Homework 6 Solutions
Com S 311
Due: Dec 8, 11:59 pm

5% bonus for submission by Dec 7 , 11:59 pm

10% Penalty for late submissions: Dec 9, 11:59PM

Note: "Do not grade" option is back. You will receive 20% credit for writing "do not grade"

This HW is on dynamic programming. For each problem, give the recurrence and give an iterative algorithm based on the recurrence. Your algorithm must not be recursive or use memoization. State the run time. Part of the grade depends on the efficiency. Please provide pseudo-code, your pseudo code need not handle array index out of bounds exceptions. Each problem is worth 40 points.

1. Given a set of positive integers $\{x_1, x_2, \cdots, x_n\}$ where $x_1 = 1$ and a positive integer W find non-negative integers $w_1, \cdots, w_n$ such that

$$\sum_{i=1}^{n} w_i x_i = W$$

and

$$\sum_{i=1}^{n} w_i$$

is minimized.

Ans. This is the coin-change problem in disguise. Suppose we have coins with denominations $x_1, x_2, \cdots, x_n$. Given W pennies, the goal is to convert W pennies into coins of various denominations so that the total number of coins is minimized.

Consider the optimal solution. How many coins with denomination of x_n could be in the optimal solution? Suppose that somone told you that the optimal solution has a many x_n's. How does this hint help arrive at the optimal solution? Note that if optimal solution uses a many x_n's, then the remaining pennies is $W - ax_n$. Thus the goal is to find, the smallest number of coins needed to convert $W - ax_n$ pennies using coins $x_1, x_2, \cdots, x_{n-1}$.

However, we do not know the value of a. There are at most $W+1$ possibilities for a(in fact at most $W/x_n + 1$)–$0, 1, \cdots, W$. Whichever possibility leads to smallest number of possibilities, that is the optimal solution.

Let $C[i][j]$ denote the smallest number of coins needed to convert j pennies using $x_1, x_2, \cdots x_i$. The goal is to compute $C[n][W]$. Based on the above discussion, we arrive at the following recurrence.

$$C[i][j] = \min \begin{cases} C[i-1][j-x_i]+1 \\ C[i-1][j-2x_i]+2 \\ C[i-1][j-3x_i]+3 \\ \cdots \\ \cdots \\ C[i-1][[j-Wx_i]+W \end{cases} \tag{1}$$

Thus to compute $C[i][j]$ we need to know at most W values from row $i-1$. This gives us the following algorithm: Compute the first row of C and use the above equation to compute $C[i][j]$ for $2 \leq i \leq n$ and $1 \leq j \leq W$. Note that to compute $C[i][j]$ we need to find minimum of (at most) W value from the previous row. Thus each $C[i][j]$ can be computed in $O(W)$ time. Thus the total time taken is $O(nW^2)$.

The above algorithm tells us the minimum number of coins needed. We can convert this into an algorithm that tells how many of each denominations are needed. Note that if the minimum of

$$C[i-1][j-x_i]+1, C[i-1][j-2x_i]+2, C[i-1][j-3x_i]+3, \cdots, \cdots, C[i-1][[j-Wx_i]+W$$

is $C[i-1][[j-\ell 3x_i]+\ell$, then this means that an optimal solution for $C[i][j]$ will contain ℓ many x_i's. This information can be encoded into the matrix C. Where $C[i][j]$ is a tuple $\langle a, b \rangle$, where b denotes the smallest number of coins needed to convert j pennies using $x_1, \cdots x_i$, and a is the number of x_i's in such solution. Note that the value a can be computed by modifying the above algorithm slightly as follows. We use $C[i][j][1]$ to denote the vale of the first component of the tuple at $C[i][j]$ and $C[i][j][2]$ to denote the value of the second component of the tuple at $C[i]j]$. Now $C[i][j][2]$ is the minimum of

$$C[i-1][j-x_i]+1, C[i-1][j-2x_i]+2, C[i-1][j-3x_i]+3, \cdots, \cdots, C[i-1][[j-Wx_i]+W$$

and if the minimum is $C[i-1][j-\ell x_i]+\ell$, then set $C[i][j][1]$ to ℓ. This information enables us to compute the number of coins of each denomination. Look at $C[n][W][1]$ is this is ℓ_n, then $w_n = \ell_n$. Now look at $C[n-1][W-\ell x_n][1]$. This tells the number of x_{n-1}'s – thus $w_{n-1} = \ell_{n-1}$ and so on.

2. Let $U = \{x_1, \cdots, x_n\}$ be a set of non-negative integers and T and k be two integers. Give an algorithm that tests whether there is a subset S of U whose size is k and the sum of integers from S equal T.

   Ans. Consider the following recurrence. Let $S[i, \ell, W]$ be true if there is a subset R of $\{x_1, \cdots x_i\}$ with size ℓ such that sum of the elements in R is W.

   Our goal is to check if $S[n, k, T]$ is true or false. Suppose that U has a subset S of size k whose sum equal T, then either x_n is that set or not. If $x_n \in S$, then it must be the case there is a $k-1$ size subset of $\{x_1, \cdots, x_{n-1}\}$ whose sum equals $T-x_n$. In this case, $S[n-1, k-1, T-x_n]$ must be true. On the other hand if $x_n \notin S$, then there must be. k-size subset of $\{x_1, \cdots, x_n\}$

whose sum equals T. In this case, $S[n-1,k,T]$ must be true. Thus $S[n,k,T]$ is true if and only if at least one of $S[n-1,k-1.T-x_n]$ or $S[n-1.k-1,T]$ is true. In general

$$S[i,\ell,W] = S[i-1,\ell-1W-x_i] \vee S[i-1,\ell-1,W]$$

Based on this recurrence we arrive at the following iterative algorithm. Let M be a $n \times k \times W$ dimensional 3-D array initialized to False. Note that $S[1,\ell,W]$ is true if and only if $\ell = 1$ and $x_i = W$. Thus we initialize the array as follows:

```
For \ell  in the range [1, k]
  for W in the range [1, T]
    M[1, \ell, W] = True only if \ell== 1 and W= x_i
```

Now we have sufficient information to complete the array M.

```
for i in the range [2, n]
   for \ell in the range [1, k]
       for W in the range [1, T]
          M[i,\ell, W] = M[i-1, \ell-1, W-x_i] OR M[i-1, \ell-1, W]
```

Finally, we output True, if and only if $M[n,k,T]$ is True. The total time is $O(nkT)$

3. You are in a rectangular maze organized in the form of $M \times N$ cells/locations. You are starting at the upper left corner (grid location: $(1,1)$) and you want to go to the lower right corner (grid location: (M,N)). From any location, you can move either to the right or to the bottom, or go diagonal. I.e., from (i,j) you can move to $(i,j+1)$ or $(i+1,j)$ or to $(i+1,j+1)$. Cost of moving right or down is 2, while the cost of moving diagonally is 3. The grid has several cells that contain diamonds of whose value lies between 1 and 10. I.e, if you land in such cells you gain an amount that is equal to the value of the diamond in the cell. Once you reach the destination your earnings is the difference between earnings and costs. Give an algorithm that will output the maximum amount that you can earn. Your algorithm need not give the path.

   Ans. Let Z denote the rectangle grid. Let $V[i][j]$ denote the value of the diamond at $Z[i][j]$. Let $P[i][j]$ denote the maximum possible earnings when we start at $Z[1][1]$ and end at $Z[i][j]$. Our goal is to compute $P[M][N]$. How can we arrive at $Z[M][N]$? We can come from $Z[M-1][N]$ or $Z[M][N-1]$ or from $Z[M-1][N-1]$. Thus the best way to reach $Z[M][N]$ is the best of the following three: Find the best possible way to come to $Z[M-1][N]$ and go to $Z[M][N]$ or find the best path to $Z[M][N-1]$ and go to $Z[M][N]$, or find the best path to $Z[M-1][N-1]$ and go to $Z[M][N]$. This gives us the following recurrence.

$$P[i][j] = \max \begin{cases} P[i-1][j] + V[i][j] - 2 \\ P[i][j-1] + V[i][j] - 2 \\ P[i-1][j-1] + V[i][j] - 3 \end{cases} \quad (2)$$

Using this we arrive the following iterative algorithm. Note that $P[1][1] = V[1][1]$. The only way to arrive at $Z[1][i]$ is by taking $i-1$ steps to the right which incurs a cost of $2(i-1)$ and and gain of $\sum_{j=1}^{i} V[1][j]$. Thus

$$P[1][1] = \sum_{j=1}^{i} V[1][j] - 2(i-1)$$

Similarly

$$P[i][1] = \sum_{j=1}^{i} V[j][1] - 2(i-1)$$

Thus we initialize the first row and first column of P as above. Now for every $2 \leq i \leq M$ and for $2 \leq j \leq N$ we set

$$P[i][j] = \max \begin{cases} P[i-1][j] + V[i][j] - 2 \\ P[i][j-1] + V[i][j] - 2 \\ P[i-1][j-1] + V[i][j] + 3 \end{cases} \tag{3}$$

Finally we output $P[M][N]$. Clearly the runtime is $O(MN)$.