Comparison-based Sorting Algorithms

REPORT

PROJECT 1

ITCS 6114 - Algorithms and Data Structures

DEPARTMENT OF COMPUTER SCIENCE

SUBMITTED TO
Dewan T. Ahmed, Ph.D.

SUBMITTED BY:

Mukesh Dasari	801208218
Yogesh Narigapalli	801193142

UNC CHARLOTTE
College of Computing and Informatics

Table of Contents

INTRODUCTION:

We used five different sorting algorithms to test output for various input sizes.

1. Insertion Sort
2. Merge Sort
3. Quick Sort
4. Modified Quick Sort
    a. Median-of-three as pivot.
    b. For small sub problem size (≤ 10), insertion sort is used.
5. Heap Sort [vector based, and insert one item at a time]

Execution Information:

1. We have run sorting algorithms for different input sizes namely "100, 500, 1000, 2000, 5000, 10000, 20000, 40000, 60000".
2. Also, we have generated a vector with random numbers and passed it as input to our sorting algorithms to record the execution time for each sorting algorithm.
3. After running the code for three different vectors of random numbers we have plotted them in a graph
4. We have also observed the performance for two special cases i.e
    a. Input vector is sorted.
    b. Input vector is reversely sorted.

Running code:

The repository contains the `main.cpp` file which is the entry point of the project and different cpp files for respective sorting algorithms. First using g++ we need to create the object `` `/usr/bin/g++ -o main main.cpp` `` and binary file and it will generate the `main` (executable) file. For running use

```
`./main <vector_size>`
Ex. ./main 100
```

Here we take vector_size which can be used to generate a vector of length vector_size and then generate a random number to insert into it.

Insertion Sort:

Description: Insertion sort is a simple sorting algorithm that virtually divides vector into two sub vectors. Left sub vector is sorted and the right sub vector is the unsorted. For each element in the right sub vector we find its position in the left sub vector by iterating until we find an element is less than the each element of the left sub vector.

Program:
```
std::vector<int> insertionSort(std::vector<int> vec){
    int i;
    for(int j=1;j<vec.size();j++){
        i = j-1;
        int key = vec[j];
        while(i>=0 && key<vec[i]){
            vec[i+1] = vec[i];
            i--;
        }
        vec[i+1] = key;
    }
    return vec;
}
```

Complexity Analysis:
Insertion sort is a recommended sorting algorithm for smaller input size of vectors. As it uses two loops to iterate through all the elements and finds its position in the left side sorted vector the time complexity is O(n^2). And as sporting is done in the same vector the space complexity is O(1).

Merge Sort:

Description: Merge Sort is a Divide and Conquer algorithm. It divides the input vector into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(vec, leftVec, rightVec) is a key process that assumes that vec[leftVec....m] and arr[m+1....rightVec] are sorted and merges the two sorted sub vectors into one.

Program:
```
std::vector<int> merge(std::vector<int> leftVec, std::vector<int> rightVec){
    std::vector<int> sortedVec;
    int i=0, j=0;
    while(leftVec.size()>0 && rightVec.size()>0){
        if(leftVec[0]<rightVec[0]){
            sortedVec.push_back(leftVec[0]);
            leftVec.erase(leftVec.begin());
        } else{
            sortedVec.push_back(rightVec[0]);
            rightVec.erase(rightVec.begin());
        }
    }
    while(i<leftVec.size()){
        sortedVec.push_back(leftVec[i]);
        i++;
    }
    while(j<rightVec.size()){
        sortedVec.push_back(rightVec[j]);
        j++;
    }
    return sortedVec;
}

std::vector<int> mergeSort(std::vector<int> vec){
    int vecLength = vec.size();
    if (vecLength == 1) return vec;
    std::vector<int> leftVec, rightVec;
    for(int i=0;i<vecLength/2;i++){
        leftVec.push_back(vec[i]);
    }
    for(int j=vecLength/2;j<vecLength;j++){
        rightVec.push_back(vec[j]);
    }
    leftVec = mergeSort(leftVec);
    rightVec = mergeSort(rightVec);
    return merge(leftVec, rightVec);
}
```

Complexity Analysis:

In merge sort, to each recursive call we divide the vector in half hence the time complexity becomes O(log(n)). And the work done at height i is O(n) so adding both the time complexity of merge sort becomes O(nlog(n)).

Quick Sort (In-Place):

Description: QuickSort is also a Divide and Conquer algorithm. It picks an element as pivot and partitions the given vector around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.

4. Pick median as pivot.

Here we used first approach in which we select pivot as the first element. The key process in quickSort is partition(). Target of partitions is, given an vector and an element x of vector as pivot, put x at its correct position in sorted vector and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Program:

```
std::vector<int> inPlacePartition(std::vector<int> &vec, int left, int right,
int pivote){
    int pivoteValue = vec[pivote];
    int i = 0, j = vec.size() - 1;
    while (i <= j){
        if (vec[i] <= vec[pivote])
            i++;
        if (vec[j] > vec[pivote])
            j--;
        if (vec[i] > vec[pivote] && vec[j] < vec[pivote]){
            int temp = vec[i];
            vec[i] = vec[j];
            vec[j] = temp;
            i++;
            j--;
        }
    }
    int pTemp = vec[pivote];
    vec[pivote] = vec[j];
    vec[j] = pTemp;
    std::vector<int> result;
    result.push_back(i);
    result.push_back(j);
    return result;
}

std::vector<int> inPlaceQuickSort(std::vector<int> &vec, int left, int
right){
    if (left >= right)
        return vec;
    int pivoteIndex = left + std::rand() % (left + (right - left));
    std::vector<int> values = inPlacePartition(vec, left, right, left);
    int i = values[0], j = values[1];
    inPlaceQuickSort(vec, left, i - 1);
    inPlaceQuickSort(vec, j + 1, right);
    return vec;
}
```

Complexity Analysis: The worst-case time complexity of Quick Sort is O(n^2), and the worst-case space complexity is O(log(n)). Quick sort in order sorts the list in its own place. It isn't in a good spot. For large input sizes, quick sort performed admirably. However, in cases where the list has already been sorted or reversed, the time waste and recursions are excessive.

Modified Quick Sort:

Description: In modified Quick Sort here we are using median of three as pivot for better results and if the problem size is less than or equal to 10 insertion sort will be used. We compare left, right and center values and swap elements if needed to get the pivot at middle which satisfies the condition vec[left] <= vec[center] < vec[right].

Program:
```
void quickInsertionSort(std::vector<int> &vec, int left, int right)
{
    int i;
    for (int j = left; j <= right; j++){
        i = j - 1;
        int key = vec[j];
        while (i >= 0 && key < vec[i]){
            vec[i + 1] = vec[i];
            i--;
        }
        vec[i + 1] = key;
    }
    return;
}

void medianQuickSort(std::vector<int> &vec, int left, int right){
    if(left+10 <= right){
        int center = (left+right)/2;
        int pivote;
        int temp;
        if (vec[center] < vec[left] && vec[center] < vec[right]){
            temp = vec[left];
            vec[left] = vec[center];
            if (vec[left] < vec[right])
                vec[center] = temp;
            if (vec[left] > vec[right]){
                vec[center] = vec[right];
                vec[right] = temp;
            }
        }
        if (vec[center] > vec[left] && vec[center] > vec[right]){
            temp = vec[right];
            vec[right] = vec[center];
            if (vec[left] < vec[right])
                vec[center] = temp;
            if (vec[left] > vec[right]){
                vec[center] = vec[left];
                vec[left] = temp;
            }
        }
        pivote = vec[center];

        int i,j;
        for(i=left, j=right-2;;){
            while(vec[i]<pivote) i++;
            while(vec[j]>pivote) j--;
            if(i<j){
                int temp = vec[i];
                vec[i] = vec[j];
                vec[j] = temp;
            } else{
                break;
            }
        }
        int swap = vec[i];
        vec[i] = vec[right-1];
        vec[right-1] = swap;

        medianQuickSort(vec, left ,i - 1);
        medianQuickSort(vec, i, right);
        return;
    }
    else{
        quickInsertionSort(vec, left, right);
        return;
    }
}

std::vector<int> modifiedQuickSort(std::vector<int> &vec){
    std::vector<int> sortedVec;
    medianQuickSort(vec,  0, vec.size() - 1);
    return vec;
}
```

Complexity Analysis:

In a modified quick sort, we traverse in the same vector so the space complexity is O(1). In algorithm, we divide vector in half and use recursion so it is O(log(n)) and each time partitions work in linear so complexity becomes O(n). So total time complexity becomes O(nlog(n)).

Heap Sort:

Description: Heap sort is a comparison based sorting technique based on Binary Heap data structure. First we generate a vector based min heap tree and then remove each element from start which will be the smallest one as per the definition of min heap. We use heapify after the element is removed from the start to satisfy the property if min heap.

Program:

```
void heapify(std::vector<int>& minHeapVec, int lastNode){
    int i = 1;
    while(i < lastNode){
        if((2*i+1) <= lastNode){
            int currentNode = minHeapVec[i];
            int leftNode = minHeapVec[2*i];
            int rightNode = minHeapVec[(2*i)+1];
            if(currentNode<leftNode && currentNode<rightNode) return;
            else if(leftNode<=rightNode && currentNode>leftNode){
                    int temp = minHeapVec[2*i];
                    minHeapVec[2*i] = minHeapVec[i];
                    minHeapVec[i] = temp;
                    i = 2*i;
            } else if(rightNode<leftNode && currentNode>rightNode){
                int temp = minHeapVec[(2*i)+1];
                minHeapVec[(2*i)+1] = minHeapVec[i];
                minHeapVec[i] = temp;
                i = (2*i)+1;
            }
        } else if(2*i <= lastNode && minHeapVec[i]>minHeapVec[2*i]){
            int temp = minHeapVec[2*i];
            minHeapVec[2*i] = minHeapVec[i];
            minHeapVec[i] = temp;
            i = 2*i;
        } else{
            return;
        }
    }
}
```

```
std::vector<int> heapSort(std::vector<int> vec){
    std::vector<int> minHeapVec;
    minHeapVec.push_back(0);
    minHeapVec.push_back(vec[0]);
    for(int i=1; i<vec.size();i++){
        minHeapVec.push_back(vec[i]);
        int lastNode = i+1;
        while(lastNode>0 && minHeapVec[lastNode]<minHeapVec[lastNode/2]){
            int temp = minHeapVec[lastNode];
            minHeapVec[lastNode] = minHeapVec[lastNode/2];
            minHeapVec[lastNode/2] = temp;
            lastNode = lastNode/2;
        }
    }

    for(int j=0;j<minHeapVec.size()-1;j++){
        vec[j]=minHeapVec[1];
        minHeapVec[1] = minHeapVec[minHeapVec.size()-j-1];
        heapify(minHeapVec,minHeapVec.size()-j-2);
    }
    return vec;
}
```

Complexity Analysis:

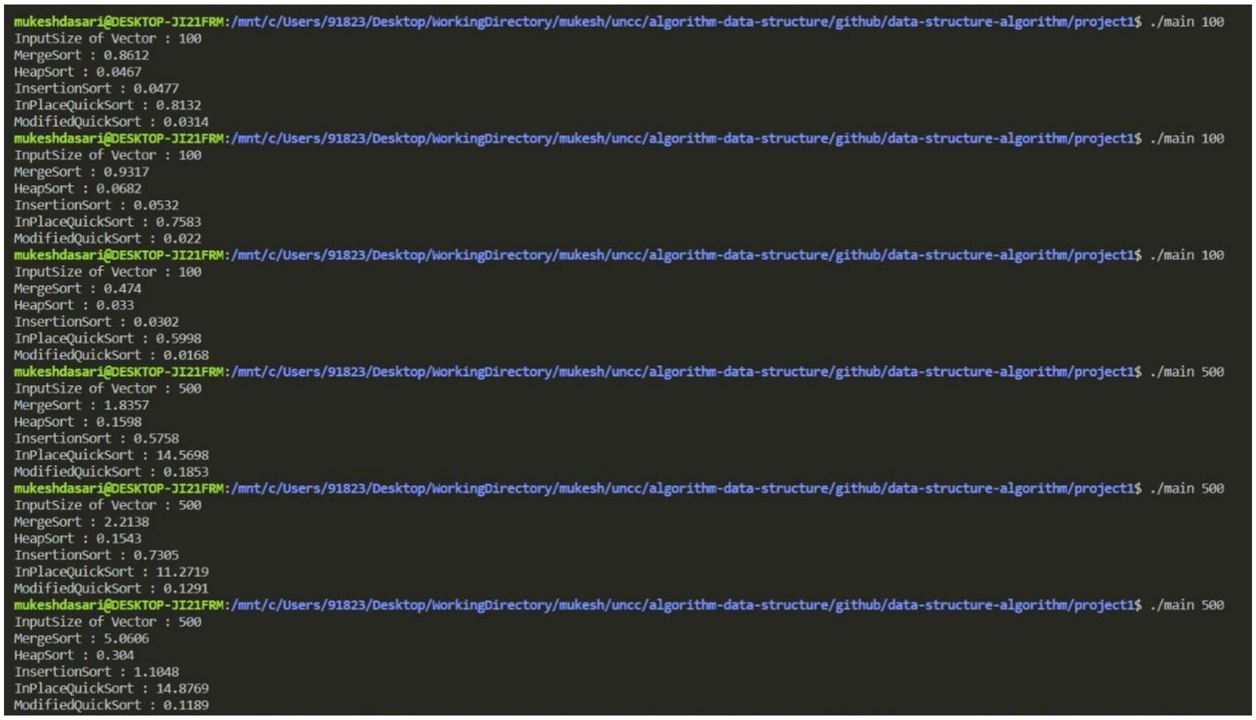
Here in the heapSort function we are creating a vector based heap by inserting each element in the vector and performing heapify in which we compare it with its parent value. So creating a vector based takes O(log(n)). And after once the vector is ready with properties of heap, we remove the first element from the vector which is the smallest once according to min heap properties. But after removing the element we again perform heapify in which from root to its childs we traverse and the worst case of it is O(n). So, the overall time complexity of an algorithm is O(nlog(n)). And it uses vector to store elements and doesn't use any other arbitrary data structure so the space complexity is O(1).

Random Number Generator:

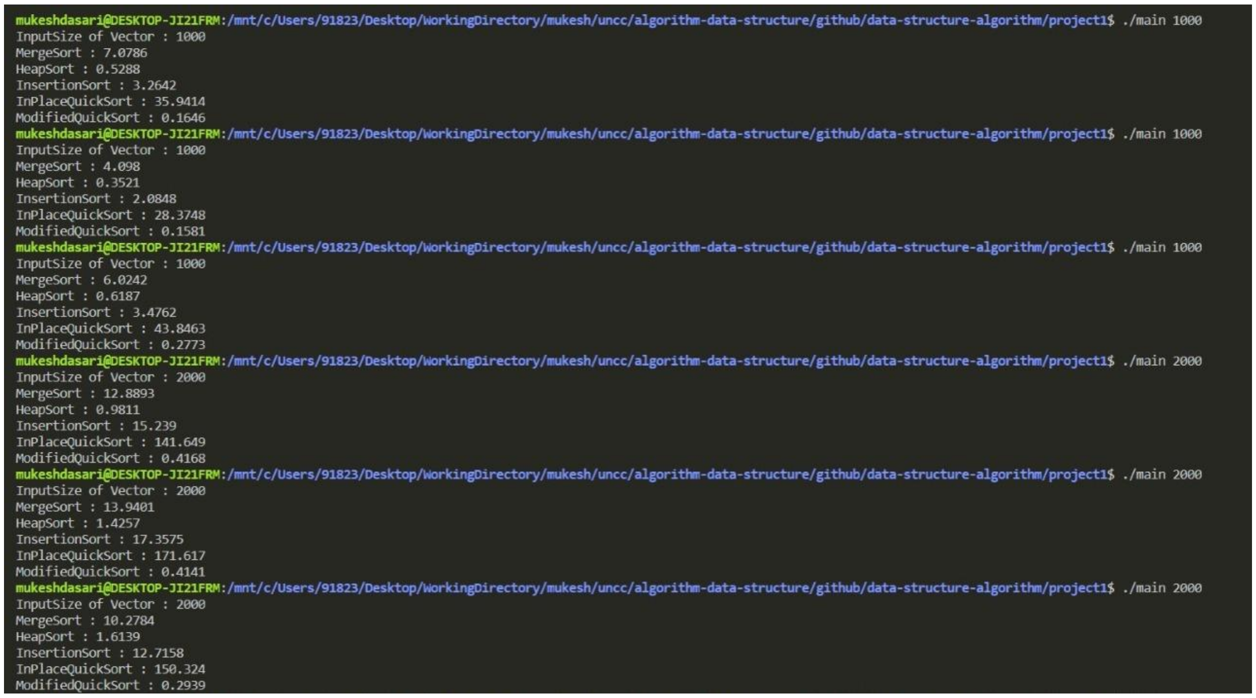
To generate vector with random numbers we have used std::srand to generate seed and using std::rand we generate random numbers and store it in respective vector. And the same vector values are passed to all sorting algorithms to measure its performance.

```
std::vector<int> inputVec1(inputSize,0);
std::srand(std::time(0));
std::generate(inputVec1.begin(),inputVec1.end(),std::rand);
```

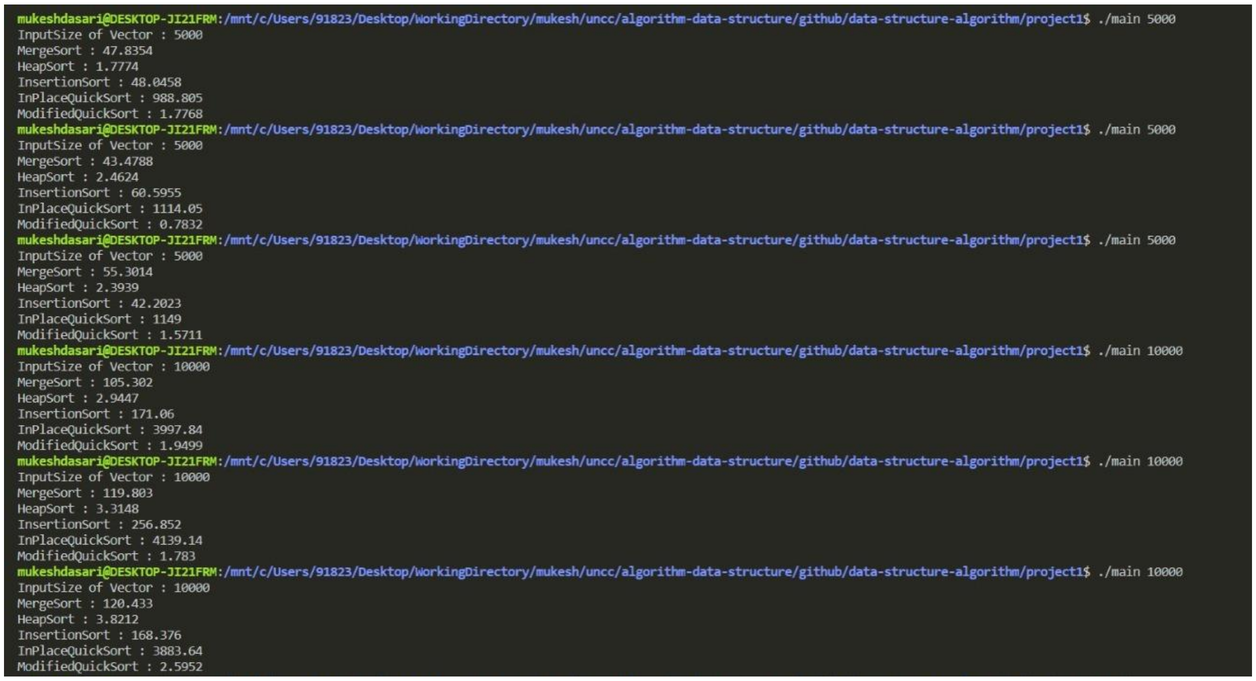
<u>Output:</u>



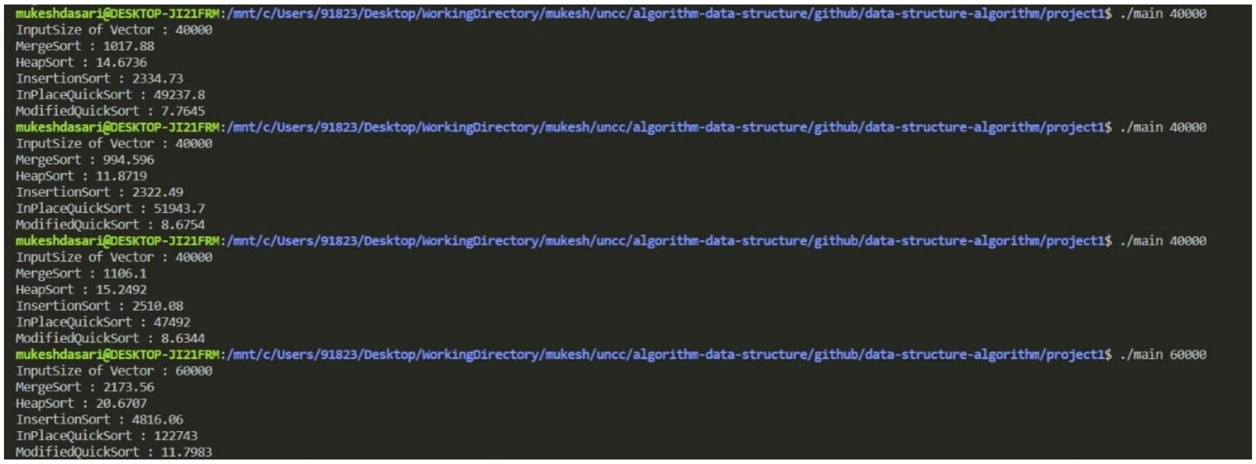
```
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 100
InputSize of Vector : 100
MergeSort : 0.8612
HeapSort : 0.0467
InsertionSort : 0.0477
InPlaceQuickSort : 0.8132
ModifiedQuickSort : 0.0314
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 100
InputSize of Vector : 100
MergeSort : 0.9317
HeapSort : 0.0682
InsertionSort : 0.0532
InPlaceQuickSort : 0.7583
ModifiedQuickSort : 0.022
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 100
InputSize of Vector : 100
MergeSort : 0.474
HeapSort : 0.033
InsertionSort : 0.0302
InPlaceQuickSort : 0.5998
ModifiedQuickSort : 0.0168
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 500
InputSize of Vector : 500
MergeSort : 1.8357
HeapSort : 0.1598
InsertionSort : 0.5758
InPlaceQuickSort : 14.5698
ModifiedQuickSort : 0.1853
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 500
InputSize of Vector : 500
MergeSort : 2.2138
HeapSort : 0.1543
InsertionSort : 0.7305
InPlaceQuickSort : 11.2719
ModifiedQuickSort : 0.1291
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 500
InputSize of Vector : 500
MergeSort : 5.0606
HeapSort : 0.304
InsertionSort : 1.1048
InPlaceQuickSort : 14.8769
ModifiedQuickSort : 0.1189
```



```
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 1000
InputSize of Vector : 1000
MergeSort : 7.0786
HeapSort : 0.5288
InsertionSort : 3.2642
InPlaceQuickSort : 35.9414
ModifiedQuickSort : 0.1646
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 1000
InputSize of Vector : 1000
MergeSort : 4.098
HeapSort : 0.3521
InsertionSort : 2.0848
InPlaceQuickSort : 28.3748
ModifiedQuickSort : 0.1581
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 1000
InputSize of Vector : 1000
MergeSort : 6.0242
HeapSort : 0.6187
InsertionSort : 3.4762
InPlaceQuickSort : 43.8463
ModifiedQuickSort : 0.2773
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 2000
InputSize of Vector : 2000
MergeSort : 12.8893
HeapSort : 0.9811
InsertionSort : 15.239
InPlaceQuickSort : 141.649
ModifiedQuickSort : 0.4168
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 2000
InputSize of Vector : 2000
MergeSort : 13.9401
HeapSort : 1.4257
InsertionSort : 17.3575
InPlaceQuickSort : 171.617
ModifiedQuickSort : 0.4141
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 2000
InputSize of Vector : 2000
MergeSort : 10.2784
HeapSort : 1.6139
InsertionSort : 12.7158
InPlaceQuickSort : 150.324
ModifiedQuickSort : 0.2939
```



```
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 5000
InputSize of Vector : 5000
MergeSort : 47.8354
HeapSort : 1.7774
InsertionSort : 48.0458
InPlaceQuickSort : 988.805
ModifiedQuickSort : 1.7768
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 5000
InputSize of Vector : 5000
MergeSort : 43.4788
HeapSort : 2.4624
InsertionSort : 60.5955
InPlaceQuickSort : 1114.05
ModifiedQuickSort : 0.7832
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 5000
InputSize of Vector : 5000
MergeSort : 55.3014
HeapSort : 2.3939
InsertionSort : 42.2023
InPlaceQuickSort : 1149
ModifiedQuickSort : 1.5711
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 10000
InputSize of Vector : 10000
MergeSort : 105.302
HeapSort : 2.9447
InsertionSort : 171.06
InPlaceQuickSort : 3997.84
ModifiedQuickSort : 1.9499
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 10000
InputSize of Vector : 10000
MergeSort : 119.803
HeapSort : 3.3148
InsertionSort : 256.852
InPlaceQuickSort : 4139.14
ModifiedQuickSort : 1.783
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 10000
InputSize of Vector : 10000
MergeSort : 120.433
HeapSort : 3.8212
InsertionSort : 168.376
InPlaceQuickSort : 3883.64
ModifiedQuickSort : 2.5952
```



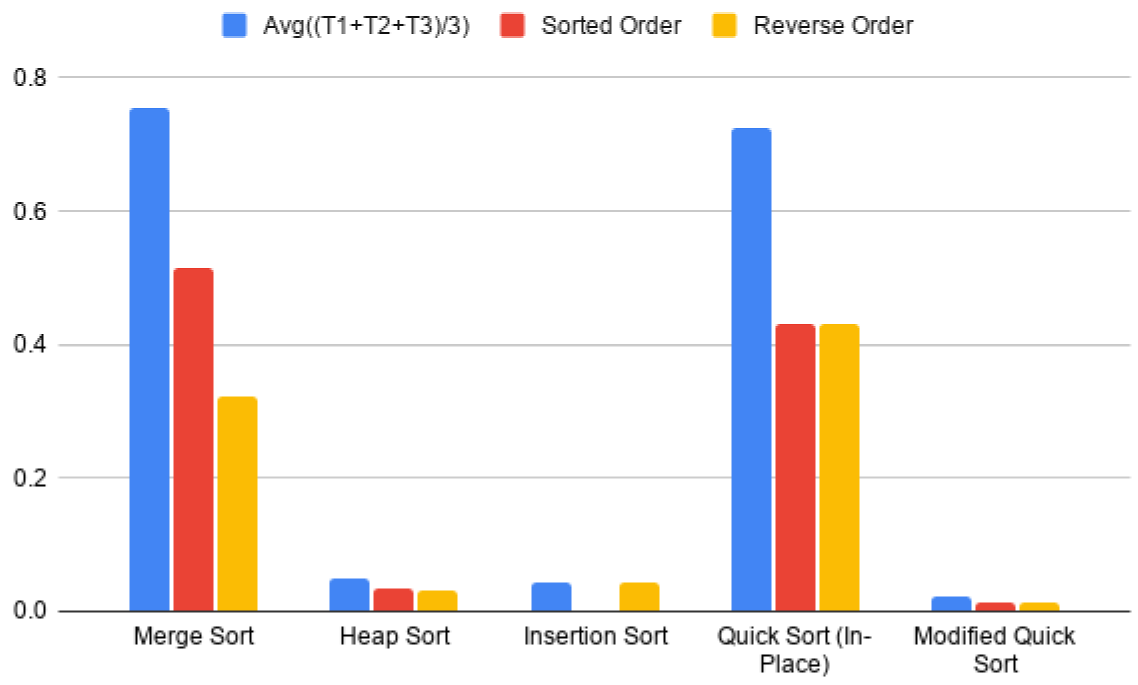
```
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 40000
InputSize of Vector : 40000
MergeSort : 1017.88
HeapSort : 14.6736
InsertionSort : 2334.73
InPlaceQuickSort : 49237.8
ModifiedQuickSort : 7.7645
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 40000
InputSize of Vector : 40000
MergeSort : 994.596
HeapSort : 11.8719
InsertionSort : 2322.49
InPlaceQuickSort : 51943.7
ModifiedQuickSort : 8.6754
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 40000
InputSize of Vector : 40000
MergeSort : 1106.1
HeapSort : 15.2492
InsertionSort : 2510.08
InPlaceQuickSort : 47492
ModifiedQuickSort : 8.6344
mukeshdasari@DESKTOP-JI21FRM:/mnt/c/Users/91823/Desktop/WorkingDirectory/mukesh/uncc/algorithm-data-structure/github/data-structure-algorithm/project1$ ./main 60000
InputSize of Vector : 60000
MergeSort : 2173.56
HeapSort : 20.6707
InsertionSort : 4816.06
InPlaceQuickSort : 122743
ModifiedQuickSort : 11.7983
```

<u>Graphs for various Input sizes:</u>

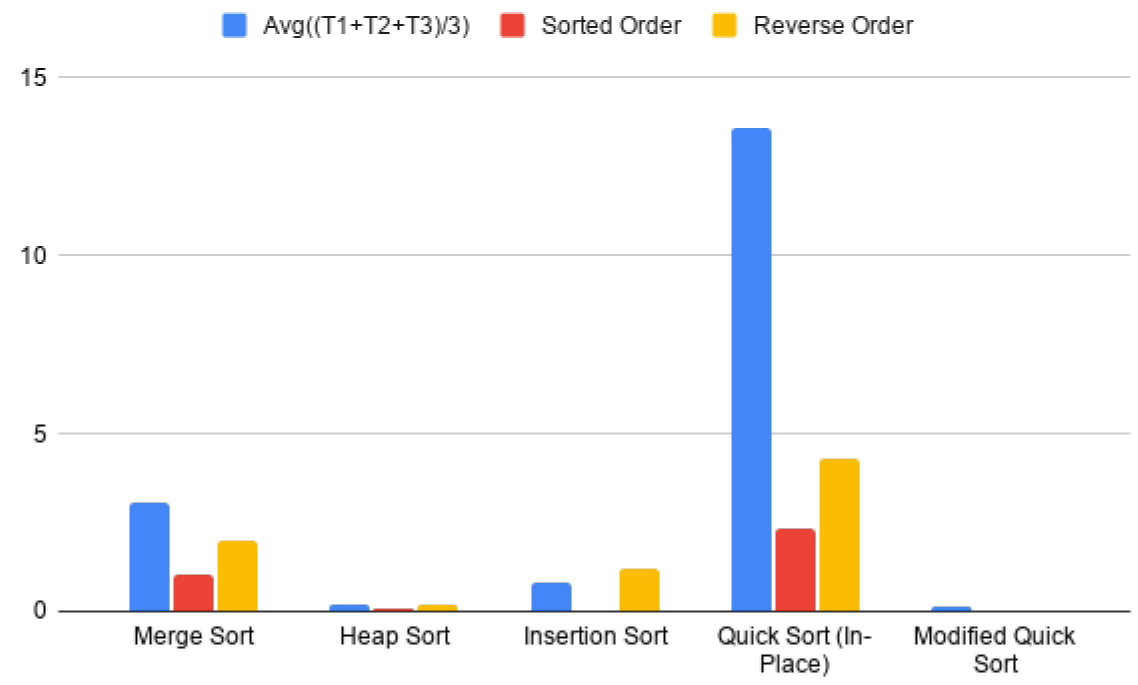
Note: We have taken average of output for three different set of random numbers (Time1,Time2,Time3) and then plotted them as per the three different conditions:

1. 1)If array is Unsorted(Unsorted Array)
2. 2)If array is Sorted(Sorted Array)
3. 3)If array is Reversely Sorted(Reversely Sorted)

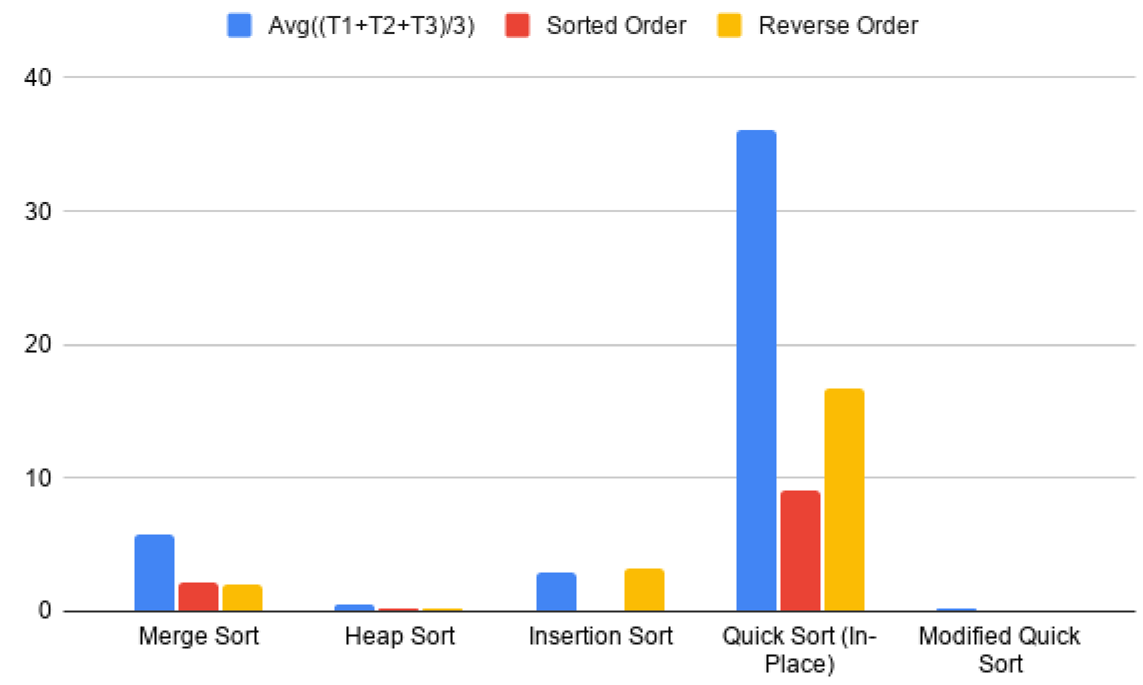
1)For input size = 100 :



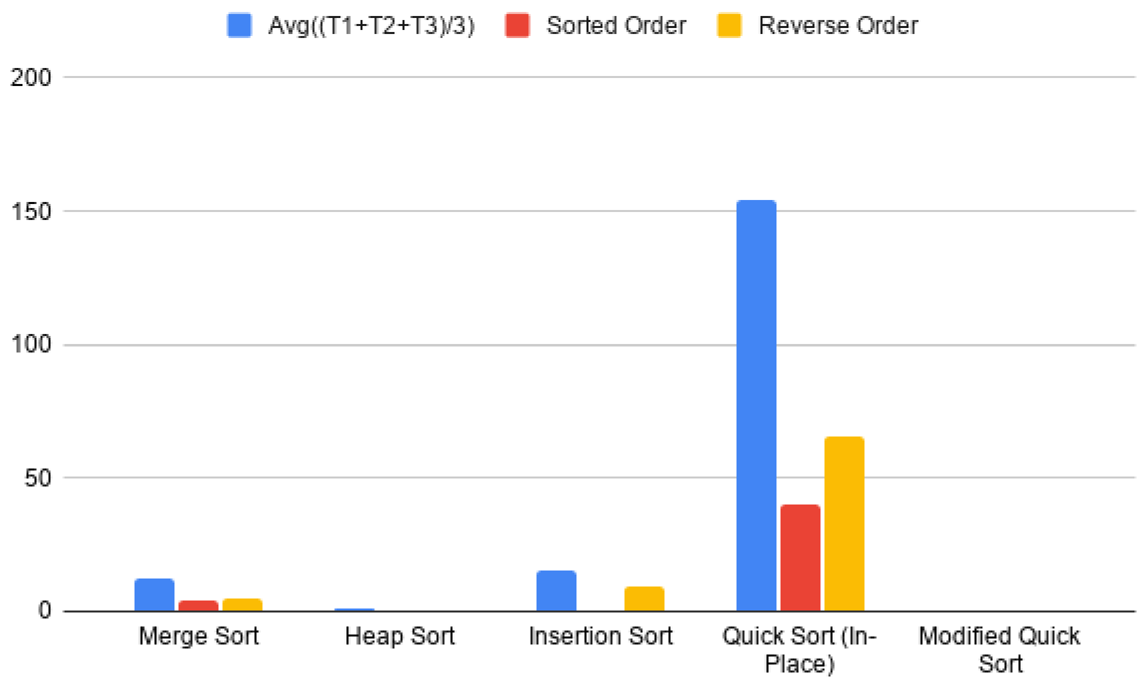
2)For input size = 500 :



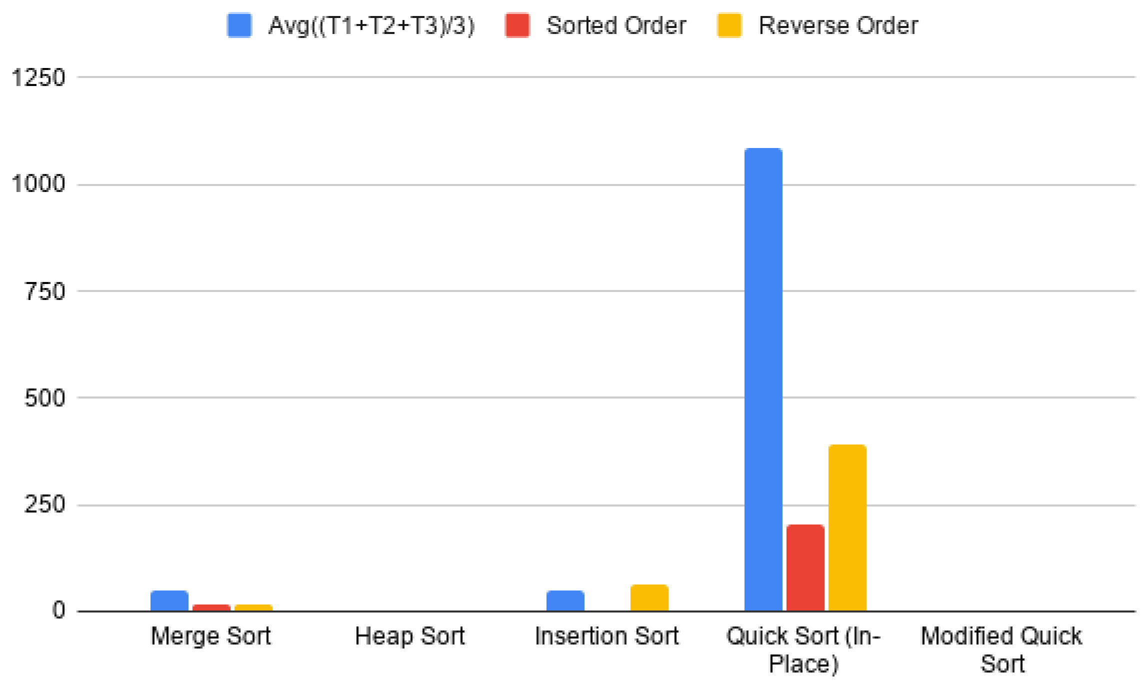
3)For input size = 1000 :



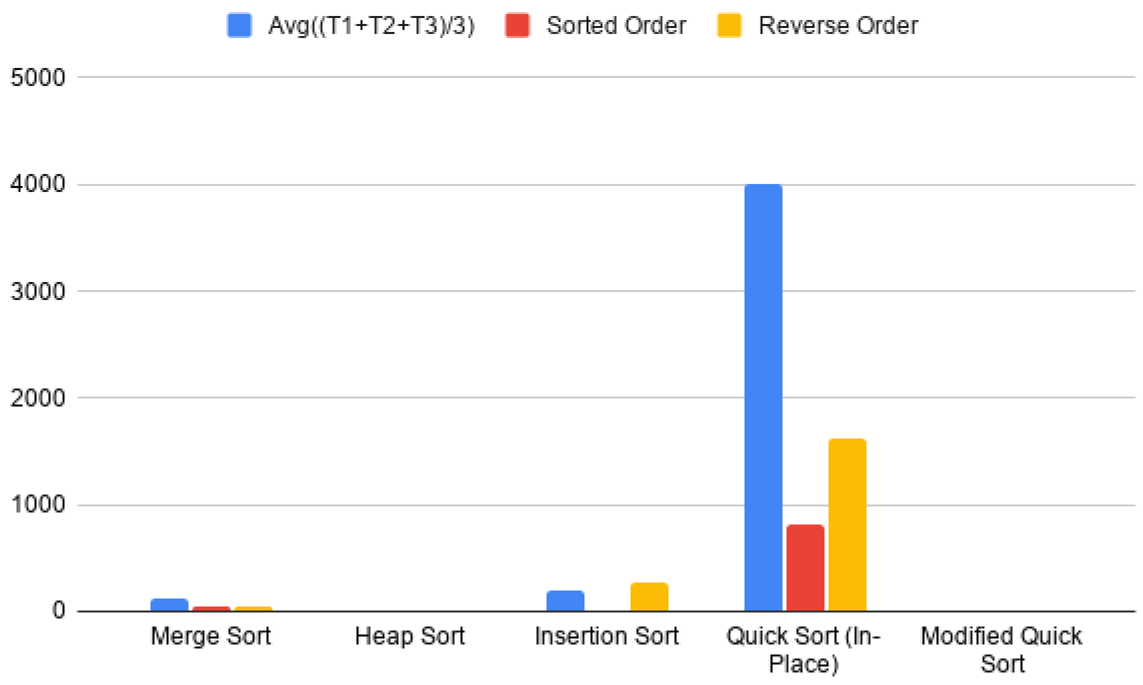
4)For input size = 2000 :



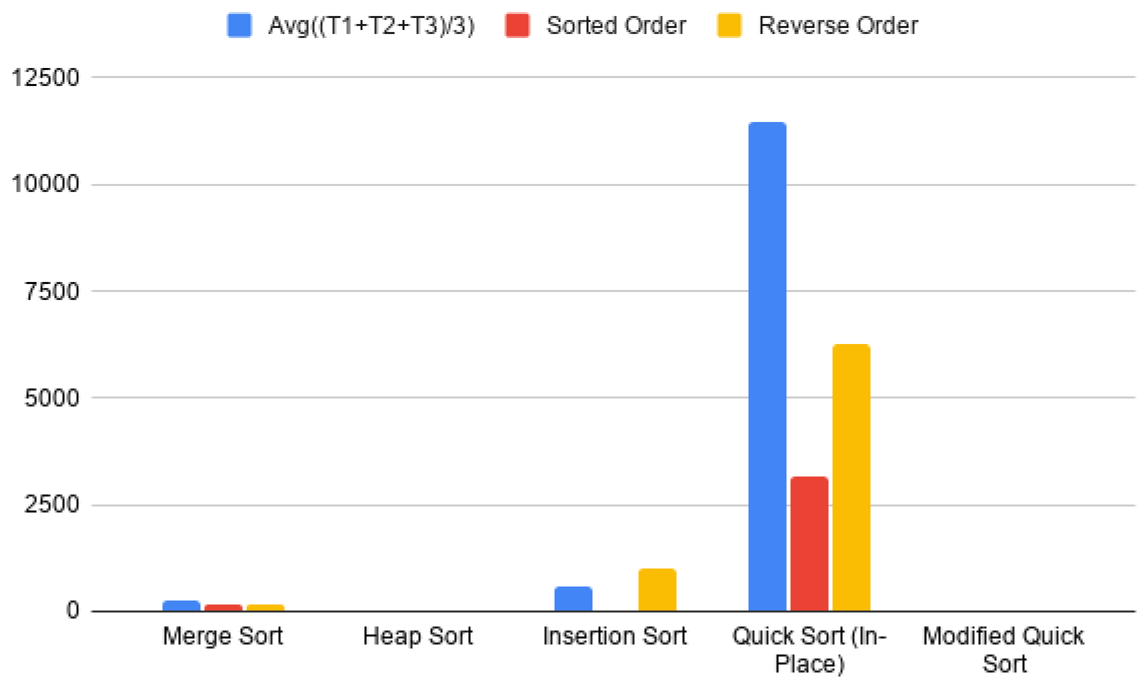
5)For input size = 5000 :



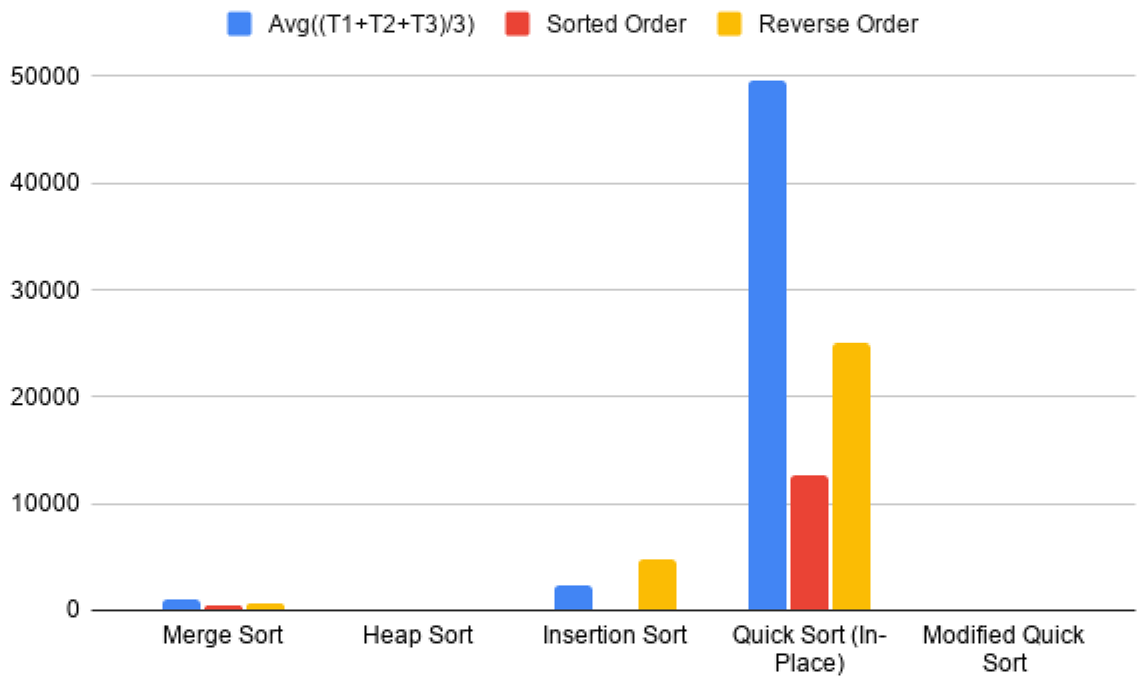
6)For input size = 10000 :



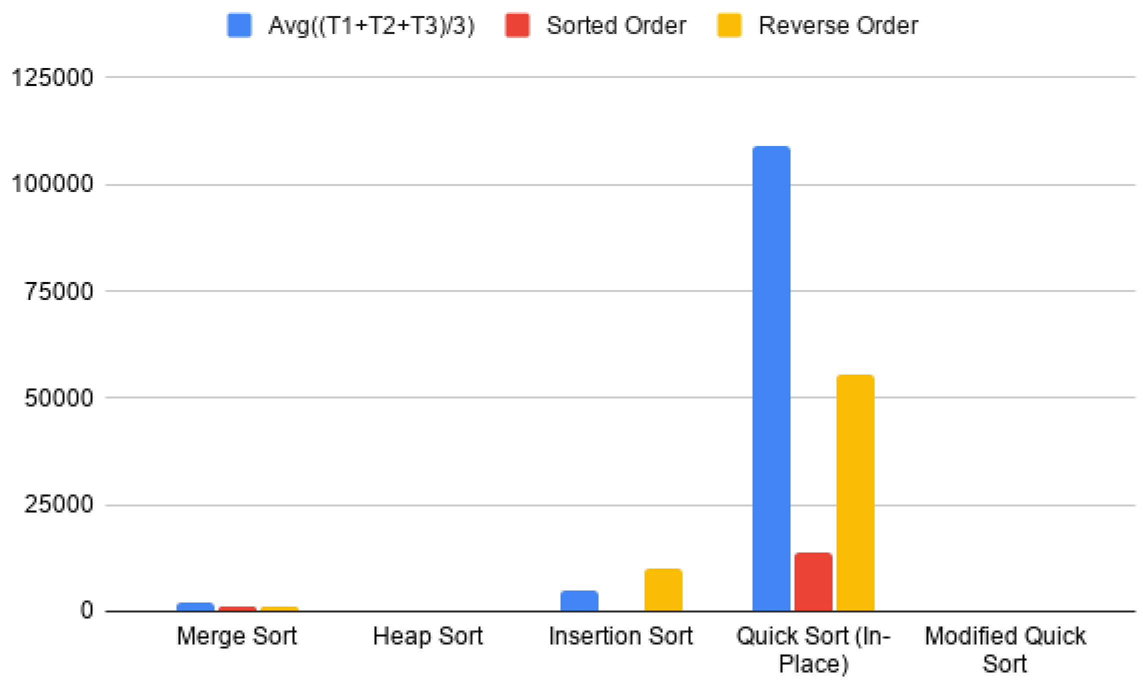
7)For input size = 20000 :



8)For input size = 40000 :



9)For input size = 60000 :


Avg((T1+T2+T3)/3)
Sorted Order
Reverse Order
Merge Sort
Heap Sort
Insertion Sort
Quick Sort (In-Place)
Modified Quick Sort