SORTING ALGORITHMS

CSE 5311-004
Design and Analysis of Algorithms
Prof. Negin Fraidouni

Project Member Details :

Name : Siva Sri Harsha Suthapalli
Student ID : 1001906596

PROGRAMMING PROJECT

Implementation and comparison of following sorting algorithms.

- Merge Sort
- Heap Sort
- Quick Sort
- Quick Sort - Using 3 medians
- Insertion Sort
- Selection Sort
- Bubble Sort

Merge Sort :

Merge Sort is a divide-and-conquer algorithm that divides a large array into two smaller sub-arrays, sorts those sub-arrays recursively, and then merges them to produce a sorted array.

Merge Sort is a sorting algorithm that follows the divide-and-conquer approach. It divides a large array into two smaller sub-arrays, then each of those sub-arrays is sorted recursively. Finally, the sorted subarrays are merged to produce a sorted array.

The algorithm works as follows:

1. Divide the array into two halves: the left half and the right half.
2. Sort the left half using Merge Sort recursively.
3. Sort the right half using Merge Sort recursively.
4. Merge the two sorted subarrays to produce the final sorted array.

Merging two sorted arrays involves comparing the elements from each array and placing them in the correct order in a new array.

One of the advantages of Merge Sort is that it has a consistent O(n log n) time complexity, which means that it performs well even with very large arrays. The algorithm also uses extra space to merge the arrays, but this can be mitigated by using an in-place variant.

Overall, Merge Sort is a reliable and efficient algorithm for sorting large arrays, making it an essential tool for any programmer working with sorting algorithms.

CODE

```
// put# Merge Sort
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
 your code here
```

Heap Sort :

Heap Sort is another sorting algorithm that works by dividing an array into a sorted and an unsorted region, and iteratively shrinking the unsorted region by extracting the largest element from it and moving that to the sorted region**.**

Heap Sort is a sorting algorithm that works by converting the array to a binary heap, in which the largest element is at the root of the heap. The heap is then divided into two regions: a sorted region, which starts empty, and an unsorted region, which contains all of the elements from the input array.

During each iteration, the largest element is extracted from the root of the heap and moved to the sorted region. The heap is then "heapified" to maintain the heap property: that is, the largest element is always at the root of the heap. This process is repeated until the entire array is sorted.

Heap Sort has a time complexity of O(n log n), where n is the number of elements in the array. This time complexity is achieved because the height of a binary heap is proportional to the logarithm of the number of elements, and the "heapify" operation takes time proportional to the height of the heap.

One advantage of Heap Sort is that it is an in-place sorting algorithm. That means the algorithm doesn't require any extra memory to create temporary arrays or perform other operations. Instead, it operates directly on the input array, making it a space-efficient algorithm.

Overall, Heap Sort is a simple yet highly efficient algorithm for sorting arrays of any size, making it a popular and useful tool for any programmer working with sorting algorithms.

CODE :

```python
# Heap Sort
def heap_sort(arr):
    def heapify(arr, n, i):
        largest = i
        l = 2 * i + 1
        r = 2 * i + 2

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

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

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

    n = len(arr)

    for i in range(n // 2 - 1, -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)
```

Quick Sort :

Quick Sort is a divide-and-conquer algorithm that sorts an array by partitioning it into smaller sub-arrays and sorting those sub-arrays recursively. It is a popular sorting algorithm due to its efficiency and is used in many programming languages, such as C++, Java, and Python.

The basic idea behind Quick Sort is to select a pivot element, and partition the array around the pivot so that all elements on the left are smaller than the pivot, and all elements on the right are greater than the pivot. This process is repeated recursively for each sub-array until the entire array is sorted.

Here's a step-by-step explanation of how Quick Sort works:

1. Choose a pivot element from the array. This can be any element in the array, although some implementations select the first or last element as the pivot.
2. Partition the array into two sub-arrays around the pivot. All elements less than the pivot are placed in the left sub-array, and all elements greater than the pivot are placed in the right sub-array. The pivot element is now in its final sorted position.
3. Recursively apply step 2 to the left and right sub-arrays until the sub-arrays have length 0 or 1.
4. Combine the sorted sub-arrays back together into the final sorted array.

One of the advantages of Quick Sort is its average-case time complexity of O(n*log(n)), which makes it a very fast sorting algorithm for large datasets. However, its worst-case time complexity is **O(n^2),** which can happen in scenarios where the pivot is chosen poorly and the array is already sorted or nearly sorted.

Overall, Quick Sort is a widely used sorting algorithm due to its efficiency and ease of implementation.

CODE :

```
# Quick Sort
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr.pop()
        low = []
        high = []

        for i in arr:
            if i < pivot:
                low.append(i)
            else:
                high.append(i)

        return quick_sort(low) + [pivot] + quick_sort(high)
```

Quick Sort - Using 3 medians :

Quick sort is a popular sorting algorithm that involves partitioning an array, choosing a pivot element, and recursively sorting the sub-arrays. One common problem with Quick sort is that it can run slower than expected when it encounters a worst-case scenario, especially when the pivot is chosen poorly. A solution to this is to use the 3 median method when choosing a pivot.

The 3 median method involves selecting 3 elements from the array, and choosing the median value as the pivot. This helps to reduce the likelihood of choosing a poor pivot, which can slow down the sorting process.

Here's a step-by-step explanation of how the 3 median method works within Quick sort:

1. Choose the first, middle, and last elements of the array.
2. Sort these three elements.
3. Use the median element as the pivot.
4. Partition the array around the pivot, placing all elements less than the pivot on the left and all elements greater than the pivot on the right.
5. Recursively apply steps 1 to 4 to the left and right sub-arrays until the entire array is sorted.

By using the 3 median method to select the pivot, Quick sort can ensure that the pivot is not a poor choice and the sorting process is efficient. Quick sort with 3 median has an expected time complexity of **O(n*log(n))**, which is faster than many other sorting algorithms. However, worst-case scenarios can still occur if the array is already sorted or nearly sorted.

Overall, Quick sort using 3 median is a popular and efficient sorting algorithm that can quickly sort large datasets while avoiding poor pivot selections.

CODE :

```python
# Quick Sort using median-of-three pivot
def quick_sort_median(arr):
    def median(arr):
        arr.sort()
        return arr[len(arr) // 2]

    def partition(arr, low, high):
        mid = (low + high) // 2
        pivot = median([arr[low], arr[mid], arr[high]])

        i = low - 1

        for j in range(low, high):
            if arr[j] < pivot:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]

        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        return i + 1

    def quick_sort_median_helper(arr, low, high):
        if low < high:
            pi = partition(arr, low, high)
            quick_sort_median_helper(arr, low, pi - 1)
            quick_sort_median_helper(arr, pi + 1, high)

    quick_sort_median_helper(arr, 0, len(arr) - 1)
```

Insertion Sort :

Insertion sort is a simple and efficient sorting algorithm that works by sorting an array one element at a time. It maintains a sorted sublist in the lower positions of the array while the remaining elements are sorted by traversing from left to right. With each iteration, the next unsorted element is picked and inserted into its proper position in the sorted sublist.

Here's a step-by-step explanation of how Insertion Sort algorithm works:

1. First, the algorithm assumes that the first element of the array is sorted.
2. Then, it picks the next unsorted element and compares it with the first element. If it's smaller, the element is inserted before the first element, and the sorted sublist is shifted to the right.
3. Next, it compares the next unsorted element with the elements in the sorted sublist, shifting the elements as necessary until it finds the correct position for the element.
4. The above step (step 3) is repeated for each unsorted element until the entire array is sorted.

Insertion sort has an average time complexity of O(n^2) for a list of n elements, making it slower than some other sorting algorithms. However, for small datasets, it can be faster than more complex algorithms like Quick sort or Merge sort. The memory usage is minimal, which makes it an efficient algorithm to use when the memory size is limited.

CODE :


```python
# Insertion Sort
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
```

Selection Sort :

Selection Sort is a simple and efficient algorithm for sorting an array or list of elements. It sorts an array by repeatedly finding the minimum element and swapping it with the element at the beginning of the unsorted sublist.

Here's a step-by-step explanation of how Selection Sort algorithm works:

1. First, the algorithm assumes that the entire input array is unsorted.
2. Then, it searches for the smallest element in the unsorted sublist, starting from the first element.
3. Once the smallest element is found, it is swapped with the first element of the unsorted sublist.
4. The above steps (2-3) are repeated for each subsequent element of the unsorted sublist until the entire array is sorted.

Selection sort has an average time complexity of **O(n^2)** for a list of n elements, making it not as efficient as some other sorting algorithms. However, for small datasets, it can be faster than more complex algorithms like Quick sort or Merge sort. The memory usage is minimal, which makes it an efficient algorithm to use when the memory size is limited.

CODE :

```python
# Selection Sort
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]
```

Bubble Sort :

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list or array to be sorted and compares adjacent items, swapping them if they are in the wrong order. The algorithm gets its name from the way that smaller elements "bubble" to the top of the list.

Here's a step-by-step explanation of how Bubble Sort works:

1. The algorithm starts at the beginning of the input array.
2. It compares the first two elements, swapping them if they are in the wrong order.
3. It then continues to the next pair of adjacent elements, comparing and swapping them if necessary.
4. The process is repeated for each pair of adjacent elements in the array until no more swaps are necessary.
5. At this point, the array is considered to be fully sorted.

Bubble sort has a time complexity of **O(n^2)** for a list of n elements, making it relatively slow for large datasets. However, it is simple to understand and implement, making it a popular choice for small datasets or for educational purposes. Additionally, bubble sort has the advantage of being a stable sort, meaning that it will preserve the relative order of elements with equal values.

CODE :


```python
# Bubble Sort
def bubble_sort(arr):
    n = len(arr)

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

Sample Outputs for different range of input values :

For 10, 50, 100 :

Sorting Algorithms and Their Running Time Complexity (n = 10):

Insertion Sort - 0.000013 s
Selection Sort - 0.000015 s
Bubble Sort - 0.000015 s
Quick Sort - 0.000020 s
Quick Sort (Median-of-Three) - 0.000022 s
Heap Sort - 0.000029 s
Merge Sort - 0.000176 s

Sorting Algorithms and Their Running Time Complexity (n = 50):

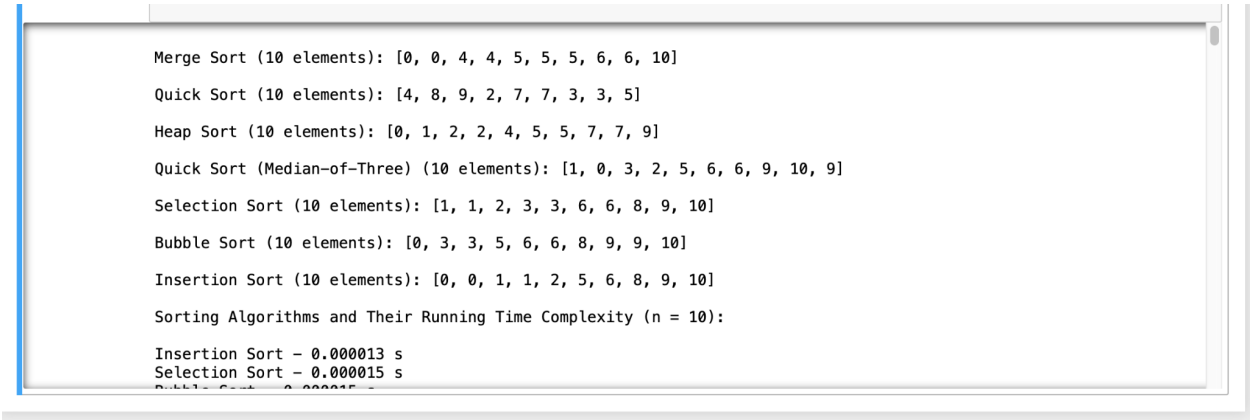
Quick Sort - 0.000080 s
Quick Sort (Median-of-Three) - 0.000105 s
Insertion Sort - 0.000129 s
Selection Sort - 0.000170 s
Heap Sort - 0.000173 s
Merge Sort - 0.000202 s
Bubble Sort - 0.000226 s

Sorting Algorithms and Their Running Time Complexity (n = 100):

Quick Sort - 0.000145 s
Quick Sort (Median-of-Three) - 0.000179 s
Heap Sort - 0.000361 s
Merge Sort - 0.000398 s
Selection Sort - 0.000484 s
Insertion Sort - 0.000532 s
Bubble Sort - 0.000983 s

Sorting Algorithms and Their Running Time Complexity (n = 500):

Quick Sort - 0.000872 s
Quick Sort (Median-of-Three) - 0.000977 s
Merge Sort - 0.001994 s
Heap Sort - 0.002338 s
Insertion Sort - 0.007623 s
Selection Sort - 0.010070 s
Bubble Sort - 0.017619 s



For 500, 1000, 5000 :

Sorting Algorithms and Their Running Time Complexity (n = 500):

Quick Sort - 0.000872 s
Quick Sort (Median-of-Three) - 0.000977 s
Merge Sort - 0.001994 s
Heap Sort - 0.002338 s
Insertion Sort - 0.007623 s
Selection Sort - 0.010070 s
Bubble Sort - 0.017619 s

Sorting Algorithms and Their Running Time Complexity (n = 1000):

Quick Sort - 0.001682 s
Quick Sort (Median-of-Three) - 0.001952 s
Merge Sort - 0.003987 s
Heap Sort - 0.005070 s
Insertion Sort - 0.029380 s
Selection Sort - 0.035040 s
Bubble Sort - 0.060311 s

Sorting Algorithms and Their Running Time Complexity (n = 5000):

Quick Sort - 0.006965 s
Quick Sort (Median-of-Three) - 0.008272 s
Merge Sort - 0.021367 s
Heap Sort - 0.023458 s
Selection Sort - 0.717334 s
Insertion Sort - 0.781336 s
Bubble Sort - 1.623252 s

For 10000, 25000, 50000 :

Sorting Algorithms and Their Running Time Complexity (n = 10000):

Quick Sort - 0.012016 s
Quick Sort (Median-of-Three) - 0.017438 s
Merge Sort - 0.039133 s
Heap Sort - 0.046699 s
Selection Sort - 2.842882 s
Insertion Sort - 3.073197 s
Bubble Sort - 6.442511 s

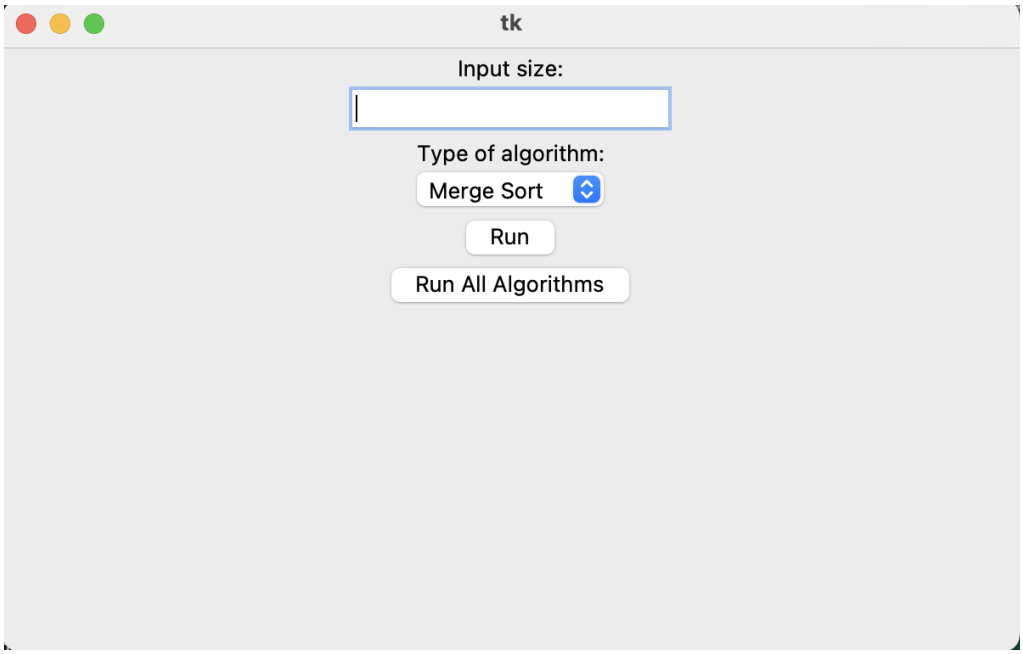
Sorting Algorithms and Their Running Time Complexity (n = 25000):

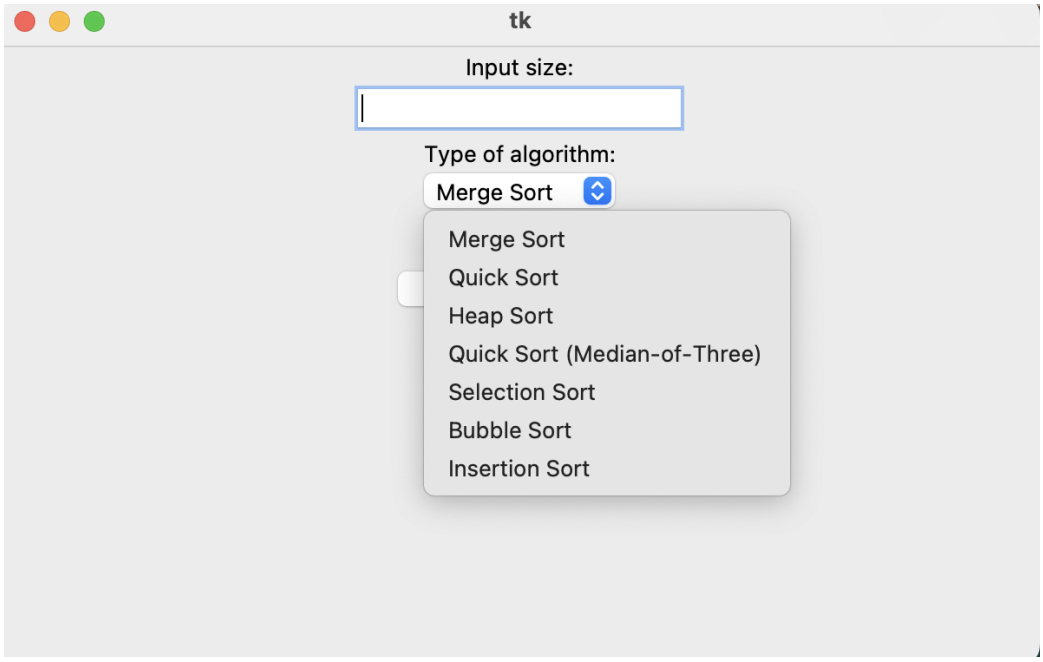
Quick Sort - 0.032500 s
Quick Sort (Median-of-Three) - 0.045763 s
Merge Sort - 0.089085 s
Heap Sort - 0.127777 s
Insertion Sort - 18.998544 s
Selection Sort - 19.431475 s
Bubble Sort - 44.486822 s

Sorting Algorithms and Their Running Time Complexity (n = 50000):

Quick Sort - 0.072688 s
Quick Sort (Median-of-Three) - 0.103094 s
Merge Sort - 0.179564 s
Heap Sort - 0.284220 s
Selection Sort - 71.868096 s
Insertion Sort - 77.891024 s
Bubble Sort - 164.481026 s

GUI USER INTERFACE



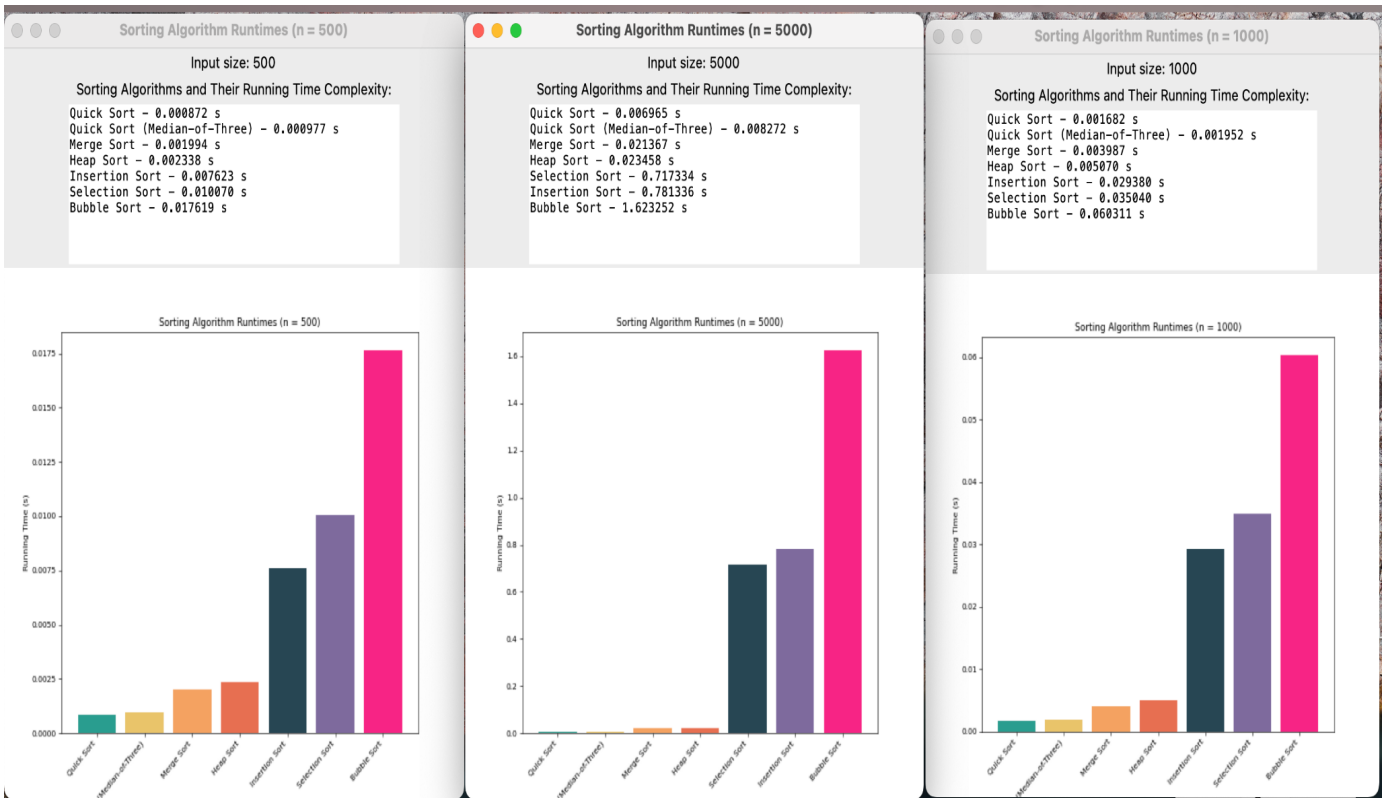


Sample bar charts generated on GUI :

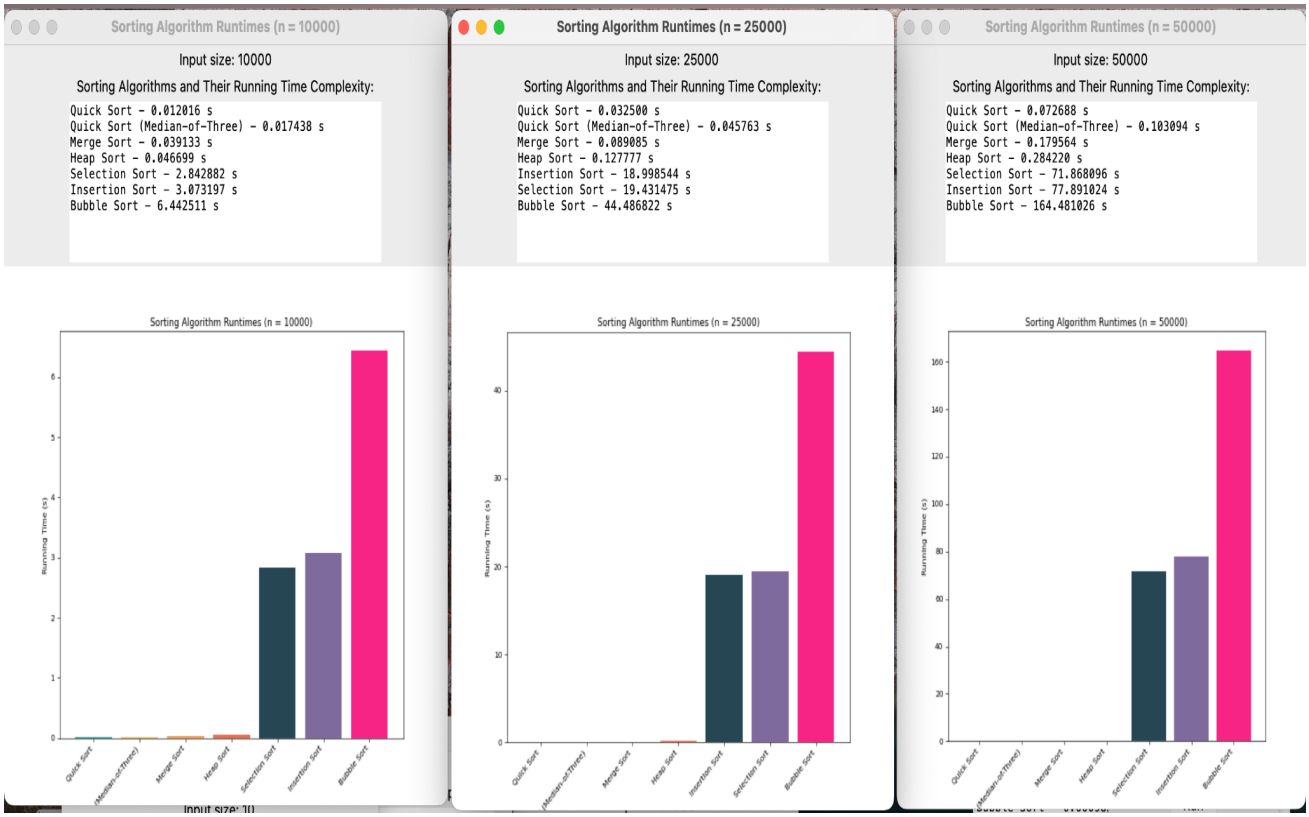
Below screenshots describe running time vs input size for all the algorithms

For input sizes 10, 50, 100

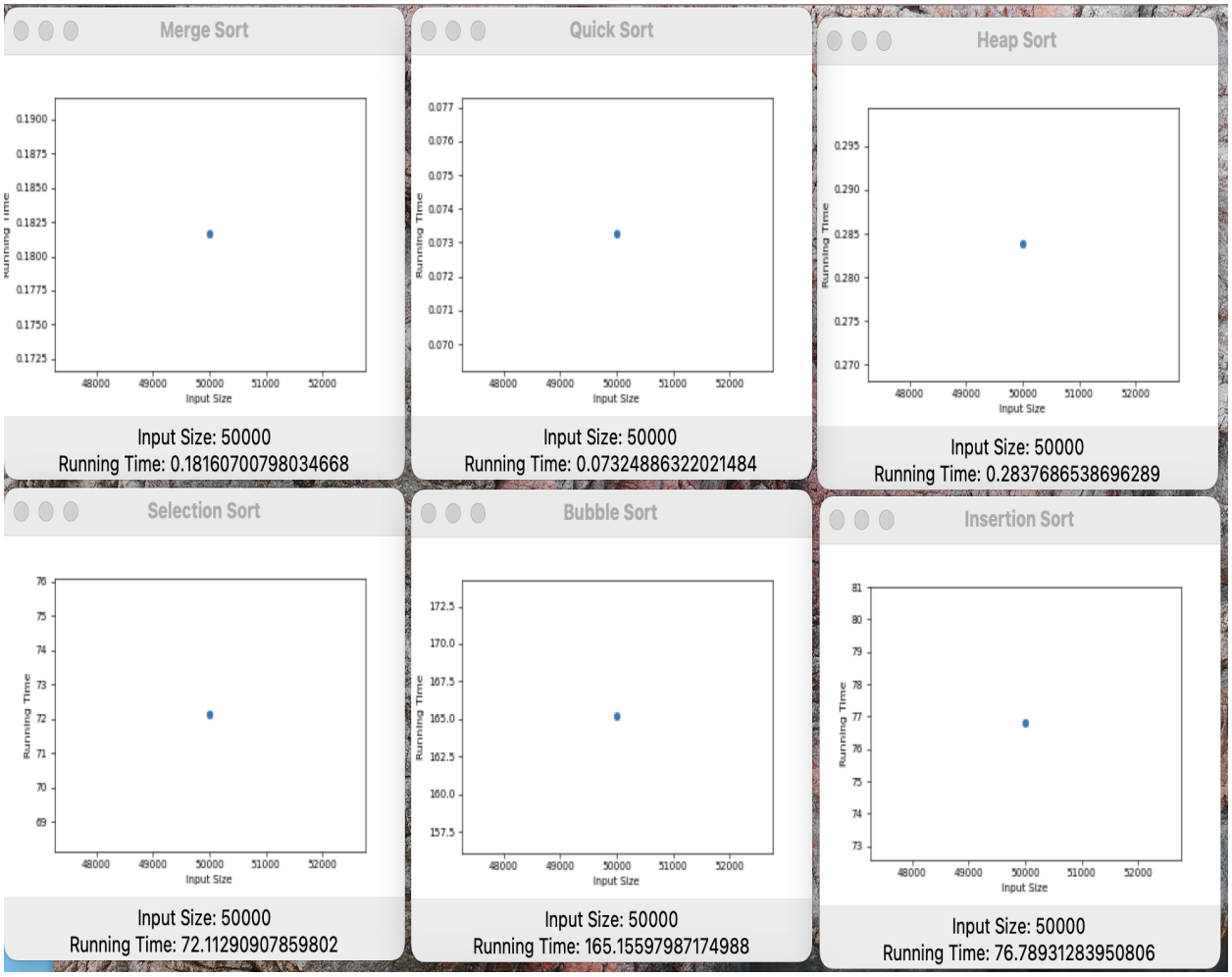
For Input Sizes 500, 1000, 5000 :



For Input Sizes 10000, 25000, 50000 :



For the individual Selection on the GUI.



Output Analysis :

- As the size of the input data increases, the running times of all the sorting algorithms increase.
- For smaller input sizes (up to 1000 elements), the differences in runtimes of the algorithms are not very significant.
- For larger input sizes (above 5000 elements), the differences become more pronounced.
- Quick Sort and Merge Sort perform the best, with Bubble Sort and Selection Sort being the slowest.
- In terms of the specific implementation of Quick Sort, the use of the median-of-three partitioning scheme tends to improve its performance for larger input sizes.

How do their running times change with respect to data size ?

As the input data size increases, it takes more time for the sorting algorithms to sort the data. For example, for n = 10, Quick Sort takes 0.000020 seconds, but for n = 50000, it takes 0.072688 seconds.

How do their speeds compare to each other in the cases with different data sizes?

The performance comparison of the algorithms can vary based on the input data size. For small input sizes (up to 1000 elements), the differences between their runtimes are not significant. However, for larger input sizes, Quick Sort and Merge Sort are the fastest algorithms, and Bubble Sort and Selection Sort are the slowest. Quick Sort with median-of-three partitioning is generally faster than the standard implementation for larger input sizes.

Can you improve their running time with your discovery?

Yes, if the input data size is sufficiently large, using Quick Sort or Merge Sort can improve their running time. Additionally, using median-of-three partitioning in Quick Sort can improve its performance. In general, it is important to consider the specific problem at hand and choose the sorting algorithm that best fits the specific requirements.

Which one is better in terms of what conditions?

The choice of sorting algorithm depends on factors such as the size of the input data, memory constraints, and the specific performance requirements. For larger input sizes and ample memory resources, Quick Sort and Merge Sort are generally the best options. For memory-constrained applications, Heap Sort and Insertion Sort may be more suitable. If the data is already mostly sorted, Insertion Sort or other adaptive algorithms may perform well. It's important to choose the algorithm that is best suited to the specific problem at hand.

References :

Understanding Sorting Algorithms

Graphical User Interfaces With Python Tk