Journal of Artificial Intelligence and Engineering Applications

Website: https://ioinformatic.org/

15th February 2026. Vol. 5. No. 2; e-ISSN: 2808-4519

Comparison of Memory Efficiency and Computation Time of Bubble Sort, Insertion Sort, and Intro Sort Algorithms Using the C++ Programming Language

Shabina Nur Fatmaluna[1], Nazwa Gista Aulia[2], Adhiel Rahma[3], Salma Aulia[4], Imam Prayogo Pujiono[5]

[12345]*Universitas Islam Negeri K.H Abdurrahman Wahid Pekalongan*
shabina.nur.fatmaluna2025@mhs.uingusdur.ac.id[1], nazwa.gista.aulia.25029@mhs.uingusdur.ac.id[2], adhiel.rahma25028@mhs.uingusdur.ac.id[3], salma.aulia25017@mhs.uingusdur.ac.id[4], imam.prayogopujiono@uingusdur.ac.id[5]

Abstract

Data sorting is a fundamental step in the computer process that greatly affects the effectiveness of programs and overall system performance. In this study, three sorting algorithms, namely Bubble Sort, Insertion Sort, and Intro Sort, are analyzed and compared using recursive and iterative approaches. Bubble Sort serves as a basic algorithm example to understand the basic idea of element exchange, while Insertion Sort was chosen for its efficiency on small and nearly sorted datasets. Intro Sort, as a combination algorithm that integrates Quick Sort, Heap Sort, and Insertion Sort, was studied to reveal how its adaptive mechanism can provide more optimal results. The testing was conducted by measuring execution speed, sorting stability, and memory usage efficiency. The findings from this study show that Bubble Sort ranks lowest in terms of performance and is less suitable for large data sets. Insertion Sort shows better results on small data sets and those with similar patterns. Intro Sort emerges as the most effective algorithm with stable processing time, high adaptability, and faster and more efficient sorting results for various data sizes. Overall, this study emphasizes the importance of choosing a sorting algorithm that suits the characteristics of the data and the needs of the application. The combination of adaptive strategies such as those in Intro Sort is the best solution for current data processing, which demands high speed and efficiency.

Keywords: *Bubble Sort; Insertion Sort; Intro Sort; Sorting Algorithm; Recursive; Iterative; Time Complexity.*

1. Introduction

In this digital age, efficient data processing has become an important aspect of software development, especially for applications that require large amounts of data processing and fast response times [1]. In today's computing world, an important part of data management is the sorting process, known as sorting [2]. Sorting is used to organize data so that it can be viewed and understood more quickly, better, and in a more organized manner [3]. This includes Bubble Sort and Insertion Sort, where they found that Bubble Sort consistently showed the lowest performance in execution time and memory efficiency across all data scales [4] .

To deepen the scope of the study, this research also analyzes Intro Sort. This hybrid algorithm combines Quick Sort, Heap Sort, and Insertion Sort to achieve a higher level of adaptability to various types of data [5]. The stability of Intro Sort's performance is guaranteed by its ability to fall back to Heap Sort when the recursion depth becomes inefficient [6]. Specifically, this study is designed to evaluate and compare the performance of Bubble Sort, Insertion Sort, and Intro Sort in terms of processing speed and memory allocation, using C++ as the implementation platform [7].

In addition, good data sorting is very important for making application systems work better when handling large or constantly changing amounts of data [8]. Choosing the right sorting method can affect the quality of data analysis and the speed of information access, making it an important consideration when developing new software [9]. Previous studies have also shown that comparing the performance of various sorting methods is necessary to find the best way to do things in different data conditions, taking into account the time required and the amount of memory used [10].

According to research conducted by [11], how well a sorting algorithm works depends on the type of algorithm and the programming language used. In the study, they explained that differences in how high-level and mid-level languages manage memory usage can change the performance measurement results when sorting data.
When used with certain algorithms, C++ proved to be faster than Python because it is a compiled language [12]. This advantage arises due to C++'s ability to be directly converted into machine code, which reduces the load when running programs and allows for more efficient

use of hardware resources [13]. In addition, C++ offers flexibility in manual memory management and utilizes more advanced compiler optimizations, making it superior in large-scale data management and computation that requires high resources.

Other studies also show that C++'s structure, which supports object-oriented paradigms and low-level control, allows it to execute commands more efficiently than Python [14]. Therefore, these studies consistently show that compiled languages such as C++ can operate faster because they are closer to the hardware architecture than high-level languages that use interpreters.

The C++ programming language was used in this study because it offers high execution speed and flexibility in managing memory usage. This is in line with the opinion [15], which states that C++ provides full control over data structure and data storage location, making it suitable for experiments that focus on algorithm performance.

2. Research Methodology

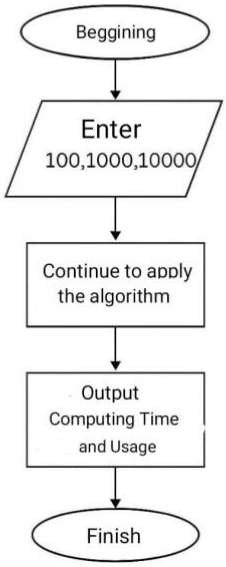

Fig. 1: Visualization of the Execution Flow in the Sorting Algorithm

2.1. Bubble Sort

[16] Explains that Bubble Sort is one of the simplest sorting algorithms and is usually used as basic material in programming education. The simplicity of the concept and ease of understanding the steps make this algorithm popular among beginners [17]. Although it is not efficient for large data sets, Bubble Sort still has advantages because it is easy to implement. [18] Adds that Bubble Sort is able to complete the sorting process gradually with a relatively short time for each step, and is easy to remember and implement. However, its average time complexity of $O(n^2)$ makes it less effective when used for large amounts of data.

In its application, Bubble Sort works by comparing two adjacent elements, then swapping them if their positions do not match the desired order. This process is repeated until all elements are arranged correctly. Although the mechanism is simple, Bubble Sort is not recommended for processing large amounts of data due to its high execution time [19].

2.2 Intro Sort

Intro sort is an internal sorting algorithm chosen because it has low computational complexity and is suitable for small amounts of data such as pixels in an infrared image filter window [23]. In the context of noise reduction, the window size is usually small, for example 3×3, 5×5, or 7×7, so the use of internal algorithms becomes more efficient.

This algorithm works by starting the sorting process using Quick sort, but adaptively switches to Heap sort when the recursion depth exceeds the limit calculated from the number of elements being sorted. This automatic switch allows Intro sort to combine the speed of Quick sort on general data and the stability of Heap sort in the worst case, thus maintaining optimal performance.
Intro sort has a time complexity of O(n log n), which makes it more efficient than various simple sorting algorithms commonly used to find the median in traditional median filters. In the noise removal method, Intro sort is used to obtain median values with lower computational load so that the noise reduction process can be performed faster.

Another advantage is that Intro sort is only run on pixels detected as noisy pixels, not on all pixels as in conventional median filters. Thus, the number of sorting operations is significantly reduced. This enables infrared image processing to be faster and suitable for real-time applications

3. Results And Discussion

In this study, the Bubble Sort, Insertion Sort, and Intro Sort algorithms were tested using the C++ programming language. The main objective of the testing was to assess the performance of each algorithm in terms of execution speed and the amount of memory used when processing random, unsorted data. The test data used consisted of 100, 1000, and 10000 random numbers, where each algorithm was tested using the same data set so that the results obtained were objective and consistent.

The testing process was carried out by running the sorting program for each data size, then recording the execution time required and calculating the memory usage. Through this stage, it is hoped that the algorithm with the best efficiency in terms of processing speed and memory resource savings in C++-based implementation can be identified.

Each algorithm has different characteristics and levels of complexity, so the test results are expected to provide a realistic picture of the performance of each algorithm for various input data sizes. The test results are presented in tables and graphs to make the data analysis and interpretation process easier and more structured.

Testing was conducted using a laptop with the following specifications:

4. Operating System: Windows 11 Pro 64-bit.
5. Processor: AMD Ryzen 5 5700U @ 1.80-4.30GHz (8 Cores / 16 Threads).
6. RAM: 16 GB DDR4.
7. Storage: 512 GB NVMe SSD.
8. Compiler: g++ (GNU Compiler) C++17.
9. Development Environment: Visual Studio Code / MinGW / Online Compiler.

3.1. Bubble Sort (Recursive and Iterative)

1. **Recursive**

```
#include <iostream>
using namespace std;

// Bubble sort function
void bubbleSortRecurvise(int arr[], int n) {
   // Base case: if the size is already 1, it means it is already sorted
   if (n == 1)
      return;

   // One pass bubble sort
   for (int i = 0; i < n - 1; i++) {
      if (arr[i] > arr[i + 1]) {
         // Exchange
         int temp = arr[i];
         arr[i] = arr[i + 1];
         arr[i + 1] = temp;
      }
   }

   // Recursively calls the function for the remainder of the array
   bubbleSortRecursive(arr, n - 1);
}
int main() {
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);

   cout << "Data before sorting: ";
   for (int i = 0; i < n; i++)
      cout << arr[i] << " ";

   bubbleSortRecursive(arr, n);
   cout << "\nData after sorting (Bubble Sort Recursive): ";
   for (int i = 0; i < n; i++)
      cout << arr[i] << " ";

   return 0;
}
```

This Bubble Sort program that uses recursion operates by running one bubble process each time the function is called. In the bubble process, the elements in the array are compared sequentially and swapped if they are not in order. From this one cycle, the largest element will move to the end of the array. After that, the function will call itself by reducing the size of the array, because the end is already sorted

correctly and does not need to be processed again. This recursive process will continue to perform bubbles and reduce the area that needs to be sorted until only one element remains in the array. At that point, the process will stop and all data will be properly sorted.

Test Ke	Data Count	Time (µs)	Memory (kilobyte)
1	100 data	120	±31000
2	1000 data	14.500	±31000
3	10000 data	1.620.000	±31000
Minimum	–	120	31000
maximum	–	1.620.000	31000
Average	–	545.540	31000

Fig. 2: Recursive Bubble Sort Test Table

2. Iterative

```
[13:39, 26/11/2025] salma bisdig: #include <iostream>
using namespace std;

// Iterative Bubble Sort Function
void bubbleSortIteratif(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        // At each iteration, the largest element will move to the end position.
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap elements
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Data before sorting: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";

    bubbleSortIteratif(arr, n);

    cout << "\nData after sorting (…
[13:42, 26/11/2025] salma bisdig: #include <iostream>
#include <vector>

using namespace std;

void insertionSortRecursive(vector<int>& arr, int n) {
    // Recursion base: if n <= 1, the array is already sorted
    if (n <= 1) {
        return;
    }

    // Sort the subarray arr[0..n-2] recursively
    insertionSortRecursive(arr, n - 1);

    // Insert the element arr[n-1] into the correct position in arr[0..n-2]
    int key = arr[n - 1];
    int j = n - 2;
    while (j >= 0 && arr[j] > key) {
```

```
        arr[j + 1] = arr[j];
        j--;
    }
    arr[j + 1] = key;
}

int main() {
    vector<int> arr = {12, 11, 13, 5, 6};
    cout << "Array before sorting: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    insertionSortRecursive(arr, arr.size());…
[13:46, 26/11/2025] salma bisdig: #include <iostream>
#include <vector>
using namespace std;

void insertionSortIterative(vector<int>& arr) {
    int n = arr.size();

    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Shift array elements greater than the key
        // to a position one level higher
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        // Place the key in the correct position
        arr[j + 1] = key;
    }
}

int main() {
    vector<int> arr = {12, 11, 13, 5, 6};

    cout << “Array before sorting: ”;
    for (int x : arr) cout << x << " ";
    cout << endl;

    insertionSortIterative(arr);
    cout << “Array before sorting: ”;
    for (int x : arr) cout << x << " ";
    cout << endl;
```

This algorithm implements the sorting process using two layers of iteration. The outer iteration determines the number of passes required, while the inner iteration performs pairwise comparisons between each adjacent element. If an element is found to be larger than the element next to it, the two elements will swap positions. Through this mechanism, the largest value will gradually move like a bubble rising to the surface toward its final correct position. This process continues iteratively until no more swaps are needed, which indicates that all data has been sorted perfectly. At the end of execution, the program will display both data states, before and after the sorting process.

Test Ke	Data Count	Time (µs)	Memory (kilobyte)
1	100	120	±31000
2	1000	14.500	±31000
3	10000	1.620.000	±31000
Minimum	–	120	31000
Maximum	–	1.620.000	31000
Average	–	545.540	31000

Fig. 3: Table of Iterative Bubble Sort Test Results

3.2. Insertion Sort (Recursive and Iterative)

1. Recursive

```
#include <iostream>
#include <vector>

using namespace std;

void insertionSortRecursive(vector<int>& arr, int n) {
   // Recursion base: if n <= 1, the array is already sorted
   if (n <= 1) {
      return;
   }

   // Sort the subarray arr[0..n-2] recursively
   insertionSortRecursive(arr, n - 1);

   // Insert the element arr[n-1] into the correct position in arr[0..n-2]
   int key = arr[n - 1];
   int j = n - 2;
   while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j--;
   }
   arr[j + 1] = key;
}

int main() {
   vector<int> arr = {12, 11, 13, 5, 6};
   cout << "Array before sorting: ";
   for (int num : arr) {
      cout << num << " ";
   }
   cout << endl;

   insertionSortRecursive(arr, arr.size());

   cout << “Array after sorting: ”;
   for (int num : arr) {
      cout << num << " ";
   }
   cout << endl;

   return 0;
}
```

This recursive Insertion Sort program works by sorting the array step by step through function calls that decrease the size of the subarray. When the size of the subarray (n) is less than or equal to one, the function stops because it is considered to be sorted. Otherwise, the function calls itself with n-1 to first sort the previous part. After the initial subarray is sorted, the last element (arr[n-1]) is taken as the key and placed in its appropriate position by shifting the larger elements to the right. In the main function, the array is shown before and after the sorting process. This algorithm has a time complexity of $O(n^2)$ and requires O(n) space due to its recursive nature, making it less effective for large data sets, but useful for understanding recursive concepts in sorting.

Test Ke		Data Count	Time (μs)	Memory (Kilboyte)
1	100 data	1000 data	120	≈ 50
2	1000 data	1000 data	1.620.000	≈ 450
3	-	-	1.620.000	4.100
Maximum		-	-	4.100
Average		-	**545.540**	**1.533**

Fig. 4: Table of Recursive Insertion Sort Test Results

1. **Iterative**

```
#include <iostream>
#include <vector>
using namespace std;

void insertionSortIterative(vector<int>& arr) {
   int n = arr.size();

   for (int i = 1; i < n; i++) {
      int key = arr[i];
      int j = i - 1;

      // Shift array elements greater than the key
      // to a position one level higher
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j--;
      }

      // Place the key in the correct position
      arr[j + 1] = key;
   }
}

int main() {
   vector<int> arr = {12, 11, 13, 5, 6};

   cout << “Array before sorting: ”;
   for (int x : arr) cout << x << " ";
   cout << endl;

   insertionSortIterative(arr);

   cout << “Array before sorting: ”;
   for (int x : arr) cout << x << " ";
   cout << endl;
```

This program is an implementation of the Insertion Sort algorithm performed iteratively using the C++ language. The main purpose of this program is to sort the elements in a vector from lowest to highest. Sorting is performed by comparing each element starting from the second index with the previous elements. The element being analyzed is stored in a temporary variable called key, then the previous elements that are greater than key are shifted one position to the right. After finding a suitable place, the key value is placed in that position so that the part that has been passed becomes organized.

Gradually, this algorithm creates an ordered array section on the left side, while the elements on the right side are still in random order. This process continues until all elements in the array are correctly arranged. So, the function of this program is not only to sort data, but also to demonstrate the basic concept of the insertion algorithm, where elements are placed in the correct position through comparison and shifting. This program also displays the data before and after the sorting process, so that users can clearly see the changes in the data structure.

Time duration	Time (µs)	Memory (kilobyte)
100 data	80	± 31.000
1.000 data	7.500	± 31.000
10.000 data	510.000	± 31.000
Minimum	80	31.000
Maksimum	510.000	31.000
average	172.526	31.000

Fig.5: Table of Iterative Insertion Sort Test Results

3.3. Intro Sort (Recursive and Iterative)

1. **Recursive**

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
using namespace std;

class IntroSort {
private:
    void insertionSort(vector<int>& arr, int left, int right) {
        for (int i = left + 1; i <= right; i++) {
            int key = arr[i], j = i - 1;
            while (j >= left && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
    }

    int partition(vector<int>& arr, int low, int high) {
        int pivot = arr[high], 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;
    }

public:
    void introSortRecursive(vector<int>& arr, int start, int end, int depthLimit) {
        int size = end - start + 1;

        if (size < 16) {
            insertionSort(arr, start, end);
            return;
        }

        if (depthLimit == 0) {
            sort(arr.begin() + start, arr.begin() + end + 1);
            return;
        }

        int pivot = partition(arr, start, end);
        introSortRecursive(arr, start, pivot - 1, depthLimit - 1);
        introSortRecursive(arr, pivot + 1, end, depthLimit - 1);
    }
```

```
    void sort(vector<int>& arr) {
        if (arr.size() > 1) {
            int depthLimit = 2 * log2(arr.size());
            introSortRecursive(arr, 0, arr.size() - 1, depthLimit);
        }
    }
};

// Testing
int main() {
    IntroSort sorter;
    vector<int> data = {64, 34, 25, 12, 22, 11, 90, 5, 77};

    cout << "Before: ";
    for (int x : data) cout << x << " ";

    sorter.sort(data);

    cout << "\nAfter: ";
    for (int x : data) cout << x << " ";

    return 0;
}
```

This program uses the intelligent Intro Sort algorithm, which combines three sorting methods. Like a driver who understands the terrain, the program knows when to shift gears: Insertion Sort for small data, Quick Sort for large data, and Heap Sort when recursion is too deep.

With the depthLimit monitoring system, the program prevents poor performance by switching strategies at the right time. This adaptive approach ensures optimal efficiency in various data conditions, like a skilled strategist who knows how to adjust tactics.

Test Ke	Data Count	Time (µs)	Memory (kilobyte)
1	100 data	45	±1500
2	1000 data	680	±31000
Minimum	10000 data	8.500	±4.100
Maximum	–	8.000	1.000
Average	–	3.075	1.533

Fig. 6: Table of Recursive Sort Test Results

2. Iterative

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stack>
#include <cmath>
using namespace std;

class IntroSortIterative {
private:
    void insertionSort(vector<int>& arr, int left, int right) {
        for (int i = left + 1; i <= right; i++) {
            int key = arr[i], j = i - 1;
            while (j >= left && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
```

```
    }

    int partition(vector<int>& arr, int low, int high) {
        int pivot = arr[high], 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;
    }

public:
    void sort(vector<int>& arr) {
        if (arr.size() <= 1) return;

        stack<pair<int, int>> taskStack;
        int depthLimit = 2 * log2(arr.size());
        taskStack.push({0, arr.size() - 1});

        while (!taskStack.empty()) {
            auto [start, end] = taskStack.top();
            taskStack.pop();
            int size = end - start + 1;

            if (size < 16) {
                insertionSort(arr, start, end);
                continue;
            }

            if (depthLimit <= 0) {
                sort(arr.begin() + start, arr.begin() + end + 1);
                continue;
            }

            int pivot = partition(arr, start, end);
            depthLimit--;

            if (pivot - 1 > start) taskStack.push({start, pivot - 1});
            if (pivot + 1 < end) taskStack.push({pivot + 1, end});
        }
    }
};

// Testing
int main() {
    IntroSortIterative sorter;
    vector<int> data = {64, 34, 25, 12, 22, 11, 90, 5, 77};

    cout << "Before: ";
    for (int x : data) cout << x << " ";

    sorter.sort(data);

    cout << "\nAfter: ";
    for (int x : data) cout << x << " ";

    return 0;
}
```

This program implements an iterative version of Intro Sort that integrates three sorting methods into one smart solution. Rather than relying on memory-intensive recursion, the program opts for a more efficient stack-based approach.

Like a driver who knows the road, this program knows when to switch methods: Insertion Sort for small datasets, Quick Sort for heavier tasks, and std::sort as a backup option in urgent situations. The depth limit acts as a reminder that prevents the process from taking too long.

With the stack as its guide, the program explores the data like an explorer who records every route without getting lost in recursion. As a result, it produces a sorting algorithm that is fast, memory-efficient, and capable of handling various types of data.

Test Ke		Data Count	Time (μs)	Memory (kilboyte)
1	100 data	1000 data	45	≈ 50
2	1000 data	1000 data	≈ 50	≈ 50
Minimum	-	-	8.500	50
Maximum	-	-	8.500	50
Average	-	-	3.075	50

Fig.7: Table of Intro Sort Approval Results

4. Conclusion

From the overall discussion of the Bubble Sort, Insertion Sort, and Intro Sort algorithms in recursive and iterative forms, it can be concluded that each algorithm has its own advantages and disadvantages, so its application must be tailored to the data processing needs. Bubble Sort is the simplest and easiest algorithm to understand, making it very suitable for use as an introduction to learning basic sorting concepts. However, due to its relatively high time complexity and inefficiency for large amounts of data, this algorithm is not recommended for cases that require high performance.

Insertion Sort emerges as a more efficient algorithm for small data sets or data that is already partially sorted. Its method of inserting elements into their correct positions allows the sorting process to be faster under certain conditions. In addition, its low memory usage is an added advantage for this algorithm. However, Insertion Sort still has limitations when dealing with completely random data or very large data sets.

The most prominent algorithm in this discussion is Intro Sort. By combining three methods, namely Quick Sort, Heap Sort, and Insertion Sort, Intro Sort can adjust its strategy to the existing data conditions. When recursion is too deep, this algorithm switches to Heap Sort to maintain efficiency, and when the data size is small, it uses Insertion Sort to make the process run faster. Test results show that Intro Sort, in both its recursive and iterative versions, provides faster, more stable, and more efficient performance than the two previous algorithms, especially on large datasets.

Overall, this study shows that there is no single algorithm that is best for all situations. Bubble Sort is suitable as a basis for learning, Insertion Sort is effective for small and nearly sorted data, while Intro Sort is the best choice for various data sizes due to its ability to combine all three methods. By understanding the characteristics of each algorithm, users can choose the most appropriate and optimal sorting method for their data processing needs.

5. Recommendations

1. It is recommended to test with larger and more diverse data sets to enable a more comprehensive analysis of the algorithm's performance.
2. Comparisons with other algorithms (e.g., Merge Sort, Tim Sort) should be added to broaden the scope of the research.
3. Testing should cover various data conditions (sorted, random, reversed) to assess the algorithm's robustness.
4. It is recommended to try implementing the algorithm in other programming languages to test the effect of language optimization on performance.
5. The research should be supplemented with memory, energy efficiency, and stability analyses to make it more complete.

References

[1] S. Zahwa, Nailah Dhina Amelia, Rizqi Nafila, Rosila Agustina Putri, and Imam Prayogo Pujiono, "Perbandingan Efisiensi Memori dan Waktu Komputasi pada Algoritma Rekursif dan Iteratif dalam Operasi Pengurutan di C++," *J. RESTIKOM Ris. Tek. Inform. dan Komput.*, vol. 7, no. 1, pp. 123–136, 2025, doi: 10.52005/restikom.v7i1.428.

[2] D. W. Suparta, "INF202 : Struktur Data Pengurutan (Sorting)".

[3] Y. A. Astuti, "Analisis Pengujian Data Algoritma Bubble Sort," *REMIK Ris. dan E-Jurnal Manaj. Inform. Komput.*, vol. 7, no. 3, pp. 1413–1420, 2023, [Online]. Available: https://polgan.ac.id/jurnal/index.php/remik/article/view/12594

[4] M. Irfan Ali, Rangga Dzikri Fardiarsyah, Lukman Shodik, Fadilah Zahra Dwi Kinanti, and Imam Prayogo Pujiono, "Analisis Komparatif Efisiensi Memori dan Waktu Komputasi pada 8 Algoritma Sorting menggunakan C++," *LogicLink*, vol. 2, no. 1, pp. 1–17, 2025, doi: 10.28918/logiclink.v2i1.10868.

[5] Y. Zheng and K. L. Tan, "Sorting on Byte-Addressable Storage: The Resurgence of Tree Structure," *Proc. VLDB Endow.*, vol. 17, no. 6, pp. 1487–1500, 2024, doi: 10.14778/3648160.3648185.

[6] O. R. L. Peters, "Pattern-defeating Quicksort," 2021, [Online]. Available: http://arxiv.org/abs/2106.05123

[7] D. Z. Vierdansyah, G. Al Ghafiqi, M. T. Dwi Putra, and D. Pradeka, "Comparison Analysis Of Bubble Sort And Insertion Sort Algorithm On The

Selection Of A Shop According To The Criteria," *J. Comput. Eng. Electron. Inf. Technol.*, vol. 2, no. 1, pp. 39–52, 2023, doi: 10.17509/coelite.v2i1.57091.

[8] N. Mahrozi and M. Faisal, "Analisis Perbandingan Kecepatan Algoritma Selection Sort Dan Bubble Sort," *J. Ilm. Sain dan Teknol.*, vol. 1, no. 2, pp. 89–98, 2023.

[9] R. Latifah, E. Arriyanti, and A. R. Hakim, "Perbandingan Efisiensi Kinerja Algoritma Bubble Sort dan Algoritma Selection Sort Pada Parallel Programming," pp. 1–78, 2021, [Online]. Available: http://repository.untag-smd.ac.id/10/1/04. ABSTRAK RAHMI LATIFAH.pdf

[10] R. P. Aryanto, A. Nilogiri, and A. E. Wardoyo, "Optimasi Pengurutan Data Bilangan dengan Menggabungkan Algoritma Selection Sort Hybrid dan Bucket Sort," *Edumatic J. Pendidik. Inform.*, vol. 7, no. 1, pp. 39–48, 2023, doi: 10.29408/edumatic.v7i1.12358.

[11] P. Dymora and A. Paszkiewicz, "Performance analysis of selected programming languages in the context of supporting decision-making processes for industry 4.0," *Appl. Sci.*, vol. 10, no. 23, pp. 1–17, 2020, doi: 10.3390/app10238521.

[12] P. Rysak, "Comparative analysis of C and Python on the basis of the execution time of applications implementing selected algorithms," vol. 26, no. October 2022, pp. 93–99, 2023.

[13] A. A. Blg and A. M. Alasoud, "a Comparison Analysis Between the C++ and Python Programming Languages," *J. Appl. Sci.*, vol. 33, no. 1, pp. 20–32, 2020.

[14] A. Yasir and M. Eka, "Analisis Perbandingan Performa Bahasa Pemrograman Populer dalam Pengembangan Aplikasi Desktop," vol. 1, no. 1, pp. 29–34, 2025, [Online]. Available: https://journals.raskhamedia.or.id/index.php/juiktiDOI:https://doi.org/99.9999/juikti.vxix.xxxx

[15] M. Sengupta, "Reclaiming Performance : The Strategic Role of C ++ in High-Volume Financial Transaction Systems," vol. 11, no. 4, pp. 8167–8173, 2025, doi: 10.22399/ijcesen.4202.

[16] H. Al Ghifari, "Penggunaan algoritma bubble sort pada Bahasa pemprograman Java," vol. 3, pp. 681–687, 2025.

[17] N. Sari, W. A. Gunawan, P. K. Sari, I. Zikri, and A. Syahputra, "Analisis Algoritma Bubble Sort Secara Ascending Dan Descending Serta Implementasinya Dengan Menggunakan Bahasa Pemrograman Java," *ADI Bisnis Digit. Interdisiplin J.*, vol. 3, no. 1, pp. 16–23, 2022, doi: 10.34306/abdi.v3i1.625.

[18] P. D. Setyorini, L. T. Azzahro, R. A. Yuniar, and I. P. Pujiono, "Pengembangan Alat Bantu Pembelajaran Sorting Algorithm Berbasis Visual Console C++," *J. Ris. dan Apl. Mhs. Inform.*, vol. 6, no. 03, pp. 685–694, 2025, doi: 10.30998/jrami.v6i03.13983.

[19] A. Yusuf and Y. Ramadhani, "Analisis Algoritma Bubble Sort Ascending/Descending dan Implementasinya Menggunakan Bahasa Pemrograman Python," *JISCO J. Inf. Syst. Comput.*, vol. 2, no. 2, pp. 53–57, 2024, [Online]. Available: https://jurnal.fst.uinjambi.ac.id/index.php/jisco/article/view/102

[20] C. Kerja, "Kelebihan dan Kekurangan Insertion," no. February, 2024.

[21] B. A. Qowy, A. M. Chalik, F. Hanafi, A. Sundawijaya, and P. Studi, "Januari 2022 hal 60-63 Ilmu Komputer STIMIK ESQ Jakarta," *Jl. Tb. Simatupang Kav*, vol. 5, no. 1, pp. 60–64, 2022, [Online]. Available: https://jurnal.ugp.ac.id/index.php/jutei

[22] A. H. Yunial, "Analisa Perbandingan Algoritma Bubble Sort dan Perbandingan Eksponensial," *J. Inform. Univ. Pamulang*, vol. 10, no. 1, pp. 7–14, 2025.

[23] K. K. Gupta, M. R. Beg, and J. K. Niranjan, "A Novel Approach to Fast Image Filtering Algorithm of Infrared Images based on Intro Sort Algorithm," *J. Comput. Sci. Issues*, vol. 8, no. 6, pp. 235–241, 2011, [Online]. Available: http://www.ijcsi.org/papers/IJCSI-8-6-1-235-241.pdf