DESIGN & ANALYSIS OF ALGORITHMS

PROJECT TITLE: SORTING ALGORITHMS

AUTHOR: SINDOORA RAVIKUMAR MURTHY

UTA ID: 1001862126

COURSE: CSE 5311-004

TABLE OF CONTENTS

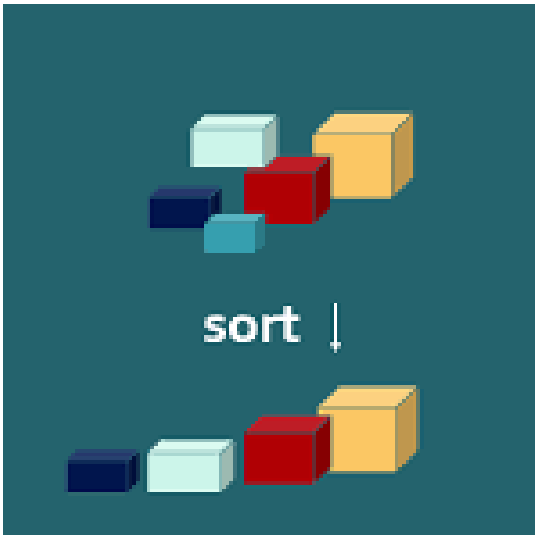
ABSTRACT

The main objective of this project is to understand different sorting algorithms, implement them using a high-level language to see their working and also draw conclusions about their time complexities.

In this project, 7 types of sorting algorithms have been implemented, i.e., ***Bubble sort, Insertion sort, Selection sort, Merge sort, Quick sort, Quick sort using three medians and Heap sort.*** For each of these algorithms we can see the Data structures used, various functions and their functionalities, the time complexity and conduct experiments for a range of inputs and plot a graph of Input size vs Run time of the algorithm.

Finally, all the 7 algorithms are run with the same input to see which is the best/most-efficient/least time-taking algorithm, considering the wide range of inputs.

SORTING ALGORITHMS



A Sorting algorithm is the logic used to arrange a collection of items in an ordered manner. Sorting Algorithms are used to rearrange a given array or list elements 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.

For example,

ARRAY	67	8	444	9	96438	4	553

Applying a sorting algorithm

ARRAY	4	8	9	67	444	553	96438

1. BUBBLE SORT

Bubble Sort is the simplest in-place algorithm sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Data Structures used: Array

Functions used:

- bubble_sort( ) : iterates through the entire array (n-1) times and compares each element with the next element and swaps if necessary. ['n' is the size of array]

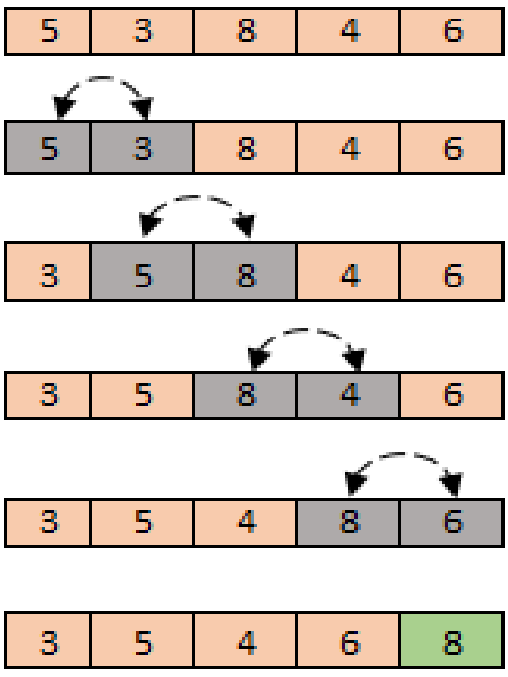
Input: array[ ] of size 'n'

Output: sorted array[ ]

Algorithm:

```
bubble_sort(array):
    for all elements in array:
        if array[i] > array[i+1]:
                swap(array[i] & array[i+1])
    return array
```

Tracing visualization:



ITERATION 1

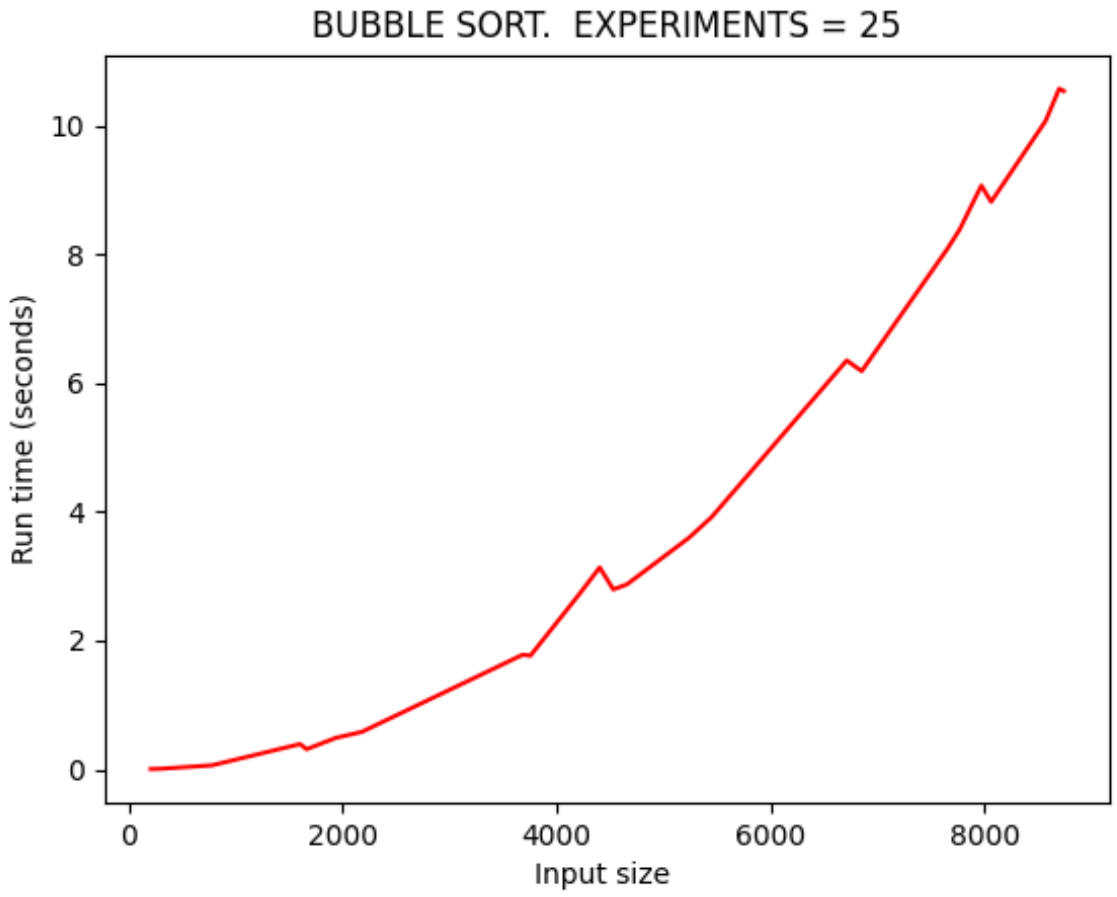
Time Complexity:

Best case	Avg case	Worst case
O(n)	$O(n^2)$	$O(n^2)$

Experimental analysis:

The Bubble sort function was executed 25 times with different number of inputs ranging from 10-10,000 in each experiment.

The following graph shows the Input size vs Run time of Bubble sort algorithm. It is clear from the graph that Bubble sort has a time complexity of $\Theta(n^2)$.



2. INSERTION SORT

Insertion sort is an in-place 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. One element from the unsorted part is consumed in every iteration to find its correct place in the sorted part of the array. Remaining elements are moved accordingly.

Data Structures used: Array

Functions used:

- insertion_sort( ) : iterates through the entire array and in each iteration the first element is initialized as the 'key'. The elements are compared with the key, key is placed in its correct position and key gets re-initialized.

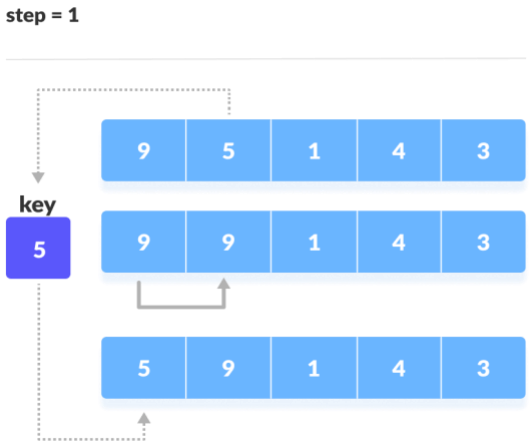
Input: array[ ] of size 'n'

Output: sorted array[ ]

Algorithm:

```
insertion_sort(array):
    for i ranging from 1 to n:
        key ← array[i]
        j ← i -1
        while j>=0 and key < array[j]:
                array[j+1] ← array[j]
                j - -
        array[j+1] ← key
    return array
```

Tracing visualization:



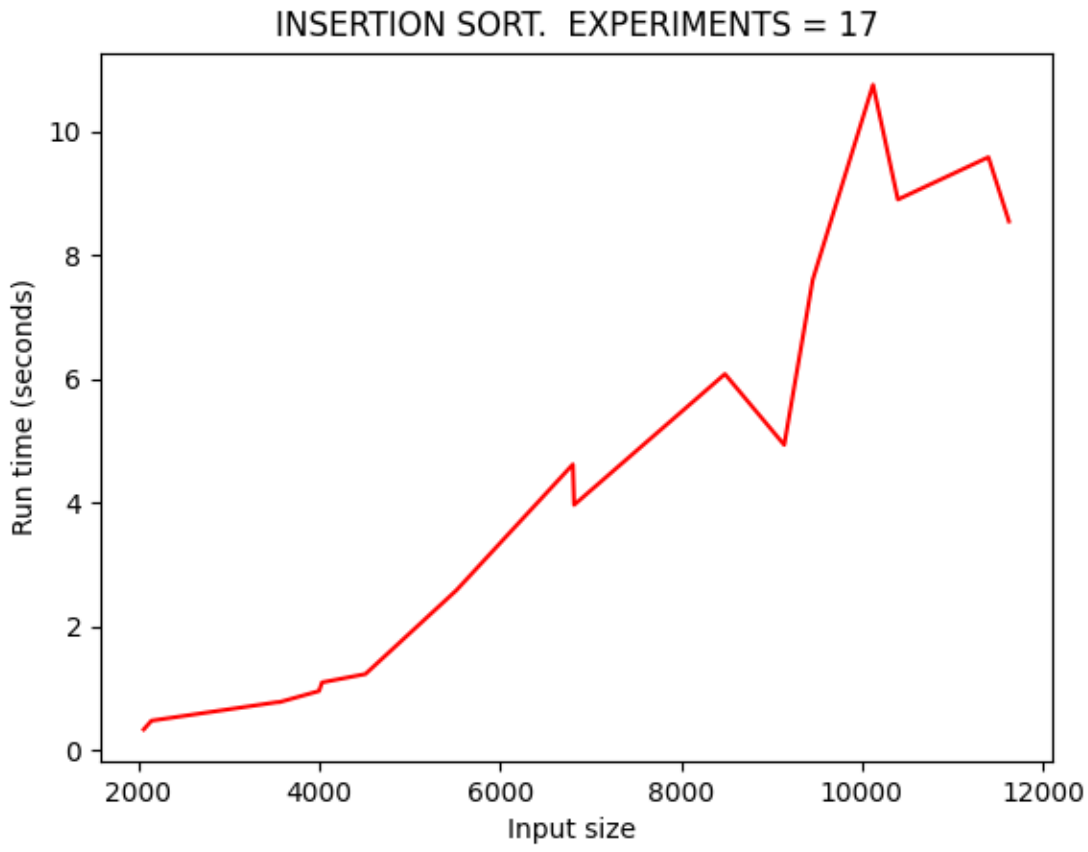
Time Complexity:

Best case	Avg case	Worst case
O(n)	$O(n^2)$	$O(n^2)$

Experimental analysis:

The Insertion sort function was executed 17 times with different number of inputs ranging from 10-12,000 in each experiment.

The following graph shows the Input size vs Run time of Insertion sort algorithm. It is clear from the graph that Insertion sort has a time complexity of $\Theta(n^2)$.



3. SELECTION SORT

The selection sort algorithm is an in-place algorithm that sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning. The algorithm maintains two subarrays i.e., the subarray which is already sorted and the remaining subarray which is unsorted.

Data Structures used: Array

Functions used:

- selection_sort( ) : iterates through the entire array and in every iteration, the minimum element from the unsorted subarray is picked and moved to the sorted subarray.

Input: array[ ] of size 'n'

Output: sorted array[ ]

Algorithm:

```
selection_sort(array):
    for i=0 to n-2:
        min ← i
    for j=i+1 to n-1:
        if array[j] < array[min]:
                min ← j
                swap ( array[j] & array[min])
    return array
```

Tracing visualization:

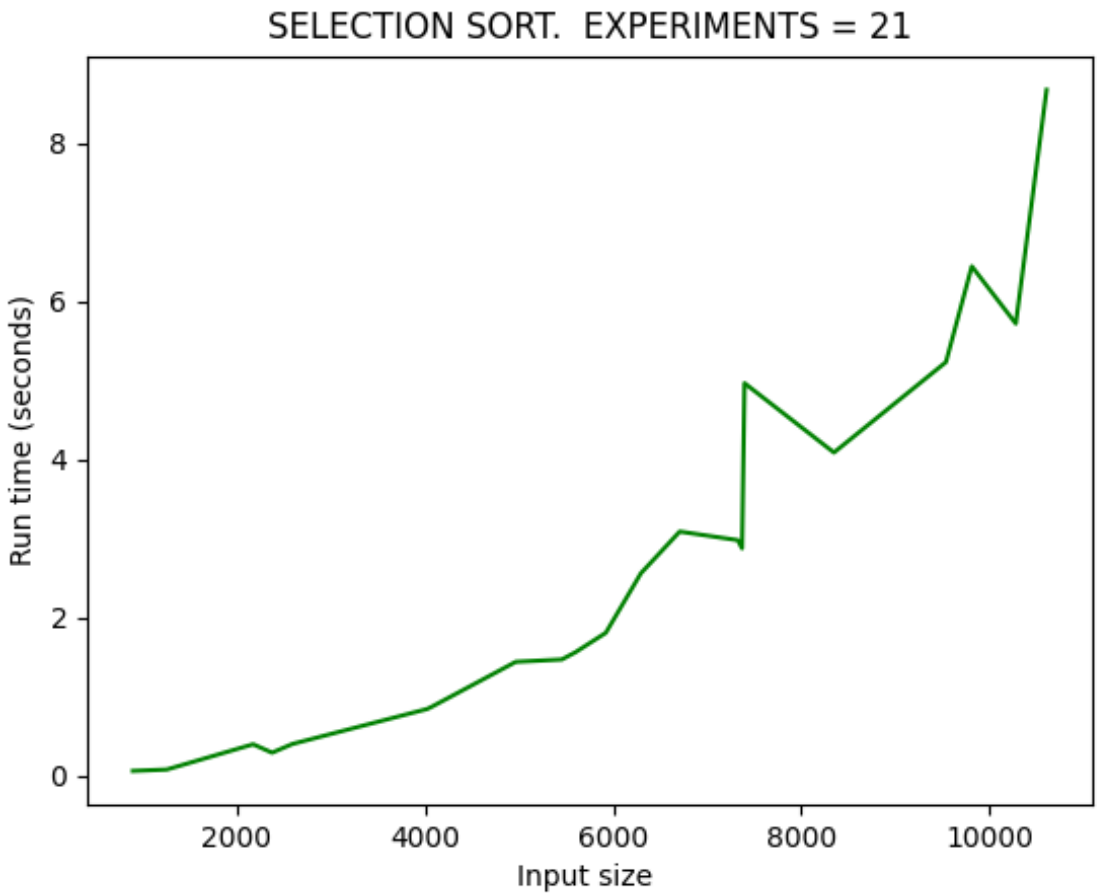
Time Complexity:

Best case	Avg case	Worst case
$O(n^2)$	$O(n^2)$	$O(n^2)$

Experimental analysis:

The Selection sort function was executed 21 times with different number of inputs ranging from 500-13,000 in each experiment.

The following graph shows the Input size vs Run time of Selection sort algorithm. It is clear from the graph that Selection sort has a time complexity of $\Theta(n^2)$.



4. MERGE SORT

Merge Sort is a Divide and Conquer algorithm. It is an out-of-place algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

Data Structures used: Array

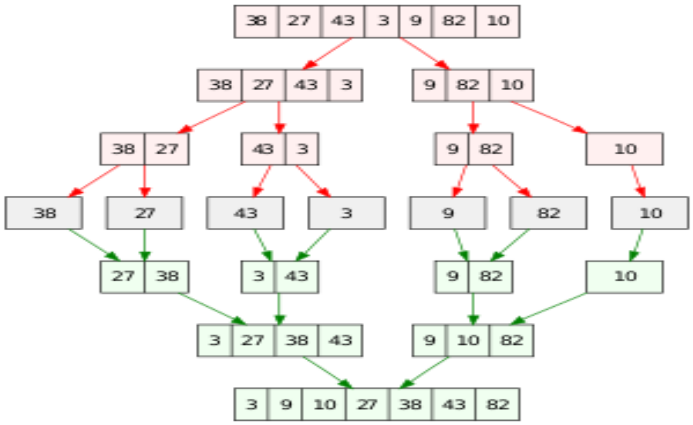
Functions used:

- merge_sort( ) : divides the array into sub-arrays recursively until each array size is 1 and then calls the merge function to compare and merge the arrays into 1.
- merge( ) : is used for merging two halves. This is a key process that merges the two sorted sub-arrays array[l..mid] and array[mid+1..r] into one.

```
Input: array[ ] of size 'n'
Output: sorted merged_array[ ]
Algorithm:
merge_sort(array):
    if n == 1:
        return array
    L1=array[low….mid]
    L2=array[mid+1…..high]
    L1=merge_sort(L1)
    L2=merge_sort(L2)
    return merge(L1,L2)

merge(array a, array b):
    array merged_array
    while( a and b have elements):
        if (a[0] > b[0]):
            add b[0] to end of merged_array
            remove b[0] from b
        else:
            add a[0] to end of merged_array
            remove a[0] from a
    while a has elements:
        add a[0] to end of merged_array
        remove a[0] from a
    while b has elements:
        add b[0] to end of merged_array
        remove b[0] from b
    return merged_array
```

Tracing visualization:



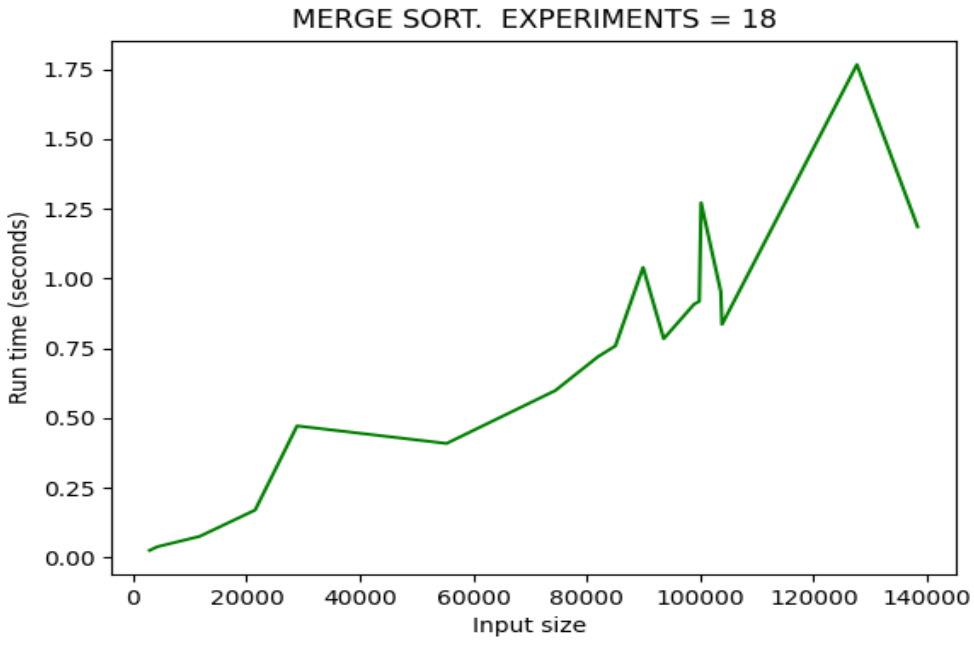
Time Complexity:

Best case	Avg case	Worst case
O(nlog(n))	O(nlog(n))	O(nlog(n))

Experimental analysis:

The Merge sort function was executed 18 times with different number of inputs ranging from 10-14,000 in each experiment.

The following graph shows the Input size vs Run time of Merge sort algorithm. It is clear from the graph that Selection sort has a time complexity of $\Theta(nlog(n))$.



5. QUICK SORT

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

Data Structures used: Array

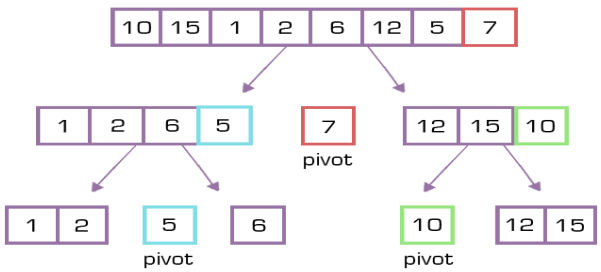
Functions used:

- quick_sort( ) : partitions the array by calling the partition function and recursively calls itself on each of the sub-arrays until each array size is 1.
- partition( ) : 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.

```
Input: array[ ] of size 'n'
Output: sorted array[ ]
Algorithm:
quick_sort(array, low, high):
    if low< high:
        index=partition(array,low,high)
        quick_sort(array,low,index-1)
        quick_sort(array,index+1,high)

partition(array, low, high):
    pivot ← array[high]
    i ← low-1
    for j=0 to high :
        if array[j] < pivot:
            i ← i +1
            swap(array[i] & array[j])
    swap( array[i+1] and array[high])
    return (i+1)
```

Tracing visualization:



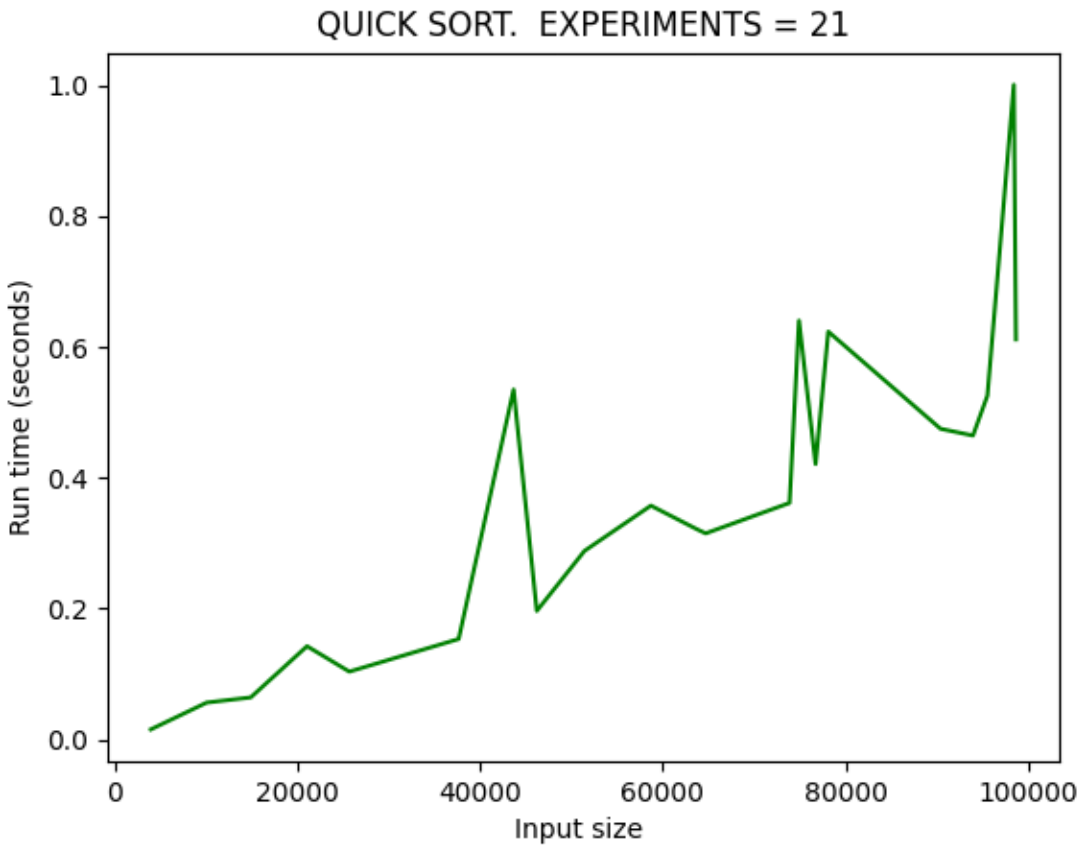
Time Complexity:

Best case	Avg case	Worst case
O(nlog(n))	O(nlog(n))	O(nlog(n))

Experimental analysis:

The Quick sort function was executed 21 times with different number of inputs ranging from 10-100,000 in each experiment.

The following graph shows the Input size vs Run time of Quick sort algorithm. It is clear from the graph that Quick sort has a time complexity of $\Theta(nlog(n))$.



6. QUICK SORT USING 3 MEDIANS

Quick Sort is a Divide and Conquer algorithm. It is an in-place algorithm. It picks 3 medians and chooses the middle elements of these 3 and applies quick sort to get the sorted array.

Data Structures used: Array

Functions used:

- median_quickSort( ) : divides the array based on 3 medians by calling the medianPivot( ) function and sorts the array by calling the quick_sort( ) function on the re-arranged array.
- medianPivot( ): picks the first, middle and the last elements of the array as 3 medians, sorts these 3 medians and picks the middle element of the 3 as the pivot and calls the partition function that divides the array based on the middle element's value.
- quick_sort( ) : partitions the array by calling the partition function and recursively calls itself on each of the sub-arrays until each array size is 1.
- partition( ) : 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.

Input: array[ ] of size 'n'

Output: sorted array[ ]

Algorithm:

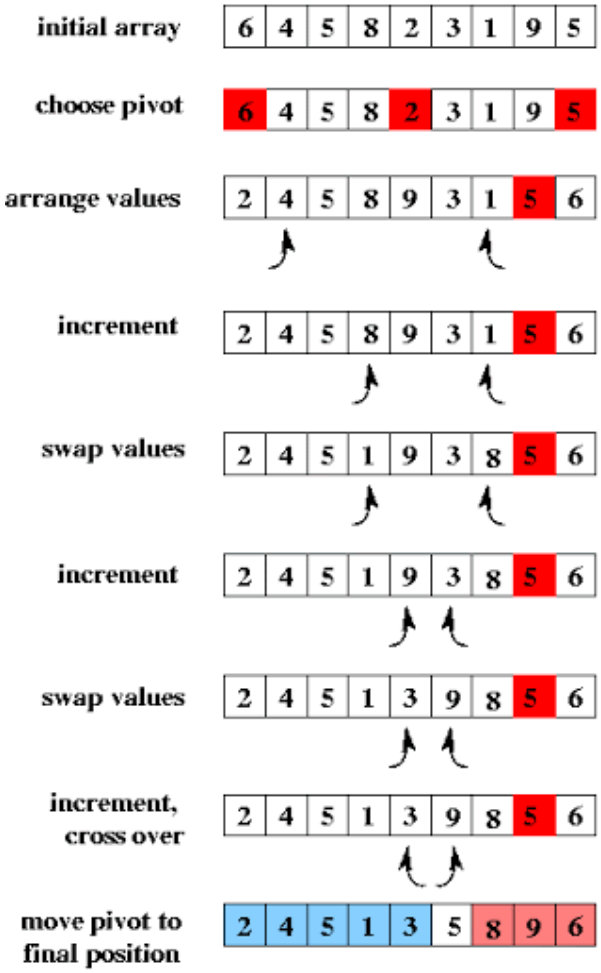
```
median_quickSort( array, low, high):
    if low>= high: return
    if low<high :
        pi ← medianPivot(array, start, end)
        quick_sort(array, start, end)

medianPivot(array, low, high):
    mediansList ← sorted( first, mid, last elements of the array)
    middle ← mediansList[1]
    swap ( array[first], array[mid] and array[last] elements based on middle's value)
    return partition(array, low, high)

quick_sort(array, low, high):
    if low< high:
        index=partition(array,low,high)
        quick_sort(array,low,index-1)
        quick_sort(array,index+1,high)
```

```
partition(array, low, high):
    pivot ← array[high]
    i ← low-1
    for j=0 to high :
        if array[j] < pivot:
                i ← i +1
                swap(array[i] & array[j])
    swap( array[i+1] and array[high])
    return (i+1)
```

Tracing visualization:



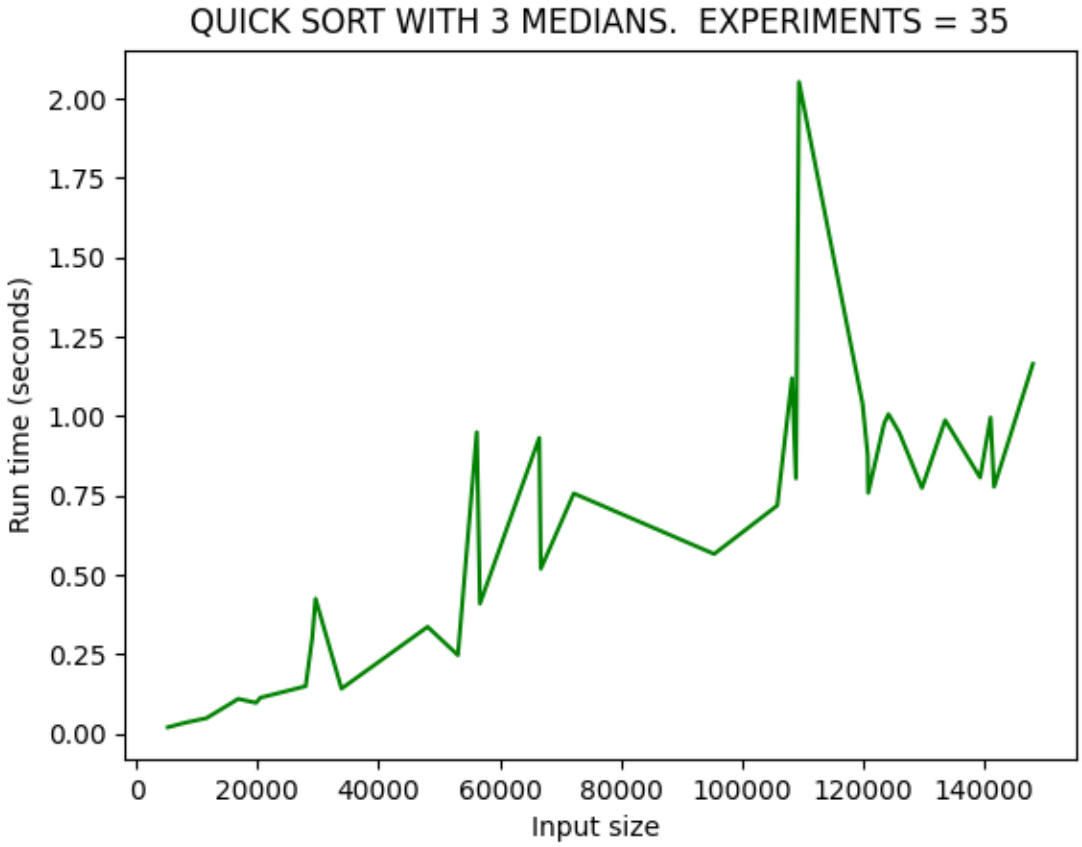
Time Complexity:

Best case	Avg case	Worst case
O(nlog(n))	O(nlog(n))	O(nlog(n))

Experimental analysis:

The Quick sort using 3 medians function was executed 35 times with different number of inputs ranging from 10-150,000 in each experiment.

The following graph shows the Input size vs Run time of Quick sort using 3 medians algorithm. It is clear from the graph that Quick sort using 3 medians has a time complexity of $\Theta(nlog(n))$.



7. HEAP SORT

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. We first find the minimum/maximum element and place the minimum/maximum element at the beginning and then heapify the entire tree. We repeat the same process for the remaining elements.

Data Structures used: Binary heap tree, array

Functions used:

- heap_sort( ) : builds a min/max heap from the input array data. Replaces the smallest/largest element at the root with the last element of the heap, thus reducing the size of heap by 1 and then calls the heapify( ) function to heapify the tree. Same process is repeated for all elements in the heap.
- heapify( ) : the parent node is compared with its children and based on whether it's a min/max heap, the appropriate swaps are made.

Input: binary heap tree of size 'n'

Output: sorted binary heap tree in array[ ] representation

Algorithm: (using max heap)

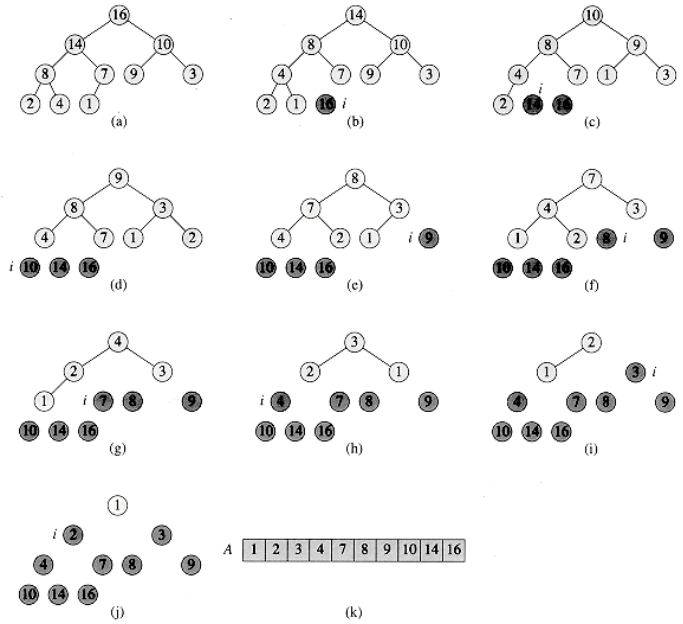
```
heap_sort(array):
    build a max heap from given array
    for all elements in heap:
        replace largest at the root node with the last element of heap
        reduce the size of heap by 1 and heapify( ) the root.

heapify(root ):
while max_heap property holds for all elements:
    if root < left_child :
        swap( root & left_child)
    if root < right_child :
        swap( root & right_child)
```

Time Complexity:

Best case	Avg case	Worst case
O(nlog(n))	O(nlog(n))	O(nlog(n))

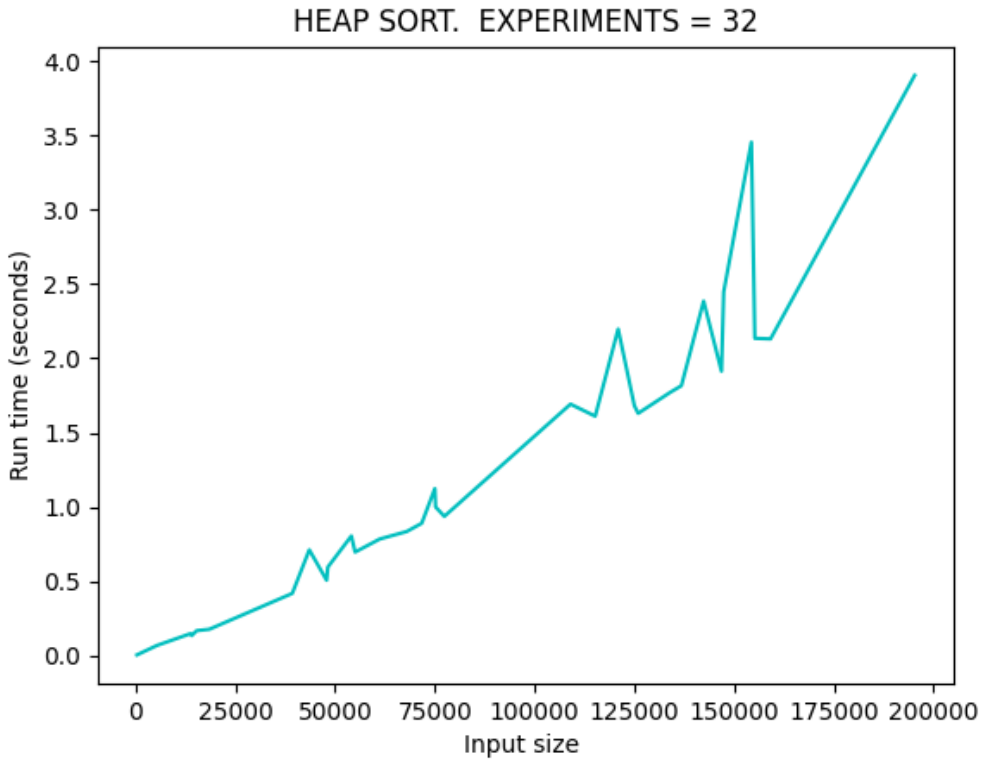
Tracing visualization:



Experimental analysis:

The Heap sort function was executed 32 times with different number of inputs ranging from 10-200,000 in each experiment.

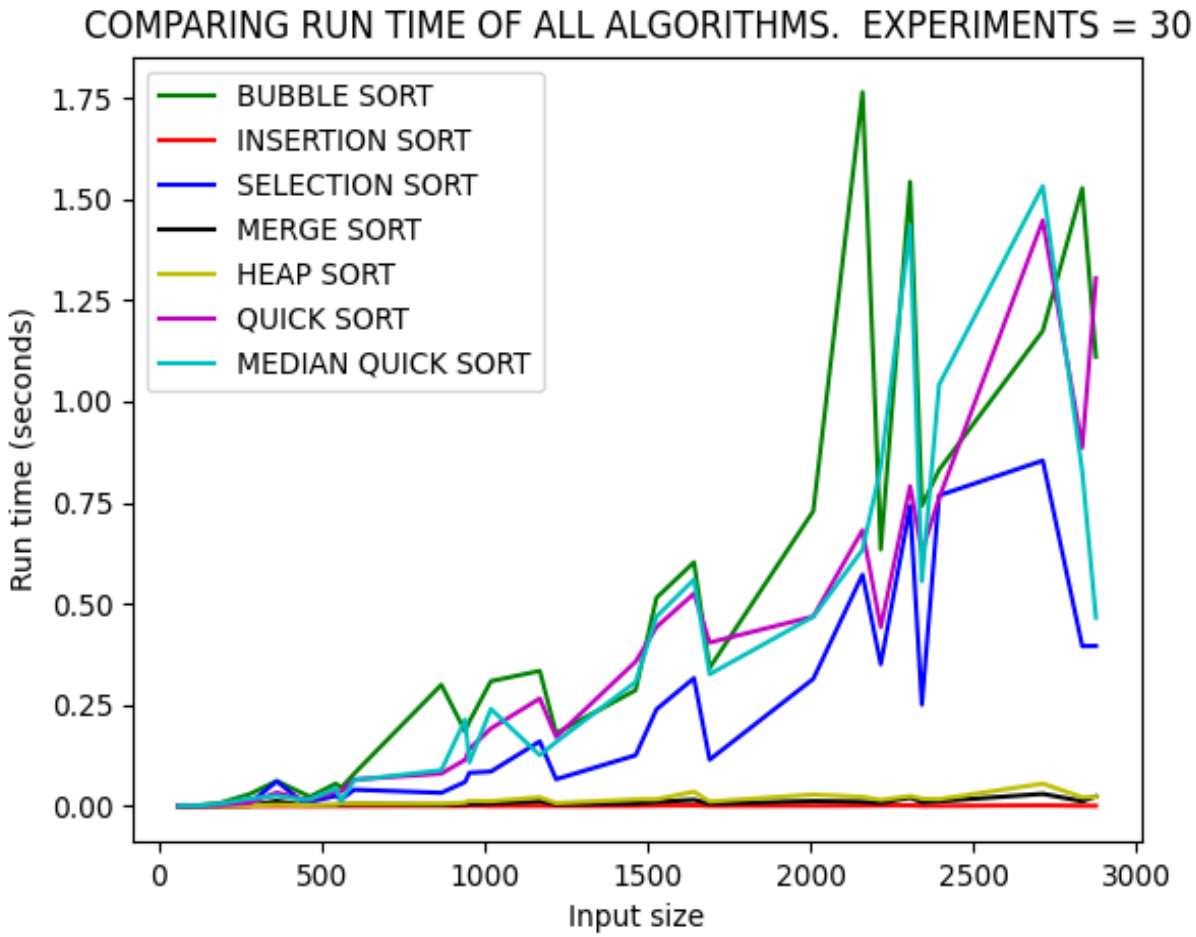
The following graph shows the Input size vs Run time of Heap sort algorithm. It is clear from the graph that Heap sort has a time complexity of $\Theta(nlog(n))$.



COMPARISON OF ALL ALGORITHMS

All the 7 algorithms, **Bubble sort, Insertion sort, Selection sort, Merge sort, Quick sort, Quick sort using three medians and Heap sort** are compared against the **"same input of unsorted numbers".**

We have a combination of algorithms whose run times vary from $\Theta(nlog(n))$ **to** $\Theta(n^2)$, hence we can see which algorithm takes the least amount of time to sort the given input.

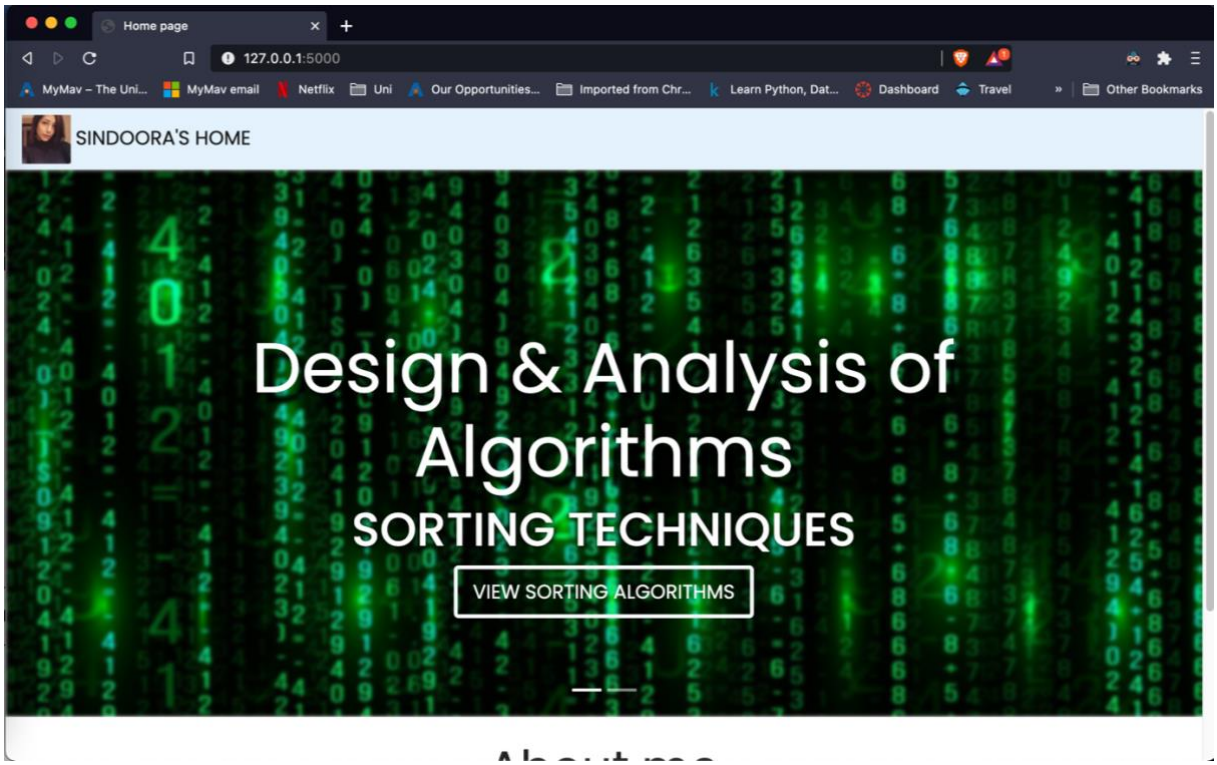


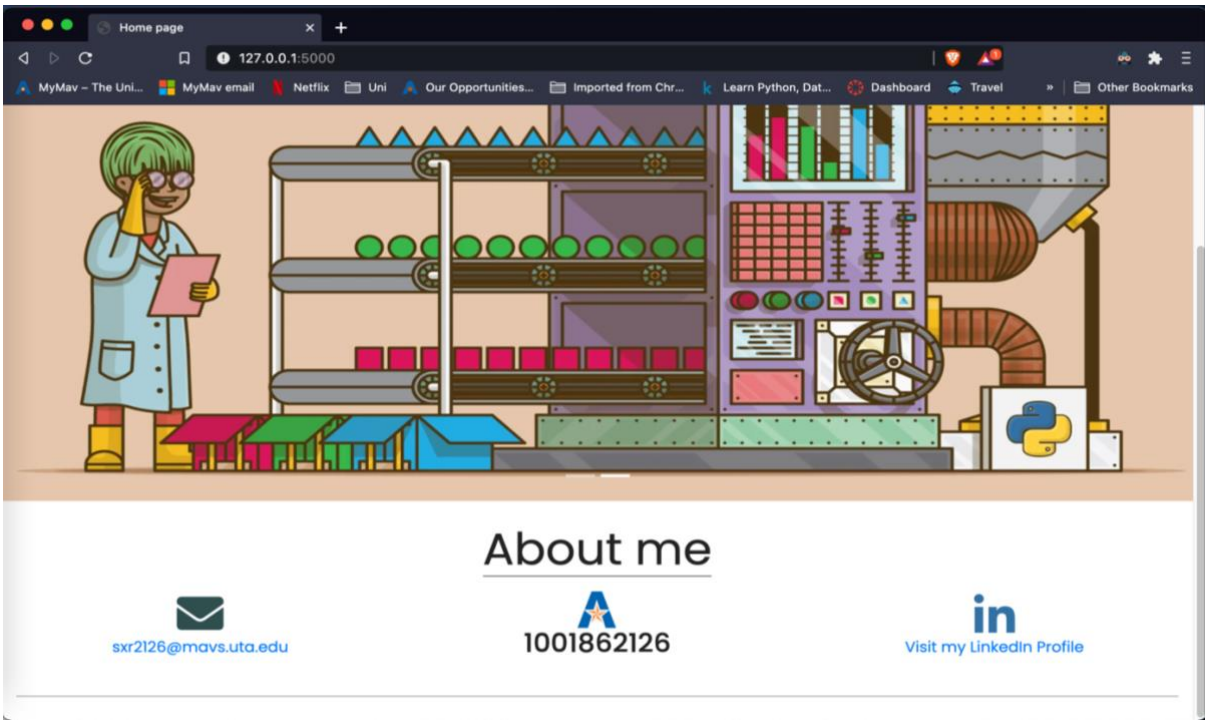
GRAPHICAL USER INTERFACE (GUI)

A GUI is created for the user to view the different sorting algorithms and select each of the sorting algorithm for 'n' number of experiments and inputs to see how the Run time varies with Input size.

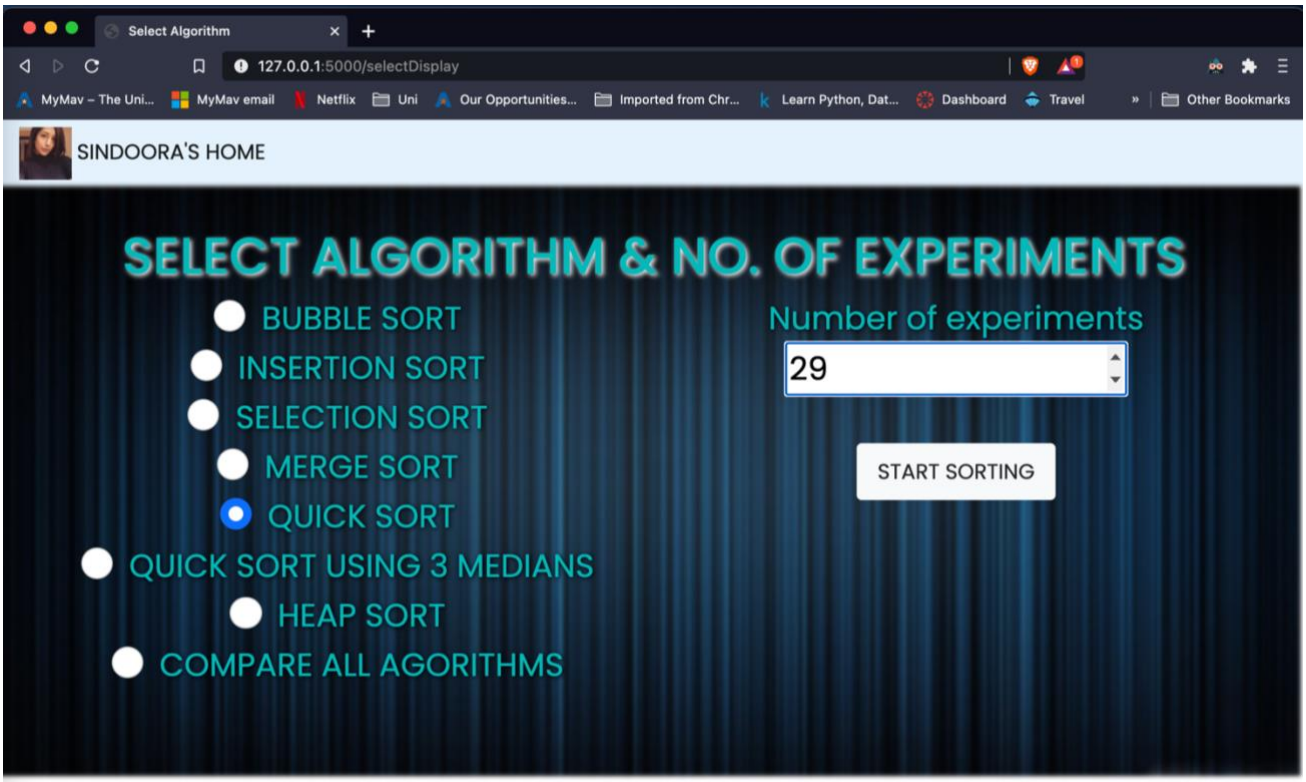
The GUI is built using **Python – Flask**, Front end is designed using **HTML5 and CSS.**

Page 1: Home Page (Scrolling)

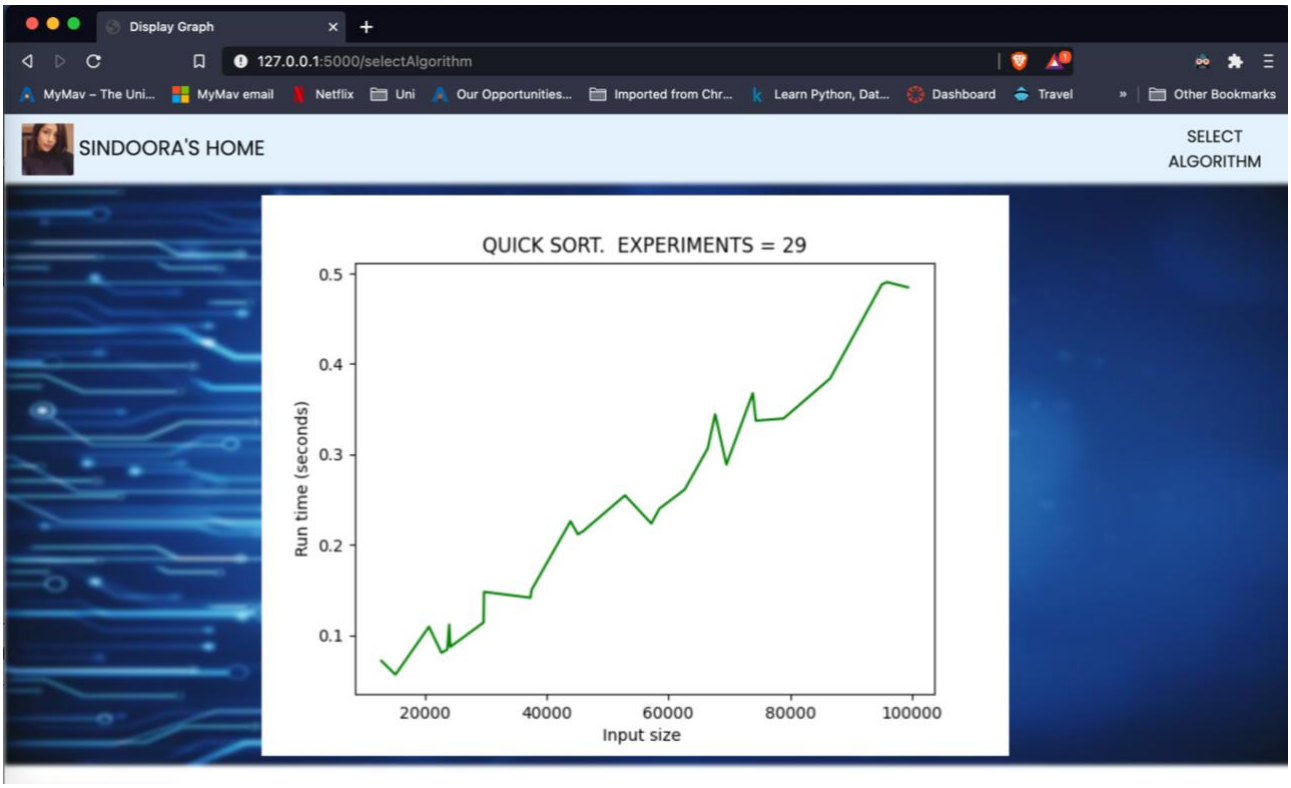




Page 2: Select Algorithm and enter number of experiments to perform with random input sizes (Eg: Input size :10-20000 which is generated using random number generation)



Page 3: Graph displaying Input size vs Run time for the algorithm



CONCLUSION

The results shown by the graphs confirm the time complexities of each of the algorithms.

- Citation:

  Tracing visualization screenshots for all algorithms – www.google.com