Homework Sheet 2

Authors	**Tutors**
Abdullah Oğuz Topçuoğlu	Maryna Dernovaia
Ahmed Waleed Ahmed Badawy Shora	Jan-Hendrik Gindorf
Yousef Mostafa Farouk	guys fill here

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 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).

Exercise 3

a) **Pseudocode:**

```
EqualSumsSorted(A[1..n], B[1..n], k):
    i := 1
    j := n
    while i <= n and j > 0:
        temp := A[i] + B[j]
        if temp > k:
            j--
        else if temp < k:
            i++
        else:
            return true
    return false
```

Correctness: The algorithm starts with the smallest element in A and the largest in B. If the sum is too large, we decrease j to reduce it; if too small, we increase i to enlarge it. Since both arrays are sorted, every possible candidate pair is checked exactly once, so the algorithm correctly finds whether $A[i] + B[j] = k$.

Running Time Analysis: Both pointers i and j move at most n times, hence the total running time is $O(n)$.

b) **Pseudocode:**

```
EqualSumsUnsorted(A[1..n], B[1..n], k):
    MergeSort(A)                  // O(n log n)
    MergeSort(B)                  // O(n log n)
    return EqualSumsSorted(A, B, k) // O(n)
```

Correctness: By sorting both arrays, we can safely apply the algorithm from part (a). Since MergeSort is correct and EqualSumsSorted is correct, their composition is correct.

Running Time Analysis: Each MergeSort takes $O(n \log n)$, and EqualSumsSorted takes $O(n)$. Hence the total time is $O(2n \log n + n) = O(n \log n)$.

c) **Pseudocode:**

```
ArithmeticMean(A[1..n], B[1..n], C[1..n]):
    MergeSort(A)     // O(n log n)
    MergeSort(B)     // O(n log n)
    for i := 1 to n: // up to n iterations
        if EqualSumsSorted(A, B, 2 * C[i]): // O(n)
            return true
    return false
```

Correctness: The algorithm checks for each $C[i]$ whether there exist $A[j]$ and $B[k]$ such that $A[j] + B[k] = 2 \times C[i]$, i.e., $C[i]$ is the arithmetic mean of two elements. If such a pair exists, it returns true.

Running Time Analysis: Sorting takes $O(n \log n)$, and the loop runs n times, each calling an $O(n)$ function. Thus, the total time is $O(n \log n + n^2) = O(n^2)$.

Exercise 4

a) (3,4),(3,5),(4,5),(1,5),(2,5)

b) the reverse sorted array [n...1], it has $\sum_{i=1}^{n}(i-1) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$ bad pairs.

c) yes, the running time of insertion sort does scale with the number of bad pairs. This is because bad pairs represent elements that are "out of order" with respect to each other. Insertion sort works by repeatedly moving each element leftward until it reaches its correct position among the elements already sorted. Each such movement resolves one of the bad pairs, because when an element is moved left past another element that was originally smaller, that "bad pair" is no longer "bad".

d)

- **Pseudocode:**

```
ModifiedInsertionSort (A[1...n]):
inversions := 0
for i := 2,...,n:
   j := i
   while j > 1 and A[j-1] > A[j]:
       swap( A[j-1], A[j])
       j--
       inversions++
return inversions
```

Correctness: Each time the algorithm swaps two adjacent elements $A[j-1]$ and $A[j]$ where $A[j-1] > A[j]$, it removes exactly one bad pair. Initially, all bad pairs are present; by the end, the array is sorted, meaning no bad pairs remain. Since every swap corresponds to the resolution of one bad pair, the final value of `inversions` is exactly the number of bad pairs in the original array.

Running Time Analysis: The outer loop runs $n-1$ times, and in the worst case (when the array is reverse sorted), the inner loop executes $1 + 2 + \cdots + (n-1) = O(n^2)$ iterations. Hence the total running time is $O(n^2)$ in the worst case, and $O(k)$ on average where k is the number of bad pairs. (Same as insertion sort)

- **Pseudocode for bonus:**

```
ModifiedMerge (A[1...m], B[1....n])
allocate array C[1...n+m]
invs := 0
```

```
i := 1
j := 1
for k = 1,..,n+m:
    if j = m+1 or (i <= n and A[i] <= B[j]):
        C[k] := A[i]
        i++
    else
        C[k] := B[j]
        j++
        invs += n - i + 1
return (C, invs)

ModifiedMergeSort (A):
invs := 0
if n > 1:   //number of bad pairs = #of bad pairs in the right half +
             #of bad pairs in the left half + #of relative inversions
    m := |_n/2_| //floor division
    invs += ModifiedMergeSort(A[1...m])
    invs += ModifiedMergeSort(A[m+1..n])
    (A[1..n], temp) := ModifiedMerge(A[1...m], A[m+1..n])
return invs + temp
```

Correctness: The algorithm divides the array into two halves. It recursively counts:

- Bad pairs within the left half,
- Bad pairs within the right half, and
- Bad pairs across halves, which occur whenever an element from the right half is placed before an element from the left half during merging.

During the merge step, each time an element from the right half $B[j]$ is placed before a remaining element in the left half $A[i]$, there are $(n - i + 1)$ bad pairs formed with all remaining elements of A. Therefore, the algorithm correctly counts every bad pair exactly once.

Running Time Analysis: The recurrence relation for the algorithm is the same as the one of merge sort

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n),$$

which solves to $T(n) = O(n \log n)$. Thus, the modified merge sort counts all bad pairs in $O(n \log n)$ time.