CS-700 Algorithms and Complexity

Assignment-1

ANAND M K
Roll Number: 242CS008

1 Introduction

In this assignment, I have analyzed and compared different sorting algorithms based on their execution time. The performance of these algorithms was compared with varying input sizes, and an in-depth analysis has been conducted.

The following sorting algorithms were considered for comparison:

- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Radix Sort

2 Data Generation and Experimental Setup

2.1 Input Data

Three different versions of input data were considered for each algorithm:

1. Elements sorted in increasing order.
2. Elements sorted in decreasing order.
3. Randomly generated elements.

The program can read the input from a file and execute the code. The format of the input files is as follows:

```
[n] //number of positive integers to sort
[element 1]
[element 2]
...
...
[element n]
```

Where n varies from 10,000 to 100,000, with intervals of 10,000. For each version of input data, there are 10 input files, each corresponding to a different input data size ranging from 10,000 to 100,000. This varied dataset allows for a more comprehensive analysis.

2.2 Machine Specifications

- CPU: 11th Gen Intel® Core™ i5-11500 @ 2.70GHz × 12
- RAM: 16 GiB
- Operating System: Ubuntu 22.04.4 LTS
- Compiler: gcc 11.4.0

2.3 Timing Mechanism

Execution time was calculated in milliseconds using the gettimeofday() function in C, which provides higher precision than the clock() function. The ¡sys/time.h¿ header file is required for gettimeofday(). The following code was used to calculate the execution time.

```
struct timeval start, end;
gettimeofday(&start, NULL);
Sorting Algorithm();
gettimeofday(&end, NULL);
double executiontime = (end.tv_sec - start.tv_sec) + ((end.
   tv_usec - start.tv_usec) / 1000000.0); // Execution time
   in seconds
executiontime = executiontime * 1000; // Execution time in
   milli seconds
```

2.4 Experiment Repetitions

Each experiment was repeated more than 10 times to verify that the measurements were valid and did not produce incorrect values.

2.5 Time Capture

Each sorting algorithm was executed with different input sizes, and the time was captured for each case. The time was calculated as the average of multiple repetitions of the experiments.

2.6 Input Generation and Selection

I have generated the input files as follows:

- **For the increasing order file:** Numbers were generated from 100,000 to 100,000 plus the size of the input file.
  Example: For an increasing order file with 20,000 numbers, the values will range from 100,000 to 120,000.

- **For the decreasing order file:** Numbers were generated from 200,000 to 200,000 minus the size of the input file.
  Example: For a decreasing order file with 20,000 numbers, the values will range from 200,000 to 180,000.

- **For the file with random values:** Numbers were generated using the `rand()` function in C, with values between 100,000 and 200,000.

2.7 Input Consistency: Were the same inputs used for all sorting algorithms?

Yes, the same set of inputs was used for all sorting algorithms.

3 Analysis of QuickSort

Three versions of QuickSort have been considered for analysis:

- **Version 1:** The first element in the array.
- **Version 2:** A random element in the array.
- **Version 3:** The median of the first, middle, and last elements in the array.

3.1 Version 1 : QuickSort with the Pivot as the First Element

- **Sorted List in Increasing Order:** Since the pivot is selected as the first element, each partitioning step is highly unbalanced. Therefore, the time complexity results in $O(n^2)$.
- **Sorted List in Decreasing Order:** Similar to the increasing order list, the pivot choice leads to highly unbalanced partitions. Thus, the time complexity also results in $O(n^2)$.
- **Random Order List:** In this case, the list is not sorted, so the pivot does not consistently produce unbalanced partitions. As a result, the algorithm performs better, and the time complexity is approximately $O(n \log n)$.

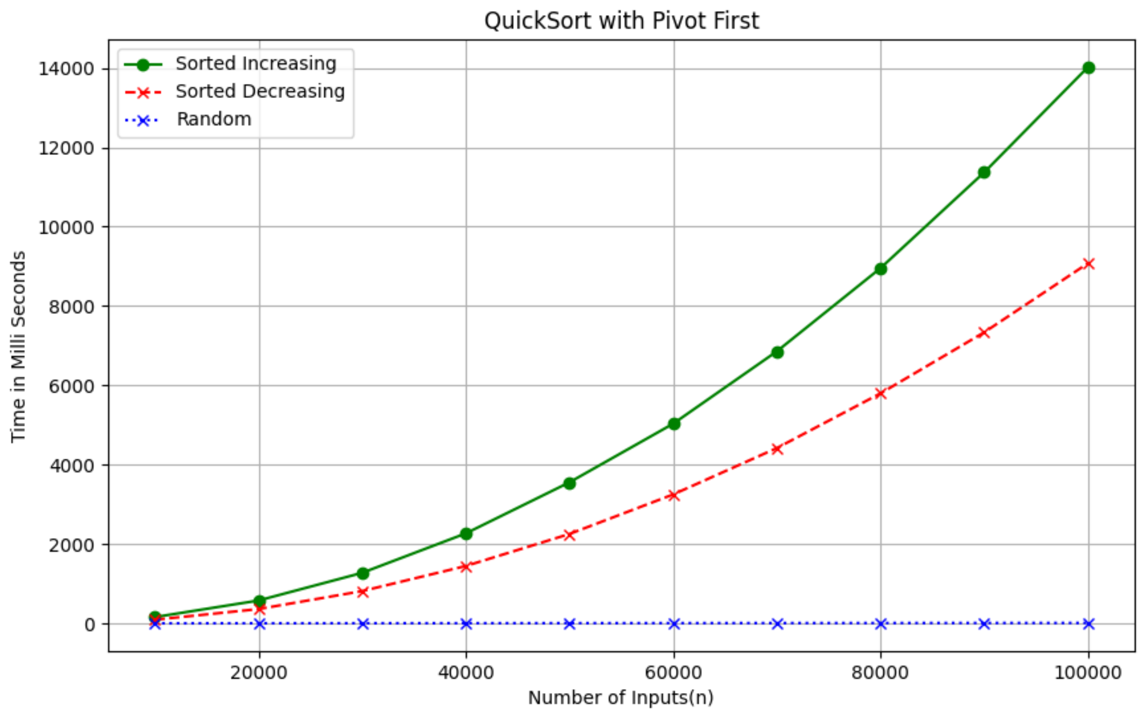

Figure 1: Performance of QuickSort with Pivot as the First Element

3.2 Version 2 : QuickSort with the Pivot as the Random Element

- **Sorted List in Increasing Order:** Since the pivot is chosen randomly, each partitioning step is less likely to be highly unbalanced. This results time complexity of $O(n \log n)$.
- **Sorted List in Decreasing Order:** Similar to the sorted increasing order list, the random pivot selection helps avoid consistently poor partitioning. Therefore, the time complexity is $O(n \log n)$.
- **Random Order List:** The random pivot selection generally provides balanced partitions and ensures good performance. The time complexity is $O(n \log n)$

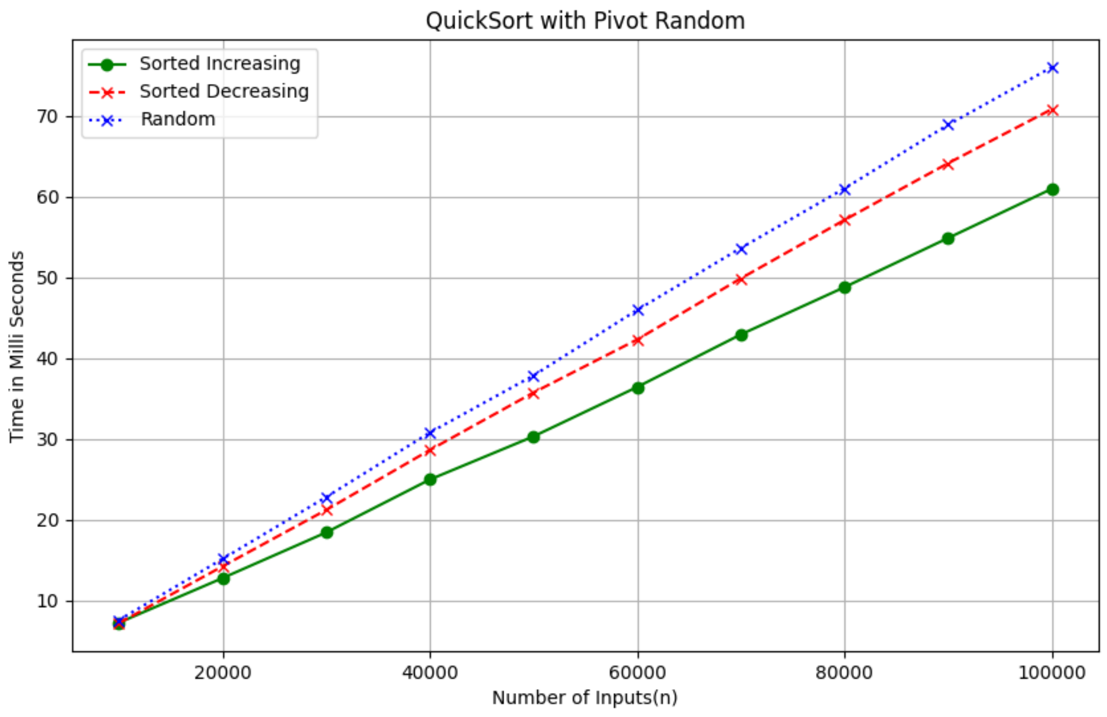

Figure 2: Performance of QuickSort with Random Pivot Choice

3.3 Version 3 : QuickSort with the Pivot as the Median of the First, Middle, and Last Elements

- **Sorted List in Increasing Order:** The median of the first, middle, and last elements as the pivot generally results in more balanced partitions, as it chooses a pivot more likely to be near the middle value. This results in an average time complexity of $O(n \log n)$.

- **Sorted List in Decreasing Order:** Similar to the increasing order list, the median as the pivot choice improves partitioning, and the time complexity results in $O(n \log n)$.

- **Random Order List:** The median pivot selection generally provides balanced partitions and ensures good performance. The time complexity is $O(n \log n)$.

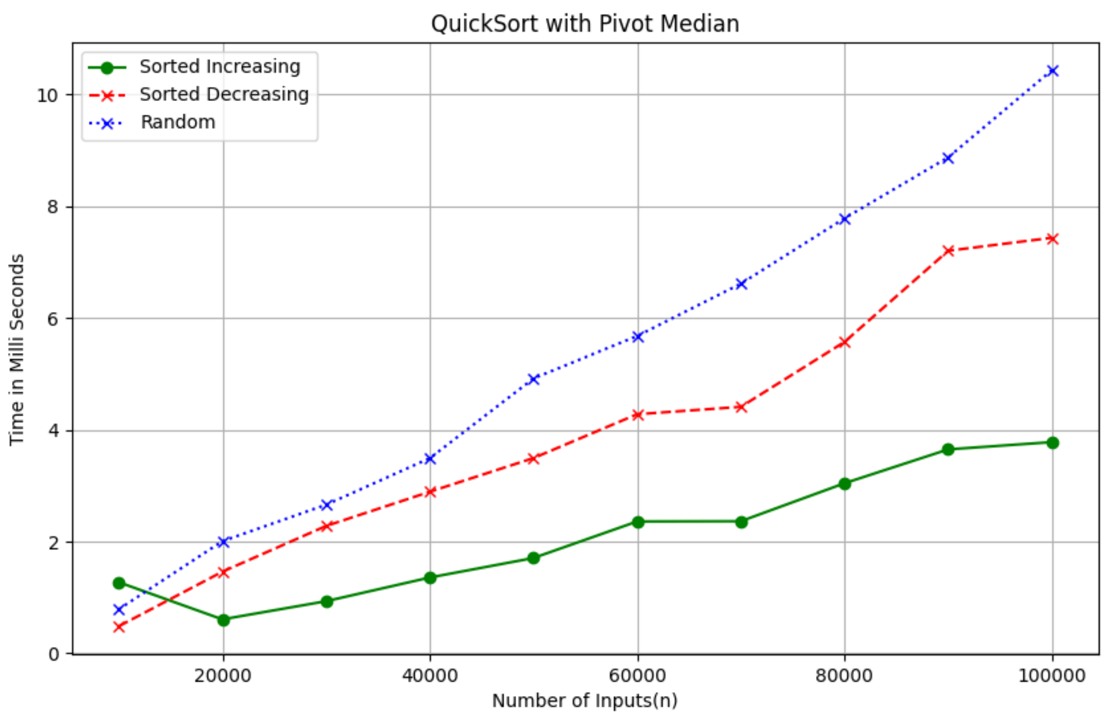

Figure 3: Performance of QuickSort with Pivot as the Median of the First, Middle, and Last Elements

3.4 Best Case Performance of Three Versions of Quick Sort

As per the above analysis, the best case occurs when the partitions are balanced, and there are fewer unbalanced partitions. This leads to a best-case time complexity of $O(n \log n)$ in all three versions.

- **Version 1:** As per the above experiments and observations, the random array gives the best case.
- **Version 2:** Even though all types of inputs result in $O(n \log n)$, considering the experiments conducted and the graph, the sorted array in increasing order gives the best times.
- **Version 3:** Even though all types of inputs result in $O(n \log n)$, considering the experiments conducted and the graph, the sorted array in increasing order gives the best times.

Following Figure-4 shows the best case running time for all the three versions of quick sort.

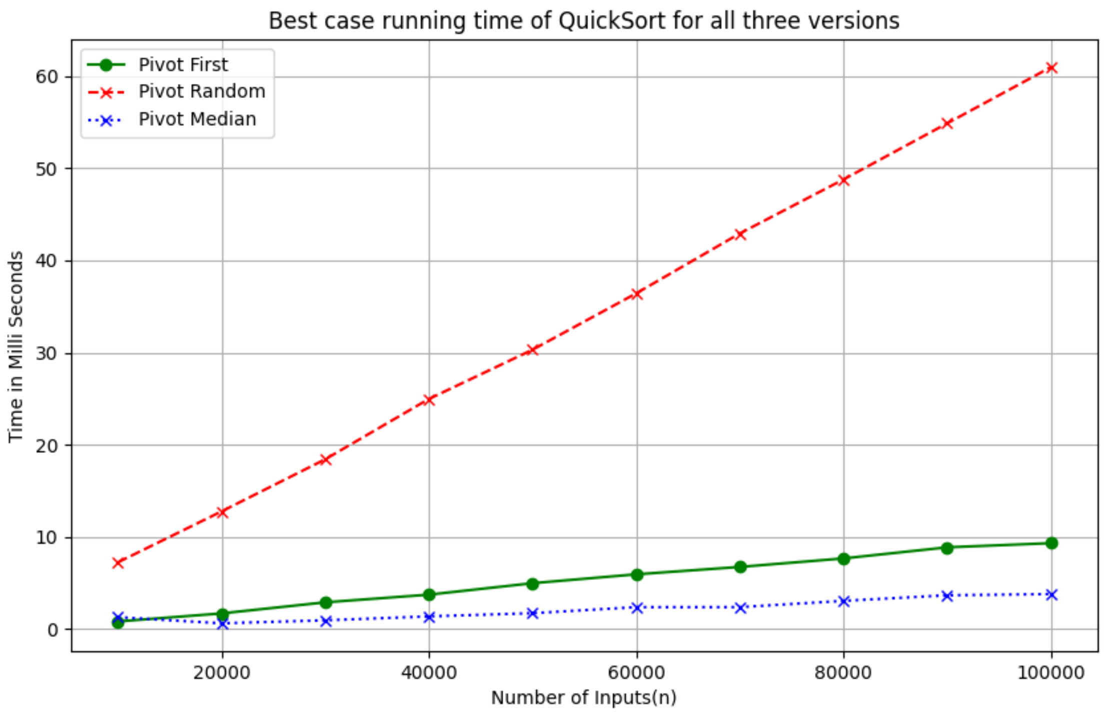

Figure 4: Best case running time of QuickSort for all three versions

3.5 Worst Case Performance of Three Versions of Quick Sort

Worst case occurs when partitions are highly unbalanced, which leads to a worst-case time complexity of $O(n^2)$.

- **Version 1:** For both sorted lists (increasing and decreasing), when the pivot is the first element, it leads to unbalanced partitions and a time complexity of $O(n^2)$. Though both the increasing and decreasing arrays have the same time complexity of $O(n^2)$, the experiments show that the increasing array results in the worst time.

- **Version 2:** When the pivot is selected randomly, there is a very low chance of getting an unbalanced partition. The worst case occurs when the randomly selected pivot consistently picks extreme values (either the smallest or largest element), which is highly unlikely. According to experiments and analysis, all three inputs result in the same time complexity of $O(n \log n)$, but the random order array yields the worst time among them.

- **Version 3:** Similar to the case with a random pivot, using the median of the first, middle, and last elements as the pivot generally prevents the worst-case scenario. According to the experiments, while all input versions have a time complexity of $O(n \log n)$, the random order array results in the worst performance.

Following Figure-5 and Figure-6 shows worst case running time of all the three versions of quick sort

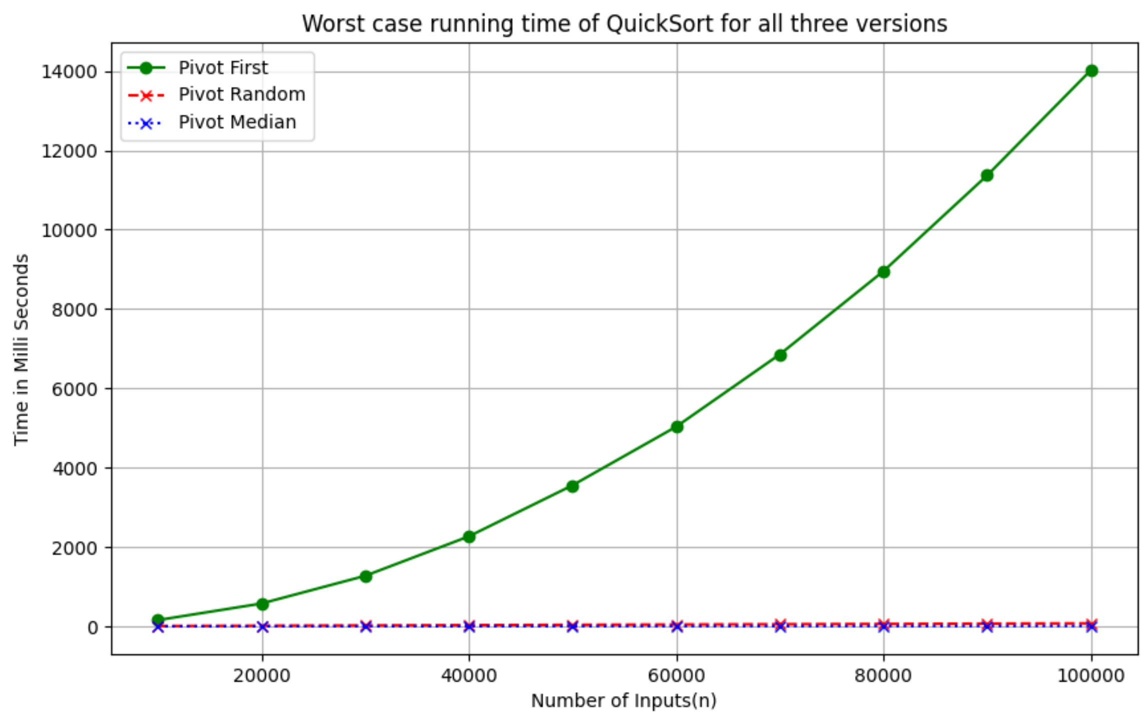

Figure 5: Worst case running time of QuickSort for all three versions

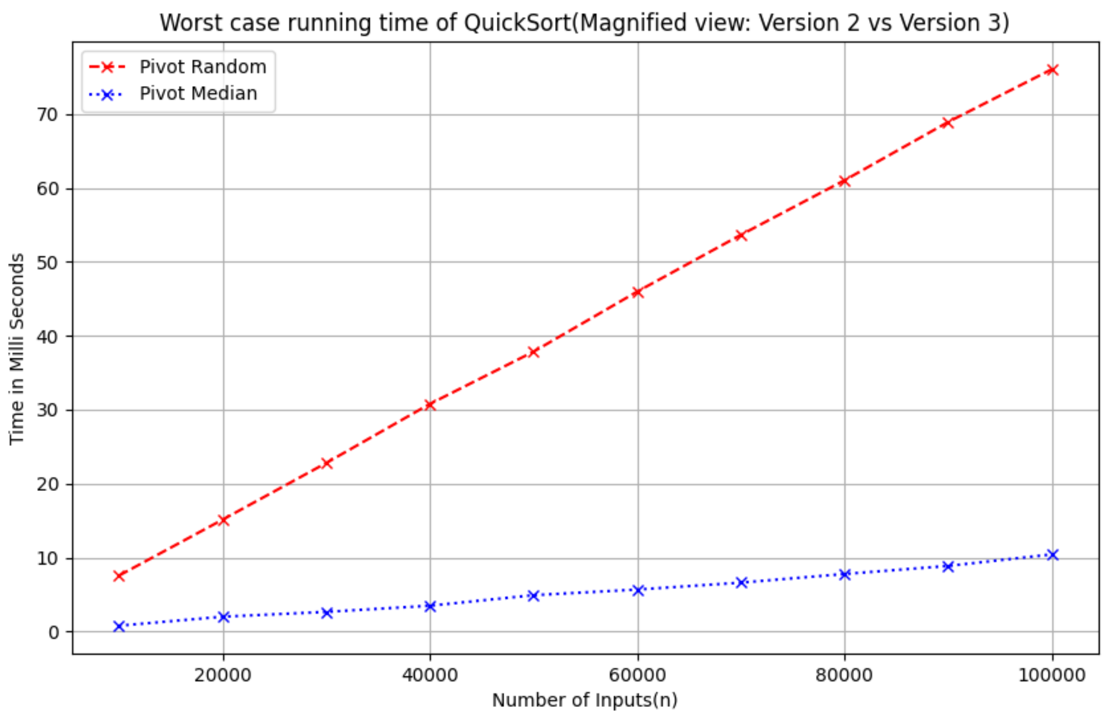

Figure 6: Worst case running time of QuickSort(Magnified view: Version 2 vs Version 3)

3.6 Average Case Performance of Three Versions of Quick Sort

All three versions of QuickSort yield an average time complexity of $O(n \log n)$.

- **Version 1:** When the array is neither fully sorted nor completely unbalanced, balanced partitions occur on average, leading to an average time complexity of $O(n \log n)$. For analyzing the average case, a different input file was considered where parts of the array are sorted, and other parts are shuffled.
- **Version 2:** When the pivot is selected randomly, the average case also results in a time complexity of $O(n \log n)$. The random pivot choice tends to create balanced partitions on average, leading to efficient sorting in most cases.
- **Version 3:** Using the median of the first, middle, and last elements as the pivot generally produces more balanced partitions, resulting in an average case time complexity of $O(n \log n)$. This method minimizes the chances of extreme unbalanced partitions, ensuring consistent performance across different input types.

Following Figure-7 and Figure-8 shows Average case running time of all the three versions of quick sort

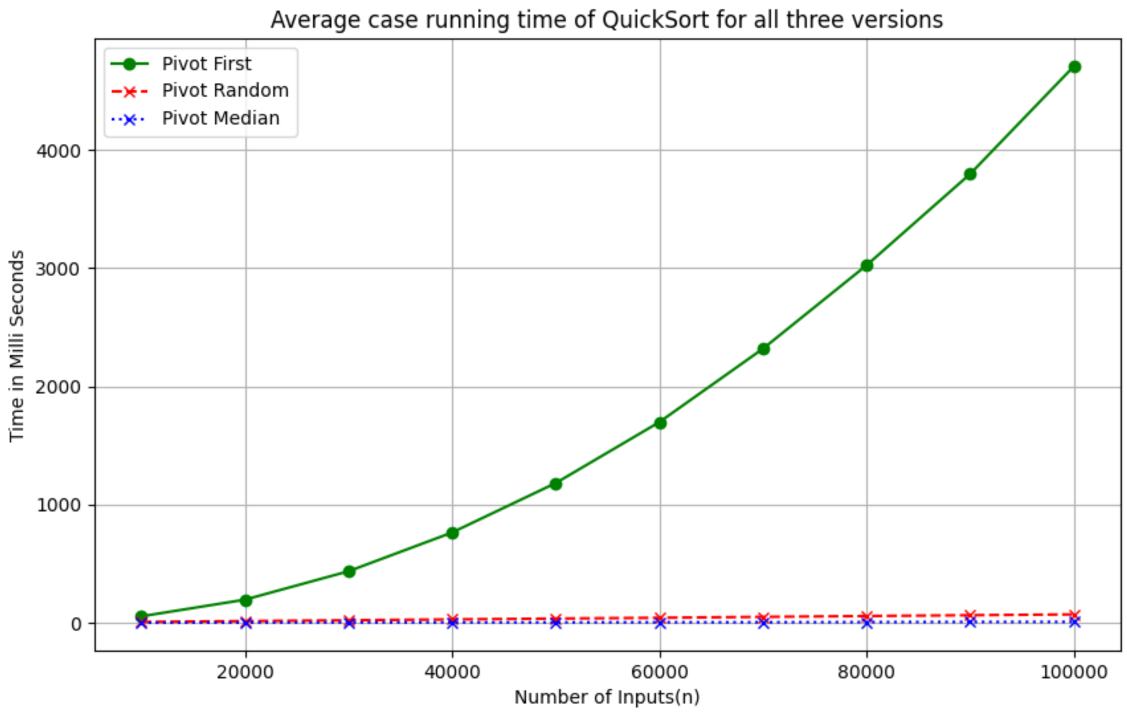

Figure 7: Average case running time of QuickSort for all three versions

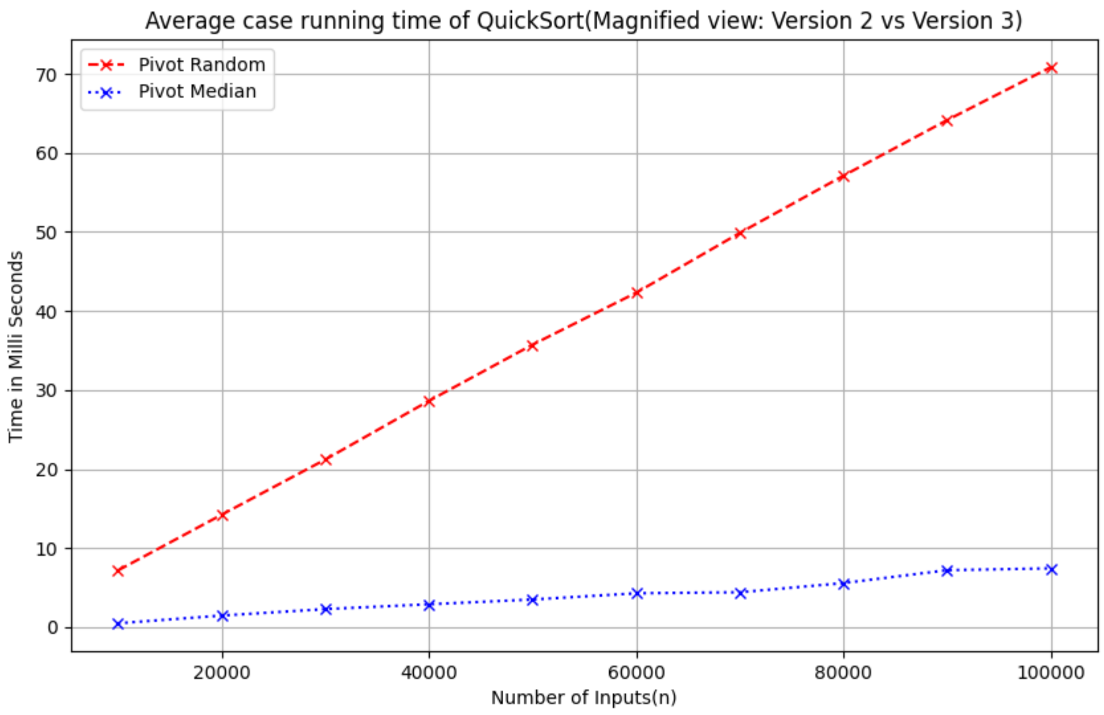

Figure 8: Worst case running time of QuickSort(Magnified view: Version 2 vs Version 3)

3.7 Conclusion of Quicksort Analysis

By considering all the analysis, the median-based pivot is the most reliable, minimizing the chances of extreme unbalanced partitions and consistently providing a time complexity of $O(n \log n)$ across all three input versions. This is followed by the random pivot, with the first-element pivot being the least favorable.