算法Project3 318104584 毛雨帆

Part1：Leetcode 题目（Hard）

分发糖果

老师想给孩子们分发糖果，有 N 个孩子站成了一条直线，老师会根据每个孩子的表现，预先给他们评分。

你需要按照以下要求，帮助老师给这些孩子分发糖果：

每个孩子至少分配到 1 个糖果。
相邻的孩子中，评分高的孩子必须获得更多的糖果。
那么这样下来，老师至少需要准备多少颗糖果呢？

示例 1:

输入: [1,0,2]
输出: 5
解释: 你可以分别给这三个孩子分发 2、1、2 颗糖果。
示例 2:

输入: [1,2,2]
输出: 4
解释: 你可以分别给这三个孩子分发 1、2、1 颗糖果。
第三个孩子只得到 1 颗糖果，这已满足上述两个条件。

Part2：算法设计与代码实现

方法一：直接法暴力求解

思路：假设流程走完，状态不变，代表是最终的结果
我们每次遍历的规则是：
假设第i位大于其左边，但他的糖果不比左边多，就将他的糖果数定为左边的数量+1
假设第i位大于其右边，但他的糖果不比右边多，就将他的糖果数定为右边的数量+1
假设状态不发生变化，此时所得到就是最终的稳定的结果了

算法流程：

使用一个一维的数组 candies去记录给学生的糖果数。首先我们给每个学生 1 个糖果。然后我们开始从左到右扫描数组。对每一个学生，如果当前的评分 ratings[i]比前一名学生的评分 ratings[i - 1]高，且 candies[i]<=candies[i - 1]，那么我们更新 candies[i] = candies[i-1] + 1。这样，这两名学生之间的糖果分配目前是正确的。同样的，我们检查当前学生的评分 ratings[i]是否比 ratings[i+1]高，如果成立，我们同样更新 candies[i]=candies[i+1] + 1。我们继续对ratings 数组重复此步骤。如果在某次遍历中，candies数组不再变化，意味着我们已经得到了最后的糖果分布，此时可以停止遍历。为了记录是否到达最终状态，我们用flag 记录每次遍历是否有糖果数目变化，如果有，则为True ，否则为False 。最终，我们可以把candies 数组中所有糖果数目加起来，得到要求数目最少的糖果数。

```
public class Solution {
    public int candy(int[] ratings) {
        int[] candies = new int[ratings.length];
        Arrays.fill(candies, 1);
        boolean flag = true;
        int sum = 0;
        while (flag) {
```

```
            flag = false;
            for (int i = 0; i < ratings.length; i++) {
                if (i != ratings.length - 1 && ratings[i] > ratings[i + 1] &&
candies[i] <= candies[i + 1]) {
                    candies[i] = candies[i + 1] + 1;
                    flag = true;
                }
                if (i > 0 && ratings[i] > ratings[i - 1] && candies[i] <=
candies[i - 1]) {
                    candies[i] = candies[i - 1] + 1;
                    flag = true;
                }
            }
        }
        for (int candy : candies) {
            sum += candy;
        }
        return sum;
    }
}
```

方法二：两次遍历法（合并规则）

思路：

根据直接法，总结归纳出两条规则，即合并规则求解。

规则定义： 设学生A 和学生B左右相邻在左边；

左规则： 当 ratings_B>ratings_Aratings B>ratings A时，B的糖比A的糖数量多。

右规则： 当 ratings_A>ratings_Bratings A>ratings B时，A 的糖比 B 的糖数量多。

相邻的学生中，评分高的学生必须获得更多的糖果等价于所有学生满足左规则且满足右规则。

算法流程：

先从左至右遍历学生成绩 ratings，按照以下规则给糖，并记录在 left 中：

先给所有学生 11 颗糖；

若 ratings_i>ratings*{i-1}ratings >ratings i−1，则第 i名学生糖比第i−1 名学生多 11 个。若 ratings_i*<=*ratings*{i-1}ratings i <=ratings i−1，则第 ii 名学生糖数量不变。（交由从右向左遍历时处理。）

经过此规则分配后，可以保证所有学生糖数量 满足左规则 。

同理，在此规则下从右至左遍历学生成绩并记录在 right 中，可以保证所有学生糖数量 满足右规则 。

最终，取以上 22 轮遍历 left 和 right 对应学生糖果数的 最大值 ，这样则 同时满足左规则和右规则 ，即得到每个同学的最少糖果数量。

复杂度分析：

时间复杂度 O(N)： 遍历两遍数组即可得到结果；

空间复杂度 O(N) ： 需要借用left，right的线性额外空间。

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        int len = ratings.size();
        if(len==0) return 0;
        vector<int> candy(len,1);
        int sum = 0;
        //从左往右：如果比前一个人分高，那么糖果比前一个人+1
        for(int i=1;i<len;++i)
            if(ratings[i]>ratings[i-1]) candy[i] = candy[i-1] + 1;
        //从右往左：如果比后一个人分高，但是糖果没他多，那么糖果比后一个人+1
        for(int i=len-2;i>=0;--i)
            if(ratings[i]>ratings[i+1] && candy[i]<=candy[i+1]) candy[i] =
candy[i+1] + 1;
        //计算结果
        for(int it:candy)
            sum += it;
        return sum;
    }
};
```

方法三：常数空间一次遍历

这个方法通过观察（如下面的图所展示）发现，为了获得最少总数的糖果，糖果的分配每次都是增加 1 的。进一步的，在分配糖果时，给一个学生的最少数目是 1 。所以，局部的分配形式一定是1, 2, 3, ..., n 或者n, ..., 2, 1，总和是n(n+1)/2.

现在我们可以把评分数组ratings 当做一些上升和下降的坡。每当坡是上升的，糖果的分配一定1, 2, 3, ..., m 这样的。同样的，如果是一个下降的坡，一定是k, ..., 2, 1 的形式。一个随之而来的情况是，每个峰都只会在这些坡中的一个出现。那么我们应该把这个峰放在上升的坡中还是下降的坡中呢?

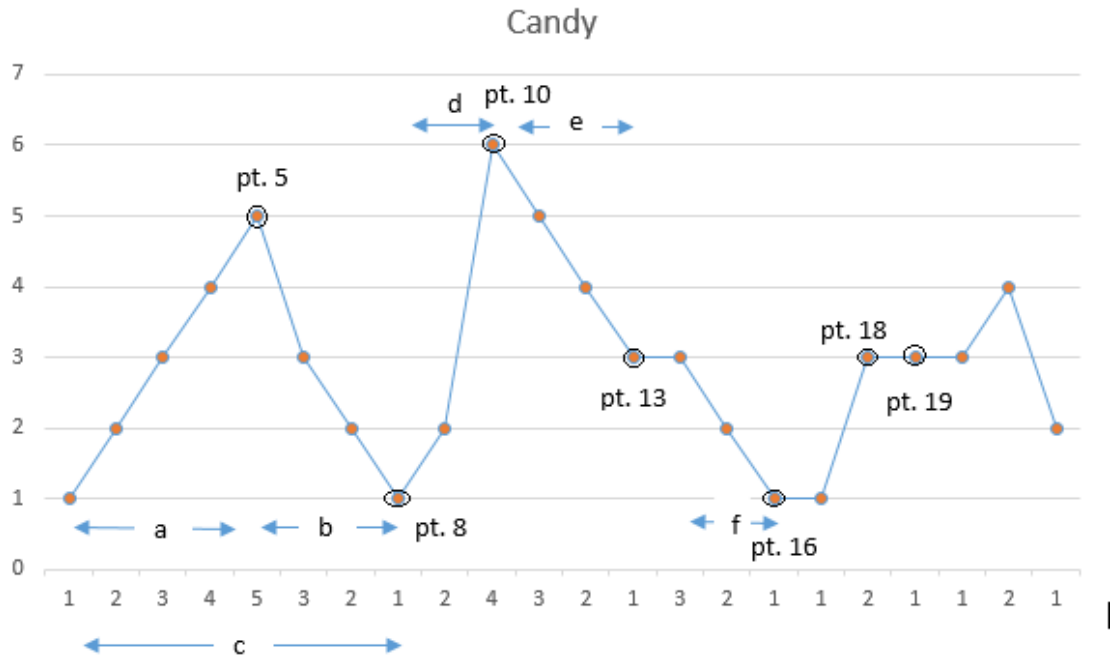
为了解决这个问题，我们观察到为了同时满足左右邻居的约束，峰值一定是上升的坡和下降的坡中所有点的最大值，所以为了决定需要的糖果数，峰点需要算在上升坡和下降坡较多点的那一边。局部谷点也只能被包括在一个坡中，但是这种情况很容易解决，因为局部谷点总是只会被分配 1 个糖果（可以在下一个坡开始计数时减去）。

接下来考虑实现，我们维护两个变量 old_slope和new_slope 来决定现在是在峰还是在谷，同时我们用 up 和 down 两个变量分别记录上升或者下降坡中的学生个数（不包括峰点）。我们总是在一个下降的坡接着上升坡（或者上升坡接一个下降坡）的时候更新 candies 的总数。

在一个山的结束处，我们决定将峰点算在上升坡还是下降坡中，决定的依据是比较up 和 down 两个变量。因此，峰值的数目应该为 max}(up, down) + 1。此时，我们将up 和down 变量重新初始化，表示一个新的山的开始。

下面的图展示了如下样例的结果。

rankings: [1 2 3 4 5 3 2 1 2 6 5 4 3 3 2 1 1 3 3 3 4 2]



从这个图中，我们可以看到糖果在局部的分配中一定是 \text{1, 2, ..., n}1, 2, ..., n 或者 \text{n, ..., 2, 1}n, ..., 2, 1 这样的形式。对于由 aa 和 bb 组成的第一座山，在分配峰点（pt. 5pt.5）糖果的时候，它应该被分配到 aa 中满足左邻居约束，bb 中的局部谷点（pt. 8pt.8）标志着第一座山（cc）的结束。在计算的时候，我们可以把这个点归属为当前的山，也可以归属到接下来的山中。点 pt.13pt.13 标记的是第二座山的结束，因为 pt.13pt.13 和 pt.14pt.14 两个学生的评分是相同的。因此，区域 ee 比区域 dd 有更多的点，局部峰点 (pt. 10pt.10) 应该被划分到 ee 区域满足右邻居的约束。现在第三座山 ff 应该被考虑为一座只有下降坡没有上升坡的山 (up=0up=0) 。类似的，因为与旁边的学生评分相同， pt.16, 18, 19pt.16,18,19 也是山的结束。

```
class Solution {
public:
    int count(int n) {
        return n*(n+1)/2;
    }

    int candy(vector<int>& ratings) {
        int n = ratings.size();
        int sum = 0;
        int up = 0, down = 0;
        int os = 0, ns = 0;
        for (int i = 1; i < n; ++i) {
            ns = ratings[i]>ratings[i-1] ? 1 : (ratings[i]<ratings[i-1] ? -1 :
0);
            // 这座山峰遍历结束，计算糖果数。
            if ((os < 0 && ns >= 0) || os > 0 && ns == 0) {
                // 这里看似好像峰顶没有加 1，其实是 count(down) 减去了 1。
                // 因为谷底是共享的，所以将谷底给了下一座山峰的上坡。
                sum += count(up) + count(down) + max(up, down);
                up = down = 0;
            }
            if (ns > 0) up++;
            else if (ns < 0) down++;
            // 如果是平原，说明谷底不会共享，之前少加的 1 再补上。
            else if (!ns) sum++;
            os = ns;
```

```
        }
        // 最后一座山峰循环里不会计算到，再加上。
        sum += count(up) + count(down) + max(up, down) + 1;
        return sum;
    }
};
```

方法四：拓扑排序法

思路：

根据给定的数组生成一个有向无序图，然后转化为一个拓扑排序问题,再根据拓扑排序的两种方法，即DFS（深度优先搜索）和减治法进行实现。把数组中的每一个位置看成一个节点，每个节点有一条指向相邻位置且比他大的数的边，就构成了一个有向无环图。采用拓扑排序从度数为0的节点开始遍历，遍历到某个节点时，就可知道当前节点需要获得的最小糖果数。

（1）减治法

```
class Solution {
    public int candy(int[] ratings) {
        Queue<Integer> queue = new LinkedList<Integer>();//维护一个队列queue，运用了模板库中得LinkList
        int[] degree = new int[ratings.length];
        //生成有向无环图，degree数组存储每个结点的入度情况
        for(int i = 0 ; i < ratings.length; i++){
            if(i - 1 >= 0 && ratings[i - 1] < ratings[i]){
                degree[i]++;//左边的人rating比i小，i结点入度加一
            }
            if(i + 1 < ratings.length && ratings[i + 1] < ratings[i]){
                degree[i]++;//右边的人rating比i小，i结点入度加一
            }
            if(degree[i] == 0){
                queue.offer(i);
            }
        }
        //减治法进行拓扑排序，排序过程中根据遍历顺序给count数组赋值，count存储每个人应给的最少糖果数
        int[] count = new int[ratings.length];
        int cur = 1;
        //queue队列存储入度为0的元素，即源，没有结点指向它
        while(!queue.isEmpty()){
            int len = queue.size();
            while(len-- > 0){
                int idx = queue.poll();//从队列中取出一个入度为0的结点
                count[idx] = cur;
                if(idx - 1 >= 0 && ratings[idx - 1] > ratings[idx]){
                    degree[idx - 1]--; //左边的人rating比它大，结点入度减一
                    if(degree[idx - 1] == 0){
                        queue.offer(idx - 1);
                    }
                }
                if(idx + 1 < ratings.length && ratings[idx + 1] > ratings[idx]){
                    degree[idx + 1]--;//右边的人rating比它大，结点入度减一
                    if(degree[idx + 1] == 0){
                        queue.offer(idx + 1);
                    }
```

```
                }
            }
            cur++;
        }
        //累加count，得最少糖果总数
        int total = 0;
        for(int i = 0; i < count.length; i++){
            total += count[i];
        }
        return total;
    }
}
```

（2）DFS深度优先搜索

与上面相似，首先构建有向无环图，然后按照DFS遍历。

具体过程如下：

初始化时，将每个结点的candy数初始化为1，每个结点的father指向自己。假设当前结点为i，将i的rating值和两个邻居进行比较，如果比邻居大，那么就添加一条邻居到i的边，同时将当前结点的father指向该邻居。这里要分两种情况，边界和中间，边界的i只需和一个邻居进行比较，中间的i需要和左右两个邻居进行比较。因为不知道图的起点在哪，为了避免重复搜索导致超时，此时father数组就起作用了。当father[i]=i且存在邻边时，就说明该结点是图的起点，可以进行DFS搜索了。DFS搜索参数有两个index表示当前结点，cur表示当前结点的candy数，每进入下一层，cur+1。最后将nums数组的所有值相加得到答案。

复杂度分析：

时间复杂度：三次遍历搜索，第一次构建图，第二次DFS搜索，第三次求总candy数，每次遍历都是O(n)的时间复杂度，因此总时间复杂度为O(n)。空间复杂度:O(n)

```
class Solution {
    vector<vector<int>> graph;
    vector<int> nums;
    void DFS(int index,int cur){
        //某一结点index比左右两个邻居都大时，选取大的那个邻居的candy数+1作为自己的candy数
        nums[index]=max(cur,nums[index]);

        for(int i=0;i<graph[index].size();i++){
            DFS(graph[index][i],cur+1);
        }
    }
public:
    int candy(vector<int>& ratings) {
        int size=ratings.size();
        if(size==0) return 0;
        if(size==1) return 1;
        graph=vector<vector<int>>(size);
        int father[size];
        //father数组很重要，能够避免重复遍历的情况出现
        for(int i=0;i<size;i++) father[i]=i;
        //构造图
        for(int i=0;i<size;i++){
            if(i==0){
                if(ratings[i]>ratings[i+1]){
                    father[i]=i+1;
                    graph[i+1].push_back(i);
```

```
                }
            }
            else if(i==size-1){
                if(ratings[i]>ratings[i-1]){
                    father[i]=i-1;
                    graph[i-1].push_back(i);
                }
            }
            else{
                if(ratings[i]>ratings[i-1]){
                    father[i]=i-1;
                    graph[i-1].push_back(i);
                }
                if(ratings[i]>ratings[i+1]){
                    father[i]=i+1;
                    graph[i+1].push_back(i);
                }
            }
        }
        nums=vector<int>(size,1);
        for(int i=0;i<size;i++){
            //结点i存在边，并且是当前的起点，才开始进行DFS，这样能避免重复搜索
            if(graph[i].size()>0&&father[i]==i){

                DFS(i,nums[i]);
            }

        }

        int ans=0;
        for(int i=0;i<size;i++){

            ans+=nums[i];
        }

        return ans;

    }
};
```

方法五：单调栈

从左到右遍历，使用单调栈，如果当前的元素大于后面的元素，说明当前的元素需要修改，但是修改到何种程度，由两个方面组成，左边+右边。左边的影响易评估；主要是右边，所以可以将其“压栈”，往后遍历，一直到递减结束；那么该元素的前一个元素应该是2，然后3， 然后4。。。到了栈顶元素，取左右影响的最大值即可。这里实际上不需要用栈记录，只需要记录开始递减的value及其index即可。

```
class Solution {
public:
    void RefreshminValueFromStartIndex(vector<int>& ans, int minValue, int
startIndex, int end)
    {
        int curV = 2;
        for (int i = end - 1; i >= startIndex; i--) {
            ans[i] = curV;
```

```
            curV++;
        }
        ans[startIndex] = max(ans[startIndex], minValue);
    }
    int GetCurrentItem(vector<int>& ratings, vector<int>& ans, int index)
    {
        if (index == 0) {
            return 1;
        }
        return ratings[index] > ratings[index-1] ? (ans[index-1] + 1) :
ans[index];
    }
    int candy(vector<int>& ratings) {
        if (ratings.size() <= 1) {
            return ratings.size();
        }

        vector<int> ans(ratings.size(), 1);
        int startIndex = -1;
        int minValue = 0;
        for (int i = 0; i < ratings.size(); i++) {
            if (i == ratings.size() - 1) {
                if (startIndex != -1) {
                    RefreshminValueFromStartIndex(ans, minValue, startIndex, i);
                } else {
                    ans[i] = GetCurrentItem(ratings, ans, i);
                }
            } else {
                if (ratings[i] > ratings[i + 1]) {
                    if (startIndex == -1) {
                        startIndex = i;
                        minValue = GetCurrentItem(ratings, ans, i);
                    }
                } else {
                    if (startIndex != -1) {
                        RefreshminValueFromStartIndex(ans, minValue, startIndex,
i);

                        startIndex = -1;
                        minValue = 0;
                    }
                    ans[i] = GetCurrentItem(ratings, ans, i);
                }
            }
        }
        int totalAns = 0;
        for (auto item : ans) {
            totalAns += item;
        }
        return totalAns;
    }
};
```

Part3:实验结果与分析

（1）测试样例（仅给出对直接法的测试，其余方法均通过验证，篇幅限制在此不表）

input: output:

input: output:

input: output:

input: output:

input: output:

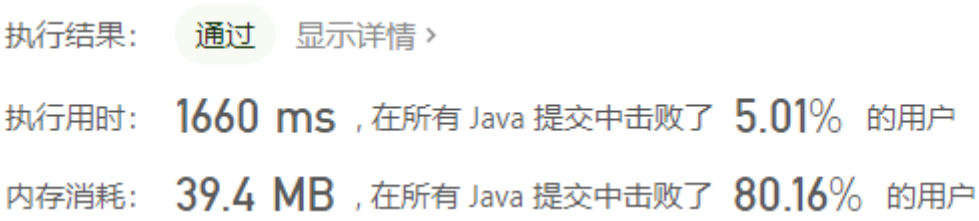
input: output:

input: output:

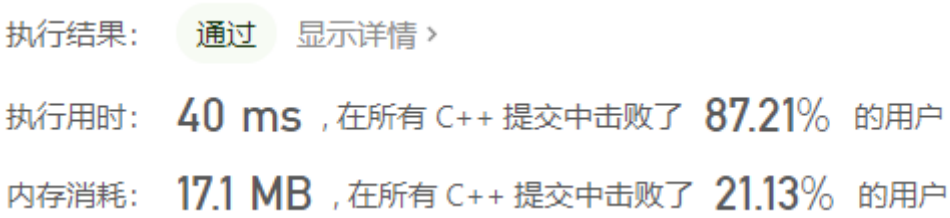
input: output:

（2）时间&空间效率分析

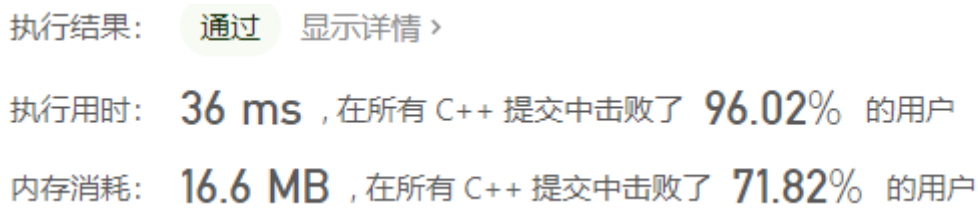
方法一：直接法暴力求解



方法二：两次遍历法（合并规则）

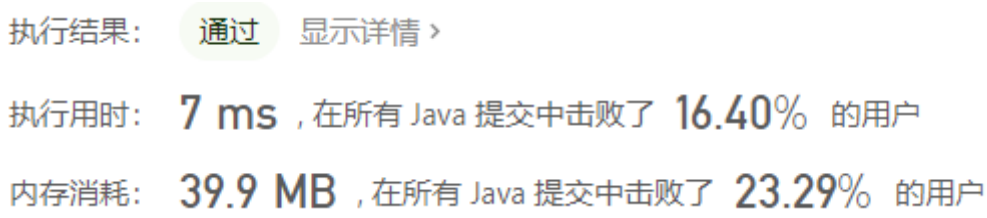


方法三：常数空间一次遍历

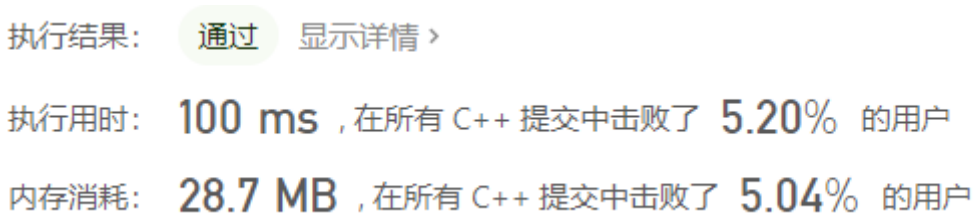


方法四：拓扑排序法

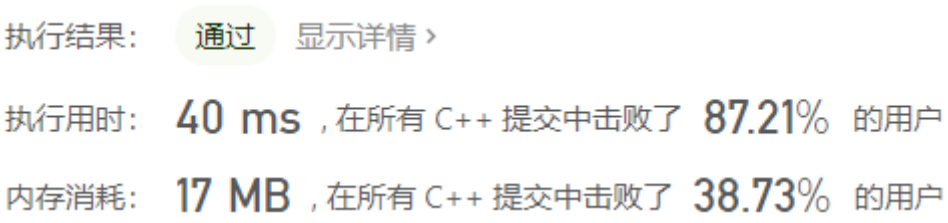
减治法：



DFS:



方法五：单调栈



小结：