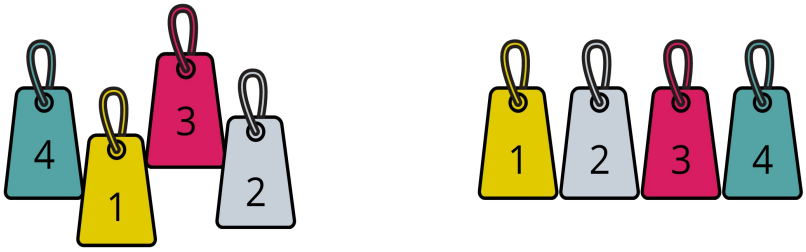
Comaprison-based Sorting Algorithms



REPORT

PROJECT 1

ITCS 6114 - Algorithms and Data Structures

DEPARTMENT OF COMPUTER SCIENCE

SUBMITTED TO
Dewan T. Ahmed, Ph.D.

SUBMITTED BY:

RISHAB SEMLANI	801201960
VARUN JAIN	801168344

UNC CHARLOTTE
College of Computing and Informatics

Table of Contents

INTRODUCTION:

Here we have implemented 5 sorting algorithms to observe performance for different 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 algorithms for different input sizes namely "1000, 2000, 3000, 5000, 10000, 20000, 30000, 40000, 50000, 60000".
2) Also, we have generated an array with random numbers and gave it as input in our code to record the execution time for each sorting technique.
3) After running the code for three different set 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 array is sorted.
    b) Input array is reversely sorted.

Insertion Sort:

About: Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Code:

```
public class sort {
      void insertionSort(Integer arr[]){
      int i=1;
      int n= arr.length;
      while (i<n){
         int element=arr[i];
         int j=i-1;
         boolean largeElementFound=false;
         while (j>=0 && arr[j]>element){
            if (!largeElementFound){
               largeElementFound=true;
            }
            arr[j+1]=arr[j];
            j--;
         }
         if(largeElementFound){
            arr[j+1]=element;
         }
         i++;
      }
   }
```

Observations: Insertion Sort worst case time complexity is O(n^2) and space complexity is O(1). Insertion Sort works pretty good if list is small. Also, it is very fast for sorted list and takes time if the list is reversely sorted.

Merge Sort:

About: Merge Sort is a Divide and Conquer algorithm. It divides the input array 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(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

Code:

```
void mergeSubArray(Integer arr[],int l, int m, int r){
      int size1= m-l+1;
      int size2=r-m;
      int arr1[]= new int[size1];
      int arr2[]= new int[size2];
      for(int i=0;i<arr1.length;i++){
         arr1[i]=arr[l+i];
      }
      for (int i=0;i<arr2.length;i++){
         arr2[i]=arr[m+1+i];
```

```
    }
    int i=0,j=0;
    int k=l;
    while (i<size1 && j<size2){
      if (arr1[i]<=arr2[j]){
        arr[k]=arr1[i];
        i++;
      }else {
        arr[k]=arr2[j];
        j++;
      }
      k++;
    }
    while (i<size1){
      arr[k]=arr1[i];
      i++;
      k++;
    }
    while (j<size2){
      arr[k]=arr2[j];
      j++;
      k++;
    }
  }
  void mergeSort(Integer arr[],int l,int r){
    if (l<r){
      int m= l+((r-l)/2);
      mergeSort(arr,l,m);
      mergeSort(arr,m+1,r);
      mergeSubArray(arr,l,m,r);
    }
  }
```

Observations: Merge Sort worst case time complexity is O(n log(n)) and space complexity is O(n). Merge Sort works well for large data sets.

Quick Sort:

About: QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array 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.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array 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.

Code:

```
int quickSort(Integer arr[],int low, int high){
      int pivot=arr[high];
      int j=low;
      int i=low-1;
      while (j<=high-1){
         if (arr[j]<pivot) {
            i++;
            swap(arr, i, j);
         }
         j++;
      }
      swap(arr,i+1,high);
      return i+1;
   }
    void quicksort(Integer arr[],int l, int r){
      while (l<r){
         int index= quickSort(arr,l,r);
         if (index-l<r-index){
            quicksort(arr,l,index-1);
            l=index+1;
         }else {
            quicksort(arr,index+1,r);
            r=index-1;
         }
      }
   }
    void swap(Integer arr[],int i,int j){
      int temp=arr[i];
      arr[i]=arr[j];
      arr[j]=temp;
    }
```

Observation: Quick Sort worst case time complexity is O(n^2) and space complexity is O(log(n)). In place quick sort sorts the array in its own place. It's not stable. In general quick sort worked pretty well for large input sizes. But in scenarios were list is already sorted or reversely sorted the time consumption and recursions exceeds too much.

Modified Quick Sort:

About: 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.

Code:

```
void modifiedQuickSort(Integer arr[], int l, int h){
      if (l<h){
         int n= h-l+1;
         int med= findMedianOfMedian(arr,l,h, n/2);
         int pos= partition(arr,l,h,med);
         modifiedQuickSort(arr,l,pos-1);
         modifiedQuickSort(arr,pos+1,h);
      }
   }
```

```
// Applying insertion sort in modified quick sort for number of elements <= 10
 int median(Integer arr[], int i, int n){
    //Arrays.sort(arr,i,i+n);
     int s=i;
     int e=i+n;
     while (i<e){
        int element=arr[i];
        int j=i-1;
        boolean largeElementFound=false;
        while (j>=s && arr[j]>element){
           if (!largeElementFound){
              largeElementFound=true;
           }
           arr[j+1]=arr[j];
           j--;
        }
        if(largeElementFound){
           arr[j+1]=element;
        }
        i++;
     }
    return arr[s+(n/2)];
}
// finding median of median to select that element as pivot
 int findMedianOfMedian(Integer arr[],int l, int r,int k){
    if (k>0 && k<=r-l+1){
       int n= r-l+1;
       int i;
       Integer median[]= new Integer[(n+9)/10];
       for (i=0;i<(n/10);i++){
          median[i]= median(arr,l+i*10,10);
       }
       if (i*10<n){
          median[i]=median(arr,l+i*10,n%10);
          i++;
       }
       int medOfMed= i==1?median[i-1]:findMedianOfMedian(median,0,i-1,i/2);
       int pos= partition(arr,l,r,medOfMed);
       if (pos-l==k-1){
          return arr[pos];
       }else if (pos-l>k-1){
          return findMedianOfMedian(arr,l,pos-1,k);
       }else {
          return findMedianOfMedian(arr,pos+1,r,k-pos+l-1);
       }
    }
    return Integer.MAX_VALUE;
}
// Partition for quick sort
 int partition(Integer arr[],int l, int r, int pivot){
    int i;
    for (i=l;i<r;i++){
       if (arr[i]==pivot){
          break;
```

```
            }
        }
        swap(arr,i, r);
        i=l;
        for (int j=l;j<r;j++){
            if (arr[j]<=pivot){
                swap(arr,i,j);
                i++;
            }
        }
        swap(arr,i,r);
        return i;
    }
```

Observation: Modified Quick Sort worst case time complexity is O(n log(n)) and overall space complexity is O(1). Though to calculate medians an array will be used whose space complexity will be O(n+9/10) and there will also be recursion calls which will store the previous state in stack.
Here for a better value of pivot median of median was taken from all elements and that was chosen as pivot. This makes the execution a bit faster compared to normal quick sort. Also since insertion sort was added for input sizes less than or equal to 10, those values were sorted very fast. Quick sort works well for large random data sets.

Heap Sort:

About: Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining elements.

Code:

```
void heapify(MinHeapTree minHeap,int index){
        int left=left(index);
        int right= right(index);
        int min=index;
        if (left<minHeap.size && minHeap.vector.get(min)>minHeap.vector.get(left)){
            min=left;
        }
        if (right<minHeap.size && minHeap.vector.get(min)>minHeap.vector.get(right)){
            min=right;
        }
        if (min!=index){
            int temp= minHeap.vector.get(index);
            minHeap.vector.set(index,minHeap.vector.get(min));
            minHeap.vector.set(min,temp);
            heapify(minHeap,min);
        }
    }
    //Insert Key in Min heap
    void insertKey(MinHeapTree minHeapTree,int key){
        if (minHeapTree.size>=minHeapTree.capacity){
            System.out.println("Heap Size Overflow");
        }
```

```
        minHeapTree.size++;
        int i= minHeapTree.size-1;
        minHeapTree.vector.add(i,key);
        while (i!=0 && minHeapTree.vector.get(parent(i))>minHeapTree.vector.get(i)){
            int temp= minHeapTree.vector.get(parent(i));
            minHeapTree.vector.set(parent(i),minHeapTree.vector.get(i));
            minHeapTree.vector.set(i,temp);
            i=parent(i);
        }
    }
    //Extract Min in Min heap
    public int extractMin(MinHeapTree minHeapTree){
        if (minHeapTree.size<=0){
            return Integer.MAX_VALUE;
        }
        if (minHeapTree.size==1){
            minHeapTree.size--;
            return minHeapTree.vector.get(0);
        }
        int root= minHeapTree.vector.get(0);
        minHeapTree.vector.set(0,minHeapTree.vector.get(minHeapTree.size-1));
        minHeapTree.size--;
        heapify(minHeapTree,0);
        return root;
    }
 int parent(int i){
        return (i-1)/2;
    }
    int left(int i){
        return 2*i+1;
    }
    int right(int i){
        return 2*i+2;
    }
```

Observation: Time complexity of heapify method is O(log(n)). Time complexity of createAndBuildHeap() is O(n) and overall Worst case time complexity is O(n log(n)) and space complexity is O(1). Heap sort is not stable and also has a poor locality of reference. Also Memory management is more complex in heap memory because it's used globally.

Random Number Generator:

Code:

```
        for (int i=0;i<inputSize1.length;i++){
            inputSize1[i]= ThreadLocalRandom.current().nextInt(1,inputSize1.length+1);
        }
        //Copying above generated data in other input arrays for other sorts to perform all sorts on one data
        inputSize1_merge=inputSize1.clone();
        inputSize1_heap=inputSize1.clone();
        inputSize1_quick=inputSize1.clone();
        inputSize1_mq=inputSize1.clone();
        for (int i=0;i<inputSize2.length;i++){
            inputSize2[i]= ThreadLocalRandom.current().nextInt(1,inputSize2.length+1);
```

```
    }
    inputSize2_merge=inputSize2.clone();
    inputSize2_heap=inputSize2.clone();
    inputSize2_quick=inputSize2.clone();
    inputSize2_mq=inputSize2.clone();
    for (int i=0;i<inputSize3.length;i++){
       inputSize3[i]= ThreadLocalRandom.current().nextInt(1,inputSize3.length+1);
    }
    inputSize3_merge=inputSize3.clone();
    inputSize3_heap=inputSize3.clone();
    inputSize3_quick=inputSize3.clone();
    inputSize3_mq=inputSize3.clone();
    for (int i=0;i<inputSize4.length;i++){
       inputSize4[i]= ThreadLocalRandom.current().nextInt(1,inputSize4.length+1);
    }
    inputSize4_merge=inputSize4.clone();
    inputSize4_heap=inputSize4.clone();
    inputSize4_quick=inputSize4.clone();
    inputSize4_mq=inputSize4.clone();
    for (int i=0;i<inputSize5.length;i++){
       inputSize5[i]= ThreadLocalRandom.current().nextInt(1,inputSize5.length+1);
    }
    inputSize5_merge=inputSize5.clone();
    inputSize5_heap=inputSize5.clone();
    inputSize5_quick=inputSize5.clone();
    inputSize5_mq=inputSize5.clone();
    for (int i=0;i<inputSize6.length;i++){
       inputSize6[i]= ThreadLocalRandom.current().nextInt(1,inputSize6.length+1);
    }
    inputSize6_merge=inputSize6.clone();
    inputSize6_heap=inputSize6.clone();
    inputSize6_quick=inputSize6.clone();
    inputSize6_mq=inputSize6.clone();
    for (int i=0;i<inputSize7.length;i++){
       inputSize7[i]= ThreadLocalRandom.current().nextInt(1,inputSize7.length+1);
    }
    inputSize7_merge=inputSize7.clone();
    inputSize7_heap=inputSize7.clone();
    inputSize7_quick=inputSize7.clone();
    inputSize7_mq=inputSize7.clone();
    for (int i=0;i<inputSize8.length;i++){
       inputSize8[i]= ThreadLocalRandom.current().nextInt(1,inputSize8.length+1);
    }
    inputSize8_merge=inputSize8.clone();
    inputSize8_heap=inputSize8.clone();
    inputSize8_quick=inputSize8.clone();
    inputSize8_mq=inputSize8.clone();
    for (int i=0;i<inputSize9.length;i++){
       inputSize9[i]= ThreadLocalRandom.current().nextInt(1,inputSize9.length+1);
    }
    inputSize9_merge=inputSize9.clone();
    inputSize9_heap=inputSize9.clone();
    inputSize9_quick=inputSize9.clone();
    inputSize9_mq=inputSize9.clone();
```

```
for (int i=0;i<inputSize10.length;i++){
    inputSize10[i]= ThreadLocalRandom.current().nextInt(1,inputSize10.length+1);
}
inputSize10_merge=inputSize10.clone();
inputSize10_heap=inputSize10.clone();
inputSize10_quick=inputSize10.clone();
inputSize10_mq=inputSize10.clone();
```

Output:

1) Output for random numbers set 1

Markers | Properties | Servers | Data Source Explorer | Snippets | Console

<terminated> sort [Java Application] C:\Program Files\Java\jdk-15.0.2\bin\javaw.exe (27-Feb-2021, 11:54:19 am – 11:57:12 am)

```
Time Complexity of different Sorting Techniques displayed for following input sizes:
1000 2000 3000 5000 10000 20000 30000 40000 50000 60000
Time Complexity of Insertion Sort if Arrays are not sorted :
18 22 18 45 197 485 1175 2454 3773 5111
Time Complexity of Insertion Sort if Arrays sorted :
0 0 0 0 0 1 0 0 1 0
Time Complexity of Insertion Sort if Arrays reverse sorted :
2 8 16 46 185 1116 1973 3457 5532 8320
Time Complexity of Merge Sort if Arrays are not sorted :
3 2 15 3 8 24 18 16 23 24
Time Complexity of Merge Sort if Arrays sorted :
1 0 1 2 4 7 13 17 24 17
Time Complexity of Merge Sort if Arrays reverse sorted :
1 1 1 1 3 5 8 15 25 20
Time Complexity of Quick Sort if Arrays are not sorted :
1 1 2 2 10 13 10 10 15 17
Time Complexity of Quick Sort if Arrays sorted :
8 51 70 226 881 3678 8137 14585 22318 31726
Time Complexity of Quick Sort if Arrays reverse sorted :
8 31 48 135 537 2104 5423 9135 13629 20300
Time Complexity of Modified Quick Sort if Arrays are not sorted :
5 6 18 35 86 117 42 60 82 91
Time Complexity of Modified Quick Sort if Arrays sorted :
1 1 2 4 8 18 28 35 46 55
Time Complexity of Modified Quick Sort if Arrays reverse sorted :
1 2 3 5 14 30 42 66 69 87
Time Complexity of heap Sort if Arrays are not sorted :
9 6 9 17 36 66 106 141 186 255
Time Complexity of heap Sort if Arrays is already sorted :
2 5 7 13 29 64 127 217 193 211
Time Complexity of heap Sort if Arrays is reversly sorted :
3 7 14 21 47 102 161 259 328 466
```

2) Output for random numbers set 2

```
<terminated> sort [Java Application] C:\Program Files\Java\jdk-15.0.2\bin\javaw.exe  (27-Feb-2021, 1:42:06 pm – 1:44:37 pm)
Time Complexity of different Sorting Techniques displayed for following input sizes:
1000 2000 3000 5000 10000 20000 30000 40000 50000 60000
Time Complexity of Insertion Sort if Arrays are not sorted :
15 35 16 31 163 441 911 1702 2765 4138
Time Complexity of Insertion Sort if Arrays sorted :
0 0 0 0 1 0 0 1 0 1
Time Complexity of Insertion Sort if Arrays reverse sorted :
2 7 16 46 178 720 1440 2835 4498 6766
Time Complexity of Merge Sort if Arrays are not sorted :
2 2 12 3 6 17 16 16 27 27
Time Complexity of Merge Sort if Arrays sorted :
0 1 1 2 4 8 14 17 25 16
Time Complexity of Merge Sort if Arrays reverse sorted :
0 0 0 1 3 5 10 14 26 18
Time Complexity of Quick Sort if Arrays are not sorted :
2 1 1 3 7 16 9 14 15 19
Time Complexity of Quick Sort if Arrays sorted :
8 48 69 222 881 3429 7085 12630 19643 28407
Time Complexity of Quick Sort if Arrays reverse sorted :
5 21 49 157 537 2011 4504 8027 12716 18460
Time Complexity of Modified Quick Sort if Arrays are not sorted :
5 3 9 47 92 109 46 62 76 93
Time Complexity of Modified Quick Sort if Arrays sorted :
1 1 2 4 8 16 27 36 55 73
Time Complexity of Modified Quick Sort if Arrays reverse sorted :
1 2 3 5 12 25 40 54 67 84
Time Complexity of heap Sort if Arrays are not sorted :
8 6 10 21 42 74 119 147 208 230
Time Complexity of heap Sort if Arrays is already sorted :
2 4 8 13 34 69 109 146 213 235
Time Complexity of heap Sort if Arrays is reversly sorted :
3 7 12 20 48 101 162 219 276 341
```

3) Output for random numbers set 3

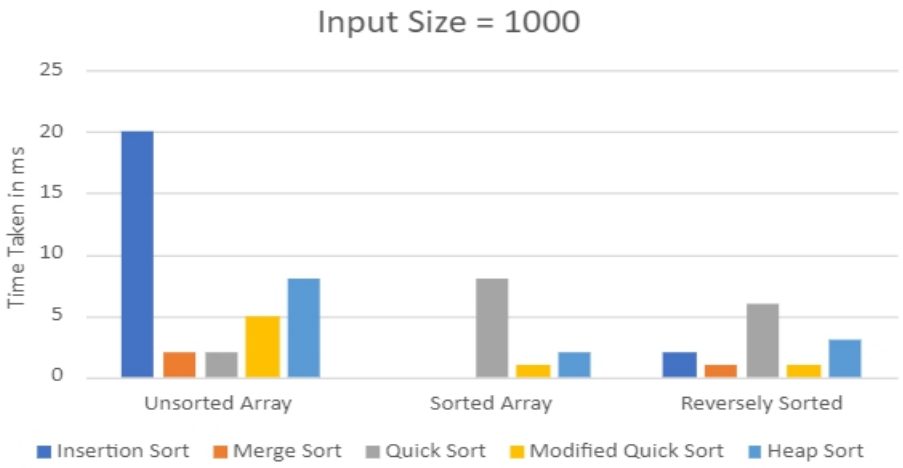
```
<terminated> sort [Java Application] C:\Program Files\Java\jdk-15.0.2\bin\javaw.exe  (27-Feb-2021, 3:15:47 pm – 3:18:10 pm)
Time Complexity of different Sorting Techniques displayed for following input sizes:
1000 2000 3000 5000 10000 20000 30000 40000 50000 60000
Time Complexity of Insertion Sort if Arrays are not sorted :
27 15 43 50 245 381 986 1802 2831 4024
Time Complexity of Insertion Sort if Arrays sorted :
0 0 0 0 0 0 0 1 0 1
Time Complexity of Insertion Sort if Arrays reverse sorted :
2 7 15 45 184 725 1471 2821 4444 6512
Time Complexity of Merge Sort if Arrays are not sorted :
2 2 14 4 11 19 19 18 27 32
Time Complexity of Merge Sort if Arrays sorted :
0 0 2 3 5 10 14 21 30 19
Time Complexity of Merge Sort if Arrays reverse sorted :
1 1 1 1 4 8 8 15 30 20
Time Complexity of Quick Sort if Arrays are not sorted :
2 2 2 3 8 39 15 22 22 21
Time Complexity of Quick Sort if Arrays sorted :
8 51 64 207 784 3154 6401 11442 18912 25793
Time Complexity of Quick Sort if Arrays reverse sorted :
5 21 48 126 501 1975 4431 8023 12419 17643
Time Complexity of Modified Quick Sort if Arrays are not sorted :
4 6 11 24 18 32 49 68 80 93
Time Complexity of Modified Quick Sort if Arrays sorted :
0 1 2 4 8 17 24 46 42 52
Time Complexity of Modified Quick Sort if Arrays reverse sorted :
1 2 3 5 12 25 40 54 70 87
Time Complexity of heap Sort if Arrays are not sorted :
8 5 9 16 34 67 101 141 201 218
Time Complexity of heap Sort if Arrays is already sorted :
2 4 7 13 28 63 96 131 169 206
Time Complexity of heap Sort if Arrays is reversly sorted :
4 8 12 20 48 101 159 236 283 337
```

Graphs for various Input sizes:

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

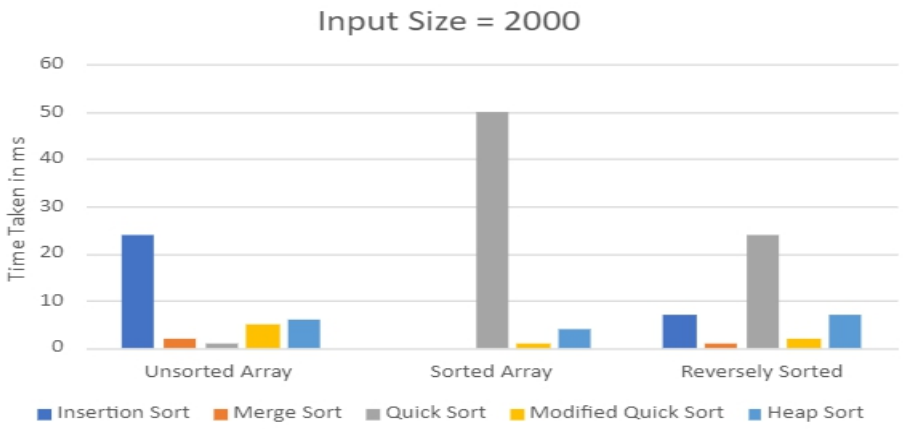
1) For input size = 1000 :

Sorting Technique	Time1	Time2	Time2	Unsorted Array	Sorted Array	Reversely Sorted
Insertion Sort	27	18	18	20	0	2
Merge Sort	2	3	3	2	0	1
Quick Sort	2	1	1	2	8	6
Modified Quick Sort	4	5	5	5	1	1
Heap Sort	8	9	9	8	2	3



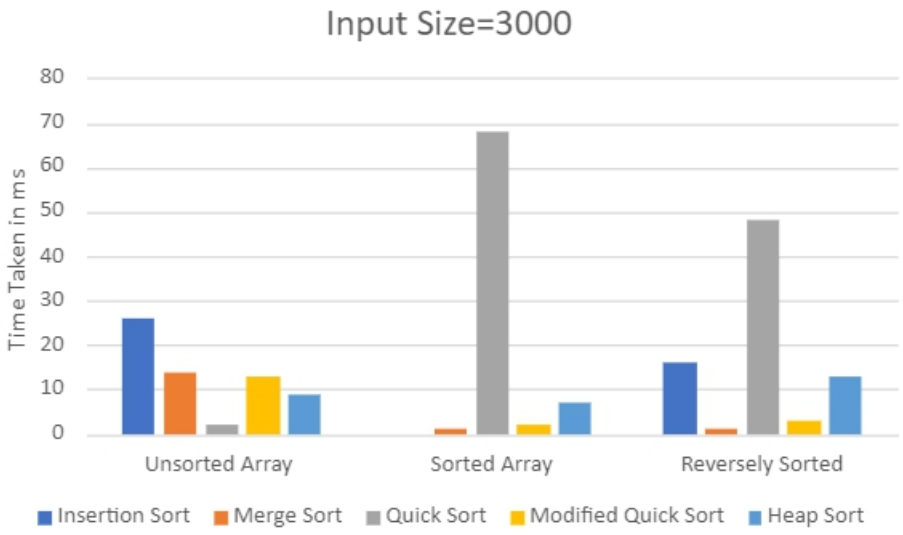
2) For input size = 2000 :

Sorting Technique	Time1	Time2	Time2	Unsorted Array	Sorted Array	Reversely Sorted
Insertion Sort	15	22	22	24	0	7
Merge Sort	2	2	2	2	0	1
Quick Sort	2	1	1	1	50	24
Modified Quick Sort	6	6	6	5	1	2
Heap Sort	5	6	6	6	4	7



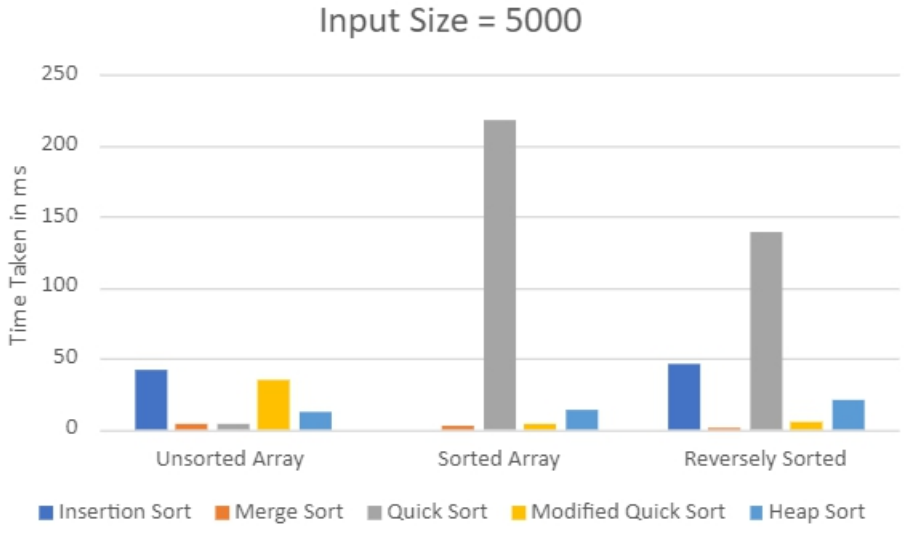
3) For input size = 3000 :

Sorting Technique	Time1	Time2	Time2	Unsorted Array	Sorted Array	Reversely Sorted
Insertion Sort	43	18	18	26	0	16
Merge Sort	14	15	15	14	1	1
Quick Sort	2	2	2	2	68	48
Modified Quick Sort	11	18	18	13	2	3
Heap Sort	9	9	9	9	7	13



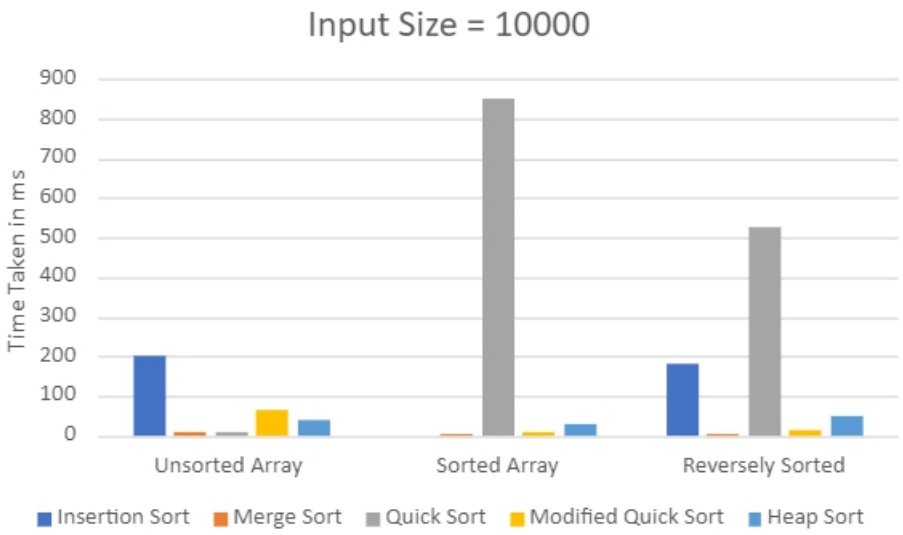
4) For input size = 5000 :

Sorting Technique	Time1	Time2	Time2	Unsorted Array	Sorted Array	Reversely Sorted
Insertion Sort	50	45	45	42	0	46
Merge Sort	4	3	3	3	2	1
Quick Sort	3	2	2	3	218	139
Modified Quick Sort	24	35	35	35	4	5
Heap Sort	16	17	17	12	13	20



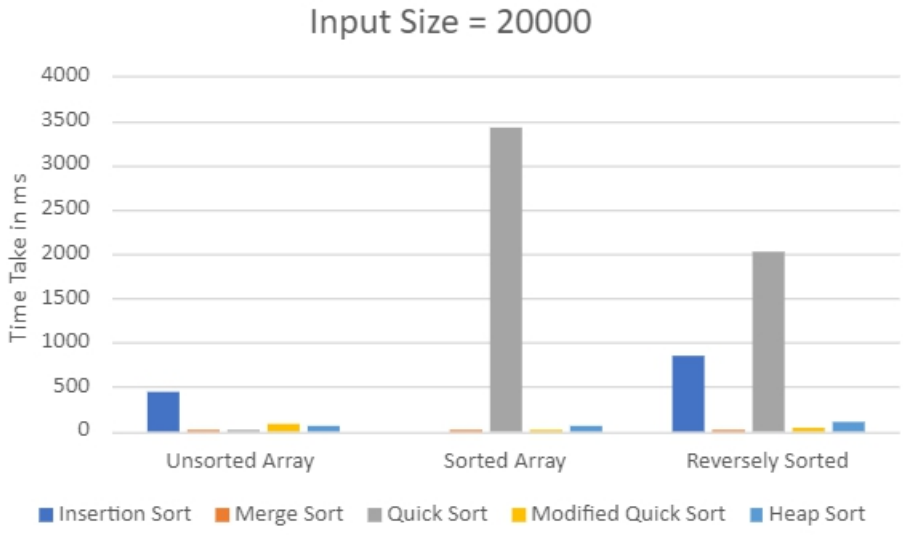
5) For input size = 10000 :

Sorting Technique	Time1	Time2	Time2	Unsorted Array	Sorted Array	Reversely Sorted
Insertion Sort	245	197	197	202	0	182
Merge Sort	11	8	8	8	4	3
Quick Sort	8	10	10	8	849	525
Modified Quick Sort	18	86	86	65	8	13
Heap Sort	34	36	36	37	30	48



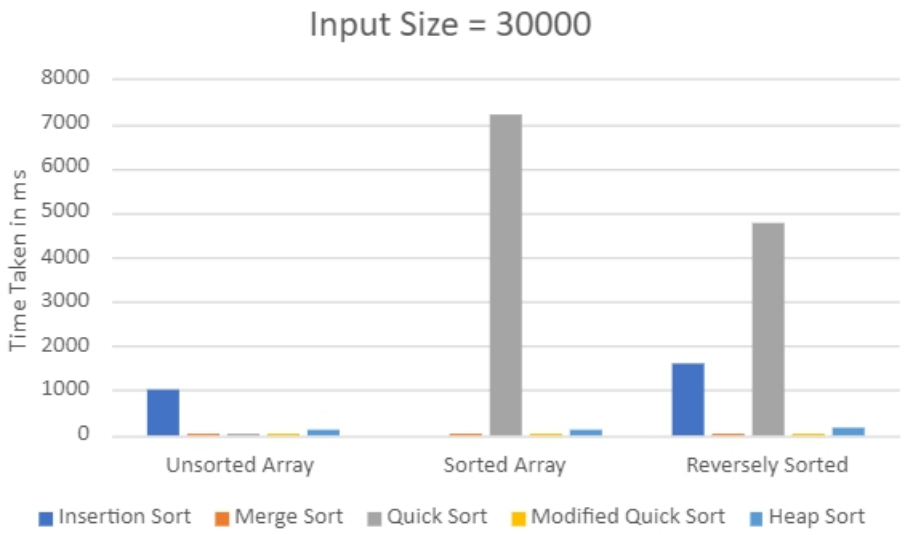
6) For input size = 20000 :

Sorting Technique	Time1	Time2	Time2	Unsorted Array	Sorted Array	Reversely Sorted
Insertion Sort	381	485	485	436	0	854
Merge Sort	19	24	24	20	8	6
Quick Sort	8	13	13	12	3420	2030
Modified Quick Sort	32	117	117	86	17	27
Heap Sort	67	66	66	69	65	101



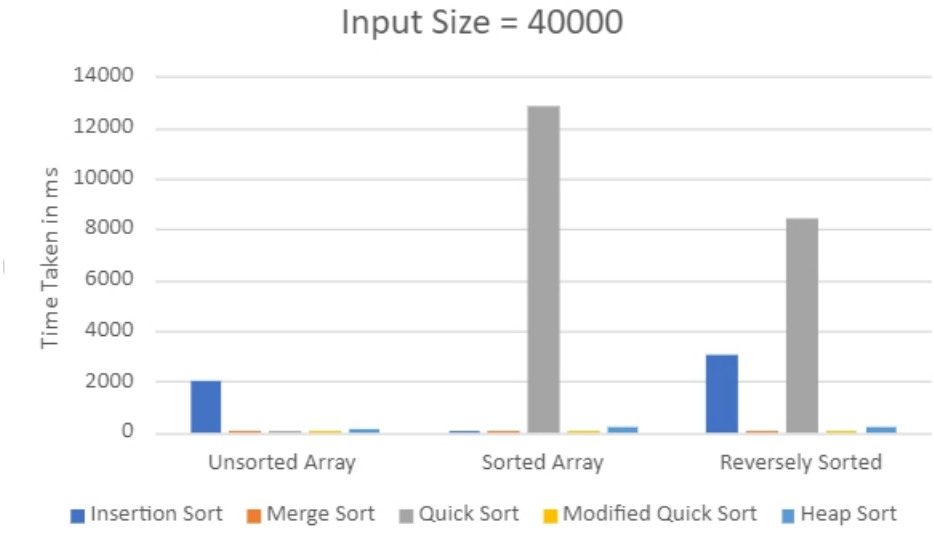
7) For input size = 30000 :

Sorting Technique	Time1	Time2	Time2	Unsorted Array	Sorted Array	Reversely Sorted
Insertion Sort	986	1175	1175	1024	0	1628
Merge Sort	19	18	18	18	14	9
Quick Sort	15	10	10	11	7208	4786
Modified Quick Sort	49	42	42	46	26	41
Heap Sort	101	106	106	109	111	161



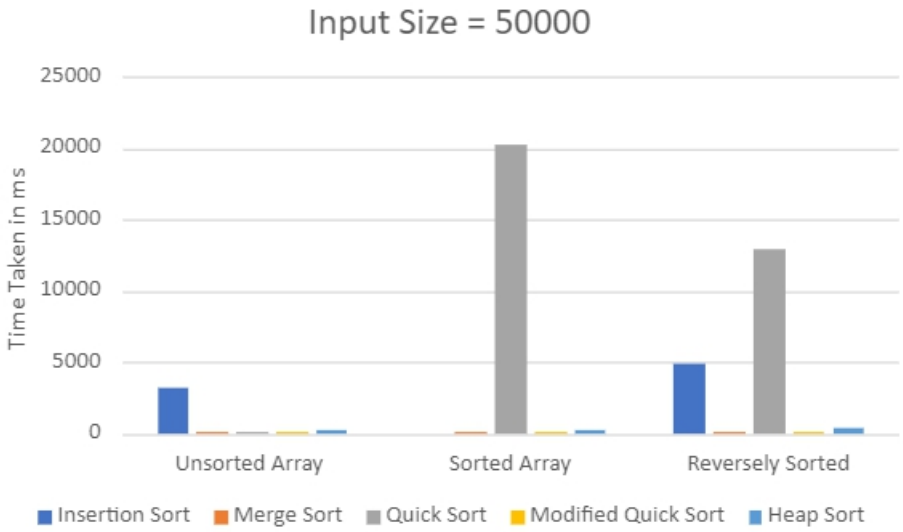
8) For input size = 40000 :

Sorting Technique	Time1	Time2	Time2	Unsorted Array	Sorted Array	Reversely Sorted
Insertion Sort	1802	2454	2454	1986	1	3038
Merge Sort	18	16	16	17	18	15
Quick Sort	22	10	10	15	12886	8395
Modified Quick Sort	68	60	60	63	39	58
Heap Sort	141	141	141	143	165	238



9) For input size = 50000 :

Sorting Technique	Time1	Time2	Time2	Unsorted Array	Sorted Array	Reversely Sorted
Insertion Sort	2831	3773	3773	3123	0	4825
Merge Sort	27	23	23	26	26	27
Quick Sort	22	15	15	17	20291	12921
Modified Quick Sort	80	82	82	79	48	69
Heap Sort	201	186	186	198	192	296



10) For input size = 60000 :

Sorting Technique	Time1	Time2	Time2	Unsorted Array	Sorted Array	Reversely Sorted
Insertion Sort	4024	5111	5111	4424	1	7199
Merge Sort	32	24	24	28	17	19
Quick Sort	21	17	17	19	28642	18801
Modified Quick Sort	93	91	91	92	60	86
Heap Sort	218	255	255	234	217	381