

COT 5405 Homework 3 Solutions

February 11, 2009

1 Grading Policy

- Maximum Score is 100.
- Each completed question is worth 15 points.
- Question 3 is graded for additional 55 points. Each subpart is worth 15/15/10/15 points respectively.
- For exceeding the page requirement which is one sheet per question is penalized for 10 points.
- Please note that, no late submissions are accepted.
- If you have a similar answer to an answer from somewhere else or someone and if you did not give any citations/references/people who discussed related to your answer, you lost additional 30 points. Note that, you can discuss your answers or explore resources, but you need to give references to your sources. This penalty will be increased for the upcoming homeworks, in case it happens again.
- Please read the regrading policy from the course syllabus. Contact TA Hale Erten regarding issues about grading (via email or during her office hours).
- Please let us know, if you find any mistakes in this solution.

2 Problem 15-4

The problem can be transformed into a tree coloring problem where we consider the supervisor tree and color each node red if the employee is attending and white otherwise. The parent of a red node must be white (if an employee is attending, his/her immediate supervisor cannot attend). We wish to color the tree so that the sum of the conviviality of the nodes is maximized. Let $v = T(x, c)$ be the conviviality of the tree rooted at the node x that is colored with color c . We can construct the following recursion:

If x is a leaf node with conviviality v and color c then:

$$T(x, c) = \begin{cases} v, & \text{if } x = RED \\ 0, & \text{if } x = WHITE \end{cases}$$

If x is not a leaf node with conviviality v and color c then:

$$T(x, c) = \begin{cases} v + \sum_i T(x.child_i, WHITE), & \text{if } x = RED \\ \sum_i \max(T(x.child_i, RED), T(x.child_i, WHITE)), & \text{if } x = WHITE \end{cases}$$

The maximal conviviality v_{max} is then given by $v_{max} = \max(T(root, WHITE), T(root, RED))$. This algorithm will take $O(n)$ time.

3 Problem 16.2-2

The solution is based on the optimal-substructure observation in the text: Let i be the highest-numbered item in an optimal solution S for W pounds and items $1, \dots, n$. Then $S' = S - i$ must be an optimal solution for $W - w_i$ pounds and items $1, \dots, i - 1$, and the value of the solution S is v_i plus the value of the subproblem solution S' . We can express this relationship in the following formula: Let $c[i, w]$ to be the value of the solution for items $1, \dots, i$ and maximum weight w . Then

$$c[i, w] = \begin{cases} 0, & \text{if } i = 0 \text{ or } w = 0 \\ c[i - 1, w], & \text{if } w_i > w \\ \max(v_i + c[i - 1, w - w_i], c[i - 1, w]), & \text{if } i > 0 \text{ and } w - w_i \end{cases}$$

? The last case says that the value of a solution for i items either includes item i , in which case it is v_i plus a subproblem solution for $i - 1$ items and the weight excluding w_i , or does not include item i , in which case it is a subproblem solution for $i - 1$ items and the same weight. That is, if the thief picks item i , he takes v_i value, and he can choose from items $1, \dots, i - 1$ up to the weight limit $w - w_i$, and get $c[i - 1, w - w_i]$ additional value. On the other hand, if he decides not to take item i , he can choose from items $1, \dots, i - 1$ up to the weight limit w , and get $c[i - 1, w]$ value. The better of these two choices should be made. The algorithm takes as inputs the maximum weight W , the number of items n , and the two sequences $v = v_1, v_2, \dots, v_n$ and $w = w_1, w_2, \dots, w_n$. It stores the $c[i, j]$ values in a table $c[0..n, 0..W]$ whose entries are computed in row-major order. (That is, the first row of c is filled in from left to right, then the second row, and so on.) At the end of the computation, $c[n, W]$ contains the maximum value the thief can take.

DYNAMIC-0-1-KNAPSACK(v, w, n, W)

for $w = 0$ to W **do**

 do $c[0, w] = 0$

end for

for $i = 0$ to n **do**

 do $c[i, 0] = 0$

```

for  $w = 1$  to  $W$  do
  if  $w_i < w$  then
    if  $v_i + c[i - 1, w - w_i] > c[i - 1, w]$  then
       $c[i, w] = v_i + c[i - 1, w - w_i]$ 
    else
       $c[i, w] = c[i - 1, w]$ 
    end if
  else
     $c[i, w] = c[i - 1, w]$ 
  end if
end for
end for

```

The set of items to take can be deduced from the c table by starting at $c[n, W]$ and tracing where the optimal values came from. If $c[i, w] = c[i - 1, w]$, then item i is not part of the solution, and we continue tracing with $c[i - 1, w]$. Otherwise item i is part of the solution, and we continue tracing with $c[i - 1, w - w_i]$.

The above algorithm takes $\Theta(nW)$ time total: $\Theta(nW)$ to fill in the c table, where there are $(n + 1)(W + 1)$ entries, each requiring $\Theta(1)$ time to compute. $O(n)$ time to trace the solution (since it starts in row n of the table and moves up one row at each step).

4 Problem 16-1

Before we go into the various parts of this problem, let us first prove once and for all that the coin-changing problem has optimal substructure. Suppose we have an optimal solution for a problem of making change for n cents, and we know that this optimal solution uses a coin whose value is c cents; let this optimal solution use k coins. We claim that this optimal solution for the problem of n cents must contain within it an optimal solution for the problem of $n - c$ cents. We use the usual cut-and-paste argument. Clearly, there are $k - 1$ coins in the solution to the $n - c$ cents problem used within our optimal solution to the n cents problem. If we had a solution to the $n - c$ cents problem that used fewer than $k - 1$ coins, then we could use this solution to produce a solution to the n cents problem that uses fewer than k coins, which contradicts the optimality of our solution.

(a) The problem we wish to solve is making change for n cents. If $n = 0$, the optimal solution is to give no coins. If $n > 0$, determine the largest coin whose value is less than or equal to n . Let this coin have value c . Give one such coin, and then recursively solve the subproblem of making change for $n - c$ cents. To prove that this algorithm yields an optimal solution, we first need to show that the greedy-choice property holds, that is, that some optimal solution to making change for n cents includes one coin of value c , where c is the largest coin value such that $c \leq n$. Consider some optimal solution. If this optimal solution includes a coin of value c , then we are done. Otherwise, this optimal solution does not include a coin of value c . We have four cases to consider:

- If $1 \leq n < 5$, then $c = 1$. A solution may consist only of pennies, and so it must contain the greedy choice.
- If $5 \leq n < 10$, then $c = 5$. By supposition, this optimal solution does not contain a nickel, and so it consists of only pennies. Replace five pennies by one nickel to give a solution with four fewer coins.
- If $10 \leq n < 25$, then $c = 10$. By supposition, this optimal solution does not contain a dime, and so it contains only nickels and pennies. Some subset of the nickels and pennies in this solution adds up to 10 cents, and so we can replace these nickels and pennies by a dime to give a solution with (between 1 and 9) fewer coins.
- If $25 \leq n$, then $c = 25$. By supposition, this optimal solution does not contain a quarter, and so it contains only dimes, nickels, and pennies. If it contains three dimes, we can replace these three dimes by a quarter and a nickel, giving a solution with one fewer coin. If it contains at most two dimes, then some subset of the dimes, nickels, and pennies adds up to 25 cents, and so we can replace these coins by one quarter to give a solution with fewer coins.

Thus, we have shown that there is always an optimal solution that includes the greedy choice, and that we can combine the greedy choice with an optimal solution to the remaining subproblem to produce an optimal solution to our original problem. Therefore, the greedy algorithm produces an optimal solution. For the algorithm that chooses one coin at a time and then recurses on subproblems, the running time is $O(k)$, where k is the number of coins used in an optimal solution. Since $k \leq n$, the running time is $O(n)$.

(b) When the coin denominations are c^0, c^1, \dots, c^k , the greedy algorithm to make change for n cents works by finding the denomination c^j such that $j = \max\{0 \leq i \leq k : c^i \leq n\}$, giving one coin of denomination c^j , and recursing on the subproblem of making change for $n - c^j$ cents. (An equivalent, but more efficient, algorithm is to give n/c^k coins of denomination c^k and $(n \bmod c^k)/c^i$ coins of denomination c^i for $i = 0, 1, \dots, k-1$.) To show that the greedy algorithm produces an optimal solution, we start by proving the following lemma:

Lemma

For $i = 0, 1, \dots, k$, let a_i be the number of coins of denomination c^i used in an optimal solution to the problem of making change for n cents. Then for $i = 0, 1, \dots, k-1$, we have $a_i < c$.

Proof

If $a_i \geq c$ for some $0 \leq i < k$, then we can improve the solution by using one more coin of denomination c^{i+1} and c fewer coins of denomination c^i . The amount for which we make change remains the same, but we use $c - 1 > 0$ fewer coins.

To show that the greedy solution is optimal, we show that any non-greedy solution is not optimal. As above, let $j = \max\{0 \leq i \leq k : c^i \leq n\}$, so that the

greedy solution uses at least one coin of denomination c^j . Consider a non-greedy solution, which must use no coins of denomination c^j or higher. Let the non-greedy solution use a_i coins of denomination c_i , for $i = 0, 1, \dots, j - 1$; thus we have $\sum_{i=0}^{j-1} a_i c^i = n$. Since $n \geq c^j$, we have that $\sum_{i=0}^{j-1} a_i c^i \geq c^j$.

Now suppose that the non-greedy solution is optimal. By the above lemma, $a_i \leq c - 1$ for $i = 0, 1, \dots, j - 1$. Thus,

$$\begin{aligned} \sum_{i=0}^{j-1} a_i c^i &\leq \sum_{i=0}^{j-1} (c - 1) c^i \\ &= (c - 1) \sum_{i=0}^{j-1} c^i \\ &= (c - 1)(c^j - 1)/(c - 1) \\ &= c^j - 1 \\ &< c^j \end{aligned}$$

which contradicts our earlier assertion that $\sum_{i=0}^{j-1} a_i c^i \geq c^j$. We conclude that the non-greedy solution is not optimal.

Since any algorithm that does not produce the greedy solution fails to be optimal, only the greedy algorithm produces the optimal solution. The problem did not ask for the running time, but for the more efficient greedy-algorithm formulation, it is easy to see that the running time is $O(k)$, since we have to perform at most k each of the division, floor, and mod operations.

(c) With actual U.S. coins, we can use coins of denomination 1, 10, and 25. When $n = 30$ cents, the greedy solution gives one quarter and five pennies, for a total of six coins. The non-greedy solution of three dimes is better. The smallest integer numbers we can use are 1, 3, and 4. When $n = 6$ cents, the greedy solution gives one 4-cent coin and two 1-cent coins, for a total of three coins. The non-greedy solution of two 3-cent coins is better.

(d) Since we have optimal substructure, dynamic programming might apply. And indeed it does. Let us define $c[j]$ to be the minimum number of coins we need to make change for j cents. Let the coin denominations be d_1, d_2, \dots, d_k . Since one of the coins is a penny, there is a way to make change for any amount $j - 1$. Because of the optimal substructure, if we knew that an optimal solution for the problem of making change for j cents used a coin of denomination d , we would have $c[j] = 1 + c[j - d]$. As base cases, we have that $c[j] = 0$ for all $j \leq 0$.

To develop a recursive formulation, we have to check all denominations, giving

$$c[j] = \begin{cases} 0 & , \text{if } j \leq 0 \\ 1 + \min_{1 \leq i \leq k} c[j - d_i] & , \text{if } j > 0 \end{cases}$$

We can compute the $c[j]$ values in order of increasing j by using a table. The following procedure does so, producing a table $c[1..n]$. It avoids even examining $c[j]$ for $j \leq 0$ by ensuring that $j - d_i$ before looking up $c[j - d_i]$. The procedure also produces a table $denom[1..n]$, where $denom[j]$ is the denomination of a coin used in an optimal solution to the problem of making change for j cents.

```
COMPUTE-CHANGE(n, d, k)
for  $j = 1$  to  $n$  do
```

```

do  $c[j] = \infty$ 
for  $i = 1$  to  $k$  do
  if  $j \geq d_i$  and  $1 + c[j - d_i] < c[j]$  then
     $c[j] = 1 + c[j - d_i]$ 
     $denom[j] = d_i$ 
  end if
end for
end for
return  $c$  and  $denom$ 

```

This procedure obviously runs in $O(nk)$ time. We use the following procedure to output the coins used in the optimal solution computed by *COMPUTE-CHANGE*:

The initial call is *GIVE-CHANGE*($n, denom$). Since the value of the first parameter decreases in each recursive call, this procedure runs in $O(n)$ time.

```

GIVE-CHANGE(  $j$ ,  $denom$ )
if  $j > 0$  then
  give one coin of denomination  $denom[j]$ 
  GIVE-CHANGE(  $j - denom[j]$ ,  $denom$ )
end if

```

Note: Solutions given are not the only possible solutions, there can be many different solutions.