Leetcode算法题：第一周

问题

1. 双指针(题号：167)：https://leetcode.com/problems/two-sum-ii-input-array-is-sorted/description/*

2. 排序：

- 快速选择、堆排序（题号：215）：https://leetcode.com/problems/kth-largest-element-in-an-array/description/
- 桶排序（题号：347）：https://leetcode.com/problems/top-k-frequent-elements/description/
- 荷兰国旗问题（题号：75）：https://leetcode.com/problems/sort-colors/description/
- 贪心（题号：455）：https://leetcode.com/problems/assign-cookies/description/

（本次leetcode中涉及到排序的问题请同学们不要调用系统库函数去实现，尝试自己手动实现。）

解答

167. 两数之和 II - 输入有序数组

关键词：数组，双指针，二分查找

给定一个已按照升序排列 的有序数组，找到两个数使得它们相加之和等于目标数。

函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。

说明:

- 返回的下标值（index1 和 index2）不是从零开始的。
- 你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。

示例:

输入: numbers = [2, 7, 11, 15], target = 9 输出: [1,2] 解释: 2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2 。

思路1：双指针

由于是有序数组，使用两个指针i, j分别指向数组的首尾两端，然后计算sum = numbers[i]+numbers[j] ，如果sum较target偏大则说明数字过大应该减小numbers[j], 因此j--; 反之同理，i++, 如果遍历结束没有答案则返回空数组；time O(n) ,space O(1); 使用哈希表则会消耗O(n) space

leetcode官方解答：

> 我们可以使用两数之和的解法在 O(n^2)O(n2) 时间 O(1)O(1) 空间暴力解决，也可以用哈希表在 O(n)时间和 O(n) 空间内解决。然而，这两种方法都没有用到输入数组已经排序的性质，我们可以做得更好。

> 我们使用两个指针，初始分别位于第一个元素和最后一个元素位置，比较这两个元素之和与目标值的大小。如果和等于目标值，我们发现了这个唯一解。如果比目标值小，我们将较小元素指针增加一。如果比目标值大，我们将较大指针减小一。移动指针后重复上述比较知道找到答案。
>
> 假设 [... , a, b, c, ... , d, e, f, ...][...,a,b,c,...,d,e,f,...] 是已经升序排列的输入数组，并且元素 b, e是唯一解。因为我们从左到右移动较小指针，从右到左移动较大指针，总有某个时刻存在一个指针移动到 b或 e的位置。不妨假设小指针县移动到了元素 b，这是两个元素的和一定比目标值大，根据我们的算法，我们会向左移动较大指针直至获得结果。
>
> **复杂度分析**
>
> - 时间复杂度：O(n)。每个元素最多被访问一次，共有 n 个元素。
> - 空间复杂度：O(1)。只是用了两个指针。

代码:

```
class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        //哈希表要用space O(n), time O(n)
        //双指针time O(n)
        int i=0,j=numbers.size()-1;
        while(i<j){
            int sum=numbers[i]+numbers[j];
            if(sum==target){
                return {i+1,j+1};
            }
            if(sum>target)
                j--;
            else
                i++;
        }
        return {};
    }
};
```

执行用时 :8 ms, 在所有 C++ 提交中击败了92.77%的用户

内存消耗 :9.4 MB, 在所有 C++ 提交中击败了79.15%的用户

215. 数组中的第K个最大元素

关键词：分治算法，堆

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1:

输入: [3,2,1,5,6,4] 和 k = 2 输出: 5

示例 2:

输入: [3,2,3,1,2,4,5,5,6] 和 k = 4 输出: 4

说明: 你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

思路1： 直接使用排序

使用排序的思路很容易想，如果按照从小到大的元素对数组排序，那么第k大的元素则为nums[nums.size()-k]。

- 使用快速排序，time O(nlog(n)), space O(1) 原地排序，改变了输入数组；

代码1:

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        quicksort(nums, 0, nums.size()-1);
        return nums[nums.size()-k];
    }
    int partition(vector<int>& nums, int start, int end){
        int pivot=nums[start];
        int i=start, j=end;
        while(i<j){
            while(i<j && nums[j]>=pivot)
                j--;
            nums[i]=nums[j];
            while(i<j && nums[i]<=pivot)
                i++;
            nums[j]=nums[i];
        }
        nums[i]=pivot;
        return i;
    }
    void quicksort(vector<int>& nums, int start, int end){
        if(start>=end) return;
        int mid=partition(nums, start, end);
        quicksort(nums, start, mid-1);
        quicksort(nums, mid+1, end);
    }
};
```

执行用时 :128 ms, 在所有 C++ 提交中击败了13.32%的用户

内存消耗 :9.8 MB, 在所有 C++ 提交中击败了21.21%的用户

- 使用归并排序，time O(n), space O(n)，因为归并排序要使用额外的空间来中转数组；

代码2

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        mergesort(nums, 0, nums.size()-1);

        return nums[nums.size()-k];
```

```
    }
    void merge(vector<int>& nums, int start, int mid, int end){
        vector<int> tmp;
        int i=start, j=mid+1;
        while(i<=mid && j<=end){
            int cur=nums[i]<=nums[j]?nums[i++]:nums[j++];
            tmp.push_back(cur);
        }
        while(i<=mid){
            tmp.push_back(nums[i++]);
        }
        while(j<=end){
            tmp.push_back(nums[j++]);
        }
        for(int k=0;k<tmp.size();k++){
            nums[start+k]=tmp[k];
        }
    }
    void mergesort(vector<int>& nums, int start, int end){
        if(start>=end) return;
        int mid=start+(end-start)/2;
        mergesort(nums, start, mid);
        mergesort(nums, mid+1, end);
        merge(nums, start, mid, end);
    }
};
```

执行用时 :60 ms, 在所有 C++ 提交中击败了28.80%的用户

内存消耗 :32.9 MB, 在所有 C++ 提交中击败了5.01%的用户

思路2： 看做top(k) 问题，维护一个最小堆

最小堆的堆顶总是最小的，如果一个数大于堆顶元素，那么就将其放入堆顶，替换之前的元素，并对堆进行重构；遍历一遍数组，得到的就是最大的前k个数，堆顶元素即为第k大元素 time O(nlogk), space O(k)

代码3： 使用priority_queue

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {

        struct cmp{
            bool operator()(int a,int b){
                return a>b;
            }
        };
        priority_queue<int, vector<int>, cmp> q;//priority_queue<int, vector<int>, greater<int>
> q;//这种写法也行
        for(int num: nums){
            if(q.size()<k){
                q.push(num);continue;
            }

            if(num>q.top()){
```

```
                q.pop();
                q.push(num);
            }
        }
        return q.top();
    }
};
```

执行用时 :24 ms, 在所有 C++ 提交中击败了58.00%的用户

内存消耗 :9.2 MB, 在所有 C++ 提交中击败了83.75%的用户

代码4： 自建小顶堆，不改变原数组；

与代码1思路一样，只是不借助stl, 自己建堆；由于使用数组建堆对于本题效率更高，因此效果最好

```
class Solution {
public:
    //将id处元素下移叶节点;
    void rebuildheap(vector<int>& heap, int id){
        int k=heap.size()-1;
        int tar=id;
        //构建小顶堆单元，大顶堆为<
        while(id<=k/2){
            if(2*id<=k && heap[id]>heap[2*id]){
                tar=2*id;
            }
            if(2*id+1<=k && heap[tar]>heap[2*id+1]){
                tar=2*id+1;
            }
            if(tar==id) break;
            swap(heap[id],heap[tar]);
            id=tar;
        }
    }
    void buildheap(vector<int>& heap){
        int k=heap.size()-1;
        for(int id=k/2;id>=1;id--){
            rebuildheap(heap,id);
        }
    }
    int findKthLargest(vector<int>& nums, int k) {
        vector<int> heap(k+1,0);
        for(int i=0;i<k;i++){
            heap[i+1]=nums[i];
        }

        buildheap(heap);
        for(int i=k;i<nums.size();i++){
            if(nums[i]>heap[1]){
                heap[1]=nums[i];
                rebuildheap(heap,1);
            }
        }
```

```
        return heap[1];
    }
};
```

执行用时 :4 ms, 在所有 C++ 提交中击败了99.97%的用户

内存消耗 :9.2 MB, 在所有 C++ 提交中击败了83.75%的用户

思路3： 基于快速排序的思路，通过分治算法来进行查找，快速排序partition会将数组分为大于pivot和小于pivot的两部分，如果大于pivot的数字恰好为k-1个，则pivot为最大值；

代码 5： 从数组中挑选一个数（我选择第一个）作为pivot，然后调整顺序，将大于pivot的放到pivot左边，小于pivot放到数组右边；采用二分法，当pivot的序号（记作mid）等于k-1时说明找到了第k大的数，当mid大于k-1时说明需要搜索mid左边,令end=mid-1; 当小于k-1时，说明需要搜索mid右边，令start=mid+1

```
class Solution {
public:
    int partition(vector<int>& nums, int start, int end){
        int pivot=nums[start];
        int i=start,j=end;
        while(i<j){
            while(i<j && nums[j]<=pivot)
                j--;
            nums[i]=nums[j];
            while(i<j && nums[i]>=pivot)
                i++;
            nums[j]=nums[i];
        }
        nums[i]=pivot;
        return i;
    }
    int findKthLargest(vector<int>& nums, int k) {
        int start=0, end=nums.size()-1;
        int res=0;
        while(start<=end){
            int mid=partition(nums,start,end);
            res=nums[mid];
            if(mid==k-1) break;
            if(mid<k-1)
                start=mid+1;
            else
                end=mid-1;
        }
        return res;
    }
};
```

执行用时 :64 ms, 在所有 C++ 提交中击败了27.52%的用户

内存消耗 :9.1 MB, 在所有 C++ 提交中击败了89.75%的用户

总结

```
可以发现，最优解还是自建最小堆 time O(nlogk), space O(k);
使用priority_queue相对耗时，其实也可以使用multiset来实现优先队列的功能
快排分治法 time O(nlogn)space O(1) 但改变了原数组;
直接快排，相比分治多一倍时间，因为分治只需要给一半的数组执行partition;
归并排序消耗空间，timeO(nlogn) space O(n);
```

347. 前K个高频元素

给定一个非空的整数数组，返回其中出现频率前 k 高的元素。

关键词：堆，哈希表，桶排序

示例 1:

```
输入: nums = [1,1,1,2,2,3], k = 2
输出: [1,2]
```

示例 2:

```
输入: nums = [1], k = 1
输出: [1]
```

说明:

- 你可以假设给定的 k 总是合理的，且 1 ≤ k ≤ 数组中不相同的元素的个数。
- 你的算法的时间复杂度必须优于 O(n log n) , n 是数组的大小。

思路1： 由于时间复杂度必须优于O(nlogn)所以直接排序不能满足要求：

```
遍历一遍数组，建立一个哈希表，统计每个元素的频率;
使用priority_queue（或自建最小堆）来选择频率最大的k个数
```

具体操作为： 来源： 五分钟学算法

- 借助 哈希表 来建立数字和其出现次数的映射，遍历一遍数组统计元素的频率
- 维护一个元素数目为 k 的最小堆
- 每次都将新的元素与堆顶元素（堆中频率最小的元素）进行比较
- 如果新的元素的频率比堆顶端的元素大，则弹出堆顶端的元素，将新的元素添加进堆中
- 最终，堆中的 k 个元素即为前 k 个高频元素

复杂度分析： 来源： 五分钟学算法

- 时间复杂度： O(nlogk)， n 表示数组的长度。首先，遍历一遍数组统计元素的频率，这一系列操作的时间复杂度是 O(n)；接着，遍历用于存储元素频率的 map，如果元素的频率大于最小堆中顶部的元素，则将顶部的元素删除并将该元素加入堆中，**这里维护堆的数目是 k** ，所以这一系列操作的时间复杂度是 O(nlogk)的；因此，总的时间复杂度是 O(nlogk) 。
- 空间复杂度：O(n)，最坏情况下（每个元素都不同），map 需要存储 n 个键值对，优先队列需要存储 k个元素，因此，空间复杂度是 O(n)。

代码1： 使用priority_queue

```
//设想利用hash map来存储每个元素的个数；采用小顶堆存储k个元素；time0(n+klogk)

class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        if(nums.size()==0) return{};
        vector<int> res;
        unordered_map<int,int> m;
        for(int num:nums){
            m[num]++;
        }
        struct cmp{
            bool operator()(pair<int,int> a,pair<int,int> b) {return a.second>b.second;}//大于才
是小顶堆
        };
        priority_queue<pair<int,int>, vector< pair<int,int> >, cmp> q;
        for(auto it:m){
            int num=it.first;
            if(q.size()<k){
                q.push(it);
            }
            if(it.second>q.top().second){
                q.pop();q.push(it);
            }
        }
        while(!q.empty()){
            res.push_back(q.top().first);
            q.pop();
        }
        return res;
    }
};
```

执行用时 :28 ms, 在所有 C++ 提交中击败了87.10%的用户

内存消耗 :11.5 MB, 在所有 C++ 提交中击败了20.47%的用户

代码2： 用vector<int>建最小堆：

```
//设想利用hash map来存储每个元素的个数；采用小顶堆存储k个元素；time0(n+klogk)
class Solution {
public:
    unordered_map<int,int> m;
    vector<int> topKFrequent(vector<int>& nums, int k) {
        if(nums.size()==0) return{};
        vector<int> res;
        for(int num:nums){
            m[num]++;
        }
        vector<int> heap(k+1,0);
        int id=1;
        for(auto it:m){
            int num=it.first;
```

```
            if(id<k){
                heap[id++]=num;continue;
            }
            if(id==k){
                heap[id++]=num;buildheap(heap);continue;
            }
            if(it.second>m[heap[1]]){
                heap[1]=num;
                rebuildheap(heap,1);
            }
            id++;
        }
        for(int i=heap.size()-1;i>=1;i--){
            res.push_back(heap[i]);
        }
        return res;
    }
    void rebuildheap(vector<int>& heap, int id){
        int k=heap.size()-1;
        int tar=id;
        //构建小顶堆单元，大顶堆为<
        while(id<=k/2){
            //注意此处比较heap元素所对应频率的大小
            if(2*id<=k && m[heap[id]]>m[heap[2*id]]){
                tar=2*id;
            }
            if(2*id+1<=k && m[heap[tar]]>m[heap[2*id+1]]){
                tar=2*id+1;
            }
            if(tar==id) break;
            swap(heap[id],heap[tar]);
            id=tar;
        }
    }
    void buildheap(vector<int>& heap){
        int k=heap.size()-1;
        for(int id=k/2;id>=1;id--){
            rebuildheap(heap,id);
        }
    }
};
```

执行用时 :28 ms, 在所有 C++ 提交中击败了87.10%的用户

内存消耗 :11.4 MB, 在所有 C++ 提交中击败了33.70%的用户

思路2

1. 通过哈希表统计频率$m[value] = frequency$，可以使用unordered_map< int, int >; 2. 然后通过桶排序（按频率分桶）来进行高频元素的选择：$bucket[frequency] = \{value1, value2, value3...\}$，可以使用vector< vector< int > >;

代码3

```cpp
//设想利用hash map来存储每个元素的个数；采用桶排序来获得前k个高频元素；
class Solution {
public:

    vector<int> topKFrequent(vector<int>& nums, int k) {
        if(nums.size()==0) return{};
        vector<int> res;
        int f_max=0;
        unordered_map<int,int> m;
        for(int num:nums){
            m[num]++;
            f_max=m[num]>f_max?m[num]:f_max;
        }
        vector<vector<int>> bucket(f_max+1,vector(0,0));
        for(auto it:m){
            bucket[it.second].push_back(it.first);
        }

        for(int i=f_max;i>=0;i--){
            for(int j=0;j<bucket[i].size();j++){
                res.push_back(bucket[i][j]);
                k--;
                if(k==0) break;
            }
            if(k==0) break;
        }
        return res;
    }
};
```

75. 颜色分类

关键词：计数排序，数组，双指针，荷兰国旗问题

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

注意: 不能使用代码库中的排序函数来解决这道题。

示例:

输入: [2,0,2,1,1,0] 输出: [0,0,1,1,2,2] **进阶:**

一个直观的解决方案是使用计数排序的两趟扫描算法。 首先，迭代计算出0、1 和 2 元素的个数，然后按照0、1、2的排序，重写当前数组。 你能想出一个仅使用常数空间的一趟扫描算法吗?

思路1:

很明显，这道题与计数排序有关，可以达到O(n)的时间复杂度，O(1)空间复杂度，但是需要两次扫描；通过 一个辅助数组来统计0,1,2的个数，然后输出到nums中，

代码1：

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        //time O(n) space O(1)的两趟计数排序
        vector<int> count(3,0);
        for(int num:nums){
            count[num]++;
        }
        int id=0;
        for(int i=0;i<3;i++){
            while(count[i]>0){
                nums[id++]=i;
                count[i]--;
            }
        }
    }
};
```

思路2： 不设置辅助数组, 通过三个指针来进行位置交换，只扫描一趟；设置L, M, H分别指向0,1,2；LM从0开始，H从最大值开始；

当nums[M]遇到1时直接M++;

当nums[M]为0时，可知本应该由L指向0，因此交换nums[L], nums[M]，并M++,L++; (为了保证M在L,M间，L++时M必须++);

当nums[M]为2时，可知本应该有M指向2，因此交换nums[H],nums[M], 并且H--；此时不必M++，M<=H为执行条件；

代码2：

```
/**
使用3个变量来分别表示3个颜色；
**/
class Solution {
public:
    void sortColors(vector<int>& nums) {
        if(nums.size()==0) return;
        int len=nums.size();
        int L=0,M=0,H=len-1;
        while(M<=H){
            if(nums[M]==1){
                M++;continue;
            }
            if(M<=H&&nums[M]==0){
                swap(nums[M],nums[L]);
                L++;M++;
            }
            if(M<=H&&nums[M]==2){
                swap(nums[M],nums[H]);

                H--;
```

```
            }
        }
    }
};
```

复杂度分析

- 时间复杂度 :由于对长度 N的数组进行了一次遍历，时间复杂度为O(N) 。
- 空间复杂度 :由于只使用了常数空间，空间复杂度为O(1) 。

455. 分发饼干

关键词：贪心算法，快速排序

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，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；并且每块饼干 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。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

注意：

你可以假设胃口值为正。 一个小朋友最多只能拥有一块饼干。

示例 1:

输入: [1,2,3], [1,1]

输出: 1

解释: 你有三个孩子和两块小饼干，3个孩子的胃口值分别是：1,2,3。 虽然你有两块小饼干，由于他们的尺寸都是1，你只能让胃口值是1的孩子满足。 所以你应该输出1。

示例 2:

输入: [1,2], [1,2,3]

输出: 2

解释: 你有两个孩子和三块小饼干，2个孩子的胃口值分别是1,2。 你拥有的饼干数量和尺寸都足以让所有孩子满足。 所以你应该输出2.

思路： 这是一道很简单的贪心算法题目，只需要每次优先给胃口小的孩子发较小的饼干即可，不予赘述；

代码1： 由于这道题是考察贪心算法，因此先直接调用sort

```
class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        int lg=g.size(),ls=s.size();
        if( lg==0 || ls==0 ) return 0;
        sort(g.begin(),g.end());
        sort(s.begin(),s.end());

        int cnt=0;
```

```
        int gi=0,si=0;
        while(gi<lg&&si<ls){
            if(g[gi]<=s[si]){
                gi++;si++;cnt++;
            }else{
                si++;
            }
        }
        return cnt;
    }
};
```

执行用时 :48 ms, 在所有 C++ 提交中击败了92.74%的用户

内存消耗 :10.4 MB, 在所有 C++ 提交中击败了23.31%的用户

代码2： 如果不允许用库函数，那么可以自己实现一个快速排序：

```
class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        int lg=g.size(),ls=s.size();
        if( lg==0 || ls==0 ) return 0;
        quicksort(g,0,lg-1);
        quicksort(s,0,ls-1);
        int cnt=0;
        int gi=0,si=0;
        while(gi<lg&&si<ls){
            if(g[gi]<=s[si]){
                gi++;si++;cnt++;
            }else{
                si++;
            }
        }
        return cnt;
    }
    int partition(vector<int>& nums,int start,int end){
        int pivot=nums[start];
        int i=start,j=end;
        while(i<j){
            //一定注意等号，不然有重复数出现时，不能执行j--和i++，会陷入死循环；
            while(i<j && nums[j]>=pivot){//
                j--;
            }
            nums[i]=nums[j];
            while(i<j && nums[i]<=pivot){
                i++;
            }
            nums[j]=nums[i];
        }
        nums[i]=pivot;
        return i;

    }
```

```
    void quicksort(vector<int>& nums,int start,int end){
        if(start>=end) return;
        int pivot=partition(nums,start,end);
        quicksort(nums,start,pivot-1);
        quicksort(nums,pivot+1,end);
    }
};
```

总结

本周基本上都是排序题目，涉及快速排序，归并排序，堆排序，计数排序，桶排序，以及双指针法，还有简单的贪心算法；