LeetCode Online Coding Interview Questions – 错题集

deepwaterooo

August 25, 2022

Contents

Chapter 1

扫描线

1.0.1 435. Non-overlapping Intervals - Medium 动态规划贪心

Given an array of intervals intervals where intervals[i] = [starti, endi], return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

1. 解题思路与分析: 动态规划 O（N^2）tle

   方法一：动态规划

   思路与算法

   题目的要求等价于「选出最多数量的区间，使得它们互不重叠」。由于选出的区间互不重叠，因此我们可以将它们按照端点从小到大的顺序进行排序，并且无论我们按照左端点还是右端点进行排序，得到的结果都是唯一的。

   这样一来，我们可以先将所有的 n 个区间按照左端点（或者右端点）从小到大进行排序，随后使用动态规划的方法求出区间数量的最大值。设排完序后这 n 个区间的左右端点分别为 $l_0, \cdots, l_{n-1}$ 以及 $r_0, \cdots, r_{n-1}$，那么我们令 f_i 表示「以区间 i 为最后一个区间，可以选出的区间数量的最大值」，状态转移方程即为：

   $$f_i = \max_{j<i \wedge r_j \le l_i} \{f_j\} + 1$$

   即我们枚举倒数第二个区间的编号 j，满足 $j < i$，并且第 j 个区间必须要与第 i 个区间不重叠。由于我们已经按照左端点进行升序排序了，因此只要第 j 个区间的右端点 r_j 没有越过第 i 个区间的左端点 l_i，即 $r_j \le l_i$，那么第 j 个区间就与第 i 个区间不重叠。我们在所有满足要求的 j 中，选择 f_j 最大的那一个进行状态转移，如果找不到满足要求的区间，那么状态转移方程中 $\min$ 这一项就为 0，f_i 就为 1。

   最终的答案即为所有 f_i 中的最大值。

```
public int eraseOverlapIntervals(int[][] a) { // 刚过去周六早上的比赛简单的当时就想出来 O(NlogN) 的解法了，这里居然还有些不通
    int n = a.length, max = 0;
    Arrays.sort(a, (x, y) -> x[0] != y[0] ? x[0] - y[0] : x[1] - y[1]); // starttime, then end time
    int [] dp = new int [n];
    Arrays.fill(dp, 1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (a[j][1] > a[i][0]) continue;
            dp[i] = Math.max(dp[i], dp[j] + 1);
        }
        max = Math.max(max, dp[i]);
    }
    return n-max;
}
```

2. 解题思路与分析: 贪心

方法二：贪心

思路与算法

我们不妨想一想应该选择哪一个区间作为首个区间。

假设在某一种**最优**的选择方法中，$[l_k, r_k]$ 是首个（即最左侧的）区间，那么它的左侧没有其它区间，右侧有若干个不重叠的区间。设想一下，如果此时存在一个区间 $[l_j, r_j]$，使得 $r_j < r_k$，即区间 j 的右端点在区间 k 的左侧，那么我们将区间 k 替换为区间 j，其与剩余右侧被选择的区间仍然是不重叠的。而当我们将区间 k 替换为区间 j 后，就得到了另一种**最优**的选择方法。

我们可以不断地寻找右端点在首个区间右端点左侧的新区间，将首个区间替换成该区间。那么当我们无法替换时，**首个区间就是所有可以选择的区间中右端点最小的那个区间**。因此我们将所有区间按照右端点从小到大进行排序，那么排完序之后的首个区间，就是我们选择的首个区间。

如果有多个区间的右端点都同样最小怎么办？由于我们选择的是首个区间，因此在左侧不会有其它的区间，那么左端点在何处是不重要的，我们只要任意选择一个右端点最小的区间即可。

当确定了首个区间之后，所有与首个区间不重合的区间就组成了一个规模更小的子问题。由于我们已经在初始时将所有区间按照右端点排好序了，因此对于这个子问题，我们无需再次进行排序，只要找出其中**与首个区间不重合**并且右端点最小的区间即可。用相同的方法，我们可以依次确定后续的所有区间。

在实际的代码编写中，我们对按照右端点排好序的区间进行遍历，并且实时维护上一个选择区间的右端点 $right$。如果当前遍历到的区间 $[l_i, r_i]$ 与上一个区间不重合，即 $l_i \geq right$，那么我们就可以贪心地选择这个区间，并将 $right$ 更新为 r_i。

```
public int eraseOverlapIntervals(int[][] intervals) {
    if (intervals.length == 0) return 0;
    Arrays.sort(intervals, (a, b)->a[1] - b[1]);
    int n = intervals.length;
    int right = intervals[0][1];
    int ans = 1;
    for (int i = 1; i < n; ++i)
        if (intervals[i][0] >= right) {
            ++ans;
            right = intervals[i][1];
        }
    return n - ans;
}
```

1.0.2 2054. Two Best Non-Overlapping Events - Medium

You are given a 0-indexed 2D integer array of events where events[i] = [startTimei, endTimei, valuei]. The ith event starts at startTimei and ends at endTimei, and if you attend this event, you will receive a value of valuei. You can choose at most two non-overlapping events to attend such that the sum of their values is maximized.

Return this maximum sum.

Note that the start time and end time is inclusive: that is, you cannot attend two events where one of them starts and the other ends at the same time. More specifically, if you attend an event with end time t, the next event must start at or after t + 1.

1. 解题思路与分析

   - 因为最多只有两个事件，联想到股票问题的只有两次交易，很容易想到从头到尾扫一遍，从尾到头扫一遍
   - 也是一道扫描线问题，借助 TreeMap, PriorityQueue 等数据结构，或 binary search 算法帮助保证正确性以及提速
   - 再熟悉一下 NavigableMap 类中的 floorKey() 和 ceilingKey() 的 API：

类型	方法	描述
K	ceilingKey(K key)	返回大于或等于给定键的最小键，如果没有这样的键，则 null
K	floorKey(K key)	返回小于或等于给定键的最大键，如果没有这样的键，则 null

```
public int maxTwoEvents(int[][] events) {
    int n = events.length, maxSeenVal = 0;     // for any timestamp - find max of values to left and max of values to right
    Arrays.sort(events, (a,b) -> a[1] - b[1]); // Sort by end time - to know what events happened previously
```

```
    TreeMap<Integer, Integer> leftMaxSeen = new TreeMap<>();
    for (int [] event : events) {
        int eventVal = event[2];
        if (eventVal >= maxSeenVal) {
            maxSeenVal = eventVal;
            leftMaxSeen.put(event[1], maxSeenVal); // save maxSeenVal event value seen so far at END timestamp
        }
    }
    // Sort by start time but in descending order - as we parse events and store max value of future events to current event start timestamp
    Arrays.sort(events, (a,b) -> b[0] - a[0]);
    int ans = 0, maxSeenRight = 0;
    for (int [] event : events) {
        int eventVal = event[2];
        if (eventVal >= maxSeenRight) {
            maxSeenRight = eventVal;
            Integer maxOnLeftKey = leftMaxSeen.floorKey(event[0] - 1);
            if (maxOnLeftKey != null)
                ans = Math.max(ans, leftMaxSeen.get(maxOnLeftKey) + maxSeenRight);
        }
    }
    return Math.max(maxSeenVal, ans); // 有一种单个事件得最大值的情况不能漏掉
}
```

- 扫描线算法

```
public int maxTwoEvents(int[][] events) {
    int result = 0, maxOfCompletedEvents = 0;
    Arrays.sort(events, (x,y) -> x[0] - y[0]);   // Sort by Start time
    PriorityQueue<int[]> inProgressQueue = new PriorityQueue<>((x,y)->x[1]-y[1]); // sorted by end time
    for (int[] currentEvent : events) {
        while (!inProgressQueue.isEmpty() && inProgressQueue.peek()[1] < currentEvent[0])
            maxOfCompletedEvents = Math.max(maxOfCompletedEvents, inProgressQueue.poll()[2]);
        result = Math.max(result, maxOfCompletedEvents + currentEvent[2]);
        inProgressQueue.offer(currentEvent);
    }
    return result;
}
```

- 二分查找: 改天补上

1.0.3 2055. Plates Between Candles - Medium

There is a long table with a line of plates and candles arranged on top of it. You are given a 0-indexed string s consisting of characters '*' and '|' only, where a '*' represents a plate and a '|' represents a candle.

You are also given a 0-indexed 2D integer array queries where queries[i] = [lefti, righti] denotes the substring s[lefti...righti] (inclusive). For each query, you need to find the number of plates between candles that are in the substring. A plate is considered between candles if there is at least one candle to its left and at least one candle to its right in the substring.

For example, s = "||**||**|*", and a query [3, 8] denotes the substring "*||**|". The number of plates between candles in this substring is 2, as each of the two plates has at least one candle in the substring to its left and right. Return an integer array answer where answer[i] is the answer to the ith query.

1. 解题思路与分析

   - 先把每个调用的最左、以及最右的蜡烛的位置找出来，假如作必要的前置处理的话，可以做到 O(1) 时间
   - 而要数这最左与最右蜡烛之间的盘子个数的话，如果我们前置数清楚所有位置蜡烛个数，我们也可以做到 O(1) 时间
   - 所以，使用三个数组，一个记录各个位置蜡烛总个数，另两个分别纪录左右端点

```
public int[] platesBetweenCandles(String t, int[][] queries) {
    int n = t.length();
    int [] sum = new int [n+1];
    int [] pre = new int [n+1], suf = new int [n+1];
    char [] s = t.toCharArray();
    for (int i = 0; i < n; i++) {
        sum[i+1] = sum[i] + (s[i] == '|' ? 1 : 0);
        pre[i+1] = s[i] == '|' ? i : pre[i]; // pre[i] matches i-1
    }
    for (int i = n-1; i >= 0; i--)
        suf[i] = s[i] == '|' ? i : suf[i+1]; // suf[i] matches i
    int [] ans = new int [queries.length];
    for (int i = 0; i < queries.length; i++) {
        int l = suf[queries[i][0]], r = pre[queries[i][1]+1]; // 注意: 右蜡烛边界
        if (l < r)
            ans[i] = r - l  - (sum[r] - sum[l]);
    }
    return ans;
}
```

1.0.4 218. The Skyline Problem - Hard

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Given the locations and heights of all the buildings, return the skyline formed by these buildings collectively.

The geometric information of each building is given in the array buildings where buildings[i] = [lefti, righti, heighti]:

lefti is the x coordinate of the left edge of the ith building. righti is the x coordinate of the right edge of the ith building. heighti is the height of the ith building. You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

The skyline should be represented as a list of "key points" sorted by their x-coordinate in the form [[x1,y1],[x2,y2],...]. Each key point is the left endpoint of some horizontal segment in the skyline except the last point in the list, which always has a y-coordinate 0 and is used to mark the skyline's termination where the rightmost building ends. Any ground between the leftmost and rightmost buildings should be part of the skyline's contour.

Note: There must be no consecutive horizontal lines of equal height in the output skyline. For instance, [...,[2 3],[4 5],[7 5],[11 5],[12 7],...] is not acceptable; the three lines of height 5 should be merged into one in the final output as such: [...,[2 3],[4 5],[12 7],...]

1. 解题思路与分析

   - 复杂度分析

   时间复杂度：O(nlogn)，其中 nn 为建筑数量。每座建筑至多只需要入队与出队一次，单次时间复杂度为 O(logn)。

   空间复杂度：O(n)，其中 nn 为建筑数量。数组 boundaries 和优先队列的空间占用均为 O(n)。

```
public List<List<Integer>> getSkyline(int[][] a) {
    List<Integer> pos = new ArrayList<>();
    for (int [] b : a) {
        pos.add(b[0]);
        pos.add(b[1]);
    }
    Collections.sort(pos);
    List<List<Integer>> ans = new ArrayList<>();
    Queue<int []> q = new PriorityQueue<>((x, y) -> y[1] - x[1]);
    int n = a.length, idx = 0;
    for (int v : pos) {
        while (idx < n && a[idx][0] <= v) {
            q.offer(new int [] {a[idx][1], a[idx][2]});
            idx++;
        }
        while (!q.isEmpty() && q.peek()[0] <= v) q.poll();
        int maxHiCur = q.isEmpty() ? 0 : q.peek()[1];
        if (ans.size() == 0 || maxHiCur != ans.get(ans.size()-1).get(1))
            ans.add(Arrays.asList(v, maxHiCur));
    }
    return ans;
}
```

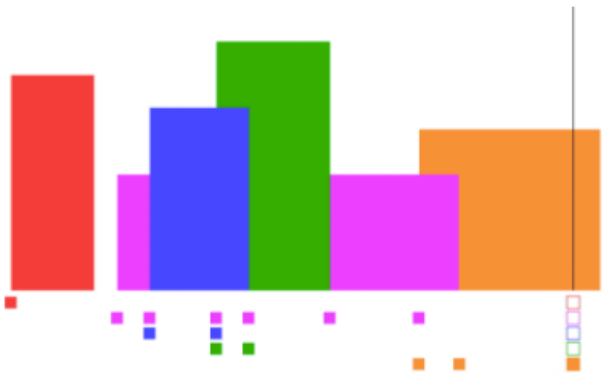
2. 解题思路与分析

   使用扫描线，从左至右扫过。如果遇到左端点，将高度入堆，如果遇到右端点，则将高度从堆中删除。使用 last 变量记录上一个转折点。O(NLogN)

   扫描线法

   使用扫描线，从左至右扫过。如果遇到左端点，将高度入堆，如果遇到右端点，则将高度从堆中删除。使用 last 变量记录上一个转折点。

   可以参考下面的图，扫描线下方的方格就是堆。



```java
public List<List<Integer>> getSkyline(int[][] buildings) { // 从左向右扫一遍过去
    List<List<Integer>> ans = new ArrayList<>();
    Map<Integer, List<Integer>> map = new TreeMap<>(); // TreeMap, 取其根据键的大小自动排序之意, 可以得到 logN 的效率
    for (int [] v : buildings) {
        map.computeIfAbsent(v[0], z -> new ArrayList<>()).add(-v[2]); // 插入左节点: 高度为负
        map.computeIfAbsent(v[1], z -> new ArrayList<>()).add(v[2]);  // 插入右节点: 高度为正
    }
    Map<Integer, Integer> heights = new TreeMap<>((a, b) -> b-a); // 保留当前位置的所有高度 重定义排序: 从大到小
    int [] last = {0, 0}; // 保留上一个位置的横坐标及高度
    for (Integer key : map.keySet()) {
        List<Integer> ys = map.get(key);
        Collections.sort(ys);
        for (int y : ys) {
            if (y < 0) // 左端点, 高度入队
                heights.put(-y, heights.getOrDefault(-y, 0) + 1);
            else {     // 右端点移除高度
                heights.put(y, heights.getOrDefault(y, 1) - 1);
                if (heights.get(y) == 0) heights.remove(y);
            }
        }
        Integer maxHeight = 0;      // 获取 heights 的最大值: 就是第一个值
        if (!heights.isEmpty())
            maxHeight = heights.keySet().iterator().next();
        if (last[1] != maxHeight) { // 如果当前最大高度不同于上一个高度, 说明其为转折点
            last[0] = key;          // 更新 last, 并加入结果集
            last[1] = maxHeight;
            ans.add(Arrays.asList(key, maxHeight));
        }
    }
    return ans;
}
```

3. 解题思路与分析: 比较慢

   思路：类似于 meeting room 的方法，用 hashmap 存《start, +1》和《end, -1》。在这里就是先用一个 List《int[]》来存储《start, height》和《end, -height》, 然后把这个 list 按照从小到大排序，最后用一个 queue 来走一遍每个点，每次遇到正值的 height，就 offer，遇到负值就 pop 此值，然后在每一个点，都 peek heap 中的最大 height，如果这个 height 和之前存在 result 里的值不同，将此点和值存进 result，否则更新 pre，继续向后走，直到走完。

```java
public List<List<Integer>> getSkyline(int[][] buildings) {
    List<List<Integer>> ans = new ArrayList<>();
    List<int[]> hite = new ArrayList<>();
    for (int [] v : buildings) {
        hite.add(new int [] {v[0], v[2]});
        hite.add(new int [] {v[1], -v[2]});
    }
    Collections.sort(hite, (a, b)-> b[0] != a[0] ? a[0] - b[0] : b[1] - a[1]);
    Queue<Integer> q = new PriorityQueue<>((a, b)-> b - a);
    q.offer(0);
    int cur = 0, pre = 0;
    for (int [] h : hite) {
        if (h[1] > 0) q.offer(h[1]);
        else q.remove(-h[1]);
        cur = q.peek();
        if (pre != cur) {
            ans.add(List.of(h[0], cur));
            pre = cur;
        }
    }
    return ans;
}
```

4. 解题思路与分析: Java divide and conquer solution beats 96% todo

   The basic idea is divide the buildings into two subarrays, calculate their skylines respectively, then merge two skylines together.

   - https://leetcode.com/problems/the-skyline-problem/discuss/61281/Java-divide-and-conquer-solution-b

```java
public List<List<Integer>> getSkyline(int[][] buildings) { // todo; divide and conque 代码被我改乱了
    return merge(buildings, 0, buildings.length-1);        // 今天这个题看累了, 改天再补这个
}
private List<List<Integer>> merge(int[][] buildings, int lo, int hi) {
    List<List<Integer>> res = new LinkedList<>();
    if (lo > hi) {
        return res;
    } else if (lo == hi) {
        res.add(List.of(buildings[lo][0], buildings[lo][2]));
        res.add(List.of(buildings[lo][1], 0));
        return res;
    }
    int mid = lo+(hi-lo)/2;
    List<List<Integer>> left = merge(buildings, lo, mid);
    List<List<Integer>> right = merge(buildings, mid+1, hi);
    int leftH = 0, rightH = 0;
```

```
            while(!left.isEmpty() || !right.isEmpty()) {
                long x1 = left.isEmpty()? Long.MAX_VALUE: left.peekFirst()[0];
                long x2 = right.isEmpty()? Long.MAX_VALUE: right.peekFirst()[0];
                int x = 0;
                if(x1 < x2) {
                    int[] temp = left.pollFirst();
                    x = temp[0];
                    leftH = temp[1];
                } else if(x1 > x2) {
                    int[] temp = right.pollFirst();
                    x = temp[0];
                    rightH = temp[1];
                } else {
                    x = left.peekFirst()[0];
                    leftH = left.pollFirst()[1];
                    rightH = right.pollFirst()[1];
                }
                int h = Math.max(leftH, rightH);
                if(res.isEmpty() || h != res.peekLast()[1])
                    res.add(List.of(x, h));
            }
            return res;
        }
```

1.0.5 715. Range Module - Hard

A Range Module is a module that tracks ranges of numbers. Design a data structure to track the ranges represented as half-open intervals and query about them.

A half-open interval [left, right) denotes all the real numbers x where left <= x < right.

Implement the RangeModule class:

RangeModule() Initializes the object of the data structure. void addRange(int left, int right) Adds the half-open interval [left, right), tracking every real number in that interval. Adding an interval that partially overlaps with currently tracked numbers should add any numbers in the interval [left, right) that are not already tracked. boolean queryRange(int left, int right) Returns true if every real number in the interval [left, right) is currently being tracked, and false otherwise. void removeRange(int left, int right) Stops tracking every real number currently being tracked in the half-open interval [left, right).

```
class Range {
    int left, right;
    public Range(int left, int right) {
        this.left = left;
        this.right = right;
    }
}
TreeSet<Range> ts;
public RangeModule() {
    ts = new TreeSet<>((a, b)->(a.left != b.left ? a.left - b.left : a.right - b.right));
}
public void addRange(int left, int right) {
    int nl = left, nr = right;
    Range high = new Range(right, Integer.MAX_VALUE);
    while (true) {
        Range r = ts.lower(high);
        if (r == null || r.right < left) break;
        if (r.right > right) nr = r.right;
        if (r.left < left) nl = r.left;
        ts.remove(r);
    }
    ts.add(new Range(nl, nr));
}
public boolean queryRange(int left, int right) {
    Range target = ts.floor(new Range(left, Integer.MAX_VALUE));
    return target != null && target.left <= left && target.right >= right;
}
public void removeRange(int left, int right) {
    Range high = new Range(right, right);
    while (true) {
        Range r = ts.lower(high);
        if (r == null || r.right <= left) break;
        if (r.right > right)
            ts.add(new Range(right, r.right));
        if (r.left < left)
            ts.add(new Range(r.left, left));
        ts.remove(r);
    }
}
```

1.0.6 352. Data Stream as Disjoint Intervals - Hard

Given a data stream input of non-negative integers a1, a2, . . . , an, summarize the numbers seen so far as a list of disjoint intervals.

Implement the SummaryRanges class:

SummaryRanges() Initializes the object with an empty stream. void addNum(int val) Adds the integer val to the stream. int[][] getIntervals() Returns a summary of the integers in the stream currently as a list of disjoint intervals [starti, endi].

```
    class Range implements Comparable<Range> {
        int bgn, end;
        public Range(int bgn, int end) {
            this.bgn = bgn;
            this.end = end;
        }
        @Override public int compareTo(Range other) {
            return this.bgn - other.bgn;
        }
    }
    TreeSet<Range> ts;
    public SummaryRanges() {
        ts = new TreeSet<Range>();
    }
    public void addNum(int val) {
        Range cur = new Range(val, val);
        Range bef = ts.floor(cur);
        Range aft = ts.ceiling(cur);
        if (bef != null && bef.end + 1 >= val) {
            cur.bgn = bef.bgn;
            cur.end = Math.max(val, bef.end);
            ts.remove(bef);
        }
        if (aft != null && aft.bgn == val + 1) {
            cur.end = aft.end;
            ts.remove(aft);
        }
        ts.add(cur);
    }
    public int[][] getIntervals() {
        int [][] ans = new int [ts.size()][2];
        int i = 0;
        for (Range cur : ts) {
            ans[i][0] = cur.bgn;
            ans[i][1] = cur.end;
            i++;
        }
        return ans;
    }
```

1.0.7 1419. Minimum Number of Frogs Croaking - Medium

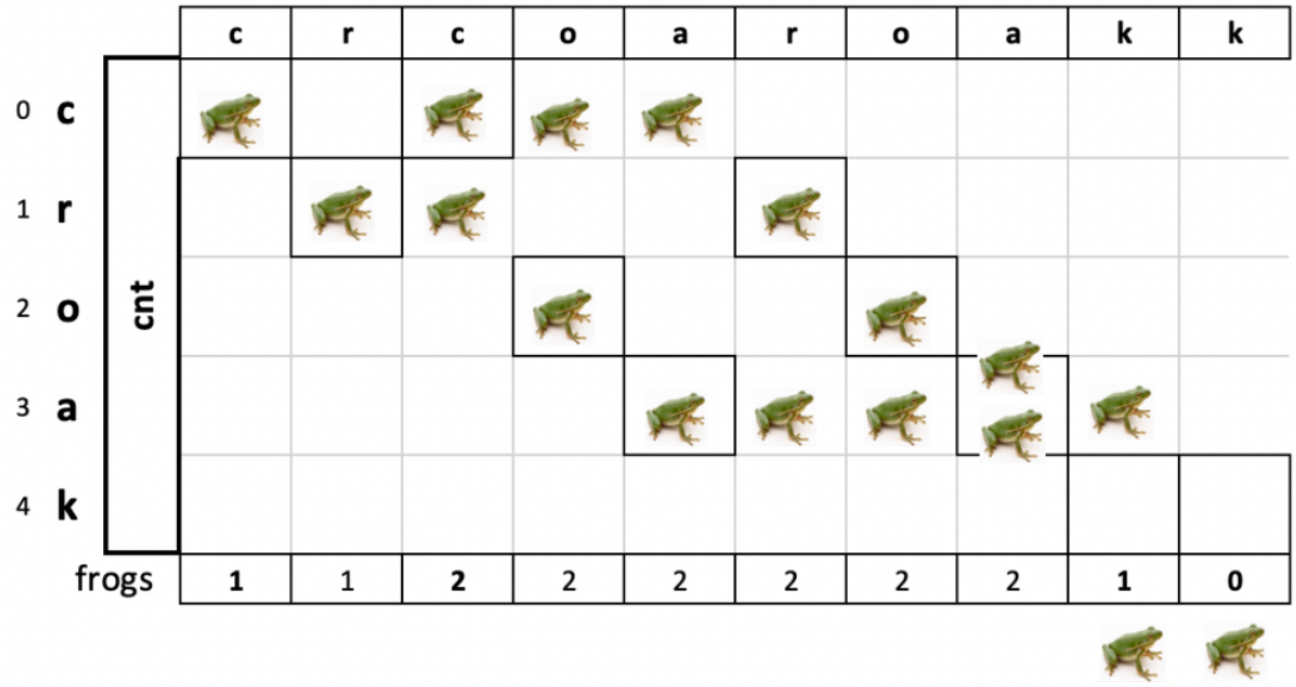
Given the string croakOfFrogs, which represents a combination of the string "croak" from different frogs, that is, multiple frogs can croak at the same time, so multiple “croak” are mixed. Return the minimum number of different frogs to finish all the croak in the given string.

A valid "croak" means a frog is printing 5 letters ‘c’ , ’ r’ , ’ o’ , ’ a’ , ’ k’ sequentially. The frogs have to print all five letters to finish a croak. If the given string is not a combination of valid "croak" return -1.

```
public int minNumberOfFrogs(String s) { // 写得有点儿乱
    int n = s.length();
    int cc = 0, cr = 0, co = 0, ca = 0, cnt = 0, max = 0;
    for (int i = 0; i < n; i++) {
        char c = s.charAt(i);
        if (c == 'c') {
            ++cc;
            ++cnt;
            max = Math.max(max, cnt);
        } else if (c == 'r') {
            if (cc == 0) return -1;
            --cc;
            ++cr;
        } else if (c == 'o') {
            if (cr == 0) return -1;
            --cr;
            ++co;
        } else if (c == 'a') {
            if (co == 0) return -1;
            --co;
            ++ca;
        } else if (c == 'k') {
            if (ca == 0) return -1;
            --ca;
            --cnt;
        }
    }
    if (cc + cr + co + ca > 0) return -1;
    return max;
}
```

We can track how many frogs are 'singing' each letter in `cnt`:

- Increase number of frogs singing this letter, and decrease number singing previous letter.
- When a frog sings 'c', we increase the number of (simultaneous) `frogs`.
- When a frog sings 'k', we decrease the number of (simultaneous) `frogs`.
- If some frog is singing a letter, but no frog sang the previous letter, we return `-1`.



Track and return the maximum number of frogs ever signing together.

> Catch: if some frog hasn't finished croaking, we need to return `-1`.

```
public int minNumberOfFrogs(String s) {
    int n = s.length();
    int cnt [] = new int [5];
    int cur = 0, max = 0;
    for (int i = 0; i < n; i++) {
        char c = s.charAt(i);
        int idx = "croak".indexOf(c);
        ++cnt[idx];
        if (idx == 0)
            max = Math.max(max, ++cur);
        else if (--cnt[idx-1] < 0)
            return -1;
        else if (idx == 4)
            --cur;
    }
    return cur == 0 ? max : -1; // 如果最后所有的青蛙都叫完了的话
}
```

- 解法三:

这个跟上面的解法差不多，优化了查询位置的时间。直接对比每一步的结果。如果当前字符位的个数比上一位多，那么说明当前位没有办法被匹配成功。

```
public int minNumberOfFrogs(String croakOfFrogs) {
    char[] ch = croakOfFrogs.toCharArray();
    int curr = 0;
    int res = 0;
    int c = 0, r = 0, o = 0, a = 0, k = 0;
    for(int i = 0; i < ch.length; i++) {
        if(ch[i] == 'c') {
            c++;
            curr++;
        } else if(ch[i] == 'r')
            r++;
        else if(ch[i] == 'o')
            o++;
         else if(ch[i] == 'a')
            a++;
         else {
            k++;
            curr--;
```

```
        }
        res = Math.max(res, curr);
        if (c < r || r < o || o < a || a < k) // 要满足所有的条件
            return -1;
    }
    if ((c == r) && (r == o ) && ( o == a) &&  (a == k))
        return res;
    return -1;
}
```

Chapter 2

单调栈

2.0.1 975. Odd Even Jump - Hard

You are given an integer array arr. From some starting index, you can make a series of jumps. The (1st, 3rd, 5th, . . .) jumps in the series are called odd-numbered jumps, and the (2nd, 4th, 6th, . . .) jumps in the series are called even-numbered jumps. Note that the jumps are numbered, not the indices.

You may jump forward from index i to index j (with i < j) in the following way:

During odd-numbered jumps (i.e., jumps 1, 3, 5, . . .), you jump to the index j such that arr[i] <= arr[j] and arr[j] is the smallest possible value. If there are multiple such indices j, you can only jump to the smallest such index j. During even-numbered jumps (i.e., jumps 2, 4, 6, . . .), you jump to the index j such that arr[i] >= arr[j] and arr[j] is the largest possible value. If there are multiple such indices j, you can only jump to the smallest such index j. It may be the case that for some index i, there are no legal jumps. A starting index is good if, starting from that index, you can reach the end of the array (index arr.length - 1) by jumping some number of times (possibly 0 or more than once).

Return the number of good starting indices.

1. 解题思路与分析: 树映射（Tree Map）

   在方法一中，原问题简化为：奇数次跳跃时，对于一些索引 i，下一步应该跳到哪里去（如果有的话）。

   我们可以使用 TreeMap，一个维护有序数据的绝佳数据结构。我们将索引 i 映射到 v = A[i] 上。

   从 i = N-2 到 i = 0 的遍历过程中，对于 v = A[i]，我们想知道比它略大一点和略小一点的元素是谁。TreeMap.lowerKey 与 TreeMap.higherKey 函数就是用来做这样一件事情的。

   了解这一点之后，解法接下来的内容就非常直接了：我们使用动态规划来维护 odd[i] 和 even[i]：从索引 i 出发奇数次跳跃与偶数次跳跃是否能到达数组末尾。

   - 复杂度分析

   时间复杂度：O(NlogN)，其中 NN 是数组 A 的长度。

   空间复杂度：O(N)

```
public int oddEvenJumps(int [] a) { // 写出这种状况：就该强调，选对数据结构，事半功倍!
    int n = a.length, ans = 1; // n-1 算答案里的一个
    boolean [] odd = new boolean [n];
    boolean [] evn = new boolean [n];
    TreeMap<Integer, Integer> m = new TreeMap<>(); // 这里用 treemap 就比用 ArrayDeque 好用
    odd[n-1] = evn[n-1] = true;
    m.put(a[n-1], n-1);
    for (int i = n-2; i >= 0; i--) {
        Integer higher = m.ceilingKey(a[i]); // higherKey() 返回的是键 key
        Integer lower = m.floorKey(a[i]);    // lowerKey()
        if (higher != null) odd[i] = evn[m.get(higher)]; // 仍需取值，而非用键
        if (lower != null) evn[i] = odd[m.get(lower)];
        if (odd[i]) ans++;
        m.put(a[i], i);
    }
    return ans;
}
```

2. 解题思路与分析: 单调栈

   首先，我们可以发现下一步应该跳到哪里只与我们当前的位置与跳跃次数的奇偶性有关系。

   对于每一种状态，接下来可以跳到的状态一定只有一种（或者接下来不能跳跃了）。如果我们使用某种方法知道了不同状态之间的转移关系，我们就可以通过一次简单的遍历解决这个问题了。

   于是，问题就简化为了：从索引 i 进行奇数次跳跃时，下一步应该跳到哪里去（如果有的话）。偶数次跳跃也是类似的。

假设当前是奇数次跳跃，让我们来搞清楚在索引 i 的位置接下来应该跳到哪里去。

我们从小到大考虑数组 A 中的元素。假设当前我们正在考虑 A[j] = v，在我们已经处理过但是还未确定下一步跳跃位置的索引中（也就是 <= v 的那些）进行搜索。如果我们找到了某些已经处理过的值 v0 = A[i] 且 i < j，那么我们就可以知道从索引 i 下一步应该跳跃到索引 j 的位置。

这种朴素的方法有一点点慢，然而我们可以使用一个很常见的技巧单调栈来加速这个过程。

我们在栈中保存所有已经处理过的索引 i ，并且时时刻刻维护这个栈中的元素是递减的。当我们增加一个新的索引 j 的时候，我们弹出栈顶比较小的索引 i < j，并且记录这些索引下一步全都会跳跃到索引 j。

然后，我们就知道所有的 oddnext[i]，也就是位于索引 i 在奇数次跳跃时将会跳到的位置。使用类似的方法，我们也可以求出 evennext[i]。有了这些信息，我们就可以使用动态规划的技巧快速建立所有可达状态。

上面的方法在时间复杂度上并没有优化，而是使用单调栈结构（Monotonic Stack）去解决问题，问题的本质还是求从某一点出发，当进行奇数跳时，他能到达什么位置；然而进行偶数跳时，他又能到达什么位置，将这些信息记录在数组当中。那么问题的关键是如何利用单调栈来求得每一个能到达的相应位置。

```
public int oddEvenJumps(int [] a) {
    int n = a.length, ans = 1;
    ArrayDeque<Integer> s = new ArrayDeque<>();
    Integer [] idx = new Integer [n];       // idx 存储的都是 A 数组的下标
    for (int i = 0; i < n; i++) idx[i] = i;
    Arrays.sort(idx, (x, y)-> a[x] - a[y]); // 将 A 数组的下标按 A 中的元素的大小进行排序
    int [] odIdx = new int [n];
    for (int i = 0; i < n; i++) {
        while (!s.isEmpty() && s.peekLast() < idx[i]) // 利用单调栈获取每一个元素进行奇数跳时所到达的位置
            odIdx[s.pollLast()] = idx[i];
        s.offerLast(idx[i]);
    }
    Arrays.sort(idx, (x, y) -> a[y] - a[x]);
    s.clear();
    int [] enIdx = new int [n];
    for (int i = 0; i < n; i++) {
        while (!s.isEmpty() && s.peekLast() < idx[i])
            enIdx[s.pollLast()] = idx[i];
        s.offerLast(idx[i]);
    }
    boolean [] odd = new boolean [n], evn = new boolean [n];
    odd[n-1] = evn[n-1] = true;
    for (int i = n-2; i >= 0; i--) {
        odd[i] = evn[odIdx[i]];
        evn[i] = odd[enIdx[i]];
        if (odd[i]) ans++;
    }
    return ans;
}
```

2.0.2 1793. Maximum Score of a Good Subarray - Hard

You are given an array of integers nums (0-indexed) and an integer k.

The score of a subarray (i, j) is defined as min(nums[i], nums[i+1], . . . , nums[j]) * (j - i + 1). A good subarray is a subarray where i <= k <= j.

Return the maximum possible score of a good subarray.

1. 解题思路与分析: 左右两个单调栈

```
public int maximumScore(int[] a, int k) {
    int n = a.length, ans = 0;
    List<Integer> li = new ArrayList<>();
    List<Integer> ri = new ArrayList<>();
    li.add(k);
    ri.add(k);
    for (int i = k-1; i >= 0; i--)
        if (a[i] < a[li.get(li.size()-1)]) li.add(i);
    for (int i = k+1; i < n; i++)
        if (a[i] < a[ri.get(ri.size()-1)]) ri.add(i);
    int i = 0, j = 0, nl = li.size(), nr = ri.size();
    while (i < nl || j < nr) {
        int l = (i == nl - 1) ? -1 : li.get(i+1);
        int r = (j == nr - 1) ? n : ri.get(j+1);
        ans = Math.max(ans, Math.min(a[li.get(i)], a[ri.get(j)]) * (r - l - 1));
        if (i == nl - 1 && j == nr - 1) break;
        else if (i == nl - 1) j++;
        else if (j == nr - 1) i++;
        else if (a[l] <= a[r]) j++;
        else i++;
    }
    return ans;
}
```

2. 解题思路与分析: 用真实的栈

- 上面的解法是用数组来代替栈以提高效率，真正用栈的实现如下：

```java
public int maximumScore(int[] a, int k) { // 这个方法速度很慢，上面一个比较快
    ArrayDeque<Integer> s = new ArrayDeque<>();
    int ans = 0;
    for (int i = 0; i < a.length; i++) {
        while (!s.isEmpty() && a[s.peekLast()] > a[i]) {
            int idx = s.pollLast();
            int l = -1;
            if (!s.isEmpty()) l = s.peekLast();
            int cur = (i - l - 1) * a[idx];
            if (l + 1 <= k && i-1 >= k) ans = Math.max(ans, cur);
        }
        s.offerLast(i);
    }
    while (!s.isEmpty()) {
        int idx = s.pollLast();
        int l = -1;
        if (!s.isEmpty()) l = s.peekLast();
        int cur = (a.length - l - 1) * a[idx];
        if (l <= k) ans = Math.max(ans, cur);
    }
    return ans;
}
```

3. 解题思路与分析: two pointers

```java
public int maximumScore(int[] a, int k) {  // O(N) Two Pointers
    int n = a.length, ans = a[k];
    int l = k, r = k, min = a[k];
    while (true) {
        while (r+1 < n && a[r+1] >= min) r++;
        while (l-1 >= 0 && a[l-1] >= min) l--;
        ans = Math.max(ans, min*(r - l + 1));
        if (l == 0 && r == n-1) break;
        if (l == 0) min = a[r+1];
        else if (r == n-1) min = a[l-1];
        else min = Math.max(a[r+1], a[l-1]);
    }
    return ans;
}
```

Chapter 3

PreSum 差分数组

3.1 2381. Shifting Letters II

You are given a string s of lowercase English letters and a 2D integer array shifts where shifts[i] = [starti, endi, directioni]. For every i, shift the characters in s from the index starti to the index endi (inclusive) forward if directioni = 1, or shift the characters backward if directioni = 0.

Shifting a character forward means replacing it with the next letter in the alphabet (wrapping around so that 'z' becomes 'a'). Similarly, shifting a character backward means replacing it with the previous letter in the alphabet (wrapping around so that 'a' becomes 'z').

Return the final string after all such shifts to s are applied.

- 真正做题考试的时候，才能意识到先前没有理解消化透彻
- 输入由小写字母组成的字串 s 和一个 2D 整数阵列 shifts。其中 shifts[i] = [starti, endi, directioni]，代表将 starti 到 endi 范围内的所有字母进行修改。若 directioni 🄵 1，则将字母向前移动一位；若🄵 0，则向后移一位。
- 向前移指的是" a" 变成" b"，而" z" 变回" a"。反之，后移指的是" b" 变成" a"，而" a" 变回" z"。求套用完整个 shifts 之后的字串。
- 有个很关键的点在於：题目只有求最终结果。意味著我们可以先算每个位置的最终移动方向，而不必逐一套用。
- 这时可以使用差分阵列来计算区间的变化量。假设我们一开始有空的差分阵列 D[0,0,0,0]，要使得闭区间 [1,2] 增加 1，可以先使 D[1]+1，而 d[2+1]-1，得到 D=[0,1,0,-1]。在拿来做前缀和，变成 [0,1,1,0]，正好是所求的原阵列。
- 照这个思路做，遍歷所有 shift，若要使字母向前移，则将区间 [a,b] 加 1；否则-1。遍歷完成后做前缀和，得到原本的变化量。
- 最后遍歷字串 s 中的所有字元 c，移动对应的步数后模 26，将值调整於 0~25 之间，转回字元并加入答案中。

```
public String shiftingLetters(String t, int[][] a) {
    int n = t.length(), m = a.length;
    char [] s = t.toCharArray();
    int [] dif = new int [n+1];
    for (int i = 0; i < m; i++) {
        if (a[i][2] == 1) {
            dif[a[i][0]] += 1;
            dif[a[i][1]+1] -= 1;
        } else {
            dif[a[i][0]] -= 1;
            dif[a[i][1]+1] += 1;
        }
// 这里不能真的这么一个一个地来，使用差分数组
        // for (int j = a[i][0]; j <= a[i][1]; j++) {
        //     if (a[i][2] == 1) dif[j]++;
        //     else dif[j]--;
        // }
    }
// change dif [] to be sum []
    for (int i = 1; i < n; i++)
        dif[i] += dif[i-1];
    for (int i = 0; i < n; i++)
        s[i] = (char)('a' + (s[i]-'a' + 26 + dif[i] % 26) % 26);
    return new String(s);
}
```

[1] DEFINITION NOT FOUND.

3.2 1674. Minimum Moves to Make Array Complementary - Medium 差分数组

You are given an integer array nums of even length n and an integer limit. In one move, you can replace any integer from nums with another integer between 1 and limit, inclusive.

The array nums is complementary if for all indices i (0-indexed), nums[i] + nums[n - 1 - i] equals the same number. For example, the array [1,2,3,4] is complementary because for all indices i, nums[i] + nums[n - 1 - i] = 5.

Return the minimum number of moves required to make nums complementary.

1. 解题思路与分析 Sweep Line / Prefix Sum Let a = min(nums[i], nums[n-i-1]), b = max(nums[i], nums[n-i-1])

   The key to this problem is how many moves do we need to make a + b == T.

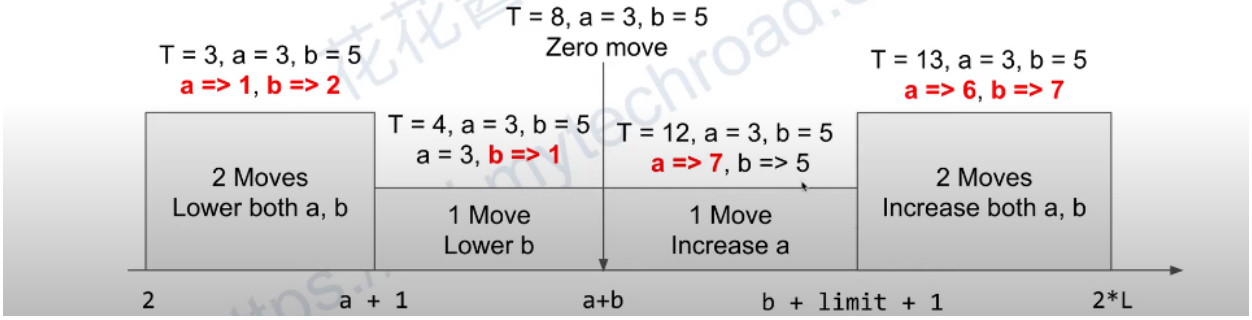
```
if 2 <= T < a + 1, two moves, lower both a and b.
if a +1 <= T < a + b, one move, lower b
if a + b == T, zero move
if a + b + 1 <= T < b + limit + 1, one move, increase a
if b + limit + 1 <= T <= 2*limit, two moves, increase both a and b.
```

- 关键是找到五个关键点，并利用扫描线、差分数组来暴搜所有数对和，并最全局最优解

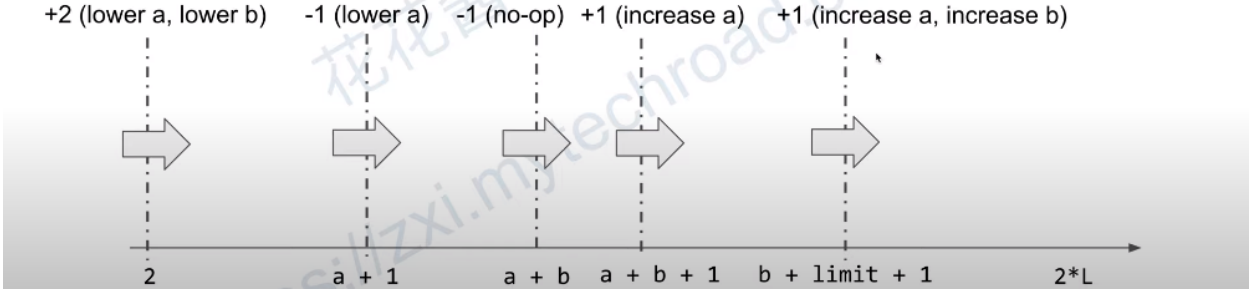
```
To better understand this problem, we can start with the brute force method:
Try all possible pair sum T from 2 to limit * 2, and check how many moves we need.

Let a = min(nums[i], nums[n - i - 1]), b = max(nums[i], nums[n - i - 1])
How many moves do we need to sum up to T? It depends on (a,b,T,limit)

Time complexity: O(limit*n) TLE 72/110 passed, Space complexity: O(1)
```




```
Optimization: If we move T in the x-axis, whenever T cross one boundary, we need to
perform some action.
We can record the delta for each pair at each boundary.
Then we only need to do a prefix sum to determine how many moves we need at the current
pair sum T in O(1). Similar to LeetCode 731. My Calendar II.
Time complexity: O(n + limit), Space complexity: O(limit)
```



- 用一个实例来看。这里强调一下：不可以使用二分搜索，因为这里的答案并不唯一。

a, b	2 / +2	a+1 / -1	a+b / -1	a+b+1 / +1	a+b+L+1 / + 1	delta
1, 3	2 / +2	2 / -1	4 / -1	5 / +1	9 / +1	{2:1,4:-1,5:1,9:1}
2, 4	2 / +2	3 / -1	6 / -1	7 / +1	11 / +1	{2:3,3:-1,4:-1,5:1,6:-1,7:1,9:1,11:1}

T	delta	Prefix sum / ans
2	3	3
3	-1	2
4	-1	1
5	1	2
6	-1	1
7	1	2
9	1	3
11	1	4

[1, 2, 2, 3]

[3, 2, 4, 3]

Since there are at most O(n) unique boundaries, we can sort them to obtain O(nlogn) time complexity and O(n) space complexity. If limit >> n.

```java
public int minMoves(int[] nums, int limit) {
    int n = nums.length;
    int [] delta = new int [2 * limit + 2]; // 差分数组
    for (int i = 0; i < n/2; i++) {
        int a = Math.min(nums[i], nums[n-1-i]);
        int b = Math.max(nums[i], nums[n-1-i]); // 把各区间所需要的操作简化到五行代码中去了
        delta[2] += 2; // dec a, decreasing b [2, a] 区间的和 所需要的变换次数都是 2 次
        --delta[a+1];  // dec a               [a + 1, a+b-1] 区间中的和 所需要变的次数都是 1 次
        --delta[a+b];  // no operations
        ++delta[a+b+1];// inc a               [a+b+1, b+limit] 区间中的和 只需要变动 1 次;
        ++delta[b+limit+1]; // inc a, inc b
    }
    int ans = n, sum = 0;
    for (int t = 2; t < 2 * limit + 2; t++) {
        sum += delta[t]; // 差分数组的前缀和等于 (共需要的操作次数)
        ans = Math.min(ans, sum);
    }
    return ans;
}
```

3.3 798. Smallest Rotation with Highest Score

Hard

308

19

Add to List

Share You are given an array nums. You can rotate it by a non-negative integer k so that the array becomes [nums[k], nums[k + 1], . . . nums[nums.length - 1], nums[2], nums[1], . . . , nums[k-1]]. Afterward, any entries that are less than or equal to their index are worth one point.

For example, if we have nums = [2,4,1,3,0], and we rotate by k = 2, it becomes [1,3,0,2,4]. This is worth 3 points because 1 > 0 [no points], 3 > 1 [no points], 0 <= 2 [one point], 2 <= 3 [one point], 4 <= 4 [one point]. Return the rotation index k that corresponds to the highest score we can achieve if we rotated nums by it. If there are multiple answers, return the smallest such index k.

1. 解题思路与分析

   答案的思路也十分巧妙，并没有采用 brute force 那种直接求每一个 K 值的得分，而是反其道而行之，对于每个数字，探究其跟 K 值之间的联系。首先我们要讨论一下边界情况，那么就是当 A[i] = 0 或 N 的情况，首先如果 A[i] = 0 的话，那么 0 这个数字在任何位置都会小于等于坐标值，所以在任何位置都会得分的，那么其实可以忽略之，因为其不会对最大值产生任何影响，同理，如果 A[i] = N 的时候，由于长度为 N 的数组的坐标值范围是 [0, N-1]，所以数字 N 在任何位置都不得分，同样也不会对最大值产生任何影响，可以忽略之。那么我们关心的数字的范围其实是 [1, N-1]。在这个范围内的数字在旋转数组的过程中，从位置 0 变到 N-1 位置的时候，一定会得分，因为此范围的数字最大就是 N-1。这个一定得的分我们在最后统一加上，基于上面的发现，我们再来分析下题目中的例子 [2, 3, 1, 4, 0]，其中红色数字表示不得分的位置：

```
A:    2   3   1   4   0   (K = 0)
A:    3   1   4   0   2   (K = 1)
A:    1   4   0   2   3   (K = 2)
```

[2]DEFINITION NOT FOUND.

```
A:    4   0   2   3   1   (K = 3)
A:    0   2   3   1   4   (K = 4)
idx:  0   1   2   3   4
```

对于某个数字 A[i]，我们想知道其什么时候能旋转到坐标位置为 A[i] 的地方，这样就可以得分了。比如上面博主标记了红色的数字 3，最开始时的位置为 1，此时是不得分的，我们想知道其什么时候能到位置 3，答案是当 K=3 的时候，其刚好旋转到位置 3，K 再增加的时候，其又开始不得分了。所以这个最后能得分的临界位置是通过 (i - A[i] + N) % N 得到，那么此时如果 K 再增加 1 的话，A[i] 就开始不得分了（如果我们 suppose 每个位置都可以得分，那么不得分的地方就可以当作是失分了），所以我们可以在这个刚好开始不得分的地方标记一下，通过-1 进行标记，这个位置就是 (i - A[i] + 1 + N) % N。我们用一个长度为 N 的 change 数组，对于每个数字，我们都找到其刚好不得分的地方，进行-1 操作，那么此时 change[i] 就表示数组中的数字在 i 位置会不得分的个数，如果我们仔细观察上面红色的数字，可以发现，由于是左移，坐标在不断减小，所以原先失分的地方，在 K+1 的时候还是失分，除非你从开头位置跑到末尾去了，那会得分，所以我们要累加 change 数组，并且 K 每增加 1 的时候，要加上额外的 1，最后 change 数组中最大数字的位置就是要求的 K 值了.

Key point

Don't calculate the score for `K=0`, we don't need it at all.

(I see almost all other solutions did)

The key point is to find out how score changes when `K++`

Time complexity:

"A will have length at most 20000."

I think it means you should find a O(N) solution.

How dosen score change when K++ ?

- **Get point**

  Each time when we rotate, we make index `0` to index `N-1`, then we get one more point.

  We know that for sure, so I don't need to record it.

- **Lose point**

  `(i - A[i] + N) % N` is the value of K making `A[i]'s` index just equal to `A[i]`.

  For example, If `A[6] = 1`, then `K = (6 - A[6]) % 6 = 5` making `A[6]` to `index 1` of new array.

  So when `K=5`, we get this point for `A[6]`

  Then if `K` is bigger when `K = (i - A[i] + 1) % N`, we start to lose this point, making our score -= 1

  All I have done is record the value of `K` for all `A[i]` where we will lose points.

- `A[i]=0`

  Rotation makes no change for it, becasue we alwars have `0 <= index`.

  However, it is covered in "get point" and "lose point".

Explanation of codes

1. Search the index where score decrease and record this changement to a list `change`.
2. A simple for loop to calculate the score for every `K` value.

   `score[K] = score[K-1] + change[K]`

   In my codes I accumulated `changes` so I get the changed score for every `K` value compared to `K=0`
3. Find the index of best score.

时间复杂度：O(n)

空间复杂度：O(n)

```
public int bestRotation(int[] a) {
    int n = a.length, ans = 0;
    int [] change = new int [n];
    for (int i = 0; i < n; ++i) change[(i - a[i] + n + 1) % n] -= 1;
    for (int i = 1; i < n; ++i) {
        change[i] += change[i-1] + 1;
        ans = change[i] > change[ans] ? i : ans;
```

```
        }
        return ans;
    }
```

3.4 1074. Number of Submatrices That Sum to Target - Hard 二维数组转化为一维数组: 数组中 target Sum 的 2D 版、presum 思想的 2d 版

Given a matrix and a target, return the number of non-empty submatrices that sum to target.

A submatrix x1, y1, x2, y2 is the set of all cells matrix[x][y] with x1 <= x <= x2 and y1 <= y <= y2.

Two submatrices (x1, y1, x2, y2) and (x1', y1', x2', y2') are different if they have some coordinate that is different: for example, if x1 != x1'.

本题也需要使用到 presum 思路来解题，只不过这个前缀和 presum 的计算对象是二维数组。对于任意一个点，presum[i][j] 代表了从 $Matrix^{2,2}$ 到 Matrix[i][j] 之间的和。

有了前缀和之后，我们可以将二维数组拆解为多个一维数组，再用一维数组的思路去解题。

拆分数组时可以以列拆分，也可以以行拆，无论如何都可以达到遍历所有子矩阵的效果。本题以列拆分为例，对于任意两列 col1 和 col2，我们可以得到所有行的前缀和

这样，二维数组就转化为了一维数组。接下来只要遍历所有列的组合即可。

```
public int numSubmatrixSumTarget(int[][] matrix, int target) {
    int m = matrix.length, n = matrix[0].length, rowSum = 0;
    int [][] sum = new int [m][n]; // sums of row sum only
    for (int i = 0; i < m; i++) {
        rowSum = 0;
        for (int j = 0; j < n; j++) {
            rowSum += matrix[i][j];
            sum[i][j] = rowSum + (i == 0 ? 0 : sum[i-1][j]);
        }
    }
    int cnt = 0, cur = 0;
    for (int j = 0; j < n; j++)     // col1
        for (int k = j; k < n; k++) {// col2
            Map<Integer, Integer> map = new HashMap<>(); // 用于记录前缀和个数
            for (int i = 0; i < m; i++) {
                cur = sum[i][k] - (j == 0 ? 0 : sum[i][j-1]);
                if (cur == target) ++cnt;
                cnt += map.getOrDefault(cur - target, 0);
                map.put(cur, map.getOrDefault(cur, 0) + 1);
            }
        }
    return cnt;
}
```

- 对 corner case 的处理相对简洁的写法

```
public int numSubmatrixSumTarget(int [][] matrix, int target) {
    int res = 0, m = matrix.length, n = matrix[0].length;
    int [][] sum = new int [m+1][n+1];
    for (int i = 1; i <= m; ++i)
        for (int j = 1; j <= n; ++j) // 对全数组纵横求和
            sum[i][j] = sum[i][j - 1] + sum[i - 1][j] - sum[i - 1][j - 1] + matrix[i - 1][j - 1];
    for (int i = 1; i <= m; ++i)
        for (int j = 1; j <= n; ++j)
            for (int p = 1; p <= i; ++p)
                for (int q = 1; q <= j; ++q) {
                    int t = sum[i][j] - sum[i][q - 1] - sum[p - 1][j] + sum[p - 1][q - 1];
                    if (t == target) ++res;
                }
    return res;
}

public int numSubmatrixSumTarget(int[][] matrix, int target) {
    int m = matrix.length, n = matrix[0].length;
    int [][] sum = new int [m][n+1];                // sums of row sum only
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            sum[i][j+1] = sum[i][j] + matrix[i][j]; // row sums only
    }
    int cnt = 0, cur = 0;
    for (int j = 0; j < n; j++)      // col1
        for (int k = j; k < n; k++) {// col2
            cur = 0;
            Map<Integer, Integer> map = new HashMap<>(); // 用于记录前缀和个数
            map.put(0, 1);
            for (int i = 0; i < m; i++) {
                cur += sum[i][k+1] - sum[i][j];
                cnt += map.getOrDefault(cur - target, 0);
                map.put(cur, map.getOrDefault(cur, 0) + 1);
            }
        }
    return cnt;
}
```

解题思路分析

首先对每一行，计算一个 presum

然后对任意两个列，[j,k], 计算 j 和 k 之间的所有数的和，因为对于每一行，我们已经计算了 presum, 所以，

对于 [j,k] 之间的树，可以通过 presum[i][k] - (j == 0 ? 0 : presum[i][j-1]) 得到某一行在 [j,k] 之间的值

这样，对于任意两列 [j,k] 之间的数，我们可以形成一个临时的一维数组，表示从第 0 行到最后一行的数，

那么，问题就转换成在这个一维数组里面，找一个连续子数组，使得这些数的和是 target，

那么，后面这个问题就转换成 leetcode 560

http://www.noteanddata.com/leetcode-560-Subarray-Sum-Equals-K-java-solution-note.html

所以，这个是 presum 思想的 2d 版, 类似的题目还有 leetcode 304 Range Sum Query 2D - Immutable

```
public int numSubmatrixSumTarget(int [][] matrix, int target) {
    int res = 0, m = matrix.length, n = matrix[0].length;
    for (int i = 0; i < m; ++i)
        for (int j = 1; j < n; ++j)
            matrix[i][j] += matrix[i][j - 1];
    for (int i = 0; i < n; ++i)
        for (int j = i; j < n; ++j) {
            Map<Integer, Integer> cntMap = new HashMap<>();
            cntMap.put(0, 1);
            int cur = 0;
            for (int k = 0; k < m; ++k) {
                cur += matrix[k][j] - (i > 0 ? matrix[k][i - 1] : 0);
                res += cntMap.getOrDefault(cur-target, 0);
                cntMap.put(cur, cntMap.getOrDefault(cur, 0) + 1);
            }
        }
    return res;
}

private int cntTarget(int [] arr, int target) { // 比较一下与上面解法、写法的不同!!!
    int cnt = 0, sum = 0;
    Map<Integer, Integer> cntMap = new HashMap<>();
    cntMap.put(0, 1);
    for (int i = 0; i < arr.length; i++) {
        sum += arr[i];
        cnt += cntMap.getOrDefault(sum - target, 0);
        cntMap.compute(sum, (k, v) -> { //
                if (v == null) v = 0;
                return v+1;
            });
        return cnt;
    }
}
public int numSubmatrixSumTarget(int[][] matrix, int target) {
    int m = matrix.length, n = matrix[0].length, cur = 0;
    int [][] sum = new int [m][n]; // sums of row sum only
    for (int i = 0; i < m; i++) {
        cur = 0;
        for (int j = 0; j < n; j++) {
            cur += matrix[i][j];
            sum[i][j] = cur;
        }
    }
    int cnt = 0;
    for (int j = 0; j < n; j++) {
        for (int k = j; k < n; k++) {
            int [] tmp = new int [m];
            for (int i = 0; i < m; i++)
                tmp[i] = sum[i][k] - (j == 0 ? 0 : sum[i][j-1]);
            int oneCnt = cntTarget(tmp, target);
            cnt += oneCnt;
        }
    }
    return cnt;
}
```

3.4.1 Java O(NlgN) optimized Brute Force with Fenwick Tree O(NlogN)

我们假设 A 的大小为 n，如果采用暴力法逐个测试，则时间复杂度为 O(n^2)，应该过不了大数据测试。

我采取的方法是：首先计算使得每个元素 A[i] 要符合条件，需要 rotate 的 K 的集合，用线段表示；然后再扫描一遍，求出这些线段中重合最多的点，那么这个点对应的 rotate 次数就是题目所要计算的 K。对应 A[i] 来讲，如果 A[i] <= i，那么它向左移动到 j 也可能维持 A[i] <= j，所以我们计算出此时它向左移动的合法区间 [0, i - A[i]]。那么 A[i] 向右移动的合法区间是多少呢？我们知道它向右移动最多移动到 n - 1，即移动 n - 1 - i 步；而最少需要移动 max(1, A[i] - i) 步，其中 1 表示 A[i] <= 的情况。那么如果这个区间合法，就可以同样构成了一个合法的移动区间 [i + 1, n - max(1, A[i] - i)]。

得到多个线段构成的合法移动区间之后，我们的任务就是求出这些区间的最大重合点。首先对 segment 中的各个点进行排序，然后采用扫描线的方法计算最大最大重合处。为了便于区分某个点是起点还是终点，我们定义一个 pair<int, bool> 来表示点，并且让起点的 bool 值为 false，终点的 bool 值为 true，这样就可以在扫描到某个点之后，先处理起点，再处理终点。

由于每个 A[i] 最多对应 2 个合法移动区间，所以 segments 大小也是 O(n) 量级的。这样可以得知，本算法的时间复杂度是 O(nlogn)，空间复杂度是 O(n)。

3.4. 1074. NUMBER OF SUBMATRICES THAT SUM TO TARGET - HARD 二维数组转化为一维数组: 数组中 TARGET SUM 的 2D 版

https://www.codeleading.com/article/62063257456/

- 不是我的代码风格，需要再学习一下

```
public int bestRotation(int[] A) {
    int LEN = A.length;
    int score, ans, ansInd, k; score = k = ans = ansInd = 0;
    for(int i = 0; i < LEN; i++) A[i] -= i;
    int[] bit = new int[3 * LEN + 1];
    for(int i = k; i < LEN; i++) bitAdd(bit, 3 * LEN + 1, vToIndex(A[i], LEN), 1);
    while (k < LEN) {
        score = bitPreSum(bit, vToIndex(-k, LEN));
        if(score > ans) {
            ansInd = k;
            ans = score;
        }
        bitAdd(bit, 3 * LEN + 1, vToIndex(A[k], LEN), -1);
        bitAdd(bit, 3 * LEN + 1, vToIndex(A[k] - LEN, LEN), 1);
        k++;
    }
    return ansInd;
}
void bitAdd(int[] bit, int bLEN, int ind, int delta) {
    for(; ind < bLEN; ind = (ind | (ind + 1))) {
        bit[ind] += delta;
    }
}
int bitPreSum(int[] bit, int ind) {
    int ans = 0;
    for(; ind >= 0; ind = (ind & (ind + 1)) - 1) {
        ans += bit[ind];
    }
    return ans;
}
int vToIndex(int v, int LEN) {
    return v + 2 * LEN;
}
```

Chapter 4

Greedy

4.1 1585. Check If String Is Transformable With Substring Sort Operations - Hard

Given two strings s and t, you want to transform string s into string t using the following operation any number of times:

Choose a non-empty substring in s and sort it in-place so the characters are in ascending order. For example, applying the operation on the underlined substring in "14234" results in "12344".

Return true if it is possible to transform string s into string t. Otherwise, return false.

A substring is a contiguous sequence of characters within a string.

```
public boolean isTransformable(String s, String t) {
    ArrayDeque<Integer> [] q = new ArrayDeque[10];
    for (int i = 0; i < 10; i++)
        q[i] = new ArrayDeque<Integer>();
    for (int i = 0; i < s.length(); i++)
        q[s.charAt(i)-'0'].offerLast(i);
    for (char c : t.toCharArray()) {
        int d = c - '0';
        if (q[d].isEmpty()) return false;
        for (int i = 0; i < d; i++)
            if (!q[i].isEmpty() && q[i].peekFirst() < q[d].peekFirst())
                return false;
        q[d].pollFirst();
    }
    return true;
}
```

4.2 1330. Reverse Subarray To Maximize Array Value - Hard

You are given an integer array nums. The value of this array is defined as the sum of |nums[i]-nums[i+1]| for all 0 <= i < nums.length-1.

You are allowed to select any subarray of the given array and reverse it. You can perform this operation only once.

Find maximum possible value of the final array.

解题思路

给出一个整数数组，定义这个数组的和为相邻元素之差的和。现在允许将数组中的某个子数组进行整体翻转，然后计算对应的和。返回数组可得到的最大和。

这个题分析一下可知。有如下几种情况。

如果最优的最大和是不翻转的情况，那么就等于是翻转整个数组。

如果最优的情况下，翻转子串的边界是数组边界，那么用O(n)的时间遍历两次，分别找出是右边界和左边界的情况就可以了。

如果最优的情况下，翻转子串的边界在数组内部，这是最复杂的情况，那么该如何考虑呢？假设我们已经计算出来数组在不翻转的情况下的和S，那么翻转之后其实只是改变了子串和整体数组的交界处的情况，其它地方均没有改变。因此假设边界左端点的值是b，挨着的前一个是a，右端点是c，挨着的后一个是d。那么变换之后的结果其实就是S+abs(d-b)+abs(c-a)-abs(b-a)-abs(d-c)=S+x。只要x大于0，就意味着增量为正，变换会导致结果优化。我们再假设a<=b, c<=d, 那么增量就是2*(c-b). 而分析可知我们要让c尽可能大，b尽可能小。因此我们用两个O(n)的时间。第一次找b，怎么找呢？就是每两个相邻的，选大的那一个，然后在所有对中选对应b最小的那一对。第二次找c就是每两个相邻的，选小的那个，然后在所有的对中选最大的c的那一对。

这里假设了abcd的关系，其实也可以假设其他情况，最后都是一样的。

时空复杂度

代码如下，时间上，O(n)，空间上，O(1)。

```
public int maxValueAfterReverse(int[] a) {
    int n = a.length, ans = 0, sum = 0;
    if (n == 1) return 0;
    for (int i = 0; i < n-1; i++) // 不发生交换情况下的解
        sum += Math.abs(a[i] - a[i+1]);
    for (int i = 0; i < n-1; i++) // 发生交换、但交换的左端点为数组头的 case, 交换的区间左端点是数组的左边界
        ans = Math.max(ans, sum + Math.abs(a[i+1]-a[0]) - Math.abs(a[i+1]-a[i]));
    for (int i = n-1; i > 0; i--) // 发生交换、但交换的左端点为数组尾的 case, 交换的区间右端点是数组的右边界
        ans = Math.max(ans, sum + Math.abs(a[n-1] - a[i-1]) - Math.abs(a[i] - a[i-1]));
    // 交换区间左右端点在数组内部
    int l = 1; // 1. 找出左端点 b: 每两个相邻的，选最大的那一个数；在所有相邻对中，选最小的
    for (int i = 2; i < n; i++)
        if (Math.max(a[i], a[i-1]) < Math.max(a[l], a[l-1])) l = i;
    int r = 0; // 2. 找出右端点 c: 每两个相邻的，先最小的那一个数；在所有相邻对中，选最大的
    for (int i = 1; i < n-1; i++)
        if (Math.min(a[i], a[i+1]) > Math.min(a[r], a[r+1])) r = i;
    ans = Math.max(ans, sum + 2 * (Math.min(a[r], a[r+1]) - Math.max(a[l], a[l-1])));
    return ans;
}
```

Chapter 5

others

5.1 Predict the Winner

You are given an integer array nums. Two players are playing a game with this array: player 1 and player 2. Player 1 and player 2 take turns, with player 1 starting first. Both players start the game with a score of 0. At each turn, the player takes one of the numbers from either end of the array (i.e., nums^2 or nums[nums.length - 1]) which reduces the size of the array by 1. The player adds the chosen number to their score. The game ends when there are no more elements in the array. Return true if Player 1 can win the game. If the scores of both players are equal, then player 1 is still the winner, and you should also return true. You may assume that both players are playing optimally.

```
private int helper( int [] arr, int i, int j) {
    if (i == j) return arr[i];
    else return Math.max(arr[i] - helper(arr, i+1, j), arr[j] - helper(arr, i, j-1));
}
public boolean PredictTheWinner(int[] nums) {
    int n = nums.length;
    if (n == 1) return true;
    return helper(nums, 0, n-1) >= 0;
}
```

5.2 Rectangle Area II

We are given a list of (axis-aligned) rectangles. Each rectangle[i] = [xi1, yi1, xi2, yi2] , where (xi1, yi1) are the coordinates of the bottom-left corner, and (xi2, yi2) are the coordinates of the top-right corner of the ith rectangle. Find the total area covered by all rectangles in the plane. Since the answer may be too large, return it modulo 109 + 7.

```
public int rectangleArea(int[][] rectangles) {
    ll = new ArrayList<>();
    long ans = 0;
    for (int [] r : rectangles)
        addRect(r, 0);
    for (int [] r : ll)
        ans = (ans + (long)(r[2] - r[0]) * (long)(r[3] - r[1])) % mod;
    return (int)ans;
}
static final int mod = (int)1e9 + 7;
List<int []> ll;
void addRect(int [] a, int i) { // i: idx
    if (i >= ll.size()) {
        ll.add(a);
        return ;
    }
    int [] r = ll.get(i);
    if (a[2] <= r[0] || a[0] >= r[2] || a[1] >= r[3] || a[3] <= r[1]) { // 被添加的，与现遍历的矩形，完全没有交集，直接往后遍历
        addRect(a, i+1);
        return ;
    } // 找出所有有交集的、交集部分——与现遍历矩形相比，多出来的部分，的四个顶点，再往后遍历
    if (a[0] < r[0]) // 左
        addRect(new int [] {a[0], a[1], r[0], a[3]}, i+1);
    if (a[2] > r[2]) // 右
        addRect(new int [] {r[2], a[1], a[2], a[3]}, i+1);
    if (a[1] < r[1]) // 下: 新增矩形 下侧 也可以有 交叠: 注意左侧、右侧前面已经加进去了，现在只加中间部分，不要重复计算
        addRect(new int [] {Math.max(a[0], r[0]), a[1], Math.min(a[2], r[2]), r[1]}, i+1);
    if (a[3] > r[3]) // 上: 新增矩形 上侧 也可以有 交叠: 注意左侧、右侧、下侧 前面已经加进去了，现在只加中间部分，不要重复计算
        addRect(new int [] {Math.max(a[0], r[0]), r[3], Math.min(a[2], r[2]), a[3]}, i+1);
}
```

5.3 Construct Binary Tree from Preorder and Postorder Traversal

Given two integer arrays, preorder and postorder where preorder is the preorder traversal of a binary tree of distinct values and postorder is the postorder traversal of the same tree, reconstruct and return the binary tree. If there exist multiple answers, you can return any of them.

```
public TreeNode constructFromPrePost(int[] preorder, int[] postorder) {
    int n = preorder.length;
    TreeNode r = new TreeNode(preorder[0]);
    if (n == 1) return r;
    Stack<TreeNode> s = new Stack<>();
    s.push(r);
    int idx = 0;
    for (int i = 1; i < n; i++) {
        TreeNode cur = new TreeNode(preorder[i]);
        if (s.peek().left == null) s.peek().left = cur;
        else s.peek().right = cur;
        s.push(cur);
        while (idx < n && postorder[idx] == s.peek().val) {
            s.pop();
            ++idx;
        }
    }
    return r;
}
```

5.4 Path Sum III

Given the root of a binary tree and an integer targetSum, return the number of paths where the sum of the values along the path equals targetSum. The path does not need to start or end at the root or a leaf, but it must go downwards (i.e., traveling only from parent nodes to child nodes).

```
private int solve(TreeNode r, int t, int value) {
    if (r == null) return 0;
    if (value + r.val == t)
        return 1 + solve(r.left, 0, 0) + solve(r.right, 0, 0);
    return solve(r.left, t, value + r.val) + solve(r.right, t, value + r.val);
}
public int pathSum(TreeNode root, int targetSum) {
    if (root == null) return 0;
    return solve(root, targetSum, 0) + pathSum(root.left, targetSum) + pathSum(root.right, targetSum);
}
```

5.5 Critical Connections in a Network

- There are n servers numbered from 0 to n - 1 connected by undirected server-to-server connections forming a network where connections[i] = [ai, bi] represents a connection between servers ai and bi. Any server can reach other servers directly or indirectly through the network.
- A critical connection is a connection that, if removed, will make some servers unable to reach some other server.
- Return all critical connections in the network in any order.

```
static class Eg {
    int u, v, next;
    // int w;
    boolean cut;
    // int num;
}
public Eg[] egs;
public int cnt;
public int [] fir; // 边的出发点
int [] low;
int [] dfn;
int recdfn;
void tarjanAddEg(int u, int v, int w) {
    egs[cnt] = new Eg();
    egs[cnt].u = u;
    egs[cnt].v = v;
    // egs[cnt].w = w;
    egs[cnt].cut = false;
    // egs[cnt].num = 0;
    egs[cnt].next = fir[u]; // ?
    fir[u] = cnt++;         // ?
}
private void initTarjan(int nodeSize, int edgeSize) {
    cnt = 0;
    egs = new Eg [edgeSize];
```

```
    low = new int [nodeSize];
    dfn = new int [nodeSize];
    fir = new int [edgeSize];
    Arrays.fill(fir, -1);
}
private void tarjan(int u, int fa) { // fa: father
    low[u] = ++recdfn;
    dfn[u] = recdfn;
    int have = 0;
    for (int i = fir[u]; i != -1; i = egs[i].next) {
        int v = egs[i].v;
        if (have == 0 && v == fa) { // 走过你来时的路
            have++;
            continue;
        }
        if (dfn[v] == 0) { // dfs 过程中还未经过该点
            tarjan(v, u);
            low[u] = Math.min(low[u], low[v]);
            if (dfn[u] < low[v]) { // 连通世外桃源与外界的路
                // 当 dfn[x] < low[y] 的时候:
                // --- 我们发现从 yy 节点出发, 在不经过 (x,y)(x,y) 的前提下, 不管走哪一条边, 我们都无法抵达 xx 节点, 或者比 xx 节点更早出现的节点
                // --- 此时我们发现 yy 所在的子树似乎形成了一个封闭圈, 那么 (x,y)(x,y) 自然也就是桥了.
                egs[i].cut = true;
                egs[i^1].cut = true; // ???
            }
        } else {
            low[u] = Math.min(low[u], dfn[v]); // 取已访问的节点的 dfs 序的最小值
        }
    }
}
private boolean findEdgeCut(int l, int r) {
    Arrays.fill(low, 0);
    Arrays.fill(dfn, 0);
    recdfn = 0;
    tarjan(l, l);
    for (int i = l; i <= r; i++) {
        if (dfn[i] == 0) return false;
    }
    return true;
}
public List<List<Integer>> criticalConnections(int n, List<List<Integer>> connections) {
    initTarjan(n, connections.size()*2);
    for (List<Integer> eg : connections) {
        tarjanAddEg(eg.get(0), eg.get(1), 1);
        tarjanAddEg(eg.get(1), eg.get(0), 1);
    }
    // boolean ans = findEdgeCut(0, n-1);
    Arrays.fill(low, 0);
    Arrays.fill(dfn, 0);
    recdfn = 0;
    tarjan(0, 0);
    List<List<Integer>> res = new ArrayList<>();
    int l = connections.size();
    for (int i = 0; i < l * 2; i += 2) { // i += 2 skipped egs[i^1] ?
        Eg eg = egs[i];
        if (eg != null && eg.cut) {
            List<Integer> t = new ArrayList<>();
            t.add(eg.u);
            t.add(eg.v);
            res.add(t);
        }
    }
    return res;
}
```

5.6 891. Sum of Subsequence Widths - Hard 考 sorting 和对 subsequence 的理解

The width of a sequence is the difference between the maximum and minimum elements in the sequence.

Given an array of integers nums, return the sum of the widths of all the non-empty subsequences of nums. Since the answer may be very large, return it modulo 109 + 7.

A subsequence is a sequence that can be derived from an array by deleting some or no elements without changing the order of the remaining elements. For example, [3,6,2,7] is a subsequence of the array [0,3,1,6,2,2,7].

1. 解题思路与分析

    - 这道题的最优解法相当的 tricky，基本有点脑筋急转弯的感觉了。在解题之前，我们首先要知道的是一个长度为 n 的数组，共有多少个子序列，如果算上空集的话，共有 2^n 个。

    那么在给数组排序之后，对于其中任意一个数字 A[i]，其前面共有 i 个数是小于等于 A[i] 的，这 i 个数字共有 2^i 个子序列，它们加上 A[i] 都可以组成一个新的非空子序列，并且 A[i] 是这里面最大的数字，那么在宽度计算的时候，就要加

上 A[i] x (2^i)，

同理，A[i] 后面还有 n-1-i 个数字是大于等于它的，后面可以形成 2^(n-1-i) 个子序列，每个加上 A[i] 就都是一个新的非空子序列，同时 A[i] 是这些子序列中最小的一个，那么结果中就要减去 A[i] x (2 ^ (n-1-i))。对于每个数字都这么计算一下，就是最终要求的所有子序列的宽度之和了。

可能你会怀疑虽然加上了 A[i] 前面 2^i 个子序列的最大值，那些子序列的最小值减去了么？其实是减去了的，虽然不是在遍历 A[i] 的时候减去，在遍历之前的数字时已经将所有该数字是子序列最小值的情况减去了，同理，A[i] 后面的那些 2^(n-1-i) 个子序列的最大值也是在遍历到的时候才加上的，所以不会漏掉任何一个数字。

在写代码的时候有几点需要注意的地方，首先，结果 res 要定义为 long 型，因为虽然每次会对 1e9+7 取余，但是不能保证不会在取余之前就已经整型溢出，所以要定义为长整型。

其次，不能直接算 2^i 和 2^(n-1-i)，很容易溢出，即便是长整型，也有可能溢出。那么解决方案就是，在累加 i 的同时，每次都乘以个 2，那么遍历到 i 的时候，也就乘到 2^i 了，防止溢出的诀窍就是每次乘以 2 之后就立马对 1e9+7 取余，这样就避免了指数溢出，同时又不影响结果。

最后，由于这种机制下的 2^i 和 2^(n-1-i) 不方便同时计算，这里又用了一个 trick，就是将 A[i] x (2^(n-1-i)) 转换为了 A[n-1-i] x 2^i，其实二者最终的累加和是相等的：

```
sum(A[i] * 2^(n-1-i)) = A[0]*2^(n-1) + A[1]*2^(n-2) + A[2]*2^(n-3) + ... + A[n-1]*2^0
sum(A[n-1 - i] * 2^i) = A[n-1]*2^0 + A[n-2]*2^1 + ... + A[1]*2^(n-2) + A[0]*2^(n-1)
```

```
public int sumSubseqWidths(int[] a) {
    long mod = (int)1e9 + 7, c = 1;
    long ans = 0;
    Arrays.sort(a);
    for (int i = 0; i < a.length; i++) {
        ans = (ans + (long)a[i] * c  - a[a.length-1-i] * c) % mod;
        c = (c << 1) % mod;
    }
    return (int)ans;
}
```

5.7 335. Self Crossing

Hard

225

433

Add to List

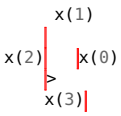
Share You are given an array of integers distance.

You start at point (0,0) on an X-Y plane and you move distance[2] meters to the north, then distance[1] meters to the west, distance[1] meters to the south, distance[2] meters to the east, and so on. In other words, after each move, your direction changes counter-clockwise.

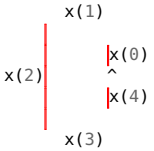
Return true if your path crosses itself, and false if it does not.

1. 解题思路与分析

   - 参考的网上大神们的解法，实际上相交的情况只有以下三种情况：

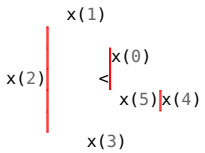


第一类是第四条边和第一条边相交的情况，需要满足的条件是第一条边大于等于第三条边，第四条边大于等于第二条边。同样适用于第五条边和第二条边相交，第六条边和第三条边相交等等，依次向后类推的情况...



第二类是第五条边和第一条边重合相交的情况，需要满足的条件是第二条边和第四条边相等，第五条边大于等于第三条边和第一条边的差值，同样适用于第六条边和第二条边重合相交的情况等等依次向后类推...

[1]DEFINITION NOT FOUND.
[2]DEFINITION NOT FOUND.



第三类是第六条边和第一条边相交的情况，需要满足的条件是第四条边大于等于第二条边，第三条边大于等于第五条边，第五条边大于等于第三条边和第一条边的差值，第六条边大于等于第四条边和第二条边的差值，同样适用于第七条边和第二条边相交的情况等等依次向后类推...

```
public boolean isSelfCrossing(int [] d) {
    int n = d.length;
    if (n < 4) return false;
    for (int i = 3; i < n; i++) { // 4 >= 2 && 1 >= 3
        // if (i % 3 == 0 && d[i] >= d[i-2] && d[i-3] >= d[i-1]) return true;
        // else if (i % 4 == 0 && d[i-1] == d[i-3] && d[i] >= d[i-2] - d[i-4]) return true; // 2 == 4 && 5 >= 3-1
        // else if (i % 5 == 0 && d[i-2] >= d[i-4] && d[i-3] >= d[i-1] && d[i-1] >= d[i-3] - d[i-5] && d[i] >= d[i-2] - d[i-4])
            if (d[i] >= d[i-2] && d[i-3] >= d[i-1]) return true;
            else if (i >= 4 && d[i-1] == d[i-3] && d[i] >= d[i-2] - d[i-4]) return true; // 2 == 4 && 5 >= 3-1
            else if (i >= 5 && d[i-2] >= d[i-4] && d[i-3] >= d[i-1] && d[i-1] >= d[i-3] - d[i-5] && d[i] >= d[i-2] - d[i-4])
            return true;
        // else if (i % 6 == 0 && d[i-4] + d[i] >= d[i-2] && d[i-1] <= d[i-3] && d[i-5] + d[i-1] >= d[i-3]) return true; // 这个条件不对
    }
    return false;
}
```

5.8 391. Perfect Rectangle - Hard

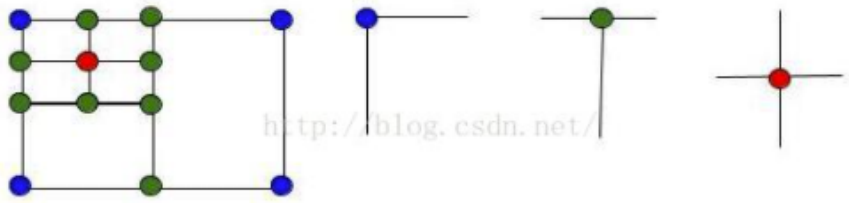
Given an array rectangles where rectangles[i] = [xi, yi, ai, bi] represents an axis-aligned rectangle. The bottom-left point of the rectangle is (xi, yi) and the top-right point of it is (ai, bi).

Return true if all the rectangles together form an exact cover of a rectangular region.

1. 解题思路与分析: 数顶点个数：掩码 O(N)

思路：一种naive的做法是先计算总共小矩阵的面积和四个边界的面积，如果不相等则返回false．否则就要计算是否有重叠，计算重叠的方式是两两检查即可．这样时间复杂度是O(n^2)，并不能过掉所有数据．所以我们需要一种更聪明的做法．

如果一堆小矩形要组成一个完美的大矩形需要满足以下条件：



1．蓝色的点一个位置只能出现一次，并且只有四个这样的点

2．绿色的点一个位置只能出现两次

3．红色的点一个位置只能出现四次

也就是只能存在以上类型的点，并且我们给一个矩形的四个角编号分别为１，２，３，４，那么出现在同一个位置的点必须为一个矩阵不同编号的点．如此条件都满足即可构成一个完美矩形．

算法实现步骤如下：

1．先遍历每一个矩阵，对每一个点进行处理，对于每一个位置的点用hash表来存储这个位置出现了几个corner，并且为了区分出现在这个位置的点是什么编号，我们可以用一个掩码来表示这个点的编号，这样也好判断是否这个位置出现了重复的编号的corner.

2. 对矩阵做好处理之后接下来就好遍历hash表查看对于每个位置来说是否满足以上规定的点的形式．任意一个点不满足条件即可返回false．

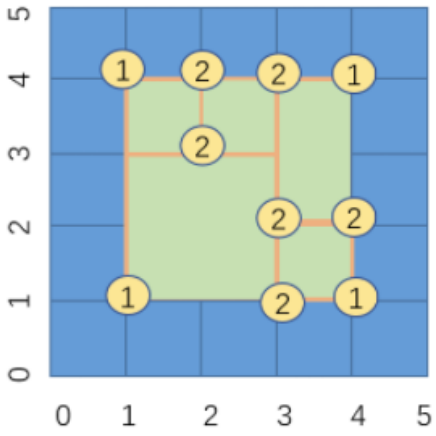
在实现的过程中为了简化代码用了一些技巧，比如为了在hash表中保存一个坐标，可以用字符串形式来保存．还有判断一个数是否是２的倍数可以用val&(-val)是否为０来判断．

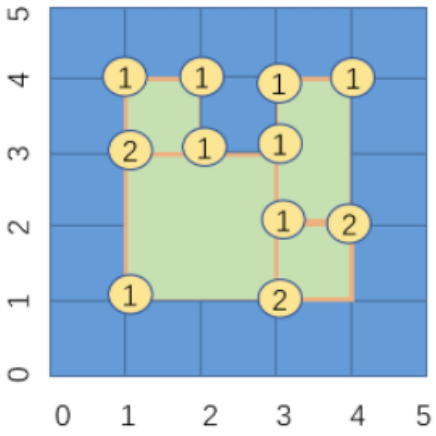
这样时间复杂度即可降为O(n)

```
public boolean isRectangleCover(int[][] arr) {
    Map<String, Integer> m = new HashMap<>();
    for (int [] a : arr)
        for (int i = 0; i < 4; i++) { // [0 1 2 3] 四种类型的顶点，第个位置每个类型最多出现一次
            String cur = a[i / 2 * 2] + "-" + a[i % 2 * 2 + 1];        // 记录从左下角开始逆时针的四个顶点 [0 1 2 3]: 左下，右下，右上，左上
            if ((m.getOrDefault(cur, 0) & (1 << i)) > 0) return false; // 同一形状、同一个角型出现了重叠
            m.put(cur, m.getOrDefault(cur, 0) | (1 << i));
        }
    int cnt = 0;
    for (Map.Entry<String, Integer> en : m.entrySet()) {
        int v = en.getValue();
        if ((v & (v-1)) == 0 && cnt++ > 4) return false; // 只出现一次的顶点最多只有四个
        // if ((v & (v-1)) > 0 && !(v == 15 || v == 10 || v == 12 || v == 5 || v == 3)) // 是 test case 太弱吗？检测不出来？
        if ((v & (v-1)) > 0 && !(v == 15 || v == 12 || v == 10 || v == 9 || v == 5 || v == 3)) // 所有合法的子集除外
            return false;
    }
    return true; // 所有的点都合法了，就不用检测面积了
}
```

2. 解题思路与分析

因此，我们还需要从点的角度进行进一步判断。下面给出一个合法的例子如左图所示，不合法的例子如右图所示，同时用数字标记了所有小矩形包包含的顶点出现的次数。可以发现，合法的例子中，出现次数为1的顶点就是最后完美矩形的四个顶点，其他的顶点都出现了两次。而不合法的例子中，出现次数为1的顶点不只是完美矩形的四个顶点。





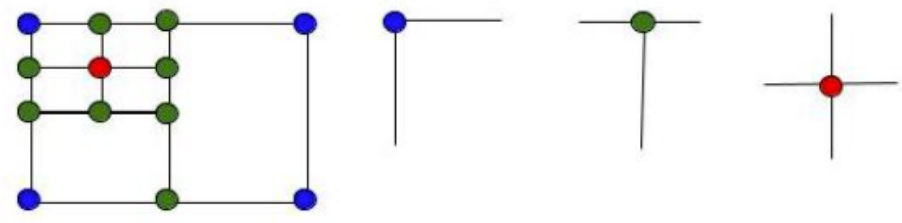
因此，我们可以在遍历小矩形的同时记录顶点出现的情况。如果当前顶点没有遍历过，则将其加入到集合中，否则将其从集合中删除。如果最后集合中只包含可能的完美矩形的四个顶点，那么说明例子是合法的，否则说明无法构成完美矩形。

总结，如果解该题需要遍历所有的小矩形，遍历过程中需要等到完美矩形的左下角和右上角坐标、矩形面积之和，以及顶点的出现情况，最后，判断小矩形面积之和和完美矩形面积之和是否相等，集合中是否只包含4个顶点且是完美矩形的四个顶点，只有这两个条件同时满足才能构成完美矩形。

```
public boolean isRectangleCover(int[][] arr) {
    Set<String> s = new HashSet<>();
    int i = Integer.MAX_VALUE, j = Integer.MAX_VALUE, x = Integer.MIN_VALUE, y = Integer.MIN_VALUE, sum = 0;
    for (int [] a : arr) {
        i = Math.min(i, a[0]);
        j = Math.min(j, a[1]);
        x = Math.max(x, a[2]);
        y = Math.max(y, a[3]);
        sum += (a[2] - a[0]) * (a[3] - a[1]);
        String bl = a[0] + "-" + a[1], tl = a[2] + "-" + a[1];
        String br = a[0] + "-" + a[3], tr = a[2] + "-" + a[3];
        if (!s.contains(bl)) s.add(bl); else s.remove(bl);
        if (!s.contains(br)) s.add(br); else s.remove(br);
        if (!s.contains(tl)) s.add(tl); else s.remove(tl);
        if (!s.contains(tr)) s.add(tr); else s.remove(tr);
    }
    String a = i + "-" + j, b = i + "-" + y;
    String c = x + "-" + j, d = x + "-" + y;
    if (!s.contains(a) || !s.contains(b) || !s.contains(c) || !s.contains(d) || s.size() != 4)
        return false;
    return sum == (x - i) * (y - j);
}
```

3. 解题思路与分析: 这个就当帮上面第一个方法再理解消化一下 c++

博主能力有限，只能去论坛中找各位大神的解法，发现下面两种方法比较fancy，也比较好理解。首先来看第一种方法，这种方法的设计思路很巧妙，利用了mask，也就是位操作Bit Manipulation的一些技巧，下面这张图来自这个帖子：



所有的矩形的四个顶点只会有下面蓝，绿，红三种情况，其中蓝表示该顶点周围没有其他矩形，T型的绿点表示两个矩形并排相邻，红点表示四个矩形相邻，那么在一个完美矩形中，蓝色的点只能有四个，这是个很重要的判断条件。我们再来看矩形的四个顶点，我们按照左下，左上，右上，右下的顺序来给顶点标号为1，2，4，8，为啥不是1，2，3，4呢，我们注意它们的二进制1(0001)，2(0010)，4(0100)，8(1000)，这样便于我们与和或的操作，我们还需要知道的一个判定条件是，当一个点是某一个矩形的左下顶点时，这个点就不能是其他矩形的左下顶点了，这个条件对于这四种顶点都要成立，那么对于每一个点，如果它是某个矩形的四个顶点之一，我们记录下来，如果在别的矩形中它又是相同的顶点，那么直接返回false即可，这样就体现了我们标记为1，2，4，8的好处，我们可以按位检查1。如果每个点的属性没有冲突，那么我们来验证每个点的mask是否合理，通过上面的分析，我们知道每个点只能是蓝，绿，红三种情况的一种，其中蓝的情况是mask的四位中只有一个1，分别就是1(0001)，2(0010)，4(0100)，8(1000)，而且蓝点只能有四个；那么对于T型的绿点，mask的四位中有两个1，那么就有六种情况，分别是12(1100), 10(1010), 9(1001), 6(0110), 5(0101), 3(0011)；而对于红点，mask的四位都是1，只有一种情况15(1111)，那么我们可以通过直接找mask是1，2，4，8的个数，也可以间接通过找不是绿点和红点的个数，看是否是四个。最后一个判定条件是每个矩形面积累加和要等于最后的大矩形的面积，那么大矩形的面积我们通过计算最小左下点和最大右上点来计算出来即可， 参见代码如下：

```
bool isRectangleCover(vector<vector<int>>& rectangles) {
    unordered_map<string, int> m;
    int min_x = INT_MAX, min_y = INT_MAX, max_x = INT_MIN, max_y = INT_MIN, area = 0, cnt = 0;
    for (auto rect : rectangles) {
        min_x = min(min_x, rect[0]);
        min_y = min(min_y, rect[1]);
        max_x = max(max_x, rect[2]);
        max_y = max(max_y, rect[3]);
        area += (rect[2] - rect[0]) * (rect[3] - rect[1]);
        if (!isValid(m, to_string(rect[0]) + "_" + to_string(rect[1]), 1)) return false; // bottom-left
        if (!isValid(m, to_string(rect[0]) + "_" + to_string(rect[3]), 2)) return false; // top-left
        if (!isValid(m, to_string(rect[2]) + "_" + to_string(rect[3]), 4)) return false; // top-right
        if (!isValid(m, to_string(rect[2]) + "_" + to_string(rect[1]), 8)) return false; // bottom-right
    }
    for (auto it = m.begin(); it != m.end(); ++it) {
        int t = it->second;
        if (t != 15 && t != 12 && t != 10 && t != 9 && t != 6 && t != 5 && t!= 3) {
            ++cnt;
        }
    }
    return cnt == 4 && area == (max_x - min_x) * (max_y - min_y);
}
bool isValid(unordered_map<string, int>& m, string corner, int type) {
    int& val = m[corner];
    if (val & type) return false;
    val |= type;
    return true;
}
```

Chapter 6

排列与组合

6.0.1 1611. Minimum One Bit Operations to Make Integers Zero - Hard

Given an integer n, you must transform it into 0 using the following operations any number of times:

Change the rightmost (0th) bit in the binary representation of n. Change the ith bit in the binary representation of n if the (i-1)th bit is set to 1 and the (i-2)th through 0th bits are set to 0. Return the minimum number of operations to transform n into 0.

1. 解题思路与分析

给定一个正整数n，将其视为二进制数。允许进行两种操作：
1、将其最低位取反；
2、如果左起第i位是1，并且第$i-1$位一直到第0位都是0，那就可以将第$i+1$位取反。
问要将其变为0至少需要多少步。

首先，对于任意n，它一定要先变成形如$1100...0$的数，才能继续变为0。这是因为要将n变为0，必须其中某一步是将最高位变为0，而最高位变为0必须要得到形如$1100...0$的数才能做。我们来证明两个命题：

1、对于$n=2^k$来说，至少需要$2^{k+1}-1$步。
数学归纳法：当$k=0$时显然。假设对于$k-1$的情况上述命题也正确，对于k的情况，由于$10...0$先要变成$110...0$，对于任意的能把$10...0$变成$110...0$的最短的操作序列，都不会改动最高位的1（如果改变了的话，说明中途已经到了$110...0$了，与最短矛盾），所以把$10...0$变成$110...0$的最短的操作序列其实就是把0变为$10...0$的操作序列（即只考虑除了最高位1的剩余数字）。而题目的两个操作实际上是在做异或运算，这个运算是可逆的，所以把0变为$10...0$的最短操作序列的逆序就是把$10...0$变为0的最短操作序列，从而根据归纳假设，其有2^k-1步，接着$110...0$变成$10...0$需要1步，然后再根据归纳假设，其经过2^k-1步变为0，所以总共步数就是$2^k-1+2^k=2^{k+1}-1$步。由数学归纳法，命题对于任意k都对。
2、将$n\to x$的最少步数等于$n\wedge x\to x\wedge x=0$的最少步数，这一点由异或运算满足结合律显然成立（左边同时做异或即可）。

我们回到原问题，设$f(n)$是答案。对于n而言，$n\to 110...0$需要的步数就等于$n\wedge 110...0\to 110...0\wedge 110...0=0$的步数，我们取$110...0$是能将$n$的最高位$1$通过异或消掉的数，令$k$是小于等于$n$的最大的$2$的幂（例如$n=10$的时候$k=2^3=8$，$n=17$的时候$k=2^4=16$），那么显然$k+k/2$就是形如$110...0$并且能将$n$的最高位$1$消掉的数。所以有

$$f(n)=f(n\wedge 110...0)+1+k-1=f(n\wedge 110...0)+k$$

只需要直接DFS就行了。因为每次参数的最高位都会被消掉，所以位数一定会越变越少，时间复杂度就是$O(\log n)$。代码如下：

```
public int minimumOneBitOperations(int n) { // O(logN)
    if (n == 0) return 0;
    int k = 1;
    while (k << 1 <= n)
        k <<= 1;
    return minimumOneBitOperations(k ^ (k >> 1) ^ n) + k;
}
public int minimumOneBitOperations(int n) {
    int ans = 0;
    while (n > 0) {
        ans ^= n;
        n /= 2;
    }
    return ans;
}
```

6.0.2 1643. Kth Smallest Instructions - Hard

Bob is standing at cell (0, 0), and he wants to reach destination: (row, column). He can only travel right and down. You are going to help Bob by providing instructions for him to reach destination.

The instructions are represented as a string, where each character is either:

'H', meaning move horizontally (go right), or 'V', meaning move vertically (go down). Multiple instructions will lead Bob to destination. For example, if destination is (2, 3), both "HHHVV" and "HVHVH" are valid instructions.

However, Bob is very picky. Bob has a lucky number k, and he wants the kth lexicographically smallest instructions that will lead him to destination. k is 1-indexed.

Given an integer array destination and an integer k, return the kth lexicographically smallest instructions that will take Bob to destination.

1. 解题思路与分析

方法一：优先确定高位 + 组合计数

思路与算法

当字符串中每一种字符的数量固定时（例如对于本题，我们需要在字符串中放入 h 个 H 和 v 个 V），如果需要求出字典序第 k 小的字符串，可以考虑从高位高位向低位依次确定每一个位置的字符。

> 如果我们在最高位放置了 H，那么剩余的 $(h-1,v)$ 就是一个规模减少的相同问题；同理如果我们在最高位放置了 V，那么剩余的 $(h,v-1)$ 也是一个规模减少的相同问题。

我们考虑最高位是放 H 还是 V。由于后者的字典序较大，因此如果最高位放 V，那么**所有最高位为 H 的字符串的字典序都比它小**，这样的字符串共有

$$o=\binom{h+v-1}{h-1}$$

个。也就是**确定了最高位为 H，剩余 $h+v-1$ 个位置中选择 $h-1$ 个放入 H，其余位置自动放入 V** 的方案数。因此：

- 如果 k 大于这个组合数 o，那么最高位一定是 V。我们将 v 减少 1，**并且需要将 k 减少 o**，这是因为剩余部分应当是包含 $(h,v-1)$ 的字典序第 $k-o$ 小的字符串；
- 如果 k 小于 o，那么最高位是 H。我们将 h 减少 1，但我们不需要改变 k 的值，这是因为剩余部分就是包含 $(h-1,v)$ 的字典序第 k 小的字符串。

这样一来，我们就可以从高位开始，依次确定每一个位置的字符了。需要注意的是，当 $h=0$ 时，我们只能放 V，无需进行判断。

代码

对于 `Python` 语言，可以使用 `math.comb()` 方便地求出组合数。但对于 `C++` 而言，由于本题会导致乘法溢出，因此可以考虑使用组合数的递推式

$$c[n][k]=c[n-1][k-1]+c[n-1][k]$$

预处理处所有可能需要用到的组合数。

本题中，可能需要计算的最大组合数为 $\binom{29}{14}$，在 `C++` 语言中，直接通过先乘法后除法的方法计算该组合数，在乘法过程中就会超出 64 位无符号整数的上限。

```
public String kthSmallestPath(int [] a, int k) {
    int m = a[0], n = a[1], mn = m + n; // m rows, n cols
    int [][] c = new int [m+n][n]; // calculate combinations
    c[0][0] = 1;
    for (int i = 1; i < m+n; i++) {
        c[i][0] = 1; // 从这些数量中选 0 个的可能性： 1--》就是什么也不选
```

```
            for (int j = 1; j <= i && j < n; j++) // 对于第 j 个数，有两种选择:
                c[i][j] = c[i-1][j] + c[i-1][j-1]; // 可以不选 j，则所有选的 j 个数由前 i-1 个数选出；或选 j，则从前 i-1 个数中选择出 j-1 个数
        }
        String ans = "";
        for (int i = 0; i < mn; i++) { // loop 出 m+n 步中，根据 k 值的大小，每一步的选择
            if (n > 0) { // 如果当前选择的是 "V" : 那么所有最高位为 H 的字符串的字典序都比它小，这样的字符串共有 cnt 种
                int cnt = c[m+n-1][n-1]; // 会有多少种选择
                if (k > cnt) {  // k 比这个值大，说明必须选 V
                    ans += "V"; // 我们将 vv 减少 11，并且需要将 kk 减少 oo，这是因为剩余部分应当是包含 (h,v-1)(h,v1) 的字典序第 k-oko 小的字符串；
                    --m; // BUG: 这里可能没有理解对，想要减少 'V'，减少的当然是行数呀
                    k -= cnt;
                } else {
                    ans += "H";
                    --n; // BUG: 减少 H，减少的是列数
                }
            } else {
                ans += "V";
                --m;
            }
        }
        return ans;
    }
```

6.0.3 1467. Probability of a Two Boxes Having The Same Number of Distinct Balls - Hard

Given 2n balls of k distinct colors. You will be given an integer array balls of size k where balls[i] is the number of balls of color i.

All the balls will be shuffled uniformly at random, then we will distribute the first n balls to the first box and the remaining n balls to the other box (Please read the explanation of the second example carefully).

Please note that the two boxes are considered different. For example, if we have two balls of colors a and b, and two boxes [] and (), then the distribution [a] (b) is considered different than the distribution [b] (a) (Please read the explanation of the first example carefully).

We want to calculate the probability that the two boxes have the same number of distinct balls.

Example: [2, 1, 1]

for first ball, we can put both ball in first bin and 0 ball in second bin OR we can put 1 ball in first bin, second ball in 2nd bin OR we can put both in second bin

for second ball, we can put ball in first bin and 0 ball in second bin, similarly, we can put 1 ball in second bin.

same thing with the third ball

Try all possible permutations recursively. And, in the end check, if there are equal number of distinct balls in both bins or not.

1. 解题思路与分析

   思路比较朴素，就是先把这 2n 个数字分成两块，对应两个括号。先不考虑括号里这些数字的排列情况。对于每一种颜色的球，可以放 0, 1, 2, . . . balls[i] 个到第一个块。

   依次类推到其他颜色的球。我们可以遍历所有的这些可能的分块方法。balls[i] <= 6, balls.length <= 8，所以一共最多有 7 ^ 8 种方法可能不算太多。

   对于上述每一种分块，都会有 P(ballsA) * P(ballsB) 种可能。P(balls) 就是我们下面要计算的，给定每种球的数量，有多少种不同的排列。

   P(balls) = sum(balls)! / (balls^2! * balls^1! * balls^3! . . . .balls[n - 1]!);

   可以从小到大依次计算

   把 balls^2 + balls^1 + . . . balls[i - 1] = sum

   P[0 : i] = p[0 : i - 1] * (sum + 1) * (sum + 2) * . . . .(sum + balls[i]) / factorial[balls[i]]

   修改了下题解，不需要用 BigInteger 了，java 的 Double 就足够大。

```
private static final double [] fact = {1, 1, 2, 6, 24, 120, 720};
private double difCnt = 0;
int n;
public double getProbability(int[] balls) { // 居然是个回塑 + 裁枝
    n = balls.length;
    double totCnt = getPermutation(balls);
    dfs(0, balls, new int [n]);
    return difCnt / totCnt;
}
private double getPermutation(int [] a) {
    double [] ans = new double [n];
    ans[0] = 1;
    int sum = a[0];
    for (int i = 1; i < n; i++) {
        ans[i] = ans[i-1];
        for (int j = sum+1; j <= sum + a[i]; j++)
```

```
            ans[i] *= j;
        ans[i] /= fact[a[i]];
        sum += a[i];
    }
    return ans[n-1];
}
private void dfs(int idx, int [] a, int [] b) {
    if (idx == n) {
        int ca = 0, cb = 0, sa = 0, sb = 0;
        for (int i = 0; i < n; i++) {
            sa += a[i];
            sb += b[i];
            if (a[i] > 0) ca ++;
            if (b[i] > 0) cb ++;
        }
        if (ca == cb && sa == sb)
            difCnt += getPermutation(a) * getPermutation(b);
        return ;
    }
    for (int i = 0; i <= a[idx]; i++) {
        a[idx] -= i;
        b[idx] += i;
        dfs(idx+1, a, b);
        b[idx] -= i;
        a[idx] += i;
    }
}
```

- 以前参考过的题解

```
public double getProbability(int[] balls) {
    int sum = Arrays.stream(balls).sum();
    double all = allCases(balls, 0, 0, 0, 0, 0, sum);
    double valid = casesWithEqualDistinctBalls(balls, 0, 0, 0, 0, 0, sum);
    return (1.0 * valid / all);
}
// disF = distinct balls in first bin
// disS = distinct balls in second bin
// f = number of balls in first bin
// s = number of balls in second bin
public double allCases(int [] arr, int pos, int f, int s, int disF, int disS, int sum) {
    if (pos == arr.length) {
        // for all cases, we just need to check if both bins have same number of balls or not
        if (f == s) return fact(sum / 2) * fact(sum / 2); // numerator of our permutations
        return 0;
    }
    // we put all balls in second bin
    double ans = 1.0 * allCases(arr, pos+1, f, s+arr[pos], disF, disS+1, sum) / fact(arr[pos]);
    // we put all balls in first bin
    ans += 1.0 * allCases(arr, pos+1, f+arr[pos], s, disF+1, disS, sum) / fact(arr[pos]);
    for (int i = 1; i < arr[pos]; i++) // 把每一种颜色的球放到两个里面盒子里都有
        ans += 1.0 * allCases(arr, pos+1, f+i, s+arr[pos]-i, disF+1, disS+1, sum) / (fact(i) * fact(arr[pos]-i));
    return ans;
}
public double casesWithEqualDistinctBalls(int [] arr, int pos, int f, int s, int disF, int disS, int sum) {
    if (pos == arr.length) {
        if (f == s && disF == disS) return fact(sum / 2) * fact(sum / 2);
        return 0;
    }
    double ans = 1.0 * casesWithEqualDistinctBalls(arr, pos+1, f, s+arr[pos], disF, disS+1, sum) / fact(arr[pos]);
    ans += 1.0 * casesWithEqualDistinctBalls(arr, pos+1, f+arr[pos], s, disF+1, disS, sum) / fact(arr[pos]);
    for (int i = 1; i < arr[pos]; i++)
        ans += 1.0 * casesWithEqualDistinctBalls(arr, pos+1, f+i, s+arr[pos]-i, disF+1, disS+1, sum) / (fact(i) * fact(arr[pos]-i));
    return ans;
}
private double fact(double n) {
    double res = 1;
    for (int i = 2; i <= n; i++)
        res = res * i;
    return res;
}
// Complexity: There can be total of (8 * 6) balls with 8 distinct. Complexity of recursion is f * s * disF * disS = O(48 * 48 * 8 * 8)
```

6.0.4 1569. Number of Ways to Reorder Array to Get Same BST - Hard

Given an array nums that represents a permutation of integers from 1 to n. We are going to construct a binary search tree (BST) by inserting the elements of nums in order into an initially empty BST. Find the number of different ways to reorder nums so that the constructed BST is identical to that formed from the original array nums.

For example, given nums = [2,1,3], we will have 2 as the root, 1 as a left child, and 3 as a right child. The array [2,3,1] also yields the same BST but [3,2,1] yields a different BST.

Return the number of ways to reorder nums such that the BST formed is identical to the original BST formed from nums.

Since the answer may be very large, return it modulo 10^9 + 7.

1. 解题思路与分析: 官方题解

   - https://leetcode-cn.com/problems/number-of-ways-to-reorder-array-to-get-same-bst/solution/jiang-z

   根节点是数组第一个数
   然后分为左右两个子树，左右子树之间的顺序不乱就可以
   假设左子树 L 长度 nL，右子树 R 长度 nR，存在方案数为 C_{nL+nR}^{nL}*f(L)*f(R)

```
public int numOfWays(int [] a) {
    int n = a.length;
    f = new int [n+1][n+1];
    f[1][0] = 1; // C_1^0 = 1
    f[1][1] = 1; // C_1^1 = 1
    for (int i = 2; i <= n; i++) // DP 求解组合数
        for (int j = 0; j <= i; j++)
            if (j == 0 || j == i) f[i][j] = 1; // C_n^0 = C_n^n = 1
            else f[i][j] = (f[i-1][j-1] + f[i-1][j]) % mod; // 选与不选第 j 个数
    return (int)((dfs(Arrays.stream(a).boxed().collect(Collectors.toList())) - 1) % mod);
}
int mod = (int)1e9 + 7;
int [][] f;
private long dfs(List<Integer> a) {
    if (a.size() <= 1) return 1;
    int root = a.get(0), n = a.size();
    List<Integer> l = new ArrayList<>();
    List<Integer> r = new ArrayList<>();
    for (int v : a)
        if (v < root) l.add(v);
        else if (v > root) r.add(v);
    long cntLeft = dfs(l), cntRight = dfs(r);
    return ((f[n-1][l.size()] * cntLeft % mod) * cntRight) % mod;
}
```

2. 解题思路与分析: 先根据数组 nums 把整棵二叉查找树 TT 建立出来

```
static final int mod = (int)1e9 + 7;
long [][] c;
public int numOfWays(int [] a) {
    int n = a.length;
    if (n == 1) return 0;
    c = new long [n][n];
    c[0][0] = 1;
    for (int i = 1; i < n; ++i) {
        c[i][0] = 1;
        for (int j = 1; j < n; ++j)
            c[i][j] = (c[i - 1][j - 1] + c[i - 1][j]) % mod;
    }
    TreeNode root = new TreeNode(a[0]);
    for (int i = 1; i < n; ++i) {
        int val = a[i];
        insert(root, val);
    }
    dfs(root);
    return (root.ans - 1 + mod) % mod;
}
public void insert(TreeNode root, int v) {
    TreeNode r = root;
    while (true) {
        ++r.size;
        if (v < r.val) {
            if (r.left == null) {
                r.left = new TreeNode(v);
                return;
            }
            r = r.left;
        } else {
            if (r.right == null) {
                r.right = new TreeNode(v);
                return;
            }
            r = r.right;
        }
    }
}
public void dfs(TreeNode r) {
    if (r == null) return;
    dfs(r.left);
    dfs(r.right);
    int lsize = r.left != null ? r.left.size : 0;
    int rsize = r.right != null ? r.right.size : 0;
    int lans = r.left != null ? r.left.ans : 1;
    int rans = r.right != null ? r.right.ans : 1;
    r.ans = (int) (c[lsize + rsize][lsize] % mod * lans % mod * rans % mod);
}
class TreeNode {
```

```
    TreeNode left;
    TreeNode right;
    int val;
    int size;
    int ans;
    TreeNode(int v) {
        this.val = v;
        this.size = 1;
        this.ans = 0;
    }
}
```

- 复杂度分析

时间复杂度：O(n^2)

时间复杂度由以下三部分组成：

预处理组合数的时间复杂度为 O(n^2)

建立二叉查找树的平均时间复杂度为 O(nlogn)。但在最坏情况下，当数组 nums 中的数单调递增或递减时，二叉查找树退化成链式结构，建立的时间复杂度为 O(n^2)

动态规划的时间复杂度为 O(n)O(n)，即为对二叉查找树进行遍历需要的时间。

空间复杂度：O(n^2)

3. 解题思路与分析: 并查集 + 乘法逆元优化

```
static final int mod = (int)1e9 + 7;
long [] fac;
long [] inv;
long [] facInv;
public int numOfWays(int[] a) { // 这个方法还要再消化一下
    int n = a.length;
    if (n == 1) return 0;
    fac = new long[n];
    inv = new long[n];
    facInv = new long[n];
    fac[0] = inv[0] = facInv[0] = 1;
    fac[1] = inv[1] = facInv[1] = 1;
    for (int i = 2; i < n; ++i) {
        fac[i] = fac[i - 1] * i % mod;
        inv[i] = (mod - mod / i) * inv[mod % i] % mod;
        facInv[i] = facInv[i - 1] * inv[i] % mod;
    }
    Map<Integer, TreeNode> found = new HashMap<Integer, TreeNode>();
    UnionFind uf = new UnionFind(n);
    for (int i = n - 1; i >= 0; --i) {
        int val = a[i] - 1;
        TreeNode node = new TreeNode();
        if (val > 0 && found.containsKey(val - 1)) {
            int lchild = uf.getroot(val - 1);
            node.left = found.get(lchild);
            node.size += node.left.size;
            uf.findAndUnite(val, lchild);
        }
        if (val < n - 1 && found.containsKey(val + 1)) {
            int rchild = uf.getroot(val + 1);
            node.right = found.get(rchild);
            node.size += node.right.size;
            uf.findAndUnite(val, rchild);
        }
        int lsize = node.left != null ? node.left.size : 0;
        int rsize = node.right != null ? node.right.size : 0;
        int lans = node.left != null ? node.left.ans : 1;
        int rans = node.right != null ? node.right.ans : 1;
        node.ans = (int) (fac[lsize + rsize] * facInv[lsize] % mod * facInv[rsize] % mod * lans % mod * rans % mod);
        found.put(val, node);
    }

    return (found.get(a[0] - 1).ans - 1 + mod) % mod;
}
class UnionFind {
    public int[] parent;
    public int[] size;
    public int[] root;
    public int n;
    public UnionFind(int n) {
        this.n = n;
        parent = new int[n];
        size = new int[n];
        root = new int[n];
        Arrays.fill(size, 1);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            root[i] = i;
```

```
            }
        }
        public int findset(int x) {
            return parent[x] == x ? x : (parent[x] = findset(parent[x]));
        }
        public int getroot(int x) {
            return root[findset(x)];
        }
        public void unite(int x, int y) {
            root[y] = root[x];
            if (size[x] < size[y]) {
                int temp = x;
                x = y;
                y = temp;
            }
            parent[y] = x;
            size[x] += size[y];
        }
        public boolean findAndUnite(int x, int y) {
            int i = findset(x);
            int j = findset(y);
            if (i != j) {
                unite(i, j);
                return true;
            }
            return false;
        }
    }
    class TreeNode {
        TreeNode left;
        TreeNode right;
        int size;
        int ans;
        TreeNode() {
            size = 1;
            ans = 0;
        }
    }
```

- 复杂度分析

时间复杂度：O(n(n))。

空间复杂度：O(n)

6.0.5 903. Valid Permutations for DI Sequence - Hard

You are given a string s of length n where s[i] is either:

'D' means decreasing, or 'I' means increasing. A permutation perm of n + 1 integers of all the integers in the range [0, n] is called a valid permutation if for all valid i:

If s[i] = 'D', then perm[i] > perm[i + 1], and If s[i] = 'I', then perm[i] < perm[i + 1]. Return the number of valid permutations perm. Since the answer may be large, return it modulo 109 + 7.

当是降序时，下一个数字不小于当前最后一个数字，反之是升序时，下一个数字小于当前最后一个数字，所以可以写出状态转移方程如下所示：

```
if (S[i-1] == 'D')    dp[i][j] += dp[i-1][k]    ( j <= k <= i )
else                  dp[i][j] += dp[i-1][k]    ( 0 <= k < j )
```

(动态规划) $O(n^3)$

题解1:动态规划。这道题首先给出了最后答案很大需要取模，所以这道题很明显是一道计数DP类的问题，接下来我们需要寻找状态表示和状态转移。很朴素的想法就是 `dp[i]` 代表了 `[0 : i]` 这 `i + 1` 个元素满足前 `i` 个序列的方案数，但是这样，我们无法直接进行状态转移，因为下一个元素放置什么除了取决于上升下降关系外，还取决于当前最后一个数字是多少。基于此，我们可以提出这样的状态表示：

`dp[i][j]` 代表将 `[0 : i]` 这 `i + 1` 个元素满足前 `i` 个DI序列并且最后一个元素是 `j` 的方案数，这里很明显 `j <= i` 。

接下来我们考虑状态转移：

考虑 `dp[2][1]` 代表的是前三个元素并且末尾是 `1` 的方案数，我们从中任取一种方案， `2,0,1` ，接下来我们需要放置元素 `3` 了。

假设此时 `s[2] = D` 。这是我们显然不能直接把 `3` 放在后面。考虑当前最后一个元素为1，那么我们可以把序列中所有大于等于1的元素都加上1得到序列 `3,0,2` （这个时候是不会改变当前序列的大小关系的），这个时候，我们再把1添加到序列末尾得到 `3,0,2,1` 。同样的我们可以把序列中所有大于等于0的元素都加上1得到序列 `3,1,2` ，这时候我们再把0添加到末尾，得到 `3,1,2,0` 。进一步的我们可以得到，对 `dp[i - 1][k]` 中的所有方案，我们都可以将该序列中所有大于等于 `j(0 <= j <= k)` 的元素加上1，并在末尾添加一个 `j` 得到 `dp[i][j]` 中的一种方案。也就是 `dp[i][j]` 可以由所有的 `j <=k < i` 对应的 `dp[i - 1][k]` 生成。

假设此时 `s[2] = I` 。这时候我们可以直接把3放在后面。考虑当前最后一个元素为1，我们同样的可以把序列中所有大等于2的元素都加上1得到 `3,0,1` ，再把2放在序列末尾得到 `3,0,1,2` 。进一步的，我们可以得到对于 `dp[i - 1][k]` 中的所有方案，我们都可以把该序列中所有大于等于 `j(k < j <= i)` 的数字都加上1，并在末尾添加一个 `j` 得到 `dp[i][j]` 中的一种方案。也就是 `dp[i][j]` 可以由所有的 `0 <= k < j` 对应的 `dp[i - 1][k]` 生成。

基于此，我们的状态转移可以写成：

当 `s[i - 1] = 'I'` 时，$dp(i,j) = \sum_{k=0}^{k=j-1} dp(i-1,k)$

当 `s[i - 1] = 'D'` 时，$dp(i,j) = \sum_{k=j}^{k=i-1} dp(i-1,k)$

状态初始化： `dp[0][0] = 1` ，刚开始只能把0放在起始位置上。

时间复杂度分析：$O(n^3)$

```
public int numPermsDISequence(String s) {
    int n = s.length(), mod = (int)1e9 + 7, res = 0;
    int [][] dp = new int [n+1][n+1];
    dp[0][0] = 1;
    for (int i = 1; i <= n; i++)
        for (int j = 0; j <= i; j++) // 考虑当前最后一个元素为 j
            if (s.charAt(i-1) == 'D')
                // 可以把序列中所有大于等于 j 的元素都加上 1 得到新序列 (这个时候是不会改变当前序列的大小关系的), 再把 j 添加到序列末尾得到
                for (int k = j; k <= i; k++)
                    dp[i][j] = (dp[i][j] + dp[i-1][k]) % mod;
            else // 考虑当前最后一个元素为 j
                 // 把该序列中所有大于等于 j(k < j <= i) 的数字都加上 1, 并在末尾添加一个 j 得到 dp[i][j] 中的一种方案
                for (int k = 0; k < j; k++) // 把序列中所有大等于 j 的元素都加上 1, 再把 j 放在序列末尾得到
                    dp[i][j] = (dp[i][j] + dp[i-1][k]) % mod;
    for (int i = 0; i <= n; i++)
        res = (res + dp[n][i]) % mod;
    return (int)res;
}
```

1. 解题思路与分析: (优化版本动态规划) O(n2)

题解 2: 动态规划优化。延续上述的思路，但是我们发现我们的重复计算太多了。

当 s[i - 1] == 'D' 的时候:

```
dp[i][j] = dp[i - 1][j] + dp[i - 1][j + 1] + dp[i - 1][j + 2] + ... + dp[i][i - 1]
dp[i][j + 1] = dp[i - 1][j + 1] + dp[i - 1][j + 2] + ... + dp[i][i - 1]
dp[i][j] = dp[i][j + 1] + dp[i - 1][j]
基于此，我们需要从后往前计算，而且从i - 1开始计算就可以了，一方面是当s[i - 1] = 'D'的时候，i不可能出现在最后一个位置上，同时也避免了j + 1越界。
```

当 s[i - 1] = 'I' 的时候

```
dp[i][j - 1] = dp[i - 1][0] + dp[i - 1][1] + ... + dp[i - 1][j - 2]
dp[i][j] = dp[i][j - 1] + dp[i - 1][j - 1]
基于此，我们需要从前往后计算，而且从1开始计算就可以了，一方式是当s[i - 1] = 'I'的时候，0不可能出现在最后一个位置上，同时也避免了j - 1越界。
dp[i][j] = dp[i - 1][0] + dp[i - 1][1] + ... + dp[i - 1][j - 1]
```

时间复杂度:O(n2)

```
static final int mod = (int)1e9 + 7;
public int numPermsDISequence(String t) {
    int n = t.length(), ans = 0;
    char [] s = t.toCharArray();
    int [][] dp = new int [n + 1][n + 1];
    dp[0][0] = 1;
    for (int i = 1; i <= n; i ++)
        if (s[i - 1] == 'D')
            for (int j = i - 1 ; j >= 0 ; j --)
                dp[i][j] = (dp[i][j + 1] + dp[i - 1][j]) % mod;
        else for (int j = 1; j <= i; j ++)
                dp[i][j] = (dp[i][j - 1] + dp[i - 1][j - 1]) % mod;
    for (int i = 0 ; i <= n ; i ++)
        ans = (ans + dp[n][i]) % mod;
    return ans;
}
```

2. 解题思路与分析

(分治/区间动态规划) $O(n^3)$

题解3:分治算法（区间动态规划）。

状态表示：`dp[i][j]`表示填充区间p`[i:j]`共`j - i + 1`个数的方案数。不考虑具体数字，只考虑数字之间的大小关系。

状态初始化：`dp[i][i] = 1`，区间内只有一个数字，数字大小关系是唯一确定的。最终返回`dp[0][n]`即可。

状态转移：我们考虑枚举当前区间最大的数可以放在哪些位置。我们知道控制区间`dp[i][j]`的是序列`s[i:j - 1]`。

如果`s[i] = 'D'`，那么最大的元素可以放在`p[i]`上。`dp[i][j] += dp[i + 1][j]`

如果`s[j - 1] = 'I'`，那么最大的元素可以放在`p[j]`上。`dp[i][j] += dp[i][j - 1]`

每一个区间其实类似于一个摆动序列，最大元素只可能放在峰值的位置上。所以我们枚举`[i + 1:j - 1]`中每一个位置`k`。如果`s[k - 1] = 'I' and s[k] = 'D'`，那么当前位置就是一个峰值。那么现在数组被划分成了两个部分`s[i:k - 1]`和`s[k + 1,j]`。区间长度为`j - i + 1`，除去最大的那个元素，那么还剩`j - i`个元素，我们需要从这`j - i`个元素中挑选出`k - i`个元素放在左半区间。这样我们既可以求解两个子区间了。

`dp[i][j] += dp[i][k - 1] * dp[k + 1][j] * C(j - i,k - i) if s[k - 1] = 'I' and s[k] = 'D' for`

。

这里我们介绍一下组合数的动态规划求解：

状态表示：`C[i][j]`代表从`i`个数选择`j`个数的所有方案数。

状态初始化：`C[i][0] = 1`代表，从`i`个数选择0个数只有一种情况什么都不选。

状态转移：`C[i][j] = C[i - 1][j - 1] + C[i - 1][j]`。考虑从`i`个数中选择`j`个数的方案数，我们考虑最后一个数字选不选，如果最后一个数字选的话，那么相当于还需要从前`i - 1`个数字中选择`j - 1`个数字，如果最后一个数字不选的话，相当于还需要从前`i - 1`个数字中选择`j`个数字。

时间复杂度分析：$O(n^3)$

```
static final int mod = (int)1e9 + 7;
public int numPermsDISequence(String t) {
    int n = t.length(), ans = 0;
    char [] s = t.toCharArray();
    long [][] dp = new long [n+1][n+1];
    long [][] c = new long [n+2][n+2];
    for (int i = 0; i <= n; i++)
        dp[i][i] = 1;
    for (int i = 0; i <= n+1; i++) {
        c[i][0] = 1;
        for (int j = 1; j <= i; j++)
            c[i][j] = (int)((c[i-1][j] + c[i-1][j-1]) % mod);
    }
    for (int len = 2; len <= n+1; len++)
        for (int i = 0; i+len-1 <= n; i++) {
            int j = i + len - 1;
            if (s[i] == 'D')   // 如果 s[i] = 'D', 那么最大的元素可以放在 p[i] 上。dp[i][j] += dp[i + 1][j]
                dp[i][j] = (dp[i][j] + dp[i+1][j]) % mod;
            if (s[j-1] == 'I') // 如果 s[j - 1] = 'I', 那么最大的元素可以放在 p[j] 上。dp[i][j] += dp[i][j - 1]
                dp[i][j] = (dp[i][j] + dp[i][j-1]) % mod;
            for (int k = i+1; k <= j-1; k++)
                if (s[k-1] == 'I' && s[k] == 'D')
                    dp[i][j] = (dp[i][j] + (1l * c[len-1][k-i]) * dp[i][k-1] % mod * dp[k+1][j] % mod) % mod;
        }
    return (int)dp[0][n];
}
```

6.0.6 233. Number of Digit One - Hard

Given an integer n, count the total number of digit 1 appearing in all non-negative integers less than or equal to n.

1. 解题思路与分析: 递归

```
[1--9]--> 1
[10--19] --> 11
[20--29] --> 1
[30--39] --> 1
....
[90--99] --> 1
[100--199] --> 100 + count(99)
[200-299] ---> count(99)
....
[900-999] ---> count(99)
[1000-1999]  -> 1000 + count(999) // 首先千位数上一直都是 1, 所以这里有 1000 个 1, 然后在个位, 十位和百位上的 1 就是 count(999)
[2000-2999] --> count(999)
[3000-3999] --> count(999)
...
[9000-9999] --> count(999)
```

上面的数看上去已经有点规律了，那么我们如果要求解符合题目要求的 n 总共有多少个 1，

a. 假设 n=2345,

那么 f(2345) = 1000 + f(345) + f(999) * 2

相当于 [0–999] 之间的 1 会需要计数两次，然后呢，加上千位数的 1 会计数 1000 次。最后，还有 345 需要计数

b. 假设 n=3568,

那么 f(3568) = 1000 + f(568) + f(999) * 3

同理，[1000-1999] 在千位数上的 1 有 1000 个，然后，[0–999] 需要计数三次，就是千位数分别数 0，1，2 的时候总共有三次，

最后，千位数是 3 的时候，568 的时候需要计数。

c. 假设 n=1729,

那么 f(1729) = 729+1 + f(999) + f(729)

考虑千位数的时候，千位数数 1 的情况和前面两个情况不太一样，这时候，千位数是 1 的数是 [1000-1729], 所以总共有 729+1 个，

然后，千位数数 0 的时候，[0–999] 就是 f(999), 千位数是 1 的时候，还有 f(729) 需要计数

```
public int countDigitOne(int n) {
    if (n <= 0) return 0;
    if (n < 10) return 1;
    int base = (int)Math.pow(10, String.valueOf(n).length()-1);
    int fst = n / base, residual = n % base;
    if (fst == 1)
        return residual + 1 + countDigitOne(residual) + countDigitOne(base - 1);
    else
        return base + countDigitOne(residual) + fst * countDigitOne(base - 1);
}
```

2. 解题思路与分析

以算百位上 1 为例子:

假设百位上是 0, 1, 和 >=2 三种情况:

case 1: n=3141092, a= 31410, b=92. 计算百位上 1 的个数应该为 3141 *100 次.

case 2: n=3141192, a= 31411, b=92. 计算百位上 1 的个数应该为 3141 *100 + (92+1) 次.

case 3: n=3141592, a= 31415, b=92. 计算百位上 1 的个数应该为 (3141+1) *100 次.

所以可以将每一位归纳成这样一个公式:

(a + 8) / 10 * m + (a % 10 == 1) * (b + 1)

需要注意的坑，虽然最终结果不会超过 int 范围，但是因为中间计算涉及乘法，所以会出现溢出，需要用 long long 存储中间变量。

```
public int countDigitOne(int n) { //
    long digit = 1L;//值为 1, 10, 100, ..., 表示当前处理的十进制数位
    int ans = 0;
    while (n >= digit) { // 枚举每一数位上 1 的个数
        int cnt1 = (int)(n / (digit * 10) * digit); // 由更高数位的值产生的当前数位 1 的出现次数
        int cnt2 = Math.min(Math.max((int)(n % (digit * 10) - digit + 1), 0), (int)digit); // 由更低数位的值产生的当前数位 1 的出现次数
        ans += cnt1 + cnt2;
        digit *= 10L; // 十进制数位左移 1 位
    }
    return ans;
}
```

3. 解题思路与分析: 数每一位上 1 的个数

   The idea is to calculate occurrence of 1 on every digit. There are 3 scenarios, for example

   if n = xyzdabc

   and we are considering the occurrence of one on thousand, it should be:

```
(1) xyz * 1000                    if d == 0
(2) xyz * 1000 + abc + 1          if d == 1
(3) xyz * 1000 + 1000             if d > 1
```

   iterate through all digits and sum them all will give the final answer

```java
public int countDigitOne(int n) {
    if (n <= 0) return 0;
    int q = n, x = 1, ans = 0;
    do {
        int digit = q % 10;
        q /= 10;
        ans += q * x;
        if (digit == 1) ans += n % x + 1;
        if (digit >  1) ans += x;
        x *= 10;
    } while (q > 0); // q > 0
    return ans;
}
public int countDigitOne(int n) {
    int count = 0;
    //依次考虑个位、十位、百位... 是 1
    //k = 1000, 对应于上边举的例子
    for (int k = 1; k <= n; k *= 10) {
        // xyzdabc
        int abc = n % k;
        int xyzd = n / k;
        int d = xyzd % 10;
        int xyz = xyzd / 10;
        count += xyz * k;
        if (d > 1) {
            count += k;
        }
        if (d == 1) {
            count += abc + 1;
        }
        //如果不加这句的话, 虽然 k 一直乘以 10, 但由于溢出的问题
        //k 本来要大于 n 的时候, 却小于了 n 会再次进入循环
        //此时代表最高位是 1 的情况也考虑完成了
        if(xyz == 0){
            break;
        }
    }
    return count;
}
public int countDigitOne(int n) {
    int count = 0;
    for (long k = 1; k <= n; k *= 10) {
        long r = n / k, m = n % k;
        // sum up the count of ones on every place k
        count += (r + 8) / 10 * k + (r % 10 == 1 ? m + 1 : 0);
    }
    return count;
}
```

4. 解题思路与分析

   复杂度分析

   时间复杂度：O(log n)O(logn)。nn 包含的数位个数与 nn 呈对数关系。

   空间复杂度：O(1)O(1)。

```java
public int countDigitOne(int n) {
// mulk 表示 10^k
// 在下面的代码中, 可以发现 k 并没有被直接使用到 (都是使用 10^k)
// 但为了让代码看起来更加直观, 这里保留了 k
    long mulk = 1;
    int ans = 0;
    for (int k = 0; n >= mulk; ++k) {
        ans += (n / (mulk * 10)) * mulk + Math.min(Math.max(n % (mulk * 10) - mulk + 1, 0), mulk);
        mulk *= 10;
    }
    return ans;
}
```

Chapter 7

binary Search

7.1 LeetCode Binary Search Summary 二分搜索法小结

- https://segmentfault.com/a/1190000016825704
- https://www.cnblogs.com/grandyang/p/6854825.html

7.1.1 标准二分查找

```
public int search(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;
    while (left <= right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] == target) return mid;
        else if (nums[mid] > target)
            right = mid - 1;
        else
            left = mid + 1;
    }
    return -1;
}
```

循环终止的条件包括：

- 找到了目标值
- left > right （这种情况发生于当 left, mid, right 指向同一个数时，这个数还不是目标值，则整个查找结束。）

left + ((right -left) » 1) 对于目标区域长度为奇数而言，是处于正中间的，对于长度为偶数而言，是中间偏左的。因此左右边界相遇时，只会是以下两种情况：

- left/mid , right (left, mid 指向同一个数，right 指向它的下一个数)
- left/mid/right （left, mid, right 指向同一个数）

即因为 mid 对于长度为偶数的区间总是偏左的，所以当区间长度小于等于 2 时，mid 总是和 left 在同一侧。

7.1.2 二分查找左边界

利用二分法寻找左边界是二分查找的一个变体，应用它的题目常常有以下几种特性之一：

- 数组有序，但包含重复元素
- 数组部分有序，且不包含重复元素
- 数组部分有序，且包含重复元素

1. 左边界查找类型 1

   类型 1 包括了上面说的第一种，第二种情况。

   既然要寻找左边界，搜索范围就需要从右边开始，不断往左边收缩，也就是说即使我们找到了 nums[mid] == target, 这个 mid 的位置也不一定就是最左侧的那个边界，我们还是要向左侧查找，所以我们在 nums[mid] 偏大或者 nums[mid] 就等于目标值的时候，继续收缩右边界，算法模板如下：

```java
public int search(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target)
            left = mid + 1;
        else
            right = mid;
    }
    return nums[left] == target ? left : -1;
}
```

返回值：nums[left] == target ? left : -1

与标准的二分查找不同：

首先，这里的右边界的更新是 right = mid，因为我们需要在找到目标值后，继续向左寻找左边界。

其次，这里的循环条件是 left < right。

因为在最后 left 与 right 相邻的时候，mid 和 left 处于相同的位置 (前面说过，mid 偏左)，则下一步，无论怎样，left, mid, right 都将指向同一个位置，如果此时循环的条件是 left <= right，则我们需要再进入一遍循环，此时，如果 nums[mid] < target 还好说，循环正常终止；否则，我们会令 right = mid，这样并没有改变 left,mid,right 的位置，将进入死循环。

事实上，我们只需要遍历到 left 和 right 相邻的情况就行了，因为这一轮循环后，无论怎样，left,mid,right 都会指向同一个位置，而如果这个位置的值等于目标值，则它就一定是最左侧的目标值；如果不等于目标值，则说明没有找到目标值，这也就是为什么返回值是 nums[left] == target ? left : -1。

```java
public int searchInsert(int[] nums, int target) {
    int len = nums.length;
    if (nums[len - 1] < target) return len;
    int left = 0;
    int right = len - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        // 等于的情况最简单，我们应该放在第 1 个分支进行判断
        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            // 题目要我们返回大于或者等于目标值的第 1 个数的索引
            // 此时 mid 一定不是所求的左边界，
            // 此时左边界更新为 mid + 1
            left = mid + 1;
        } else {
            // 既然不会等于，此时 nums[mid] > target
            // mid 也一定不是所求的右边界
            // 此时右边界更新为 mid - 1
            right = mid - 1;
        }
    }
    // 注意：一定得返回左边界 left，
    // 理由是对于 [1,3,5,6]，target = 2，返回大于等于 target 的第 1 个数的索引，此时应该返回 1
    // 在上面的 while (left <= right) 退出循环以后，right < left，right = 0 ，left = 1
    // 根据题意应该返回 left，
    // 如果题目要求你返回小于等于 target 的所有数里最大的那个索引值，应该返回 right
    return left;
}
```

2. 左边界查找类型 2

左边界查找的第二种类型用于数组部分有序且包含重复元素的情况，这种条件下在我们向左收缩的时候，不能简单的令 right = mid，因为有重复元素的存在，这会导致我们有可能遗漏掉一部分区域，此时向左收缩只能采用比较保守的方式，代码模板如下：

```java
public int search(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target)
            left = mid + 1;
        else if (nums[mid] > target)
            right = mid;
        else
            right--;
    }
    return nums[left] == target ? left : -1;
}
```

它与类型 1 的唯一区别就在于对右侧值的收缩更加保守。这种收缩方式可以有效地防止我们一下子跳过了目标边界从而导致了搜索区域的遗漏。

7.1.3 二分查找右边界

```
public int search(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;
    while (left < right) {
        int mid = left + ((right - left) >> 1) + 1;
        if (nums[mid] > target)
            right = mid - 1;
        else
            left = mid;
    }
    return nums[right] == target ? right : -1;
}
```

- 循环条件：left < right
- 中间位置计算：mid = left + ((right -left) » 1) + 1
- 左边界更新：left = mid
- 右边界更新：right = mid - 1
- 返回值：nums[right] == target ? right : -1

这里大部分和寻找左边界是对称着来写的，唯独有一点需要尤其注意——中间位置的计算变了，我们在末尾多加了 1。这样，无论对于奇数还是偶数，这个中间的位置都是偏右的。

对于这个操作的理解，从对称的角度看，寻找左边界的时候，中间位置是偏左的，那寻找右边界的时候，中间位置就应该偏右呗，但是这显然不是根本原因。根本原因是，在最后 left 和 right 相邻时，如果 mid 偏左，则 left, mid 指向同一个位置，right 指向它们的下一个位置，在 nums[left] 已经等于目标值的情况下，这三个位置的值都不会更新，从而进入了死循环。所以我们应该让 mid 偏右，这样 left 就能向右移动。这也就是为什么我们之前一直强调查找条件，判断条件和左右边界的更新方式三者之间需要配合使用。

右边界的查找一般来说不会单独使用，如有需要，一般是需要同时查找左右边界。

7.1.4 二分查找左右边界

前面我们介绍了左边界和右边界的查找，那么查找左右边界就容易很多了——只要分别查找左边界和右边界就行了。

7.1.5 二分查找极值

二分查找还有一种有趣的变体是二分查找极值点，之前我们使用 nums[mid] 去比较的时候，常常是和给定的目标值 target 比，或者和左右边界比较，在二分查找极值点的应用中，我们是和相邻元素去比，以完成某种单调性的检测。关于这一点，我们直接来看一个例子就明白了。

Find Peak Element

A peak element is an element that is greater than its neighbors.

Given an input array nums, where nums[i] nums[i+1], find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that nums[-1] = nums[n] = -.

这一题的有趣之处在于他要求求一个局部极大值点，并且整个数组不包含重复元素。所以整个数组甚至可以是无序的——你可能很难想象我们可以在一个无序的数组中直接使用二分查找，但是没错！我们确实可以这么干！谁要人家只要一个局部极大值即可呢。

```
public int findPeakElement(int[] nums) {
    int left = 0;
    int right = nums.length - 1;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] < nums[mid + 1])
            left = mid + 1;
        else
            right = mid;
    }
    return left;
}

public int binarySearch2(int[] nums, int target) {
    // left 和 right 都在数组下标范围内
    // [left, right]
    int left = 0;
    int right = nums.length - 1;
    // while 循环跳出的条件是 left > right
    // 所以如果没找到 target 的话，也不需要特判了
    while (left <= right) {
```

```
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;
        else if (nums[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    // 如果没找到就只能返回-1
    return -1;
}
// 模板二，适合判断当前 index 和 index + 1 之间的关系。
// right 指针一开始的定义是在数组下标范围外的，[left, right)，所以在需要移动 right 指针的时候不能写成 right = mid。这样会遗漏掉一些下标的判断。
public int binarySearch3(int[] nums, int target) {
    // right 不在下标范围内
    // [left, right)
    int left = 0;
    int right = nums.length;
    // while 循环跳出的条件是 left == right
    // 这个模板比较适合判断当前 index 和 index + 1 之间的关系
    // left < right, example, left = 0, right = 1
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;
        else if (nums[mid] < target)
            left = mid + 1;
        else
            // 因为搜索范围是左闭右开所以这里不能-1
            right = mid;
    }
    // 最后的特判
    if (left != nums.length && nums[left] == target)
        return left;
    return -1;
}
// while 条件不满足的时候，left + 1 == right，两下标应该指向某个下标 i 和 i + 1。这样如果有什么特殊的值需要判断，应该不是 left 就是 right 了。
public int binarySearch1(int[] nums, int target) {
    // left 和 right 都在数组下标范围内
    // [left, right]
    int left = 0;
    int right = nums.length - 1;
    // 举例，start - 0, end = 3
    // 中间隔了起码有 start + 1 和 start + 2 两个下标
    // 这样跳出 while 循环的时候，start + 1 == end
    // 才有了最后的两个判断
    while (left + 1 < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;
        else if (nums[mid] < target)
            left = mid;
        else
            right = mid;
    }
    // 特判
    if (nums[left] == target) return left;
    if (nums[right] == target) return right;
    // 如果没找到就只能返回-1
    return -1;
}
```

Template #1:	Template #2:	Template #3:

// Pre-processing
...
left = 0; right = length - 1;
while (left <= right) {
 mid = left + (right - left) / 2;
 if (nums[mid] == target) {
 return mid;
 } else if(nums[mid] < target) {
 left = mid + 1;
 } else
 right = mid - 1;
 }
...
// right + 1 == left
// No more candidate
	// Pre-processing
	...
	left = 0; right = length;
	while (left < right) {
	 mid = left + (right - left) / 2;
	 if(nums[mid] < target) {
	 left = mid + 1;
	 } else {
	 right = mid;
	 }
	
	...
	// left == right
	// 1 more candidate
	// Post-Processing
		// Pre-processing
		...
		left = 0; right = length - 1;
		while (left + 1 < right) {
		 mid = left + (right - left) / 2;
		 if (num[mid] < target) {
		 left = mid;
		 } else {
		 right = mid;
		 }
		
		...
		// left + 1 == right
		// 2 more candidates
		// Post-Processing

7.1.6 第一类：需查找和目标值完全相等的数

这是最简单的一类，也是我们最开始学二分查找法需要解决的问题，比如我们有数组 [2, 4, 5, 6, 9]，target = 6，那么我们可以写出二分查找法的代码如下：

```
int find(vector<int>& nums, int target) {
    int left = 0, right = nums.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;
        else if (nums[mid] < target) left = mid + 1;
        else right = mid;
    }
    return -1;
}
```

会返回 3，也就是 target 的在数组中的位置。注意二分查找法的写法并不唯一，主要可以变动地方有四处：
第一处是 right 的初始化，可以写成 nums.size() 或者 nums.size() - 1。
第二处是 left 和 right 的关系，可以写成 left < right 或者 left <= right。
第三处是更新 right 的赋值，可以写成 right = mid 或者 right = mid - 1。
第四处是最后返回值，可以返回 left，right，或 right - 1。

- 但是这些不同的写法并不能随机的组合，像博主的那种写法，
  - 若 right 初始化为了 nums.size()，那么就必须用 left < right，而最后的 right 的赋值必须用 right = mid。
  - 但是如果我们 right 初始化为 nums.size() - 1，那么就必须用 left <= right，并且 right 的赋值要写成 right = mid - 1，不然就会出错。

所以博主的建议是选择一套自己喜欢的写法，并且记住，实在不行就带简单的例子来一步一步执行，确定正确的写法也行。
第一类应用实例：
Intersection of Two Arrays

7.1.7 第二类：查找第一个不小于目标值的数，可变形为查找最后一个小于目标值的数

这是比较常见的一类，因为我们要查找的目标值不一定会在数组中出现，也有可能是跟目标值相等的数在数组中并不唯一，而是有多个，那么这种情况下 nums[mid] == target 这条判断语句就没有必要存在。比如在数组 [2, 4, 5, 6, 9] 中查找数字 3，就会返回数字 4 的位置；在数组 [0, 1, 1, 1, 1] 中查找数字 1，就会返回第一个数字 1 的位置。我们可以使用如下代码：

```
int find(vector<int>& nums, int target) {
    int left = 0, right = nums.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) left = mid + 1;
        else right = mid;
    }
    return right;
}
```

最后我们需要返回的位置就是 right 指针指向的地方。在 C++ 的 STL 中有专门的查找第一个不小于目标值的数的函数 lower_bound，在博主的解法中也会时不时的用到这个函数。但是如果面试的时候人家不让使用内置函数，那么我们只能老老实实写上面这段二分查找的函数。

这一类可以轻松的变形为查找最后一个小于目标值的数，怎么变呢。我们已经找到了第一个不小于目标值的数，那么再往前退一位，返回 right - 1，就是最后一个小于目标值的数。

第二类应用实例：

Heaters，Arranging Coins，Valid Perfect Square，Max Sum of Rectangle No Larger Than K，Russian Doll Envelopes

第二类变形应用：Valid Triangle Number

7.1.8 第三类：查找第一个大于目标值的数，可变形为查找最后一个不大于目标值的数

这一类也比较常见，尤其是查找第一个大于目标值的数，在 C++ 的 STL 也有专门的函数 upper_bound，这里跟上面的那种情况的写法上很相似，只需要添加一个等号，将之前的 nums[mid] < target 变成 nums[mid] <= target，就这一个小小的变化，其实直接就改变了搜索的方向，使得在数组中有很多跟目标值相同的数字存在的情况下，返回最后一个相同的数字的下一个位置。比如在数组 [2, 4, 5, 6, 9] 中查找数字 3，还是返回数字 4 的位置，这跟上面那查找方式返回的结果相同，因为数字 4 在此数组中既是第一个不小于目标值 3 的数，也是第一个大于目标值 3 的数，所以 make sense；在数组 [0, 1, 1, 1, 1] 中查找数字 1，就会返回坐标 5，通过对比返回的坐标和数组的长度，我们就知道是否存在这样一个大于目标值的数。参见下面的代码：

```
int find(vector<int>& nums, int target) {
    int left = 0, right = nums.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] <= target) left = mid + 1;
        else right = mid;
    }
    return right;
}
```

这一类可以轻松的变形为查找最后一个不大于目标值的数，怎么变呢。我们已经找到了第一个大于目标值的数，那么再往前退一位，返回 right - 1，就是最后一个不大于目标值的数。比如在数组 [0, 1, 1, 1, 1] 中查找数字 1，就会返回最后一个数字 1 的位置 4，这在有些情况下是需要这么做的。

第三类应用实例：

Kth Smallest Element in a Sorted Matrix

第三类变形应用示例：

Sqrt(x)

7.1.9 第四类：用子函数当作判断关系（通常由 mid 计算得出）

这是最令博主头疼的一类，而且通常情况下都很难。因为这里在二分查找法重要的比较大小的地方使用到了子函数，并不是之前三类中简单的数字大小的比较，比如 Split Array Largest Sum 那道题中的解法一，就是根据是否能分割数组来确定下一步搜索的范围。类似的还有 Guess Number Higher or Lower 这道题，是根据给定函数 guess 的返回值情况来确定搜索的范围。对于这类题目，博主也很无奈，遇到了只能自求多福了。

第四类应用实例：

Split Array Largest Sum，Guess Number Higher or Lower，Find K Closest Elements，Find K-th Smallest Pair Distance，Kth Smallest Number in Multiplication Table，Maximum Average Subarray II，Minimize Max Distance to Gas Station，Swim in Rising Water，Koko Eating Bananas，Nth Magical Number

7.1.10 第五类：其他（通常 target 值不固定）

有些题目不属于上述的四类，但是还是需要用到二分搜索法，比如这道 Find Peak Element，求的是数组的局部峰值。由于是求的峰值，需要跟相邻的数字比较，那么 target 就不是一个固定的值，而且这道题的一定要注意的是 right 的初始化，一定要是 nums.size() - 1，这是由于算出了 mid 后，nums[mid] 要和 nums[mid+1] 比较，如果 right 初始化为 nums.size() 的话，mid+1 可能会越界，从而不能找到正确的值，同时 while 循环的终止条件必须是 left < right，不能有等号。

类似的还有一道 H-Index II，这道题的 target 也不是一个固定值，而是 len-mid，这就很意思了，跟上面的 nums[mid+1] 有异曲同工之妙，target 值都随着 mid 值的变化而变化，这里的 right 的初始化，一定要是 nums.size() - 1，而 while 循环的终止条件必须是 left <= right，这里又必须要有等号，是不是很头大 -.-!!!

其实仔细分析的话，可以发现其实这跟第四类还是比较相似，相似点是都很难 -.-!!!，第四类中虽然是用子函数来判断关系，但大部分时候 mid 也会作为一个参数带入子函数进行计算，这样实际上最终算出的值还是受 mid 的影响，但是 right 却可以初始化为数组长度，循环条件也可以不带等号，大家可以对比区别一下 ~

第五类应用实例：

Find Peak Element

H-Index II

7.2 793. Preimage Size of Factorial Zeroes Function

Let f(x) be the number of zeroes at the end of x!. Recall that x! = 1 * 2 * 3 * ... * x and by convention, 0! = 1.

For example, f(3) = 0 because 3! = 6 has no zeroes at the end, while f(11) = 2 because 11! = 39916800 has two zeroes at the end. Given an integer k, return the number of non-negative integers x have the property that f(x) = k.

```
private long numberOfTrailingZeros(long v) {
    long cnt = 0;
    for (; v > 0; v /= 5)
        cnt += v / 5;
    return cnt;
}
public int preimageSizeFZF(int k) {
    long left = 0, right = 5l * (k + 1);
    while (left < right) {
        long mid = left + (right - left) / 2;
        long cnt = numberOfTrailingZeros(mid);
        if (cnt == k) return 5;
        if (cnt < k) left = mid + 1;
        else right = mid;
    }
    return 0;
}
```

- 下面这种解法是把子函数融到了 while 循环内，使得看起来更加简洁一些，解题思路跟上面的解法一模一样，参见代码如下：

```
public int preimageSizeFZF(int k) {
    long left = 0, right = 5l * (k + 1);
    while (left < right) {
        long mid = left + (right - left) / 2, cnt = 0;
        for (long i = 5; mid / i > 0; i *= 5)
            cnt += mid / i;
        if (cnt == k) return 5;
        if (cnt < k) left = mid + 1;
        else right = mid;
    }
    return 0;
}
```

下面这种解法也挺巧妙的，也是根据观察规律推出来的，我们首先来看 x 为 1 到 25 的情况：

```
x:    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
f(x): 0 0 0 0 1 1 1 1 1 2  2  2  2  2  3  3  3  3  3  4  4  4  4  4  6
g(x): 0 0 0 0 1 0 0 0 0 1  0  0  0  0  1  0  0  0  0  1  0  0  0  0  2
```

这里，f(x) 是表示 x! 末尾零的个数，而 g(x) = f(x) - f(x-1)，其实还可以通过观察发现，f(x) = sum(g(x)).

再仔细观察上面的数字，发现 g(x) 有正值的时候都是当 x 是 5 的倍数的时候，那么来专门看一下 x 是 5 的倍数时的情况吧：

```
x:    5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105 110 115 120 125
g(x): 1 1  1  1  2  1  1  1  1  2  1  1  1  1  2  1  1  1  1   2   1   1   1   1   3
```

仔细观察上面的红色数字，g(x)=1 时，是 5 的倍数，g(x)=2 时，都是 25 的倍数，g(x)=3 时，是 125 的倍数，那么就有：

```
g(x) = 0     if x % 5 != 0,
g(x) >= 1    if x % 5 == 0,
g(x) >= 2   if x % 25 == 0.
```

如果继续将上面的数字写下去，就可以发现规律，g(x) 按照 1 1 1 1 x 的规律重复五次，第五次的时候 x 自增 1。再继续观察：

当 x=25 时，g(x)=2，此时 K=5 被跳过了。

当 x=50 时，g(x)=2，此时 K=11 被跳过了。

当 x=75 时，g(x)=2，此时 K=17 被跳过了。

当 x=100 时，g(x)=2，此时 K=23 被跳过了。

当 x=125 时，g(x)=3，此时 K=29，30 被跳过了。

进一步，可以发现如下规律：

5(=1*5), 11(=6*1+5), 17(=6*2+5), 23(=6*3+5), 29(=6*4+5), 30(=6*5), 36(=31+5), 42(=31+6+5), 48(=31+6*2+5)

这些使得 x 不存在的 K，出现都是有规律的，它们减去一个特定的基数 base 后，都是余 5，而余 1，2，3，4 的，都是返回 5。那么这个基数 base，实际是 1，6，31，156，...，是由 base = base * 5 + 1，不断构成的，通过这种不断对基数取余的操作，可以最终将 K 降为小于等于 5 的数，就可以直接返回结果了，参见代码如下：

```
public int preimageSizeFZF(int k) {
    if (k < 5) return 5;
    int base = 1;
    while (base * 5 + 1 <= k)
        base = base * 5 + 1;
    if (k / base == 5) return 0;
    return preimageSizeFZF(k % base);
}
```

7.3 2040. Kth Smallest Product of Two Sorted Arrays - Hard

Given two sorted 0-indexed integer arrays nums1 and nums2 as well as an integer k, return the kth (1-based) smallest product of nums1[i] * nums2[j] where 0 <= i < nums1.length and 0 <= j < nums2.length.

Thinking Process

- I can put the index pair for the two arrays in a priority queue and compute the answer gradually. However, the K can be up to 10^9. This will lead to TLE.
- The element can be negative. Maybe I need to know the number of negative elements and handle 4 different combinations: (negative array 1, negative array 2), (negative array 1, positive array 2), (positive array 1, negative array 2), (positive array 1, positive array 2). At least, I can know the number of products of each combination and locate k-th product among them.
- Even though I know which combination the k-th product belongs to, it doesn't guarantee I can use the priority queue approach. I need another hint.
- Continue with above, I think I need some way to eliminate some number of products step by step to reach the goal.
- Since the array is sorted, if I turn my attention on nums1[i] x nums2[j], I can know there are j + 1 products which are less than or equal to nums1[i] x nums2[j] that are generated by nums1[i]. Then I realize that I should try the binary search.

Algorithm

- Binary search the answer
- For each nums1[i], count the number of products that are less than or equal to the current guess

```
private static long INF = (long)1e10;
public long kthSmallestProduct(int[] a, int[] b, long k) {
    int m = a.length, n = b.length;
    long lo = -INF-1, hi = INF + 1;
    while (lo < hi) {
        long mid = lo + (hi - lo) / 2, cnt = 0;
        for (int  i : a) { // 对于数组 a 中的每一个数与 b 中元素的乘积，数 <=mid 的个数，二分搜索
            if (i >= 0) {
                int l = 0, r = n-1, p = 0;
                while (l <= r) {
                    int c = l + (r - l) / 2;
                    long mul = i * (long)b[c];
                    if (mul <= mid) {
                        p = c + 1;
                        l = c + 1;
                    } else r = c - 1;
                }
                cnt += p;
            } else { // i < 0
                int l = 0, r = n-1, p = 0;
                while (l <= r) {
                    int c = l + (r - l) / 2;
                    long mul = i * (long)b[c];
                    if (mul <= mid) {
                        p = n - c; // i < 0, 数右边 <= mid 的个数
                        r = c - 1;
                    } else l = c + 1;
                }
                cnt += p;
            }
        }
        if (cnt >= k) hi = mid;
        else lo = mid + 1l;
    }
    return lo;
}
```

- 另一种相当于换汤不换药的写法

```
private static long INF = (long)1e10;
public long kthSmallestProduct(int[] a, int[] b, long k) {
    long lo = -INF, hi = INF;
    while (lo < hi) {
        long mid = lo + hi + 1 >> 1;
        if (f(a, b, mid) < k) lo = mid;
        else hi = mid - 1;
    }
    return lo;
}
private long f(int [] a, int [] b, long mid) {
    long cnt = 0;
    for (int v : a) {
        int l = 0, r = b.length;
        if (v < 0) {
            while (l < r) {
                int m = l + r >> 1;
                if ((long)v * b[m] >= mid) l = m + 1;
                else r = m;
            }
            cnt += b.length - l;
        } else { // v >= 0
            while (l < r) {
                int m = l + r >> 1;
                if ((long)v * b[m] < mid) l = m + 1;
                else r = m;
            }
            cnt += l;
        }
    }
    return cnt;
}
```