dataStructure

deepwaterooo

2021 年 12 月 3 日

目录

Chapter 1

Segment Tree 与 Binary Index Tree 线段树与树状数组

线段树（segment tree），顾名思义，是用来存放给定区间（segment, or interval）内对应信息的一种数据结构。与树状数组（binary indexed tree）相似，线段树也用来处理数组相应的区间查询（range query）和元素更新（update）操作。与树状数组不同的是，线段树不止可以适用于区间求和的查询，也可以进行区间最大值，区间最小值（Range Minimum/Maximum Query problem）或者区间异或值的查询。

对应于树状数组，线段树进行更新（update）的操作为 O(logn)，进行区间查询（range query）的操作也为 O(logn)。

1.0.1 1157. Online Majority Element In Subarray - Hard

Design a data structure that efficiently finds the majority element of a given subarray.

The majority element of a subarray is an element that occurs threshold times or more in the subarray.

Implementing the MajorityChecker class:

MajorityChecker(int[] arr) Initializes the instance of the class with the given array arr. int query(int left, int right, int threshold) returns the element in the subarray arr[left...right] that occurs at least threshold times, or -1 if no such element exists.

```
private class Node {
    private int bgn;
    private int end;
    private int val;
    private int cnt; // sum
    private Node left;
    private Node right;
    public Node(int bgn, int end, int val, int cnt) {
        this.bgn = bgn;
        this.end = end;
        this.val = val;
        this.cnt = cnt;
        this.left = null;
        this.right = null;
    }
    public Node(int bgn, int end, int val, int cnt, Node left, Node right) {
        this.bgn = bgn;
        this.end = end;
        this.val = val;
        this.cnt = cnt;
        this.left = left;
        this.right = right;
    }
}
public void update(int index, int val) {
    updateTree(root, index, val);
}
public int cntRange(int left, int right) {
    return cntRangeFromTree(root, left, right);
}
private int cntRangeFromTree(Node r, int i, int j) {
    if (r == null || i < r.bgn || i > r.end) return 0;
    else if (i <= r.bgn && j >= r.end) return r.cnt;
    else return cntRangeFromTree(r.left, i, j) + cntRangeFromTree(r.right, i, j);
}
private void updateTree(Node r, int i, int va) {
    if (r == null || i < r.bgn || i > r.end) return;
    else if (r.bgn == r.end && r.bgn == i) r.cnt = va;
```

```
    else {
        updateTree(r.left, i, va);
        updateTree(r.right, i, va);
        int cnt = 0;
        if (r.left != null) cnt += r.left.cnt;
        if (r.right != null) cnt += r.right.cnt;
        r.cnt = cnt;
    }
}
private Node buildTree(int [] arr, int i, int j) {
    if (i > j) return null;
    else if (i == j)
        return new Node(i, i, arr[i], 1);
    else {
        int mid = i + (j-i)/2;
        Node left = buildTree(arr, i, mid);
        Node right = buildTree(arr, mid+1, j);
        if (left.val == right.val)
            return new Node(i, j, left.val, left.cnt + right.cnt, left, right);
        else {
            if (left.cnt > right.cnt)
                return new Node(i, j, left.val, left.cnt-right.cnt, left, right);
            else return new Node(i, j, right.val, right.cnt-left.cnt, left, right);
        }
    }
}
// 排序数组中 第一个大于 tar 的下标
int upper_bound(List<Integer> list, int tar) {
    int l = 0, r = list.size();
    while (l < r) {
        int mid = l + (r-l)/2;
        if (list.get(mid) <= tar) l = mid+1;
        else r = mid;
    }
    return l;
}
// 排序数组中 第一个大于等于 tar 的下标
int lower_bound(List<Integer> list, int tar) {
    int l = 0, r = list.size()-1;
    while (l < r) {
        int mid = l + (r-l)/2;
        if (list.get(mid) < tar) l = mid+1;
        else r = mid;
    }
    return l;
}
/**
 * 构建线段树
 * @param arr 被构建数组
 * @param l 构建节点的左值 表示查询区域左边界
 * @param r 构建节点的右值 表示查询区域右边界
 * @return 以构建完成的线段树节点
 * */
private SegTreeNode buildTree(int[] arr, int l, int r) {
    if (l > r) return null;
    // 初始一个线段树节点
    SegTreeNode root = new SegTreeNode(l, r);
    // 叶子节点
    if (l == r) {
        // 众数就是当前值 计数为 1
        root.val = arr[l]; root.count = 1;
        return root;
    }

    int mid = (l+r)/2;
    // 构建左子节点
    root.left = buildTree(arr, l, mid);
    // 构建右子节点
    root.right = buildTree(arr, mid+1, r);
    // 整合父节点
    makeRoot(root);
    return root;
}
/**
 * 整合一个父节点
 * @param root 被整合节点
 * */
private void makeRoot(SegTreeNode root) {
    if (null == root) return;
    // 如果该节点有左子节点 该节点的值" 先" 等于左子节点
    if (root.left != null) {
        root.val = root.left.val;
        root.count = root.left.count;
    }
    // 如果该节点还有右子节点 融合父节点和子节点
```

```
    if (root.right != null) {
        if (root.val == root.right.val)
            root.count = root.count + root.right.count;
        else {
            if (root.count >= root.right.count)
                root.count = root.count - root.right.count;
            else {
                root.val = root.right.val;
                root.count = root.right.count - root.count;
            }
        }
    }
}
/**
 * 查询线段树
 * @param root 被查询节点
 * @param l 需要查询的范围左边界
 * @param r 需要查询的范围右边界
 * */
private void searchSegTree(Node root, int l, int r) {
    if (root == null || l > r) return;
    if (root.bgn > r || root.end < l) return;

    // 当查询边界 覆盖 节点边界 该节点就是查询区域
    if (root.bgn >= l && root.end <= r) {
        if (key == root.val) cnt += root.cnt;
        else if (cnt <= root.cnt) {
            key = root.val;
            cnt = root.cnt - cnt;
        } else cnt = cnt - root.cnt;
        return;
    }

    int mid = (root.end + root.bgn)/2;
    // root.bgn <= l <= mid 左节点也可以是查询区域
    if (l <= mid)  // 这两个查询条件再好好想想 !!!!!!!!!!!!!!!
        searchSegTree(root.left, l, r);
    // mid+1 <= r <= root.end 右节点也可以是查询区域
    if (r >= mid+1)
        searchSegTree(root.right, l, r);
}
// https://books.halfrost.com/leetcode/ChapterFour/1100~1199/1157.Online-Majority-Element-In-Subarray/ 也有一个直观图
// https://www.cnblogs.com/slowbirdoflsh/p/11381565.html 思路比较清晰
HashMap<Integer, List<Integer>> idx = new HashMap<>();
private Node root;
int key = 0, cnt = 0;
public MajorityChecker(int[] arr) {
    root = buildTree(arr, 0, arr.length-1);
    levelPrintTree(root);
    idx = new HashMap<>();
    for (int i = 0; i < arr.length; i++) {
        if (!idx.containsKey(arr[i]))
            idx.put(arr[i], new ArrayList<>());
        idx.get(arr[i]).add(i);
    }
}
public int countRangeSum(int[] nums, int lower, int upper) {
    MajorityChecker mc = new MajorityChecker(nums);
}
public int query(int left, int right, int threshold) {
    // 初始化 所查询众数 key 及辅助判断的计数 cnt
    key = 0; cnt = 0;
    // 查询线段树
    searchSegTree(root, left, right);
    // 如果查询区域没有众数 即 key 没被更改
    // 或者
    // 所查询出来的众数 在原数组中根本没有超出阈值的能力
    System.out.println("key: " + key);
    System.out.println("(idx.get(key) == null): " + (idx.get(key) == null));

    if (key == 0 || idx.get(key).size() < threshold) return -1;

    // 上确界 排序数组中 第一个大于 right 的下标
    int r = upper_bound(idx.get(key), right);
    // 下确界 排序数组中 第一个大于等于 left 的下标
    int l = lower_bound(idx.get(key), left);
    cnt = r - l;
    return cnt >= threshold ? key : -1;
}
```

1.0.2 1825. Finding MK Average - Hard

You are given two integers, m and k, and a stream of integers. You are tasked to implement a data structure that calculates the MKAverage for the stream.

The MKAverage can be calculated using these steps:

If the number of the elements in the stream is less than m you should consider the MKAverage to be -1. Otherwise, copy the last m elements of the stream to a separate container. Remove the smallest k elements and the largest k elements from the container. Calculate the average value for the rest of the elements rounded down to the nearest integer. Implement the MKAverage class:

MKAverage(int m, int k) Initializes the MKAverage object with an empty stream and the two integers m and k. void addElement(int num) Inserts a new element num into the stream. int calculateMKAverage() Calculates and returns the MKAverage for the current stream rounded down to the nearest integer.

```
// 根据题意需要找到前 k 大的数，又需要求区间和，就自然想到线段树. 写起来较不容易出错。
// 维护 2 个线段树数组，一个记录数的个数，一个记录区间值，
// 注意一般线段树中 [s, e] 指固定的区间，这里类似线段数求第 k 小的数，所以 [s,e] 指第 s 小的值到第 e 小的值的区间。
LinkedList<Integer> q;
int [] cnt;
long[] sum;
int m,k;
public MKAverage(int m, int k) {
    q = new LinkedList<>();
    cnt = new int[400001]; // space: 4N
    sum = new long[400001];
    this.m = m;
    this.k = k;
}
public void addElement(int num) {
    if (q.size() == m) {
        int v = q.pollFirst();
        insert(1, 0, 100000, v, -1);
    }
    insert(1, 0, 100000, num, 1);
    q.addLast(num);
}
public int calculateMKAverage() {
    if (q.size() < m)return -1;
    int s = k+1, e = m-k;
    return (int)(query(1, 0, 100000, s, e)/(m-2*k));
}
void insert(int idx, int l, int r, int v, long d) {
    cnt[idx] += d;
    sum[idx] += d*v;
    if (l == r) return;
    int m = l + (r-l)/2;
    if (v <= m)
        insert(idx<<1, l, m, v, d);    // 向左子树查询
    else
        insert(idx<<1|1, m+1, r, v, d);// 向右子树查询
}
long query(int idx, int l, int r, int s, int e){//线段中第 s 个到第 e 个
    if (l == r) {//起始和结束最多出现 2 次此情况
        int c = e-s+1;
        return (long)c*l;
    } else if (cnt[idx] == e-s+1) {
        return sum[idx];
    } else {
        int m = (l+r)/2;
        int c1 = cnt[idx<<1];
        int c2 = cnt[idx<<1|1];
        if (c1 >= e){
            return query(idx<<1, l, m, s, e);
        } else if (c1 >= s){
            return query(idx<<1, l, m, s, c1)+query(idx<<1|1, m+1, r, 1, e-c1);
        } else {//c1<s
            return query(idx<<1|1, m+1, r, s-c1, e-c1);
        }
    }
}
```

- 数状数组的解法: 另外第一次看到别人二分 + 树状数组也能求前 k 大的值。

```
// We can have a queue to maintain m elements
// Use two Fenwick tree, 1 for count and 1 for prefix sum
// Do 2 times binary search for the first k elements and the last k elements by using the count from our first fenwick tree
// We can get the sum by subtrating the sum of first k elements and sum of last k element by using our second fenwick tree
Queue<Integer> q = new LinkedList<>();
```

```
FenWick fone, ftwo;
int [] cnt = new int [100010];
long sum = 0;
int m,k;
public MKAverage(int m, int k) {
    this.m = m;
    this.k = k;
    long A [] = new long [100010];
    long B [] = new long [100010];
    fone = new FenWick(A);
    ftwo = new FenWick(B);
}
public void addElement(int num) {
    q.add(num);
    sum += num;
    fone.update(num, 1);
    ftwo.update(num, num);
    cnt[num]++;
}
public int calculateMKAverage() {
    if (q.size() < m) return -1;
    while (q.size() > m) {
        int cur = q.poll();
        cnt[cur]--;
        sum -= cur;
        fone.update(cur, -1);
        ftwo.update(cur, -cur);
    }
    // binary search for the first k (there may be duplicated)
    int l = 0, r = cnt.length-1;
    int i = -1, j = -1; // pos1, pos2
    while (l <= r) { // 二分查找总计数
        int m = (r + l) / 2;
        long count = fone.sumRange(0, m);
        if (count >= k) {
            i = m;
            r = m -1;
        } else l = m+1;
    }
    // binary search for the last k (there may be duplicated)
    l = 0;
    r = cnt.length-1;
    while (l <= r) {
        int m = l + (r-l)/2;
        long count = fone.sumRange(m, cnt.length-1);
        if (count >= k) {
            j = m;
            l = m + 1;
        } else r = m-1;
    }
    long sum1 = ftwo.sumRange(0,  i);
    long sum2 = ftwo.sumRange(j, cnt.length-1);
    long cnt1 = fone.sumRange(0, i);
    long cnt2 = fone.sumRange(j, cnt.length-1);
    if (cnt1 > k)
        sum1 -= i*(cnt1-k);
    if (cnt2 > k)
        sum2 -= j*(cnt2-k);
    long remain = sum - sum1 - sum2; // 总和, 减去两边最小最大各 K 个数的和
    return (int)(remain / (m-2*k));
}
class FenWick {
    long tree []; //1-index based
    long A [];
    long arr[];
    public FenWick(long [] A) {
        this.A = A;
        arr = new long [A.length];
        tree = new long [A.length + 1];
    }
    public void update(int i, int v) {
        arr[i] += v;
        i++;
        while (i < tree.length) {
            tree[i] += v;
            i += (i & -i); // 这是的原理细节再回去复习一下
        }
    }
    public long sumRange(int i, int j) {
        return pre(j+1)-pre(i);
    }
    public long pre(int i) {
        long sum = 0;
        while (i > 0) {
            sum += tree[i];
```

```
            i -= (i & -i);
        }
        return sum;
    }
}
```

1.0.3 699. Falling Squares - Hard

There are several squares being dropped onto the X-axis of a 2D plane.

You are given a 2D integer array positions where positions[i] = [lefti, sideLengthi] represents the ith square with a side length of sideLengthi that is dropped with its left edge aligned with X-coordinate lefti.

Each square is dropped one at a time from a height above any landed squares. It then falls downward (negative Y direction) until it either lands on the top side of another square or on the X-axis. A square brushing the left/right side of another square does not count as landing on it. Once it lands, it freezes in place and cannot be moved.

After each square is dropped, you must record the height of the current tallest stack of squares.

Return an integer array ans where ans[i] represents the height described above after dropping the ith square.

1. 解题思路与分析: O(N^2) 本能土办法 方块的大小不是固定的，有可能很大，但是不管方块再大，只要有一点点部分搭在其他方块上面，整个方块都会在上面，并不会掉下来，让我们求每落下一个方块后的最大高度。我们知道返回的是每落下一个方块后当前场景中的最大高度，那么返回的数组的长度就应该和落下方块的个数相同。所以我们可以建立一个 heights 数组，其中 heights[i] 表示第 i 块方块落下后所在的高度，那么第 i 块方块落下后场景的最大高度就是 [0, i] 区间内的最大值。那么我们在求出 heights 数组后，只要不停返回 [0, i] 区间内的最大值即可。继续来看，这道题的难点就是方块重叠的情况，我们先来想，如果各个方块不重叠，那么 heights[i] 的高度就是每个方块自身的高度。一旦重叠了，就得在已有的基础上再加上自身的高度。那么我们可以采用 brute force 的思想，对于每个一个下落的方块，我们都去看和后面将要落下的方块有没有重叠，有的话，和后面将要落下的方块的位置相比较，取二者中较大值为后面要落下的方块位置高度 heights[j]。判读两个方块是否重叠的方法是如果方块 2 的左边界小于方块 1 的右边界，并且方块 2 点右边界大于方块 1 点左边界。就拿题目中的例子 1 来举例吧，第一个下落的方块的范围是 [1, 3]，长度为 2，则 heights[1]=2，然后我们看其和第二个方块 [2, 5] 是否重叠，发现是重叠的，则 heights[2]更新为 2，再看第三个方块 [6, 7]，不重叠，不更新。然后第二个方块落下，此时累加高度，则 heights[2]=5，再看第三个方块，不重叠，不更新。然后第三个方块落下，heights[3]=1。此时我们 heights 数组更新好了，然后我们开始从头遍历，维护一个当前最大值 curMax，每次将 [0, i] 中最大值加入结果 res 即可，

```
public List<Integer> fallingSquares(int[][] p) {
    List<Integer> ans = new ArrayList<>();
    int n = p.length, cur = 0;
    int [] hi = new int [n];
    for (int i = 0; i < n; i++) {
        int len = p[i][1], l = p[i][0], r = l + len;
        hi[i] += len;
        for (int j = i+1; j < n; j++) { // 采用 brute force 的思想，对于每个一个下落的方块，我们都去看和后面将要落下的方块有没有重叠
            int ll = p[j][0], rr = ll + p[j][1]; // 有的话，和后面将要落下的方块的位置相比较，取二者中较大值为后面要落下的方块位置高度 heights[j]
            // [[6,1],[9,2],[2,4]] 因为不能保证是从左往下延 x 轴顺序掉落，所以加上 l < rr 也很重要 确保不管左右边有交叠
            if (ll < r && l < rr) // 判读两个方块是否重叠的方法是如果方块 2 的左边界小于方块 1 的右边界，并且方块 2 点右边界大于方块 1 点左边界
                hi[j] = Math.max(hi[j], hi[i]); // 这里提前检查是否重叠，是为后来的更新打下 base，再接下来遍历 i 时直接累加就可以了
        }
    }
    for (int v : hi) {
        cur = Math.max(cur, v);
        ans.add(cur);
    }
    return ans;
}
```

2. 解题思路与分析：线段树 + 离散化

想象 x xx 轴是地面，如果某个方块掉落的过程中遇到了之前的某个方块（擦边而过不算），则该方块会叠到上面。现在给定一个长 n nn 数组 A AA，A [i] A[i]A[i] 存了第 i ii 个掉落的方块的信息，其中 A [i] [0] A[i]^{1}A[i]^{1}表示它的左下角的 x xx 坐标，A [i] [1] A[i]^{2}A[i]^{2}表示它的边长。要求返回一个长 n nn 数组 B BB，使得 B [i] B[i]B[i] 表示在 A [i] A[i]A[i] 掉落之后，当前所有方块的最高点的 y yy 坐标。

思路是线段树 + 离散化。可以将 x xx 坐标离散化，这样可以节省存储空间（离散化的过程其实就是将一个数组 d dd 排序后去重，然后将每个数映射到它的下标。这样在线段树建树的时候，就只需维护 [0 , l d 1] [0,l$_d$-1][0,l$_{d-1}$]

xx 坐标和右端点的 x xx 坐标分别是 a aa 和 b bb，边长是 c cc，那么我们需要实现两个操作，第一是查询 (a , b) (a,b)(a,b)

[1] DEFINITION NOT FOUND.

[2] DEFINITION NOT FOUND.

[3] DEFINITION NOT FOUND.

里的最大值 M MM（注意这里查询的是开区间 (a , b) (a,b)(a,b) 的最大值，因为下落的方块擦着另一个方块的边的话，是不会叠上去的），另一个是将 [a , b] [a,b][a,b] 里所有值都变成 M + c M+cM+c。本质上是要求一个数据结构可以查询区间最大值，以及将区间修改为某一值，这可以用线段树 + 懒标记来做到。在离散化之后，为了使得区间 (a , b) (a,b)(a,b) 非空（注意这里 a aa 和 b bb 都是离散化之后的值，此时 (a , b) = [a + 1 , b 1] (a,b)=[a+1,b-1](a,b)=[a+1,b1]），我们可以在离散化的时候将方块的中点也加入一起做离散化，但是这会导致中点变成非整数，这里将原坐标乘以 2 22 就行了。

```
class SegTree {  // 实现一下带懒标记的线段树：这棵树好强大
    class Node { // v 是 [l, r] 区间的最大值，lazy 是懒标记
        int l, r, v, lazy;
        public Node(int l, int r) {
            this.l = l;
            this.r = r;
        }
    }
    private Node [] tr;
    public SegTree(int size) {
        tr = new Node[size << 2]; // * 4
        build(1, 0, size - 1);
    }
    public void build(int u, int l, int r) { // 下标从 1 开始 自顶向下
        tr[u] = new Node(l, r);
        if (l == r) return;
        int mid = l + r >> 1; // / 2
        build(u << 1, l, mid);          // 分别构建左右子树
        build(u << 1 | 1, mid + 1, r);
    }
    private void pushup(int u) { // 最大树：当前节点的值为左右子节点的最大值
        tr[u].v = Math.max(tr[u << 1].v, tr[u << 1 | 1].v);
    }
    private void pushdown(int u) { // 下传懒标记
        int c = tr[u].lazy;
        if (c != 0) {
            tr[u].lazy = 0;
            tr[u << 1].v = tr[u << 1 | 1].v = c; // 根据父节点的值，懒标记下传一层，更新左右子树的值，并更新下传而来的标记
            tr[u << 1].lazy = tr[u << 1 | 1].lazy = c;
        }
    }
    public void update(int u, int l, int r, int c) {
        // for (int i = 1; i < tr.length; i++)
        //     if (tr[i] != null)
        //         System.out.println("[" + tr[i].l + "," + tr[i].r + "], v: " + tr[i].v + ", lazy: " + tr[i].lazy);

        if (l <= tr[u].l && tr[u].r <= r) { // 任务不需要下发，可以用懒标记懒住
            tr[u].v = tr[u].lazy = c;       // 整棵树落在左右区间内，可以发懒，暂不下传
            // System.out.println("[" + tr[u].l + "," + tr[u].r + "], v: " + tr[u].v + ", lazy: " + tr[u].lazy);
            return;
        }
        pushdown(u); // 任务不得不下发，则先下发给两个孩子
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) update(u << 1, l, r, c); // 回归调用，下传更新至左右子节点
        if (mid + 1 <= r) update(u << 1 | 1, l, r, c);
        pushup(u); // 孩子完成了任务，再修改自己的值
    }
    public int query(int u, int l, int r) {
        if (l <= tr[u].l && tr[u].r <= r) return tr[u].v;
        pushdown(u);
        int res = 0, mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) res = Math.max(res, query(u << 1, l, r));
        if (mid + 1 <= r) res = Math.max(res, query(u << 1 | 1, l, r));
        return res;
    }
    public int query() {
        return tr[1].v;
    }
}
public List<Integer> fallingSquares(int[][] positions) {
    List<Integer> xs = new ArrayList<>();
    for (int[] p : positions) {
        int a = p[0], b = a + p[1];
        xs.add(a * 2);
        xs.add(b * 2);
        xs.add(a + b);
    }
    xs = unique(xs); // 排序并去重
    SegTree segTree = new SegTree(xs.size());
    List<Integer> res = new ArrayList<>();
    for (int [] p : positions) {
        int a = p[0], b = a + p[1];
        a = get(a * 2, xs);
        b = get(b * 2, xs);
        int h = segTree.query(1, a + 1, b - 1);
```

```
            segTree.update(1, a, b, h + p[1]);
            res.add(segTree.query());
        }
        return res;
    }
    private int get(int x, List<Integer> xs) { // 找到 x 在离散化之后的值是多少，其实就是求 xs 里 x 的下标，可以二分来找到
        int l = 0, r = xs.size() - 1;
        while (l < r) {
            int m = l + r >> 1;
            if (xs.get(m) >= x)
                r = m;
            else
                l = m + 1;
        }
        return l;
    }
    private List<Integer> unique(List<Integer> list) { // 将 list 排序后去重
        list.sort(Integer::compareTo);
        int j = 0;
        for (int i = 0; i < list.size(); i++)
            if (i == 0 || list.get(j - 1) != list.get(i))
                list.set(j++, list.get(i));
        return list.subList(0, j); // subList()
    }
```

1.0.4 1483. Kth Ancestor of a Tree Node - Hard 倍增法 binary lifting

You are given a tree with n nodes numbered from 0 to n - 1 in the form of a parent array parent where parent[i] is the parent of ith node. The root of the tree is node 0. Find the kth ancestor of a given node.

The kth ancestor of a tree node is the kth node in the path from that node to the root node.

Implement the TreeAncestor class:

TreeAncestor(int n, int[] parent) Initializes the object with the number of nodes in the tree and the parent array. int getKthAncestor(int node, int k) return the kth ancestor of the given node node. If there is no such ancestor, return -1.

1. 解题思路与分析: 倍增 binary lifting

给定一个n个节点的树，每个节点编号$0 \sim n-1$，另外给出每个节点的父亲节点是谁，以数组p给出。要求设计一个数据结构可以应答这样的询问，每次询问给出一个节点编号u和一个正整数k，问u的第k个祖先是谁（即u向上走k步是谁）。如果该祖先不存在则返回-1。

可以用倍增法。设$f[u][k]$是节点u向上走2^k步能走到的节点，那么有

$$f[u][k] = f[f[u][k-1]][k-1]$$

初始条件$f[u][0] = p[u]$，从而可以递推出f数组。因为一共有n个节点，树最高就是n，也就是说向上最多能走$n-1$步，我们找到最小的$2^k \geq n-1$，把f的第二维开$k = \lfloor \log_2 n - 1 \rfloor + 1$这么长，以保证询问都能被正确应答。接下来考虑如何应答询问，可以用类似快速幂的思想，比方说k的二进制表示是1101_2，那么$1101_2 = 2^0 + 2^2 + 2^3$，那么可以先求$u_1 = f[u][0]$，这样就跳了2^0步，再接着求$u_2 = f[u_1][2]$，这样又跳了2^2步，再接者求$f[u_2][3]$，这样就将k步全跳完了。中途如果发现要跳的幂次大于了f的第二维能取的最大值，或者跳到了-1，则直接返回-1。代码如下：

- 预处理时间复杂度 O(nlogn)，每次询问时间 O(logn)，空间 O(nlogn)。

```
    private int [][] p;
    private int log;
    public TreeAncestor(int n, int[] parent) {
        log = (int) (Math.log(n - 1) / Math.log(2)) + 1;
        p = new int[n][log];
        for (int i = 0; i < parent.length; i++) // 初始化 p 数组
            p[i][0] = parent[i];
        for (int i = 1; i < log; i++) // 按公式递推 p 数组
            for (int j = 0; j < n; j++)
                if (p[j][i-1] != -1)
                    p[j][i] = p[p[j][i-1]][i-1];
                else p[j][i] = -1;
    }
    public int getKthAncestor(int node, int k) {
        int pow = 0;
```

```
        while (k > 0) {
            if (pow >= log || node == -1) return -1;
            if ((k & 1) == 1)
                node = p[node][pow];
            k >>= 1;
            pow++;
        }
        return node;
    }
```

2. 解题思路与分析

```
    Map<Integer, List<Integer>> adj;
    int [][] par;
    public TreeAncestor(int n, int[] parent) {
        par = new int [n][30]; // 30 , 16: 不能证它是一棵很平衡的二叉树
        adj = new HashMap<>();
        for (int i = 0; i < n; i++) {
            Arrays.fill(par[i], -1);
            adj.put(i, new ArrayList<>());
        }
        for (int i = 0; i < parent.length; i++)
            if (parent[i] != -1) {
                adj.get(parent[i]).add(i); // 自顶向下: 父 --》子节点
                par[i][0] = parent[i];     // 每个子节点的第一个父节点 (2^0 = 1), 即为父节点 // 自底向上: 子节点: 2^0 父节点、 2^1 节点、 2^2 节点
            }
        dfs(0);
    }
    public int getKthAncestor(int node, int k) {
        for (int i = 0; k > 0; i++, k >>= 1) // k /= 2
            if ((k & 1) == 1) {
                node = par[node][i];
                if (node < 0) return -1;
            }
        return node;
    }
    private void dfs(int idx) { // 自顶向下: 从父节点遍历子节点
        for (int i = 1; par[idx][i-1] >= 0; i++) // 穷追塑源: 一直找到整棵树的根节点: 0
            par[idx][i] = par[par[idx][i-1]][i-1]; // 这里多想想
        for (int next : adj.get(idx))
            dfs(next);
    }
```

1.0.5 236 二叉树的最近公共祖先

Chapter 2

Trie

应用 Trie 树最直观的定义就是 LinkedList of HashMap。所以 Trie 和 HashMap 都可以用来查询某个单词是否在字典当中。我们需要知道他们的优缺点。优点：支持字符级别的查询，比如说我们需要在 matrix 当中通过 traverse 构造单词，那么这个单词是一个一个字符形成的，我们可以在 traverse 的每一步去检验当前路径是否可以形成 valid word。另外，对于含有 regex 符号的字符串，我们需要一个字符一个字符的考虑，这种情况下我们也需要通过 trie 去查找。节省空间，相同的 prefix 只存一遍，而 HashMap 需要存很多遍。缺点：实现起来较麻烦，大部分题目使用 Trie 都是 overkill，所以除非需要支持字符级别的查询，否则 HashMap 更好。操作：三个操作：insert search startWith 其中 insert 记得把最后一个 node 标记为 isEnd = true。其中 search 和 startWith 都可以通过同一个 searchHelper helper method 来实现，我们只需要 return 最后一个 node 就可以，如果 isEnd == true，那么说明找到一个完整的单词，否则至少找到了 prefix。别忘了使用 trie 的第一步是 preprocess，把字典里的所有 word 加入到 trie 树当中。题目

2.1 208. Implement Trie (Prefix Tree)

2.1.1 212. Word Search II

2.1.2 211. Add and Search Word - Data structure design (Facebook 店面)

2.1.3 14. Longest Common Prefix (这道题可以稍作改编，比如说 **string list** 会经常 **update**，会经常 **query**，那这时很明显用 **trie** 更好)

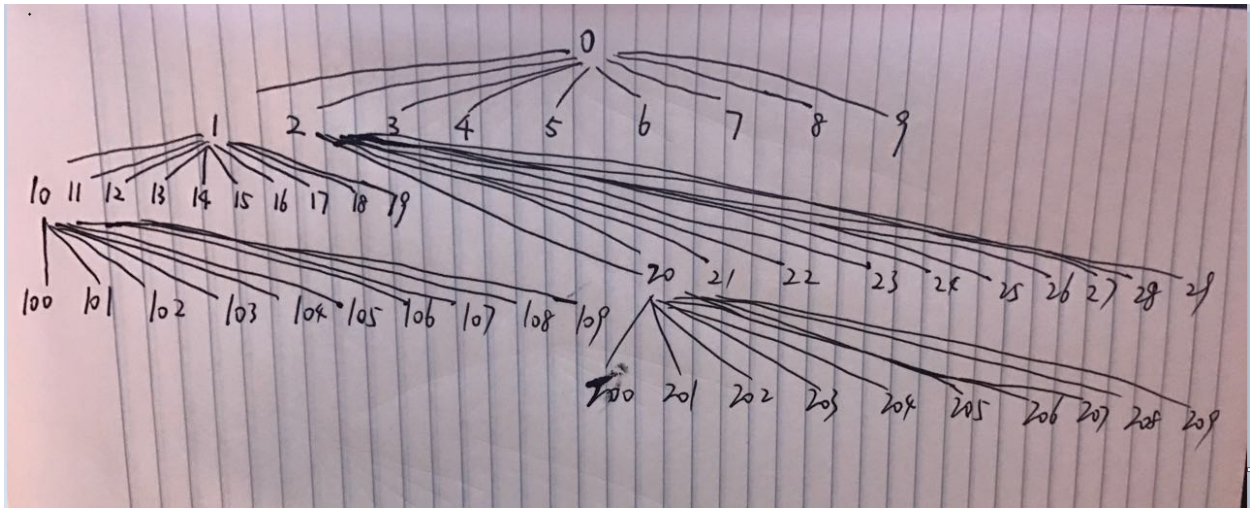
2.1.4 440. K-th Smallest in Lexicographical Order - Hard

Given two integers n and k, return the kth lexicographically smallest integer in the range [1, n].

1. 解题思路与分析 就像 dfs 时当我们需要两个字符串，遍历字符串，我们并不需要看的去遍历字符串，我们只要移动下标就可以了

   这里我们并不需要真的去建和遍历这样一个字典，我们只要理清数字个数之间的关系就可以了

   还要一个典型案例，把它找出来。。。。todo



Initially, image you are at node 1 (variable: curr),
the goal is move (k - 1) steps to the target node x. (substract steps from k after moving)
when k is down to 0, curr will be finally at node x, there you get the result.

we don't really need to do a exact k steps preorder traverse of the denary tree, **the idea is to calculate the steps between curr and curr + 1 (neighbor nodes in same level), in order to skip some unnecessary moves.**

Main function
Firstly, calculate how many steps curr need to move to curr + 1.

1. if the steps <= k, we know we can move to curr + 1, and narrow down k to k - steps.
2. else if the steps > k, that means the curr + 1 is actually behind the target node x in the preorder path, we can't jump to curr + 1. What we have to do is to move forward only 1 step (curr * 10 is always next preorder node) and repeat the iteration.

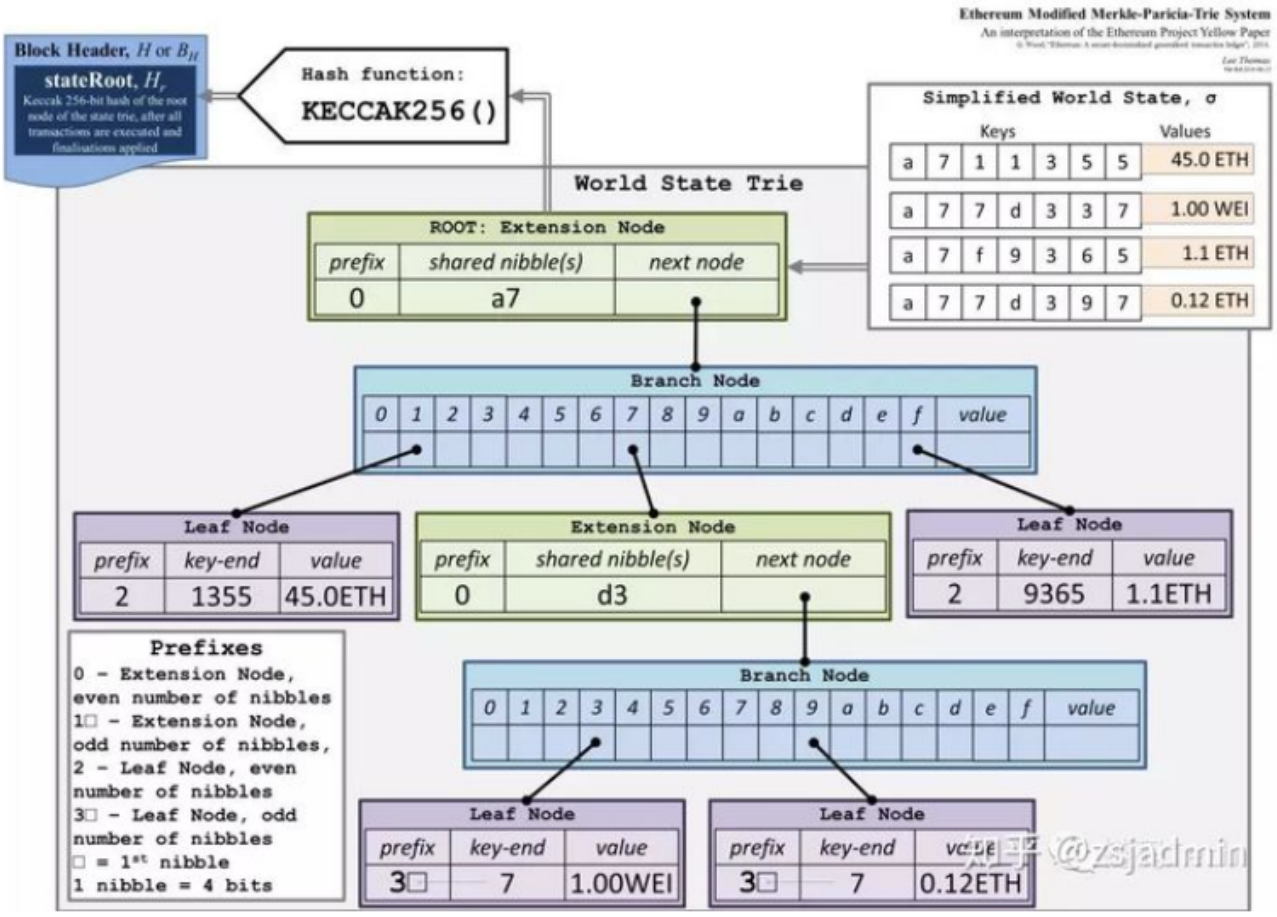
calSteps function

1. how to calculate the steps between curr and curr + 1?
   Here we come up a idea to calculate by level.
   Let n1 = curr, n2 = curr + 1.
   n2 is always the next right node beside n1's right most node (who shares the same ancestor "curr")
   (refer to the pic, 2 is right next to 1, 20 is right next to 19, 200 is right next to 199).
2. so, if n2 <= n, what means n1's right most node exists, we can simply add the number of nodes from n1 to n2 to steps.
3. else if n2 > n, what means n (the biggest node) is on the path between n1 to n2, add (n + 1 - n1) to steps.
4. organize this flow to "steps += Math.min(n + 1, n2) - n1; n1 *= 10; n2 *= 10;"

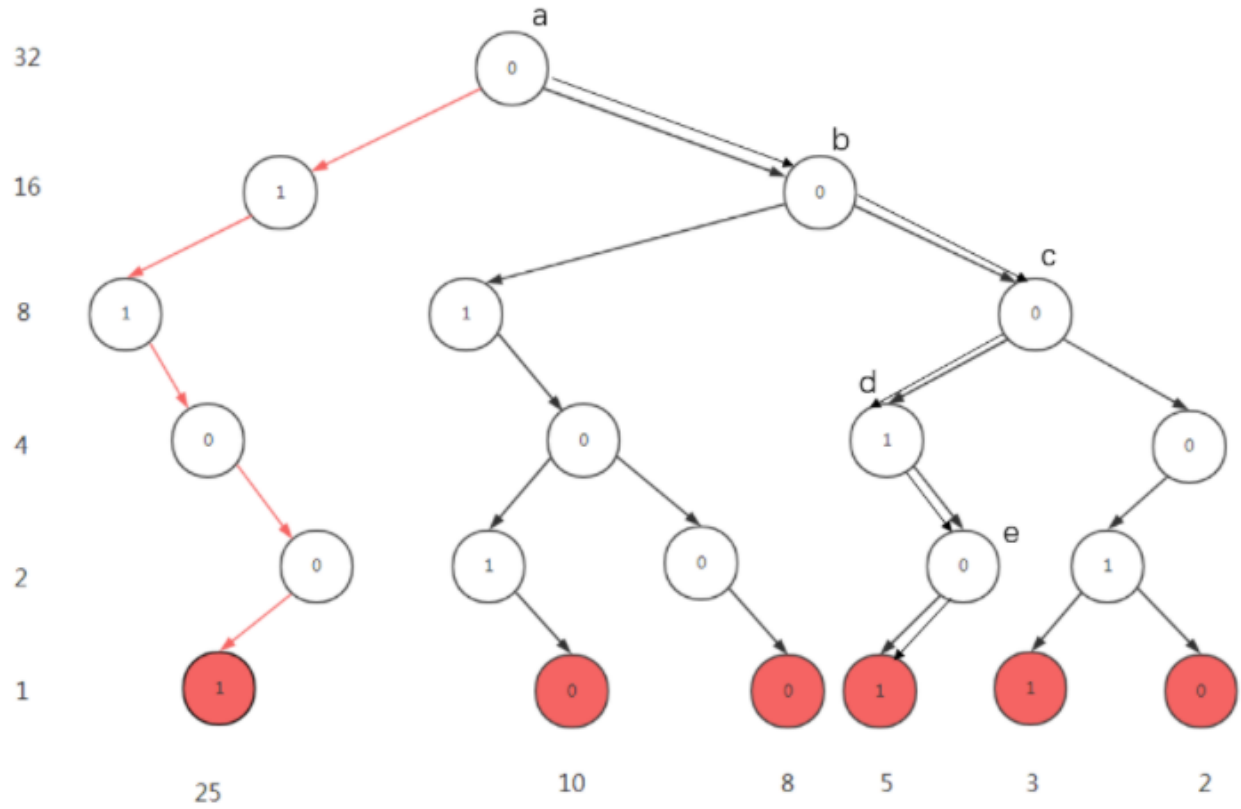
```
private int calSteps(int n, long n1, long n2) { // n1 和 n2 得是 long 类型的, int 会产生溢出, 不能通过这个案例: 输入 n=681692778, k=351251360, 预期结果
    int steps = 0;
    while (n1 <= n) {
        steps += Math.min(n2, n+1) - n1;
        n1 *= 10;
        n2 *= 10;
    }
    return steps;
}
public int findKthNumber(int n, int k) {
    int cur = 1; //根据题意, 第一个数是 1
    --k;         //第一个是 1, 所以再找出 k-1 个数后就知道第 k 个数是多少了
    while (k > 0) {
        int steps = calSteps(n, cur, cur+1);
        if (steps <= k) { //横向扩展, 相当于 +steps,
            cur += 1;
            k -= steps;
        } else {          //steps > k; 纵向扩展, 相当于 +1
            cur *= 10;
            k -= 1;
        }
    }
    return cur;
}
```

2.1.5 421. Maximum XOR of Two Numbers in an Array

Given an integer array nums, return the maximum result of nums[i] XOR nums[j], where 0 <= i <= j < n.



以样例 `[3, 10, 5, 25, 2, 8]` 为例，建完树之后可以得到下图：



左儿子为 1 的分支，右儿子为 0 的分支。
然后依次枚举每个数，在 Trie 树中找到与它异或结果最大的数。
这一步可以贪心来做：

从高位到低位，依次在 Trie 树中遍历，每次尽量走到与当前位不同的分支，这样可以使得找到的数与当前数在当前二进制位的异或结果是 1，从而可以得到尽量大的结果。

如上图所示，我们用 25 来举例说明，它的二进制表示是 (11001)：

```
最初指针在根节点(编号是 a 的点)，我们从 25 的二进制表示的最高位开始枚举；
  由于最高位是 1，我们走到 0 分支，走到 b 点；
  次高位是 1，我们继续往右儿子走，走到 c 点；
  下一位是 0，我们往左走，走到 d 点；
  下一位是 0，我们希望往左走，但发现左儿子不存在，所以只能往右走，走到 e 点；
  最后一位是 1，我们希望往右走，但发现右儿子不存在，所以只能往左走，最终走到 5；
所以和 25 异或值最大的数是 5，25 ^ 5 = 28。
```

```
public class Trie {
    private class Node { // 这我自己写的乱代码，贴在这里很不相关，也需要先测试一下
        public int val;
        public boolean isExist;
        public Node [] next;
        public Node(boolean isExist) {
            this.isExist = isExist;
            next = new Node[2];
            val = 0;
        }
        public Node() { this(false); }
        public Node(int va) {
            this(true);
            this.val = va;
        }
    }
    private Node root;
    public Trie() { root = new Node(); }
    public void insert(int va) {
        Node cur = root;
        for (int i = 31; i >= 0; i--) {
            int tmp = (va >> i) & 1;
            if (cur.next[tmp] == null)
                cur.next[tmp] = new Node();
            cur = cur.next[tmp];
        }
        cur.isExist = true;
    }
    public int search(int va) {
        int max = 0;
        Node cur = root;
        for (int i = 31; i >= 0; i--) {
            int t = (va >> i) & 1;
            if (cur.next[t^1] != null) {
                max += (1 << i);
                cur = cur.next[t^1];
            } else cur = cur.next[t&1];
        }
        return max;
    }
}
```

2.1.6 另一种位操作法

- 学到了异或操作的一个重要性质：$a^b = c$, 则有 $a^c = b$，且 $b^c = a$;

我们还需要用上一个异或的特性，假设 a 和 b 产生了最终的答案 max，即 a ^ b = x，那么根据异或的特性，a ^ x = b。同理，a 和 b 的最高位（前 n 位）也有相同的性质。

先以最高位为例子，我们可以把所有的数字的最高位放到一个 HashSet 里面，然后使用 1 与 set 里面的所有数字进行异或，如果得出的结果仍然在 set 里面，那么最终结果的最高位必然为 1，否则为 0。也即，先假定结果为 1，然后与 set 中所有数字异或，假定 a 与 1 异或得到结果 b（a ^ 1 = b），而 b 仍然在 set 里面，那么说明 set 中有两个数字异或能得到 1（a ^ b = 1）。否则，set 中没有两个数字能够异或得到 1，那么最终结果的最高位为 1 的假设失败，说明最终结果的最高位为 0。以此类推可以得到第二位、第三位。。。的数字。

再做一下推广，我们将所有数字的前 N 位放到一个 HashSet 里面，然后使用之前 N-1 位得到的最大值前缀 prefix 与 set 里面的所有数字进行异或，如果得出的结果仍然在 set 中，那么第 N 位必然为 1，否则为 0。

举个例子，给定数组 [14, 11, 7, 2]，二进制表示分别为 [1110, 1011, 0111, 0010]。题目说了，数字最长不会超过 32 位，所以应从 i = 31 开始，但是所有数字中最多位 4 位数，简单起见，我直接从最高位 i=3 开始

```
[14,   11,   7,    2]
[1110, 1011, 0111, 0010]
```

```
1. i = 3, set = {1000, 0000} => max = 1000
2. i = 2, set = {1100, 1000, 0100, 0000} => max = 1100
3. i = 1, set = {1110, 1010, 0110, 0010} => max = 1100
4. i = 0, set = {1110, 1011, 0111, 0010} => max = 1100
```

```
public int findMaximumXOR(int[] nums) { // 这种解法没有用到上面的这个 trie 呀
    int n = nums.length;
    int mask = 0, max = 0;
    HashSet<Integer> s = new HashSet<>();
    for (int i = 31; i >= 0; --i) { // i == 31 时
        mask = mask | 1 << i;     // 为获取前 n 位的临时变量
        for (int va : nums)
            s.add(va & mask);     // 将所有数字的前 n 位放入 set 中
        int tmp = max | (1 << i); // 假定第 n 位为 1, 前 n-1 位 max 为之前迭代求得
        for (Integer va : s)
            if (s.contains(va ^ tmp)) { // 查看 `b`是否在 // i == 31, (va^tmp):  -2147483648
                max = tmp;              // b 存在, 第 n 位为 1
                break;
            }
        s.clear();
    }
    return max;
}
// 此解法时间复杂度为 O(32n)=O(n), 空间复杂度上, 我们使用了一个 HashSet 用于存储所有数字, 因此空间复杂度是 O(n)
```

2.1.7 1617. Count Subtrees With Max Distance Between Cities - Hard

There are n cities numbered from 1 to n. You are given an array edges of size n-1, where edges[i] = [ui, vi] represents a bidirectional edge between cities ui and vi. There exists a unique path between each pair of cities. In other words, the cities form a tree.

A subtree is a subset of cities where every city is reachable from every other city in the subset, where the path between each pair passes through only the cities from the subset. Two subtrees are different if there is a city in one subtree that is not present in the other.

For each d from 1 to n-1, find the number of subtrees in which the maximum distance between any two cities in the subtree is equal to d.

Return an array of size n-1 where the dth element (1-indexed) is the number of subtrees in which the maximum distance between any two cities is equal to d.

Notice that the distance between the two cities is the number of edges in the path between them.

- So apparently the brute-force approach passed this question. I guess for future contests, I should really pay attention to the input size. . .

1. 解题思路与分析: 自己凭感觉写的，看别人的代码（尤其写得比较烦琐的前提下）好累，不如自己的代码简炼

```
public int[] countSubgraphsForEachDiameter(int n, int[][] edges) {
    int m = n-1, range = 1 << m, root = 0, cnt = 0;
    int [] ans = new int [m];
    for (int i = 1; i < range; i++) {
        root = -1;
        Map<Integer, List<Integer>> adj = new HashMap<>();
        for (int j = 0; j < m; j++)  // m edges
            if (((i >> j) & 1) == 1) {
                int [] e = edges[j];
                if (root == -1) root = e[0];
                adj.computeIfAbsent(e[0], z -> new ArrayList<>()).add(e[1]);
                adj.computeIfAbsent(e[1], z -> new ArrayList<>()).add(e[0]);
            }
        cnt = Integer.bitCount(i);
        Set<Integer> vis = new HashSet<>();
        max = 1;
        dfs(root, -1, adj, vis);
        if (vis.size() != cnt + 1) continue;
        ans[max-1]++;
    }
    return ans;
}
int max = 1;
private int dfs(int u, int p, Map<Integer, List<Integer>> m, Set<Integer> vis) { // 树的最大直径:
    vis.add(u);
    if (m.get(u).size() == 1 && m.get(u).get(0) == p) return 1; // 叶子节点
    int fst = 0, sec = 0;
    for (Integer v : m.get(u)) {
        if (v == p) continue;
        int cur = dfs(v, u, m, vis);
```

```
        if (cur >= fst) {
            sec = fst;
            fst = cur;
        } else sec = Math.max(sec, cur);
    }
    max = Math.max(max, fst + sec); // bug: 这里 fst + sec 不需要 +1
    return fst + 1;
}
```

- 另一种位操作法

```
One way in which we can find the diameter of a tree is using DFS, just like if our tree is represented using tree nodes instead of as grpah
    1. Make a call to DFS from any node as root, lets say 1 as root
    2. Maintain a global max parameter
    3. For each call to dfs, of all current nodes children (excluding parent)
       find top two distances from current node to any leaf reachable from current node
    4. Sum of these top two distances froms the longes path passing through current node to all its children. Update if this path is maximum
    5. return 1 + top distance for this dfs call. Need to add 1 since,
       max length of path that can be reached from current ndoe is current ndoe + max distance reachable from current ndoes's children
```

```
public int [] countSubgraphsForEachDiameter(int n, int[][] edges) {
    ans = new int [n-1];
    for (int [] i : edges) { // if our node is 5, we store it as 1 << 4 which is 2^4
        graph.computeIfAbsent(1 << (i[0]-1), ArrayList::new).add(1 << (i[1]-1));
        graph.computeIfAbsent(1 << (i[1]-1), ArrayList::new).add(1 << (i[0]-1));
    }
    int range = (1 << n) - 1;  // (int)Math.pow(2, n) - 1;
    for (int subset = 3; subset <= range; subset++) {
        boolean isPowerOf2 = subset != 0 && (subset & (subset - 1)) == 0; // is power of 2
        if (isPowerOf2) continue;      // Single node subtrees can be excluded.
        max = 0; vis = 0;
        dfs(subset, Integer.highestOneBit(subset), -1); // Integer.highestOneBit(subset): subset: 0b1100, highest: 0b1000
        if (vis == subset)   // If visited is not equal to our current subset, all nodes are not reachable.
            ans[max - 1] ++; // In otherwords is not a proper subtree, hence dont include in the maxwer
    }
    return ans;
}
Map<Integer, List<Integer>> graph = new HashMap<>();
int max = 0, vis = 0;
int [] ans;
private int dfs(int subset, int cur, int pre) {
    if ((subset & cur) == 0) return 0; // 只遍历子集中存在的节点，换句话说，只遍历子集中存在的边，这样总图只建一遍就可以了
    vis = vis | cur;
    int fstMax = 0, sndMax = 0;
    for (Integer next : graph.get(cur)) {
        if (next == pre) continue;
        int dist = dfs(subset, next, cur);
        if (dist > fstMax) {
            sndMax = fstMax;
            fstMax = dist;
        } else sndMax = Math.max(sndMax, dist);
    }
    max = Math.max(max, fstMax + sndMax); // top two distances from this node c
// top distance this cur node to any leaf is topdistance from c's children + 1. Adding 1 since we need to include cur node
    return 1 + fstMax;
}
```

- 以前参考过的代码

```
public int [] countSubgraphsForEachDiameter(int n, int[][] edges) {
    int [] res = new int [n-1];
    List<List<int []>> subsets = new ArrayList<>();
    generateSubsets(edges, new ArrayList<int []>(), subsets, 0);
    for (List<int []> subset : subsets)
        solve(subset, res);
    return res;
}
private void solve(List<int []> subset, int [] res) {
    if (!isValidGraph(subset)) return;
    Map<Integer, List<Integer>> graph = new HashMap<>();
    for (int [] eg : subset) {
        graph.computeIfAbsent(eg[0], k -> new ArrayList<>()).add(eg[1]);
        graph.computeIfAbsent(eg[1], k -> new ArrayList<>()).add(eg[0]);
    }
    int max = 1;
    for (Integer key : graph.keySet()) {
        if (graph.get(key).size() == 1) {
            int [] longest = new int [] {1}; // 减少 global 变量的数量
            Set<Integer> vis = new HashSet<>();
```

```
                vis.add(key);
                dfs(graph, vis, key, longest, 0);
                max = Math.max(max, longest[0]);
            }
        }
        res[max - 1]++;
    }
    private void dfs(Map<Integer, List<Integer>> graph, Set<Integer> vis, int idx, int [] longest, int level) {
        longest[0] = Math.max(longest[0], level);
        for (Integer node : graph.get(idx))
            if (vis.add(node)) // Set.add(element) return false if it contains element already
                dfs(graph, vis, node, longest, level + 1);
    }
    private boolean isValidGraph(List<int []> subset) {
        Set<Integer> nodes = new HashSet<>();
        for (int [] cur : subset) {
            nodes.add(cur[0]);
            nodes.add(cur[1]);
        }
        return nodes.size() - 1 <= subset.size();
    }
    private void generateSubsets(int [][] arr, List<int []> cur, List<List<int []>> res, int idx) {
        if (idx == arr.length) return; // arr.length <= 15, 用回塑法直接生成 subsets, 但是这是相对耗时的操作
        for (int i = idx; i < arr.length; i++) {
            cur.add(arr[i]);
            res.add(new ArrayList<>(cur));
            generateSubsets(arr, cur, res, i+1);
            cur.remove(cur.size()-1);
        }
    }
```

2.1.8 1938. Maximum Genetic Difference Query - Hard 离线算法、离线思维、批量处理、顺序无关

There is a rooted tree consisting of n nodes numbered 0 to n - 1. Each node's number denotes its unique genetic value (i.e. the genetic value of node x is x). The genetic difference between two genetic values is defined as the bitwise-XOR of their values. You are given the integer array parents, where parents[i] is the parent for node i. If node x is the root of the tree, then parents[x] == -1.

You are also given the array queries where queries[i] = [nodei, vali]. For each query i, find the maximum genetic difference between vali and pi, where pi is the genetic value of any node that is on the path between nodei and the root (including nodei and the root). More formally, you want to maximize vali XOR pi.

Return an array ans where ans[i] is the answer to the ith query.

```
// 可以从根节点开始，对整棵树进行一次深度优先遍历，即：
// 当我们第一次遍历到某一节点 ii 时，我们将 ii 放入「数据结构」中；
// 当我们遍历完所有节点 ii 的子节点，即将回溯到 ii 的父节点前，我们将 ii 从「数据结构」中移除。
// 这样一来，我们就可以通过「离线」的思想将每一个询问在遍历到节点 \textit{val}_ival 时进行求解。这是因为，如果当前正在遍历节点 \textit{val}_ival
// 那么数据结构中就存放着所有从根节点到节点 \textit{val}_ival 的路径上的所有节点。
// 此时，我们只需要找出数据结构中使得 p_i \oplus \textit{val}_ip 达到最大值的节点 p_ip 即可。
// 而深度优先搜索过程中，当前入队的部分正是该节点及其所有层级的父节点，因此可结合 DFS 方法进行离线搜索。
// 对最大异或值的计算，可结合字典树方法进行。
// 本题需涉及对字典树中数值的删除操作，为简化代码，可在字典树的节点中设计一个计数器，记录当前该节点对应的数字个数，从而避免删除实际节点。
public class Trie {
    static final int H = 18; // 树高度，本题 val<=2*10^5<2^18
    Trie [] next;
    int cnt;                 // 当前节点对应的数值个数，简化删除操作
    public Trie() {
        this.next = new Trie[2];
        this.cnt = 0;
    }
    public void insert(int va) { // 插入数值
        Trie r = this;
        for (int i = H-1; i >= 0; i--) {
            int bit = (va >> i) & 1;
            if (r.next[bit] == null)
                r.next[bit] = new Trie();
            r = r.next[bit];
            r.cnt++;
        }
    }
    private void removeVal(int v) { // 删除数值
        Trie r = this;
        for (int i = H-1; i >= 0; i--) {
            int bit = (v >> i) & 1;
            r = r.next[bit];
            r.cnt--;
        }
```

```
    }
    public int search(int va) { // 针对数值查询当前字典树对应的最大异或值
        Trie r = this;
        int max = 0;
        for (int i = H-1; i >= 0; i--) {
            int bit = (va >> i) & 1 ^ 1;
            if (r == null) return -1;
            if (r.next[bit] != null && r.next[bit].cnt > 0) {
                max += (1 << i);
                r = r.next[bit];
            } else
                r = r.next[bit ^ 1];
        }
        return max;
    }
}
private void dfs(int idx) { // 深度优先搜索
    trie.insert(idx);       // 当前节点加入字典树
    if (queVal.containsKey(idx)) // 处理针对当前节点的查询
        for (int i = 0; i < queVal.get(idx).size(); i++)
            ans[queId.get(idx).get(i)] = trie.search(queVal.get(idx).get(i));
    if (tree.containsKey(idx))   // 当前节点存在子节点
        for (int n : tree.get(idx))
            dfs(n);
    trie.removeVal(idx);         // 从字典树中删除当前节点
}
Map<Integer, List<Integer>> tree;  // 树中各个节点对应的子节点
Map<Integer, List<Integer>> queVal;// 树中各个节点对应的查询值
Map<Integer, List<Integer>> queId; // 树中各个节点对应的 queries 下标
Trie trie;                         // 字典树根节点
int [] ans;
public int[] maxGeneticDifference(int[] parents, int[][] queries) {
    int n = parents.length, m = queries.length, root = -1;
    this.tree = new HashMap<>();
    for (int i = 0; i < n; i++) { // 记录树中各个节点对应的子节点
        if (parents[i] != -1) {   // Note: 当作有向树图来处理 !!!
            tree.computeIfAbsent(parents[i], k -> new ArrayList<>());
            tree.get(parents[i]).add(i);
        } else root = i;
    }
    this.queVal = new HashMap<>();
    this.queId = new HashMap<>();
    for (int i = 0; i < m; i++) {
        int nid = queries[i][0], val = queries[i][1];
        queVal.computeIfAbsent(nid, k -> new ArrayList<>()).add(val);
        queId.computeIfAbsent(nid, k -> new ArrayList<>()).add(i);
    }
    this.ans = new int [m];
    this.trie = new Trie();
    dfs(root);
    return ans;
}
```

复杂度分析

时间复杂度：O((n+q) log C)O((n+q)logC)，其中 qq 是数组 *queries*queries 的长度，log C = 18logC=18 是本题中最大的数的二进制表示的位数。在深度优先遍历的过程中，访问的节点个数为 nn，每个节点需要 O(log C)O(logC) 的时间在一开将其加入字典树以及回溯前将其从字典树中移除。对于数组 *queries*queries 中的每一个询问，我们需要 O(log C)O(logC) 的时间得到答案。因此总时间复杂度为 O((n+q) log C)O((n+q)logC)。

空间复杂度：O(nlog C + q)O(nlogC+q)。我们需要 O(n)O(n) 的空间存储树本身，O(n log C)O(nlogC) 的空间存储字典树，O(q)O(q) 的空间存储将询问进行离线，分配到每个节点上。

2.1.9 792. Number of Matching Subsequences - Medium

Given a string s and an array of strings words, return the number of words[i] that is a subsequence of s.

A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

For example, "ace" is a subsequence of "abcde".

```
// 我们需要使用每个字典中的单词去和 S 比较，看它是否是 S 的子序列。不过这种比较非常耗费时间，因此我们需要对 S 进行一下预处理。
// 首先定义一个二维数组 arr[][]，其中 arr[i][j] 代表距离 S 中第 i 位字符最近的 j 字符的位置。
// 换句话说，我们需要遍历一边字符串，记录下字符串 S 每一位上的字符，在它右侧距离它最近的 a-z 分别在哪。
public int numMatchingSubseq(String s, String[] words) {
    int n = s.length();
    int [][] arr = new int [n][26]; // 预处理用的数组
    for (int i = n-2; i >= 0; i--) {// 预处理
        arr[i] = Arrays.copyOf(arr[i+1], 26);
        arr[i][s.charAt(i+1)-'a'] = i+1;
```

```
    }
    int res = 0, idxAtS = 0, idx = 0, cur = 0;
    for (String v : words) {        // 比较每一个单词
        idxAtS = 0;                 // 对应 S 的下标
        idx = 0;                    // 当前单词下标
        if (v.charAt(0) == s.charAt(0)) { // 如果当前单词首字符等于 S 首字符
            idx ++;                 // 当前单词下标加一
            if (v.length() == 1) res++;      // 如果当前单词长度只有 1，说明当前单词已经遍历结束，结果加一
        }
        while (idx < v.length()) {            // 继续比较单词接下来的字符，在 S 中是否存在
            cur = v.charAt(idx) - 'a';
            if (arr[idxAtS][cur] == 0) break; // 如果 indexAtS 之后不存在 c，当前单词不合法
            idxAtS = arr[idxAtS][cur]; // 将 indexAtS 更新为 c 在 S 中的位置
            if (++idx == v.length()) res++;     // index 加一，如果 index 为单词最后一位，代表单词中所有字符均在 S 中找到
        }
    }
    return res;
}
```

2.1.10 472. Concatenated Words - Hard

Given an array of strings words (without duplicates), return all the concatenated words in the given list of words.

A concatenated word is defined as a string that is comprised entirely of at least two shorter words in the given array.

Example 1:

```
Input: words = ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratcatdogcat"]
Output: ["catsdogcats","dogcatsdog","ratcatdogcat"]
Explanation: "catsdogcats" can be concatenated by "cats", "dog" and "cats";
"dogcatsdog" can be concatenated by "dog", "cats" and "dog";
"ratcatdogcat" can be concatenated by "rat", "cat", "dog" and "cat".
```

- 切记：dfs 深搜 + 记忆

```
// 切记:  dfs 深搜 + 记忆 // Trie with memo, Time: o(m*2^n)
public class Trie {
    boolean isWord;
    Trie [] children;
    public Trie() {
        isWord = false;
        children = new Trie[26];
    }
}
public void insert(String word) {
    Trie cur = root;
    for (int i = 0; i < word.length(); i++) {
        char c = word.charAt(i);
        if (cur.children[c-'a'] == null)
            cur.children[c-'a'] = new Trie();
        cur = cur.children[c-'a'];
    }
    cur.isWord = true;
}
public boolean isConcatenated(String word, int idx, int cnt, HashMap<Integer, Boolean> memo) {
    if (memo.containsKey(idx)) return memo.get(idx);
    if (idx == word.length()) {
        memo.put(idx, cnt > 1);
        return cnt > 1;
    }
    Trie cur = root;
    for (int i = idx; i < word.length(); i++) {
        char c = word.charAt(i);
        if (cur.children[c-'a'] == null) {
            memo.put(idx, false);
            return false;
        } else {
            cur = cur.children[c-'a'];
            if (cur.isWord && isConcatenated(word, i+1, cnt+1, memo)) {
                memo.put(idx, true);
                return true;
            }
        }
    }
    memo.put(idx, false);
    return false;
}
Trie root = new Trie();
public List<String> findAllConcatenatedWordsInADict(String[] words) {
```

```
    for (String word : words)
        insert(word);
    List<String> res = new ArrayList<>();
    for (String word : words)
        if (isConcatenated(word, 0, 0, new HashMap<Integer, Boolean>()))
            res.add(word);
    return res;
}
```

- 一种稍微优化了一下的方法，逻辑就相对复杂一点儿，参考一下

```
public class Trie { // Trie with memo, Time: o(m*2^n)
    boolean isKey;
    Trie [] child;
    public Trie() {
        this.isKey = false;
        child = new Trie[26];
    }
    public void insert(String s) {
        int [] memo = new int [s.length()];
        Trie p = this;
        char [] sArr = s.toCharArray();
        boolean added = false;
        for (int i = 0; i < sArr.length; i++) {
            char c = sArr[i];
            if (p.child[c-'a'] == null)
                p.child[c-'a'] = new Trie();
            p = p.child[c-'a'];
            if (p.isKey && isConcatenated(s, i+1, 0, memo) && !added) {
                res.add(s);
                added = true;
            }
        }
        p.isKey = true;
    }     // 这么看来，我还没能透彻理解 dfs 深搜中的重复，什么时候应该拥有记忆?!!!
    public boolean isConcatenated(String s, int start, int cnt, int [] memo) {
        if (start == s.length() && cnt > 0) return true;
        if (memo[start] != 0) return memo[start] == 1;
        Trie p = this;
        char [] sArr = s.toCharArray();
        for (int i = start; i < sArr.length; i++) {
            char c = sArr[i];
            Trie cur = p.child[c-'a'];
            if (cur == null) {
                memo[start] = -1;
                return false;
            } else {
                if (cur.isKey && isConcatenated(s, i+1, cnt+1, memo)) {
                    memo[start] = 1;
                    return true;
                }
                p = cur;
            }
        }
        memo[start] = -1;
        return false;
    }
}
// Sort the words based on length
// Use trie to store words: while adding, checking if it is concatenated
// While checking, use dfs + memo
List<String> res = new ArrayList<>();
public List<String> findAllConcatenatedWordsInADict(String[] words) {
    Arrays.sort(words, (x, y) -> Integer.compare(x.length(), y.length()));
    Trie tree = new Trie();
    for (String word : words)
        tree.insert(word);
    return res;
}
```

2.1.11 1948. Delete Duplicate Folders in System - Hard

Due to a bug, there are many duplicate folders in a file system. You are given a 2D array paths, where paths[i] is an array representing an absolute path to the ith folder in the file system.

For example, ["one", "two", "three"] represents the path "/one/two/three". Two folders (not necessarily on the same level) are identical if they contain the same non-empty set of identical subfolders and underlying subfolder structure. The folders do not need to be at the root level to be identical. If two or more folders are identical, then mark the folders as well as all their subfolders.

For example, folders "/a" and "/b" in the file structure below are identical. They (as well as their subfolders) should all be marked:

```
/a
/a/x
/a/x/y
/a/z
/b
/b/x
/b/x/y
/b/z
```

However, if the file structure also included the path "/b/w", then the folders "/a" and "/b" would not be identical. Note that "/a/x" and "/b/x" would still be considered identical even with the added folder.

Once all the identical folders and their subfolders have been marked, the file system will delete all of them. The file system only runs the deletion once, so any folders that become identical after the initial deletion are not deleted.

Return the 2D array ans containing the paths of the remaining folders after deleting all the marked folders. The paths may be returned in any order.

```
public class Node {
    String name;
    Map<String, Node> children = new HashMap<>();
    private String hashCode = null;
    public Node (String name) {
        this.name = name;
    }
    public void add(List<String> path) {
        Node cur = this;
        for (String file : path) {
            if (!cur.children.containsKey(file))
                cur.children.put(file, new Node(file));
            cur = cur.children.get(file);
        }
    }
    public String getHashCode() {
        if (hashCode == null)
            hashCode = compueteHash();
        return hashCode;
    }
    private String compueteHash() {
        StringBuilder sb = new StringBuilder();
        List<Node> nodes = new ArrayList<>();
        for (Node n : children.values())
            nodes.add(n);
        if (nodes.size() == 0) return null;
        nodes.sort((a, b) -> a.name.compareTo(b.name));
        for (Node n : nodes) {
            sb.append('(');
            sb.append(n.name + n.getHashCode());
            sb.append(')');
        }
        return sb.toString();
    }
}
private void getGoodFiles(Node node, Map<String, Integer> occurs, List<String> cur, List<List<String>> ans) {
    if (occurs.containsKey(node.getHashCode()) && occurs.get(node.getHashCode()) > 1) return;
    cur.add(node.name);
    ans.add(new ArrayList<>(cur));
    for (Node n : node.children.values())
        getGoodFiles(n, occurs, cur, ans);
    cur.remove(cur.size()-1);
}
private void findOccurs(Node node, Map<String, Integer> occurs) {
    String key = node.getHashCode();
    if (key != null)
        occurs.put(key, occurs.getOrDefault(node.getHashCode(), 0) + 1);
    for (Node n : node.children.values())
        findOccurs(n, occurs);
}
Node root;
public List<List<String>> deleteDuplicateFolder(List<List<String>> paths) {
    root = new Node("");
    for (List<String> path : paths)
        root.add(path);
    Map<String, Integer> occurs = new HashMap<>();
    findOccurs(root, occurs);
    List<List<String>> ans = new ArrayList<>();
    for (Node n : root.children.values())
        getGoodFiles(n, occurs, new ArrayList<>(), ans);
```

```
    return ans;
}
```

Chapter 3

Tree 树结构：各种新型数据结构

3.0.1 687. Longest Univalue Path

Given the root of a binary tree, return the length of the longest path, where each node in the path has the same value. This path may or may not pass through the root.

The length of the path between two nodes is represented by the number of edges between them.

- 此题与求二叉树的最长路径边长相似，只是此题要求是节点值相同的路径，也就是说在找最长路径的时候，还需要判断节点值，要是不相同，就重置为 0，在此期间，我们使用一个全局变量来存储最长节点值相同路径的边长。

```
private int topDownTraverse(TreeNode r) {
    if (r == null) return 0;
    int left = topDownTraverse(r.left);
    int right = topDownTraverse(r.right);
    if (r.left == null || r.left.val != r.val) left = 0;
    if (r.right == null || r.right.val != r.val) right = 0;
    max = Math.max(max, left + right);
    return Math.max(left, right) + 1;
}
int max = 0;
public int longestUnivaluePath(TreeNode root) {
    if (root == null) return 0;
    topDownTraverse(root);
    return max;
}
```

3.0.2 652. Find Duplicate Subtrees

Given the root of a binary tree, return all duplicate subtrees.

For each kind of duplicate subtrees, you only need to return the root node of any one of them.

Two trees are duplicate if they have the same structure with the same node values.

```
private String duplicate(TreeNode node) {
    if(node == null) return "X";
    String l = duplicate(node.left);
    String r = duplicate(node.right);
    String s = Integer.toString(node.val) + "-" + l + "-" + r;
    map.put(s, map.getOrDefault(s, 0)+1);
    if (map.get(s) == 2)
        list.add(node);
    return s;
}
HashMap<String,Integer> map = new HashMap<>();
ArrayList list = new ArrayList<>();
public List findDuplicateSubtrees(TreeNode root) {
    duplicate(root);
    return list;
}
```

- 看一下构造的图的效果图

```
      1 -> root
    2, 3,  ->
4, #| 2, 4,  ->
```

```
#.#| 4, #| #.#|  ->
#.#|  ->

map.size(): 4
3-2-4-X-X-X-4-X-X, 1
1-2-4-X-X-X-3-2-4-X-X-X-4-X-X, 1
2-4-X-X-X, 2
4-X-X, 3

res.size(): 2
TREE Level order traversal:
      4 -> root
    #.#|  ->

TREE Level order traversal:
      2 -> root
    4, #|  ->
#.#|  ->
```

- 一种 dfs 的写法

```java
HashSet<String> set, added;
List<TreeNode> list;
public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
    set = new HashSet();
    added = new HashSet();
    list = new ArrayList();
    StringBuilder ret = dfs(root);
    return list;
}
private StringBuilder dfs(TreeNode root){
    if (root == null) return null;
    StringBuilder sbL = dfs(root.left), sbR = dfs(root.right);
    if (sbL == null && sbR == null){
        sbL = new StringBuilder();
        sbL.append(root.val);
    } else if (sbL != null){
        sbL.append(" " + root.val);
        if (sbR != null){
            sbL.append(' ');
            sbL.append(sbR);
        } else sbL.append(" n");
    } else if (sbL == null){
        if (sbR != null){
            sbR.insert(0, " n " + root.val);
            sbL = sbR;
        }
    }
    String temp = sbL.toString();
    if (set.contains(temp) && !added.contains(temp)){
        list.add(root);
        added.add(temp);

    }
    set.add(temp);
    return sbL;
}
```

- 这个跑起来很高效，可惜我看不懂。。。。。以后再慢慢消化吧
- https://leetcode.com/problems/find-duplicate-subtrees/discuss/1418487/Java-beats-99.5-in-time

---

```java
Map<Integer, Integer> count;           // frequency of each subtree represented in string
Map<List<Integer>, Integer> numberMap; // ** not hashset since it cannot reserve element order
List<TreeNode> ans;
int globalNumber = 1;
public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
    count = new HashMap();
    numberMap = new HashMap();
    ans = new ArrayList();
    collect(root);
    return ans;
}
public int collect(TreeNode node) {
    if (node == null) return 0;
    int leftNumber = collect(node.left);
    int rightNumber = collect(node.right);
    List<Integer> numberExp = new ArrayList<>(); // construct expression
    numberExp.add(node.val);
    numberExp.add(leftNumber);
```

```
    numberExp.add(rightNumber);
    if (!numberMap.containsKey(numberExp)) { // update numberMap
        numberMap.put(numberExp, globalNumber);
        globalNumber++;
    }
    // check number frequency. if == 2, meaning duplication then add to result
    int rootNumber = numberMap.get(numberExp).intValue();
    count.put(rootNumber, count.getOrDefault(rootNumber, 0)+1);
    if (count.get(rootNumber) == 2) // not >=2, otherwise ans will have duplicated nodes
        ans.add(node);
    return rootNumber;
}
```

```
count.size(): 4
1, 3
2, 2
3, 1
4, 1
numberMap.size(): 4
2, 1, 0,
2
3, 2, 1,
3
1, 2, 3,
4
4, 0, 0,
1
```

3.0.3 979. Distribute Coins in Binary Tree

You are given the root of a binary tree with n nodes where each node in the tree has node.val coins. There are n coins in total throughout the whole tree.

In one move, we may choose two adjacent nodes and move one coin from one node to another. A move may be from parent to child, or from child to parent.

Return the minimum number of moves required to make every node have exactly one coin.

```
private int dfs(TreeNode r) { // 统计把自身，左右子树都平衡，需要移动的 coins 个数
    if (r == null) return 0;
    int left = dfs(r.left);      // 左、右子树缺多少
    int right = dfs(r.right);
    res += Math.abs(left) + Math.abs(right); // 左，右子树和自身都平衡需要的移动数
    return left + right + r.val-1;
}
int res;
public int distributeCoins(TreeNode root) {
    res = 0;
    return res;
}
```

3.0.4 1696. Jump Game VI

You are given a 0-indexed integer array nums and an integer k. You are initially standing at index 0. In one move, you can jump at most k steps forward without going outside the boundaries of the array. That is, you can jump from index i to any index in the range [i + 1, min(n - 1, i + k)] inclusive. You want to reach the last index of the array (index n - 1). Your score is the sum of all nums[j] for each index j you visited in the array. Return the maximum score you can get.

```
public int maxResult(int[] nums, int k) { // O(N) DP with double ended queue
    int n = nums.length;
    int [] dp = new int[n];
    ArrayDeque<Integer> q = new ArrayDeque<>();
    for (int i = 0; i < n; i++) {
        while (!q.isEmpty() && q.peekFirst() < i-k) // 头大尾小
            q.removeFirst();
        dp[i] = nums[i] + (q.isEmpty() ? 0 : dp[q.peekFirst()]);
        while (q.size() > 0 && dp[q.peekLast()] <= dp[i])
            q.removeLast();
        q.addLast(i);
    }
    return dp[n-1];
}
public int maxResult(int[] nums, int k) { // BigO: O (NlogN)
    int n = nums.length;
```

```
    int [] dp = new int[n];
    Queue<int []> q = new PriorityQueue<>(Comparator.comparingInt(e -> -e[0]));
    for (int i = 0; i < n; i++) {
        while (!q.isEmpty() && q.peek()[1] + k < i)
            q.poll();
        dp[i] = nums[i] + (q.isEmpty() ? 0 : q.peek()[0]);
        q.add(new int[] {dp[i], i});
    }
    return dp[n-1];
}
```

3.0.5 1932. Merge BSTs to Create Single BST

You are given n BST (binary search tree) root nodes for n separate BSTs stored in an array trees (0-indexed). Each BST in trees has at most 3 nodes, and no two roots have the same value. In one operation, you can:

Select two distinct indices i and j such that the value stored at one of the leaves of trees[i] is equal to the root value of trees[j]. Replace the leaf node in trees[i] with trees[j]. Remove trees[j] from trees. Return the root of the resulting BST if it is possible to form a valid BST after performing n - 1 operations, or null if it is impossible to create a valid BST.

A BST (binary search tree) is a binary tree where each node satisfies the following property:

Every node in the node's left subtree has a value strictly less than the node's value. Every node in the node's right subtree has a value strictly greater than the node's value. A leaf is a node that has no children.

```
public TreeNode canMerge(List<TreeNode> trees) {
    final int size = trees.size();
    final Map<Integer, TreeNode> roots = new HashMap<>(size);
    for (final TreeNode node : trees)
        roots.put(node.val, node);
    for (final TreeNode node : trees) {
        if (roots.containsKey(node.val)) { // 这里判断：是因为接下来 buildTree 会将可以合并的子树键值对删除并回收利用建大树了
            final TreeNode root = buildTree(roots, node);
            roots.put(root.val, root);    // update root node
        }
    }
    if (roots.size() != 1) return null;   // 无法合并所有的子树
    final TreeNode root = roots.values().iterator().next(); // 只有这一颗树根
    return isValid(root, Integer.MIN_VALUE, Integer.MAX_VALUE) ? root : null;
}
private TreeNode buildTree(Map<Integer, TreeNode> roots, TreeNode node) { // 用 recursion 把所有需要/可以合并的子树建成一棵完整大树，方法很传神
    final TreeNode next = roots.remove(node.val); // map.remove() 返回值：如果存在 key，则删除并返回 value；如果不存在则返回 null
    if (next != null) {
        if (next.left != null) node.left = buildTree(roots, next.left);
        if (next.right != null) node.right = buildTree(roots, next.right);
    }
    return node;
}
private boolean isValid(TreeNode node, int min, int max) { // 这些个递归写得很传功力，要活学活用到出神入化。。。。。
    if (node == null) return true;
    final int value = node.val;
    if (value <= min || value >= max) return false;
    return isValid(node.left, min, value) && isValid(node.right, value, max);
}
```

3.0.6 968. Binary Tree Cameras

You are given the root of a binary tree. We install cameras on the tree nodes where each camera at a node can monitor its parent, itself, and its immediate children. Return the minimum number of cameras needed to monitor all nodes of the tree.

```
// 对于每个节点，有一下三种 case：
// case（1）：如果它有一个孩子，且这个孩子是叶子（状态 0），则它需要摄像头，res ++，然后返回 1，表示已经给它装上了摄像头。
// case（2）：如果它有一个孩子，且这个孩子是叶子的父节点（状态 1），那么它已经被覆盖，返回 2。
// case（0）：否则，这个节点无孩子，或者说，孩子都是状态 2，那么我们将这个节点视为叶子来处理。
// 由于 dfs 最终返回后，整棵树的根节点的状态还未处理，因此需要判断，若根节点被视为叶子，需要在其上加一个摄像头。
private int dfs(TreeNode r) {
    // 空节点不需要被覆盖，归入情况 2
    if (r == null) return 2; // do not need cover
    int left = dfs(r.left);  // 递归求左右孩子的状态
    int right = dfs(r.right);
    // 获取左右孩子状态之后的处理
    // 有叶子孩子，加摄像头，归入情况 1
    if (left == 0 || right == 0) {
        res ++;
        return 1;
    }
```

```
    // 孩子上有摄像头，说明此节点已被覆盖，情况 2;
    if (left == 1 || right == 1) return 2;
    return 0;
}
int res = 0;
public int minCameraCover(TreeNode root) {
    // 若根节点被视为叶子，需要在其上加一个摄像头
    return (dfs(root) == 0 ? 1 : 0) + res;
}
```

3.0.7 Create Sorted Array through Instructions

Given an integer array instructions, you are asked to create a sorted array from the elements in instructions. You start with an empty container nums. For each element from left to right in instructions, insert it into nums. The cost of each insertion is the minimum of the following: The number of elements currently in nums that are strictly less than instructions[i]. The number of elements currently in nums that are strictly greater than instructions[i]. For example, if inserting element 3 into nums = [1,2,3,5], the cost of insertion is min(2, 1) (elements 1 and 2 are less than 3, element 5 is greater than 3) and nums will become [1,2,3,3,5]. Return the total cost to insert all elements from instructions into nums. Since the answer may be large, return it modulo 109 + 7

```
// https://blog.csdn.net/qq_28033719/article/details/112506925
private static int N = 100001;
private static int [] tree = new int [N]; // 拿元素值作为 key 对应 tree 的下标值
public int lowbit(int i) {
    return i & -i;
}
public void update(int i, int v) { // 更新父节点
    while (i <= N) {
        tree[i] += v;
        i += lowbit(i);
    }
}
public int getSum(int i) { // 得到以 i 为下标 1-based 的所有子、叶子节点的和，也就是 [1, i] 的和，1-based
    int ans = 0;
    while (i > 0) {
        ans += tree[i];
        i -= lowbit(i);
    }
    return ans;
}
public int createSortedArray(int[] instructions) {
    int n = instructions.length;
    long res = 0;
    Arrays.fill(tree, 0);
    for (int i = 0; i < n; i++) {
        //              严格小于此数的个数 严格大于此数的个数：为总个数（不含自己） - 小于自己的个数
        res += Math.min(getSum(instructions[i]-1), i-getSum(instructions[i]));
        update(instructions[i], 1);
    }
    return (int)(res % ((int)Math.pow(10, 9) + 7));
}
```

Chapter 4

Tree

4.1 979. Distribute Coins in Binary Tree

You are given the root of a binary tree with n nodes where each node in the tree has node.val coins. There are n coins in total throughout the whole tree.

In one move, we may choose two adjacent nodes and move one coin from one node to another. A move may be from parent to child, or from child to parent.

Return the minimum number of moves required to make every node have exactly one coin.

```
private int dfs(TreeNode r) { // 统计把自身，左右子树都平衡，需要移动的 coins 个数
    if (r == null) return 0;
    int left = dfs(r.left);      // 左、右子树缺多少
    int right = dfs(r.right);
    res += Math.abs(left) + Math.abs(right); // 左，右子树和自身都平衡需要的移动数
    return left + right + r.val-1;
}
int res;
public int distributeCoins(TreeNode root) {
    res = 0;
    return res;
}
```

4.2 687. Longest Univalue Path

Given the root of a binary tree, return the length of the longest path, where each node in the path has the same value. This path may or may not pass through the root.

The length of the path between two nodes is represented by the number of edges between them.

- 此题与求二叉树的最长路径边长相似，只是此题要求是节点值相同的路径，也就是说在找最长路径的时候，还需要判断节点值，要是不相同，就重置为 0，在此期间，我们使用一个全局变量来存储最长节点值相同路径的边长。

```
private int topDownTraverse(TreeNode r) {
    if (r == null) return 0;
    int left = topDownTraverse(r.left);
    int right = topDownTraverse(r.right);
    if (r.left == null || r.left.val != r.val) left = 0;
    if (r.right == null || r.right.val != r.val) right = 0;
    max = Math.max(max, left + right);
    return Math.max(left, right) + 1;
}
int max = 0;
public int longestUnivaluePath(TreeNode root) {
    if (root == null) return 0;
    topDownTraverse(root);
    return max;
}
```

4.3 652. Find Duplicate Subtrees

Given the root of a binary tree, return all duplicate subtrees.

For each kind of duplicate subtrees, you only need to return the root node of any one of them.
Two trees are duplicate if they have the same structure with the same node values.

```
private String duplicate(TreeNode node) {
    if(node == null) return "X";
    String l = duplicate(node.left);
    String r = duplicate(node.right);
    String s = Integer.toString(node.val) + "-" + l + "-" + r;
    map.put(s, map.getOrDefault(s, 0)+1);
    if (map.get(s) == 2)
        list.add(node);
    return s;
}
HashMap<String,Integer> map = new HashMap<>();
ArrayList list = new ArrayList<>();
public List findDuplicateSubtrees(TreeNode root) {
    duplicate(root);
    return list;
}
```

- 看一下构造的图的效果图

```
      1 -> root
    2, 3,  ->
4, #| 2, 4,  ->
#.#| 4, #| #.#|  ->
#.#|  ->

map.size(): 4
3-2-4-X-X-X-4-X-X, 1
1-2-4-X-X-X-3-2-4-X-X-X-4-X-X, 1
2-4-X-X-X, 2
4-X-X, 3

res.size(): 2
TREE Level order traversal:
      4 -> root
    #.#|  ->

TREE Level order traversal:
      2 -> root
    4, #|  ->
#.#|  ->
```

- 一种 dfs 的写法

```
HashSet<String> set, added;
List<TreeNode> list;
public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
    set = new HashSet();
    added = new HashSet();
    list = new ArrayList();
    StringBuilder ret = dfs(root);
    return list;
}
private StringBuilder dfs(TreeNode root){
    if (root == null) return null;
    StringBuilder sbL = dfs(root.left), sbR = dfs(root.right);
    if (sbL == null && sbR == null){
        sbL = new StringBuilder();
        sbL.append(root.val);
    } else if (sbL != null){
        sbL.append(" " + root.val);
        if (sbR != null){
            sbL.append(' ');
            sbL.append(sbR);
        } else sbL.append(" n");
    } else if (sbL == null){
        if (sbR != null){
            sbR.insert(0, " n " + root.val);
            sbL = sbR;
        }
    }
    String temp = sbL.toString();
    if (set.contains(temp) && !added.contains(temp)){
        list.add(root);
        added.add(temp);

    }
```

```
    set.add(temp);
    return sbL;
}
```

- 这个跑起来很高效，可惜我看不懂。。。。。以后再慢慢消化吧
- https://leetcode.com/problems/find-duplicate-subtrees/discuss/1418487/Java-beats-99.5-in-time

```
Map<Integer, Integer> count;           // frequency of each subtree represented in string
Map<List<Integer>, Integer> numberMap; // ** not hashset since it cannot reserve element order
List<TreeNode> ans;
int globalNumber = 1;
public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
    count = new HashMap();
    numberMap = new HashMap();
    ans = new ArrayList();
    collect(root);
    return ans;
}
public int collect(TreeNode node) {
    if (node == null) return 0;
    int leftNumber = collect(node.left);
    int rightNumber = collect(node.right);
    List<Integer> numberExp = new ArrayList<>(); // construct expression
    numberExp.add(node.val);
    numberExp.add(leftNumber);
    numberExp.add(rightNumber);
    if (!numberMap.containsKey(numberExp)) { // update numberMap
        numberMap.put(numberExp, globalNumber);
        globalNumber++;
    }
    // check number frequency. if == 2, meaning duplication then add to result
    int rootNumber = numberMap.get(numberExp).intValue();
    count.put(rootNumber, count.getOrDefault(rootNumber, 0)+1);
    if (count.get(rootNumber) == 2) // not >=2, otherwise ans will have duplicated nodes
        ans.add(node);
    return rootNumber;
}
```

```
count.size(): 4
1, 3
2, 2
3, 1
4, 1
numberMap.size(): 4
2, 1, 0,
2
3, 2, 1,
3
1, 2, 3,
4
4, 0, 0,
1
```