CSC373 Tutorial 3, Summer 2015

TA: Eric Zhu

Problems are based on tutorial exercises from previous offering of the course by Francois Pitt (Fall 2014).

Brief review of dynamic programming:

- Recursive structure of sub-problems
- Data structure for storing optimal values for sub-problems (why)
- Iterations to compute optimal values of sub-problems
- Reconstruction of optimal solution (s)

Making Change

Consider the problem of making change when the denominations are arbitrary.

Input: Positive integer "amount" n, positive integer "denominations" $d = [d_1, d_2, ..., d_m]$ and $d_1 < d_2 < ... < d_m$.

Output: **All possible ways** to make change: lists of "coins" $c = [c_1, c_2, ..., c_k]$ where each c_i is in d, repeated coins are allowed ($c_i = c_j$ for $i \neq j$), $\sum_{i=1}^{k} c_i = n$. If no solution is possible, $k = 0$ and the list is empty.

Recursive Structure

Finding a recursive structure of sub-problems - we need to divide a problem into smaller problems, and obtain the solutions based on the solutions of the smaller problems.

Problem: find solution giving amount n and denominations $d = [d_1, d_2, ..., d_m]$: $S(n, [d_1, ..., d_m])$

Sub-problems:

1. Find solutions giving amount n and denominations $d_1, d_2, ..., d_{m-1}$, i.e. solve the original problem without using d_m: $S(n, [d, ..., d_{m-1}])$

2. Find solutons giving amount $n - d_m$and denominations $d_1, d_2, ..., d_m$, i.e. use d_m once, and solve a smaller problem with the same denominations: $S(n - d_m, [d_1, ..., d_m])$

Solution: take union of the solutions for sub-problems 1 and 2:

$S(n, [d_1, ..., d_m]) = S(n, [d, ..., d_{m-1}]) \cup (S(n - d_m, [d_1, ..., d_m]) + [d_m])$ if $n - d_m \geq 0$, and

$S(n, [d_1, ..., d_m]) = S(n, [d, ..., d_{m-1}])$ if $n - d_m < 0$, as there is no solution to make negative change with positive coins.

For example:

$$
\begin{aligned}
S(5, [1, 2, 5]) &= S(5, [1, 2]) \cup (S(0, [1, 2, 5]) + [5]) \\
&= S(5, [1]) \cup (S(3, [1, 2])) \cup \{[] + [5]\} \\
&... \\
&= \{[1, 1, 1, 1, 1]\} \cup \{[1, 1, 1, 2], [1, 2, 2]\} \cup \{[5]\}
\end{aligned}
$$

We need a data structure to store the number of solutions to the sub-problems. Remember edit distance?

We can use a table (2-D array) of size $(n+1) \times m$, because there are 2 input variables (n and d) and all their values are discrete, and we need an additional row to store the sub-probelms involve zero amounts ($n = 0$).

What is the recurrence relation for the values in the table?

Example: $n = 5$ and $d = \{1, 2, 5\}$

Amount	1	2	5
0	1		
1			
2			
3			
4			
5			

Row i represents amount i, and column j represents using the denominations up to d_j. Each cell in the table stores the number of solutions to the sub-problem $S(i, [d_1, ..., d_j])$. For example, cell at row 0, column 1, represented as $(0, 1)$, stores the number of solutions to the sub-problem $S(0, [1])$, and the number of solutions is 1 - just use 0 coins to make $0 change. Cell $(1, 2)$ is $S(1, [1, 2])$, and so on.

Amount	1	2	5
0	1		
1	1		
2	1		
3	1		
4	1		
5	1		

Based on the division of sub-problems mentioned earlier, each cell in the table can be derived from computed cells.

For example, at cell $(1,1)$, $S(1,[1]) = S(1,[]) \cup S(0,[1])$. There is no solution to $S(1,[])$ as you cannot make change with no coins, and there is 1 solution to $S(0,[1])$ as stored in cell $(0,1)$. Thus, the number of solution to $S(1,[1])$ is $1+0=1$. Put a 1 at cell $(1,1)$. The first colum can be computed following the same pattern.

Amount	1	2	5
0	1	1	
1	1	1	
2	1		
3	1		
4	1		
5	1		

At cell $(1, 2)$, $S(1, [1, 2]) = S(1, [1])$. But the number solutions to $S(1, [1])$ is already stored in cell $(1, 1)$. Thus, the number of solutions just 1. Put a 1 at cell $(1, 2)$.

Amount	1	2	5
0	1	1	
1	1	1	
2	1	2	
3	1		
4	1		
5	1		

At cell $(2,2)$, $S(2,[1,2]) = S(2,[1]) \cup (S(0,[1,2]) + [2])$. The number solutions to $S(2,[1])$ is already stored in cell $(2,1)$, and the number of solutions to $S(0,[1,2])$ is stored in cell $(0,2)$. Thus, the number of solutions $1+1=2$. Put a 2 at cell $(2,2)$.

The complete steps for computing the table column-by-column.

Step 1:

Amount	1	2	5
0			
1			
2			
3			
4			
5			

Step 2:

Amount	1	2	5
0	1		
1	1		
2	1		
3	1		
4	1		
5	1		

Step 3:

Amount	1	2	5
0	1	1	
1	1	1	
2	1	2	
3	1	2	
4	1	3	
5	1	3	

Step 4:

Amount	1	2	5
0	1	1	1
1	1	1	1
2	1	2	2
3	1	2	2
4	1	3	3
5	1	3	4

The bottom-right cell stores the number of solutions to the original problem.

What is the running time?

What is the space (memory) requirement?

Reconstruct All Solutions

1. Store back trace pointers in each cell of the table, and each pointer points to the cell from which the current cell obtains a non-zero value.

2. Use depth-first search starting from the bottom-right cell to reconstruct the solutions

3. Moving left: solution skips the denomination of the current column

4. Moving up: solution uses one coin with the denomination of the current column

What is the running time of reconstruction? How to make it more efficient?

Hint: the reconstruction can be thought of as another dynamic programming problem.

Improve Efficiency

Can we do better than using $O(mn)$ space?

Yes! Because when we are working on the table one column at a time, only the previous column is required, and we do not need the values we have used.

So we can store only one column, and over-write each value as we compute the new one.

Example: $n = 5$ and $d = \{1, 2, 5\}$

Step 1:

Amount	
0	1
1	
2	
3	
4	
5	

Step 2:

Amount	
0	1
1	1
2	1
3	1
4	1
5	1

Step 3:

Amount	
0	1
1	1
2	2
3	2
4	3
5	3

Step 4:

Amount	
0	1
1	1
2	2
3	2
4	3
5	4

Can we store only one row? (No, why?)

How to modify the new algorithm to allow reconstruction of optimal solutions?