1 Review

1.1 Ideas

We have already seen in *divide and conquer* how we can divide the problem into subproblems, recursively solve the subproblems, and combine those solutions to get the answer for the original problem (think merge-sort).

But for some problems, this naive scheme leads to an exponential blow-up in time complexity (think about the recursive algorithm for Fibonacci numbers). To avoid recalculations, we can use a lookup table. This technique is known as *memoization.*

And to make it more effective, we can build the lookup table in a bottom-up fashion, until we reach the original problem. This approach is called *dynamic programming.*

1.2 Steps

1. Define a set of subproblems that can lead to the answer of the original problem. Think: what information would make solving this problem easier? For example, calculating the n-th Fibonacci number is much simpler if we already have the $(n-1)$-th and $(n-2)$-th numbers.
2. Try to find a relation between the subproblems you found above, and the current problem you're trying to solve. Write the recursion solution of the original problem from the solutions of subproblems.
3. Build the solution lookup table in a bottom-up fashion, until reaching the original problem.
   - Initialize the lookup table.
   - Fill the other terms through iteration by using the recurrence relation.

2 Exercises

2.1 Making change with coins

Exercise (Basic Problem)**.** *Given a general monetary system with M different coins of value $\{c_1, c_2, \ldots, c_M\}$, devise an algorithm that can make change for amount N using a minimum number of coins. What is the complexity of your algorithm?*
(Remember we leaned last time that the greedy algorithm does not work for such general cases.)

Solution.

1. Define $X[n]$ as the minimum number of coins needed to make change for amount n. How can we make this problem easier to solve? Well, in the same fashion that calculating the the n-th Fibonacci number is easy given the previous two, $X[n]$ is easy to calculate if we know $X[n-c_k]$ (the minimum number of coins needed to make change for $n-c_k$) for all k.
2. What's the relation? If we know the least number of coins to make $n-c_k$, we can just take the minimum and add 1. Let c_{min} be the smallest coin denomination. The recurion will then be:

$$X[n] = \begin{cases} 0 & n = 0 \\ \infty & n < c_{min} \\ \min_k\{X[n-c_k]\} + 1 & \text{otherwise} \end{cases}$$

3. Initialize $X[0] = 0$ and $X[n] = \infty$ for $n < c_{min}$. Then fill up the other $X[n]$ until $n = N$ using the above recurrence relation.

The time complexity of the algorithm is $O(MN)$. We have N inputs, and for each we take a min over up to M values.

The space complexity of the algorithm is $O(N)$ through straight-forward memoization.

Exercise (Extension)**.** *Change your algorithm to compute how many different possible ways of making changes for amount N with the given coins exist.*

Solution.

1. Define subproblem $X(m,n)$ as the number of different possible ways to make changes for amount n using coins $\{c_1, c_2, \ldots, c_m\}$. How can we make this problem easier to solve? Well, what if we just consider c_m? The number of ways to make change where the "last" coin we use is c_m depends just on the number of ways to make change using the coins $\{c_1, c_2, \ldots, c_{m-1}\}$.
2. What's the relation? The total number of ways is given by the sum of the number of ways to make change for $n, n-c_m, n-2c_m, \ldots, n-\lfloor\frac{n}{c_m}\rfloor c_m$ without using c_m (ie, using $\{c_1, c_2, \ldots, c_{m-1}\}$). Then we can write the recursion

$$X(m,n) = \begin{cases} 1 & n = 0, m = 0 \\ 0 & m = 0 \\ \sum_{0 \le k \le \lfloor n/c_m \rfloor} X(m-1, n-kc_m) & \text{otherwise} \end{cases}$$

   We discuss the base cases first. There's 1 way to make change for \$0 with no coins (the empty set). Secondly, it's impossible to create $n > 0$ change if no coins are available. The last equation means that to make changes of amount n with $\{c_1, c_2, \ldots, c_m\}$, we can either use 0 coins of c_m,1 coin of c_m, 2 coins of $c_m, \cdots, \lfloor n/c_m \rfloor$ coins of c_m, and use the $\{c_1, c_2, \ldots, c_{m-1}\}$ to make changes for the rest of them. The total number of ways to make change for n is then the sum of all of these options. This is analogous to the subset sum problem, and can guarantee no replicated combination.
3. Initialize $X[0][0] = 1$ and $X[0][n] = 0$ for $n > 0$. Then fill up the other $X[n][m]$ until $m = M$ and $n = N$ using the recurrence relation (for each n, calculate all m).

The complexity of the algorithm is $O(MN^2)$. There are a total of MN inputs to our function, and on each input, we take the sum of up to N elements.

The space complexity of the algorithm is $O(MN)$ using memoization. We can improve this to $O(N)$ if we note that $X(m,n)$ only depends on the values of $X(m-1,n')$.

2.2 Robot on a grid

Exercise (Basic problem). *Imagine a robot sitting on the upper-left corner of an $M \times N$ grid. The robot can only move in two directions at each step: right or down. Write a program to compute the number of possible paths for the robot to get to the lower-right corner. What is the complexity of your program?*

Solution.

1. Define $X[m,n]$ as the number of possible paths to square (m,n). The robot could only have come from $(m-1,n)$ or $(m,n-1)$, so this problem would be easy to solve if we knew $X[m-1,n]$ and $X[m,n-1]$.
2. The number of possible paths to the square (m,n) is simply the sum of the paths to $(m-1,n)$ and $(m,n-1)$ because the robot can only move down and right. The recursion will be:

$$X[m,n] = \begin{cases} 1 & n = 1 \text{ or } m = 1 \\ X[m-1,n] + X[m,n-1] & \text{otherwise} \end{cases}$$

3. Initialize $X[1,n] = X[m,1] = 1$, and fill up the other $X[m,n]$ until $m = M, n = N$. Fill up either column by column, row by row, or shell by shell outward from the top left corner.

The time complexity of the algorithm is $O(MN)$. We have a total of MN inputs and we spend constant time on each.

The space complexity of the algorithm is $O(\min\{M,N\})$. We can fill up either by column or by row by keeping around two columns/rows at a time.

Exercise (Mathematician's solution). *Can you derive a mathematical formula to directly find number of possible paths? What is the complexity for a computer program to compute this formula?*

Solution.
The robot needs to move $(M-1)+(N-1)$ steps to get to the lower-right corner, among which $M-1$ should be downwards and $N-1$ should be rightwards. So the problem is how many possible ways to pick $M-1$ steps among $(M-1)+(N-1)$ steps, which is

$$\binom{(M-1)+(N-1)}{M-1} = \frac{((M-1)+(N-1))!}{(M-1)!(N-1)!}$$

The complexity of computing this mathematical formula is $O(M+N)$ using DP as discussed in class.

The space complexity is $O(1)$.

Exercise (Extension). *Imagine that certain squares on the grid are occupied by some obstacles (probably your fellow robots from your union, but they don't move). Change your program to find the number of possible paths to get to the lower-right corner without going through any of those occupied squares.*

Solution.
The first and second step are the same as the basic problem. In the third step, we need to initialize the occupied squares to be zero, and remember not to update them in the recursion. The space complexity is $O(MN)$ since we might need to store information on all the occupied squares.

2.3 Balanced partition

Exercise (A wedge: positive integer subset sum problem). *Given a set of M positive integers $A = \{a_1, a_2, \ldots, a_M\}$, and a pre-defined number N, design a program to find out whether it is possible to find a subset of A such that the sum of elements in this subset is N.*
Example: $A = \{1, 4, 12, 20, 9\}$ and $N = 14$, we can find a subset $\{1, 4, 9\}$ such that $1 + 4 + 9 = 14$.

Solution. 1. Define

$$X(m,n) = \begin{cases} \text{true} & \text{if it is possible to find a subset of } \{a_1, a_2, \ldots, a_m\} \text{ such that the element sum is } n \\ \text{false} & \text{otherwise} \end{cases}$$

If it's possible to form n using just $\{a_1, a_2, \cdots, a_{m-1}\}$ then it's definitely possible to from n using the entire set. However, we also have the option of forming $n - a_m$ using $\{a_1, a_2, \cdots, a_{m-1}\}$, since then we can use a_m in our larger array to form n.

2. To combine the above, if either option works, then $X(m,n)$ is true, otherwise false. The recursion will be

$$X(m,n) = \begin{cases} \text{true} & n = 0 \\ \text{false} & m = 0, n > 0 \\ X(m-1, n) & a_m > n \\ X(m-1, n) \text{ or } X(m-1, n - a_m) & \text{otherwise} \end{cases}$$

3. Initialize $X(m, 0) = \text{true}, \forall m \geq 0; X(0, n) = \text{false}, \forall n \geq 1$, and fill up the other $X(m,n)$ until $m = M, n = N$.

The time complexity of this algorithm is $O(MN)$. There are a total of MN, each with a constant time computation.

The space complexity is $O(N)$ because $X(m,n)$ depends only on $X(m-1, n')$.

Exercise (Balanced partition problem). *Given a set of M positive integers $A = \{a_1, a_2, \ldots, a_M\}$, find a partition A_1, A_2, such that $|S_1 - S_2|$ is minimized, where S_i is the sum of elements in subset A_i. (A partition of A means two subset A_1, A_2 such that $A = A_1 \bigcup A_2$ and $A_1 \bigcap A_2 = \emptyset$.)*
Example: $A = \{1, 4, 12, 20, 9\}$.The best balanced partition we can find is $A_1 = \{1, 12, 9\}, S_1 = 1+12+9 = 22$ and $A_2 = \{4, 20\}, S_2 = 4 + 20 = 24$.

Solution.
Let $S = \sum_{m=0}^{M} a_m$, and the lookup table $X(m,n)$ in previous problem until $m = M, n = \lfloor S/2 \rfloor$, and find maximum n such that $X(M,n)$ is true. Then we define this set as A_1, and the complementary set as A_2. It can be easily shown that this partition is the most balanced.

2.4 Palindrome

A palindrome is a word (or a sequence of numbers) that can be read the same way in either direction, for example "abaccaba" is a palindrome.

Exercise (Palindrome)**.** *Design an algorithm to compute what is the minimum number of characters you need to remove from a given string to get a palindrome.*
Example: you need to remove at least 2 characters of string "abbaccdaba" to get the palindrome "abaccaba".

Solution.

1. Define $X(i,j)$ the minimum number of characters we need to remove in order to make substring S[i,i+1,...,j] a palindrome.
2. The recurion will be

$$X(i,j) = \begin{cases} 0 & i = j, i = j+1 \\ X(i+1, j-1) & \text{if S[i]=S[j]} \\ \min\{X(i+1,j), X(i,j-1)\} + 1 & \text{otherwise} \end{cases}$$

3. Initialize $X(i,i) = 0 = X(i+1,i), \forall i \geq 0$, and fill up the other $X(i,j)$ until $i = 0, j = \text{S.length} - 1$.

The complexity of this algorithm is $O(N^2)$, where $N = \text{S.length}$.

The space complexity is $O(N^2)$.