# 算法

## 第一章 动态规划

动态规划需要满足的两个基本条件：

1、最优子结构；

2、重叠子问题；

### 1.1 除数博弈（Week0001\_\_20190714）

**1025. Divisor Game（easy）**

Alice and Bob take turns playing a game, with Alice starting first.

Initially, there is a number N on the chalkboard. On each player's turn, that player makes a move consisting of:

Choosing any x with 0 < x < N and N % x == 0.

Replacing the number N on the chalkboard with N - x.

Also, if a player cannot make a move, they lose the game.

Return True if and only if Alice wins the game, assuming both players play optimally.

**Example 1:**

**Input:** 2

**Output:** true

**Explanation:** Alice chooses 1, and Bob has no more moves.

**Example 2:**

**Input:** 3

**Output:** false

**Explanation:** Alice chooses 1, Bob chooses 1, and Alice has no more moves.

**Note:**

1. 1 <= N <= 1000

解答1(动态规划)：本题首先我们对N = 1~10进行分析，发现后面的数值计算依赖于前面比其小的数值，所以我们可以采用动态规划算法，构建dp[]。

class Solution {

public:

bool divisorGame(int N) {

if(N > 1000 || N < 1)

return false;

vector<bool> dp(1001, false);

dp[1] = false;

for(int value = 2; value <= N; value++)

{

for(int factor = 1; factor < value; factor++)

{

if((value % factor == 0) && !dp[value-factor])

{

dp[value] = true;

break;

}

}

}

return dp[N];

}

};

解答2(数学)：须知：奇数的因子必为奇数，奇数与其因子之差必为偶数；偶数的因子或奇或偶，偶数与其因子之差或奇或偶。本题博弈中，得奇数者必败，得偶数者必胜。采用数学归纳法证明如下：

1、显然divisorGame(1)== false和divisorGame(2)== true正确；

2、假设对于任何正数n，divisorGame(2 \* n – 1)== false和divisorGame(2 \* n)== true。我们将证明它是2 \* (n + 1) -1和2 \* (n + 1)。

2.1、2 \* (n + 1) -1是奇数，所以它的任何因子都是奇数，因此差值2 \* (n + 1) -1 - x（x是任意2 \* (n + 1) -1的因子）是偶数，但正如我们所知divisorGame（2 \* n）== true对于任何正数n都是真，这意味着任何可能的Alice的动作都会引起Bob的胜利，所以divisorGame(2 \* (n + 1) -1) == false。

2.2、2 \* (n + 1)是偶数，Alice只需选择x == 1.因为我们在2.1中得到，divisorGame(2 \* (n + 1) -1) == false，所以divisorGame(2 \* (n + 1))== true。

class Solution {

public:

bool divisorGame(int N) {

if(N > 1000 || N < 1)

return false;

return !(N & 1);

}

};

### 1.2 买卖股票的最佳时机（Week0001\_\_20190720）

股票问题系列可以采用动态规划问题来解答。我们假设T[i][k][0]代表第i天，最多k次交易，完成第i天动作手里目前有0支股票时的最大收益；T[i][k][1]代表第i天，最多k次交易，完成第i天动作手里目前有1支股票时的最大收益。

基本条件满足：

T[-1][k][0] = 0，T[-1][k][1] = -Infinity；

T[i][0][0] = 0，T[i][0][1] = -Infinity；

T[-1][k][0]和T[i][0][0]代表事实，所以为0；T[-1][k][1]和T[i][0][1]代表不可能发生的事，所以为负无穷大。

通用递推式如下：

T[i][k][0] = max(T[i-1][k][0]，T[i-1][k][1] + prices[i])

T[i][k][1] = max(T[i-1][k][1]，T[i-1][k-1][0] - prices[i])

参考链接如下：https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/discuss/108870/Most-consistent-ways-of-dealing-with-the-series-of-stock-problems

#### 1.2.1买卖股票的最佳时机I

**121. Best Time to Buy and Sell Stock（easy）**

Say you have an array for which the ith element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (i.e., buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Note that you cannot sell a stock before you buy one.

Example 1:

Input: [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.

Not 7-1 = 6, as selling price needs to be larger than buying price.

Example 2:

Input: [7,6,4,3,1]

Output: 0

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

解答1：初步分析，我们发现要使收益最大，则需低买高卖，且差值最大，这样问题抽象为数学即是，找到最大的一对数，不要求连续，但这对数的差值最大（后面的数减去前面的数）。我们首先想到的是采用暴力法遍历所有可能的数值对。

class Solution {

public:

int maxProfit(vector<int>& prices) {

int len = prices.size();

int result = 0;

for(int i = 0; i < len-1; i++)

{

for(int j = i+1; j < len; j++)

{

int value = prices[j] - prices[i];

if(value > result)

result = value;

}

}

return result;

}

};

解法2：暴力法时间复杂度为，我们可以继续优化。其实，数组中的某个数并不需要与前面每个数都计算差值，只要和它之前的最小值计算差值即可。

class Solution {

public:

int maxProfit(vector<int>& prices) {

int len = prices.size();

int minValue = 0x7fffffff;

int result = 0;

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

{

if(prices[i] < minValue)

minValue = prices[i];

else if(prices[i] - minValue > result)

result = prices[i] - minValue;

}

return result;

}

};

解法3：由于e-a等价于b-a+c-b+d-c+e-d，即该问题可以等价于求连续子数组的最大和问题，不过，数组要变为差值数组。

class Solution {

public:

int maxProfit(vector<int>& prices) {

int len = prices.size();

int profit = 0;

int maxprofit = 0;

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

{

profit = max(0, profit += (prices[i]-prices[i-1]));

maxprofit = max(profit, maxprofit);

}

return maxprofit;

}

};

解法4：采用动态规划递推式，这里k等于1；其递推式如下：

T[i][1][0] = max(T[i-1][1][0]，T[i-1][1][1] + prices[i])

T[i][1][1] = max(T[i-1][1][1]，-prices[i])

这里由于i只依赖与i-1，所以可以采用空间压缩。

未采用空间压缩代码：

class Solution {

public:

int maxProfit(vector<int>& prices) {

int len = prices.size();

if(len < 2)

return 0;

int T[len+1][2];

T[0][0] = 0;

T[0][1] = 0x80000000;

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

{

T[i][0] = max(T[i-1][0], T[i-1][1] + prices[i-1]);

T[i][1] = max(T[i-1][1], -prices[i-1]);

}

return T[len][0];

}

};

采用空间压缩代码：

class Solution {

public:

int maxProfit(vector<int>& prices) {

int len = prices.size();

if(len < 2)

return 0;

int T\_i10 = 0;

int T\_i11 = 0x80000000;

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

{

T\_i10 = max(T\_i10, T\_i11 + prices[i]);

T\_i11 = max(T\_i11, -prices[i]);

}

return T\_i10;

}

};

#### 1.2.2买卖股票的最佳时机II

**122. Best Time to Buy and Sell Stock II（easy）**

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

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

Example 1:

Input: [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.

Example 2:

Input: [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: [7,6,4,3,1]

Output: 0

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

解答1：本题与上题的主要区别在于，可以交易多次，也就是说对于数组求出数组中任意相邻数之差的差值数组，统计差值数组中的正数即可。

class Solution {

public:

int maxProfit(vector<int>& prices) {

int len = prices.size();

int result = 0;

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

{

if(prices[i] - prices[i-1] > 0)

{

result += (prices[i] - prices[i-1]);

}

}

return result;

}

};

解答2：采用动态规划递推式，这里k等于正无穷大；所以其中：

T[i-1][k-1][0] = T[i-1][k][0]

其递推式如下：

T[i][k][0] = max(T[i-1][k][0]，T[i-1][k][1] + prices[i])

T[i][k][1] = max(T[i-1][k][1]，T[i-1][k][0] - prices[i])

这里由于i只依赖与i-1，所以可以采用空间压缩。

采用空间压缩代码：

class Solution {

public:

int maxProfit(vector<int>& prices) {

int len = prices.size();

if(len < 2)

return 0;

int T\_ik0 = 0;

int T\_ik1 = 0x80000000;

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

{

int T\_ik0\_old = T\_ik0;

T\_ik0 = max(T\_ik0, T\_ik1 + prices[i]);

T\_ik1 = max(T\_ik1, T\_ik0\_old-prices[i]);

}

return T\_ik0;

}

};

#### 1.2.3买卖股票的最佳时机III

**123. Best Time to Buy and Sell Stock III（hard）**

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most two transactions.

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

Example 1:

Input: [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: [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: [7,6,4,3,1]

Output: 0

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

解答1：题目中要求最大交易次数不大于2次，那么我们可以将最大收益看做两次交易之和，求取最大值。

class Solution {

public:

int maxProfit(vector<int>& prices) {

int len = prices.size();

if(len <= 1)

return 0;

vector<int> firstMaxProfit(len);

int minPrice = prices[0];

int firstMax = 0;

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

{

if(prices[i] < minPrice)

minPrice = prices[i];

else if(prices[i] - minPrice > firstMax)

firstMax = prices[i] - minPrice;

firstMaxProfit[i] = firstMax;

}

vector<int> secondMaxProfit(len);

int maxPrice = prices[len-1];

int secondMax = 0;

for(int i = len-1; i >= 0; i--)

{

if(prices[i] > maxPrice)

maxPrice = prices[i];

else if(maxPrice - prices[i] > secondMax)

secondMax = maxPrice - prices[i];

secondMaxProfit[i] = secondMax;

}

int maxProfit = firstMaxProfit[len-1];

for(int i = 0; i < len-1; i++)

{

if(firstMaxProfit[i] + secondMaxProfit[i+1] > maxProfit)

maxProfit = firstMaxProfit[i] + secondMaxProfit[i+1];

}

return maxProfit;

}

};

解答2：采用动态规划法，其递推式如下：

T[i][2][0] = max(T[i-1][2][0], T[i-1][2][1] + prices[i])

T[i][2][1] = max(T[i-1][2][1], T[i-1][1][0] - prices[i])

T[i][1][0] = max(T[i-1][1][0], T[i-1][1][1] + prices[i])

T[i][1][1] = max(T[i-1][1][1], -prices[i])

采用空间压缩法代码如下：

class Solution {

public:

int maxProfit(vector<int>& prices) {

int len = prices.size();

if(len < 2)

return 0;

int T\_i10 = 0;

int T\_i11 = 0x80000000;

int T\_i20 = 0;

int T\_i21 = 0x80000000;

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

{

T\_i20 = max(T\_i20, T\_i21 + prices[i]);

T\_i21 = max(T\_i21, T\_i10 - prices[i]);

T\_i10 = max(T\_i10, T\_i11 + prices[i]);

T\_i11 = max(T\_i11, -prices[i]);

}

return T\_i20;

}

};

#### 1.2.4买卖股票的最佳时机IV

**188. Best Time to Buy and Sell Stock IV（hard）**

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most k transactions.

Note:You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Example 1:

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

Output: 2

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

Example 2:

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

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.

解答1：此处需要对k进行分别处理，如果k大于prices数组长度的一半时，其处理方式等同于k等于正无穷大；

class Solution {

public:

int maxProfit(int k, vector<int>& prices) {

int len = prices.size();

if(len < 2)

return 0;

if(k > len/2)

{

int T\_ik0 = 0;

int T\_ik1 = 0x80000000;

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

{

int T\_ik0\_old = T\_ik0;

T\_ik0 = max(T\_ik0, T\_ik1 + prices[i]);

T\_ik1 = max(T\_ik1, T\_ik0\_old - prices[i]);

}

return T\_ik0;

}

int T\_ik0[k+1];

int T\_ik1[k+1];

memset(T\_ik0, 0, sizeof(T\_ik0));

for(int i = 0; i < k+1; i++)

{

T\_ik1[i] = 0x80000000;

}

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

{

for(int j = k; j > 0; j--)

{

T\_ik0[j] = max(T\_ik0[j], T\_ik1[j] + prices[i]);

T\_ik1[j] = max(T\_ik1[j], T\_ik0[j-1] - prices[i]);

}

}

return T\_ik0[k];

}

};

#### 1.2.5带有冷却时间的买卖股票最好时间

**309. Best Time to Buy and Sell Stock with Cooldown（medium）**

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

Example:

Input: [1,2,3,0,2]

Output: 3

Explanation: transactions = [buy, sell, cooldown, buy, sell]

解法1：本题是k取正无穷大，且冷却一天，采用动态规划法，其递推公式如下：

T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i])

T[i][k][1] = max(T[i-1][k][1], T[i-2][k][0] - prices[i])

采用空间压缩法代码如下：

class Solution {

public:

int maxProfit(vector<int>& prices) {

int len = prices.size();

if(len < 2)

return 0;

int T\_ik0 = 0;

int T\_ik1 = 0x80000000;

int T\_ik0\_pre = T\_ik0;

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

{

int T\_ik0\_old = T\_ik0;

T\_ik0 = max(T\_ik0, T\_ik1 + prices[i]);

T\_ik1 = max(T\_ik1, T\_ik0\_pre - prices[i]);

T\_ik0\_pre = T\_ik0\_old;

}

return T\_ik0;

}

};

#### 1.2.6买卖股票的最佳时机含手续费

**714. Best Time to Buy and Sell Stock with Transaction Fee（medium）**

Your are given an array of integers prices, for which the i-th element is the price of a given stock on day i; and a non-negative integer fee representing a transaction fee.

You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction. You may not buy more than 1 share of a stock at a time (ie. you must sell the stock share before you buy again.)

Return the maximum profit you can make.

Example 1:

Input: prices = [1, 3, 2, 8, 4, 9], fee = 2

Output: 8

Explanation: The maximum profit can be achieved by:

Buying at prices[0] = 1

Selling at prices[3] = 8

Buying at prices[4] = 4

Selling at prices[5] = 9

The total profit is ((8 - 1) - 2) + ((9 - 4) - 2) = 8.

Note:

0 < prices.length <= 50000.

0 < prices[i] < 50000.

0 <= fee < 50000.

解法1：本题是k取正无穷大，且每笔交易需支付手续费，采用动态规划法，其递推公式如下：

T[i][k][0] = max(T[i-1][k][0], T[i-1][k][1] + prices[i] -fee)

T[i][k][1] = max(T[i-1][k][1], T[i-1][k][0] - prices[i])

采用空间压缩法代码如下：

class Solution {

public:

int maxProfit(vector<int>& prices, int fee) {

int len = prices.size();

if(len < 2)

return 0;

int T\_ik0 = 0;

int T\_ik1 = -prices[0];

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

{

int T\_ik0\_old = T\_ik0;

T\_ik0 = max(T\_ik0, T\_ik1 + prices[i] - fee);

T\_ik1 = max(T\_ik1, T\_ik0\_old - prices[i]);

}

return T\_ik0;

}

};