Name: Niranjana Subramanian
Student ID: 10020246305
Subject code and Course: CSE5311 Design and Analysis of Algorithms
Section: 006

Programming Assignment

Introduction:

I have chosen Project1 Sorting Algorithms as my programming assignment. Sorting algorithms are used to sort an array or list in either ascending or descending order. I have implemented Bubble sort, Insertion sort, Merge sort, Heap sort, Regular Quick sort, Quick sort with 3 medians, and Selection sort using python on Visual Studio Code as the IDE. I have also implemented GUI using Flask, Html, CSS, and Bootstrap. The comparison between each and all algorithms has been done using Google Charts.

Algorithms:

1. Bubble Sort:

This sorting algorithm checks each of the adjacent elements in the string from left to right and swaps them when they are out of order.

Time Complexity:

Best Case	Worst Case
O(n)	O(n^2)

```
def bub_sort(arr):
    sort = arr[:]
    init_time = time.time()
    for j in range(len(sort)):
        for l in range(len(sort)-j-1):
            if sort[l+1] < sort[l]:
                sort[l], sort[l+1] = sort[l+1], sort[l]
    final_time = time.time()
    exec_time = final_time - init_time
    return sort,exec_time
```

Name: Niranjana Subramanian
Student ID: 10020246305
Subject code and Course: CSE5311 Design and Analysis of Algorithms
Section: 006

Programming Assignment

2. Insertion Sort:

This sorting algorithm divides the array into two parts sorted and unsorted. Values from the unsorted part are selected and put into the correct position in the sorted part.

Time Complexity:

Best Case	Worst Case
O(n)	O(n^2)

```
def insert_sort_f(arr):
    sort = arr[:]
    init_time = time.time()
    for n in range(1,len(sort)):
        k = sort[n]
        i = n
        while i > 0 and k < sort[i-1]:
            sort[i] = sort[i-1]
            i -= 1
        sort[i] = k
    final_time = time.time()
    exec_time = final_time - init_time
    return sort,exec_time
```

3. Merge Sort:

This sorting algorithm divides the list into halves and then iterates through the 2 halves, and halves them down further into their smaller parts so that it can sort easily.

Time Complexity:

Best Case	Worst Case
O(nlogn)	O(nlogn)

Name: Niranjana Subramanian
Student ID: 10020246305
Subject code and Course: CSE5311 Design and Analysis of Algorithms
Section: 006

Programming Assignment

```
def msort(arr_l,arr_r):

    sort = []
    l = 0
    r = 0
    while len(sort) < len(arr_r) + len(arr_l):
        if arr_l[l] <= arr_r[r]:
            sort.append(arr_l[l])
            l += 1
        else:
            sort.append(arr_r[r])
            r += 1

        if len(arr_l)  == l:
            sort.extend(arr_r[r:])
            break
        if len(arr_r)  == r:
            sort.extend(arr_l[l:])
            break

    return sort

def msort_f(arr):

    if len(arr) < 2:
        return arr

    m = len(arr)//2

    r_arr = msort_f(arr[m:])
    l_arr = msort_f(arr[0:m])

    sort = msort(l_arr,r_arr)

    return sort
```

4. Heap Sort:

This sorting algorithm is a comparison based on the binary heap data structure. It finds the minimum element and places it at the beginning. The same process is repeated until all elements are sorted.

Time Complexity:

Best Case	Worst Case
O(nlogn)	O(nlogn)

Name: Niranjana Subramanian
Student ID: 10020246305
Subject code and Course: CSE5311 Design and Analysis of Algorithms
Section: 006

Programming Assignment

```
def hsort(sort, m, n):
    max = n
    lnode_ = 2*n + 1
    rnode_ = 2*n + 2
    sort
    if lnode_ < m and sort[max] < sort[lnode_]:
        max = lnode_

    if rnode_ < m and sort[max] < sort[rnode_]:
        max = rnode_

    if max != n:
        sort[max], sort[n] =  sort[n], sort[max]
        hsort(sort, m, max)


def hsort_f(arr):
    sort = arr[:]
    i_t = time.time()
    alen_ = len(sort)

    for i in range(alen_//2 -1, -1, -1):
        hsort(sort,alen_, i )

    for i in range(alen_-1 , 0, -1):
        sort[0], sort[i] = sort[i] , sort[0]
        hsort(sort,i, 0 )
    f_t = time.time()
    ex_time = f_t - i_t
    return sort,ex_time
```

5. Quick Sort:

This sorting algorithm divides the whole array into 2 parts based on the pivot. One sub-array contains values lesser than the pivot and the other sub-array contains values greater than the pivot.

Time Complexity:

Best Case	Worst Case
O(nlogn)	O(n^2)

Name: Niranjana Subramanian
Student ID: 10020246305
Subject code and Course: CSE5311 Design and Analysis of Algorithms
Section: 006

Programming Assignment

- Regular Quicksort (using the last element as the pivot)

```
def qsort(sort,s,e):
    a = s-1
    q = sort[e]
    for m in range(s,e,1):
        if sort[m] < q:
            a+=1
            sort[m], sort[a] = sort[a], sort[m]
    sort[e], sort[a+1] = sort[a+1], sort[e]
    return a+1


def qsort_f(sort,s,e):

    if len(sort) == 1:
        return sort

    if s < e:
        pindex_ = qsort(sort,s,e)

        qsort_f(sort, s, pindex_-1)
        qsort_f(sort, pindex_+1, e)
```

- 3 Median

```
def qsort_3(sort,s,e):
    f_pivot = sort[s]
    mid_pivot = sort[e//2]
    l_pivot = sort[e]
    m_arr = [f_pivot,mid_pivot,l_pivot]
    m_arr.sort()
    median = m_arr[1]
    sort[s] = m_arr[0]
    sort[e] = m_arr[1]
    sort[e//2] = m_arr[2]


def qsort_3f(sort,s,e):
    if s < e:
        qsort_3(sort,s,e)
        qsort_f(sort,s,e)
```

Name: Niranjana Subramanian
Student ID: 10020246305
Subject code and Course: CSE5311 Design and Analysis of Algorithms
Section: 006

Programming Assignment

6. Selection Sort:

This sorting algorithm works by bringing the smallest element in the array by bringing it to the beginning of the array.

Time Complexity:

Best Case	Worst Case
O(n^2)	O(n^2)

```
def select_sort(arr):
    sort = arr[:]
    init_time = time.time()
    for j in range(0,len(sort)-1):
        min = j
        for l in range(j+1, len(sort)):
            if  sort[min] > sort[l]:
                min = l
        sort[min], sort[l] = sort[j], sort[min]
    final_time = time.time()
    exec_time = final_time - init_time
    return sort,exec_time
```

GUI Implementation:

1. Enter the length of the array.
2. Select the sorting algorithm.
3. Click the ‘Click to Sort’ button.
4. On the same page you get the input with random numbers and the output of the sorted array.
5. Time complexity is calculated and given in Milliseconds at the end of the page.
6. To compare the time complexity of each and all algorithms against the input size, click the ‘To Compare Algorithms’ link at the top of the page.

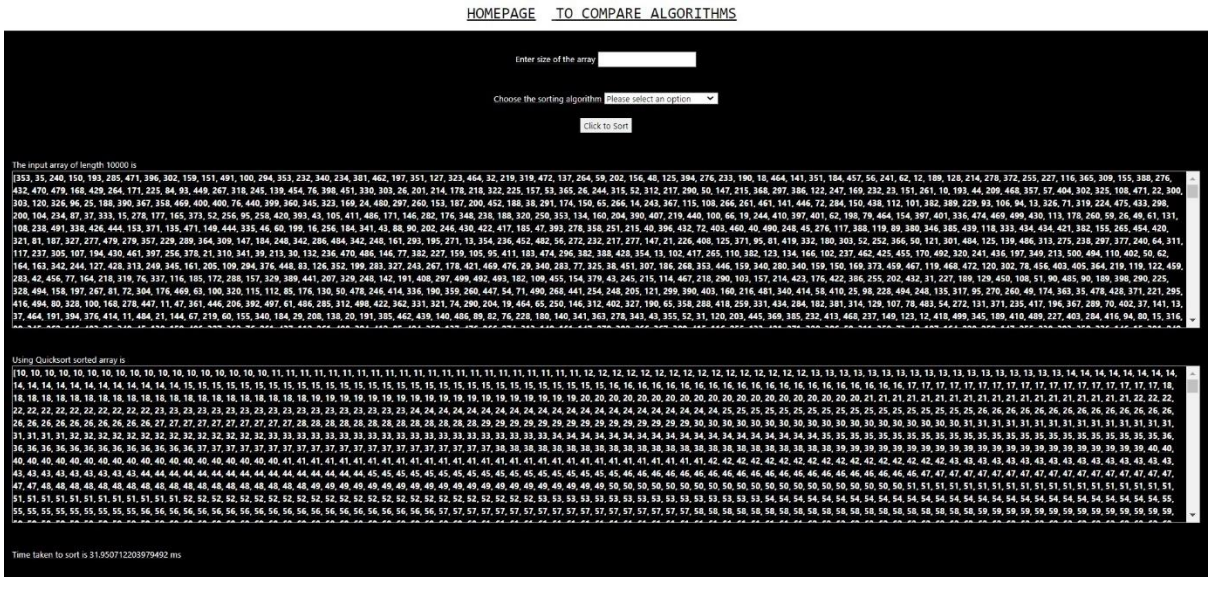
Name: Niranjana Subramanian
Student ID: 10020246305
Subject code and Course: CSE5311 Design and Analysis of Algorithms
Section: 006

Programming Assignment

7. It directs you to a page where you enter the input size of the array.
8. Click ‘Click to Compare.
9. It produces the result in a graph on a new redirected page.

The following are the snapshots of the GUI:

home.html:



Name: Niranjana Subramanian
Student ID: 10020246305
Subject code and Course: CSE5311 Design and Analysis of Algorithms
Section: 006

Programming Assignment

compare.html

HOMEPAGE TO COMPARE ALGORITHMS

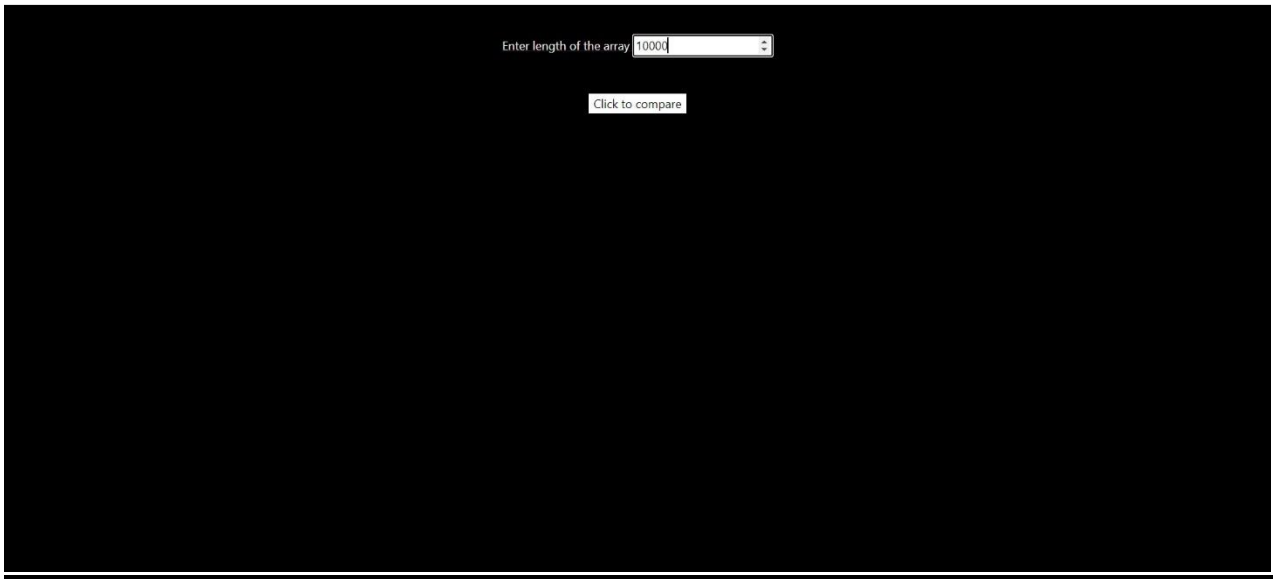
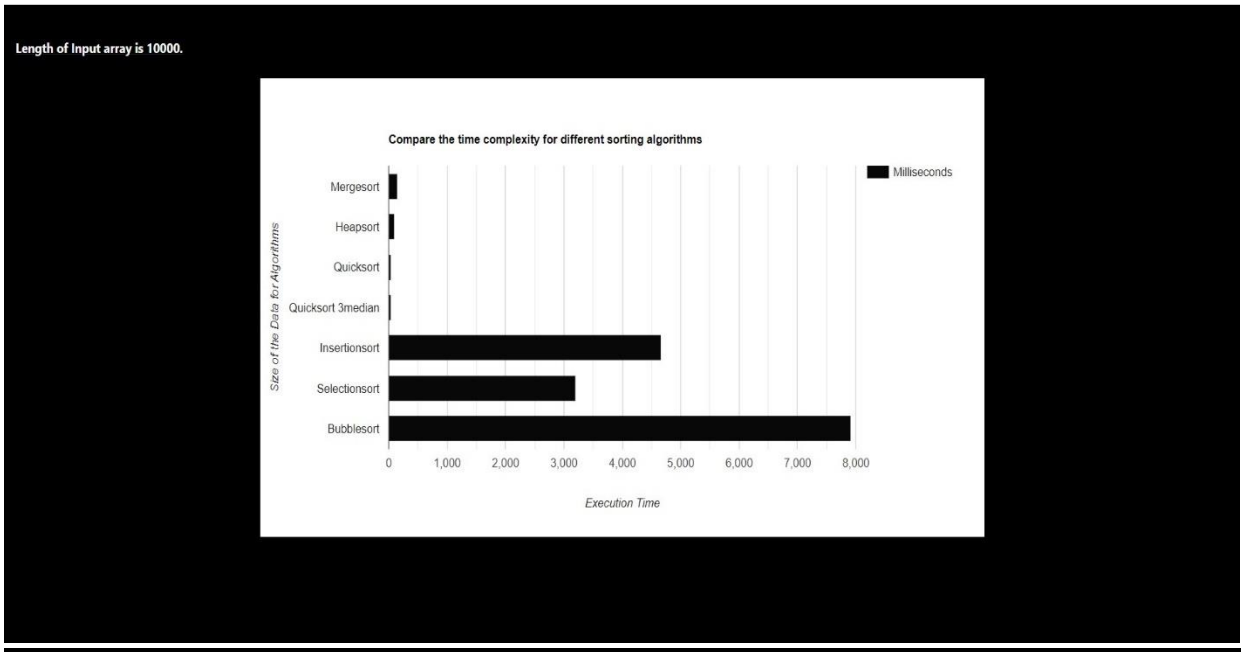


chart.html

HOMEPAGE TO COMPARE ALGORITHMS

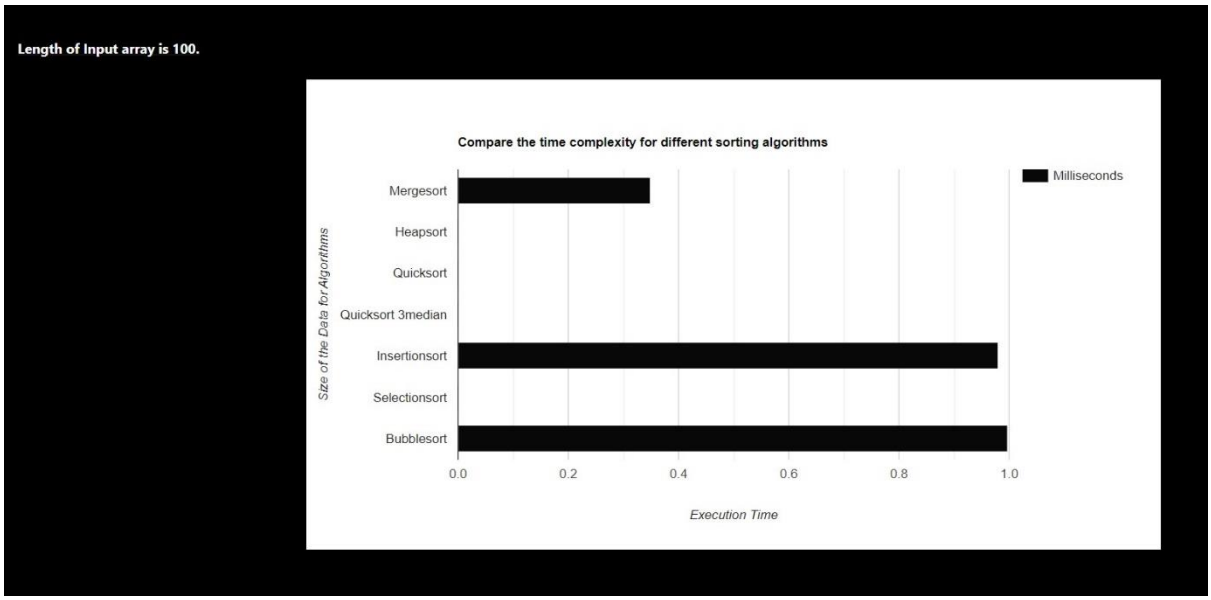


Name: Niranjana Subramanian
Student ID: 10020246305
Subject code and Course: CSE5311 Design and Analysis of Algorithms
Section: 006

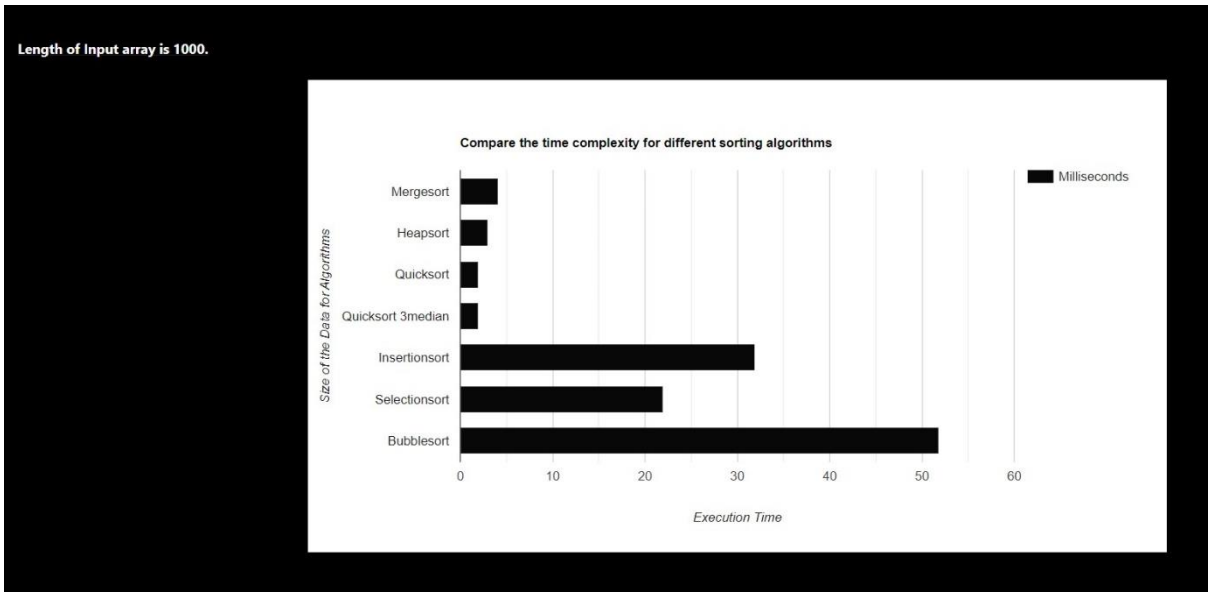
Programming Assignment

Graph Analysis:

- When the input size is 100:



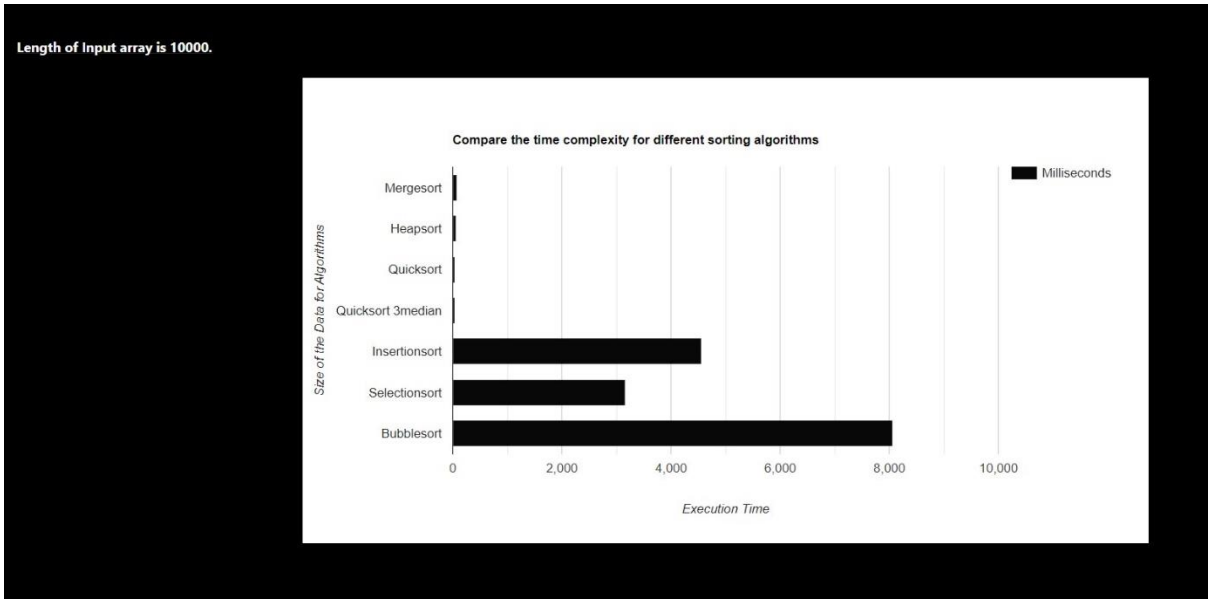
- When the input size is 1000:



Name: Niranjana Subramanian
Student ID: 10020246305
Subject code and Course: CSE5311 Design and Analysis of Algorithms
Section: 006

Programming Assignment

- When the input size is 10000:



We can see from the above snapshots that as the input size increases, the time complexity also increases. Bubble Sort consumes a lot of time and is hence not efficient. Hence for huge input sizes, sorting algorithms like Quick Sort, Heap Sort, and Merge Sort is preferred, and for smaller input sizes, sorting algorithms such as Insertion Sort and Selection Sort is preferred.

Conclusion:

The project compares each sorting algorithm against the input size. For each input size, the time complexity for each sorting algorithm is calculated and the graph analysis is done.