Dynamic Programming Practice session

Omkar Deshpande

Richard Bellman

CHOICE OF THE NAME DYNAMIC PROGRAMMING

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes.

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities" (p. 159).

“Mathematics:

Some interviewers ask basic discrete math questions. This is more prevalent at Google than at other companies because we are surrounded by counting problems, probability problems, and other Discrete Math 101 situations. Spend some time before the interview refreshing your memory on (or teaching yourself) the essentials of combinatorics and probability. You should be familiar with n-choose-k problems and their ilk - the more the better.”

https://careers.google.com/how-we-hire/interview/#interviews-for-software-engineering-and-technical-roles

509. Fibonacci Number

Easy 279 160 Favorite Share

The **Fibonacci numbers**, commonly denoted `F(n)` form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from `0` and `1`. That is,

```
F(0) = 0,   F(1) = 1
F(N) = F(N - 1) + F(N - 2), for N > 1.
```

Given `N`, calculate `F(N)`.

Example 1:

```
Input: 2
Output: 1
Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1.
```

Example 2:

```
Input: 3
Output: 2
Explanation: F(3) = F(2) + F(1) = 1 + 1 = 2.
```

Leonardo of Pisa (1170-1250)

Pisa

Location of Pisa in Italy

The leaning tower of Pisa

Leonardo of Pisa
(1170-1250)

filius Bonaccio, “son of Bonaccio”: Fibonacci

Liber abaci (Book of Counting)

Chapter 1:

“These are the nine figures of the Indians:

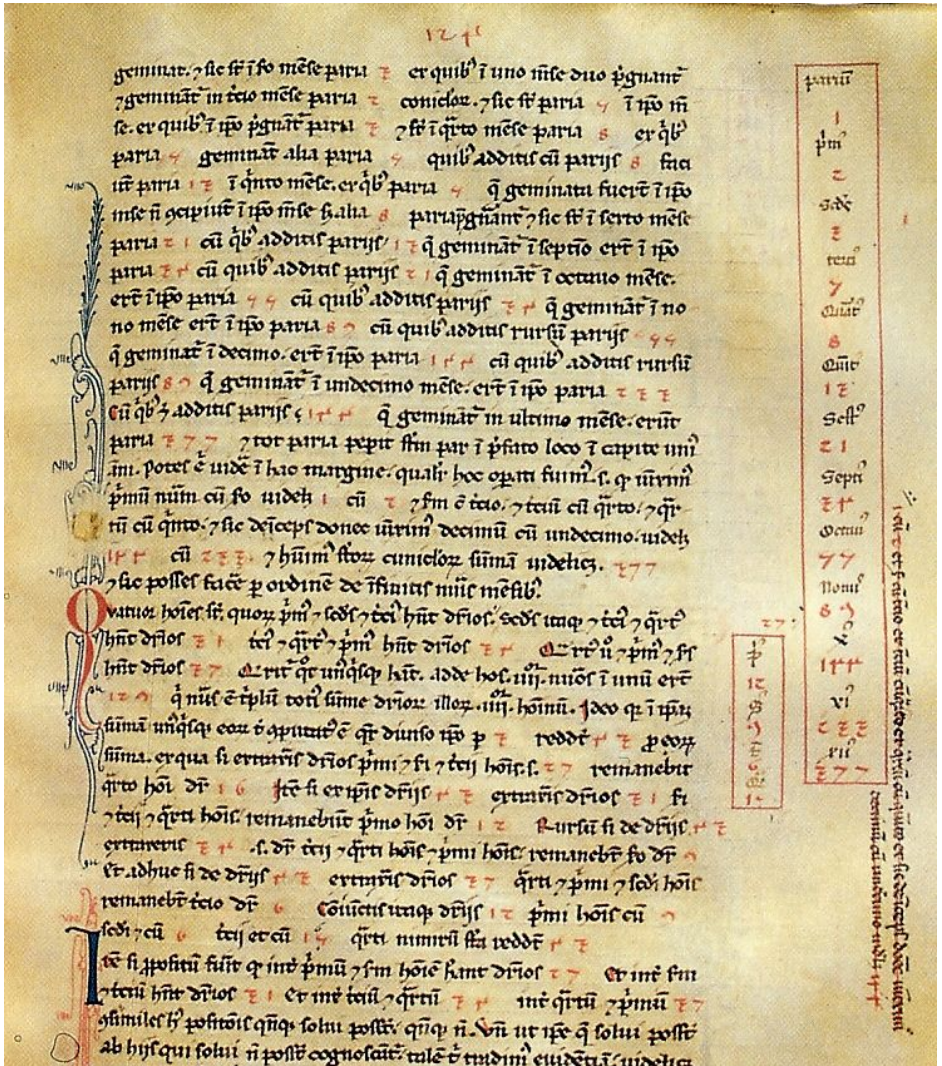
9 8 7 6 5 4 3 2 1.

With these nine figures, and with this sign 0… any number may be written, as will be demonstrated below.”

Liber abaci, chapter 12

A man put one pair of (newborn) rabbits in a certain place entirely surrounded by a wall. How many pairs of rabbits can be produced from that pair in a year, if the nature of these rabbits is such that every month, each pair bears a new pair which from the second month on becomes productive?

(Assume no rabbits die)



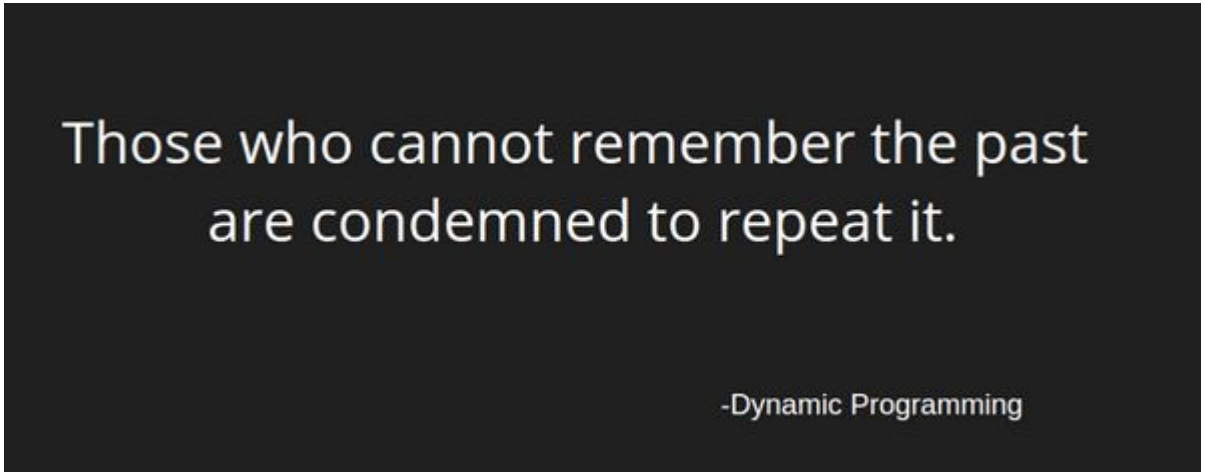
"It is ironic that Leonardo, who made valuable contributions to mathematics, is remembered today mainly because a 19th-century French number theorist, Édouard Lucas... attached the name Fibonacci to a number sequence that appears in a trivial problem in Liber abaci"

(Martin Gardner, Mathematical Circus)

New month’s adults = All the rabbit pairs from previous month

New month’s young = Previous month’s adults = Previous to previous month’s rabbit pairs

Exponential solution



Working definition: DP = Recursion without repetition

Memoized solution

DP solution that runs in linear time, linear space

DP solution that runs in linear time, O(1) space

70. Climbing Stairs

Easy 2667 94 Favorite Share

You are climbing a stair case. It takes *n* steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

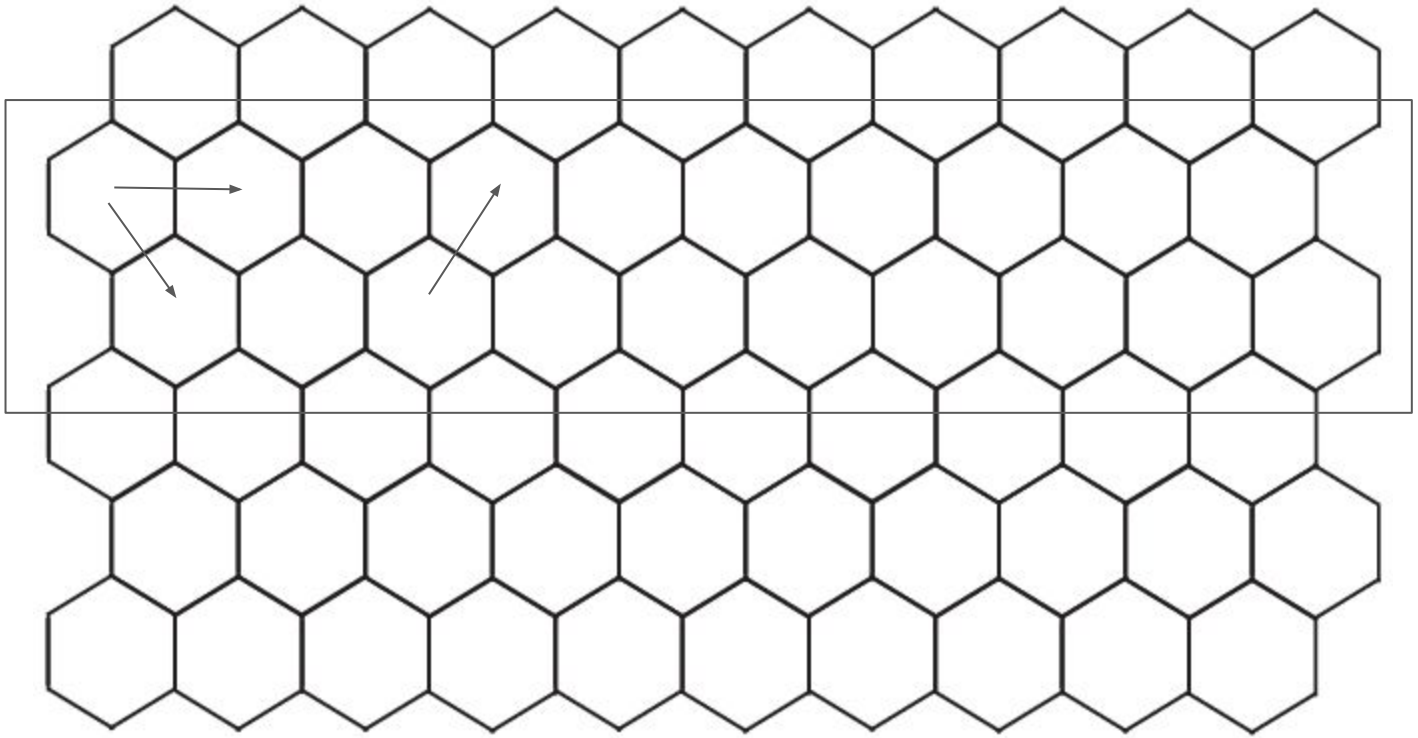
Note: Given *n* will be a positive integer.

Example 1:

```
Input: 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```



Tiling an n-board with squares, dominoes, and triominoes

1137. N-th Tribonacci Number

Easy 68 13 Favorite Share

The Tribonacci sequence T_n is defined as follows:

$T_0 = 0$, $T_1 = 1$, $T_2 = 1$, and $T_{n+3} = T_n + T_{n+1} + T_{n+2}$ for n >= 0.

Given `n`, return the value of T_n.

Example 1:

```
Input: n = 4
Output: 4
Explanation:
T_3 = 0 + 1 + 1 = 2
T_4 = 1 + 1 + 2 = 4
```

Example 2:

```
Input: n = 25
Output: 1389537
```

DP solution in O(n) time, O(1) space

790. Domino and Tromino Tiling

Medium 276 142 Favorite Share

We have two types of tiles: a 2x1 domino shape, and an "L" tromino shape. These shapes may be rotated.

```
XX  <- domino

XX  <- "L" tromino
X
```

Given N, how many ways are there to tile a 2 x N board? **Return your answer modulo 10^9 + 7**.

(In a tiling, every square must be covered by a tile. Two tilings are different if and only if there are two 4-directionally adjacent cells on the board such that exactly one of the tilings has both squares occupied by a tile.)

```
Example:
Input: 3
Output: 5
Explanation:
The five different ways are listed below, different letters indicates different tiles:
XYZ XXZ XYY XXY XYY
XYZ YYZ XZZ XYY XXY
```

Shortest path counting

A chess rook can move horizontally or vertically to any square in the same row or in the same column of a chessboard. Find the number of shortest paths by which a rook can move from one corner of a chessboard to the diagonally opposite corner

a) Use a dynamic programming algorithm
b) Use elementary combinatorics

62. Unique Paths

Medium 1903 133 Favorite Share

A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

Above is a 7 x 3 grid. How many possible unique paths are there?

Note: *m* and *n* will be at most 100.

Example 1:

```
Input: m = 3, n = 2
Output: 3
Explanation:
From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:
1. Right -> Right -> Down
2. Right -> Down -> Right
3. Down -> Right -> Right
```

Example 2:

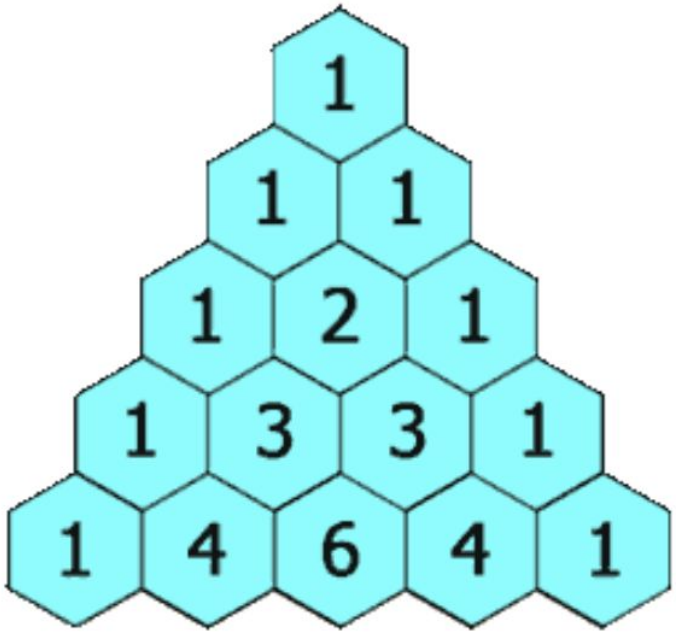
```
Input: m = 7, n = 3
Output: 28
```

```
8
9           table = [ [1 for j in range(m)] for i in range(n)]
10
11          for row in range(1,n):
12              for col in range(1,m):
13                  table[row][col] = table[row-1][col] + table[row][col-1]
14
15          return table[n-1][m-1]
16
```

118. Pascal's Triangle

Easy 848 84 Favorite Share

Given a non-negative integer *numRows*, generate the first *numRows* of Pascal's triangle.



In Pascal's triangle, each number is the sum of the two numbers directly above it.

Example:

```
Input: 5
Output:
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

Properties of Pascal's triangle

$C(n,k) = C(n-1,k) + C(n-1,k-1)$

Symmetry: $C(n,k) = C(n,n-k)$

Row sum = 2^n

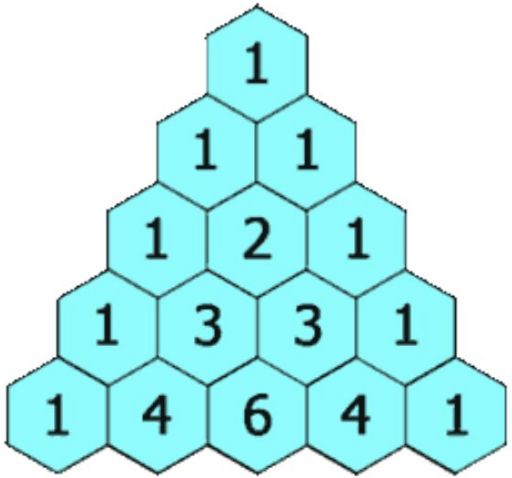
Hockey stick identity

119. Pascal's Triangle II

Easy 547 178 Favorite Share

Given a non-negative index k where $k \le 33$, return the k^{th} index row of the Pascal's triangle.

Note that the row index starts from 0.



In Pascal's triangle, each number is the sum of the two numbers directly above it.

Example:

```
Input: 3
Output: [1,3,3,1]
```

Binomial Coefficient | DP-9

Following are common definition of Binomial Coefficients.

1. A binomial coefficient C(n, k) can be defined as the coefficient of X^k in the expansion of (1 + X)^n.
2. A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k-element subsets (or k-combinations) of an n-element set.

The Problem

Write a function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k). For example, your function should return 6 for n = 4 and k = 2, and it should return 10 for n = 5 and k = 2.

Dynamic Programming Practice session 2

Omkar Deshpande

63. Unique Paths II

Medium 1007 169 Favorite Share

A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as `1` and `0` respectively in the grid.

Note: *m* and *n* will be at most 100.

Example 1:

Input:
```
[
  [0,0,0],
  [0,1,0],
  [0,0,0]
]
```
Output: 2
Explanation:
There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right

```
class Solution(object):
    def uniquePathsWithObstacles(self, obstacleGrid):
        """
        :type obstacleGrid: List[List[int]]
        :rtype: int
        """

        table = [ [0 for j in range(len(obstacleGrid[0]))] for i in range(len(obstacleGrid))]

        #First fill in the base cases
        if obstacleGrid[0][0] == 1:
            table[0][0] = 0
        else:
            table[0][0] = 1

        #Fill in row 0
        for col in range(1, len(obstacleGrid[0])):
            if obstacleGrid[0][col] == 1:
                table[0][col] = 0
            else:
                table[0][col] = table[0][col-1]

        #Fill in col 0
        for row in range(1, len(obstacleGrid)):
            if obstacleGrid[row][0] == 1:
                table[row][0] = 0
            else:
                table[row][0] = table[row-1][0]

        #Now do the main traversal
        for row in range(1, len(obstacleGrid)):
            for col in range(1, len(obstacleGrid[0])):
                if obstacleGrid[row][col] == 1:
                    table[row][col] = 0
                else:
                    table[row][col] = table[row-1][col] + table[row][col-1]
```

What if the grid was not regular but in the shape of a general graph?

From Counting problems to Optimization

“Principle of optimality”

64. Minimum Path Sum

Medium 1650 43 Favorite Share

Given a *m* x *n* grid filled with non-negative numbers, find a path from top left to bottom right which *minimizes* the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Example:

```
Input:
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
Output: 7
Explanation: Because the path 1→3→1→1→1 minimizes the sum.
```

Optimization problem

Optimal substructure + Overlapping subproblems

```python
    def minPathSum(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """

        table = [ [0 for j in range(len(grid[0]))]  for i in range(len(grid))]

        #Base cases
        table[0][0] = grid[0][0]

        #Row 0
        for col in range(1,len(grid[0])):
            table[0][col] = table[0][col-1] + grid[0][col]

        #Column 0
        for row in range(1, len(grid)):
            table[row][0] = table[row-1][0] + grid[row][0]

        #General traversal
        for row in range(1,len(grid)):
            for col in range(1, len(grid[0])):
                table[row][col] = grid[row][col] + min(table[row-1][col],table[row][col-1])

        return table[len(grid)-1][len(grid[0])-1]
```

746. Min Cost Climbing Stairs

Easy 1250 293 Favorite Share

On a staircase, the `i`-th step has some non-negative cost `cost[i]` assigned (0 indexed).

Once you pay the cost, you can either climb one or two steps. You need to find minimum cost to reach the top of the floor, and you can either start from the step with index 0, or the step with index 1.

Example 1:

```
Input: cost = [10, 15, 20]
Output: 15
Explanation: Cheapest is start on cost[1], pay that cost and go to the top.
```

Example 2:

```
Input: cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]
Output: 6
Explanation: Cheapest is start on cost[0], and only step on 1s, skipping cost[3].
```

```python
    def minCostClimbingStairs(self, cost):
        """
        :type cost: List[int]
        :rtype: int
        """
        cost.append(0)

        table = [0 for index in range(len(cost))]

        table[0] = cost[0]
        table[1] = cost[1]

        for i in range(2,len(table)):
            table[i] = cost[i] + min(table[i-1],table[i-2])

        return table[-1]
```

322. Coin Change

Medium 2270 83 Favorite Share

You are given coins of different denominations and a total amount of money *amount*. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return `-1`.

Example 1:

```
Input: coins = [1, 2, 5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
```

Example 2:

```
Input: coins = [2], amount = 3
Output: -1
```

Note:
You may assume that you have an infinite number of each kind of coin.

```python
    def coinChange(self, coins, amount):
        """
        :type coins: List[int]
        :type amount: int
        :rtype: int
        """

        table = [float("inf") for index in range(amount+1)]

        table[0] = 0

        for index in range(1,amount+1):
            mincoins = float("inf")
            for c in coins:
                if index - c >= 0:
                    mincoins = min(mincoins,table[index-c]+1)
            table[index] = mincoins

        if table[amount] == float("inf"):
            return -1
        else:
            return table[amount]

```

120. Triangle

Medium 1334 153 Favorite Share

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

```
[
     [2],
    [3,4],
   [6,5,7],
  [4,1,8,3]
]
```

The minimum path sum from top to bottom is `11` (i.e., **2** + **3** + **5** + **1** = 11).

Note:

Bonus point if you are able to do this using only *O*(*n*) extra space, where *n* is the total number of rows in the triangle.

```python
    def minimumTotal(self, triangle):
        """
        :type triangle: List[List[int]]
        :rtype: int
        """

        #Set up table with the same size as the triangular grid
        table = [ [0 for cols in range(rows+1) ] for rows in range(len(triangle))]

        #Set up base cases
        table[0][0] = triangle[0][0]

        for row in range(1, len(triangle)):
            #The left wall
            table[row][0] = table[row-1][0] + triangle[row][0]
            #The right wall
            table[row][row] = table[row-1][row-1] + triangle[row][row]

        #General traversal
        #Start from row 2. Within each row, the number of cols is row index + 1.
        #So the column index will vary from 1 to rowindex-1, having excluded the first and last columns
        for row in range(2, len(table)):
            for col in range(1,row):
                table[row][col] = triangle[row][col] + min(table[row-1][col-1], table[row-1][col])

        #The answer isn't the value in a particular cell, but the minimum of all values in the last row
        return min(table[-1])

```

198. House Robber

Easy 3051 95 Favorite Share

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

Example 1:

```
Input: [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
             Total amount you can rob = 1 + 3 = 4.
```

Example 2:

```
Input: [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
             Total amount you can rob = 2 + 9 + 1 = 12.
```

```python
    def rob(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """

        if len(nums) == 0:
            return 0
        if len(nums) == 1:
            return nums[0]

        table = [0 for index in range(len(nums))]
        #table[i] = max amount of money you can rob tonight from houses 0...i-1

        #Fill base cases
        table[0] = nums[0]
        table[1] = max(nums[0],nums[1])

        #Recursive case
        #The last house, say i, could either be robbed or not robbed.
        #If it is robbed, then the prev house cannot be robbed. So consult the solution from houses 0...i-2
        #If it is not robbed, then consult the solution from houses 0...i-1

        for i in range(2,len(nums)):
            table[i] = max(nums[i]+table[i-2],table[i-1])

        return table[len(nums)-1]
```

213. House Robber II

Medium 1065 38 Favorite Share

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**.

Example 1:

```
Input: [2,3,2]
Output: 3
Explanation: You cannot rob house 1 (money = 2) and then rob house 3 (money = 2),
             because they are adjacent houses.
```

Example 2:

```
Input: [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
             Total amount you can rob = 1 + 3 = 4.
```

```
    def rob(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """

        if len(nums) == 0:
            return 0
        if len(nums) == 1:
            return nums[0]
        if len(nums) == 2:
            return max(nums[0],nums[1])
        if len(nums) == 3:
            return max(nums[0],nums[1],nums[2])
        if len(nums) == 4:
            return max(nums[0]+nums[2],nums[1]+nums[3])

        table = [0 for index in range(len(nums))]
        #We need to somehow break the circle into a line
        #Let's say the houses were numbered 0...n-1
        #Case 1: Suppose the first house, 0, was robbed.
        #In that case, house n-1 couldn't be robbed, and house 1 couldn't be robbed. We have to consult the best solution for houses 2...n-2 treating them as if they lie along a line
        table[2] = nums[2]
        table[3] = max(nums[2],nums[3])
        for i in range(4,len(nums)-1):
            table[i] = max(nums[i] + table[i-2],table[i-1])

        case1 = nums[0] + table[len(nums)-2] #The solution for this scenario

```

```
31          #Case 2: Now suppose the first house was not robbed.
32          #In that case, we have to consult the best solution for houses 1...n-1 as if they lie along a line
33          table[1] = nums[1]
34          table[2] = max(nums[1],nums[2])
35          for i in range(3,len(nums)):
36              table[i] = max(nums[i] + table[i-2],table[i-1])
37
38          case2 = table[len(nums)-1] #The solution for the second scenario
39
40          return max(case1,case2) #Optimal solution is the best of the two scenarios
41
```

983. Minimum Cost For Tickets

Medium 668 15 Favorite Share

In a country popular for train travel, you have planned some train travelling one year in advance. The days of the year that you will travel is given as an array `days`. Each day is an integer from `1` to `365`.

Train tickets are sold in 3 different ways:

- a 1-day pass is sold for `costs[0]` dollars;
- a 7-day pass is sold for `costs[1]` dollars;
- a 30-day pass is sold for `costs[2]` dollars.

The passes allow that many days of consecutive travel. For example, if we get a 7-day pass on day 2, then we can travel for 7 days: day 2, 3, 4, 5, 6, 7, and 8.

Return the minimum number of dollars you need to travel every day in the given list of `days`.

Example 1:

```
Input: days = [1,4,6,7,8,20], costs = [2,7,15]
Output: 11
Explanation:
For example, here is one way to buy passes that lets you travel your travel plan:
On day 1, you bought a 1-day pass for costs[0] = $2, which covered day 1.
On day 3, you bought a 7-day pass for costs[1] = $7, which covered days 3, 4, ..., 9.
On day 20, you bought a 1-day pass for costs[0] = $2, which covered day 20.
In total you spent $11 and covered all the days of your travel.
```

Example 2:

```
Input: days = [1,2,3,4,5,6,7,8,9,10,30,31], costs = [2,7,15]
Output: 17
Explanation:
For example, here is one way to buy passes that lets you travel your travel plan:
On day 1, you bought a 30-day pass for costs[2] = $15 which covered days 1, 2, ..., 30.
On day 31, you bought a 1-day pass for costs[0] = $2 which covered day 31.
In total you spent $17 and covered all the days of your travel.
```

Note:

1. `1 <= days.length <= 365`
2. `1 <= days[i] <= 365`
3. `days` is in strictly increasing order.
4. `costs.length == 3`
5. `1 <= costs[i] <= 1000`

```python
    def mincostTickets(self, days, costs):
        """
        :type days: List[int]
        :type costs: List[int]
        :rtype: int
        """

        table = [0 for i in range(len(days))]

        table[0] = min(costs) #If it was only a 1-day travel plan, then pick the lowest cost ticket

        for i in range(1, len(days)):
            #table[i] to be filled with the cost of travel on days[0..i]
            #days[i] can be covered by a 1-day/7-day/30-day pass

            #If day i is covered by a 1-day pass, then we have to pick the optimal solution for all previous days
            case1 = table[i-1] + costs[0]

            #If day i is covered by a 7-day pass, then we have to assume that the 6 previous days were also covered by it.
            j = i-1
            while j >= 0 and days[j] >= days[i] - 6:
                j -= 1
            if j >= 0:
                case2 = table[j] + costs[1]
            else:
                case2 = costs[1]
```

```
29              #If day i is covered by a 30-day pass, then we have to assume that the 29 previous days were also covered by it.
30              j = i-1
31              while j >= 0 and days[j] >= days[i] - 29:
32                  j -= 1
33              if j >= 0:
34                  case3 = table[j] + costs[2]
35              else:
36                  case3 = costs[2]
37
38              table[i] = min(case1, case2, case3)
39
40          return table[len(days)-1]
41
```

Dynamic Programming Practice session 3

Omkar Deshpande

Recap

Dynamic Programming (bottom-up tabulation) on two kinds of problems:

1. (Non-optimization) counting problems
   - Overlapping subproblems
2. (Combinatorial Optimization) problems where we found the “best” sequence (one that maximized or minimized some objective function)
   - Overlapping subproblems
   - Optimal substructure (required for correctness)

Both these problems would otherwise (without DP) lead to exponential time complexity, since the number of possible sequences to consider are far too many. Caching the results of overlapping subproblems is necessary to curb it.

Non-optimization counting problems

1. Number of ways to climb n stairs (1D)
2. Number of ways to go across two rows of a hexagonal grid using n moves to the right. (1D)
3. Number of ways to tile an n-board using squares, dominoes and trominoes. (1D)
4. Number of ways to tile a 2 x n board using dominoes and L-shaped trominoes. (1D)
5. Number of unique paths from the top-left to bottom-right corner of a grid. (2D)
6. Number of unique paths from the top-left to bottom-right corner of a grid with obstacles at arbitrary locations. (2D)

In every case, exponential time complexity is curbed via DP to get a polynomial time algorithm (or pseudo-polynomial time algorithm).

Also applies to

- Computing the nth Fibonacci number (classic 1D DP)
- Computing C(n,k) (classic 2D DP)

Combinatorial Optimization problems

The adjective “combinatorial” means that without DP, the combinatorial explosion of possibilities would lead to an exponential time complexity.

After all, we are optimizing permutations and combinations (which are “combinatorial objects”, always exponential in number) by trying to pick the “best”.

Combinatorial optimization problems

1. Find the min cost path from the top left to bottom right corner of a grid (2D)
2. Find the min cost path up a flight of n stairs. (1D)
3. Find the min number of coins needed to make up change for a specified amount of money. (1D)
4. Find the min post path from the top tip to the bottom row of a triangular grid (2D)
5. **Find the maximum amount of money you can rob from non-adjacent houses in a row (1D)**
6. **Find the maximum amount of money you can rob from non-adjacent houses in a circle (1D)**
7. Find the min cost itinerary to travel by train through a country using 1-day/7-day/30-day passes (1D)

f(i,j) = min cost from the top-left corner to cell (i,j)
f(i,j) = min[f(i-1,j), f(i,j-1)] + cost of cell (i,j)

f(i) = min cost of starting from below and climbing up to stair i
f(i) = min over all preceding steps j [f(j)] + cost of stair i

f(a) = min number of coins needed to make up amount a
f(a) = min $[f(a - c_i)]$ (for all coin denominations c_i) + 1

f(r,k) = min cost path from the top tip of the triangle to row r, entry k
f(r,k) = min [f(r-1,k) + f(r-1,k-1)] + cost of cell (r,k)

f(i) = maximum amount of money you can rob from houses 0..i
f(i) = max [f(n-2) + cash in house i, f(n-1)]
(for both versions of the house robbery problem)

f(i) = min cost itinerary for day[0]....day[i] (note: the travel days may not be contiguous)
f(i) = min [f(i-1) + cost of ending 1-day pass, f(j) + cost of ending 7-day pass, f(k) + cost of ending 30-day pass] where j,k are the rightmost days not covered by the ending 7-day/30-day pass

Recall: Recursion class

When we enumerated permutations and combinations, we treated them like a **sequence** of blanks to be filled in : __ __ __ __ __ __ __ __ from left to right.

Here, we are trying to identify the “best” (max/min) sequence of blanks, among an exponential number of possibilities. Combinatorial enumeration can’t usually be done in polynomial time, but combinatorial optimization can be, IF there is optimal substructure and overlapping subproblems. This allows us to use dynamic programming.

Branch-and-Bound doesn’t generally lead to polynomial-time worst case complexity, and Greedy strategy often doesn’t work (and always needs a proof wherever it works)

We imagined an optimal solution laid out in a sequence.

We argued that if the last (right-most) element of the optimal solution was examined, there might be a bunch of options for what that rightmost element could be (in general): say the options are case1, case2, case3, …, case k. For each option, look at what the preceding element could be, and formulate the objective function accordingly, for a general preceding element or prefix of the optimal solution.

Since the prefixes are also optimal (optimal substructure), we could use the prefix solutions to figure out the overall optimal solution.

In this way, we used a decrease-and-conquer strategy to write a recursive formula for the optimal solution (and in general for any prefix of the optimal solution) in terms of optimal solutions for the smaller prefixes (which are the smaller subproblems).

We visualized the different subproblems as vertices in a dependency graph, a DAG. The dependency arrows went from smaller subproblems to larger subproblems.

We saw that the solutions to the subproblems (or prefixes) need to be built up in a convenient topological sort order.

We allocated a table with enough space to hold the solutions to all the subproblems.

We first filled in the base case values at the extreme end of the table, and then for each subsequent entry in the interior of the table, used the recursive formula to calculate and fill in the optimal values for larger subproblems, until the optimal value for the original problem got filled in at the end (or at least can be calculated from the filled table at the end).

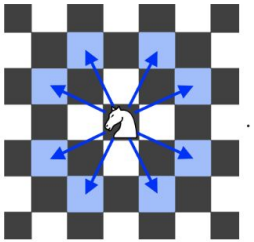
We analysed the time and space complexity of the algorithm.

Note: The time complexity for filling in the table = The time complexity for processing the DAG in topological sort order.

935. Knight Dialer

Medium 309 108 Favorite Share

A chess knight can move as indicated in the chess diagram below:



1	2	3
4	5	6
7	8	9
	0	

This time, we place our chess knight on any numbered key of a phone pad (indicated above), and the knight makes `N-1` hops. Each hop must be from one key to another numbered key.

Each time it lands on a key (including the initial placement of the knight), it presses the number of that key, pressing `N` digits total.

How many distinct numbers can you dial in this manner?

Since the answer may be large, **output the answer modulo** `10^9 + 7`.

Example 1:

```
Input: 1
Output: 10
```

```python
    def knightDialer(self, N):
        """
        :type N: int
        :rtype: int
        """

        #f(n,d) = Number of distinct n-digit numbers which end at digit d
        table = [ [0 for _ in range(10) ] for _ in range(N+1)]

        for digit in range(10):
            table[1][digit] = 1

        for i in range(2,N+1):
            table[i][0] = table[i-1][4] + table[i-1][6]
            table[i][1] = table[i-1][8] + table[i-1][6]
            table[i][2] = table[i-1][7] + table[i-1][9]
            table[i][3] = table[i-1][4] + table[i-1][8]
            table[i][4] = table[i-1][3] + table[i-1][9] + table[i-1][0]
            table[i][5] = 0
            table[i][6] = table[i-1][0] + table[i-1][7] + table[i-1][1]
            table[i][7] = table[i-1][2] + table[i-1][6]
            table[i][8] = table[i-1][1] + table[i-1][3]
            table[i][9] = table[i-1][4] + table[i-1][2]

        return sum(table[-1]) % (10**9 + 7)

```

276. Paint Fence

Easy 483 143 Favorite Share

There is a fence with n posts, each post can be painted with one of the k colors.

You have to paint all the posts such that no more than two adjacent fence posts have the same color.

Return the total number of ways you can paint the fence.

Note:
n and k are non-negative integers.

Example:

```
Input: n = 3, k = 2
Output: 6
Explanation: Take c1 as color 1, c2 as color 2. All possible ways are:

            post1  post2  post3
 -----      -----  -----  -----
   1         c1     c1     c2
   2         c1     c2     c1
   3         c1     c2     c2
   4         c2     c1     c1
   5         c2     c1     c2
   6         c2     c2     c1
```

```python
    def numWays(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: int
        """

        if n == 0:
            return 0

        same = [0 for i in range(n)] #same[i] = Number of ways to paint the fence so that posts i-1 and i have same color
        different = [0 for i in range(n)] #different[i] = #ways to paint fence s.t. posts i-1 and i have different colors
        total = [0 for i in range(n)] #Number of ways to paint the fences from 0 to i

        same[0] = 0 #A single post, hence no prev post to color with the same color. This is like a domino tile
        different[0] = k
        total[0] = k

        for i in range(1,n):
            same[i] = different[i-1]
            different[i] = total[i-1]*(k-1)
            total[i] = same[i] + different[i]

        return total[n-1]
```

256. Paint House

Easy 456 46 Favorite Share

There are a row of *n* houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a `n x 3` cost matrix. For example, `costs[0][0]` is the cost of painting house 0 with color red; `costs[1][2]` is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

Note:
All costs are positive integers.

Example:

```
Input: [[17,2,17],[16,16,5],[14,3,19]]
Output: 10
Explanation: Paint house 0 into blue, paint house 1 into green, paint house 2 into blue.
             Minimum cost: 2 + 5 + 3 = 10.
```

```python
    def minCost(self, costs):
        """
        :type costs: List[List[int]]
        :rtype: int
        """

        if len(costs) == 0:
            return 0

        red = [0 for i in range(len(costs))]
        blue = [0 for i in range(len(costs))]
        green = [0 for i in range(len(costs))]

        red[0] = costs[0][0]
        blue[0] = costs[0][1]
        green[0] = costs[0][2]

        #The last house could be painted red, blue or green

        #Case 1: If the last house is red, then the previous house would be either green or blue
        #Case 2: If the last house is blue, then the previous house could be green or red
        #Case 3: If the last house is green, then the previous house could be red or blue

        for i in range(1, len(costs)):
            red[i] = costs[i][0] + min(blue[i-1],green[i-1])
            blue[i] = costs[i][1] + min(red[i-1],green[i-1])
            green[i] = costs[i][2] + min(red[i-1], blue[i-1])

        return min(red[len(costs)-1], blue[len(costs)-1], green[len(costs)-1])

```

1 Rod cutting

Suppose you have a rod of length n, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. A piece of length i is worth p_i dollars.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Figure 15.1 A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.

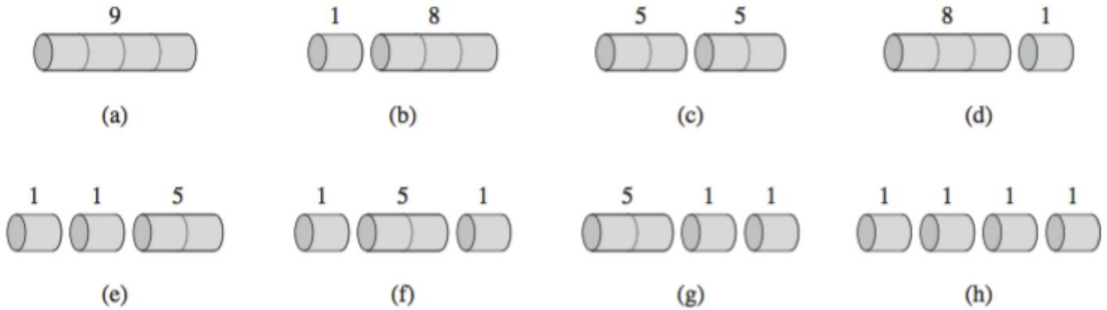


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

For example, if you have a rod of length 4, there are eight different ways to cut it, and the best strategy is cutting it into two pieces of length 2, which gives you 10 dollars.

```
def rodcutting(n,lengths,prices):
  #n = length of the original rod
  #lengths and prices are the price values for rods of lengths 1, 2, 3, 4, ... which could go well beyond the length n

  if n == 0:
    return 0
  if n == 1:
    return prices[0]

  table = [0 for _ in range(n+1)]
  table[0] = 0
  table[1] = prices[0]

  #In general, if a rod has length i, then we focus on the last piece of it.
  #What could the length of the last piece be?
  #It no longer has a constant number of choices. It could vary from 1 to i itself
  #Let's treat the case where we don't cut the rod at all to be the default, and see if we can do better
  #So we will vary the length of the last cut piece from 1 to i-1

  for i in range(2, n+1):
    #Need to compute and set value of table[i]
    best = prices[i-1] #prices array is zero indexed
    for cutlength in range(1,i):
      if prices[cutlength-1] + table[i-cutlength] > best:
        best = prices[cutlength-1] + table[i-cutlength]
    table[i] = best

  return table[n]


print rodcutting(4, [1,2,3,4,5,6,7,8,9,10],[1, 5, 8, 9, 10, 17, 17, 20, 24, 30])
```

343. Integer Break

Medium 674 186 Favorite Share

Given a positive integer *n*, break it into the sum of **at least** two positive integers and maximize the product of those integers. Return the maximum product you can get.

Example 1:

```
Input: 2
Output: 1
Explanation: 2 = 1 + 1, 1 × 1 = 1.
```

Example 2:

```
Input: 10
Output: 36
Explanation: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36.
```

Note: You may assume that *n* is not less than 2 and not larger than 58.

```python
    def integerBreak(self, n):
        """
        :type n: int
        :rtype: int
        """

        #What could the last piece be?
        #The last possible piece could be of length 1,2,3, ... , n
        #Let f(n) = Max possible product obtained from 1..n
        #In general, f(i) = max possible product obtainable from rod of length i
        #The overall rod must be broken at least once, but the smaller pieces don't have to be broken further in general, so let's ignore the requirement that the overall rod must be broken at least once, and handle it as a special case.

        if n == 2: #A rod of length 2 must be broken into 1 and 1
            return 1

        table = [ 0 for _ in range(n+1)] #table[0] has no meaning but just having an array of this size to keep f(i) stored in index i
        table[1] = 1
        table[2] = 2 #Note that subproblem rods don't have to be split up (unlike the special case above with the overall rod size being 2)
        for i in range(3,n+1):
            if i == n: #Special case: The overall rod must be split at least once, so we won't consider it remaining at length n. Pick any valid split (e.g, 1 and n-1) and initialize best-so-far to be 1*n-1)
                best = n-1
            else:
                best = i #If the rod was not cut at all, this would be the default product
            #Set table[i]
            for j in range(1,i): #If it was cut, then the last piece could be of length 1 to i-1
                if table[j] * table[i-j] > best:
                    best = table[j] * table[i-j]
            table[i] = best #The best product (including the no-split case) defines the value of table[i]

        return table[n]
```

Dynamic Programming Practice session 4

Omkar Deshpande

Recap

We looked at more combinatorial optimization problems.

The general pattern for solving them was the same:

1. Think of the optimal solution as something that will be laid out in a sequence.

   E.g, a sequence of moves of length 1 or 2 in the n stairs problem.
   A sequence of ~ m+n horizontal or vertical moves in the unique paths / min cost path in a 2D grid problem.
   A sequence of rob/don't rob decisions for the n house robbers problem.
   A sequence of colors for the posts in the Paint Fence problem.
   A sequence of colors for the houses in the Paint House problem.

2. Focus on the last (rightmost) element of that sequence. Why only the rightmost element? Because you are a lazy manager who just wants to complete the last step of the overall project, depending on your reports to do all the rest of the work.

As a lazy manager, I will only think of
- the last move in a unique paths/ min cost path problem or n stairs problem
- the decision for the last house in the house robbers problem
- the last digit in the knight dialers problem
- the last house or fence post to be colored

3. Think about the different choices/options for what the rightmost element could be, and for each of those options, what the preceding subproblem (neighbor) could be, and what information from those preceding subproblem neighbors would be required for you to solve your problem.

- The last move could be from the previous step or the previous-to-previous step. Or I could reach the bottom right cell from the neighbor on the top or the neighbor to my left.
- The last decision could be to rob/not rob that last house. That depends on whether the previous house was robbed/not robbed.
- The last day could be covered by a 1-day/7-day/30-day pass. That depends on how the previous n-1 days, n-7 days, or n-30 days were covered.
- The last piece of the rod could be 1, 2, …. n units long. That depends on how the previous section of the rod of length n-1 / n-2 / .. could be cut to maximize profit.
- The last digit could be 0…. 9 in the phone dialpad. That depends on whether the previous digit was a 4 or 6, a 6 or 8, and so on.
- The last color could be any of the k colors (if the previous two colors were different) or k-1 colors (if the previous two colors were the same).

4. Remember that each preceding subproblem is going to be solved by one of your immediate reports. You will get to work AFTER they have done their work.

5. Because they are equally lazy like you, they too will be working on only the rightmost bit of their solution, and relying on their subordinates for the solution to the remaining prefix of the problem.

6. In this way, down the corporate hierarchy, the workers will be working on smaller and smaller prefixes of the original problem.

7. The leaf-level workers will first solve the base-case versions of the subproblems.

8. In the recursion class, every worker had exactly one manager they were reporting to. There was a lot of redundant work being done in the corporate hierarchy, as several subtrees were working on solving the same set of subproblems. (Overlapping subproblems) The number of workers employed was exponential.

9. That redundancy is going to be eliminated by DP by “laying off” large subtrees of workers. In the reorganized corporate hierarchy, every worker is working on a unique subproblem. Since the number of unique subproblems will be polynomial, the number of workers will also be polynomial (we have a much smaller company now!).

10. Instead of a huge corporate tree, we now have a much-reduced corporate DAG. Think of every subproblem in the DAG as being solved by an individual worker.

11. Each worker can now have multiple managers they report to. Each manager (as before) continues to have multiple reports (but they may not be the sole manager for any of those reports).

12. Every worker in the hierarchy is using the solutions from his/her reports to build the solution for a slightly larger prefix, and handing it above to all the managers he/she reports to.

13. You are the root manager at the top, and you will combine the solutions from your immediate reports to build the solution for the overall problem.

14. In the top-down approach we had for combinatorial enumeration, you first filled in the leftmost blank, and handed the remaining subproblem (a suffix of the original problem) down to your reports. They successively filled in more and more of the blanks, until the leaf-level nodes would complete individual solutions (and there were an exponential number of them).

15. DP is bottom-up. Each leaf node fills in the leftmost blank, and hands the solution up to the managers above, who in turn fill in the leftmost blank for their slightly larger subproblem, and this process goes purely “bottom-up” until at the top-level, you fill in the last remaining blank (the rightmost blank) and generate the complete solution.

16. Whether top-down or bottom-up, the optimal solution sequence (or the counts of the number of sequences) is being constructed from left to right. So the prefix (partial solution) grows into the full solution at the end.

17. (Coming back to the DP approach)
- Each worker is counting or optimizing a slightly larger prefix than the workers below them (or equivalently, workers before them in topological sort order).
- The solutions to the earlier prefixes are already available when a worker gets to work on his/her task. Those earlier solutions are cached and just need to be “looked up” from some kind of table.
- After computing the solution to his/her problem, an individual worker will then write that solution value into another cell of the same table. There is no recursive call going back to a caller. The next worker in top sort order will automatically get to work on the next subproblem.

18. Coming up with the recurrence formula:

I am given an overall task on a problem of size n. Since I will worry only about the rightmost piece of the optimal sequence (or the rightmost blank for a counting problem), I will think about what information I need from my reports to construct the solution for my problem.
And that is what every worker will do: think “I am given an overall task of size i. Since I will worry only about the rightmost piece of the optimal sequence (or the rightmost blank for a counting problem), I will think about what information I need from my reports to construct the solution for my problem.”

Whatever information I need from my reports, it will have the ***same form*** as the information that a general worker will need from their reports, and what they will need to pass to their own managers (who will expect the same information from them). The difference between the workers is only in the size of the problem (prefix) that each worker is working on.

The structure of that information will determine whether we need additional parameters for the function apart from the problem size (i), and whether we need multiple functions to be computed.

Since that information is going to be stored in the table/cache, the dimensions/size of the table (and even the number of tables) will depend on the form of that information.

E.g, if (for my task) I just need to know how many paths there were up to my two previous neighbors, that means each worker needs to calculate the number of paths leading up to them, and pass on that information to their manager.

If I need to know not just the total number of ways to color the previous n-1 fence posts, but also how many of them had the last two fences of the same color (or different color).... that means every worker needs to calculate the number of ways to color the (prefix) fence posts upto post i, such that the last fences were of the same color (or different color).

If I need to know not just the best way to color n-1 houses, but the best way to color them so that the rightmost house among them is green (or red, or blue), every worker in the hierarchy working on every prefix must compute the same information and pass it on to the immediate manager.

265. Paint House II

Hard 384 14 Favorite Share

There are a row of *n* houses, each house can be painted with one of the *k* colors. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a `n x k` cost matrix. For example, `costs[0][0]` is the cost of painting house 0 with color 0; `costs[1][2]` is the cost of painting house 1 with color 2, and so on... Find the minimum cost to paint all houses.

Note:
All costs are positive integers.

Example:

```
Input: [[1,5,3],[2,9,4]]
Output: 5
Explanation: Paint house 0 into color 0, paint house 1 into color 2. Minimum cost: 1 + 4 = 5;
             Or paint house 0 into color 2, paint house 1 into color 0. Minimum cost: 3 + 2 = 5.
```

Follow up:
Could you solve it in *O*(*nk*) runtime?

```python
    def minCostII(self, costs):
        """
        :type costs: List[List[int]]
        :rtype: int
        """
        n = len(costs)
        if n == 0:
            return 0

        k = len(costs[0])



        #In general, from our reports, we need information about the best way to color the first n-1 houses ending with a specific color.

        table = [ [0 for _ in range(k) ] for _ in range(n) ]

        #Base case
        for color in range(k):
            table[0][color] = costs[0][color]

        #Main traversal
        for i in range(1,n):
            #For each house i, compute the best way to color the houses up to that house ending with each specific color

            for color in range(k):
                #Set table[i][k], which depends on table[i-k][c] where c is different from k
                best = float("inf")
                for prevcolor in range(k):
                    if prevcolor == color:
                        continue
                    best = min(best,table[i-1][prevcolor])
                table[i][color] = best + costs[i][color]

        return min(table[-1])
```

72. Edit Distance

Hard 2520 40 Favorite Share

Given two words *word1* and *word2*, find the minimum number of operations required to convert *word1* to *word2*.

You have the following 3 operations permitted on a word:

1. Insert a character
2. Delete a character
3. Replace a character

Example 1:

```
Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')
```

Example 2:

```
Input: word1 = "intention", word2 = "execution"
Output: 5
Explanation:
intention -> inention (remove 't')
inention -> enention (replace 'i' with 'e')
enention -> exention (replace 'n' with 'x')
exention -> exection (replace 'n' with 'c')
exection -> execution (insert 'u')
```

AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGTCGATTTGCCCGAC

-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC

Definition

Given two strings $x = x_1x_2...x_M$, $y = y_1y_2...y_N$,

an alignment is an assignment of gaps to positions 0,…, M in x, and 0,…, N in y, so as to line up each letter in one sequence with either a letter, or a gap in the other sequence

```python
    def minDistance(self, word1, word2):
        """
        :type word1: str
        :type word2: str
        :rtype: int
        """

        table = [ [0 for _ in range(1+len(word2))] for _ in range(1+len(word1))]

        for col in range(1,1+len(word2)):
            table[0][col] = col

        for row in range(1, 1+len(word1)):
            table[row][0] = row

        for row in range(1, 1+len(word1)):
            for col in range(1, 1+len(word2)):
                if word1[row-1] == word2[col-1]:
                    s = 0
                else:
                    s = 1
                table[row][col] = min(table[row-1][col] + 1, table[row][col-1] + 1, table[row-1][col-1] + s)

        return table[-1][-1]

```

1143. Longest Common Subsequence

Medium 218 8 Favorite Share

Given two strings `text1` and `text2`, return the length of their longest common subsequence.

A *subsequence* of a string is a new string generated from the original string with some characters(can be none) deleted without changing the relative order of the remaining characters. (eg, "ace" is a subsequence of "abcde" while "aec" is not). A *common subsequence* of two strings is a subsequence that is common to both strings.

If there is no common subsequence, return 0.

Example 1:

```
Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.
```

Example 2:

```
Input: text1 = "abc", text2 = "abc"
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.
```

Example 3:

```
Input: text1 = "abc", text2 = "def"
Output: 0
Explanation: There is no such common subsequence, so the result is 0.
```

```python
    def longestCommonSubsequence(self, text1, text2):
        """
        :type text1: str
        :type text2: str
        :rtype: int
        """

        table = [ [0 for _ in range(1+len(text2)) ] for _ in range(1+len(text1))]

        for row in range(1, 1+len(text1)):
            for col in range(1, 1+len(text2)):
                if text1[row-1] == text2[col-1]:
                    s = 1
                else:
                    s = 0
                table[row][col] = max(table[row-1][col], table[row][col-1], table[row-1][col-1] + s)

        return table[-1][-1]
```

583. Delete Operation for Two Strings

Medium 788 21 Favorite Share

Given two words *word1* and *word2*, find the minimum number of steps required to make *word1* and *word2* the same, where in each step you can delete one character in either string.

Example 1:

```
Input: "sea", "eat"
Output: 2
Explanation: You need one step to make "sea" to "ea" and another step to make "eat" to "ea".
```

```
    def minDistance(self, word1, word2):
        """
        :type word1: str
        :type word2: str
        :rtype: int
        """

        table = [ [0 for _ in range(1+len(word2))] for _ in range(1+len(word1))]

        for col in range(1,1+len(word2)):
            table[0][col] = col

        for row in range(1, 1+len(word1)):
            table[row][0] = row

        for row in range(1, 1+len(word1)):
            for col in range(1, 1+len(word2)):
                if word1[row-1] == word2[col-1]:
                    s = 0
                else:
                    s = 3
                table[row][col] = min(table[row-1][col] + 1, table[row][col-1] + 1, table[row-1][col-1] + s)

        return table[-1][-1]
```

1092. Shortest Common Supersequence

Hard 183 6 Favorite Share

Given two strings `str1` and `str2`, return the shortest string that has both `str1` and `str2` as subsequences. If multiple answers exist, you may return any of them.

(A string S is a subsequence of string T if deleting some number of characters from T (possibly 0, and the characters are chosen anywhere from T) results in the string S.)

Example 1:

```
Input: str1 = "abac", str2 = "cab"
Output: "cabac"
Explanation:
str1 = "abac" is a subsequence of "cabac" because we can delete the first "c".
str2 = "cab" is a subsequence of "cabac" because we can delete the last "ac".
The answer provided is the shortest such string that satisfies these properties.
```

```
    def shortestCommonSupersequence(self, str1, str2):
        """
        :type str1: str
        :type str2: str
        :rtype: str
        """

        table = [ [0 for _ in range(1+len(str2)) ] for _ in range(1+len(str1))]

        for row in range(1, 1+len(str1)):
            for col in range(1, 1+len(str2)):
                if str1[row-1] == str2[col-1]:
                    s = 1
                else:
                    s = 0
                table[row][col] = max(table[row-1][col], table[row][col-1], table[row-1][col-1] + s)

```

```
        #Now do a traceback
        result = []
        row = len(str1)
        col = len(str2)
        while row != 0 and col != 0:
            #Check where the value in table[row][col] came from
            if table[row][col] == table[row-1][col]:
                result.append(str1[row-1])
                row -= 1
            elif table[row][col] == table[row][col-1]:
                result.append(str2[col-1])
                col -= 1
            else:
                result.append(str1[row-1])
                row -= 1
                col -= 1

        while row != 0:
            result.append(str1[row-1])
            row -= 1

        while col != 0:
            result.append(str2[col-1])
            col -= 1

        result.reverse()
        return "".join(result)
```

Dynamic Programming Practice session 5

Omkar Deshpande

Recap

We looked at more combinatorial optimization problems.

The general pattern for solving them was the same:

1. Think of the optimal solution as something that will be laid out in a sequence.

   E.g, a sequence of moves of length 1 or 2 in the n stairs problem.
   A sequence of ~ m+n horizontal or vertical moves in the unique paths / min cost path in a 2D grid problem.
   A sequence of rob/don't rob decisions for the n house robbers problem.
   A sequence of colors for the posts in the Paint Fence problem.
   A sequence of colors for the houses in the Paint House problem.

2. Focus on the last (rightmost) element of that sequence. Why only the rightmost element? Because you are a lazy manager who just wants to complete the last step of the overall project, depending on your reports to do all the rest of the work.

As a lazy manager, I will only think of
- the last move in a unique paths/ min cost path problem or n stairs problem
- the decision for the last house in the house robbers problem
- the last digit in the knight dialers problem
- the last house or fence post to be colored

3. Think about the different choices/options for what the rightmost element could be, and for each of those options, what the preceding subproblem (neighbor) could be, and what information from those preceding subproblem neighbors would be required for you to solve your problem.

- The last move could be from the previous step or the previous-to-previous step. Or I could reach the bottom right cell from the neighbor on the top or the neighbor to my left.
- The last decision could be to rob/not rob that last house. That depends on whether the previous house was robbed/not robbed.
- The last day could be covered by a 1-day/7-day/30-day pass. That depends on how the previous n-1 days, n-7 days, or n-30 days were covered.
- The last piece of the rod could be 1, 2, …. n units long. That depends on how the previous section of the rod of length n-1 / n-2 / .. could be cut to maximize profit.
- The last digit could be 0…. 9 in the phone dialpad. That depends on whether the previous digit was a 4 or 6, a 6 or 8, and so on.
- The last color could be any of the k colors (if the previous two colors were different) or k-1 colors (if the previous two colors were the same).

4. Remember that each preceding subproblem is going to be solved by one of your immediate reports. You will get to work AFTER they have done their work.

5. Because they are equally lazy like you, they too will be working on only the rightmost bit of their solution, and relying on their subordinates for the solution to the remaining prefix of the problem.

6. In this way, down the corporate hierarchy, the workers will be working on smaller and smaller prefixes of the original problem.

7. The leaf-level workers will first solve the base-case versions of the subproblems.

8. In the recursion class, every worker had exactly one manager they were reporting to. There was a lot of redundant work being done in the corporate hierarchy, as several subtrees were working on solving the same set of subproblems. (Overlapping subproblems) The number of workers employed was exponential.

9. That redundancy is going to be eliminated by DP by “laying off” large subtrees of workers. In the reorganized corporate hierarchy, every worker is working on a unique subproblem. Since the number of unique subproblems will be polynomial, the number of workers will also be polynomial (we have a much smaller company now!).

10. Instead of a huge corporate tree, we now have a much-reduced corporate DAG. Think of every subproblem in the DAG as being solved by an individual worker.

11. Each worker can now have multiple managers they report to. Each manager (as before) continues to have multiple reports (but they may not be the sole manager for any of those reports).

12. Every worker in the hierarchy is using the solutions from his/her reports to build the solution for a slightly larger prefix, and handing it above to all the managers he/she reports to.

13. You are the root manager at the top, and you will combine the solutions from your immediate reports to build the solution for the overall problem.

14. In the top-down approach we had for combinatorial enumeration, you first filled in the leftmost blank, and handed the remaining subproblem (a suffix of the original problem) down to your reports. They successively filled in more and more of the blanks, until the leaf-level nodes would complete individual solutions (and there were an exponential number of them).

15. DP is bottom-up. Each leaf node fills in the leftmost blank, and hands the solution up to the managers above, who in turn fill in the leftmost blank for their slightly larger subproblem, and this process goes purely “bottom-up” until at the top-level, you fill in the last remaining blank (the rightmost blank) and generate the complete solution.

16. Whether top-down or bottom-up, the optimal solution sequence (or the counts of the number of sequences) is being constructed from left to right. So the prefix (partial solution) grows into the full solution at the end.

17. (Coming back to the DP approach)
- Each worker is counting or optimizing a slightly larger prefix than the workers below them (or equivalently, workers before them in topological sort order).
- The solutions to the earlier prefixes are already available when a worker gets to work on his/her task. Those earlier solutions are cached and just need to be “looked up” from some kind of table.
- After computing the solution to his/her problem, an individual worker will then write that solution value into another cell of the same table. There is no recursive call going back to a caller. The next worker in top sort order will automatically get to work on the next subproblem.

18. Coming up with the recurrence formula:

I am given an overall task on a problem of size n. Since I will worry only about the rightmost piece of the optimal sequence (or the rightmost blank for a counting problem), I will think about what information I need from my reports to construct the solution for my problem.
And that is what every worker will do: think “I am given an overall task of size i. Since I will worry only about the rightmost piece of the optimal sequence (or the rightmost blank for a counting problem), I will think about what information I need from my reports to construct the solution for my problem.”

Whatever information I need from my reports, it will have the ***same form*** as the information that a general worker will need from their reports, and what they will need to pass to their own managers (who will expect the same information from them). The difference between the workers is only in the size of the problem (prefix) that each worker is working on.

The structure of that information will determine whether we need additional parameters for the function apart from the problem size (i), and whether we need multiple functions to be computed.

Since that information is going to be stored in the table/cache, the dimensions/size of the table (and even the number of tables) will depend on the form of that information.

Subproblem patterns

In all problems we have seen so far, we have been trying to count or optimize permutations/combinations that can be visualised as sequences.

If the overall problem is the sequence $x_1, x_2, x_3, \ldots, x_n$,

the subproblems are prefixes of the form $x_1, x_2, \ldots, x_i$

91. Decode Ways

Medium 1727 1969 Favorite Share

A message containing letters from `A-Z` is being encoded to numbers using the following mapping:

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

Given a **non-empty** string containing only digits, determine the total number of ways to decode it.

Example 1:

```
Input: "12"
Output: 2
Explanation: It could be decoded as "AB" (1 2) or "L" (12).
```

Example 2:

```
Input: "226"
Output: 3
Explanation: It could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).
```

Accepted 800,540 | Submissions 1,907,951

```
    def numDecodings(self, s):
        """
        :type s: str
        :rtype: int
        """

        #Let f(n) = Total number of ways to decode a string of length n
        #Look at the last digit. Where could it have come from?
        #Either it came by itself from some letter. But for that to happen, the digit shouldn't be 0.
        #Or it (along with the previous digit) came from a letter. For that to happen, the previous digit should be 1 OR (previous digit should be 2 and this digit should be from 0 to 6)
        #f(i) = (conditionally) f(i-1) + (conditionally) f(n-2)

        table = [0]*(1+len(s))
        table[0] = 1
        if s[0] == "0":
            table[1] = 0
        else:
            table[1] = 1
        for i in range(2,1+len(s)):
            #table[i] = Number of ways of decoding the first i characters of s
            table[i] = 0 #Initialization of table[i]
            if s[i-1] != "0": #If the ith digit is not 0, it could have come by itself from a letter
                table[i] += table[i-1] #So find out the number of ways to decode the remaining prefix
            if s[i-2] == "1" or (s[i-2] == "2" and s[i-1] in ["6", "5", "4", "3", "2", "1", "0"]):
                #The (i-1)st digit is a 1 or it (together with the ith digit) make up a number from 20 to 26
                table[i] += table[i-2]

        return table[len(s)]

```

Subproblem patterns

Even when the input was in the form of two strings

$x_1, x_2, x_3, \ldots, x_m$ and $y_1, y_2, y_3, \ldots, y_n$

we visualised an alignment of the sequences and tried to optimize the alignment, treating it as a single sequence.

The subproblems were prefixes of the form $x_1, x_2, \ldots, x_i$ and $y_1, y_2, \ldots, y_j$

97. Interleaving String

Hard 934 54 Favorite Share

Given *s1*, *s2*, *s3*, find whether *s3* is formed by the interleaving of *s1* and *s2*.

Example 1:

```
Input: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
Output: true
```

Example 2:

```
Input: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbbaccc"
Output: false
```

```
    def isInterleave(self, s1, s2, s3):
        """
        :type s1: str
        :type s2: str
        :type s3: str
        :rtype: bool
        """

        #Imagine taking the interleaving of s1 and s2, and pulling apart the characters from the two strings
        #Add gaps in the other string to visualize it as an alignment.
        #There are no matches/mismatches in the alignment.
        #There is a combinatorial explosion in the number of such alignments.
        #Given s3, can it be visualized as such an alignment too? That is the question given to me.
        #As a lazy manager, I will look at only the last alignment (a, _) or (_, a) pair to make the decision.
        #The last character 'a' could in theory have come from either s1 or s2.
        #If it matches the last character of s1, I will hire someone to find out if s3 minus the last character could be the interleaving of s1 minus the last character, and s2.
        #If it matches the last character of s2, I will hire someone to find out if s3 minus the last character could be the interleaving of s2 minus the last character, and s1.
        #If either of them come back with the answer "True", my answer will also be "True". Otherwise it will be False.
        #f(i,j) = f(i-1,j) & s1[i-1] == s3[i+j-1]  OR f(i,j-1) & s2[j-1] == s3[i+j-1]

        if len(s1) + len(s2) != len(s3):
            return False

        table = [ [False]*(1+len(s2)) for _ in range(1+len(s1)) ]
```

```
        #Base cases
        table[0][0] = True
        for col in range(1,1+len(s2)):
            table[0][col] = table[0][col-1] and (s2[col-1] == s3[col-1])
            #The prefix of s2 should exactly match the prefix of s3

        for row in range(1,1+len(s1)):
            table[row][0] = table[row-1][0] and (s1[row-1] == s3[row-1])
            #The prefix of s1 should exactly match the prefix of s3

        #Main traversal for the recursive case
        for row in range(1, 1+len(s1)):
            for col in range(1, 1 + len(s2)):
                table[row][col] = (table[row-1][col] and s1[row-1] == s3[row+col-1]) or (table[row][col-1] and s2[col-1] == s3[row+col-1])

        return table[-1][-1]
```

416. Partition Equal Subset Sum

Medium 1517 40 Favorite Share

Given a **non-empty** array containing **only positive integers**, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Note:

1. Each of the array element will not exceed 100.
2. The array size will not exceed 200.

Example 1:

```
Input: [1, 5, 11, 5]

Output: true

Explanation: The array can be partitioned as [1, 5, 5] and [11].
```

Example 2:

```
Input: [1, 2, 3, 5]

Output: false

Explanation: The array cannot be partitioned into equal sum subsets.
```

```python
    def canPartition(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """

        #If the sum in both subsets is to be equal, then the total sum of the numbers must be even
        total = sum(nums)
        if total % 2 == 1:
            return False

        #We are searching for a subset that adds up to half of the total sum
        k = total / 2
```

```
        #Here, each element can be included or excluded from the subset
        #As a lazy manager, I will make the decision only for the last number in the subset
        #If I include the last number, then I want to know from my reports whether they could find a subset from the previous n-1 numbers adding up to k - the last number.
        #If I exclude the last number, then I want to know from my reports whether they could find a subset from the previous n-1 numbers adding up to k.
        #If either of these reports came back with a yes answer, then my own answer would be a yes.
        #f(n,k) = True if there exists a subset among the first n numbers adding up to k, False otherwise
        #f(n,k) = f(n-1,k - value of nth number) or f(n-1, k)
        #Base cases: f(0,0) = True
        #f(i,0) = True because we can exclude all i elements to get a subset sum of 0
        #f(0,k) = False because we cannot get a non-zero subset sum by including 0 elements

        #Build a table with (n+1)(k+1) elements
        table = [ [True]*(1+k) for _ in range(1+len(nums))]
        for target in range(1,1+k):
            table[0][target] = False
```

```
        #Do a full traversal of the dependency DAG table now
        for numindex in range(1, 1+len(nums)):
            for target in range(1,1+k):
                #table[numindex][target] = True if the first numindex numbers can form a subset adding up to target
                #table[numindex][target] = table[numindex-1][target] or table[numindex-1][target - nums[numindex-1]]
                table[numindex][target] = table[numindex-1][target]
                if target >= nums[numindex-1]:
                    table[numindex][target] = table[numindex][target] or table[numindex-1][target - nums[numindex-1]]

        return table[-1][-1]
```

```
27          #Base cases
28          #Build a table with (n+1)(k+1) elements
29          memo = {}
30          #table = [ [True]*(1+k) for _ in range(1+len(nums))]
31          memo[(0,0)] = True
32
33          for numindex in range(1,1+len(nums)):
34              memo[(numindex,0)] = True
35
36          for target in range(1,1+k):
37              #table[0][target] = False
38              memo[(0,target)] = False
39
```

Memoization

```
        #Do a full traversal of the dependency DAG table now
        #for numindex in range(1, 1+len(nums)):
        #    for target in range(1,1+k):
        def helper(numindex,target):
                if (numindex,target) in memo:
                    return memo[(numindex,target)]
                #table[numindex][target] = True if the first numindex numbers can form a subset adding up to target
                #table[numindex][target] = table[numindex-1][target] or table[numindex-1][target - nums[numindex-1]]
                #table[numindex][target] = table[numindex-1][target]
                #if target >= nums[numindex-1]:
                #    table[numindex][target] = table[numindex][target] or table[numindex-1][target - nums[numindex-1]]
                if target >= nums[numindex-1]:
                    memo[(numindex,target)] = helper(numindex-1,target-nums[numindex-1]) or helper(numindex-1,target)
                else:
                    memo[(numindex,target)] = helper(numindex-1,target)

                return memo[(numindex,target)]

        return helper(len(nums),k)
```

Order of these two could affect running time!

Minimization problem: Compute the shortest path from s to d in the dependency DAG.

Maximization problem: Compute the longest path from s to d in the dependency DAG

Counting problem: Compute the number of paths from s to d in the dependency DAG.

Decision problem: Figure out if there exists a path from s to d in the dependency DAG.

Knapsack problem

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His bag (“knapsack”) will hold a total weight of at most W pounds. There are n items to pick from, of weight w1, w2, … wn and dollar value v1, v2, ... , vn. What is the most valuable combination of items he can fit into his bag?

E.g, W = 10
Item1: weight = 6, value = 30$
Item2: weight = 3, value = 14$
Item3: weight = 4, value = 16$
Item4: weight = 2, value = 9$

Three versions of Knapsack

1. Only one copy of each item available. (0-1 Knapsack)
2. Unlimited quantities of each item available. (Unbounded Knapsack)
3. One copy of each item, but the robber can steal a fraction of an item also. (Fractional Knapsack or Continuous Knapsack)

53. Maximum Subarray

Easy 5185 202 Favorite Share

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Example:

```
Input: [-2,1,-3,4,-1,2,1,-5,4],
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

```python
    def maxSubArray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """


        #f(i) = maximum subarray sum ending at index i
        #Either extend the subarray from index i-1 or begin a new one
        #f(i) = max(f(i-1) + ith number, ith number)

        table = [0 for _ in range(len(nums))]

        table[0] = nums[0]
        for i in range(1,len(nums)):
            table[i] = max(table[i-1] + nums[i] , nums[i])
        return max(table)
```

Dynamic Programming Live Class

Omkar Deshpande

Recap of foundation videos

Dynamic Programming is doing recursion without repetition.

Two variants:

1. Memoization (top-down)
2. Tabulation (bottom-up)

We generally (though not always) prefer bottom-up tabulation over top-down memoization.

Classic DP problems

- Computing the nth Fibonacci number (classic 1D DP)
- Computing C(n,k) (classic 2D DP)

(Non-optimization) Counting problems

1. Number of ways to climb n stairs (1D)
2. Number of unique paths from the top-left to bottom-right corner of a grid. (2D)

Overlapping subproblems

Combinatorial Optimization problems

The adjective “combinatorial” means that without DP, the combinatorial explosion of possibilities would lead to an exponential time complexity.

After all, we are optimizing permutations and combinations (which are “combinatorial objects”, always exponential in number) by trying to pick the “best”.

1. Min cost way to climb n stairs (1D)
2. Maximum path sum in a 2D grid (2D)
3. Minimum number of coins making up change (1D)

Overlapping subproblems + Optimal substructure

Recap

We looked at counting and optimization problems.

The general pattern for solving them was the same:

1. Think of the optimal solution as something that will be laid out in a sequence.

   E.g, a sequence of moves of length 1 or 2 in the n stairs problem.
   A sequence of ~ m+n horizontal or vertical moves in the unique paths / min cost path in a 2D grid problem.

Decrease-and-conquer

2. Focus on the last (rightmost) element of that sequence. Why only the rightmost element? Because you are a lazy manager who just wants to complete the last step of the overall project, depending on your reports to do all the rest of the work.

As a lazy manager, I will only think of
- the last move in a unique paths/ min cost path problem or n stairs problem
- the last coin in the coin change problem

3. Think about the different choices/options for what the rightmost element could be, and for each of those options, what the preceding subproblem (neighbor) could be, and what information from those preceding subproblem neighbors would be required for you to solve your problem.

- The last move could be from the previous step or the previous-to-previous step. Or I could reach the bottom right cell from the neighbor on the top or the neighbor to my left.

4. Remember that each preceding subproblem is going to be solved by one of your immediate reports. You will get to work AFTER they have done their work.

5. Because they are equally lazy like you, they too will be working on only the rightmost bit of their solution, and relying on their subordinates for the solution to the remaining prefix of the problem.

6. In this way, down the corporate hierarchy, the workers will be working on smaller and smaller prefixes of the original problem.

7. The leaf-level workers will first solve the base-case versions of the subproblems.

8. In the recursion class, every worker had exactly one manager they were reporting to. There was a lot of redundant work being done in the corporate hierarchy, as several subtrees were working on solving the same set of subproblems. (Overlapping subproblems) The number of workers employed was exponential.

9. That redundancy is going to be eliminated by DP by “laying off” large subtrees of workers. In the reorganized corporate hierarchy, every worker is working on a unique subproblem. Since the number of unique subproblems will be polynomial, the number of workers will also be polynomial (we have a much smaller company now!).

10. Instead of a huge corporate tree, we now have a much-reduced corporate DAG. Think of every subproblem in the DAG as being solved by an individual worker.

11. Each worker can now have multiple managers they report to. Each manager (as before) continues to have multiple reports (but they may not be the sole manager for any of those reports).

12. Every worker in the hierarchy is using the solutions from his/her reports to build the solution for a slightly larger prefix, and handing it above to all the managers he/she reports to.

13. You are the root manager at the top, and you will combine the solutions from your immediate reports to build the solution for the overall problem.

14. In the top-down approach we had for combinatorial enumeration, you first filled in the leftmost blank, and handed the remaining subproblem (a suffix of the original problem) down to your reports. They successively filled in more and more of the blanks, until the leaf-level nodes would complete individual solutions (and there were an exponential number of them).

15. DP is bottom-up. Each leaf node fills in the leftmost blank, and hands the solution up to the managers above, who in turn fill in the leftmost blank for their slightly larger subproblem, and this process goes purely “bottom-up” until at the top-level, you fill in the last remaining blank (the rightmost blank) and generate the complete solution.

16. Whether top-down or bottom-up, the optimal solution sequence (or the counts of the number of sequences) is being constructed from left to right. So the prefix (partial solution) grows into the full solution at the end.

17. (Coming back to the DP approach)
- Each worker is counting or optimizing a slightly larger prefix than the workers below them (or equivalently, workers before them in topological sort order).
- The solutions to the earlier prefixes are already available when a worker gets to work on his/her task. Those earlier solutions are cached and just need to be “looked up” from some kind of table.
- After computing the solution to his/her problem, an individual worker will then write that solution value into another cell of the same table. There is no recursive call going back to a caller. The next worker in top sort order will automatically get to work on the next subproblem.

18. Coming up with the recurrence formula:

I am given an overall task on a problem of size n. Since I will worry only about the rightmost piece of the optimal sequence (or the rightmost blank for a counting problem), I will think about what information I need from my reports to construct the solution for my problem.
And that is what every worker will do: think “I am given an overall task of size i. Since I will worry only about the rightmost piece of the optimal sequence (or the rightmost blank for a counting problem), I will think about what information I need from my reports to construct the solution for my problem.”

Whatever information I need from my reports, it will have the ***same form*** as the information that a general worker will need from their reports, and what they will need to pass to their own managers (who will expect the same information from them). The difference between the workers is only in the size of the problem (prefix) that each worker is working on.

The structure of that information will determine whether we need additional parameters for the function apart from the problem size (i), and whether we need multiple functions to be computed.

Since that information is going to be stored in the table/cache, the dimensions/size of the table (and even the number of tables) will depend on the form of that information.

139. Word Break

Medium 2758 147 Favorite Share

Given a **non-empty** string *s* and a dictionary *wordDict* containing a list of **non-empty** words, determine if *s* can be segmented into a space-separated sequence of one or more dictionary words.

Note:

- The same word in the dictionary may be reused multiple times in the segmentation.
- You may assume the dictionary does not contain duplicate words.

Example 1:

```
Input: s = "leetcode", wordDict = ["leet", "code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".
```

Example 2:

```
Input: s = "applepenapple", wordDict = ["apple", "pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
             Note that you are allowed to reuse a dictionary word.
```

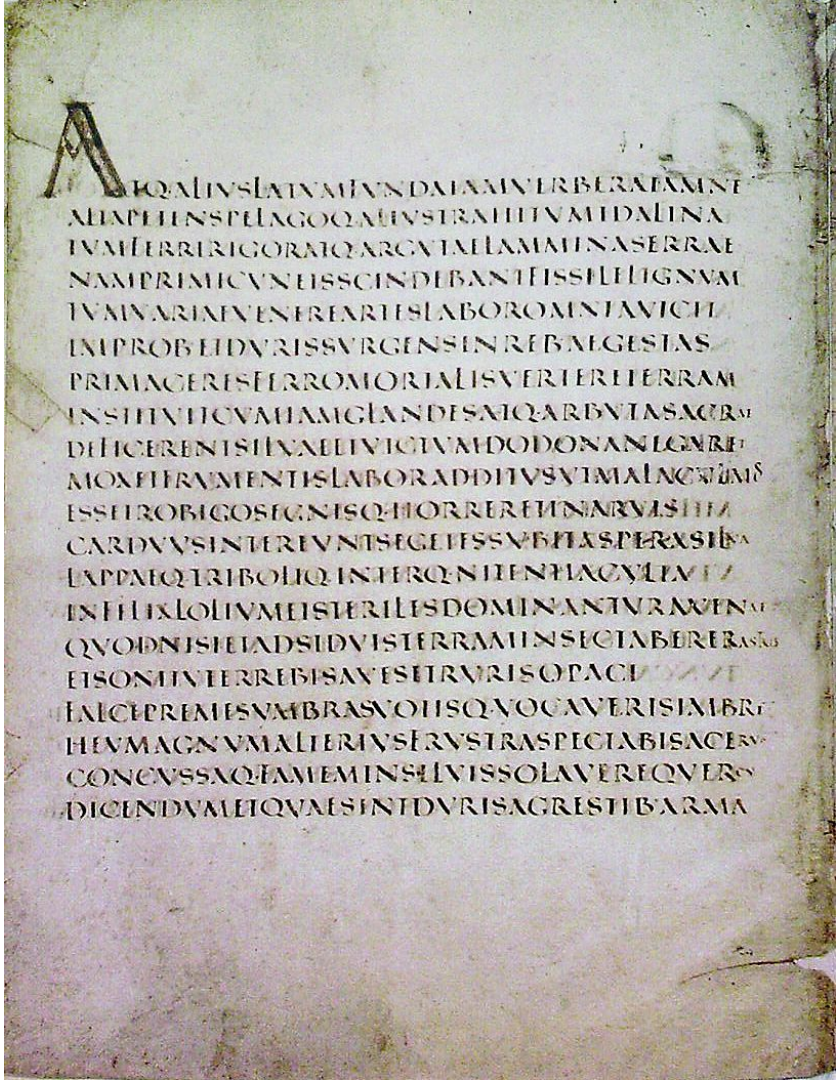
Example 3:

```
Input: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
Output: false
```

BOTHEARTHANDSATURNSPIN

Can this be segmented into English words at all?

Decision Problem: Yes/No answer



```
    def wordBreak(self, s, wordDict):
        """
        :type s: str
        :type wordDict: List[str]
        :rtype: bool
        """
        #Decision problem

        #The dictionary should be a hash map instead of a list for fast lookup
        hmap = {}
        for word in wordDict:
            hmap[word] = 1

        #f(n) = True if s[0...n-1] can be split into one or more dictionary words
        #f(i) = OR over all j (The last j characters are in the dict and f(i-j) is True)

        table = [False]*(1+len(s))
        #Base case needed?
        #table[0] = True

        for i in range(1,len(table)):
            #Compute and fill in table[i]
            table[i] = (s[:i] in hmap) #Entire string is a valid word
            for j in range(1,i): #If it has to be broken from the right end into a piece of length j
                #First i-j characters should return True
                #The characters after them, starting from i-j+1 'th character up to the ith character should make up a valid dictionary word
                if s[i-j:i] in hmap and table[i-j] == True:
                    table[i] = True

        return table[-1]
```

Counting version

Count all different possible word-break arrangements for the string s. This integer could be large so output it modulo 10^9 + 7.

```
    def wordBreak(self, s, wordDict):
        """
        :type s: str
        :type wordDict: List[str]
        :rtype: bool
        """
        #Counting problem

        #The dictionary should be a hash map instead of a list for fast lookup
        hmap = {}
        for word in wordDict:
            hmap[word] = 1

        #f(n) = The number of valid splits of s[0...n-1] into dictionary words
        #f(i) = Summation over all j: f(i-j), where the last j characters are in the dict

        table = [0]*(1+len(s))
        #Base case needed?
        #table[0] = True

        for i in range(1,len(table)):
            #Compute and fill in table[i]
            if s[:i] in hmap:
                table[i] += 1 #Entire string is a valid word
            for j in range(1,i): #If it has to be broken from the right end into a piece of length j
                #First i-j characters should return their count
                #The characters after them, starting from i-j+1 'th character up to the ith character should make up a vali
dictionary word
                if s[i-j:i] in hmap:
                    table[i] += table[i-j]

        return table[-1] % 1000000007
```

Optimization version

Find the best possible word break arrangement for the string s. Assume that each word is associated with a score in the dictionary, indicating how common it is.

72. Edit Distance

Hard 2520 40 Favorite Share

Given two words *word1* and *word2*, find the minimum number of operations required to convert *word1* to *word2*.

You have the following 3 operations permitted on a word:

1. Insert a character
2. Delete a character
3. Replace a character

Example 1:

```
Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')
```

Example 2:

```
Input: word1 = "intention", word2 = "execution"
Output: 5
Explanation:
intention -> inention (remove 't')
inention -> enention (replace 'i' with 'e')
enention -> exention (replace 'n' with 'x')
exention -> exection (replace 'n' with 'c')
exection -> execution (insert 'u')
```

AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGTCGATTTGCCCGAC

-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---
TAG-CTATCAC--GACCGC--GGTCGATTTGCCCGAC

Definition

Given two strings $x = x_1x_2...x_M$, $y = y_1y_2...y_N$,

an alignment is an assignment of gaps to positions 0,…, M in x, and 0,…, N in y, so as to line up each letter in one sequence with either a letter, or a gap in the other sequence

```python
    def minDistance(self, word1, word2):
        """
        :type word1: str
        :type word2: str
        :rtype: int
        """

        table = [ [0 for _ in range(1+len(word2))] for _ in range(1+len(word1))]

        for col in range(1,1+len(word2)):
            table[0][col] = col

        for row in range(1, 1+len(word1)):
            table[row][0] = row

        for row in range(1, 1+len(word1)):
            for col in range(1, 1+len(word2)):
                if word1[row-1] == word2[col-1]:
                    s = 0
                else:
                    s = 1
                table[row][col] = min(table[row-1][col] + 1, table[row][col-1] + 1, table[row-1][col-1] + s)

        return table[-1][-1]

```

877. Stone Game

Medium 411 700 Favorite Share

Alex and Lee play a game with piles of stones. There are an even number of piles **arranged in a row**, and each pile has a positive integer number of stones `piles[i]`.

The objective of the game is to end with the most stones. The total number of stones is odd, so there are no ties.

Alex and Lee take turns, with Alex starting first. Each turn, a player takes the entire pile of stones from either the beginning or the end of the row. This continues until there are no more piles left, at which point the person with the most stones wins.

Assuming Alex and Lee play optimally, return `True` if and only if Alex wins the game.

Example 1:

```
Input: [5,3,4,5]
Output: true
Explanation:
Alex starts first, and can only take the first 5 or the last 5.
Say he takes the first 5, so that the row becomes [3, 4, 5].
If Lee takes 3, then the board is [4, 5], and Alex takes 5 to win with 10 points.
If Lee takes the last 5, then the board is [3, 4], and Alex takes 4 to win with 9 points.
This demonstrated that taking the first 5 was a winning move for Alex, so we return true.
```

```
    def stoneGame(self, piles):
        """
        :type piles: List[int]
        :rtype: bool
        """

        #The given problem is of size n.
        #As a lazy manager, I would try picking the item on extreme left and find what happens. Or I would pick the item on the extreme right and see what happens. I would then choose the better of the two moves. If I find my opponent is losing in either of those resulting subproblems, I will go for that choice and make sure I win.
        #A subproblem would no longer be just a prefix of the original problem but more like a substring.

        table = [ [False]*len(piles) for _ in range(len(piles))]

        for diag in range(len(piles)):
            table[diag][diag] = True

        for row in range(len(piles)-2,-1,-1):
            for col in range(diag,len(piles)):
                if table[row][col-1] == False:
                    table[row][col] = True
                if table[row+1][col] == False:
                    table[row][col] = True

        return table[0][len(piles)-1]
```

Matrix Chain Multiplication

Problem Statement:

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

(ABC)D = (AB)(CD) = A(BCD) = ...

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Clearly, the first parenthesization requires less number of operations.

Given an array ***mtxSizes***[], which represents the chain of matrices such that the ith matrix Ai is of dimension ***mtxSizes***[i-1] x ***mtxSizes***[i], we need to write a function that should return the minimum number of multiplications needed to multiply the chain. Length of chain of matrices is ***n***, and thus size of ***mtxSizes*** is (***n***+1).

Input/Output Format For The Function:

Input Format:

You will be given an integer array ***mtxSizes***.

Output Format:

Return an integer ***minOps***, denoting the minimum number of operations needed.

```python
def  minMultiplicationCost(mtxSizes):
    n = len(mtxSizes) - 1
    if n == 1:
        return 0

    #Suppose the final multiplication is the optimal way to multiply the optimal result of matrices 0..i with the optimal
        result of matrices i+1...n-1

    #f(i,j) = Optimal way to multiple matrices i, i+1, ... j
    #f(i,j) = min over all k [f(i,k) + f(k+1,j)] where k varies from i to j-1

    table = [ [0]*n for _ in range(n)]
    for m in range(n):
        table[m][m] = 0

    for row in range(n-2,-1,-1):
        for col in range(row+1,n):
            table[row][col] = float("inf")
            for k in range(row,col):
                table[row][col] = min(table[row][col], table[row][k] + table[k+1][col] + mtxSizes[row]*mtxSizes[k+1]
                    *mtxSizes[col+1])

    return table[0][n-1]
```