CH-231-A
hw6_p1.cpp
Flori Kusari
fkusari@jacobs-university.de

Bubble Sort & Stable and Adaptive Sorting

Problem 6.1

Pseudo-Code

```
1. bubbleSort(A[])  // Name of Function which uses an Array
2.    n = length(A)    // Setting a variable to the size of the Array A
3.    for i from 1 to n-1    // Iterated loop through elements
4.        // Flag "swapped" is initialized to false
5.        swapped = false
6.        // Last i elements are in the correct place already
7.        for j from 0 to n-i-1
8.            // Compare the neighboring elements
9.            if A[j] > A[j+1] then
10.               // If elements are in the wrong order, swap
11.               swap(A[j], A[j+1])
12.               swapped = true
13.        // If no elements were swapped, then break
14.        if not swapped then // If Swapped == false
15.            break
16.    end for // ends for loop
17. end bubbleSort() //end of function
```

B) Time-Complexities

1. **Worst-case time complexity:** ***$O(n^2)$* -** When the array is in reverse order, every element will be compared and swapped with every other element which means that in the worst possible case (excluding the possible errors in compilers of code, of course) the time-complexity of the algorithm will be ***$O(n^2)$***. The entire array is continuously being sorted until it finally is perfect so the worst case will not differ much from the average case. Still, in a way, it can be understood that based on how the algorithm works the worst case is the one involving the most amount of swaps and intuitively we can understand that the case involving the most swaps is the case in which the entire array is reversed.
2. **Average-case time complexity:** ***$O(n^2)$* -** Similarly to the worst-case scenario if one element is out of order that means that technically the whole sorted array is out of order which would lead the algorithm to go through all comparisons until everything was sorted leads to the same outcome as the worst-case time complexity of ***$O(n^2)$*.** This may seem counter-intuitive but when it is understood that the algorithm works by comparing and swapping the idea makes sense because for every swap the algorithm must check for other ones just in case the whole list is unsorted.
3. **Best-case time complexity:** ***$O(n)$* -** The best-case time complexity scenario is self-explanatory. The best case is that the array has already been sorted and this way the algorithm will only go through the array once and realize that everything is in the right order and not do anything. Although no change is occurring, the algorithm must know that nothing needs to be changed. The bare minimum is that it goes through every element before finally ending and returning the list without any changes. Although there are no changes, the algorithm still iterates through the array leaving a time complexity of ***$O(n)$*.**

(C) Stability of Sorting Algorithms

Known Definition: "***Stability*** refers to the ability of a sorting algorithm to maintain the relative order of records with equal keys (i.e., values). In other words, if two elements are equal, a stable sorting algorithm ensures they remain in the same order as they appeared in the input." (Wikipedia Definition)

- **Insertion Sort** - In ***Insertion Sort,*** each new element is inserted in the sorted array part "*without changing the relative order of elements with equal keys*". It shifts elements one position to the right, as it scans backward through the sorted portion, and it creates space for the new element but any elements with equal keys. Since none of that goes against the definition of stability, we can confidently conclude that ***Insertion Sort*** is **<u>stable</u>**.

- **Merge Sort** - Merge Sort carefully merges two sorted halves of the whole. If two elements from the two halves are equal, it will always take the element from the left half first, which in turn preserves their initial order from the inputted array. Since none of that goes against the definition of stability, we can confidently conclude that ***Merge Sort*** is **stable**.
- **Heap Sort** - Heap Sort builds a heap from the input and repeatedly extracts the max or min from the heap to build a sorted array. The process of heapification does not take the original order of equal elements into account and this will likely rearrange the order of the equal elements (Although in some lucky cases, it may preserve the order, it will not always do so). Since that goes against the very definition of stability, we can safely conclude that ***Heap Sort*** is **NOT stable**.
- **Bubble Sort** - In Bubble Sort, each pair of neighboring elements is swapped ONLY if they are in the wrong order. For equal elements, no swap is performed since they are equal and thus not in the wrong order. This preserves their original order and will not cause issues in any case even if both the equal elements are sorted wrongly. Taking the very logic behind ***Bubble Sort*** into consideration, we can conclude that ***Bubble Sort*** is **stable** by the very definition given above.

(D) Adaptive Sorting Algorithms

Known Definition: "***Adaptivity*** refers to the sorting algorithm's ability to take advantage of the existing order in its input, performing better (i.e., faster or with fewer operations) on partially sorted arrays." (Wikipedia Definition)

Insertion Sort

Adaptive: ***Insertion Sort*** performs significantly better on partially sorted arrays. It inserts elements into their correct position faster because it requires fewer comparisons and shifts for elements that are already close to their final sorted positions and it gets better and better the more sorted that the array already is. From the very way that the ***Insertion Sort*** works we can conclude that it must be adaptive since it follows the definition of adaptability perfectly

Merge Sort

Not Adaptive: By definition, the ***Merge Sort*** divides the array into subarrays, recursively sorts the subarrays, and then merges them into a single sorted array. It has a divide-and-conquer approach that does not take initial ordering into account meaning that the very logic that it uses is **NOT adaptive** since it will work the same consistently and it will not take into account anything and will only focus on the goal, not the means.

Heap Sort

Not Adaptive: ***Heap Sort***'s performance is not affected by the initial order of the input. Constructing the heap and then sorting it involves steps that are not based on the array's initial order, leading to a consistent performance every time regardless of initial order. The only aspect that will affect the algorithm would be the size of the array so there is no good argument to make for ***Heap Sort*** being adaptive.

Bubble Sort

"Arguably" **Adaptive**: The basic ***Bubble Sort*** algorithm is not adaptive, but an improved version that includes a flag to check if any swaps have occurred during a pass can detect when the array is already sorted. Although this can improve the algorithm by a little, the changes are nowhere near as significant as the ones from Insertion Sort and thus we can say that the basic version is ***NOT adaptive*** while the "improved" version is ***somewhat adaptive.*** Since the question probably refers to the traditional or "basic" ***Bubble Sort*** we can say that it is not adaptive and will have a consistent performance.