**Group Members: Chaitanya Darade, Tanvi Rasam**

**Project 1: Comparison-based Sorting Algorithms**

**Project Overview** :

In this project, the following algorithms are implemented in Python language:

* Insertion sort
* Merge sort
* Heapsort [vector based, and insert one item at a time]
* In-place quicksort (any random item or the first or the last item of your input can be pivot).
* Modified quicksort
  + Use median-of-three as pivot.
  + For small sub-problem of size ≤ 10, use insertion sort.

**Data Structures Used:**

1. Lists
2. Numpy Arrays
3. Numpy Matrix
4. Strings

**Complexity Analysis:**

1. **Insertion Sort**

**Insertion sort** is a sorting algorithm that builds a final sorted array (sometimes called a list) one element at a time. While sorting is a simple concept, it is a basic principle used in complex computer programs such as file search, data compression, and path finding. Running time is an important thing to consider when selecting a sorting algorithm since efficiency is often thought of in terms of speed.

Insertion sort has an average and worst-case running time of O(n^2), so in most cases, a faster algorithm is more desirable.

**Time Complexity:** O(n^2)

**Auxiliary Space:**O(1)

**Boundary Cases**: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

**Algorithmic Paradigm:** Incremental Approach

**Sorting In Place:** Yes

**Stable:** Yes

**Online:** Yes

**Uses:** Insertion sort is used when number of elements is small. It can also be useful when input is almost sorted, only few elements are misplaced in complete big array.

1. **Merge Sort**

**Merge sort** is an efficient sorting algorithm that uses a **divide-and-conquer** approach to order elements in an array. Sorting is a key tool for many problems in computer science. For example, inputting a list of names to a sorting algorithm can return them in alphabetical order, or a sorting algorithm can order a list of basketball players by how many points they each scored. Running time is an important thing to consider when selecting a sorting algorithm since efficiency is often thought of in terms of speed. Mergesort runs in a guaranteed **O(nlogn)** time, which is significantly faster than the average- and worst-case running times of several other sorting algorithms

**Time complexity:**

It takes O(1) time to divide the problem into two parts. To divide the problem, the algorithm computes the middle of the list by taking the length of the list and dividing by two, which takes constant time.

Two equally large subproblems are produced. Each half takes T(n/2) time, so solving the subproblems takes a total of 2T(n/2) time.  
This is the merge step of mergesort. This step takes O(n) time, as shown in the analysis of the merge algorithm.  
So solution to this recurrence is **O(nlogn)**Mergesort runs in **O(nlogn)** time in its best case, worst case, and average case. That means that no matter what the input, mergesort will operate in ***O*(*n*log*n*)** time.

**Auxiliary Space:** O(n)

**Algorithmic Paradigm:**Divide and Conquer

**Sorting In Place:** No in a typical implementation

**Stable:** Yes

1. **Heap Sort**

**Heapsort** is a comparison-based sorting algorithm that uses a binary heap data structure. Like mergesort, heapsort has a running time of *O*(*n*log*n*), and like insertion sort, heapsort sorts **in-place**, so no extra space is needed during the sort.

The binary heap data structure allows the heapsort algorithm to take advantage of the heap's **heap properties** and the heapsort algorithm makes use of the efficient running time for inserting to and deleting from the heap.

**Time complexity:**

It takes **O(logn)** for **heapify** and **O(n)** for **constructing a heap**. Hence, the overall time complexity of **heap sort**using**min heap** or**max heap** is **O(nlogn)**

**Auxiliary Space:  O(1)**

**Sorting In Place:** Yes

**Stable:** No

1. **And 5. Quick Sort In Place and Modified**

**Quicksort** is a **fast** sorting algorithm that takes a **divide and conquer** approach to sorting lists. While sorting is a simple concept, it is a basic principle used in complex programs such as file search, data compression, and pathfinding. Running time is an important thing to consider when selecting a sorting algorithm since efficiency is often thought of in terms of speed. Quicksort has a very slow worst-case running time, but a fast average and best-case running time.

The answer depends on strategy for choosing pivot. In early versions of Quick Sort where leftmost (or rightmost) element is chosen as pivot, the **worst** occurs in following cases.

**1)** **Array is already sorted in same order.  
2) Array is already sorted in reverse order.  
3) All elements are same (special case of case 1 and 2)**

Since these cases are very common use cases, the problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or **(especially for longer partitions) choosing the median of the first, middle and last element** of the partition for the pivot. With these modifications, the worst case of Quick sort has less chances to occur, but worst case can still occur if the input array is such that the maximum (or minimum) element is always chosen as pivot.

Picking a good pivot is the key for a fast implementation of quicksort; however, it is difficult to determine what a good pivot might be. The partitioning step takes time proportional to the number of elements being partitioned, so reducing the number of elements in each partition would give a faster runtime. The best-case pivot would divide the array into two equal parts, which would halve the problem size. However, this means that the pivot is the median of the elements, and in order to find the median, we would need an already sorted array. Since the goal of quicksort is to sort an array, we can’t rely on having a pivot equal to the median of the elements.

**Time complexity:**

Quicksort will have a best-case running time when the pivot at each recursive call is equal to the median element of the subarray. This means that, at each step, the problem size is being halved, and the array can be sorted with**log*n*** nested calls. Each call takes **O(n)** time (from the division step), so the total run time of the best-case quicksort is ***O*(*n*log*n*).**

**Auxiliary Space:  O(n)**

**Boundary Cases :**

* 1. **Worst case complexity: O(n^2):**

In the worst case, all elements are either less than or greater than the pivot. In other words, if the pivot is the smallest or largest element of the array.

* 1. **Best case complexity:** **O(nlogn)**

The best case would be when both arrays are of the same length, in which case it would take **2T((n-1)/2)** to solve both of the subproblems.

**Sorting In Place:** Yes

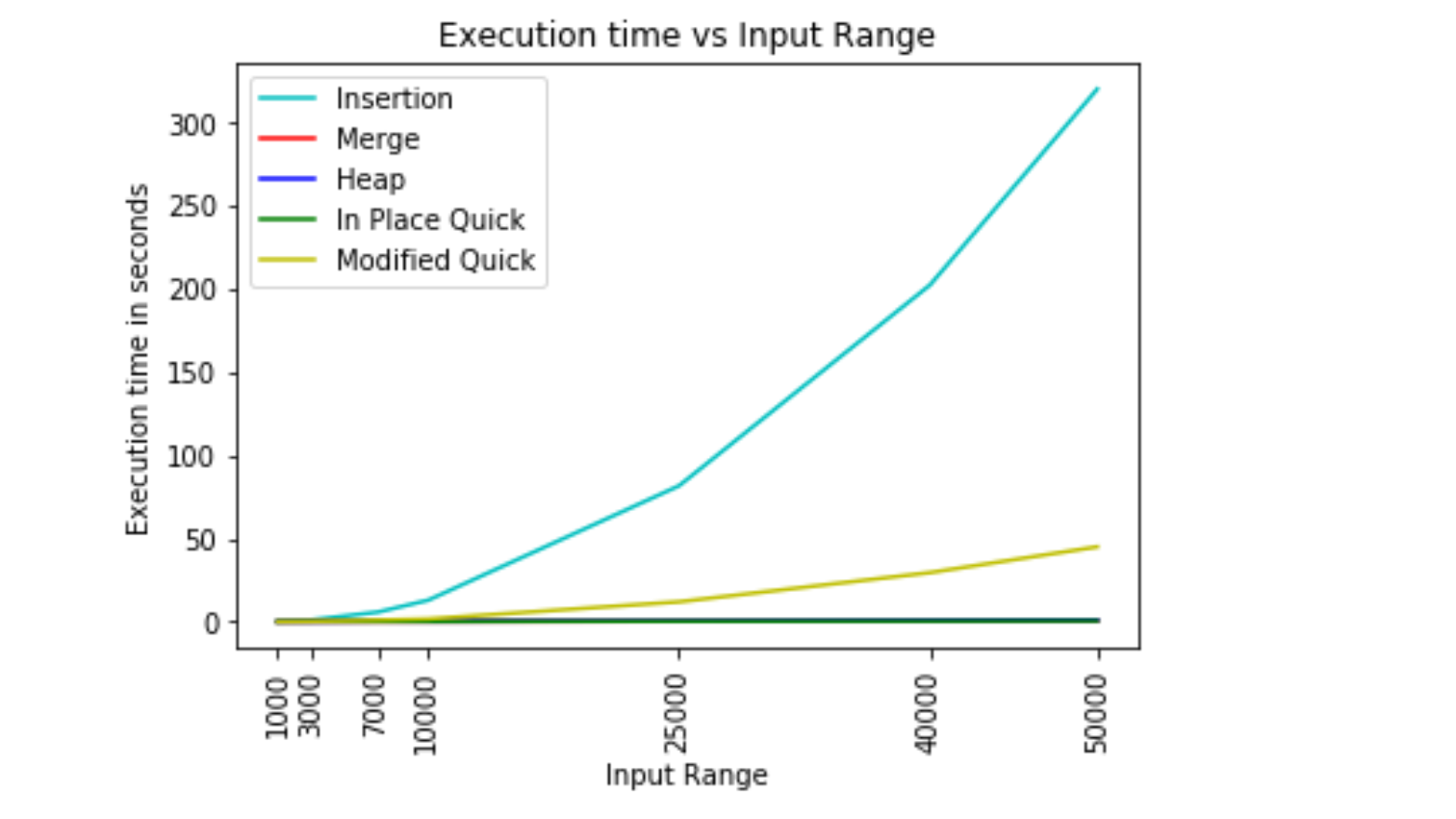
**Stable:** Yes

**Results Visualization:**

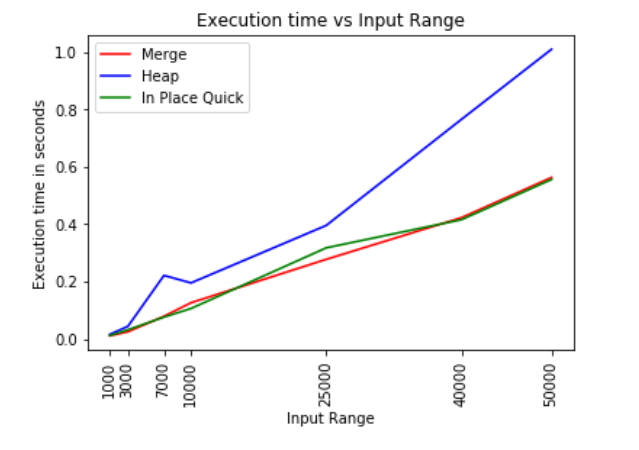
**1) Unsorted array**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Range** | 1000 | 3000 | 7000 | 10000 | 20000 | 40000 | 50000 |
| **Algorithms** |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| Insertion | 0.151 | 1.05 | 5.788 | 1.291 | 8.171 | 202.38 | 320.15 |
| Merge | 0.012 | 0.026 | 0.08 | 0.127 | 0.278 | 0.423 | 0.563 |
| Heap | 0.017 | 0.045 | 0.222 | 0.196 | 0.396 | 0.765 | 1.009 |
| In Place Quick | 0.013 | 0.033 | 0.077 | 0.107 | 0.318 | 0.416 | 0.556 |
| Modified Quick | 0.035 | 0.217 | 0.953 | 1.636 | 11.95 | 29.49 | 45 |

**T1:Algorithms and Average execution time (in seconds)**

****

**Fig 1.1: Execution Time Vs Input Range for Unsorted Array**

****

**Fig 1.2: Execution Time Vs Input Range of O(nlogn) Algorithms**

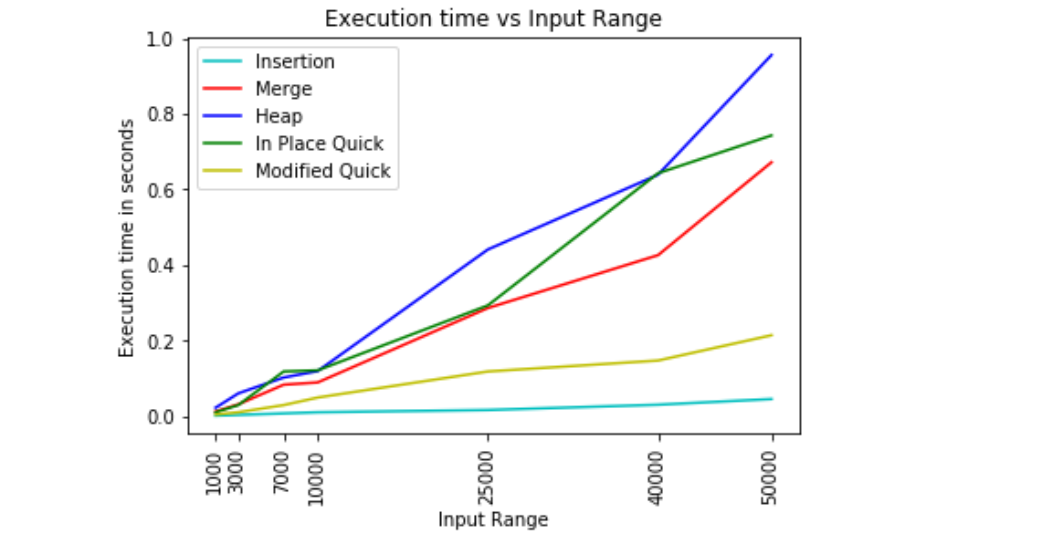
**Observation:**

* As we know, Insertion sort takes more execution time as number of elements increases i.e O(n^2). Fig 1.1 shows, execution time for insertion sort is exponentially increasing as increase in input range. Other sorting techniques have performed comparatively efficient.
* Sorting techniques such as merge, heap and quick sort have same time complexity i.e O(nlogn). Thus, their execution time for different ranges is almost the same. Fig 1.2 shows the same.

**2) Sorted Array**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Range** | 1000 | 3000 | 7000 | 10000 | 20000 | 40000 | 50000 |
| **Algorithms** |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| Insertion | 0.001 | 0.003 | 0.007 | 0.01 | 0.016 | 0.03 | 0.045 |
| Merge | 0.012 | 0.031 | 0.083 | 0.089 | 0.286 | 0.426 | 0.672 |
| Heap | 0.022 | 0.06 | 0.102 | 0.119 | 0.441 | 0.639 | 0.956 |
| In Place Quick | 0.01 | 0.029 | 0.118 | 0.121 | 0.293 | 0.643 | 0.743 |
| Modified Quick | 0.004 | 0.01 | 0.029 | 0.049 | 0.118 | 0.147 | 0.214 |

**T2 : Algorithms and Average execution time (in seconds)**

****

**Fig 2 : Execution Time Vs Input Range for Sorted Array**

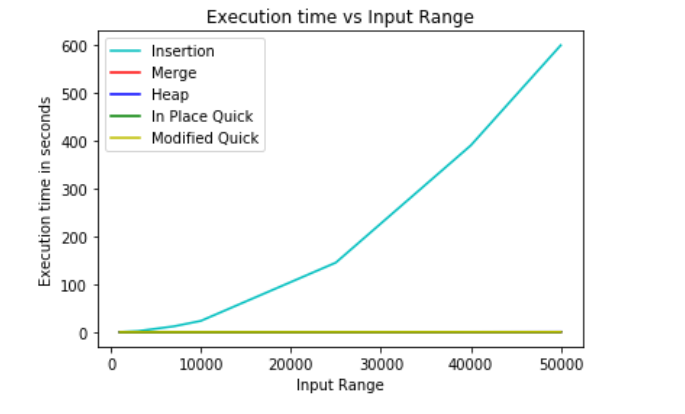
**Observation:**

* As we know, Insertion sort takes minimum execution time if input array is sorted, O(n). Also, in quick sort, **choosing the median of the first, middle and last element is best choice amongst others when data input data is sorted**. Fig 2. shows insertion and modified quick sort performed better comparatively.
* For normal quick sort if the first or last element was chosen as the pivot, the time taken will be more than unsorted array as here a skewed tree is generated thus giving a complexity of O(n^2)
* But we are randomly choosing the pivot in our case thus the time remains the same
* Merge Sort and Heap Sort is unaffected by the input array type

**3) Reverse Sorted Array**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Range** | 1000 | 3000 | 7000 | 10000 | 20000 | 40000 | 50000 |
| **Algorithms** |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| Insertion | 0.25 | 2.18 | 12.13 | 23.55 | 145.26 | 390.15 | 599.55 |
| Merge | 0.009 | 0.021 | 0.084 | 0.096 | 0.289 | 0.483 | 0.499 |
| Heap | 0.009 | 0.034 | 0.086 | 0.086 | 0.38 | 0.514 | 0.7 |
| In Place Quick | 0.006 | 0.026 | 0.083 | 0.087 | 0.307 | 0.454 | 0.48 |
| Modified Quick | 0.006 | 0.032 | 0.059 | 0.061 | 0.142 | 0.24 | 0.388 |

**T3: Algorithms and Average execution time (in seconds)**



**Fig 3.1 : Execution Time Vs Input Range for Reversely sorted**



**Fig 3.2: Execution Time Vs Input Range for Reversely sorted**

**Observation:**

* Insertion sort is used when number of elements is small. It can also be useful when input is almost sorted, however, it has worst behavior on input data which is reversely sorted as shown in fig 3.1.
* Sorting techniques such as merge, heap and quick sort have same time complexity i.e O(nlogn). Fig 3.2 shows the same.