COMP 251: Data Structures and Algorithms Winter 2006

Dynamic Programming #1

Lecturer/TA: Ethan Kim

1 Introduction

Recall from previous lectures on divide and conquer that sometimes it is important to notice if the problem can be broken down to subproblems. To use the *divide and conquer* technique we asked ourselves the following question: Can we recursively solve a problem Π by dividing into subproblems $(\pi_1, \pi_2, \ldots)$ and merge the solutions to the subproblems? However, there are times when such approaches won't necessarily result in efficient algorithms. Consider the problem of computing the Fibonacci number $fib(n)$, for a positive integer n. By the definition $fib(n) = fib(n-1)+fib(n-2)$, we can design a simple divide-and-conquer-like algorithm as follows:

1. If $n = 1$ or $n = 2$ Then return 1.
2. Else return $fib(n-1) + fib(n-2)$.

A quick "trial by hand" tells us that this algorithm is quite inefficient.

$$\begin{aligned} fib(6) &= fib(5) + fib(4) \\ &= fib(4) + \underline{\text{fib(3) + fib(3)}} + fib(2) \quad \text{[overlapping subproblems!]} \\ &= fib(3) + fib(2) + fib(2) + fib(1) + \ldots \end{aligned}$$

The problem lies on the fact that there are overlapping subproblems in the recursion tree, and thus the algorithm solves the same subproblem repeatedly. A better algorithm is achieved by simply building the solutions for the subproblems in a *bottom-up* fashion:

1. Let A be an array of size n.
2. $A[1] := A[2] := 1$
3. For $i := 3$ to n
   - $A[i] = A[i-1] + A[i-2]$
4. return $A[n]$.

This is a (very simple) example of dynamic programming. By solving the subproblem from the bottom of the recursion tree, we never revisit the same subproblem again. Let's look at another example.

2 Binomial Coefficient

Recall that the binomial coefficient $\binom{n}{k}$ is defined as follows:

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 0 & \text{otherwise.} \end{cases}$$

Here, we will assume that the last case will not happen, since we can simply check if $0 \leq k \leq n$ and return 0 if it doesn't hold. As before, if we simply apply the spirit of divide and conquer approach, the following algorithm can be designed:

1. If $k = 0$ or $k = n$ Then return 1.
2. Else return $C(n-1, k-1) + C(n-1, k)$

Again, this algorithm would revisit the same subproblem over and over. Since the final result is obtained by adding up a number of 1's, the running time of the algorithm is in $\Omega(\binom{n}{k})$. Now, we wish to obtain a better (more effcient) algorithm by using dynamic programming – and this is by the famous *Pascal's triangle*. To compute the value $\binom{n}{k}$, we need to look at two previous values, namely $\binom{n-1}{k-1}$ and $\binom{n-1}{k}$. Thus, the algorithm would do the following:

1. For $i = 0$ to k
   $C[i][i] = 1$ [*Initialize the diagonal to* 1]
2. For $i = 0$ to n
   $C[i][0] = 1$ [*Initialize the leftmost column to* 1]
3. For $i = 2$ to n
   For $j = 1$ to $i - 1$ [*We only fill in below the diagonal.*]
   $C[i][j] = C[i-1][j-1] + C[i-1][j]$
4. return $C[n][k]$.

The algorithm clearly takes $O(nk)$.

3 Making Change

For this problem, imagine that you are working as a cashier at a local grocery store. Your manager tells you that, when customers pay for their groceries, you should give out their change using minimum number of coins(for whatever reason nobody understood). Being the employee of the month for last month, you would like to keep that reputation by following this order... You first try to use the greedy algorithm. That is, if the customer needs to receive $9.87, you would give a

$5, two $2 coins, a $1 coin, etc.. However, you quickly observe that this doesn't work in general. Let's look at how to solve this using dynamic programming.

First, for simplicity, assume that we have unlimited supply of each coins. The input to the problem is given as (n, S) and a weight function w, where n is the number of different coin types, S is the change you must make using those coins, and $w : \{1, \ldots, n\} \rightarrow Z^+$ is the value of each coin types. For instance, there may be 3 coin types, each worth $1, $4, $6, and we would want to make $8 using those coins. (Note: using the above mentioned greedy approach, you would make $8 by giving the customer one $6 and two $1's, which is clearly not minimum number of coins!)

Now, let's think about what it means to make S dollars using n coin types. To make the correct change, you have two choices:

1. You may decide not to use the last coin type. This means your problem is now a smaller subproblem: make S dollars using the first $n-1$ coin types.
2. You may decide to use the last coin type. Then, you can think of your problem as follows: make $S - w(n)$ dollars using all n coin types. Once this smaller problem is solved, we simply add one to the total number of coins used, since we just used the n^{th} coin type once.

Above observation leads us to a nice dynamic programming algorithm. Let's build our table. The table T has n rows and $S+1$ columns. For each cell $T[i][j]$, we store the minimum number of coins needed to make j dollars using *only the first i coin types*. First, initialize the table by setting $T[i][0] = 0$, since no coins are needed to make $0. For other values of i and j, either we decide not to use i^{th} coin type, in which case $T[i][j] = T[i-1][j]$, or we decide to use one of i^{th} coin type, in which case $T[i][j] = 1 + T[i, j - w(i)]$. Since we want to minimize the number of coins used, we choose the smaller of the two:

$$T[i][j] = min(\ T[i-1][j]\ ,\ 1 + T[i][j - w(i)]\ ).$$

However, there are some degenerate cases. If $i = 1$, $T[i-1][j]$ falls outside the table. This is absurd, since if we are at $i = 1$, and decide not to choose the i^{th} (first) type of coins, we cannot make any value of S. So, it is convenient to think of such elements as having the value $+\infty$. Similarly, if $j < w(i)$, $T[i][j - w(i)]$ falls outside the table. But this case is absurd, too, since if we want to make change for j dollars, we wouldn't give out a coin worth more than j dollars. Again, we think of such elements as having the value $+\infty$. Since we are looking for smaller of the two, we simply pick the one that does not have $+\infty$ value. If they both turn out to be $+\infty$, we just set $T[i][j] = +\infty$. We are now ready to give the full algorithm.

1. For $i = 1$ to n

   $T[i][0] = 0$

2. For $i = 1$ to n

   For $j = 1$ to S

   - If $i = 1$ and $j < w(i)$ Then $T[i][j] = +\infty$
   - Else If $i = 1$ Then $T[i][j] = 1 + T[1, j - w(i)]$
   - Else If $j < w(i)$ Then $T[i][j] = T[i-1][j]$
   - Else $T[i][j] = min(\ T[i-1][j]\ ,\ 1 + T[i][j - w(i)]\ )$

3. return $T[n][S]$.

The minimum number of coins required to make S dollars would be stored at $T[n][S]$. Note that the solution may not always exist. For instance, if we are to make a change of \$17 (an odd number), but all our denominations are even numbers, we cannot make the change at all. In such cases, our algorithm returns the artificial value of $+\infty$. The analysis of the algorithm is easy. We do constant amount of work for each element in the table, and thus the running time is $O(nS)$.

Now, one may consider a different (but more realistic) version of the problem: instead of only returning the minimum number of coins needed, we would like to know, for each coin type, exactly how many coins make up the desired amount. This is easy, since we have just done the work. Starting from $T[n][S]$, our algorithm would "walk-back" the path that led us to $T[n][S]$. If $T[n][S] = +\infty$, there is nothing to do. Otherwise, do the following starting from $T[n][S]$. Suppose we're at $T[i][j]$. The value we stored in $T[i][j]$ tells us the following two cases:

1. $T[i][j] = T[i-1][j]$
2. $T[i][j] = 1 + T[i][j - w(n)]$

If the first case is true, we don't need to give the i^{th} coins, so we go up to $T[i-1][j]$. If the second case is true, we give one coin of i^{th} type and move left to $T[i][j - w(i)]$. If both cases hold, we can take either paths, since both would end up being the same number of coins in total. Continuing in this way, we eventually reach the first column of T, and then the algorithm halts.

For the running time of this "walk-back" phase of the algorithm, we make two types of moves: 1) we make $n-1$ steps to the row above(corresponding to not using a coin of the current denomination), and 2) we make $T[n][S]$ steps to the left(corresponding to handing over a coin). Since each of these steps can be made in constant time, the time required is $\Theta(n + T[n][S])$.

3.1 Exercises

1. Consider a different version of the problem. Instead of minimizing the number of coins to give, we would like to know *how many different combinations* of the coins make up the given amount. Design an algorithm for this problem.

2. Another version of coin change problem: In the above discussion, we assumed that we have unlimited supply of coins for each coin type. Suppose otherwise. If we have finite number of coins per each type, how would you solve the problem? That is, given n(the number of different coin types), S(amount which we want to make using the coins), $w : \{1, \ldots, n\} \rightarrow Z^+$(value for each coin type), and $c : \{1, \ldots, n\} \rightarrow Z^+$ (the number of avaialbe coins for each type), compute the minimum number of coins needed to make S.