算法设计之贪心算法

(1)122. 买卖股票的最佳时机 II

Description

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。如果允许你多次买卖一支股票，设计一个算法来计算你所能获取的最大利润。

Solution

参考121. 买卖股票的最佳时机

思路是：在价格的波谷买入，波峰卖出。相当于所有递增差值(就是相邻两点差大于0)都是利润

时间复杂度： O(n)

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

class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int result = 0;
        for (int i=1; i<prices.size(); i++) {
            result += max(0, prices[i]-prices[i-1]); //差值大于0就加
        }
        return result;
    }
};
```

```
int main(){
    Solution solu;
    vector<int> nums = {7,1,5,3,6,4};
    cout<<solu.maxProfit(nums);
    return 0;
}
```

---

(2)860. 柠檬水找零

Description

每一杯柠檬水的售价为 5元。每位顾客只买一杯，然后向你付 5元、或10元、或 20元。你必须给每个顾客正确找零。注意，一开始你手头没有任何零钱。

如果你能给每位顾客正确找零，返回 true ，否则返回 false 。

Solution

典型的贪心算法，先从大钱找起

时间复杂度： O(n)

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

class Solution {
public:
    bool lemonadeChange(vector<int>& bills) {
        int cnt10 = 0, cnt5 = 0; //记录5元、10元张数。支付最大20元，所以20不能用于找零
        for(int i=0; i<bills.size(); i++) {
            if(bills[i] == 20){ //找零时从10元找起
                if(cnt10 && cnt5) {cnt10--;cnt5--;} //找一张10、一张5
                else if(cnt5>=3)  cnt5-=3; //找3张5
                else  return false;
            }
            else if(bills[i] == 10){
                cnt10 ++;
                if(cnt5) cnt5--;
                else return false;
            }
            else if(bills[i] == 5){
```

```
                cnt5 ++;
            }
        }
        return true;
    }
};

int main(){
    Solution solu;
    vector<int> nums = {5,10,20};
    cout << boolalpha << solu.lemonadeChange(nums)
    return 0;
}
```

---

(3)455. 分发饼干

Description

每个孩子最多只能给一块饼干。对每个孩子 i ，都有一个胃口值 gi ，这是能让孩子们满足胃口的饼干的最小尺寸；每块饼干 j ，都有一个尺寸 sj 。如果 sj >= gi ，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。

输出满足越多数量的孩子这个最大数值。

Solution

胃口的饼干先升序排序
从最小饼干分起，分给胃口最小的朋友
以此递进
分不出去的饼干舍弃

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

class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        sort(g.begin(), g.end()); //升序排序
        sort(s.begin(), s.end());

        int count = 0;
        for(int i=0,j=0; i<s.size() && j<g.size(); ) {
```

```
            if(s[i]>=g[j]){ //当前饼干能满足当前孩子
                i++;
                j++;
                count++;
            }
            else //当前饼干不能满足当前孩子 ， 拿下一块
                i++;
        }
        return count;
    }
};

int main() {
    Solution solu;
    vector<int> g = {1,2};
    vector<int> s = {1,2,3};
    cout<<solu.findContentChildren(g, s);
    return 0;
}
```

---

(4)874. 模拟行走机器人

Description

机器人在一个无限大小的网格上行走，从点 (0, 0) 处开始出发，面向北方。该机器人可以接收以下三种类型的命令：

-2：向左转 90 度

-1：向右转 90 度

1 <= x <= 9：向前移动 x 个单位长度

在网格上有一些格子被视为障碍物。如果机器人试图走到障碍物上方，那么它将停留在障碍物的前一个网格方块上，但仍然可以继续该路线的其余部分。

求从原点到机器人的最大欧式距离的平方。

Input and Output

> 输入: commands = [4,-1,4,-2,4], obstacles = [[2,4]]
> 输出: 65
> 解释: 机器人在左转走到 (1, 8) 之前将被困在 (1, 4) 处

Solution

本题就像马踏棋盘，没有什么快捷方法，顶多在转向和查找上做改进

一、首要解决的是机器人转向，用0~3标记东南西北。向右就+1，向左就-1(因为对4取余故+3就是-1)。这个算是本题难点，方法很巧妙，要学习。

二、其次是判断障碍物，使用set类型，方便查找。每走一步在前进过程中都要判断

三、最后走到最后不一定就是最远的，有可能往回走，所以没走一步都要计算距离

```
#include <set>
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

class Solution {
public:
    int robotSim(vector<int>& commands, vector<vector<int>>& obstacles) {
        int x = 0, y = 0; //机器人位置
        int dir = 0; //机器人朝向，0朝北，1朝东，2朝南，3朝西
        int result = 0; //机器人与原点最大欧式平方
        int g[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}}; //四个方向位置计算
方式

        set<pair<int, int> > obstacles_set;//vector转set，方便查找
        for(auto e : obstacles){
            obstacles_set.insert({e[0], e[1]});
        }

        for(auto c : commands){
            switch(c){
                case -2: //向左转
                        dir = (dir + 3) % 4;
                    break;

                case -1: //向右转
                        dir = (dir + 1) % 4;
                    break;

                default: //前进 c 步
                        while(c--){
                            int nx = x + g[dir][0];
                            int ny = y + g[dir][1];
                            if (obstacles_set.find({nx, ny}) != obstacles
_set.end()) break;
                            x = nx;
                            y = ny;
                        }
            }
            result = max(result, x * x + y * y);
        }
        return result;
    }
};
```

```
int main(){
    Solution solu;
    vector<int> commands = {4,-1,3};
    vector<vector<int>> obstacles = {};
    cout<<solu.robotSim( commands, obstacles );
    return 0;
}
```

---

(5)861. 翻转矩阵后的得分

Description

有一个二维矩阵 A，其中每个元素的值为 0 或 1 。

翻转是指选择任一行或列，该行或列中的每一个值 0 变 1，1 变 0

在做出任意次数的移动后，将该矩阵的每一行都按照二进制数来解释，矩阵的得分就是这些数字的总和。

求返回尽可能高的分数。

Solution

贪心思想

- 每行的第一位必须是1 –> 行变换确定了，接下来是列变换
- 再考虑每一列，一旦该列0的个数多于1的个数，就翻转该列

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

class Solution {
private:
    void rot_row(int row, vector<vector<int> > &A){ //翻转行
        for(int col=0; col<A[0].size(); col++){
            A[row][col] ^= 1;
        }
    }

    void rot_col(int col, vector<vector<int> > &A){ //翻转列
        for(int row=0; row<A.size(); row++){
            A[row][col] ^= 1;
```

```
        }
    }

public:
    int matrixScore(vector<vector<int> >& A) {
        if(A.empty() || A[0].empty()) return 0;

        int row_number = A.size();
        int col_number = A[0].size();
        for(int i=0; i<row_number; i++){
            if(A[i][0] == 0) rot_row(i, A); //每行的第一位必须是1
        }

        for(int j=1; j<col_number; j++){
            int cnt_one = 0;
            for(int i=0; i<row_number; i++)
                cnt_one += A[i][j];
            if(cnt_one*2 < A.size()) rot_col(j, A); //一旦该列0的个数多于1的
个数，就翻转该列
        }

        int result = 0;
        for(int i=0; i<row_number; i++){ //计算分数
            int line_score = 0;
            for(int j=0; j<col_number; j++){
                line_score += A[i][j]<<(col_number-j-1);
            }
            result += line_score;
        }
        return result;
    }
};

int main(){
    Solution Solu;
    vector<vector<int> > matrix = { {0,0,1,1},{1,0,1,0},{1,1,0,0} };
    cout <<  Solu.matrixScore(matrix);
    return 0;
}
```

(6)392.判断子序列

Description

给定字符串 s 和 t ，判断 s 是否为 t 的子序列。s 和 t 中仅包含英文小写字母。
字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置

形成的新字符串。（例如，”ace”是”abcde”的一个子序列，而”aec”不是）。

Solution

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

class Solution {
public:
    bool isSubsequence(string s, string t) {
        if(s.length()==0) return true;

        int si = 0;
        for(auto c : t){
            if(si<s.size() && c==s[si]){
                si++;
            }
        }
        return si>=s.size();
    }
};

int main(){
    Solution Solu;
    string s = "leeeeetcode";
    string t = "yyyyylyyyyyyyyyyyyyyyyyy";
    cout << boolalpha << Solu.isSubsequence(s,t);
    return 0;
}
```

---

(7)134. 加油站

Description

在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 gas[i] 升。从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

Solution

- 如果Σgas_cost<0，则不存在那个点能环绕一周，返回-1
- 如果Σgas_cost>=0，是否一定存在某个点能环绕一周？？

  我们画出汽车油量变化的折线图(先假设汽车从0号加油站出发)，通过折线图可知这个点就是车油箱油量最低点。当我 们把汽车出发点选为折线中车油箱油量最低点时，相当于把横坐标下移到过改点，这时再看折线图就会发现车油箱油量始终保持大于等于0(在横坐标上方)，也就是说满足绕环路行驶一周。

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

class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int sum = 0; //记录汽车油量变化
        int minqua = 0; //记录汽车油量最低值
        int result = 0;
        for(int i=0; i<gas.size(); i++) {
            if( sum < minqua ){
                minqua = sum;
                result = i;
            }
            sum += gas[i]-cost[i];
        }

        return sum<0 ? -1:result;
    }
};

int main(){
    Solution solu;
    vector<int> gas = {1,2,3,4,5};
    vector<int> cost = {3,4,5,1,2};
    cout << solu.canCompleteCircuit(gas,cost);
    return 0;
}
```

---

(8)452.用最少数量的箭引爆气球

Description

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。

一支弓箭可以沿着x轴从不同点完全垂直地射出。
找到使得所有气球全部被引爆，所需的弓箭的最小数量。

Solution

讲下思路：
因为气球数量不为0，所以怎么也得先来一发啊，然后这一箭能覆盖的最远位置就是第一个气球的结束点，用变量end来表示。
然后我们开始遍历剩下的气球:

- 如果当前气球的起始点<=end，说明有重合，之前那一箭可覆盖到当前的气球，end更新为两个气球结束点之间较小的那个
- 如果当前气球的起始点>end，说明前面的箭无法覆盖到当前的气球，那么就得再来一发result++，end更新为当前气球的结束点了

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

class Solution {
public:
    int findMinArrowShots(vector<pair<int, int>>& points) {
        if (points.empty()) return 0;

        sort(points.begin(), points.end()); //按第一个数字升序排列，如果第一个数字相同，那么按第二个数字升序排列

        int result = 1, end = points[0].second;
        for (int i = 1; i < points.size(); ++i) {
            if(points[i].first <= end)
                end = min(end, points[i].second);
            else{
                result++;
                end = points[i].second;
            }
        }
        return result;
    }
};

int main(){
    Solution Solu;
    vector<pair<int, int> > points =  { {10,16}, {2,8}, {1,6}, {7,12} };
    cout <<  Solu.findMinArrowShots(points);
    return 0;
}
```

(9)435. 无重叠区间

Description

给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

注意:可以认为区间的终点总是大于它的起点。区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。

Solution

> - 普及下区间重叠情况（边界相互接触不视为重叠）
> - 重叠的情况有4种： 两种相交，两种包含
> - 不重叠的情况有2种：A在B前面，A在B后面

> - 判断区间重叠的方法：
> - 正向判断，列出四种重叠的情况， 满足其一，则重叠；
> 表面看比较复杂，但可优化正向判断：只要满足 max(A.start,B.start)
> <min(A.end,B,end)，则A，B重叠。
> - 逆向判断，列出两种不重叠的情况，满足其一，则不重叠；
> 逆向判断：只要满足其一 (A.end<=B.start || A.start>=B.end) ， 则A,B不重叠

本题求的是除区间的数量，而不是求移除那个，数出需要移除的区间数量较难
我们反向思考。需移除区间的最小数量 = 区间总数 - 重叠于某一块按一块算的区间数
所以我们需计算 重叠按一块算的区间数
那么本题就和题目 452.用最少数量的箭引爆气球 很像了
注：我们会把重叠于某一块的一些区间的看成一个大区间，下面所说的就是指这种大区间

因为 大区间 最小为1，用变量end来记 和第一块大区间的右边结束点。
然后我们开始遍历剩下的大区间:

如果当前大区间的起始点<end，说明有重叠，end更新为两个大区间结束点之间较小的那个

如果当前大区间的起始点>end，说明无重叠，那么result++，end更新为当前大区间的结束点了

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

struct Interval {
    int start;
```

```
    int end;
    Interval() : start(0), end(0) {}
    Interval(int s, int e) : start(s), end(e) {}
};

class Solution {
public:
    int eraseOverlapIntervals(vector<Interval>& intervals) {
        if(intervals.empty()) return 0;

        // 先排序
        vector<pair<int, int> > points;
        for(auto e : intervals) points.push_back({e.start, e.end});
        sort(points.begin(), points.end());

        int result = 1, end = points[0].second;
        for (int i = 1; i < points.size(); ++i) {
            if(points[i].first < end) //有重叠
                end = min(end, points[i].second);
            else{ //没有重叠
                result++;
                end = points[i].second;
            }
        }
        return intervals.size() - result;
    }
};

int main(){
    Solution Solu;
    vector<Interval> intervals =  { {1,4}, {2,3}, {3,5} };
    cout <<  Solu.eraseOverlapIntervals(intervals);
    return 0;
}
```