

第一题

递归情况下，如果n取得足够大，暂且不说费时的问题，直接就会因为递归次数太多，函数堆栈溢出而程序奔溃。

用一个数组来保存曾经计算过的数据来避免重复计算。这种思想便是**动态规划**。

动态转移的基本思想可以认为是建立起**某一状态**和**之前状态**的一种**转移表示**。通过不断循环重复这一状态转换进行迭代直到找到最优解。

1. 最优子结构
2. 边界条件
3. 状态转移方程

动态规划一般也只能应用于有最优子结构的问题。最优子结构的意思是局部最优解能决定全局最优解(对有些问题这个要求并不能完全满足，故有时需要引入一定的近似)。简单地说，问题能够分解成子问题来解决。

状态转换方程：就是原结构向子结构转移的方程。

第二题

若要求解第三题，只需要修改cooldown的值

- 第二题：cooldown = 2：卖完过两天才能买
- 第三题：cooldown = 0：卖完当天就能买

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main() {
    vector<int> prices = { 7,1,5,3,6,4 }; //1,2,9,1,6,0 /1,2,3,0,2/7,1,5,3,6,4
    int day = prices.size();
    int (*dp)[2] = new int[day][2];
    int cooldown = 2; //卖完过两天才能买
    //int cooldown = 0; //卖当天就能买
    /*-----首先将第i天分为--持有、不持有-----*/
    /*i表示第i天，0or1表示现在的状态是持有股票还是不持有股票。
    dp[2][1]表示的状态是第2天，此时手上还有股票。
    dp[2][0]表示的状态是第2天，此时手上没有股票。*/

    dp[0][0] = 0, dp[0][1] = -prices[0];
    for (int i = 1; i < day; i++) {
        /*dp[i][0] = max(dp[i-1][0],dp[i-1][1] + prices[i])
           max(选择 rest,          选择 sell)
        解释：今天我没有持有股票，有两种可能：
        要么是我昨天就没有持有，然后今天选择 rest，所以我今天还是没有持有；
        要么是我昨天持有股票，但是今天我 sell 了，所以我今天没有持有股票了。*/
        dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] + prices[i]);

        /*dp[i][1] = max(dp[i-1][1],dp[i-1-cooldown][0]-prices[i])
           max(    选择 rest    ,          选择 buy          )
        解释：今天我持有股票，有两种可能：
        要么是我昨天持有股票，然后今天选择 buy，所以我今天还是持有；
        要么是我昨天没有持有股票，但是今天我 buy 了，所以我今天持有股票了。*/
        dp[i][1] = max(dp[i - 1][1], dp[i - cooldown][0] - prices[i]);
    }
```

解释：今天我持有着股票，有两种可能：

要么我昨天就持有着股票，然后今天选择 **rest**，所以我今天还持有着股票；

要么我`cooldown`期前本没有持有，但今天我选择 **buy**，所以今天我就持有股票了。*/

```
if (i <= cooldown-1) {
    dp[i][1] = max(dp[i - 1][1], dp[0][0] - prices[i]);
}
else {
    dp[i][1] = max(dp[i - 1][1], dp[i - cooldown][0] - prices[i]);
}
}

// 倒叙get所有操作
string* ope = new string[day];
int* hold = new int[day]; // 0不持有，1持有
hold[day - 1] = false;
for (int i = day-1; i > 0; i--) {
    if (hold[i] == 0) { //第i天不持有
        if (dp[i][0] == dp[i - 1][0]) { // 昨天就没有持有，然后今天选择 rest
            ope[i] = "do nothing";
            hold[i - 1] = hold[i];
        }
        else { //昨天持有股票，但是今天我 sell
            ope[i] = "sell";
            hold[i-1] = true; // 前一天持有
        }
    }
    else { // 第i天持有
        if (dp[i][1] == dp[i - 1][1]){ // 昨天就持有，然后今天选择 rest
            ope[i] = "do nothing";
            hold[i - 1] = hold[i];
        }
        else { //cooldown前本没有持有，但今天我选择 buy
            ope[i] = "buy";
            for (int j = 1; j < cooldown; j++) {
                ope[--i] = "cooldown";
            }
            hold[i-1] = false; // cooldown前未持有
        }
    }
}
}
if (hold[0] == 1)
    ope[0] = "buy";
else
    ope[0] = "rest";

return 0;
}
```

例一：

```

6   int main() {
7       vector<int> prices = { 1, 2, 3, 0, 2 }; //1, 2, 9, 1, 6, 0 /1, 2, 3, 0, 2 /7, 1, 5, 3
8       int day = prices.size();
9       int (*dp)[2] = new int[day][2];
10      int cooldown = 2; //卖完过两天才能买
11      //int cooldown = 0; //卖当天就能买
12
13      //i表示第i天 0或1表示现在状态是持有股票还是不持有股票
14  }

```

未找到相关问题

名称	值	类型
dp,5	0x000001d669b2bc20 {0x000001d669b2bc20 {0, -1}, 0x000001d6...	int[5][2]
dp[0]	0x000001d669b2bc20 {0, -1}	int[2]
dp[1]	0x000001d669b2bc28 {1, -1}	int[2]
dp[2]	0x000001d669b2bc30 {2, -1}	int[2]
dp[3]	0x000001d669b2bc38 {2, 1}	int[2]
dp[4]	0x000001d669b2bc40 {3, 1}	int[2]
ope,5	0x000001d669b30b18 {"buy", "sell", "cooldown", "buy"...	std::string[5]
ope[0]	"buy"	std::string
ope[1]	"sell"	std::string
ope[2]	"cooldown"	std::string
ope[3]	"buy"	std::string
ope[4]	"sell"	std::string

其中Output = dp[4][0] = 3

第三题

```

int maxProfit(vector<int>& prices) {
    int totalProfite = 0;
    for (int i = 1; i < prices.size(); i++){
        //只要当天利润P>0, 买卖股票, 利润增加。如果当天利润P≤0, 不进行操作。
        if (prices[i - 1] < prices[i])
            totalProfite += (prices[i]-prices[i-1]);
        //prices[i]-prices[i-1]记入总利润
    }
    return totalProfite;
}

```

ope,6	0x0000022746d742b8 {"do nothing", "buy", "s
ope[0]	"do nothing"
ope[1]	"buy"
ope[2]	"sell"
ope[3]	"buy"
ope[4]	"sell"
ope[5]	"do nothing"