<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iv/description/>

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

Find the maximum profit you can achieve. You may complete at most k transactions: i.e. you may buy at most k times and sell at most k times.

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

**Example 1:**

Input: k = 2, prices = [2,4,1]

Output: 2

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

**Example 2:**

Input: k = 2, prices = [3,2,6,5,0,3]

Output: 7

Explanation: Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit = 6-2 = 4. Then buy on day 5 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

**Constraints:**

1 <= k <= 100

1 <= prices.length <= 1000

0 <= prices[i] <= 1000

**Attempt 1: 2023-11-02**

**Solution 1: Native DFS (10 min, TLE 206/210)**

class Solution {

public int maxProfit(int k, int[] prices) {

return helper(prices, 0, 1, k);

}

private int helper(int[] prices, int index, int buy, int limit) {

if(index >= prices.length || limit <= 0) {

return 0;

}

int profit = 0;

if(buy == 1) {

int not\_buy = helper(prices, index + 1, 1, limit);

int buy\_it = helper(prices, index + 1, 0, limit) - prices[index];

profit = Math.max(not\_buy, buy\_it);

} else {

int not\_sell = helper(prices, index + 1, 0, limit);

int sell\_it = helper(prices, index + 1, 1, limit - 1) + prices[index];

profit = Math.max(not\_sell, sell\_it);

}

return profit;

}

}

Time Complexity:O(2^n)

Space Complexity:O(n)

**Solution 2: DFS + Memoization (10 min)**

class Solution {

public int maxProfit(int k, int[] prices) {

Integer[][][] memo = new Integer[prices.length + 1][2][k + 1];

return helper(prices, 0, 1, k, memo);

}

private int helper(int[] prices, int index, int buy, int limit, Integer[][][] memo) {

if(index >= prices.length || limit <= 0) {

return 0;

}

if(memo[index][buy][limit] != null) {

return memo[index][buy][limit];

}

int profit = 0;

if(buy == 1) {

int not\_buy = helper(prices, index + 1, 1, limit, memo);

int buy\_it = helper(prices, index + 1, 0, limit, memo) - prices[index];

profit = Math.max(not\_buy, buy\_it);

} else {

int not\_sell = helper(prices, index + 1, 0, limit, memo);

int sell\_it = helper(prices, index + 1, 1, limit - 1, memo) + prices[index];

profit = Math.max(not\_sell, sell\_it);

}

return memo[index][buy][limit] = profit;

}

}

Time Complexity:O(2\*k\*N)

Space Complexity:O(2\*k\*N) + O(N)

**Solution 3: DP (10 min)**

class Solution {

public int maxProfit(int k, int[] prices) {

int n = prices.length;

int[][][] dp = new int[n + 1][2][k + 1];

dp[n][0][0] = 0;

for(int i = n - 1; i >= 0; i--) {

for(int buy = 0; buy <= 1; buy++) {

for(int limit = 1; limit <= k; limit++) {

int profit = 0;

if(buy == 1) {

profit = Math.max(dp[i + 1][1][limit], dp[i + 1][0][limit] - prices[i]);

} else {

profit = Math.max(dp[i + 1][0][limit], dp[i + 1][1][limit - 1] + prices[i]);

}

dp[i][buy][limit] = profit;

}

}

}

return dp[0][1][k];

}

}

Time Complexity:O(2\*k\*N)

Space Complexity:O(2\*k\*N)

**Solution 4: DP + Space Optimization (10 min)**

class Solution {

public int maxProfit(int k, int[] prices) {

int n = prices.length;

int[][] dpPrev = new int[2][k + 1];

int[][] dp = new int[2][k + 1];

dp[0][0] = 0;

for(int i = n - 1; i >= 0; i--) {

for(int buy = 0; buy <= 1; buy++) {

for(int limit = 1; limit <= k; limit++) {

int profit = 0;

if(buy == 1) {

profit = Math.max(dpPrev[1][limit], dpPrev[0][limit] - prices[i]);

} else {

profit = Math.max(dpPrev[0][limit], dpPrev[1][limit - 1] + prices[i]);

}

dp[buy][limit] = profit;

}

}

dpPrev = dp.clone();

}

return dp[1][k];

}

}

Time Complexity:O(2\*k\*N)

Space Complexity:O(2\*k)

**Solution 5: State Machine (30 min)**

class Solution {

public int maxProfit(int k, int[] prices) {

int[] buy = new int[k + 1];

int[] sell = new int[k + 1];

Arrays.fill(buy, Integer.MIN\_VALUE);

for(int price : prices) {

for(int i = 1; i <= k; i++) {

buy[i] = Math.max(buy[i], sell[i - 1] - price);

sell[i] = Math.max(sell[i], buy[i] + price);

}

}

return sell[k];

}

}

Time Complexity:O(N)

Space Complexity:O(N)

**Refer to**

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iv/solutions/54125/very-understandable-solution-by-reusing-problem-iii-idea/>

Re: [A Concise DP Solution in Java](https://leetcode.com/topic/8984/a-concise-dp-solution-in-java)

In Problem III (At most two transaction), I try to understand and solve this problem from state machine perspective inspired by this amazing post: <https://discuss.leetcode.com/topic/30680/share-my-dp-solution-by-state-machine-thinking>

Then we conclude that we can use constant variable to represent 4 states and get a very concise solution as followed.

public int maxProfit(int[] prices) {

int buy1 = Integer.MIN\_VALUE, sell1 = 0, buy2 = Integer.MIN\_VALUE, sell2 = 0;

for (int price : prices) {

buy1 = Math.max(buy1, -price);

sell1 = Math.max(sell1, buy1 + price);

buy2 = Math.max(buy2, sell1 - price);

sell2 = Math.max(sell2, buy2 + price);

}

return sell2;

}

Now for Problem IV, we can make at most K transaction rather than only two. Why not reuse the idea above? The only edge case is the first buy which has no previous sell. So here we create two int[k + 1] array to use sell[0] as a buffer region. Here is the solution.

public int maxProfit(int k, int[] prices) {

int[] buy = new int[k + 1], sell = new int[k + 1];

Arrays.fill(buy, Integer.MIN\_VALUE);

for (int price : prices) {

for (int i = 1; i <= k; i++) {

buy[i] = Math.max(buy[i], sell[i - 1] - price);

sell[i] = Math.max(sell[i], buy[i] + price);

}

}

return sell[k];

}

Yes, that's all. While this will TLE for large test case, so it's necessary to add a special case handling to pass that. But don't be confused, the basic idea is only the piece of data above indeed.

if (k >= prices.length / 2) { // if k >= n/2, then you can make maximum number of transactions

int profit = 0;

for (int i = 1; i < prices.length; i++)

if (prices[i] > prices[i - 1]) profit += prices[i] - prices[i - 1];

return profit;

}