**Day-8**

**Dynamic Programming**

**Student Name: Kamalpreet Singh UID: 22BCS11720**

**Branch: BE - CSE Section/Group: FL\_IOT-603-A**

**Semester: 5th Date of Performance:27-12-24**

# [N-th Tribonacci Number](https://leetcode.com/problems/n-th-tribonacci-number/)

The Tribonacci sequence Tn is defined as follows:

T0 = 0, T1 = 1, T2 = 1, and Tn+3 = Tn + Tn+1 + Tn+2 for n >= 0.

Given n, return the value of Tn.

**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**

**Constraints: 0 <= n <= 37**

**The answer is guaranteed to fit within a 32-bit integer, ie. answer <= 2^31 - 1.**

## CODE:

def tribonacci(n: int) -> int: if n == 0:

return 0

if n in (1, 2):

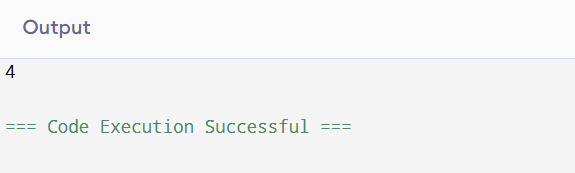
return 1

dp = [0, 1, 1]

for i in range(3, n + 1):

dp.append(dp[i - 1] + dp[i - 2] + dp[i - 3]) return dp[n]

print(tribonacci(4))



***Easy***

# [Climbing Stairs](https://leetcode.com/problems/climbing-stairs/)

You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

**Example 1: Input: n = 2**

**Output: 2**

**Explanation: There are two ways to climb to the top.**

* 1. **1 step + 1 step**
  2. **2 steps Constraints:1 <= n <= 45**

## CODE:

def climbStairs(n: int) -> int: if n == 1:

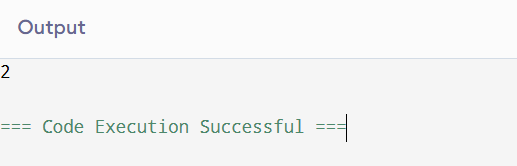
return 1

dp = [0] \* (n + 1) dp[1], dp[2] = 1, 2

for i in range(3, n + 1):

dp[i] = dp[i - 1] + dp[i - 2] return dp[n]

print(climbStairs(2)) # Output: 2



***Medium*:**

# [Longest Palindromic Substring](https://leetcode.com/problems/longest-palindromic-substring/)

Given a string s, return the longest palindromic substring in s.

**Example 1: Input: s = "babad"**

**Output: "bab"**

**Explanation: "aba" is also a valid answer.**

**Example 2: Input: s = "cbbd"**

**Output: "bb"**

**Constraints: 1 <= s.length <= 1000**

**s consist of only digits and English letters.**

## CODE:

def longestPalindrome(s: str) -> str:

def expand\_around\_center(left, right):

while left >= 0 and right < len(s) and s[left] == s[right]: left -= 1

right += 1

return left + 1, right - 1

start, end = 0, 0

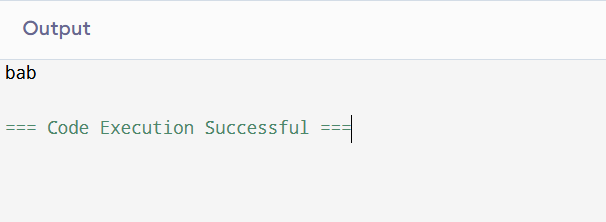
for i in range(len(s)):

l1, r1 = expand\_around\_center(i, i)

l2, r2 = expand\_around\_center(i, i + 1) if r1 - l1 > end - start:

start, end = l1, r1 if r2 - l2 > end - start:

start, end = l2, r2 return s[start:end + 1]

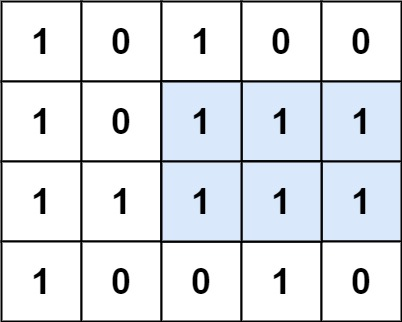
print(longestPalindrome("babad")) # Output: "bab" or "aba"

***Hard***

# [Maximal Rectangle](https://leetcode.com/problems/maximal-rectangle/)

Given a rows x cols binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

**Example-**



**Input:matrix= [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0 "]]**

**Output: 6**

**Explanation: The maximal rectangle is shown in the above picture.**

**Constraints:**

**rows == matrix.length cols == matrix[i].length 1 <= row, cols <= 200**

**matrix[i][j] is '0' or '1'.**

**CODE:**

def maximalRectangle(matrix): if not matrix:

return 0

def largest\_histogram\_area(heights): stack = []

max\_area = 0 heights.append(0)

for i, h in enumerate(heights):

while stack and heights[stack[-1]] > h: height = heights[stack.pop()]

width = i if not stack else i - stack[-1] - 1 max\_area = max(max\_area, height \* width)

stack.append(i) return max\_area

cols = len(matrix[0]) heights = [0] \* cols max\_area = 0

for row in matrix:

for j in range(cols):

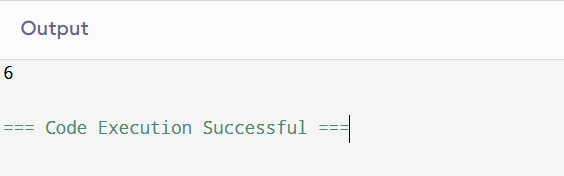
heights[j] = heights[j] + 1 if row[j] == "1" else 0

max\_area = max(max\_area, largest\_histogram\_area(heights)) return max\_area

matrix = [["1", "0", "1", "0", "0"], ["1", "0", "1", "1", "1"], ["1", "1", "1", "1", "1"],

["1", "0", "0", "1", "0"]]

print(maximalRectangle(matrix))



***Very Hard***

# [Cherry Pickup](https://leetcode.com/problems/cherry-pickup/)

You are given an n x n grid representing a field of cherries, each cell is one of three possible integers.

0 means the cell is empty, so you can pass through,

1 means the cell contains a cherry that you can pick up and pass through, or

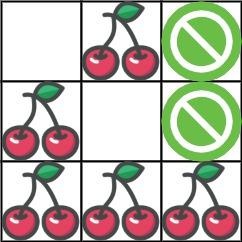
-1 means the cell contains a thorn that blocks your way.

Return the maximum number of cherries you can collect by following the rules below:

Starting at the position (0, 0) and reaching (n - 1, n - 1) by moving right or down through valid path cells (cells with value 0 or 1).

After reaching (n - 1, n - 1), returning to (0, 0) by moving left or up through valid path cells. When passing through a path cell containing a cherry, you pick it up, and the cell becomes an empty cell 0.

If there is no valid path between (0, 0) and (n - 1, n - 1), then no cherries can be collected.



**Input: grid = [[0,1,-1],[1,0,-1],[1,1,1]]**

**Output: 5**

**Explanation: The player started at (0, 0) and went down, down, right right to reach (2, 2).**

**4 cherries were picked up during this single trip, and the matrix becomes [[0,1,-1],[0,0,- 1],[0,0,0]].**

**Then, the player went left, up, up, left to return home, picking up one more cherry. The total number of cherries picked up is 5, and this is the maximum possible.**

**Example 2: Input: grid = [[1,1,-1],[1,-1,1],[-1,1,1]]**

**Output: 0**

**Constraints:**

**n == grid.length**

**n == grid[i].length 1 <= n <= 50**

**grid[i][j] is -1, 0, or 1.**

**grid[0][0] != -1**

**grid[n - 1][n - 1] != -1**

## CODE:

def cherryPickup(grid): n = len(grid)

dp = [[[float('-inf')] \* n for \_ in range(n)] for in range(n)] dp[0][0][0] = grid[0][0]

for r1 in range(n): for c1 in range(n):

for c2 in range(n): r2 = r1 + c1 - c2

if 0 <= r2 < n and grid[r1][c1] != -1 and grid[r2][c2] != -1: cherries = grid[r1][c1]

if c1 != c2:

cherries += grid[r2][c2] prev = max(

dp[r1 - 1][c1][c2] if r1 > 0 else float('-inf'),

dp[r1][c1 - 1][c2] if c1 > 0 else float('-inf'),

dp[r1 - 1][c1][c2 - 1] if r1 > 0 and c2 > 0 else float('-inf'), dp[r1][c1 - 1][c2 - 1] if c1 > 0 and c2 > 0 else float('-inf'),

)

dp[r1][c1][c2] = prev + cherries if prev != float('-inf') else float('-inf') return max(0, dp[n - 1][n - 1][n - 1])

grid = [[0, 1, -1], [1, 0, -1], [1, 1, 1]]

print(cherryPickup(grid))

