

**Department of Electrical and Computer Engineering**

**North South University**

**CSE225**

**Assignment 1 Report**

**Performance Comparison of Sorting Algorithms**

***Submitted By***

Saif Mohammed

2121913642

***Course Details***

CSE225 (Section 12)

Data Structures and Algorithms

***Submitted To***

Dr. Mohammad Rezwanul Huq

Associate Professor (Part-time) (MRH1)

Department of Electrical and Computer Engineering

North South University

***Submission Date***

September 5, 2023

**INDEXES**

**Section-1: Pseudocodes -------------------------------------------------- 3**

**Section-2: Performance Comparison Table ------------------------ 5**

**Section 3: Performance and Evaluation Graphs ------------------ 6**

**Section 4: Discussion and Conclusion ------------------------------- 9**

**Performance Comparison of Sorting Algorithms**

1. **Pseudocodes**

**Bubble Sort Pseudocode**

void bubbleSort(int\* dataSet, int size) {

int temp = 0;

for (int i = 0; i < size; i++) {

for (int j = 0; j < size - i - 1; j++) {

if (dataSet[j] > dataSet[j + 1]) {

temp = dataSet[j];

dataSet[j] = dataSet[j + 1];

dataSet[j + 1] = temp;

}

}

}

}

**Selection Sort Pseudocode**

void selectionSort(int\* dataSet, int size) {

for (int i = 0; i < size - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < size; j++) {

if (dataSet[j] < dataSet[minIndex]) {

minIndex = j;

}

}

swap(dataSet[i], dataSet[minIndex]);

}

}

**Insertion Sort Pseudocode**

void insertionSort(int\* dataSet, int size) {

for (int i = 1; i < size; i++) {

int current = dataSet[i];

int j = i - 1;

while (j >= 0 && dataSet[j] > current) {

dataSet[j + 1] = dataSet[j];

j--;

}

dataSet[j + 1] = current;

}

}

**Merge Sort Pseudocode**

void merge(int \*dataSet, int low, int mid, int high) {

int i, j, k;

int lengthLeft = mid - low + 1;

int lengthRight = high - mid;

int arrLeft[lengthLeft], arrRight[lengthRight];

for (int i= 0; i < lengthLeft; i++) {

arrLeft[i] = dataSet[low + i];

}

for (int i = 0; i < lengthRight; i++) {

arrRight[i] = dataSet[mid + 1 + i];

}

i = 0

j = 0;

k = low;

while (i < lengthLeft && j < lengthRight) {

if (arrLeft[i] <= arrRight[j]) {

dataSet[k] = arrLeft[i];

i++;

} else {

dataSet[k] = arrRight[j];

j++;

}

k++;

}

while (i < lengthLeft) {

dataSet[k] = arrLeft[i];

k++;

i++;

}

while (j < lengthRight) {

dataSet[k] = arrRight[j];

k++;

j++;

}

}

void mergeSort(int \*dataSet, int low, int high) {

int mid;

if (low < high) {

mid = (low + high) / 2;

mergeSort(dataSet, low, mid);

mergeSort(dataSet, mid + 1, high);

merge(dataSet, low, mid, high);

}

}

1. **Performance Comparison Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Size | Bubble Sort | Selection Sort | Insertion Sort | Merge Sort |
| 100000 | 19s | 7s | 3s | 0s |
| 200000 | 80s | 25s | 14s | 0s |
| 300000 | 182 | 57s | 32s | 0s |
| 400000 | 323s | 102s | 56s | 1s |

1. **Performance Evaluation Graph**

1. **Discussion and Conclusion**

In this assignment, we conducted a comparative analysis of four sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, and Merge Sort, and assessed their performance on different dataset sizes. The findings from our analysis provide valuable insights into the suitability of these algorithms for various scenarios.

Bubble Sort is a straightforward sorting algorithm, but it exhibits poor performance, especially for larger datasets. With its quadratic time complexity (O(n^2)), it is not recommended for sorting large datasets due to its inefficiency.

Selection Sort, while an improvement over Bubble Sort, also has a time complexity of O(n^2). It is more efficient than Bubble Sort but still not ideal for larger datasets. Selection Sort works by repeatedly finding the minimum element and moving it to the beginning of the unsorted portion of the dataset.

Insertion Sort, on the other hand, performs better than both Bubble Sort and Selection Sort, particularly when the dataset is partially sorted. Its time complexity is also O(n^2), but it has a more favorable performance profile for smaller datasets. It works by iteratively inserting elements into their correct positions within the sorted portion of the dataset.

Merge Sort, the most efficient among the algorithms we tested, demonstrated consistent and superior performance. With a time complexity of O(n log n), Merge Sort outperforms the other sorting algorithms, especially when dealing with larger datasets. It divides the dataset into smaller sub-arrays, sorts them individually, and then merges them to produce the final sorted result.

The selection of a sorting algorithm should be based on the specific characteristics of the dataset and the performance requirements of the task. For larger datasets, the efficiency of the algorithm becomes paramount. Bubble Sort and Selection Sort, while conceptually simple, are not suitable for sorting large datasets due to their quadratic time complexities. Their use should be limited to small datasets or educational purposes. Insertion Sort can be a viable option when dealing with partially sorted datasets or small datasets where its simplicity is an advantage. However, its performance degrades for larger datasets. Merge Sort emerges as the top-performing algorithm among those we tested. Its efficiency with a time complexity of O(n log n) makes it the preferred choice for sorting large datasets, and it consistently produces sorted results. The nature of the data, whether partially sorted or completely random, should influence the choice of sorting algorithm. Merge Sort's stability and efficiency make it a versatile choice.

In conclusion, the choice of a sorting algorithm should be made carefully, taking into account the size of the dataset and the desired performance characteristics. Merge Sort stands out as the recommended choice for sorting large datasets efficiently, while Bubble Sort, Selection Sort, and Insertion Sort are better suited for specific use cases or educational purposes. Understanding the strengths and weaknesses of sorting algorithms is essential for making informed decisions in real-world applications.