Report on Sorting Algorithms Performance

Sorting Algorithms Analyzed

We analyze three normal (iterative) sorting algorithms:

1. **Bubble Sort**
2. **Selection Sort**
3. **Insertion Sort**

We also analyze two recursive sorting algorithms: 4. **Merge Sort** 5. **Quick Sort** (with various pivot strategies)

Performance Analysis

The performance is rated on three data configurations:

- **Best Case:** Array is already sorted.
- **Worst Case:** Array is sorted in reverse order.
- **Average Case:** Array is shuffled 30 times, and the average sorting time is computed.

1. Bubble Sort

- **Best Case:** O(n)
  Bubble Sort performs well only in the best case due to early termination when no swaps are needed. However, it is still slower than advanced algorithms like Merge Sort and Quick Sort.
- **Worst Case:** O(n^2)
  In the reverse order, Bubble Sort compares every element multiple times, making it one of the slowest algorithms.
- **Average Case:** O(n^2)
  It remains inefficient compared to other sorting algorithms for large datasets.

2. Selection Sort

- **Best Case:** O(n^2)
  Selection Sort always makes n2n^2n2 comparisons regardless of input, making it slower than Bubble Sort for sorted data.
- **Worst Case:** O(n^2)
  The algorithm performs consistently poorly in this case due to repetitive element selection.
- **Average Case**: O(n^2)
  While predictable, it is outperformed by both Insertion Sort and the recursive algorithms.

3. Insertion Sort

- **Best Case:** O(n)
  Insertion Sort shines here, requiring only n−1n-1n−1 comparisons when the array is already sorted.

- **Worst Case:** O(n^2)
  When the array is reversed, Insertion Sort has to shift each element to the correct position, making it as slow as Bubble Sort.

- **Average Case:** O(n^2)
  Despite its quadratic time complexity, Insertion Sort is typically faster than Bubble Sort and Selection Sort.

4. Merge Sort

- **Best Case:**O(nlogn)
  Merge Sort's performance is consistent across all cases since it divides the array into halves and performs sorting and merging.

- **Worst Case:** O(nlogn)
  Even in the reversed order, the performance does not degrade.

- **Average Case:** O(nlogn)
  Merge Sort consistently performs well and is a reliable choice for larger datasets.

5. Quick Sort (Random Pivot)

- Best Case: O(nlog⁡n)O(n \log n)O(nlogn)
  Quick Sort performs excellently when partitioning is balanced. The randomized pivot strategy ensures this for sorted data.

- Worst Case: O(n2)O(n^2)O(n2)
  If the pivot selection is poor (e.g., smallest or largest element), Quick Sort degrades to quadratic time.

- Average Case: O(nlog⁡n)O(n \log n)O(nlogn)
  On average, Quick Sort is faster than Merge Sort due to its in-place partitioning, making it one of the fastest algorithms.

Rankings Based on Performance

Best Case (Sorted Data)

1. **Insertion Sort** (Fastest for small data)
2. **Merge Sort**
3. **Quick Sort**
4. **Bubble Sort**

5. **Selection Sort** (Slowest)

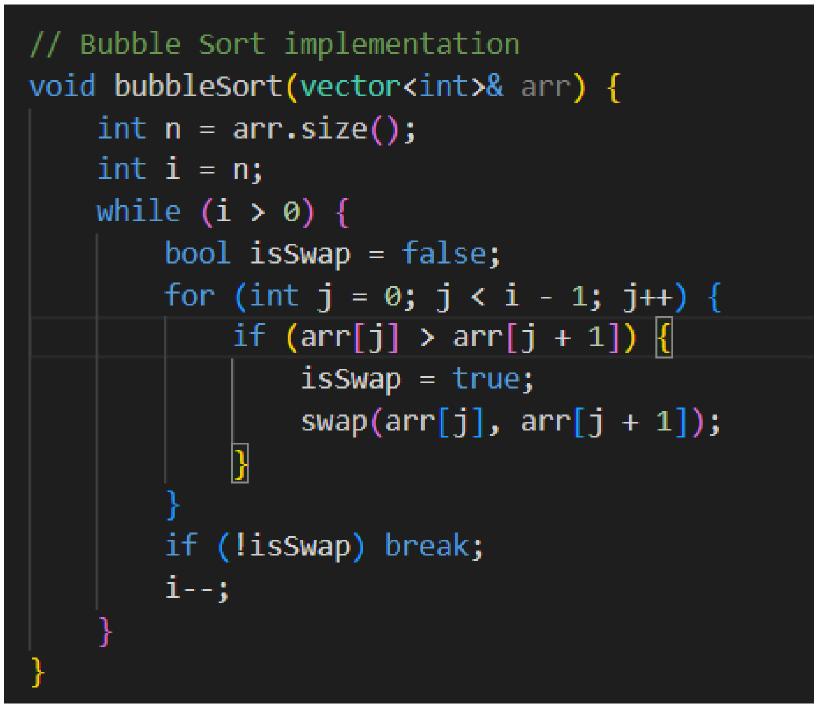
Worst Case (Reversed Data)

1. **Merge Sort** (Consistently fast)
2. **Quick Sort (Random Pivot)** (Slower for poor pivots)
3. **Insertion Sort**
4. **Bubble Sort**
5. **Selection Sort** (Predictably slow)

Average Case (Shuffled Data)

1. **Quick Sort (Random Pivot)** (Fastest overall)
2. **Merge Sort**
3. **Insertion Sort** (For small datasets)
4. **Bubble Sort**
5. **Selection Sort** (Slowest due to n2n^2n2 complexity)

Bubble Sort:



```cpp
// Bubble Sort implementation
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    int i = n;
    while (i > 0) {
        bool isSwap = false;
        for (int j = 0; j < i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                isSwap = true;
                swap(arr[j], arr[j + 1]);
            }
        }
        if (!isSwap) break;
        i--;
    }
}
```

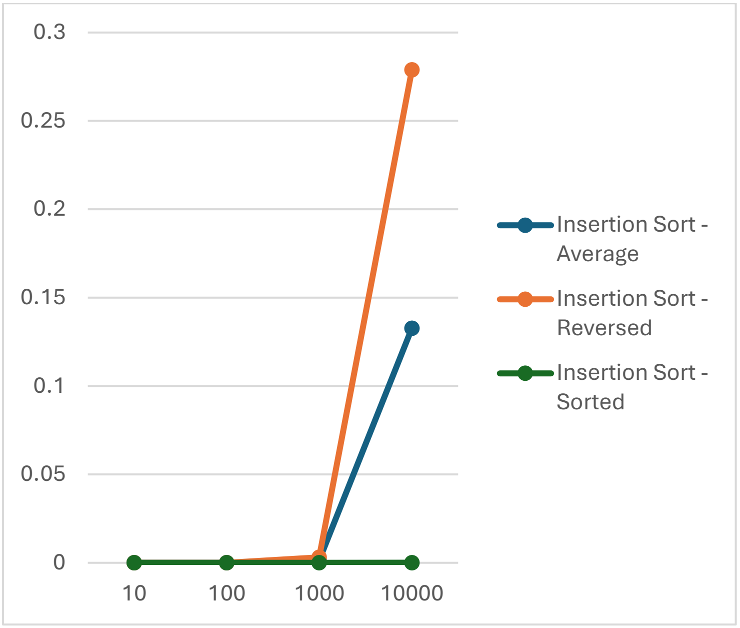
Function Description

1. **Inputs and Outputs**:
   - **Input**: A vector of integers (vector<int>& arr) passed by reference.
   - **Output**: The function sorts the input vector arr in ascending order in-place (no new array is created).
2. **Algorithm**:
   - The algorithm repeatedly iterates through the array, comparing adjacent elements and swapping them if they are in the wrong order.
   - The largest element "bubbles" to its correct position at the end of the array after each pass.
   - The function also incorporates an **optimization** to exit early if no swaps occur in a pass, indicating that the array is already sorted.

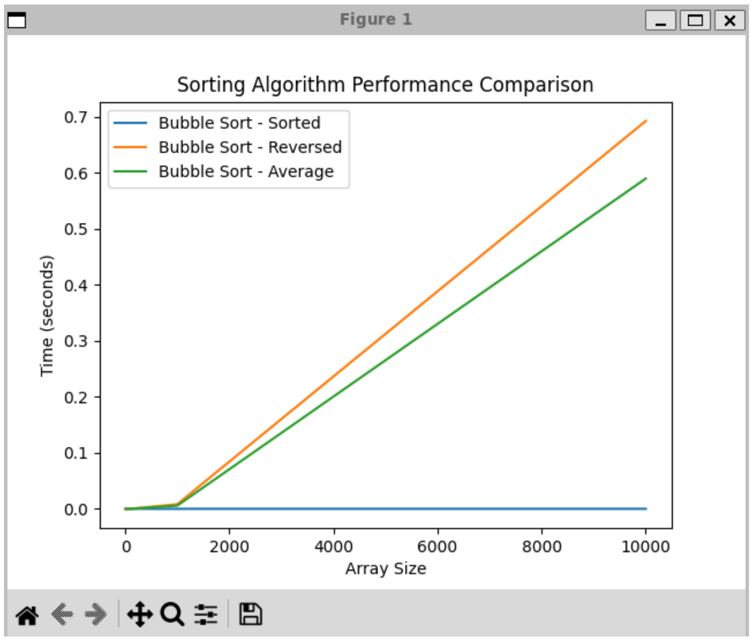
Below is a Line graph where arrays of size: 10, 100, 1000, 10000 are considered and plotted. The orange line shows the worst time complexity and green line shows best time complexity and blue line is the average time complexity with 30 diferent permutations of the arrays.

Time Complexity: Best – O(n), Worst and average- O(n^2).

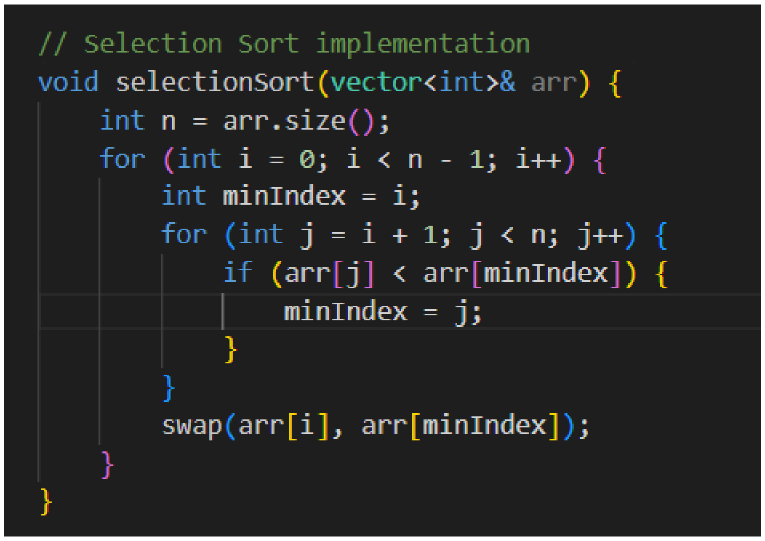
Graph Plotted in Excel:



Graph plotted using Matplotlib.CPP:



Selection Sort:



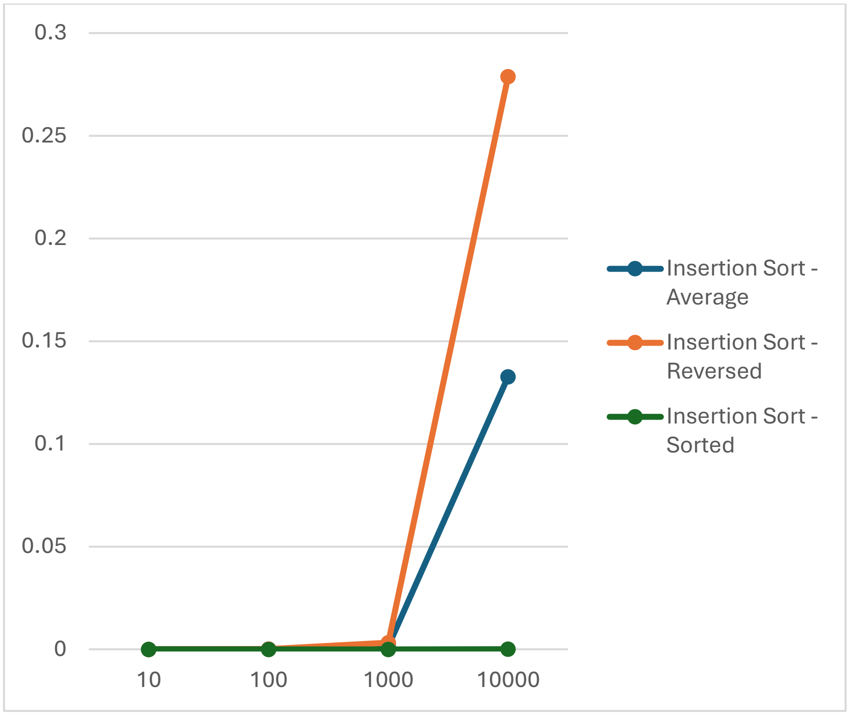
```
// Selection Sort implementation
void selectionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        swap(arr[i], arr[minIndex]);
    }
}
```

Function Description

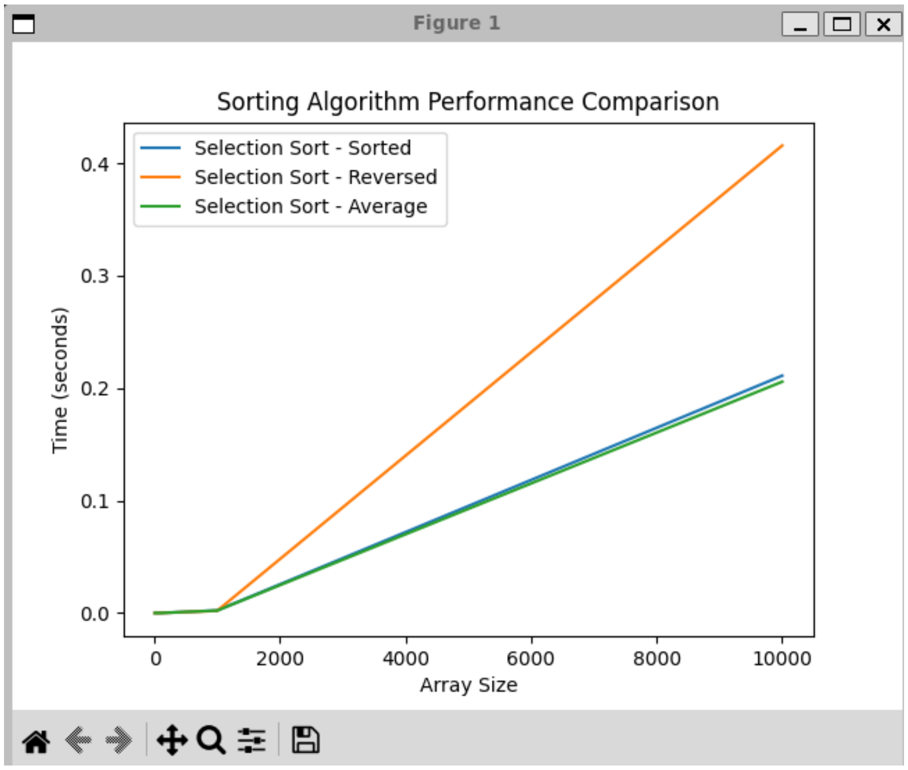
1. **Inputs and Outputs**:
    - **Input**: A vector of integers (vector<int>& arr) passed by reference.
    - **Output**: The function sorts the input vector arr in ascending order in-place (no additional memory is used).
2. **Algorithm**:
    - The algorithm divides the array into two parts: a sorted portion and an unsorted portion.
    - It repeatedly selects the smallest element from the unsorted portion and swaps it with the first element of the unsorted portion, expanding the sorted portion by one element after each pass.
    -

Below is a Line graph where arrays of size: 10, 100, 1000, 10000 are considered and plotted. The orange line shows the worst time complexity and green line shows best time complexity and blue line is the average time complexity with 30 diferent permutations of the arrays. As we can see, everytime the time complexity is O(n^2) though performance varies.

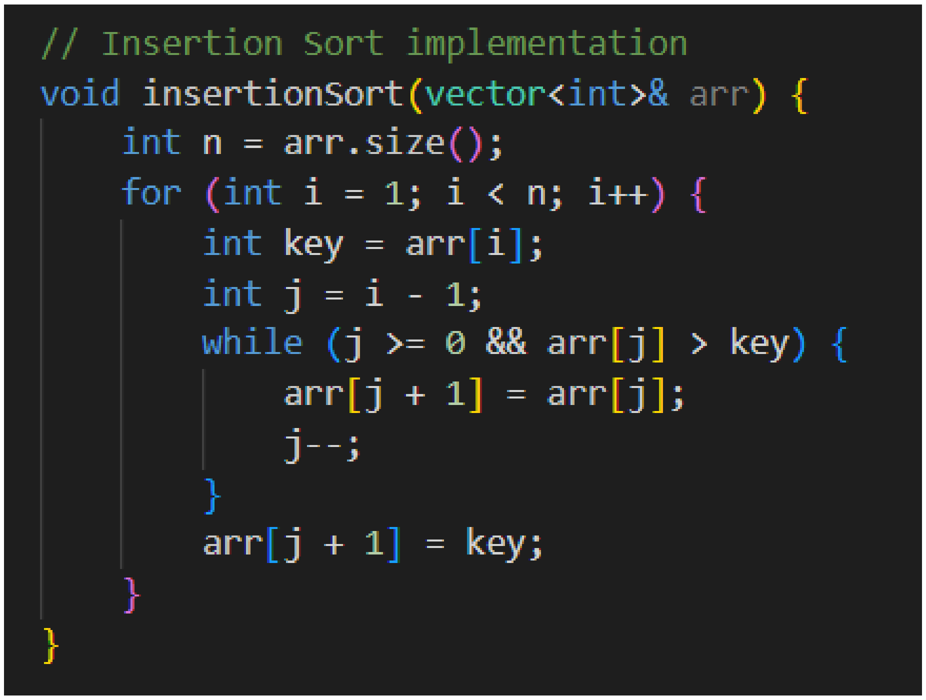
Graph Plotted in Excel:



Graph plotted using Matplotlib.CPP:



Insertion Sort:



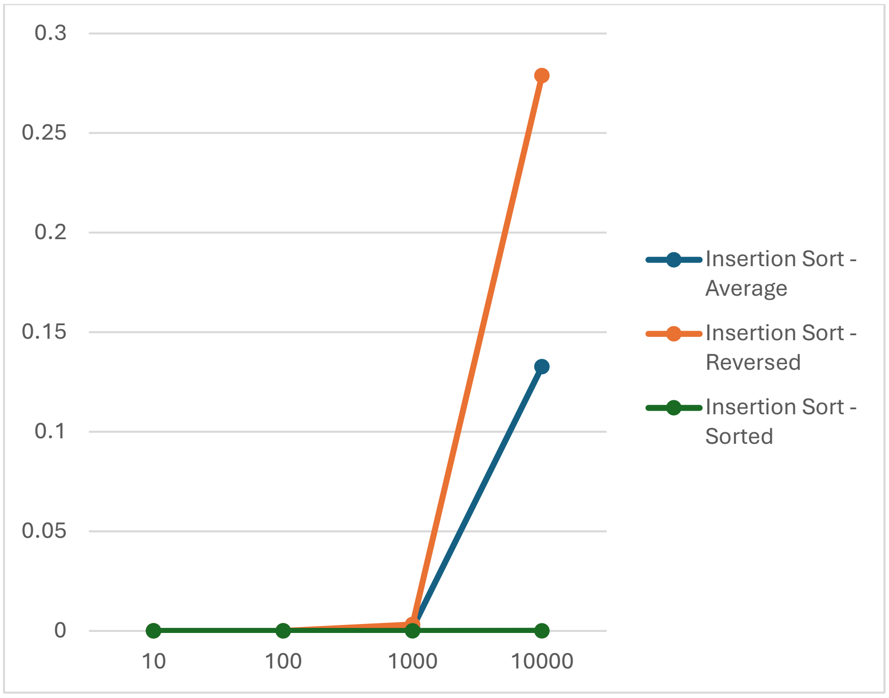
```
// Insertion Sort implementation
void insertionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Function Description

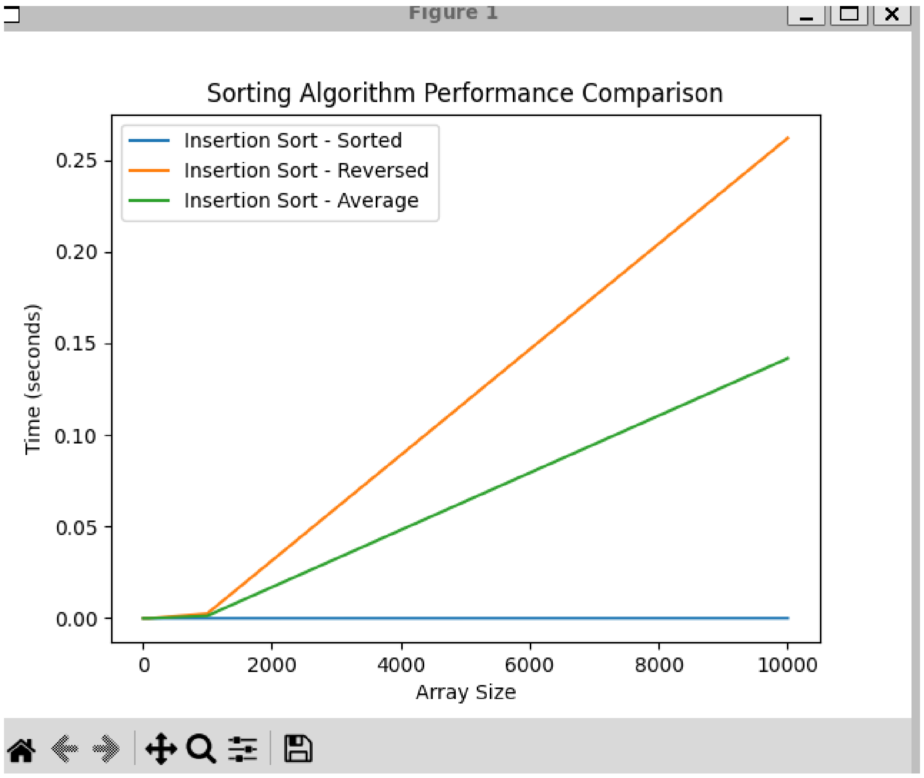
1. **Inputs and Outputs**:
    - **Input**: A vector of integers (vector<int>& arr) passed by reference.
    - **Output**: The function sorts the input vector arr in ascending order in-place (no additional memory is used).
2. **Algorithm**:
    - The algorithm divides the array into two parts: a sorted portion and an unsorted portion.
    - It repeatedly selects the smallest element from the unsorted portion and swaps it with the first element of the unsorted portion, expanding the sorted portion by one element after each pass.
    -

Below is a Line graph where arrays of size: 10, 100, 1000, 10000 are considered and plotted. The orange line shows the worst time complexity and green line shows best time complexity and blue line is the average time complexity with 30 diferent permutations of the arrays. As we can see, the worst and average time complexity is O(n^2) and the best time complexity is O(n).

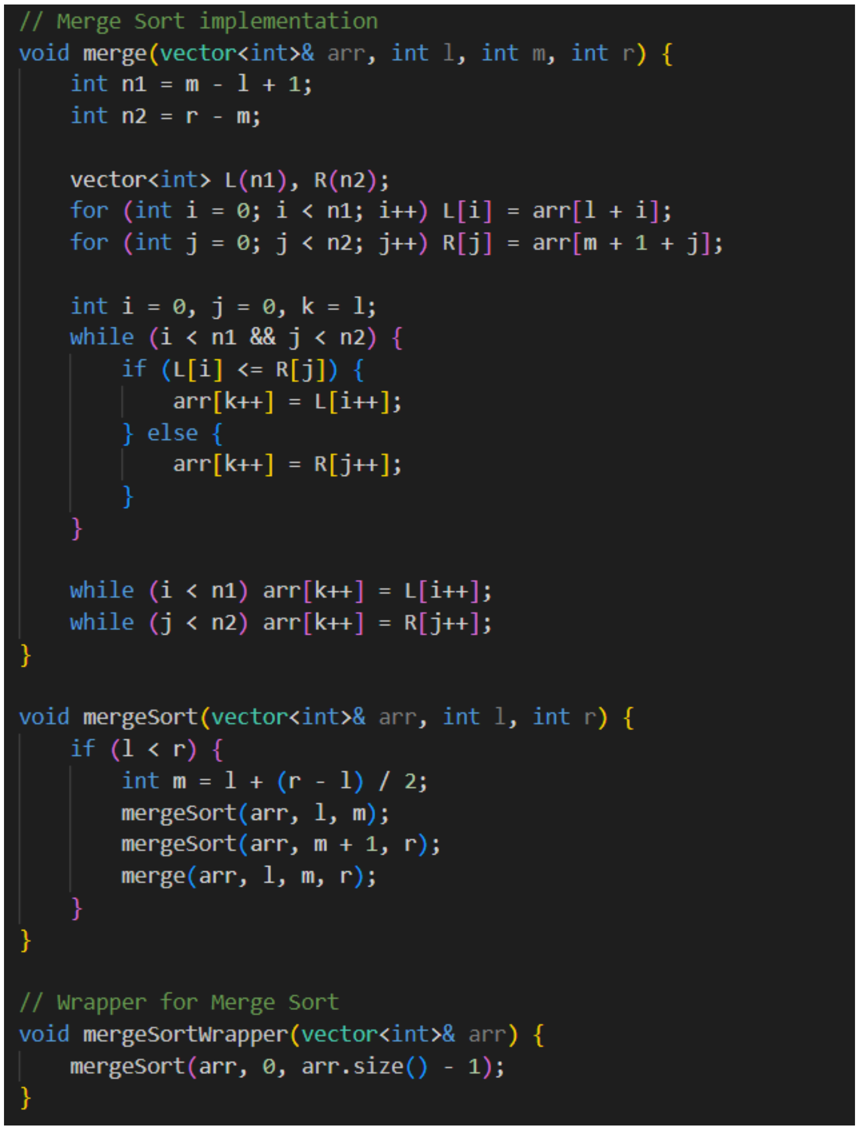
Graph Plotted in Excel:



Graph plotted using Matplotlib.CPP:



Merge Sort:



```
// Merge Sort implementation
void merge(vector<int>& arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void mergeSort(vector<int>& arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

// Wrapper for Merge Sort
void mergeSortWrapper(vector<int>& arr) {
    mergeSort(arr, 0, arr.size() - 1);
}
```

mergeSort Function

1. **Purpose**:
   - Recursively divides the array into two halves until each half contains a single element, then merges the halves back together in sorted order.
2. **Parameters**:
   - arr: The array to be sorted.
   - l: The starting index of the portion to be sorted.

- r: The ending index of the portion to be sorted.

3. **Algorithm**:
   - Base case: If l >= r, the array is already sorted (a single element).
   - Recursively divide the array:
     - Calculate the middle index, m = l + (r - l) / 2.
     - Sort the left half (arr[l..m]).
     - Sort the right half (arr[m+1..r]).
   - Merge the two sorted halves using the merge function.

---

mergeSortWrapper Function

- A wrapper function to simplify the call to mergeSort, sorting the entire array by specifying its full range (0 to arr.size() - 1).

Performance Observations

Best Case (Sorted Data):

- Even when the array is already sorted, Merge Sort does not adapt and always divides and merges.
- Time Complexity: (nlogn).

Worst Case (Reverse-Sorted Data):

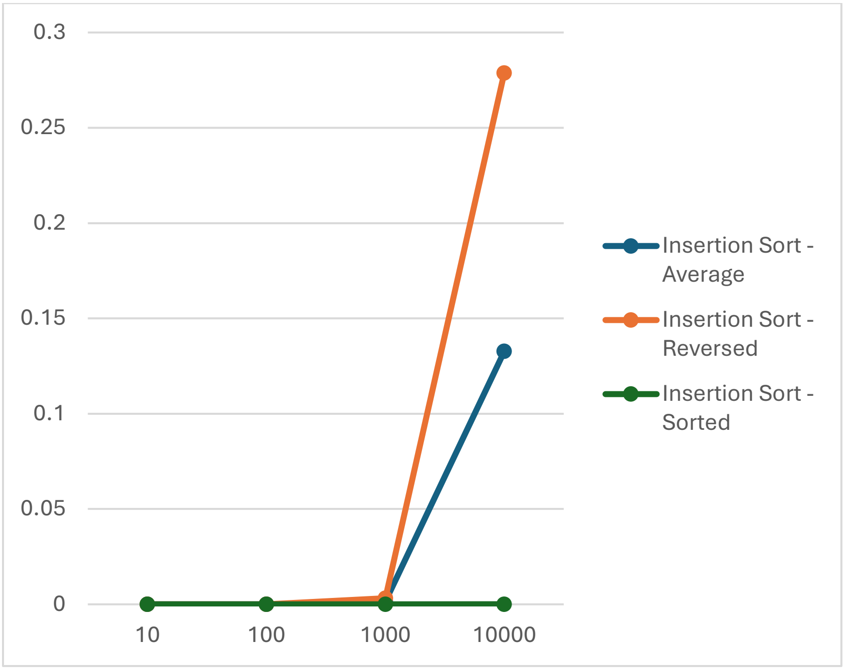
- Reverse-sorted data is treated the same as sorted data since Merge Sort divides and merges irrespective of order.
- Time Complexity: (nlogn).

Average Case (Random Data):

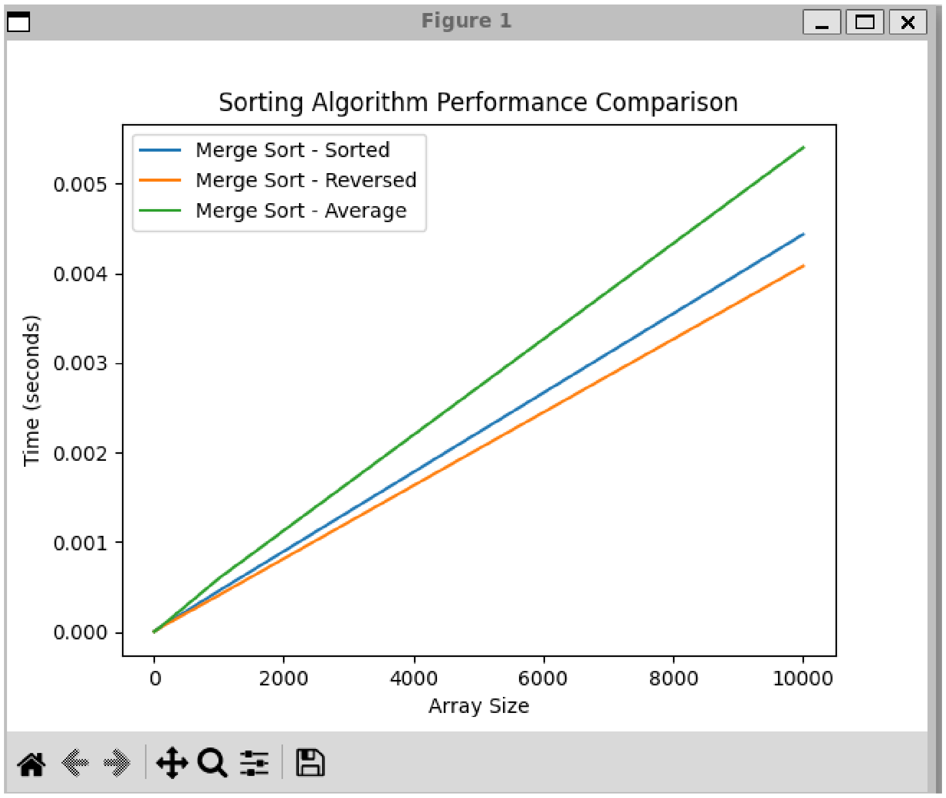
- Merge Sort performs the same operations for any random arrangement of data.
- Time Complexity: O(nlogn).

Below is a Line graph where arrays of size: 10, 100, 1000, 10000 are considered and plotted. The orange line shows the worst time complexity and green line shows best time complexity and blue line is the average time complexity with 30 diferent permutations of the arrays. As we can see, the time complexity is always O(nlogn) but the space complexity is O(n) because of extra space for new arrays.

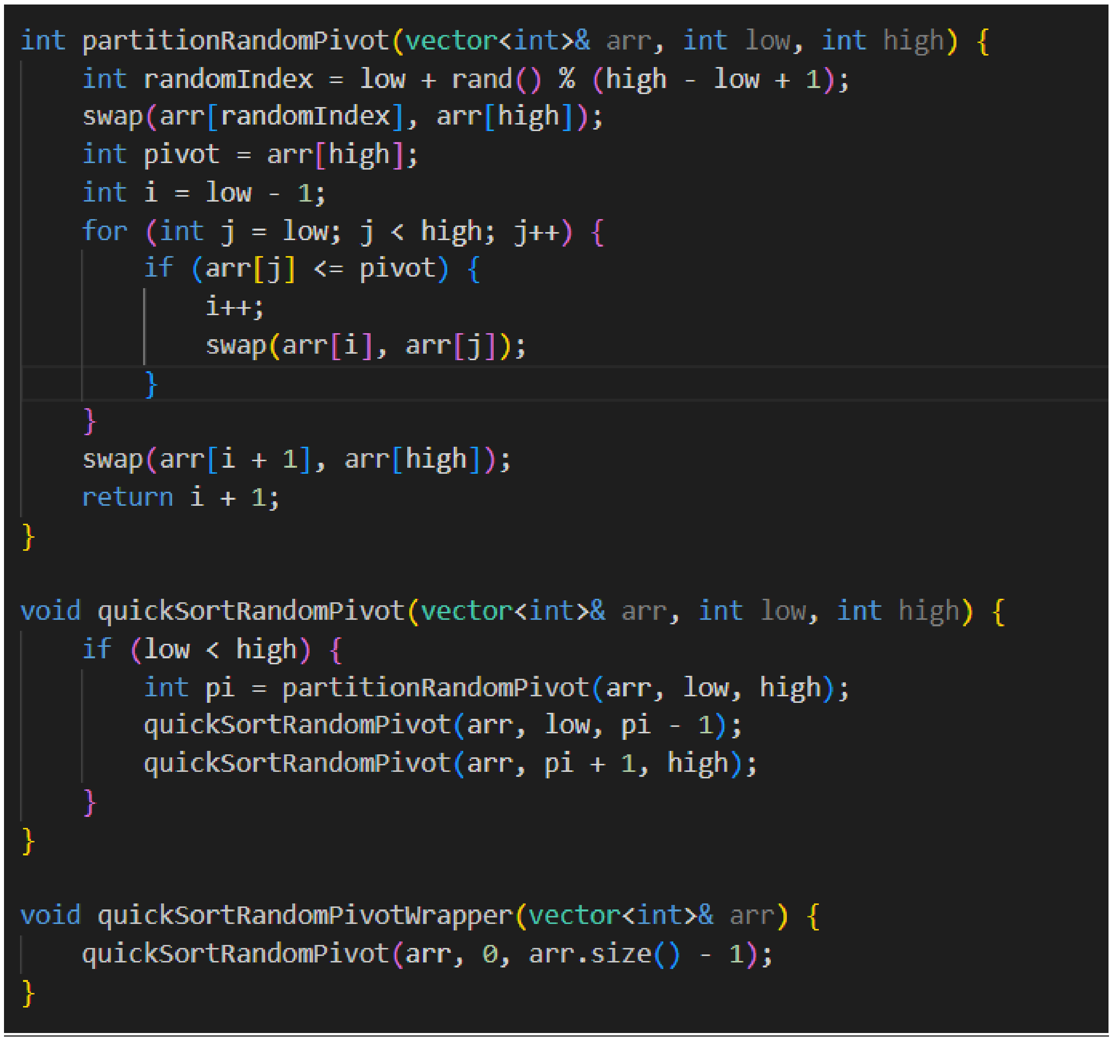
Graph Plotted in Excel:



Graph plotted using Matplotlib.CPP:



Quick Sort (Selecting Pivot as Random Element) :



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

void 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);
    }
}

void quickSortRandomPivotWrapper(vector<int>& arr) {
    quickSortRandomPivot(arr, 0, arr.size() - 1);
}
```

Function Description:

partitionRandomPivot

1. Purpose:
    - Partitions the array into two sections: one containing elements less than or equal to the pivot and the other containing elements greater than the pivot.
    - The pivot element is selected randomly to reduce the likelihood of worst-case performance.

2. Parameters:
    - arr: The array to be partitioned.
    - low: The starting index of the subarray.
    - high: The ending index of the subarray.
3. Algorithm:
    - Randomly select a pivot index between low and high.
    - Swap the randomly chosen pivot with the last element in the subarray (arr[high]).
    - Use two pointers to rearrange elements:
        - i: Tracks the position for smaller elements.
        - Iterate through the array (j), swapping elements when they are smaller than or equal to the pivot.
    - Swap the pivot back to its correct position in the array (i + 1).
    - Return the index of the pivot (i + 1).

Time Complexity:

Best Case:

- In the best-case scenario, the pivot divides the array into two nearly equal halves at every step.
- Time Complexity: O(nlogn).

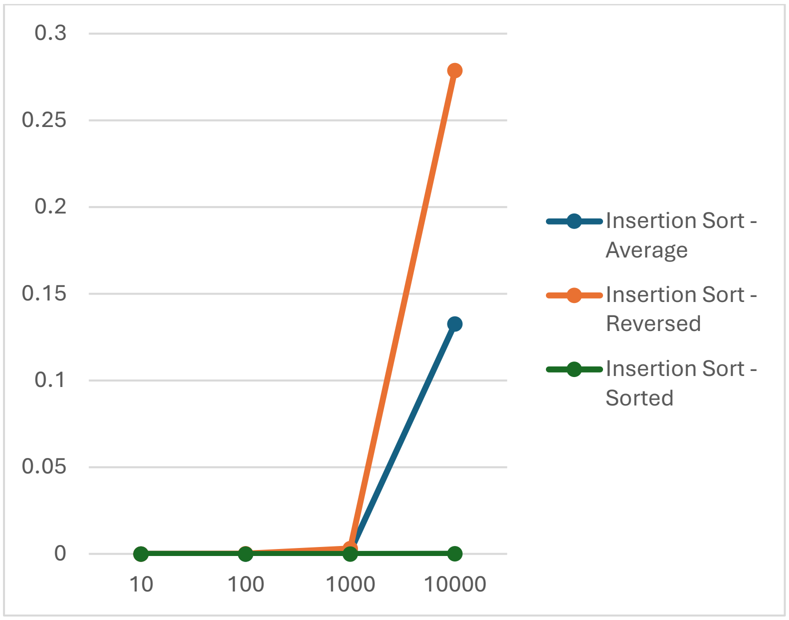
Average Case:

- On average, the random pivot helps maintain a balanced partitioning.
- Time Complexity: O(nlogn).

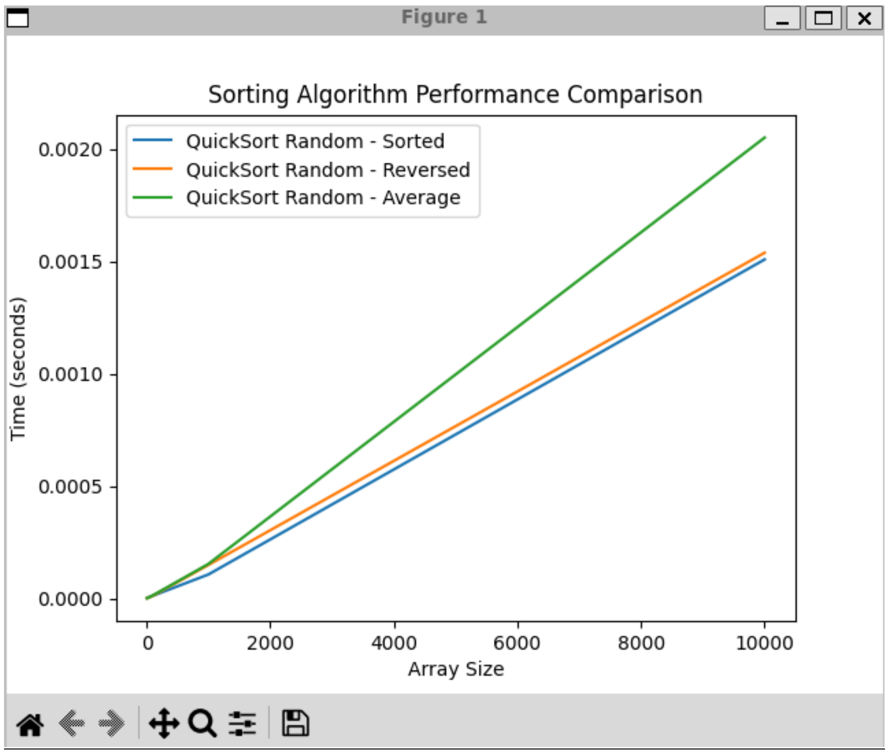
Worst Case:

- If the pivot selection consistently results in unbalanced partitions (e.g., one side contains all elements except one), the performance degrades to:
- Time Complexity: O(n^2)
- Note: Using a random pivot greatly reduces the likelihood of encountering the worst case.

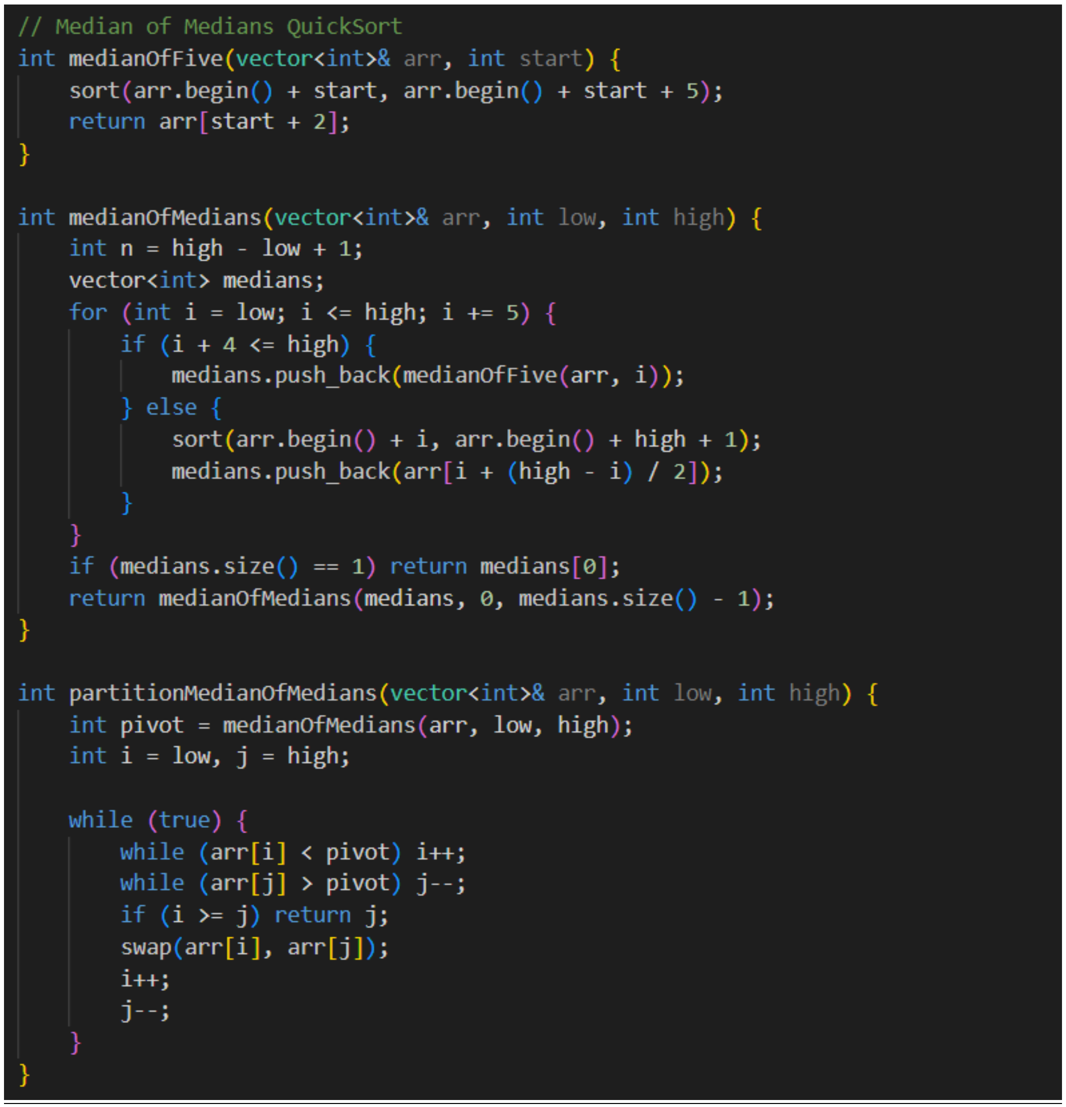
Graph Plotted on Excel:



Graph Plotted using Matplotlib.Cpp:



Quick Sort using Median of Medians(Division of 5):

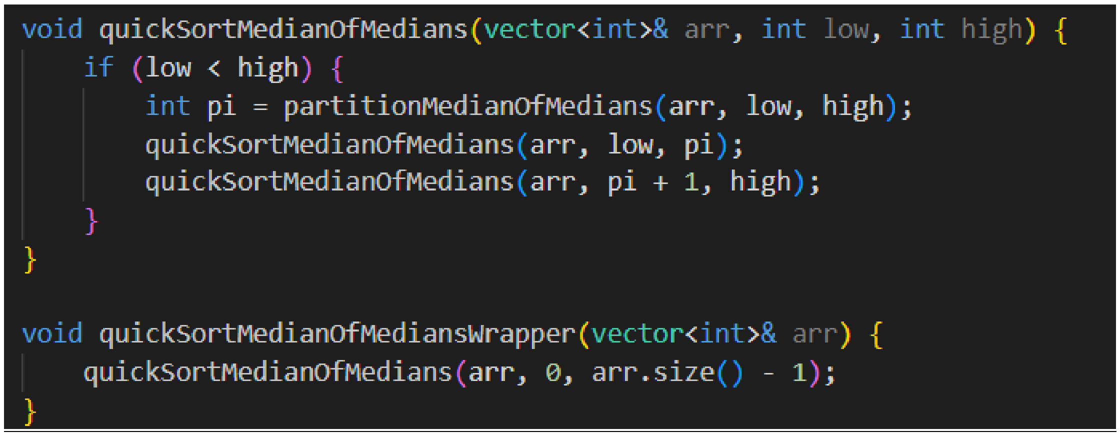


```cpp
// Median of Medians QuickSort
int medianOfFive(vector<int>& arr, int start) {
    sort(arr.begin() + start, arr.begin() + start + 5);
    return arr[start + 2];
}

int medianOfMedians(vector<int>& arr, int low, int high) {
    int n = high - low + 1;
    vector<int> medians;
    for (int i = low; i <= high; i += 5) {
        if (i + 4 <= high) {
            medians.push_back(medianOfFive(arr, i));
        } else {
            sort(arr.begin() + i, arr.begin() + high + 1);
            medians.push_back(arr[i + (high - i) / 2]);
        }
    }
    if (medians.size() == 1) return medians[0];
    return medianOfMedians(medians, 0, medians.size() - 1);
}

int partitionMedianOfMedians(vector<int>& arr, int low, int high) {
    int pivot = medianOfMedians(arr, low, high);
    int i = low, j = high;

    while (true) {
        while (arr[i] < pivot) i++;
        while (arr[j] > pivot) j--;
        if (i >= j) return j;
        swap(arr[i], arr[j]);
        i++;
        j--;
    }
}
```



```cpp
void quickSortMedianOfMedians(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partitionMedianOfMedians(arr, low, high);
        quickSortMedianOfMedians(arr, low, pi);
        quickSortMedianOfMedians(arr, pi + 1, high);
    }
}

void quickSortMedianOfMediansWrapper(vector<int>& arr) {
    quickSortMedianOfMedians(arr, 0, arr.size() - 1);
}
```

Function Descriptions

medianOfFive

1. Purpose:
    - Finds the median of a small group of up to 5 elements.
    - Sorting the group and selecting the middle element ensures accurate median computation.
2. Parameters:
    - arr: The array containing the group.
    - start: The starting index of the group.
3. Steps:
    - Sorts the group of 5 elements (or fewer if near the array's end).
    - Returns the median, which is the middle element (arr[start + 2]).

---

medianOfMedians

1. Purpose:
    - Recursively computes the median of medians as the pivot for partitioning.
    - Breaks the array into groups of 5, computes their medians, and recursively finds the median of those medians.
2. Parameters:
    - arr: The array to process.
    - low: The starting index of the range.
    - high: The ending index of the range.
3. Steps:
    - Divide the array into groups of 5 elements.
    - Compute the median for each group using medianOfFive.
    - Recursively find the median of these medians.
    - This ensures a well-balanced pivot for partitioning.
4. Result:
    - Returns a robust pivot that guarantees better performance, even in worst-case scenarios.

partitionMedianOfMedians

1. Purpose:
    - Partitions the array using the pivot provided by medianOfMedians.
2. Parameters:
    - arr: The array to partition.
    - low: The starting index of the range.
    - high: The ending index of the range.
3. Steps:
    - Obtain the pivot using medianOfMedians.
    - Use two pointers (i and j) to rearrange the elements:
        - Move elements smaller than the pivot to the left.
        - Move elements larger than the pivot to the right.
    - The pivot is placed in its correct sorted position, and the partition index is returned.

---

quickSortMedianOfMedians

1. Purpose:
    - Implements the recursive QuickSort algorithm using partitionMedianOfMedians.
2. Parameters:
    - arr: The array to sort.
    - low: The starting index of the range.
    - high: The ending index of the range.
3. Steps:
    - Base case: If the range has one or zero elements, it's already sorted.
    - Recursive case:
        - Partition the array using partitionMedianOfMedians.
        - Recursively sort the left and right subarrays.

---

quickSortMedianOfMediansWrapper

- Provides a simple interface for sorting an entire array by calling quickSortMedianOfMedians with appropriate parameters.

Time Complexity

Best Case:

- The pivot divides the array into two equal halves at each step.
- Time Complexity: O(nlog⁡n)O(n \log n)O(nlogn).

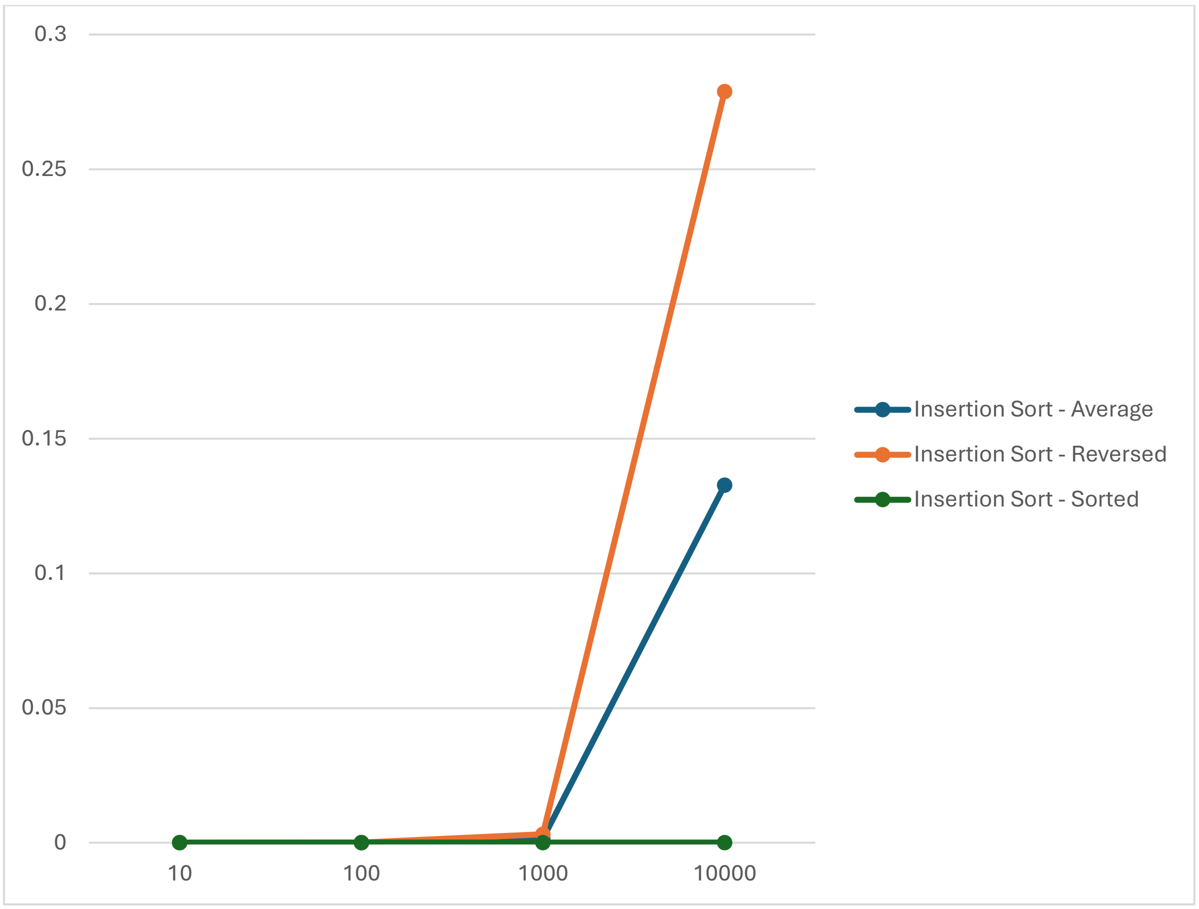
Worst Case:

- The Median of Medians guarantees a pivot that divides the array into two parts where neither part has more than 70% of the elements.
- Worst-case Time Complexity: O(nlog⁡n)O(n \log n)O(nlogn), as the partitioning is well-balanced.

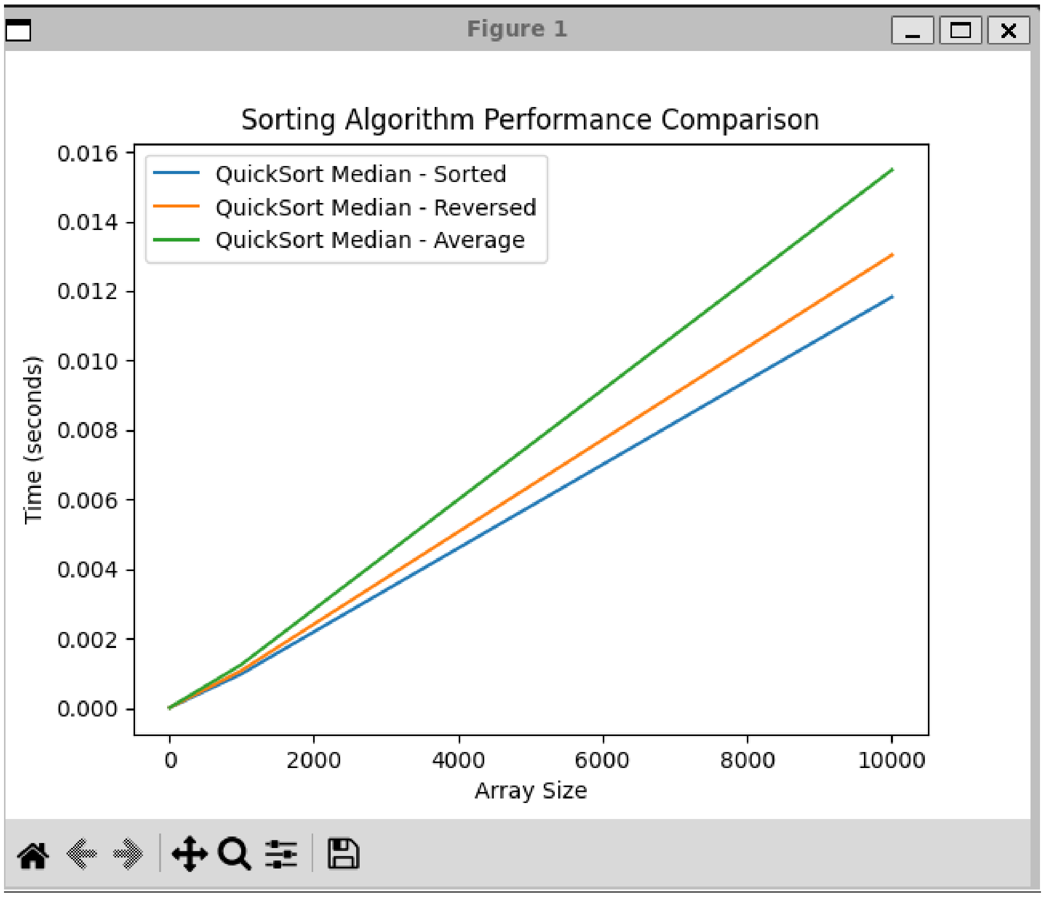
Average Case:

- Similar to the best case due to the robust pivot selection.
- Time Complexity: O(nlog⁡n)O(n \log n)O(nlogn).

Graph Plotted in Excel:



Graph Plotted in Matplotlib.CPP:

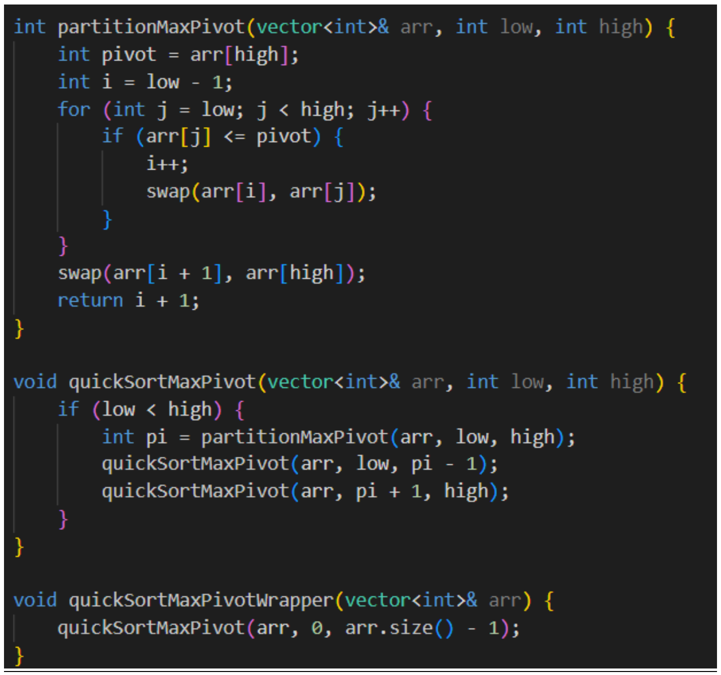


Space Complexity

1. **Auxiliary Space:**
   - The algorithm is in-place for partitioning, so no extra memory is required apart from the recursion stack

Quick Sort by selecting Minimum or Maximum as Pivot:

Maximum:

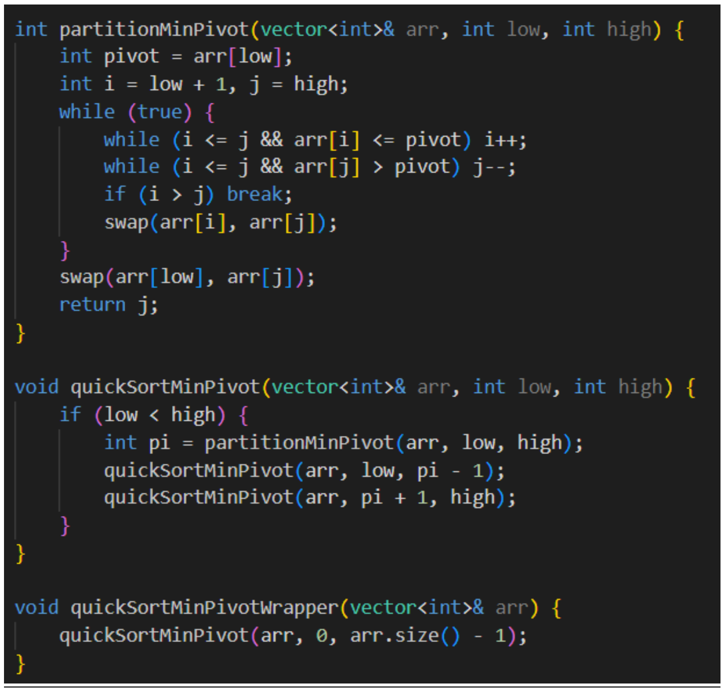


```cpp
int partitionMaxPivot(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i +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

void quickSortMaxPivot(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partitionMaxPivot(arr, low, high);
        quickSortMaxPivot(arr, low, pi - 1);
        quickSortMaxPivot(arr, pi + 1, high);
    }
}

void quickSortMaxPivotWrapper(vector<int>& arr) {
    quickSortMaxPivot(arr, 0, arr.size() - 1);
}
```

Minimum:



```cpp
int partitionMinPivot(vector<int>& arr, int low, int high) {
    int pivot = arr[low];
    int i = low + 1, j = high;
    while (true) {
        while (i <= j && arr[i] <= pivot) i++;
        while (i <= j && arr[j] > pivot) j--;
        if (i > j) break;
        swap(arr[i], arr[j]);
    }
    swap(arr[low], arr[j]);
    return j;
}

void quickSortMinPivot(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partitionMinPivot(arr, low, high);
        quickSortMinPivot(arr, low, pi - 1);
        quickSortMinPivot(arr, pi + 1, high);
    }
}

void quickSortMinPivotWrapper(vector<int>& arr) {
    quickSortMinPivot(arr, 0, arr.size() - 1);
}
```

<u>Performance Analysis for Minimum Pivot:</u>

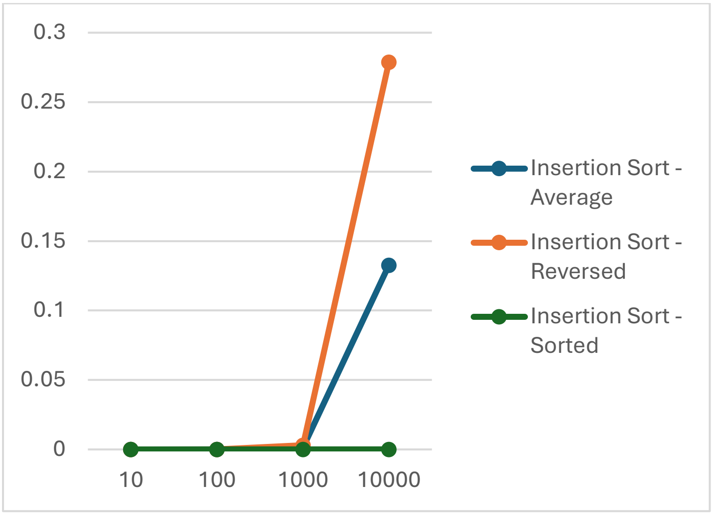
1. **<u>Best Case (Sorted Data):</u>**
    - Already sorted data leads to unbalanced partitioning.
    - Time Complexity: O(n^2) due to the smallest element always being chosen as the pivot.
2. **<u>Worst Case (Reverse-Sorted Data):</u>**
    - Completely unbalanced partitioning (like the best case).
    - Time Complexity:O(n^2)
3. **<u>Average Case (Random Data):</u>**
    - Random partitioning quality depends on input distribution.
    - Expected Time Complexity: O(nlogn) but constant factors are higher due to poor pivot selection.

<u>Performance Analysis for Maximum Pivot:</u>

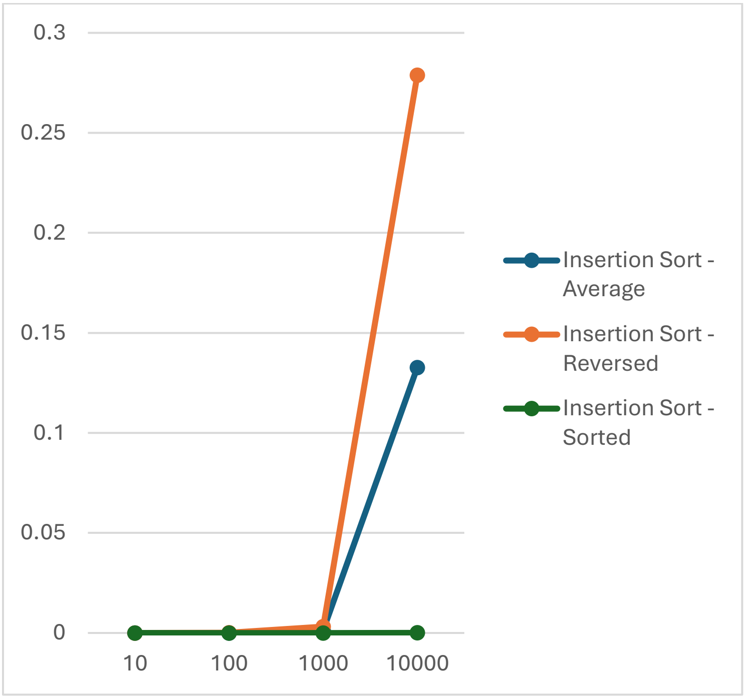
1. **<u>Best Case (Sorted Data)</u>**:
    - Already sorted data causes unbalanced partitioning.
    - Time Complexity: O(n^2) due to the largest element always being chosen as the pivot.
2. **Worst Case (Reverse-Sorted Data)**:
    - Completely unbalanced partitioning.
    - Time Complexity: O(n^2)
3. **Average Case (Random Data)**:
    - Similar to Min Pivot QuickSort.
    - Expected Time Complexity: O(nlogn), but constant factors are higher due to poor pivot selection.

Graph Plotted using Excel:

Maximum Pivot:

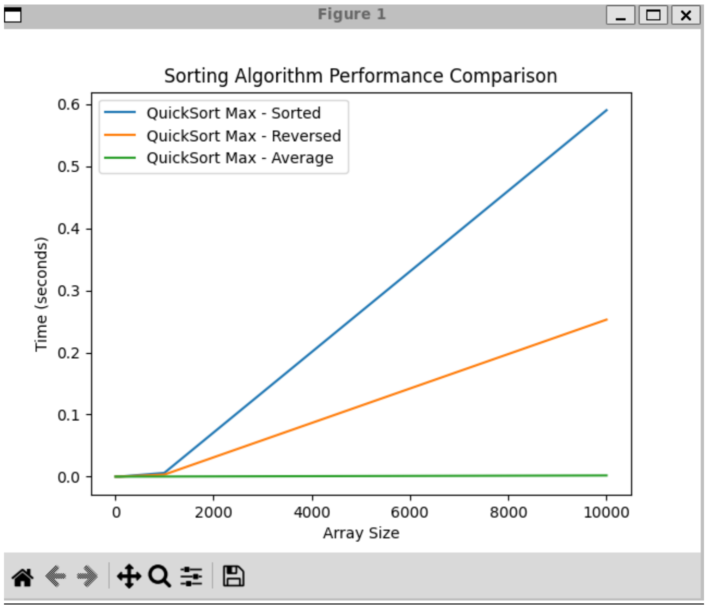


Minimum Pivot:

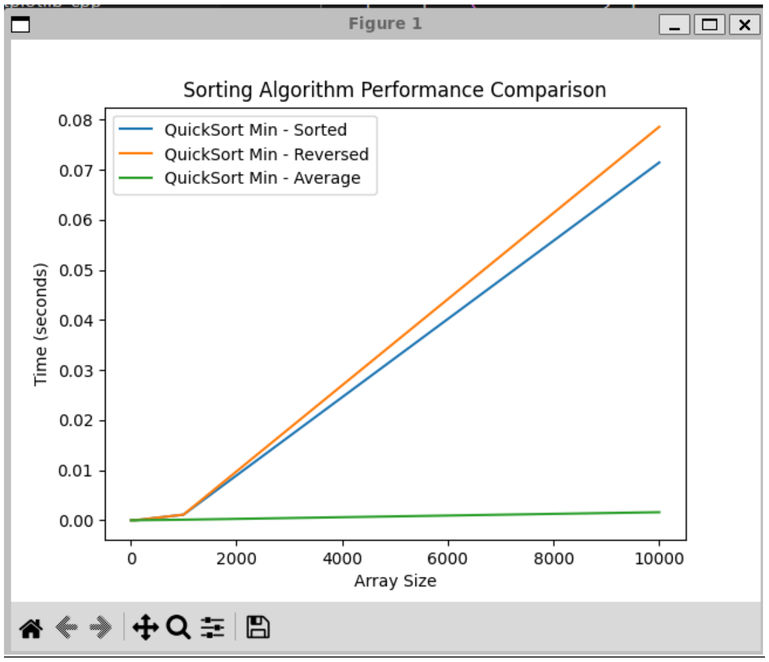


Graph Plotted Using Matplotlib.Cpp:

Maximum Pivot:



Minimum Pivot:



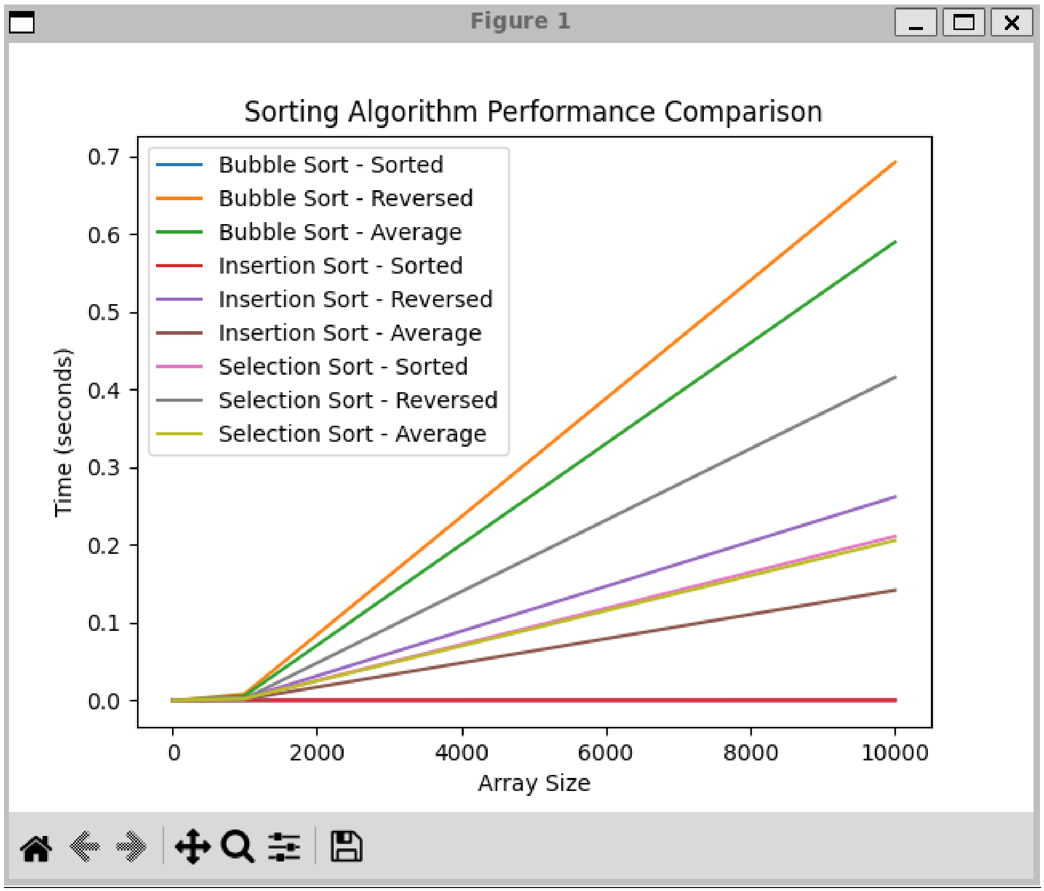
Key Observations:

- Sorted Data:
  - When sorted, Min Pivot always leads to a completely unbalanced partition, resulting in O(n^2) performance.
  - Max Pivot performs similarly for sorted data.
- Reversed Data:
  - Min Pivot performs poorly with reversed data O(n^2).
  - Max Pivot also struggles, as it consistently chooses the worst possible pivot.
- Average Data:
  - For random inputs, partitions are not as heavily skewed, and the average-case performance for both Min and Max pivots can closely approachO(nlogn), explaining why the graph appears close to the x-axis.

Conclusion:

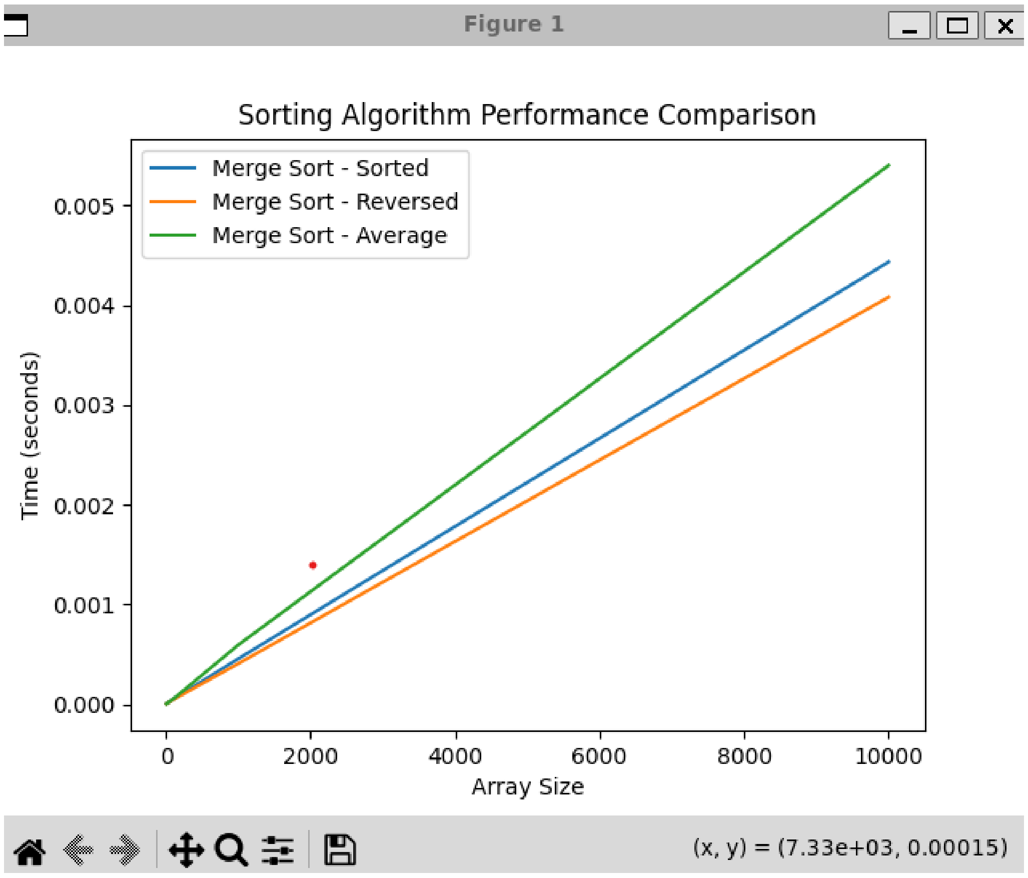
Algorithm	Best Case	Worst Case	Average Case	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
QuickSort (Random)	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(\log n)$
QuickSort (Min Pivot)	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(\log n)$
QuickSort (Max Pivot)	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(\log n)$
QuickSort (Median of Medians)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$

Sorting Algorithm Comparison for Bubble Selection and Inserton Sorts:



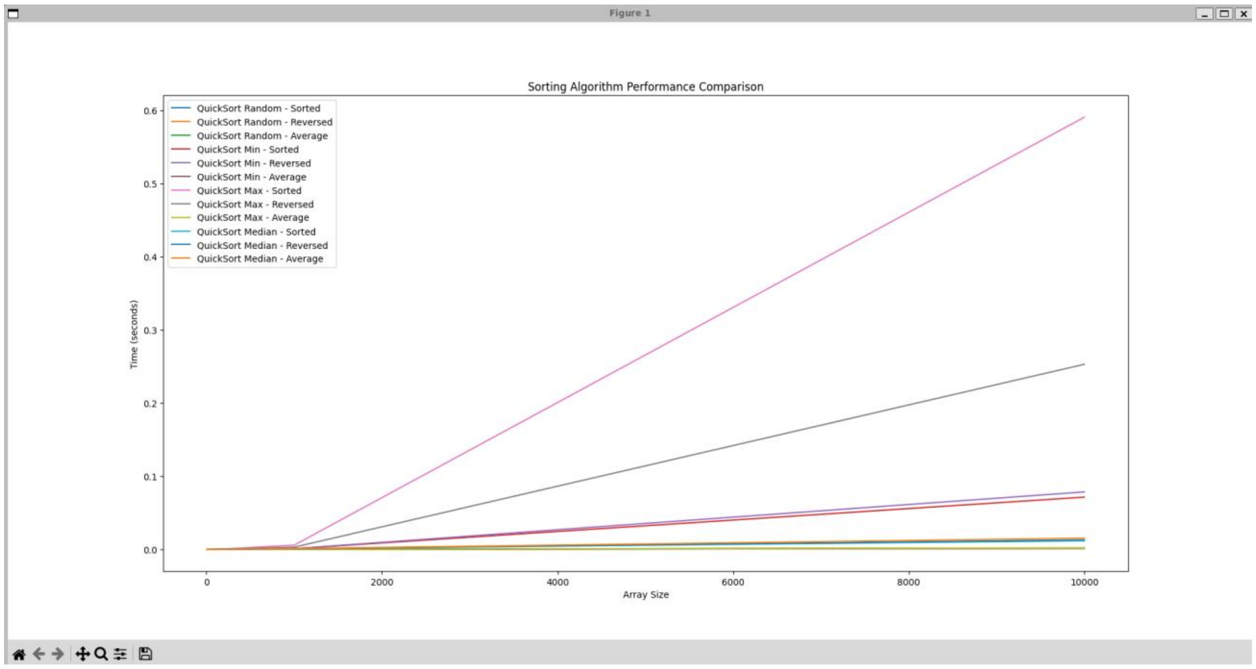
Insertion Sort and Bubble sort gives O(n) for best time complexity and O(n^2) for worst time complexity and average time complexity whereas Selection sort gives O(n^2) for all time complexities.

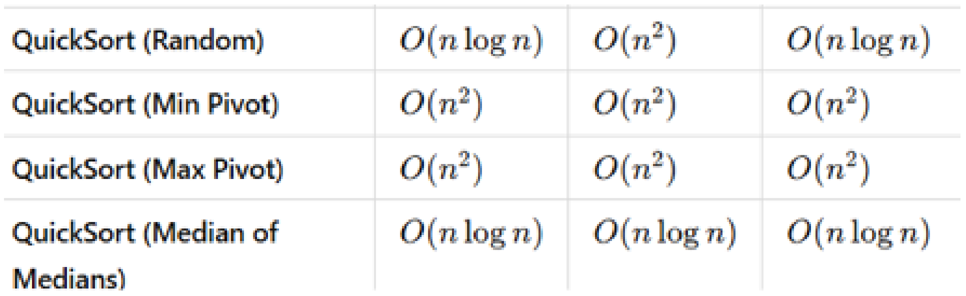
Merge Sort:



Merge Sort always gives O(nlogn) for all the time Complexities.

<u>Quick Sort Including all the conisderations(Max, Min, Median of Medians, Random):</u>





QuickSort (Random)	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
QuickSort (Min Pivot)	$O(n^2)$	$O(n^2)$	$O(n^2)$
QuickSort (Max Pivot)	$O(n^2)$	$O(n^2)$	$O(n^2)$
QuickSort (Median of Medians)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$