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(Number of possible inputs). 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

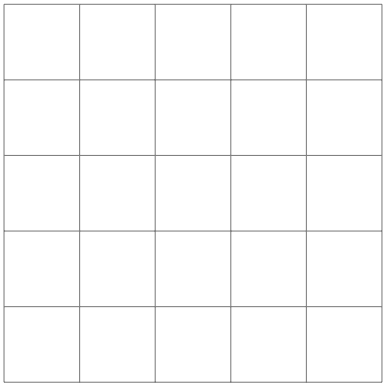
2.1 Robots on A Grid

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 left corner without going through any of those occupied squares?



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. How would you modify your algorithm to return the actual collection of coins?

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.

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".

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 $((T \oplus F) \vee F)$.