Assignment 1 - Sorting

S1_A1_8

Code Explanation

Under each code there are some dogstrings where we have explained what each code does and all the explanations needed.
However, we also explained in the document some part of the code with its complexities.

Bubble Sort

- **Time Complexity:**
  - **Best case: O(n)**
  - **Worst case: $O(n^2)$**
- **Space Complexity: O(n)**

The function implements the bubble sort algorithm, which repeatedly traverses the array, compares adjacent elements, and swaps them if they are in the wrong order. The goal is to sort the array by "bubbling" larger elements to the end of the array and smaller elements to the beginning.

Regarding complexity, in the best-case scenario, the array is already sorted. During the first pass through the array, no swaps are necessary, and the loop terminates after the first traversal. As the algorithm checks each element once, the best-case time complexity is O(n), where n is the number of elements in the array.

In the worst-case scenario, the array is sorted in reverse order. The algorithm will perform multiple passes through the array, making a full round of comparisons and swaps for each element. This results in a worst-case time complexity of $O(n^2)$ because the number of comparisons grows quadratically with the size of the array.

When it comes to space complexity, this depends on the size of the input array, which is a copy. It needs a space complexity of O(n) to be stored. In addition, the variables current and sorted need a constant space and we can conclude that the space complexity is O(n).

Insertionsort

- **Time Complexity:**
  - **Best case: O(n)**
  - **Worst case: O(n^2)**
- **Space Complexity: O(n)**

If the array is already sorted (best-case), the “while” loop does not modify the position of any element (it runs only one time for every “i”). Consequently, n-1 comparisons are made and the best case time complexity is O(n).

In the worst case, the array is reversed, and each element needs to be shifted. The “while” loop runs “i” times, making (n(n-1))/2 comparisons. Then, the worst-case time complexity is O(n^2).

The algorithm uses a constant amount of space for variables, but making a copy of the array takes O(n) space. So, the space complexity is O(n).

Quicksort

- **Time Complexity:**
  - **Best case: O(n/log n)**
  - **Worst case: O(n^2)**
- **Space Complexity:**
  - **Best case: O(log n)**
  - **Worst case: O(n)**

The partition function of quicksort rearranges elements around the pivot, and the quicksort recursively performs the same to subarrays.

Given the best case, the array is divided into two parts of the same length. Each recursive call processes the half of the array, so the recursion depth is of log n. At the same time, the partitioning program makes n comparisons at each level. In consequence, the best-case time complexity is of O(n/logn).

In contrast, in the worst case the pivot provokes unequal partitions. In this case, the algorithm can be compared to a sorting one that takes quadratic time and makes n recursive calls with n-1, n-2, …, 1 comparisons. The worst-case time complexity is of O(n^2).

For recursion, the stack is used. The recursion depth is of O(n), being this the space complexity. Despite this, if the partitions are equal, the recursion depth and hence the space complexity is log n. Also, the partitioning in quicksort swaps elements in the input array, so no more arrays are necessary and hence no extra space.

Sorting Algorithm

- **Time Complexity:**
  - **Best case: O(n)**
  - **Worst case: O(n^2)**
- **Space Complexity: O(n)**

If the input array is sorted, there will be only one iteration through it with n-1 comparisons and no other action. The sorted flag will be set to “true”. So, the best-case time complexity is O(n).
Nevertheless, the worst case that could take place would be when the array is reversed. The algorithm would start making n-1 comparisons, then n-2, and like this until there is only one comparison finally. In total, the comparisons made would be (n(n-1))/2, making the worst-case time complexity O(n^2).
Because the function creates a copy of the initial array, something that takes O(n) memory, the space complexity is O(n). Take into account that the other variables (sorted, current, loop..) are constant (O(1)).

Linear search

- **Time Complexity:**
  - **Best case: O(1)**
  - **Worst case: O(n)**

  the algorithm performs n comparisons. Linear complexity
- **Space Complexity: O(1)**

  Constant complexity

The function of linear search is the traversal of an array with the aim of finding a concrete element (x). Like this, it provides the index of the first occurrence of the element or -1 in the case it is not found.
To evaluate the time complexity, we should keep in mind the position of the element x in the array. On the one hand, the best case would take place if the element is detected in the first comparison, being then time complexity O(1). On the other hand, the worst case happens when x is not in the array or it is at the last position. To know that, the loop has to iterate through all the elements in this array (as many times as number of elements that are present in the array). The loop would have to iterate n times, making the worst-case time complexity O(n).
When it comes to space complexity, it is O(1). This is due to the fact that the program needs a constant amount of space, no matter the size of the input, and because the other variables used (index, elem and i) are scalars.

Binary search

- **Time Complexity:**
  - **Best case: O(1)**
  - **Worst case: O(log n)**
- **Space Complexity: O(1)**

Binary search makes the search space the half each time, so the best-case time complexity is when the element (x) is in the middle of the array already and so the search does not keep on going. In this case, the time complexity is of O(1).

The worst case would be when the array constantly has to be divided in half until arriving to an array of an only one element. Every time a halving occurs, the number of elements to iterate through are the half. Because of this, the search space goes from n to n/2, n/4 and successively. This way, the worst-case time complexity is O(log n).

Binary Search Recursive

- **Time Complexity: O(log n)**

  The same as Binary Search
- **Space Complexity: O(log n)**

  Now the space complexity is not constant like in the iterative version, but O(log2 n) due to the additional space required by the recursive function calls.

Hanoi Tower

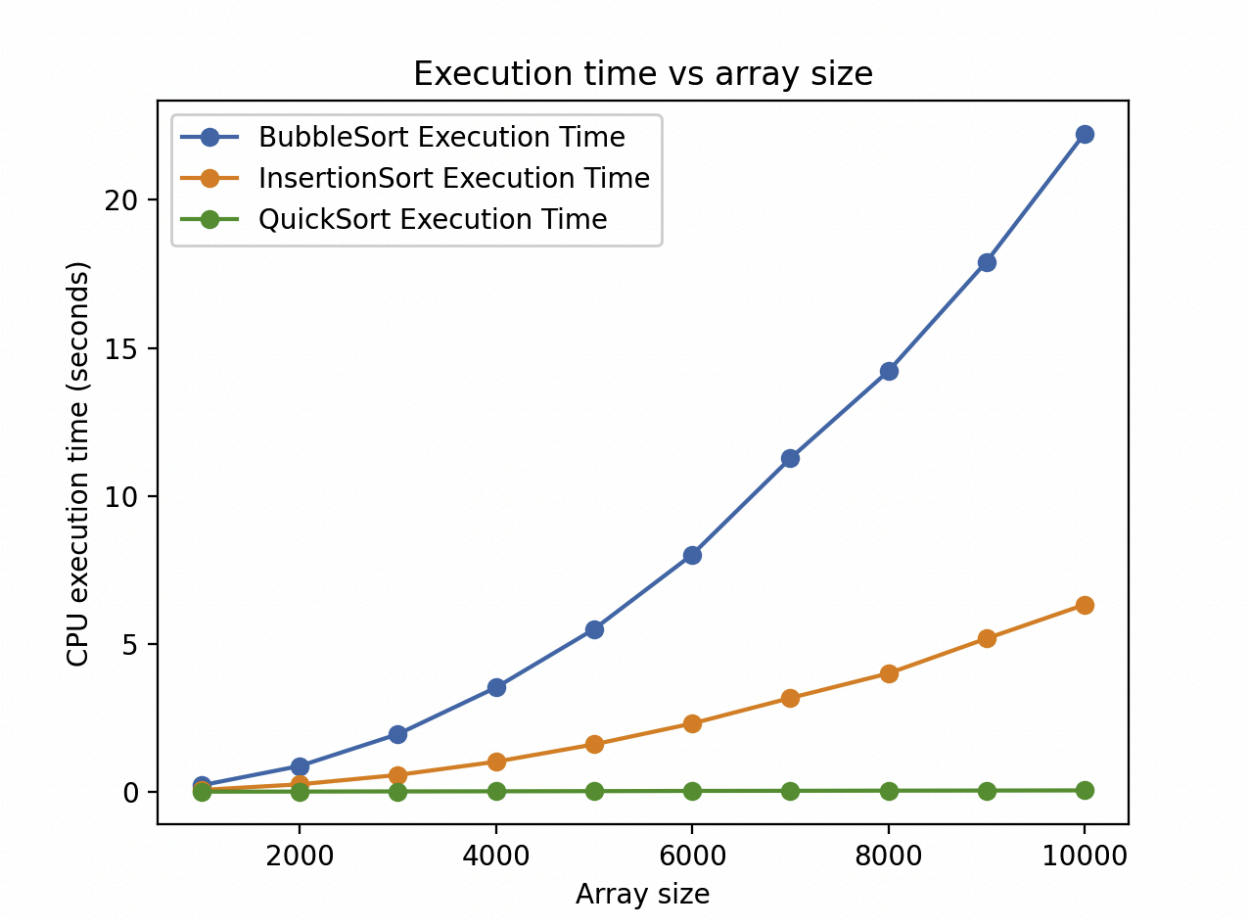
- **Time Complexity: O(2^n).**
- **Space Complexity: O(n)**

To solve the Hanoi Tower recursively, you move n-1 disks from the initial rod to the auxiliary one. You then move the largest disk to the final rod. So you end up moving n-1 disks from the auxiliary rod to the final one. Mathematically expressed, $T(n) = 2T(n-1)+1$

The number of moves grows exponentially with the number of disks, and we can see that the complexity is O(2^n). With each extra disk, the number of moves doubles.

The space complexity is O(n), because recursion is using stack. Its depth is proportional to n (one recursive call per disk is being made).

Graphs



The BubbleSort shows a steep increase in execution time as the array size grows, reflecting its inefficiency, especially with larger arrays.

The insertionSort performs better than BubbleSort, with a slower rise in the execution time.

Lastly, QuickSort has the best performance, with minimal increase in execution time even as the array size grows, demonstrating its efficiency with large arrays.

The linear Search shows a slight increase in execution time as the array size grows, though it remains relatively low. The different curves observed make sense, considering we chose a random target (x) for each array. Since the linear search traverses the whole array in order, execution time will be directly correlated to where the index of the random target (x) was located for each array size (i.e for array size 6000, the index was closer to the start of the array, etc).

The binary Search and Binary Search Recursive both maintain a very low and constant execution time across all array sizes, demonstrating their efficiency.

Overall, Binary Search and Binary Search Recursive are the most efficient, with negligible differences between them, while Linear Search performs slightly worse but still maintains low execution times.

Self Assessment of the group

We reached a unanimous self-assessment and are all happy to work together. There has been strong teamwork and communication throughout. We've supported each other, especially with exercises that turned out to be more challenging than expected. Moreover, we all worked together to do the document and explain the codes and graphs.

- **Josep Cubedo 10**

Has been a key member of this group, contributing valuable knowledge and helping others understand and complete the exercises. He has worked hard and shown great cooperation.

He made sure to make all the graphs with the right information and help others with the sorting exercises.

- **Miruna Ghiveci 9.5**

Has made many contributions and shown great interest in working. She has been very cooperative and a hard worker.

She also was able to complete the Hanoi Tower exercise and help others with the sorting exercises.

- **Laura Rigau 9.5**

Laura has also made contributions and successfully completed some exercises. She has shown great communication and worked hard.

She was able to do the quicksorting exercise and also help to do the sorting exercises.

- **Anna Ymbern 9.5**

Has also worked hard, helping to understand exercises and collaborating with others to complete some of the exercises. She has demonstrated lots of communication.

She did the Binary sorting exercises and helped others with the rest of sorting.