# 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.

## 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;

}

## 1186. Maximum Subarray Sum with One Deletion

Medium

Given an array of integers, return the maximum sum for a **non-empty** subarray (contiguous elements) with at most one element deletion. In other words, you want to choose a subarray and optionally delete one element from it so that there is still at least one element left and the sum of the remaining elements is maximum possible.

Note that the subarray needs to be **non-empty** after deleting one element.

**Example 1:**

**Input:** arr = [1,-2,0,3]

**Output:** 4

**Explanation:** Because we can choose [1, -2, 0, 3] and drop -2, thus the subarray [1, 0, 3] becomes the maximum value.

**Example 2:**

**Input:** arr = [1,-2,-2,3]

**Output:** 3

**Explanation:** We just choose [3] and it's the maximum sum.

**Example 3:**

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

**Output:** -1

**Explanation:** The final subarray needs to be non-empty. You can't choose [-1] and delete -1 from it, then get an empty subarray to make the sum equals to 0.

**Constraints:**

* 1 <= arr.length <= 10^5
* -10^4 <= arr[i] <= 10^4

### Analysis:

Consider delete one element, we can simply add the maximum subarray ending one position before current position and maximum subarray starting one position after current position.

/// <summary>

/// Leet code #1186. Maximum Subarray Sum with One Deletion

///

/// Given an array of integers, return the maximum sum for a non-empty

/// subarray (contiguous elements) with at most one element deletion.

/// In other words, you want to choose a subarray and optionally delete

/// one element from it so that there is still at least one element left

/// and the sum of the remaining elements is maximum possible.

/// Note that the subarray needs to be non-empty after deleting one element.

///

/// Example 1:

/// Input: arr = [1,-2,0,3]

/// Output: 4

/// Explanation: Because we can choose [1, -2, 0, 3] and drop -2, thus the

/// subarray [1, 0, 3] becomes the maximum value.

///

/// Example 2:

/// Input: arr = [1,-2,-2,3]

/// Output: 3

/// Explanation: We just choose [3] and it's the maximum sum.

///

/// Example 3:

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

/// Output: -1

/// Explanation: The final subarray needs to be non-empty. You can't

/// choose [-1] and delete -1 from it, then get an empty subarray to make

/// the sum equals to 0.

///

/// Constraints:

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

/// 2. -10^4 <= arr[i] <= 10^4

/// </summary>

int LeetCodeArray::maximumSum(vector<int>& arr)

{

vector<int> dp1(arr.size()), dp2(arr.size());

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

{

dp1[i] = arr[i];

if (i > 0) dp1[i] = max(dp1[i-1] + arr[i], dp1[i]);

}

for (int i = arr.size() - 1; i >=0; i--)

{

dp2[i] = arr[i];

if (i < arr.size() - 1) dp2[i] = max(dp2[i + 1] + arr[i], dp2[i]);

}

int result = INT\_MIN;

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

{

result = max(result, dp1[i]);

if (i > 0 && i < arr.size() - 1)

{

result = max(result, dp1[i - 1] + dp2[i + 1]);

}

}

return result;

}