# LeetCode Day 37 Simple DP

In this chapter, we cover some simple DP algorithm. The core idea of DP is that you calculate the result based on the previous calculated result, this normally happened in the array and you need to scane the array properly.

## 562. Longest Line of Consecutive One in Matrix

Medium

Given a 01 matrix **M**, find the longest line of consecutive one in the matrix. The line could be horizontal, vertical, diagonal or anti-diagonal.

**Example:**

**Input:**

[[0,1,1,0],

[0,1,1,0],

[0,0,0,1]]

**Output:** 3

**Hint:** The number of elements in the given matrix will not exceed 10,000.

### Analysis:

There are four directions, left, up, diag and anti-diag, the previous result is either on previous rows or previous columns, so you only need one scan and just keep the result in 4 directions.

/// <summary>

/// Leet code #562. Longest Line of Consecutive One in Matrix

/// Given a 01 matrix M, find the longest line of consecutive one in the

/// matrix. The line could be horizontal, vertical, diagonal or

/// anti-diagonal.

///

/// Example:

///

/// Input:

/// [

/// [0,1,1,0],

/// [0,1,1,0],

/// [0,0,0,1]

/// ]

/// Output: 3

/// Hint: The number of elements in the given matrix will not exceed 10,000.

/// </summary>

int LeetCodeDP::longestLine(vector<vector<int>>& M)

{

vector<vector<vector<int>>> dp;

int max\_length = 0;

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

{

dp.push\_back(vector<vector<int>>(M[i].size(), vector<int>(4, 0)));

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

{

// if zero we ignore

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

{

dp[i][j][0] = (i > 0 && M[i - 1][j] == 1) ?

dp[i - 1][j][0] + 1 : 1;

dp[i][j][1] = (j > 0 && M[i][j - 1] == 1) ?

dp[i][j - 1][1] + 1 : 1;

dp[i][j][2] = (i > 0 && j > 0 && M[i - 1][j - 1] == 1) ?

dp[i - 1][j - 1][2] + 1 : 1;

dp[i][j][3] = (i > 0 && j < M[i].size() - 1 &&

M[i - 1][j + 1] == 1) ?

dp[i - 1][j + 1][3] + 1 : 1;

}

for (size\_t k = 0; k < 4; k++)

{

max\_length = max(max\_length, dp[i][j][k]);

}

}

}

return max\_length;

}

## 91. Decode Ways

Medium

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).

### Analysis:

Move forward is bettwe than look back, and add the current sum to the next position, if the current position is ‘1’ or ‘2’, we may add the current sym to the one after next position.

/// <summary>

/// Leet code 91. Decode Ways

///

/// 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).

/// </summary>

int LeetCodeDP::numDecodings(string s)

{

if (s.empty()) return 0;

vector<int> dp(s.size()+1);

dp[0] = 1;

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

{

if (s[i] != '0') dp[i+1] += dp[i];

if (i < s.size() - 1)

{

if ((s[i] == '1') ||

(s[i] == '2' && s[i + 1] >= '0' && s[i + 1] <= '6'))

{

dp[i + 2] += dp[i];

}

}

}

return dp[s.size()];

}

## 576. Out of Boundary Paths

Medium

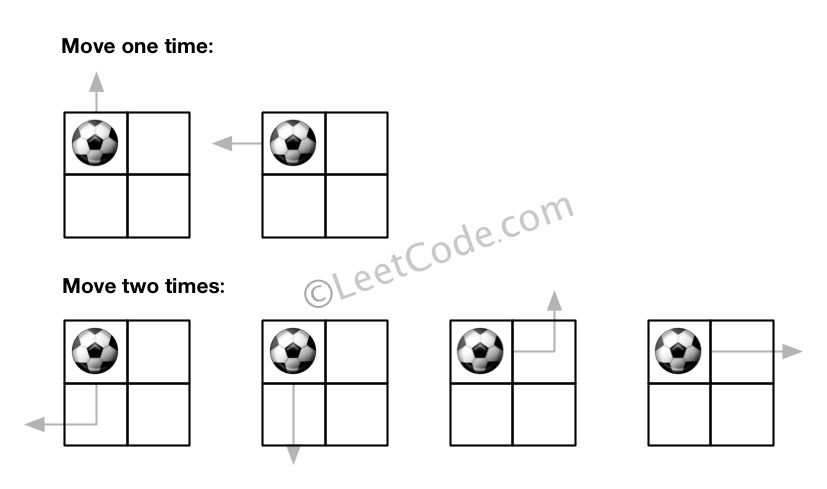
There is an **m** by **n** grid with a ball. Given the start coordinate **(i,j)** of the ball, you can move the ball to **adjacent** cell or cross the grid boundary in four directions (up, down, left, right). However, you can **at most** move **N** times. Find out the number of paths to move the ball out of grid boundary. The answer may be very large, return it after mod 109 + 7.

**Example 1:**

**Input:** m = 2, n = 2, N = 2, i = 0, j = 0

**Output:** 6

**Explanation:**

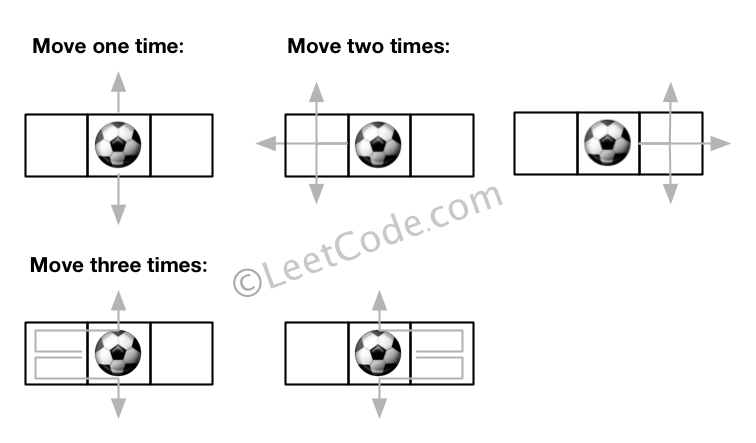


**Example 2:**

**Input:** m = 1, n = 3, N = 3, i = 0, j = 1

**Output:** 12

**Explanation:**



**Note:**

1. Once you move the ball out of boundary, you cannot move it back.
2. The length and height of the grid is in range [1,50].
3. N is in range [0,50].

### Analysis:

For such problem you need to iterate, iterate over from current position to next position, iterate from current grid to next grid.

/// <summary>

/// Leet code #576. Out of Boundary Paths

///

/// There is an m by n grid with a ball. Given the start coordinate (i,j) of

/// the ball, you can move the ball to adjacent cell or cross the grid boundary

/// in four directions (up, down, left, right). However, you can at most move N

/// times. Find out the number of paths to move the ball out of grid boundary.

/// The answer may be very large, return it after mod 10^9 + 7.

///

/// Example 1:

/// Input:m = 2, n = 2, N = 2, i = 0, j = 0

/// Output: 6

/// Explanation:

///

///

/// Example 2:

/// Input:m = 1, n = 3, N = 3, i = 0, j = 1

/// Output: 12

/// Explanation:

///

/// Note:

/// Once you move the ball out of boundary, you cannot move it back.

/// The length and height of the grid is in range [1,50].

/// N is in range [0,50].

/// </summary>

int LeetCodeDP::findPaths(int m, int n, int N, int i, int j)

{

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

dp[i][j] = 1;

int result = 0;

int M = 1000000007;

vector<vector<int>> directions = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };

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

{

vector<vector<int>> next\_dp(m, vector<int>(n));

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

{

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

{

if (dp[r][c] == 0) continue;

for (size\_t d = 0; d < directions.size(); d++)

{

int next\_r = r + directions[d][0];

int next\_c = c + directions[d][1];

if (next\_r < 0 || next\_r >= m || next\_c < 0 || next\_c >= n)

{

result = (result + dp[r][c]) % M;

}

else

{

next\_dp[next\_r][next\_c] =

(next\_dp[next\_r][next\_c] + dp[r][c]) % M;

}

}

}

}

dp = next\_dp;

}

return result;

}

## 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;

}