ALGORITHM EFFICIENCY ANALYZER TOOL

CPSC 535: Advanced Algorithms (Fall 2023)

Project 1

GROUP 3 - Algo Wizards

Team Members:

Anthony Martinez

James Kim

Anshika Khandelwal

Pravallika Bahadur

Tejashwa Tiwari

Rishitha Bathini

Param Venkat Vivek Kesireddy

Roles:

GUI Developer - Anthony Martinez

Algorithm Specialist - James kim

Data Visualization Experts - Anshika Khandelwal & Pravallika Bahadur

Testing & Documentation Leads - Tejashwa Tiwari & Rishitha Bathini

Project Coordinator - Param Venkat Vivek Kesireddy

Date of Submission - 09/16/2023

TABLE OF CONTENTS

1. ABSTRACT

The "Algorithm Efficiency Analyzer Tool" is a web-application designed to help users with a comprehensive understanding of sorting algorithms' performance based on different metrics such as time and memory. Sorting is a fundamental operation when writing algorithms, and the tool focuses on eight prominent sorting techniques, including Bubble Sort, Selection Sort, Merge Sort, Counting Sort, Heap Sort, Insertion Sort, Quick Sort, and Radix Sort.

This tool addresses the need for a practical solution to analyse and compare the efficiency of various sorting algorithms. Users can input a list of numbers of their choice and get a bar-graph with the time taken by every sorting algorithm to sort the given input.

In summary, this project aims to provide a practical and user-friendly solution for algorithm efficiency analysis, ultimately contributing to better-informed algorithm selection and enhanced problem-solving capabilities.

2. INTRODUCTION

A Sorting algorithm is used to arrange elements in a specific order. There are various sorting algorithms that can be used to sort a given range of elements in a list/array. The Algorithms Efficiency Analyzer Tool is an application which provides the user a way to enter a range of elements as an input and the tool sorts the numbers using the different sorting algorithms and outputs the time taken by every sorting algorithm to execute the sorting.

There are a couple sorting techniques we will be looking at through this tool to understand what sorting techniques are best for which scenarios. This tool will also give a deeper result-based efficiency output which will explain the complexity of these algorithms.

The different sorting algorithms are -

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
6. Counting Sort
7. Heap Sort

The efficiency of any sorting algorithm is determined by the time complexity and space complexity. Analysing the efficiency of algorithms, considering both time and space

complexity, is crucial for several reasons. It helps us select faster algorithms, optimise memory usage, and identify bottlenecks in software.

In today's fast-paced digital world, efficient algorithms are vital for responsive applications, cost-effective cloud computing, and scalable solutions, making efficiency analysis a cornerstone of modern computing.

Sorting Algorithms often reduce the complexity of a problem – by having direct applications in searching algorithms, database algorithms, divide and conquer methods, data structure algorithms, and many more it is important for us to have a deeper understanding of when we should use sorting algorithms.

3. ROLES AND RESPONSIBILITIES

- GUI Developer
    - The graphical user interface (GUI) was created and implemented by **Anthony Martinez** in his capacity as the GUI Developer. He was responsible for creating the GUI's layout and design to make sure it was simple to use and understand.
    - He added interactive components that let users choose sorting methods and input data. Anthony guaranteed responsiveness, feedback, and easy user engagement.
    - Additionally, he worked closely with Anshika and Pravallika to manage faulty inputs and seamlessly incorporate visualisations into the GUI in order to improve user experience.
- Algorithm Specialist
    - **James Kim** was entrusted with developing and refining the tool's sorting algorithms as the only algorithm specialist. His obligations included:
    - putting into practice the stated sorting algorithms, which included Quicksort, Linear Sorting, Medians & Order Statistics, and Heapsort.
    - enhancing the algorithms to ensure their accuracy and excellent performance while taking time and space complexity into account.
    - testing the built algorithms thoroughly to confirm their accuracy and effectiveness.
    - the development of user-friendly interfaces for choosing and using sorting algorithms in close collaboration with the GUI developer.

- Data Visualization Expert
    - As Data Visualization Experts, **Anshika Khandelwal** and **Pravallika Bahadur** were in charge of producing visual representations of algorithm effectiveness.
    - They had to use the matplotlib software to create visual aids that showed each sorting algorithm's effectiveness, including bar graphs or scatter plots.
    - To effortlessly include these visualisations into the application's user interface, they worked closely with the GUI Developer.
- Testing & Documentation Lead
    - **Tejashwa Tiwari and Rishitha Bathini** were responsible for ensuring the reliability and documentation of the project in their role as Testing & Documentation Leads.
    - Their responsibilities included developing a comprehensive test suite to validate the functionality of the tool under various scenarios and with different input sizes and datasets.
    - They also documented the codebase, including code comments, explanations, and best practices. Furthermore, they created clear and detailed documentation of the testing processes that were followed, including test cases, results, and any anomalies encountered.
    - Their role also involved overseeing the quality assurance process to identify and address any glitches or bugs.
- Project Coordinator
    - As the project coordinator, **Param Venkat Vivek Kesireddy** was in charge of managing the team's schedule and overseeing the project's development.
    - Using Git, Venkat tracked code contributions, provided efficient team coordination to promote smooth communication and collaboration throughout the project, and dealt with any conflicts that emerged during development.
    - He also facilitated discussions, dealt with any problems or obstacles that arose throughout the project, and kept the project's overall direction and progress on track.
    - He also created and maintained project schedules to ensure that tasks were appropriately assigned to team members and completed on time through Jira and Github Repository.

4. METHODOLOGY

1. **Project Overview:** This methodology outlines the approach, tools, and technologies we used to build a tool for analyzing the efficiency of different sorting algorithms. The tool provides insights into the time taken by each algorithm to sort an array.
2. **Objective:** The primary objective of this project is to create a web-based application that allows users to select various sorting algorithms and input arrays, then visualize the time it takes for each algorithm to sort the given array. The tool is built using a combination of React.js for the front-end and Flask for the back-end.
3. **Front-End Development:**
    a. Framework: The front-end of the application is developed using React.js.
    b. Design Library: Material UI, is used to design the user interface components for a modern and intuitive user experience.
    c. Components: Create React components for user input (array selection, algorithm selection), visualization of sorting process (using ChartJS), and display of sorting results.
    d. State Management: Utilize React's state management to handle user interactions and data flow.
4. **Back-End Development:**
    a. Framework: Flask, a micro web framework for Python, is used to build the back-end server.
    b. Hosting: The Flask server is hosted on PythonAnywhere, ensuring the application is accessible from anywhere.
    c. Static Build: The React.js application is built into a static build, and this build is served by Flask when users access the root URL ('/').
5. **Sorting Algorithms:**
    a. Implement various sorting algorithms like Bubble Sort, Quick Sort, Merge Sort, etc., as separate functions or classes in Python.
    b. Ensure each sorting algorithm accepts unsorted arrays and returns sorted arrays and accepts edge cases.
    c. Implement timing mechanisms to record the execution time of each sorting algorithm.
6. **API Endpoint:**
    a. Create a Flask route ('/sort') that handles GET requests for sorting.
    b. This endpoint will accept query parameters 'unsorted_arr' (the array to be sorted) and 'algorithm' (the selected sorting algorithm).
    c. Upon receiving a request, the endpoint will call the corresponding sorting algorithm function, record the execution time, and return the sorted array and execution time in the response.
7. **Front-End Interaction:**
    a. Users input an array and select a sorting algorithm through the user interface.
    b. Upon submission, a GET request is made to the '/sort' endpoint with the chosen parameters.

   c. The application displays the sorted array and the time taken for the selected sorting algorithm to complete the task.
8. **Visualization**:
   a. Implement a visual representation of the sorting process to enhance user understanding and engagement.
   b. Use animations or step-by-step updates to show how the elements are rearranged during the sorting.
9. **Testing:**
   a. Conduct thorough testing of both front-end and back-end components.
   b. Ensure the tool functions correctly for various input scenarios and handles errors gracefully.
10. **Deployment**:
   a. Deploy the application on a web server, making it accessible to users.
   b. Ensure proper configurations for the server and domain settings.

By following this methodology, we were able to successfully build a tool that analyzes the efficiency of different sorting algorithms and provides valuable insights into their performance. Users can interact with the tool to visualize and compare the sorting times of various algorithms on different input arrays and the tool can be accessed from anywhere.

5. IMPLEMENTED ALGORITHMS

There are various types of sorting algorithms, they can be categorised based on multiple parameters. Some of the parameters are -

1. Count of Swaps or Inversions Required - Refers to the no. of times an algorithm swaps elements to sort the input.
2. Number of Comparisons - Refers to the number of times an algorithm compares the elements to sort the input. These correlate to the best case and worst case scenarios and are denoted using Big-O notation.
3. Recursions - Depends if an algorithm uses recursion or not. Some like Quick sort use recursion to sort the input while others like selection / insertion sort use non-recursive techniques.
4. Stability of the algorithm - Stable sorting algorithms maintain the relative order of elements with equal values, or keys. Unstable sorting algorithms do not maintain the relative order of elements with equal values / keys.
5. Complexity or Space Required: Some algorithms are efficient and can sort a list without creating a new list which keeps the complexity low O(1) whereas there are out-of-place sorting algorithms which creates a new list while sorting everytime.

Based on the factors listed above, there are several sorting algorithms which can help you sort your list/array. Different algorithms are created because different sorting algorithms are

designed to excel in specific scenarios and under particular conditions.

1. **Bubble Sort:** Bubble sort is a basic sorting algorithm that goes through a list, swapping adjacent values if they're in the wrong order to arrange them from lowest to highest.
    a.

```
function bubbleSort(arr):
  n = length(arr)
  swapped = true

  while swapped is true:
    swapped = false

    for i from 1 to n-1:
      if arr[i-1] > arr[i]:
        swap(arr[i-1], arr[i])
        swapped = true

    n = n - 1
```

2. Pseudo Code: **Quick Sort:** It works by dividing a list into smaller parts and then rearranging them based on whether they're greater or smaller than a chosen "pivot" element.
    a. Pseudo Code:

```
function quicksort(arr):
  if length(arr) <=1;
    return arr
  pivot = selectPivot(arr)
  left = elements in arr less than pivot
  right = elements in arr greater than pivot

  left = quicksort(left)
  right = quicksort(right)

  return concatenate(left, pivot, right)
```

3. **Insertion Sort:** Insertion sort works by picking one element at a time and inserting it into its correct position among the already sorted elements, gradually building a sorted list.
    a. Pseudo Code:

```
function insertionSort(arr):
    for i from 1 to length(arr) - 1:
        key = arr[i]
        j = i - 1

        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1

        arr[j + 1] = key
```

4. **Merge Sort:** Merge sort divides the unsorted list into smaller halves, sorts each half, and then merges them back together in sorted order. It repeats this process until the entire list is sorted.
    a. Pseudo Code:

```
function mergeSort(arr):
    if length(arr) <= 1:
        return arr

    mid = length(arr) / 2
    left = first half of arr
    right = second half of arr

    left = mergeSort(left)
    right = mergeSort(right)

    return merge(left, right)

function merge(left, right):
    result = empty array

    while left is not empty and right is not empty:
        if left[0] <= right[0]:
            append left[0] to result
            remove the first element from left
        else:
            append right[0] to result
            remove the first element from right

    append remaining elements of left to result
    append remaining elements of right to result
```

```
return result
```

5. **Selection Sort:** Selection sort works by repeatedly selecting the smallest (or largest) element from the unsorted part of the list and moving it to the beginning (or end) of the sorted part, incrementally building a sorted list.
    a. Pseudo Code:

```
function selectionSort(arr):
  n = length(arr)

  for i from 0 to n-1:
    minIndex = i

    for j from i+1 to n-1:
      if arr[j] < arr[minIndex]:
        minIndex = j

    if minIndex ≠ i:
      swap arr[i] with arr[minIndex]
```

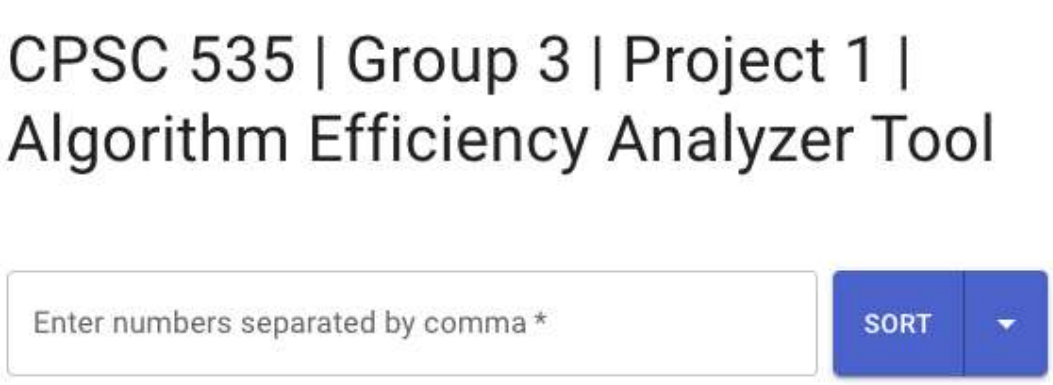
6. **Count Sort:** Counting sort works by counting the frequency of each element in the input list, creating a count array. It then constructs a sorted output array by mapping each element to its correct position based on its count and order in the original list.
    a. Pseudo Code:

```
function countSort(arr, max):
  count = array of size (max + 1) initialised with zeros
  output = empty array

  for element in arr:
    count[element] += 1

  for i from 0 to max:
    while count[i] > 0:
      append i to output
      count[i] -= 1

  return output
```

7. **Heap Sort**: Heap sort works by first transforming the input array into a max-heap, where the largest element is at the root. It repeatedly removes the root element (the largest) and restores the heap property, resulting in a sorted list.
    a. Pseudo Code:

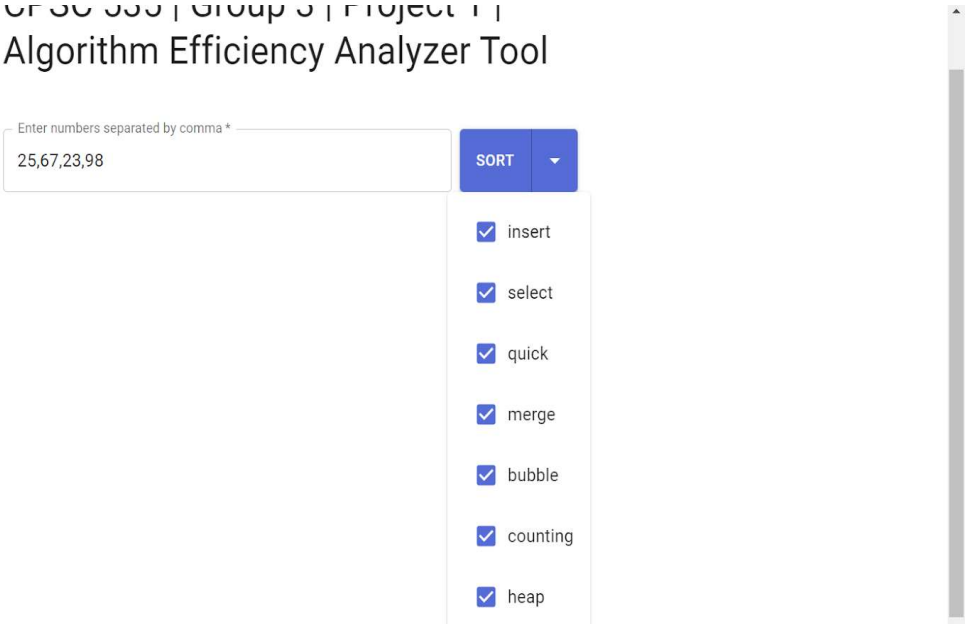
```
function heapSort(arr):
    n = length(arr)
    for i from (n / 2) down to 0:
        heapify(arr, n, i)

    for i from n - 1 down to 0:
        swap arr[0] with arr[i]
        heapify(arr, i, 0)

function heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest ≠ i:
        swap arr[i] with arr[largest]
        heapify(arr, n, largest)
```

6. GUI DESIGN

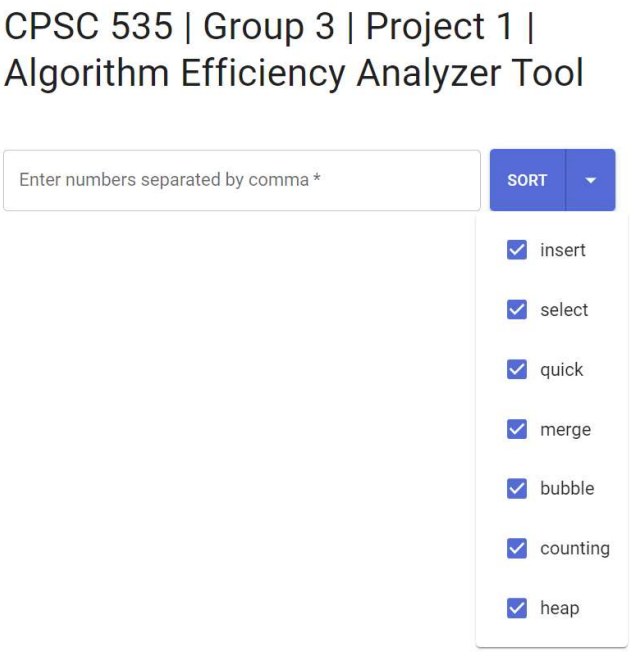


This graphical user interface (GUI) serves as the user interface for our application. To initiate the sorting process, the first step is to input the numbers or data elements that you wish to sort. Once these values have been provided, our system will proceed to generate a sorted

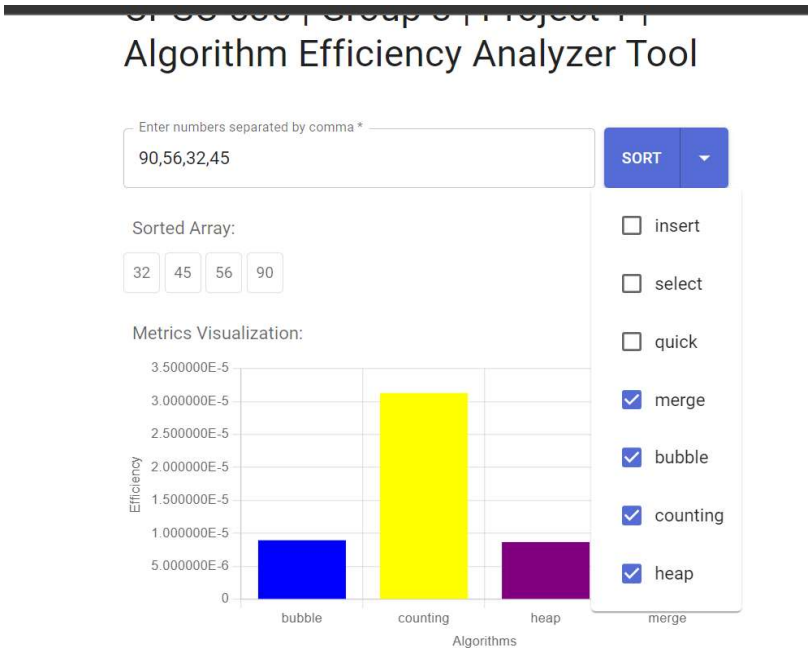
array based on the input.



After entering the numbers and initiating the sorting process, you can observe the resulting sorted array displayed on the screen. This provides a visual representation of the data in its sorted order.

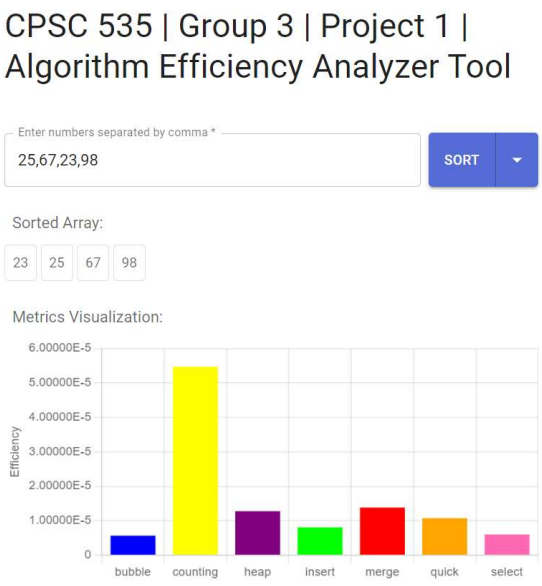


A list of integers can be entered, and the user can choose a method to sort the array. The array is then sorted, and the results are shown in a console window .



We can select which sorting algorithms we have to use based on specific input data.

7. VISUALISATION

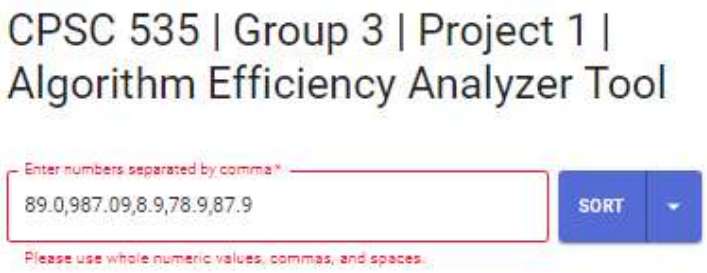


The bar graph shows the average time taken by six different sorting algorithms to sort a set of numbers of increasing size. The algorithms are bubble, counting, heap, merge, quick, and

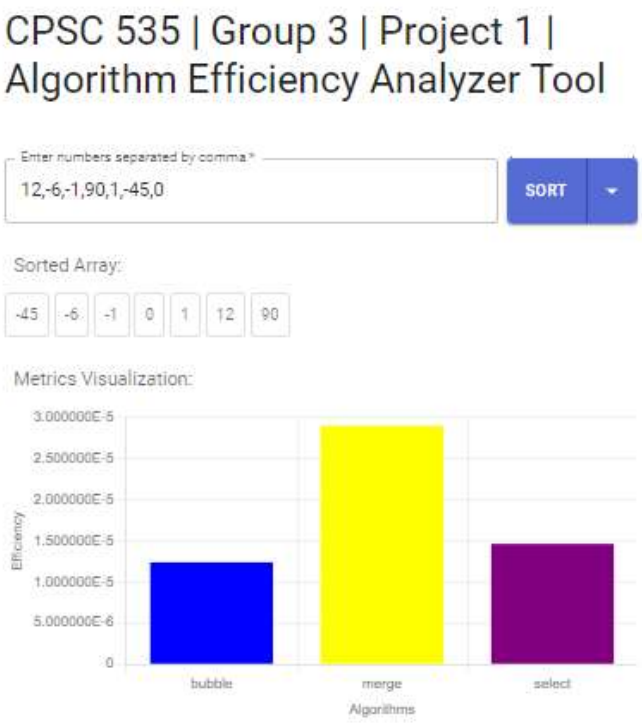
select. The x-axis represents the size of the input set (in number of elements) and the y-axis represents the time taken in seconds.

8. TESTING

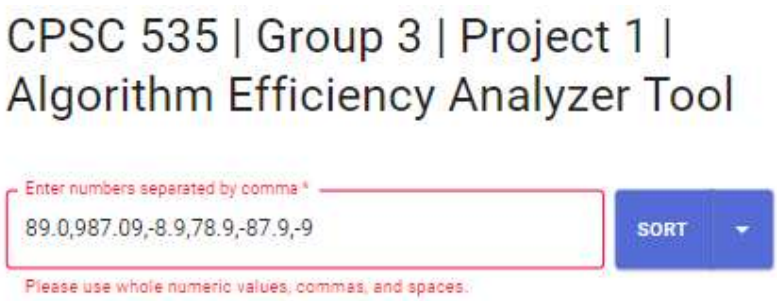
- During the manual testing process, it was observed that the system encounters difficulties when handling float data types, leading to an error message stating, "Please use whole numeric values, commas, and spaces." This indicates that the system is not designed to support float types and expects inputs to be composed of integers, commas, and spaces.



- Negative numbers are sorted correctly, and the algorithm produces accurate output

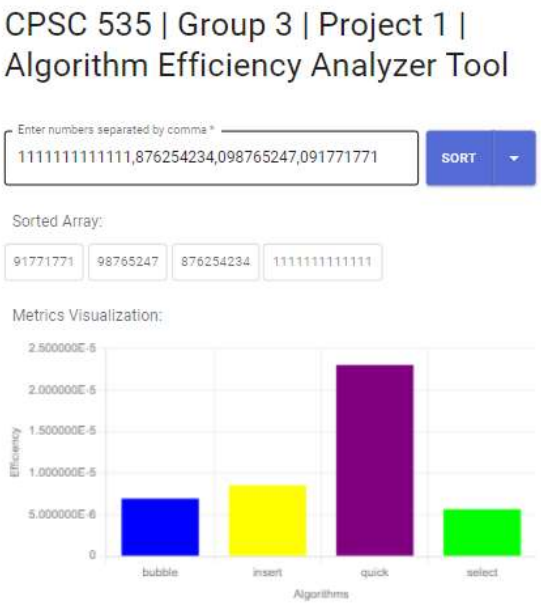


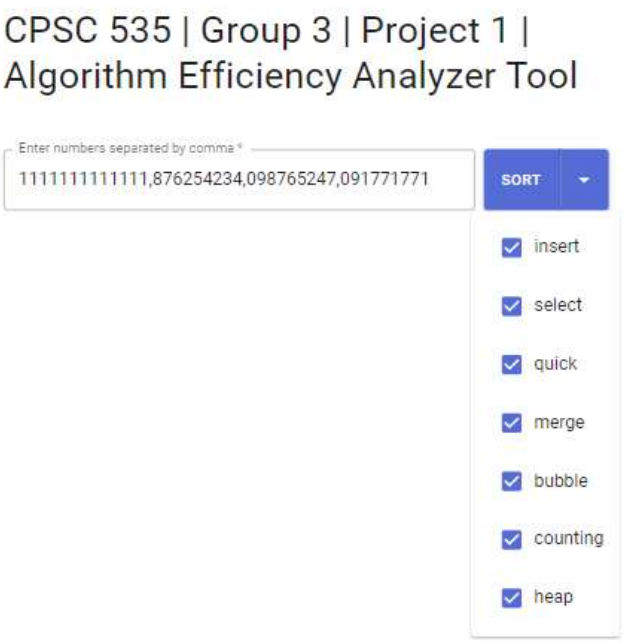
- It was observed that negative numbers and decimals are causing an error, but the algorithm still proceeds to sort the array and generate the output.



- For testing purposes, we focused on evaluating the performance of the algorithm when sorting arrays containing large numbers, specifically those with up to 8 digits. Our observations indicated that as we increased the number of digits within a single array element, the execution time also increased significantly.
- Additionally, it's important to highlight that when the array contained multiple elements with long integers, either the sorting process took an extended amount of time or, in some cases, caused the application to become unresponsive.
- It was noted that while the system executed one specific sorting method flawlessly, the performance of all other sorting methods, excluding counting sort, remained consistently effective when dealing with numbers containing a larger number of digits.

- Furthermore, it was observed that while the system executed one type of sort perfectly, as we kept selecting different sorting methods, the execution time increased significantly, and the system occasionally became unresponsive.





9. CHALLENGES FACED

Algorithm optimization and guaranteeing seamless GUI and data visualisation component integration presented difficulties for our team. Iterative development and strong cooperation allowed us to overcome these challenges.

- It was difficult to design a user-friendly and intuitive GUI. In order for users to submit data, choose algorithms, and examine results without ending up confused, we had to build a layout that is simple to browse and comprehend.
- It was difficult to produce efficient representations of the method, such as graphs or charts. We must choose the right visualisation packages and create powerful graphical displays of the data.
- Making a product that users find beneficial and simple to use requires doing user testing to get input and make adjustments depending on user preferences and pain areas.
- It can be difficult to design the tool such that it can efficiently handle both small and large datasets since sorting algorithm performance might vary greatly depending on the size of the input.
- It is essential to guarantee that the implemented sorting algorithms function properly. To detect and correct any algorithmic mistakes, we need thorough testing and debugging processes.
- To keep track of activities, problems, and progress, we used project management tools such as Jira and other tools to keep the team organised.

10. CONCLUSION

In conclusion, our team's creation of the Algorithm Efficiency Analyzer Tool, which enables users to research and assess the effectiveness of various sorting algorithms, was a success.

This project improved our comprehension of data visualisation, GUI design, and algorithm efficiency. This effort advances our knowledge of sorting algorithms while simultaneously serving as an important instructional tool that makes difficult ideas understandable to a wider audience. The primary objective of this project was to develop a web-based tool that empowers users to explore the performance characteristics of various sorting techniques.

Throughout the project's lifecycle, we employed a comprehensive approach that incorporated the utilization of modern technologies and best practices. With prospective upgrades including new sorting algorithms, customization choices, complicated data structure analysis, real-time benchmarking, instructional materials, and cross-platform compatibility, the Algorithm Efficiency Analyzer Tool has a promising future.

11. APPENDICES

A. Sorting Algorithms Used
    a. Bubble Sort
    b. Selection Sort
    c. Insertion Sort
    d. Merge Sort
    e. Quick Sort
    f. Heap Sort
    g. Counting Sort
B. Tools and Technologies Used
    a. Front-End:
        i. React.js
        ii. Material UI Design Library (MUI)
        iii. ChartJS
    b. Back-End:
        i. Flask (Python Web Framework)
        ii. PythonAnywhere (Hosting Service)
    c. Development Tools:
        i. Code Editor (e.g., Visual Studio Code, PyCharm)
        ii. Git for Version Control
        iii. GitHub for Collaboration and Code Repository
    d. Management
        i. Jira for Project Management