Analysis of Sorting and Searching Algorithm

Mohd Gulam Mohd Hasim Ansari
(Master of Technology)
Department of Computer Science and Engineering,NITK Surathkal

Assignment No - 1

1 Introduction

In this assignment we have practically analyze and compare the performance of various sorting and searching algorithms in terms of execution time. The algorithms studied include Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Radix Sort, Linear Search and Binary Search. The performance study, experimental setup, and implementation of these algorithms are covered in depth in this report.

2 Sorting Algorithms

2.1 Bubble Sort

Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted. Below is the code snippet of optimized bubble sort algorithm used for analysis.

```
bubbleSort(vector<int>numbers, int size) {
     bool swapped;

    for (int i = 0; i < size - 1; i++) {
        swapped = false;
        for (int j = 0; j < size - i - 1; j++) {
            if (numbers[j] > numbers[j + 1]) {
                swap(numbers[j], numbers[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
```

2.2 Insertion Sort

Insertion Sort builds the final sorted array one item at a time. It one-by-one picks elements from the unsorted portion and inserts them into their correct position. Below is the code snippet of Insertion sort used for analysis.

```
 insertionSort(vector<int>numbers, int size){
    for (int i = 1; i < size; ++i) {
        int key = numbers[i];
        int j = i - 1;

        while (j >= 0 && numbers[j] > key) {
            numbers[j + 1] = numbers[j];
            j = j - 1;
        }
        numbers[j + 1] = key;
    }
```

2.3 Merge Sort

To create the sorted array, it recursively divides the input array into smaller sub-arrays, sorts those sub-arrays, and then merges them back together.To put it simply, merge sort involves splitting an array in half, sorting each half separately, and then merging the sorted halves back together. Until the entire array is sorted, these steps are repeated. Below is the code snippet of Insertion sort used for analysis.

```
merge(vector<int> &numbers, int low, int mid, int high) {
    vector<int> temp;
    int left = low;
    int right = mid + 1;

    //storing elements in the temporary array in a sorted manner

    while (left <= mid && right <= high) {
        if (numbers[left] <= numbers[right]) {
            temp.push_back(numbers[left]);
            left++;
        }
        else {
            temp.push_back(numbers[right]);
            right++;
        }
    }

    while (left <= mid) {
        temp.push_back(numbers[left]);
        left++;
    }

    while (right <= high) {
        temp.push_back(numbers[right]);
        right++;
    }

    for (int i = low; i <= high; i++) {
        numbers[i] = temp[i - low];
    }
}

mergeSort(vector<int> &numbers, int low, int high) {
    if (low >= high) return;
    int mid = (low + high) / 2 ;
    mergeSort(numbers, low, mid);  // left half
    mergeSort(numbers, mid + 1, high); // right half
    merge(numbers, low, mid, high);  // merging sorted halves
}
```

2.4 Quick Sort

QuickSort is a sorting method that uses the Divide and Conquer algorithm as its basis. It selects an element to act as a pivot and divides the array around it by positioning the pivot correctly within the sorted array. After first Iteration all the elements to the left of the pivot element is less than or equal to pivot element and all the elements to the right of the pivot is greater than pivot element. Running time of quick sort heavily depend on selection of pivot element technique.Three pivot choices are considered for the analysis purpose:

2.4.1 Pivot Choice 1: First Element

It picks first element of an array as pivot element. Below is the code snippet considered for the analysis.

```
partitionFirstPivot(vector<int>& arr, int low, int high) {
    int pivot = arr[low];  // First element as pivot
    int i = low + 1;

    for (int j = low + 1; j <= high; j++) {
        if (arr[j] < pivot) {
            swap(arr[i], arr[j]);
            i++;
        }
    }
    swap(arr[low], arr[i - 1]);
    return i - 1;
}
```

```
quickSortFirstPivot(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partitionFirstPivot(arr, low, high);
        quickSortFirstPivot(arr, low, pi - 1);
        quickSortFirstPivot(arr, pi + 1, high);
    }
}
```

2.4.2 Pivot Choice 2: Random Element

It picks random element of an array as pivot element. Below is the code snippet considered for the analysis.

```
quickSortRandomPivot(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partitionRandomPivot(arr, low, high);
        quickSortRandomPivot(arr, low, pi - 1);
        quickSortRandomPivot(arr, pi + 1, high);
    }
}
```

```
partitionRandomPivot(vector<int>& arr, int low, int high) {
    int randomIndex = low + rand() % (high - low + 1);
    swap(arr[low], arr[randomIndex]);
    return partitionFirstPivot(arr, low, high); }
```

2.4.3 Pivot Choice 3: Median of Three

It picks median of first, middle and last element of an array as pivot element. Below is the code snippet considered for the analysis.

```
quickSortMedianPivot(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partitionMedianPivot(arr, low, high);
        quickSortMedianPivot(arr, low, pi - 1);
        quickSortMedianPivot(arr, pi + 1, high);
    }
}

medianOfThree(vector<int>& arr, int low, int mid, int high) {
    if (arr[low] > arr[mid]) swap(arr[low], arr[mid]);
    if (arr[low] > arr[high]) swap(arr[low], arr[high]);
    if (arr[mid] > arr[high]) swap(arr[mid], arr[high]);
    return mid;  // The middle element is now the median
}
```

```
partitionMedianPivot(vector<int>& arr, int low, int high) {
    int mid = low + (high - low) / 2;
    int medianIndex = medianOfThree(arr, low, mid, high);
    swap(arr[low], arr[medianIndex]);
    return partitionFirstPivot(arr, low, high);
    }
```

2.5 Heap Sort

A sorting method based on comparison and the Binary Heap data structure is called heap sort. It resembles the selection sort method in which the minimum element is located first and positioned at the start. For the remaining elements, carry out the same procedure again.Below is the code snippet of Heap Sort used for the analysis.

```
 heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;


    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);

        heapify(arr, n, largest);
    }
}

heapSort(vector<int>& arr, int n) {

    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);


    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

2.6 Radix Sort

Utilizing place value is the fundamental principle of Radix Sort.Radix Sort processes integer keys digit by digit starting from the least significant digit to the most significant digit. Below is the code snippet of Radix sort considered for the analysis.

```
countingSort(vector<int>& arr, int n, int exp) {
    vector<int> output(n);
    int count[10] = {0};

    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

radixSort(vector<int>& arr, int n) {
    int max_val = max(arr);
    for (int exp = 1; max_val / exp > 0; exp *= 10)
        countingSort(arr, n, exp);
}
```

3 Experimental Setup and Data Generation

3.1 Machine Specifications

The experiments were conducted on a machine with the following specifications:

- CPU: Intel(R) Core(TM) i5- 8250U CPU @ 1.60 GHz
- RAM: 8 GB
- Operating System: Microsoft Windows-11
- Compiler: g++ (MinGW.org GCC-6.3.0-1) 6.3.0

3.2 Timing Mechanism

Execution times were measured in **Microseconds** using the **chrono** library in C++ to get high-resolution timings. The following code snippet was used to measure the time:

```
auto start = chrono::high_resolution_clock::now();
// Execute sorting algorithm
auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::microseconds>(end - start).count();
```

3.3 Input Data

Three types of input numbers arrays with variable sizes were created and stored in txt files:

- Random order elements
- Elements sorted in increasing order
- Elements sorted in decreasing order
- File sizes range from 10,000 to 100,000, with intervals of 10,000.
- Each file contains numbers from 1 to its maximum size (e.g., a 10,000-size file contains numbers from 1 to 10,000).

The input sizes were varied to study performance and provide multiple data points for plotting the Execution Time vs. Input Size graph.

3.4 Experiment Repetitions

Each experiment was repeated **20** times to ensure stable and reliable measurements.

3.5 Time Capture

The reported times represent the averages of the execution times calculated across all repetitions of each experiment for corresponding data size.

3.6 Selection Of Inputs

We generated the input numbers as follows:

- For the increasing order list, the numbers were generated from 1 to the size of the list.

- For the decreasing order list, the numbers were generated from the size of the list down to 1.

- To generate the random order list, we used the rand() function from the C++ library to randomly shuffle these numbers.

This ensures a variety of input data for analyzing the performance of the algorithms.

3.7 Same Input For All Sorting Algorithm or Not?

Yes, I have used same input files for all the sorting algorithm's execution time analysis.

4 Quick Sort Versions Result Analysis

The three versions of "Quick Sort" algorithms based on pivot element selection mechanism are as follows :

- Version 1: The first element of the list as pivot element
- Version 2: A random of the list as pivot element
- Version 3: The median of first element, middile element and last element of the list as pivot

We have to analyse the running time for Best Case, Worst Case and Average case of above three versions of algorithms for the corresponding case inputs(Increasing Order, Random Order, Decreasing Order).

4.0.1 Best Case Performance of Three Versions of Quick Sort

The best case scenario occurs for quick sort when there is no significant unbalanced partitions of numbers list with the time complexity of O(nlogn) where n is the number of elements in the list.

- Quick Sort(Version 1): It gives the best case running time for random order inputs because it does not divide the numbers list into skewed manner.
- Quick Sort(Version 2): As per experimental observation it gives best case running time for increasing order numbers list.
- Quick Sort(Version 3): As per experimental observation it gives best case running time for random order numbers list.

Following Figure-1 shows the best case running time for all the three versions of quick sort.

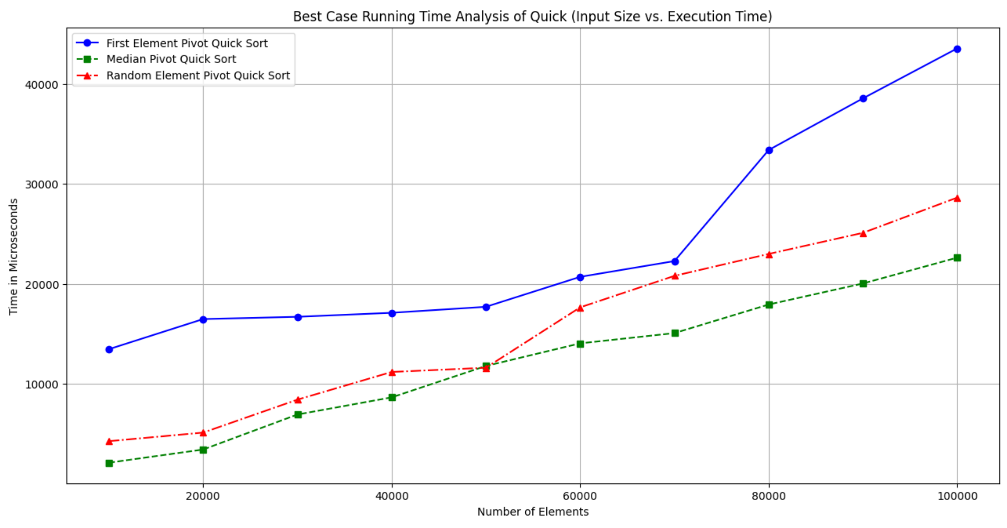

Figure 1: Best Case Running Time of All Three Versions of Quick Sort

4.0.2 Worst Case Performance of Three Versions of Quick Sort

The worst case scenario for quick sort occurs when the pivot is the smallest or largest element, leading to unbalanced partitions with the time complexity of $O(n^2)$ where n is the number of elements in the list.

- Quick Sort(Version 1): It gives the worst case running time for increasing/decreasing order inputs because it does divide the numbers list into skewed manner thus partitions are highly unbalanced.
- Quick Sort(Version 2): As per experimental observation it gives worst case running time for random order numbers list.
- Quick Sort(Version 3): As per experimental observation it gives worst case running time for decreasing order numbers list.

Following Figure-2 and Figure-3 shows worst case running time of all the three versions of quick sort.

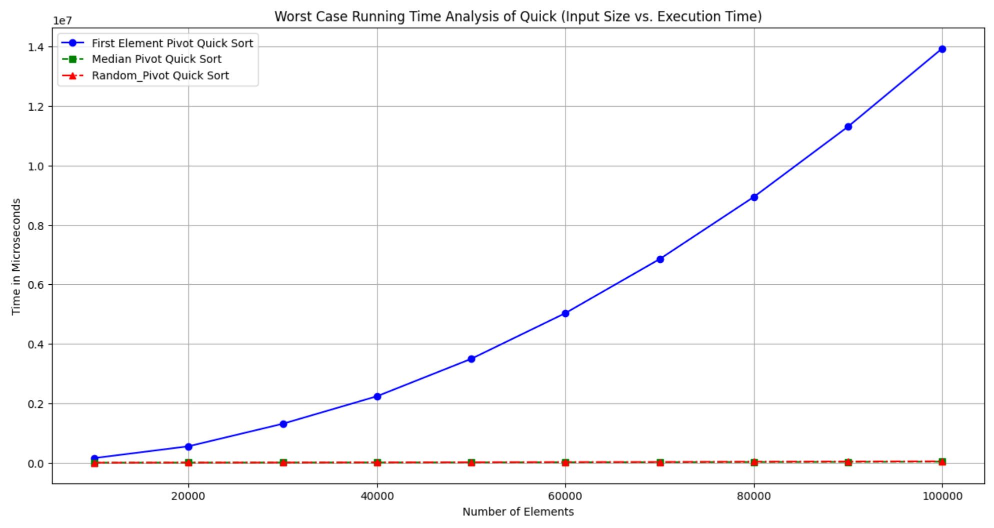

Figure 2: Worst Case Running Time of All Three Versions of Quick Sort

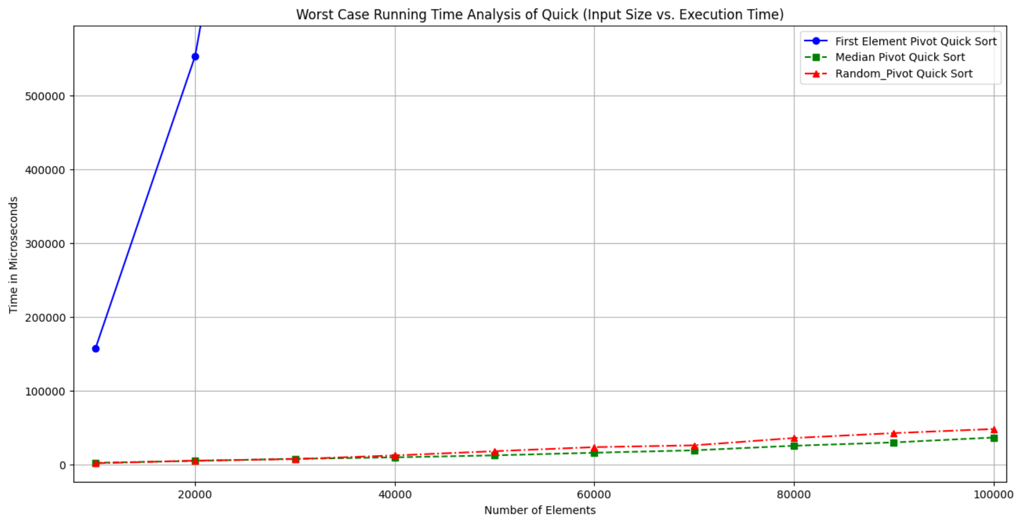

Figure 3: Magnified view: Worst Case Running Time of All Three Versions of Quick Sort

4.0.3 Average Case Performance of Three Versions of Quick Sort

Average Case Performance: It gives the average case running time for for all the versions of quick sort with time complexity of $O(n \log n)$ where n is the number of elements in the list.

- Quick Sort(Version 1): Generally performs well on average but can degrade to O(n^2) if the input is already sorted or nearly sorted.

- Quick Sort(Version 2): As per experimental observation it reduces the chance of consistently picking a poor pivot, especially in cases of sorted inputs.
- Quick Sort(Version 3): As per experimental observation it gives average case running time irrespective of input orders.

  Magnified view:

Following Figure-2 and Figure-3 shows worst case running time of all the three versions of quick sort.

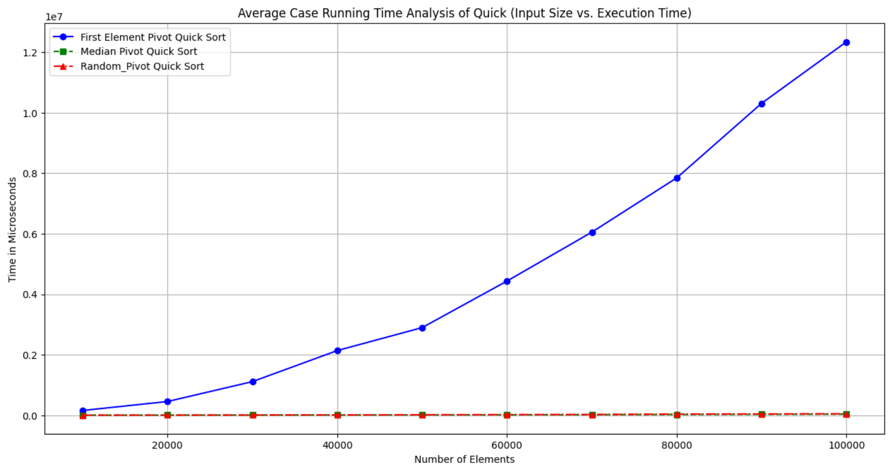

Figure 4: Average Case Running Time of All Three Versions of Quick Sort

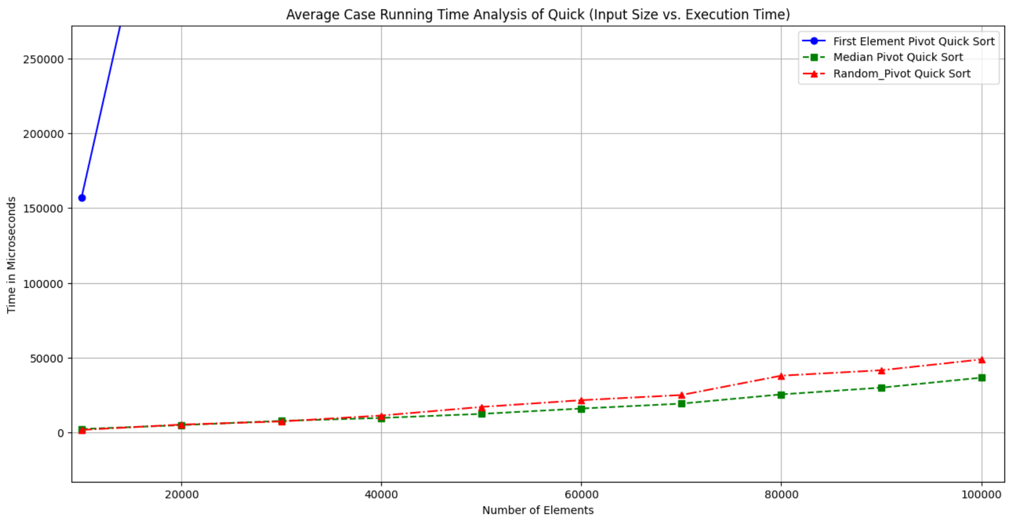

Figure 5: Magnified view: Average Case Running Time of All Three Versions of Quick Sort

4.1 Conclusion of Quick Sort Versions

While all three versions achieve O(nlogn) average-case time complexity, the median of first element, middle element and last element outperforms the others in due to better pivot selection. The random pivot approach is also effective, while the first element pivot selection mechanism is less reliable depending on the input order and it gives worst performance on sorted numbers list.

Pivot Selection Strategy	**Average Case Complexity**	**Worst Case Complexity**	**Best Performance**	**Worst Performance**	**Overall Effectiveness**
First Element	$O(n \log n)$	$O(n^2)$	Random Order	Increasing/Decreasing Order	Least Effective
Random Element	$O(n \log n)$	$O(n^2)$ (Less Likely)	Random Order	Less likely to hit worst case	Moderately Effective
Median of Three	$O(n \log n)$	$O(n \log n)$ (More Balanced)	Random Order	None (Performs well on all)	Most Effective

Table 1: Comparison of Quick Sort Pivot Selection Strategies

5 Performance of All Six Sorting Algorithms

5.1 Best Case Performance

As per the execution time analysis for all the algorithms,best case performance of algorithm is dependent on order of input numbers. Below is the practical observation of performance in terms of **execution time** for all algorithms.

- Bubble Sort: This algorithm performs best when the input numbers are already sorted.
- Insertion Sort: It also performs best when the input numbers are already sorted.
- Merge Sort: It is practically not dependent on order of input numbers.
- Quick Sort (Median as Pivot): It performs best for random order input numbers.
- Heap Sort: It performs best for random order input numbers.
- Radix Sort:It is practically not dependent on order of input numbers.

Following Figure-6 and Figure-7 shows the best case running time analysis of the sorting algorithms.

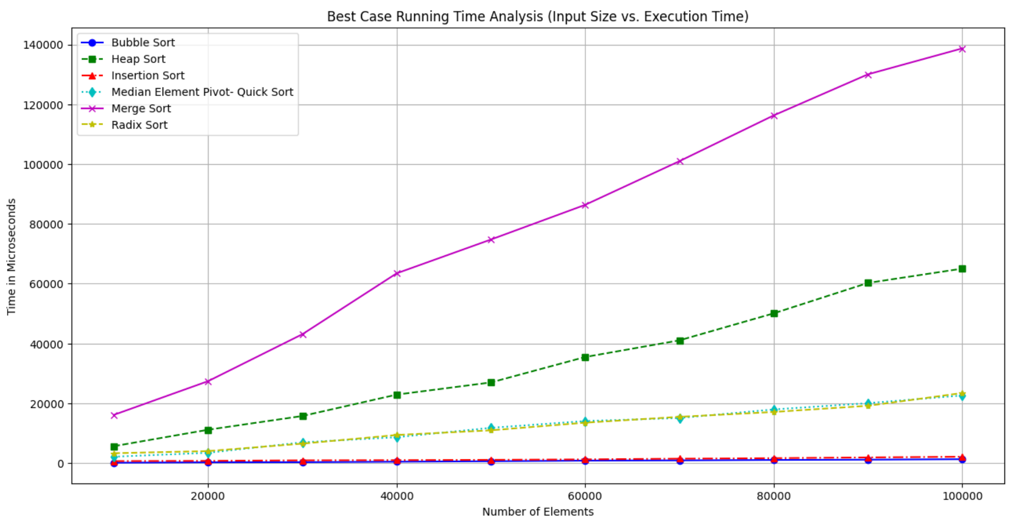

Figure 6: Best Case Running Time of All Sorting Algorithms

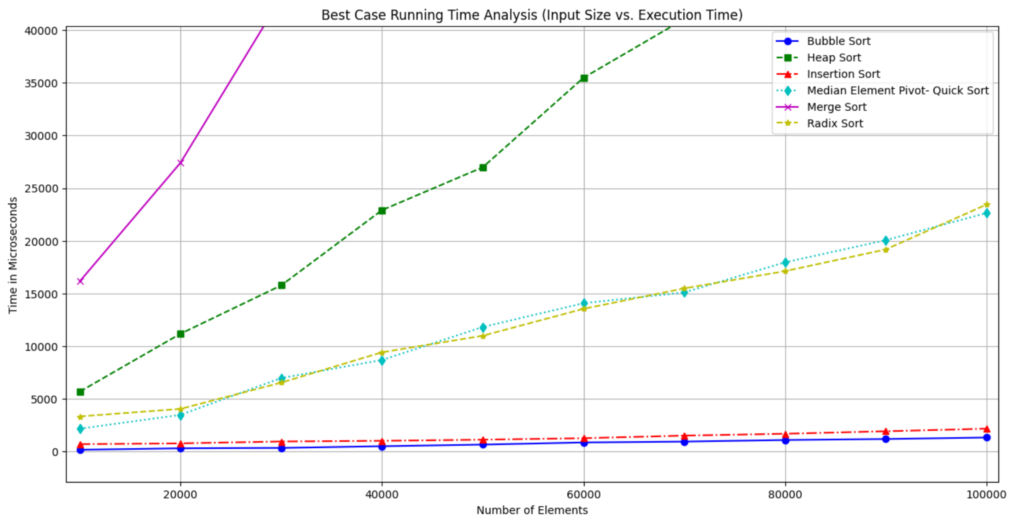

Figure 7: Magnified view: Best Case Running Time of All Sorting Algorithms

5.2 Worst Case Performance

- Bubble Sort: This algorithm performs worst when input numbers are reverse order sorted due to maximum comparisions.
- Insertion Sort: It also performs worst when input numbers are reverse order sorted due to maximum comparisions.
- Merge Sort: It is practically not dependent on order of input numbers.It is practically not dependent on order of input numbers.

- Quick Sort (Median as Pivot):It's performance is worst when numbers list are sorted.
- Heap Sort: The heapify process has to do more work because elements are initially in increasing order, which is the opposite of what a max-heap requires. However, since the entire array needs to be reorganized thus it performs worst in case of sorted order.
- Radix Sort: It is practically not dependent on order of input numbers.

Following Figure-8 and Figure-9 shows the worst case running time analysis of the sorting algorithms.

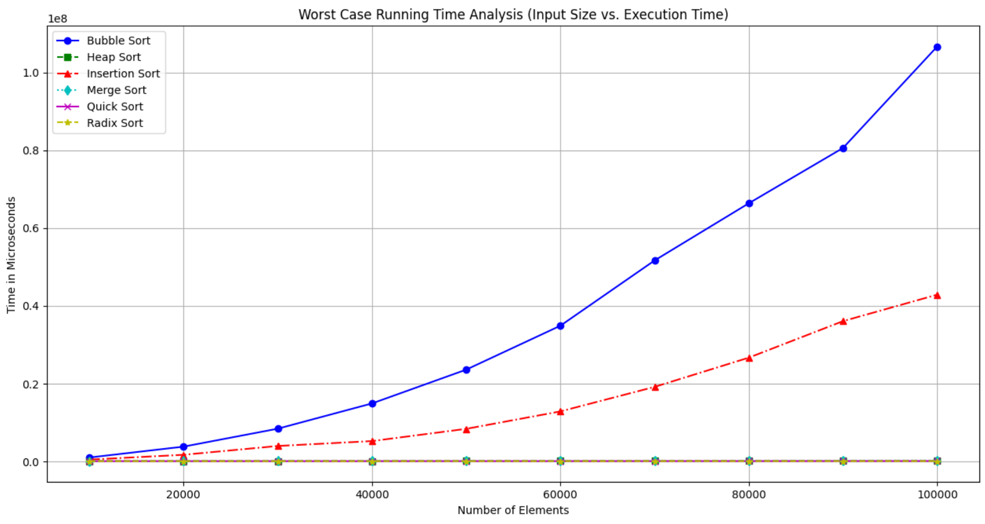

Figure 8: Worst Case Running Time of All Sorting Algorithms

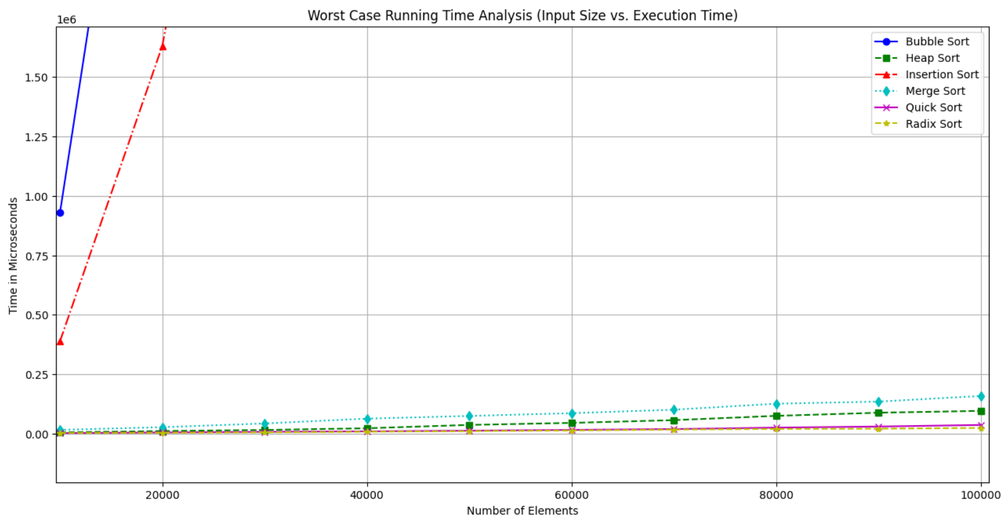

Figure 9: Magnified view: Worst Case Running Time of All Sorting Algorithms

5.3 Average Case Performance

- Bubble Sort: It gives average case performance for random order input numbers.
- Insertion Sort: It gives average case performance for random order input numbers.
- Merge Sort:It is practically not dependent on order of input numbers.It is practically not dependent on order of input numbers.
- Quick Sort (Median as Pivot):
- Heap Sort:It gives average case performance for increasing/decreasing order input numbers.
- Radix Sort:It is practically not dependent on order of input numbers.

Following Figure-10 and Figure-11 shows the average case running time analysis of the sorting algorithms.

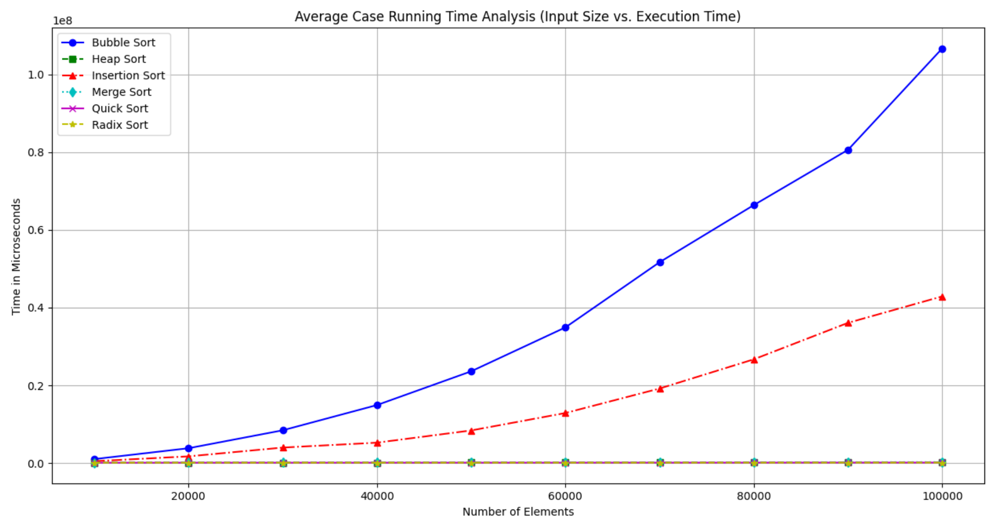

Figure 10: Average Case Running Time of All Sorting Algorithms

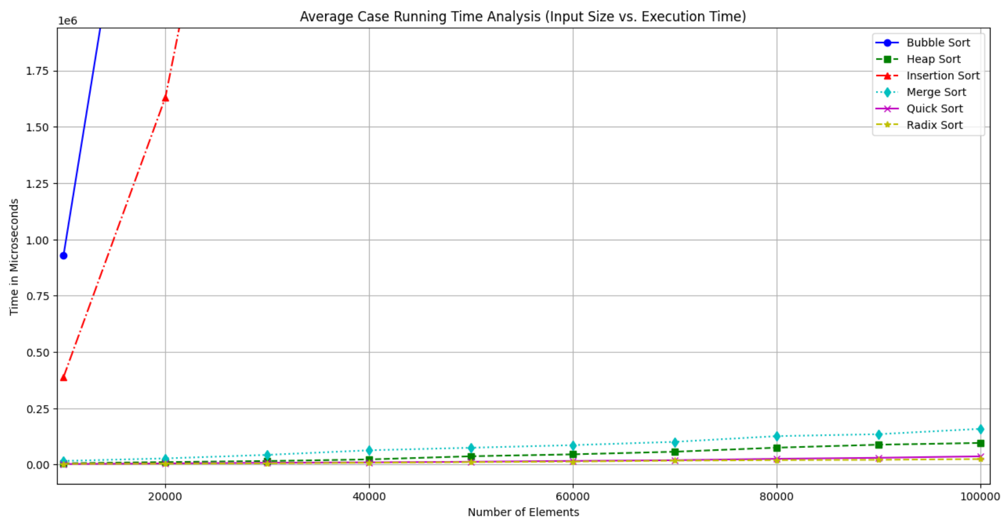

Figure 11: Magnified view: Average Case Running Time of All Sorting Algorithms

6 Correlation Between Number of Comparision With Execution Time