Analysis of Algorithms: Solutions 4

```
                             X
                             X
                             X
                             X
                             X
number of                    X     X
homeworks                    X     X
                          X  X     X  X  X
                          X  X  X  X  X  X
                          X  X  X  X  X  X
                    X     X  X  X  X  X  X
                    X  X  X  X  X  X  X  X
           --------------------------------
           0  1  2  3  4  5  6  7  8  9  10
                        grades
```

The histogram shows the distribution of grades for the homeworks submitted on time.

Problem 1
Consider a computer environment where the control flow of a program can split three ways after a single comparison $a_i : a_j$, according to whether $a_i < a_j$, $a_i = a_j$, or $a_i > a_j$. Argue that the number of these three-way comparisons required to sort an n-element array is $\Omega(n \lg n)$.

Suppose that all numbers in the input array are *distinct*. Then, the comparison algorithm never encounters the "$a_i = a_j$" situation, and we may represent its control flow by a binary decision tree, like we did in class. The height of this decision tree is $\Omega(n \log n)$; hence, the worst-case complexity of sorting an array of distinct numbers is $\Omega(n \log n)$. Therefore, the complexity of sorting an arbitrary array is also $\Omega(n \log n)$.

Problem 2
Using Figure 9.2 (page 176) in the textbook as a model, illustrate the operation of Counting-Sort on the array $A = \langle 7, 1, 3, 1, 2, 4, 5, 7, 2, 4, 3 \rangle$.

The values in the array C are as follows:

	1	2	3	4	5	6	7
after first loop:	0	0	0	0	0	0	0
after second loop:	2	2	2	2	1	0	2
after third loop:	2	4	6	8	9	9	11
after fourth loop:	0	2	4	6	8	9	9

Problem 3

Briefly describe how to adapt (a) MERGE-SORT and (b) QUICK-SORT to sort elements stored in a linked list. Give the time complexity of your algorithms; is it the same as the complexity of sorting an array?

We assume that every element x of a linked list has two fields, $next[x]$ and $key[x]$. The *next* field points to the next element of the linked list, whereas *key* contains a numeric value. If x is the last element in the list, then $next[x]$ is NIL.

We describe the modified versions of MERGE-SORT and QUICK-SORT procedures. The time complexity of both procedures is the same as that for sorting arrays.

(a) The MERGE-SORT procedure gets two arguments, the first element of a linked list and the number of elements in the list. The procedure finds the middle of the list and cuts it in two sublists, y and z. Then, it makes recursive calls to sort these sublists. The MERGE procedure is similar to that for arrays; however, it may be implemented to sort in-place.

```
MERGE-SORT(x, n)   ▷ x is the first element; n is the number of elements.
if n > 1
   then q ← ⌊n/2⌋
        y ← x
        for i ← 1 to q − 1   ▷ Find the middle of the list.
           do x ← next[x]
        z ← next[x]   ▷ Beginning of the second sublist.
        next[x] ← NIL   ▷ End of the first sublist.
        y ← MERGE-SORT(y, q)   ▷ Sort the first sublist.
        z ← MERGE-SORT(z, n − q)   ▷ Sort the second sublist.
        return MERGE(y, z)   ▷ Return the sorted list.
```

(b) The QUICK-SORT procedure gets the first element of a linked list, and calls PARTITION to divide the input list into two lists. The PARTITION procedure uses the *key* value of the first element as the "pivot" for partitioning, and constructs two new linked lists: one with the values smaller than the pivot, and the other with the values larger than the pivot; it returns both lists. After calling PARTITION, the QUICK-SORT procedure makes recursive calls to sort the two lists, and then appends the second sorted list to the end of the first one.

```
QUICK-SORT(x)   ▷ x is the first element of the list.
if next[x] ≠ NIL   ▷ More than one element?
   then y, z ← PARTITION(x)   ▷ PARTITION returns two lists.
        y ← QUICK-SORT(y)
        z ← QUICK-SORT(z)
        x ← y
        while next[x] ≠ NIL   ▷ Find the end of the first list.
           do x ← next[x]
        next[x] ← z   ▷ Append the second list to the end of the first one.
        return y
```

PARTITION(x)
$k \leftarrow key[x]$ ▷ k is the pivot for partitioning.
$y \leftarrow$ NIL ▷ List of elements smaller than k.
$z \leftarrow$ NIL ▷ List of elements greater than k.
while $x \neq$ NIL
    do $x\text{-}next \leftarrow next[x]$
        if $key[x] \leq k$
            then ▷ Add x to the smaller-element list.
                $next[x] \leftarrow y$
                $y \leftarrow x$
            else ▷ Add x to the larger-element list.
                $next[x] \leftarrow z$
                $z \leftarrow x$
        $x \leftarrow x\text{-}next$ ▷ Move to the next element.
return y, z

Problem 4

Consider an array $A[1..n]$ whose elements are distinct integer numbers, and describe an algorithm that finds the largest and second largest element of this array using $(n+\lceil\log n\rceil-2)$ comparisons.

We may think of this problem as a tournament among n chess players. Every number is a player, and every comparison is a game, which reveals the stronger of the two players. We need to plan $(n+\lceil\log n\rceil-2)$ games that reveal the strongest and second strongest player.

First, we break the players in pairs and make them play, thus revealing the stronger one in each pair. These stronger players enter the second round of the tournament. If the number of players is odd, then one of them does not play in the first round.

In the second round, we again break the players in pairs and make them play, and then move the winners to the third round. We repeat this operation until finding the strongest player. The total number of games for identifying this player is $n-1$.

Note that the final winner of the tournament has played $\lceil\lg n\rceil$ games, and the *second strongest player is one of those who have lost to the winner.* Thus, there are $\lceil\lg n\rceil$ people who have lost to the winner, and we have to find the strongest one among them. We run a smaller tournament among these people, which involves $\lceil\lg n\rceil-1$ games and reveals the second strongest player. The total number of games is $(n+\lceil\log n\rceil-2)$.