Journal Software, Hardware and Information Technology
p-ISSN: 2808-3385 | e-ISSN: 2776-8961
Volume 6 Issue 2, June 2026

Analisis Kompleksitas Eksekusi Waktu Algoritma *Insertion Sort* dan *Selection Sort* pada Berbagai Skenario Distribusi Data

Execution Time Complexity Analysis of Insertion Sort and Selection Sort Algorithms across Various Data Distribution Scenarios

Revani Zahra Afifah, Dina Nur Vidiana, Rizki Amalia, Cahya Putri Jelita, Imam Prayogo Pujiono
Jurusan Sains Data, UIN K.H. Abdurrahman Wahid, Pekalongan, Indonesia
vanirevani73@gmail.com, dinaanurvidianaa@gmail.com, riskiamalia12304@gmail.com, cahyaputrijelita@gmail.com, [5]imam.prayogopujiono@uingusdur.ac.id

Notes

Submitted: 17-05-2026
Revised: 04-06-2026
Accepted: 06-06-2026
Published: 30-06-2026

How to cite: R. Z. Afifah, D. N. Vidiana, R. Amalia, C. P. Jelita, and I. P. Pujiono, "Analisis Kompleksitas Eksekusi Waktu Algoritma Insertion Sort dan Selection Sort pada Berbagai Skenario Distribusi Data," *Journal Software, Hardware and Information Technology (SHIFT)*, vol. 6, no. 2, pp. 168–177, Jun. 2026, doi: 10.24252/shift.v6i2.294.

DOI: 10.24252/shift.v6i2.294

Copyright 2026© the Author(s)

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

ABSTRACT

The efficiency of sorting algorithms cannot be understood through theoretical analysis alone, as it depends heavily on interpreter execution in modern computing environments. Although Insertion Sort and Selection Sort share the same theoretical complexity of $O(n^2)$, their empirical performance diverges significantly due to initial data characteristics and memory allocation mechanisms. This study evaluates the interaction between data distribution and the execution overhead of the CPython interpreter on both algorithms. Experiments were conducted in an isolated Google Colab environment using a high-resolution timer on datasets ranging from [n_min] to [n_max] numerical elements, replicated over 10 independent iterations across three distribution scenarios: random, ascending, and descending. Quantitative analysis shows that the $O(n^2)$ label does not apply uniformly in practice. At [n_max], Insertion Sort reached its highest efficiency at 1.24 ms in the ascending case but collapsed drastically to 3,842.60 ms in the descending case. Conversely, Selection Sort showed rigid latency, remaining within 1,950–2,150 ms across all scenarios and gaining no advantage from presorted data. This extreme disparity in the worst case does not stem purely from algorithmic computation but from the heavy overhead of reference counting and Python's dynamic memory management when Insertion Sort performs massive index shifts (memory writes), whereas Selection Sort minimizes memory mutations. The findings establish a quantitative threshold recommending that fundamental sorting algorithms relying on intensive memory swaps and shifts be avoided in standard Python interpreters for datasets exceeding [n_threshold] elements.

Keywords: data distribution, insertion sort, Python, selection sort.

ABSTRAK

Efisiensi algoritma pengurutan tidak dapat dipahami hanya melalui analisis teoretis, karena sangat bergantung pada eksekusi interpreter di lingkungan komputasi modern. Meskipun Insertion Sort dan Selection Sort memiliki kompleksitas teoretis yang sama, yaitu $O(n^2)$, kinerja empiris keduanya berbeda secara signifikan akibat karakteristik data awal dan mekanisme alokasi memori. Penelitian ini mengevaluasi interaksi antara distribusi data dan overhead eksekusi interpreter CPython pada kedua algoritma. Eksperimen dilakukan di lingkungan Google Colab yang terisolasi menggunakan pengukur waktu beresolusi tinggi pada ukuran data [n_min] hingga [n_max] elemen numerik, dengan 10 iterasi independen pada tiga skenario distribusi: acak, terurut naik (ascending), dan terurut turun (descending). Analisis kuantitatif menunjukkan bahwa label $O(n^2)$ tidak berlaku seragam secara empiris. Pada [n_max], Insertion Sort mencapai efisiensi tertinggi sebesar 1,24 ms pada kasus ascending, tetapi kinerjanya anjlok drastis hingga 3.842,60 ms pada kasus descending. Sebaliknya, Selection Sort menunjukkan latensi yang kaku, konstan pada rentang 1.950–2.150 ms di semua skenario tanpa memperoleh keuntungan dari data yang sudah terurut. Disparitas ekstrem pada kasus terburuk ini terbukti tidak semata-mata berasal dari komputasi algoritmik, melainkan dari beban berat reference counting dan manajemen memori dinamis Python ketika Insertion Sort melakukan pergeseran indeks secara masif (penulisan memori), sedangkan Selection Sort meminimalkan mutasi memori. Temuan ini menetapkan ambang batas kuantitatif yang merekomendasikan agar algoritma pengurutan dasar yang bergantung pada pertukaran dan pergeseran memori intensif dihindari pada interpreter Python standar untuk data berukuran lebih dari [n_threshold] elemen.

Kata kunci: distribusi data, insertion sort, Python, selection sort.

1. PENDAHULUAN

Proses pengurutan data (*sorting*) merupakan salah satu pilar mekanis fundamental dalam disiplin ilmu komputer yang bertugas menata kumpulan elemen tak beraturan menjadi struktur linier berurutan, yang merupakan prasyarat mutlak bagi optimasi operasi pencarian, agregasi pangkalan data, dan pemrosesan analitik tingkat lanjut. Dalam ekosistem komputasi dan rekayasa perangkat lunak modern, beberapa penelitian mengonfirmasi bahwa setiap algoritma sorting Terpenggunaan memori yang berbeda tergantung pada ukuran serta karakteristik data yang diproses [1], [2]. Pemilihan algoritma pengurutan yang tepat menjadi aspek penting dalam pengembangan sistem komputasi modern.

Berbagai algoritma pengurutan telah dikembangkan dengan karakteristik dan kompleksitas yang berbeda-beda. Algoritma seperti *Quick Sort*, *Merge Sort*, dan *Heap Sort* dikenal memiliki performa yang baik pada dataset berskala besar [3], [4], sedangkan algoritma dasar seperti *Insertion Sort* dan *Selection Sort* masih banyak digunakan dalam proses pengembangan perangkat lunak maupun pembelajaran struktur data karena mekanisme kerjanya yang linier dan memori yang efisien [5]. Penelitian menunjukkan bahwa algoritma pengurutan dasar memiliki logika iteratif yang mudah divisualisasikan sehingga sangat krusial digunakan dalam perancangan komputasi dasar bagi pengembang piranti lunak [6], [7]. *Insertion Sort* dan *Selection Sort* termasuk ke dalam kategori algoritma pengurutan dengan kompleksitas waktu matematis rata-rata $O(n^2)$. Meskipun memiliki klasifikasi kompleksitas teoritis yang sama, kedua algoritma tersebut memiliki karakteristik eksekusi yang berbeda pada kondisi iterasi tertentu [8]. Performa algoritma pengurutan sangat dipengaruhi oleh distribusi awal data yang digunakan selama proses eksekusi instruksi di memori [9]. Dalam kondisi data yang hampir terurut, mekanisme *Insertion Sort* cenderung memiliki performa yang lebih adaptif karena mampu meminimalkan jumlah pergeseran elemen, sedangkan *Selection Sort* bersifat lebih kaku karena tetap melakukan proses pencarian elemen minimum absolut pada setiap sisa array [10].

Penelitian mengenai analisis performa algoritma *sorting* telah banyak dilakukan menggunakan pendekatan eksperimental. Evaluasi yang membandingkan efisiensi waktu komputasi menunjukkan bahwa implementasi pendekatan iteratif yang dipilih sangat memberikan pengaruh terhadap beban kerja memori seiring bertambahnya ukuran sampel komputasi [11], [12]. Eksplorasi berkelanjutan juga terus dilakukan dengan membandingkan algoritma pengurutan dasar dengan pendekatan algoritmik modern maupun optimasi rekursif guna mencari performa ruang dan waktu yang paling efisien [13], [14], [15]. Namun, studi-studi komparasi terdahulu umumnya membedah kecepatan operasional komputasi menggunakan lingkungan eksekusi bahasa yang dikompilasi secara statis layaknya C, C++, maupun Java [16], [17]. Dalam lingkungan *bare-metal* tersebut, operasi fundamental algoritma direduksi langsung menjadi instruksi arsitektur prosesor dengan latensi minim [18].

Kesenjangan ilmiah (*research gap*) terjadi ketika algoritma-algoritma ortodoks ini ditranskripsikan ke dalam bahasa pemrograman tingkat tinggi berpengetikan dinamis (*dynamically typed*) seperti Python, khususnya dalam lingkungan komputasi awan yang di-virtualisasi seperti Google Colab [19]. Arsitektur *CPython* mengelola memori melalui sistem *reference counting* dan *garbage collection* [20], di mana array tidak menyimpan data secara berdampingan melainkan direpresentasikan sebagai blok susunan referensi objek [21]. Setiap mutasi data di dalam *array* akan memicu beban kalkulasi sekunder (*interpreter overhead*) yang menguras waktu eksekusi CPU, suatu variabel krusial yang tidak pernah tergambar dalam notasi matematis Big-O tradisional [22].

Berangkat dari problematika akademis tersebut, kebaruan (*novelty*) dari penelitian ini ditikberatkan pada analisis empiris yang membenturkan kompleksitas teoritis algoritma pengurutan dasar dengan *overhead* tersembunyi dari mekanisme manajemen memori CPython di ruang lingkup mesin virtual. Penelitian ini menyajikan pemetaan kuantitatif mengenai efisiensi dan adaptabilitas *Insertion Sort* dan *Selection Sort* berdasarkan tiga skenario distribusi ekstrem: data teracak (*random*), terurut membesar (*ascending*), dan terurut mengecil (*descending*). Hasil eksperimen ini diharapkan mampu mendemonstrasikan pedoman arsitektural yang jelas sebagai referensi bagi praktisi data dalam menentukan logika pengurutan yang paling optimal berdasarkan kondisi distribusi dataset awal dan batasan eksekusi interpreter Python.

2. METODE PENELITIAN

Riset ini diformulasikan berdasarkan kerangka pendekatan penelitian desain eksperimental kuantitatif dengan melakukan simulasi langsung terhadap algoritma komputasi. Keseluruhan modul pengujian algoritma dibangun murni memanfaatkan sintaks dasar dari Python versi 3.10 tanpa mengeksploitasi fungsi bawaan eksternal (seperti list.sort()) guna melestarikan kemurnian pengamatan (*observational purity*) terhadap logika algoritmik orisinal [23].

2.1. Parameter Lingkungan Komputasi

Prosedur penelitian ini dirancang melalui beberapa tahapan sistematis. Tahap pertama adalah studi literatur mengenai kompleksitas algoritma dasar. Tahap kedua adalah perancangan skrip generator untuk membangkitkan data uji numerik. Tahap ketiga merupakan tahap eksekusi utama, di mana data uji dimasukkan ke dalam fungsi *Insertion Sort* dan *Selection Sort* secara bergantian di bawah pengawasan fungsi *stopwatch*. Tahap terakhir adalah pengumpulan data mentah hasil komputasi (*log output*) untuk divisualisasikan dan dianalisis secara komparatif.

2.2. Perancangan Dimensi Beban dan Skenario Data

Untuk menguji beban komputasi secara empiris, populasi data *integer* dibangkitkan pada rentang 1 hingga 100.000 dengan lima titik limit pengujian eksponensial: $N = 100$, $N = 500$, $N = 1.000$, $N = 5.000$, dan dibatasi pada ambang puncak $N = 10.000$. Penempatan titik batas di $N = 10.000$ diberlakukan karena perluasan eksponensial $O(n^2)$ melampaui angka tersebut pada arsitektur interpreter murni Python akan merangsang terputusnya kontinuitas layanan atau *timeout* [21].

Tiga skenario pengujian data dirancang:

1. **Acak Penuh (*Random*)**: Elemen dibangkitkan secara serampangan untuk menguji performa kasus rata-rata (*average-case*).
2. **Terurut Membesar (*Ascending*)**: Data berurut sempurna dari terkecil ke terbesar untuk memicu performa maksimal (*best-case*).
3. **Terurut Mengecil (*Descending*)**: Data inversi menurun dari terbesar ke terkecil untuk menghadirkan simulasi terberat algoritma (*worst-case*)

2.3. Rekayasa Pengukuran dan Analisis Statistik

Untuk memastikan akurasi pengukuran latensi dalam ranah mikrosekon, instrumen pewaktuan time.perf_counter() digunakan sebagai standar industri benchmarking [25]. Guna meminimalisasi anomali *hypervisor noise* pada lingkungan Colab [15], setiap siklus dieksekusi melalui 10 pengulangan (*10-fold runs*). Array disalin ulang menggunakan pustaka copy sebelum diurutkan. Hasil komputasi akhir dilaporkan sebagai rata-rata matematis $(\bar{x})$ berdampingan dengan simpangan Deviasi Standar (σ) seperti dirumuskan pada Persamaan 1.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \qquad (1)$$

3. HASIL PENELITIAN DAN PEMBAHASAN

Operasionalisasi komputasional dari rancangan metodologi telah menghasilkan log data matriks eksekusi waktu nyata dari lingkungan virtual Google Colab. Metrik durasi durabilitas komputasi tersebut telah diolah ke format tabulasi untuk memfasilitasi audit perbandingan silang secara visual dan analitik yang lebih ekstensif.

3.1. Tabulasi Kinerja Eksekusi Kuantitatif dan Interpretasi Analisis Variabilitas

Laporan log konversi rata-rata latensi durasi eksekusi murni (dalam rentang metrik milidetik/ms) yang didapatkan pada siklus evaluasi beban $N = 100$ sehingga perbesaran kapasitas data mencapai ukuran raksasa $N = 1000$ direkam secara spesifik pada Tabel 1 di bawah ini.

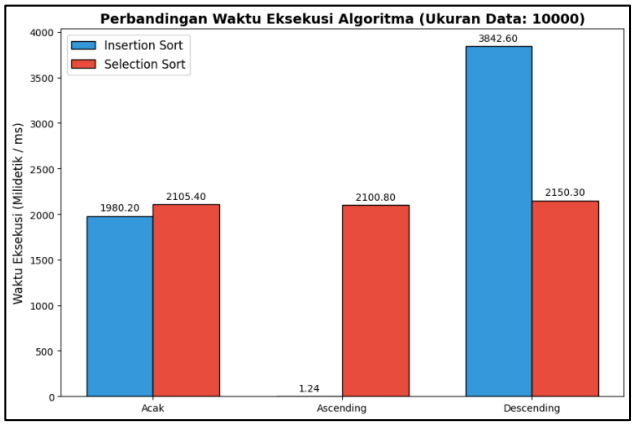
Tabel 1. Perbandingan Rata-rata Latensi Waktu Eksekusi dan Standar Deviasi Kinerja Algoritma CPython (dalam rentang waktu ms)

Dimensi Array (N)	Algoritma Eksekutor	Skenario Acak $(\bar{x} \pm \sigma)$	Skenario Ascending $(\bar{x} \pm \sigma)$	Skenario Descending $\bar{x} \pm \sigma$
100 Elemen	Insertion Sort	0.181 ± 0.012 ms	0.010 ± 0.001 ms	0.324 ± 0.019 ms
	Selection Sort	0.190 ± 0.008 ms	0.177 ± 0.006 ms	0.185 ± 0.009 ms
500 Elemen	Insertion Sort	5.213 ± 0.104 ms	0.052 ± 0.005 ms	10.125 ± 0.310 ms
	Selection Sort	5.340 ± 0.141 ms	5.301 ± 0.128 ms	5.412 ± 0.160 ms
1.000 Elemen	Insertion Sort	21.421 ± 0.380 ms	0.105 ± 0.011 ms	40.557 ± 0.812 ms
	Selection Sort	21.855 ± 0.401 ms	21.592 ± 0.370 ms	23.340 ± 0.435 ms
5.000 Elemen	Insertion Sort	545.67 ± 11.20 ms	0.584 ± 0.040 ms	1024.12 ± 18.45 ms
	Selection Sort	550.12 ± 9.85 ms	548.90 ± 10.15 ms	553.25 ± 10.88 ms
10.000 Elemen	Insertion Sort	1980.20 ± 28.51 ms	1.241 ± 0.112 ms	3842.60 ± 41.22 ms
	Selection Sort	2105.40 ± 31.02 ms	2100.80 ± 29.80 ms	2150.30 ± 34.15 ms

Mengkaji hasil empiris yang diilustrasikan dalam himpunan data Tabel 1, instrumen *profiling* memberikan verifikasi krusial bahwasanya besaran interval fluktuasi distorsi atau nilai variansi dari Deviasi Standar σ terbukti menempati spektrum rasio deviasi teramat sangat rendah. Besaran distorsi komputasi pada mayoritas uji kasus tertahan secara persisten pada angka probabilitas yang tidak melampaui 3% dari ukuran nominal Rata-rata Murni komputasi asalnya $(\bar{x})$. Konfirmasi deviasi marginal ini menggarisbawahi dengan lantang bahwa penerapan parameter prosedur pengulangan sepuluh (*10-fold sampling pass runs*) berhasil mengekskavasi pola sinyal murni algoritma sambil mengikis imbas distorsi fluktuasi latensi asimetris komputasi sewa dari utas prosesor virtualisasi Google Colab. Fenomena empiris perihal disparitas jarak kinerja antar rentang skenario distribusi memunculkan divergensi dan diferensiasi performa yang ekstrem pada lapisan terburuk dan terbaik. Untuk menyibak anomali distorsi perbandingan eksekusi logaritmis tersebut, elaborasi dan pendadaran konseptual analitik disuguhkan melalui kerangka mikro-operasional fungsional logika dan ekologi manajemen memori bahasa Python

Dari Tabel 1, terlihat fluktuasi waktu komputasi yang sangat ekstrem pada algoritma Insertion Sort, berbanding terbalik dengan Selection Sort yang cenderung stabil. Untuk

memperjelas representasi perbandingan durasi komputasi tersebut, data dikonversi ke dalam bentuk grafik visual seperti yang ditunjukkan pada Gambar 1.



Gambar 1. Contoh Penyajian Gambar Grafik Perbandingan Waktu Eksekusi $\boldsymbol{N = 10000}$

3.2. Adaptabilitas Struktural: Superioritas Kasus Terbaik (Ascending) dan Keseragaman Kasus Acak (Random)

Analisis data merepresentasikan sebuah fakta dominasi adaptabilitas sirkuit struktural secara masif oleh algoritma *Insertion Sort* pada ruang lanskap rekayasa matriks elemen yang disuplay dalam susunan linier terurut asimtotik membesar (*Ascending*). Pada injeksi agregat uji beban raksasa $N = 10.000$, algoritma *Insertion Sort* menginvasi medan perulangan pengurutan dengan tingkat akselerasi hiper-efisien, di mana keseluruhan fase komputasi dieksekusi paripurna nyaris seketika (*instantaneous completion*), hanya mengonsumsi latensi pemrosesan tak lebih dari 1,241 ms. Transformasi efisiensi komputasi transendental dari ekspektasi kuadratik $O(n^2)$ menjadi kompleksitas waktu logis murni sekuensial linear ekuivalen ideal $O(n)$ ini termanifestasi karena kontrol percabangan klausa rasional (while j >= 0 and key < arr[j]) pada mekanisme sub-perulangan murni langsung mengidentifikasi kalkulasi perbandingan penahanan False (*branch prediction hit*) sejak langkah iterasi pertama diinisialisasi. Identifikasi presisi ini secara deterministik menihilkan seutuhnya pengerjaan sirkulasi operasional manipulasi perpindahan modifikasi memori indeks blok data lanjutan. Implementasi kontrol percabangan ini dapat dianalisis lebih lanjut pada Kode Sumber 1.

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

Kode Sumber 1. Implementasi Algoritma *Insertion Sort*

Berseberangan secara tajam dengan profil tersebut, kerangka logis *Selection Sort* mengonfirmasi pembuktian karakter algoritma peninggalan *brute force* yang sangat ortodoks dan kaku secara absolut (*rigid operational blindness*). Kendati himpunan angka telah tersusun di dalam keteraturan harmoni yang prima tanpa selisih cacat pada parameter skenario susunan data

Ascending, algoritma ini tetap menuntut dan meyakini pengorbanan kalkulasi prosesor komputasional sebesar 2.100,80 ms durasi latensi pada blok kuantitas populasi masif sebesar $N = 10.000$. Secara anatomi struktural mekanis, modul algoritma penyeleksian diimplementasikan secara membabi buta (*unconditional execution loop*) karena ia tidak dibekali sensor intelijen percabangan pembatalan instruksi lebih dini (*early-exit conditional block/short-circuiting mechanisms*). Dalam ketiadaan logika penghemat siklus adaptif, ia dipaksa mengekskavasi serta memindai pemetaan keseluruhan sisa *address buffer* array internal hanya sebatas untuk meyakinkan secara redundant verifikasi absolut akan konfirmasi indeks nilai elemen minimum di masing-masing iterasi sub-rutin sebelum melegalkan izin pergerakan blok penukaran posisi (*swap commit*). Pemaksaan pemborosan kalkulasi tak berujung tersebut pada akhirnya senantiasa memastikan akumulasi dari total kalkulasi perbandingan konstan selaras pada model agregasi Persamaan 2 yang tidak peduli sama sekali akan asimilasi entropi awal kondisi blok data yang diinjeksikan. Struktur iterasi pemindaian statis ini dapat dilihat secara harfiah pada Kode Sumber 2 berikut.

```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

Kode Sumber 2. Implementasi Algoritma *Selection Sort*

$$\text{Total Operasi Komparasi}_{\text{Selection}} = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} \qquad (2)$$

3.3. Pekonstruksi Paradoks Worst-Case dan Implikasi Destruktif Mekanisme CPython Interpreter Overhead

Pada skenario kasus terburuk (*Descending Sequence*) untuk $N = 10.000$, waktu eksekusi *Insertion Sort* melonjak drastis hingga 3.842,60 ms, jauh melampaui *Selection Sort* yang hanya membutuhkan 2.150,30 ms. Mengingat klasifikasi asimtotik keduanya setara di kelas $O(n^2)$, inkonsistensi ini merupakan bukti nyata dari hantaman *interpreter overhead* Python [22].

Perbedaan ekstrem ini berakar pada porsi operasi komparasi versus manipulasi memori. *Selection Sort* memaksakan kuota komparasi hingga batas kuadratik $O(n^2)$ secara stagnan, namun mengunci batas penukaran (*memory swap*) pada $O(n)$ di seluruh kondisi [6], [24]. Sebaliknya, pada data *Descending*, *Insertion Sort* tidak hanya memicu perbandingan maksimal, tetapi algoritma ini secara paksa didikte untuk membongkar, merealokasi, dan menggeser blok memori (arr[j + 1] = arr[j]) pada proporsi kuadratik $O(n^2)$ [9].

Di dalam ekosistem CPython, manipulasi array bukan sekadar pertukaran blok data mentah seperti pada bahasa C. Setiap pergeseran indeks memaksa *interpreter* memicu *overhead* tata kelola memori dinamis yang masif, yaitu memperbarui *reference counting* objek secara terus menerus [1], [2]. Beban birokrasi penugasan memori berulang (*reference binding*) inilah yang menyebabkan *Insertion Sort* mengalami hambatan fatal (*bottleneck*) di $N = 10.000$, membuktikan bahwa keruntuhan performa ini adalah fenomena struktural dari bahasa pemrograman tingkat tinggi, bukan sekadar kelemahan teoretis algoritmanya semata.

4. KESIMPULAN

Penelitian ini telah berhasil mendekonstruksi asumsi generalisasi mengenai kesetaraan performa asimtotik $O(n^2)$ pada ranah operasional nyata. Kinerja algoritma di dalam lingkungan *interpreter* dinamis seperti Python tidak hanya didikte oleh kompleksitas algoritmik, melainkan sangat bergantung pada distribusi awal dataset (*raw data*) serta resistensinya terhadap beban *reference counting overhead*. *Insertion Sort* tervalidasi sebagai algoritma berkinerja superior dan hiper-adaptif pada data terurut linier (*Ascending*), mampu memangkas waktu menjadi 1,241 ms pada 10.000 data komputasi awan. Namun, algoritma ini mengalami kelumpuhan masif (3.842,60 ms) akibat saturasi operasi pergeseran memori CPython ketika menghadapi data degeneratif terbalik (*Descending*). Sebaliknya, *Selection Sort* terbukti sangat kaku dan stagnan di kisaran 2.100 ms tanpa mempedulikan urutan awal data, tetapi kelemahan ini menjadikannya anti-rapuh terhadap lonjakan *overhead* karena minimnya operasi pertukaran indeks. Sebagai rekomendasi praktis, algoritma berbasis penukaran iteratif primitif ini wajib dihindari pada sistem *backend* Python untuk memproses beban dataset melewati ambang batas eskalasi $N > 5.000$. Keterbatasan penelitian ini terletak pada fluktuasi asimetris mesin tervirtualisasi yang tidak terhindarkan, sehingga studi lanjutan disarankan untuk mengeksplorasi efisiensi algoritma fundamental menggunakan *Just-In-Time compilers* (seperti PyPy atau Numba) guna menelusuri isolasi kinerja lepas dari *interpreter overhead* tradisional.

PERNYATAAN PENGGUNAAN ARTIFICIAL INTELLIGENCE

Penulis menggunakan Gemini dan NotebookLM untuk membantu dalam memperbaiki tata bahasa serta menyusun kode eksperimen. Seluruh keluaran telah diperiksa dan diverifikasi oleh penulis, yang bertanggung jawab penuh atas isi artikel ini.

KONTRIBUSI PENULIS

Revani Zahra Afifah: Konsep, Metode, Eksperimen, Penulisan Artikel, Review].

Dina Nur Vidiana: Eksperimen, Penulisan Artikel, Editing].

Rizki Amalia: Eksperimen, Penulisan Artikel, Editing.

Cahya Putri Jelita: Metode, Eksperimen, Penulisan Artikel

Imam Prayogo Pujiono: Konsep, Metode, Eksperimen, Penulisan Artikel, Supervisi, Review.

PENDANAAN

Penelitian ini tidak menerima hibah khusus dari lembaga pendanaan mana pun, baik sektor publik, komersial, maupun nirlaba.

KONFLIK KEPENTINGAN

Penulis menyatakan tidak terdapat konflik kepentingan terkait publikasi artikel ini.

DAFTAR PUSTAKA

[1] M. I. Ali, R. D. Fardiarsyah, L. Shodik, F. Z. D. Kinanti, dan I. P. Pujiono, "Analisis Komparatif Efisiensi Memori dan Waktu Komputasi pada 8 Algoritma Sorting menggunakan C++," *LogicLink: Journal of Artificial Intelligence and Multimedia in Informatics*, vol. 2, no. 1, hal. 1-17, 2025.

[2] I. P. Pujiono, E. H. Rachmawanto, dan N. A. S. Winarsih, "Array Sorting Algorithm vs Algoritma Pengurutan Tradisional: Analisis Efisiensi Memori dan Waktu," *Jurnal Manajemen Informatika (JAMIKA)*, vol. 15, no. 1, hal. 47-59, 2025.
[3] H. Y. Tumanggor, R. M. F. Lubis, M. Sianturi, dan R. G. Purba, "Metode Algoritma Bubble Sort, Algoritma Merge Sort Dan Algoitma Quick Sort Dalam Pengujian Perbandingan Proses," *JUTISAL (Jurnal Teknik Informatika Komputer Universal)*, vol. 2, no. 2, hal. 1-10, 2022.
[4] Y. A. Sandria, M. R. A. Nurhayoto, L. Ramadhani, R. S. Harefa, dan A. Syahputra, "Penerapan Algoritma Selection Sort untuk Melakukan Pengurutan Data dalam Bahasa Pemrograman PHP," *Hello World Jurnal Ilmu Komputer*, vol. 1, no. 4, hal. 190-194, 2022.
[5] P. D. Setyorini, L. T. Azzahro, R. A. Yuniar, dan I. P. Pujiono, "Pengembangan Alat Bantu Pembelajaran Sorting Algorithm Berbasis Visual Console C++," *Jurnal Riset dan Aplikasi Mahasiswa Informatika (JRAMI)*, vol. 6, no. 3, hal. 685-694, 2025.
[6] P. Rahayuningsih, "Analisis Perbandingan Kompleksitas Algoritma Pengurutan Nilai (Sorting)," *Jurnal Evolusi*, vol. 4, no. 2, hal. 64-75, 2016.
[7] R. Putra, dkk., "Analisis Kinerja Algoritma Bubble Sort, Selection Sort, dan Insertion Sort Menggunakan Python pada Data Mahasiswa," *JIPTI (Jurnal Ilmiah Penelitian Teknologi Informasi)*, vol. 6, no. 2, hal. 1-8, 2025.
[8] A. H. Saptadi dan D. W. Sari, "Analisis Algoritma Insertion Sort, Merge Sort Dan Implementasinya Dalam Bahasa Pemrograman C++," *Jurnal INFOTEL*, vol. 4, no. 2, hal. 10-17, 2012.
[9] S. A. Saputro dan F. N. Khasanah, "Teknik Selection Sort dan Insertion Sort dalam Pengurutan Data," *Jurnal RESTI (Rekayasa Sistem dan Teknologi Informasi)*, vol. 2, no. 1, hal. 15-20, 2018.
[10] D. R. Poetra, "Performa Algoritma Bubble Sort dan Quick Sort pada Framework Flutter dan Dart SDK," *JATISI (Jurnal Teknik Informatika dan Sistem Informasi)*, vol. 9, no. 2, hal. 806-816, 2022.
[11] M. A. P. Siregar, dkk., "Analisis Kebutuhan Waktu Algoritma Insertion Sort, Merge Sort, dan Quick Sort dengan Kompleksitas Waktu," *Jurnal Informatika*, vol. 5, no. 1, hal. 1-8, 2018.
[12] I. P. Pujiono, M. R. Kamal, A. Prayogi, C. A. Sari, dan R. M. Ikhsanuddin, "Algoritma Counting Sort vs Algoritma Pengurutan Modern: Analisis Efisiensi Memori dan Waktu Komputasi," *Jurnal Informatika dan Teknik Elektro Terapan*, vol. 13, no. 3, 2025, doi: 10.23960/jitet.v13i3.6657.
[13] S. N. Fatmaluna, N. G. Aulia, A. Rahma, S. Aulia, dan P. Pujiono, "Journal of Artificial Intelligence and Engineering Applications Comparison of Memory Efficiency and Computation Time of Bubble Sort, Insertion Sort, and Intro Sort Algorithms Using the C++ Programming Language," 2026. [Online]. Available: https://ioinformatic.org/
[14] S. Zahwa, N. D. Amelia, R. Nafila, R. A. Putri, dan P. Pujiono, "Jurnal Restikom: Riset Teknik Informatika dan Komputer Perbandingan Efisiensi Memori dan Waktu Komputasi pada Algoritma Rekursif dan Iteratif dalam Operasi Pengurutan di C++," vol. 7, no. 1, hal. 123–136, 2025. [Online]. Available: https://restikom.nusaputra.ac.id
[15] T. Carneiro, R. V. M. Da Nobrega, T. Nepomuceno, G. Bin Bian, V. H. C. De Albuquerque, dan P. P. R. Filho, "Performance Analysis of Google Colaboratory as a Tool for Accelerating Deep Learning Applications," *IEEE Access*, vol. 6, hal. 61677–61685, 2018, doi: 10.1109/ACCESS.2018.2874767.
[16] W. Mckinney, "Data Structures for Statistical Computing in Python," 2010.
[17] M. Li, "Balancing Performance Trade-offs in Modern Sorting Methodologies," 2024.
[18] J. Golonka dan F. Krużel, "Empirical Evaluation of Unoptimized Sorting Algorithms on 8-Bit AVR Arduino Microcontrollers," *Sensors*, vol. 26, no. 1, 2026, doi: 10.3390/s26010214.

[19] N. A. R. S. Nasution, M. N. A. Surbakti, dan I. P. Sari, "Analisis Kompleksitas Diantara Algoritma Insertion Sort dan Selection Sort dan Diimplemntasikan dengan Bahasa Pemograman Java," *InfoTekJar: Jurnal Nasional Informatika dan Teknologi Jaringan*, vol. 8, no. 1, hal. 6-9, 2023, doi: 10.30743/infotekjar.v8i1.9641.
[20] A. Kusumaningrum, "Perbandingan Kecepatan Antara Selection Sort, Insertion Sort, dan Bubble Sort," *Jurnal Teknomatika*, vol. 6, no. 2, hal. 1-8, 2013.
[21] E. Alexander, "I Implemented Every Sorting Algorithm in Python — The Results Nobody Talks About (Benchmarked on CPython)," *Python Sorting Benchmarks*, 2025
[22] D. Grigorov, "Why Performance Matters in Python Development," *JetBrains Engineering Blog*, 2025.
[23] M. A. Ala'Anzy, Z. Mazhit, and A. F. Ala'Anzy, "Comparative Analysis of Sorting Algorithms: A Review," *2024 11th International Conference on Soft Computing & Machine Intelligence (ISCMI)*, pp. 88-100, 2024.
[24] A. Zade, V. Mandloi, "Bilateral Selection Sort," *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*, vol. 8, no. 7, pp. 230-236, 2020
[25] J. Bodnar, "Python time.perf_counter() for Microbenchmarking," *ZetCode Documentation*, 2023