JITET (Jurnal Informatika dan Teknik Elektro Terapan)
Vol. 13 No. 3, pISSN: 2303-0577 eISSN: 2830-7062 http://dx.doi.org/10.23960/jitet.v13i3.6657

ALGORITMA COUNTING SORT VS ALGORITMA PENGURUTAN MODERN: ANALISIS EFISIENSI MEMORI DAN WAKTU KOMPUTASI

Imam Prayogo Pujiono[1*], Muhammad Rikzam Kamal[2], Arditya Prayogi[3], Christy Atika Sari[4], Rohmatulloh Muhamad Ikhsanuddin[5]

[1,2,3]Program Studi Informatika, Universitas Islam Negeri K.H. Abdurrahman Wahid Pekalongan; Jl. Pahlawan Km.5 Rowolaku Kajen Kab. Pekalongan 51161; (0285) 412575
[4]Program Studi Teknik Informatika, Universitas Dian Nuswantoro; Jl. Imam Bonjol No. 207, Pendrikan Kidul, Kota Semarang 50131; (024) 3517261
[5]Program Studi Sains Data, Universitas Putra Bangsa; Jl. Ronggowarsito No.18, Sudagaran, Kedawung, Kabupaten Kebumen 54361; 087732746000

Keywords:
Sorting Algorithm, Counting Sort, Memory Efficiency, Computation Time, Performance Analysis.

Corespondent Email:
imam.prayogopujiono@uingusdur.ac.id

Abstrak. Penelitian ini bertujuan untuk menganalisis efisiensi memori dan waktu komputasi algoritma Counting Sort dibandingkan dengan algoritma pengurutan modern seperti Heap Sort, Quick Sort, Merge Sort, dan Shell Sort. Fokus penelitian adalah pada dataset numerik acak dengan rentang terbatas, yang relevan untuk aplikasi praktis di bidang informatika. Penelitian ini menggunakan pendekatan eksperimental, dengan dataset berukuran 100, 1.000, dan 10.000 elemen yang dihasilkan dalam rentang 1 hingga 99, dan diimplementasikan dalam bahasa pemrograman Java untuk pengujian performa. Berdasarkan hasil eksperimen, Counting Sort mencatat waktu komputasi yang jauh lebih rendah, terutama pada dataset besar (10.000 elemen), di mana performanya hampir 6-10 kali lebih cepat dibandingkan algoritma lainnya. Namun, dalam hal efisiensi memori, Counting Sort memerlukan penggunaan memori yang lebih tinggi pada dataset kecil (100 elemen) dan sedang (1.000 elemen) dibandingkan algoritma *in-place* seperti Heap Sort dan Quick Sort. Pada dataset besar, penggunaan memorinya tetap kompetitif, bahkan lebih hemat dibandingkan Merge Sort. Penelitian ini menyimpulkan bahwa Counting Sort merupakan pilihan optimal untuk mengurutkan dataset numerik dengan rentang terbatas, terutama dalam aplikasi yang menuntut pengolahan data cepat dan hemat sumber daya, seperti sistem tertanam atau IoT. Temuan ini memberikan kontribusi pada pemilihan algoritma pengurutan yang lebih tepat berdasarkan karakteristik dataset.

Abstract. *This research aims to analyse the memory efficiency and computation time of the Counting Sort algorithm compared to modern sorting algorithms such as Heap Sort, Quick Sort, Merge Sort, and Shell Sort. The research focuses on random numerical datasets with a limited range, which are relevant for practical applications in informatics. This research uses an experimental approach, with datasets of 100, 1,000, and 10,000 elements generated in the range of 1 to 99, and implemented in Java programming language for performance testing. Based on the experimental results, Counting Sort recorded significantly lower computation time, especially on large datasets (10,000 elements), where it performed almost 6-10 times faster than other algorithms. However, regarding memory efficiency, Counting Sort requires higher memory usage on small (100 elements) and medium (1,000 elements) datasets than in-place algorithms such as Heap Sort and Quick Sort.*

Its memory usage remains competitive on large datasets, even more efficient than Merge Sort. This research concludes that Counting Sort is optimal for sorting numerical datasets with limited ranges, especially in applications that demand fast and resource-efficient data processing, such as embedded systems or IoT. The findings contribute to selecting a more appropriate sorting algorithm based on the dataset's characteristics.

1. PENDAHULUAN

Perkembangan teknologi informasi yang pesat telah mendorong peningkatan signifikan dalam volume data yang dihasilkan dan diproses setiap hari. Dalam konteks ini, efisiensi algoritma pengurutan menjadi krusial, terutama dalam aplikasi yang menuntut pengolahan data secara cepat dan hemat sumber daya [1][2]. Algoritma pengurutan tidak hanya berperan dalam menyusun data secara teratur, tetapi juga menjadi fondasi bagi banyak operasi komputasi lanjutan, seperti pencarian, pengindeksan, dan analisis data [3][4]. Meskipun algoritma pengurutan modern seperti Heap Sort, Quick Sort, Merge Sort, dan Shell Sort telah lama dikenal dan digunakan secara luas karena efisiensinya dalam berbagai skenario [5][6], terdapat algoritma lain yang sering diabaikan namun berpotensi unggul dalam kondisi tertentu, yaitu Counting Sort. Penelitian ini bertujuan untuk menganalisis efisiensi memori dan waktu komputasi dari algoritma Counting Sort dibandingkan dengan algoritma pengurutan modern tersebut, dengan fokus pada dataset numerik acak dengan rentang nilai antara 1 hingga 99 serta dengan jumlah data (dataset) berukuran 100, 1.000, dan 10.000 elemen.

Secara umum, algoritma pengurutan dapat diklasifikasikan ke dalam dua kategori utama: yaitu algoritma berbasis perbandingan dan algoritma non-perbandingan. Algoritma berbasis perbandingan, seperti Heap Sort, Quick Sort, Merge Sort, dan Shell Sort, mengurutkan data dengan membandingkan elemen-elemennya, yang secara teoretis memiliki batas bawah kompleksitas waktu O(n log n) [7]. Di sisi lain, algoritma non-perbandingan, seperti Counting Sort, memanfaatkan sifat khusus dari data untuk mencapai efisiensi yang lebih tinggi, terutama pada dataset dengan rentang nilai yang terbatas. Dalam rata-rata kasus, Counting Sort dapat mencapai kompleksitas waktu O(n + k) [8], di mana k adalah rentang dari elemen data, yang dalam banyak kasus dapat lebih efisien daripada O(n log n). Namun, keunggulan ini sering kali dibayangi oleh keterbatasan Counting Sort dalam menangani data dengan rentang nilai yang luas atau data non-integer [9].

Penelitian sebelumnya telah banyak membahas performa algoritma pengurutan berbasis perbandingan dalam berbagai konteks. Sebagai contoh, studi terdahulu membandingkan Quick Sort dan Merge Sort pada dataset besar, menunjukkan bahwa Quick Sort lebih cepat dalam rata-rata kasus tetapi rentan terhadap kasus terburuk [10]. Studi lain menganalisis Heap Sort dan Shell Sort, menyoroti kestabilan dan efisiensi memori dari Heap Sort [11]. Namun, penelitian yang secara khusus membandingkan Counting Sort dengan empat algoritma pengurutan modern dalam hal efisiensi memori dan waktu komputasi pada dataset dengan rentang terbatas masih terbatas. Celah inilah yang menjadi fokus utama penelitian ini.

Penelitian ini bertujuan untuk menjawab pertanyaan: "Bagaimana efisiensi memori dan waktu komputasi dari Counting Sort dibandingkan dengan Heap Sort, Quick Sort, Merge Sort, dan Shell Sort pada dataset numerik acak dengan rentang 1-99?" Untuk mencapai tujuan ini, penelitian akan melibatkan pengujian empiris pada dataset dengan ukuran bervariasi dan rentang nilai yang dibatasi, menggunakan pendekatan analisis komparatif. Hasil pengujian akan dianalisis untuk menentukan algoritma mana yang paling efisien dalam hal penggunaan memori dan kecepatan komputasi.

Signifikansi dari penelitian ini terletak pada potensinya untuk memberikan panduan praktis dalam memilih algoritma pengurutan yang optimal berdasarkan karakteristik dataset, khususnya dalam aplikasi yang memerlukan pengolahan data cepat, seperti sistem tertanam atau aplikasi real-time. Selain itu, penelitian ini berkontribusi pada literatur akademik dengan mengisi celah pengetahuan mengenai performa

Counting Sort dalam konteks modern, yang sering kali diabaikan dalam diskusi tentang algoritma pengurutan. Keaslian penelitian ini terletak pada pendekatan komparatif yang komprehensif dan fokus pada dataset dengan rentang terbatas, yang relevan dengan tren terkini dalam pengolahan data sensor, IoT, dan analisis data skala besar yang sering kali melibatkan data numerik dengan rentang nilai yang dapat diprediksi.

Dengan demikian, penelitian ini diharapkan dapat memberikan wawasan baru dan rekomendasi yang berharga bagi para peneliti dan praktisi di bidang informatika dan ilmu komputer, khususnya dalam mengoptimalkan proses pengurutan data untuk aplikasi yang sensitif terhadap efisiensi memori dan waktu komputasi.

2. TINJAUAN PUSTAKA

2.1. ALGORITMA *SORTING*

Algoritma pengurutan (*sorting*) adalah proses menyusun elemen data dalam urutan tertentu, yang menjadi dasar bagi banyak operasi dalam ilmu komputer, seperti pencarian dan analisis data. Secara umum, algoritma sorting dapat dibagi ke dalam dua kategori, yaitu berbasis perbandingan dan non-perbandingan [12].

- Algoritma berbasis perbandingan, seperti Heap Sort, Quick Sort, Merge Sort, dan Shell Sort, bekerja dengan membandingkan elemen secara berpasangan dan memiliki batas bawah kompleksitas waktu O(n log n). Algoritma ini fleksibel untuk berbagai tipe data, namun efisiensinya bergantung pada jumlah perbandingan yang diperlukan.
- Algoritma non-perbandingan, seperti Counting Sort, memanfaatkan informasi khusus tentang data, seperti rentang nilai, untuk mengurutkan tanpa perbandingan langsung, yang memungkinkan kompleksitas waktu lebih rendah dalam kondisi tertentu.

2.2. ALGORITMA COUNTING SORT

Counting Sort adalah algoritma non-perbandingan yang efisien untuk mengurutkan data integer dengan rentang terbatas. Algoritma ini bekerja dengan menghitung frekuensi kemunculan setiap nilai dalam dataset dan menggunakan informasi tersebut untuk menempatkan elemen pada posisi yang benar dalam array terurut [13].

Kompleksitas waktunya adalah O(n + k), di mana n adalah jumlah elemen dan k adalah rentang nilai data. Ketika k jauh lebih kecil dari n, Counting Sort mendekati O(n), yang lebih efisien dibandingkan algoritma berbasis perbandingan. Namun, kelemahannya adalah penggunaan memori tambahan sebesar O(k) untuk array penghitung, yang dapat menjadi signifikan jika rentang data besar [14]. Oleh karena itu, Counting Sort sangat cocok untuk dataset dengan rentang terbatas, misalnya menggunakan rentang 1-99. Untuk memperjelas algoritma Counting Sort, berikut sintak Counting Sort dalam bahasa pemrograman Java:

```
public static void SortingCounting(int arr[]) {
    int n = arr.length;
    if (n == 0) return;

    // Temukan nilai maksimum dalam array
    int max = arr[0];
    for (int i = 1; i < n; i++) {
      if (arr[i] > max) {
        max = arr[i];
      }
    }

    // Inisialisasi array penghitung
    int[] count = new int[max + 1];

    // Hitung kemunculan setiap nilai
    for (int i = 0; i < n; i++) {
      count[arr[i]]++;
    }

    // Akumulasikan array penghitung
    for (int i = 1; i <= max; i++) {
      count[i] += count[i - 1];
    }

    // Buat array output
    int[] output = new int[n];

    // Bangun array output dari belakang ke depan
    for (int i = n - 1; i >= 0; i--) {
      output[count[arr[i]] - 1] = arr[i];
      count[arr[i]]--;
    }

    // Salin array output kembali ke array input
    for (int i = 0; i < n; i++) {
      arr[i] = output[i];
    }
}
```

2.3. EFISIENSI ALGORITMA *SORTING*

Efisiensi algoritma diukur melalui dua aspek utama: waktu komputasi dan penggunaan memori. Waktu komputasi mengacu pada durasi yang dibutuhkan algoritma untuk menyelesaikan tugas, yang dianalisis melalui kompleksitas waktu dalam notasi Big O. Penggunaan memori, atau efisiensi ruang, mengukur jumlah memori yang dibutuhkan algoritma, yang dapat mempengaruhi performa pada sistem dengan sumber daya terbatas [15].

Algoritma *in-place*, seperti Heap Sort dan Quick Sort, meminimalkan penggunaan memori tambahan, sedangkan algoritma seperti Merge Sort dan Counting Sort memerlukan ruang tambahan yang bergantung pada ukuran dataset atau rentang data [16]. Dalam penelitian ini, efisiensi memori dan waktu komputasi diukur secara empiris untuk membandingkan performa Counting Sort dengan algoritma pengurutan modern pada dataset numerik acak dengan rentang terbatas.

3. METODE PENELITIAN

Penelitian ini dilakukan dengan menggunakan bahasa pemrograman Java dan *Integrated Development Environment* (IDE) NetBeans versi 14 untuk mengembangkan serta menjalankan kode program. Pengujian dijalankan pada laptop dengan spesifikasi sebagai berikut:

- Prosesor AMD Ryzen 7 3750H 4-Core dengan kecepatan 2.3 GHz
- SSD sebesar 512 GB
- Memori RAM sebesar 8 GB
- Sistem operasi Windows 10 64-Bit

Desain penelitian mengadopsi metode eksperimental untuk menganalisis efisiensi memori dan waktu komputasi algoritma Counting Sort dibandingkan dengan algoritma pengurutan modern (Heap Sort, Quick Sort, Merge Sort, dan Shell Sort). Pendekatan eksperimental dipilih karena memungkinkan pengujian langsung terhadap kinerja algoritma dalam kondisi terkontrol, sehingga menghasilkan data yang terukur dan dapat dibandingkan secara objektif [17]. Desain ini mendukung tujuan penelitian untuk mengevaluasi performa algoritma secara sistematis, dengan fokus pada dua parameter utama: penggunaan memori dan waktu komputasi. Untuk memperjelas proses, alur kerja penelitian meliputi tahap persiapan dataset, implementasi algoritma, pengujian performa, pengumpulan data, dan analisis komparatif.

Prosedur eksperimen dimulai dengan pembuatan dataset numerik acak yang memiliki rentang nilai terbatas, yaitu antara 1 hingga 99. Dataset yang dibuat dibagi menjadi tiga ukuran yaitu: kecil (100 elemen), sedang (1.000 elemen), dan besar (10.000 elemen), guna mengevaluasi skalabilitas algoritma. Dataset dihasilkan menggunakan fungsi pembangkit bilangan acak dalam bahasa pemrograman Java, yang dipilih karena kemampuannya dalam pengolahan data dan pengukuran performa yang akurat. Fungsi yang digunakan dapat dilihat dibawah ini:

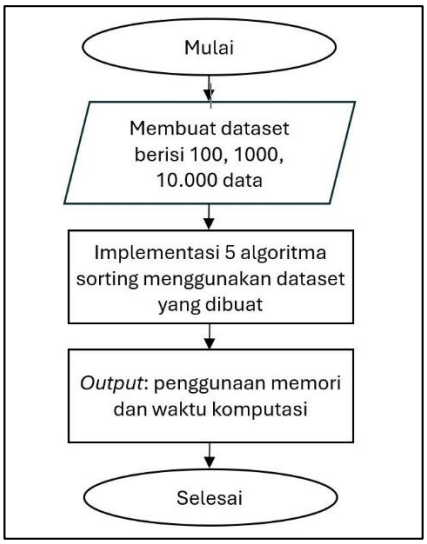
```
//Fungsi membuat dataset dalam bentuk array
public static int[] createArray(int count) {
      //Membuat array dengan ukuran 'count'
      int[] arr = new int[count];
      for (int i = 0; i < count; i++) {
         //Menghasilkan nilai acak 1-99
         arr[i] = (int) (Math.random() * 99 + 1);
      }
      //Menghasilkan dataset dalam bentuk array
      return arr;
}
```

Selanjutnya, kelima algoritma diimplementasikan dalam bahasa pemrograman Java dan diverifikasi kebenaran fungsionalitasnya sebelum pengujian. Pengujian performa dilakukan dengan menjalankan setiap algoritma pada ketiga ukuran dataset, mengukur waktu komputasi menggunakan System.nanoTime() dan penggunaan memori melalui Runtime.getRuntime().totalMemory() serta Runtime.getRuntime().freeMemory().

Pengumpulan data dilakukan secara sistematis melalui serangkaian eksperimen berulang. Setiap algoritma diuji tiga kali pada masing-masing ukuran dataset untuk memastikan konsistensi hasil dan meminimalkan fluktuasi acak. Pengujian dilakukan dalam lingkungan yang terisolasi, dengan hanya aplikasi yang relevan yang dijalankan untuk menghindari interferensi eksternal. Data yang dikumpulkan mencakup waktu komputasi dalam satuan nanodetik dan penggunaan memori dalam byte, yang kemudian direkam untuk setiap iterasi pengujian. Proses ini memastikan bahwa data yang diperoleh mencerminkan performa

algoritma secara akurat dalam kondisi yang terkontrol.

Analisis data dilakukan dengan menghitung rata-rata waktu komputasi dan penggunaan memori dari tiga pengujian untuk setiap kombinasi algoritma dan ukuran dataset. Teknik analisis komparatif diterapkan untuk membandingkan efisiensi antar algoritma, dengan memperhatikan tren performa seiring bertambahnya ukuran dataset. Untuk memvalidasi hasil, pengujian tambahan dilakukan pada dataset dengan rentang nilai yang lebih luas dan lebih sempit, guna memastikan apakah temuan dapat digeneralisasikan atau hanya berlaku dalam kondisi tertentu. Pendekatan ini memungkinkan penelitian untuk menyajikan evaluasi yang komprehensif dan obyektif, memberikan wawasan yang relevan mengenai keunggulan dan keterbatasan Counting Sort dibandingkan algoritma pengurutan modern dalam konteks efisiensi memori dan waktu komputasi. Gambar 1 menunjukkan gambaran dari tahapan pada penelitian ini.



Gambar 1. Diagram Alur Penelitian

4. HASIL DAN PEMBAHASAN

Hasil pengujian menunjukkan perbedaan yang signifikan dalam efisiensi memori dan waktu komputasi antara Counting Sort dan algoritma pengurutan modern yang diuji, yaitu: Heap Sort, Quick Sort, Merge Sort, dan Shell Sort. Pengujian dilakukan pada dataset dengan ukuran 100, 1.000, dan 10.000 elemen, masing-masing terdiri dari bilangan bulat acak dalam rentang 1 hingga 99, Dataset pengujian dapat dilihat pada link berikut: bit.ly/4miAHrn. Data eksperimen utama disajikan dalam Tabel 1 dan Tabel 2, yang merangkum rata-rata waktu komputasi dalam nanodetik dan penggunaan memori dalam byte untuk setiap algoritma berdasarkan tiga kali pengulangan demi memastikan akurasi. Dokumentasi eksperimen tiap algoritma dapat dilihat pada link berikut: bit.ly/4dnsvSL.

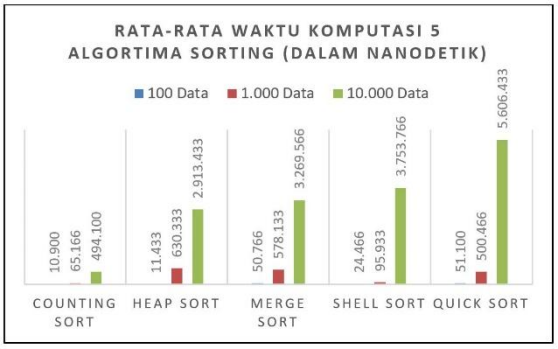
Tabel 1. Rata-rata Waktu Komputasi

Ukuran Dataset	Counting Sort	Heap Sort	Quick Sort	Merge Sort	Shell Sort
100	10.900	11.433	51.100	50.766	24.466
1.000	65.166	630.333	500.466	578.133	95.933
10.000	494.100	2.913.433	5.606.433	3.269.566	3.753.766

Tabel 2. Rata-rata Penggunaan Memori

Ukuran Dataset	Counting Sort	Heap Sort	Quick Sort	Merge Sort	Shell Sort
100	482.750	41.960	41.960	482.368	41.960
1.000	482.760	482.368	482.368	482.368	41.960
10.000	482.760	482.368	482.368	965.976	482.368

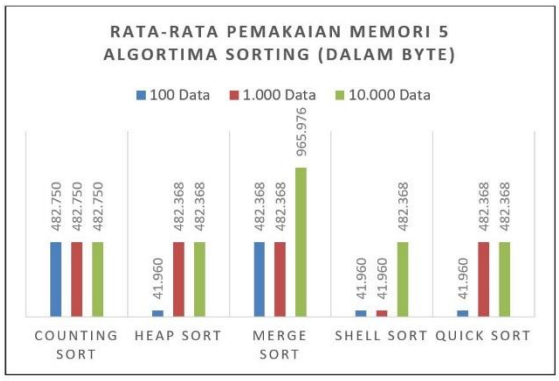
Untuk memberikan gambaran visual yang lebih jelas, Gambar 2 dan Gambar 3 mengilustrasikan rata-rata waktu komputasi dan rata-rata pemakaian memori pada 5 algoritma *sorting* berdasarkan ukuran dataset.



Gambar 2. Rata-rata Waktu Komputasi

Dari gambar 2. terlihat bahwa waktu komputasi kelima algoritma *sorting* menunjukkan variasi signifikan seiring peningkatan ukuran data dari 100, 1.000, hingga 10.000 elemen. Counting Sort menonjol sebagai algoritma dengan performa tercepat pada ketiga ukuran data, terutama pada 10.000 data, yang mengindikasikan kompleksitas waktu mendekati linear (O(n)) atau lebih rendah dibanding algoritma lainnya. Hal ini mungkin disebabkan oleh pendekatan non-comparison yang mengurangi operasi perbandingan. Di sisi lain, Heap Sort dan Merge Sort menunjukkan

tren pertumbuhan waktu yang lebih stabil, sesuai dengan karakteristik kompleksitas O(n log n). Shell Sort dan Quick Sort menunjukkan hasil yang beragam, Quick Sort mungkin mengalami performa yang tidak maksimal karena pemilihan pivot yang kurang optimal, sementara Shell Sort bergantung pada efisiensi gap sequence yang digunakan.



Gambar 3. Rata-rata Penggunaan Memori

Dari gambar 3. terlihat bahwa algoritma Counting Sort menunjukkan karakteristik penggunaan memori yang unik dibandingkan algoritma pengurutan modern seperti Heap Sort, Merge Sort, Shell Sort, dan Quick Sort. Pada dataset berisi 100 elemen data, Counting Sort memerlukan alokasi memori yang sama dengan Merge Sort namun lebih tinggi dari algoritma lainnya. Pada dataset berisi 1.000 elemen, hanya algoritma Shell Sort yang memerlukan alokasi memori paling kecil dibanding algoritma lainnya, hal ini karena Shell Sort menggunakan pendekatan berbasis insertion sort dengan optimasi *gap sequence*. Hasil ini mempertegas bahwa algoritma modern seperti Shell Sort, Heap Sort, dan Quick Sort lebih unggul dibanding Counting Sort dalam efisiensi memori untuk dataset berukuran 100 hingga 1.000.

Dari data pada Tabel 1 dan Gambar 2, Counting Sort secara konsisten menunjukkan waktu komputasi yang lebih rendah untuk semua ukuran dataset dibandingkan algoritma pengurutan lainnya (Heap Sort, Quick Sort, Merge Sort, dan Shell Sort). Pada dataset kecil berukuran 100 elemen, perbedaan waktu relatif kecil dan mungkin tidak terlihat dalam aplikasi praktis. Namun, pada dataset besar dengan 10.000 elemen, Counting Sort lebih cepat 6 sampai 10 kali dibandingkan algoritma modern lainnya. Keunggulan ini dapat dijelaskan melalui kompleksitas teoritis Counting Sort, yaitu O(n + k), di mana k adalah rentang data (dalam hal ini 99). Ketika n jauh lebih besar dari k, perilaku algoritma mendekati O(n), yang secara signifikan lebih efisien dibandingkan kompleksitas O(n log n) dari algoritma lainnya (Heap Sort, Quick Sort, Merge Sort, dan Shell Sort).

Dari data pada Tabel 2 dan Gambar 3, menunjukkan bahwa Counting Sort memerlukan lebih banyak memori dibandingkan algoritma *in-place* seperti Heap Sort, Quick Sort, dan Shell Sort, tetapi lebih hemat dibandingkan Merge Sort, terutama pada dataset besar. Hal ini disebabkan oleh kebutuhan Counting Sort akan array tambahan sebesar k, yang dalam penelitian ini tetap konstan pada 100 elemen, sehingga kontribusinya terhadap total memori relatif kecil dibandingkan ukuran dataset. Sebaliknya, Merge Sort membutuhkan ruang tambahan sebesar O(n), yang menyebabkan penggunaan memori meningkat secara linier dengan ukuran dataset, sebagaimana terlihat pada Tabel 2 dan Gambar 3. Dengan demikian, Counting Sort tidak selalu menjadi pilihan optimal untuk pengurutan dataset kecil, terutama jika efisiensi memori menjadi prioritas. Algoritma *in-place* seperti Heap Sort, Shell Sort, dan Quick Sort menunjukkan konsistensi dalam penggunaan memori yang minimal, menjadikannya lebih sesuai untuk aplikasi dengan keterbatasan sumber daya.

Temuan ini selaras dengan teori algoritma pengurutan (*sorting*) dan mendukung penelitian sebelumnya yang menegaskan efisiensi Counting Sort untuk data dengan rentang terbatas. Sebagai contoh, penelitian oleh peneliti [18] yang memaparkan tingginya efisiensi algoritma Counting Sort untuk dataset besar dengan rentang nilai yang diketahui, hasil penelitian menunjukkan waktu eksekusi Counting Sort yang lebih cepat (2,57 detik untuk 1 juta elemen) dibandingkan algoritma lain seperti Quick Sort (6,34 detik). Selain itu peneliti [19][20] juga menemukan bahwa Counting Sort memiliki keunggulan dalam stabilitas dan kompleksitas waktu yang rendah, yaitu O(n + k), di mana n adalah jumlah elemen dan k adalah rentang nilai, kelebihan ini menjadikannya efisien untuk data dengan rentang terbatas. Tak berbeda jauh dengan

penelitian sebelumnya, peneliti [21] membandingkan 5 Algoritma *Sorting* menggunakan bahasa pemrograman Python, Radix Sort yang merupakan varian dari Counting Sort untuk data integer unggul dalam kompleksitas waktu dalam semua kasus, namun memiliki penggunaan memori lebih tinggi dibandingkan beberapa algoritma lain.

Dampak dari penelitian ini terlihat pada aplikasi praktis di bidang yang melibatkan pengolahan data numerik dengan rentang terbatas, seperti sistem tertanam, pengolahan data sensor, atau aplikasi real-time pada perangkat IoT. Dalam konteks ini, kecepatan tinggi Counting Sort dapat meningkatkan performa sistem, menjadikannya pilihan yang layak dibandingkan algoritma pengurutan modern. Meskipun algoritma seperti Quick Sort atau Merge Sort lebih fleksibel untuk data dengan rentang luas atau tipe data kompleks, Counting Sort memiliki keunggulan dalam skenario data dengan rentang terbatas, sehingga memberikan panduan berbasis bukti bagi pengembang perangkat lunak.

Secara keseluruhan, penelitian ini menegaskan bahwa Counting Sort lebih unggul dalam efisiensi waktu dibandingkan Heap Sort, Quick Sort, Merge Sort, dan Shell Sort untuk dataset numerik dengan rentang terbatas, sambil tetap mempertahankan penggunaan memori yang kompetitif. Temuan ini secara langsung menjawab pertanyaan penelitian mengenai perbandingan efisiensi memori dan waktu komputasi antara Counting Sort dan algoritma pengurutan modern, sekaligus memenuhi tujuan awal untuk memberikan analisis mendalam yang dapat diandalkan berdasarkan data eksperimen. Hasil ini tidak hanya memperkuat teori yang ada, tetapi juga menawarkan implikasi praktis yang relevan untuk optimasi algoritma dalam aplikasi spesifik.

5. KESIMPULAN

a. Penelitian ini menunjukkan bahwa algoritma Counting Sort secara signifikan lebih unggul dalam efisiensi waktu komputasi dibandingkan algoritma pengurutan modern seperti Heap Sort, Quick Sort, Merge Sort, dan Shell Sort pada dataset numerik acak dengan rentang terbatas (1-99). Berdasarkan hasil eksperimen, Counting Sort mencatat waktu komputasi yang jauh lebih rendah, terutama pada dataset besar (10.000 elemen), di mana performanya hampir 6-10 kali lebih cepat dibandingkan algoritma lainnya, dengan kompleksitas waktu mendekati O(n). Namun, dalam hal efisiensi memori, Counting Sort memerlukan penggunaan memori yang lebih tinggi pada dataset kecil (100 elemen) dan sedang (1.000 elemen) dibandingkan algoritma in-place seperti Heap Sort dan Quick Sort. Pada dataset besar, penggunaan memorinya tetap kompetitif, bahkan lebih hemat dibandingkan Merge Sort yang membutuhkan ruang O(n).
b. Temuan ini secara eksplisit menjawab pertanyaan penelitian: "Bagaimana efisiensi memori dan waktu komputasi dari Counting Sort dibandingkan dengan Heap Sort, Quick Sort, Merge Sort, dan Shell Sort pada dataset numerik acak dengan rentang 1-99?" Counting Sort terbukti lebih efisien dalam waktu komputasi untuk semua ukuran dataset, tetapi kurang efisien dalam penggunaan memori pada dataset kecil dan sedang dibandingkan beberapa algoritma modern. Pada dataset besar, keunggulannya dalam waktu komputasi tetap dominan tanpa mengorbankan efisiensi memori secara berlebihan.
c. Dampak praktis dari penelitian ini signifikan, terutama untuk aplikasi yang memerlukan pengolahan data cepat dengan sumber daya terbatas, seperti sistem tertanam, pengolahan data sensor, atau perangkat IoT. Counting Sort dapat menjadi pilihan optimal dalam skenario tersebut, meningkatkan performa sistem dengan memanfaatkan karakteristik dataset numerik berentang terbatas. Sehingga temuan ini memberikan panduan berbasis bukti bagi pengembang dan peneliti dalam memilih algoritma pengurutan yang sesuai dengan kebutuhan spesifik aplikasi mereka.

DAFTAR PUSTAKA

[1] N. Cherezova, D. Mihhailov, S. Devadze, dan A. Jutman, “HLS-based Optimization of Tau Triggering Algorithm for LHC: a case study,”

dalam *2022 18th Biennial Baltic Electronics Conference (BEC)*, IEEE, 2022, hlm. 1–6.

[2] N. Nurhamni, "GEOGRAPHIC INFORMATION SYSTEM (GIS) USES A* ALGORITHM FOR SORTING NEAREST UMKM LOCATIONS," *Jurnal Informatika dan Teknik Elektro Terapan*, vol. 13, no. 2, 2025.

[3] J. Zhou, J. Zhang, X. Zhang, T. Xiao, D. Ma, dan C. Gong, "A Hybrid Vectorized Merge Sort on ARM NEON," dalam *International Conference on Algorithms and Architectures for Parallel Processing*, Springer, 2024, hlm. 26–36.

[4] I. P. Pujiono, E. H. Rachmawanto, dan N. A. S. Winarsih, "Array Sorting Algorithm vs Traditional Sorting Algorithm: Memory and Time Efficiency Analysis," *Jurnal Manajemen Informatika (JAMIKA)*, vol. 15, no. 1, hlm. 47–59, Apr 2025, doi: 10.34010/jamika.v15i1.13230.

[5] F. R. Wibowo dan M. Faisal, "Comparative analysis of sorting algorithms: TimSort Python and classical sorting methods," *JISA (Jurnal Informatika dan Sains)*, vol. 7, no. 1, hlm. 11–18, 2024.

[6] S. Wijaya, F. Fauziah, dan T. W. Harjanti, "Perbandingan Algoritma Sorting dengan Menggunakan Bahasa Pemograman Javascript dalam Penggunaan Waktu Komputasi dan Penggunaan Memori," *STRING (Satuan Tulisan Riset dan Inovasi Teknologi)*, vol. 8, no. 3, hlm. 294–302, 2024.

[7] E. Caizergues, F. Durand, dan F. Mathieu, "Anytime sorting algorithms (extended version)," *arXiv preprint arXiv:2405.08564*, 2024.

[8] P. Kumar, A. Gangal, S. Kumari, dan S. Tiwari, "Recombinant sort: N-dimensional cartesian spaced algorithm designed from synergetic combination of hashing, bucket, counting and radix sort," *arXiv preprint arXiv:2107.01391*, 2021.

[9] P. Kumar, A. Gangal, S. Kumari, dan S. Tiwari, "Recombinant sort: N-dimensional cartesian spaced algorithm designed from synergetic combination of hashing, bucket, counting and radix sort," *arXiv preprint arXiv:2107.01391*, 2021.

[10] O. E. Taiwo, A. O. Christianah, A. N. Oluwatobi, dan K. A. Aderonke, "Comparative study of two divide and conquer sorting algorithms: Quicksort and mergesort," *Procedia Comput Sci*, vol. 171, hlm. 2532–2540, 2020.

[11] O. Folorunso, O. R. Vincent, dan O. Salako, "An Exploratory Study of Critical Factors Affecting the Efficiency of Sorting Techniques (Shell, Heap and Treap)," *arXiv preprint arXiv:1203.1250*, 2012.

[12] A. Fenyi, M. Fosu, dan B. Appiah, "Comparative analysis of comparison and non comparison based sorting algorithms," 2020.

[13] S. Kumar dan P. Singla, "Sorting using a combination of Bubble Sort, Selection Sort & Counting Sort," *Mathematical Sciences and Computing*, vol. 2, hlm. 30–43, 2019.

[14] K. Goel, P. Dwivedi, dan O. Sharma, "Performance analysis of various sorting algorithms: Comparison and optimization," dalam *2023 11th International Conference on Intelligent Systems and Embedded Design (ISED)*, IEEE, 2023, hlm. 1–5.

[15] A. Chakraborty, "Calculation of the Comparative Efficiency of Algorithms Using a Single Metric," *arXiv preprint arXiv:2406.15510*, 2024.

[16] S. Edelkamp, A. Weiß, dan S. Wild, "QuickXsort: A fast sorting scheme in theory and practice," *Algorithmica*, vol. 82, no. 3, hlm. 509–588, 2020.

[17] I. P. Pujiono, R. B. Trianto, dan F. M. Hana, "Perbandingan Efisiensi Memori dan Waktu Komputasi Pada 7 Algoritma Sorting Menggunakan Bahasa Pemrograman Java," *Jurnal Sistem Informasi dan Sistem Komputer*, vol. 9, no. 2, hlm. 2018–230, Jul 2024.

[18] K. Kala, S. K. Budhani, R. S. Bisht, D. K. Bisht, dan K. S. Bumrah, "A novel sorting method for real and integer numbers: An extension of counting sort," *Int J Eng Adv Technol*, vol. 8, no. 6s4, hlm. 36–39, 2020.

[19] M. A. Ala'Anzy, Z. Mazhit, A. F. Ala'Anzy, A. Algarni, R. Akhmedov, dan A. Bauyrzhan, "Comparative Analysis of Sorting Algorithms: A Review," dalam *2024 11th International Conference on Soft Computing & Machine Intelligence (ISCMI)*, IEEE, 2024, hlm. 88–100.

[20] C. Song dan H. Li, "Improvement of counting sorting algorithm," *Journal of Computer and Communications*, vol. 11, no. 10, hlm. 12–22, 2023.

[21] M. Marcellino, D. W. Pratama, S. S. Suntiarko, dan K. Margi, "Comparative of advanced sorting algorithms (quick sort, heap sort, merge sort, intro sort, radix sort) based on time and memory usage," dalam *2021 1st international conference on computer science and artificial intelligence (ICCSAI)*, IEEE, 2021, hlm. 154–160.