Problem 3 (20 points). You are given two lists A and B, each of which is sorted in ascending order. It is guaranteed that all numbers in A and B are distinct. Given an integer k with $1 \le k \le |A|+|B|$, design an $O(\log|A|+\log|B|)$ time algorithm for computing the k-th smallest element in the union of A and B.

Answer. Define recursion function select-in-two-sorted-arrays (A, a_1, a_2, B, b_1, b_2, k) return the k-th smallest element in the union of $A[a_1 \cdots a_2]$ and $B[b_1 \cdots b_2]$. The pseudo-code for it is as follows.

```
function select-in-two-sorted-arrays (A, a1, a2, B, b1, b2, k)
    if a1 > a2: return B[b1 + k − 1];
    if b1 > b2: return A[a1 + k − 1];
    if k = 1: return the smaller one between A[a1] and B[b1];
    let a = (a1 + a2)/2;
    let b = (b1 + b2)/2;
    m = a2 − a1 + 1;
    n = b2 − b1 + 1;
    if A[a] < B[b]:
      if k < m/2 + n/2: return select-in-two-sorted-arrays (A, a1, a2, B, b1, b − 1, k);
      if k ≥ m/2 + n/2: return select-in-two-sorted-arrays (A, a + 1, a2, B, b1, b2, k − a + a1 − 1);
    else:
      if k < m/2 + n/2: return select-in-two-sorted-arrays (A, a1, a − 1, B, b1, b2, k);
      if k ≥ m/2 + n/2: return select-in-two-sorted-arrays (A, a1, a2, B, b + 1, b2, k − b + b1 − 1);
    end if
end function
```

Call select-in-two-sorted-arrays (A, 1, $|A|$, B, 1, $|B|$, k) will give the k-th smallest element in $A \cup B$.

In each iteration, either $a_2 - a_1$ is reduced by half, or of $b_2 - b_1$ is reduced by half. Therefore, the running time is $O(\log|A|+\log|B|)$.

Problem 5 (20 points). Let $S[1 \cdots n]$ be an array with n *distinct* integers. We say two indices (i, j) form an inversion if we have $i < j$ and $S[i] > S[j]$. Design an divide-and-conquer algorithm that counts the number of inversions in S. Your algorithm should run in $O(n \cdot \log n)$ time. For example, if you are given $S = (3,8,5,2,9)$, then your algorithm should return 4. The 4 inversions are $(3,2),(8,5),(8,2),(5,2)$.

Answer. Define recursive function count-inversions (S) returns (S', N), where S' is the sorted list of S, and N is the number of inversions in array S. The pseudo-code for this function is below.

---

function count-inversions (S)

  if $(|S| = 1)$: return 0;

  let $n = |S|$;

  $(S_1, N_1) =$ count-inversion $(S[1 \cdots n/2])$;

  $(S_2, N_2) =$ count-inversion $(S[n/2+1 \cdots n])$;

  $(S_3, N_3) =$ count-inversions-between-two-sorted-arrays (S_1, S_2);

  return $(S_3, N_1 + N_2 + N_3)$;

end function

The function count-inversions-between-two-sorted-arrays (S_1, S_2) used above takes two sorted list S_1 and S_2 as input, and returns (S_3, N_3), where S_3 is the sorted list for all elements in S_1 and S_2, and N_3 is the number of inversions across S_1 and S_2, i.e., number of pair (i, j) such that $S_1[i] > S_2[j]$. The pseudo-code for this function is below.

```
function count-inversions-between-two-sorted-lists (S_1, S_2)
    let k_1 = 1, k_2 = 1, k_3 = 1, N = 0, init list S_3;
    while (k_1 ≤ |S_1| and k_2 ≤ |S_2|)
      if (S_1[k_1] < S_2[k_2])
        S_3[k_3] = S[k_1];
        k_1 = k_1 + 1;
        k_3 = k_3 + 1;
      else
        N = N + (|S_1| − k_1) + 1;
        S_3[k_3] = S[k_2];
        k_2 = k_2 + 1;
        k_3 = k_3 + 1;
      end if
    end while
    while (k_1 ≤ |S_1|)
      k_1 = k_1 + 1;
      S_3[k_3] = S[k_1];
      k_3 = k_3 + 1;
    end while
    while (k_2 ≤ |S_1|)
      k_2 = k_2 + 1;
      S_3[k_3] = S[k_2];
      k_3 = k_3 + 1;
    end while
    return (S_3, N);
end function
```

Call count-inversions (S), and the second returned parameter gives the number of inversions in S.

The function of count-inversions-between-two-sorted-arrays takes linear time. Hence, the recursion for the entire algorithm is $T(n) = 2 \cdot T(n/2) + O(n)$. Therefore the running time is $O(n \cdot \log n)$.