Today we will cover dynamic programming questions, and you will get to solve them yourselves on LeetCode. Please create a LeetCode account beforehand on `https://leetcode.com/`.

1 Introduction

Dynamic programming is basically recursion, except you compute some of the answers to recursive subproblems in advance, because you think you will use them multiple times, and you want to save work.

As an example, consider the problem of computing the nth Fibonacci number. The Fibonacci numbers are

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_2 &= F_0 + F_1 = 1 \\ F_3 &= F_1 + F_2 = 2 \\ F_4 &= F_2 + F_3 = 3 \\ F_5 &= F_3 + F_4 = 5 \\ &\vdots \end{aligned}$$

After the first two Fibonacci numbers, each Fibonacci number is the sum of the previous two Fibonacci numbers.

1.1 Recursive solution

We can solve this problem recursively, as follows:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

While this solution is simple, it also takes $O(2^n)$ time, since each recursive call recurses on two subproblems that are almost as large as the original problem.

1.2 Dynamic programming solution

Fortunately, we can drastically reduce the amount of work by noticing that the recursion makes a lot of the same calls over and over again, and preemptively computing the lower

recursive calls in advance. The idea is to first compute `fib(0)` and `fib(1)`, and then use the results of these calls to compute `fib(3)`, and use the results of those calls to compute `fib(4)`, and keep preemptively computing the calls all the way up to `fib(n)`.

Consider the following solution:

```
def fib(n):
    A = [0, 1]
    for i in xrange(2, n + 1):
        A.append(A[i - 1] + A[i - 2])
    return A[n]
```

This solution runs in linear time, and computes the solution in a "bottom-up" manner.

1.3 More space-efficient solution

If you want to solve this problem in $O(1)$ space, you don't even have to store an array. You can write something along the lines of

```
def fib2(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    a_i_minus_2 = 0
    a_i_minus_1 = 1
    for i in xrange(2, n + 1):
        a_i = a_i_minus_1 + a_i_minus_2
        a_i_minus_2 = a_i_minus_1
        a_i_minus_1 = a_i
    return a_i
```

However, the array thing is more similar to other dynamic programming problems you will have to solve in this class, so I wanted to give you an example of an array.

1.4 Dynamic programming strategies

1. The first step is to decide what the subproblems are, and then write the recurrence relation that relates the answers to the larger subproblems to the answers to the smaller subproblems. At that point, you could solve the problem by recursion, but the dynamic programming approach is more efficient, since recursive calls to a given subproblem are only made once, instead of being repeated a lot.
2. Often you will want to populate an array, where each entry in the array represents a solution to a subproblem. In the simplest cases, the array will be one-dimensional.

However, sometimes the arrays might be two-dimensional, or three-dimensional, or there may be more than one array.

3. In order to populate the array, you should iterate through the entries, typically using a for loop. If you have a two-dimensional array, there should be two for loops.
4. Often the array size will be slightly larger than the total size of the problem.
5. Always remember to implement the base case!

2 Problems

For solutions and code, see the next page.

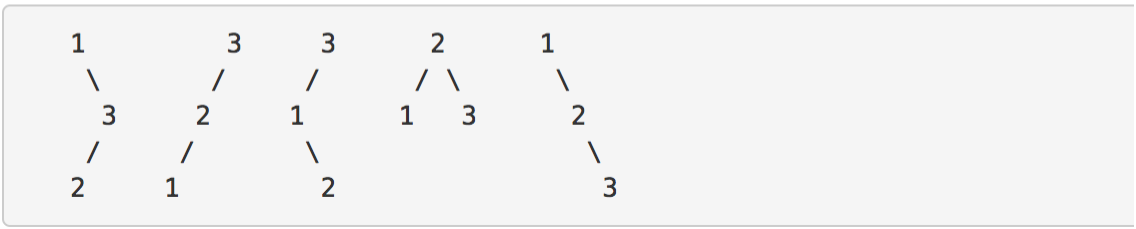
2.1 Unique Binary Search Trees:

`https://leetcode.com/problems/unique-binary-search-trees/#/description`

Given *n*, how many structurally unique **BST's** (binary search trees) that store values 1...*n*?

For example,

Given *n* = 3, there are a total of 5 unique BST's.



2.2 Edit Distance:

`https://leetcode.com/problems/edit-distance/#/description`

Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

a) Insert a character b) Delete a character c) Replace a character

3 Solutions

3.1 Unique binary search trees

Suppose you have a binary search tree with n distinct numbers. Let $A[n]$ be the number of such trees (where $A[0] = 1$). Any binary search tree must have a root. Once we have chosen the (zero-indexed) root $0 \leq k \leq n-1$, there are $n-1$ numbers left to place. k of those numbers must be in the left subtree, and $n-1-k$ numbers must be in the right subtree.

Therefore, once we have chosen the root k, there are $A[k]$ ways to arrange the left subtree, and $A[n-1-k]$ ways to arrange the right subtree. Summing over all possible roots, we get

$$A[n] = \sum_{k=0}^{n-1} A[k] * A[n-1-k]$$

We use this recurrence relation to compute $A[1], A[2], \ldots$ in sequential order, saving the values as we go along, until we reach $A[n]$.

3.2 Edit distance

As with the longest common subsequence problem, our subproblems will be prefixes of the strings A and B. Let $d[i, j]$ be the number of moves it takes to convert the string $A[0..i]$ to $B[0..j]$. As before, we concentrate on what we should do with the last letters of the string, a_n and b_m.

If $a_n = b_m$, the optimal thing to do is not to touch them, and just work with the letters before the last letter. So $d[n, m] = d[n-1, m-1]$.

If $a_n \neq b_m$, then we have three viable options. The first option is we can add the character b_m to the end of string A. (Adding any other character would be strictly less optimal.) The second option is to delete a character from the end of string A. Finally, we can change the last character of string A into the last character of string B. (Changing it into any other character would be strictly less optimal.)

If we take the first option, we have $d[n, m] = 1 + d[n, m-1]$. The second option gives $d[n, m] = 1 + d[n-1, m]$. The third option gives $d[n, m] = 1 + d[n-1, m-1]$. Finally, taking the minimum of these three options gives us the optimal edit distance.

4 Code

4.1 Unique binary search trees

```
def numTrees(n):
    A = [0 for i in range(n + 1)]
    A[0] = 1
    for i in xrange(1, n + 1):
        for k in xrange(0, i):
            A[i] += A[k] * A[i - 1 - k]
    return A[n]
```

4.2 Edit distance

```
def minDistance(word1, word2):
    d = [[0 for j in range(len(word2) + 1)] for i in range(len(word1) + 1)]
    for i in xrange(len(word1) + 1):
        d[i][0] = i
    for j in xrange(len(word2) + 1):
        d[0][j] = j

    for i in xrange(1, len(word1) + 1):
        for j in xrange(1, len(word2) + 1):
            if word1[i - 1] == word2[j - 1]:
                d[i][j] = d[i - 1][j - 1]
            else:
                d[i][j] = 1 + min(min(d[i - 1][j], d[i][j - 1]), d[i - 1][j - 1])

    return d[len(word1)][len(word2)]
```