1 Review

1.1 Ideas

We already see in divide and conquer that how we can divide the problem into subproblems, recursively solve the subproblems, and combine those solutions to get the answer of the original problem. But for some problems, this scheme ends up into exponential growing complexity because we will be recomputing the answer of same subproblems over and over again (think about the recursive algorithm of Fibonacci numbers). To avoid recalculations, we use a lookup table. And to make it more effectively, we 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.
2. Write the recursion solution of the original problem from the solution 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 by recursion.

2 Exercises

2.1 Making change with coins

Exercise (Follow up the left-over problem of last section). Given a general monetary system with $M$ different coins of value $\{c_1, c_2, \ldots, c_M\}$, devise an algorithm that can make changes for amount $N$ using minimum number of coins. What is the complexity of your algorithm?

(Remember we learnt last time that greedy algorithm does not work for such general cases.)

Solution. 1. Define $X[n]$ is the minimum number of coins needed to make change of amount $n$.
   2. The recursion will be
      $$X[n] = \min\{X[n - c_1], X[n - c_2], \ldots, X[n - c_M]\} + 1$$
      Note that the minimum only consider coin $c_m$ if $n \geq c_m$. If $n < c_m$, $X[n - c_m]$ is considered as infinity.
   3. Initialize $X[0] = 0$, and fill up the other $X[n]$ until $n = N$.

The complexity of the algorithm is $O(MN)$. 
Exercise (Extension). Change your algorithm to compute how many different possible ways of making changes for amount \( N \) with the given coins.

Solution.
Define subproblem \( X(m, n) \) be the number of different possible ways to make changes of amount \( n \) using coins \( \{c_1, c_2, \ldots, c_m\} \). Then we can write the recursion

\[
X(m, n) = X(m-1, n) + X(m-1, n-c_m) + X(m-1, n-2c_m) + \cdots \nonumber
\]

\[
= \sum_{0 \leq k \leq \lfloor n/c_m \rfloor} X(m-1, n-kc_m) \nonumber
\]

The equation means to make changes of amount \( n \) with \( \{c_1, c_2, \ldots, c_m\} \), we can either use \( 0c_m, 1c_m, 2c_m, \ldots, \lfloor n/c_m \rfloor c_m \), and use the \( \{c_1, c_2, \ldots, c_{m-1}\} \) to make changes for the rest of them. This is analogous to the subset sum problem, and can guarantee no replicated combination.

We also need to be careful about the initial condition, which can be cleverly set as

\[
X(0, 0) = 1, X(0, n) = 0, \forall n \geq 1
\]

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 the lower-right corner. What is the complexity of your program?

Solution.
1. Define \( X[m, n] \) is the number of possible paths of getting square \((m, n)\).

2. The recursion will be

\[
X[m, n] = X[m-1, n] + X[m, n-1]
\]

3. Initialize \( X[1, n] = X[m, 1] = 1 \), and fill up the other \( X[m, n] \) until \( m = M, n = N \).

The complexity of the algorithm is \( O(MN) \).

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 compute to this formula?

Solution.
The robot needs to move \( (M-1) + (N-1) \) steps to get 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 \( O(M + N) \).
Exercise (Extension). Imagine that certain squares on the grid are occupied by some obstacles (probably your fellow robots, but they don’t move), change your program to find the number of possible paths to get the lower-right corner without going through any of those occupied squares.

Solution. The first and second step is 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 by recursion.

2.3 Multiplying matrices

Exercise (Multiplying matrices problem). Multiplying a series of matrices can often be a tedious task. However, recall that we know that matrix multiplication is associative. Because of this, we can greatly reduce the number of operations we need to do a computation. For example, suppose we were to do the multiplication \(ABC\), where \(A\) is a \(10 \times 4\) matrix, \(B\) is a \(4 \times 6\) matrix, and \(C\) is a \(6 \times 20\) matrix. Then working \((AB)C\) would require \(10 \times 4 \times 6 + 10 \times 6 \times 20 = 1440\) operations, while \(A(BC)\) would require \(4 \times 6 \times 20 + 10 \times 4 \times 20 = 1280\) operations. Design an algorithm that would return the minimum number of computations needed to multiply a series of matrices.

Solution. Suppose you have \(n\) matrices \(A[1], A[2], \ldots, A[n]\) where you desire to find \(A[1] \ast A[2] \ast \ldots \ast A[n]\). Define a lookup table \(B[i][j]\) that is an \(n \times n\) matrix where \(B[i][j]\) holds the number of operations needed to do the computation \(A[i] \ast A[i+1] \ast \ldots \ast A[j]\). \(B\) would first be initialized with \(B[i][i] = 0\). In order to make the computation for \(B[i][j]\), we can simply iterate through all possible \(i \leq k < j\) and determine the minimum possible value of \(B[i][k] + \text{row}(i) \ast \text{col}(k) \ast \text{col}(j) + B[k+1][j]\), where \(\text{row}(i)\) and \(\text{col}(i)\) return the number of rows or columns of the \(i\)th matrix, respectively. It follows rather naturally, whether one uses a top-down or bottom-up approach, that this algorithm runs in \(O(n^3)\) time.

2.4 Palindrome

A palindrome is a word (or a sequence of numbers) that can be read the same say 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 recursion will be

\[
X(i, j) = \begin{cases} 
X(i + 1, j - 1) & \text{if } S[i]=S[j] \\
\min\{X(i + 1, j), X(i, j - 1)\} + 1 & \text{otherwise}
\end{cases}
\]
3. Initialize \( X(i, i) = 0 = X(i + 1, i), \forall i \geq 0, \) and fill up the other \( X(i, j) \) until \( i = 0, j = S.\text{length} - 1. \) The complexity of this algorithm is \( O(N^2) \), where \( N = S.\text{length}. \)