Algorithm Efficiency Analyzer Tool

Team Members:

- Neema Tabarani (Algorithm Specialist)
- Asad Abdul (Algorithm Specialist)
- Mrunal Patil (GUI Developer)
- Akshay Sakpal (Data Visualization Expert)
- Adam Harb (Testing and Documentation Lead)
- Brandon Kilpatrick (Project Coordinator)

Abstract: What is an Algorithm Efficiency Analyyzer Tool?

The Algorithm Efficiency Analyzer Tool is a type of collaborative project that looks to deepen our understanding of algorithm efficiency, GUI Design, and data visualization techniques to help prepare for real world scenarios. The tool allow users to input a list of numbers, selecting a sorting algorithm, and visualize its efficiency through a spontaneous GUI interface

Introduction: What is the objective of the project?

The objective of the project is to provide a practical tool for users to touch on and comprehend the efficiency of numerous sorting algorithms. By implementing Heapsort, Quicksort, Counting Sort, Merge Sort, Insertion Sort, and Bubble Sort, our goal is to offer a hands-on experience in algorithm analysis. The technologies used include 'tkinter' for GUI Development and 'matplotlib' for Data Visualization

Roles and Responsibilities

Neema Tabarani and Asad Abdul (Algorithm Specialist)

- Implemented sorting algorithms: Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Counting Sort, Radix Sort, and QuickSelect Method
- Enhanced and tested the algorithm for correctness

Mrunal Patil (GUI Developer)

- Developed the GUI layout and design
- Guarantee smooth user interaction and feedback
- Integrated visualization produced by the Data Visualization Expert (Ashkay Patil)

Akshay Sakpal (Data Visualization Expert)

- Created visual representation of algorithm efficiency using ‘matplotlib’
- Collaborated with GUI Developer (Mrunal Patil) to integrate visualization into the application

Adam Harb (Testing and Documentation Lead)

- Developed a comprehensive test suite
- Make sure the application run without glitches
- Documented the code and testing processes followed

Brandon Kilpatrick (Project Coordinator)

- Coordinated between team members to ensure smooth integration.
- Handled version control, timelines, and task distribution.

Methodology: What does the team follow?

The team followed an iterative development process, utilizing Git for version control and maintaining regular communication and ensuring a meeting is being held once a week. The workflow involved collaboration between the GUI Developer (Mrunal Patil), Algorithm Specialist (Neem and Asad), and Data Visualization (Akshay Sakpal) to integrate their respective components smoothly.

Implemented Algorithm

Bubble Sort:

- Description: Bubble sort algorithm is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted
- Examples: [5 1 4 2 8] -> [1 5 4 2 8] First compare the first 2 elements 1 and 5. Second, swap 5 and 4 to result in [1 4 5 2 8]. Third, swap 5 and 2 to result in [1 4 2 5 8]. Now, the algorithms are in order and we cannot swap the last 2 elements, so we repeat the process starting from the first 2 elements. 1 and 4 cannot be swapped, but 4 and 2 can result in [1 2 4 5 8]. Now, the array is sorted in corrected orders and the algorithm is complete.
- Pseudocode:

```
function bubbleSort(A, n):
    for i in range(n - 1):
        swapped = False
        for j in range(n - i - 1):
            if A[j] > A[j + 1]:
                # Swap A[j] and A[j + 1]
                temp = A[j]
                A[j] = A[j + 1]
                A[j + 1] = temp
                swapped = True
           # If no two elements were swapped in inner loop, the list is
sorted
        if not swapped:
            break
```

- Time Complexity: Worst case ($O(n^2)$) and best case $O(n)$

Insertion Sort:

- Description: Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on a large list than more advanced algorithms such as quicksort, heapsort, or merge sort.
- Examples: [3 8 5 4 1 9 -2] -> [3 5 8 4 1 9 -2] -> [3 4 5 8 1 9 -2] -> [1 3 4 5 8 9 -2] -> [-2 1 3 4 5 8 9]. Initially, a sorted subset consist of only one first element at index 0. Then for each iteration, insertion sort removes the next element from the unsorted subset, finds the location it belongs within the sorted subset and inserts it there, It repeats until no input elements remain.
- Pseudocode:

```
i ← 1
while i < length(A):
    j ← i
    while j > 0 and A[j-1] > A[j]:
        swap A[j] and A[j-1]
        j ← j - 1
    end while
    i ← i + 1
end while
```

- Time Complexity: Best Case (O(n)), Average Case (O(n^2)), and Worst case(O(n^2))

Merge Sort:

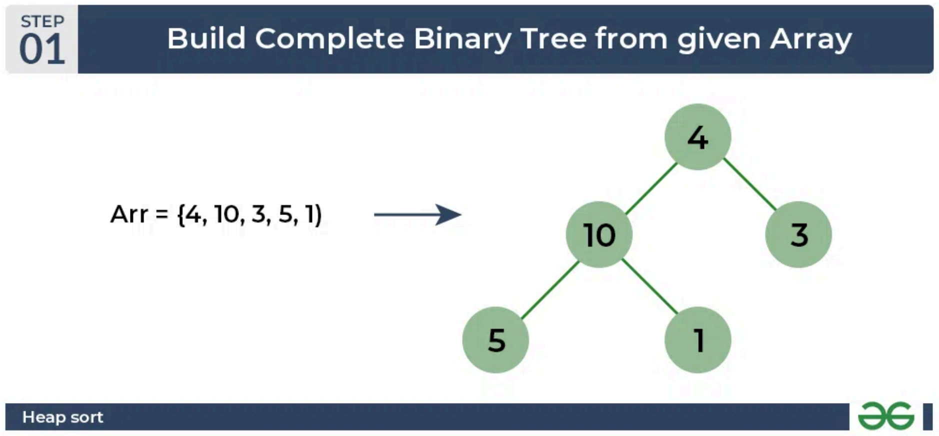
- Description: Merge Sort is a divide and conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.
- Examples: A = [38, 27, 43, 3, 9, 82, 10]. First Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted) [38], [27], [43], [3], [9], [82], [10]. Next, repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list [27, 38], [3, 43], [9, 82], [10]-> [3, 27, 38, 43], [9, 10, 82] -> [3, 9, 10, 27, 38, 43, 82] which is the sorted array.
- Pseudocode:

```
function mergeSort(arr, start, end):
    if start >= end:
        return
    mid = (start + end) / 2
    mergeSort(arr, start, mid)
    mergeSort(arr, mid + 1, end)
    merge(arr, start, mid, end)
```

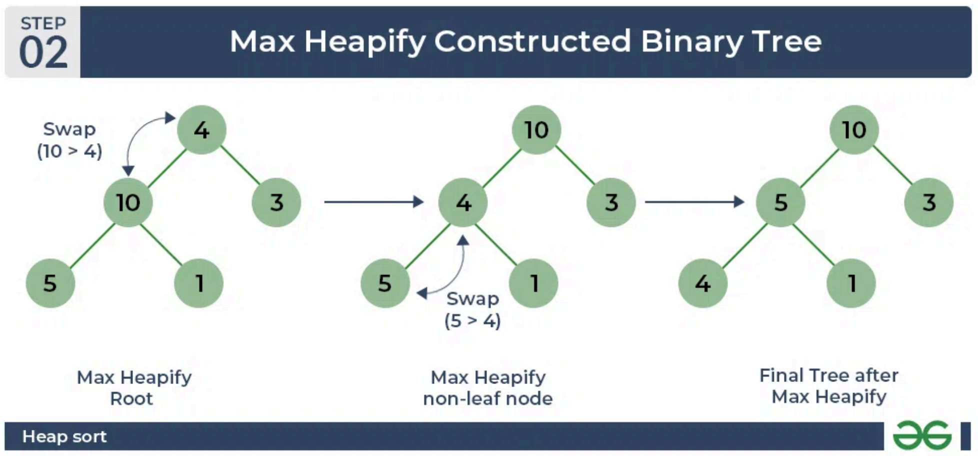
- Time Complexity: Merge sort is fast and has a time complexity of O(nlogn) for the best, worst, and average case as merge sort divides the array in two halves and takes linear time to merge two halves.

Heap Sort:

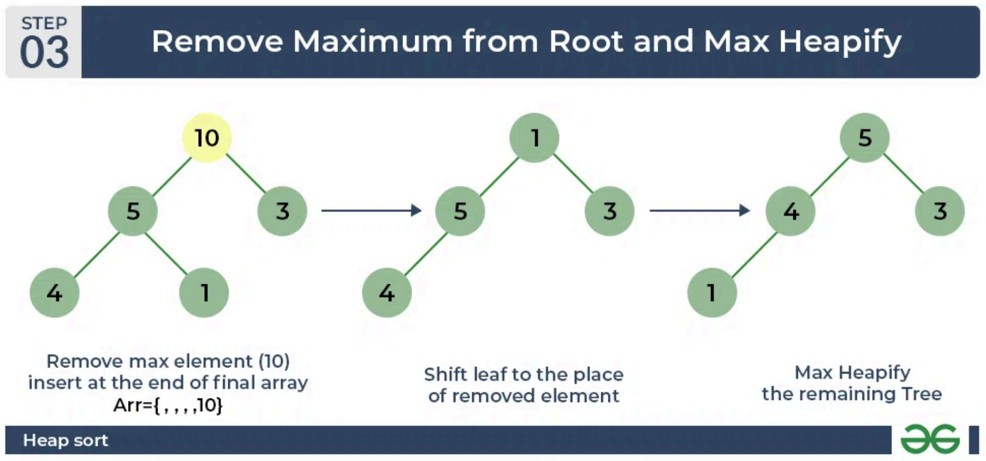
- Description: Heap Sort is a comparison-based sorting algorithm. It uses binary heap data structure. Heap sort can be thought of as an improved selection sort: like selection sort, heap sort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region.
- Example: Unsorted array: arr[] = [4, 10, 3, 5, 1]



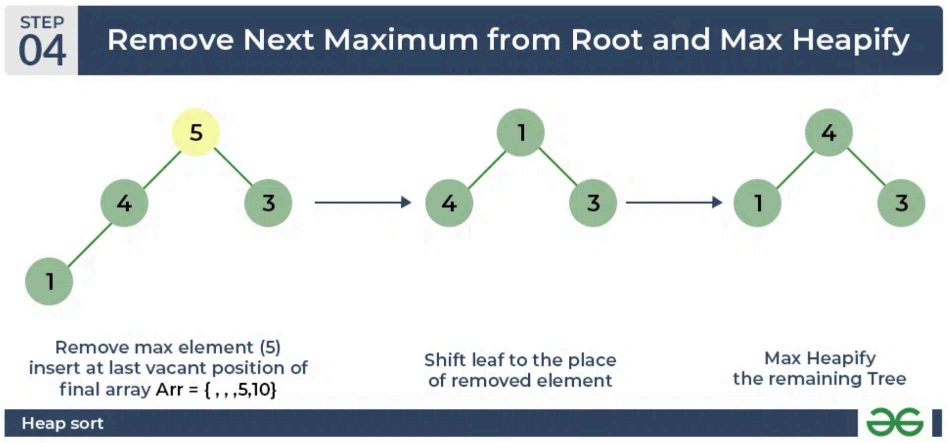
Transform into max heap: After that, the task is to construct a tree from that unsorted array and try to convert it into max heap. To transform a heap into a max-heap, the parent node should always be greater than or equal to the child nodes Here, in this example, as the parent node 4 is smaller than the child node 10, thus, swap them to build a max-heap. Now, 4 as a parent is smaller than the child 5, thus swap both of these again and the resulted heap and array should be like this:


STEP 02
Max Heapify Constructed Binary Tree
Swap (10 > 4)
Swap (5 > 4)
Max Heapify Root
Max Heapify non-leaf node
Final Tree after Max Heapify
Heap sort

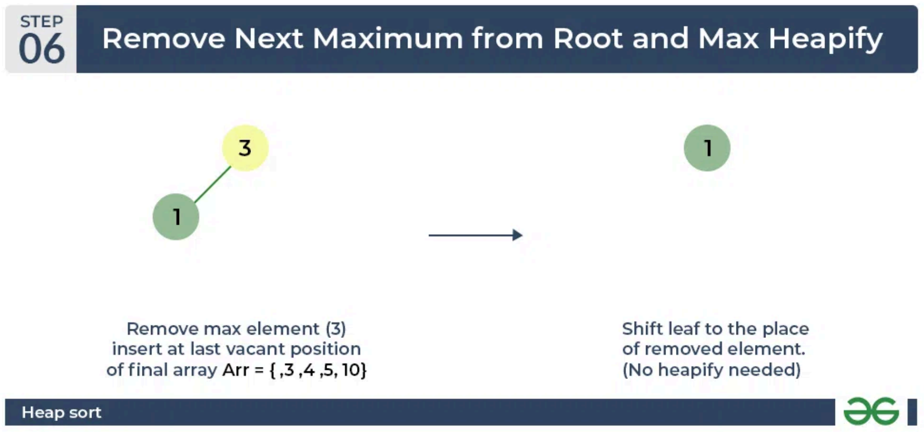
Perform heap sort: Remove the maximum element in each step (i.e., move it to the end position and remove that) and then consider the remaining elements and transform it into a max heap. Delete the root element (10) from the max heap. In order to delete this node, try to swap it with the last node, i.e. (1). After removing the root element, again heapify it to convert it into max heap. Resulted heap and array should look like this:


STEP 03
Remove Maximum from Root and Max Heapify
10
5
3
4
1
1
5
3
4
5
4
3
1
Remove max element (10) insert at the end of final array Arr={ , , , ,10}
Shift leaf to the place of removed element
Max Heapify the remaining Tree
Heap sort

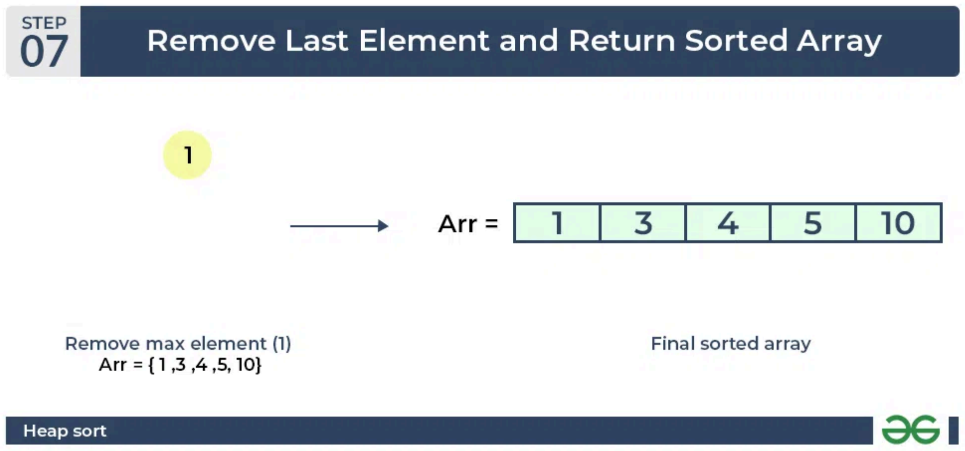
Repeat the above steps and it will look like the following:


STEP 04
Remove Next Maximum from Root and Max Heapify
5
4
3
1
1
4
3
4
1
3
Remove max element (5) insert at last vacant position of final array Arr = { , , ,5,10}
Shift leaf to the place of removed element
Max Heapify the remaining Tree
Heap sort

Now remove the root (i.e. 3) again and perform heapify.


STEP 06
Remove Next Maximum from Root and Max Heapify
3
1
1
Remove max element (3)
insert at last vacant position
of final array Arr = { ,3 ,4 ,5, 10}
Shift leaf to the place
of removed element.
(No heapify needed)
Heap sort

Now when the root is removed once again it is sorted. and the sorted array will be like arr[] = {1, 3, 4, 5, 10}.



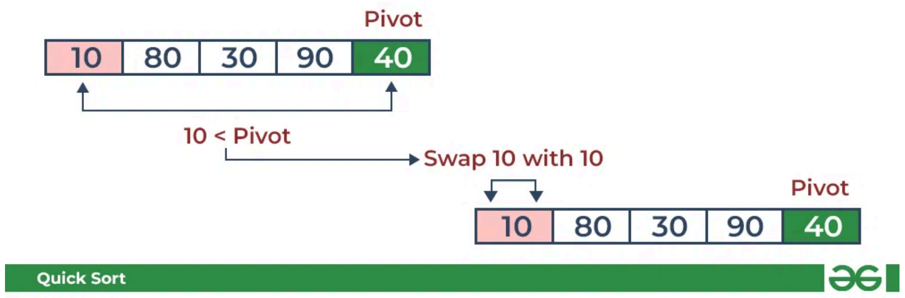
- Pseudocode

```
function heapSort(A):
    buildMaxHeap(A)
    for i from length(A) down to 2:
        swap A[1] with A[i]
        heapSize = heapSize - 1
        maxHeapify(A, 1)

function buildMaxHeap(A):
    heapSize = length(A)
    for i from floor(length(A)/2) down to 1:
        maxHeapify(A, i)

function maxHeapify(A, i):
    left = 2*i
    right = 2*i + 1
    if left <= heapSize and A[left] > A[i]:
        largest = left
    else:
        largest = i
    if right <= heapSize and A[right] > A[largest]:
        largest = right
    if largest != i:
        swap A[i] with A[largest]
        maxHeapify(A, largest)
```

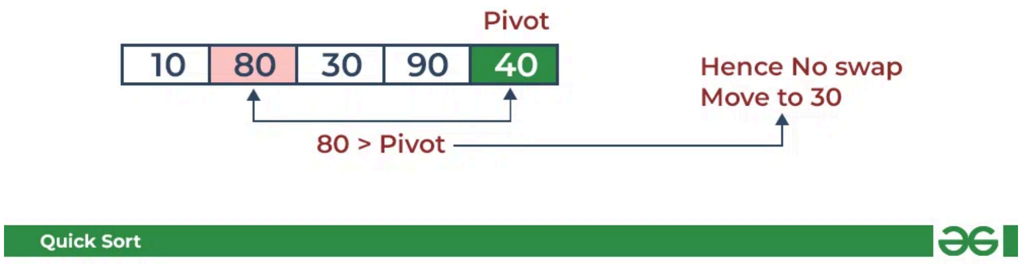
- Time Complexity: O(nlogn) for best, worst, and average case

Quick Sort:

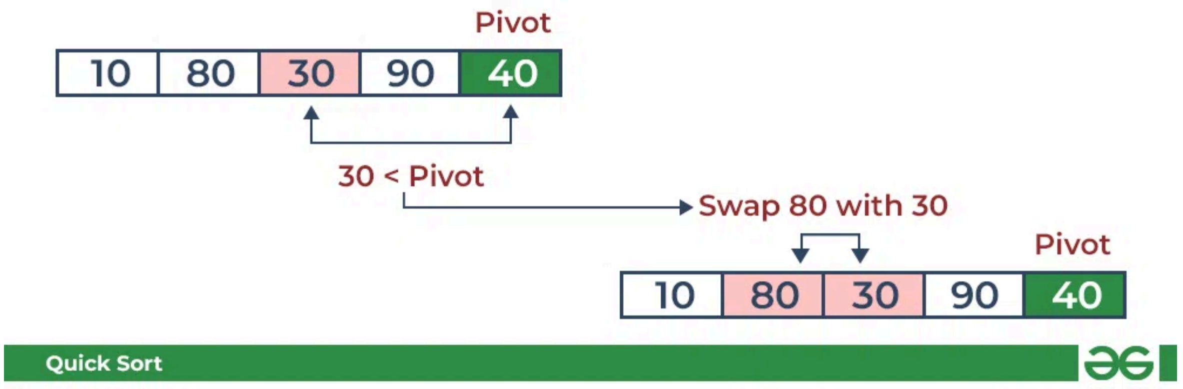
- Description: Quick Sort is a divide and conquer algorithm. It works by selecting a ‘pivot’ element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.
- Examples: Consider unsorted arrays arr[] = [10, 80, 30, 90, 40]
  Compare 10 with the pivot and as it is less than pivot arrange it accordingly.



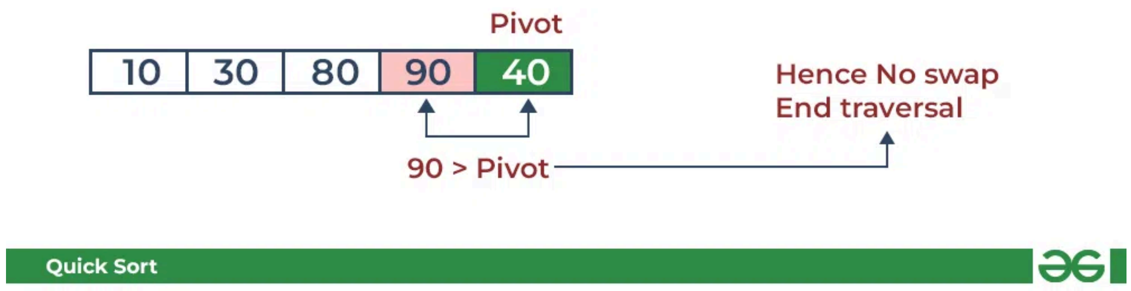
Compare 80 with the pivot. It is greater than pivot.



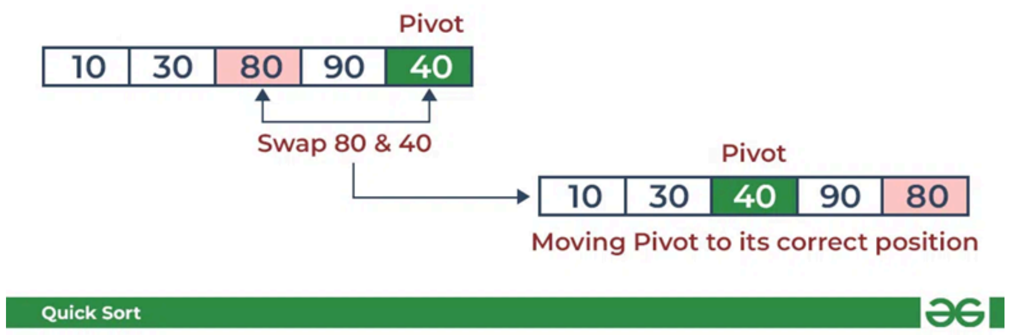
Compare 30 with pivot. It is less than pivot so arrange it accordingly.



Compare 90 with the pivot. It is greater than the pivot.



Arrange the pivot in its correct position.



- Pseudocode:

```
function quicksort(array)
  if length(array) $\le$ 1
    return array
 select and remove a pivot value 'pivot' from array
 create empty lists less and greater
 for each x in array
    if x $\le$ pivot then append x to less
    else append x to greater
 return concatenate(quicksort(less), pivot, quicksort(greater))
```

- Time Complexity:
  - Average Case: O(nlogn)
  - Worst Case: $O(n^2)$

Counting Sort

- Description: Counting Sort is not a comparison sort and works by counting the number of objects having distinct key values
- Examples: Let’s consider an array A = [2, 5, 3, 0, 2, 3, 0, 3]. Here’s how Counting Sort would sort this array:
    - o Find the maximum element. In this case, it’s 5.
    - o Initialize a count array of length max+1 (6 in this case) with all elements as 0.
    - o Store the count of each unique element of the input array at their respective indices in the count array. For example, the count of element 2 in the input array is 3. So, store 3 at index 2 in the count array1.
    - o Store the cumulative sum or prefix sum of the elements of the count array. This will help in placing the elements of the input array at the correct index in the output array1.
    - o Iterate from the end of the input array. Update the output array and also update the count array1
- Pseudocode:

```
function CountingSort(A, max):
    count = array of max+1 zeros
    output = array of same length as A

    # Store the count of each element in A
    for number in A:
        count[number] += 1

    # Store the cumulative count
    for i in range(1, len(count)):
        count[i] += count[i - 1]

    # Place the elements in the output array
    for number in reversed(A):
        output[count[number] - 1] = number
        count[number] -= 1

    return output
```

- Time Complexity:
    - o O(n+k) where n is the size of the input array and k is the range of the input

Linear Sorting:

- Description: Linear Sorting refers to a group of algorithms that can sort data in linear time.
- Example: Let's consider an array A = [2, 5, 3, 0, 2, 3, 0, 3]. Here's how Counting Sort would sort this array:
    - Find the maximum element. In this case, it's 5.
    - Initialize a count array of length max+1 (6 in this case) with all elements as 0.
    - Store the count of each unique element of the input array at their respective indices in the count array. For example, the count of element 2 in the input array is 3. So, store 3 at index 2 in the count array1.
    - Store the cumulative sum or prefix sum of the elements of the count array. This will help in placing the elements of the input array at the correct index in the output array1.
    - Iterate from the end of the input array. Update the output array and also update the count array1
- Pseudocode:

```
function CountingSort(A, max):
    count = array of max+1 zeros
    output = array of same length as A

    # Store the count of each element in A
    for number in A:
        count[number] += 1

    # Store the cumulative count
    for i in range(1, len(count)):
        count[i] += count[i - 1]

    # Place the elements in the output array
    for number in reversed(A):
        output[count[number] - 1] = number
        count[number] -= 1

    return output
```

- Time Complexity:
    - O(n + k) where n is the size of the input array and k is the range of the input

Medians & Order Statistics:

- Descriptions: Medians & Order Statistics are concepts in the field of statistics and computer science that deal with the relationship between elements in a dataset. The ith order statistic of a set of n elements is the ith smallest element. For example, the minimum of a set of elements is the first order statistic (i = 1), and the maximum one is the nth order statistic (i = n). The median of a set is the "halfway point" of the set. When n is odd, the median is unique, and i = (n + 1)/2; but when n is even, there are two such medians, occurring at i = n/2, and i = n/2 + 1.
- Example: Let's consider a set of 5 values, of the weights of rats: x1 = 603, x2 = 780, x3 = 710, x4 = 742, x5 = 630. We can arrange the set of data in ascending / increasing order. So the observed order statistics are : y1 = 603 < y2 = 630 < y3 = 710 < y4 = 742 < y5 = 780.
- Pseudocode:

```
function FindMinAndMax(A):
  if length[A] % 2 == 0:
    if A[1] > A[2]:
      min = A[2]
      max = A[1]
    else:
      min = A[1]
      max = A[2]
    for i = 2 to length[A]/2:
      large = max(large, max(A[2i - 1], A[2i]))
      small = min(small, min(A[2i - 1], A[2i]))
  else:
    min = max = A[1]
    for i = 2 to length[A]/2:
      large = max(large, max(A[2i - 1], A[2i]))
      small = min(small, min(A[2i - 1], A[2i]))
    large = max(large, A[length[A]])
    small = min(small, A[length[A]])
  return (large, small)
```

- Time Complexity:
    - Finding min and max is 3n/2

Radix Sort:

- Radix Sort is a non-comparative sorting algorithm and it is the most efficient and fastest linear sorting algorithm. Radix Sort was developed to sort large integers. It works by sorting elements based on each digits value, starting from the least significant digits and moving towards the most significant.
- Example: Consider arr[] = [170, 45, 75, 90, 802, 24, 2, 66]
    - First Pass (Unit Place):
        - Buckets [[170, 90], [802], [], [], [24], [45, 75], [66], [], [2], []]
        - Concatenated: [170, 90, 802, 24, 45, 75, 66, 2]
    - Second Pass (Tens Place):
        - Buckets: [[802, 2], [], [24], [], [45], [170, 75], [66, 90], [], [], []]
        - Concatenated: [802, 2, 24, 45, 170, 75, 66, 90]
    - Third Pass (Hundreds Place):
        - Buckets: [[2, 24, 45, 66, 75, 90], [170], [802], [], [], [], [], [], []]
        - Concatenated: [2, 24, 45, 66, 75, 90, 170, 802]
    - After the third pass, the array is sorted. The number of passes is determined by the numbers of digits in the maximum number in the array. In this case the maximum number is 802, which has three digits, ao there are 3 passes
- Pseudocode:

```
radixSort(arr):
  maxNum = maximum element in arr
  digitPlace = 1
 while maxNum / digitPlace > 0:
    countingSort(arr, digitPlace)
    digitPlace *= 10
countingSort(arr, digitPlace):
  n = length of arr
  output = new array of size n
  count = new array of size 10, initialized to 0
  `Count occurrences of each digit at the given digitPlace
  for i from 0 to n - 1:
    digit = (arr[i] / digitPlace) % 10
```

~~~
        count[digit] += 1

     Calculate cumulative count

    for i from 1 to 9:

        count[i] += count[i - 1]

    `` Build the output array

    for i from n - 1 to 0:

        digit = (arr[i] / digitPlace) % 10

        output[count[digit] - 1] = arr[i]

count[digit] -= 1

    ```Copy the output array to arr

    for i from 0 to n - 1:

        arr[i] = output[i]
~~~

- ➢ Time Complexity:
    - ○ O(nd) where n is the size of the array and d is the number of digits in the largest number

Bucket Sort:

- Description: Bucket Sort, also known as bin sort, is a sorting algorithm that divides an array's elements into several buckets. The buckets are then sorted one at a time, either using a different sorting algorithms or by recursively applying the bucket sorting algorithm
- Examples: Consider arr[] = [0.42, 0.32, 0.33, 0.52, 0.37, 0.47, 0.51]
    - Step 1 (Create Bucket): Creat an array of size 10 where each slot represent a bucket
    - Step 2 (Insert Elements): Insert elements into the bucket from the input array based on their range. For example, for element 0.23, we get 0.23 * 10 = 2.3. Convert the result to an integer, which gives us the bucket index. In this case, 2.3 is converted to the integer 2. Insert the element into the bucket corresponding to the calculated index. Repeat these steps for all elements in the input array.
    - Step 3 (Sort Buckets): Sor the elements within each bucket. In this example, we use a stable sorting algorithm (like quicksort) to sort the elements within each bucket
    - Step 4 (Gather Elements): Gather the elements from each bucket and put them back into the original array. Iterate through each bucket in order, Insert each individual element from the bucket into the original array. Once an element is copied, it is removed from the bucket. Repeat this process for all buckets until all elements have been gathered.
    - After the fourth step, the array is sorted. The number of passes is determined by the number of digits in the maximum number in the array, In this case the maximum number is 0.52, which has two digits, so there are two passes.
- Pseudocode:

```
function bucketSort(array, bucketSize)

 if array.length === 0 then

  return array

 end if

      Determine minimum and maximum values

 minValue = array[0]

 maxValue = array[0]

 for i from 1 to n - 1

  if array[i] < minValue then
```

```
      minValue = array[i]
    else if array[i] > maxValue then
      maxValue = array[i]
    end if
  end for
        Initialize buckets
  bucketCount = floor((maxValue - minValue) / bucketSize) + 1
  buckets = new array of bucketCount
  for i from 0 to bucketCount - 1
    buckets[i] = []
  end for
          Distribute input array values into buckets
  for i from 0 to array.length - 1
    buckets[floor((array[i] - minValue) / bucketSize)].push(array[i])
  end for
        Sort buckets and place back into input array
  array.length = 0
  for i from 0 to buckets.length - 1
    insertionSort(buckets[i])
    for j from 0 to buckets[i].length - 1
      array.push(buckets[i][j])
    end for
  end for
  return array
end function
```

- ➢ Time Complexity:
    - ○ Average time: O(n + k)
    - ○ Worst time: O(n^2)

QuickSelect Method Algorithm:

- ➢ Description: QuickSelect is a selection algorithm to find the k-th smallest element in an unordered list. It is related to the quick sort sorting algorithm. The difference is, instead of recurring for both sides (after finding pivot), it recurs only for the part that contains the k-th smallest element.
- ➢ Example: Consider arr[] = [7, 10, 4, 3, 20, 15]
    - ○ To find third smallest element in the array, first choose a pivot: For simplicity, let’s choose the last element, 15
    - ○ Partition the array around the pivot: This step rearranges the elements so that all elements less than the pivot come before all elements greater than the pivot. For our array, the partitioning might look like this: arr[] = [7, 10, 4, 3, 15, 20]. The pivot, 15, is now in its final sorted position.
    - ○ Check the pivot’s index against k: The pivot is the 5th smallest element, but we’re looking for the 3rd smallest, so we need to recur on the left part of the array.
    - ○ Recur on the appropriate part of the array: Since k is less than the pivot’s index, we recur on the left part of the array, We choose a new pivot, say 10, and partition around it: arr[] = [7, 4, 3, 10, 15, 20]. Now, the pivot 10 is the 4th smallest element. Since k is less than the pivot’s index, we again recur on the left part of the array
    - ○ Continue until the pivot’s index is equal to k: We continue choosing pivots, partitioning, and recurring on the appropriate parts of the array until the pivot’s index is equal to k. At that point, the pivot is the k-th smallest element, and we return it. In this example, the 3rd smallest element is 7.
- ➢ Pseudocode:

```
function quickSelect(list, left, right, k)

  if left == right

    return list[left]

     Select a pivotIndex between left and right

  pivotIndex = partition(list, left, right, pivotIndex)



  if k == pivotIndex
  return list[k]
else if k < pivotIndex
  return quickSelect(list, left, pivotIndex - 1, k)
else
  return quickSelect(list, pivotIndex + 1, right, k)
```

- Time Complexity:
  - Average time: O(n)
  - Worst time: O(n^2)

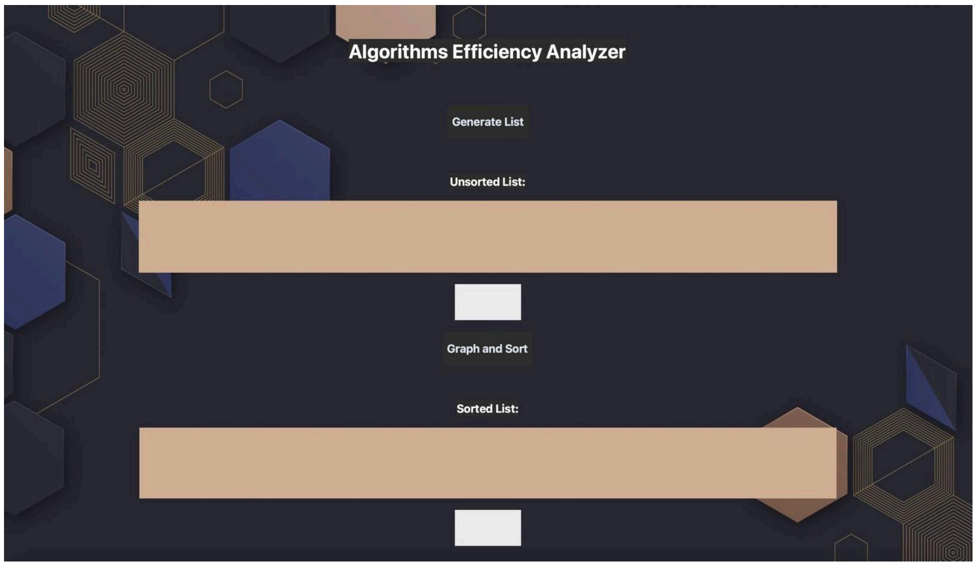
:

GUI Designs (Mrunal Patil):

- GUI was designed to be interactive, intuitive, and capable of handling potential invalid inputs

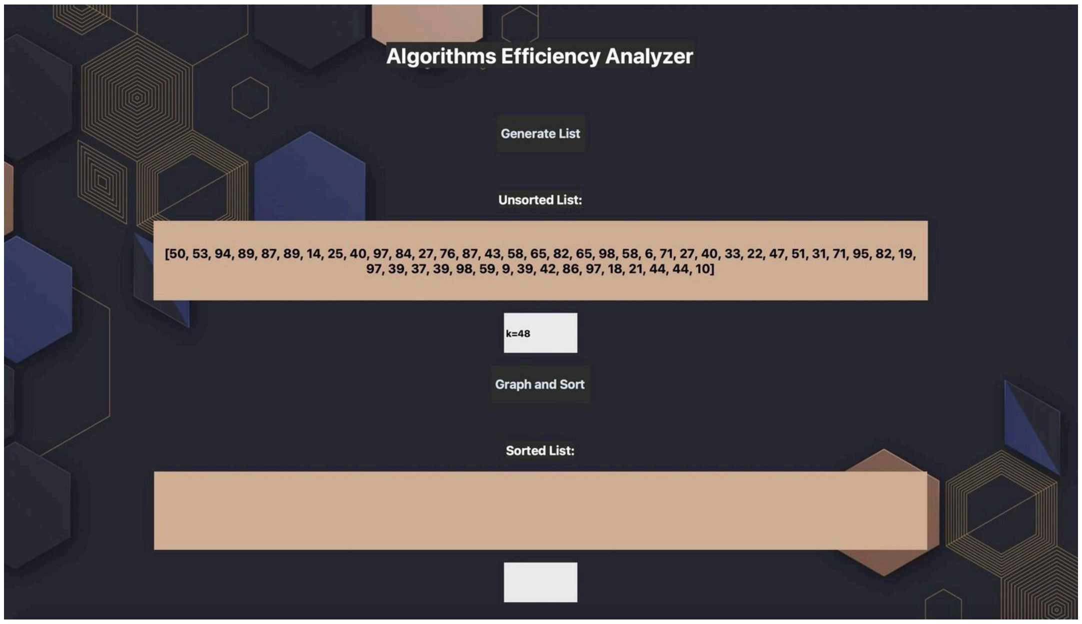
Screenshot of the Algorithm Analyzer Main Interface:

- ➢ The main interface shows the users with the option to generate a list of numbers and view the unsorted list before sorting. This interface provides a clear starting point for users to interact with the tool.



Generate List of Unsorted Arrays:

➢ Once the users press the button ‘Generate List’, the users can see the list of unsorted numbers. This allows the users to verify the input data before proceeding with sorting, The unsorted list view ensures transparency and allows the user to understand the data being analyzed. ‘k=48’ is the number of elements listed in the array.

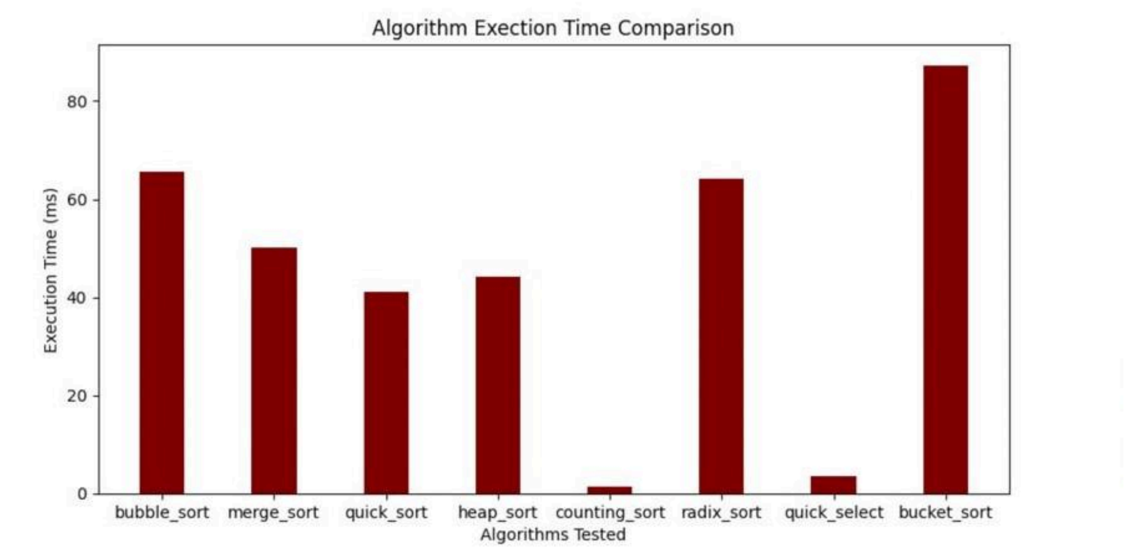


```
[50, 53, 94, 89, 87, 89, 14, 25, 40, 97, 84, 27, 76, 87, 43, 58, 65, 82, 65, 98, 58, 6, 71, 27, 40, 33, 22, 47, 51, 31, 71, 95, 82, 19,
97, 39, 37, 39, 98, 59, 9, 39, 42, 86, 97, 18, 21, 44, 44, 10]
```

k=48

Graph and Sort:

- After viewing the unsorted array list, users can now select the button ‘Graph and Sort’ to view the sorting algorithm and initialize the sorting process. The graph visually represents the order of each sorting algorithm. Users can view the sorting algorithm performance in real time, improving their understanding of algorithm logicality. The title of the bar graph is ‘Algorithm Execution Time Comparison’ with the x-axis showing Algorithm Tested of each sorting algorithm and the y-axis showing Execution Time in miliseconds.



- These screenshots demonstrate the GUI’s layout and design, highlighting simplicity, clarity, and functionality. The GUI design stregthen user interaction and feedback, furthering a smooth user experience throughout the algorithm analysis process.

Visualization:

- Visual Aids like a bar graph were creating using ‘matplotlib’ to display the efficiency of each algorithm
- Collaboration between the GUI Developer and Data Visualization Expert ensured a smooth integration

1. Bar Graph
    a. The bar graph visually represents the logicality of each sorting algorithm, showcasing the time taken to sort a given dataset. Refer to the screenshot included above in the GUI Design section for a visual representation of the bar graph.

Testing:

- Comprehensive test cases were developed to ensure the reliability of the tool
- Results of tests with different input sizes and datasets were documented.
- Any anomalies encountered during testing were addressed promptly

Challenges Faced:

- Challenge: Ensuring smooth collaboration between team members
    - Resolution: Regular team meetings mostly online and effective use of version control tools diminished collaboration challenges
- Challenge: Optimizing sorting algorithms for correctness
    - Resolution: Through testing and peer reviews helped identify and resolved certain issues

Conclusion:

- The Algorithm Efficiency Analyzer Tool successfully achieved its objective of providing an insightful exploration of sorting algorithm efficiency. The collaboration between team members and the application of diverse skills contributed to a robust and valuable tool for algorithm analysis

References:

- Github links that helped with information related to Sorting Algorithms:
  - https://github.com/jwdotpark/2021_fall/blob/c7187f3fac25d8af81f0e04d2fab88585b67cdce/practical_ds_a/scavenge_hunting.md#bubble-sort
  - https://github.com/JSchoppe/CommonAlgorithmsCSharp/blob/87dbf411c608fea6527dbab74618ba5326a046e3/AlgorithmSamples/SortingAlgorithms/README.md
- GeeksforGeeks:
  - https://www.geeksforgeeks.org/quick-sort/
  - https://www.geeksforgeeks.org/heap-sort/

Acknowledgements:

- To all my team members for their dedicated hard work and support throughout the project:
  - Brandon Kilpatrick for his great work as a Project Coordinator
  - Neema Tabarani and Asad Abdul for getting the algorithms to work efficiently
  - Mrunal Patil for her hard work as a GUI Developer
  - Akshay Sakpal for his hard work as a Data Visualization Expert