Homework Sheet 2

Authors: Abdullah Oğuz Topçuoğlu & Ahmed Waleed Ahmed Badawy Shora & Yousef Mostafa Farouk

Exercise 1

Given an array A[1, . . . , n] of integers and an integer k, we can decide whether there are two distinct entries in A that are k-similar by following these steps:

1. Sort the array A in ascending order. This can be done in O(n log n) time using merge sort from the lecture.

2. After sorting we are gonna start from the beginning of the sorted array and compare each element with the next element to check if their difference is less than or equal to k and greater than zero (greater than zero because elements needs to be distinct).

Pseudocode:

```
fun AreKSimilar(A[1...n], k)
   MergeSort(A)  // from the lecture, O(n log n) time
   for i = 1 to n-1 do
      if A[i+1] - A[i] <= k and A[i+1] != A[i] then
         return True
   return False
```

Correctness:
After sorting the array, all elements that are k-similar will be positioned next to each other. By iterating through the sorted array and checking the difference between consecutive elements, we can determine if there are any two distinct entries that are k-similar. If we find such a pair we return true, otherwise, we return false after checking all pairs.
Running Time Analysis:
The sorting step takes O(n log n) time. The subsequent loop iterates through the array once, taking O(n) time. Therefore, the overall time complexity of the algorithm is O(n log n) + O(n) = O(n log n).

Exercise 2

We can solve this problem by following these steps:

1. Create a copy of the original array A and sort it. Lets call the sorted array B. (O(n log n) time)

2. Iterate over the original array A and for each element get the first index of that element in B. (O(n log n) time) (O(n) for iterating over A and O(log n) for binary searching in B)

3. The index we found in step 2 is the number of elements smaller than the current element because B is sorted.

Pseudocode:

```
fun SmallerEntries(A[1...n])
   B := Copy(A)  // O(n) time
   MergeSort(B)  // from the lecture, O(n log n) time
   for i = 1 to n do
      R[i] := BinarySearchFirstIndex(B, A[i]) - 1 // O(log n) time
   return R

// providing this function here because we havent seen this in the lecture yet
fun BinarySearchFirstIndex(B[1...n], x)
   low := 1
   high := n
   result := n + 1
   while low <= high do
      mid := (low + high) / 2
      if B[mid] >= x then
         result := mid
         high := mid - 1
      else
         low := mid + 1
   return result
```

Correctness:
By sorting the array A into B, we can easily determine the number of elements smaller than each element in A. The binary search function finds the first occurrence of each element in the sorted array B, and since B is sorted, the index of this occurrence minus one gives the count of elements smaller than the current element.
Running Time Analysis:
The sorting step takes O(n log n) time. The loop iterates through the array A, taking O(n) time, and for each element, we perform a binary search in B which takes O(log n) time. Therefore, the overall time complexity of the algorithm is O(n log n) + O(n log n) = O(n log n).