DAT600: Algorithm Theory - Assignment 1

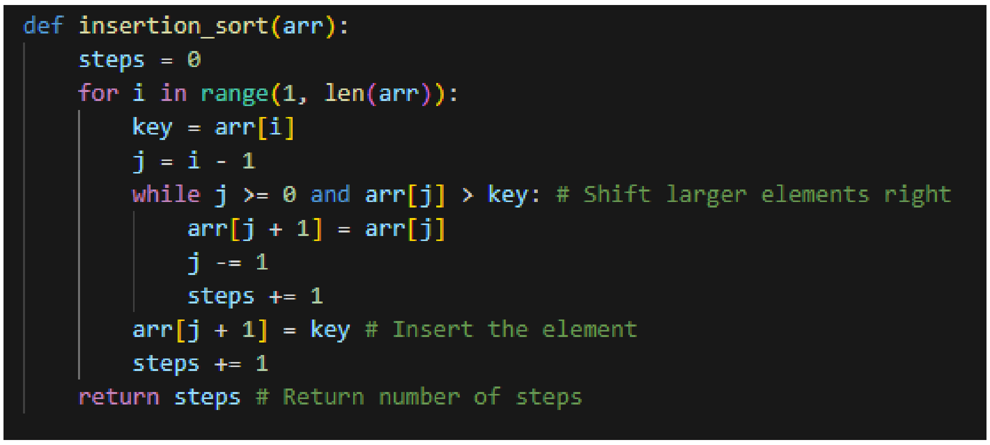
Antón Maestre Gómez - Daniel Linfon Ye Liu

Task 1 Counting the steps:

For this task, we have worked on an IPYNB file, programming code that allows us to implement the four algorithms.

Insertion-Sort

The **Insertion Sort** algorithm sorts an array by building a sorted portion step by step. It works by taking each element and inserting it into its correct position in the already sorted part of the array.



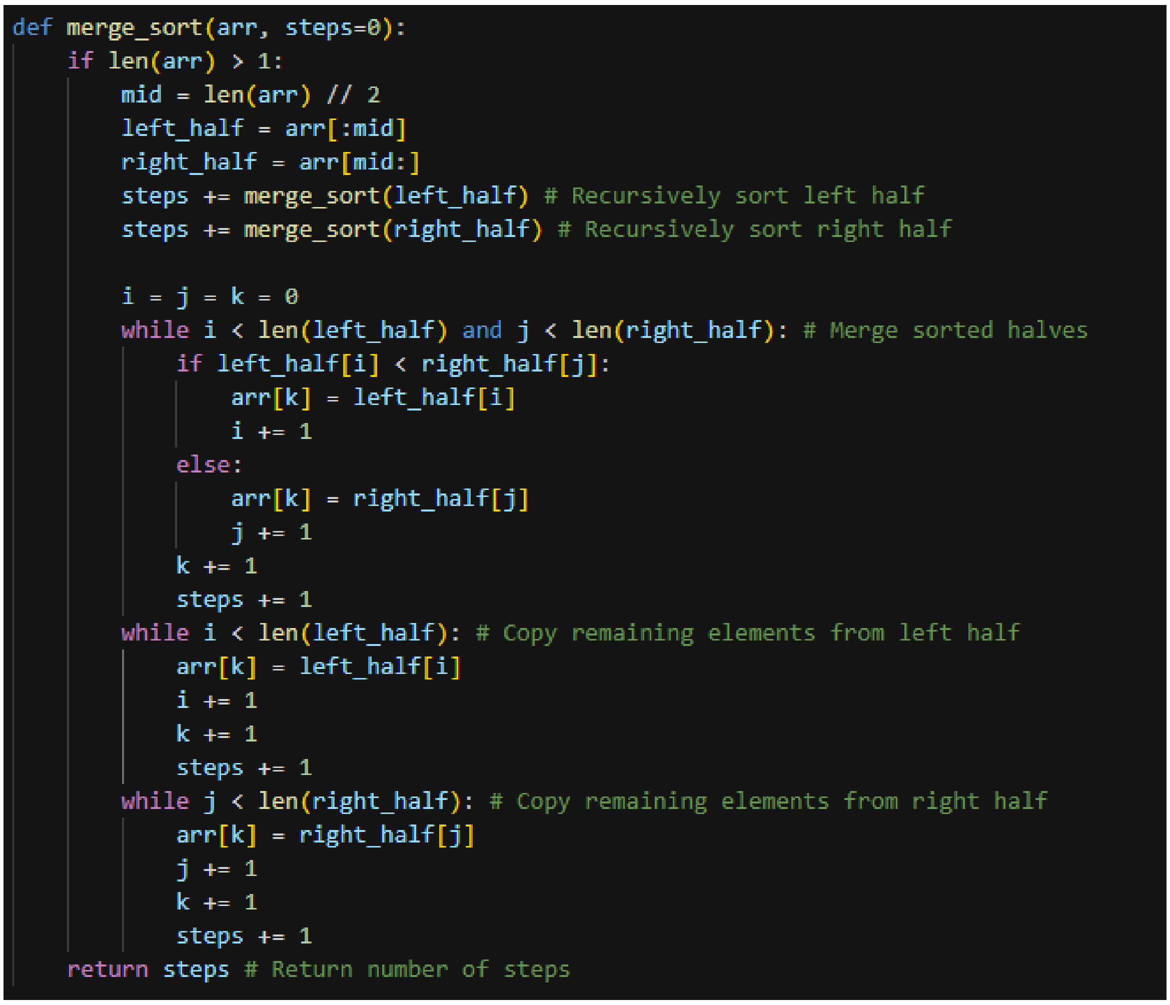
```python
def insertion_sort(arr):
    steps = 0
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key: # Shift larger elements right
            arr[j + 1] = arr[j]
            j -= 1
            steps += 1
        arr[j + 1] = key # Insert the element
        steps += 1
    return steps # Return number of steps
```

In this implementation, the algorithm starts from the **second element** (index 1), compares it with previous elements and **shifts larger ones** to the right. Then, it inserts the element (key) at its correct position. This process is repeated for all elements until the array is sorted.

This implementation also tracks the **number of operations performed** for complexity analysis.

Merge-Sort

The **Merge Sort** algorithm is a divide and conquer sorting technique. It works by recursively dividing the array into two halves, sorting each half separately, and then merging the sorted halves back together.



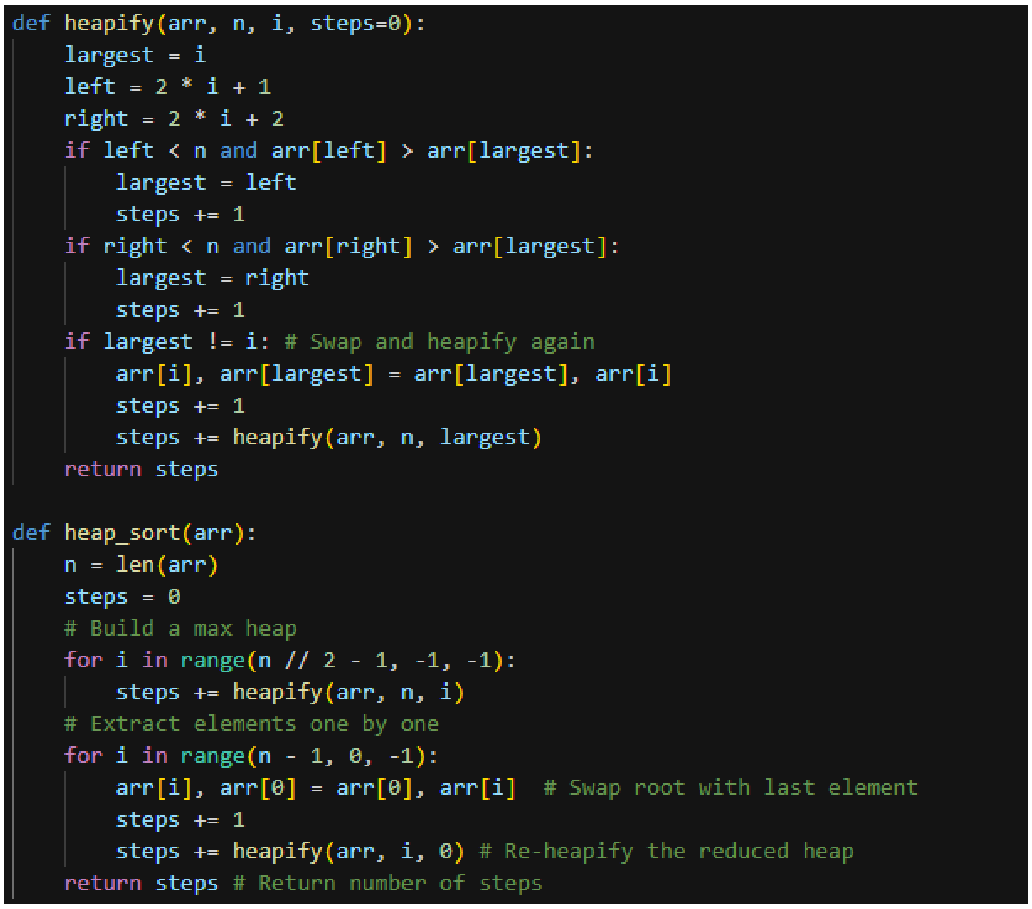
```python
def merge_sort(arr, steps=0):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]
        steps += merge_sort(left_half) # Recursively sort left half
        steps += merge_sort(right_half) # Recursively sort right half

        i = j = k = 0
        while i < len(left_half) and j < len(right_half): # Merge sorted halves
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1
            steps += 1
        while i < len(left_half): # Copy remaining elements from left half
            arr[k] = left_half[i]
            i += 1
            k += 1
            steps += 1
        while j < len(right_half): # Copy remaining elements from right half
            arr[k] = right_half[j]
            j += 1
            k += 1
            steps += 1
    return steps # Return number of steps
```

In this implementation, the algorithm **recursively splits** the array until each subarray has a single element. Then, it merges the subarrays by comparing elements from both halves and placing them in sorted order. This process continues until the entire array is sorted.

This implementation also tracks the **number of operations performed** for complexity analysis.

Heap-Sort

The **Heap Sort** algorithm is a comparison-based sorting technique that uses a binary heap structure. It works by first building a max heap, where the largest element is at the root, and then extracting the maximum element one by one, placing it at the end of the array.



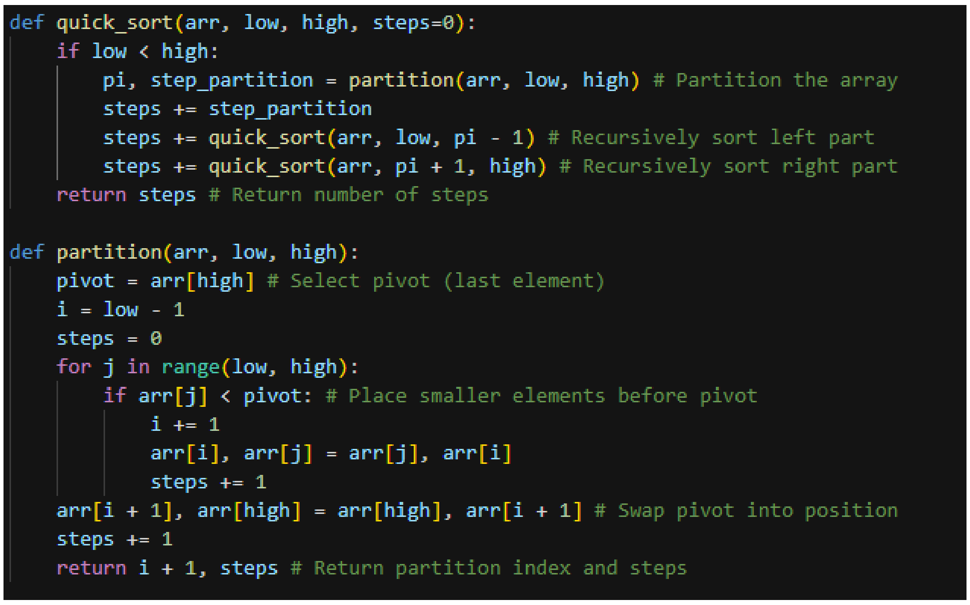
```python
def heapify(arr, n, i, steps=0):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
        steps += 1
    if right < n and arr[right] > arr[largest]:
        largest = right
        steps += 1
    if largest != i: # Swap and heapify again
        arr[i], arr[largest] = arr[largest], arr[i]
        steps += 1
        steps += heapify(arr, n, largest)
    return steps

def heap_sort(arr):
    n = len(arr)
    steps = 0
    # Build a max heap
    for i in range(n // 2 - 1, -1, -1):
        steps += heapify(arr, n, i)
    # Extract elements one by one
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Swap root with last element
        steps += 1
        steps += heapify(arr, i, 0) # Re-heapify the reduced heap
    return steps # Return number of steps
```

In this implementation, the algorithm **first heapifies** the array (converts the array into a max heap, largest element at the root). Then, it **repeatedly swaps** the root (largest element) with the last element and **re-heapifies** the reduced heap. The process continues until the entire array is sorted.

As the previous ones, this implementation also tracks the **number of operations performed** for complexity analysis.

Quicksort

The **Quick Sort** algorithm is a **divide and conquer** sorting technique. It works by selecting a **pivot element**, partitioning the array so that elements smaller than the pivot are on the left and larger ones are on the right, and then recursively sorting both halves.

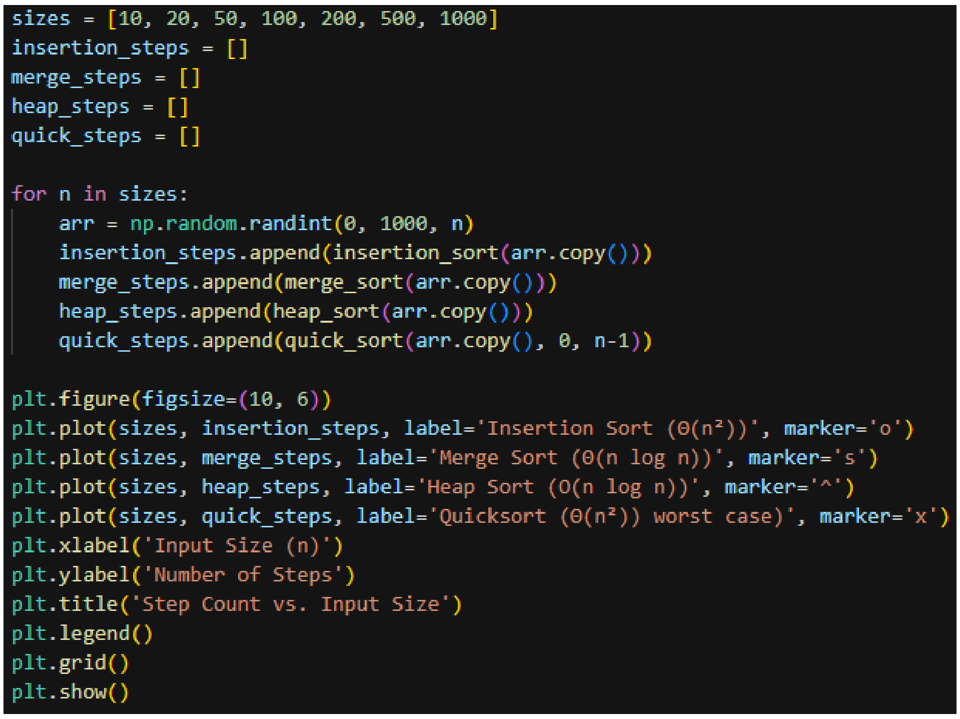


```python
def quick_sort(arr, low, high, steps=0):
    if low < high:
        pi, step_partition = partition(arr, low, high) # Partition the array
        steps += step_partition
        steps += quick_sort(arr, low, pi - 1) # Recursively sort left part
        steps += quick_sort(arr, pi + 1, high) # Recursively sort right part
    return steps # Return number of steps

def partition(arr, low, high):
    pivot = arr[high] # Select pivot (last element)
    i = low - 1
    steps = 0
    for j in range(low, high):
        if arr[j] < pivot: # Place smaller elements before pivot
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
            steps += 1
    arr[i + 1], arr[high] = arr[high], arr[i + 1] # Swap pivot into position
    steps += 1
    return i + 1, steps # Return partition index and steps
```

In this implementation, the algorithm **chooses the last element as the pivot**, partitions the array around it, and then recursively applies Quick Sort to the left and right subarrays.

The **number of operations performed** is also tracked in this implementation.

Analysis

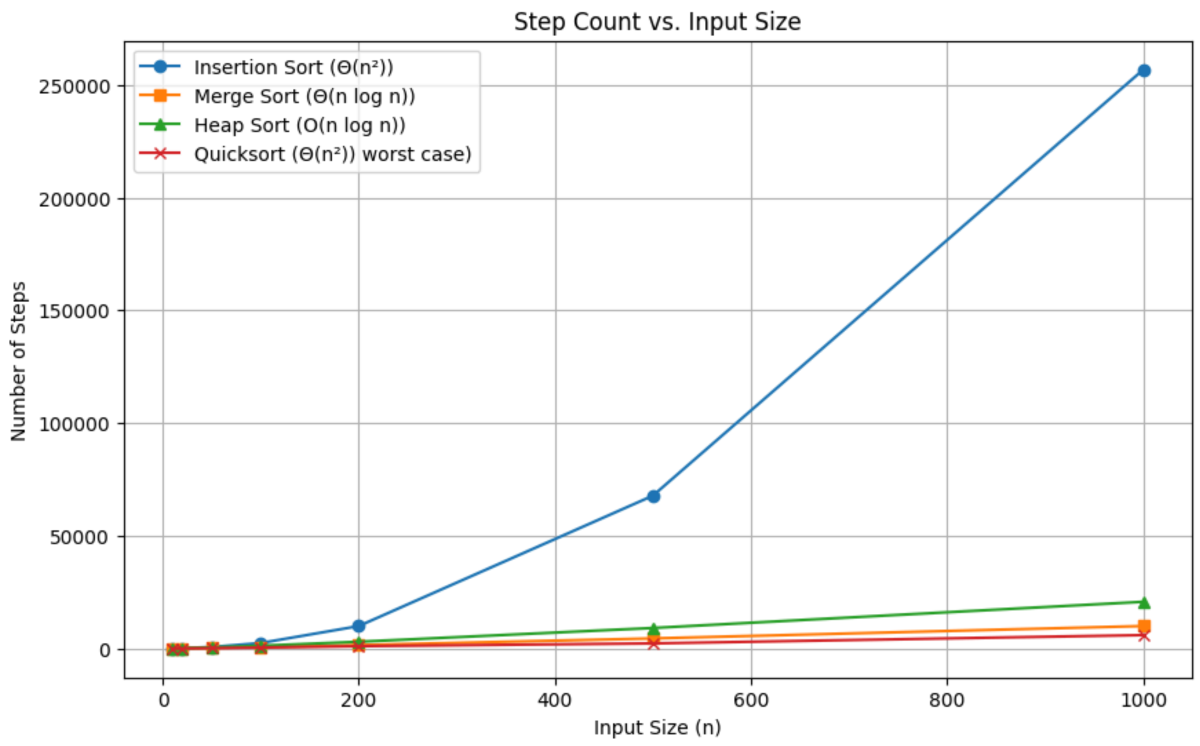


```python
sizes = [10, 20, 50, 100, 200, 500, 1000]
insertion_steps = []
merge_steps = []
heap_steps = []
quick_steps = []

for n in sizes:
    arr = np.random.randint(0, 1000, n)
    insertion_steps.append(insertion_sort(arr.copy()))
    merge_steps.append(merge_sort(arr.copy()))
    heap_steps.append(heap_sort(arr.copy()))
    quick_steps.append(quick_sort(arr.copy(), 0, n-1))

plt.figure(figsize=(10, 6))
plt.plot(sizes, insertion_steps, label='Insertion Sort (Θ(n²))', marker='o')
plt.plot(sizes, merge_steps, label='Merge Sort (Θ(n log n))', marker='s')
plt.plot(sizes, heap_steps, label='Heap Sort (O(n log n))', marker='^')
plt.plot(sizes, quick_steps, label='Quicksort (Θ(n²)) worst case)', marker='x')
plt.xlabel('Input Size (n)')
plt.ylabel('Number of Steps')
plt.title('Step Count vs. Input Size')
plt.legend()
plt.grid()
plt.show()
```

This analysis evaluates the **step count** of four sorting algorithms (**Insertion Sort, Merge Sort, Heap Sort, and Quick Sort**) for different input sizes. The input sizes tested are [10, 20, 50, 100, 200, 500, 1000]. For each size, a **random array** is generated using **np.random.randint(0, 1000, n)**, ensuring variability in the dataset.

Each sorting algorithm is applied to a **copy** of the array to prevent modifications from affecting other tests. The number of **steps performed** during execution is recorded, allowing for a fair comparison of algorithm efficiency. This step count represents the number of basic operations executed, providing insight into their empirical performance.

The recorded step counts are plotted against input sizes, with different markers and labels indicating the **expected asymptotic complexity**. **Insertion Sort** follows a quadratic growth pattern $\Theta(n^2)$, making it inefficient for large inputs. **Merge Sort and Heap Sort** show log-linear growth $\Theta(n \log n)$, confirming their efficiency over quadratic algorithms. **Quicksort** exhibits variability, as its worst-case complexity is $\Theta(n^2)$, though its average case is usually $O(n \log n)$.



This visualization helps validate the **theoretical complexities** by comparing **empirical step counts** with their expected growth rates.

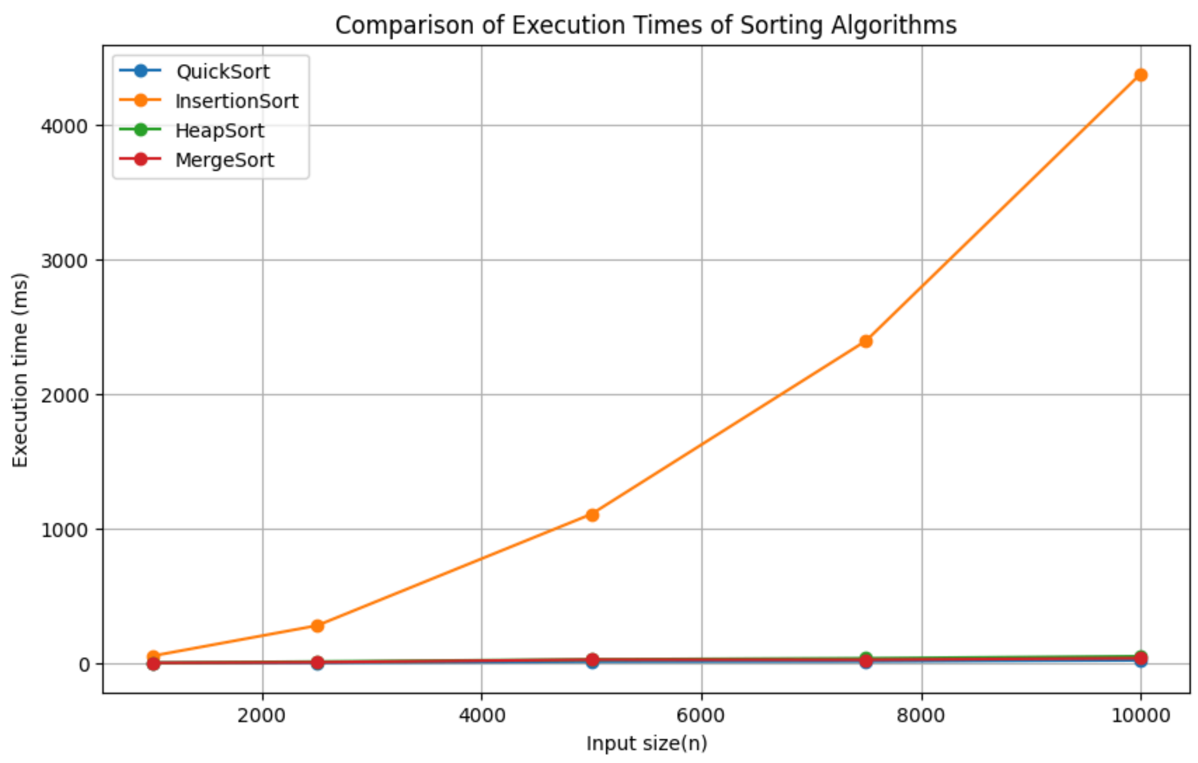
The results show how, as the input size increases, the inefficiency of **Insertion Sort** becomes more evident, and to a lesser extent, **Heap Sort** as well. Thus, the divide-and-conquer algorithms **Merge Sort** and **Quicksort** are the most efficient, with the latter being the least affected by the increase in input size.

Task 2 Compare true execution time

The execution times for sorting algorithms were compared between **Python** and **Java**, revealing significant performance differences. 3 executions of each algorithm have been carried out to obtain an average time:

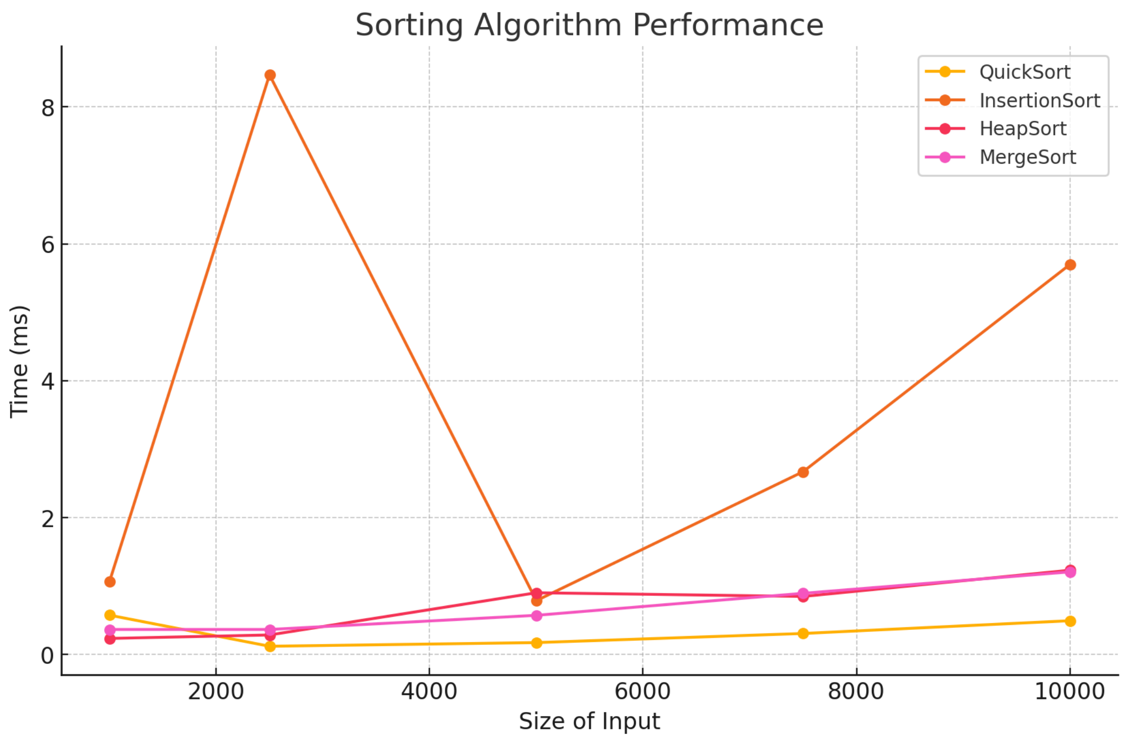
Python

Algorithm/Size(n)	1000	2500	5000	7500	10000
Quicksort	3.3246 ms	4.6546 ms	12.6429 ms	14.2951 ms	22.1150 ms
InsertionSort	54.3368 ms	279.9706 ms	1108.034 ms	2394.9713 ms	4375.8606 ms
HeapSort	3.3175 ms	12.6481 ms	28.8420 ms	36.9933 ms	52.3372 ms
MergeSort	3.6567 ms	7.6672 ms	27.2604 ms	26.0647 ms	37.7131 ms



Java

Algorithm/Size(n)	1000	2500	5000	7500	10000
Quicksort	0.5764 ms	0.1224 ms	0.1758 ms	0.3089 ms	0.4951 ms
InsertionSort	1.0631 ms	8.4674 ms	0.7884 ms	2.6719 ms	5.6977 ms
HeapSort	0.2364 ms	0.2873 ms	0.9036 ms	0.8498 ms	1.2328 ms
MergeSort	0.3674 ms	0.3673 ms	0.5741 ms	0.8954 ms	1.2087 ms



Analysis

1. QuickSort

- **Python:** The execution time for QuickSort increases with the input size, but it remains fairly fast, ranging from 3.32 ms for 1000 elements to 22.12 ms for 10000 elements.
- **Java**: In Java, QuickSort is significantly faster, with times ranging from 0.12 ms for 2500 elements to 0.50 ms for 10000 elements. The execution times in Java are much lower compared to Python

2. InsertionSort

- **Python**: InsertionSort shows much higher execution times, which is expected due to its $O(n^2)$ time complexity. For example, for 10000 elements, it takes 4375.86 ms, which is considerably slower than the other algorithms.
- **Java**: In Java, although InsertionSort is still slower than QuickSort, the times aren't as high as in Python. For 10000 elements, it takes 5.69 ms, showing a substantial difference in efficiency between the two languages for this algorithm.

3. HeapSort

- **Python**: HeapSort is relatively fast in Python, with times ranging from 3.32 ms for 1000 elements to 52.34 ms for 10000 elements. It's not as fast as QuickSort but still performs well.
- **Java**: In Java, HeapSort is very fast in comparison, with times ranging from 0.24 ms for 1000 elements to 1.23 ms for 10000 elements. Here, Java outperforms Python by a significant margin.

4. MergeSort

- **Python**: MergeSort has moderate execution times in Python, ranging from 3.66 ms for 1000 elements to 37.71 ms for 10000 elements. This behavior is quite consistent, with $O(nlogn)$ complexity.
- **Java**: In Java, MergeSort is also very fast, with times of 0.37 ms for 1000 elements and 1.21 ms for 10000 elements. Like HeapSort, MergeSort is faster in Java than in Python.

The differences in execution times are likely the result of the internal implementations and optimizations that each language performs. While both languages have their own efficiencies, **Java** shows a significant advantage in sorting algorithm performance in this case. This could be due to several factors, such as differences in the language implementations, memory management, and optimizations in the Java execution engine (JVM). If performance is crucial, especially for large datasets, Java is the better choice here.

Task 3 Basic proofs:

Proof 1: $(n + a)^b = \Theta(n^b)$

We want to prove that for any real constants a and b where $b > 0$, the following holds:

$$(n + a)^b = \Theta(n^b)$$

Using the Binomial Expansion, we get:

$$(n + a)^b = n^b(1 + a/n)^b$$

As $n \to \infty$, the term $(1 + a/n)^b$ approaches 1, implying that:

$$C_1 n^b \leq (n + a)^b \leq C_2 n^b$$

for some positive constants C_1, C_2, which confirms that $(n + a)^b = \Theta(n^b)$

Proof 2: $n^2/log\, n = o(n^2)$

We now want to prove that:

$$n^2/log\, n = o(n^2)$$

which means:

$$lim_{n \to \infty}[(n^2/log\, n)/n^2] = 0$$

so the proof is:

$$lim_{n \to \infty}[(n^2/log\, n)/n^2] = lim_{n \to \infty}[1/log\, n] = 0$$

since $log\, n \to \infty$ as $n \to \infty$. This shows that $n^2/log\, n$ grows strictly slower than n^2, meaning that $n^2/log\, n = o(n^2)$.

Proof 3: $n^2 \neq o(n^2)$

We need to prove that n^2 **is not** asymptotically smaller than n^2,

$$n^2 \neq o(n^2)$$

which means:

$$lim_{n\to\infty} n^2/n^2 \neq 0$$

as we can deduce:

$$lim_{n\to\infty} n^2/n^2 \ = 1 \neq 0$$

This shows that n^2 is **not** strictly smaller in growth than n^2, so $n^2 \neq o(n^2)$.

Task 4 Divide and Conquer Analysis:

Master theorem

$$T(n) \ = \ 3T(n/2) \ + \ \Theta(n)$$

$$\Downarrow$$

$$T(n) \ = \ aT(n/b) \ + \ f(n)$$

where:

- a = 3
- b = 2
- f(n) = Θ(n)

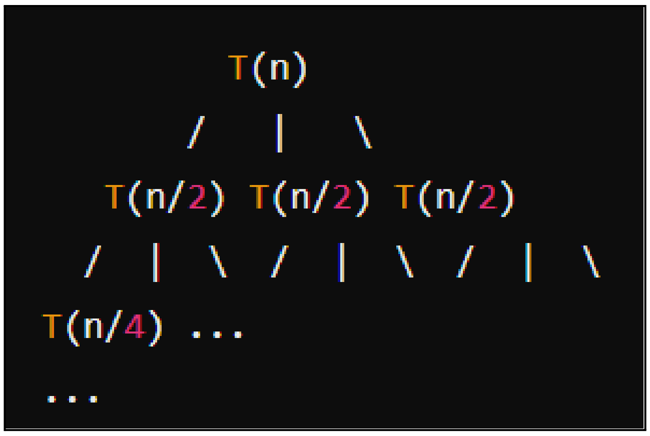
$IF\ f(n) \ = \ O(n^c)\ with\ c \ < \ log_b a\ \ then$:

$c \ = \ 1 \ < \ log_2 3 \ = 1.585 \Rightarrow$ **CASE 1**

$$T(n) \ = \ \Theta(n^{1.585})$$

Recursion tree method

$$T(n) \ = \ 3T(n/2) \ + \ \Theta(n)$$



```
        T(n)
      /   |   \
   T(n/2) T(n/2) T(n/2)
  /  |  \  /  |  \  /  |  \
T(n/4) ...
...
```

Recursion tree

Level 0: *O(n) = O(n)*

Level 1: *3 · O(n/2)= O(n)*

Level 2: *9 · O(n/4) = O(n)*

Each level continues adding up ***O(n).***

To determine the depth of the tree, the size of the subproblem is reduced as $n/2^k$, until $n/2^k = 1$. Solving $n = 2^k$, we obtain:

$$k = log_2 n$$

Each level has cost $O(n)$ and there are $O(log\, n)$ levels, so the complexity is:

$$T(n) = \Theta(n^{log_2 3})$$

$$\Downarrow$$

$$T(n) = \Theta(n^{1.585})$$

GitHub Files

https://github.com/AntonMG4/DAT600_AlgorithmTheory/tree/main/Assignment1