Problem 1 (10 points). Let $S[1 \cdots n]$ be an *sorted* array with n *distinct* integers in ascending order. We select an integer k uniformly at random from $\{1, 2, \cdots, n\}$.

1. What is the probability that $S[k]$ is no less than the i-th smallest number in S *and* no larger than the j-th smallest number in S (assume that $i < j$)?
2. What is the probability that $S[k]$ is no larger than the i-th smallest number in S *or* no less than the j-th smallest number in S (assume that $i < j$)?

Solution.

1. $(j - i + 1)/n$.
2. $(i + n - j + 1)/n$.

Problem 2 (10 points). Let $S[1 \cdots n]$ be an array with n *distinct* integers. We select an integer k uniformly at random from $\{1, 2, \cdots, n\}$.

1. What is the probability that $S[k]$ is no less than the i-th smallest number in S *and* no larger than the j-th smallest number in S (assume that $i < j$)?
2. What is the probability that $S[k]$ is no larger than the i-th smallest number in S *or* no less than the j-th smallest number in S (assume that $i < j$)?

Solution.

1. $(j - i + 1)/n$.
2. $(i + n - j + 1)/n$.

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.

Solution. ~~Without loss of generality, assume that $|A| \le k$ and $|B| \le k$, otherwise we can only keep the first k elements of A or B.~~ (This is only true in the beginning but not during the recursion anymore.) We will again use the idea of *binary search* (refer to Problem 3 of Homework 2). Consider the middle elements of $A[m/2]$ and $B[n/2]$, where $m = |A|$ and $n = |B|$. We show that we can discard half numbers in one of the two arrays by comparing $A[m/2]$ and $B[n/2]$. Suppose that we have $A[m/2] < B[n/2]$. We consider two cases.
The first case is when we have $k < m/2 + n/2$, and we can claim that we can discard the second half of array B, i.e., the k-th smallest element must be not in $B[n/2 \cdots n]$. To see this, consider how many numbers are *guaranteed to be smaller than* $B[n/2]$. Clearly, the first $n/2 - 1$ numbers in B are smaller than $B[n/2]$. In array A, the first $m/2$ numbers are also guaranteed to be smaller than $B[n/2]$ as we have $A[m/2] < B[n/2]$. So the numbers in $A \cup B$ that can are guaranteed to be smaller than $B[n/2]$ is at least $m/2 + n/2 - 1 \ge k$, as we have $m/2 + n/2 > k$. This implies that $B[n/2]$ is at least the $(k+1)$-th smallest number in $A \cup B$. Hence, we can discard all numbers in $B[n/2 \cdots n]$, and the k-th smallest number in $A \cup B$ will be exactly the k-th smallest number in the union of A and $B[1 \cdots n/2 - 1]$.
The second case is when we have $k \ge m/2 + n/2$, then we can safely discard the first half of array A, i.e., the k-th smallest element must be not in $A[1 \cdots m/2]$. To see this, consider how many numbers *might be smaller than* $A[m/2]$. Clearly, the first $m/2 - 1$ numbers in A are smaller than $A[m/2]$. In array B, the first $n/2 - 1$ numbers might be smaller than $A[m/2]$, as all numbers in $B[n/2 \cdots n]$ are larger than $A[m/2]$. So the maximum numbers in $A \cup B$ that can be smaller than $A[m/2]$ is $m/2 - 1 + n/2 - 1 =$

$m/2+n/2-2 \le k-2$. This implies that $A[m/2]$ is at most the $(k-1)$-th smallest number in $A \cup B$. Hence, we can discard the first $m/2$ numbers in A, and the k-th smallest number in $A \cup B$ will be exactly the $(k-m/2)$-th smallest number in the union of $A[m/2+1, m]$ and B. Symmetric results can be obtained if we have $A[m/2] > B[n/2]$.

We define that the 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 $B[b_1 \cdots b_2]$. There are 2 possible base cases: if $a_1 > a_2$ (resp. $b_1 > b_2$) then it means that A is empty (resp. B is empty), and we can immediately locate the desired element in B (resp. in A); if we have $k = 1$, then we can compare the first elements of the arrays and return the smaller one.

```
function select-in-two-sorted-arrays ($A$, $a_1$, $a_2$, $B$, $b_1$, $b_2$, $k$)
    if $a_1 > a_2$: return $B[b_1+k-1]$;
    if $b_1 > b_2$: return $A[a_1+k-1]$;
    if $k = 1$: return the smaller one between $A[a_1]$ and $B[b_1]$;
    let $a = (a_1+a_2)/2$;
    let $b = (b_1+b_2)/2$;
    $m = a_2 - a_1 + 1$;
    $n = b_2 - b_1 + 1$;
    if $A[a] < B[b]$:
      if $k < m/2+n/2$: return select-in-two-sorted-arrays ($A$, $a_1$, $a_2$, $B$, $b_1$, $b-1$, $k$);
      if $k \ge m/2+n/2$: return select-in-two-sorted-arrays ($A$, $a+1$, $a_2$, $B$, $b_1$, $b_2$, $k-a+a_1-1$);
    else:
      if $k < m/2+n/2$: return select-in-two-sorted-arrays ($A$, $a_1$, $a-1$, $B$, $b_1$, $b_2$, $k$);
      if $k \ge m/2+n/2$: return select-in-two-sorted-arrays ($A$, $a_1$, $a_2$, $B$, $b+1$, $b_2$, $k-b+b_1-1$);
    end if
end function
```

We can call select-in-two-sorted-arrays (A, 1, $|A|$, B, 1, $|B|$, k) to compute the k-th smallest element in $A \cup B$.

To analyze the running time, notice that 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 4 (20 points). Let $S[1 \cdots n]$ be an array with n *distinct* integers. Given an integer k with $1 \le k \le n$, design an algorithm to partition S into S_L (integers in S that are smaller than $S[k]$), $S[k]$, and S_R (integers in S that are larger than $S[k]$) using at most constant amount of extra memory.

Solution. We maintain two pointers, k_1 and k_2, pointing to the first and last elements in the beginning, i.e., $k_1 = 1$ and $k_2 = n$. We will move k_1 right and move k_2 left until we find that $S[k_1] > S[k]$, and $S[k_2] < S[k]$. When this happens, we swap $S[k_1]$ and $S[k_2]$. To avoid handling multiple cases when k_1 or k_2 equals to k, we can first swap $S[1]$ and $S[k]$ to protect $S[k]$ and swap them again in the end. The pseudo-code is as follows.

```
function partition-in-place ($S$, $k$)
    let $k_1 = 1$;
    let $k_2 = n$;
    swap $S[1]$ and $S[k]$; /* now $S[1]$ is the pivot */
    while ($k_1 < k_2$)
        while ($S[k_1] < S[1]$) $k_1 = k_1 + 1$;
        while ($S[k_2] > S[1]$) $k_2 = k_2 - 1$;
        if ($k_1 < k_2$): swap $S[k_1]$ and $S[k_2]$;
    end while;
    swap $S[k_2]$ and $S_1$;
    /*$S[1 \cdots k_2 - 1]$, $S[k_2]$, and $S[k_2 + 1 \cdots n]$ will be the desired partition*/
end function
```

The above algorithm takes linear time and only uses constant number of memory units.

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 $(3,2),(8,5),(8,2),(5,2)$.

Solution. We can design a divide-and-conquer algorithm to count the number of inversions. 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. We can recursively call $(S_1, N_1) =$ count-inversions ($S[1 \cdots n/2]$), and $(S_2, N_2) =$ count-inversions ($S[n/2+1 \cdots n]$). To compute the total number of inversions in S, we need to add up N_1 and N_2, and also the inversions between the two halves of S. Since now we have the sorted lists of the two halves, which are stored in S_1 and S_2, we can use them to count inversions. Similar to the merge-two-sorted-arrays function, we can maintain two pointers and scan both arrays, once we have that $S_1[i] > S_2[j]$, then we know all numbers in S_1 that are at index i and beyond $S[i]$ are larger than $S_2[j]$, i.e., we have $|S_1| - i + 1$ inversions by comparing all elements in S_1 to $S_2[j]$. The pseudo-code for counting the number of inversions between two sorted arrays is below.

```
function count-inversions-between-two-sorted-lists ($S_1$, $S_2$) /* merge $S_1\&S_2$, and count cross inversions
    let $k_1 = 1$, $k_2 = 1$, $k_3 = 1$, $N = 0$;
    init list $S_3$; /* $S_3$ will store the merged list of $S_1$ and $S_2$;
    while ($k_1 \leq |S_1|$ and $k_2 \leq |S_2|$)
        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 ($k_1 \leq |S_1|$)
        $S_3[k_3] = S[k_1]$;
        $k_1 = k_1 + 1$;
        $k_3 = k_3 + 1$;
    end while
    while ($k_2 \leq |S_1|$)
        $S_3[k_3] = S[k_2]$;
        $k_2 = k_2 + 1$;
        $k_3 = k_3 + 1$;
    end while
    return $S_3$, $N$;
end function
```

The entire divide-and-conquer algorithm for computing inversions with array S 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 merge-step of count-inversions-between-two-sorted-arrays takes linear time. Hence, the recursion for the above algorithm is $T(n) = 2 \cdot T(n/2) + O(n)$. Therefore the running time is $O(n \cdot \log n)$.