Project Report of CSE 5311

on Sorting Algorithms

by

Nahin Kumar Dey

UTA ID:1002204142

Dept. of Computer Science and Engineering

The University of Texas at Arlington

500 UTA Blvd, Arlington, TX 76010

April 2024

Nahin Kumar Dey
UTA ID: 1002204142

Contents

Nahin Kumar Dey
UTA ID: 1002204142

Part 01: The Algorithms

1. Merge Sort:

Merge Sort follows a divide-and-conquer strategy. It recursively divides the unsorted list into sub-lists containing a single element (considered sorted). These sub-lists are then merged back together in a sorted manner.

Data Structure: Liner data structures like arrays and linked lists. Temporary arrays might be used during the merge process.

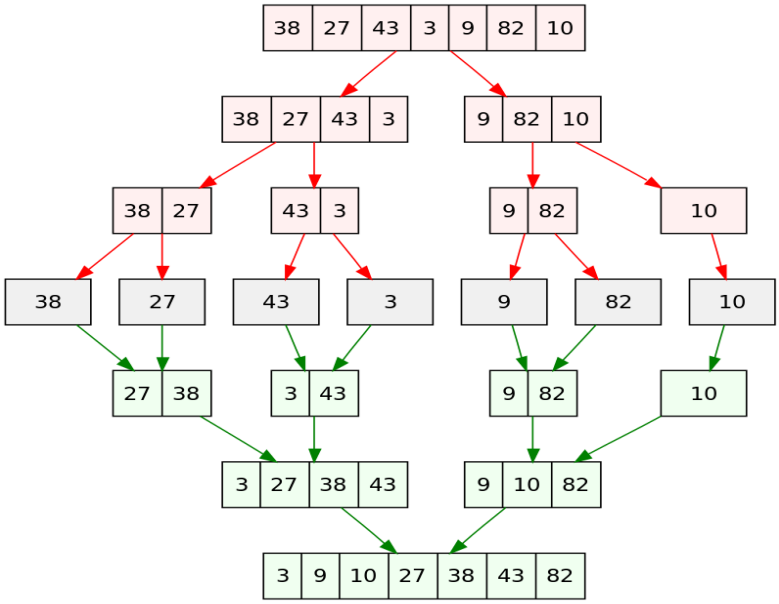

Figure 01: Merge sort [1]

Algorithm:

1. **Divide**: The function mergeSort recursively divides the array into halves until single-element sub-arrays are reached (base case).
2. **Conquer**: Each sub-array of size 1 is considered sorted.
3. **Merge**: The function merge takes two sorted sub-arrays and combines them into a single sorted sub-array within the original array. This merging process compares elements from both sub-arrays and inserts them into the final sorted sub-array in the correct order.

Pseudocode:

```
MergeSort(array):
  if length of array <= 1:
    return array
  mid = length of array // 2
  left_half = MergeSort(first half of array)
```

```
    right_half = MergeSort(second half of array)
    return Merge(left_half, right_half)

Merge(left_half, right_half):
    merged_array = []
    i = j = 0
    while i < length of left_half and j < length of right_half:
        if left_half[i] <= right_half[j]:
            append left_half[i] to merged_array
            increment i
        else:
            append right_half[j] to merged_array
            increment j
    append remaining elements of left_half and right_half to merged_array
    return merged_array
```

Below is the **implementation** of the above approach:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
```

```
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Complexity:

Time Complexity:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n log n)

Space Complexity:

O(n) due to temporary arrays used during the merge process.

References:

1. https://en.wikipedia.org/wiki/Merge_sort
2. https://github.com/Devinterview-io/sorting-algorithms-interview-questions
3. https://dev.to/ladunjexa/sorting-algorithms-from-classic-to-modern-o07?comments_sort=top

2. Heap Sort:

Heap Sort utilizes a heap data structure to sort the elements. It builds a max heap from the input array, where the largest element is at the root. The largest element is then swapped with the last element and the heap property is restored on the remaining sub-array. This process (extracting the largest element and fixing the heap) is repeated until the entire array is sorted.

Data Structure:

Primary data structure: Array (used to represent a heap).

Nahin Kumar Dey
UTA ID: 1002204142

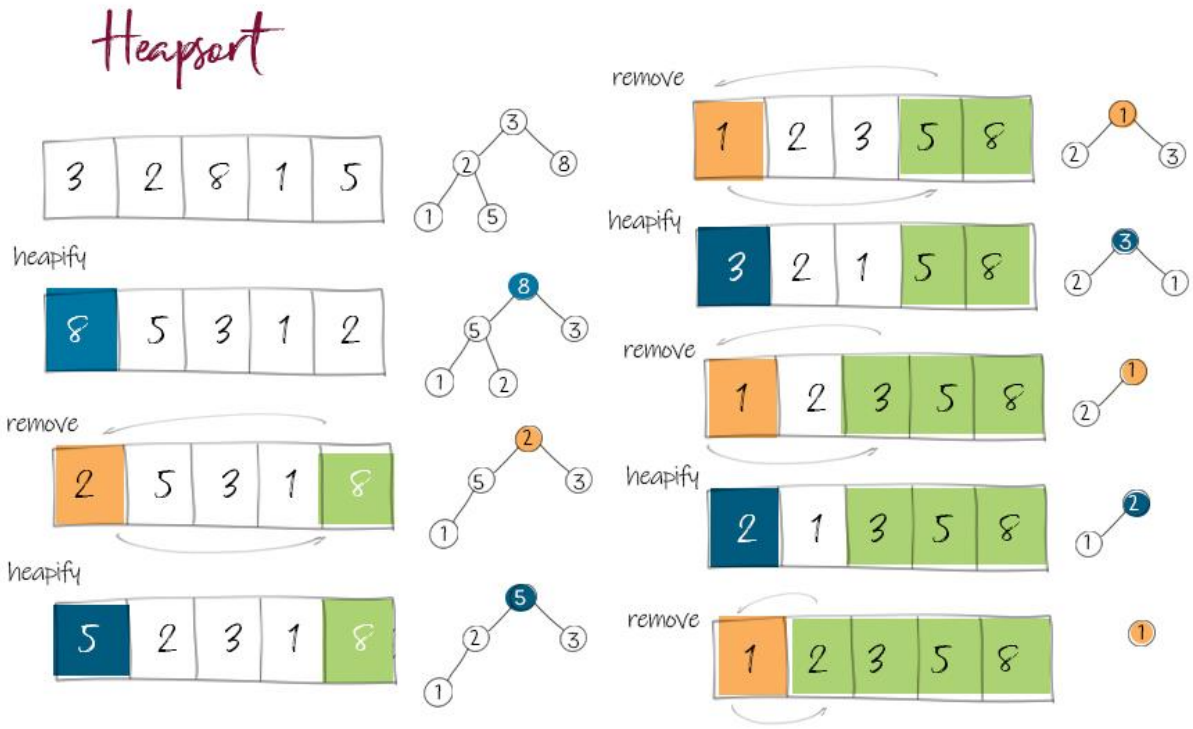

Figure 02: Heap sort [4]

Algorithm:

1. **Build Heap**: The function HeapSort first builds a max-heap from the input array using the Heapify function. This function ensures that the largest element is at the root and the subtree rooted at each node has the property that the root is greater than the child nodes.
2. **Extract Maximum**: After building the max heap, HeapSort iteratively extracts the largest element (which is at the root) from the heap and swaps it with the last element in the unsorted sub-array.
3. **Heapify Down**: The Heapify function is called again on the reduced sub-array (excluding the last element) to restore the max-heap property. This ensures the largest element remains at the root of the reduced heap.
4. **Repeat**: Steps 2 and 3 are repeated until all elements have been extracted from the heap (which is equivalent to having all elements sorted in the array).

Pseudocode:

```
HeapSort(array):
    n = length of array
    # Build max heap
    for i from n // 2 - 1 down to 0:
```

```
        heapify(array, n, i)
    # Extract elements from heap
    for i from n - 1 down to 0:
        swap array[0] with array[i]
        heapify(array, i, 0)


heapify(array, n, i):
    largest = i
    left_child = 2 * i + 1
    right_child = 2 * i + 2
    if left_child < n and array[left_child] > array[largest]:
        largest = left_child
    if right_child < n and array[right_child] > array[largest]:
        largest = right_child
    if largest != i:
        swap array[i] with array[largest]
        heapify(array, n, largest)
```

Below is the **implementation** of the above approach:

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2


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


    if r < n and arr[r] > arr[largest]:
        largest = r
```

```
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)
    for i in range(n, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

Complexity:

Time Complexity:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n log n)

Space Complexity:

O(1) auxiliary space (in-place algorithm).

References:

1. https://en.wikipedia.org/wiki/Heapsort
2. https://github.com/Devinterview-io/sorting-algorithms-interview-questions
3. https://dev.to/ladunjexa/sorting-algorithms-from-classic-to-modern-o07?comments_sort=top
4. https://www.lavivienpost.net/wp-content/uploads/2022/02/heapsort.jpg

Nahin Kumar Dey
UTA ID: 1002204142

3. Quick Sort:

Quick Sort is a divide-and-conquer sorting algorithm that works by selecting a pivot element from the array and partitioning the remaining elements into two sub-arrays:

- Elements less than the pivot
- Elements greater than or equal to the pivot

These sub-arrays are then recursively sorted, and finally, the pivot is placed in its correct sorted position.

Data Structure: Primarily uses the array to store the elements.

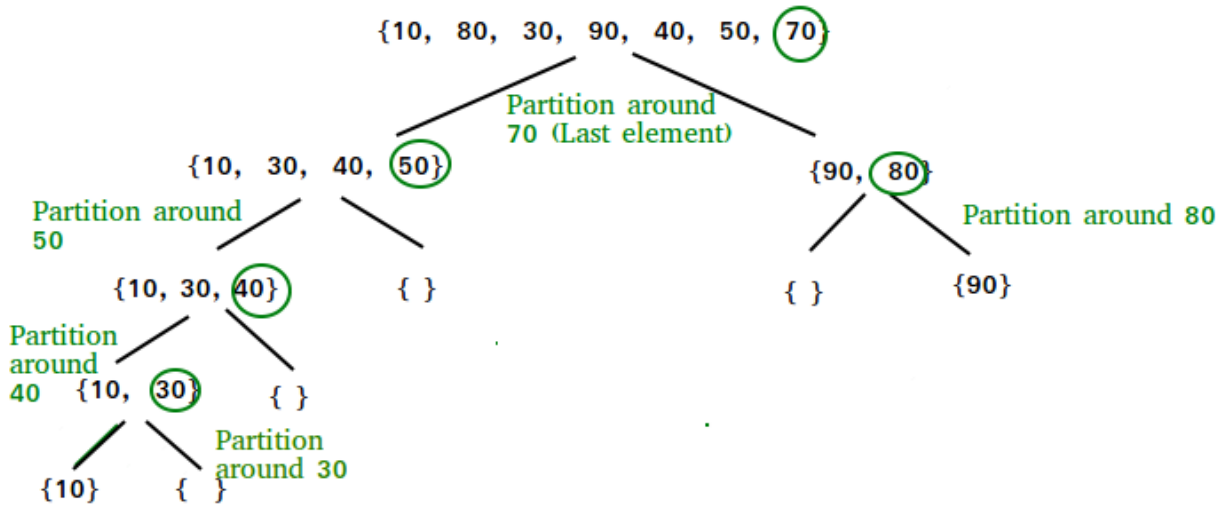

Figure 03: Quick sort [4]

Algorithm:

1. **Choose Pivot**: The pivot element can be chosen in various ways, such as the first, last, or median element. Here, we'll use the last element for simplicity.
2. **Partition**: The partition function rearranges the array such that all elements less than the pivot are placed before it, and all elements greater than or equal to the pivot are placed after it. The function returns the final index of the pivot element in its sorted position.
3. **Recursive Calls**: The QuickSort function recursively sorts the two sub-arrays created by the partitioning process: the elements less than the pivot and the elements greater than or equal to the pivot.

Pseudocode:

```
QuickSort(arr, low, high)
  if low < high
    pi = partition(arr, low, high)
```

```
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)


Partition(arr, low, high)
    pivot = arr[high]
    i = low - 1
    for j = low to high - 1
        if arr[j] <= pivot
            i++
            swap arr[i] with arr[j]
    swap arr[i + 1] with arr[high]
    return i + 1
```

Below is the **implementation** of the above approach:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

Complexity:

Time Complexity:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: $O(n^2)$

Space Complexity:

O(log n) due to the recursive calls.

Nahin Kumar Dey
UTA ID: 1002204142

References:

1. https://en.wikipedia.org/wiki/Quicksort
2. https://github.com/Devinterview-io/sorting-algorithms-interview-questions
3. https://dev.to/ladunjexa/sorting-algorithms-from-classic-to-modern-o07?comments_sort=top
4. https://www.geeksforgeeks.org/quick-sort/

4. Quick Sort (Median of 3):

It is a variant of the traditional Quick Sort algorithm that aims to improve its average-case performance by choosing a better pivot element.

Data Structure: Primary data structure: Array

Algorithm:

1. Median of 3 Pivot Selection:
    a. Before partitioning, select three elements from the array:
        i. First element (index 0)
        ii. Last element (index n-1)
        iii. Middle element (index (n-1) // 2)
    b. Find the median of these three elements. This can be done by sorting the three elements and selecting the middle one.
    c. Use the median element as the pivot.
2. Partitioning and Recursion:
    a. The rest of the algorithm follows the standard Quick Sort approach:
        i. Partition the array around the chosen pivot element.
        ii. Recursively sort the two sub-arrays created by the partition (elements less than the pivot and elements greater than or equal to the pivot).

Pseudocode:

```
quick_sort_median3(arr, low, high):
  if low < high:
    pivot_index = median_of_three(arr, low, high)
    pivot_new_index = partition(arr, low, high, pivot_index)
    quick_sort_median3(arr, low, pivot_new_index - 1)
    quick_sort_median3(arr, pivot_new_index + 1, high)

median_of_three(arr, low, high):
  mid = (low + high) // 2
```

```
    # Order the first, middle, last elements and take the middle one as the pivot
    if arr[low] > arr[mid]:
        arr[low], arr[mid] = arr[mid], arr[low]
    if arr[low] > arr[high]:
        arr[low], arr[high] = arr[high], arr[low]
    if arr[mid] > arr[high]:
        arr[mid], arr[high] = arr[high], arr[mid]
    # Now the median is at the mid index
    return mid


partition(arr, low, high, pivot_index):
    pivot_value = arr[pivot_index]
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]  # Move pivot to end
    store_index = low
    for i in range(low, high):  # Exclude the pivot itself at 'high'
        if arr[i] <= pivot_value:
            arr[i], arr[store_index] = arr[store_index], arr[i]
            store_index += 1
    arr[store_index], arr[high] = arr[high], arr[store_index]
    return store_index
```

Below is the **implementation** of the above approach:

```
def quick_sort_median(arr):
    if len(arr) <= 1:
        return arr
    if len(arr) < 3:
        return quick_sort(arr)
    mid_index = len(arr) // 2
    median_of_three = [arr[0], arr[mid_index], arr[-1]]
    pivot = sorted(median_of_three)[1]
```

```
left = [x for x in arr if x < pivot]

middle = [x for x in arr if x == pivot]

right = [x for x in arr if x > pivot]

return quick_sort(left) + middle + quick_sort(right)
```

Complexity:

Time Complexity:

Best Case: O(n log n)

Average Case: O(n log n)

Worst Case: O(n^2) (Improved probability due to better pivot choice)

Space Complexity:

O(log n) due to recursion stack space.

References:

1. https://www.cs.cornell.edu/courses/JavaAndDS/files/sort3Quicksort3.pdf
2. https://github.com/Devinterview-io/sorting-algorithms-interview-questions
3. https://dev.to/ladunjexa/sorting-algorithms-from-classic-to-modern-o07?comments_sort=top
4. https://www.reddit.com/r/math/comments/2yi8wd/how_to_do_quick_sort_algorithm_using_median_of_3/

5. Insertion Sort:

Insertion Sort is an algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort or merge sort. It is suitable for small data sets or scenarios where the array is partially sorted. It excels in situations where random access to elements is needed during the sorting process.

Data Structure: Primary data structure: Array

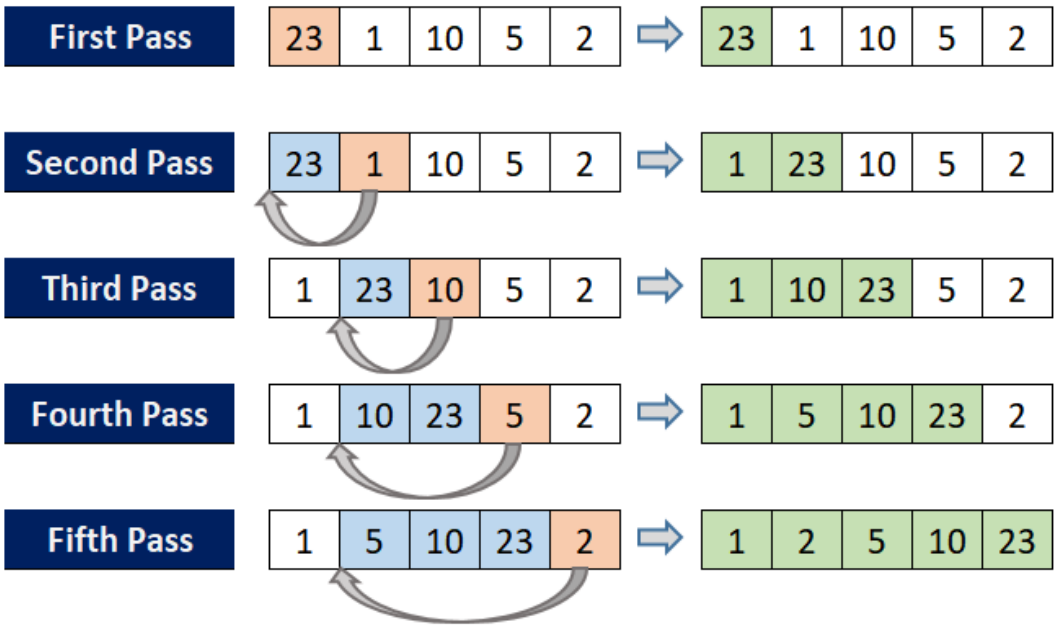

Figure 04: Insertion sort [4]

Algorithm:

1. **Start**: Consider the first element as the sorted sub-array (size 1).
2. **Iterate**: For each element (starting from the second element), compare it to the elements in the sorted sub-array.
    a. If the element is greater than all elements in the sub-array, place it at the end.
    b. Otherwise, shift larger elements in the sub-array from one position to the right and insert the element at the correct position.
3. **Repeat**: Continue iterating for all remaining elements, gradually building the sorted sub-array.

Pseudocode:

```
insertionSort(array, n)

 for i = 1 to n - 1 do

  key = array[i]  // Element to be inserted

  j = i - 1


  // Move elements of sorted sub-array that are greater than the key

  while j >= 0 and array[j] > key do

   array[j + 1] = array[j]

   j = j - 1
```

```
        endfor


        array[j + 1] = key  // Insert the key at its correct position
      endfor
```

Below is the **implementation** of the above approach:

```
def insertion_sort(arr):
  for i in range(1, len(arr)):
    key = arr[i]
    j = i - 1
    while j >= 0 and key < arr[j]:
      arr[j + 1] = arr[j]
      j -= 1
    arr[j + 1] = key
```

Complexity:

Time Complexity:

Worst-case: $O(n^2)$ (occurs when the array is in reverse order)

Average-case: $O(n^2)$

Best-case: O(n) (already sorted array)

Space Complexity:

O(1) (in-place sorting)

References:

1. https://en.wikipedia.org/wiki/Insertion_sort
2. https://github.com/Devinterview-io/sorting-algorithms-interview-questions
3. https://dev.to/ladunjexa/sorting-algorithms-from-classic-to-modern-o07?comments_sort=top
4. https://miro.medium.com/v2/resize:fit:765/0*1zi2XtjiLXa3LYZh.PNG

6. Selection Sort:

Selection Sort is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list. It iteratively finds the minimum element in the unsorted sub-array and

swaps it with the first element. This process continues, reducing the unsorted sub-array by one position in each iteration.

Data Structure: Primary data structure: Array

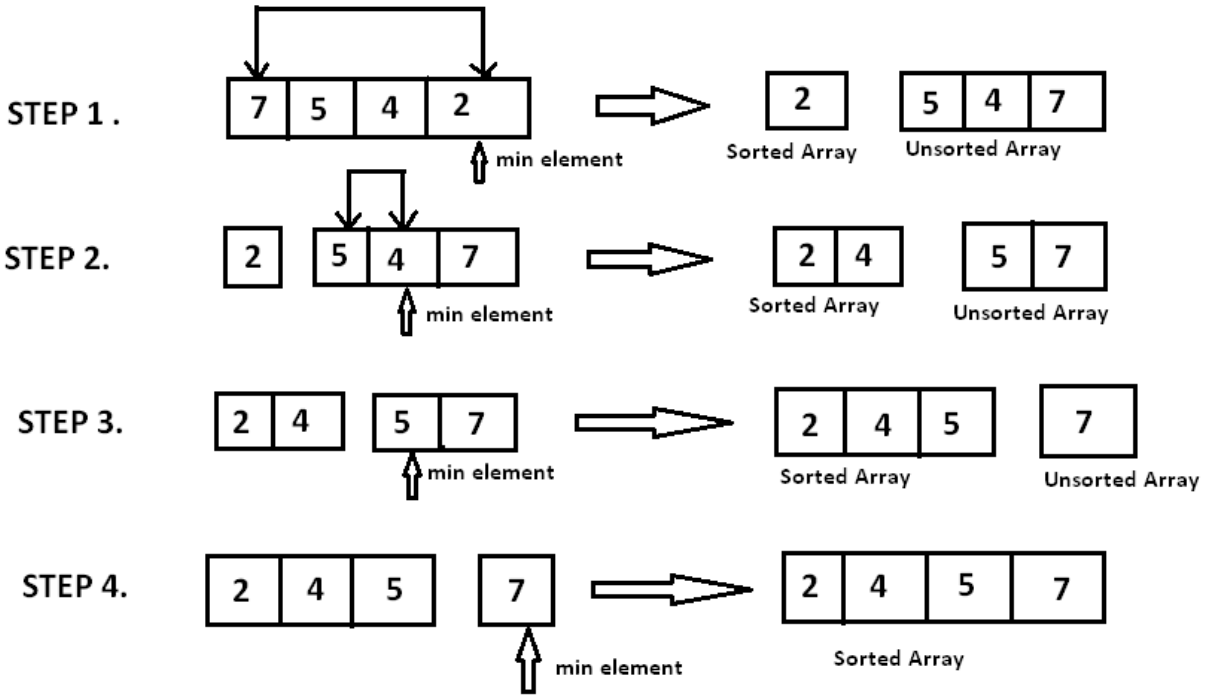

Figure 05: Selection sort [4]

Algorithm:

1. **Iterate**: For each element (except the last), find the minimum element in the unsorted sub-array (from the current element to the end).
2. **Swap**: Swap the minimum element with the first element of the unsorted sub-array.
3. **Repeat**: Continue iterating for the remaining elements, gradually building the sorted sub-array at the beginning.

Pseudocode:

```
SelectionSort(arr)
  n = length(arr)
  for i from 0 to n - 1
    min_idx = i
    for j from i + 1 to n
```

```
        if arr[j] < arr[min_idx]
            min_idx = j
    swap arr[i] with arr[min_idx]
```

Below is the **implementation** of the above approach:

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_index = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
```

Complexity:

Time Complexity:

Best case: $O(n^2)$ (compares each element with every other element)

Average case: $O(n^2)$

Worst case: (n^2)

Space Complexity:

O(1) (in-place sorting)

References:

1. https://en.wikipedia.org/wiki/Selection_sort
2. https://github.com/Devinterview-io/sorting-algorithms-interview-questions
3. https://dev.to/ladunjexa/sorting-algorithms-from-classic-to-modern-o07?comments_sort=top
4. https://he-s3.s3.amazonaws.com/media/uploads/2888f5b.png

7. Bubble Sort:

Bubble Sort 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 no swaps are needed, which means the list is sorted.

Data Structure: Primary data structure: Array

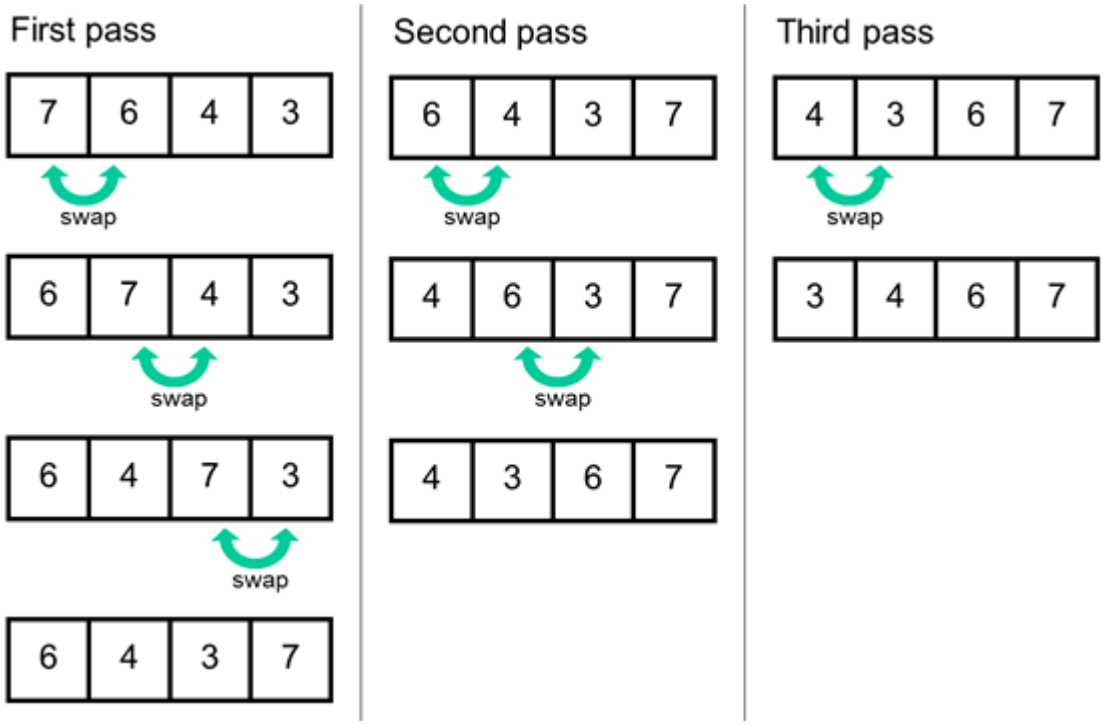

Figure 06: Bubble sort [4]

Algorithm:

1. Repeatedly iterates through the array.
    a. In each iteration, compare adjacent elements.
    b. If elements are in the wrong order (larger element comes first), swap them.
2. This "bubbling" process pushes larger elements towards the end of the array.
3. Continues iterating until no swaps occur in a pass, indicating the array is sorted.

Pseudocode:

```
BubbleSort(arr)
  n = length(arr)
  for i from 0 to n - 1
    swapped = false
    for j from 0 to n - i - 1
      if arr[j] > arr[j + 1]
        swap arr[j] with arr[j + 1]
        swapped = true
    if not swapped
      break
```

Below is the **implementation** of the above approach:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Complexity:

Time Complexity:

Worst-case: $O(n^2)$ (occurs when the array is in reverse order)

Average-case: $O(n^2)$

Best-case: O(n) (already sorted array)

Space Complexity:

O(1) (in-place sorting)

References:

1. https://en.wikipedia.org/wiki/Bubble_sort
2. https://github.com/Devinterview-io/sorting-algorithms-interview-questions
3. https://dev.to/ladunjexa/sorting-algorithms-from-classic-to-modern-o07?comments_sort=top
4. https://www.computersciencebytes.com/wp-content/uploads/2016/10/bubble_sort.png

Nahin Kumar Dey
UTA ID: 1002204142

Part 02: Experimental setup and results

Array generation:
Generates random data (python list) based on the user-given input size. It generates the list with the help of randint() function. The generated list contents are in between any number between 0 to 9999.

Implementation:

```
def generate_data(size):
    return [random.randint(0, 10000) for _ in range(1, size+1)]
```

Measure sorting time:

Measures the sorting time for a given sorting function and data.

- The time function starts by calling the time() function and for any given input size the data gets generated using the generate_data function and that data is passed to the selected sorting algorithm.
- After that the time() function is called again and the difference is measured.
- For the second implementation perf_counter() has been used which is more precise than time.time() for timing intervals.
- Also, data.copy() is used to make sure the same unsorted list is used for each sort to ensure fairness in comparison.

Implementation:

```
def measure_time(sort_func, data):
    start_time = time.time()
    sort_func(data.copy())
    end_time = time.time()
    return end_time - start_time
```

or

```
def measure_time(sort_func, data):
    start_time = perf_counter()
    sort_func(data.copy())  # Copy data to ensure each sort starts with the same input
    end_time = perf_counter()
    return end_time - start_time
```

Nahin Kumar Dey
UTA ID: 1002204142

Graphical User Interface:

Users can select one or more sorting algorithms, enter the input size, and view the runtime results of the selected algorithms. It provides a convenient way to compare the performance of different sorting algorithms. For the design of the user interface, **Tkinter** has been used in Python.

Define GUI Functions:

1. create_gui(): Creates and configures the GUI elements.
    a. Inside create_gui(), we define the following functions:
        i. run_algorithm(): Retrieves the selected algorithms and input size, generates random data, runs the selected algorithms on the data, and displays the runtime results.
        ii. add_algorithm_combobox(): Adds a new algorithm selection dropdown when the user clicks the "Add Algorithm" button.

Create the GUI:

1. Inside create_gui(), we create a new Tkinter window (root) and set its title.
2. Labels, dropdowns, entry fields, buttons, and a text label are defined to display the runtime results.
3. Widgets are arranged using the grid() method to place them in rows and columns within the window.

Execute the GUI:

1. We call the create_gui() function to create and display the GUI.
2. root.mainloop() starts the event loop, allowing the GUI to handle user interactions and events.

Implementation:

```
def create_gui():

   def run_algorithm():

      selected_algorithms = [algorithm_comboboxes[i].get() for i in
range(len(algorithm_comboboxes))]

      size = int(input_size_entry.get())
```

```
        data = generate_data(size)
        #print(data, "\n")
        result_text = ""
        for algorithm_name in selected_algorithms:
            if algorithm_name == "Merge Sort":
                time_taken = measure_time(merge_sort, data)
                print("Taken time for Merge Sort:", time_taken)
            elif algorithm_name == "Heap Sort":
                time_taken = measure_time(heap_sort, data)
                print("Taken time for Heap Sort:", time_taken)
            elif algorithm_name == "Quick Sort":
                time_taken = measure_time(quick_sort, data)
                print("Taken time for Quick Sort:", time_taken)
            elif algorithm_name == "Quick Sort (Median of 3)":
                time_taken = measure_time(quick_sort_median, data)
                print("Taken time for Quick Sort (Median of 3):)", time_taken)
            elif algorithm_name == "Insertion Sort":
                time_taken = measure_time(insertion_sort, data)
                print("Taken time for Insertion Sort:", time_taken)
            elif algorithm_name == "Selection Sort":
                time_taken = measure_time(selection_sort, data)
                print("Taken time for Selection Sort:", time_taken)
            elif algorithm_name == "Bubble Sort":
                time_taken = measure_time(bubble_sort, data)
                print("Taken time for Bubble Sort:", time_taken)
            result_text += f"Runtime for {algorithm_name}: {time_taken:.9f} seconds\n"

        result_label.config(text=result_text)
```

```
    def add_algorithm_combobox():
        new_algorithm_combobox = ttk.Combobox(root, values=algorithms,
state="readonly")
        new_algorithm_combobox.grid(row=len(algorithm_comboboxes) + 4, column=0,
padx=10, pady=5)
        algorithm_comboboxes.append(new_algorithm_combobox)

    root = tk.Tk()
    root.title("Sorting Algorithm Comparison")

    algorithm_label = ttk.Label(root, text="Select Algorithm(s):")
    algorithm_label.grid(row=0, column=0, padx=10, pady=5)

    algorithms = [
        "Merge Sort",
        "Heap Sort",
        "Quick Sort",
        "Quick Sort (Median of 3)",
        "Insertion Sort",
        "Selection Sort",
        "Bubble Sort"
    ]
    algorithm_comboboxes = []
    initial_algorithm_combobox = ttk.Combobox(root, values=algorithms,
state="readonly")
    initial_algorithm_combobox.grid(row=0, column=1, padx=10, pady=5)
    algorithm_comboboxes.append(initial_algorithm_combobox)

    add_algorithm_button = ttk.Button(root, text="Add Algorithm",
command=add_algorithm_combobox)
```

```
    add_algorithm_button.grid(row=0, column=2, padx=10, pady=5)

    input_size_label = ttk.Label(root, text="Input Size:")
    input_size_label.grid(row=1, column=0, padx=10, pady=5)

    input_size_entry = ttk.Entry(root)
    input_size_entry.grid(row=1, column=1, padx=10, pady=5)

    run_button = ttk.Button(root, text="Run Algorithm(s)", command=run_algorithm)
    run_button.grid(row=2, column=0, columnspan=3, padx=10, pady=5)

    result_label = ttk.Label(root, text="", wraplength=400)
    result_label.grid(row=3,column=0, columnspan=3, padx=10, pady=(20, 10))  # Adjusted padding
    root.mainloop()
create_gui()
```

Experimental Results:

How do running times change with respect to data size:

Scenario 1:
Array size: 50

- Run time for Merge Sort: 0.00022429999080486596 seconds
- Run time for Heap Sort: 0.00019340001745149493 seconds
- Run time for Quick Sort: 0.00020760000916197896 seconds
- Run time for Quick Sort (Median of 3):) 0.00017900002421811223 seconds
- Run time for Insertion Sort: 0.00012229999992996454 seconds
- Run time for Selection Sort: 0.000155699992319569 seconds
- Run time for Bubble Sort: 0.00021130000823177397 seconds

As per the comparison of runtime, we can see that **Insertion Sort** takes less time and **Merge sort** took the most compared to other algorithms.

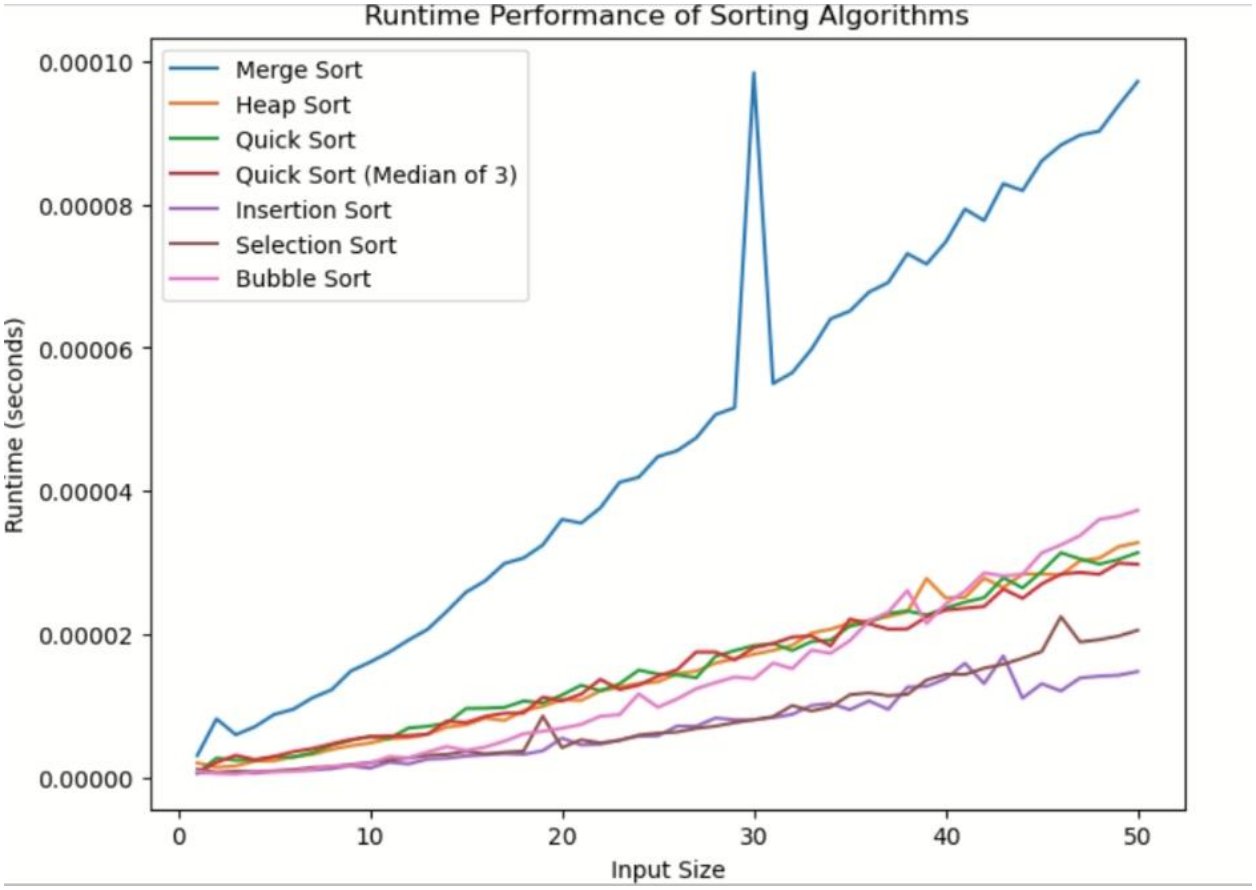

Figure 07: Run time change with respect to data size (input size: 50)

Scenario 2:

Array size: 500

- Run time for Merge Sort: 0.0021572000114247203 seconds
- Run time for Heap Sort: 0.0010028000106103718 seconds
- Run time for Quick Sort: 0.0005578999989666045 seconds
- Run time for Quick Sort (Median of 3): 0.0005329000123310834 seconds
- Run time for Insertion Sort: 0.0028432999970391393 seconds
- Run time for Selection Sort: 0.003472000011242926 seconds
- Run time for Bubble Sort: 0.0068113000015728176 seconds

As per the comparison of runtime, we can see that **Quick Sort (Median of 3)** takes less time and **Bubble Sort** took the most compared to other algorithms.

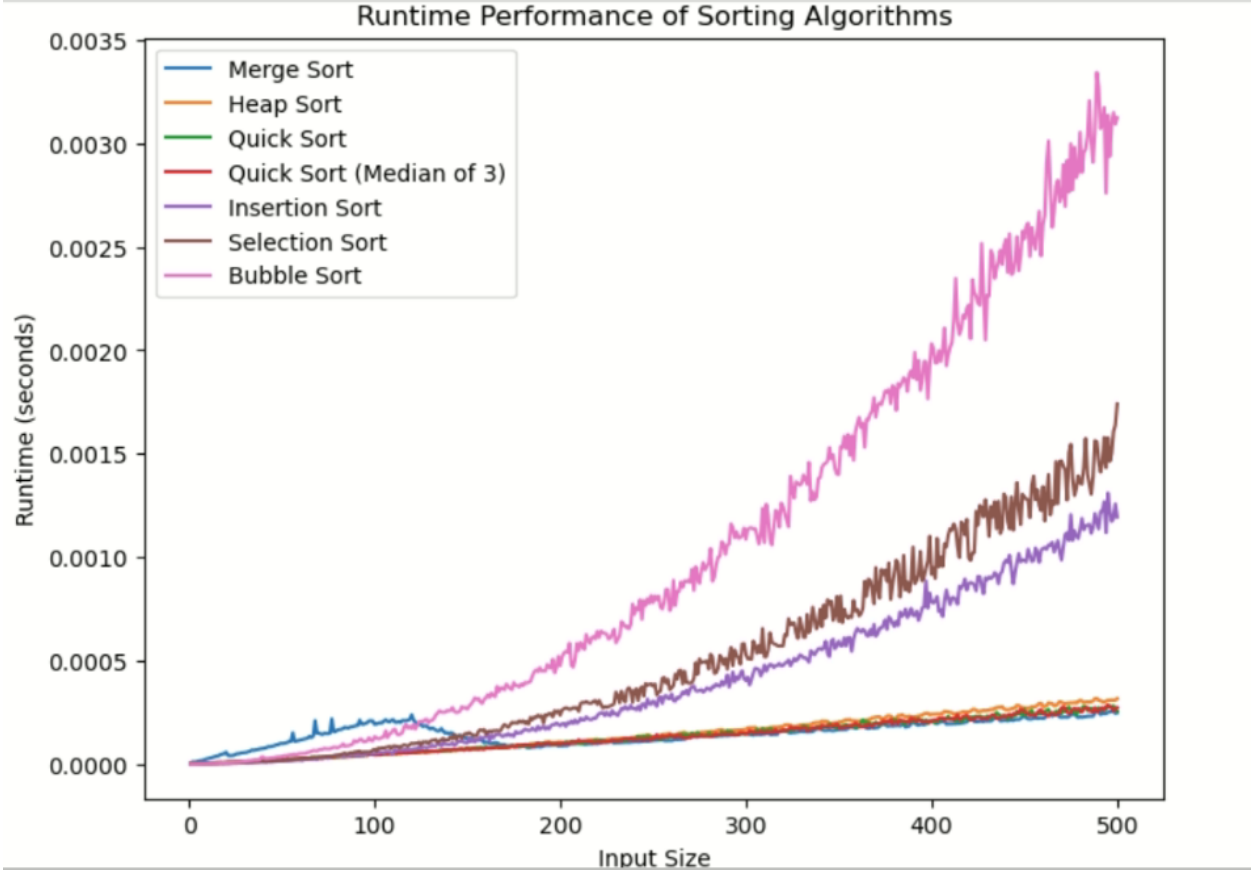

Figure 08: Run time change with respect to data size (input size: 500)

Scenario 3:

Array size: 1000

- Run time for Merge Sort: 0.0023363999789580703 seconds
- Run time for Heap Sort: 0.0030682999931741506 seconds
- Run time for Quick Sort: 0.0018467000045347959 seconds
- Run time for Quick Sort (Median of 3):) 0.0014140999992378056 seconds
- Run time for Insertion Sort: 0.013612500013550743 seconds
- Run time for Selection Sort: 0.014985899993916973 seconds
- Run time for Bubble Sort: 0.030032899987418205 seconds

As per the comparison of runtime, we can see that **Quick Sort (Median of 3)** takes less time and **Bubble Sort** took the most compared to other algorithms.

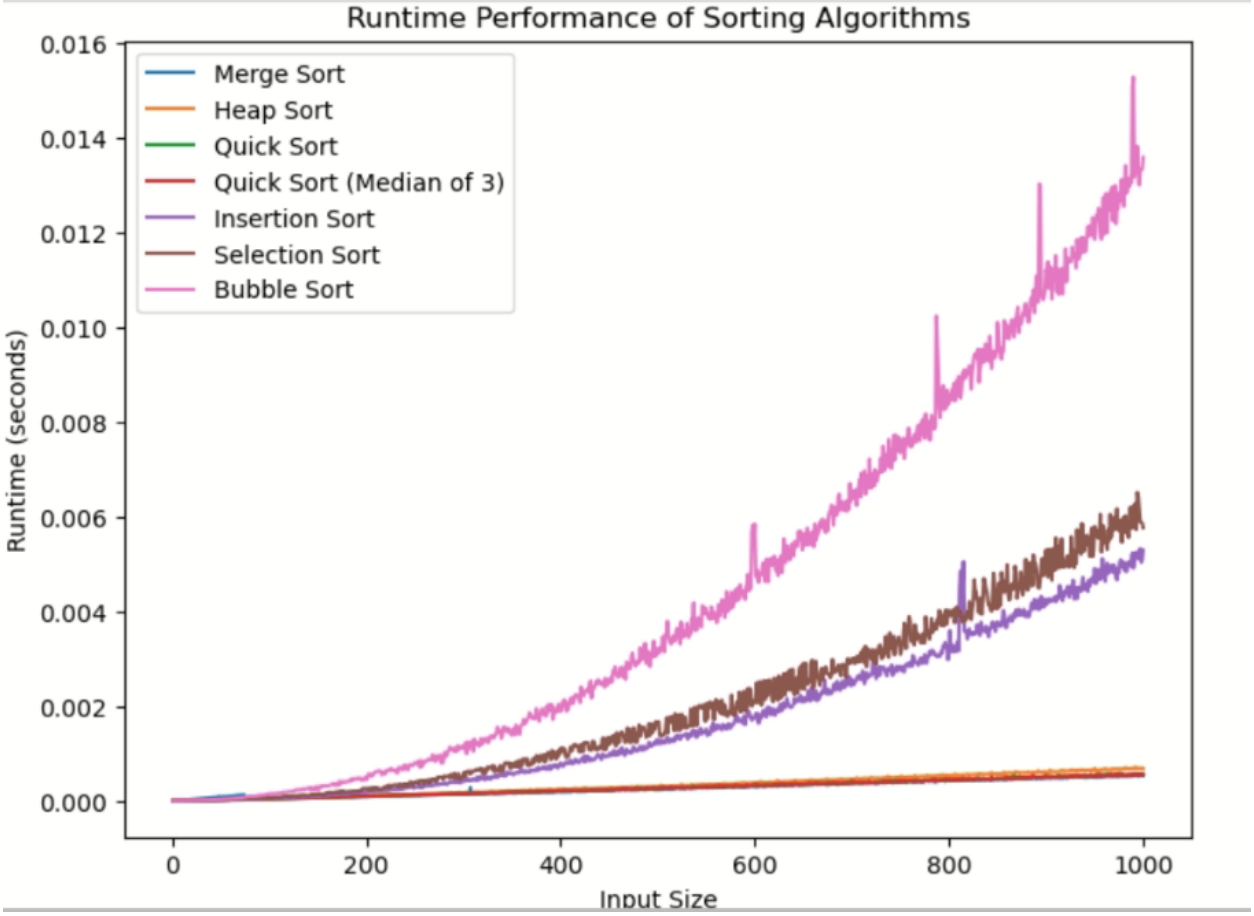

Figure 09: Run time change with respect to data size (input size: 1000)

Scenario 4:

Array size: 5000

- Run time for Merge Sort: 0.008025200018892065 seconds
- Run time for Heap Sort: 0.010327299998607486 seconds
- Run time for Quick Sort: 0.006610800017369911 seconds
- Run time for Quick Sort (Median of 3):) 0.006113299983553588 seconds
- Run time for Insertion Sort: 0.348651999986032 seconds
- Run time for Selection Sort: 0.36229250000906177 seconds
- Run time for Bubble Sort: 0.8395690000033937 seconds

As per the comparison of runtime, we can see that **Quick Sort (Median of 3)** takes less time and **Bubble Sort** took the most compared to other algorithms.

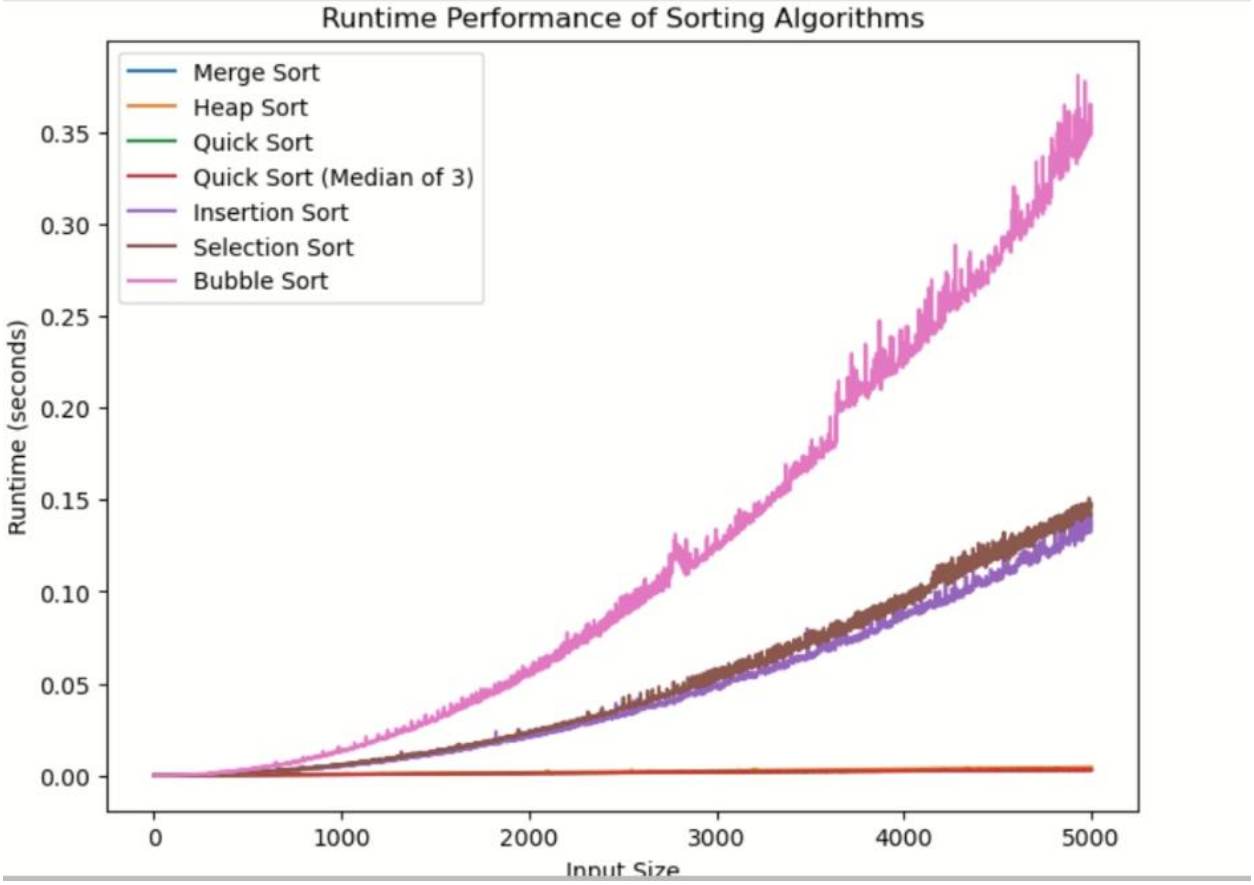

Figure 10: Run time change with respect to data size (input size: 5000)

Scenario 5:
Array size: 8000

- Run time for Merge Sort: 0.014695499994559214 seconds
- Run time for Heap Sort: 0.018527900014305487 seconds
- Run time for Quick Sort: 0.011994200001936406 seconds
- Run time for Quick Sort (Median of 3):) 0.010362100001657382 seconds
- Run time for Insertion Sort: 0.908546299993759 seconds
- Run time for Selection Sort: 0.9690126999921631 seconds
- Run time for Bubble Sort: 2.1715497000259347 seconds

As per the comparison of runtime, we can see that **Quick Sort (Median of 3)** takes less time and **Bubble Sort** took the most compared to other algorithms.

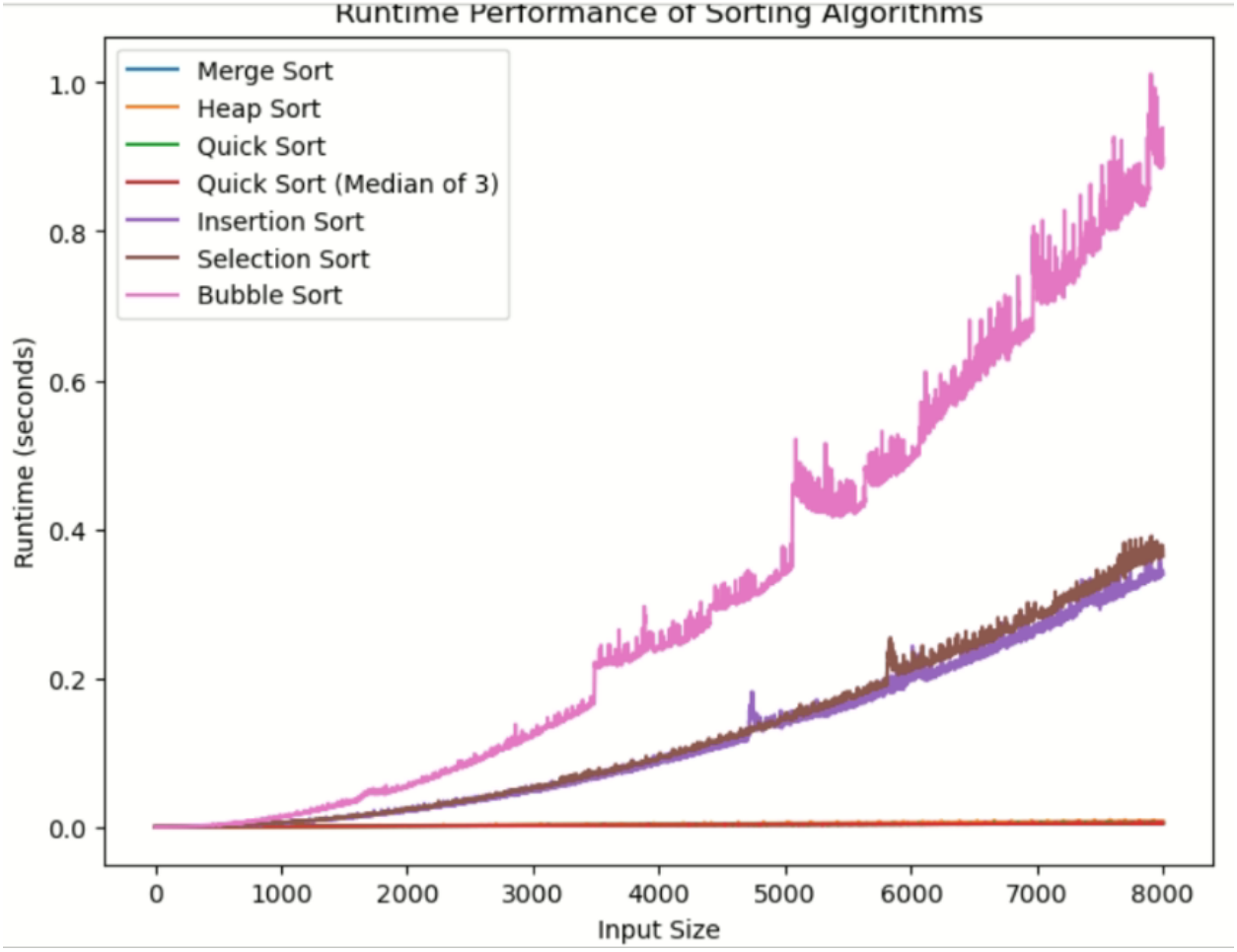

Figure 11: Run time change with respect to data size (input size: 8000)

Which one is better in terms of what conditions?

- Based on experimental results, if dealing with large amounts of data and needing a robust, fast algorithm, and can afford extra memory, **Mergesort** is excellent.
- **Quicksort**, especially with optimizations like the median of three, is often faster in practice.
- For limited memory environments, **Heapsort** offers good performance without additional space requirements.
- **Insertion Sort** is ideal for small or nearly sorted datasets due to its simplicity and efficiency under these conditions.
- **Selection Sort** and **Bubble Sort** are generally less efficient and best used for educational purposes or very small datasets.

Nahin Kumar Dey
UTA ID: 1002204142

Demonstration of the GUI:

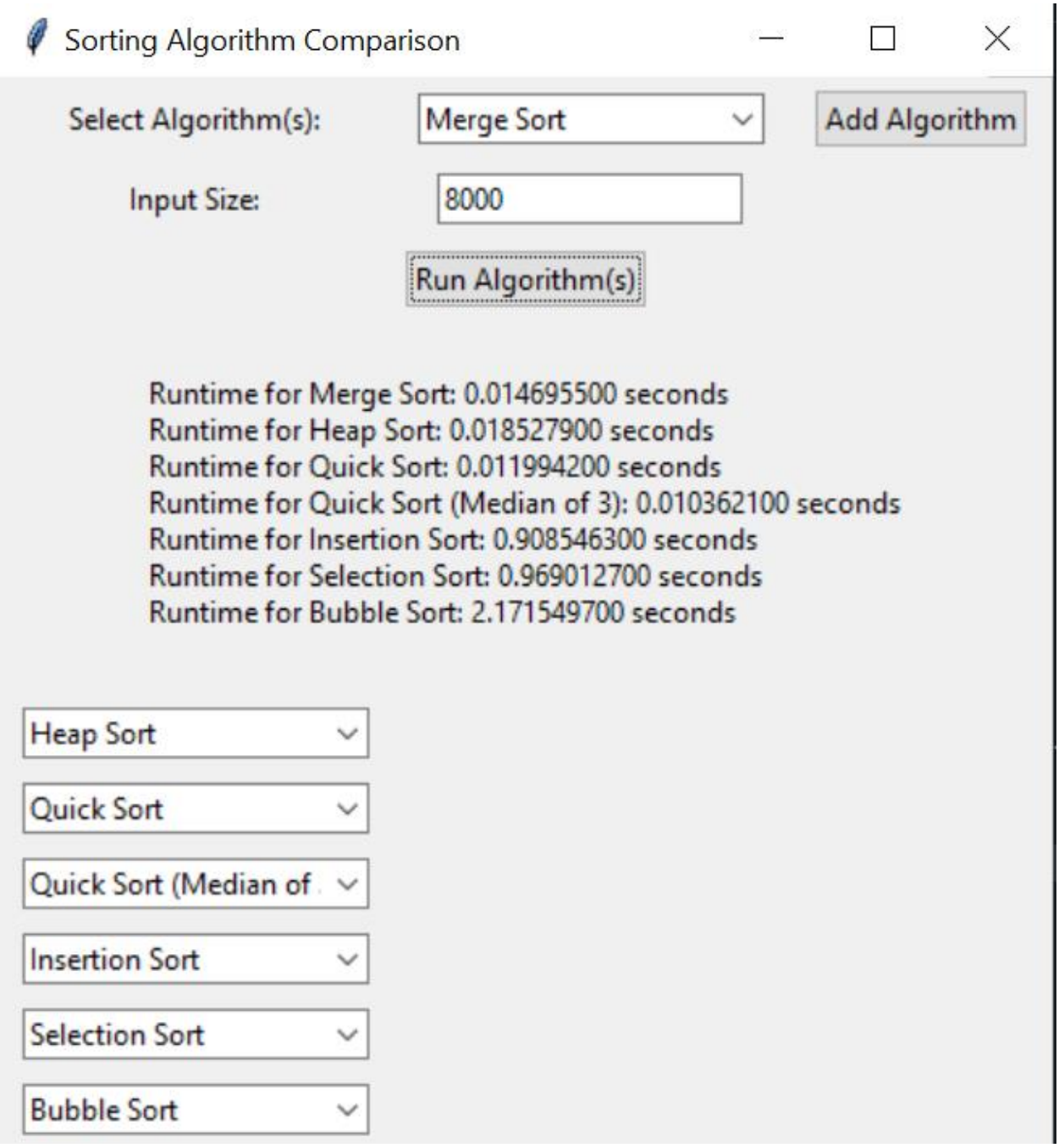

Figure 12: Screenshot of GUI implementation

Nahin Kumar Dey
UTA ID: 1002204142

References:

1. https://docs.python.org/3/library/tkinter.html
2. https://www.geeksforgeeks.org/python-gui-tkinter/
3. https://stackoverflow.com/questions/45441885/how-can-i-create-a-dropdown-menu-from-a-list-in-tkinter
4. https://www.geeksforgeeks.org/dropdown-menus-tkinter/
5. https://www.udacity.com/blog/2021/09/create-a-timer-in-python-step-by-step-guide.html
6. https://www.geeksforgeeks.org/time-perf_counter-function-in-python/
7. https://dev.to/chroline/visualizing-algorithm-runtimes-in-python-f92
8. https://medium.com/@maniksingh256/how-to-plot-the-performance-of-various-algorithms-in-python-6c420e42676e
9. https://www.pythonguis.com/tutorials/create-buttons-in-tkinter/
10. https://www.tutorialspoint.com/python/tk_button.htm