第 339 场周赛

2609. 最长平衡子字符串

给你一个仅由 `0` 和 `1` 组成的二进制字符串 `s` 。

如果子字符串中 **所有的** `0` **都在** `1` **之前** 且其中 `0` 的数量等于 `1` 的数量，则认为 `s` 的这个子字符串是平衡子字符串。请注意，空子字符串也视作平衡子字符串。

返回 `s` 中最长的平衡子字符串长度。

子字符串是字符串中的一个连续字符序列。

示例 1：

> 输入：s = "01000111"
> 输出：6
> 解释：最长的平衡子字符串是 "000111" ，长度为 6 。

示例 2：

> 输入：s = "00111"
> 输出：4
> 解释：最长的平衡子字符串是 "0011" ，长度为 4 。

示例 3：

> 输入：s = "111"
> 输出：0
> 解释：除了空子字符串之外不存在其他平衡子字符串，所以答案为 0 。

提示：

- `1 <= s.length <= 50`
- `'0' <= s[i] <= '1'`

```
class Solution {
    public int findTheLongestBalancedSubstring(String s) {
        char[] str = s.toCharArray();
        int ans = 0;
        int zero = 0;
        int one = 0;
        int i = 0;
        while (i < str.length) {
            zero = 0;
            one = 0;
            while (i < str.length && str[i] == '0') {
                zero++;
                i++;
            }
```

```
            while (i < str.length && str[i] == '1') {
                one++;
                i++;
            }
            // System.out.println(zero + " " + one);
            ans = Math.max(ans, (int)Math.min(one, zero) * 2);
        }
        return ans;
    }
}
```

2610. 转换二维数组

给你一个整数数组 `nums` 。请你创建一个满足以下条件的二维数组：

- 二维数组应该 **只** 包含数组 `nums` 中的元素。
- 二维数组中的每一行都包含 **不同** 的整数。
- 二维数组的行数应尽可能 **少** 。

返回结果数组。如果存在多种答案，则返回其中任何一种。

请注意，二维数组的每一行上可以存在不同数量的元素。

示例 1：

> 输入：nums = [1,3,4,1,2,3,1]
> 输出：[[1,3,4,2],[1,3],[1]]
> 解释：根据题目要求可以创建包含以下几行元素的二维数组：
>
> - 1,3,4,2
> - 1,3
> - 1
>
> nums 中的所有元素都有用到，并且每一行都由不同的整数组成，所以这是一个符合题目要求的答案。
> 可以证明无法创建少于三行且符合题目要求的二维数组。

示例 2：

> 输入：nums = [1,2,3,4]
> 输出：[[4,3,2,1]]
> 解释：nums 中的所有元素都不同，所以我们可以将其全部保存在二维数组中的第一行。

提示：

- `1 <= nums.length <= 200`
- `1 <= nums[i] <= nums.length`

```
class Solution {
    public List<List<Integer>> findMatrix(int[] nums) {
        int[] hash = new int[nums.length + 1];
        List<List<Integer>> ans = new ArrayList<>();

        for (int i = 0; i < nums.length; i++) {
            hash[nums[i]]++;
```

```
        }

        for (int i = 1; i < hash.length; i++) {
            if (hash[i] == 0) {
                continue;
            }
            while (hash[i] > ans.size()) {
                ans.add(new ArrayList<>());
            }

            for (int j = 0; j < hash[i]; j++) {
                List<Integer> list = ans.get(j);
                list.add(i);
            }
        }
        return ans;
    }
}
```

2611. 老鼠和奶酪

有两只老鼠和 `n` 块不同类型的奶酪，每块奶酪都只能被其中一只老鼠吃掉。

下标为 `i` 处的奶酪被吃掉的得分为：

- 如果第一只老鼠吃掉，则得分为 `reward1[i]` 。
- 如果第二只老鼠吃掉，则得分为 `reward2[i]` 。

给你一个正整数数组 `reward1` ，一个正整数数组 `reward2` ，和一个非负整数 `k` 。

请你返回第一只老鼠恰好吃掉 `k` 块奶酪的情况下，**最大** 得分为多少。

示例 1：

> 输入：reward1 = [1,1,3,4], reward2 = [4,4,1,1], k = 2
> 输出：15
> 解释：这个例子中，第一只老鼠吃掉第 2 和 3 块奶酪（下标从 0 开始），第二只老鼠吃掉第 0 和 1 块奶酪。
> 总得分为 4 + 4 + 3 + 4 = 15 。
> 15 是最高得分。

示例 2：

> 输入：reward1 = [1,1], reward2 = [1,1], k = 2
> 输出：2
> 解释：这个例子中，第一只老鼠吃掉第 0 和 1 块奶酪（下标从 0 开始），第二只老鼠不吃任何奶酪。
> 总得分为 1 + 1 = 2 。
> 2 是最高得分。

提示：

- `1 <= n == reward1.length == reward2.length <= 105`

- `1 <= reward1[i], reward2[i] <= 1000`
- `0 <= k <= n`

```
class Solution {
    public int miceAndCheese(int[] reward1, int[] reward2, int k) {
        PriorityQueue<Node> pq = new PriorityQueue<>((x, y) -> (y.key - x.key));
        HashSet<Integer> set = new HashSet<>();
        int ans = 0;

        for (int i = 0; i < reward1.length; i++) {
            Node node = new Node(reward1[i] - reward2[i], i);
            pq.add(node);
        }

        for (int i = 0; i < k; i++) {
            Node node = pq.poll();
            ans += reward1[node.idx];
            set.add(node.idx);
        }

        for (int i = 0; i < reward2.length; i++) {
            if (set.contains(i)) {
                continue;
            }
            ans += reward2[i];
        }
        return ans;
    }
}

class Node {
    int key;
    int idx;

    public Node(int key, int idx) {
        this.key = key;
        this.idx = idx;
    }
}
```

2612. 最少翻转操作数

给你一个整数 `n` 和一个在范围 `[0, n - 1]` 以内的整数 `p` ，它们表示一个长度为 `n` 且下标从 **0** 开始的数组 `arr` ，数组中除了下标为 `p` 处是 `1` 以外，其他所有数都是 `0` 。

同时给你一个整数数组 `banned` ，它包含数组中的一些位置。 `banned` 中第 i 个位置表示 `arr[banned[i]] = 0` ，题目保证 `banned[i] != p` 。

你可以对 `arr` 进行 **若干次** 操作。一次操作中，你选择大小为 `k` 的一个 **子数组** ，并将它 **翻转** 。在任何一次翻转操作后，你都需要确保 `arr` 中唯一的 `1` 不会到达任何 `banned` 中的位置。换句话说，`arr[banned[i]]` 始终 **保持** `0` 。

请你返回一个数组 `ans` ，对于 `[0, n - 1]` 之间的任意下标 `i` ， `ans[i]` 是将 `1` 放到位置 `i` 处的 **最少** 翻转操作次数，如果无法放到位置 `i` 处，此数为 `-1` 。

- **子数组** 指的是一个数组里一段连续 **非空** 的元素序列。
- 对于所有的 `i` ， `ans[i]` 相互之间独立计算。

- 将一个数组中的元素 **翻转** 指的是将数组中的值变成 **相反顺序** 。

示例 1：

> 输入：n = 4, p = 0, banned = [1,2], k = 4
> 输出：[0,-1,-1,1]
> 解释：k = 4，所以只有一种可行的翻转操作，就是将整个数组翻转。一开始 1 在位置 0 处，所以将它翻转到位置 0 处需要的操作数为 0 。
> 我们不能将 1 翻转到 banned 中的位置，所以位置 1 和 2 处的答案都是 -1 。
> 通过一次翻转操作，可以将 1 放到位置 3 处，所以位置 3 的答案是 1 。

示例 2：

> 输入：n = 5, p = 0, banned = [2,4], k = 3
> 输出：[0,-1,-1,-1,-1]
> 解释：这个例子中 1 一开始在位置 0 处，所以此下标的答案为 0 。
> 翻转的子数组长度为 k = 3 ， 1 此时在位置 0 处，所以我们可以翻转子数组 [0, 2]，但翻转后的下标 2 在 banned 中，所以不能执行此操作。
> 由于 1 没法离开位置 0 ，所以其他位置的答案都是 -1 。

示例 3：

> 输入：n = 4, p = 2, banned = [0,1,3], k = 1
> 输出：[-1,-1,0,-1]
> 解释：这个例子中，我们只能对长度为 1 的子数组执行翻转操作，所以 1 无法离开初始位置。

提示：

- `1 <= n <= 105`
- `0 <= p <= n - 1`
- `0 <= banned.length <= n - 1`
- `0 <= banned[i] <= n - 1`
- `1 <= k <= n`
- `banned[i] != p`
- `banned` 中的值 **互不相同** 。

```
class Solution {
    public int[] minReverseOperations(int n, int p, int[] banned, int k) {
        boolean[] check = new boolean[n];
        for (int i = 0; i < banned.length; i++) {
            check[banned[i]] = true;
        }

        Queue<int[]> queue = new LinkedList<>();
        queue.offer(new int[]{p, 0});
        int[] res = new int[n];
        for (int i = 0; i < n; i++) {
            res[i] = -1;
        }
        res[p] = 0;
        UnionFind unionFind = new UnionFind(n);
```

```
        while (!queue.isEmpty()) {
            int[] po = queue.poll();
            int l = Math.max(0, po[0] - k + 1);
            int r = Math.min(n - 1, po[0] + k - 1);
            int len = (l + k - 1) - po[0];
            int realL = l + len;
            len = r - po[0];
            int realR = (r - k + 1) + len;
            while (realR >= realL) {
                int nextIdx = realR;
                if (res[nextIdx] == -1 && !check[nextIdx]) {
                    queue.offer(new int[]{nextIdx, po[1] + 1});
                    res[nextIdx] = po[1] + 1;
                }
                if (realR - 2 < realL) {
                    break;
                }
                unionFind.add(realR, realR - 2);
                realR = unionFind.find(realR);
            }
        }
        return res;
    }
}

class UnionFind {
    int[] parent;

    public UnionFind(int n) {
        parent = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    public void add(int a, int b) {
        int rA = find(a);
        int rB = find(b);
        if (rA < rB) {
            parent[rB] = rA;
        } else {
            parent[rA] = rB;
        }
    }

    public int find(int a) {
        if (parent[a] == a) {
            return a;
        }
        return parent[a] = find(parent[a]);
    }
}
```

第 101 场双周赛

2605. 从两个数字数组里生成最小数字

给你两个只包含 1 到 9 之间数字的数组 `nums1` 和 `nums2` ，每个数组中的元素 **互不相同** ，请你返回 **最小** 的数字，两个数组都 **至少** 包含这个数字的某个数位。

示例 1：

> 输入：nums1 = [4,1,3], nums2 = [5,7]
> 输出：15
> 解释：数字 15 的数位 1 在 nums1 中出现，数位 5 在 nums2 中出现。15 是我们能得到的最小数字。

示例 2：

> 输入：nums1 = [3,5,2,6], nums2 = [3,1,7]
> 输出：3
> 解释：数字 3 的数位 3 在两个数组中都出现了。

提示：

- `1 <= nums1.length, nums2.length <= 9`
- `1 <= nums1[i], nums2[i] <= 9`
- 每个数组中，元素 **互不相同** 。

```
class Solution {
    public boolean isFind(int[] arr, int num) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == num) {
                return true;
            }
        }
        return false;
    }
    public int minNumber(int[] nums1, int[] nums2) {
        int ans = 1;
        while (true) {
            int tmp = ans;
            boolean flag1 = false;
            boolean flag2 = false;
            while (tmp != 0) {
                int bit = tmp % 10;
                // System.out.println(tmp +  " " + bit);
                if (isFind(nums1, bit)) {
                    flag1 = true;
                }
                if (isFind(nums2, bit)) {
                    flag2 = true;
                }
                tmp /= 10;
            }
            if (flag1 && flag2) {
                return ans;
            }
            ans++;
        }
```

```
    }
}
```

2606. 找到最大开销的子字符串

给你一个字符串 `s` ，一个字符 **互不相同** 的字符串 `chars` 和一个长度与 `chars` 相同的整数数组 `vals` 。

子字符串的开销 是一个子字符串中所有字符对应价值之和。空字符串的开销是 `0` 。

字符的价值 定义如下：

- 如果字符不在字符串 chars 中，那么它的价值是它在字母表中的位置（下标从 1 开始）。
    - 比方说， `'a'` 的价值为 `1` ， `'b'` 的价值为 `2` ，以此类推， `'z'` 的价值为 `26` 。
- 否则，如果这个字符在 `chars` 中的位置为 `i` ，那么它的价值就是 `vals[i]` 。

请你返回字符串 `s` 的所有子字符串中的最大开销。

示例 1：

> 输入：s = "adaa", chars = "d", vals = [-1000]
> 输出：2
> 解释：字符 "a" 和 "d" 的价值分别为 1 和 -1000 。
> 最大开销子字符串是 "aa" ，它的开销为 1 + 1 = 2 。
> 2 是最大开销。

示例 2：

> 输入：s = "abc", chars = "abc", vals = [-1,-1,-1]
> 输出：0
> 解释：字符 "a" ，"b" 和 "c" 的价值分别为 -1 ，-1 和 -1 。
> 最大开销子字符串是 "" ，它的开销为 0 。
> 0 是最大开销。

提示：

- `1 <= s.length <= 105`
- `s` 只包含小写英文字母。
- `1 <= chars.length <= 26`
- `chars` 只包含小写英文字母，且 **互不相同** 。
- `vals.length == chars.length`
- `-1000 <= vals[i] <= 1000`

```
class Solution {
    public int maximumCostSubstring(String s, String chars, int[] vals) {
        HashMap<Character, Integer> map = new HashMap<>();
        for (int i = 0; i < chars.length(); i++) {
            map.put(chars.charAt(i), i);
        }

        char[] str = s.toCharArray();
        int ans = 0;
```

```
        int preSum = 0;
        for (int i = 0; i < str.length; i++) {
            int val;
            int idx = map.getOrDefault(str[i], -1);
            // System.out.println(i + " " + idx);
            if (idx == -1) {
                val = str[i] - 'a' + 1;
            } else {
                val = vals[idx];
            }
            preSum = Math.max(preSum, 0) + val;
            ans = Math.max(ans, preSum);
        }
        return ans;
    }
}
```

2607. 使子数组元素和相等

给你一个下标从 **0** 开始的整数数组 `arr` 和一个整数 `k` 。数组 `arr` 是一个循环数组。换句话说，数组中的最后一个元素的下一个元素是数组中的第一个元素，数组中第一个元素的前一个元素是数组中的最后一个元素。

你可以执行下述运算任意次：

- 选中 `arr` 中任意一个元素，并使其值加上 `1` 或减去 `1` 。

执行运算使每个长度为 `k` 的 **子数组** 的元素总和都相等，返回所需要的最少运算次数。

子数组 是数组的一个连续部分。

示例 1：

> 输入：arr = [1,4,1,3], k = 2
> 输出：1
> 解释：在下标为 1 的元素那里执行一次运算，使其等于 3 。
> 执行运算后，数组变为 [1,3,1,3] 。
>
> - 0 处起始的子数组为 [1, 3] ，元素总和为 4
> - 1 处起始的子数组为 [3, 1] ，元素总和为 4
> - 2 处起始的子数组为 [1, 3] ，元素总和为 4
> - 3 处起始的子数组为 [3, 1] ，元素总和为 4

示例 2：

> 输入：arr = [2,5,5,7], k = 3
> 输出：5
> 解释：在下标为 0 的元素那里执行三次运算，使其等于 5 。在下标为 3 的元素那里执行两次运算，使其等于 5 。
> 执行运算后，数组变为 [5,5,5,5] 。
>
> - 0 处起始的子数组为 [5, 5, 5] ，元素总和为 15
> - 1 处起始的子数组为 [5, 5, 5] ，元素总和为 15
> - 2 处起始的子数组为 [5, 5, 5] ，元素总和为 15
> - 3 处起始的子数组为 [5, 5, 5] ，元素总和为 15

提示：

- `1 <= k <= arr.length <= 105`
- `1 <= arr[i] <= 109`

```
class Solution {
    public long makeSubKSumEqual(int[] arr, int k) {
        UnionFind uf = new UnionFind(arr.length);
        for (int i = 0; i < arr.length; i++) {
            uf.union(i, (i + k) % arr.length); // i 和 i + k 的元素是同一组，同组元素都必须
相等
        }

        HashMap<Integer, ArrayList<Integer>> map = new HashMap<>();
        for (int i = 0; i < arr.length; i++) {
            int key = uf.find(i);
            ArrayList<Integer> list = map.getOrDefault(key, null);
            if (list == null) {
                list = new ArrayList<>();
            }
            list.add(arr[i]);
            map.put(key, list);
        }

        long ans = 0;
        for (Integer key : map.keySet()) {
            ArrayList<Integer> list =  (ArrayList<Integer>)map.get(key);
            Collections.sort(list);
            int mid = list.size() / 2;
            int target = list.get(mid);
            for (Integer i : list) {
                ans += Math.abs(target - i);
            }
        }
        return ans;
    }
}
class UnionFind {
    int[] parent;
    int[] rank;

    public UnionFind(int len) {
        parent = new int[len];
        rank = new int[len];
        for (int i = 0; i < len; i++) {
            parent[i] = i;
            rank[i] = 1;
        }
    }

    public int find(int p) {
        if (parent[p] == p) {
            return p;
        }
        return parent[p] = find(parent[p]);
    }
```

```
    public void union(int p, int q) {
        int pRoot = find(p);
        int qRoot = find(q);
        if (pRoot == qRoot) {
            return;
        }
        if (rank[pRoot] < rank[qRoot]) {
            parent[pRoot] = qRoot;
        } else {
            parent[qRoot] = pRoot;
            rank[pRoot]++;
        }
    }
}
```

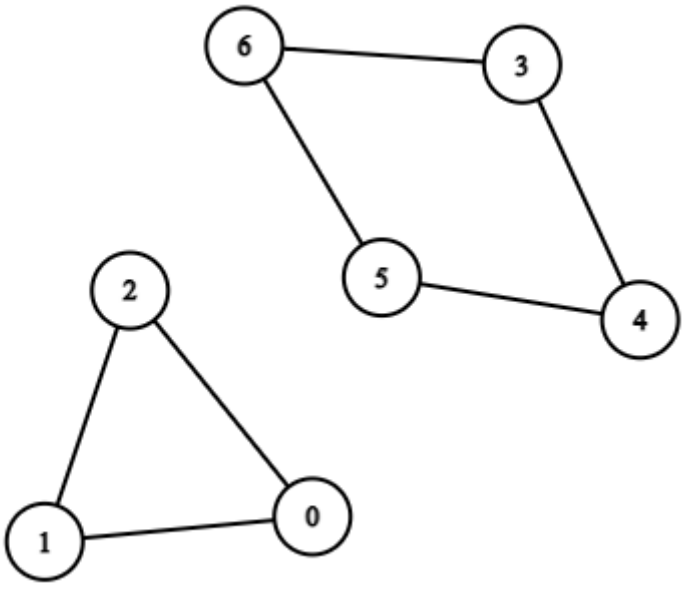
2608. 图中的最短环

现有一个含 `n` 个顶点的 **双向** 图，每个顶点按从 `0` 到 `n - 1` 标记。图中的边由二维整数数组 `edges` 表示，其中 `edges[i] = [ui, vi]` 表示顶点 `ui` 和 `vi` 之间存在一条边。每对顶点最多通过一条边连接，并且不存在与自身相连的顶点。

返回图中 **最短** 环的长度。如果不存在环，则返回 `-1` 。

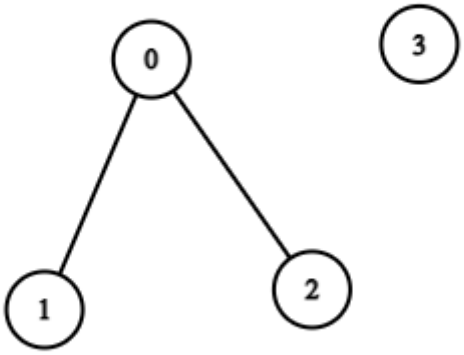
环 是指以同一节点开始和结束，并且路径中的每条边仅使用一次。

示例 1：



> 输入：n = 7, edges = [[0,1],[1,2],[2,0],[3,4],[4,5],[5,6],[6,3]]
> 输出：3
> 解释：长度最小的循环是：0 -> 1 -> 2 -> 0

示例 2：



> 输入：n = 4, edges = [[0,1],[0,2]]
> 输出：-1
> 解释：图中不存在循环

提示：

- `2 <= n <= 1000`
- `1 <= edges.length <= 1000`
- `edges[i].length == 2`
- `0 <= ui, vi < n`
- `ui != vi`
- 不存在重复的边

```
class Solution {
    public int dijkstra(ArrayList<Integer>[] graph, int start, int end, int n) {
        int[] distance = new int[n];
        for (int i = 0; i < n; i++) {
            distance[i] = Integer.MAX_VALUE;
        }

        distance[start] = 0;
        PriorityQueue<Node> pq = new PriorityQueue<>((a, b) -> (a.distance -
b.distance));
        pq.add(new Node(start, 0));
        while (!pq.isEmpty()) {
            Node cur = pq.poll();
            if (distance[cur.x] != cur.distance) {
                continue;
            }
            ArrayList<Integer> list = graph[cur.x];
            for (int i = 0; i < list.size(); i++) {
                int val = list.get(i);
                if ((cur.x == start && val == end)||
                    (cur.x == end && val == start)) {
                    continue;
                }
```

```
                if (distance[cur.x] + 1 < distance[val]) {
                    distance[val] = distance[cur.x] + 1;
                    pq.add(new Node(val, distance[cur.x] + 1));
                }
            }
        }
        return distance[end] == Integer.MAX_VALUE ? -1 : distance[end];
    }

    public int findShortestCycle(int n, int[][] edges) {
        ArrayList<Integer>[] graph = new ArrayList[n];
        for (int i = 0; i < n; i++) {
            graph[i] = new ArrayList<>();
        }
        for (int[] edge : edges) {
            graph[edge[0]].add(edge[1]);
            graph[edge[1]].add(edge[0]);
        }
        int min = Integer.MAX_VALUE;
        for (int[] edge : edges) {
            int distance = dijkstra(graph, edge[0], edge[1], n);
            if (distance != -1 && distance < min) {
                min = distance;
            }
        }
        return min == Integer.MAX_VALUE ? -1 : min + 1;
    }
}

class Node {
    int x;
    int distance;

    public Node(int x, int distance) {
        this.x = x;
        this.distance = distance;
    }
}
```

第 338 场周赛

2600. K 件物品的最大和

袋子中装有一些物品，每个物品上都标记着数字 `1` 、 `0` 或 `-1` 。

给你四个非负整数 `numOnes` 、 `numZeros` 、 `numNegOnes` 和 `k` 。

袋子最初包含：

- `numOnes` 件标记为 `1` 的物品。
- `numZeroes` 件标记为 `0` 的物品。
- `numNegOnes` 件标记为 `-1` 的物品。

现计划从这些物品中恰好选出 `k` 件物品。返回所有可行方案中，物品上所标记数字之和的最大值。

示例 1：

> 输入：numOnes = 3, numZeros = 2, numNegOnes = 0, k = 2
> 输出：2
> 解释：袋子中的物品分别标记为 {1, 1, 1, 0, 0} 。取 2 件标记为 1 的物品，得到的数字之和为 2 。
> 可以证明 2 是所有可行方案中的最大值。

示例 2：

> 输入：numOnes = 3, numZeros = 2, numNegOnes = 0, k = 4
> 输出：3
> 解释：袋子中的物品分别标记为 {1, 1, 1, 0, 0} 。取 3 件标记为 1 的物品，1 件标记为 0 的物品，得到的数字之和为 3 。
> 可以证明 3 是所有可行方案中的最大值。

提示：

- `0 <= numOnes, numZeros, numNegOnes <= 50`
- `0 <= k <= numOnes + numZeros + numNegOnes`

```
class Solution {
    public int kItemsWithMaximumSum(int numOnes, int numZeros, int numNegOnes, int k)
{
        int ans = 0;
        if (numOnes >= k) {
            return k;
        }
        ans += numOnes;
        k -= numOnes;
        if (numZeros >= k) {
            return ans;
        }
        k -= numZeros;
        return ans - k;
    }
}
```

2601. 质数减法运算

给你一个下标从 **0** 开始的整数数组 `nums` ，数组长度为 `n` 。

你可以执行无限次下述运算：

- 选择一个之前未选过的下标 `i` ，并选择一个 **严格小于** `nums[i]` 的质数 `p` ，从 `nums[i]` 中减去 `p` 。

如果你能通过上述运算使得 `nums` 成为严格递增数组，则返回 `true` ；否则返回 `false` 。

严格递增数组 中的每个元素都严格大于其前面的元素。

示例 1：

> 输入：nums = [4,9,6,10]
> 输出：true
> 解释：
> 在第一次运算中：选择 i = 0 和 p = 3 ，然后从 nums[0] 减去 3 ，nums 变为 [1,9,6,10] 。
> 在第二次运算中：选择 i = 1 和 p = 7 ，然后从 nums[1] 减去 7 ，nums 变为 [1,2,6,10] 。
> 第二次运算后，nums 按严格递增顺序排序，因此答案为 true 。

示例 2：

> 输入：nums = [6,8,11,12]
> 输出：true
> 解释：nums 从一开始就按严格递增顺序排序，因此不需要执行任何运算。

示例 3：

> 输入：nums = [5,8,3]
> 输出：false
> 解释：可以证明，执行运算无法使 nums 按严格递增顺序排序，因此答案是 false 。

提示：

- `1 <= nums.length <= 1000`
- `1 <= nums[i] <= 1000`
- `nums.length == n`

```
class Solution {
    public boolean isPrime(int num) {
        for (int i = 2; i * i <= num; i++) {
            if (num % i == 0) {
                return false;
            }
        }
        return true;
    }
    public int getNumsI(int[] ) {
        int tmp = 1001;
        for (Integer prime : arr) {
            if (nums[0] - prime > 0) {
                tmp = Math.min(tmp, nums[0] - prime);
            }
        }
        return
    }
    public boolean primeSubOperation(int[] nums) {
        ArrayList<Integer> arr = new ArrayList<>();
        for (int i = 2; i <= 1000; i++) {
            if (isPrime(i)) {
                arr.add(i);
            }
        }


        nums[0] = Math.min(tmp, nums[0]);
```

```
        System.out.println(nums[0]);
        for (int i = 1; i < nums.length; i++) {
            tmp = 1001;
            for (Integer prime : arr) {
                if (nums[i] - prime > nums[i - 1]) {
                    tmp = Math.min(tmp, nums[i] - prime);
                }
            }
            nums[i] = Math.min(tmp, nums[i]);
            // System.out.println(i + " " + nums[i] + " " + nums.length);
            if (nums[i] <= nums[i - 1]) {
                return false;
            }
        }
        return true;
    }
}
```

2602. 使数组元素全部相等的最少操作次数

给你一个正整数数组 `nums` 。

同时给你一个长度为 `m` 的整数数组 `queries` 。第 `i` 个查询中，你需要将 `nums` 中所有元素变成 `queries[i]` 。你可以执行以下操作 **任意** 次：

- 将数组里一个元素 **增大** 或者 **减小** `1` 。

请你返回一个长度为 `m` 的数组 `answer` ，其中 `answer[i]` 是将 `nums` 中所有元素变成 `queries[i]` 的 **最少** 操作次数。

注意，每次查询后，数组变回最开始的值。

示例 1：

> 输入：nums = [3,1,6,8], queries = [1,5]
> 输出：[14,10]
> 解释：第一个查询，我们可以执行以下操作：
>
> - 将 nums[0] 减小 2 次，nums = [1,1,6,8] 。
> - 将 nums[2] 减小 5 次，nums = [1,1,1,8] 。
> - 将 nums[3] 减小 7 次，nums = [1,1,1,1] 。
>
> 第一个查询的总操作次数为 2 + 5 + 7 = 14 。
> 第二个查询，我们可以执行以下操作：
>
> - 将 nums[0] 增大 2 次，nums = [5,1,6,8] 。
> - 将 nums[1] 增大 4 次，nums = [5,5,6,8] 。
> - 将 nums[2] 减小 1 次，nums = [5,5,5,8] 。
> - 将 nums[3] 减小 3 次，nums = [5,5,5,5] 。
>
> 第二个查询的总操作次数为 2 + 4 + 1 + 3 = 10 。

示例 2：

> 输入：nums = [2,9,6,3], queries = [10]
> 输出：[20]
> 解释：我们可以将数组中所有元素都增大到 10 ， 总操作次数为 8 + 1 + 4 + 7 = 20 。

提示：

- `n == nums.length`
- `m == queries.length`
- `1 <= n, m <= 105`
- `1 <= nums[i], queries[i] <= 109`

```
class Solution {
    public int binarySearch(int[] arr, int l, int r, int target) {
        while (l < r) {
        int mid = l + (r - l) / 2;
        if (target <= arr[mid])
            r = mid;
        else
            l = mid + 1;
        }
        return r;
    }
    public List<Long> minOperations(int[] nums, int[] queries) {
        int n = nums.length;
        Arrays.sort(nums);
        ArrayList<Long> ans = new ArrayList<>(queries.length);
        long[] prefix = new long[nums.length + 1];
        prefix[0] = 0;
        for (int i = 1; i < prefix.length; i++) {
            prefix[i] = prefix[i - 1] + nums[i - 1];
        }
        long sum = prefix[n];

        for (int query : queries) {
        int i = binarySearch(nums, 0, n, query);
        long leftSum = (long)query * i - prefix[i];
        long rightSum = sum - prefix[i] - (long)query * (n - i);
        ans.add(leftSum + rightSum);
        }
        return ans;
    }
}
```

2603. 收集树中金币

给你一个 `n` 个节点的无向无根树，节点编号从 `0` 到 `n - 1` 。给你整数 `n` 和一个长度为 `n - 1` 的二维整数数组 `edges` ，其中 `edges[i] = [ai, bi]` 表示树中节点 `ai` 和 `bi` 之间有一条边。再给你一个长度为 `n` 的数组 `coins` ，其中 `coins[i]` 可能为 `0` 也可能为 `1` ， `1` 表示节点 `i` 处有一个金币。

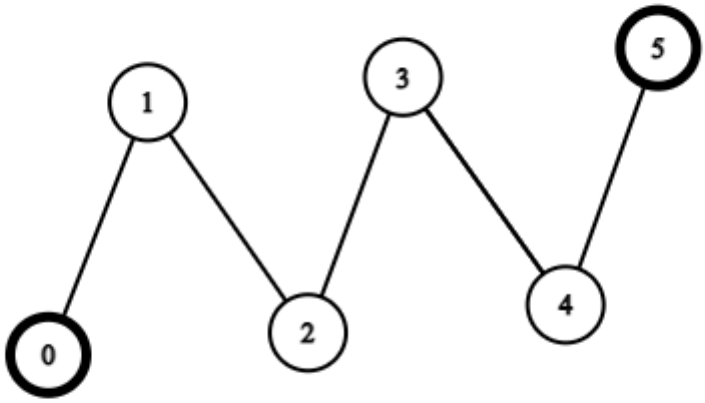
一开始，你需要选择树中任意一个节点出发。你可以执行下述操作任意次：

- 收集距离当前节点距离为 `2` 以内的所有金币，或者
- 移动到树中一个相邻节点。

你需要收集树中所有的金币，并且回到出发节点，请你返回最少经过的边数。

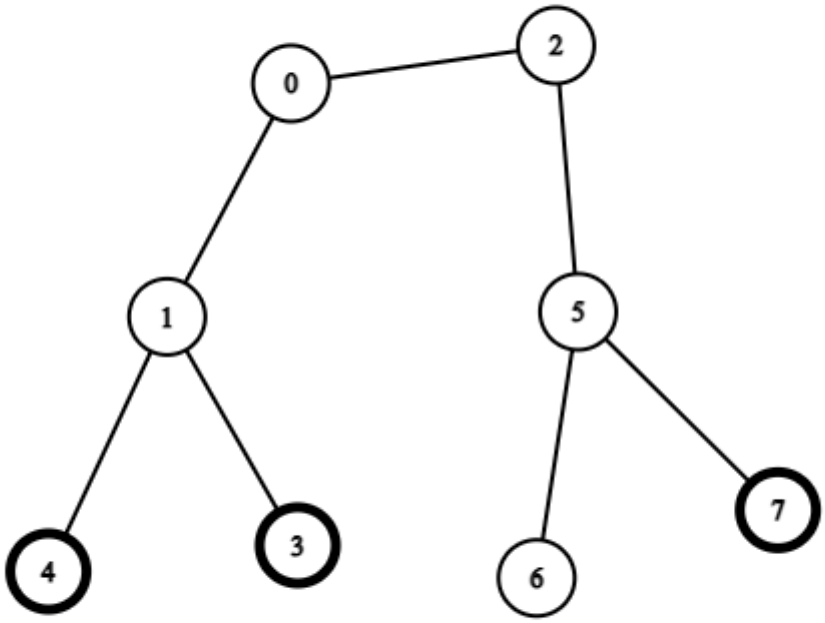
如果你多次经过一条边，每一次经过都会给答案加一。

示例 1：



> 输入：coins = [1,0,0,0,0,1], edges = [[0,1],[1,2],[2,3],[3,4],[4,5]]
> 输出：2
> 解释：从节点 2 出发，收集节点 0 处的金币，移动到节点 3 ，收集节点 5 处的金币，然后移动回节点 2 。

示例 2：



> 输入：coins = [0,0,0,1,1,0,0,1], edges = [[0,1],[0,2],[1,3],[1,4],[2,5],[5,6],[5,7]]
> 输出：2
> 解释：从节点 0 出发，收集节点 4 和 3 处的金币，移动到节点 2 处，收集节点 7 处的金币，移动回节点 0 。

提示：

- `n == coins.length`
- `1 <= n <= 3 * 10^4`
- `0 <= coins[i] <= 1`
- `edges.length == n - 1`
- `edges[i].length == 2`
- `0 <= ai, bi < n`
- `ai != bi`
- `edges` 表示一棵合法的树。

```
class Solution {
    public int collectTheCoins(int[] coins, int[][] edges) {
        ArrayList<Integer>[] graph = new ArrayList[coins.length];
        for (int i = 0; i < graph.length; i++) {
            graph[i] = new ArrayList<>();
        }
        int[] deg = new int[coins.length];
```

```
        for (int i = 0; i < edges.length; i++) {
            graph[edges[i][0]].add(edges[i][1]);
            graph[edges[i][1]].add(edges[i][0]);
            deg[edges[i][0]]++;
            deg[edges[i][1]]++;
        }

        // 无向图拓扑排序, 去掉没有金币的子树
        Queue<Integer> que = new LinkedList<>();
        for (int i = 0; i < coins.length; i++) {
            if (deg[i] == 1 && coins[i] == 0) {
                que.add(i);
            }
        }
        while (!que.isEmpty()) {
            int u = que.poll();
            for (int i = 0; i < graph[u].size(); i++) {
                int v = graph[u].get(i);
                if (--deg[v] == 1 && coins[v] == 0) {
                    que.add(v);
                }
            }
        }

        for (int i = 0; i < coins.length; i++) {
            if (deg[i] == 1 && coins[i] == 1) {
                que.add(i);
            }
        }
        if (que.size() <= 1) {
            return 0;
        }
        int[] times = new int[coins.length];
        while (!que.isEmpty()) {
            int u = que.poll();
            for (int i = 0; i < graph[u].size(); i++) {
                int v = graph[u].get(i);
                if (--deg[v] == 1) {
                    times[v] = times[u] + 1;
                    que.add(v);
                }
            }
        }
        int ans = 0;
        for (int i = 0; i < edges.length; i++) {
            if (times[edges[i][0]] >= 2 && times[edges[i][1]] >= 2) {
                ans += 2;
            }
        }
        return ans;
    }
}
```

第 337 场周赛

2595. 奇偶位数

给你一个 **正** 整数 `n` 。

用 `even` 表示在 `n` 的二进制形式（下标从 **0** 开始）中值为 `1` 的偶数下标的个数。

用 `odd` 表示在 `n` 的二进制形式（下标从 **0** 开始）中值为 `1` 的奇数下标的个数。

返回整数数组 `answer` ，其中 `answer = [even, odd]` 。

示例 1：

> 输入：n = 17
> 输出：[2,0]
> 解释：17 的二进制形式是 10001 。
> 下标 0 和 下标 4 对应的值为 1 。
> 共有 2 个偶数下标，0 个奇数下标。

示例 2：

> 输入：n = 2
> 输出：[0,1]
> 解释：2 的二进制形式是 10 。
> 下标 1 对应的值为 1 。
> 共有 0 个偶数下标，1 个奇数下标。

提示：

- `1 <= n <= 1000`

```
class Solution {
    public int[] evenOddBit(int n) {
        int[] arr = new int[32];
        int size = 0;

        while (n != 0) {
            arr[size++] = n % 2;
            n /= 2;
        }

        int[] ans = new int[2];
        int even = 0;
        int odd = 0;

        for (int i = 0; i < size; i++) {
            if (arr[i] == 1) {
                if (i % 2 == 0) {
                    even++;
                } else {
                    odd++;
                }
            }
        }
        ans[0] = even;
```

```
            ans[1] = odd;
            return ans;
        }
    }
```

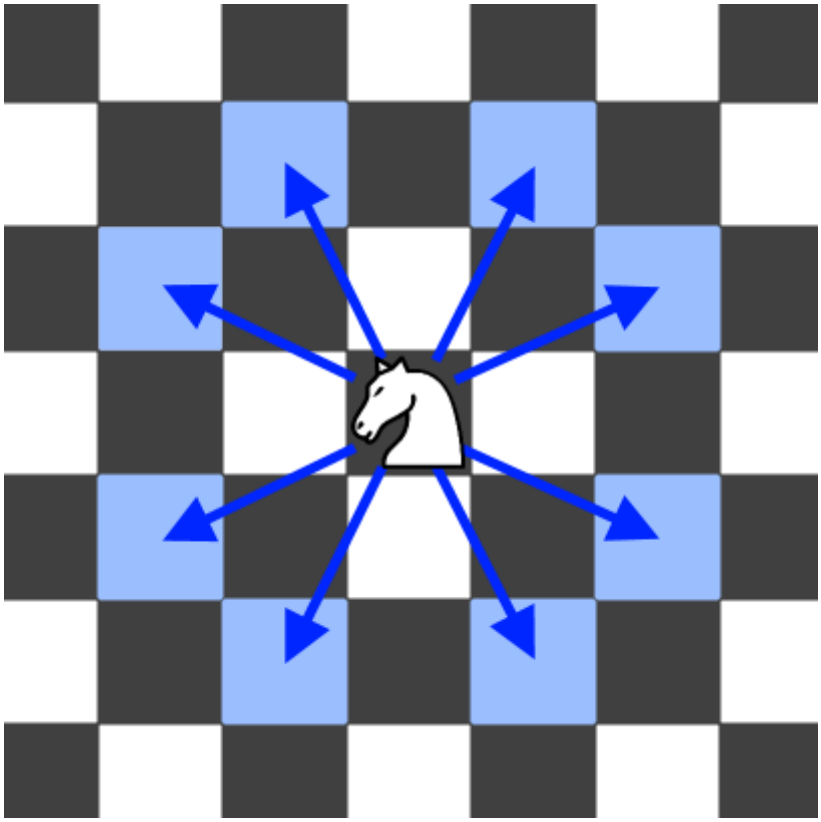
2596. 检查骑士巡视方案

骑士在一张 `n x n` 的棋盘上巡视。在有效的巡视方案中，骑士会从棋盘的 **左上角** 出发，并且访问棋盘上的每个格子 **恰好一次** 。

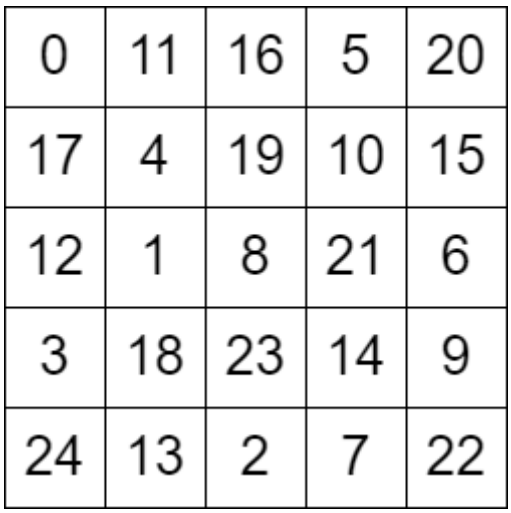
给你一个 `n x n` 的整数矩阵 `grid` ，由范围 `[0, n * n - 1]` 内的不同整数组成，其中 `grid[row][col]` 表示单元格 `(row, col)` 是骑士访问的第 `grid[row][col]` 个单元格。骑士的行动是从下标 0 开始的。

如果 `grid` 表示了骑士的有效巡视方案，返回 `true`；否则返回 `false`。

注意，骑士行动时可以垂直移动两个格子且水平移动一个格子，或水平移动两个格子且垂直移动一个格子。下图展示了骑士从某个格子出发可能的八种行动路线。



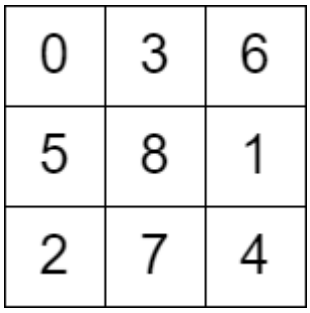
示例 1：



0	11	16	5	20
17	4	19	10	15
12	1	8	21	6
3	18	23	14	9
24	13	2	7	22

输入：grid = [[0,11,16,5,20],[17,4,19,10,15],[12,1,8,21,6],[3,18,23,14,9],[24,13,2,7,22]]
输出：true
解释：grid 如上图所示，可以证明这是一个有效的巡视方案。

示例 2：



0	3	6
5	8	1
2	7	4

输入：grid = [[0,3,6],[5,8,1],[2,7,4]]
输出：false
解释：grid 如上图所示，考虑到骑士第 7 次行动后的位置，第 8 次行动是无效的。

提示：

- `n == grid.length == grid[i].length`
- `3 <= n <= 7`
- `0 <= grid[row][col] < n * n`
- `grid` 中的所有整数 **互不相同**

```
class Solution {
    public boolean backTrack(int[][] grid, int i, int x, int y) {
        if (x < 0 || x >= grid.length || y < 0 || y >= grid.length) {
            return false;
        }

        if (grid[x][y] != i) {
            return false;
        }

        if (i == grid.length * grid.length - 1) {
            return true;
        }
```

```
        int[] arrX = new int[]{-2, -1, -2, -1, 1, 2, 1, 2};
        int[] arrY = new int[]{1, 2, -1, -2, 2, 1, -2, -1};

        boolean res = false;
        for (int j = 0; j < arrX.length; j++) {
            res |= backTrack(grid, i + 1, x + arrX[j], y + arrY[j]);
        }

        return res;
    }
    public boolean checkValidGrid(int[][] grid) {
        return backTrack(grid, 0, 0, 0);
    }
}
```

2597. 美丽子集的数目

给你一个由正整数组成的数组 `nums` 和一个 **正** 整数 `k` 。

如果 `nums` 的子集中，任意两个整数的绝对差均不等于 `k` ，则认为该子数组是一个 **美丽** 子集。

返回数组 `nums` 中 **非空** 且 **美丽** 的子集数目。

`nums` 的子集定义为：可以经由 `nums` 删除某些元素（也可能不删除）得到的一个数组。只有在删除元素时选择的索引不同的情况下，两个子集才会被视作是不同的子集。

示例 1：

> 输入：nums = [2,4,6], k = 2
> 输出：4
> 解释：数组 nums 中的美丽子集有：[2], [4], [6], [2, 6] 。
> 可以证明数组 [2,4,6] 中只存在 4 个美丽子集。

示例 2：

> 输入：nums = [1], k = 1
> 输出：1
> 解释：数组 nums 中的美丽数组有：[1] 。
> 可以证明数组 [1] 中只存在 1 个美丽子集。

提示：

- `1 <= nums.length <= 20`
- `1 <= nums[i], k <= 1000`

```
class Solution {
    HashMap<Integer, Integer> map;
    int ans;
    public void backTrack(int[] nums, int i, int k) {
        if (i == nums.length) {
            ans++;
            return;
        }
```

```
        backTrack(nums, i + 1, k);
        if ((map.getOrDefault(nums[i] - k, 0) == 0) && (map.getOrDefault(nums[i] + k,
0) == 0)) {
            int val = map.getOrDefault(nums[i], 0);
            map.put(nums[i], val + 1);
            backTrack(nums, i + 1, k);
            map.put(nums[i], val);
        }
    }

    public int beautifulSubsets(int[] nums, int k) {
        map = new HashMap<>();
        ans = -1; // 去掉空集
        backTrack(nums, 0, k);
        return ans;
    }
}
```

2598. 执行操作后的最大 MEX

给你一个下标从 0 开始的整数数组 `nums` 和一个整数 `value` 。

在一步操作中，你可以对 `nums` 中的任一元素加上或减去 `value` 。

- 例如，如果 `nums = [1,2,3]` 且 `value = 2` ，你可以选择 `nums[0]` 减去 `value` ，得到 `nums = [-1,2,3]` 。

数组的 MEX (minimum excluded) 是指其中数组中缺失的最小非负整数。

- 例如， `[-1,2,3]` 的 MEX 是 `0` ，而 `[1,0,3]` 的 MEX 是 `2` 。

返回在执行上述操作 **任意次** 后， `nums` 的最大 MEX 。

示例 1：

> 输入：nums = [1,-10,7,13,6,8], value = 5
> 输出：4
> 解释：执行下述操作可以得到这一结果：
>
> - nums[1] 加上 value 两次，nums = [1,0,7,13,6,8]
> - nums[2] 减去 value 一次，nums = [1,0,2,13,6,8]
> - nums[3] 减去 value 两次，nums = [1,0,2,3,6,8]
>
> nums 的 MEX 是 4 。可以证明 4 是可以取到的最大 MEX 。

示例 2：

> 输入：nums = [1,-10,7,13,6,8], value = 7
> 输出：2
> 解释：执行下述操作可以得到这一结果：
>
> - nums[2] 减去 value 一次，nums = [1,-10,0,13,6,8]
>
> nums 的 MEX 是 2 。可以证明 2 是可以取到的最大 MEX 。

提示：

- `1 <= nums.length, value <= 105`
- `-109 <= nums[i] <= 109`

```
class Solution {
    public int findSmallestInteger(int[] nums, int value) {
        HashMap<Integer, Integer> map = new HashMap<>();

        for (int i = 0; i < nums.length; i++) {
            int num = (nums[i] % value + value) % value;
            int val = map.getOrDefault(num, 0);
            map.put(num, val + 1);
        }

        for (int i = 0; i <= (int)1e9; i++) {
            int val = map.getOrDefault(i % value, 0);
            if (val > 0) {
                map.put(i % value, val - 1);
            } else {
               return i;
            }
        }

        return -1;
    }
}
```

第 100 场双周赛

2591. 将钱分给最多的儿童

给你一个整数 `money` ，表示你总共有的钱数（单位为美元）和另一个整数 `children` ，表示你要将钱分配给多少个儿童。

你需要按照如下规则分配：

- 所有的钱都必须被分配。
- 每个儿童至少获得 `1` 美元。
- 没有人获得 `4` 美元。

请你按照上述规则分配金钱，并返回 **最多** 有多少个儿童获得 **恰好** `8` 美元。如果没有任何分配方案，返回 `-1` 。

示例 1：

> 输入：money = 20, children = 3
> 输出：1
> 解释：
> 最多获得 8 美元的儿童数为 1 。一种分配方案为：
>
> - 给第一个儿童分配 8 美元。
> - 给第二个儿童分配 9 美元。

- 给第三个儿童分配 3 美元。
  没有分配方案能让获得 8 美元的儿童数超过 1 。

示例 2：

输入：money = 16, children = 2
输出：2
解释：每个儿童都可以获得 8 美元。

提示：

- `1 <= money <= 200`
- `2 <= children <= 30`

```
class Solution {
    public int distMoney(int money, int children) {
        int ans = 0;
        if (money < children) {
            return -1;
        }
        money = money - children;
        ans = money / 7;
        if ((ans > children) || (ans == children && money % 7 != 0)) {
            return children - 1;
        }
        if (money % 7 == 3 && ans == children - 1) {
            ans -= 1;
        }
        return ans;
    }
}
```

2592. 最大化数组的伟大值

给你一个下标从 0 开始的整数数组 `nums` 。你需要将 `nums` 重新排列成一个新的数组 `perm` 。

定义 `nums` 的 **伟大值** 为满足 `0 <= i < nums.length` 且 `perm[i] > nums[i]` 的下标数目。

请你返回重新排列 `nums` 后的 **最大** 伟大值。

示例 1：

输入：nums = [1,3,5,2,1,3,1]
输出：4
解释：一个最优安排方案为 perm = [2,5,1,3,3,1,1] 。
在下标为 0, 1, 3 和 4 处，都有 perm[i] > nums[i] 。因此我们返回 4 。

示例 2：

输入：nums = [1,2,3,4]
输出：3
解释：最优排列为 [2,3,4,1] 。
在下标为 0, 1 和 2 处，都有 perm[i] > nums[i] 。因此我们返回 3 。

提示：

- `1 <= nums.length <= 105`
- `0 <= nums[i] <= 109`

```
class Solution {
    public int maximizeGreatness(int[] nums) {
        if (nums.length == 1) {
            return 0;
        }
        Arrays.sort(nums);
        int l = 0;
        for (int r = 1; r < nums.length; r++) {
            if (nums[l] < nums[r]) {
               l++;
            }
        }
        return l;
    }
}
```

2593. 标记所有元素后数组的分数

给你一个数组 `nums` ，它包含若干正整数。

一开始分数 `score = 0` ，请你按照下面算法求出最后分数：

- 从数组中选择最小且没有被标记的整数。如果有相等元素，选择下标最小的一个。
- 将选中的整数加到 `score` 中。
- 标记 **被选中元素**，如果有相邻元素，则同时标记 **与它相邻的两个元素** 。
- 重复此过程直到数组中所有元素都被标记。

请你返回执行上述算法后最后的分数。

示例 1：

> 输入：nums = [2,1,3,4,5,2]
> 输出：7
> 解释：我们按照如下步骤标记元素：
>
> - 1 是最小未标记元素，所以标记它和相邻两个元素：[2,1,3,4,5,2] 。
> - 2 是最小未标记元素，所以标记它和左边相邻元素：[2,1,3,4,5,2] 。
> - 4 是仅剩唯一未标记的元素，所以我们标记它：[2,1,3,4,5,2] 。
>
> 总得分为 1 + 2 + 4 = 7 。

示例 2：

> 输入：nums = [2,3,5,1,3,2]
> 输出：5
> 解释：我们按照如下步骤标记元素：
>
> - 1 是最小未标记元素，所以标记它和相邻两个元素：[2,3,5,1,3,2] 。

- 2 是最小未标记元素，由于有两个 2 ，我们选择最左边的一个 2 ，也就是下标为 0 处的 2 ，以及它右边相邻的元素：[2,3,5,1,3,2] 。
- 2 是仅剩唯一未标记的元素，所以我们标记它：[2,3,5,1,3,2] 。
  总得分为 1 + 2 + 2 = 5 。

提示：

- `1 <= nums.length <= 105`
- `1 <= nums[i] <= 106`

```
class Solution {
    public long findScore(int[] nums) {
        boolean[] flag = new boolean[nums.length];
        HashMap<Integer, ArrayList<Integer>> map = new HashMap<>();
        long ans = 0;

        for (int i = 0; i < nums.length; i++) {
            ArrayList<Integer> val = map.getOrDefault(nums[i], null);
            if (val == null) {
                val = new ArrayList<>();
            }
            val.add(i);
            map.put(nums[i], val);
        }

        Arrays.sort(nums);
        for (int i = 0; i < nums.length; i++) {
            ArrayList<Integer> val = map.getOrDefault(nums[i], null);
            int idx = val.get(0);
            val.remove(0);
            map.put(nums[i], val);
            if (flag[idx]) {
                continue;
            }

            flag[idx] = true;
            ans += nums[i];
            if (idx - 1 >= 0) {
                flag[idx - 1] = true;
            }
            if (idx + 1 < nums.length) {
                flag[idx + 1] = true;
            }
        }
        return ans;
    }
}
```

2594. 修车的最少时间

给你一个整数数组 `ranks` ，表示一些机械工的 **能力值** 。 `ranksi` 是第 `i` 位机械工的能力值。能力值为 `r` 的机械工可以在 `r * n2` 分钟内修好 `n` 辆车。

同时给你一个整数 `cars` ，表示总共需要修理的汽车数目。

请你返回修理所有汽车 **最少** 需要多少时间。

注意：所有机械工可以同时修理汽车。

示例 1：

> 输入：ranks = [4,2,3,1], cars = 10
> 输出：16
> 解释：
>
> - 第一位机械工修 2 辆车，需要 4 * 2 * 2 = 16 分钟。
> - 第二位机械工修 2 辆车，需要 2 * 2 * 2 = 8 分钟。
> - 第三位机械工修 2 辆车，需要 3 * 2 * 2 = 12 分钟。
> - 第四位机械工修 4 辆车，需要 1 * 4 * 4 = 16 分钟。
>
> 16 分钟是修理完所有车需要的最少时间。

示例 2：

> 输入：ranks = [5,1,8], cars = 6
> 输出：16
> 解释：
>
> - 第一位机械工修 1 辆车，需要 5 * 1 * 1 = 5 分钟。
> - 第二位机械工修 4 辆车，需要 1 * 4 * 4 = 16 分钟。
> - 第三位机械工修 1 辆车，需要 8 * 1 * 1 = 8 分钟。
>
> 16 分钟时修理完所有车需要的最少时间。

提示：

- `1 <= ranks.length <= 105`
- `1 <= ranks[i] <= 100`
- `1 <= cars <= 106`

```
class Solution {
    public long repairCars(int[] ranks, int cars) {
        long ans = 0;
        PriorityQueue<Pair> pq = new PriorityQueue<>((x, y) -> ((int)(x.time -
y.time)));

        for (int i = 0; i < ranks.length; i++) {
            pq.add(new Pair(ranks[i], 1, ranks[i]));
        }
        for (int i = 0; i < cars; i++) {
            Pair pair = pq.poll();
            ans = pair.time;
            // System.out.println(pair.r + " " + pair.n + " " + ans);
            long tmp = (long)Math.pow(pair.n + 1, 2);
            pq.add(new Pair(pair.r, pair.n + 1, (long)pair.r * tmp));
        }
        return ans;
    }
}

class Pair {
    int r;
```

```
    int n;
    long time;

    public Pair(int r, int n, long time) {
        this.r = r;
        this.n = n;
        this.time = time;
    }
}
```

第 336 场周赛

2586. 统计范围内的元音字符串数

给你一个下标从 0 开始的字符串数组 `words` 和两个整数：`left` 和 `right` 。

如果字符串以元音字母开头并以元音字母结尾，那么该字符串就是一个 **元音字符串** ，其中元音字母是 `'a'` 、`'e'` 、`'i'` 、`'o'` 、`'u'` 。

返回 `words[i]` 是元音字符串的数目，其中 `i` 在闭区间 `[left, right]` 内。

示例 1：

> 输入：words = ["are","amy","u"], left = 0, right = 2
> 输出：2
> 解释：
>
> - "are" 是一个元音字符串，因为它以 'a' 开头并以 'e' 结尾。
> - "amy" 不是元音字符串，因为它没有以元音字母结尾。
> - "u" 是一个元音字符串，因为它以 'u' 开头并以 'u' 结尾。
>
> 在上述范围中的元音字符串数目为 2 。

示例 2：

> 输入：words = ["hey","aeo","mu","ooo","artro"], left = 1, right = 4
> 输出：3
> 解释：
>
> - "aeo" 是一个元音字符串，因为它以 'a' 开头并以 'o' 结尾。
> - "mu" 不是元音字符串，因为它没有以元音字母开头。
> - "ooo" 是一个元音字符串，因为它以 'o' 开头并以 'o' 结尾。
> - "artro" 是一个元音字符串，因为它以 'a' 开头并以 'o' 结尾。
>
> 在上述范围中的元音字符串数目为 3 。

提示：

- `1 <= words.length <= 1000`
- `1 <= words[i].length <= 10`
- `words[i]` 仅由小写英文字母组成
- `0 <= left <= right < words.length`

```
class Solution {
    public boolean isVowel(char c) {
        char[] str = new char[]{'a', 'e', 'i', 'o', 'u'};
        for (int i = 0; i < str.length; i++) {
            if (c == str[i]) {
                return true;
            }
        }
        return false;
    }

    public int vowelStrings(String[] words, int left, int right) {
        int ans = 0;
        for (int i = left; i <= right; i++) {
            char[] str = words[i].toCharArray();
            if (isVowel(str[0]) && isVowel(str[str.length - 1])) {
                ans++;
            }
        }
        return ans;
    }
}
```

2587. 重排数组以得到最大前缀分数

给你一个下标从 0 开始的整数数组 `nums` 。你可以将 `nums` 中的元素按 **任意顺序** 重排（包括给定顺序）。

令 `prefix` 为一个数组，它包含了 `nums` 重新排列后的前缀和。换句话说， `prefix[i]` 是 `nums` 重新排列后下标从 `0` 到 `i` 的元素之和。 `nums` 的 **分数** 是 `prefix` 数组中正整数的个数。

返回可以得到的最大分数。

示例 1：

> 输入：nums = [2,-1,0,1,-3,3,-3]
> 输出：6
> 解释：数组重排为 nums = [2,3,1,-1,-3,0,-3] 。
> prefix = [2,5,6,5,2,2,-1] ， 分数为 6 。
> 可以证明 6 是能够得到的最大分数。

示例 2：

> 输入：nums = [-2,-3,0]
> 输出：0
> 解释：不管怎么重排数组得到的分数都是 0 。

提示：

- `1 <= nums.length <= 105`
- `-106 <= nums[i] <= 106`

```
class Solution {
```

```
    public int maxScore(int[] nums) {
        Arrays.sort(nums);

        long cnt = 0;
        int ans = 0;
        for (int i = nums.length - 1; i >= 0; i--) {
            if (cnt + nums[i] > 0) {
                ans++;
                // System.out.println("i " + ans);
                cnt += nums[i];
            }
        }

        return ans;
    }
}
```

2588. 统计美丽子数组数目

给你一个下标从 **0** 开始的整数数组 `nums` 。每次操作中，你可以：

- 选择两个满足 `0 <= i, j < nums.length` 的不同下标 `i` 和 `j` 。
- 选择一个非负整数 `k` ，满足 `nums[i]` 和 `nums[j]` 在二进制下的第 `k` 位（下标编号从 **0** 开始）是 `1` 。
- 将 `nums[i]` 和 `nums[j]` 都减去 `2k` 。

如果一个子数组内执行上述操作若干次后，该子数组可以变成一个全为 `0` 的数组，那么我们称它是一个 **美丽** 的子数组。

请你返回数组 `nums` 中 **美丽子数组** 的数目。

子数组是一个数组中一段连续 **非空** 的元素序列。

示例 1：

> 输入：nums = [4,3,1,2,4]
> 输出：2
> 解释：nums 中有 2 个美丽子数组：[4,3,1,2,4] 和 [4,3,1,2,4] 。
>
> - 按照下述步骤，我们可以将子数组 [3,1,2] 中所有元素变成 0 ：
>   - 选择 [3, 1, 2] 和 k = 1 。将 2 个数字都减去 21 ，子数组变成 [1, 1, 0] 。
>   - 选择 [1, 1, 0] 和 k = 0 。将 2 个数字都减去 20 ，子数组变成 [0, 0, 0] 。
> - 按照下述步骤，我们可以将子数组 [4,3,1,2,4] 中所有元素变成 0 ：
>   - 选择 [4, 3, 1, 2, 4] 和 k = 2 。将 2 个数字都减去 22 ，子数组变成 [0, 3, 1, 2, 0] 。
>   - 选择 [0, 3, 1, 2, 0] 和 k = 0 。将 2 个数字都减去 20 ，子数组变成 [0, 2, 0, 2, 0] 。
>   - 选择 [0, 2, 0, 2, 0] 和 k = 1 。将 2 个数字都减去 21 ，子数组变成 [0, 0, 0, 0, 0] 。

示例 2：

> 输入：nums = [1,10,4]
> 输出：0
> 解释：nums 中没有任何美丽子数组。

提示：

- `1 <= nums.length <= 105`
- `0 <= nums[i] <= 106`

```
class Solution {
    public long beautifulSubarrays(int[] nums) {
        HashMap<Integer, Integer> map = new HashMap<>();
        map.put(0, 1); // 如果前缀的异或和是0， 不需要再在它之前找同样为0的数都可以构成美丽子数组

        int sum = 0;
        long ans = 0;
        for (int i = 0; i < nums.length; i++) {
            sum ^= nums[i];
            int cnt = map.getOrDefault(sum, 0); // sum和i之前的cnt个前缀和为sum的数两两组合可以使得子数组是美丽子数组
            ans += cnt;
            map.put(sum, cnt + 1);
        }
        return ans;
    }
}
```

2589. 完成所有任务的最少时间

你有一台电脑，它可以 **同时** 运行无数个任务。给你一个二维整数数组 `tasks` ，其中 `tasks[i] = [starti, endi, durationi]` 表示第 `i` 个任务需要在 **闭区间** 时间段 `[starti, endi]` 内运行 `durationi` 个整数时间点（但不需要连续）。

当电脑需要运行任务时，你可以打开电脑，如果空闲时，你可以将电脑关闭。

请你返回完成所有任务的情况下，电脑最少需要运行多少秒。

示例 1：

> 输入：tasks = [[2,3,1],[4,5,1],[1,5,2]]
> 输出：2
> 解释：
>
> - 第一个任务在闭区间 [2, 2] 运行。
> - 第二个任务在闭区间 [5, 5] 运行。
> - 第三个任务在闭区间 [2, 2] 和 [5, 5] 运行。
>
> 电脑总共运行 2 个整数时间点。

示例 2：

> 输入：tasks = [[1,3,2],[2,5,3],[5,6,2]]
> 输出：4
> 解释：
>
> - 第一个任务在闭区间 [2, 3] 运行
> - 第二个任务在闭区间 [2, 3] 和 [5, 5] 运行。
> - 第三个任务在闭区间 [5, 6] 运行。
>
> 电脑总共运行 4 个整数时间点。

提示：

- `1 <= tasks.length <= 2000`
- `tasks[i].length == 3`
- `1 <= starti, endi <= 2000`
- `1 <= durationi <= endi - starti + 1`

```
class Solution {
    public int findMinimumTime(int[][] tasks) {
        int ans = 0;
        boolean[] isVisited = new boolean[2001];

        Arrays.sort(tasks, (x, y) -> (x[1] - y[1]));
        for (int i = 0; i < tasks.length; i++) {
            for (int j = tasks[i][1]; j >= tasks[i][0]; j--) {
                if (tasks[i][2] <= 0 || isVisited[j]) {
                    continue;
                }
                tasks[i][2]--;
                isVisited[j] = true;
                ans++;
                for (int k = i + 1; k < tasks.length; k++) {
                    if (tasks[k][0] > j || tasks[k][2] <= 0) {
                        continue;
                    }
                    tasks[k][2]--;
                }
            }
        }

        for (int i = 0; i < tasks.length; i++) {
            if (tasks[i][2] > 0) {
                ans += tasks[i][2];
            }
        }
        return ans;
    }
}
```

第 335 场周赛

2582. 递枕头

`n` 个人站成一排，按从 `1` 到 `n` 编号。

最初，排在队首的第一个人拿着一个枕头。每秒钟，拿着枕头的人会将枕头传递给队伍中的下一个人。一旦枕头到达队首或队尾，传递方向就会改变，队伍会继续沿相反方向传递枕头。

- 例如，当枕头到达第 `n` 个人时，TA 会将枕头传递给第 `n - 1` 个人，然后传递给第 `n - 2` 个人，依此类推。

给你两个正整数 `n` 和 `time` ，返回 `time` 秒后拿着枕头的人的编号。

示例 1：

> 输入：n = 4, time = 5
> 输出：2
> 解释：队伍中枕头的传递情况为：1 -> 2 -> 3 -> 4 -> 3 -> 2 。
> 5 秒后，枕头传递到第 2 个人手中。

示例 2：

> 输入：n = 3, time = 2
> 输出：3
> 解释：队伍中枕头的传递情况为：1 -> 2 -> 3 。
> 2 秒后，枕头传递到第 3 个人手中。

提示：

- `2 <= n <= 1000`
- `1 <= time <= 1000`

```
class Solution {
    public int passThePillow(int n, int time) {
        int ans = 1;
        boolean isAdd = true;
        for (int i = 0; i < time; i++) {
            if (ans < n && isAdd) {
                ans++;
                if (ans == n) {
                    isAdd = false;
                }
            } else if (ans > 1 && !isAdd) {
                ans--;
                if (ans == 1) {
                    isAdd = true;
                }
            }
        }
        return ans;
    }
}
```

2583. 二叉树中的第 K 大层和

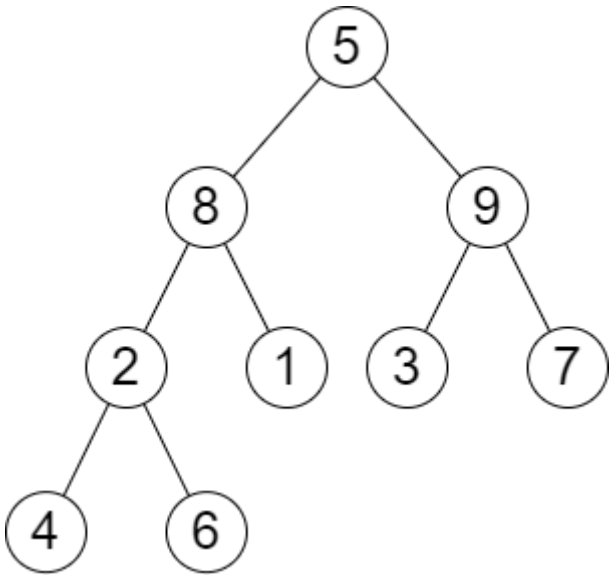
给你一棵二叉树的根节点 `root` 和一个正整数 `k` 。

树中的 **层和** 是指 **同一层** 上节点值的总和。

返回树中第 `k` 大的层和（不一定不同）。如果树少于 `k` 层，则返回 `-1` 。

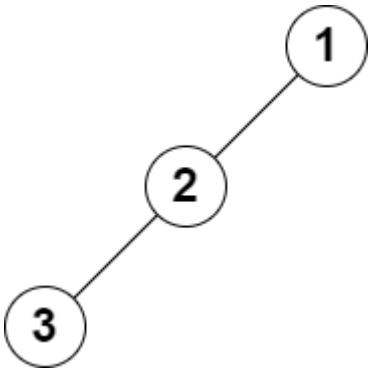
注意，如果两个节点与根节点的距离相同，则认为它们在同一层。

示例 1：



> 输入：root = [5,8,9,2,1,3,7,4,6], k = 2
> 输出：13
> 解释：树中每一层的层和分别是：
>
> - Level 1: 5
> - Level 2: 8 + 9 = 17
> - Level 3: 2 + 1 + 3 + 7 = 13
> - Level 4: 4 + 6 = 10
>
> 第 2 大的层和等于 13 。

示例 2：



> 输入：root = [1,2,null,3], k = 1
> 输出：3
> 解释：最大的层和是 3 。

提示：

- 树中的节点数为 `n`
- `2 <= n <= 105`
- `1 <= Node.val <= 106`
- `1 <= k <= n`

```
class Solution {
    public long kthLargestLevelSum(TreeNode root, int k) {
        long[] arr = new long[(int)1e5];
        int sizeOfArr = 0;
```

```
        ArrayList<TreeNode> lists = new ArrayList<>();
        lists.add(root);
        while (lists.size() > 0) {
            int size = lists.size();
            long sum = 0;
            for (int i = 0; i < size; i++) {
                TreeNode node = lists.remove(0);
                if (node.left != null) {
                    lists.add(node.left);
                }
                if (node.right != null) {
                    lists.add(node.right);
                }
                sum += node.val;
            }

            arr[sizeOfArr++] = sum;
        }

        if (k > sizeOfArr) {
            return -1;
        }
        long[] newArr = new long[sizeOfArr];
        for (int i = 0; i < sizeOfArr; i++) {
            newArr[i] = arr[i];
        }
        Arrays.sort(newArr);
        return newArr[sizeOfArr - k];
    }
}
```

2584. 分割数组使乘积互质

给你一个长度为 `n` 的整数数组 `nums` ，下标从 **0** 开始。

如果在下标 `i` 处 **分割** 数组，其中 `0 <= i <= n - 2` ，使前 `i + 1` 个元素的乘积和剩余元素的乘积互质，则认为该分割 **有效** 。

- 例如，如果 `nums = [2, 3, 3]` ，那么在下标 `i = 0` 处的分割有效，因为 `2` 和 `9` 互质，而在下标 `i = 1` 处的分割无效，因为 `6` 和 `3` 不互质。在下标 `i = 2` 处的分割也无效，因为 `i == n - 1` 。

返回可以有效分割数组的最小下标 `i` ，如果不存在有效分割，则返回 `-1` 。

当且仅当 `gcd(val1, val2) == 1` 成立时， `val1` 和 `val2` 这两个值才是互质的，其中 `gcd(val1, val2)` 表示 `val1` 和 `val2` 的最大公约数。

示例 1：

index	prefixproduct	suffixproduct	gcd
0	4	12600	4
1	28	1800	4
2	224	225	1
3	3360	15	15
4	10080	5	5

> 输入：nums = [4,7,8,15,3,5]
> 输出：2
> 解释：上表展示了每个下标 i 处的前 i + 1 个元素的乘积、剩余元素的乘积和它们的最大公约数的值。
> 唯一一个有效分割位于下标 2 。

示例 2：

index	prefixproduct	suffixproduct	gcd
0	4	12600	4
1	28	1800	4
2	420	120	60
3	3360	15	15
4	10080	5	5

> 输入：nums = [4,7,15,8,3,5]
> 输出：-1
> 解释：上表展示了每个下标 i 处的前 i + 1 个元素的乘积、剩余元素的乘积和它们的最大公约数的值。
> 不存在有效分割。

提示：

- `n == nums.length`
- `1 <= n <= 104`
- `1 <= nums[i] <= 106`

```
class Solution {
    public int findValidSplit(int[] nums) {
        int len = (int)1e6 + 1;
```

```
        HashMap<Integer, Integer> map = new HashMap<>();
        int[] right = new int[nums.length];
        Arrays.fill(right, -1);

        for (int i = 0; i < nums.length; i++) {
            for (int j = 2; j * j <= nums[i]; j++) {
                if (nums[i] % j == 0) {
                    int leftIdx = map.getOrDefault(j, -1);
                    // System.out.println(leftIdx);
                    if (leftIdx == -1) {
                        map.put(j, i);
                    } else {
                        right[leftIdx] = i;
                    }
                    nums[i] /= j;
                }
            }
            if (nums[i] > 1) {
                int leftIdx = map.getOrDefault(nums[i], -1);
                if (leftIdx == -1) {
                    map.put(nums[i], i);
                } else {
                    right[leftIdx] = i;
                }
            }
        }
        int ans = 0;
        for (int i = 0; i < nums.length; i++) {
            if (i > ans) {
                return ans;
            }
            ans = Math.max(ans, right[i]);
        }
        return -1;
    }
}
```

2585. 获得分数的方法数

考试中有 `n` 种类型的题目。给你一个整数 `target` 和一个下标从 0 开始的二维整数数组 `types` ，其中 `types[i] = [counti, marksi]` 表示第 `i` 种类型的题目有 `counti` 道，每道题目对应 `marksi` 分。

返回你在考试中恰好得到 `target` 分的方法数。由于答案可能很大，结果需要对 `109 +7` 取余。

注意，同类型题目无法区分。

- 比如说，如果有 `3` 道同类型题目，那么解答第 `1` 和第 `2` 道题目与解答第 `1` 和第 `3` 道题目或者第 `2` 和第 `3` 道题目是相同的。

示例 1：

> 输入：target = 6, types = [[6,1],[3,2],[2,3]]
> 输出：7
> 解释：要获得 6 分，你可以选择以下七种方法之一：

- 解决 6 道第 0 种类型的题目：1 + 1 + 1 + 1 + 1 + 1 = 6
- 解决 4 道第 0 种类型的题目和 1 道第 1 种类型的题目：1 + 1 + 1 + 1 + 2 = 6
- 解决 2 道第 0 种类型的题目和 2 道第 1 种类型的题目：1 + 1 + 2 + 2 = 6
- 解决 3 道第 0 种类型的题目和 1 道第 2 种类型的题目：1 + 1 + 1 + 3 = 6
- 解决 1 道第 0 种类型的题目、1 道第 1 种类型的题目和 1 道第 2 种类型的题目：1 + 2 + 3 = 6
- 解决 3 道第 1 种类型的题目：2 + 2 + 2 = 6
- 解决 2 道第 2 种类型的题目：3 + 3 = 6

示例 2：

输入：target = 5, types = [[50,1],[50,2],[50,5]]
输出：4
解释：要获得 5 分，你可以选择以下四种方法之一：

- 解决 5 道第 0 种类型的题目：1 + 1 + 1 + 1 + 1 = 5
- 解决 3 道第 0 种类型的题目和 1 道第 1 种类型的题目：1 + 1 + 1 + 2 = 5
- 解决 1 道第 0 种类型的题目和 2 道第 1 种类型的题目：1 + 2 + 2 = 5
- 解决 1 道第 2 种类型的题目：5

示例 3：

输入：target = 18, types = [[6,1],[3,2],[2,3]]
输出：1
解释：只有回答所有题目才能获得 18 分。

提示：

- `1 <= target <= 1000`
- `n == types.length`
- `1 <= n <= 50`
- `types[i].length == 2`
- `1 <= counti, marksi <= 50`

```
class Solution {
    public int waysToReachTarget(int target, int[][] types) {
        int[][] dp = new int[types.length + 1][target + 1];

        for (int i = 0; i < dp.length; i++) {
            dp[i][0] = 1;
        }

        for (int i = 1; i < dp.length; i++) {
            for (int j = 1; j <= target; j++) {
                for (int k = 0; k <= types[i - 1][0]; k++) {
                    // System.out.println(i +  " " + j + " " + k + " " + (k * types[i
- 1][1]));
                    if (j - k * types[i - 1][1] < 0) {
                        dp[i][j] = dp[i][j];
                    } else {
                        dp[i][j] = (dp[i][j] + dp[i - 1][j - types[i - 1][1] * k]) %
((int)1e9 + 7);
                    }
                }
```

```
            }
        }

        // for (int i = 0; i < dp.length; i++) {
        //     for (int j = 0; j < dp[0].length; j++) {
        //         System.out.print(dp[i][j] + " ");
        //     }
        //     System.out.println();
        // }
        return dp[dp.length - 1][target];
    }
}
```

第 099 场双周赛

2578. 最小和分割

给你一个正整数 `num` ，请你将它分割成两个非负整数 `num1` 和 `num2` ，满足：

- `num1` 和 `num2` 直接连起来，得到 num 各数位的一个排列。
  - 换句话说， `num1` 和 `num2` 中所有数字出现的次数之和等于 `num` 中所有数字出现的次数。
- `num1` 和 `num2` 可以包含前导 0 。

请你返回 `num1` 和 `num2` 可以得到的和的 **最小** 值。

注意：

- `num` 保证没有前导 0 。
- `num1` 和 `num2` 中数位顺序可以与 `num` 中数位顺序不同。

示例 1：

> 输入：num = 4325
> 输出：59
> 解释：我们可以将 4325 分割成 num1 = 24 和 num2 = 35 ，和为 59 ，59 是最小和。

示例 2：

> 输入：num = 687
> 输出：75
> 解释：我们可以将 687 分割成 num1 = 68 和 num2 = 7 ，和为最优值 75 。

提示：

- `10 <= num <= 109`

```
class Solution {
    public int splitNum(int num) {
        int[] arr = new int[32];
        int size = 0;
        while (num > 0) {
            arr[size++] = num % 10;
            num /= 10;
```

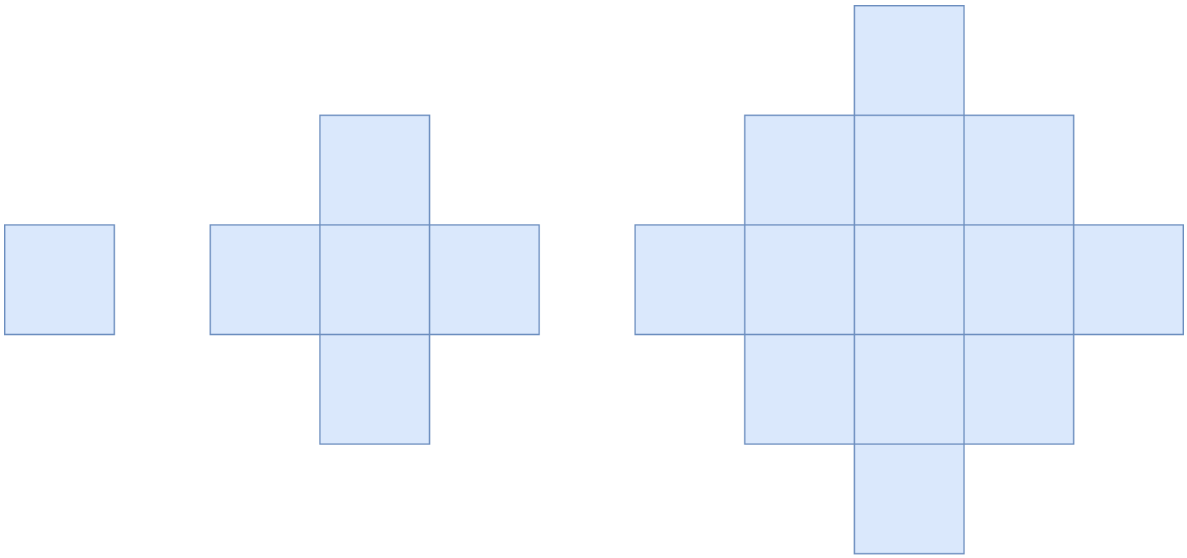
```
        }
        int[] newArr = new int[size];
        for (int i = 0; i < size; i++) {
            newArr[i] = arr[i];
        }
        Arrays.sort(newArr);
        int l = 0;
        int r = 0;
        for (int i = 0; i < size; i++) {
            if (i % 2 == 0) {
                l = l * 10 + newArr[i];
            } else {
                r = r * 10 + newArr[i];
            }
        }
        return l + r;
    }
}
```

2579. 统计染色格子数

有一个无穷大的二维网格图，一开始所有格子都未染色。给你一个正整数 `n` ，表示你需要执行以下步骤 `n` 分钟：

- 第一分钟，将 **任一** 格子染成蓝色。
- 之后的每一分钟，将与蓝色格子相邻的 **所有** 未染色格子染成蓝色。

下图分别是 1、2、3 分钟后的网格图。



n=1 **n=2** **n=3**

请你返回 `n` 分钟之后 **被染色的格子** 数目。

示例 1：

> 输入：n = 1
> 输出：1
> 解释：1 分钟后，只有 1 个蓝色的格子，所以返回 1 。

示例 2：

> 输入：n = 2
> 输出：5
> 解释：2 分钟后，有 4 个在边缘的蓝色格子和 1 个在中间的蓝色格子，所以返回 5 。

提示：

- `1 <= n <= 105`

```
class Solution {
    public long coloredCells(int n) {
        return (long)n * 2 * (n - 1) + 1;
    }
}
```

2580. 统计将重叠区间合并成组的方案数

给你一个二维整数数组 `ranges` ，其中 `ranges[i] = [starti, endi]` 表示 `starti` 到 `endi` 之间（包括二者）的所有整数都包含在第 `i` 个区间中。

你需要将 `ranges` 分成 **两个** 组（可以为空），满足：

- 每个区间只属于一个组。
- 两个有 **交集** 的区间必须在 **同一个** 组内。

如果两个区间有至少 **一个** 公共整数，那么这两个区间是 **有交集** 的。

- 比方说，区间 `[1, 3]` 和 `[2, 5]` 有交集，因为 `2` 和 `3` 在两个区间中都被包含。

请你返回将 `ranges` 划分成两个组的 **总方案数** 。由于答案可能很大，将它对 `109 + 7` **取余** 后返回。

示例 1：

> 输入：ranges = [[6,10],[5,15]]
> 输出：2
> 解释：
> 两个区间有交集，所以它们必须在同一个组内。
> 所以有两种方案：
>
> - 将两个区间都放在第 1 个组中。
> - 将两个区间都放在第 2 个组中。

示例 2：

> 输入：ranges = [[1,3],[10,20],[2,5],[4,8]]
> 输出：4
> 解释：
> 区间 [1,3] 和 [2,5] 有交集，所以它们必须在同一个组中。
> 同理，区间 [2,5] 和 [4,8] 也有交集，所以它们也必须在同一个组中。
> 所以总共有 4 种分组方案：
>
> - 所有区间都在第 1 组。

- 所有区间都在第 2 组。
- 区间 [1,3] ，[2,5] 和 [4,8] 在第 1 个组中，[10,20] 在第 2 个组中。
- 区间 [1,3] ，[2,5] 和 [4,8] 在第 2 个组中，[10,20] 在第 1 个组中。

提示：

- `1 <= ranges.length <= 105`
- `ranges[i].length == 2`
- `0 <= starti <= endi <= 109`

```
class Solution {
    public int getPow(int a, int n) {
        if (n == 0) {
            return 1;
        }
        if (n == 1) {
            return a;
        }

        long half1 = getPow(a, n / 2) % ((int)1e9 + 7);
        long half2 = getPow(a, n - n / 2) % ((int)1e9 + 7);
        return (int)(half1 * half2 % ((int)1e9 + 7));
    }

    public int countWays(int[][] ranges) {
        if (ranges.length == 1) {
            return 2;
        }
        int cnt = 0;
        Arrays.sort(ranges, new Comparator<int[]>() {
            public int compare(int[] a, int[] b) {
                if (a[0] == b[0]) {
                    return a[1] - b[1];
                }
                return a[0] - b[0];
            }
        });

        int l = 0;
        for (int i = 1; i < ranges.length; i++) {
            if (i == ranges.length - 1 && ranges[l][1] < ranges[i][0]) {
                cnt += 2;
                break;
            }
            if (ranges[l][1] < ranges[i][0] || i == ranges.length - 1) {
                cnt++;
                l = i;
            } else {
                ranges[l][1] = Math.max(ranges[i][1], ranges[l][1]);
            }

        }
        return getPow(2, cnt);
    }
}
```

2581. 统计可能的树根数目

Alice 有一棵 `n` 个节点的树，节点编号为 `0` 到 `n - 1` 。树用一个长度为 `n - 1` 的二维整数数组 `edges` 表示，其中 `edges[i] = [ai, bi]` ，表示树中节点 `ai` 和 `bi` 之间有一条边。

Alice 想要 Bob 找到这棵树的根。她允许 Bob 对这棵树进行若干次 **猜测** 。每一次猜测，Bob 做如下事情：

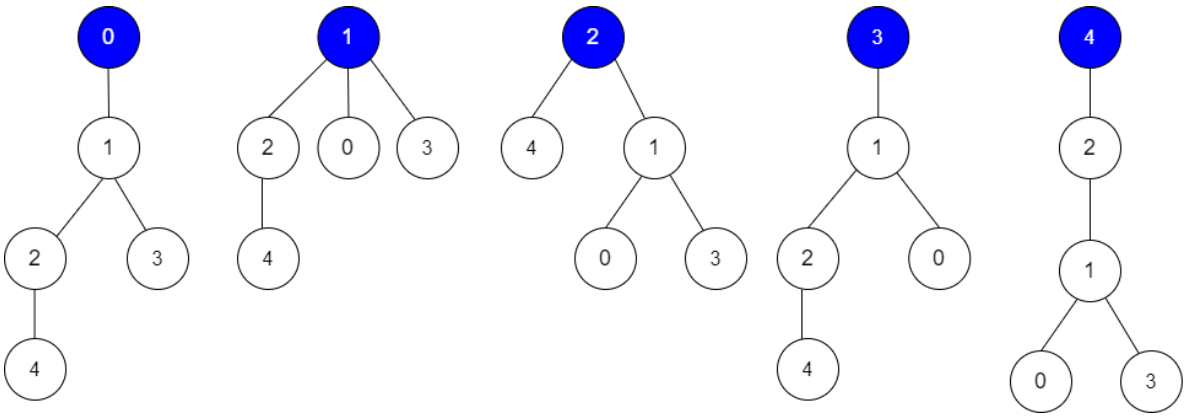
- 选择两个 **不相等** 的整数 `u` 和 `v` ，且树中必须存在边 `[u, v]` 。
- Bob 猜测树中 `u` 是 `v` 的 **父节点** 。

Bob 的猜测用二维整数数组 `guesses` 表示，其中 `guesses[j] = [uj, vj]` 表示 Bob 猜 `uj` 是 `vj` 的父节点。

Alice 非常懒，她不想逐个回答 Bob 的猜测，只告诉 Bob 这些猜测里面 **至少** 有 `k` 个猜测的结果为 `true` 。

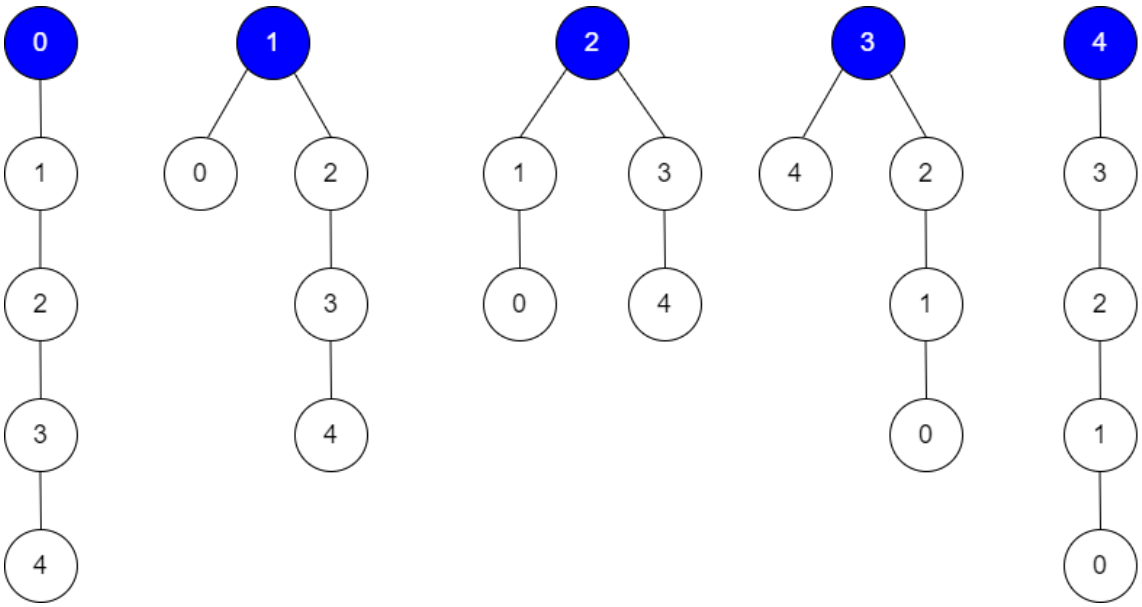
给你二维整数数组 `edges` ， Bob 的所有猜测和整数 `k` ，请你返回可能成为树根的 **节点数目** 。如果没有这样的树，则返回 `0` 。

示例 1：



输入：edges = [[0,1],[1,2],[1,3],[4,2]], guesses = [[1,3],[0,1],[1,0],[2,4]], k = 3
输出：3
解释：
根为节点 0 ， 正确的猜测为 [1,3], [0,1], [2,4]
根为节点 1 ， 正确的猜测为 [1,3], [1,0], [2,4]
根为节点 2 ， 正确的猜测为 [1,3], [1,0], [2,4]
根为节点 3 ， 正确的猜测为 [1,0], [2,4]
根为节点 4 ， 正确的猜测为 [1,3], [1,0]
节点 0 ， 1 或 2 为根时，可以得到 3 个正确的猜测。

示例 2：



> 输入：edges = [[0,1],[1,2],[2,3],[3,4]], guesses = [[1,0],[3,4],[2,1],[3,2]], k = 1
> 输出：5
> 解释：
> 根为节点 0 ， 正确的猜测为 [3,4]
> 根为节点 1 ， 正确的猜测为 [1,0], [3,4]
> 根为节点 2 ， 正确的猜测为 [1,0], [2,1], [3,4]
> 根为节点 3 ， 正确的猜测为 [1,0], [2,1], [3,2], [3,4]
> 根为节点 4 ， 正确的猜测为 [1,0], [2,1], [3,2]
> 任何节点为根，都至少有 1 个正确的猜测。

提示：

- `edges.length == n - 1`
- `2 <= n <= 105`
- `1 <= guesses.length <= 105`
- `0 <= ai, bi, uj, vj <= n - 1`
- `ai != bi`
- `uj != vj`
- `edges` 表示一棵有效的树。
- `guesses[j]` 是树中的一条边。
- `guesses` 是唯一的。
- `0 <= k <= guesses.length`

```
class Solution {
    ArrayList<Integer>[] edge;
    HashMap<Node, Integer> map;  // 统计Bob对Node中a是b的父亲猜了多少次
    int cur, ans, K;

    public void dfs(Node node) {
        cur += map.getOrDefault(node, 0);
        for (int i = 0; i < edge[node.b].size(); i++) {
            if (edge[node.b].get(i) != node.a) { // 因为是无向图，需要避免来回横跳
                dfs(new Node(node.b, edge[node.b].get(i)));
            }
        }
    }
```

```
    public void reRoot(Node node) {
        if (cur >= K) {
            ans += 1;
        }
        for (int i = 0; i < edge[node.b].size(); i++) {  // 换根，从node.b 换到
edge[node.b].get(i)
            if (edge[node.b].get(i) != node.a) {
                int back = cur;
                cur -= map.getOrDefault(new Node(node.b, edge[node.b].get(i)), 0);
                cur += map.getOrDefault(new Node(edge[node.b].get(i), node.b), 0);
                reRoot(new Node(node.b, edge[node.b].get(i)));
                cur = back;
            }
        }
    }

    public int rootCount(int[][] edges, int[][] guesses, int k) {
        int n = edges.length;
        int m = guesses.length;
        K = k;
        cur = 0;
        ans = 0;
        edge = new ArrayList[n + 1];
        map = new HashMap<>();

        for (int i = 0; i < edge.length; i++) {
            edge[i] = new ArrayList<>();
        }
        for (int i = 0; i < n; i++) {
            edge[edges[i][0]].add(edges[i][1]);
            edge[edges[i][1]].add(edges[i][0]);
        }
        for (int i = 0; i < m; i++) {
            Node node = new Node(guesses[i][0], guesses[i][1]);
            int cnt = map.getOrDefault(node, 0);
            map.put(node, cnt + 1);
        }
        dfs(new Node(-1, 0)); // 统计 0 作为根节点Bob猜对的次数

        reRoot(new Node(-1, 0));
        return ans;
    }
}

class Node {
    int a; // 表示从a到b
    int b;
    public Node(int a, int b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public boolean equals(Object obj) {
        Node node = (Node) obj;
        if (this.a == node.a && this.b == node.b) {
            return true;
```

```
        }
        return false;
    }

    @Override
    public int hashCode() {
        return (a << 16) | b;
    }
}
```

第 334 场周赛

2574. 左右元素和的差值

给你一个下标从 0 开始的整数数组 `nums` ，请你找出一个下标从 0 开始的整数数组 `answer` ，其中：

- `answer.length == nums.length`
- `answer[i] = |leftSum[i] - rightSum[i]|`

其中：

- `leftSum[i]` 是数组 `nums` 中下标 `i` 左侧元素之和。如果不存在对应的元素，`leftSum[i] = 0` 。
- `rightSum[i]` 是数组 `nums` 中下标 `i` 右侧元素之和。如果不存在对应的元素，`rightSum[i] = 0` 。

返回数组 `answer` 。

示例 1：

> 输入：nums = [10,4,8,3]
> 输出：[15,1,11,22]
> 解释：数组 leftSum 为 [0,10,14,22] 且数组 rightSum 为 [15,11,3,0] 。
> 数组 answer 为 [|0 - 15|,|10 - 11|,|14 - 3|,|22 - 0|] = [15,1,11,22] 。

示例 2：

> 输入：nums = [1]
> 输出：[0]
> 解释：数组 leftSum 为 [0] 且数组 rightSum 为 [0] 。
> 数组 answer 为 [|0 - 0|] = [0] 。

提示：

- `1 <= nums.length <= 1000`
- `1 <= nums[i] <= 105`

```
class Solution {
    public int[] leftRigthDifference(int[] nums) {
        int[] leftSum = new int[nums.length];
        int[] rightSum = new int[nums.length];
        int[] ans = new int[nums.length];

        leftSum[0] = nums[0];
```

```
        rightSum[nums.length - 1] = nums[nums.length - 1];
        for (int i = 1, j = nums.length - 2; i < leftSum.length; i++, j--) {
            leftSum[i] = leftSum[i - 1] + nums[i];
            rightSum[j] = rightSum[j + 1] + nums[j];
        }

        for (int i = 0; i < ans.length; i++) {
            ans[i] = Math.abs(leftSum[i] - rightSum[i]);
        }
        return ans;
    }
}
```

2575. 找出字符串的可整除数组

给你一个下标从 0 开始的字符串 `word` ，长度为 `n` ，由从 `0` 到 `9` 的数字组成。另给你一个正整数 `m` 。

`word` 的 **可整除数组** `div` 是一个长度为 `n` 的整数数组，并满足：

- 如果 `word[0,...,i]` 所表示的 **数值** 能被 `m` 整除，`div[i] = 1`
- 否则，`div[i] = 0`

返回 `word` 的可整除数组。

示例 1：

> 输入：word = "998244353", m = 3
> 输出：[1,1,0,0,0,1,1,0,0]
> 解释：仅有 4 个前缀可以被 3 整除："9"、"99"、"998244" 和 "9982443" 。

示例 2：

> 输入：word = "1010", m = 10
> 输出：[0,1,0,1]
> 解释：仅有 2 个前缀可以被 10 整除："10" 和 "1010" 。

提示：

- `1 <= n <= 105`
- `word.length == n`
- `word` 由数字 `0` 到 `9` 组成
- `1 <= m <= 109`

```
class Solution {
    public int[] divisibilityArray(String word, int m) {
        int[] ans = new int[word.length()];
        long mod = 0;
        for (int i = 0; i < word.length(); i++) {
            long num = mod * 10 + word.charAt(i) - '0';
            mod = num % m;
            if (mod == 0) {
                ans[i] = 1;
```

```
            }
        }
        return ans;
    }
}
```

2576. 求出最多标记下标

给你一个下标从 0 开始的整数数组 `nums` 。

一开始，所有下标都没有被标记。你可以执行以下操作任意次：

- 选择两个 **互不相同且未标记** 的下标 `i` 和 `j` ，满足 `2 * nums[i] <= nums[j]` ，标记下标 `i` 和 `j` 。

请你执行上述操作任意次，返回 `nums` 中最多可以标记的下标数目。

示例 1：

> 输入：nums = [3,5,2,4]
> 输出：2
> 解释：第一次操作中，选择 i = 2 和 j = 1 ，操作可以执行的原因是 2 * nums[2] <= nums[1] ，标记下标 2 和 1 。
> 没有其他更多可执行的操作，所以答案为 2 。

示例 2：

> 输入：nums = [9,2,5,4]
> 输出：4
> 解释：第一次操作中，选择 i = 3 和 j = 0 ，操作可以执行的原因是 2 * nums[3] <= nums[0] ，标记下标 3 和 0 。
> 第二次操作中，选择 i = 1 和 j = 2 ，操作可以执行的原因是 2 * nums[1] <= nums[2] ，标记下标 1 和 2 。
> 没有其他更多可执行的操作，所以答案为 4 。

示例 3：

> 输入：nums = [7,6,8]
> 输出：0
> 解释：没有任何可以执行的操作，所以答案为 0 。

提示：

- `1 <= nums.length <= 105`
- `1 <= nums[i] <= 109`

```
class Solution {
    public int maxNumOfMarkedIndices(int[] nums) {
        Arrays.sort(nums);
        int cnt = 0;
        int j = nums.length / 2;
        for (int i = 0; i < nums.length / 2; i++) {
            while (j < nums.length) {
```

```
                if (nums[i] * 2 <= nums[j]) {
                    cnt++;
                    j++;
                    break;
                }
                j++;
            }
        }
        return cnt * 2;
    }
}
```

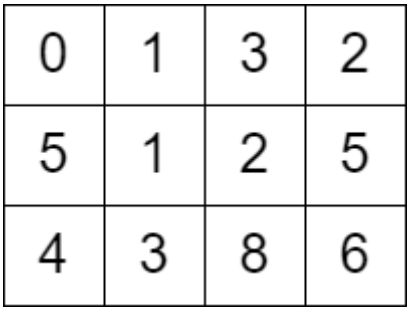
2577. 在网格图中访问一个格子的最少时间

给你一个 `m x n` 的矩阵 `grid` ，每个元素都为 **非负** 整数，其中 `grid[row][col]` 表示可以访问格子 `(row, col)` 的 **最早** 时间。也就是说当你访问格子 `(row, col)` 时，最少已经经过的时间为 `grid[row][col]` 。

你从 **最左上角** 出发，出发时刻为 `0` ，你必须一直移动到上下左右相邻四个格子中的 **任意** 一个格子（即不能停留在格子上）。每次移动都需要花费 1 单位时间。

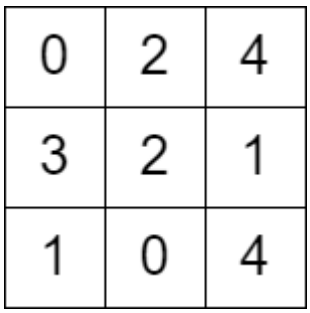
请你返回 **最早** 到达右下角格子的时间，如果你无法到达右下角的格子，请你返回 `-1` 。

示例 1：



> 输入：grid = [[0,1,3,2],[5,1,2,5],[4,3,8,6]]
> 输出：7
> 解释：一条可行的路径为：
>
> - 时刻 t = 0 ， 我们在格子 (0,0) 。
> - 时刻 t = 1 ， 我们移动到格子 (0,1) ， 可以移动的原因是 grid[0][1] <= 1 。
> - 时刻 t = 2 ， 我们移动到格子 (1,1) ， 可以移动的原因是 grid[1][1] <= 2 。
> - 时刻 t = 3 ， 我们移动到格子 (1,2) ， 可以移动的原因是 grid[1][2] <= 3 。
> - 时刻 t = 4 ， 我们移动到格子 (1,1) ， 可以移动的原因是 grid[1][1] <= 4 。
> - 时刻 t = 5 ， 我们移动到格子 (1,2) ， 可以移动的原因是 grid[1][2] <= 5 。
> - 时刻 t = 6 ， 我们移动到格子 (1,3) ， 可以移动的原因是 grid[1][3] <= 6 。
> - 时刻 t = 7 ， 我们移动到格子 (2,3) ， 可以移动的原因是 grid[2][3] <= 7 。
>
> 最终到达时刻为 7 。这是最早可以到达的时间。

示例 2：



> 输入：grid = [[0,2,4],[3,2,1],[1,0,4]]
> 输出：-1
> 解释：没法从左上角按题目规定走到右下角。

提示：

- `m == grid.length`
- `n == grid[i].length`
- `2 <= m, n <= 1000`
- `4 <= m * n <= 105`
- `0 <= grid[i][j] <= 105`
- `grid[0][0] == 0`

```
class Solution {
    public int minimumTime(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;

        if (grid[0][1] > 1 && grid[1][0] > 1) {
            return -1;
        }
        PriorityQueue<Node> que = new PriorityQueue<>((node1, node2) ->
(node1.distance - node2.distance));

        int[][] dis = new int[m][n];
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                dis[i][j] = Integer.MAX_VALUE;
            }
        }

        que.add(new Node(0, 0, 0));
        dis[0][0] = 0;

        int[] movX = new int[]{0, -1, 0, 1};
        int[] movY = new int[]{1, 0, -1, 0};
        while (!que.isEmpty()) {
            Node node = que.poll();
            if (dis[node.x][node.y] != node.distance) {
                continue;
            }
            for (int i = 0; i < 4; i++) {
                int newX = node.x + movX[i];
                int newY = node.y + movY[i];
                if (newX < 0 || newX >= m || newY < 0 || newY >= n) {
```

```
                    continue;
                }
                int newDistance = node.distance + 1;
                int waitTime = Math.max(0, grid[newX][newY] - newDistance);
                waitTime += waitTime % 2 == 0 ? 0 : 1;
                newDistance += waitTime;
                if (newDistance < dis[newX][newY]) {
                    dis[newX][newY] = newDistance;
                    que.add(new Node(newX, newY, newDistance));
                }
            }
        }
        int ans = dis[m - 1][n - 1];
        return ans == Integer.MAX_VALUE ? -1 : ans;
    }
}

class Node {
    int x;
    int y;
    int distance;

    public Node(int x, int y, int distance) {
        this.x = x;
        this.y = y;
        this.distance = distance;
    }
}
```

第 333 场周赛

2570. 合并两个二维数组 - 求和法

给你两个 **二维** 整数数组 `nums1` 和 `nums2.`

- `nums1[i] = [idi, vali]` 表示编号为 `idi` 的数字对应的值等于 `vali` 。
- `nums2[i] = [idi, vali]` 表示编号为 `idi` 的数字对应的值等于 `vali` 。

每个数组都包含 **互不相同** 的 id ， 并按 id 以 **递增** 顺序排列。

请你将两个数组合并为一个按 id 以递增顺序排列的数组，并符合下述条件：

- 只有在两个数组中至少出现过一次的 id 才能包含在结果数组内。
- 每个 id 在结果数组中 **只能出现一次** ， 并且其对应的值等于两个数组中该 id 所对应的值求和。如果某个数组中不存在该 id ， 则认为其对应的值等于 `0` 。

返回结果数组。返回的数组需要按 id 以递增顺序排列。

示例 1：

> 输入：nums1 = [[1,2],[2,3],[4,5]], nums2 = [[1,4],[3,2],[4,1]]
> 输出：[[1,6],[2,3],[3,2],[4,6]]
> 解释：结果数组中包含以下元素：
>
> - id = 1 ， 对应的值等于 2 + 4 = 6 。
> - id = 2 ， 对应的值等于 3 。

- id = 3 ， 对应的值等于 2 。
- id = 4 ， 对应的值等于5 + 1 = 6 。

示例 2：

> 输入：nums1 = [[2,4],[3,6],[5,5]], nums2 = [[1,3],[4,3]]
> 输出：[[1,3],[2,4],[3,6],[4,3],[5,5]]
> 解释：不存在共同 id ， 在结果数组中只需要包含每个 id 和其对应的值。

提示：

- `1 <= nums1.length, nums2.length <= 200`
- `nums1[i].length == nums2[j].length == 2`
- `1 <= idi, vali <= 1000`
- 数组中的 id 互不相同
- 数据均按 id 以严格递增顺序排列

```
class Solution {
    public int[][] mergeArrays(int[][] nums1, int[][] nums2) {
        int len = 0;
        int[] tmp = new int[1001];

        for (int i = 0; i < nums1.length; i++) {
            if (tmp[nums1[i][0]] == 0) {
                len++;
            }
            tmp[nums1[i][0]] += nums1[i][1];
        }

        for (int i = 0; i < nums2.length; i++) {
            if (tmp[nums2[i][0]] == 0) {
                len++;
            }
            tmp[nums2[i][0]] += nums2[i][1];
        }

        int[][] ans = new int[len][2];
        int size = 0;
        for (int i = 0; i < 1001; i++) {
            if (tmp[i] == 0) {
                continue;
            }
            ans[size][0] = i;
            ans[size][1] = tmp[i];
            size++;
        }
        return ans;
    }
}
```

2571. 将整数减少到零需要的最少操作数

给你一个正整数 `n` ， 你可以执行下述操作 **任意** 次：

- `n` 加上或减去 `2` 的某个 **幂**

返回使 `n` 等于 `0` 需要执行的 **最少** 操作数。

如果 `x == 2i` 且其中 `i >= 0` ，则数字 `x` 是 `2` 的幂。

示例 1：

> 输入：n = 39
> 输出：3
> 解释：我们可以执行下述操作：
>
> - n 加上 20 = 1 ，得到 n = 40 。
> - n 减去 23 = 8 ，得到 n = 32 。
> - n 减去 25 = 32 ，得到 n = 0 。
>
> 可以证明使 n 等于 0 需要执行的最少操作数是 3 。

示例 2：

> 输入：n = 54
> 输出：3
> 解释：我们可以执行下述操作：
>
> - n 加上 21 = 2 ，得到 n = 56 。
> - n 加上 23 = 8 ，得到 n = 64 。
> - n 减去 26 = 64 ，得到 n = 0 。
>
> 使 n 等于 0 需要执行的最少操作数是 3 。

提示：

- `1 <= n <= 105`

```
class Solution {
    public int minOperations(int n) {
        int ans = 0;
        int[] arr = new int[32];
        int size = 0;

        while (n > 0) {
            arr[size++] = n % 2;
            n /= 2;
        }

        for (int i = 0; i < 32; i++) {
            if (arr[i] == 1 && arr[i + 1] == 1) {
                int j = i;
                while (j < size && arr[j] == 1) {
                    arr[j] = 0;
                    j++;
                }
                ans++;
                arr[j] = 1;
            }
        }

        for (int i = 0; i < 32; i++) {
```

```
            if (arr[i] == 1) {
                ans++;
            }
        }
        return ans;
    }
}
```

2572. 无平方子集计数

给你一个正整数数组 `nums` 。

如果数组 `nums` 的子集中的元素乘积是一个 **无平方因子数** ，则认为该子集是一个 **无平方** 子集。

无平方因子数 是无法被除 `1` 之外任何平方数整除的数字。

返回数组 `nums` 中 **无平方** 且 **非空** 的子集数目。因为答案可能很大，返回对 `109 + 7` 取余的结果。

`nums` 的 **非空子集** 是可以由删除 `nums` 中一些元素（可以不删除，但不能全部删除）得到的一个数组。如果构成两个子集时选择删除的下标不同，则认为这两个子集不同。

示例 1：

> 输入：nums = [3,4,4,5]
> 输出：3
> 解释：示例中有 3 个无平方子集：
>
> - 由第 0 个元素 [3] 组成的子集。其元素的乘积是 3 ，这是一个无平方因子数。
> - 由第 3 个元素 [5] 组成的子集。其元素的乘积是 5 ，这是一个无平方因子数。
> - 由第 0 个和第 3 个元素 [3,5] 组成的子集。其元素的乘积是 15 ，这是一个无平方因子数。
>
> 可以证明给定数组中不存在超过 3 个无平方子集。

示例 2：

> 输入：nums = [1]
> 输出：1
> 解释：示例中有 1 个无平方子集：
>
> - 由第 0 个元素 [1] 组成的子集。其元素的乘积是 1 ，这是一个无平方因子数。
>
> 可以证明给定数组中不存在超过 1 个无平方子集。

提示：

- `1 <= nums.length <= 1000`
- `1 <= nums[i] <= 30`

```
class Solution {
    public boolean isSquareFree(int num) {
        int[] arr = new int[]{4, 8, 9, 12, 16, 18, 20, 24, 25, 27, 28};
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == num) {
                return true;
            }
        }
```

```
        return false;
    }

    public int squareFreeSubsets(int[] nums) {
        HashMap<Integer, Long> map = new HashMap<>();
        map.put(0, 1l);
        for (int i = 0; i < nums.length; i++) {
            if (isSquareFree(nums[i])) {
                continue;
            }
            int state = 0;
            int tmp = nums[i];
            for (int j = 2; j * j <= nums[i]; j++) {
                if (tmp % j == 0) {
                    state |= (1 << j);
                    tmp /= j;
                }
            }
            if (tmp > 1) {
                state |= (1 << tmp);
            }

            HashMap<Integer, Long> tmpMap = new HashMap<>();
            for (int key : map.keySet()) {
                if ((key & state) == 0) {
                    tmpMap.put(key | state, map.get(key));
                }
            }

            for(int key : tmpMap.keySet()){
                map.put(key, (map.getOrDefault(key, 0l) + tmpMap.get(key)) % ((int)1e9
+ 7));
            }
        }
        long ans = 0;
        for(int key : map.keySet()){
            ans += map.get(key);
        }

        return (int)((ans - 1) % ((int)1e9 + 7));
    }
}
```

2573. 找出对应 LCP 矩阵的字符串

对任一由 `n` 个小写英文字母组成的字符串 `word` ，我们可以定义一个 `n x n` 的矩阵，并满足：

- `lcp[i][j]` 等于子字符串 `word[i,...,n-1]` 和 `word[j,...,n-1]` 之间的最长公共前缀的长度。

给你一个 `n x n` 的矩阵 `lcp` 。返回与 `lcp` 对应的、按字典序最小的字符串 `word` 。如果不存在这样的字符串，则返回空字符串。

对于长度相同的两个字符串 `a` 和 `b` ，如果在 `a` 和 `b` 不同的第一个位置，字符串 `a` 的字母在字母表中出现的顺序先于 `b` 中的对应字母，则认为字符串 `a` 按字典序比字符串 `b` 小。例如， `"aabd"` 在字典上小于 `"aaca"` ，因为二者不同的第一位置是第三个字母，而 `'b'` 先于 `'c'` 出现。

示例 1：

> 输入：lcp = [[4,0,2,0],[0,3,0,1],[2,0,2,0],[0,1,0,1]]
> 输出："abab"
> 解释：lcp 对应由两个交替字母组成的任意 4 字母字符串，字典序最小的是 "abab" 。

示例 2：

> 输入：lcp = [[4,3,2,1],[3,3,2,1],[2,2,2,1],[1,1,1,1]]
> 输出："aaaa"
> 解释：lcp 对应只有一个不同字母的任意 4 字母字符串，字典序最小的是 "aaaa" 。

示例 3：

> 输入：lcp = [[4,3,2,1],[3,3,2,1],[2,2,2,1],[1,1,1,3]]
> 输出：""
> 解释：lcp[3][3] 无法等于 3 ，因为 word[3,...,3] 仅由单个字母组成；因此，不存在答案。

提示：

- `1 <= n == ``lcp.length == ``lcp[i].length <= 1000`
- `0 <= lcp[i][j] <= n`

```
class Solution {
    public boolean isFilled(char[] str, int i) {
        return (str[i] >= 'a' && str[i] <= 'z') ? true : false;
    }
    public String findTheString(int[][] lcp) {
        char[] str = new char[lcp.length];
        int idx = 0;

        for (char i = 'a'; i <= 'z'; i++) {
            if (idx >= lcp.length) {
                break;
            }
            while (idx < lcp.length && isFilled(str, idx)) {
                idx++;
            }
            for (int j = idx; j < lcp.length; j++) {
                if (lcp[idx][j] > 0) { // lcp[i][j] > 0必然对应着str[i] == str[j]
                    str[j] = i;
                }
            }
        }

        while (idx < lcp.length) { // 26个字母不够用
            if (!isFilled(str, idx)) {
                return "";
            }
            idx++;
        }

        int expectVal;
        for (int i = lcp.length - 1; i >= 0; i--) {
```

```
            for (int j = lcp.length - 1; j >= 0; j--) {
                if (i == lcp.length - 1 || j == lcp.length - 1) {
                    expectVal = str[i] == str[j] ? 1 : 0;
                } else {
                    expectVal = str[i] == str[j] ? lcp[i + 1][j + 1] + 1 : 0;
                }
                if (lcp[i][j] != expectVal) {
                    return "";
                }
            }
        }
        return new String(str);
    }
}
```

第 098 场双周赛

2566. 替换一个数字后的最大差值

给你一个整数 `num` 。你知道 Danny Mittal 会偷偷将 `0` 到 `9` 中的一个数字 **替换** 成另一个数字。

请你返回将 `num` 中 **恰好一个** 数字进行替换后，得到的最大值和最小值的差为多少。

注意：

- 当 Danny 将一个数字 `d1` 替换成另一个数字 `d2` 时，Danny 需要将 `nums` 中所有 `d1` 都替换成 `d2` 。
- Danny 可以将一个数字替换成它自己，也就是说 `num` 可以不变。
- Danny 可以将数字分别替换成两个不同的数字分别得到最大值和最小值。
- 替换后得到的数字可以包含前导 0 。
- Danny Mittal 获得周赛 326 前 10 名，让我们恭喜他。

示例 1：

> 输入：num = 11891
> 输出：99009
> 解释：
> 为了得到最大值，我们将数字 1 替换成数字 9 ，得到 99899 。
> 为了得到最小值，我们将数字 1 替换成数字 0 ，得到 890 。
> 两个数字的差值为 99009 。

示例 2：

> 输入：num = 90
> 输出：99
> 解释：
> 可以得到的最大值是 99（将 0 替换成 9），最小值是 0（将 9 替换成 0）。
> 所以我们得到 99 。

提示：

- `1 <= num <= 108`

```java
class Solution {
    public int getMaxNum(int[] arr, int size) {
        int maxNum = 0;
        int replace = 0;
        int i = size- 1;
        while (i >= 0 && arr[i] == 9) {
            maxNum = maxNum * 10 + arr[i];
            i--;
        }
        replace = i >= 0 ? arr[i] : 9;
        for (; i >= 0; i--) {
            if (arr[i] == replace) {
                maxNum = maxNum * 10 + 9;
            } else {
                maxNum = maxNum * 10 + arr[i];
            }
        }
        return maxNum;
    }

    public int getMinNum(int[] arr, int size) {
        int minNum = 0;
        int replace = arr[size- 1];
        for (int i = size- 1; i >= 0; i--) {
            if (arr[i] == replace) {
                minNum = minNum * 10 + 0;
            } else {
                minNum = minNum * 10 + arr[i];
            }
        }
        return minNum;
    }

    public int minMaxDifference(int num) {
        int[] arr = new int[9];
        int size = 0;

        while (num != 0) {
            arr[size++] = num % 10;
            num /= 10;
        }

        int maxNum = getMaxNum(arr, size);
        int minNum = getMinNum(arr, size);
        return maxNum - minNum;
    }
}
```

2567. 修改两个元素的最小分数

给你一个下标从 0 开始的整数数组 `nums` 。

- `nums` 的 **最小** 得分是满足 `0 <= i < j < nums.length` 的 `|nums[i] - nums[j]|` 的最小值。
- `nums` 的 **最大** 得分是满足 `0 <= i < j < nums.length` 的 `|nums[i] - nums[j]|` 的最大值。
- `nums` 的分数是 **最大** 得分与 **最小** 得分的和。

我们的目标是最小化 `nums` 的分数。你 **最多** 可以修改 `nums` 中 2 个元素的值。

请你返回修改 `nums` 中 **至多两个** 元素的值后，可以得到的 **最小分数** 。

`|x|` 表示 `x` 的绝对值。

示例 1：

> 输入：nums = [1,4,3]
> 输出：0
> 解释：将 nums[1] 和 nums[2] 的值改为 1 ， nums 变为 [1,1,1] 。|nums[i] - nums[j]| 的值永远为 0 ，所以我们返回 0 + 0 = 0 。

示例 2：

> 输入：nums = [1,4,7,8,5]
> 输出：3
> 解释：
> 将 nums[0] 和 nums[1] 的值变为 6 ， nums 变为 [6,6,7,8,5] 。
> 最小得分是 i = 0 且 j = 1 时得到的 |nums[i] - nums[j]| = |6 - 6| = 0 。
> 最大得分是 i = 3 且 j = 4 时得到的 |nums[i] - nums[j]| = |8 - 5| = 3 。
> 最大得分与最小得分之和为 3 。这是最优答案。

提示：

- `3 <= nums.length <= 105`
- `1 <= nums[i] <= 109`

```
class Solution {
    public int minimizeSum(int[] nums) {
        Arrays.sort(nums);

        int a = nums[nums.length - 1] - nums[2];
        int b = nums[nums.length - 2] - nums[1];
        int c = nums[nums.length - 3] - nums[0];
        return Math.min(a, Math.min(b, c));
    }
}
```

2568. 最小无法得到的或值

给你一个下标从 **0** 开始的整数数组 `nums` 。

如果存在一些整数满足 `0 <= index1 < index2 < ... < indexk < nums.length` ，得到 `nums[index1] | nums[index2] | ... | nums[indexk] = x` ，那么我们说 `x` 是 **可表达的** 。换言之，如果一个整数能由 `nums` 的某个子序列的或运算得到，那么它就是可表达的。

请你返回 `nums` 不可表达的 **最小非零整数** 。

示例 1：

> 输入：nums = [2,1]
> 输出：4
> 解释：1 和 2 已经在数组中，因为 nums[0] | nums[1] = 2 | 1 = 3 ，所以 3 是可表达的。由于 4 是不可表达的，所以我们返回 4 。

示例 2：

> 输入：nums = [5,3,2]
> 输出：1
> 解释：1 是最小不可表达的数字。

提示：

- `1 <= nums.length <= 105`
- `1 <= nums[i] <= 109`

```
class Solution {
    public int minImpossibleOR(int[] nums) {
        int len = (int)1e9;
        for (int i = 0; i < len; i++) {
            boolean find = false;
            int num = (int)Math.pow(2, i);
            for (int j = 0; j < nums.length; j++) {
                if (nums[j] == num) {
                    find = true;
                    break;
                }
            }
            if (!find) {
                return num;
            }
        }
        return -1;
    }
}
```

2569. 更新数组后处理求和查询

给你两个下标从 0 开始的数组 `nums1` 和 `nums2` ，和一个二维数组 `queries` 表示一些操作。总共有 3 种类型的操作：

1. 操作类型 1 为 `queries[i] = [1, l, r]` 。你需要将 `nums1` 从下标 `l` 到下标 `r` 的所有 `0` 反转成 `1` 或将 `1` 反转成 `0` 。 `l` 和 `r` 下标都从 0 开始。
2. 操作类型 2 为 `queries[i] = [2, p, 0]` 。对于 `0 <= i < n` 中的所有下标，令 `nums2[i] = nums2[i] + nums1[i] * p` 。
3. 操作类型 3 为 `queries[i] = [3, 0, 0]` 。求 `nums2` 中所有元素的和。

请你返回一个数组，包含所有第三种操作类型的答案。

示例 1：

> 输入：nums1 = [1,0,1], nums2 = [0,0,0], queries = [[1,1,1],[2,1,0],[3,0,0]]
> 输出：[3]
> 解释：第一个操作后 nums1 变为 [1,1,1] 。第二个操作后，nums2 变成 [1,1,1] ，所以第三个操作的答案为 3 。所以返回 [3] 。

示例 2：

> 输入：nums1 = [1], nums2 = [5], queries = [[2,0,0],[3,0,0]]
> 输出：[5]
> 解释：第一个操作后，nums2 保持不变为 [5] ，所以第二个操作的答案是 5 。所以返回 [5] 。

提示：

- `1 <= nums1.length,nums2.length <= 105`
- `nums1.length = nums2.length`
- `1 <= queries.length <= 105`
- `queries[i].length = 3`
- `0 <= l <= r <= nums1.length - 1`
- `0 <= p <= 106`
- `0 <= nums1[i] <= 1`
- `0 <= nums2[i] <= 109`

```
class Solution {
    SegmentTree st;
    public long[] handleQuery(int[] nums1, int[] nums2, int[][] queries) {
        st = new SegmentTree(nums1.length, nums1);
        ArrayList<Long> ans = new ArrayList<>();
        long sum = 0;

        for (int i = 0; i < nums2.length; i++) {
            sum += nums2[i];
        }
        for (int i = 0; i < queries.length; i++) {
            if (queries[i][0] == 1) {
                st.modify(0, queries[i][1], queries[i][2]);
            }
            if (queries[i][0] == 2) {
                sum += (long)st.nodes[0].sum * queries[i][1]; // 0 * p还是0，这里不计算
            }
            if (queries[i][0] == 3) {
                ans.add(sum);
            }
        }

        long[] arr = new long[ans.size()];
        for (int i = 0;  i < ans.size(); i++) {
            arr[i] = ans.get(i);
        }
        return arr;
    }
}

public class SegmentTree {
```

```
    Node[] nodes;

    public SegmentTree(int num, int[] val) {
        nodes = new Node[num * 4];
        for (int i = 0; i < nodes.length; i++) {
            nodes[i] = new Node(0, 0, 0);
        }
        build(0, 0, num - 1, val);
    }

    public void build(int idx, int l, int r, int[] val) {
        nodes[idx].left = l;
        nodes[idx].right = r;
        nodes[idx].tag = false;
        if (l == r) {
            nodes[idx].sum = val[l];
            return;
        }
        int mid = l + (r - l) / 2;
        if (l <= mid) {
            build(idx * 2 + 1, l, mid, val);
        }
        if (mid < r){
            build(idx * 2 + 2, mid + 1, r, val);
        }
        nodes[idx].sum = nodes[idx * 2 + 1].sum + nodes[idx * 2 + 2].sum;
    }

    public void pushUp(int idx) {
        nodes[idx].sum = nodes[idx * 2 + 1].sum + nodes[idx * 2 + 2].sum;
    }

    public void pushDown(int idx) {
        if (nodes[idx].left == nodes[idx].right) {
            return;
        }

        if (!nodes[idx].tag) {
            return;
        }
        nodes[idx * 2 + 1].tag = !nodes[idx * 2 + 1].tag;
        nodes[idx * 2 + 1].sum = nodes[idx * 2 + 1].getLen() - nodes[idx * 2 + 1].sum;
        nodes[idx * 2 + 2].tag = !nodes[idx * 2 + 2].tag;
        nodes[idx * 2 + 2].sum = nodes[idx * 2 + 2].getLen() - nodes[idx * 2 + 2].sum;
        nodes[idx].tag = false;
    }
    public void modify(int idx, int left, int right) {
        if ((nodes[idx].left >= left) && (nodes[idx].right <= right)) {
            nodes[idx].tag = !nodes[idx].tag;
            nodes[idx].sum = nodes[idx].getLen() - nodes[idx].sum; // 相当于翻转nums1的0
和1
            return;
        }
        pushDown(idx);
        int mid = nodes[idx].left + (nodes[idx].right - nodes[idx].left) / 2;
        if (left <= mid) {
            modify(idx * 2 + 1, left, right);
        }
```

```
        if (mid < right) {
            modify(idx * 2 + 2, left, right);
        }
        pushUp(idx);
    }
}

class Node {
    int left;
    int right;
    int sum;
    boolean tag;

    public Node (int left, int right, int sum) {
        this.left = left;
        this.right = right;
        this.sum = sum;
    }
    public int getLen() {
        return right - left + 1;
    }
}
```

第 332 场周赛

2562. 找出数组的串联值

给你一个下标从 0 开始的整数数组 `nums` 。

现定义两个数字的 **串联** 是由这两个数值串联起来形成的新数字。

- 例如， `15` 和 `49` 的串联是 `1549` 。

`nums` 的 **串联值** 最初等于 `0` 。执行下述操作直到 `nums` 变为空：

- 如果 `nums` 中存在不止一个数字，分别选中 `nums` 中的第一个元素和最后一个元素，将二者串联得到的值加到 `nums` 的 **串联值** 上，然后从 `nums` 中删除第一个和最后一个元素。
- 如果仅存在一个元素，则将该元素的值加到 `nums` 的串联值上，然后删除这个元素。

返回执行完所有操作后 `nums` 的串联值。

示例 1：

> 输入：nums = [7,52,2,4]
> 输出：596
> 解释：在执行任一步操作前，nums 为 [7,52,2,4] ，串联值为 0 。
>
> - 在第一步操作中：
>   我们选中第一个元素 7 和最后一个元素 4 。
>   二者的串联是 74 ，将其加到串联值上，所以串联值等于 74 。
>   接着我们从 nums 中移除这两个元素，所以 nums 变为 [52,2] 。
> - 在第二步操作中：
>   我们选中第一个元素 52 和最后一个元素 2 。
>   二者的串联是 522 ，将其加到串联值上，所以串联值等于 596 。

接着我们从 nums 中移除这两个元素，所以 nums 变为空。
由于串联值等于 596 ，所以答案就是 596 。

示例 2：

输入：nums = [5,14,13,8,12]
输出：673
解释：在执行任一步操作前，nums 为 [5,14,13,8,12] ，串联值为 0 。

- 在第一步操作中：
  我们选中第一个元素 5 和最后一个元素 12 。
  二者的串联是 512 ，将其加到串联值上，所以串联值等于 512 。
  接着我们从 nums 中移除这两个元素，所以 nums 变为 [14,13,8] 。
- 在第二步操作中：
  我们选中第一个元素 14 和最后一个元素 8 。
  二者的串联是 148 ，将其加到串联值上，所以串联值等于 660 。
  接着我们从 nums 中移除这两个元素，所以 nums 变为 [13] 。
- 在第三步操作中：
  nums 只有一个元素，所以我们选中 13 并将其加到串联值上，所以串联值等于 673 。
  接着我们从 nums 中移除这个元素，所以 nums 变为空。
  由于串联值等于 673 ，所以答案就是 673 。

提示：

- `1 <= nums.length <= 1000`
- `1 <= nums[i] <= 104`

```
int getRightBitNum(int num) {
    int res = 0;
    while (num != 0) {
        res++;
        num /= 10;
    }

    return res;
}
long long findTheArrayConcVal(int* nums, int numsSize){
    int l = 0;
    int r = numsSize - 1;
    long long res = 0;

    while (l < r) {
        int bitNum = getRightBitNum(nums[r]);
        // printf("%d \n", bitNum);
        res += (long long)nums[l] * pow(10, bitNum) + nums[r];
        l++;
        r--;
    }

    if (l == r) {
        res += nums[l];
    }

    return res;
```

```
}
```

2563. 统计公平数对的数目

给你一个下标从 0 开始、长度为 `n` 的整数数组 `nums` ，和两个整数 `lower` 和 `upper` ，返回 **公平数对的数目** 。

如果 `(i, j)` 数对满足以下情况，则认为它是一个 **公平数对** ：

- `0 <= i < j < n`，且
- `lower <= nums[i] + nums[j] <= upper`

示例 1：

> 输入：nums = [0,1,7,4,4,5], lower = 3, upper = 6
> 输出：6
> 解释：共计 6 个公平数对：(0,3)、(0,4)、(0,5)、(1,3)、(1,4) 和 (1,5) 。

示例 2：

> 输入：nums = [1,7,9,2,5], lower = 11, upper = 11
> 输出：1
> 解释：只有单个公平数对：(2,3) 。

提示：

- `1 <= nums.length <= 105`
- `nums.length == n`
- `-109 <= nums[i] <= 109`
- `-109 <= lower <= upper <= 109`

```
class Solution {
    public int findLeftIndex(int[] nums, int lower, int upper, int i) {
        int l = i;
        int r = nums.length - 1;
        int mid;
        int majar = i;

        while (l <= r) {
            mid = l + (r - l) / 2;
            if (nums[i] + nums[mid] < lower) {
                l = mid + 1;
            } else if (nums[i] + nums[mid] > upper) {
                r = mid - 1;
            }  else {
                majar = mid;
                // System.out.println("left " + i + " " + majar);
                r = mid - 1;
            }
        }

       return majar;
    }
```

```
     public int findRightIndex(int[] nums, int lower, int upper, int i) {
        int l = i;
        int r = nums.length - 1;
        int mid;
        int majar = i;

        while (l <= r) {
            mid = l + (r - l) / 2;
            if (nums[i] + nums[mid] < lower) {
                l = mid + 1;
            } else if (nums[i] + nums[mid] > upper) {
                r = mid - 1;
            }  else {
                majar = mid;
                // System.out.println("right " + i + " " + majar);
                l = mid + 1;
            }
        }

       return majar;
    }

    public long countFairPairs(int[] nums, int lower, int upper) {
        long res = 0;
        Arrays.sort(nums);

        for (int i = 0; i < nums.length - 1; i++) {
            int left = findLeftIndex(nums, lower, upper, i);
            int right = findRightIndex(nums, lower, upper, i);
            if (i == left) {
                res += right - left;
            } else {
                res += right - left + 1;
            }

        }

        return res;
    }
}
```

2564. 子字符串异或查询

给你一个 **二进制字符串** `s` 和一个整数数组 `queries` ，其中 `queries[i] = [firsti, secondi]` 。

对于第 `i` 个查询，找到 `s` 的 **最短子字符串** ，它对应的 **十进制**值 `val` 与 `firsti` **按位异或** 得到 `secondi` ，换言之， `val ^ firsti == secondi` 。

第 `i` 个查询的答案是子字符串 `[lefti, righti]` 的两个端点（下标从 0 开始），如果不存在这样的子字符串，则答案为 `[-1, -1]` 。如果有多个答案，请你选择 `lefti` 最小的一个。

请你返回一个数组 `ans` ，其中 `ans[i] = [lefti, righti]` 是第 `i` 个查询的答案。

子字符串 是一个字符串中一段连续非空的字符序列。

示例 1：

> 输入：s = "101101", queries = [[0,5],[1,2]]
> 输出：[[0,2],[2,3]]
> 解释：第一个查询，端点为 [0,2] 的子字符串为 "101"，对应十进制数字 5，且 5 ^ 0 = 5，所以第一个查询的答案为 [0,2]。第二个查询中，端点为 [2,3] 的子字符串为 "11"，对应十进制数字 3，且 3 ^ 1 = 2 。所以第二个查询的答案为 [2,3] 。

示例 2：

> 输入：s = "0101", queries = [[12,8]]
> 输出：[[-1,-1]]
> 解释：这个例子中，没有符合查询的答案，所以返回 [-1,-1] 。

示例 3：

> 输入：s = "1", queries = [[4,5]]
> 输出：[[0,0]]
> 解释：这个例子中，端点为 [0,0] 的子字符串对应的十进制值为 1，且 1 ^ 4 = 5 。所以答案为 [0,0] 。

提示：

- `1 <= s.length <= 104`
- `s[i]` 要么是 `'0'`，要么是 `'1'`。
- `1 <= queries.length <= 105`
- `0 <= firsti, secondi <= 109`

```
class Solution {
    public int[][] substringXorQueries(String s, int[][] queries) {
        /* 如果 a ^ b = c, 则 a ^ c = b, b ^ c = a, 所以可以根据first和second来找字符串 */
        HashMap<Integer, int[]> map = new HashMap<>();
        char[] str = s.toCharArray();
        for (int i = 0; i < str.length; i++) {
            if (str[i] == '0') {
                map.put(0, new int[]{i, i});
                break;
            }
        }
        for(int i = 0; i < str.length; i++){
            if (str[i] == '0') { // 不能包含前导0
                continue;
            }
            for(int j = i + 1; j - i <= 31 && j <= str.length; j++){
                int num = Integer.valueOf(s.substring(i, j), 2);
                if(map.containsKey(num)){
                    continue;
                }
                map.put(num, new int[]{i, j - 1});
            }
        }

        int[][] result = new int[queries.length][2];
```

```
        for(int i = 0; i < queries.length; i++){
            int a = queries[i][0];
            int c = queries[i][1];
            result[i] = map.getOrDefault(a ^ c, new int[]{-1, -1});
        }

        return result;
    }
}
```

2565. 最少得分子序列

给你两个字符串 `s` 和 `t` 。

你可以从字符串 `t` 中删除任意数目的字符。

如果没有从字符串 `t` 中删除字符，那么得分为 `0` ，否则：

- 令 `left` 为删除字符中的最小下标。
- 令 `right` 为删除字符中的最大下标。

字符串的得分为 `right - left + 1` 。

请你返回使 `t` 成为 `s` 子序列的最小得分。

一个字符串的 **子序列** 是从原字符串中删除一些字符后（也可以一个也不删除），剩余字符不改变顺序得到的字符串。（比方说 `"ace"` 是 `"***a\***b***c\***d***e\***"` 的子序列，但是 `"aec"` 不是）。

示例 1：

> 输入：s = "abacaba", t = "bzaa"
> 输出：1
> 解释：这个例子中，我们删除下标 1 处的字符 "z" （下标从 0 开始）。
> 字符串 t 变为 "baa" ，它是字符串 "abacaba" 的子序列，得分为 1 - 1 + 1 = 1 。
> 1 是能得到的最小得分。

示例 2：

> 输入：s = "cde", t = "xyz"
> 输出：3
> 解释：这个例子中，我们将下标为 0， 1 和 2 处的字符 "x" ， "y" 和 "z" 删除（下标从 0 开始）。
> 字符串变成 "" ，它是字符串 "cde" 的子序列，得分为 2 - 0 + 1 = 3 。
> 3 是能得到的最小得分。

提示：

- `1 <= s.length, t.length <= 105`
- `s` 和 `t` 都只包含小写英文字母。

```
class Solution {
    public int minimumScore(String s, String t) {
        char[] s1 = s.toCharArray();
        char[] t1 = t.toCharArray();
```

```
        int n = s1.length, m = t1.length;

        int[] postfix = new int[n + 1]; // postfix[i]表示以i为起点的t的后缀匹配的得分
        postfix[n] = m;
        for (int i = n - 1, j = m - 1; i >= 0; i--) {
            if (j >= 0 && s1[i] == t1[j]) {
                j--;
            }
            postfix[i] = j + 1;
        }

        int ans = postfix[0];
        if (ans == 0) {
            return 0;
        }
        for (int i = 0, j = 0; i < n; i++) {
            if (s1[i] == t1[j]) {
                j++;
                ans = Math.min(ans, postfix[i + 1] - j);
            }
        }
        return ans;
    }
}
```

第 331 场周赛

2558. 从数量最多的堆取走礼物

给你一个整数数组 `gifts` ，表示各堆礼物的数量。每一秒，你需要执行以下操作：

- 选择礼物数量最多的那一堆。
- 如果不止一堆都符合礼物数量最多，从中选择任一堆即可。
- 选中的那一堆留下平方根数量的礼物（向下取整），取走其他的礼物。

返回在 `k` 秒后剩下的礼物数量。

示例 1：

> 输入：gifts = [25,64,9,4,100], k = 4
> 输出：29
> 解释：
> 按下述方式取走礼物：
>
> - 在第一秒，选中最后一堆，剩下 10 个礼物。
> - 接着第二秒选中第二堆礼物，剩下 8 个礼物。
> - 然后选中第一堆礼物，剩下 5 个礼物。
> - 最后，再次选中最后一堆礼物，剩下 3 个礼物。
>
> 最后剩下的礼物数量分别是 [5,8,9,4,3] ，所以，剩下礼物的总数量是 29 。

示例 2：

> 输入：gifts = [1,1,1,1], k = 4
> 输出：4
> 解释：
> 在本例中，不管选中哪一堆礼物，都必须剩下 1 个礼物。
> 也就是说，你无法获取任一堆中的礼物。
> 所以，剩下礼物的总数量是 4 。

提示：

- `1 <= gifts.length <= 103`
- `1 <= gifts[i] <= 109`
- `1 <= k <= 103`

```
class Solution {
    public long pickGifts(int[] gifts, int k) {
        PriorityQueue<Integer> pq = new PriorityQueue<>((x, y) -> (y - x));
        for (int i = 0; i < gifts.length; i++) {
            pq.add(gifts[i]);
        }

        for (int i = 0; i < k; i++) {
            Integer val = pq.poll();
            val = (int)Math.sqrt(val);
            pq.add(val);
        }
        long ans = 0;
        while (!pq.isEmpty()) {
            ans += pq.poll();
        }
        return ans;
    }
}
```

2559. 统计范围内的元音字符串数

给你一个下标从 **0** 开始的字符串数组 `words` 以及一个二维整数数组 `queries` 。

每个查询 `queries[i] = [li, ri]` 会要求我们统计在 `words` 中下标在 `li` 到 `ri` 范围内（**包含**这两个值）并且以元音开头和结尾的字符串的数目。

返回一个整数数组，其中数组的第 `i` 个元素对应第 `i` 个查询的答案。

注意：元音字母是 `'a'` 、 `'e'` 、 `'i'` 、 `'o'` 和 `'u'` 。

示例 1：

> 输入：words = ["aba","bcb","ece","aa","e"], queries = [[0,2],[1,4],[1,1]]
> 输出：[2,3,0]
> 解释：以元音开头和结尾的字符串是 "aba"、"ece"、"aa" 和 "e" 。
> 查询 [0,2] 结果为 2（字符串 "aba" 和 "ece"）。
> 查询 [1,4] 结果为 3（字符串 "ece"、"aa"、"e"）。
> 查询 [1,1] 结果为 0 。
> 返回结果 [2,3,0] 。

示例 2：

> 输入：words = ["a","e","i"], queries = [[0,2],[0,1],[2,2]]
> 输出：[3,2,1]
> 解释：每个字符串都满足这一条件，所以返回 [3,2,1] 。

提示：

- `1 <= words.length <= 105`
- `1 <= words[i].length <= 40`
- `words[i]` 仅由小写英文字母组成
- `sum(words[i].length) <= 3 * 105`
- `1 <= queries.length <= 105`
- `0 <= queries[j][0] <= queries[j][1] < words.length`

```
class Solution {
    public boolean isVowelChar(char c) {
        int[] vowelList = new int[]{'a', 'e', 'i', 'o', 'u'};
        for (int i = 0; i < vowelList.length; i++) {
            if (c == vowelList[i]) {
                return true;
            }
        }
        return false;
    }
    public int[] vowelStrings(String[] words, int[][] queries) {
        int[] dp = new int[words.length];
        int[] ans = new int[queries.length];

        if (isVowelChar(words[0].charAt(0)) &&
isVowelChar(words[0].charAt(words[0].length() - 1))) {
            dp[0] = 1;
        } else {
            dp[0] = 0;
        }

        for (int i = 1; i < words.length; i++) {
           char[] str = words[i].toCharArray();
            if (isVowelChar(str[0]) && isVowelChar(str[str.length - 1])) {
                dp[i] = dp[i - 1] + 1;
            } else {
                dp[i] = dp[i - 1];
            }
        }

        for (int i = 0; i < queries.length; i++) {
            if (queries[i][0] == 0) {
                ans[i] = dp[queries[i][1]];
            } else {
                ans[i] = dp[queries[i][1]] - dp[queries[i][0] - 1];
            }
        }

        return ans;
```

```
    }
}
```

2560. 打家劫舍 IV

沿街有一排连续的房屋。每间房屋内都藏有一定的现金。现在有一位小偷计划从这些房屋中窃取现金。

由于相邻的房屋装有相互连通的防盗系统，所以小偷 **不会窃取相邻的房屋** 。

小偷的 **窃取能力** 定义为他在窃取过程中能从单间房屋中窃取的 **最大金额** 。

给你一个整数数组 `nums` 表示每间房屋存放的现金金额。形式上，从左起第 `i` 间房屋中放有 `nums[i]` 美元。

另给你一个整数 `k` ，表示窃贼将会窃取的 **最少** 房屋数。小偷总能窃取至少 `k` 间房屋。

返回小偷的 **最小** 窃取能力。

示例 1：

> 输入：nums = [2,3,5,9], k = 2
> 输出：5
> 解释：
> 小偷窃取至少 2 间房屋，共有 3 种方式：
>
> - 窃取下标 0 和 2 处的房屋，窃取能力为 max(nums[0], nums[2]) = 5 。
> - 窃取下标 0 和 3 处的房屋，窃取能力为 max(nums[0], nums[3]) = 9 。
> - 窃取下标 1 和 3 处的房屋，窃取能力为 max(nums[1], nums[3]) = 9 。
>
> 因此，返回 min(5, 9, 9) = 5 。

示例 2：

> 输入：nums = [2,7,9,3,1], k = 2
> 输出：2
> 解释：共有 7 种窃取方式。窃取能力最小的情况所对应的方式是窃取下标 0 和 4 处的房屋。返回 max(nums[0], nums[4]) = 2 。

提示：

- `1 <= nums.length <= 105`
- `1 <= nums[i] <= 109`
- `1 <= k <= (nums.length + 1)/2`

```
class Solution {
    public int minCapability(int[] nums, int k) {
        if (nums.length == 1) {
            return nums[0];
        }

        int[] dp = new int[nums.length];
        int l = 0;
        int r = (int)1e9;
        int mid = 0;
        while (l + 1 < r) {
```

```
            mid = l + (r - l) / 2; // 当最大金额为 mid 的时候所能窃取的满足条件的最大房屋个数
            // System.out.println(mid);
            if (nums[0] > mid) {
                dp[0] = 0;
            } else {
                dp[0] = 1;
            }
            if (nums[1] > mid) {
                dp[1] = dp[0];
            } else {
                dp[1] = 1;
            }
            for (int i = 2; i < nums.length; i++) {
                if (nums[i] > mid) {
                    dp[i] = dp[i - 1];
                } else {
                    dp[i] = Math.max(dp[i - 2] + 1, dp[i - 1]);
                }
            }
            // System.out.println(dp[nums.length - 1]);
            if (dp[nums.length - 1] >= k) {
                r = mid;
            } else {
                l = mid;
            }
        }
        return r;
    }
}
```

2561. 重排水果

你有两个果篮，每个果篮中有 `n` 个水果。给你两个下标从 0 开始的整数数组 `basket1` 和 `basket2` ，用以表示两个果篮中每个水果的成本。

你希望两个果篮相等。为此，可以根据需要多次执行下述操作：

- 选中两个下标 `i` 和 `j` ，并交换 `basket1` 中的第 `i` 个水果和 `basket2` 中的第 `j` 个水果。
- 交换的成本是 `min(basket1i,basket2j)` 。

根据果篮中水果的成本进行排序，如果排序后结果完全相同，则认为两个果篮相等。

返回使两个果篮相等的最小交换成本，如果无法使两个果篮相等，则返回 `-1` 。

示例 1：

> 输入：basket1 = [4,2,2,2], basket2 = [1,4,1,2]
> 输出：1
> 解释：交换 basket1 中下标为 1 的水果和 basket2 中下标为 0 的水果，交换的成本为 1 。此时，basket1 = [4,1,2,2] 且 basket2 = [2,4,1,2] 。重排两个数组，发现二者相等。

示例 2：

> 输入：basket1 = [2,3,4,1], basket2 = [3,2,5,1]
> 输出：-1
> 解释：可以证明无法使两个果篮相等。

提示：

- `basket1.length == bakste2.length`
- `1 <= basket1.length <= 105`
- `1 <= basket1i,basket2i <= 109`

```
class Solution {
    public long minCost(int[] basket1, int[] basket2) {
        HashMap<Integer, Integer> map = new HashMap<>();

        for (int i = 0; i < basket1.length; i++) {
            int val = map.getOrDefault(basket1[i], 0);
            map.put(basket1[i], val + 1);
            val = map.getOrDefault(basket2[i], 0);
            map.put(basket2[i], val - 1);
        }

        int tmp = Integer.MAX_VALUE;
        ArrayList<Integer> list1 = new ArrayList<>();
        ArrayList<Integer> list2 = new ArrayList<>();
        for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
            int key = entry.getKey();
            int val = entry.getValue();

            if (val % 2 != 0) {
                return -1;
            }
            tmp = Math.min(tmp, key);
            for (int i = 0; i < Math.abs(val) / 2; i++) {
                if (val > 0) {
                    list1.add(key);
                } else {
                    list2.add(key);
                }

            }
        }

        Collections.sort(list1);
        Collections.sort(list2, (x, y) -> (y - x));
        long ans = 0;
        for (int i = 0; i < list1.size(); i++) {
            // System.out.println(list1.get(i) + " " + list2.get(i) + " " + tmp * 2);
            ans += Math.min(Math.min(list1.get(i), list2.get(i)), tmp * 2);
        }
        return ans;
    }
}
```

第 097 场双周赛

2553. 分割数组中数字的数位

给你一个正整数数组 `nums` ，请你返回一个数组 `answer` ，你需要将 `nums` 中每个整数进行数位分割后，按照 `nums` 中出现的 **相同顺序** 放入答案数组中。

对一个整数进行数位分割，指的是将整数各个数位按原本出现的顺序排列成数组。

- 比方说，整数 `10921` ，分割它的各个数位得到 `[1,0,9,2,1]` 。

示例 1：

> 输入：nums = [13,25,83,77]
> 输出：[1,3,2,5,8,3,7,7]
> 解释：
>
> - 分割 13 得到 [1,3] 。
> - 分割 25 得到 [2,5] 。
> - 分割 83 得到 [8,3] 。
> - 分割 77 得到 [7,7] 。
>
> answer = [1,3,2,5,8,3,7,7] 。 answer 中的数字分割结果按照原数字在数组中的相同顺序排列。

示例 2：

> 输入：nums = [7,1,3,9]
> 输出：[7,1,3,9]
> 解释：nums 中每个整数的分割是它自己。
> answer = [7,1,3,9] 。

提示：

- `1 <= nums.length <= 1000`
- `1 <= nums[i] <= 105`

```
class Solution {
    public int[] separateDigits(int[] nums) {
        ArrayList<Integer> arr = new ArrayList<>();

        for (int i = 0; i < nums.length; i++) {
            char[] str = String.valueOf(nums[i]).toCharArray();
            for (int j = 0; j < str.length; j++) {
                arr.add(str[j] - '0');
            }
        }

        int[] ans = new int[arr.size()];
        for (int i = 0; i < ans.length; i++) {
            ans[i] = arr.get(i);
        }
        return ans;
    }
}
```

2554. 从一个范围内选择最多整数 I

给你一个整数数组 `banned` 和两个整数 `n` 和 `maxSum` 。你需要按照以下规则选择一些整数：

- 被选择整数的范围是 `[1, n]` 。
- 每个整数 **至多** 选择 **一次** 。
- 被选择整数不能在数组 `banned` 中。
- 被选择整数的和不超过 `maxSum` 。

请你返回按照上述规则 **最多** 可以选择的整数数目。

示例 1：

> 输入：banned = [1,6,5], n = 5, maxSum = 6
> 输出：2
> 解释：你可以选择整数 2 和 4 。
> 2 和 4 在范围 [1, 5] 内，且它们都不在 banned 中，它们的和是 6 ，没有超过 maxSum 。

示例 2：

> 输入：banned = [1,2,3,4,5,6,7], n = 8, maxSum = 1
> 输出：0
> 解释：按照上述规则无法选择任何整数。

示例 3：

> 输入：banned = [11], n = 7, maxSum = 50
> 输出：7
> 解释：你可以选择整数 1, 2, 3, 4, 5, 6 和 7 。
> 它们都在范围 [1, 7] 中，且都没出现在 banned 中，它们的和是 28 ，没有超过 maxSum 。

提示：

- `1 <= banned.length <= 104`
- `1 <= banned[i], n <= 104`
- `1 <= maxSum <= 109`

```
class Solution {
    public int maxCount(int[] banned, int n, int maxSum) {
        HashSet<Integer> set = new HashSet<>();
        for (int i = 0; i < banned.length; i++) {
            set.add(banned[i]);
        }

        int ans = 0;
        int sum = 0;
        for (int i = 1; i <= n; i++) {
            if (set.contains(i) || sum + i > maxSum) {
                continue;
            }

            sum += i;
            ans++;
        }
        return ans;
    }
```

```
}
```

2555. 两个线段获得的最多奖品

在 **X轴** 上有一些奖品。给你一个整数数组 `prizePositions` ，它按照 **非递减** 顺序排列，其中 `prizePositions[i]` 是第 `i` 件奖品的位置。数轴上一个位置可能会有多件奖品。再给你一个整数 `k` 。

你可以选择两个端点为整数的线段。每个线段的长度都必须是 `k` 。你可以获得位置在任一线段上的所有奖品（包括线段的两个端点）。注意，两个线段可能会有相交。

- 比方说 `k = 2` ，你可以选择线段 `[1, 3]` 和 `[2, 4]` ，你可以获得满足 `1 <= prizePositions[i] <= 3` 或者 `2 <= prizePositions[i] <= 4` 的所有奖品 `i` 。

请你返回在选择两个最优线段的前提下，可以获得的 **最多** 奖品数目。

示例 1：

> 输入：prizePositions = [1,1,2,2,3,3,5], k = 2
> 输出：7
> 解释：这个例子中，你可以选择线段 [1, 3] 和 [3, 5] ，获得 7 个奖品。

示例 2：

> 输入：prizePositions = [1,2,3,4], k = 0
> 输出：2
> 解释：这个例子中，一个选择是选择线段 [3, 3] 和 [4, 4] ，获得 2 个奖品。

提示：

- `1 <= prizePositions.length <= 105`
- `1 <= prizePositions[i] <= 109`
- `0 <= k <= 109`
- `prizePositions` 有序非递减。

```
class Solution {
    public int maximizeWin(int[] prizePositions, int k) {
        // 在prizePositions数组中选择下标i, j, 如果prizePositions[j] - prizePositions[i]
<= k, 则获得的奖品数为j - i + 1
        int[] prefix = new int[prizePositions.length + 1];
        prefix[0] = 0; // 表示什么都不选
        int ans = 0;
        int l = 0;
        for (int r = 0; r < prizePositions.length; r++) {
            while (prizePositions[r] - prizePositions[l] > k) {
                l++;
            }
            ans = Math.max(ans, r - l + 1 + prefix[l]);
            prefix[r + 1] = Math.max(prefix[r], r - l + 1);
        }
        return ans;
    }
}
```

2556. 二进制矩阵中翻转最多一次使路径不连通

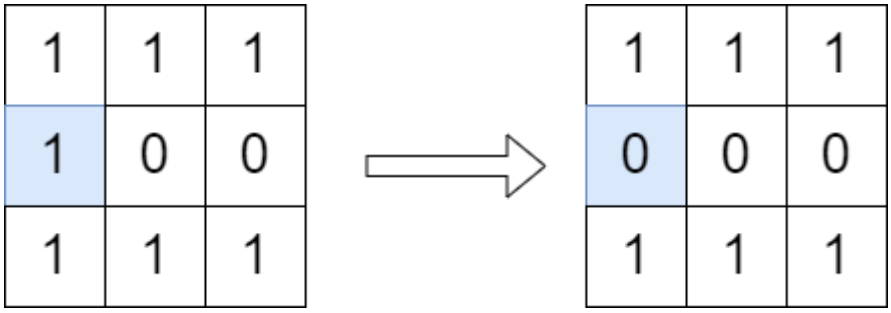
给你一个下标从 0 开始的 `m x n` **二进制** 矩阵 `grid` 。你可以从一个格子 `(row, col)` 移动到格子 `(row + 1, col)` 或者 `(row, col + 1)` ，前提是前往的格子值为 `1` 。如果从 `(0, 0)` 到 `(m - 1, n - 1)` 没有任何路径，我们称该矩阵是 **不连通** 的。

你可以翻转 **最多一个** 格子的值（也可以不翻转）。你 **不能翻转** 格子 `(0, 0)` 和 `(m - 1, n - 1)` 。

如果可以使矩阵不连通，请你返回 `true` ，否则返回 `false` 。

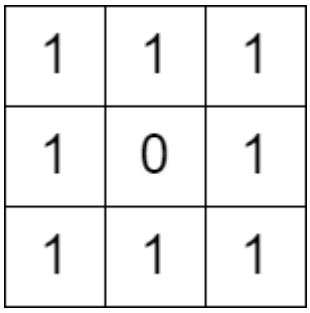
注意 ，翻转一个格子的值，可以使它的值从 `0` 变 `1` ，或从 `1` 变 `0` 。

示例 1：



> 输入：grid = [[1,1,1],[1,0,0],[1,1,1]]
> 输出：true
> 解释：按照上图所示我们翻转蓝色格子里的值，翻转后从 (0, 0) 到 (2, 2) 没有路径。

示例 2：



> 输入：grid = [[1,1,1],[1,0,1],[1,1,1]]
> 输出：false
> 解释：无法翻转至多一个格子，使 (0, 0) 到 (2, 2) 没有路径。

提示：

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 1000`
- `1 <= m * n <= 105`
- `grid[0][0] == grid[m - 1][n - 1] == 1`

```
class Solution {
    public boolean isConvertValid(int row, int col, int m, int n) {
```

```
        if (row == 0 && col == 0) {
            return false;
        }
        if (row == m - 1 && col == n - 1) {
            return false;
        }
        return true;
    }
    public boolean dfs(int[][] grid, int row, int col) {
        int m = grid.length;
        int n = grid[0].length;
        if (row < 0 || row >= m || col < 0 || col >= n) {
            return false;
        }

        if (grid[row][col] == 0) {
            return false;
        }
        if (isConvertValid(row, col, m, n)) {
            grid[row][col] = 0;
        }

        if (row == m - 1 && col == n - 1) {
            return true;
        }

        return dfs(grid, row + 1, col) || dfs(grid, row, col + 1);
    }

    public boolean isPossibleToCutPath(int[][] grid) {
        if (!dfs(grid, 0, 0)) { // 本身就不通
            return true;
        }

        // 连续两次dfs判断两条路径有没有交集，如果有，说明反转这个交集就可以不连通
        return !dfs(grid, 0, 0);
    }
}
```