三数之和

- 题解

算法分析

排序 + 双指针

- 1、枚举每个数，表示该数 `nums[i]` 已被确定，在排序后的情况下，通过双指针 `l` ， `r` 分别从左边 `l = i + 1` 和右边 `n - 1` 往中间靠拢，找到 `nums[i] + nums[l] + nums[r] == 0` 的所有符合条件的搭配
- 2、在找符合条件搭配的过程中，假设 `sum = nums[i] + nums[l] + nums[r]`
  - 若 `sum > 0` ，则 `r` 往左走，使 `sum` 变小
  - 若 `sum < 0` ，则 `l` 往右走，使 `sum` 变大
  - 若 `sum == 0` ，则表示找到了与 `nums[i]` 搭配的组合 `nums[l]` 和 `nums[r]` ，存到 `ans` 中
- 3、判重处理
  - 确定好 `nums[i]` 时， `l` 需要从 `i + 1` 开始
  - 当 `nums[i] == nums[i - 1]` ，表示当前确定好的数与上一个一样，需要直接 `continue`
  - 当找符合条件搭配时，即 `sum == 0` ,需要对相同的 `nums[l]` 和 `nums[r]` 进行判重出来

```java
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        int length = nums.length;
        if(length < 3) return new ArrayList<>();
        // 1.排序
        Arrays.sort(nums);

        // i [i + 1, n - 1] left == i + 1 < n
        List<List<Integer>> results = new ArrayList<>();
        for(int i = 0; i < length - 2; i++){
            int left = i + 1, right = length - 1;
            if(i > 0 && nums[i] == nums[i - 1]) continue;
            while(left < right){
                int curThreeSum = nums[i] + nums[left] + nums[right];
                if(curThreeSum < 0){
                    left++;
                }else if(curThreeSum > 0){
                    right--;
                }else{ // left < right && curThreeSum == 0
                    // left 由 左侧的left 向右
                    if(left - 1 > i && nums[left] == nums[left - 1]) left++;
                    else if(right < length - 1 && nums[right] == nums[right + 1]) right--;
                    else{
                        List<Integer> result = new ArrayList<>(Arrays.asList(nums[i], nums[left]
                        left++;
                        right--;
                        results.add(result);
                    }

                }
            }
        }

        return results;

    }
}
```

Points

- 对于left,right重复情况的判断
  - 在满足nums[left] + nums[right] + nums[i] == 0,但是是属于重复情况,需要跳过时
    - left > i + 1(因为nums[i] == nums[left], 在left == i + 1时并不认为是重复情况)

直线上最多的点

算法1

(哈希表) $O(n^2)$

题解：首先我们知道任意两个不同的点就可以唯一确定一条直线。那么我们现在就枚举每一个点作为固定点，将剩下的点按照与该点的斜率划分不同的组。这里需要额外注意的地方是：重复点（我们使用duplicate计数），和固定点在同一条垂线上的点(我们使用vertical计数)。我们使用 `<string,int>` 的哈希表来保存斜率与该斜率的其他点的个数。对于任意两个点的斜率$key = \frac{y_1-y_2}{x_1-x_2}$，我们将分子和分母进行约分后采取字符串的形式保存来避免精度问题。

- 求两个数字的公约数
- 字符串存储分子分母约分后面的结果

```
class Solution {
    public int maxPoints(int[][] p) {
        if(p.length <= 1) return 1;

        // 从points[i] --> points[j]的斜率
        int maxValue = 1;
        for(int i = 0; i < p.length; i++){
            Map<String, Integer> map = new HashMap<>();
            for(int j = i + 1; j < p.length; j++){
                int deltaX = p[i][0] - p[j][0];
                int deltaY = p[i][1] - p[j][1];
                int k = gcd(deltaX, deltaY);
                String key = deltaX / k + "/" + deltaY / k;
                map.put(key, map.getOrDefault(key, 0) + 1);
                maxValue = Math.max(maxValue, map.get(key));
            }
        }

        return maxValue + 1;
    }

    static int gcd(int a, int b){
        if(b == 0) return a;
        return gcd(b, a % b);


    }
}
```

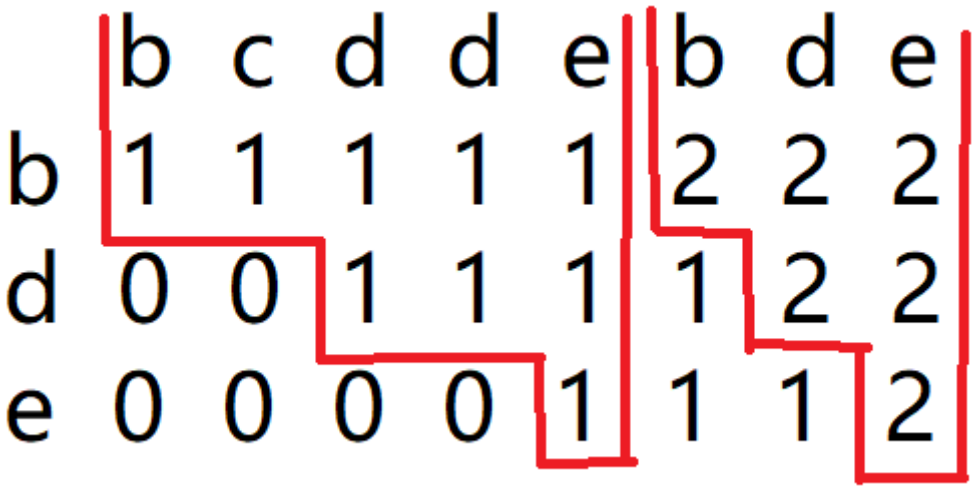
Points

- gcd写法
- 点.length <= 1特判

- 每当遍历到达一个点时,使用一个新的map来记录当前点出发的斜率相同次数, 共线点数 == 相同斜率数 + 1

最小窗口子序列

- 参考解法DP
  - https://leetcode-cn.com/problems/minimum-window-subsequence/solution/dong-tai-gui-hua-by-matrix95/
  - https://leetcode-cn.com/problems/minimum-window-subsequence/solution/du-chuang-zai-zi-fu-pi-pei-biao-zhong-xun-zhao-zui/



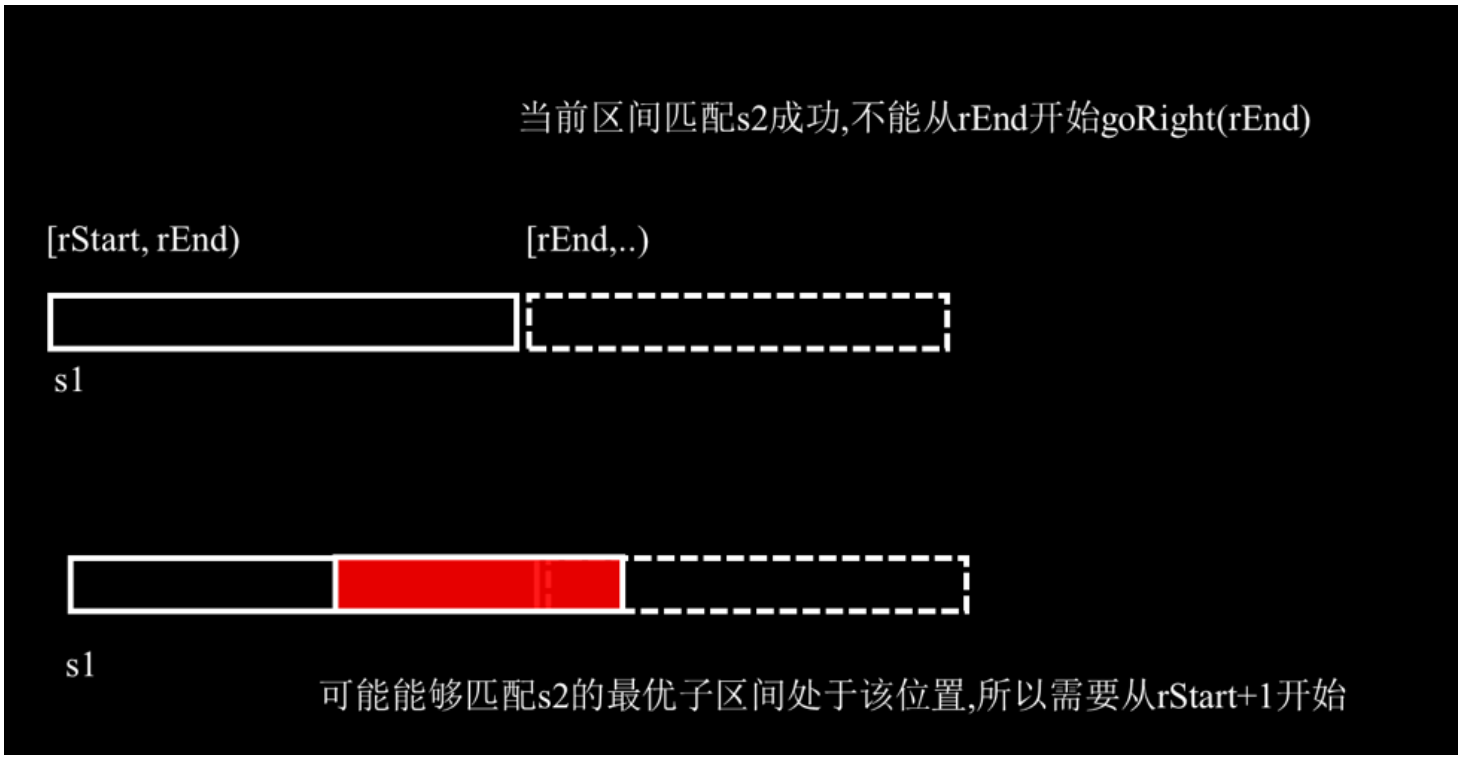
- 参考解法滑窗

```
class Solution {
    static char[] ss1;
    static char[] ss2;
    // ss1的长度 >= ss2
    public String minWindow(String s1, String s2) {
        // 校验ss1,ss2的长度
        if(s1.isEmpty() || s2.isEmpty()) return "";
        if(s2.length() > s1.length()) return "";

        ss1 = s1.toCharArray();
        ss2 = s2.toCharArray();
        // System.out.println(Arrays.toString(ss1));
        int minLength = ss1.length + 1; // 最短子字符串长度
        int mStart = -1, mEnd = -1; // 最短子字符串对应的起始位置,终止位置

        int rStart = 0;// 向右出发的开始位置
        boolean isContains = true; // s1是否包含s2
        while(true){
            int rEnd = goRight(rStart); // 向右出发的结束位置
            if(rStart == 0 && rEnd == -1){ // 第一次向右出发查找失败,s1不包含s2
                isContains = false;
                break;
            }
            if(rEnd == -1) break;

            int rLength = rEnd - rStart; // 向右匹配的长度
            // System.out.println(rStart + "-->" + (rEnd - 1) + "length: " + rLength);

            int lStart = rEnd - 1; // 向左出发的开始位置
            int lEnd = goLeft(lStart);

            int lLength = lStart - lEnd;
            // System.out.println((lEnd + 1) + "<--" + (lStart) + "length: " + lLength);

            if(rLength == lLength && lLength < minLength){
                mStart = rStart;
                mEnd = rEnd;
                minLength = lLength;
            }
            else if(rLength > lLength && lLength < minLength){
                mStart = lEnd + 1;
                mEnd = lStart + 1; // 范围左闭右开
                minLength = lLength;
            }
            // System.out.println("minLength: " + minLength +
            //                    "mStart: " + mStart +
            //                   "mEnd: " + mEnd);

            if(rLength == lLength){
                rStart = rStart + 1; // 易错
            }else if(rLength > lLength){
```

```
                rStart = lEnd + 1;
            }
        }
        if(!isContains || minLength == ss1.length + 1) return "";
        return s1.substring(mStart, mEnd);
    }

    // 从ss1的start位置出发,一直 → 匹配ss2,直到匹配ss2结束

    static int goRight(int start){
        int j = 0; // ss2的匹配位置
        int i = start;
        while(i < ss1.length){
            if(ss1[i] == ss2[j]){
                j++;
            }
            i++;
            if(j >= ss2.length) break;
        }
       //  System.out.println(start + " " + i);
        if(j < ss2.length) return -1; // 匹配ss2失败
        return i;
    }

    // 从ss1的start位置出发, 一直 ← 匹配ss2,直到匹配ss2结束
      static int goLeft(int start){
        int j = ss2.length - 1; // ss2的匹配位置
        int i = start;
        while(i >= 0){
            if(ss1[i] == ss2[j]){
                j--;
            }
            i--;
            if(j < 0) break;
        }
        // 向左走匹配失败的情况应该不太可能 if(j >= 0)
        return i;
    }


}
```

Points

- rStart = rStart + 1; // 易错
- 代表以下情况



最长连续序列

- 可以考虑并查集的**按秩合并**的思路:
  - 每次将树高较低的树合并到树高较高的树中
- 并查集的解法
  - 路径压缩 $O(logn)$
  - 按秩合并 $O(logn)$
  - 路径压缩 + 按秩合并 $O(1)$
- 本题类似于并查集的做法采用的是类似**并查集按秩合并**的思路
- 本题非并查集解法
  - 类似于递推公式
    - 先将数组中的所有元素用set存起来
    - 枚举到x时,检查x+1是否在set中,如果存在,检查x+2,...,一直枚举到y为止,此时以x开始的最长连续子序列长度为$y-x+1$,这个过程可以称为**递推**
    - 这样如果逐一枚举x,时间复杂度还是较高
      - **优化**: 当枚举到x时,假如x-1在集合中,那么表明x不能作为序列的开头,则跳过当前x,换言之,只枚举连续子序列的最小值

```
class Solution {
    public int longestConsecutive(int[] nums) {
        if(nums.length == 0) return 0;

        Set<Integer> set = new HashSet<>();
        for(int num : nums){
            set.add(num);
        }
        int maxLength = 1;
        for(int num : nums){
            if(set.contains(num - 1)){
                continue;
            }
            // num是一段子序列的起点
            int numCopy = num;
            int curLength = 1;
            while(set.contains(numCopy + 1)){
                curLength += 1;
                maxLength = Math.max(maxLength, curLength);
                numCopy += 1;
            }
        }
        return maxLength;

    }
}
```

用两个栈实现队列

```
class MyQueue {
    Stack<Integer> stk1;
    Stack<Integer> stk2;
    /** Initialize your data structure here. */
    public MyQueue() {
        stk1 = new Stack<>();
        stk2 = new Stack<>();
    }

    /** Push element x to the back of queue. */
    public void push(int x) {
        stk1.push(x);
    }

    /** Removes the element from in front of queue and returns that element. */
    public int pop() {
        if(stk2.isEmpty()){
            while(!stk1.isEmpty()){
                int num = stk1.pop();
                stk2.push(num);
            }
        }
        return stk2.pop();
    }

    /** Get the front element. */
    public int peek() {
        if(stk2.isEmpty()){
           while(!stk1.isEmpty()){
                int num = stk1.pop();
                stk2.push(num);
            }
        }
        return stk2.peek();
    }

    /** Returns whether the queue is empty. */
    public boolean empty() {
        return stk1.isEmpty() && stk2.isEmpty();
    }
}

/**
 * Your MyQueue object will be instantiated and called as such:
 * MyQueue obj = new MyQueue();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.peek();
 * boolean param_4 = obj.empty();
 */
```

数组中第k个最大元素

- 三解法

partition解法

```
public int partition(int lo, int hi){
      int idx = (int)(lo + Math.random() * (hi - lo + 1));
      int pivot = arr[idx];
      swap(idx, lo);
      int j = lo;
      for(int i = lo; i < hi + 1; i++){
          if(arr[i] < pivot){
              j++;
              swap(j, i);
          }
      }
      swap(j, lo);
      return j;
  }
```

Points

- Java中生成[lo, hi]的随机一个整数的方式
    - `int idx = (int)(lo + Math.random() * (hi - lo + 1));`

小根堆解法

整体思路

数组中第k个最大元素,即建立[第k大, 第k-1大, 第k-2大,...第1大]的小根堆

- (1)前k个元素建成小根堆,小根堆的容量为k
- (2)剩下的元素和堆顶比较,如果大于堆顶
    - 蒋当前元素 和 堆顶元素**值交换**
    - **将堆顶下滤**
    - (**注意:堆顶元素对应的元素的下标是1,为什么值交换？为什么下滤**)
        - 值交换是因为当前元素值 > 小根堆堆顶元素值,小根堆堆顶因为较小,此时不可能成为第k大元素

- **下滤是因为需要维护小根堆的性质,root < leftChild and root < rightChild**,如果此时被取代后的堆是全数组最大值,那么真正较小的第k大元素永远没法放进来

todo:时间,空间复杂度分析

```java
class Solution {
    public int findKthLargest(int[] nums, int k) {
        MinHeap hp = new MinHeap(k);
        for(int i = 0; i < k; i++){
            hp.add(nums[i]);
        }
       // System.out.println("i:  " + Arrays.toString(hp.heapArr));
       // System.out.println(hp.heapArr[1]);
       // System.out.println(hp.length);
        for(int i = k; i < nums.length; i++){
            if(nums[i] > hp.peek()){
              hp.heapArr[1] = nums[i];
              hp.down(1);
             // System.out.println("i:  " + Arrays.toString(hp.heapArr));
            }
        }
        return hp.peek();
    }
    class MinHeap {
    int length; // 当前堆中的元素个数
    int[]  heapArr; // 用于模拟堆的数组
    static final int DEFAULT_CAP = 10_006; // 堆的默认容量
    MinHeap(){
        length = 0;
        heapArr = new int[DEFAULT_CAP];
    }

     MinHeap(int k){
        length = 0;
        heapArr = new int[k + 1];
    }



    // 下滤:如果父亲节点大于两个孩子,则父亲节点应该下滤,直到堆底
    /**
     * 待下滤的元素的坐标:
     * @param idx
     */
    public void down(int idx){
        int curMin = heapArr[idx]; // 三个节点的最小值
        int curMinIdx = idx; // 三个节点的最小值对应的点的下标

        int lChild = 2 * idx;
        int rChild = 2 * idx + 1;
        // length需要修改一下
        if(lChild <= length && heapArr[lChild] < curMin) {
            curMin = heapArr[lChild];
            curMinIdx = lChild;
        }
        if(rChild <= length && heapArr[rChild] < curMin) {
```

```
            curMin = heapArr[rChild];
            curMinIdx = rChild;
        }
        // 如果发生了交换,下滤curMinIdx
        if(curMinIdx != idx){
            swap(curMinIdx, idx);
            down(curMinIdx);
        }
    }

    // 上滤:如果孩子节点小于父亲节点,则应该上滤,直到堆顶
    public void up(int idx){
        if(idx == 1) return; // 易错
        int curMin = heapArr[idx]; // 三个节点的最小值
        int curMinIdx = idx; // 三个节点的最小值对应的点的下标

        int parent = idx / 2;
        if(parent !=  idx && curMin < heapArr[parent]){
            curMinIdx = parent;
        }
        if(curMinIdx != idx){
            swap(curMinIdx, idx);
            up(curMinIdx);
        }

    }


    public void swap(int x, int y){
        int temp = heapArr[x];
        heapArr[x] = heapArr[y];
        heapArr[y] = temp;
    }

    public void add(int num){
        length++;
        heapArr[length] = num;
        up(length); // 易错点
    }


    public void poll(){
        heapArr[1] = heapArr[length];
        length--;
        down(1);
    }

    public int peek(){
        return heapArr[1];
    }
```

```
    }
}
```

Points

- down,up操作入参都是下标
- up操作,递归基的终点是 `idx == 1` ,因为堆中有效元素的下标从1开始
- length表示堆中元素的数量

大根堆解法

- 数组一共有n个数,那么维护一个大小为n的大顶堆
- poll() k次,poll()完k次之后,此时堆顶的元素就是当前数组中第k大的元素

```
class Solution {
    public int findKthLargest(int[] nums, int k) {
        int n = nums.length;
        MaxHeap hp = new MaxHeap(n);
        for(int i = 0; i < n; i++){
            hp.add(nums[i]);
        }

        for(int i = 0; i < k - 1; i++){
            hp.poll();
        }
        return hp.peek();
    }


    /*
    大顶堆:孩子节点比双亲都大
 */
    public class MaxHeap {
        int length; // 当前堆中的元素个数
        int[] heap; // 用于模拟堆的数组
        static final int DEFAULT_CAP = 10_000; // 堆的默认容量

        MaxHeap(){
            length = 0;
            heap = new int[DEFAULT_CAP];
        }

        MaxHeap(int n){
            length = 0;
            heap = new int[n + 1];
        }



        // 下滤:如果父亲节点小于两个孩子,则父亲节点应该下滤,直到堆底
        /**
         * 待下滤的元素的坐标:
         * @param idx
         */
        // 即:父元素不是父-子的最大者
        private void down(int idx){
            int maxIdx = idx; // 三个节点的最大值对应的点的下标

            int lChild = 2 * idx;
            int rChild = 2 * idx + 1;
            if(lChild <= length && heap[lChild] > heap[maxIdx]) {
                maxIdx = lChild;
            }
            if(rChild <= length && heap[rChild] > heap[maxIdx]) {
                maxIdx = rChild;
```

```
            }

            if(maxIdx != idx){
                swap(maxIdx, idx);
                down(maxIdx);
            }
        }

        // 上滤:如果孩子节点大于父亲节点,则应该上滤,直到堆顶
        // 当前节点,不是父亲-孩子中的较小者
        private void up(int idx){
            int minIdx= idx; // 三个节点的最小值对应的点的下标
            int parent = idx / 2;
            if(parent > 0 && heap[minIdx] > heap[parent]){
                minIdx = parent;
            }
            if(minIdx != idx){
                swap(minIdx, idx);
                up(minIdx);
            }

        }

        private void swap(int x, int y){
            int temp = heap[x];
            heap[x] = heap[y];
            heap[y] = temp;
        }

        public void add(int num){
            length++;
            heap[length] = num;
            up(length);
        }

        public void poll(){
            swap(1, length);
            length--;
            down(1);
        }

        public int peek(){
            return heap[1];
        }
    }

}
```

数据流中的中位数

使用对顶堆

- 规定:
  - 如果对顶堆中的元素为奇数,如2m + 1
  - 大顶堆的元素的数量为m + 1, 小顶堆的元素数量为m
  - 此时数据流中的中位数为大顶堆堆顶

当cnt为偶数例如2m

- 小顶堆存放[m + 1, 2m] --> 堆顶[cnt / 2 + 1, cnt]
- 大顶堆存放[1, m] --> [1, cnt / 2]堆顶

添加一个数字num时

- 如果 <= 大顶堆堆顶 添加到大顶堆中,此时大顶堆共有m + 1个元素
- 如果 >= 小顶堆堆顶, 添加到小顶堆中
  - num ← 小顶堆poll()
  - 将num添加到大顶堆中

当cnt为奇数,例如2m + 1

- 小顶堆存放[m + 2, 2m + 1] --> 堆顶[cnt / 2 + 2, cnt]
- 大顶堆存放[1, m + 1] --> [1, cnt / 2 + 1]堆顶

添加一个数字num时

- 如果 <= 大顶堆堆顶 添加到大顶堆中,此时大顶堆共有2m + 2个元素
  - num ← 大顶堆poll()
  - 将num添加到小顶堆中
- 如果 >= 小顶堆堆顶, 添加到小顶堆中
  - num ← 小顶堆poll()
  - 将num添加到大顶堆中