Compulsory 2

Link to GitHub repository: https://github.com/Zartok89/Complulsory_2

About the chosen sorting algorithm and how they work

Selection Sort: "Selection sort is a simple comparison-based sorting algorithm. The main idea behind the algorithm is to divide the input list into two parts: a sorted part and an unsorted part. Initially, the sorted part is empty, and the unsorted part contains all the elements. The algorithm repeatedly selects the smallest (or largest, depending on the sorting order) element from the unsorted part and swaps it with the leftmost unsorted element, moving the boundary between the two parts one element to the right."

1. Find the Minimum: Scan through the unsorted portion of the list and find the smallest element.
2. Swap: Swap the smallest element found in the unsorted part with the first element in the unsorted part.
3. Move the Boundary: Increment the boundary that separates the sorted and unsorted parts.
4. Repeat: Continue the process until the entire list is sorted.

Merge Sort: "Merge Sort is another divide-and-conquer algorithm used for sorting an array or a list of elements. The main idea behind Merge Sort is to divide the unsorted list into n sublists, each containing one element (since a list of one element is considered sorted), and then repeatedly merge sublists to produce new sorted sublists until there's only one sublist remaining."

1. Divide: Split the unsorted list into two halves. This is the divide part.
2. Conquer (Recursive Sort): Recursively sort both halves.
3. Merge (combine) them to produce a single sorted list.
4. Base Case: If the array has only one element, return it (since it's already sorted).

Quick Sort: "Quick Sort is a divide-and-conquer algorithm for sorting an array or a list of elements. 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 sorted recursively."

1. Choose a Pivot: The choice of the pivot selection and partitioning scheme significantly affects the algorithm's performance. Common methods include picking the first element, the middle element, or using a random element. More advanced methods might use the median of the array.
2. Partitioning: Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it. After this partitioning, the pivot is in its final sorted position.
3. Recursive Sort: Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.
4. Base Case: If the array has 1 or 0 elements, return (an array with a single element is always sorted).

Advantages of chosen sorting methods and its Time and Space Complexity:

Selection Sort: One of its advantage lies within its simplicity. It is made for sorting small lists where its simplicity and ease of implementation outweigh its inefficiency.

- Sorts the list by making changes directly to the input list and does not require any additional memory.

Time Complexity: Best Case: O(n^2) Even if the list is already sorted, the algorithm still goes through all the elements for each element. Average Case: O(n^2). Worst Case: O(n^2). For all cases, Selection Sort takes O(n^2) time due to its nested loops. It performs the same number of comparisons regardless of the initial ordering of the input.

Merge Sort: Merge Sort uses more memory compared to algorithms like Quick Sort, it has a consistent performance and is stable, which can make it preferable in some situations.

- It ensures a consistent O(n log n) performance regardless of the initial ordering of the input, which can be an advantage over algorithms like Quick Sort that can degrade to O(n^2) in certain scenarios.
- It is particularly good for situations where the data being sorted doesn't fit into memory. The merge operation can be effectively performed on chunks of data at a time, which makes it suitable for sorting datasets that are larger than the available RAM.
- It is a stable sort, which means the relative order of equal sort items is preserved.

Time Complexity: Best , Average, and Worst Case: O(n log n). For all cases, Merge Sort takes O(n log n) time. The "divide" step takes O(n log n) time (as the array is divided in half each time) and the "merge" step operates in O(n) time (since each item in the array will, over the course of the algorithm, be looked at once when merging). Multiplying these together gives the total time complexity.

Quick Sort: One of the advantages of Quick Sort is that it can be implemented to sort "in-place", meaning it sorts the array by making changes to the input array itself and does not need to use much additional memory.

- Despite its worst-case time complexity, Quick Sort is often faster in practice than other O(n log n) algorithms like Merge Sort or Heap Sort due to its smaller hidden constants and good cache performance.

Time Complexity: Best and Average Case: O(n log n) In the average case, the algorithm divides the array into two roughly equal pieces. Worst Case: O(n^2) This can occur when the pivot is the smallest or largest element in the array, leading to an unbalanced partition. However, the worst case can be largely avoided with a good choice of the pivot. Worst-case can be mitigated: By using techniques like "median-of-three" pivot selection (where the pivot is the median of the first, middle, and last element of the array), or randomized pivot selection.

Amount of time each of the algorithms used with int = 10, 100, 1000, 10000;

Algorithm	Int Amount and Time Taken (milliseconds)
Selection Sort	Int = 10, Time Taken = 0 ms Int = 100, Time Taken = 0 ms Int = 1000, Time Taken = 2 ms Int = 10000, Time Taken = 231 ms
Merge Sort	Int = 10, Time Taken = 0 ms Int = 100, Time Taken = 0 ms Int = 1000, Time Taken = 1 ms Int = 10000, Time Taken = 13 ms
Quick Sort	Int = 10, Time Taken = 0 ms Int = 100, Time Taken = 0 ms Int = 1000, Time Taken = 0 ms Int = 10000, Time Taken = 2 ms