1 DP Overview

1.1 Ideas

Dynamic programming is an 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. Explain what parameters it has, what they represent, what the function itself represents, and what inputs are needed 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 $i$th character to the $j$th 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 runtime analysis and a space analysis. If specific data structures are necessary to improve computation speed, you should explicitly state them. If multiple solutions exist, choose the most time-efficient, and then choose the most space-efficient.
– **Runtime.** The runtime of a DP algorithm can 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 \(O(\text{number of possible inputs})\). 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 \(w \rightarrow v\). 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 \(w \rightarrow v\).

### 2 Randomized Algorithms Overview

#### 2.1 Ideas

In contrast to the deterministic algorithms we’ve seen thus far, randomized algorithms rely on random decision-making ("coin tosses"). Note that the algorithms’ input is *not* random and is always assumed to be worst-case.

There are two main types of randomized algorithms:

- **Las Vegas:** Runtime is a random variable, but correctness is certain. We aim to 1. bound maximum expected runtime or 2. prove that runtime is small with high probability.

- **Monte Carlo:** Correctness is random, but the runtime is fixed. We aim to 1. find a low runtime and 2. bound the maximum probability that the output is incorrect.

#### 2.2 Review Examples

(a) **Frievalds’:** Given three \(n \times n\) matrices \(A, B, C\), we want to check whether \(C = A \times B\). We do so by choosing \(k\) random binary vectors \(x^1, x^2, ..., x^k\) and outputting true if and only if \(C x^i = A B x^i\) for all \(i\). This is a Monte Carlo algorithm with deterministic runtime \(\Theta(kn^2)\) and probability at most \((\frac{1}{2})^k\) of failure.
(b) **QuickSort:** We want to sort an array $A$ of $n$ elements. We recursively choose a random pivot element, splitting the remaining elements into two subarrays depending on whether they are smaller or larger than the pivot, and recursively sort each subarray. This is a Las Vegas algorithm with expected runtime $\Theta(n \log n)$.

(c) **QuickSelect:** We want to choose the $k$th smallest element in an array $A$ of $n$ elements, where $1 \leq k \leq n$. Similarly to QuickSort, we choose a random pivot element, split the remaining elements into two subarrays depending on whether they are smaller or larger than the pivot, and search the array containing the $k$th smallest element. This is a Las Vegas algorithm with expected runtime $O(n)$.

### 3 Exercises

#### 3.1 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?

### Solution

(a) Let’s label the grid as $(0, 0)$ at the bottom left and $(M - 1, N - 1)$ at the top right.

- **Definition:** Define $X[m, n]$ to be the number of possible paths to reach 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 runtime 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 upper-right corner, among which \(M - 1\) should be upwards 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.

### 3.2 Greedy.c Again

**Exercise 2.** In the last section, we explored the greedy algorithm for making change and saw that there exists 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?

**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:

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

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 runtime complexity is \(O(MN)\). The space complexity is simply \(O(N)\).
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 \( X[n - c_i] \).

### 3.3 Palindrome

**Exercise 3.** 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. Design an algorithm to compute the minimum number of characters you need to remove from a given string to get a palindrome. For example, you need to remove at least 2 characters of the string “abbaccdaba” to get the palindrome “abaccaba”.

**Solution**

- **Definition:** Define \( X[i, j] \) as 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 \( s[1, n] \).
- **Recursion:** The recursion will be
  
  \[
  X[i, j] = \begin{cases} 
  0 & j - i \leq 0 \\
  X[i + 1, j - 1] & \text{if } S[i] = S[j] \\
  \min\{X[i + 1, j], X[i, j - 1]\} + 1 & \text{otherwise}
  \end{cases}
  \]

  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 runtime 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.

### 3.4 Boolean Parenthesization

**Exercise 4.** The boolean Parenthesization problem asks us to count the number of ways to fully parenthesize a boolean expression so that it evaluates to \text{true}. Let \( T, F, \land, \lor, \oplus \) represent \text{true}, \text{false}, and, or, and xor respectively. For example, given the expression \( T \oplus F \lor F \), there are 2 ways to make it evaluate to \text{true}, namely: \( (T \oplus (F \lor F)) \) and \( ((T \oplus F) \lor F) \).

**Solution**

Let the literals be labeled \( x_1, x_2, ..., x_n \) and the operations \( y_1, y_2, ..., y_{n-1} \). 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 \text{true}. Likewise, define \( F[i, j] \) to be the same, except that the evaluation is \text{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: \((x_1 y_1 x_2 \ldots x_j y_j (x_{j+1} y_{j+1} \ldots x_n)\).

\[
T[i, j] = \begin{cases} 
T[i, k] \cdot T[k + 1, j] & y_k = \land \\
T[i, k] \cdot T[k + 1, j] + T[i, k] \cdot F[k + 1, j] + F[i, k] \cdot T[k + 1, j] & y_k = \lor \\
F[i, k] \cdot T[k + 1, j] + T[i, k] \cdot T[k + 1, j] & y_k = \lor
\end{cases}
\]

\[
F[i, j] = \begin{cases} 
F[i, k] \cdot F[k + 1, j] + T[i, k] \cdot F[k + 1, j] + F[i, k] \cdot T[k + 1, j] & y_k = \land \\
F[i, k] \cdot F[k + 1, j] + T[i, k] \cdot T[k + 1, j] & y_k = \lor \\
F[i, k] \cdot F[k + 1, j] + T[i, k] \cdot T[k + 1, j] & y_k = \oplus
\end{cases}
\]

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 runtime is \( O(n^3) \). The space complexity is \( 2n^2 = O(n^2) \).

### 3.5 Probability Problems

**Exercise 5.** Suppose Alex’s closet contains \( n \) tech shirts. Each day, Alex randomly chooses one of the shirts to wear, and then puts it back in the closet without laundering. What is the expected number of days until Alex has worn all of the shirts at least once?

**Solution**

Let \( T \) be the total time it takes for Alex to wear all \( n \) shirts, and let \( t_i \) be the additional time it takes him to wear the \( i \)th unique shirt after \( i - 1 \) unique shirts have been worn. By linearity, and using the fact that \( t_i \sim Geom\left(\frac{n-(i-1)}{n}\right)\),

\[
E(T) = \sum_i E(t_i) = n \sum_i \frac{1}{i}
\]

This question is also known as the "coupon collector’s problem."

**Exercise 6.** This weekend, you decide to go to a casino and gamble. You start with \( k \) dollars, and you decide that if you ever have \( n \geq k \) dollars, you will take your winnings and go home. Assume that at each step you either win $1 or lose $1 (with equal probability).

(a) What is the probability that you lose all your money?

(b) How many steps are expected to occur before you either run out of money or earn \( n \) and decide to leave?
Solution

(a) Let $P_k$ be the probability that you win if you currently have $\$k$.

Set up a recurrence relation: $P_0 = 0$, $P_n = 1$, $P_k = 1/2P_{k+1} + 1/2P_{k-1}$. It seems like $P_k$ is the average of $P_{k-1}$ and $P_{k+1}$ for all $0 < k < n$, which suggests a linear function some sort. Thus, the linear function that would take $P_0 = 0$ and $P_n = 1$ is $P_k = k/n$, which we confirm to be a correct solution.

Solving, we find $P_k = k/n$, so you lose all your money with probability $1 - k/n$.

(b) Let $E_k$ be the expected number of steps remaining if you currently have $\$k$.

Set up a recurrence relation as in the previous problem: $E_0 = 0$, $E_n = 0$, $E_k = 1/2E_{k-1} + 1/2E_{k+1} + 1$. We guess the form of the solution to be a quadratic function, as each term is equal to the average of its two surrounding terms plus a constant. Seeing that $E_k$ has “roots” at $k = 0$ and $k = n$, our function must be of the form $E_k = ak(n - k)$ for some constant $a$. As we have

$$E_1 = \frac{1}{2}E_0 + \frac{1}{2}E_2 + 1 = \frac{1}{2}E_2 + 1,$$

we have $E_1 = a(n - 1)$ and $E_2 = 2a(n - 2)$. Solving, we see that $a = 1$. Thus, we see that $E_k = k(n - k)$.