LeetCode Training Day 10 DP I Look Back

Dynamic Programming is an algorithm to resolve the problem based on result of sub problem. For example if we want to calculate the optimal result in an array with scope of [0, n], then we can see if we can build the answer from the sub result of [0,n-1], [0,n-2]…[0,0].

The iteration of the input data is normally the key for the solution. On each sub type of the DP program, they will have specific pattern.

In this chapter we will cover the basic pattern of DP.

In Dynamic Programing, we sometimes have a pattern as “To be or not to be”. This means on every position we will face a choice, to take the action or not to take the action, this will give us two states on every position. We need to keep the most optimal choice in each state.

**70. Climbing Stairs**

Easy

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

**Example 2:**

**Input:** n = 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

**Constraints:**

* 1 <= n <= 45

### Analysis:

Accumulate the previous result to each position.

/// <summary>

/// Leet code #70. Climbing Stairs

/// 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?

/// </summary>

int LeetCodeDP::climbStairs(int n)

{

vector<int> steps;

steps.push\_back(1);

steps.push\_back(1);

for (int i = 2; i <= n; i++)

{

steps.push\_back(steps[i - 2] + steps[i - 1]);

}

return steps[n];

}

## 62. Unique Paths

Medium

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?

**Example 1:**

A picture containing table

Description automatically generated

**Input:** m = 3, n = 7

**Output:** 28

**Example 2:**

**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 -> Down -> Down

2. Down -> Down -> Right

3. Down -> Right -> Down

**Example 3:**

**Input:** m = 7, n = 3

**Output:** 28

**Example 4:**

**Input:** m = 3, n = 3

**Output:** 6

**Constraints:**

* 1 <= m, n <= 100
* It's guaranteed that the answer will be less than or equal to 2 \* 109.

### Analysis:

This is a very simple problem. On every position you need to add the accumulated path from left or from top.

/// <summary>

/// Leet code #62. Unique Paths

///

/// 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 3 x 7 grid. How many possible unique paths are there?

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

/// </summary>

int LeetCodeDP::uniquePaths(int m, int n)

{

vector<vector<int>> matrix(m, vector<int>(n));

for (int i = 0; i < m; i++)

{

for (int j = 0; j < n; j++)

{

if ((i == 0) && (j == 0))

{

matrix[i][j] = 1;

}

else if (i == 0)

{

matrix[i][j] = matrix[i][j - 1];

}

else if (j == 0)

{

matrix[i][j] = matrix[i - 1][j];

}

else

{

matrix[i][j] = matrix[i][j - 1] + matrix[i - 1][j];

}

}

}

return matrix[m - 1][n - 1];

}

## 63. Unique Paths II

Medium

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 space is marked as 1 and 0 respectively in the grid.

**Example 1:**

A picture containing shoji

Description automatically generated

**Input:** obstacleGrid = [[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

**Example 2:**

Shape

Description automatically generated

**Input:** obstacleGrid = [[0,1],[0,0]]

**Output:** 1

**Constraints:**

* m == obstacleGrid.length
* n == obstacleGrid[i].length
* 1 <= m, n <= 100
* obstacleGrid[i][j] is 0 or 1.

### Analysis:

It is again a very simple problem. On every position you need to add the accumulated path from left or from top.

/// <summary>

/// Leet code #63. Unique Paths II

///

/// Follow up for "Unique Paths":

/// 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.

/// For example,

/// [

/// [0,0,0],

/// [0,1,0],

/// [0,0,0]

/// ]

/// There is one obstacle in the middle of a 3x3 grid as illustrated below.

/// The total number of unique paths is 2.

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

/// </summary>

int LeetCodeDP::uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid)

{

int m = obstacleGrid.size();

int n = obstacleGrid[0].size();

vector<vector<int>> matrix(m, vector<int>(n));

for (int i = 0; i < m; i++)

{

for (int j = 0; j < n; j++)

{

if (obstacleGrid[i][j] == 1)

{

matrix[i][j] = 0;

}

else if ((i == 0) && (j == 0))

{

matrix[i][j] = 1;

}

else if (i == 0)

{

matrix[i][j] = matrix[i][j - 1];

}

else if (j == 0)

{

matrix[i][j] = matrix[i - 1][j];

}

else

{

matrix[i][j] = matrix[i][j - 1] + matrix[i - 1][j];

}

}

}

return matrix[m - 1][n - 1];

}

## 64. Minimum Path Sum

Medium

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 1:**

Calendar

Description automatically generated

**Input:** grid = [[1,3,1],[1,5,1],[4,2,1]]

**Output:** 7

**Explanation:** Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.

**Example 2:**

**Input:** grid = [[1,2,3],[4,5,6]]

**Output:** 12

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 200
* 0 <= grid[i][j] <= 100

### Analysis:

On every position, we remember the minimum path sum to reach this position, and we can move on from left to right, top to bottom.

// <summary>

/// Leet code #64. Minimum Path Sum

///

/// 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.

/// </summary>

int LeetCodeDP::minPathSum(vector<vector<int>>& grid)

{

vector<vector<int>> matrix(grid.size(), vector<int>(grid[0].size()));

for (size\_t i = 0; i < grid.size(); i++)

{

for (size\_t j = 0; j < grid[0].size(); j++)

{

if ((i == 0) && (j == 0))

{

matrix[i][j] = grid[i][j];

}

else if (i == 0)

{

matrix[i][j] = matrix[i][j - 1] + grid[i][j];

}

else if (j == 0)

{

matrix[i][j] = matrix[i - 1][j] + grid[i][j];

}

else

{

matrix[i][j] = min(matrix[i - 1][j], matrix[i][j - 1]) + grid[i][j];

}

}

}

return matrix[grid.size() - 1][grid[0].size() - 1];

}

## 118. Pascal's Triangle

Easy

Given an integer numRows, return the first numRows of **Pascal's triangle**.

In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown:



**Example 1:**

**Input:** numRows = 5

**Output:** [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

**Example 2:**

**Input:** numRows = 1

**Output:** [[1]]

**Constraints:**

* 1 <= numRows <= 30

### Analysis:

Calculate from last line.

/// <summary>

/// Leet code #118. Pascal's Triangle

/// Given numRows, generate the first numRows of Pascal's triangle.

/// For example, given numRows = 5,

///

/// Return

/// [

/// [1],

/// [1,1],

/// [1,2,1],

/// [1,3,3,1],

/// [1,4,6,4,1]

/// ]

/// </summary>

vector<vector<int>> LeetCodeDP::generatePascalTriangle(int numRows)

{

vector<vector<int>> result;

if (numRows <= 0) return result;

for (int i = 0; i < numRows; i++)

{

vector<int> prev\_level;

if (i > 0) prev\_level = result.back();

vector<int> level;

for (int j = 0; j <= i; j++)

{

if ((j == 0) || (j == i))

{

level.push\_back(1);

}

else

{

level.push\_back(prev\_level[j - 1] + prev\_level[j]);

}

}

result.push\_back(level);

}

return result;

}

## 119. Pascal's Triangle II

Easy

Given an integer rowIndex, return the rowIndexth (**0-indexed**) row of the **Pascal's triangle**.

In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown:



**Example 1:**

**Input:** rowIndex = 3

**Output:** [1,3,3,1]

**Example 2:**

**Input:** rowIndex = 0

**Output:** [1]

**Example 3:**

**Input:** rowIndex = 1

**Output:** [1,1]

**Constraints:**

* 0 <= rowIndex <= 33

**Follow up:** Could you optimize your algorithm to use only O(rowIndex) extra space?

### Analysis:

Only keep track on last line.

// <summary>

/// Leet code #119. Pascal's Triangle II

/// Given an index k, return the kth row of the Pascal's triangle.

/// For example, given k = 3,

/// Return [1,3,3,1].

/// Note:

/// Could you optimize your algorithm to use only O(k) extra space?

/// </summary>

vector<int> LeetCodeDP::getPascalTriangleRow(int rowIndex)

{

vector<int> level;

if (rowIndex < 0) return level;

for (int i = 0; i <= rowIndex; i++)

{

vector<int> prev\_level = level;

level.clear();

for (int j = 0; j <= i; j++)

{

if ((j == 0) || (j == i))

{

level.push\_back(1);

}

else

{

level.push\_back(prev\_level[j - 1] + prev\_level[j]);

}

}

}

return level;

}

## 120. Triangle

Medium

Given a triangle array, return *the minimum path sum from top to bottom*.

For each step, you may move to an adjacent number of the row below. More formally, if you are on index i on the current row, you may move to either index i or index i + 1 on the next row.

**Example 1:**

**Input:** triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]

**Output:** 11

**Explanation:** The triangle looks like:

2

3 4

6 5 7

4 1 8 3

The minimum path sum from top to bottom is 2 + 3 + 5 + 1 = 11 (underlined above).

**Example 2:**

**Input:** triangle = [[-10]]

**Output:** -10

**Constraints:**

* 1 <= triangle.length <= 200
* triangle[0].length == 1
* triangle[i].length == triangle[i - 1].length + 1
* -104 <= triangle[i][j] <= 104

**Follow up:** Could you do this using only O(n) extra space, where n is the total number of rows in the triangle?

### Analysis:

We just need to accumulate the minimum sum on each line and pass it down, since on every position p only the previous line p-1 and p can impact current position, so we just need to remember p-1 and p from previous line. Please save them fore the sum[j] get updated.

If you want to make the life easy store the whole previous line sum is also acceptable.

/// <summary>

/// LeetCode #120. Triangle

/// 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.

/// </summary>

int LeetCodeDP::minimumTotal(vector<vector<int>>& triangle)

{

vector<int> sum(triangle.size());

if ((triangle.empty()) || (triangle[0].empty())) return 0;

for (size\_t i = 0; i < triangle.size(); i++)

{

int prev = sum[0];

for (size\_t j = 0; j < triangle[i].size(); j++)

{

int current = sum[j];

if (i == 0)

{

sum[j] = triangle[i][j];

}

else if (j == 0)

{

sum[j] = sum[j] + triangle[i][j];

}

else if (j == triangle[i].size() - 1)

{

sum[j] = prev + triangle[i][j];

}

else

{

sum[j] = min(prev, sum[j]) + triangle[i][j];

}

prev = current;

}

}

int minimum\_sum = sum[0];

for (size\_t i = 1; i < sum.size(); i++)

{

if (minimum\_sum > sum[i]) minimum\_sum = sum[i];

}

return minimum\_sum;

}

## 121. Best Time to Buy and Sell Stock

Easy

You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

**Example 1:**

**Input:** prices = [7,1,5,3,6,4]

**Output:** 5

**Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

**Example 2:**

**Input:** prices = [7,6,4,3,1]

**Output:** 0

**Explanation:** In this case, no transactions are done and the max profit = 0.

**Constraints:**

* 1 <= prices.length <= 105
* 0 <= prices[i] <= 104

### Analysis:

This is the first time we see a “To be or not to be” problem. On every day we will face some choices, shall we buy the stock if we have not bought before? Shall we sell the stock if we have bought before? Shall we take no action?

Assume at every day, the optimal buy action is that if we can get a lower price today, otherwise we should keep the previous buy price. The optimal for sell action is that if we can sell the stock today based on the best buy on previous days. If any sell action cause negative profit (loss money) then we do not do anything. A sell action should clear any buy price.

/// <summary>

/// Leet Code 121. Best Time to Buy and Sell Stock

///

/// Easy

///

/// You are given an array prices where prices[i] is the price of a given

/// stock on the ith day.

///

/// You want to maximize your profit by choosing a single day to buy one

/// stock and choosing a different day in the future to sell that stock.

///

/// Return the maximum profit you can achieve from this transaction. If

/// you cannot achieve any profit, return 0.

///

///

/// Example 1:

/// Input: prices = [7,1,5,3,6,4]

/// Output: 5

/// Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6),

/// profit = 6-1 = 5.

/// Note that buying on day 2 and selling on day 1 is not allowed because

/// you must buy before you sell.

///

/// Example 2:

/// Input: prices = [7,6,4,3,1]

/// Output: 0

/// Explanation: In this case, no transactions are done and the max

/// profit = 0.

///

/// Constraints:

/// 1. 1 <= prices.length <= 10^5

/// 2. 0 <= prices[i] <= 10^4

/// </summary>

int LeetCodeDP::maxProfitOneTxn(vector<int>& prices)

{

if (prices.size() == 0)

{

return 0;

}

int best\_buy = INT\_MAX;

int best\_profit = 0;

for (size\_t i = 0; i < prices.size(); i++)

{

best\_buy = min(best\_buy, prices[i]);

best\_profit = max(best\_profit, prices[i] - best\_buy);

}

return best\_profit;

}

## 122. Best Time to Buy and Sell Stock II

Medium

You are given an integer array prices where prices[i] is the price of a given stock on the ith day.

On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.

Find and return *the****maximum****profit you can achieve*.

**Example 1:**

**Input:** prices = [7,1,5,3,6,4]

**Output:** 7

**Explanation:** Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.

Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.

Total profit is 4 + 3 = 7.

**Example 2:**

**Input:** prices = [1,2,3,4,5]

**Output:** 4

**Explanation:** Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.

Total profit is 4.

**Example 3:**

**Input:** prices = [7,6,4,3,1]

**Output:** 0

**Explanation:** There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.

**Constraints:**

* 1 <= prices.length <= 3 \* 104
* 0 <= prices[i] <= 104

### Analysis:

On every day you can make a buy action based on previous sale state, or sell action based on previous buy state.

/// <summary>

/// Leet Code 122. Best Time to Buy and Sell Stock II

///

/// Medium

///

/// You are given an integer array prices where prices[i] is the price of

/// a given stock on the ith day.

///

/// On each day, you may decide to buy and/or sell the stock. You can only

/// hold at most one share of the stock at any time. However, you can buy

/// it then immediately sell it on the same day.

///

/// Find and return the maximum profit you can achieve.

///

/// Example 1:

/// Input: prices = [7,1,5,3,6,4]

/// Output: 7

/// Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5),

/// profit = 5-1 = 4.

/// Then buy on day 4 (price = 3) and sell on day 5 (price = 6),

/// profit = 6-3 = 3.

/// Total profit is 4 + 3 = 7.

///

/// Example 2:

/// Input: prices = [1,2,3,4,5]

/// Output: 4

/// Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5),

/// profit = 5-1 = 4.

/// Total profit is 4.

///

/// Example 3:

/// Input: prices = [7,6,4,3,1]

/// Output: 0

/// Explanation: There is no way to make a positive profit, so we never buy the

/// stock to achieve the maximum profit of 0.

///

/// Constraints:

/// 1. 1 <= prices.length <= 3 \* 10^4

/// 2. 0 <= prices[i] <= 10^4

/// </summary>

int LeetCodeDP::maxProfitManyTxns(vector<int>& prices)

{

int prev\_buy = INT\_MIN;

int prev\_sell = 0;

int result = 0;

for (size\_t i = 0; i < prices.size(); i++)

{

int buy = max(prev\_sell - prices[i], prev\_buy);

int sell = max(prev\_sell, prices[i] + prev\_buy);

result = max(result, sell);

prev\_buy = buy;

prev\_sell = sell;

}

return result;

}

## 198. House Robber

Medium

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 systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight****without alerting the police***.

**Example 1:**

**Input:** nums = [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:** nums = [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.

**Constraints:**

* 1 <= nums.length <= 100
* 0 <= nums[i] <= 400

### Analysis:

This is another “To be or not to be” problem. On every house x, you have two choice, robber this house, so you get the accumulate max value from house x-2, plus this house, or you have accumulated value in x-1 house and skip current one.

/// <summary>

/// Leet code #198. House Robber

/// 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.

/// </summary>

int LeetCodeDP::rob(vector<int>& nums)

{

if (nums.size() == 0) return 0;

vector<int> matrix(nums.size());

for (size\_t i = 0; i < nums.size(); i++)

{

// You start with first house

if (i == 0)

{

matrix[i] = nums[i];

}

// when at the second house, you simply compare the value of these two houses

else if (i == 1)

{

matrix[i] = max(nums[i - 1], nums[i]);

}

// when you have more than two houses, you need to think whether

// you should steal the current house or the previous house

else

{

matrix[i] = max(matrix[i - 2] + nums[i], matrix[i - 1]);

}

}

return matrix[nums.size() - 1];

}

## 213. House Robber II

Medium

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 a security system connected, and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight****without alerting the police***.

**Example 1:**

**Input:** nums = [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:** nums = [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 3:**

**Input:** nums = [1,2,3]

**Output:** 3

**Constraints:**

* 1 <= nums.length <= 100
* 0 <= nums[i] <= 1000

### Analysis:

You do this by try house 0 to n-2 and then try house 1 to n-1 in two iterations. On every iteration, you compare the accumulated value in 2 house before plus current house or accumulated value in one house before.

/// <summary>

/// Leet code #213. House Robber II

///

/// 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.

/// </summary>

int LeetCodeDP::robII(vector<int>& nums)

{

// special case

if (nums.empty()) return 0;

if (nums.size() == 1) return nums[0];

int last\_prev = 0;

int sum = 0;

int result = 0;

for (size\_t i = 0; i < nums.size() - 1; i++)

{

int prev = sum;

sum = max(last\_prev + nums[i], sum);

last\_prev = prev;

}

result = sum;

last\_prev = 0;

sum = 0;

for (size\_t i = 1; i < nums.size(); i++)

{

int prev = sum;

sum = max(last\_prev + nums[i], sum);

last\_prev = prev;

}

result = max(result, sum);

return result;

}

## 256. Paint House

Medium

There is a row of n houses, where each house can be painted one of 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 an n x 3 cost matrix costs.

* For example, costs[0][0] is the cost of painting house 0 with the color red; costs[1][2] is the cost of painting house 1 with color green, and so on...

Return *the minimum cost to paint all houses*.

**Example 1:**

**Input:** costs = [[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.

**Example 2:**

**Input:** costs = [[7,6,2]]

**Output:** 2

**Constraints:**

* costs.length == n
* costs[i].length == 3
* 1 <= n <= 100
* 1 <= costs[i][j] <= 20

### Analysis:

One every house the minimum cost for color X is the minimum of cost of all different colors in previous house, plus the cost of color X in this house.

/// <summary>

/// Leet code 256. Paint House

///

/// 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.

/// </summary>

int LeetCodeDP::minCost(vector<vector<int>>& costs)

{

vector<vector<int>> sum;

if (costs.size() == 0) return 0;

for (size\_t i = 0; i < costs.size(); i++)

{

sum.push\_back(vector<int>(3));

if (i == 0)

{

sum[i][0] = costs[i][0];

sum[i][1] = costs[i][1];

sum[i][2] = costs[i][2];

}

else

{

sum[i][0] = costs[i][0] + min(sum[i - 1][1], sum[i - 1][2]);

sum[i][1] = costs[i][1] + min(sum[i - 1][0], sum[i - 1][2]);

sum[i][2] = costs[i][2] + min(sum[i - 1][0], sum[i - 1][1]);

}

}

return min(min(sum[costs.size() - 1][0], sum[costs.size() - 1][1]), sum[costs.size() - 1][2]);

}

## 1746. Maximum Subarray Sum After One Operation

Medium

You are given an integer array nums. You must perform **exactly one** operation where you can **replace** one element nums[i] with nums[i] \* nums[i].

Return *the****maximum****possible subarray sum after****exactly one****operation*. The subarray must be non-empty.

**Example 1:**

**Input:** nums = [2,-1,-4,-3]

**Output:** 17

**Explanation:** You can perform the operation on index 2 (0-indexed) to make nums = [2,-1,**16**,-3]. Now, the maximum subarray sum is 2 + -1 + 16 = 17.

**Example 2:**

**Input:** nums = [1,-1,1,1,-1,-1,1]

**Output:** 4

**Explanation:** You can perform the operation on index 1 (0-indexed) to make nums = [1,**1**,1,1,-1,-1,1]. Now, the maximum subarray sum is 1 + 1 + 1 + 1 = 4.

**Constraints:**

* 1 <= nums.length <= 105
* -104 <= nums[i] <= 104

### Analysis:

On every position of the array, you can choose to replace the current item plus maximum previous sum without operation, or you can add maximum previous sum with operation plus current one. When accumulated sum become negative you can reset of zero.

/// <summary>

/// Leet code 1746. Maximum Subarray Sum After One Operation

///

/// Medium

///

/// You are given an integer array nums. You must perform exactly one

/// operation where you can replace one element nums[i] with

/// nums[i] \* nums[i].

/// Return the maximum possible subarray sum after exactly one operation.

/// The subarray must be non-empty.

///

/// Example 1:

/// Input: nums = [2,-1,-4,-3]

/// Output: 17

/// Explanation: You can perform the operation on index 2

/// (0-indexed) to make nums = [2,-1,16,-3]. Now, the maximum

/// subarray sum is 2 + -1 + 16 = 17.

///

/// Example 2:

/// Input: nums = [1,-1,1,1,-1,-1,1]

/// Output: 4

/// Explanation: You can perform the operation on index 1 (0-indexed)

/// to make nums = [1,1,1,1,-1,-1,1]. Now, the maximum subarray

/// sum is 1 + 1 + 1 + 1 = 4.

///

/// Constraints:

/// 1. 1 <= nums.length <= 10^5

/// -10^4 <= nums[i] <= 10^4

/// </summary>

int LeetCodeDP::maxSumAfterOperation(vector<int>& nums)

{

int sum = 0, sum\_op = 0;

int result = INT\_MIN;

for (size\_t i = 0; i < nums.size(); i++)

{

if (i == 0)

{

sum = nums[i];

sum\_op = nums[i] \* nums[i];

}

else

{

sum\_op = max(sum + nums[i] \* nums[i], sum\_op + nums[i]);

sum = max(0, sum + nums[i]);

}

result = max(result, sum\_op);

}

return result;

}

## 276. Paint Fence

Medium

You are painting a fence of n posts with k different colors. You must paint the posts following these rules:

* Every post must be painted **exactly one** color.
* There **cannot** be three or more **consecutive** posts with the same color.

Given the two integers n and k, return *the****number of ways****you can paint the fence*.

**Example 1:**

Icon

Description automatically generated

**Input:** n = 3, k = 2

**Output:** 6

**Explanation:** All the possibilities are shown.

Note that painting all the posts red or all the posts green is invalid because there cannot be three posts in a row with the same color.

**Example 2:**

**Input:** n = 1, k = 1

**Output:** 1

**Example 3:**

**Input:** n = 7, k = 2

**Output:** 42

**Constraints:**

* 1 <= n <= 50
* 1 <= k <= 105
* The testcases are generated such that the answer is in the range [0, 231 - 1] for the given n and k.

### Analysis:

On every fence we can count it into two categories, one is the same color as previous fence, we call it same, another is the different color as the previous fence, we call it diff. Current diff is always equal to previous same, because if you have same color previously you must paint a different color, current diff is equal to previous any category (same + diff) \* (k-1) because you can paint k-1 color (not same color) as last fence.

/// <summary>

/// Leet Code 276. Paint Fence

///

/// Medium

///

/// You are painting a fence of n posts with k different colors. You must

/// paint the posts following these rules:

///

/// Every post must be painted exactly one color.

/// There cannot be three or more consecutive posts with the same color.

/// Given the two integers n and k, return the number of ways you can paint

/// the fence.

///

/// Example 1:

/// Input: n = 3, k = 2

/// Output: 6

/// Explanation: All the possibilities are shown.

/// Note that painting all the posts red or all the posts green is invalid

/// because there cannot be three posts in a row with the same color.

///

/// Example 2:

/// Input: n = 1, k = 1

/// Output: 1

///

/// Example 3:

/// Input: n = 7, k = 2

/// Output: 42

///

/// Constraints:

/// 1. 1 <= n <= 50

/// 2. 1 <= k <= 10^5

/// 3. The testcases are generated such that the answer is in the range

/// [0, 2^31 - 1] for the given n and k.

/// </summary>

int LeetCodeDP::numWaysPaintFence(int n, int k)

{

int diff = 0, same = 0;

for (int i = 1; i <= n; i++)

{

if (i == 1)

{

diff = 0;

same = k;

}

else if (i == 2)

{

diff = k \* (k - 1);

same = k;

}

else

{

int temp = diff;

diff = same \* (k - 1) + diff \* (k - 1);

same = temp;

}

}

return same + diff;

}

# Advanced Problems

## 123. Best Time to Buy and Sell Stock III

Hard

You are given an array prices where prices[i] is the price of a given stock on the ith day.

Find the maximum profit you can achieve. You may complete **at most two transactions**.

**Note:** You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

**Example 1:**

**Input:** prices = [3,3,5,0,0,3,1,4]

**Output:** 6

**Explanation:** Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.

**Example 2:**

**Input:** prices = [1,2,3,4,5]

**Output:** 4

**Explanation:** Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.

Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engaging multiple transactions at the same time. You must sell before buying again.

**Example 3:**

**Input:** prices = [7,6,4,3,1]

**Output:** 0

**Explanation:** In this case, no transaction is done, i.e. max profit = 0.

**Example 4:**

**Input:** prices = [1]

**Output:** 0

**Constraints:**

* 1 <= prices.length <= 105
* 0 <= prices[i] <= 105

### Analysis:

On every day the first buy is based on 0 transaction, the second buy is based on first sell, the result is maximum of none transaction (0), one sell or two sell. Such method is actually extendable to K transactions

/// <summary>

/// Leet Code 123. Best Time to Buy and Sell Stock III

///

/// Hard

///

/// You are given an array prices where prices[i] is the price of a given

/// stock on the ith day.

///

/// Find the maximum profit you can achieve. You may complete at most two

/// transactions.

///

/// Note: You may not engage in multiple transactions simultaneously

/// (i.e., you must sell the stock before you buy again).

///

/// Example 1:

/// Input: prices = [3,3,5,0,0,3,1,4]

/// Output: 6

/// Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3),

/// profit = 3-0 = 3.

/// Then buy on day 7 (price = 1) and sell on day 8 (price = 4),

/// profit = 4-1 = 3.

///

/// Example 2:

/// Input: prices = [1,2,3,4,5]

/// Output: 4

/// Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5),

/// profit = 5-1 = 4.

/// Note that you cannot buy on day 1, buy on day 2 and sell them later,

/// as you are engaging multiple transactions at the same time. You must

/// sell before buying again.

///

/// Example 3:

/// Input: prices = [7,6,4,3,1]

/// Output: 0

/// Explanation: In this case, no transaction is done, i.e. max profit = 0.

///

/// Example 4:

/// Input: prices = [1]

/// Output: 0

///

/// Constraints:

/// 1. 1 <= prices.length <= 10^5

/// 2. 0 <= prices[i] <= 10^5

/// </summary>

int LeetCodeDP::maxProfitTwoTxns(vector<int>& prices)

{

vector<int> prev\_buy(2), prev\_sell(2);

prev\_buy[0] = prev\_buy[1] = INT\_MIN;

int result = 0;

for (size\_t i = 0; i < prices.size(); i++)

{

vector<int> buy(2), sell(2);

buy[0] = max(prev\_buy[0], 0 - prices[i]);

sell[0] = max(prev\_sell[0], prev\_buy[0] + prices[i]);

buy[1] = max(prev\_buy[1], prev\_sell[0] - prices[i]);

sell[1] = max(prev\_sell[1], prev\_buy[1] + prices[i]);

result = max(result, max(sell[0], sell[1]));

prev\_buy = buy;

prev\_sell = sell;

}

return result;

}

## 265. Paint House II

Hard

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 an n x k cost matrix costs.

* 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...

Return *the minimum cost to paint all houses*.

**Example 1:**

**Input:** costs = [[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.

**Example 2:**

**Input:** costs = [[1,3],[2,4]]

**Output:** 5

**Constraints:**

* costs.length == n
* costs[i].length == k
* 1 <= n <= 100
* 2 <= k <= 20
* 1 <= costs[i][j] <= 20

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

### Analysis:

On every house, you want to know the cheapest accumulated cost in the previous house with a different color, so you only need to keep track two cheapest cost on every house, with 2 colors. In the next house you definitely will get a different color from these two colors for the next house.

/// <summary>

/// Leet code #265. Paint House II

///

/// Hard

///

/// 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?

/// </summary>

int LeetCodeDP::minCostII(vector<vector<int>>& costs)

{

int result = 0;

if (costs.empty() || costs[0].empty()) return result;

int n = costs.size();

int m = costs[0].size();

vector<pair<int, int>> prev\_cost = { { INT\_MAX, -1 }, { INT\_MAX, -1 } };

for (int i = 0; i < n; i++)

{

vector<pair<int, int>> curr\_cost = { { INT\_MAX, -1 }, { INT\_MAX, -1 } };

for (int j = 0; j < m; j++)

{

pair<int, int> cost = make\_pair(costs[i][j], j);

if (i > 0)

{

for (int k = 0; k < 2; k++)

{

if (cost.second != prev\_cost[k].second)

{

cost.first += prev\_cost[k].first;

break;

}

}

}

for (int k = 0; k < 2; k++)

{

if (cost.first < curr\_cost[k].first)

{

swap(cost, curr\_cost[k]);

}

}

}

prev\_cost = curr\_cost;

}

return prev\_cost[0].first;

}