HW 4

1. *Suppose the input to RSelect consists of n items all with the same key value. Suppose each pivot is chosen as the leftmost item in the subarray currently being processed. How many comparisons does RSelect perform in this case when seeking the smallest item? What about if you are seeking the largest item? Justify your answers.*

   Ans:

   > 1. When seeking the smallest item, the pivot should compare all the items in the subarray to find who is smaller or larger than that. As all the items have the same key value and the value that is smaller or equal to the pivot moves to the left, the pivot will move to the rightmost. Hence, each time a new pivot is compared, it will finally move to the rightmost of the subarray, which will be $n-1+n-2+n-3+\cdots+1=\frac{n(n-1)}{2}$
   > 2. For the largest one, it will be found after the comparison of the first pivot, which is $n-1$.

2. *Give an O(n) time algorithm to complete the sort of array A. You may assume that n is an integer multiple of m. Justify your running time bound, and explain why your algorithm is correct.*

   Ans:

   > As we can see, every item in the r-th sequence is smaller than every item in the (r + 2)nd sequence; in particular, i.e. $A[rm+m]<A[(r+2)m+1]$ for $0\leq r\leq n/m-3$. Therefore, we can iterate the whole array by the odd values of r and the even values of r, respectively. Then we will get two sorted arrays at the cost of cn.
   >
   > Now it's similar to MergeSort. We merge two sorted arrays into one array, which will cost only cn.
   >
   > Then the time complexity would be $O(n)$.

3. *Suppose you are given two sorted arrays A[1 : m] and B[1 : n] as input, where 1 ≤ m ≤ n. Give an algorithm to find, for each item A[i], how many items in B are smaller than A[i]. You may assume all the items are distinct. Your algorithm should run in time O(m+m log(n/m)). You may assume that n is an integer multiple of m. Justify your running time bound and explain why your algorithm is correct.*

   Ans:

   > First, divide array B by blocks, whose size is n/m and we only take the first value of the block into account. Therefore, when comparing with A, we only use m values in array B to find the items. For example, values in B are like $\{B[i],B[i+n/m],\dots B[n]\}$.
   >
   > Then the idea of the algorithm is:
   >
   > ```
   > if B[i]<A[j]:
   >   i<-i+n/m
   > else
   >   BinarySearch(B[i-n/m:i+i]) to find the index that less than A[j]
   >   j<-j+1
   > ```

> As we can see, the index i will start from 0 to n, and each step is n/m, then it will iterate m times, so the cost is m.
>
> Every time each item A[i] wants to find an exact item in the block, which is the largest item in B but smaller than A, it will execute a binary search in its block, searching in size n/m. This step will execute m times as array B has m values and each time the cost is $\log(n/m)$, so the total cost will be $m\log(n/m)$.
>
> Hence, the overall cost is $O(m + m\log(n/m))$.

4. *Suppose you are given an array A[1 : n] of integers in the range [1, m], where m > n. Suppose that you are also given an uninitialized array B[1 : m] of integers (this means you cannot assume anything about the values in B initially). Show how to identify all duplicate items in A in time O(n) (the first instance of an item in increasing i index order is a nonduplicate, the others are duplicates). You can report a duplicate item as the pair (A[i], i). You will need to use array B to help.*

   Ans:

   > 1. Iterate all values of array A: for index i in loop, $0 \leq i < n$, assign $B[A[i]] = 0$. As it will iterate n times, the cost will be $O(n)$ here.
   > 2. Iterate all values of array A: for index i in loop, $0 \leq i < n$, if $B[A[i]] = 0$, assign $B[A[i]] = 1$. Otherwise, report $pair(A[i], i)$ when finding $B[A[i]] = 1$.
   >
   > The total cost will be $O(n) + O(n) = O(n)$.