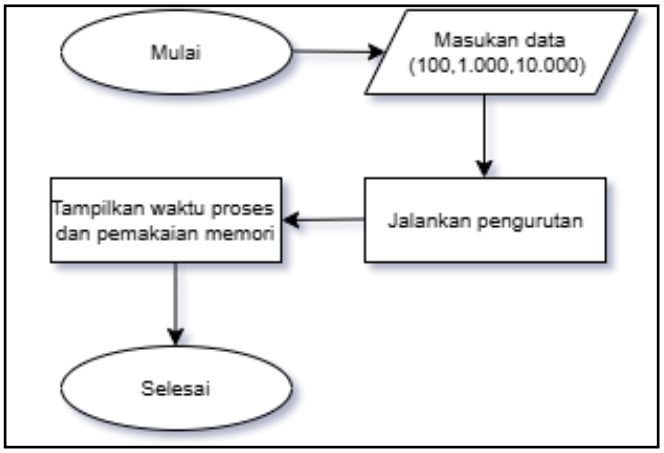
Pada penelitian ini, langkah awal dilakukan dengan membangun dataset dalam tiga kategori ukuran—data kecil (100 elemen), data sedang (1.000 elemen), dan data besar (10.000 elemen)—dengan memanfaatkan array yang diisi nilai acak antara 1 hingga 99. Rentang ini

dipilih untuk menyederhanakan distribusi data sekaligus menguji performa algoritma dalam kondisi nilai yang terbatas dan berpotensi mengandung banyak duplikasi, agar evaluasi efisiensi waktu dan penggunaan memori dapat dilakukan secara lebih konsisten. Proses pembuatan dataset dilakukan melalui fungsi C++ `generateRandomArray(n)` yang hanya dipanggil sekali di awal, sebelum pengujian dimulai; array hasilnya kemudian disalin ke masing-masing fungsi pengujian algoritma. Dengan demikian, waktu eksekusi dan penggunaan memori yang tercatat tidak termasuk proses pembuatan array, melainkan hanya mencerminkan performa pengurutan. Contoh kode yang digunakan adalah sebagai berikut:

```
//------------------------------------------------------------
// Bagian 1: Pengukuran dan Fungsi Pendukung
// ------------------------------------------------------------
SIZE_T getMemoryUsage() {
    PROCESS_MEMORY_COUNTERS pmc;
    if (GetProcessMemoryInfo(GetCurrentProcess(), &pmc, sizeof(pmc)))
{
        return pmc.WorkingSetSize;
    }
    return 0;
}
// Bersihkan working set memori sebelum pengujian
// Dipanggil SEBELUM timer dimulai, sehingga Sleep(100) tidak
mempengaruhi pengukuran waktu
void forceCleanMemory() {
    SetProcessWorkingSetSize(GetCurrentProcess(), (SIZE_T)-1,
(SIZE_T)-1);
    Sleep(100);
}
// Fungsi untuk membuat array acak berukuran n dengan nilai setiap
elemen antara 1 dan 99
// Dipanggil sekali di luar blok pengukuran agar tidak mempengaruhi
hasil
vector<int> generateRandomArray(int n) {
    vector<int> arr(n);
    for (int &val : arr) {
        val = rand() % 99 + 1;  // menghasilkan nilai acak dari 1
hingga 99
    }
    return arr;
}
```

Dataset yang dihasilkan berupa 10.000 elemen tersebut kemudian dibagi menjadi tiga variabel, yaitu ArrayKecil (100 elemen pertama), ArraySedang (1.000 elemen pertama), dan ArrayBesar (seluruh 10.000 elemen), sehingga setiap algoritma sorting diuji dengan data yang konsisten. Setiap pengujian pada masing-masing array dilakukan tiga kali dengan data input yang sama. Tujuannya adalah untuk menghitung nilai rata-rata waktu komputasi dan penggunaan memori sehingga meminimalkan efek fluktuasi hasil pada satu kali pengujian saja. Selain itu, fungsi `getMemoryUsage()` digunakan untuk mengukur penggunaan memori proses yang sedang berjalan dengan memanfaatkan Windows API melalui pemanggilan `GetProcessMemoryInfo()`, yang mengembalikan nilai WorkingSetSize sebagai indikator jumlah memori yang dipakai, Fungsi `forceCleanMemory()` digunakan untuk membersihkan cache memori sistem sebelum pengukuran waktu dimulai, guna memastikan hasil pengujian yang lebih akurat. Pemanggilan fungsi ini dilakukan sebelum pencatatan waktu komputasi `(start = high_resolution_clock::now())`, sehingga jeda waktu akibat `Sleep(100)` tidak memengaruhi hasil pengukuran.Selanjutnya, Setiap algoritma pengurutan

kemudian diimplementasikan menggunakan bahasa pemrograman C++. Proses pengujian dilakukan dengan memanfaatkan dataset yang telah ditetapkan, dengan hasil pengujian berupa data waktu komputasi dan penggunaan memori yang disusun dalam tabel untuk memudahkan analisis perbandingan kinerja algoritma.



Gambar 1. Visualisasi Alur Eksekusi pada Algoritma Pengurutan

3. HASIL DAN PEMBAHASAN

Penelitian ini dilakukan menggunakan bahasa pemrograman C++ dengan *Visual Studio Code* (VSCode) sebagai *Integrated Development Environment* (IDE) dan ekstensi Code Runner untuk menjalankan kode program. Compiler yang digunakan adalah MinGW 64-bit untuk memastikan kompatibilitas eksekusi dan dokumentasi hasil pengujian dilakukan menggunakan Snipping Tool untuk mengambil screenshot. Spesifikasi perangkat yang digunakan adalah:

1. Sistem Operasi: Windows 11 Pro (arsitektur 64-bit)
2. RAM: 8 GB
3. CPU: Intel Core i5-5300U (@2.30 GHz)
4. Penyimpanan: SSD 128 GB

Penelitian ini mengevaluasi kinerja delapan algoritma pengurutan, yakni *Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Quick Sort*, *Merge Sort*, *Heap Sort*, *Counting Sort*, dan *Radix Sort*. Semua algoritma tersebut diuji dengan menggunakan dataset yang telah distandarisasi dan dikelompokkan ke dalam tiga kategori ukuran: ArrayKecil terdiri dari 100 elemen, ArraySedang mencakup 1.000 elemen, dan ArrayBesar memuat 10.000 elemen. Data yang digunakan dihasilkan secara acak dengan nilai numerik antara 1 hingga 99 untuk menjaga objektivitas pengujian.

Dalam protokol pengujian, setiap algoritma dijalankan sebanyak tiga kali pada masing-masing kategori untuk memastikan konsistensi hasil. Secara rinci, percobaan untuk dataset 100 elemen dilakukan pada uji ke-1 hingga ke-3, untuk dataset 1.000 elemen pada uji ke-4 hingga ke-6, dan untuk dataset 10.000 elemen pada uji ke-7 hingga ke-9. Selama pengujian, hanya dua aplikasi yang aktif. Visual Studio Code untuk eksekusi kode dan *Snipping Tools* untuk dokumentasi visual melalui *screenshot* untuk meminimalkan gangguan pada CPU dan memori. Pengukuran kinerja dilakukan dengan mencatat waktu komputasi dalam satuan nanodetik dan memonitor penggunaan memori menggunakan metrik WorkingSetSize. Dataset lengkap beserta dokumentasi hasil pengujian masing-masing algoritma dapat diakses melalui tautan yang telah disediakan.

Dataset uji dapat diakses melalui tautan berikut: *Link Dataset Array (https://bit.ly/4j3THqF)*, *Bubble Sort(https://bit.ly/3SIR1UA)*, *Counting Sort(https://bit.ly/3Ff88dB)*, *Heap Sort(https://bit.ly/3Ff8UYb)*, *Insertion Sort(https://bit.ly/3H2j4vN)*, *Merge Sort(https://bit.ly/43uejTs)*, *Quick Sort(https://bit.ly/4mn8LTb)*, *Radix Sort(https://bit.ly/4mcPMuq)* dan *Selection Sort(https://bit.ly/3FmDT4t)*. Berikut adalah penjelasan dari pengujian tiap algoritma.

3.1 Bubble Sort

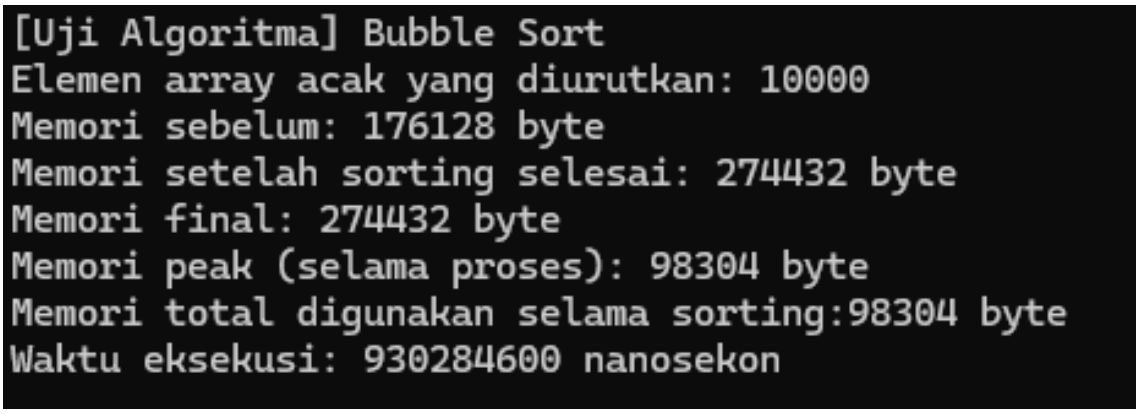
```
// Fungsi bubbleSort melakukan perbandingan antar-elemen secara
berulang
// dan menukar pasangan elemen yang keliru urutannya, hingga array
terurut
void bubbleSort(vector<int>& arr) {
    int n = (int)arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}
```

Pada kode tersebut, algoritma *Bubble Sort* diimplementasikan dalam bahasa C++. Seluruh data uji telah dimasukkan secara langsung ke dalam sistem dan dieksekusi berdasarkan skema pengujian yang telah dirancang sebelumnya. Tahap uji coba pertama hingga ketiga memanfaatkan 100 data (Dataset Kecil), pengujian keempat hingga keenam memanfaatkan 1.000 data (Dataset Sedang), sedangkan pengujian ketujuh hingga kesembilan melibatkan 10.000 data (Dataset Besar). Hasil pengukuran waktu komputasi dan konsumsi memori selama proses pengurutan dengan *Bubble Sort* dapat diamati secara lengkap pada Tabel 1.

Tabel 1. Hasil Uji *Bubble Sort*

Percobaan ke	Ukuran Data	Durasi Eksekusi (nanodetik)	Konsumsi Memori (byte)
1	100	261.800	57.344
2	100	268.900	57.344
3	100	394.100	57.344
4	1.000	18.924.200	61.440
5	1.000	18.770.900	61.440
6	1.000	24.997.200	61.440
7	10.000	930.284.600	98.304
8	10.000	937.016.700	98.304
9	10.000	926.018.800	98.304

Berdasarkan Tabel 1, waktu komputasi rata-rata untuk *Bubble Sort* pada dataset berukuran 100 data adalah 308.266 nanodetik. Nilai ini meningkat signifikan menjadi 20.897.433 nanodetik saat dataset diperbesar menjadi 1.000 data, dan melonjak hingga 931.106.700 nanodetik untuk 10.000 data. Sementara itu, penggunaan memori menunjukkan pola yang stabil: 57.344 byte untuk 100 data, 61.440 byte untuk 1.000 data, 98.340 byte untuk 10.000 data



Gambar 2. Keluaran Uji ke-7 (10.000 data) untuk Algoritma *Bubble Sort*

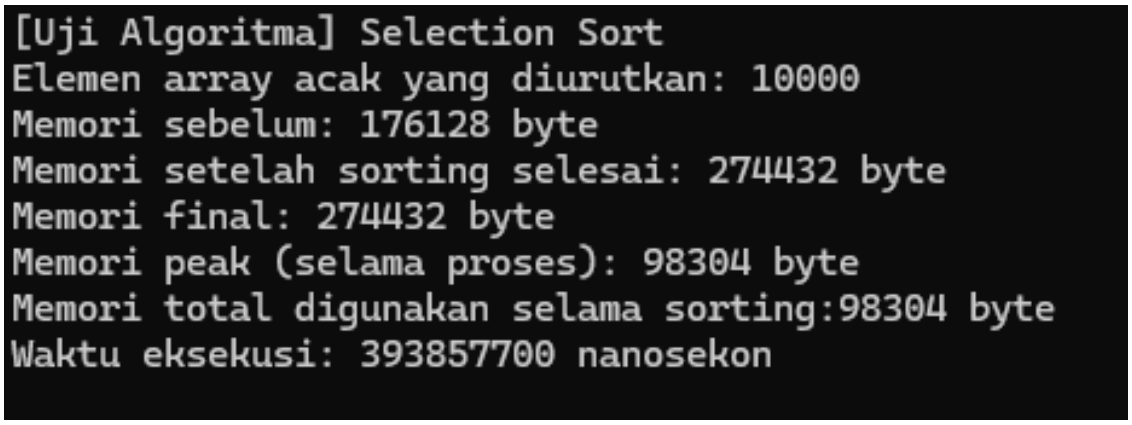
3.2 Selection Sort

```
// Fungsi selectionSort mencari elemen terkecil dan menempatkannya di
posisi awal,
// dilanjutkan ke posisi berikutnya hingga array terurut
void selectionSort(vector<int>& arr) {
    int n = (int)arr.size();
    for (int i = 0; i < n - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        swap(arr[i], arr[minIdx]);
    }
}
```

Pada kode tersebut, algoritma *Selection Sort* diimplementasikan dalam bahasa C++. Seluruh data uji telah dimasukkan secara langsung ke dalam sistem dan dieksekusi berdasarkan skema pengujian yang telah dirancang sebelumnya. Tahap uji coba pertama hingga ketiga memanfaatkan 100 data (Dataset Kecil), pengujian keempat hingga keenam memanfaatkan 1.000 data (Dataset Sedang), sedangkan pengujian ketujuh hingga kesembilan melibatkan 10.000 data (Dataset Besar). Hasil pengukuran waktu komputasi dan konsumsi memori selama proses pengurutan dengan *Selection Sort* dapat diamati secara lengkap pada Tabel 2.

Tabel 2. Hasil Uji *Selection Sort*

Percobaan ke	Ukuran Data	Durasi Eksekusi (nanodetik)	Konsumsi Memori (byte)
1	100	154.200	57.344
2	100	156.500	57.344
3	100	224.600	57.344
4	1.000	3.889.700	61.440
5	1.000	11.081.600	61.440
6	1.000	11.071.000	61.440
7	10.000	393.857.700	98.304
8	10.000	385.037.300	98.304
9	10.000	384.939.900	98.304



Gambar 3. Keluaran Uji ke-7 (10.000 data) untuk Algoritma *Selection Sort*

Berdasarkan Tabel 2, waktu komputasi rata-rata untuk *Selection Sort* pada dataset berukuran 100 data adalah 178.433 nanodetik. Nilai ini meningkat signifikan menjadi 8.680.766 nanodetik saat dataset diperbesar menjadi 1.000 data, dan melonjak hingga 387.944.966 nanodetik untuk 10.000 data. Sementara itu, penggunaan memori menunjukkan pola yang stabil: 57.344 byte untuk 100 data, 61.440 byte untuk 1.000 data, 98.304 byte untuk 10.000 data.

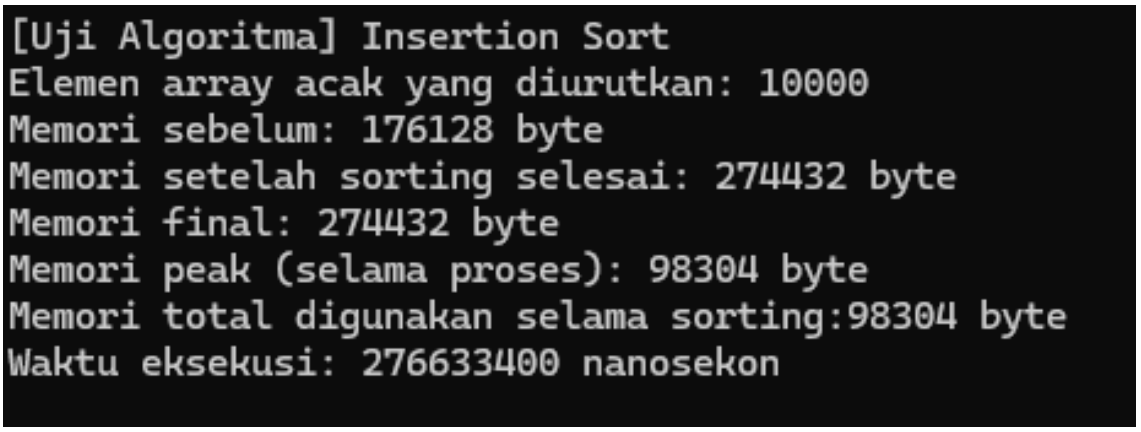
3.3 Insertion Sort

```
// Fungsi insertionSort menyisipkan elemen satu per satu pada posisi
yang tepat
// dengan cara menggeser elemen yang lebih besar ke kanan
void insertionSort(vector<int>& arr) {
    int n = (int)arr.size();
    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;
    }
}
```

Pada kode tersebut, algoritma *Insertion Sort* diimplementasikan dalam bahasa C++. Seluruh data uji telah dimasukkan secara langsung ke dalam sistem dan dieksekusi berdasarkan skema pengujian yang telah dirancang sebelumnya. Tahap uji coba pertama hingga ketiga memanfaatkan 100 data (Dataset Kecil), pengujian keempat hingga keenam memanfaatkan 1.000 data (Dataset Sedang), sedangkan pengujian ketujuh hingga kesembilan melibatkan 10.000 data (Dataset Besar). Hasil pengukuran waktu komputasi dan konsumsi memori selama proses pengurutan dengan *Insertion Sort* dapat diamati secara lengkap pada Tabel 3.

Tabel 3. Hasil Uji *Insertion Sort*

Percobaan ke	Ukuran Data	Durasi Eksekusi (nanodetik)	Konsumsi Memori (byte)
1	100	176.100	57.344
2	100	160.600	57.344
3	100	104.300	57.344
4	1.000	7.674.300	61.440
5	1.000	7.530.700	61.440
6	1.000	7.743.600	61.440
7	10.000	276.633.400	98.304
8	10.000	275.695.900	98.304
9	10.000	256.874.300	98.304



Gambar 4. Keluaran Uji ke-7 (10.000 data) untuk Algoritma *Insertion Sort*

Berdasarkan Tabel 3, waktu komputasi rata-rata untuk *Insertion Sort* pada dataset berukuran 100 data adalah 147.000 nanodetik. Nilai ini meningkat signifikan menjadi 7.649.533 nanodetik saat dataset diperbesar menjadi 1.000 data, dan melonjak hingga 269.734.533 nanodetik untuk 10.000 data. Sementara itu, penggunaan memori menunjukkan pola yang stabil: 57.344 byte untuk 100 data, 61.440 byte untuk 1.000 data, 98.304 byte untuk 10.000 data.

3.4 Quick Sort

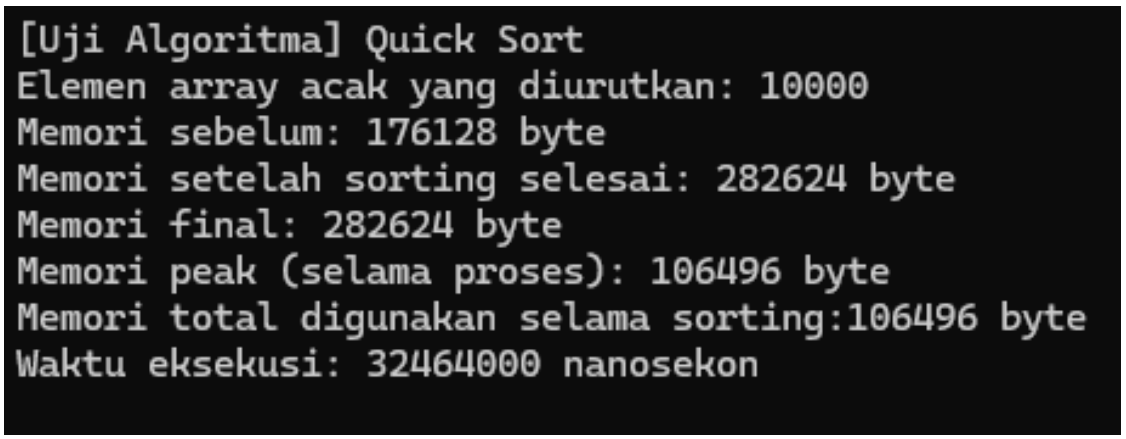
```
// Fungsi partitionFunc menempatkan pivot di posisi yang benar di
dalam array
// dengan cara memindah elemen yang lebih kecil pivot ke kiri, dan
yang lebih besar ke kanan
int partitionFunc(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}
// Fungsi quickSortImpl mengurutkan array melalui partitionFunc,
```

```
// lalu memanggil dirinya sendiri (rekursif) untuk sub-array kiri &
kanan pivot
void quickSortImpl(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partitionFunc(arr, low, high);
        quickSortImpl(arr, low, pi - 1);
        quickSortImpl(arr, pi + 1, high);
    }
}
// Fungsi quickSort menyiapkan pemanggilan quickSortImpl
void quickSort(vector<int>& arr) {
    if (!arr.empty()) {
        quickSortImpl(arr, 0, (int)arr.size() - 1);
    }
}
```

Pada kode tersebut, algoritma *Quick Sort* diimplementasikan dalam bahasa C++. Seluruh data uji telah dimasukkan secara langsung ke dalam sistem dan dieksekusi berdasarkan skema pengujian yang telah dirancang sebelumnya. Tahap uji coba pertama hingga ketiga memanfaatkan 100 data (Dataset Kecil), pengujian keempat hingga keenam memanfaatkan 1.000 data (Dataset Sedang), sedangkan pengujian ketujuh hingga kesembilan melibatkan 10.000 data (Dataset Besar). Hasil pengukuran waktu komputasi dan konsumsi memori selama proses pengurutan dengan *Quick Sort* dapat diamati secara lengkap pada Tabel 4.

Tabel 4. Hasil Uji *Quick Sort*

Percobaan ke	Ukuran Data	Durasi Eksekusi (nanodetik)	Konsumsi Memori (byte)
1	100	105.200	57.344
2	100	107.400	57.344
3	100	91.200	57.344
4	1.000	883.800	65.536
5	1.000	18.770.900	61.440
6	1.000	24.997.200	61.440
7	10.000	930.284.600	98.304
8	10.000	937.016.700	98.304
9	10.000	926.018.800	98.304



Gambar 5. Keluaran Uji ke-7 (10.000 data) untuk Algoritma *Quick Sort*

Berdasarkan Tabel 4, waktu komputasi rata-rata untuk *Quick Sort* pada dataset berukuran 100 data adalah 101.266 nanodetik. Nilai ini meningkat signifikan menjadi 14.883.966 nanodetik saat dataset diperbesar menjadi 1.000 data, dan melonjak hingga 931.106.700 nanodetik untuk 10.000 data. Sementara itu, penggunaan memori 57.344 byte untuk 100 data, 62.805 byte untuk 1.000 data, 98.304 byte untuk 10.000 data

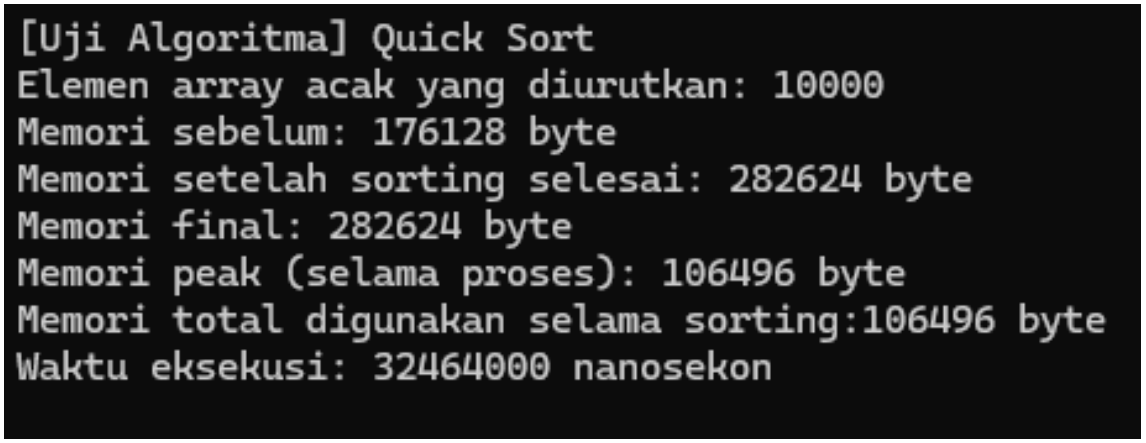
3.5 Merge Sort

```
// Fungsi merge menggabungkan dua sub-array terurut menjadi satu
secara berurutan
void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
// Fungsi mergeSortImpl membagi array menjadi sub-array lebih kecil
(rekursif),
// lalu memanggil merge untuk menggabungkannya kembali
void mergeSortImpl(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSortImpl(arr, left, mid);
        mergeSortImpl(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
// Fungsi mergeSort memanggil mergeSortImpl jika array tidak kosong
void mergeSort(vector<int>& arr) {
    if (!arr.empty()) {
        mergeSortImpl(arr, 0, (int)arr.size() - 1);
    }
}
```

Pada kode tersebut, algoritma *Merge Sort* diimplementasikan dalam bahasa C++. Seluruh data uji telah dimasukkan secara langsung ke dalam sistem dan dieksekusi berdasarkan skema pengujian yang telah dirancang sebelumnya. Tahap uji coba pertama hingga ketiga memanfaatkan 100 data (Dataset Kecil), pengujian keempat hingga keenam memanfaatkan 1.000 data (Dataset Sedang), sedangkan pengujian ketujuh hingga kesembilan melibatkan 10.000 data (Dataset Besar). Hasil pengukuran waktu komputasi dan konsumsi memori selama proses pengurutan dengan *Merge Sort* dapat diamati secara lengkap pada Tabel 5.

Tabel 5. Hasil Uji *Merge Sort*

Percobaan ke	Ukuran Data	Durasi Eksekusi (nanodetik)	Konsumsi Memori (byte)
1	100	105.200	57.344
2	100	107.400	57.344
3	100	91.200	57.344
4	1.000	883.800	65.536
5	1.000	18.770.900	61.440
6	1.000	24.997.200	61.440
7	10.000	930.284.600	98.304
8	10.000	937.016.700	98.304
9	10.000	926.018.800	98.304



Gambar 6. Keluaran Uji ke-7 (10.000 data) untuk Algoritma *Merge Sort*

Berdasarkan Tabel 5, waktu komputasi rata-rata untuk *Merge Sort* pada dataset berukuran 100 data adalah 530.100 nanodetik. Nilai ini meningkat signifikan menjadi 2.063.933 nanodetik saat dataset diperbesar menjadi 1.000 data, dan melonjak hingga 23.271.666 nanodetik untuk 10.000 data. Sementara itu, penggunaan memori menunjukan pola peningkatan bertahap sesuai skala dataset 203.434 byte untuk 100 data, 227.677 byte untuk 1.000 data, 319.488 byte untuk 10.000 data.

3.6 Heap Sort

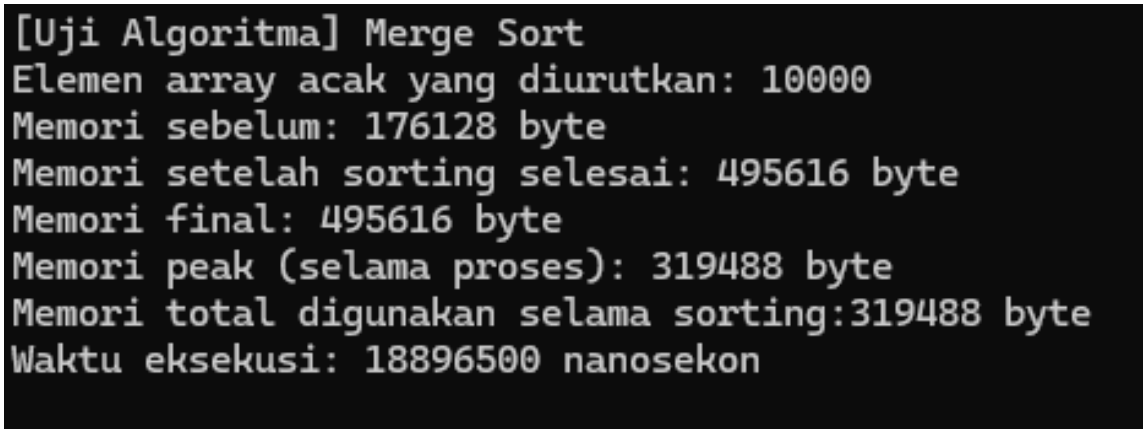
```
// Fungsi heapify membangun max-heap di sekitar indeks i
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest]) largest = left;
    if (right < n && arr[right] > arr[largest]) largest = right;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
// Fungsi heapSort membangun max-heap dari array, lalu menukar elemen
teratas
// dengan elemen akhir secara bertahap hingga array terurut
void heapSort(vector<int>& arr) {
    int n = (int)arr.size();
```

```
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

Pada kode tersebut, algoritma *Heap Sort* diimplementasikan dalam bahasa C++. Seluruh data uji telah dimasukkan secara langsung ke dalam sistem dan dieksekusi berdasarkan skema pengujian yang telah dirancang sebelumnya. Tahap uji coba pertama hingga ketiga memanfaatkan 100 data (Dataset Kecil), pengujian keempat hingga keenam memanfaatkan 1.000 data (Dataset Sedang), sedangkan pengujian ketujuh hingga kesembilan melibatkan 10.000 data (Dataset Besar). Hasil pengukuran waktu komputasi dan konsumsi memori selama proses pengurutan dengan *Heap Sort* dapat diamati secara lengkap pada Tabel 6.

Tabel 6. Hasil Uji *Heap Sort*

Percobaan ke	Ukuran Data	Durasi Eksekusi (nanodetik)	Konsumsi Memori (byte)
1	100	522.900	204.800
2	100	536.900	200.704
3	100	530.500	204.800
4	1.000	2.575.500	229.376
5	1.000	2.679.500	229.376
6	1.000	936.800	225.280
7	10.000	18.896.500	319.488
8	10.000	25.787.800	319.488
9	10.000	25.130.700	319.488



Gambar 7. Keluaran Uji ke-7 (10.000 data) untuk Algoritma *Heap Sort*

Berdasarkan Tabel 6, waktu komputasi rata-rata untuk *Heap Sort* pada dataset berukuran 100 data adalah 147.000 nanodetik. Nilai ini meningkat signifikan menjadi 7.649.533 nanodetik saat dataset diperbesar menjadi 1.000 data, dan melonjak hingga 269.734.533 nanodetik untuk 10.000 data. Sementara itu, penggunaan memori menunjukkan pola yang stabil: 57.344 byte untuk 100 data, 61.440 byte untuk 1.000 data, 98.304 byte untuk 10.000 data.

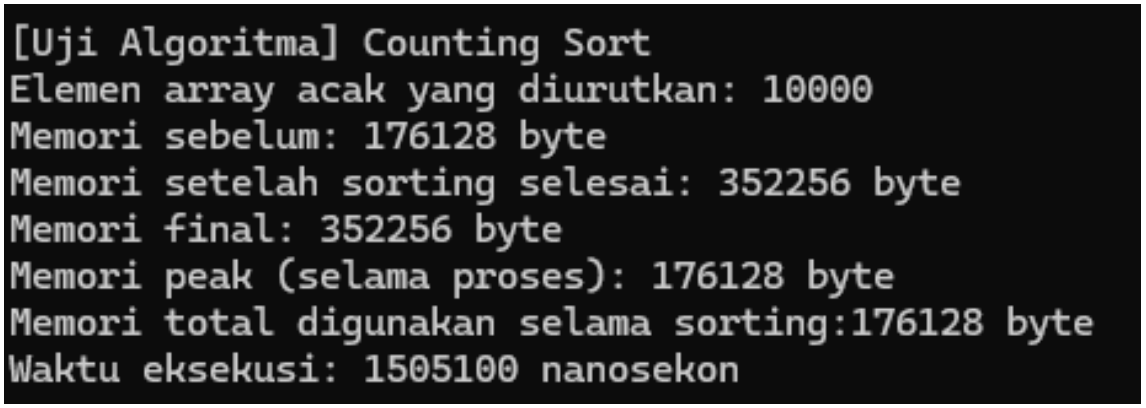
3.7 Counting Sort

```
// Fungsi countingSort menghitung frekuensi setiap nilai di array,
// lalu menyusun ulang elemen ke array terurut
void countingSort(vector<int>& arr) {
    if (arr.empty()) return;
    int maxVal = *max_element(arr.begin(), arr.end());
    vector<int> count(maxVal + 1, 0);
    for (int num : arr) {
        count[num]++;
    }
    int idx = 0;
    for (int i = 0; i <= maxVal; i++) {
        while (count[i]-- > 0) {
            arr[idx++] = i;
        }
    }
}
```

Pada kode tersebut, algoritma *Counting Sort* diimplementasikan dalam bahasa C++. Seluruh data uji telah dimasukkan secara langsung ke dalam sistem dan dieksekusi berdasarkan skema pengujian yang telah dirancang sebelumnya. Tahap uji coba pertama hingga ketiga memanfaatkan 100 data (Dataset Kecil), pengujian keempat hingga keenam memanfaatkan 1.000 data (Dataset Sedang), sedangkan pengujian ketujuh hingga kesembilan melibatkan 10.000 data (Dataset Besar). Hasil pengukuran waktu komputasi dan konsumsi memori selama proses pengurutan dengan *Counting Sort* dapat diamati secara lengkap pada Tabel 7.

Tabel 7. Hasil Uji *Counting Sort*

Percobaan ke	Ukuran Data	Durasi Eksekusi (nanodetik)	Konsumsi Memori (byte)
1	100	284.500	131.072
2	100	161.100	131.072
3	100	66.600	135.168
4	1.000	241.600	135.168
5	1.000	277.900	135.168
6	1.000	267.500	131.072
7	10.000	1.505.100	176.128
8	10.000	1.457.700	176.128
9	10.000	1.486.700	176.128



Gambar 8. Keluaran Uji ke-7 (10.000 data) untuk Algoritma *Counting Sort*

Berdasarkan Tabel 7, waktu komputasi rata-rata untuk *Counting Sort* pada dataset berukuran 100 data adalah 170.733 nanodetik. Nilai ini meningkat signifikan menjadi 262.333 nanodetik saat dataset diperbesar menjadi 1.000 data, dan melonjak hingga 1.483.166 nanodetik untuk 10.000 data. Sementara itu, penggunaan memori menunjukan pola peningkatan bertahap sesuai skala dataset 132.437 byte untuk 100 data, 133.802 byte untuk 1.000 data, 176.128 byte untuk 10.000 data.

3.8 Radix Sort

```
// Fungsi radixSort mengurutkan angka berdasarkan digit dari satuan,
puluhan, ratusan, dst.
// Menggunakan countingSort sebagai metode utama pada tiap digit
void radixSort(vector<int>& arr) {
    if (arr.empty()) return;
    int maxVal = *max_element(arr.begin(), arr.end());
    for (int exp = 1; maxVal / exp > 0; exp *= 10) {
        vector<int> output(arr.size());
        vector<int> count(10, 0);

        for (int num : arr) count[(num / exp) % 10]++;
        for (int i = 1; i < 10; i++) count[i] += count[i - 1];

        for (int i = (int)arr.size() - 1; i >= 0; i--) {
            int digit = (arr[i] / exp) % 10;
            output[--count[digit]] = arr[i];
        }
        arr = output;
    }
}
```

Pada kode tersebut, algoritma *Radix Sort* diimplementasikan dalam bahasa C++. Seluruh data uji telah dimasukkan secara langsung ke dalam sistem dan dieksekusi berdasarkan skema pengujian yang telah dirancang sebelumnya. Tahap uji coba pertama hingga ketiga memanfaatkan 100 data (Dataset Kecil), pengujian keempat hingga keenam memanfaatkan 1.000 data (Dataset Sedang), sedangkan pengujian ketujuh hingga kesembilan melibatkan 10.000 data (Dataset Besar). Hasil pengukuran waktu komputasi dan konsumsi memori selama proses pengurutan dengan *Radix Sort* dapat diamati secara lengkap pada Tabel 8.

Tabel 8. Hasil Uji *Counting Sort*

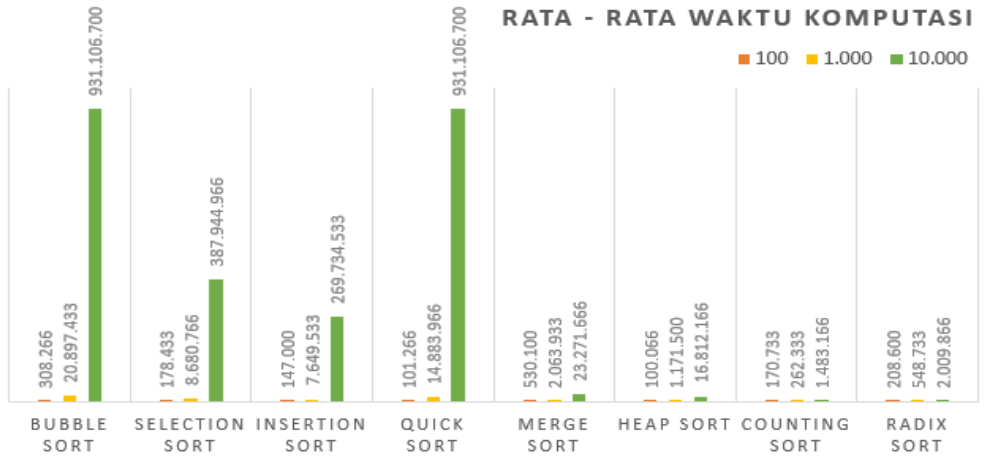
Percobaan ke	Ukuran Data	Durasi Eksekusi (nanodetik)	Konsumsi Memori (byte)
1	100	105.200	57.344
2	100	107.400	57.344
3	100	91.200	57.344
4	1.000	883.800	65.536
5	1.000	18.770.900	61.440
6	1.000	24.997.200	61.440
7	10.000	930.284.600	98.304
8	10.000	937.016.700	98.304
9	10.000	926.018.800	98.304

```
[Uji Algoritma] Radix Sort
Elemen array acak yang diurutkan: 10000
Memori sebelum: 176128 byte
Memori setelah sorting selesai: 372736 byte
Memori final: 372736 byte
Memori peak (selama proses): 196608 byte
Memori total digunakan selama sorting:196608 byte
Waktu eksekusi: 3634900 nanosekon
```

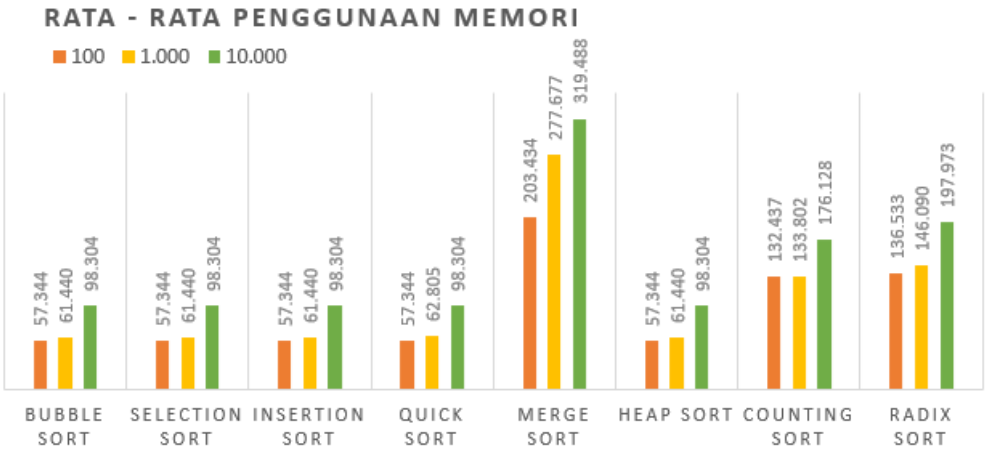
Gambar 9. Keluaran Uji ke-7 (10.000 data) untuk Algoritma *Radix Sort*

Berdasarkan Tabel 8, waktu komputasi rata-rata untuk *Counting Sort* pada dataset berukuran 100 data adalah 208.600 nanodetik. Nilai ini meningkat signifikan menjadi 548.733 nanodetik saat dataset diperbesar menjadi 1.000 data, dan melonjak hingga 2.099.867 nanodetik untuk 10.000 data. Sementara itu, penggunaan memori menunjukan pola peningkatan bertahap sesuai skala dataset 136.533 byte untuk 100 data, 146.091 byte untuk 1.000 data, 197.973 byte untuk 10.000 data.

Berdasarkan serangkaian pengujian algoritma yang telah dilaksanakan, rekapitulasi rata-rata hasil untuk setiap algoritma dapat dilihat pada Gambar 9 dan Gambar 10 di bawah ini.



Gambar 10. Rata – Rata Waktu Komputasi pada 8 Algoritma Pengurutan



Gambar 11. Rata – Rata Penggunaan Memori pada 8 Algoritma Pengurutan

Berdasarkan data yang tercantum dalam Gambar 10 dan Gambar 11, yang menunjukkan nilai rata-rata pengujian delapan metode pengurutan pada tiga tingkatan ukuran data (100, 1.000, dan 10.000), dapat disimpulkan bahwa:

1. Pada data 100, sebagian besar algoritma (*Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Quick Sort*, *Heap Sort*) menggunakan memori serupa sebesar 57.344 byte, sementara *Merge Sort* menjadi yang paling boros memori dengan 203.434 byte. Dalam hal waktu komputasi, *Quick Sort* tercepat dengan 101.266 nanodetik, sedangkan *Merge Sort* paling lambat 530.100 nanodetik.
2. Saat data meningkat menjadi 1.000, algoritma seperti *Bubble Sort*, *Selection Sort*, *Insertion Sort*, dan *Heap Sort* menggunakan memori 61.440 byte, sementara *Merge Sort* kembali membutuhkan memori tertinggi 277.677 byte. *Counting Sort* mencatat waktu tercepat 262.333 nanodetik, dan *Bubble Sort* tetap paling lambat 20.897.433 nanodetik.
3. Pada data 10.000, *Merge Sort* kembali menjadi yang paling boros memori 319.488 byte, sementara algoritma lainnya (*Bubble Sort*, *Selection Sort*, *Insertion Sort*, *Quick Sort*, *Heap Sort*) menggunakan 98.304 byte. *Counting Sort* menunjukkan efisiensi waktu terbaik 1.483.166 nanodetik, sedangkan *Bubble Sort* dan *Quick Sort* menjadi yang terlambat dengan 931.106.700 nanodetik.

Secara keseluruhan, *Merge Sort* konsisten membutuhkan memori tertinggi di semua kategori, sementara *Bubble Sort* selalu menjadi algoritma paling lambat, terutama pada data besar. *Counting Sort* menonjol sebagai algoritma paling efisien waktu pada skala data 10.000.

4. KESIMPULAN

Berdasarkan hasil pengujian terhadap delapan algoritma pengurutan pada tiga skala dataset, diperoleh bahwa *Heap Sort* menunjukkan waktu eksekusi tercepat pada data kecil, sedangkan *Counting Sort* dan *Radix Sort* unggul pada data sedang. Untuk data besar, *Counting Sort* memberikan performa terbaik dalam hal efisiensi waktu. Di sisi lain, *Bubble Sort* konsisten menjadi algoritma dengan performa terendah di semua skala. *Counting Sort* juga menunjukkan efisiensi memori yang stabil, menjadikannya algoritma paling optimal secara keseluruhan untuk berbagai ukuran data. Temuan ini dapat dijadikan rujukan dalam pemilihan algoritma pengurutan yang sesuai dengan skala data dan kebutuhan efisiensi komputasi.

DAFTAR PUSTAKA

Arifin, R. W., & Setiyadi, D. (2020). Algoritma Metode Pengurutan Bubble Short dan Quick Sort Dalam Bahasa Pemprogaman c++. *Information System For Educators and Profesionals*, *2*(No.4), 178–187.

Frühwirth, T. (2010). Union-find algorithm. In *Constraint Handling Rules*. https://doi.org/10.1017/cbo9780511609886.014

Knuth, D. E. (1998). *Art of Computer Programming - Volume 3 (Sorting & Searching)* (Second Edi). Addison-Wesley Professional.

Munir, R., & Lidya, L. (2016). *Algoritma dan Pemrograman dalam Bahasa Pascal, C, Dan C++ Edisi Keenam*. Informatika Bandung.

Poetra, D. R. (2022). Performa Algoritma Bubble Sort dan Quick Sort pada Framework Flutter dan Dart SDK(Studi Kasus Aplikasi E-Commerce). *JATISI (Jurnal Teknik Informatika Dan Sistem Informasi)*, *9*(2), 806–816. https://doi.org/10.35957/jatisi.v9i2.1886

Pratama, A., Desiani, A., & Irmeilyana, I. (2018). Analisis Kebutuhan Waktu Algoritma INsertion Sort, Merge Sort, dab Quick Sort dengan Kompleksitas Waktu. *In Computer Science and ICT*, *Vol 2*(No. 1), 95-1–6.

Pujiono, I. P., Rachmawanto, E. H., Anisa, N., & Winarsih, S. (2025). *Array Sorting Algorithm vs Algoritma Pengurutan Tradisional : Analisis Efisiensi Memori dan Waktu Array Sorting Algorithm vs Traditional Sorting Algorithm : Memory and Time Efficiency Analysis*. *15*(April), 47–59.

Pujiono, I. P., Trianto, R. B., & Hana, F. M. (2024). Perbandingan Efisiensi Memori dan Waktu Komputasi Pada 7 Algoritma Sorting Menggunakan Bahasa Pemrograman Java. *Simkom*, *9*(2), 218–230. https://doi.org/10.51717/simkom.v9i2.481

Rahayuningsih, P. (2016). Analisis Perbandingan Kompleksitas Algoritma Pengurangan NIlai (Sorting). *Jurnal Evolusi*, *No. 4*, 64–75.

Sandria, Y. A., Nurhayoto, M. R. A., Ramadhani, L., Harefa, R. S., & Syahputra, A. (2022). Penerapan Algoritma Selection Sort untuk Melakukan Pengurutan Data dalam Bahasa Pemrograman PHP. *Hello World Jurnal Ilmu Komputer*, *1*(4), 190–194. https://doi.org/10.56211/helloworld.v1i4.187

Saptadi, A. H., & Sari, D. W. (2012). Analisis Algoritma Insertion Sort, Merge Sort Dan Implementasinya Dalam Bahasa Pemrograman C++. *JURNAL INFOTEL - Informatika Telekomunikasi Elektronika*, *4*(2), 10. https://doi.org/10.20895/infotel.v4i2.103

Saputro, F. E., & Khasanah, F. N. (2018). Teknik Selection Sort dan Bubble Sort Menggunakan Borland C++. *Jurnal Mahasiswa Bina Insani*, *Vol. 2*(No. 2), 136–145.

Syafnidawaty. (2020). Pengertian Logika Fuzzy. *06 April 2020*, *II*(September), 3.

Ullah, Z. (2016). Understanding Sorting Techniques Using C++. *International Journal of Novel Research in Computer Science and Software Engineering*, *3*(1), 171–199.

Yagci, Y., & Mishra, M. K. (2016). Data and structures. In *Handbook of Vinyl Polymers: Radical Polymerization, Process, and Technology: Second Edition*. https://doi.org/10.1201/9781420015133.pt5