PROJECT –1

Comparison-based Sorting Algorithms

DEPARTMENT OF COMPUTER SCIENCE

ITCS-6114
Algorithms and Data Structures

Submitted To
Dewan T. Ahmed, Ph.D.

Submitted By:

Yash kanodia
801136232

Dipesh Chakrani
801080983

1.0 Introduction

1.1 Sort Algorithms:

A sorting algorithm is an algorithm made up of a series of instructions that takes an array as input, performs specified operations on the array, sometimes called a list, and outputs a sorted array. All sorting algorithms share the goal of outputting a sorted list, but the way that each algorithm goes about this task can vary. When working with any kind of algorithm, it is important to know how fast it runs and in how much space it operates—in other words, its time complexity and space complexity.

1.2 Insertion Sort:

Insertion Sort is a comparison-based sorting algorithm. A sub-list of sorted elements is maintained. For example, the first element is initially imagined to be the only element in the sorted sub-list. Then the element to be inserted in this sorted sub-list, must be compared with the existing elements in it and then after finding a place, it must be inserted there. The array is scanned sequentially, and the items are moved from unsorted to the sorted sub-list. Since the sorted sub-list is imagined to be the first character of the array, comparison and insertion is done using the same single array.

Time Complexity:

In worst case, each element is compared with all the other elements in the sorted array. For N elements, there will be $N2$ comparisons. Therefore, the time complexity is O($N2$)

Code:

```
public class InsertionSort {
        public int[] sort(int[] A) {
            for (int i = 1; i < A.length; i++) {
                int key = A[i];
                int j = i;
                while (j > 0 && A[j - 1] > key) {
                    A[j] = A[j - 1];
                    j = j - 1;
                }
                A[j] = key;
            }
            return A;
        }
    }
```

1.3 Merge Sort:

Merge sort is a sorting technique based on divide and conquer technique. The array sent to the merge sort is first divided into two halves, and then elements of the two halves are compared and arrays are merged. But this partition is carried on till only one element is left for partition and then the merging begins.

Time Complexity:

The list of size N is divided into a max of log (N) parts, and the merging of all sub-lists into a single list takes O(N) time, the worst case run time of this algorithm is O($NLogN$).

Code:

```
public class MergeSort {
    public static int[] merge(int[] left, int[] right)
    {

        int n = left.length + right.length;
        int[] B = new int[n];
        int i = 0, j = 0, k = 0;
        while (i < left.length || j < right.length)
        {
            if (i < left.length && j < right.length)
            {
                if (left[i] <= right[j])
                {
                    B[k] = left[i];
                    i++;
                    k++;
                }
                else
                {
                    B[k] = right[j];
                    j++;
                    k++;
                }
            }
            else if (i < left.length)
            {
                B[k] = left[i];
                i++;
                k++;
            }
            else if (j < right.length)
```

```
            {
                B[k] = right[j];
                j++;
                k++;
            }
        }
        return B;
    }

    public int[] partition(int[] A) {
        //long startTime = System.currentTimeMillis();
        int mid;
        if (A.length <= 1)
        {
            return A;
        }
        int[] right;
        mid = A.length / 2;
        int[] left = new int[mid];
        if (A.length % 2 == 0)
        {
            right = new int[mid];
        } else
        {
            right = new int[mid + 1];
        }

        for (int i = 0; i < mid; i++)
        {
            left[i] = A[i];
        }
        int x = 0;

        for (int k = mid; k < A.length; k++)
        {
            if (x < right.length)
            {
                right[x] = A[k];
                x++;
            }
        }
        left = partition(left);
        right = partition(right);
        int[] result = new int[A.length];
        result = merge(left, right);
        return result;
    }
}
```

1.4 Heapsort:

Heaps can be used in sorting an array. In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array. Consider an array Arr which is to be sorted using Heap Sort.

• Initially build a max heap of elements in Arr.

• The root element, that is $Arr[1]$, will contain maximum element of Arr. After that, swap this element with the last element of Arr and **heapify** the max heap excluding the last element which is already in its correct position and then decrease the length of heap by one.

• Repeat the step 2, until all the elements are in their correct position.

Time Complexity:

Heapify has complexity O($LogN$) , build_maxheap has complexity O(N) and we run max_heapify $N-1$ times in heap_sort function, therefore complexity of heap_sort function is O($NLogN$).

Code:

```
import java.util.Vector;

public class HeapSort {
    //code goes here
    void swapval(Vector<Integer> vector, int i) {
        int temp = (int) vector.get(i);
        vector.set(i, vector.get(i / 2));
        vector.set(i / 2, temp);
    }
      void swap(Vector<Integer> vector, int a, int b){
        int temp= vector.get(a);
        vector.set(a,vector.get(b));
        vector.set(b,temp);

    }
     void heapify(Vector<Integer> arr, int n, int i)
    {
        int smallest = i; // Initialize largest as root
        int l = 2 * i ; // left = 2*i + 1
        int r = 2 * i + 1; // right = 2*i + 2

        // If left child is larger than root
        if (l < n && arr.get(l) < arr.get(smallest))
            smallest = l;

        // If right child is larger than largest so far
        if (r < n && arr.get(r) < arr.get(smallest))
            smallest = r;

        // If largest is not root
        if (smallest != i) {
            swap(arr,i,smallest);
```

```
33
34              // Recursively heapify the affected sub-tree
35              heapify(arr, n, smallest);
36          }
37      }
38
39      void heap_sort(int[] arr){
40          Vector<Integer> vec = new Vector<>(arr.length);
41
42          int j;
43          vec.add(0,null);
44          int n=0,m=0;
45          for (int i = 0; i < arr.length; i++) {
46              n = n + 1;
47              vec.add(n, arr[i]);
48              j = n;
49              while (j > 1 && vec.get(j) < vec.get(j / 2)) {
50                  swapval(vec, j);
51                  j = j / 2;
52              }
53          }
54
55
56          for(int i=vec.size()-1; i>=1; i-- ){
57              arr[m++]=vec.get(1);
58              swap(vec,1,i);
59              heapify(vec,i,1);
60          }
61      }
62  }
63
```

1.5 In-Place Quick sort:

Quick sort is a highly efficient sorting algorithm and the partitioning method to divide array into smaller arrays. A large array is partitioned into two arrays using an element called pivot. This pivot is selected randomly every time and is such that one side of the pivot has elements lesser than it and the other side has elements greater than the pivot value. Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.

Time complexity:

This algorithm is quite efficient for large-sized input as its average and worst-case complexity are of O(n2), where n is the number of items.

Code:

```
import java.util.Random;

public class Randomized_QuickSort {
    public static void swap(int[] arr, int a, int b)
    {
        int temp = arr[a];
        arr[a] = arr[b];
        arr[b] = temp;
    }
    public int[] QuickSort(int[] A, int start, int end) {

        if (end > start) {
            Random rand = new Random();
            int pIndex = start + rand.nextInt(end - start + 1);
            swap(A, pIndex, end);
            int pivot = A[end];
            int index = partition(A, start, end, pivot);
            QuickSort(A, start, index - 1);
            QuickSort(A, index + 1, end);
        }
```

```
        return A;
    }

    public static int partition(int[] A, int start, int end, long pivot) {
        int startPtr = start - 1;
        int endPtr = end;
        while (true) {
            while (A[++startPtr] < pivot)
                ;
            while (endPtr > 0 && A[--endPtr] > pivot)
                ;
            if (startPtr >= endPtr)
                break;
            else
                swap(A, startPtr, endPtr);
        }
        swap(A, startPtr, end);
        return startPtr;
    }
}
```

1.6 Modified Quick sort:

It is like the usual quicksort but includes two modifications. Firstly, the pivot element is not randomly selected rather the median of three elements i.e. leftmost, rightmost and center is calculated to be used as the pivot. Secondly, whenever the size of the unsorted array is lesser than or equal to 10, insertion sort is called for further sorting.

Time Complexity:

Modified Quicksort has worst case complexity O($N2$) , best case complexity O($NLogN$).

Code:

```
public class QuickSortMedian {
    public void quicksort( Comparable [ ] a ) {
        quicksortmed( a, 0, a.length - 1 );
    }

    private static final int CUTOFF = 10;

    private static void quicksortmed( Comparable[] a, int first, int last ) {
        if( first + CUTOFF > last )
            insertionSort( a, first, last );
        else {
            // Sort low, middle, high
            int middle = ( first + last ) / 2;
            if( a[ middle ].compareTo( a[ first ] ) < 0 )
                swap( a, first, middle );
            if( a[ last ].compareTo( a[ first ] ) < 0 )
                swap( a, first, last );
            if( a[last ].compareTo( a[ middle ] ) < 0 )
                swap( a, middle, last );

            // Place pivot at position high - 1
            swap( a, middle, last - 1 );
            Comparable pivot = a[ last - 1 ];

            int i, j;
            for( i = first, j = last - 1; ; ) {
                while( a[ ++i ].compareTo( pivot ) < 0 )
                    ;
                while( pivot.compareTo( a[ --j ] ) < 0 )
                    ;
```

```
            if( i >= j )
                break;
            QuickSortMedian.swap( a, i, j );
        }

        swap( a, i, last - 1 );

        quicksortmed( a, first, i - 1 );
        quicksortmed( a, i + 1, last );
    }
}



private static void insertionSort( Comparable [ ] a, int low, int high ) {
    for (int p = low + 1; p <= high; p++) {
        Comparable tmp = a[p];
        int j;

        for (j = p; j > low && tmp.compareTo(a[j - 1]) < 0; j--)
            a[j] = a[j - 1];
        a[j] = tmp;
    }
}

public static final void swap( Object [ ] a, int index1, int index2 ) {
    Object tmp = a[ index1 ];
    a[ index1 ] = a[ index2 ];
    a[ index2 ] = tmp;
}}
```

2. Program Structure

The programming language used for this project was Java. We've divided the program into five classes.
Sorting: Class- Contains the main method
1. Generates input array using an in-built function
2. Passes the input array and calls functions to perform sorting on it
Functions:
Global Variables: None

Insertion Sort: Class- sorts the passed array using insertion sort 1. sort: Takes the array as a parameter and sorts it using insertion sort
Functions:
1. sort: Takes the array as a parameter and sorts it using insertion sort
Global Variables: None

Merge Sort: Class- sorts the passed array using merge sort
Functions:
1. partition: divides the array into two equal parts
2. merge: combines the two partitioned arrays into one array after comparing the elements
Global Variables: None

Randomized quick sort: Class- This class sorts the passed array using Randomized in-place quick sort, that is, by using any random element as the pivot
Functions:
1. swap: swaps the values at indexes passed to it
2. QuickSort: sorts all the elements between the start and end index passed to it
3. partition: divides the array using a random pivot each time, such that all elements on one side are lesser than pivot and on one side are greater than pivot
Global Variables: None

Quick Sort Median: Class- This class sorts the passed array using quick sort by using median-of-three as the pivot element
Functions:
1. swap: swaps two elements
2. InsertionSort: when the number of elements is less than 10, insertion sort is used for sorting the array
3. Quicksorted: usual quick sort is performed but the pivot element is always the median and calculates the median of three numbers.
Global Variables: None

ReverseSort : Class- This class reversely sorts the passed array and then can be used by each sorting algorithm to analyze.
Functions:
1. rsort: reversely sorts the array.
Global Variables: None

2.1 Input

In-built Random function is used for generating random input numbers.

2.2 Functionality

The program is initiated by the main class ‘Sorting’, which generates an array of random integers using an in-built Random function. This array is then passed to different functions where the respective sorting techniques are performed. The main class passes the array as a parameter to the InsertionSort class, MergeSort class, Randomized_quick_sort class and QuickSortMedian class. These functions return the sorted array. An in-built function is used for recording the start and stop time of processing of each algorithm to record the execution time. Once the sorting is done by all the algorithms, the execution time is printed on the screen.

Results:

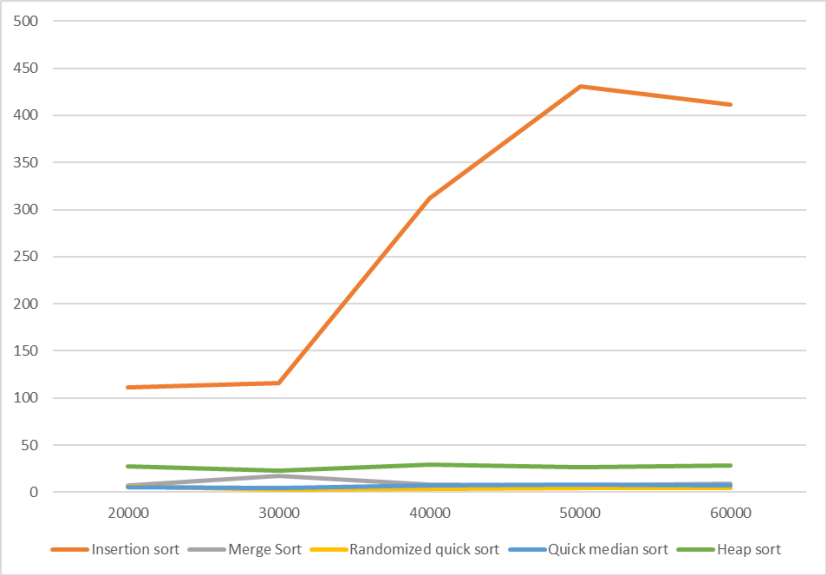
1. Unsorted Array

```
Choose one:
 1. Unsorted Array
 2. Sorted Array
 3. Reversely Sorted Array
1

Runnung for Array size 20000 : Iteration 1

Sorting using insertion sort: 146.0milliseconds

Sorting using merge sort 12.0milliseconds

Sorting using randomized quick sort 17.0milliseconds

Sorting using Quick median sort 12.0milliseconds

Sorting using Heap sort 62.0milliseconds

Runnung for Array size 20000 : Iteration 2

Sorting using insertion sort: 145.0milliseconds

Sorting using merge sort 5.0milliseconds

Sorting using randomized quick sort 1.0milliseconds

Sorting using Quick median sort 3.0milliseconds

Sorting using Heap sort 11.0milliseconds

Runnung for Array size 20000 : Iteration 3

Sorting using insertion sort: 44.0milliseconds

Sorting using merge sort 4.0milliseconds

Sorting using randomized quick sort 1.0milliseconds

Sorting using Quick median sort 2.0milliseconds
```

```
Sorting using merge sort 11.0milliseconds

Sorting using randomized quick sort 6.0milliseconds

Sorting using Quick median sort 11.0milliseconds

Sorting using Heap sort 42.0milliseconds


Runnung for Array size 60000 : Iteration 2

Sorting using insertion sort: 435.0milliseconds

Sorting using merge sort 11.0milliseconds

Sorting using randomized quick sort 3.0milliseconds

Sorting using Quick median sort 6.0milliseconds

Sorting using Heap sort 22.0milliseconds


Runnung for Array size 60000 : Iteration 3

Sorting using insertion sort: 331.0milliseconds

Sorting using merge sort 5.0milliseconds

Sorting using randomized quick sort 4.0milliseconds

Sorting using Quick median sort 6.0milliseconds

Sorting using Heap sort 21.0milliseconds
```

```
Sorting time Insert sort for input arrays: [111.66666666666667, 115.66666666666667, 312.3333333333333, 430.3333333333333, 411.6666666666667]
Sorting time Merge sort for input arrays: [7.0, 17.333333333333332, 8.333333333333334, 7.666666666666667, 9.0]
Sorting time Ramdomised quick sort for input arrays: [6.333333333333333, 2.6666666666666665, 4.0, 4.333333333333333, 4.333333333333333]
Sorting time Quick Median sort for input arrays: [5.666666666666667, 4.333333333333333, 7.0, 8.0, 7.666666666666667]
Sorting time Heap sort for input arrays: [27.666666666666668, 23.0, 29.666666666666668, 26.333333333333332, 28.333333333333332]
```

Array size	Insertion sort	Merge Sort	Randomized quick sort	Quick median sort	Heap sort
20000	111.66	7	6.33	5.66	27.66
30000	115.66	17.33	2.66	4.33	23
40000	312.33	8.33	4	7	29.66
50000	430.33	7.66	4.33	8	26.33
60000	411.66	9	4.33	7.66	28.33

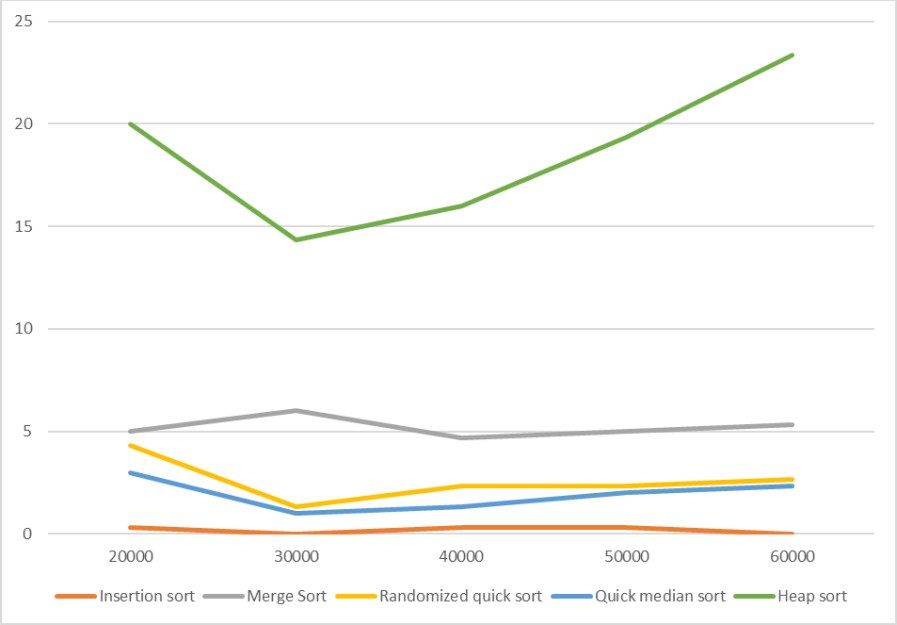


2. Sorted Array

```
Choose one:
 1. Unsorted Array
 2. Sorted Array
 3. Reversely Sorted Array
2

Runnung for Array size 20000 : Iteration 1

Sorting using insertion sort: 0.0milliseconds

Sorting using merge sort 7.0milliseconds

Sorting using randomized quick sort 10.0milliseconds

Sorting using Quick median sort 8.0milliseconds

Sorting using Heap sort 42.0milliseconds


Runnung for Array size 20000 : Iteration 2

Sorting using insertion sort: 0.0milliseconds

Sorting using merge sort 4.0milliseconds

Sorting using randomized quick sort 1.0milliseconds

Sorting using Quick median sort 1.0milliseconds

Sorting using Heap sort 10.0milliseconds
```

```
Runnung for Array size 60000 : Iteration 2

Sorting using insertion sort: 0.0milliseconds

Sorting using merge sort 5.0milliseconds

Sorting using randomized quick sort 3.0milliseconds

Sorting using Quick median sort 2.0milliseconds

Sorting using Heap sort 19.0milliseconds


Runnung for Array size 60000 : Iteration 3

Sorting using insertion sort: 0.0milliseconds

Sorting using merge sort 5.0milliseconds

Sorting using randomized quick sort 2.0milliseconds

Sorting using Quick median sort 2.0milliseconds

Sorting using Heap sort 24.0milliseconds

-----------------------------------------------------------------------------------------------------------------------------------------------------------

Sorting time Insert sort for input arrays: [0.3333333333333333, 0.0, 0.3333333333333333, 0.3333333333333333, 0.0]
Sorting time Merge sort for input arrays: [5.0, 6.0, 4.666666666666667, 5.0, 5.333333333333333]
Sorting time Ramdomised quick sort for input arrays: [4.333333333333333, 1.3333333333333333, 2.3333333333333335, 2.3333333333333335, 2.6666666666666665]
Sorting time Quick Median sort for input arrays: [3.0, 1.0, 1.3333333333333333, 2.0, 2.3333333333333335]
Sorting time Heap sort for input arrays: [20.0, 14.333333333333334, 16.0, 19.333333333333332, 23.333333333333332]
```

Array size	Insertion sort	Merge Sort	Randomized quick sort	Quick median sort	Heap sort
20000	0.33	5	4.33	3	20
30000	0	6	1.33	1	14.33
40000	0.33	4.66	2.33	1.33	16
50000	0.33	5	2.33	2	19.33
60000	0	5.33	2.66	2.33	23.33

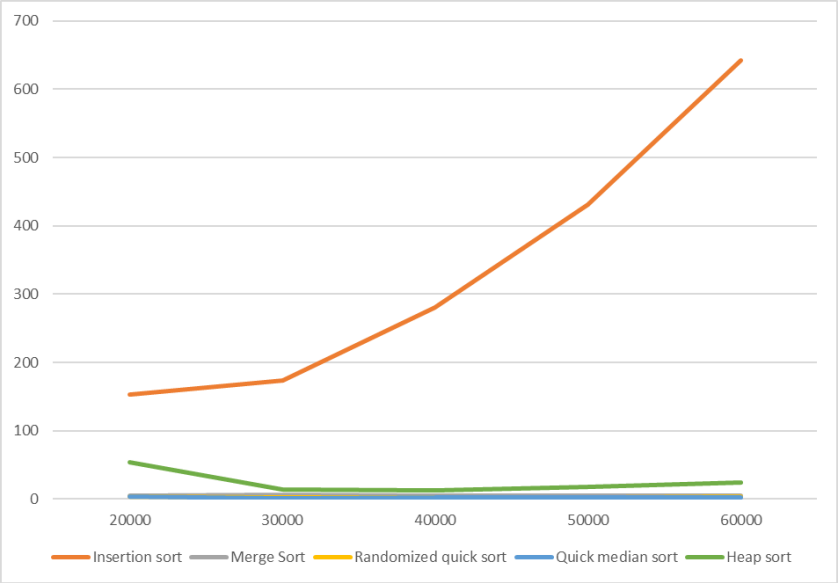


3. Reversely Sorted Array

```
Choose one:
 1. Unsorted Array
 2. Sorted Array
 3. Reversely Sorted Array
3

Runnung for Array size 20000 : Iteration 1

Sorting using insertion sort: 193.0milliseconds

Sorting using merge sort 6.0milliseconds

Sorting using randomized quick sort 8.0milliseconds

Sorting using Quick median sort 7.0milliseconds

Sorting using Heap sort 71.0milliseconds


Runnung for Array size 20000 : Iteration 2

Sorting using insertion sort: 196.0milliseconds

Sorting using merge sort 4.0milliseconds

Sorting using randomized quick sort 1.0milliseconds

Sorting using Quick median sort 1.0milliseconds

Sorting using Heap sort 42.0milliseconds
```

```
Runnung for Array size 60000 : Iteration 2

Sorting using insertion sort: 643.0milliseconds

Sorting using merge sort 6.0milliseconds

Sorting using randomized quick sort 3.0milliseconds

Sorting using Quick median sort 2.0milliseconds

Sorting using Heap sort 20.0milliseconds


Runnung for Array size 60000 : Iteration 3

Sorting using insertion sort: 620.0milliseconds

Sorting using merge sort 5.0milliseconds

Sorting using randomized quick sort 3.0milliseconds

Sorting using Quick median sort 2.0milliseconds

Sorting using Heap sort 28.0milliseconds

----------------------------------------------------------------------------------------------------------------------------------

Sorting time Insert sort for input arrays: [153.0, 173.0, 280.0, 431.3333333333333, 642.3333333333334]
Sorting time Merge sort for input arrays: [4.666666666666667, 6.0, 4.333333333333333, 4.666666666666667, 5.333333333333333]
Sorting time Ramdomised quick sort for input arrays: [3.3333333333333335, 2.0, 2.0, 2.6666666666666665, 3.0]
Sorting time Quick Median sort for input arrays: [3.0, 1.3333333333333333, 1.6666666666666667, 2.0, 2.3333333333333335]
Sorting time Heap sort for input arrays: [53.666666666666664, 13.333333333333334, 13.0, 17.666666666666668, 24.333333333333332]
```

Array size	Insertion sort	Merge Sort	Randomized quick sort	Quick median sort	Heap sort
20000	153	4.66	3.33	3	53.66
30000	173	6	2	1.33	13.33
40000	280	4.33	2	1.66	13
50000	431.33	4.66	2.66	2	17.66
60000	642.33	5.33	3	2.33	24.33



References:

1) www.stackoverflow.com
2) www.geeksforgeeks.com