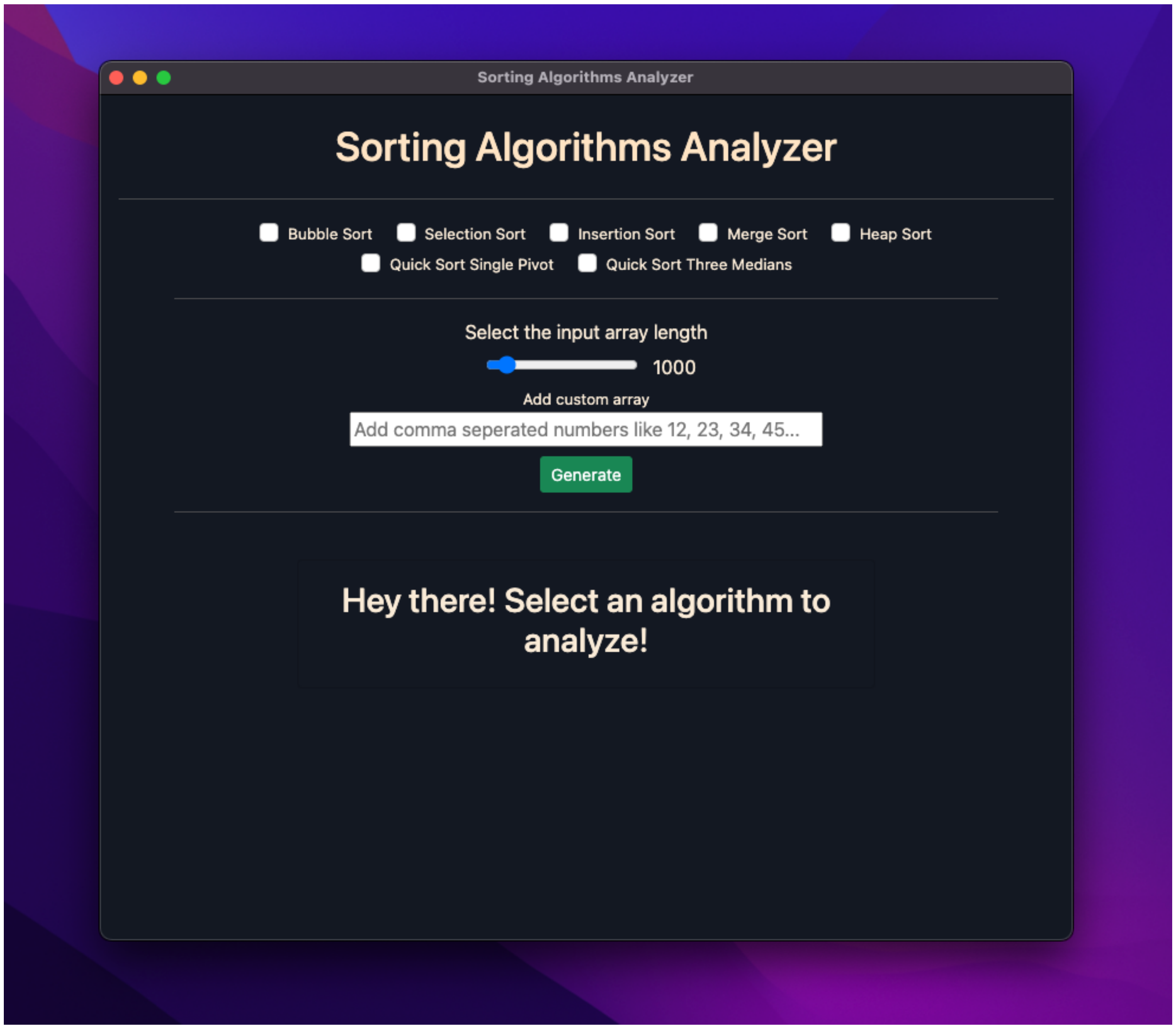
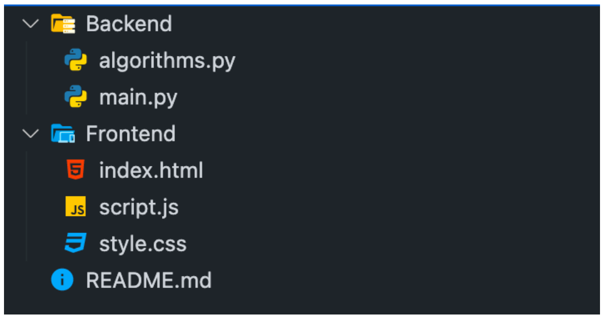
Soham Santosh Tembe
UTA ID: 1001960847
5311 Programming Project

Introduction


Sorting Algorithms Analyzer
Sorting Algorithms Analyzer
Bubble Sort
Selection Sort
Insertion Sort
Merge Sort
Heap Sort
Quick Sort Single Pivot
Quick Sort Three Medians
Select the input array length
1000
Add custom array
Add comma seperated numbers like 12, 23, 34, 45...
Generate
Hey there! Select an algorithm to analyze!
Sorting Algorithms Analyzer
(Figure 1)

I have selected Project 1 (Sorting Algorithms) as my programming project for Design and Analysis of Algorithms. The project mainly uses two programming languages Python and JavaScript to complete the Backend and the Frontend of the project.

File Structure

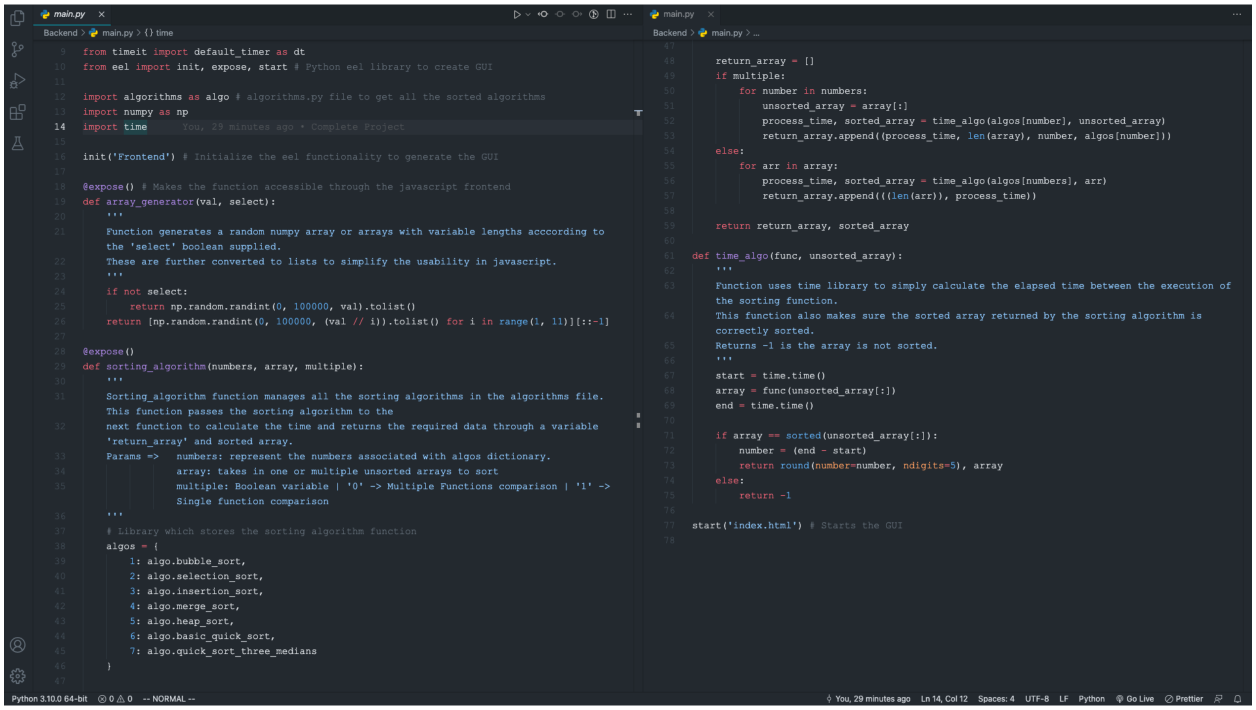


```
Backend
    algorithms.py
    main.py
Frontend
    index.html
    script.js
    style.css
README.md
```

(Figure 2)

The folder tree structure looks like as represented in Figure 2. The folder Backend contains all the Python files which handle the backend processing of the GUI. The Frontend of the project is mainly handled by 'script.js' file while the 'index.html', 'style.css' works as the structuring and styling to the GUI.

Python Backend Files



```python
from timeit import default_timer as dt
from eel import init, expose, start # Python eel library to create GUI

import algorithms as algo # algorithms.py file to get all the sorted algorithms
import numpy as np
import time

init('Frontend') # Initialize the eel functionality to generate the GUI

@expose() # Makes the function accessible through the javascript frontend
def array_generator(val, select):
    '''
    Function generates a random numpy array or arrays with variable lengths acccording to
    the 'select' boolean supplied.
    These are further converted to lists to simplify the usability in javascript.
    '''
    if not select:
        return np.random.randint(0, 100000, val).tolist()
    return [np.random.randint(0, 100000, (val // i)).tolist() for i in range(1, 11)][::-1]

@expose()
def sorting_algorithm(numbers, array, multiple):
    '''
    Sorting_algorithm function manages all the sorting algorithms in the algorithms file.
    This function passes the sorting algorithm to the
    next function to calculate the time and returns the required data through a variable
    'return_array' and sorted array.
    Params =>   numbers: represent the numbers associated with algos dictionary.
                array: takes in one or multiple unsorted arrays to sort
                multiple: Boolean variable | '0' -> Multiple Functions comparison | '1' ->
                Single function comparison
    '''
    # Library which stores the sorting algorithm function
    algos = {
        1: algo.bubble_sort,
        2: algo.selection_sort,
        3: algo.insertion_sort,
        4: algo.merge_sort,
        5: algo.heap_sort,
        6: algo.basic_quick_sort,
        7: algo.quick_sort_three_medians
    }

    return_array = []
    if multiple:
        for number in numbers:
            unsorted_array = array[:]
            process_time, sorted_array = time_algo(algos[number], unsorted_array)
            return_array.append((process_time, len(array), number, algos[number]))
    else:
        for arr in array:
            process_time, sorted_array = time_algo(algos[numbers], arr)
            return_array.append(((len(arr)), process_time))

    return return_array, sorted_array

def time_algo(func, unsorted_array):
    '''
    Function uses time library to simply calculate the elapsed time between the execution of
    the sorting function.
    This function also makes sure the sorted array returned by the sorting algorithm is
    correctly sorted.
    Returns -1 is the array is not sorted.
    '''
    start = time.time()
    array = func(unsorted_array[:])
    end = time.time()

    if array == sorted(unsorted_array[:]):
        number = (end - start)
        return round(number=number, ndigits=5), array
    else:
        return -1

start('index.html') # Starts the GUI
```

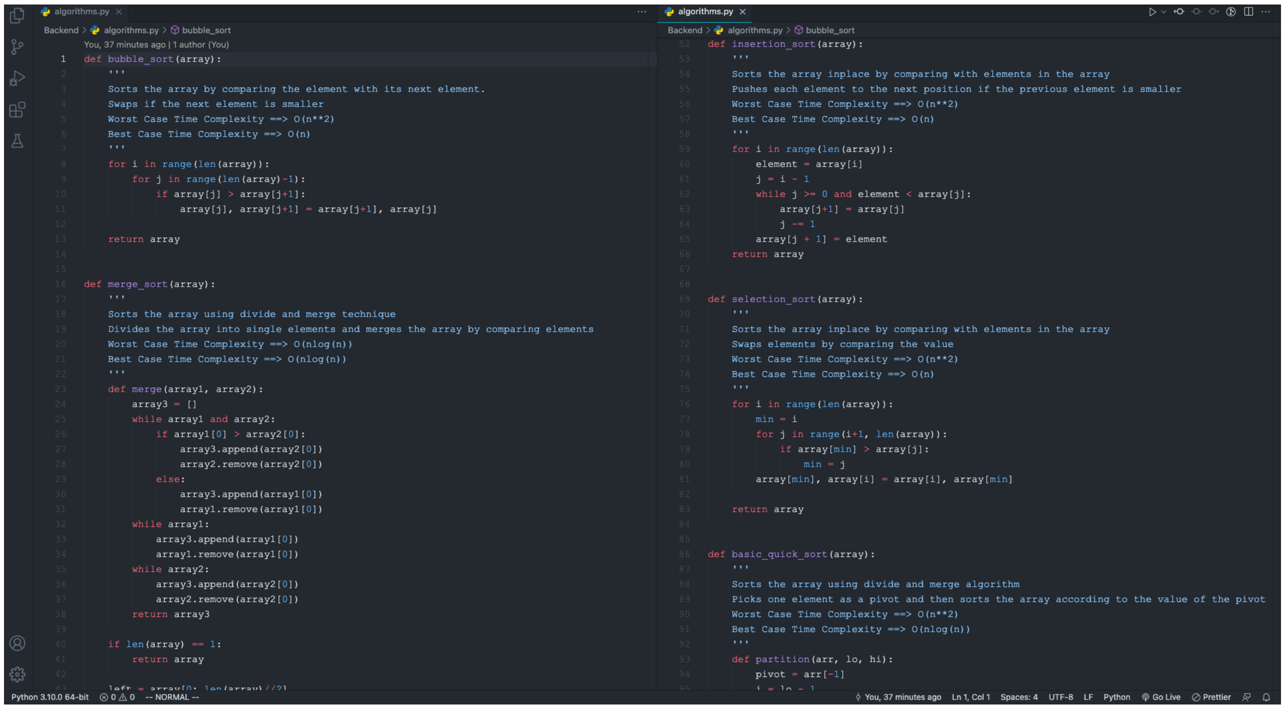
(Figure 3)

The 'main.py' is responsible to completely run the program from the start. It uses Python Eel library which helps create the GUI that can be communicated with using JavaScript, HTML and CSS.

The file includes packages like Python Eel, NumPy, time for different uses along the way.

The data structures used includes a dictionary which stores the references to the sorting algorithm functions from the 'algorithms.py' file. All the inputs to the sorting algorithms are stored in python lists.

This file contains three functions, one which generates randomized unsorted NumPy arrays converted to python lists according to the input given by the user from the GUI. The second function provides the desired output to the Frontend by deciding whether the output is for single algorithm results or multiple algorithm comparison. The third function calculates the time required by any sorting algorithm to execute the function and verifies whether the output given by the algorithm is valid by checking with the python built-in sorting method.



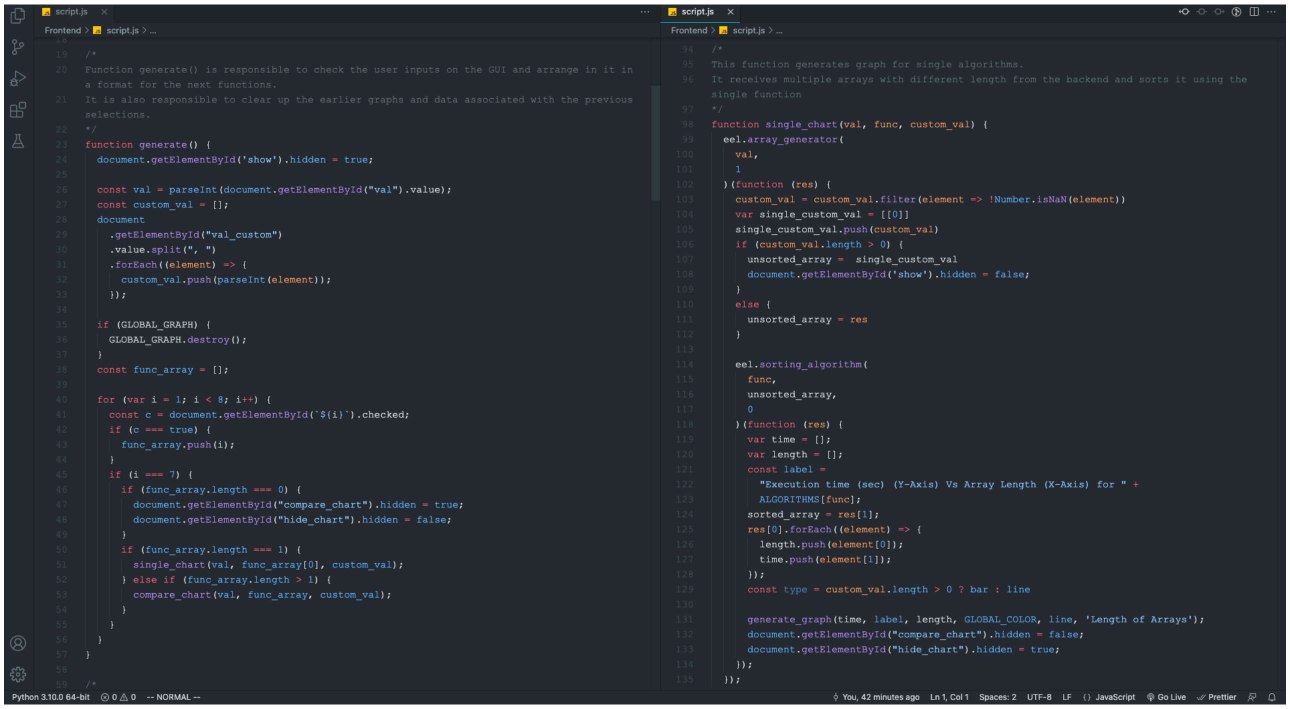
```
def bubble_sort(array):
    '''
    Sorts the array by comparing the element with its next element.
    Swaps if the next element is smaller
    Worst Case Time Complexity ==> O(n**2)
    Best Case Time Complexity ==> O(n)
    '''
    for i in range(len(array)):
        for j in range(len(array)-1):
            if array[j] > array[j+1]:
                array[j], array[j+1] = array[j+1], array[j]

    return array


def merge_sort(array):
    '''
    Sorts the array using divide and merge technique
    Divides the array into single elements and merges the array by comparing elements
    Worst Case Time Complexity ==> O(nlog(n))
    Best Case Time Complexity ==> O(nlog(n))
    '''
    def merge(array1, array2):
        array3 = []
        while array1 and array2:
            if array1[0] > array2[0]:
                array3.append(array2[0])
                array2.remove(array2[0])
            else:
                array3.append(array1[0])
                array1.remove(array1[0])
        while array1:
            array3.append(array1[0])
            array1.remove(array1[0])
        while array2:
            array3.append(array2[0])
            array2.remove(array2[0])
        return array3

    if len(array) == 1:
        return array

    left = array[0: len(array)//2]
```

```
def insertion_sort(array):
    '''
    Sorts the array inplace by comparing with elements in the array
    Pushes each element to the next position if the previous element is smaller
    Worst Case Time Complexity ==> O(n**2)
    Best Case Time Complexity ==> O(n)
    '''
    for i in range(len(array)):
        element = array[i]
        j = i - 1
        while j >= 0 and element < array[j]:
            array[j+1] = array[j]
            j -= 1
        array[j + 1] = element
    return array


def selection_sort(array):
    '''
    Sorts the array inplace by comparing with elements in the array
    Swaps elements by comparing the value
    Worst Case Time Complexity ==> O(n**2)
    Best Case Time Complexity ==> O(n)
    '''
    for i in range(len(array)):
        min = i
        for j in range(i+1, len(array)):
            if array[min] > array[j]:
                min = j
        array[min], array[i] = array[i], array[min]

    return array


def basic_quick_sort(array):
    '''
    Sorts the array using divide and merge algorithm
    Picks one element as a pivot and then sorts the array according to the value of the pivot
    Worst Case Time Complexity ==> O(n**2)
    Best Case Time Complexity ==> O(nlog(n))
    '''
    def partition(arr, lo, hi):
        pivot = arr[-1]
        i = lo - 1
```

(Figure 4)

Figure 4 shows the file 'algorithms.py' which includes all the functions with sorting algorithms. All the functions use python lists as input and return the sorted python list. The file also includes comments noting the time complexity and related information about each of the function in the file.

Frontend Files



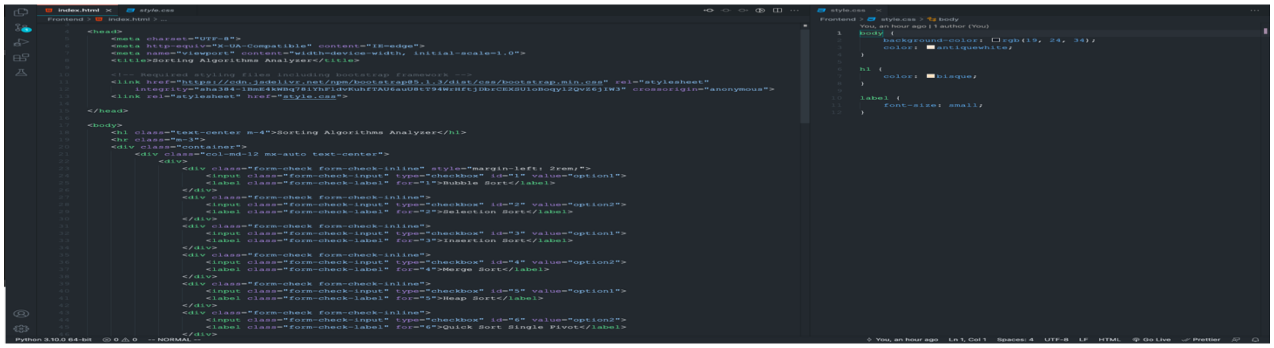
```javascript
/*
Function generate() is responsible to check the user inputs on the GUI and arrange in it in
a format for the next functions.
It is also responsible to clear up the earlier graphs and data associated with the previous
selections.
*/
function generate() {
  document.getElementById('show').hidden = true;

  const val = parseInt(document.getElementById("val").value);
  const custom_val = [];
  document
    .getElementById("val_custom")
    .value.split(", ")
    .forEach((element) => {
      custom_val.push(parseInt(element));
    });

  if (GLOBAL_GRAPH) {
    GLOBAL_GRAPH.destroy();
  }
  const func_array = [];

  for (var i = 1; i < 8; i++) {
    const c = document.getElementById(`${i}`).checked;
    if (c === true) {
      func_array.push(i);
    }
    if (i === 7) {
      if (func_array.length === 0) {
        document.getElementById("compare_chart").hidden = true;
        document.getElementById("hide_chart").hidden = false;
      }
      if (func_array.length === 1) {
        single_chart(val, func_array[0], custom_val);
      } else if (func_array.length > 1) {
        compare_chart(val, func_array, custom_val);
      }
    }
  }
}

/*
```

```javascript
/*
This function generates graph for single algorithms.
It receives multiple arrays with different length from the backend and sorts it using the
single function
*/
function single_chart(val, func, custom_val) {
  eel.array_generator(
    val,
    1
  )(function (res) {
    custom_val = custom_val.filter(element => !Number.isNaN(element))
    var single_custom_val = [[0]]
    single_custom_val.push(custom_val)
    if (custom_val.length > 0) {
      unsorted_array =  single_custom_val
      document.getElementById('show').hidden = false;
    }
    else {
      unsorted_array = res
    }

    eel.sorting_algorithm(
      func,
      unsorted_array,
      0
    )(function (res) {
      var time = [];
      var length = [];
      const label =
        "Execution time (sec) (Y-Axis) Vs Array Length (X-Axis) for " +
        ALGORITHMS[func];
      sorted_array = res[1];
      res[0].forEach((element) => {
        length.push(element[0]);
        time.push(element[1]);
      });
      const type = custom_val.length > 0 ? bar : line

      generate_graph(time, label, length, GLOBAL_COLOR, line, 'Length of Arrays');
      document.getElementById("compare_chart").hidden = false;
      document.getElementById("hide_chart").hidden = true;
    });
  });
```

(Figure 5)

The Frontend includes the JavaScript file shown in (Figure 5). The file includes functions that send API requests to the backend using the eel library functions generated through python. This file uses JavaScript Hash maps and lists to store the inputs and outputs of the algorithms. This file is mainly responsible for the User and Backend interaction.

The JavaScript file uses the inputs from the user and the outputs from the Backend to generate charts using the Chart.js library. The basic usage code snippet on Chart.js website has been used to build the charts. The functions introduced in the JavaScript library decide the type of the chart generated.



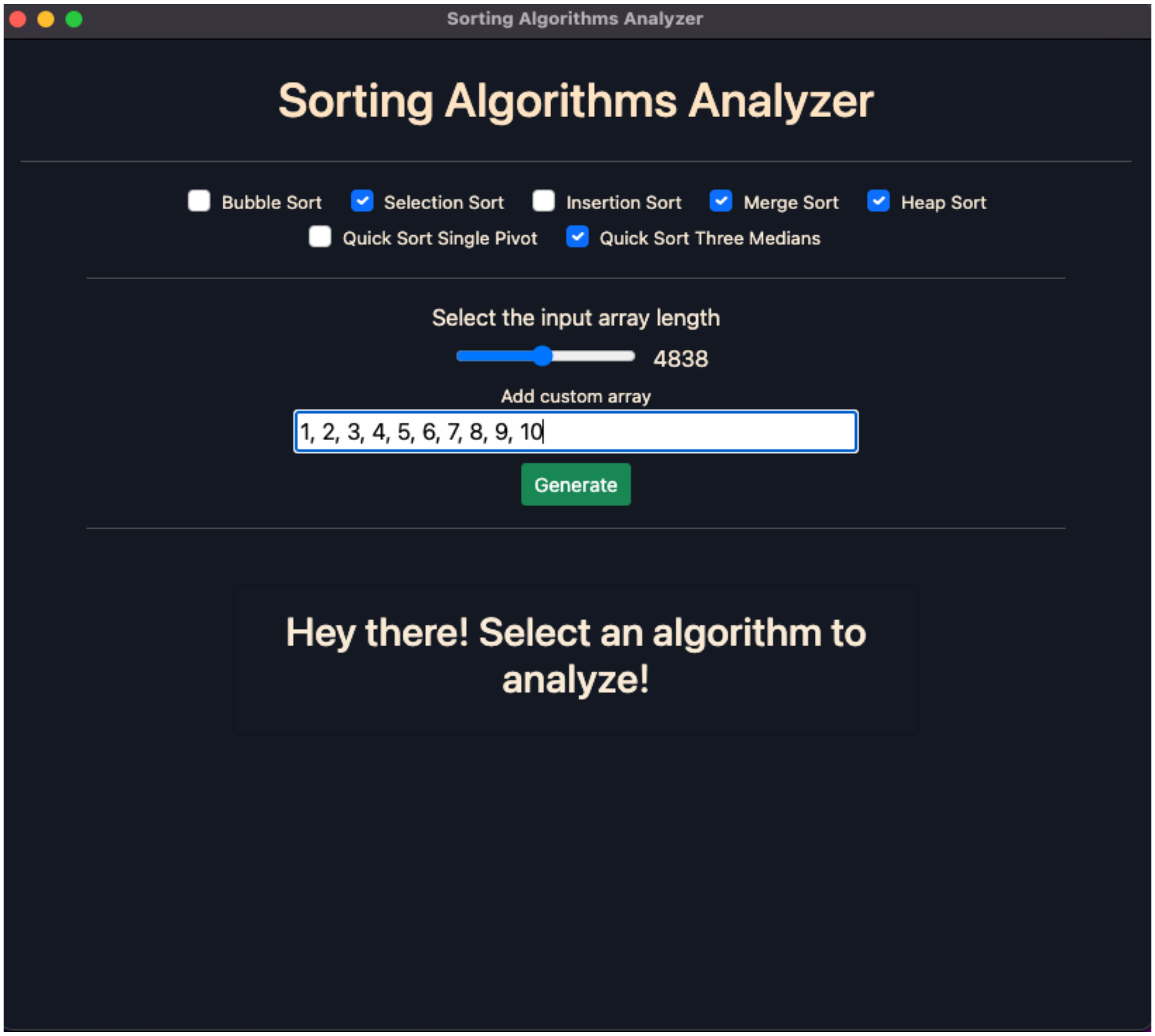
```html
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Sorting Algorithms Analyzer</title>

    <!-- Required styling files including bootstrap framework -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet"
        integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
    <link rel="stylesheet" href="style.css">

</head>

<body>
    <h1 class="text-center m-4">Sorting Algorithms Analyzer</h1>
    <hr class="m-3">
    <div class="container">
        <div class="col-md-12 mx-auto text-center">
            <div>
                <div class="form-check form-check-inline" style="margin-left: 2rem;">
                    <input class="form-check-input" type="checkbox" id="1" value="option1">
                    <label class="form-check-label" for="1">Bubble Sort</label>
                </div>
                <div class="form-check form-check-inline">
                    <input class="form-check-input" type="checkbox" id="2" value="option2">
                    <label class="form-check-label" for="2">Selection Sort</label>
                </div>
                <div class="form-check form-check-inline">
                    <input class="form-check-input" type="checkbox" id="3" value="option1">
                    <label class="form-check-label" for="3">Insertion Sort</label>
                </div>
                <div class="form-check form-check-inline">
                    <input class="form-check-input" type="checkbox" id="4" value="option2">
                    <label class="form-check-label" for="4">Merge Sort</label>
                </div>
                <div class="form-check form-check-inline">
                    <input class="form-check-input" type="checkbox" id="5" value="option1">
                    <label class="form-check-label" for="5">Heap Sort</label>
                </div>
                <div class="form-check form-check-inline">
                    <input class="form-check-input" type="checkbox" id="6" value="option2">
                    <label class="form-check-label" for="6">Quick Sort Single Pivot</label>
                </div>
```

```css
body {
    background-color: rgb(19, 24, 34);
    color: antiquewhite;
}

h1 {
    color: bisque;
}

label {
    font-size: small;
}
```

(Figure 6)

Graphical User Interface Working

The user interface is very interactive and easy-to-use. The Sorted algorithms selection depends on the check boxes selected by the user. All the selected algorithms are then taken into consideration for the further process. The user can scroll the range bar and select the maximum length of the array that needs to be fed to the sorting algorithm. User may add and input array as well in the provided text box. The user input is always given priority over randomly generated arrays for sorting.



Bubble Sort [✓ Selection Sort] Insertion Sort [✓ Merge Sort] [✓ Heap Sort]
Quick Sort Single Pivot [✓ Quick Sort Three Medians]

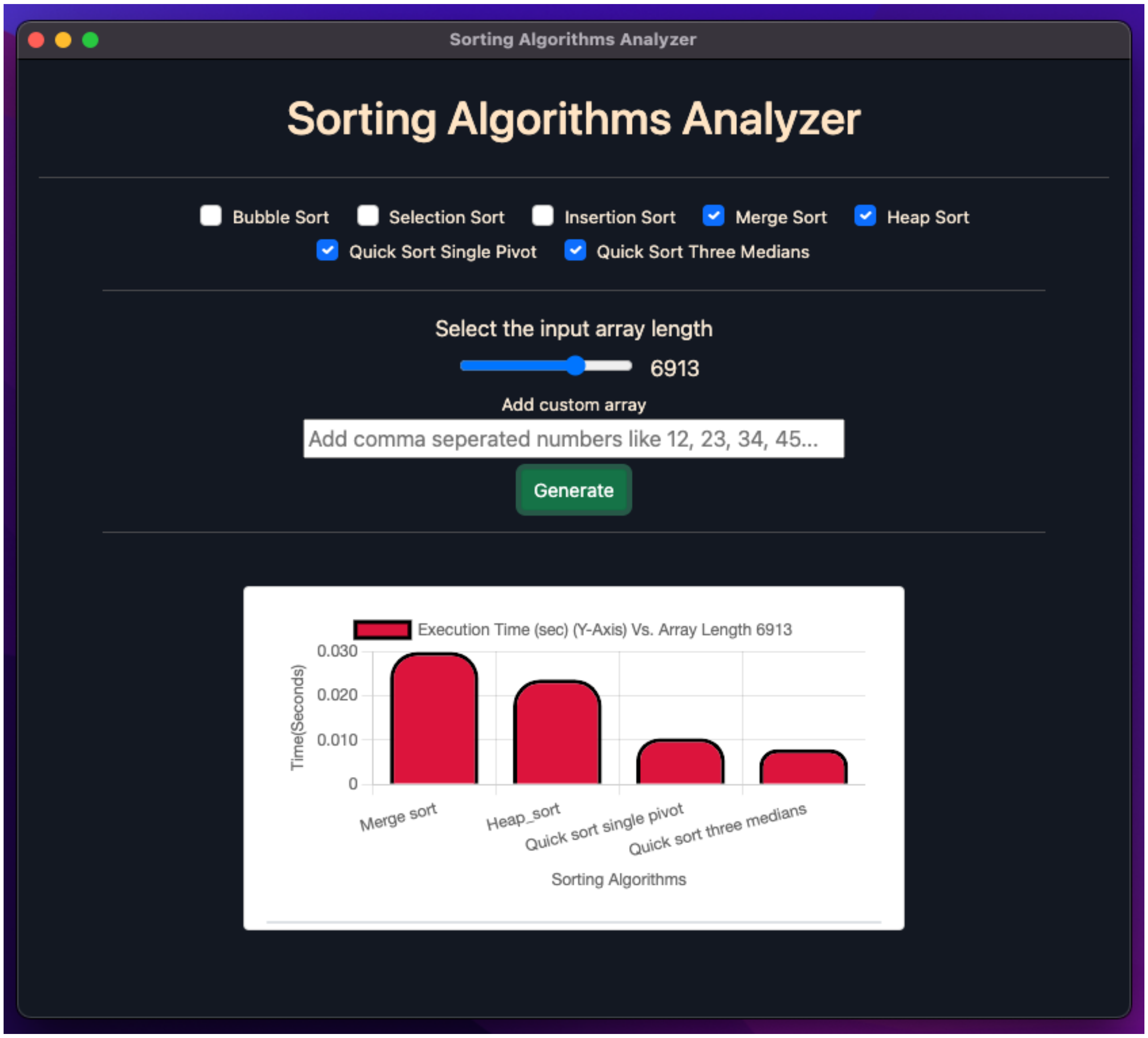
Select the input array length
4838

Add custom array
1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Hey there! Select an algorithm to analyze!

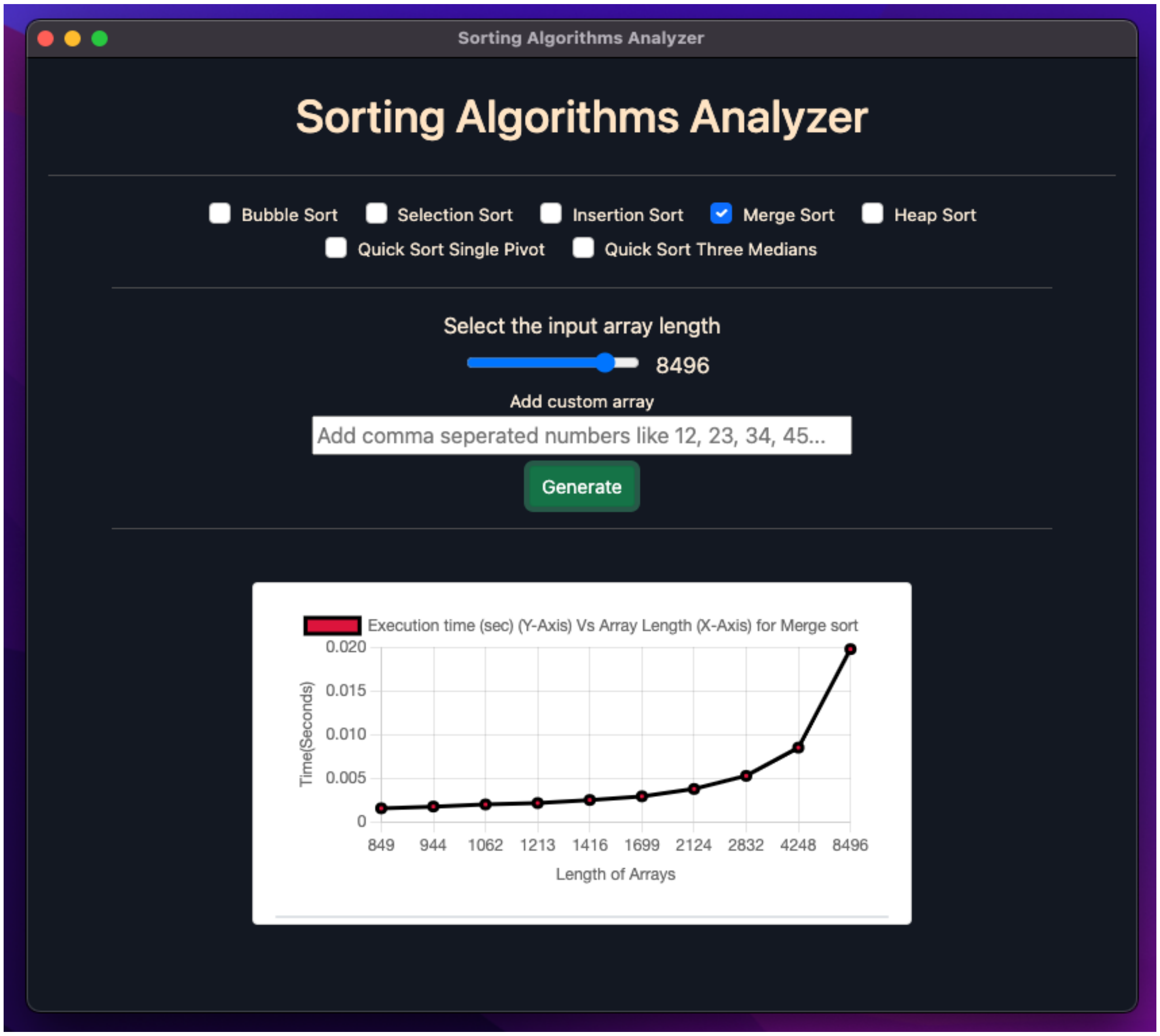
(Figure 7)

If the user selects multiple arrays 'Bar graph' is generated with a comparison of execution times of the selected algorithms for the user input array or an array of length specified by the user.


Sorting Algorithms Analyzer
Sorting Algorithms Analyzer
Bubble Sort
Selection Sort
Insertion Sort
✓ Merge Sort
✓ Heap Sort
✓ Quick Sort Single Pivot
✓ Quick Sort Three Medians
Select the input array length
6913
Add custom array
Add comma seperated numbers like 12, 23, 34, 45...
Generate
🟥 Execution Time (sec) (Y-Axis) Vs. Array Length 6913
0.030
0.020
0.010
0
Time(Seconds)
Merge sort
Heap_sort
Quick sort single pivot
Quick sort three medians
Sorting Algorithms

(Figure 8)

Similarly, when the user selects a single sorting algorithm, a 'Line Graph' is generated by the program over a range of arrays of different length with maximum length being the one specified by the user.

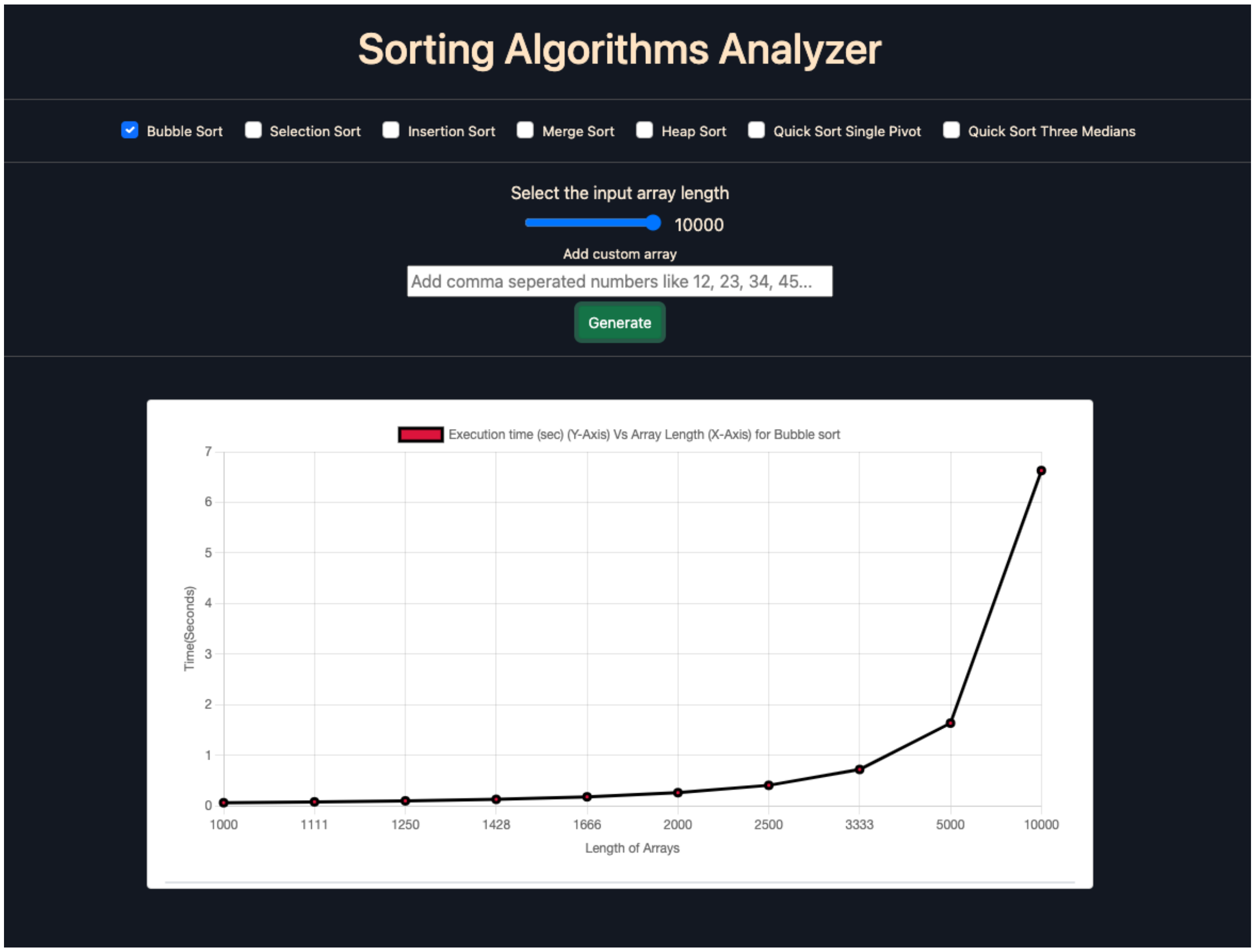


(Figure 9)

Experimental Results

For every algorithm, randomly generated NumPy arrays are used which help us understand in depth the time complexity of each of the following sorting algorithms.

1) Bubble Sort:
   The algorithm has a time complexity of O(n^2) which means every element in the array is visited twice during the execution of function. This means the graph should be exponential.



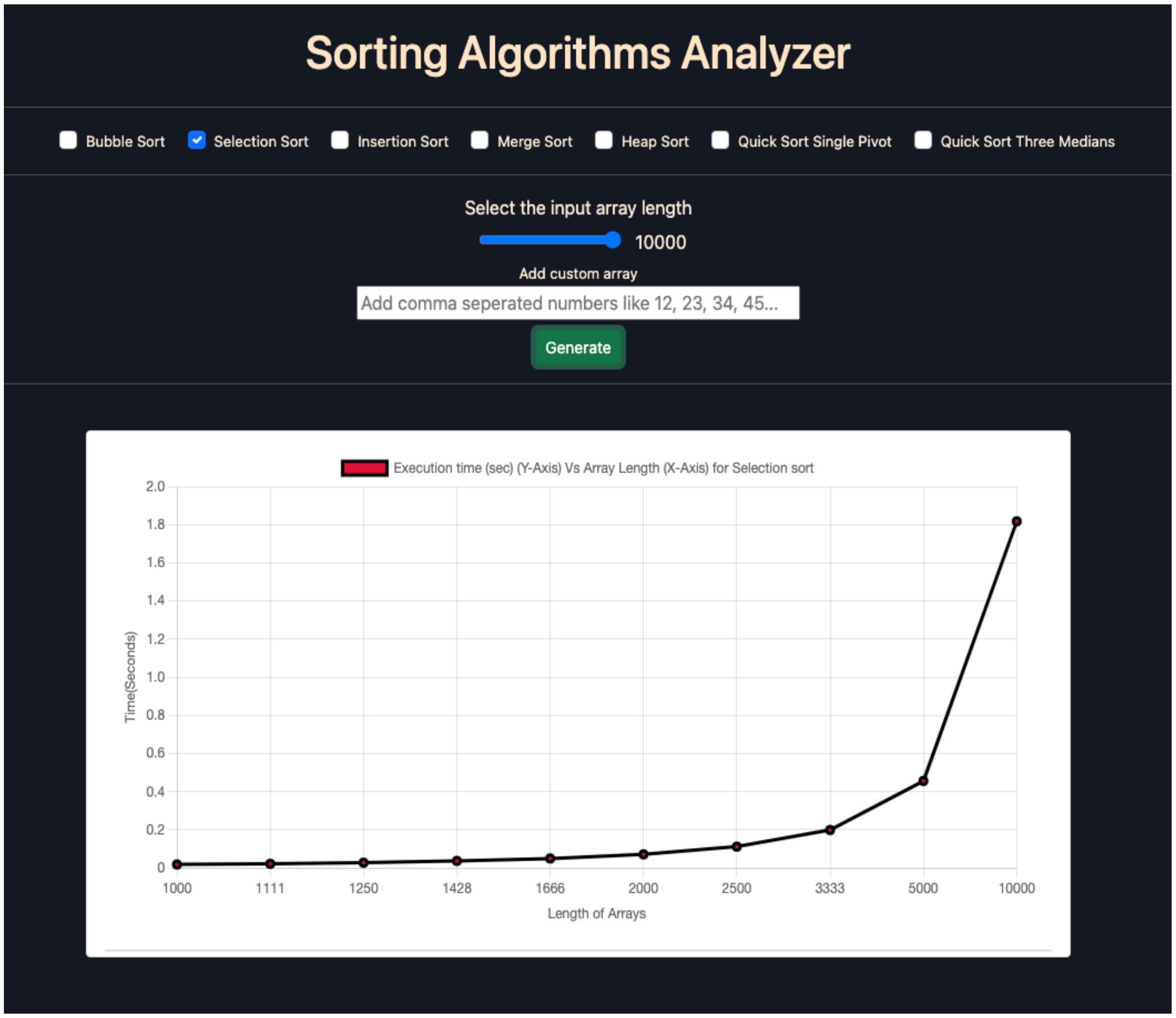
Sorting Algorithms Analyzer

✓ Bubble Sort ☐ Selection Sort ☐ Insertion Sort ☐ Merge Sort ☐ Heap Sort ☐ Quick Sort Single Pivot ☐ Quick Sort Three Medians

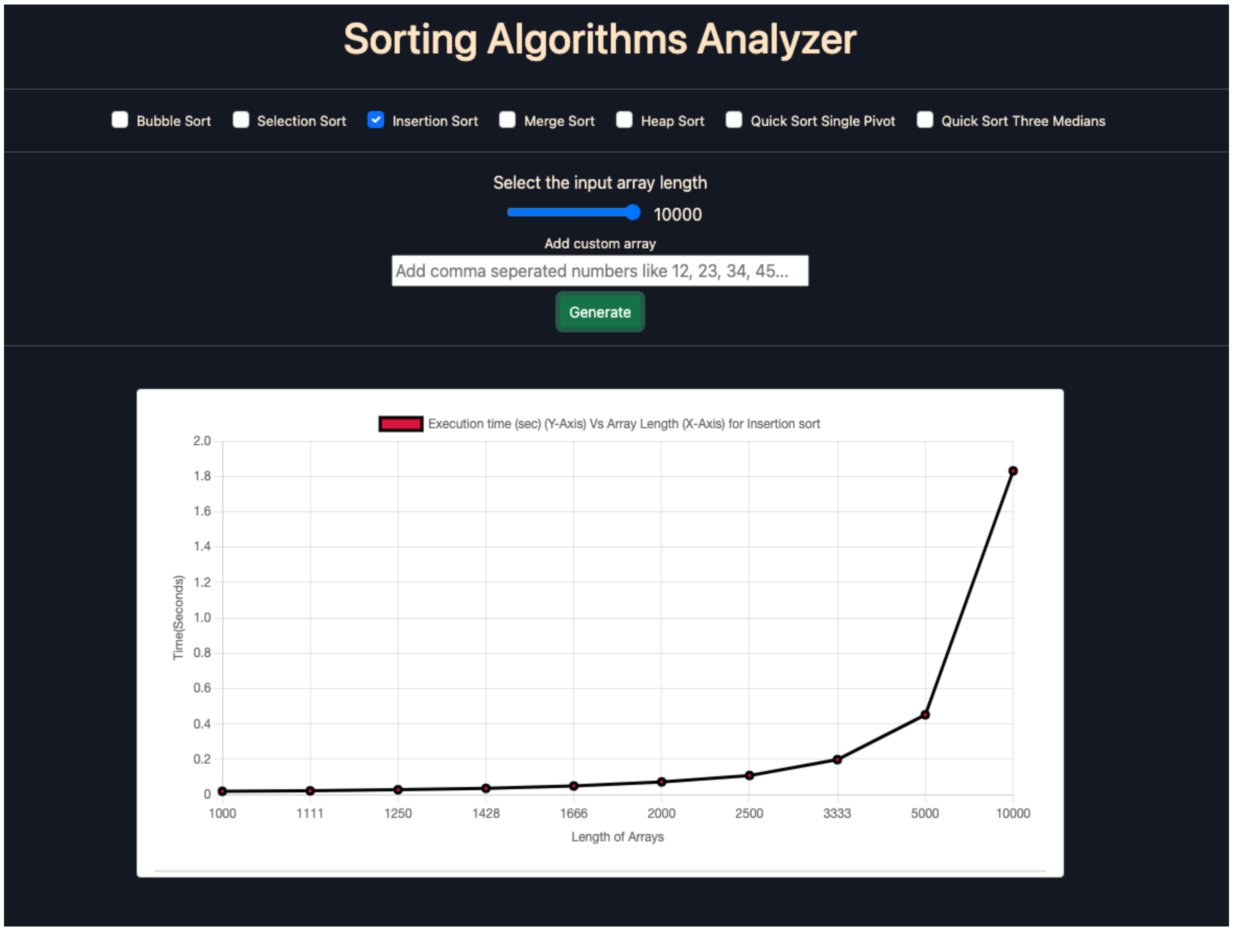
Select the input array length: 10000

Add custom array: Add comma seperated numbers like 12, 23, 34, 45...

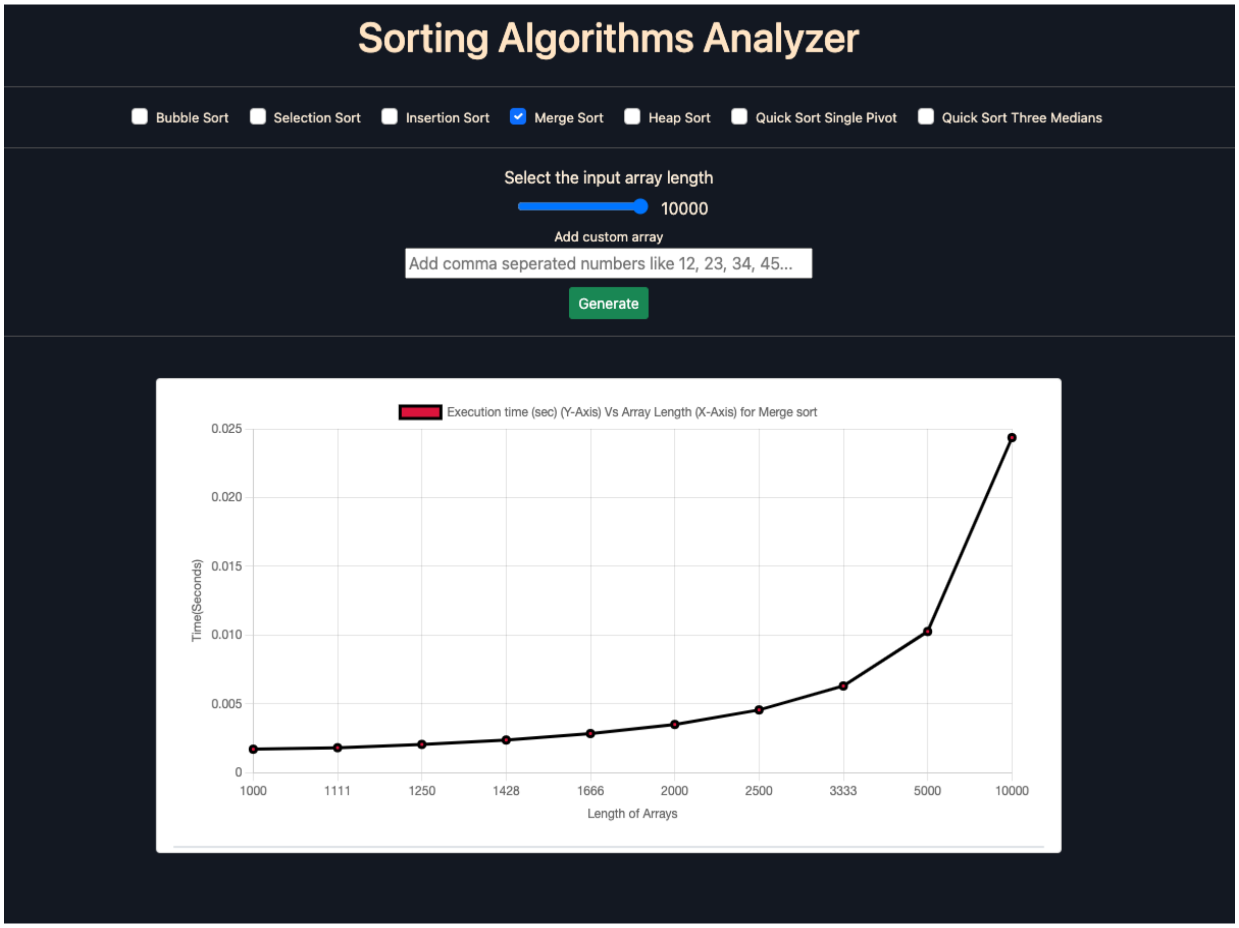
2) Selection Sort:
The algorithm has a complexity of O(n^2) which means the elements of an array are visited twice and the plotted graph of execution time should be exponential.


Sorting Algorithms Analyzer
☐ Bubble Sort
☑ Selection Sort
☐ Insertion Sort
☐ Merge Sort
☐ Heap Sort
☐ Quick Sort Single Pivot
☐ Quick Sort Three Medians
Select the input array length
10000
Add custom array
Add comma seperated numbers like 12, 23, 34, 45...
Generate
🟥 Execution time (sec) (Y-Axis) Vs Array Length (X-Axis) for Selection sort
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0
Time(Seconds)
1000
1111
1250
1428
1666
2000
2500
3333
5000
10000
Length of Arrays

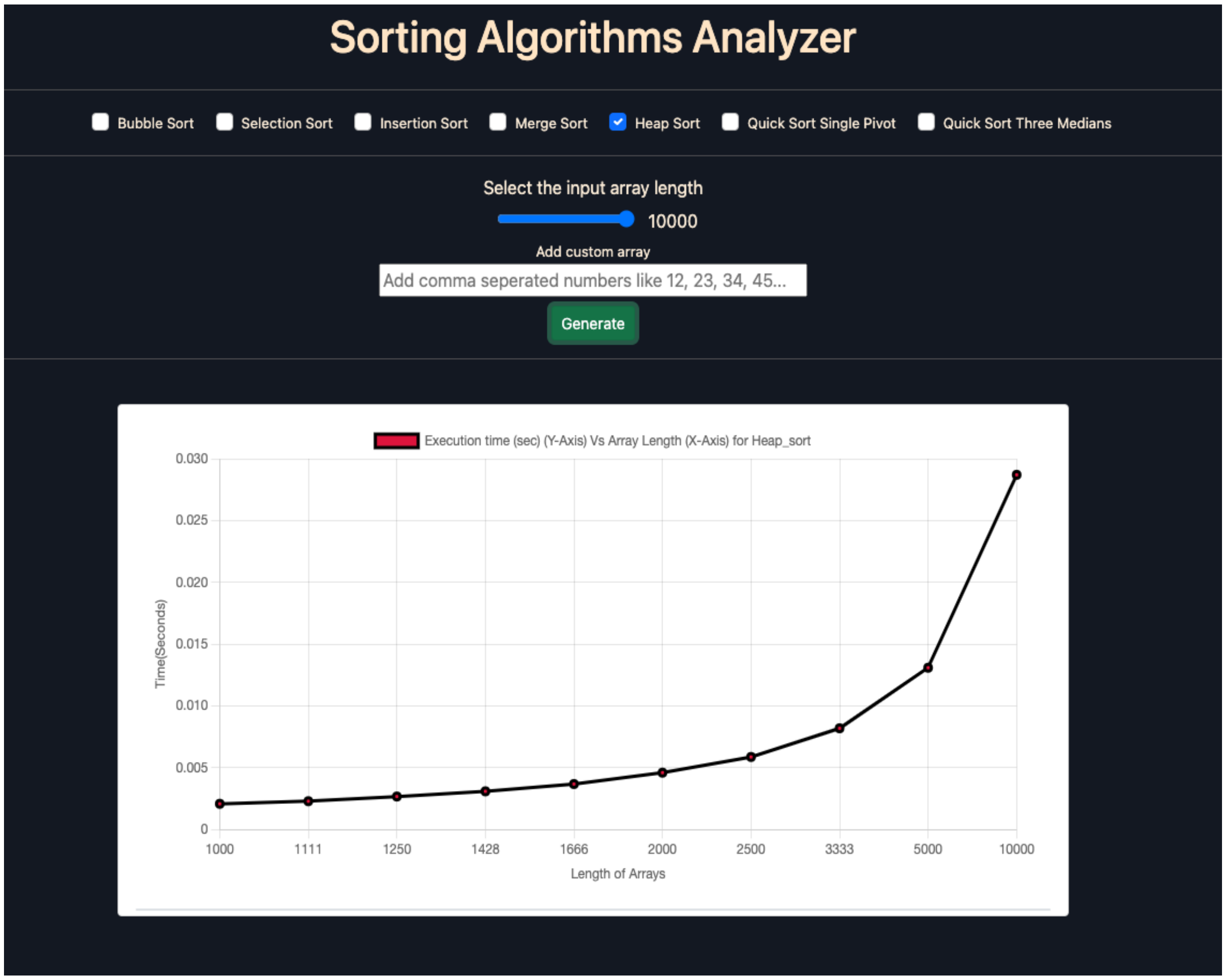
3) Insertion Sort:
The insertion sort algorithm is one of the fastest algorithms for a sorted array or an array with a short length. Although, the time complexity of the algorithm is O(n^2) the graph of the insertion sort is exponential. The algorithm shows relatively quick runtimes for smaller arrays and increases rapidly for greater length arrays.



4) Merge Sort:
   Merge sort is a divide and rule algorithm which has the worst-case time complexity of O(nlog(n)). This means the algorithm has considerably low runtime even when the array length increases a lot.



5) Heap Sort:
Heap sort is an algorithm which builds heap from the input array and then sorts the elements accordingly. This algorithm has a time complexity of O(nlog(n)).



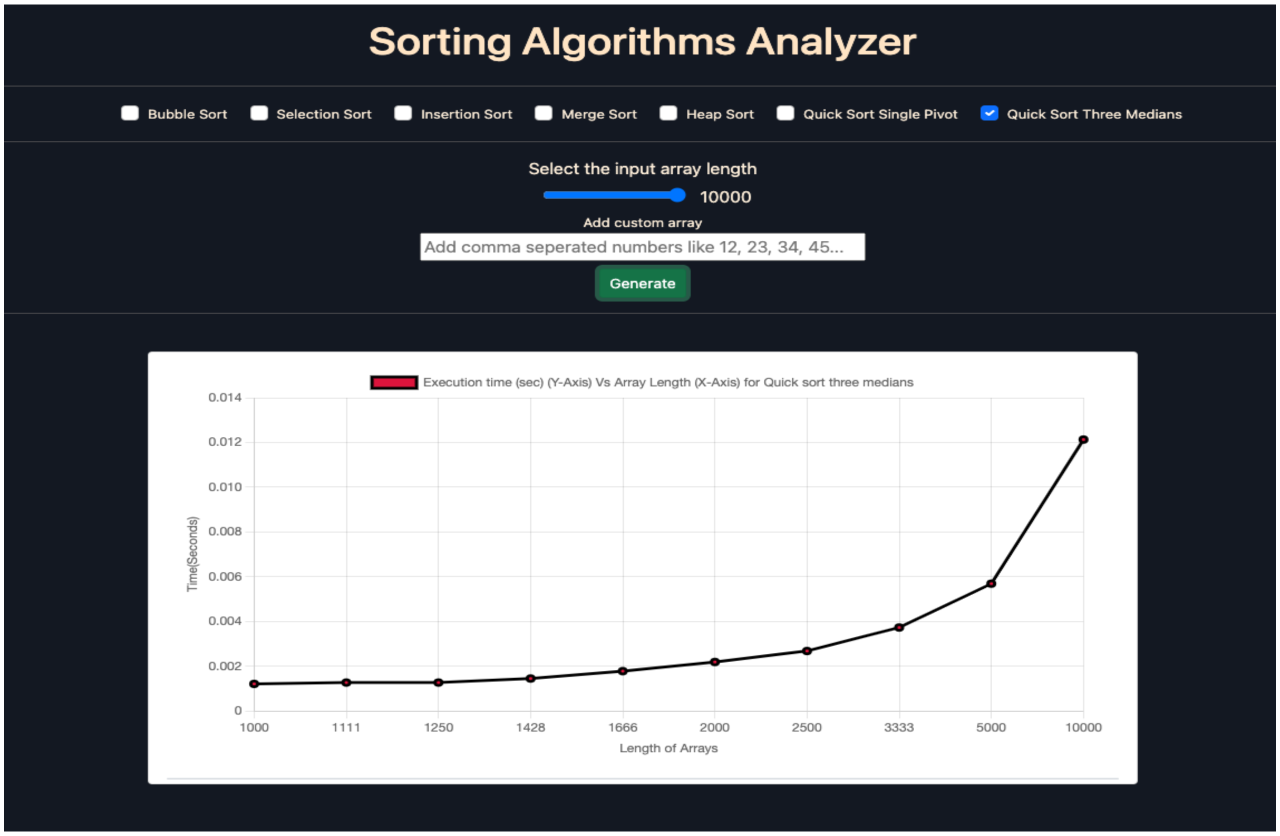
Heap Sort ✓

6) Quick Sort

The file includes two types of quick sort

- Regular Quick Sort which selects a pivot from the array and partitions the array according with respect to the pivot
- Quick Sort Median of Three selects the median among the first, last and middle elements of the array as the pivot and then partitions the array around it.

The time complexity for both the algorithms is O(nlog(n)).



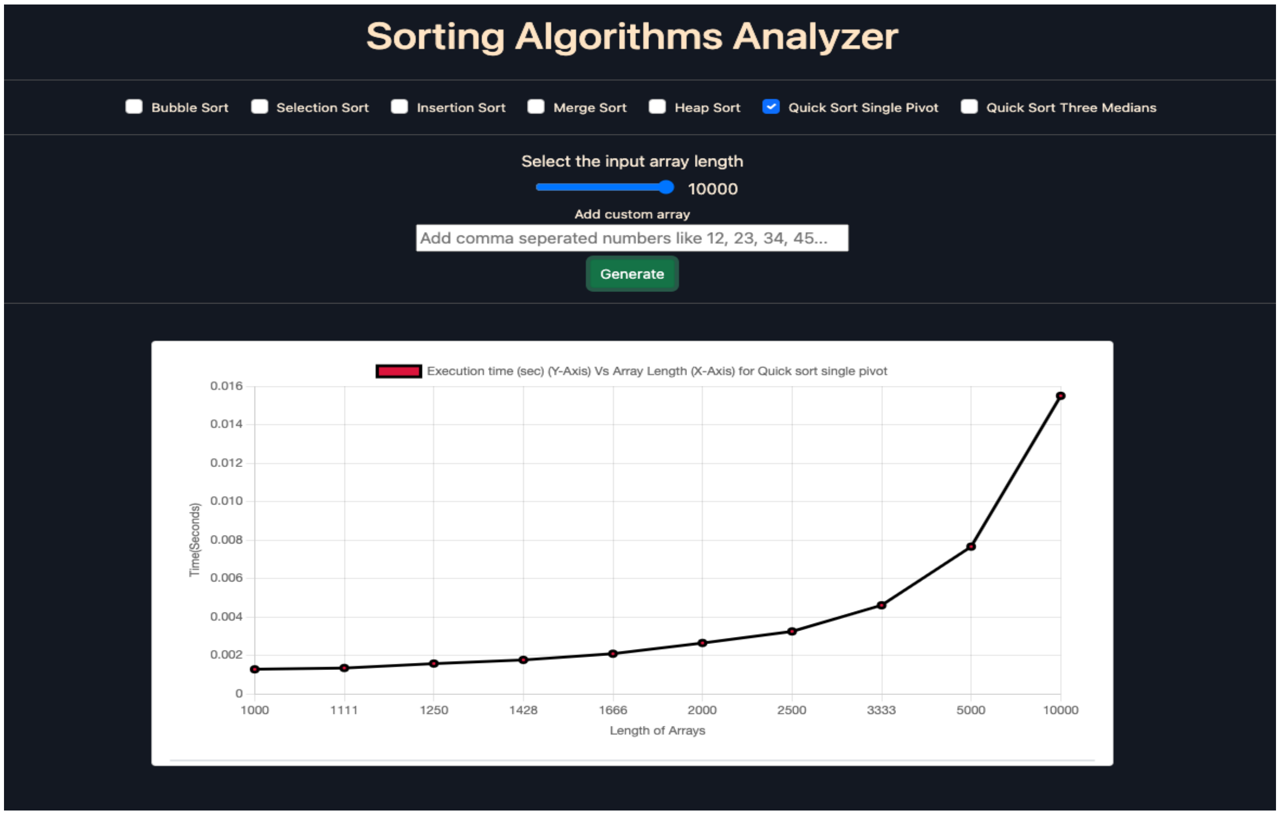
Sorting Algorithms Analyzer

☐ Bubble Sort ☐ Selection Sort ☐ Insertion Sort ☐ Merge Sort ☐ Heap Sort ☐ Quick Sort Single Pivot ✓ Quick Sort Three Medians

Select the input array length 10000

Add custom array

Add comma seperated numbers like 12, 23, 34, 45...



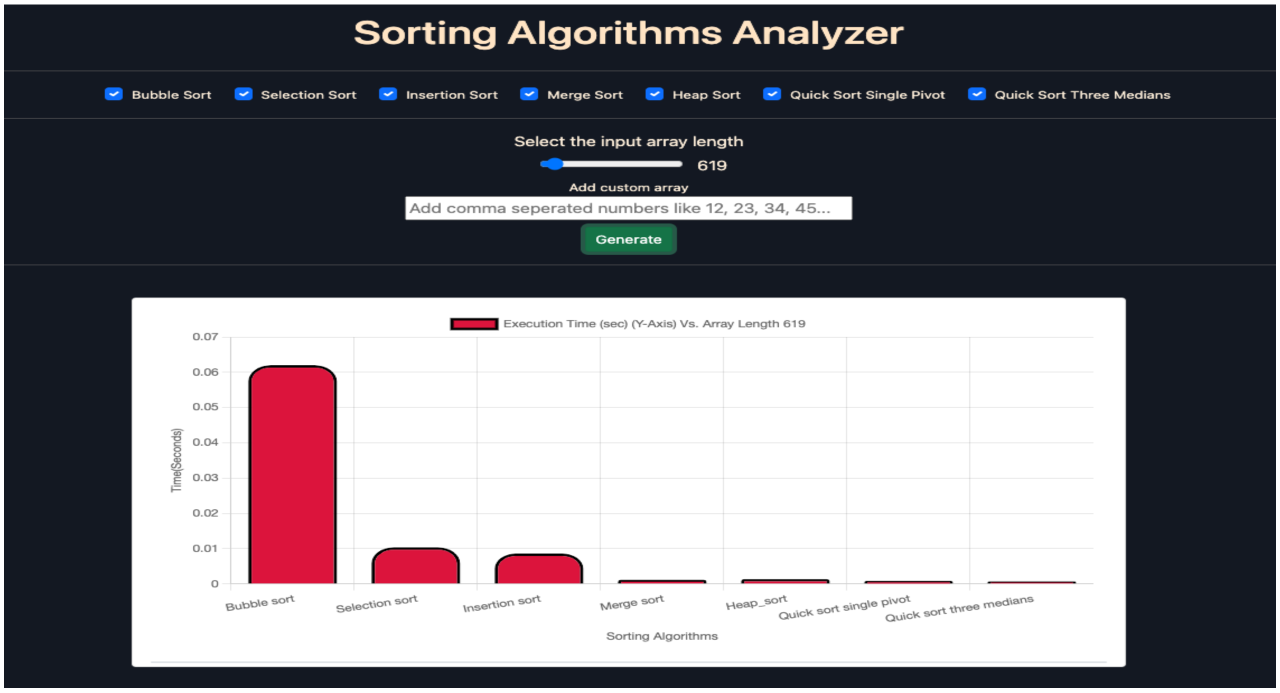
Sorting Algorithms Analyzer

☐ Bubble Sort ☐ Selection Sort ☐ Insertion Sort ☐ Merge Sort ☐ Heap Sort ☑ Quick Sort Single Pivot ☐ Quick Sort Three Medians

Select the input array length: 10000

Add custom array: Add comma seperated numbers like 12, 23, 34, 45...

When all the arrays are compared together the GUI generates a bar graph comparing all the algorithms with the same array.



Sorting Algorithms Analyzer

☑ Bubble Sort ☑ Selection Sort ☑ Insertion Sort ☑ Merge Sort ☑ Heap Sort ☑ Quick Sort Single Pivot ☑ Quick Sort Three Medians

Select the input array length: 619

Add custom array: Add comma seperated numbers like 12, 23, 34, 45...

Conclusions

The time complexities of the algorithms are very important to manage the execution cost to the program. Faster the algorithm better the management of the array during the execution. The GUI plots the graph exactly to demonstrate the need to implement time efficient algorithms.