Sorting Algorithms - A Comparative Study

65070501039 B. Pongpon, 65070501055 U. Sorrawit, 65070501069 D. Kanitsorn,
65070501083 S. Panurut, 65070501092 B. Saranunt

Department of Computer Engineering, Faculty of Engineering
King Mongkut's University of Technology Thonburi, Bangkok, Thailand

November 30, 2023

Abstract

The experiment depicts a comparative study of five sorting algorithms, aiming to illustrate the behavior of sorting algorithms and serve as a guideline for selecting the most efficient algorithm based on the characteristics of real-world data. The tests were conducted in a controlled environment with large datasets—unsorted unique, normal distribution, and sorted unique—to measure CPU runtimes and illustrate the growth characteristics of each algorithm. Our study positions Quicksort as the primary choice for handling large datasets. However, when considering data with low standard deviation, Mergesort demonstrated superiority in scenarios with low variability among datapoints, i.e., scenarios with a significant number of identical values. For highly sorted arrays, Insertionsort or Mergesort may offer greater efficiency when adding new datapoints to a sorted array. With our testing method, this analysis can be directly referred to for the selection of sorting algorithms based on real-world data statistics—size and standard deviation. However, the unexpected behavior, namely Branch Prediction, is not considered in this study.

Keywords: Sorting Algorithms, Comparison Sort, Time Complexity, Measuring CPU Runtime, Sorting in normal distributed data

1. Introduction

Sorting algorithms are commonly found and play fundamental roles in numerous computational processes [4]. This paper presents a comparative study of five sorting algorithms—Bubblesort, Insertionsort, Selectionsort, Mergesort, and Quicksort—with the aim of illustrating the growth characteristics and efficiency of each algorithm through CPU runtime measurements in a controlled environment. These algorithms were implemented in the C language [5] and executed in the Ubuntu local environment.

Performance measurements were conducted on four cases: unsorted unique data, providing a general average measurement for each algorithm relative to the size of the array; normal distributed data, commonly encountered in real-world data collection, used to test CPU runtime influenced by two factors—array size and standard deviation; and sorted unique data, employed to identify the best case for some algorithms and to evaluate scenarios where a small portion of new data is added to an already sorted array. Our main goal is to provide a practical guideline for sorting algorithm selection based on the statistical characteristics of real-world datasets, namely size and variability.

2. Literature Review

Upon surveying papers related to the Comparative Study of Sorting Algorithms [1], we have identified fundamental aspects for our work, specifically the testing environment and the size of the data. Details on these aspects will be elaborated in methodology.

However, upon review, we have noticed a lack of test cases in past studies, potentially leading to narrow conclusions in comparative studies. To conduct an analysis in a broader context, we conduct a review on the characteristics of real-world data[3] to comprehensive evaluation of the sorting algorithms.

2.1 Sorting Algorithms

a. Bubble Sort

The bubble Sort Algorithm, invented by Edward Harry Friend, is the simplest brute-force sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order until every entry in the list is sorted. Bubble sort algorithm got its name from its behavior that element in the wrong place **"Bubbling Up"** to the correct place.[2][6]

Algorithm Bubble Sort

1: **Input:** An array A[1...n]
2: **Output:** An array A[1...n] sorted in ascending order
3:
4: **for** i = 1 to $n-1$ **do**
5: **for** i = 1 to $n-i$ **do**
6: **if** A[j+1] < A[j] **then**
7: swap(a_j,a_j+1)

b. Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time, known for being one of the earliest sorting algorithms devised and served as a foundation for more advanced sorting techniques. The algorithm is based on the idea of dividing the array into a sorted and an unsorted region and iteratively inserting elements from the unsorted region into their correct position in the sorted region. Despite its simplicity, insertion sort has been widely studied and employed in various contexts.[4][6]

Algorithm Insertion Sort

1: **Input:** **A** list a_1,a_2,...,a_n
2: **Output:** **A** list a_1,a_2,...,a_n sorted
3:
4: **for** i = 2 to n **do**
5: v $\leftarrow a_i$
6: $j \leftarrow i-1$
7: **while** $j \geq 1$ and a_j < v **do**
8: $a_j+1 \leftarrow a_j$
9: $j \leftarrow j-1$
10: $a_j+1 \leftarrow v$

c. Selection Sort

Selection Sort Algorithm, one of the first algorithms devised for sorting purposes, works by repeatedly selecting the smallest or largest element from the unsorted portion of the list and moving that element to the sorted portion of the list.[6]

Algorithm Selection Sort

1: **Input:** An array A[1...n]
2: **Output:** An array A[1...n] sorted in ascending order
3:
4: **for** i = 1 to $n-1$ **do**
5: min $\leftarrow$ i
6: **for** $j = i+1$ to n **do**
7: **if** A[j] < A[min] **then**
8: min $\leftarrow$ j
9: swap(A[i],A[min]);

d. Merge Sort

Merge sort algorithms, invented in 1945 by John von Neumann, is a divide-and-conquer algorithm that recursively divides the array into halves, sorts each half, and then merges the sorted halves. It boasts a stable time complexity of O(n log n) in the worst, average, and best cases, making it more efficient for large datasets compared to the quadratic time complexities of Bubble, Insertion, and Selection sorts. Its drawback is the additional memory requirement for the merging process.[6]

Note : This algorithm uses with merge sort to merge two arrays to main sorted array

Algorithm Merge(A,l,m,r)

1: **Input:** An array A[l...r] divided into two partition which are A[l...m] and A[m+1...r]
2: **Output:** Sorted array A[l...r]
3:
4: n_1 size of A[l...m]
5: n_2 size of A[m+1...r]
6: i,j,k $\leftarrow$ 0
7:
8: **while** i < n_1 and j < n_2 **do**
9: **if** A[l...i] < A[m+1+j] **then**
10: B[k] $\leftarrow$ A[l+i]
11: i $\leftarrow$ i+1
12: **else**
13: B[k] $\leftarrow$ A[m+1+j]
14: j $\leftarrow$ j+1
15: k $\leftarrow$ k+1
16: **if** i = n_1 **then**
17: Copy A[j...n_2-1] to B[k...n_1+n_2-1]
18: **else**
19: Copy A[j...n_1-1] to B[k...n_1+n_2-1]
20: Copy B to A[l...r]

Note : This is the Recursive version of merge sort that is used in this experiment

Algorithm MergeSort

1: **Input:** An array A[l...r] of orderable element
2: **Output:** An array A[l...r] sorted in non-decreasing order
3:
4: **if** A has more than one element **then**
5: m $\leftarrow \lfloor$(l+r)/2$\rfloor$
6: MergeSort(A[l...m])
7: MergeSort(A[m+1...r])
8: Merge(A,l,m,r)

e. Quick Sort

Quicksort, firstly described by Tony Hoare, employs a divide-and-conquer strategy. The algorithm selects a "pivot" element from the array and partitions the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The process is then applied recursively to the sub-arrays.[6]

Algorithm HoarePartition(A[l...r])

1: **Input:** Array A[l...r] of orderable elements
2: **Output:** A partition of A[l...r] with the split position as return value
3:
4: p ← A[l]
5: s ← l
6: j ← r + 1
7: **repeat**
8: **repeat** i ← i + 1 **until** A[i] ≥ p
9: **repeat** j ← j − 1 **until** A[i] ≤ p
10: swap(A[i],A[j])
11: **until** i ≥ j 12: swap(A[i],A[j])
13: swap(A[l],A[j])
14: **Return** j

Algorithm QuickSort

1: **Input:** An array A[l...r] of orderable element
2: **Output:** An array A[l...r] sorted in non-decreasing order
3:
4: **if** l < r **then**
5: s ← Partition(A[l...r])
6: QuickSort(A[l...s-1)
7: QuickSort(A[s+1...r])

f. Algorithms Time Complexity

Algorithm	Time Complexity		
	Best	Average	Worse
Bubble Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(nlog(n))$	$\Theta(nlog(n))$	$O(nlog(n))$
Quick Sort	$\Omega(nlog(n))$	$\Theta(nlog(n))$	$O(n^2)$

Table 1: Sorting Algorithm Growth function [7]

3. Methodology

This study aims to compare the efficiency of five sorting algorithms - Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, and Quick Sort in C programming language.

3.1 Test Environment

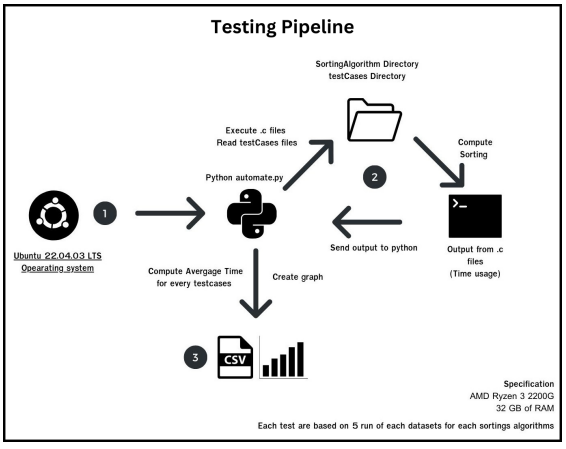


Figure 1: Test-bench Architecture

To ensure fairness of comparison between the algorithms we implemented them on a computer using the C programming language with the purpose of finding the efficiency of five sorting algorithms on the Ubuntu 22.04.3 LTS Desktop (64-bit) operating system. Python module is being used for testing process automation, CPU runtime measurement, and data visualization.
- computer specifications

- AMD Ryzen™ 3 2200G @ 3.5 GHz base clock
- 32 Gigabytes of RAM @ 2666 MHz bus speed
- NVIDIA® GeForce® GTX 950 2GB GDDR5

- compiler specifications

- gcc (Ubuntu 11.4.0-1ubuntu1˜ 22.04)

In the program, we use a library named "time.h" and use a function named "clock" to get the number of clock ticks elapsed since the program was launched to get the number of seconds used by the CPU, you will need to divide by CLOCKS_PER_SEC.

3.2 Test Data

3.2.1 Dataset Selection

One of our goals is to conduct an analysis in a broader context. The Test cases have been generated based on the characteristics of real-world data[3] to ensuring a comprehensive evaluation of the sorting algorithms. The dataset are created using Numpy in order to config the statistical properties; size, average, and standard deviation. The following dataset types will be used in testing.

a) Random-Ordered Unique Data:
This dataset type comprises randomly generated unique values. Purpose: Evaluate the algorithms' performance on unsorted, distinct datasets, simulating scenarios where data has no inherent order.

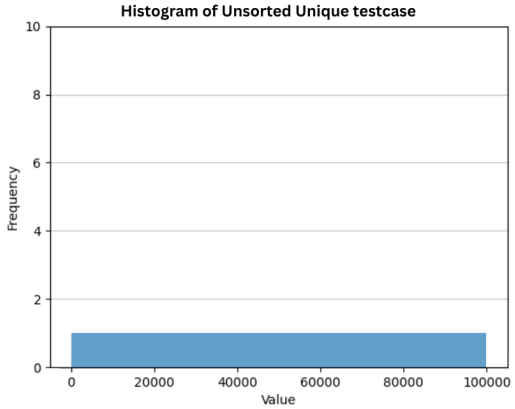


Figure 2: Distribution of Unsorted Unique Test set

b) Random Normal Distribution Data:
This dataset type includes randomly generated data following a normal distribution. Purpose: Mimic real-world scenarios[3] where data often conforms to a common statistical distribution, allowing for an evaluation of the algorithms in such contexts.

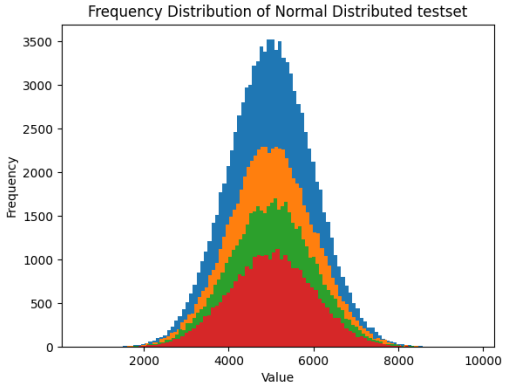


Figure 3: Distribution of Normal Distributed Testcase

c) Normal Distribution Data with Varying Standard Deviations:
Multiple datasets with normal distributions will be generated, varying in standard deviation. Purpose: Assess how the algorithms respond to datasets with different levels of variability. Lower standard deviations represent datasets with less uniqueness, while higher standard deviations introduce more variability.

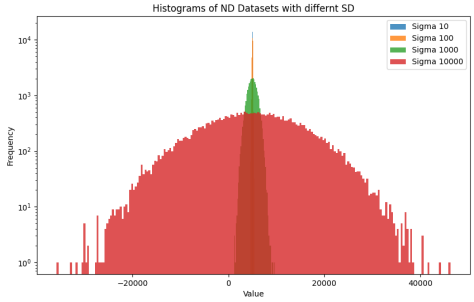
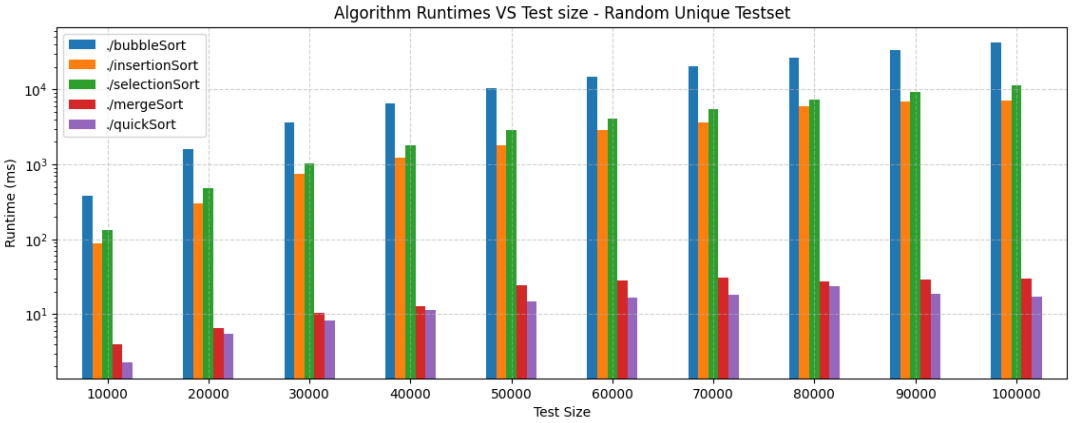


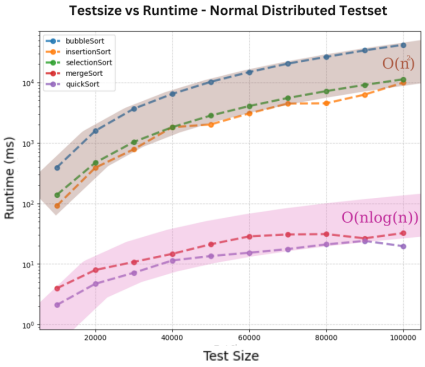
Figure 4: Distribution of Testcase varying by SD

d) Sorted Data Datasets are pre-sorted in ascending order will be utilized. Purpose:
Investigate the behavior of algorithms when the input data is already partially or fully sorted, providing insights into their adaptability in scenarios where data is close to its final state.

4. Result & Discussion

4.1 Performance on Unsorted Unique Testcase





Test Size	Algorithm				
	Bubble	Insert	Select	Merge	Quick
10,000	386.4	87.0	133.6	3.9	2.2
20,000	1581.2	306.1	479.8	6.5	5.5
30,000	3620.6	752.6	1041.3	10.3	8.2
40,000	6587.3	1231.6	1827.5	12.8	11.3
50,000	10371.2	1792.2	2845.2	24.3	14.9
60,000	14893.1	2893.8	4061.1	27.9	16.9
70,000	20748.2	3634.2	5538.9	30.5	18.3
80,000	26497.7	6061.0	7244.0	27.0	23.8
90,000	33803.4	6960.0	9166.9	29.4	19.0
100,000	42042.7	7078.7	11303.3	30.1	17.2

Figure 5: Performance Analysis on Unsorted Unique Testcase. [top] a bar graph illustrating the relationship between runtime and dataset size; [bottom-left] the growth characteristics of each algorithm; [bottom-right] Raw test data of runtime vs array size.

In our investigation of sorting algorithms using unsorted unique datasets of varying sizes, we sought to illustrate the average runtime efficiency of each algorithm on logarithmic scale. [Figure 5] highlights the superiority of divide and conquer algorithms, specifically merge sort and quicksort, in handling large datasets. Quicksort demonstrated optimal performance, completing the sorting of 100,000 unique elements in approximately 17 milliseconds. Conversely, the general approach algorithms, namely Bubblesort, insertionsort, and selectionsort, exhibited significantly higher runtimes in random unique datasets.

Upon behavioral observation, algorithms with quadratic time complexity, such as Bubblesort, insertionsort, and selectionsort, demonstrated a growth behavior proportional to n-square, as depicted in the brown area of [Figure 5]. In contrast, divide and conquer algorithms, mergesort and quicksort, displayed a runtime growth trend consistent with nlog(n). This behavior was especially evident in medium-sized array sorting scenarios, suggesting the need for larger arrays to further explore the performance of divide and conquer algorithms in future experiments.

4.2 Performance on Normal Distributed Testcase

Test Size	Algorithm				
	Bubble	Insert	Select	Merge	Quick
10,000	392.4	91.2	138.6	3.9	2.1
20,000	1587.6	389.7	471.7	7.9	4.7
30,000	3676.7	787.3	1039.4	10.8	7.1
40,000	6473.9	1823.5	1824.2	14.7	11.4
50,000	10255.0	2025.4	2837.5	21.2	13.5
60,000	14880.0	3082.4	4078.8	28.6	15.3
70,000	20487.5	4482.0	5535.8	30.7	17.6
80,000	26522.4	4530.4	7203.8	31.2	21.0
90,000	33965.2	6305.2	9133.5	26.5	24.1
100,000	41971.1	9943.3	11276.5	32.3	19.7

Table 2: Performace on Normal Distributed

SD	Algorithm				
	Bubble	Insert	Select	Merge	Quick
10	3573.1	758.2	1032.7	9.7	49.5
100	3626.6	782.6	1040.7	11.6	6.2
1,000	3677.3	786.4	1037.1	11.7	5.6
10,000	3646.6	728.3	1040.3	11.3	5.6
100,000	3709.7	808.2	1035.4	9.5	4.5

Table 3: Performace on Normal Distributed based on SD

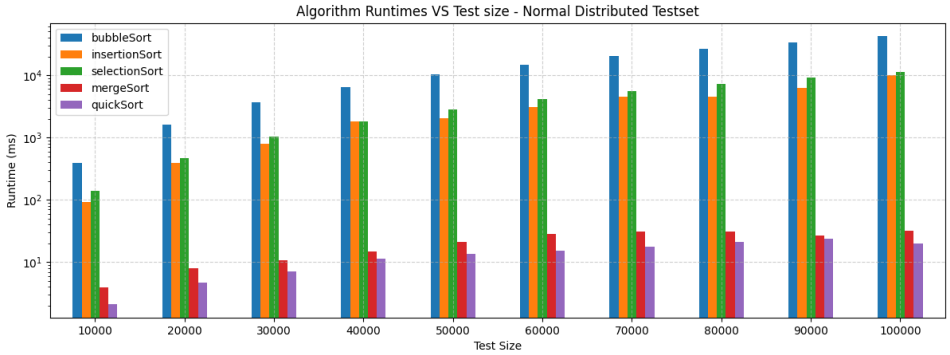


Figure 6: Chart Representing the Relationship Between Run-Time and Size in Normal Distributed Testcase

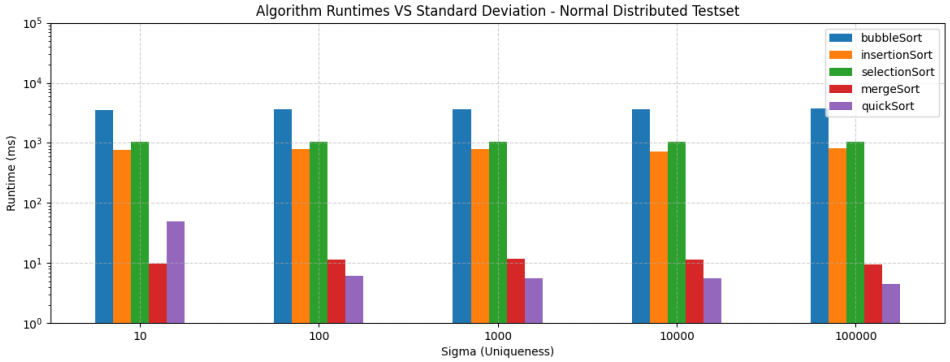
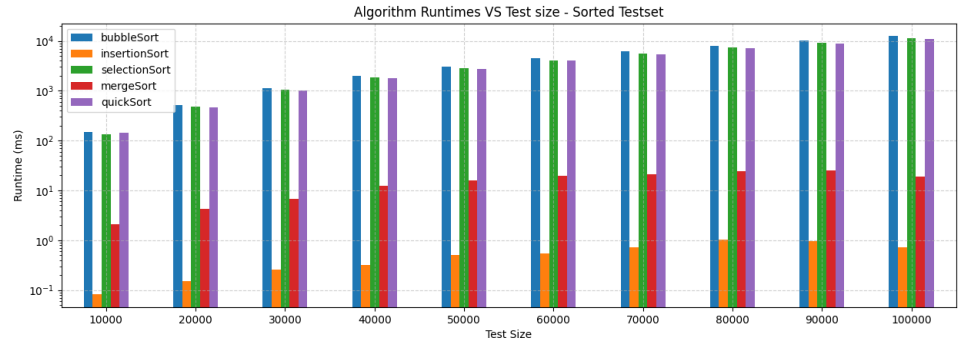


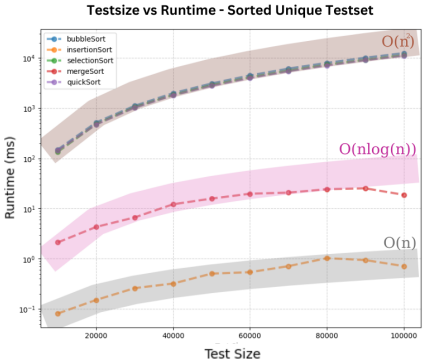
Figure 7: Chart Representing the Relationship Between Run-Time and Standard Deviation in Normal Distributed Testcase

The examination of sorting algorithms in normal distribution test cases, with varying sizes and standard deviations (σ), revealed interesting insights, as shown in [Figure 6]. For unsorted normal distributed test cases with $\mu = 5000$ and $\sigma = 1000$ across different sizes (ranging from 10,000 to 100,000), all sorting algorithms demonstrated average case runtime behavior, similar to unsorted unique datasets. Notably, merge sort and quicksort stood out with significantly lower runtimes as data size increased, following an $O(nlog(n))$ growth function. Conversely, Bubblesort, insertionsort, and selectionsort exhibited higher runtimes with a growth behavior of $O(n^2)$.

As shown in [Figure 7], In the investigation of varying standard deviations (σ) with a fixed mean ($\mu = 5000$) and size (30,000). Quicksort, employing Hoare Partition, emerged as the only algorithm influenced by standard deviation. Its efficiency improved as the standard deviation increased, indicating better performance with higher data variability and uniqueness. In contrast, other algorithms, including Bubblesort, insertionsort, selectionsort, and mergesort, showed no significant change in runtime as the standard deviation increased. This observation aligns with the understanding that standard deviation corresponds directly to data variance (σ^2), providing valuable insights into algorithm performance under different levels of data variability.

4.3 Performance on Sorted Unique Testcase





Test Size	Algorithm				
	Bubble	Insert	Select	Merge	Quick
10,000	149.5	0.1	134.0	2.1	143.3
20,000	518.6	0.2	479.5	4.3	471.9
30,000	1111.7	0.3	1043.3	6.7	1016.0
40,000	1985.5	0.3	1827.7	12.2	1805.6
50,000	3099.5	0.5	2871.5	15.8	2774.2
60,000	4474.6	0.5	4097.8	19.8	3988.6
70,000	6123.5	0.7	5589.6	20.8	5437.7
80,000	7967.8	1.0	7302.4	24.3	7094.8
90,000	10099.9	0.9	9167.6	25.3	8999.3
100,000	12462.7	0.7	11369.4	18.8	11070.4

Figure 8: Performance Analysis on Sorted Unique Testcase. [top] a bar graph illustrating the relationship between runtime and dataset size; [bottom-left] the growth characteristics of each algorithm; [bottom-right] raw test data of Runtimes vs Array size

In the context of sorted unique arrays, shown in [Figure 8], our findings revealed that Insertionsort outperformed other algorithms, completing the sorting of 100,000 elements in a mere 0.7 milliseconds. Theoretical considerations, such as the one-time traversal characteristic of Insertionsort in sorted arrays, supported its O(n) growth rate. Mergesort consistently exhibited a growth rate of $O(nlogn)$. On the other hand, Bubblesort, selectionsort, and quicksort displayed more runtimes with similar behavior with a growth rate of $O(n^2)$, despite the absence of a swap function in these algorithms.

4.4 Sorting Algorithm Selection Guideline

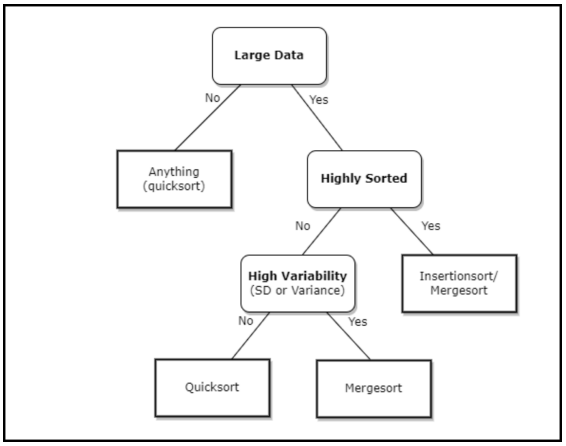


Figure 9: Algorithm Selection Guideline

The sorting algorithm selection guideline, as shown in [Figure 9], has been formulated based on our comparative study using a large test set. In datasets with fewer than 10,000 elements, runtimes do not exhibit significant differences. However, in most general cases, Quicksort is the preferable choice. For larger-sized arrays, Quicksort maintains its position as the primary algorithm in most scenarios. Notably, Merge-sort demonstrated superiority in situations with low variability among data points. Furthermore, for highly sorted arrays, Insertionsort or Mergesort may offer greater efficiency when adding new data points.

4.5 Discussion

a. While observing the growth characteristics of algorithms with O(nlog(n))—specifically Quicksort and Mergesort—we note that the test set within the range of 100,000 may not sufficiently depict a clear growth function for these algorithms.

b. The unexpected behavior, potentially identified as "branch prediction", could be the cause of unstable runtime measurements.

c. Due to an unidentified cause, memory consumption cannot be measured precisely in our testing environment.

5. Conclusion

In this experiment, we assessed the performance of Bubblesort, Insertionsort, Selectionsort, Mergesort, and Quicksort across Unsorted Unique, Normal Distributed (varied by size and standard deviation), and Sorted Unique test cases, mirroring real-world data characteristics. Implementing the algorithms in C, following CPE231 pseudocode, we measured CPU runtimes in a controlled environment.

The results consistently favored Quicksort with Hoare's partition, showcasing significantly lower runtimes across most scenarios. This positions Quicksort as the primary choice for handling large datasets. However, exceptionally, Mergesort demonstrated superiority in scenarios with low variability among datapoints, and for highly sorted arrays, Insertionsort or Mergesort may offer greater efficiency when adding new datapoints.

In practical terms, these findings provide guidance for algorithm selection based on specific data characteristics. Quicksort excels in diverse datasets, while Mergesort serves as an alternative for scenarios with lower variability. In cases of highly sorted arrays, Insertionsort or Mergesort may be more preferable. These insights underscore the importance of customizing algorithmic choices to the unique attributes of the data under consideration.

6. References

[1] Opeyemi Adesina. "A Comparative Study of Sorting Algorithms". In: *African Journal of Computing ICT* 6 (Dec. 2013), pp. 199–206.

[2] Khalid Alkharabsheh et al. "Review on Sorting Algorithms A Comparative Study". In: *International Journal of Computer Science and Security (IJCSS)* 7 (Jan. 2013).

[3] Kaine Black. *9 Important Data Distributions Real World Examples for Each*. MLearning.ai. Feb. 2022. URL: `https://medium.com/mlearning-ai/9-important-data-distributions-real-world-examples-for-each-b804d9d95fe7#1893` (visited on 11/16/2023).

[4] Thomas H Cormen et al. *Introduction to algorithms*. 4th ed. The Mit Press, 2022.

[5] George T Heineman, Stanley Selkow, and Gary Pollice. *Algorithms in a Nutshell*. 1st ed. O'reilly, 2009.

[6] Anany Levitin. *Introduction to the design and analysis of algorithms*. 3rd ed. Upper Saddle River, NJ: Pearson, Sept. 2011. Chap. 3.

[7] Eric Rowell. *Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell*. Bigocheatsheet.com. 2019. URL: `https://www.bigocheatsheet.com/`.