Solution 6

1. Given an array $A[1..n]$ of n numbers, we can calculate the maximum in A only by comparing numbers. Give an algorithm that finds the maximum and second maximum using exactly $n + O(\log n)$ comparisons.

   Solution:

   The idea is to use the binary decision tree to calculate the maximum. All the values in A are initialized as leaves in the tree. It takes n comparisons to get the root value as the maximum and there is a path in the tree for the root because it "wins" every comparison. The second maximum in A is exactly the maximum siblings on this path. Since the path is $\log n$ long, get the max siblings takes $O(\log n)$ time.

   ```
   MAX($A[1..n]$)
   for $i \leftarrow 1$ to $n$
      $T[\log n][i-1] \leftarrow A[i]$
   for $l \leftarrow \log n ... 0$
      for $i \leftarrow 1$ to $2^l$
            $T[l+1][i/2] \leftarrow \max(T[l][i], T[l][i+1])$
   max$\leftarrow T[0][0]$
   follow the path of $T[0][0]$ in the $T$, select the maximum of $T[0][0]$ 's siblings
   ```

2. Suppose $A_n = a_1, a_2, ..., a_n$ is a set of distinct coin types, where each a_i is a positive integer. Suppose also that $a_1 < a_2 < ... < a_n$. The coin-changing problem is defined as follows. Given a positive integer C, find the smallest number of coins from A_n that add up to C, given that an unlimited number of coins of each type are available.

   (a) Suppose $a_1 = 1$. A greedy algorithm will make change by using the larger coins first, using as many coins of each type as it can before it moves to the next lower denomination. Show that this algorithm does not necessarily generate a solution with the minimum number of coins.

   Solution:

   For the counterexample, consider the denomination is $\{1,3,4\}$ and you want to generate \$6\$. The greedy algorithm will output $\{4,1,1\}$. However, the minimum number of combinations of coins is $\{3,3\}$.

   (b) Show that if $A_n = \{1, c, c^2, ..., c^{n-1}\}, c \geq 2$, then the greedy algorithm always gives a minimal solution for the coin-changing problem.

   Solution:

   By way of contradiction, assume the optimal solution is $\{v_0, v_1, ..., v_n\}$ is different with greedy solution $\{g_0, g_1, ..., g_n\}$ where v_i,g_i give the number of coins for type c^i. We first prove the following lemma,

   Lemma 1. *$\forall i \in \{0, ..., n-1\}$, $v_i < c$*

Proof. Assume $v_i > c$, we can replace the change $v_i c^i$ by $(c^{i+1} + (v_i - c)c^i$. So the number of coins is reduced by $(c-1)$, contradiction. □

Now, we prove the optimality of greedy solution by showing every $v_i = g_i$

Theorem 2. $\forall i \in \{0, ..., n\}, v_i = g_i$.

Proof. By way of contraction, there is some $v_i \neq g_i$. We look for the first difference between two solutions from n down to 1. W.l.o.g., there exists $v_j \neq g_j$ and $\forall i > j, v_i = g_i$. By the greedy rule, we have $v_j < g_j$. Moreover, $S = \sum_{k=1}^{j} v_j c_k = \sum_{k=1}^{j} g_j c_k$ by the choice of j. However,

$$S = \sum_{k=1}^{j} v_j c_k \leq \sum_{k=1}^{j-1}(c-1)c_k + v_j c_j = c^j - 1 + v_j c^j = c^j(v_j+1) - 1 \leq c_j g_j - 1 \leq \sum_{k=1}^{j} g_j c_k - 1 = S - 1$$

which is a contradiction. □

(c) Design an $O(nC)$ algorithm that gives a minimum solution for the general case of A_n.
Solution:
Let function $f(C)$ denote the minimum number of coins needed to generate change for C. Consider the following recursion for computing f:

$$\begin{cases} f(C) = \min_{i \in [n]: a_i \leq C} f(C - a_i) + 1 & \text{if } C > 0 \\ f(0) = 0 \end{cases}$$

This allows us to have a dynamic programming algorithm. The algorithm tries to compute the elements of a one-dimensional array $F[0...C]$, where $F[0] = 0$,and for i from 1 to C the algorithm computes $F[i]$ using the above recursive formula. The final output of the algorithm is $F[C]$. The time complexity is $O(nC)$ because computing each of the m elements of array F takes $O(k)$ time using the above recursive formula.

3. Given n items $\{1, ..., n\}$, and each item i has a nonnegative weight w_i and a distinct value v_i. We can carry a maximum weight of W in a knapsack. The items are blocks of cheeses so that for each item i, we can take only a fraction x_i of it, where $0 \leq x_i \leq 1$. Then item i contributes weight $x_i w_i$ and value $x_i v_i$ in the knapsack. The continuous knapsack problem asks you to take a fraction of each item to maximize the total value, subject to the total weight does not exceed W. The original knapsack problem is equivalent to say x_i is either 0 or 1 depends whether the item is in or not. Give an $O(n)$ algorithm for the continuous knapsack problem.

Solution:

```
FractionalKnapsack($w[1...n], v[1...n], W$)
    f[1...n] ←[0, 0, ..., 0] // result. f[i]: fraction to take from item i
    items = list[Item]
    for $i$ ← 1...$n$ // initialize items
        items[i] = Item(index=i, weight=w[i], value=v[i], ratio=w[i]/v[i])
    Search(items, $W$, $f$)
    return $f$
```

```
SEARCH(items[1...n], $W$, $f$)
  if COMPUTEWEIGHT(items) < $W$ // COMPUTEWEIGHT: sum of item weights
      for$i \leftarrow 1...n$
          f[items[i].index] = 1 // take all items
      return
  l, p = PARTITION(items)
  // After PARTITION by ratio:
  // items[1...l] ratio>pivot.ratio
  // items[l+1...p] ratio == pivot.ratio
  // items[p+1...n] ratio < pivot.ratio
  $W_1$ = COMPUTEWEIGHT(items[1...l])
  if $W_1 < W$
      SEARCH(items[1...l], $W$, $f$)
  else
      // put items[1...l] into knapsack
      for $i \leftarrow 1...l$
          f[items[i].index] = 1
      // try to put items[l+1...p] into knapsack one by one
      for $i \leftarrow l + 1...p$
          item = items[i]
          fraction = max(1, $(W - W_1)$/item.weight)
          f[item.index] = fraction
          $W_1$ += fraction*item.weight
          if $W1 == W$
              return
      //continue searching in items[p+1...n]
      SEARCH(items[p+1...n], $W - W_1$, $f$)
```

4. Recall the problem of finding the number of inversions. We are given an array $A[1..n]$ of n numbers, which we assume are all distinct, and we define an inversion to be a pair $i < j$ such that $A[i] > A[j]$. The problem is to count the number of inversions in A. Let's call a pair (i, j) is a *significant inversion* if $i < j$ and $A[i] > 2A[j]$. Give an $O(n \log n)$ algorithm to count the number of significant inversions in A.

   Solution:

   The idea is same as the counting the inversion, except we use thedefinition of *significant inversion.* So, the merge phase will have two parts, one for counting the significant inversion, the other is an ordinary merge phase in the merge sort algorithm since we need to return a sorted list after the merge.

```
SigInver($A[1..n]$)
    if $n = 0$
        return 0
    $c_l, L[1..n/2] \leftarrow$ SigInver($A[1..n/2]$)
        // $c_l$ is the # of sig. inv. in the left half and $L[1..n/2]$ is the sorted left half
    $c_l, R[1..n/2] \leftarrow$ SigInver($A[n/2+1..n]$)
    $c \leftarrow c_l + c_r$
    $i, j \leftarrow 1$
    while $L$ and $R$ are not empty
        if $L[i] \geq 2R[j]$
            $c \leftarrow c+$ number of elements remaining in $L$
            $j \leftarrow j + 1$
        else
            $i \leftarrow i + 1$
    $A' \leftarrow$merge $L$ and $R$ in sorted order
    return $c, A'$
```