1 Overview

1.1 Ideas

Dynamic Programming is a extremely useful problem solving technique that allows us to solve larger problems by breaking them into sub-problems and combining them together. We use a dictionary or look-up table to guarantee that we only solve each sub-problem *once*.

There are 2 ways that we usually implement DP algorithms:

- **Recursion with Memoization:** Start with a recursive function f. Every time we call f on an input, first check in a dictionary to see if f on that input has been calculated before. If so, return the answer immediately. If not, run f on this input normally and store the output in the dictionary.
- **Bottom Up Dynamic Programming:** We start with a multi-dimensional array X. We start filling in the entries of X one at a time depending on the recurrence that X satisfies. Eventually, we will fill in X on our desired input and then we stop.

Theoretically, the two strategies achieve the same complexity bounds, although bottom-up DP can be space saving in some situations. For this section, we will use *recursive function* f and lookup table X interchangeably.

1.2 Solution Organization

A good solution to a dynamic programming part will consist of the following 3 parts.

- **Definition:** Define your recursive function or look-up table in words and explain what each of its arguments are. Make sure to state what it outputs and what inputs you need to feed in to get the final answer.
  - *Good.* For the string reconstruction problem, let $D[i, j]$ be the boolean value indicating whether the substring of s from the ith character to the jth character has a concatenation of strings from the dictionary. $D[1, n]$ would give us our final answer.
  - *Bad.* Let $D[i, j]$ be whether a substring can be broken into words of the dictionary. (How do i and j relate to this substring? Is $D[i, j]$ a list of dictionary words or just a boolean value?)
- **Recursion:** Give both a verbal and mathematical description of the recursion used, including the base cases. A piecewise function is usually a good way to do this. Remember, you **must** include an English explanation of why your recursion works and is correct.
- **Analysis:** Include both a run-time analysis and a space analysis.
  - *Run-time.* The run-time of a DP algorithm can always be calculated by:

$$\text{Number of possible inputs} \cdot \text{Time to combine recursive calls}$$

For the string reconstruction problem, there are n^2 possible inputs, and each takes $O(n)$ time to take the boolean OR of up to n elements from the recursive calls.

- *Space.* The space complexity of a DP algorithm is always $O(\text{Number of possible inputs} \cdot \text{size of output})$. However, in some cases, it is possible to do better if you do not need to store *all* your recursive calls' answers to build up your solution. For example, in Fibonacci, you only need to store the last 2 Fibonacci numbers, even though your function has n possible inputs, so the space can be made $O(1)$ instead of $O(n)$.

1.3 How to Approach Problems

A common theme in Dynamic programming is the idea of *exhaustion*. I know I'll find the optimal answer because I'm taking the best solution out of all possible ways to get a solution.

(a) **String Reconstruction:** If it is possible to break this string into dictionary words, then the first of these words must end somewhere. Check all possible places where the first word could end, and recursively check if the remaining letters can be broken into strings.

(b) **Edit Distance:** If last characters of two string are different, then that difference must have come about through an insert, delete or replace. Check all three of those possibilities and recursive appropriately depending on which operation took place.

(c) **Shortest Paths:** The shortest path from u to v must have passed through one of the nodes w with the edge $w \to v$. Find the best previous node w in this shortest path by taking the minimum over all such possibilities of going from u to w in the shortest way possible and then finally $w \to v$.

2 Exercises

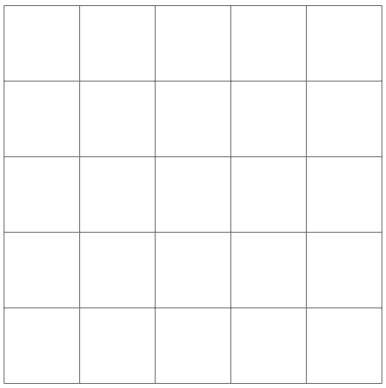
2.1 Robots on A Grid

Exercise 1. Imagine a robot sitting on the lower-left corner of an $M \times N$ grid. The robot can only move in two directions at each step: right or up.

(a) Design an algorithm to compute the number of possible paths for the robot to get to the top-right corner.

(b) Can you derive a mathematical formula to directly find the number of possible paths?

(c) Imagine that certain squares on the grid are occupied by some obstacles (probably your fellow robots, but they don't move). How should you modify your algorithm to find the number of possible paths to get the top right corner without going through any of those occupied squares?



Solution

(a) Let's label the grid as $(0, 0)$ at the bottom left and (m, n) at the top right.

- **Definition:** Define $X[m, n]$ to be the number of possible paths of getting square (m, n) starting from $(0, 0)$. Our goal is to calculate $X[M, N]$.
- **Recursion:** The recursion will be

$$X[m,n] = \begin{cases} 1 & m = 0, n = 0 \\ 0 & m < 0 \text{ or } n < 0 \\ X[m-1,n] + X[m,n-1] & \text{otherwise} \end{cases}$$

  In order to reach (m, n), you must have come in from $(m - 1, n)$ or $(n, m - 1)$. Those are the only possibilities, so we add up the solutions to those sub-problems.

- **Analysis:** There are MN states and each one takes $O(1)$ time to calculate (adding up 2 numbers). Thus, the run-time is $O(MN)$. If we filled out the table in a way such that we fill $X[m, \cdot]$ and then move on to $X[m + 1, \cdot]$, then we would only need to hold $O(N)$ elements at once. On the other hand, if we fill $X[\cdot, n]$ before moving on to $X[\cdot, n + 1]$, we will need $O(M)$ space. Therefore, the space complexity is $O(\min(M, N))$.

(b) 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 $O(M + N)$.

(c) Wherever a square is occupied, we set X of that square to be 0 and make sure that we never update it in the recursion.

2.2 Greedy.c Again

Exercise 2. In the last section, we explored the greedy algorithm for making change and saw that there exist coin denominations such that the greedy algorithm is not correct. Given a general monetary system with M different coins of value $\{c_1, c_2, \ldots, c_M\}$, devise an algorithm that returns

the minimum number of coins needed to make change for N cents. You may assume that $c_M \leq N$. How would you modify your algorithm to return the actual collection of coins?

Solution

- **Definition:** Let $X[n]$ be the minimum number of coins needed to make change for n cents. We would like to calculate $X[N]$.
- **Recursion:** We make the observation that if we took a coin out of the optimal solution, the remaining coins would still be the best way to make that many cents with these coins. Therefore, we exhaustively try to find the last coin that was used in the optimal solution:

$$X[n] = \begin{cases} \min\{X[n-c_1], X[n-c_2], \cdots, X[n-c_M]\} + 1 & n > 0 \\ 0 & n = 0 \\ \infty & n < 0 \end{cases}$$

  Note that we say $X[n] = \infty$ if $n < 0$ as a mathematical way to say that it is impossible to make change for a negative amount of coins. Since we are doing a min in our recursive case, we will never choose an impossible solution.

- **Analysis:** There are N possible inputs to X and it takes $O(M)$ time to compute each one (taking the minimum over M numbers). Therefore, the run-time complexity is $O(MN)$. For the space complexity, notice that in order to calculate $X[n]$, we only need the values $X[k]$ for k between $n - c_M$ and $n - c_1$. Therefore, we don't need to carry around all of X, but rather only the past c_M entries of X. The space complexity is therefore $O(c_M)$. Remember that we assumed in the problem $c_M \leq N$. Note that $O(N)$ is certainly correct as well, but it is not as tight of a bound as $O(c_M)$. As an extreme example, if someone asked you to make change for 9999 cents with pennies and nickles, the space you would need will be around 5, not 9999.

If we wanted the actual collection of coins used, we could make $X[n]$ return a *list* of coins used to make n cents with the minimum number of coins. In the first case with $n > 0$, we would change it so that we would try to find which of $\text{len}(X[n - c_i])$ is the smallest, and then append c_i onto $X[n - c_i]$.

Exercise 3. Here is a completely different problem about making change. The problem is to calculate the *number of ways* to make N cents using coins with denominations $\{c_1, c_2 \ldots c_M\}$. Note that we are not looking for the minimum number of coins.

(a) What is wrong with the following recurrence?

Let $X[n]$ be the number of ways to make change for n cents. Then,

$$X[n] = \begin{cases} \sum_{i=1}^{M} X[n - c_i] & n > 0 \\ 1 & n = 0 \\ 0 & n < 0 \end{cases}$$

The number of ways to make change for n cents is the sum of the number of ways to make change for $n - c_i$ cents, for $i = 1$ through m.

(b) Devise a correct algorithm to this problem.

Solution

(a) This grossly overcounts the number of ways to make change. The order of the coins we use should not matter, but this solution counts different orderings as different solutions.

(b)

- **Definition:** We can avoid overcounting by including another parameter in X which tells us which coins we are allowed to use. This way, all our solutions will use the coins in descending or ascending order, so every ordering of the coins will only be counted once. Let $X[n, m]$ be the number of ways to make n cents using only the coins $\{c_1, c_2 \dots c_m\}$. The answer we are looking for is $X[N, M]$.
- **Recurrence:** We can either use a coin of value c_m or we can choose not to and only use the remaining $m - 1$ coins. If we do, we still have the option of using any of the coins of value $\{c_1, c_2 \dots c_m\}$, but if we do not use a coin of value c_m, then we can only use coins of value $\{c_1, c_2 \dots c_{m-1}\}$. Thus, we get the following recurrence:

$$X[n, m] = \begin{cases} 1 & n = 0 \text{ and } m = 0 \\ 0 & n > 0 \text{ and } m = 0 \\ 0 & n < 0 \\ X[n - c_m, m] + X[n, m - 1] & \text{otherwise} \end{cases}$$

- **Analysis:** There are NM states of X and each one takes constant time to calculate, so this takes $O(NM)$ time. The space complexity is $O(N)$ because we see that $X[n, m]$ only depends on $X[\cdot, m]$ and $X[\cdot, m - 1]$, so when we are building up this table, we only need to keep 2 arrays of length $O(N)$.

2.3 Palindrome

Exercise 4. 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. 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

- **Definition:** 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. We are looking for $s[1, n]$.
- **Recusion:** The recursion will be

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

If $j - i \leq 1$, then we already have a palindrome. Otherwise, if the first and last letters do not

match, then we are forced to remove one of them. Recursively try to remove each of them and take the path that gives you the fewest number of letters removed later.

- **Analysis:** The run-time is $O(n^2)$ because there are n^2 states and each state takes $O(1)$ time to calculate. The space complexity is $O(n^2)$ and cannot be improved.

2.4 Boolean Parenthesization

Exercise 5. The boolean Parenthesization problem asks us to count the number of ways to fully parenthesize a boolean expression so that it evaluates to *true*. Let $T, F, \wedge, \vee, \oplus$ represent true, false, and, or, and xor respectively. For example, given the expression $T \oplus F \vee F$, there are 2 ways to make it evaluate to true, namely: $(T \oplus (F \vee F))$ and $((T \oplus F) \vee F)$.

Solution
Let the literals be labeled $x_1, x_2 \dots x_n$ and the operations $y_1, y_2 \dots y_{n-1}$. This implies that operation y_i is sandwiched between literals x_i and x_{i+1}.

- **Definition:** Let $T[i, j]$ be the number of ways to parenthesize the sub-expression from literal x_i to x_j, which would include y_i through y_{j-1} such that the boolean expression evaluates to *true*. Likewise, define $F[i, j]$ to be the same, except that the evaluation is *false*. Our desired answer is $T[1, n]$.

- **Recursion:** We try all operations that is the last to be used. For example, if we choose operation y_j, then we would parenthesize as such: $(x_1 y_1 x_2 \dots x_j) y_j (x_{j+1} y_{j+1} \dots x_n)$.

$$T[i,j] = \sum_{k=i}^{j-1} \begin{cases} T[i,k] \cdot T[k+1,j] & y_k = \wedge \\ T[i,k] \cdot T[k+1,j] + T[i,k] \cdot F[k+1,j] + F[i,k] \cdot T[k+1,j] & y_k = \vee \\ T[i,k] \cdot F[k+1,j] + F[i,k] \cdot T[k+1,j] & y_k = \oplus \end{cases}$$

$$F[i,j] = \sum_{k=i}^{j-1} \begin{cases} F[i,k] \cdot F[k+1,j] + T[i,k] \cdot F[k+1,j] + F[i,k] \cdot T[k+1,j] & y_k = \wedge \\ F[i,k] \cdot F[k+1,j] & y_k = \vee \\ F[i,k] \cdot F[k+1,j] + T[i,k] \cdot T[k+1,j] & y_k = \oplus \end{cases}$$

  The base cases here are $T[i, i] = 1$ if $x_i = T$ and 0 otherwise, as well as $F[i, i] = 1$ if $x_i = F$ and 0 otherwise.

- **Analysis:** We can have these two recursive functions call each other and that will build up the solutions to the entire T and F tables. We will not get stuck in an infinite loop because each time the recursion always occurs on inputs where $j - i$ strictly decreases. There are $2n^2$ states for the two functions, and each state takes $O(n)$ time to compute so the run-time is $O(n^3)$. The space is $2n^2 = O(n^2)$.