2222-CSE-5311-004 DESIGN ANALYSIS AND ALGORITHMS

PROJECT

NAME: NAVYASHREE BUDHIHAL MUTT

UTA ID:1001965572

Sorting Techniques: Sorting techniques is used to rearrange a given array of element according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure. This project will display the important properties of different sorting techniques including their time complexity.

Sorting Techniques implemented in this project:

- Merge sort
- Heap sort
- Quick sort
- Insertion sort
- Selection sort
- Bubble sort

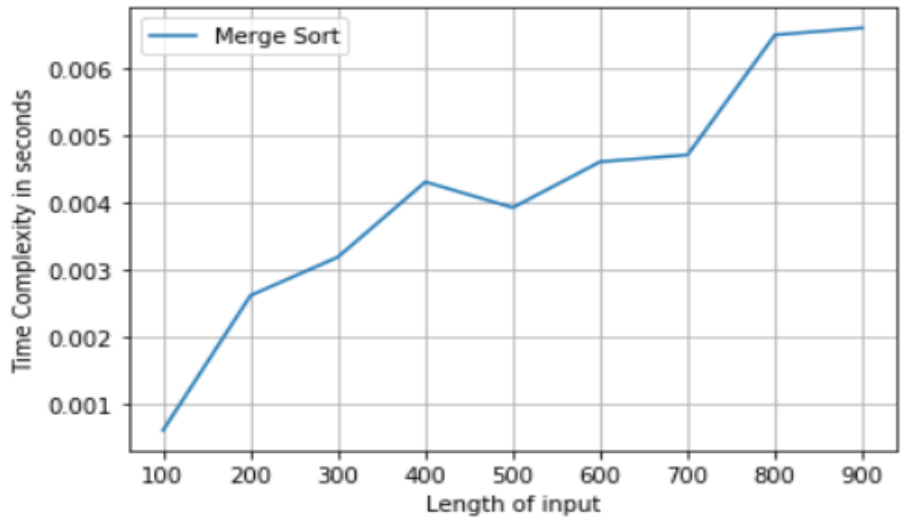
MERGE SORT: -

Merge Sort is a divide and conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge_sort () function is used for merging two halves. The merge_sort () is key process that assumes that array is sorted and merges the two sorted sub-arrays into one. The time complexity is O (n log n) in best case, worst case and average case.

Graph for run time for different input sizes for merge sort is as shown below:

```
100  elements Merge Sort time:  0.0006222999999963008
200  elements Merge Sort time:  0.0026269999999968263
300  elements Merge Sort time:  0.003198000000004697
400  elements Merge Sort time:  0.004315800000000536
500  elements Merge Sort time:  0.003933500000002255
600  elements Merge Sort time:  0.004615799999996284
700  elements Merge Sort time:  0.00471809999999806
800  elements Merge Sort time:  0.006503600000002052
900  elements Merge Sort time:  0.006606600000004903
```

```
C:\Users\Administrator\Anaconda3\lib\site-packages\ipykernel_launcher.py:5: Dep
ed in Python 3.3 and will be removed from Python 3.8: use time.perf_counter or
  """
C:\Users\Administrator\Anaconda3\lib\site-packages\ipykernel_launcher.py:7: Dep
ed in Python 3.3 and will be removed from Python 3.8: use time.perf_counter or
  import sys
```



QUICK SORT:

Quick Sort is also a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

The steps for median of three is:

1. Pick an element, called a pivot, from the list.

2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way).

3. After this partitioning, the pivot is in its final position. This is called the partition operation.

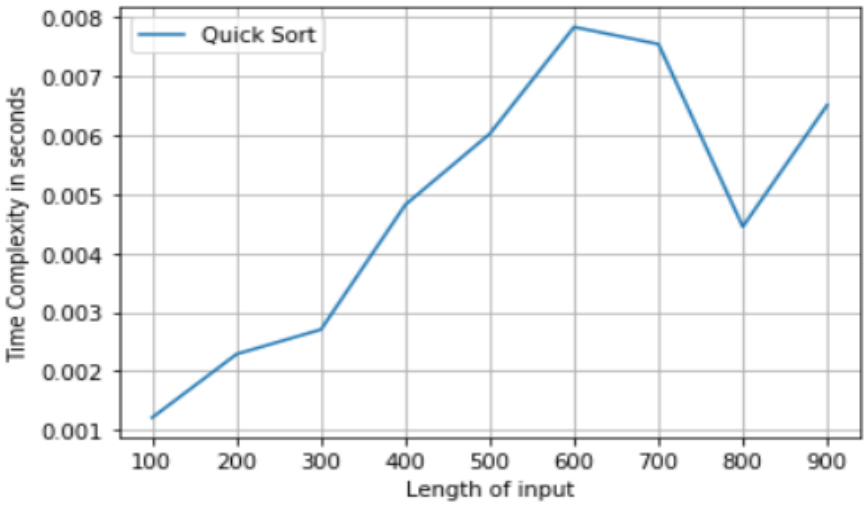
4. Recursively sort the sub-list of lesser elements and the sub-list of greater elements. Quicksort with median-of-three partitioning functions nearly the same as normal quicksort with the only difference being how the pivot item is selected.

In normal quicksort the first element is automatically the pivot item. This causes normal quicksort to function very inefficiently when presented with an already sorted list. The division will always end up producing one sub-array with no elements and one with all the elements. In quicksort with median-of-three partitioning the pivot item is selected as the median between the first element, the last element, and the middle element.

O(n) is best case and o(n^2) is worst case.

Graph for run time for different input sizes for quick sort is as shown below:

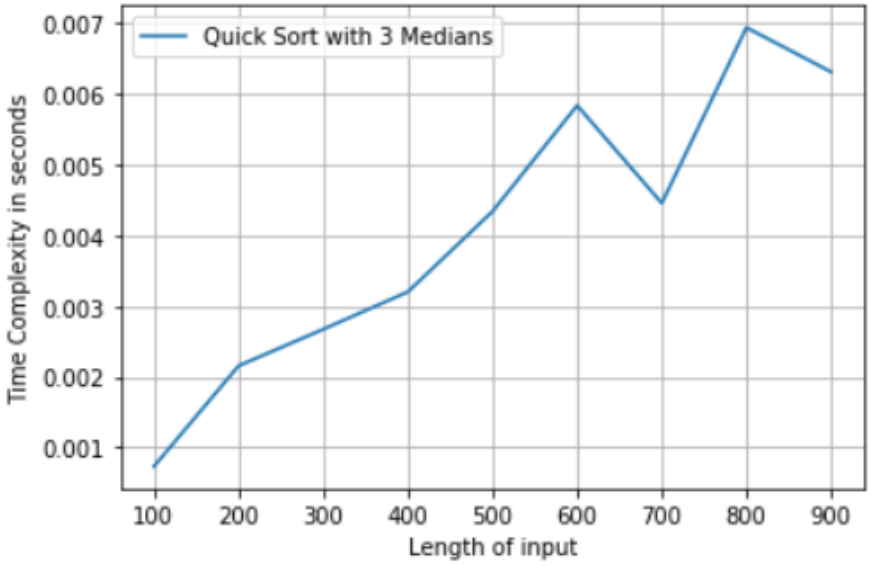
```
100 Elements Quick Sort time:  0.0012130999999993008
200 Elements Quick Sort time:  0.0022897999999997865
300 Elements Quick Sort time:  0.0027062999999998283
400 Elements Quick Sort time:  0.004823500000000536
500 Elements Quick Sort time:  0.006025700000000356
600 Elements Quick Sort time:  0.007837699999999614
700 Elements Quick Sort time:  0.007547900000000496
800 Elements Quick Sort time:  0.004449900000000895
900 Elements Quick Sort time:  0.006513700000001066
```



Graph for run time for different input sizes for quick sort with 3 medians is as shown below:

```
100 Elements Quick Sort with 3 medians time:  0.0007332999998652667
200 Elements Quick Sort with 3 medians time:  0.0021568000001934706
300 Elements Quick Sort with 3 medians time:  0.002673500000128115
400 Elements Quick Sort with 3 medians time:  0.003203499999926862
500 Elements Quick Sort with 3 medians time:  0.004341700000168203
600 Elements Quick Sort with 3 medians time:  0.0058404999999766005
700 Elements Quick Sort with 3 medians time:  0.004457600000023376
800 Elements Quick Sort with 3 medians time:  0.006942900000012742
900 Elements Quick Sort with 3 medians time:  0.006315699999959179
```

```
C:\Users\Administrator\Anaconda3\lib\site-packages\ipykernel_launcher
ed in Python 3.3 and will be removed from Python 3.8: use time.perf_c
  """
C:\Users\Administrator\Anaconda3\lib\site-packages\ipykernel_launcher
ed in Python 3.3 and will be removed from Python 3.8: use time.perf_c
  import sys
```



HEAP SORT:

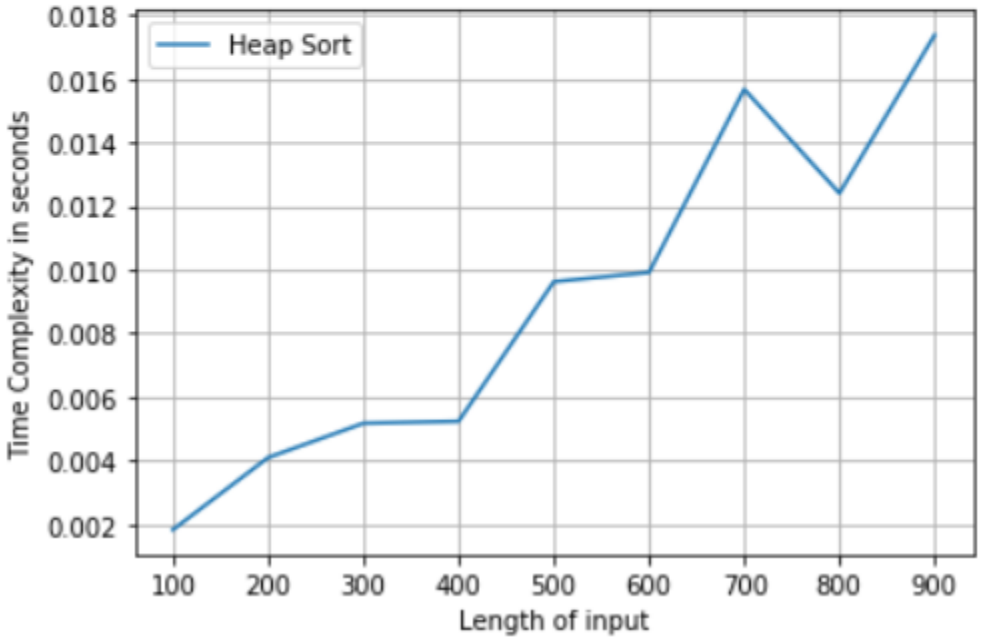
Heap sort is a comparison-based sorting algorithm.

It divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region.Heap sort is slower than quick sort. - Heap sort build a max heap from the input data. The largest item is stored at the root of the heap. It replace root with the last item of the heap followed by reducing the size of heap by 1.

O(n log n) is best case and worst case.

Graph for run time for different input sizes for heap sort is as shown below:

```
100 Elements Heap Sort time:  0.00181940000038594
200 Elements Heap Sort time:  0.004087499999513966
300 Elements Heap Sort time:  0.0051640999990922865
400 Elements Heap Sort time:  0.005226600000241888
500 Elements Heap Sort time:  0.009618500000215136
600 Elements Heap Sort time:  0.009909100001095794
700 Elements Heap Sort time:  0.015669500000512926
800 Elements Heap Sort time:  0.012400900001011905
900 Elements Heap Sort time:  0.017374900000504567
```



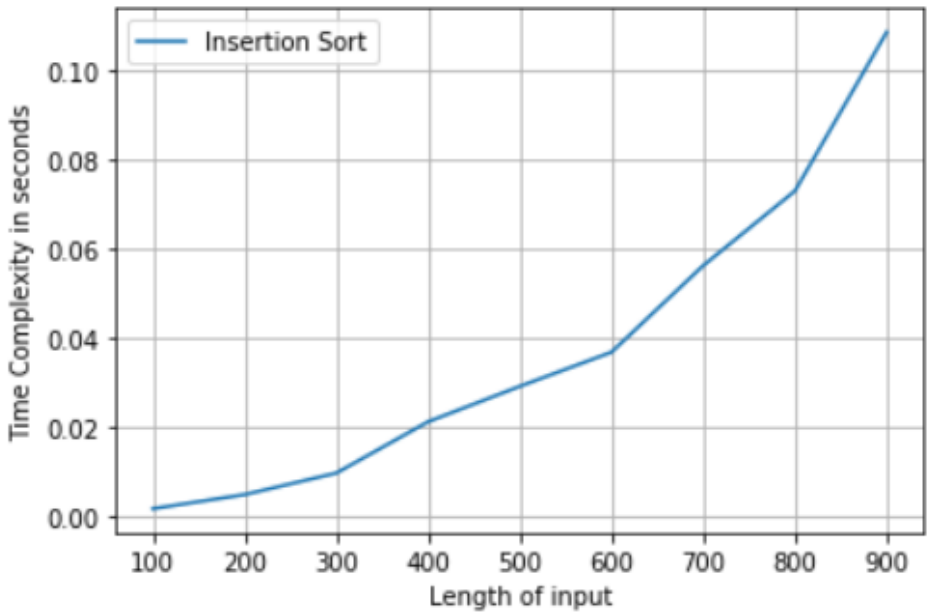
INSERTION SORT:

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. - Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. - At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain. - It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

O(n) is best case and O(n^2) is worst case.

Graph for run time for different input sizes for insertion sort is as shown below:

```
100 Elements Insertion Sort time:  0.0018613000011100667
200 Elements Insertion Sort time:  0.0050123000000894535
300 Elements Insertion Sort time:  0.009838199999649078
400 Elements Insertion Sort time:  0.021335600000384147
500 Elements Insertion Sort time:  0.029271299999891198
600 Elements Insertion Sort time:  0.03692649999902642
700 Elements Insertion Sort time:  0.05632559999867226
800 Elements Insertion Sort time:  0.07305780000024242
900 Elements Insertion Sort time:  0.10863670000071579
```



SELECTION SORT:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

The subarray which is already sorted.

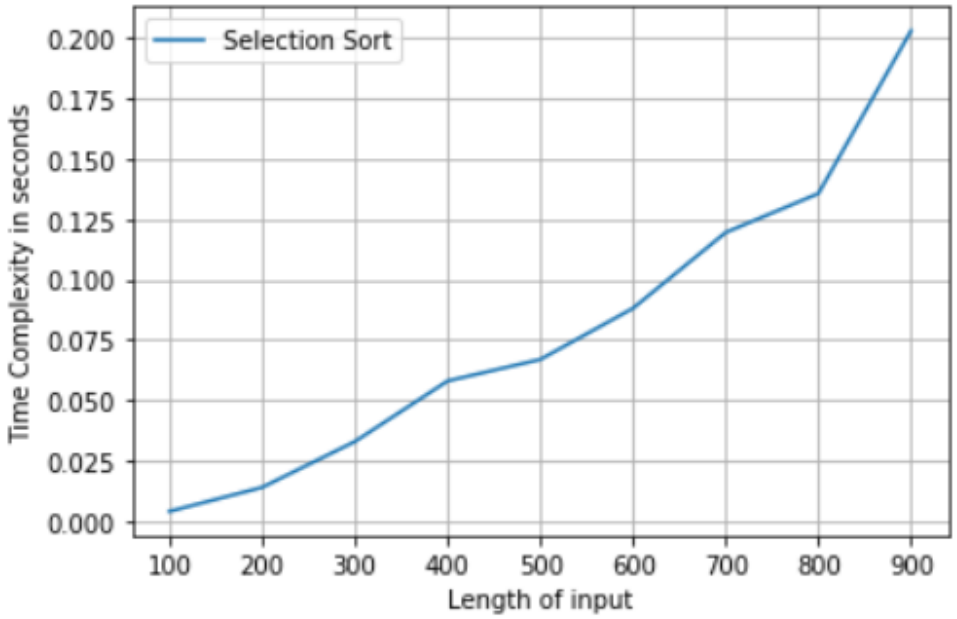
Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

O(n^2) is the worst case, best case and average case.

Graph for run time for different input sizes for selection sort is as shown below:

```
100 Elements Selection Sort time:  0.004191199999695527
200 Elements Selection Sort time:  0.014105300000665011
300 Elements Selection Sort time:  0.03303480000067793
400 Elements Selection Sort time:  0.05807160000040312
500 Elements Selection Sort time:  0.06695209999998042
600 Elements Selection Sort time:  0.08810069999890402
700 Elements Selection Sort time:  0.11943009999959031
800 Elements Selection Sort time:  0.13551709999956074
900 Elements Selection Sort time:  0.2027952999997069
```



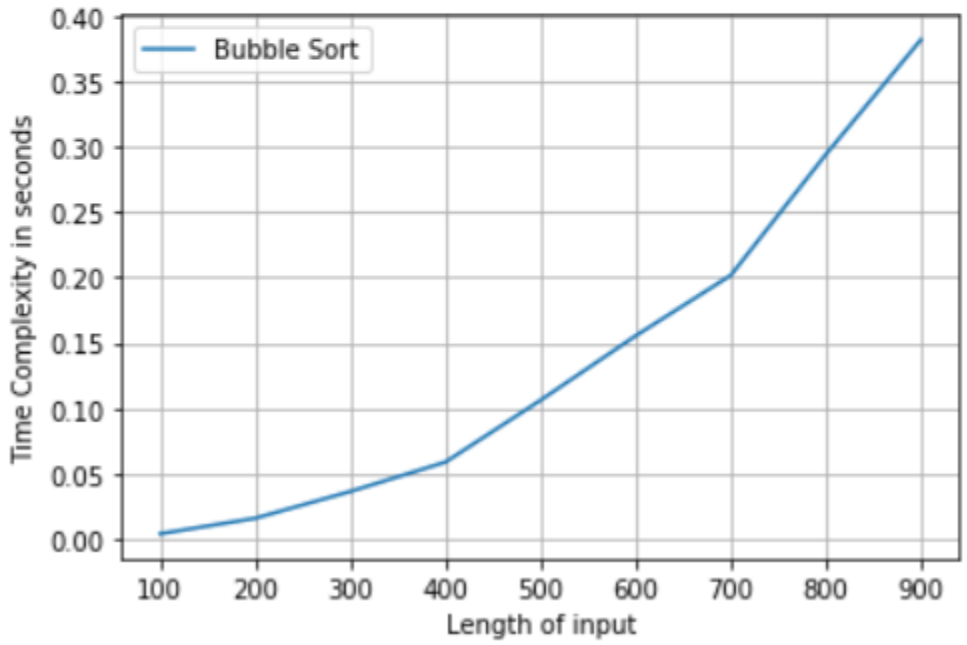
BUBBLE SORT:

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. Best case is when elements in the array is sorted.

O(n) is best case and o(n^2) is worst case.

Graph for run time for different input sizes for bubble sort is as shown below:

```
100 Elements Bubble Sort time:  0.00345019999986107
200 Elements Bubble Sort time:  0.015333800000007614
300 Elements Bubble Sort time:  0.03590049999911571
400 Elements Bubble Sort time:  0.05844240000078571
500 Elements Bubble Sort time:  0.10592999999971653
600 Elements Bubble Sort time:  0.15538250000099652
700 Elements Bubble Sort time:  0.20179859999916516
800 Elements Bubble Sort time:  0.29462539999985893
900 Elements Bubble Sort time:  0.38273579999986396
```



So as per above observations of the runt time of all sorting techniques:

Quick sort is better for the large data inputs.