EECS 3101: Design and Analysis of Algorithms
Tutorial 7 – Solutions

1. (4 points) Suppose $A_n = \{a_1, a_2, \ldots, a_n\}$ is a set of distinct coin types, where each a_i is an integer, $1 \leq i \leq n$. Suppose also that $a_1 < a_2 < \ldots < a_n$. The coin changing problem is defined as follows. Given an 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. Design a dynamic programming algorithm that take inputs A_n, C and outputs the minimum number of coins needed to solve the coin-changing problem. Prove optimal substructure and analyze the running time of your algorithm.

   NOTE: A solution may not exist for certain instances of the problem. In such cases the algorithm should output "No solution".

   Solution: Since we do not assume that $a_1 = 1$, we may not be able to make change for all amounts. For example if $a_1 = 2$, we cannot make change for 1 cent.

   Again, there are many possible dynamic programming formulations for this problem. One of them is as follows. Let $N(a, i)$ be the minimum of coins needed to make change for amount a using coins of type $1, 2, \ldots, i$. Then the answer to the given problem is $N(C, n)$. As usual, we need to prove optimal substructure to be sure that dynamic programming can be used.

   Optimal substructure: Suppose you know that the optimal solution for the problem $N(C, n)$ contains m_n coins of type n. Then we must show that the subproblem $N(C - a_n m_n, n-1)$ is solved optimally by the coins of denomination 1 through $n-1$ in the optimal solution. This is the usual cut-and-paste argument. If there is a way to make change for this problem using fewer coins, simply augment this solution with m_n coins of type n. Then this augmented solution for $N(C, n)$ makes change for amount C using fewer coins than the optimal which is a contradiction. Therefore, the problem has the optimal substructure property.

   Recurrence: The number of coins, m_n of type n in an optimal solution must satisfy $0 \leq m_n \leq \lfloor C/a_n \rfloor$. This gives us the following recurrence: $N(a, 1) = a$, $1 \leq a \leq C$, and for $i > 1$.

   $$N(a, i) = \min_{0 \leq j \leq \lfloor a/c_i \rfloor} j + N(a - ja_i, i-1).$$

   Note that the algorithm must handle the case where no solution exists. One way to do this is to define $N(a, i) = \infty$ when $a < 0$.

   Algorithm: This recurrence can be used to fill up the $C \times n$ table $N(a, i)$ row-by-row (i.e., $i = 1, \ldots, n$), from left to right (i.e., $a = 1, 2, \ldots, C$). The number of coins actually used can be computed if the value of j at each step is stored in an auxilliary array.

   Running time: For each row i, the number of steps required to compute every cell in the table is $O(a/c_i)$. Assuming that the denominations c_i are constants (i.e. independent of C) an upper bound on the total number of steps required is $O(C^2 n)$.

Alternative solution: It turns out that a different formulation gives a more efficient solution. Let $N(a)$ be the minimum of coins needed to make change for amount a. Then the answer to the given problem is $N(C)$. Again, we need to prove optimal substructure to be sure that dynamic programming can be used.

Optimal substructure: Suppose you know that the optimal solution for the problem $N(C)$ contains a coin of type n. Then we must show that the subproblem $N(C - a_n)$ is solved optimally by the remaining coins in the optimal solution. The usual cut-and-paste argument works. If there is a way to make change for this problem using fewer coins, simply augment this solution with a coins of type n. Then this augmented solution for $N(C)$ makes change for amount C using fewer coins than the optimal which is a contradiction. Therefore, the problem has the optimal substructure property.

Recurrence: If we number the coins in an optimal solution in some order, the last coin is of type n for some n. This gives us the following recurrence: $N(a_i) = 1$, $1 \leq i \leq n$, and for other $a > 1$.

$$N(a) = 1 + \min_{0 \leq j \leq n} \{N(a - a_j)\}.$$

In the above, we use the convention $N(a) = \infty$ if $a < 0$.

Algorithm: This recurrence can be used to fill up the $C \times 1$ table $N(a)$, from left to right (i.e., $a = 1, 2, \ldots, C$). The coins actually used can be computed if the value of j at each step is stored in an auxilliary array.

Running time: For each row i, the number of steps required to compute every cell in the table is $O(n)$. Thus an upper bound on the total number of steps required is $O(Cn)$.

2. Problem 15.4-5 on page 356 in Edition 2, page 397 in Edition 3.

   Solution: The simplest and most elegant solution to this problem is the following: sort the sequence $A[1..n]$ by value and store it in array $B[1..n]$, and then compute LCS(A, B). The actual sequence can be constructed by the technique described in the book to get the longest common subsequence.

   Correctness: We need to prove that the algorithm described above is correct; i.e., if the length of the LCS is ℓ and the optimal solution has length s, we need to prove that $\ell = s$.

   First, we prove that $\ell \leq s$. The LCS provides a sequence of numbers whose indices are increasing and values are increasing. Clearly this sequence must have length no larger than the optimal solution. Next, we show that $s \leq \ell$. Given an optimal solution to our problem, it is a subsequence of A and it is also a subsequence of B. In other words, it is a common subsequence of the sequences A, B defined above. Using the correctness of LCS, we have $\ell \leq s$. Therefore, it follows that $s = \ell$.

   Running time: The two sorts take $\Theta(n \log n)$ time. The LCS takes $\Theta(n^2)$ time. So the running time of our algorithm is $\Theta(n^2)$.

   Solution 2: To find the LIS without using LCS, define the quantity $T(m)$ to be the length of the LIS of $A[1..m]$ that includes $A[m]$. The latter condition allows us to decide easily if a new element can be added to the existing LIS, since we know the last value of the LIS.

Then, the LIS of $A[1..n]$ is $\max_{1 \le m \le n} T(m)$.

The recurrence for $T(m)$ is : $T(m) = 1$ if $m = 1$, and
$T(m) = 1 + \max_j T(j)$ where $m > 1$, $1 \le j < m$ and $A[j] \le A[m]$.

The correctness proof is trivial, using the standard cut-and-paste argument and is omitted.

The numbers $T(j)$ are evaluated starting from $j = 1$ and ending in $j = n$. Evaluating $T(j)$ takes time $O(j)$ and thus the algorithm runs in time $\Theta(n^2)$.