算法讨论班第 26 期——陈良涛

2016 年 3 月 17 日

309. Best Time to Buy and Sell Stock with Cooldown

Say you have an array for which the i^{th} 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:

```
prices = [1, 2, 3, 0, 2]

maxProfit = 3

transactions = [buy, sell, cooldown, buy, sell]
```

解题思路：

这是一个利用动态规划求最大值的问题，特殊之处在于每一天面临两种选择。第一种选择是买或者不买，第二种选择是卖或者不卖。因此用两个数组分别记录这两种情况下第 i 天的最大收益，S[i]表示第 i 天卖或者不卖获得第最大收益，B[i]表示第 i 天买或者不买可以获得的最大收益。因此可以写出状态转移方程：

$$S[i]=\max\{S[i-1],B[i-1]+prices[i]\}$$

$$B[i]=\max\{B[i-1],S[i-2]-prices[i]\}$$

S 的状态转移方程表示的含义是：第 i 天选择卖与不卖，如果不卖，状态没有迁移，S[i]＝S[i－1]，如果卖，说明第 i 天之前肯定买过，B[i-1]已经记录里前 i－1 天买股票可以获得的最大收益，所以，S[i]＝B[i-1]+prices[i]。

B 的状态转移方程表示的含义是：第 i 天选择买与不买，如果不买，状态没有迁移，B[i]＝B[i－1]，如果买，因为题目要求卖掉之后休息一天才能继续买，且 S[i-2]记录里前 i－2 天卖股票可以获得的最大收益，因此 B[i]=S[i-2]-prices[i]。

代码如下：

```java
public class Solution {
    public int maxProfit(int[] prices) {
        if(prices==null||prices.length<2){
            return 0;
        }
        int len = prices.length;
        int[] sellPro = new int[len];
        int[] buyPro = new int[len];
        buyPro[0] -= prices[0];
        sellPro[1] = Math.max(sellPro[0],buyPro[0]+prices[1]);
        buyPro[1] = Math.max(buyPro[0],0-prices[1]);
        for(int i=2;i<len;i++){
            sellPro[i] = Math.max(sellPro[i-1],buyPro[i-1]+prices[i]);
            buyPro[i] = Math.max(buyPro[i-1],sellPro[i-2]-prices[i]);
        }
        return Math.max(sellPro[len-1],buyPro[len-1]);
    }
}
```

时间复杂的为 O（n），空间复杂度为：O（n）。

310. Minimum Height Trees

For a undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.

Format

The graph contains `n` nodes which are labeled from `0` to `n - 1`. You will be given the number `n` and a list of undirected `edges` (each edge is a pair of labels).

You can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, `[0, 1]` is the same as `[1, 0]` and thus will not appear together in `edges`.

Example 1:

Given `n = 4`, `edges = [[1, 0], [1, 2], [1, 3]]`

```
        0
        |
        1
```

```
    / \
    2   3
```

return `[1]`

Example 2:

Given `n = 6`, `edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]`

```
  0  1  2
   \ | /
     3
     |
     4
     |
     5
```

return `[3, 4]`

解题思路：

如果想把一条裤子装进箱子里，如果箱子不够长，我们通常会把裤子从中间折叠一次，再放进箱子里（这个比喻可能不是很恰当）。本题也是同样的道理，一张图肯定存在至少一条最长路径，并且多条最长路径肯定存在共同的中点 M（可以进行数学证明），以 M 为根节点把图拎起来，则得到 MHT。

那么如何得到最长路径呢？随机选择一个节点 u，用深度优先遍历，记下每个节点到 u 的距离，离 u 最远的点 v 即为最长路径的一个端点。然后从 v 开始再进行一次深度优先遍历，离 v 最远的点即为最长路径的另一个端点。期间把路径记下来，即可轻易得到中点 M。

代码如下：

```java
 //bfs find dist to each node
 private void bfs(List<Integer>[] es,int[] dist,int[] pre,int start,int n){
     boolean[] visited = new boolean[n];
     Queue<Integer> queue = new ArrayDeque<Integer>();
     queue.add(start);
     dist[start] = 0;
     pre[start] = -1;
     visited[start] = true;
     while(!queue.isEmpty()){
         int u = queue.poll();
         for(int v:es[u]){
             if(!visited[v]){
                 visited[v] = true;
                 dist[v] = dist[u] + 1;
                 pre[v] = u;
                 queue.add(v);
             }
         }
     }
 }


public List<Integer> findMinHeightTrees(int n, int[][] edges) {
    List<Integer> res = new ArrayList<Integer>();
    if(n<1||edges==null) return res;
    //es records nodes that each node can reach
    List<Integer>[] es = new List[n];
    for(int i=0;i<n;i++) es[i] = new ArrayList<Integer>();
    for(int i=0;i<edges.length;i++){
        int u = edges[i][0];
        int v = edges[i][1];
        es[u].add(v);
        es[v].add(u);
    }
    int[] tmpDist = new int[n];
    int[] pre = new int[n];
    bfs(es,tmpDist,pre,0,n);
    //find the farthest node from 0
    int farthest = 0;
    for(int i=0;i<n;i++){
        if(tmpDist[farthest]<tmpDist[i]) farthest = i;
    }
    //find the longest path in the graph
    int[] dist = new int[n];
    bfs(es,dist,pre,farthest,n);
    int start = 0;
    for(int i=0;i<n;i++){
        if(dist[i]>dist[start]) start = i;
    }
    //make the longest path
    List<Integer> path = new ArrayList<Integer>();
    path.add(start);
    while(pre[start]!=-1){
        start = pre[start];
        path.add(start);
    }
    if(path.size()%2==1) return Arrays.asList(path.get(path.size()/2));
    else return Arrays.asList(path.get(path.size()/2-1),path.get(path.size()/2));
}
```

时间复杂度为 O（n），空间复杂度为大于 O（n），小于 O（n^2）。

322. Coin Change

You are given coins of different denominations and a total amount of money *amount*. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return `-1`.

Example 1:

coins = `[1, 2, 5]`, amount = `11`

return `3` (11 = 5 + 5 + 1)

Example 2:

coins = `[2]`, amount = `3`

return `-1`.

Note:
You may assume that you have an infinite number of each kind of coin.

解题思路：

这道题是完全背包问题（Complete Knapsack Problem）的变体。第 i 个硬币可以不拿，也可以拿多个，且隐含着最多可以拿 amount/coins[i]个。有 N 种硬币，一种一种地拿，则状态转移方程为：

F(i,amount)=min{F(i-1,amount),F(i-1,amount-k*coins[i])+k}

因为每种硬币最多可以拿 amount/coins[i]个，因此这种解法时间复杂度为$O(N*\sum_{i=0}^{N} amount/coins[i])$。

改进一下可以得到时间复杂度为 O（N＊amount）的解法，改进后的状态转移方程为：

F(i,amount)=min{F(i-1,amount),F(i,amount-coins[i])+1}

F(i,amount-coins[i])+1 的含义是，第 i 种硬币已经拿过了，现在要再拿一个。

代码如下：

```java
public class Solution {
    public int coinChange(int[] coins, int amount) {
        if(coins==null||coins.length<1||amount<1){
            return 0;
        }
        int[] nums = new int[amount+1];
        nums[0] = 0;
        for(int i=1;i<=amount;i++){
            nums[i] = Integer.MAX_VALUE-1;
        }
        int sum = -1;
        for(int i=0;i<coins.length;i++){
            for(int j=coins[i];j<=amount;j++){
                if(nums[j-coins[i]]+1<nums[j]){
                    nums[j] = nums[j-coins[i]] + 1;
                    if(j==amount) sum = j;
                }
            }
        }
        if(sum==amount) return nums[amount];
        else return -1;
    }
}
```

扩展：

Best Time to Buy and Sell Stock I

题意：用一个数组表示股票每天的价格，数组的第 i 个数表示股票在第 i 天的价格。 如果只允许进行一次交易，也就是说只允许买一支股票并卖掉，求最大的收益。

分析：动态规划法。从前向后遍历数组，记录当前出现过的最低价格，作为买入价格，并计算以当天价格出售的收益，作为可能的最大收益，整个遍历过程中，出现过的最大收益就是所求。

```java
public class Solution {
    public int maxProfit(int[] prices) {
        if (prices.length < 2) return 0;

        int maxProfit = 0;
        int curMin = prices[0];

        for (int i = 1; i < prices.length; i++) {
            curMin = Math.min(curMin, prices[i]);
            maxProfit = Math.max(maxProfit, prices[i] - curMin);
        }

        return maxProfit;
    }
}
```

代码：时间 O(n)，空间 O(1)。

Best Time to Buy and Sell Stock II

题目：用一个数组表示股票每天的价格，数组的第 i 个数表示股票在第 i 天的价格。交易次数不限，但一次只能交易一支股票，也就是说手上最多只能持有一支股票，求最大收益。

分析：贪心法。从前向后遍历数组，只要当天的价格高于前一天的价格，就算入收益。

```java
public class Solution {
    public int maxProfit(int[] prices) {
        if (prices.length < 2) return 0;

        int maxProfit = 0;
        for (int i = 1; i < prices.length; i++) {
            int diff = prices[i] - prices[i - 1];
            if (diff > 0) {
                maxProfit += diff;
            }
        }

        return maxProfit;
    }
}
```

代码：时间 O(n)，空间 O(1)。

Best Time to Buy and Sell Stock III

题意：用一个数组表示股票每天的价格，数组的第 i 个数表示股票在第 i 天的价格。最多交易两次，手上最多只能持有一支股票，求最大收益。

分析：动态规划法。以第 i 天为分界线，计算第 i 天之前进行一次交易的最大收益 preProfit[i]，和第 i 天之后进行一次交易的最大收益 postProfit[i]，最后遍历一遍，max{preProfit[i] + postProfit[i]} (0≤i≤n-1)就是最大收益。第 i 天之前和第 i 天之后进行一次的最大收益求法同 Best Time to Buy and Sell Stock I。

```java
public class Solution {
    public int maxProfit(int[] prices) {
        if (prices.length < 2) return 0;

        int n = prices.length;
        int[] preProfit = new int[n];
        int[] postProfit = new int[n];

        int curMin = prices[0];
        for (int i = 1; i < n; i++) {
            curMin = Math.min(curMin, prices[i]);
            preProfit[i] = Math.max(preProfit[i - 1], prices[i] - curMin);
        }

        int curMax = prices[n - 1];
        for (int i = n - 2; i >= 0; i--) {
            curMax = Math.max(curMax, prices[i]);
            postProfit[i] = Math.max(postProfit[i + 1], curMax - prices[i]);
        }

        int maxProfit = 0;
        for (int i = 0; i < n; i++) {
            maxProfit = Math.max(maxProfit, preProfit[i] + postProfit[i]);
        }

        return  maxProfit;
    }
}
```

代码：时间 O(n)，空间 O(n)。

Best Time to Buy and Sell Stock IV

题意：用一个数组表示股票每天的价格，数组的第 i 个数表示股票在第 i 天的价格。最多交易 k 次，手上最多只能持有一支股票，求最大收益。

分析：特殊动态规划法。传统的动态规划我们会这样想，到第 i 天时进行 j 次交易的最大收益，要么等于到第 i-1 天时进行 j 次交易的最大收益（第 i 天价格低于第 i-1 天的价格），要么等于到第 i-1 天时进行 j-1 次交易，然后第 i 天进行一次交易（第 i 天价格高于第 i-1 天价格时）。于是得到动规方程如下（其中 diff = prices[i] – prices[i – 1]）：

profit[i][j] = max(profit[i – 1][j], profit[i – 1][j – 1] + diff)
看起来很有道理，但其实不对，为什么不对呢？因为 diff 是第 i 天和第 i-1 天的差额收益，如果第 i-1 天当天本身也有交易呢，那么这两次交易就可以合为一次交易，这样 profit[i – 1][j – 1] + diff 实际上只进行了 j-1 次交易，而不是最多可以的 j 次，这样得到的最大收益就小了。

那么怎样计算第 i 天进行交易的情况的最大收益，才会避免少计算一次交易呢？我们用一个局部最优解和全局最有解表示到第 i 天进行 j 次的收益，这就是该动态规划的特殊之处。

用 local[i][j]表示到达第 i 天时，最多进行 j 次交易的局部最优解；用 global[i][j]表示到达第 i 天时，最多进行 j 次的全局最优解。它们二者的关系如下（其中 diff = prices[i] – prices[i – 1]）：

local[i][j] = max(global[i – 1][j – 1] + max(diff, 0), local[i – 1][j] + diff)

global[i][j] = max(global[i – 1][j], local[i][j])

其中的 local[i – 1][j] + diff 就是为了避免第 i 天交易和第 i-1 天交易合并成一次交易而少一次交易收益。 参考：http://www.cnblogs.com/grandyang/p/4295761.html

```java
public class Solution {
    public int maxProfit(int k, int[] prices) {
        if (prices.length < 2) return 0;
        if (k >= prices.length) return maxProfit2(prices);

        int[] local = new int[k + 1];
        int[] global = new int[k + 1];

        for (int i = 1; i < prices.length ; i++) {
            int diff = prices[i] - prices[i - 1];

            for (int j = k; j > 0; j--) {
                local[j] = Math.max(global[j - 1], local[j] + diff);
                global[j] = Math.max(global[j], local[j]);
            }
        }

        return global[k];
    }


    public int maxProfit2(int[] prices) {
        int maxProfit = 0;

        for (int i = 1; i < prices.length; i++) {
            if (prices[i] > prices[i - 1]) {
                maxProfit += prices[i] - prices[i - 1];
            }
        }

        return maxProfit;
    }
}
```

代码：时间 O(nk)，空间 O(k)。