分发糖果

两次遍历

将「相邻的孩子中，评分高的孩子必须获得更多的糖果」这句话拆分为两个规则，分别处理：

- 左规则：当 $ratings[i-1] < ratings[i]$ 时，i 号学生的糖果数量将比 $i-1$ 号孩子的糖果数量多。
- 右规则：当 $ratings[i] > ratings[i+1]$ 时，i 号学生的糖果数量将比 $i+1$ 号孩子的糖果数量多。

我们遍历该数组两次，处理出每一个学生分别满足左规则或右规则时，最少需要被分得的糖果数量。每个人最终分得的糖果数量即为这两个数量的最大值。

具体地，以左规则为例：我们从左到右遍历该数组，假设当前遍历到位置 i，如果有 $ratings[i-1] < ratings[i]$ 那么 i 号学生的糖果数量将比 $i-1$ 号孩子的糖果数量多，我们令 $left[i] = left[i-1] + 1$ 即可，否则我们令 $left[i] = 1$。

在实际代码中，我们先计算出左规则 $left$ 数组，在计算右规则的时候只需要用单个变量记录当前位置的右规则，同时计算答案即可。

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        int n = ratings.size();
        vector<int> left(n);
        for (int i = 0; i < n; i++) {
            if (i > 0 && ratings[i] > ratings[i - 1]) {
                left[i] = left[i - 1] + 1;
            } else {
                left[i] = 1;
            }
        }
        int right = 0, ret = 0;
        for (int i = n - 1; i >= 0; i--) {
            if (i < n - 1 && ratings[i] > ratings[i + 1]) {
                right++;
            } else {
                right = 1;
            }
            ret += max(left[i], right);
        }
        return ret;
    }
};
```

常数空间遍历

从左到右枚举每一个同学，记前一个同学分得的糖果数量为 pre：

- 如果当前同学比上一个同学评分高，说明我们就在最近的递增序列中，直接分配给该同学 $pre+1$ 个糖果即可。
- 否则我们就在一个递减序列中，我们直接分配给当前同学一个糖果，并把该同学所在的递减序列中所有的同学都再多分配一个糖果，以保证糖果数量还是满足条件：

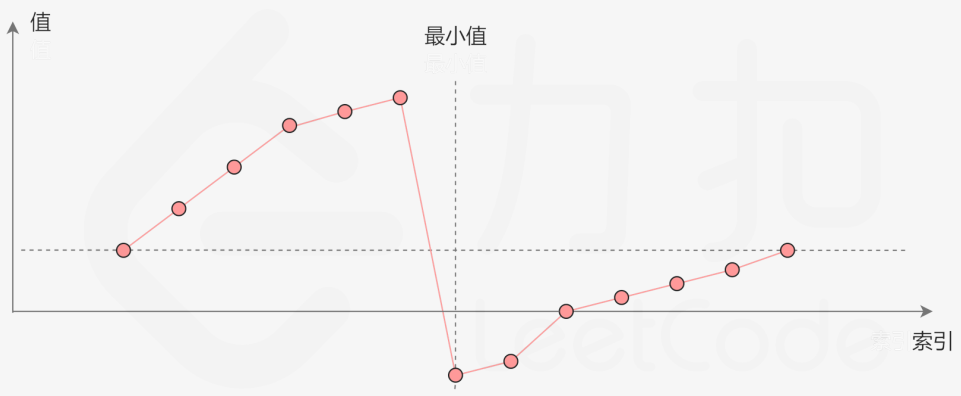
    - 我们无需显式地额外分配糖果，只需要记录当前的递减序列长度，即可知道需要额外分配的糖果数量；
    - 同时注意当当前的递减序列长度和上一个递增序列等长时，需要把最近的递增序列的最后一个同学也并进递减序列中。
- 这样，我们只要记录当前递减序列的长度 dec，最近的递增序列的长度 inc 和前一个同学分得的糖果数量 pre 即可

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        int n = ratings.size();
        int ret = 1;
        int inc = 1, dec = 0, pre = 1;
        for (int i = 1; i < n; i++) {
            if (ratings[i] >= ratings[i - 1]) {
                dec = 0;
                pre = ratings[i] == ratings[i - 1] ? 1 : pre + 1;
                ret += pre;
                inc = pre;
            } else {
                dec++;
                if (dec == inc) {
                    dec++;
                }
                ret += dec;
                pre = 1;
            }
        }
        return ret;
    }
};
```

寻找旋转排序数组中的最小值

二分查找

一个不包含重复元素的升序数组在经过旋转之后，可以得到下面可视化的折线图：



考虑数组中的最后一个元素 x：在最小值右侧的元素（不包括最后一个元素本身），它们的值一定都严格小于 x；而在最小值左侧的元素，它们的值一定都严格大于 x。因此，我们可以根据这一条性质，通过二分查找的方法找出最小值：

在二分查找的每一步中，左边界为 low，右边界为 $high$，区间的中点为 $pivot$，最小值就在该区间内。我们将中轴元素 $nums[pivot]$ 与右边界元素 $nums[high]$ 进行比较，可能会有以下的三种情况：

- 第一种情况是 $nums[pivot] < nums[high]$。这说明 $nums[pivot]$ 是最小值右侧的元素，因此我们可以忽略二分查找区间的右半部分。
- 第二种情况是 $nums[pivot] > nums[high]$。如下图所示，这说明 $nums[pivot]$ 是最小值左侧的元素，因此我们可以忽略二分查找区间的左半部分。
- 由于数组不包含重复元素，并且只要当前的区间长度不为 1，$pivot$ 就不会与 $high$ 重合；而如果当前的区间长度为 1，这说明我们已经可以结束二分查找了。因此不会存在 $nums[pivot] = nums[high]$ 的情况。

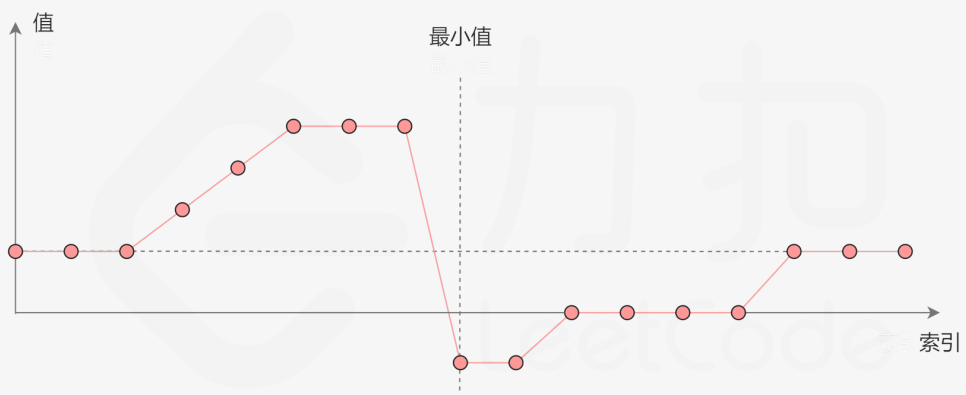
当二分查找结束时，我们就得到了最小值所在的位置：

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        int low = 0;
        int high = nums.size() - 1;
        while (low < high) {
            int pivot = low + (high - low) / 2;
            if (nums[pivot] < nums[high]) {
                high = pivot;
            }
            else {
                low = pivot + 1;
            }
        }
        return nums[low];
    }
};
```

寻找旋转排序数组中的最小值2（存在重复元素）

二分查找

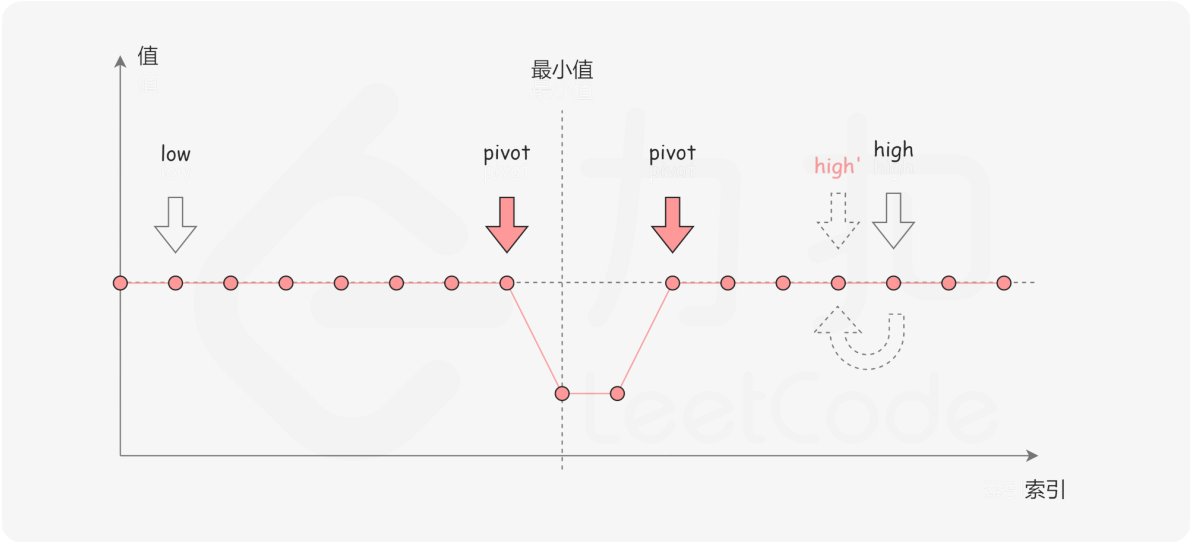
一个包含重复元素的升序数组在经过旋转之后，可以得到下面可视化的折线图：



考虑数组中的最后一个元素 x：在最小值右侧的元素（不包括最后一个元素本身），它们的值一定都小于等于 x；而在最小值左侧的元素，它们的值一定都大于等于 x。因此，我们可以根据这一条性质，通过二分查找的方法找出最小值：

在二分查找的每一步中，左边界为 low，右边界为 $high$，区间的中点为 $pivot$，最小值就在该区间内。我们将中轴元素 $nums[pivot]$ 与右边界元素 $nums[high]$ 进行比较，可能会有以下的三种情况：

- 第一种情况是 $nums[pivot] < nums[high]$。这说明 $nums[pivot]$ 是最小值右侧的元素，因此我们可以忽略二分查找区间的右半部分。
- 第二种情况是 $nums[pivot] > nums[high]$。如下图所示，这说明 $nums[pivot]$ 是最小值左侧的元素，因此我们可以忽略二分查找区间的左半部分。
- 第三种情况是 $nums[pivot] == nums[high]$。如下图所示，由于重复元素的存在，我们并不能确定 $nums[pivot]$ 究竟在最小值的左侧还是右侧，因此我们不能莽撞地忽略某一部分的元素。我们唯一可以知道的是，由于它们的值相同，所以无论 $nums[high]$ 是不是最小值，都有一个它的「替代品」$nums[pivot]$，因此我们可以忽略二分查找区间的右端点。




```
class Solution {
public:
    int findMin(vector<int>& nums) {
        int low = 0;
        int high = nums.size() - 1;
        while (low < high) {
            int pivot = low + (high - low) / 2;
            if (nums[pivot] < nums[high]) {
                high = pivot;
            }
            else if (nums[pivot] > nums[high]) {
                low = pivot + 1;
            }
            else {
                high -= 1;
            }
        }
        return nums[low];
    }
};
```

LRU 缓存机制

哈希表+双向链表

LRU 缓存机制可以通过 **哈希表辅以双向链表实现**，用一个哈希表和一个双向链表维护所有在缓存中的键值对。

- 双向链表按照被使用的顺序存储了这些键值对，靠近头部的键值对是最近使用的，而靠近尾部的键值对是最久未使用的；
- 哈希表即为普通的哈希映射，通过缓存数据的键映射到其在双向链表中的位置。

这样以来，我们首先使用哈希表进行定位，找出缓存项在双向链表中的位置，随后将其移动到双向链表的头部，即可在 $O(1)$ 的时间内完成 `get` 或者 `put` 操作。

在双向链表的实现中，使用一个 **伪头部（dummy head）** 和 **伪尾部（dummy tail）** 标记界限，这样在添加节点和删除节点的时候就不需要检查相邻的节点是否存在。

```
struct DLinkedNode {
    int key, value;
    DLinkedNode* prev;
    DLinkedNode* next;
    DLinkedNode(): key(0), value(0), prev(nullptr), next(nullptr) {}
    DLinkedNode(int _key, int _value): key(_key), value(_value), prev(nullptr),
next(nullptr) {}
};

class LRUCache {
private:
    unordered_map<int, DLinkedNode*> cache;
    DLinkedNode* head;
    DLinkedNode* tail;
    int size;
    int capacity;

public:
    LRUCache(int _capacity): capacity(_capacity), size(0) {
        // 使用伪头部和伪尾部节点
        head = new DLinkedNode();
        tail = new DLinkedNode();
        head->next = tail;
        tail->prev = head;
    }

    int get(int key) {
        if (!cache.count(key)) {
            return -1;
        }
        // 如果 key 存在，先通过哈希表定位，再移到头部
        DLinkedNode* node = cache[key];
        moveToHead(node);
        return node->value;
    }

    void put(int key, int value) {
        if (!cache.count(key)) {
            // 如果 key 不存在，创建一个新的节点
            DLinkedNode* node = new DLinkedNode(key, value);
            // 添加进哈希表
            cache[key] = node;
```

```
            // 添加至双向链表的头部
            addToHead(node);
            ++size;
            if (size > capacity) {
                // 如果超出容量，删除双向链表的尾部节点
                DLinkedNode* removed = removeTail();
                // 删除哈希表中对应的项
                cache.erase(removed->key);
                // 防止内存泄漏
                delete removed;
                --size;
            }
        }
        else {
            // 如果 key 存在，先通过哈希表定位，再修改 value，并移到头部
            DLinkedNode* node = cache[key];
            node->value = value;
            moveToHead(node);
        }
    }

    void addToHead(DLinkedNode* node) {
        node->prev = head;
        node->next = head->next;
        head->next->prev = node;
        head->next = node;
    }

    void removeNode(DLinkedNode* node) {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }

    void moveToHead(DLinkedNode* node) {
        removeNode(node);
        addToHead(node);
    }

    DLinkedNode* removeTail() {
        DLinkedNode* node = tail->prev;
        removeNode(node);
        return node;
    }
};
```