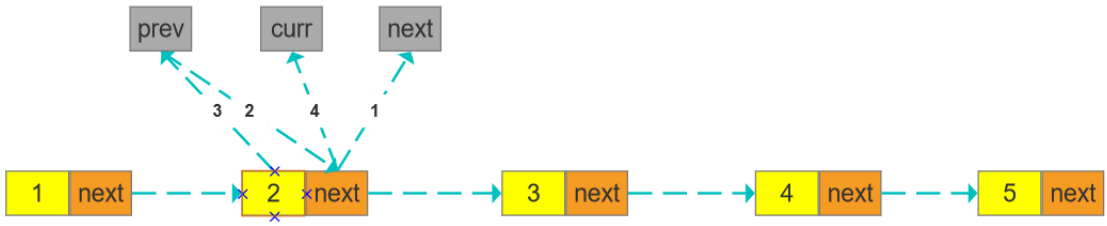
反转链表

反转一个单链表。

```
输入: 1->2->3->4->5
输出: 5->4->3->2->1
```

解法1：迭代，重复某一过程，每一次处理结果作为下一次处理的初始值，这些初始值类似于状态、每次处理都会改变状态、直至到达最终状态

从前往后遍历链表，将当前节点的next指向上一个节点，因此需要一个变量存储上一个节点prev，当前节点处理完需要寻找下一个节点，因此需要一个变量保存当前节点curr，处理完后要将当前节点赋值给prev，并将next指针赋值给curr，因此需要一个变量提前保存下一个节点的指针next



1、将下一个节点指针保存到next变量 next = curr.next

2、将下一个节点的指针指向prev，curr.next = prev

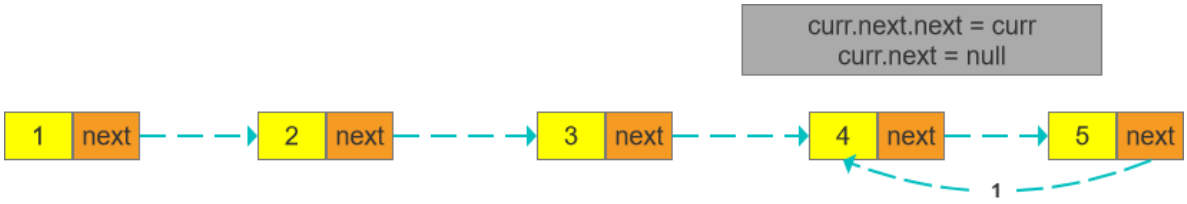
3、准备处理下一个节点，将curr赋值给prev

4、将下一个节点赋值为curr，处理一个节点

解法2：递归：以相似的方法重复，类似于树结构，先从根节点找到叶子节点，从叶子节点开始遍历

大的问题(整个链表反转)拆成性质相同的小问题(两个元素反转)curr.next.next = curr

将所有的小问题解决，大问题即解决



只需每个元素都执行curr.next.next = curr，curr.next = null两个步骤即可

为了保证链不断，必须从最后一个元素开始

```
public class ReverseList {
    static class ListNode{
        int val;
```

```
        ListNode next;

        public ListNode(int val, ListNode next) {
            this.val = val;
            this.next = next;
        }
    }

    public static ListNode iterate(ListNode head){
        ListNode prev = null,curr,next;
        curr = head;
        while(curr != null){
            next = curr.next;
            curr.next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }

    public static ListNode recursion(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode newHead = recursion(head.next);
        head.next.next = head;
        head.next = null;
        return newHead;
    }


    public static void main(String[] args) {
        ListNode node5 = new ListNode(5,null);
        ListNode node4 = new ListNode(4,node5);
        ListNode node3 = new ListNode(3,node4);
        ListNode node2 = new ListNode(2,node3);
        ListNode node1 = new ListNode(1,node2);
        //ListNode node = iterate(node1);
        ListNode node_1 = recursion(node1);
        System.out.println(node_1);
    }
}
```

统计N以内的素数

素数：只能被1和自身整除的数，0、1除外

解法一：暴力算法

直接从2开始遍历，判断是否能被2到自身之间的数整除

```java
public int countPrimes(int n) {
    int ans = 0;
    for (int i = 2; i < n; ++i) {
        ans += isPrime(i) ? 1 : 0;
    }
    return ans;
}
//i如果能被x整除，则x/i肯定能被x整除，因此只需判断i和根号x之中较小的即可
public boolean isPrime(int x) {
    for (int i = 2; i * i <= x; ++i) {
        if (x % i == 0) {
            return false;
        }
    }
    return true;
}
```

解法2：埃氏筛

利用合数的概念(非素数)，素数*n必然是合数，因此可以从2开始遍历，将所有的合数做上标记

```java
public static int eratosthenes(int n) {
    boolean[] isPrime = new boolean[n];
    int ans = 0;
    for (int i = 2; i < n; i++) {
        if (!isPrime[i]) {
            ans += 1;
            for (int j = i * i; j < n; j += i) {
                isPrime[j] = true;
            }
        }
    }
    return ans;
}
```

将合数标记为true，j = i * i 从 2 * i 优化而来，系数2会随着遍历递增（j += i，相当于递增了系数2），每一个合数都会有两个比本身要小的因子(0,1除外)，2 * i 必然会遍历到这两个因子

当2递增到大于根号n时，其实后面的已经无需再判断（或者只需判断后面一段），而2到根号n、实际上在 i 递增的过程中已经计算过了，i 实际上就相当于根号n

例如：n = 25 会计算以下

2 * 4 = 8

3 * 4 = 12

但实际上8和12已经标记过，在n = 17时已经计算了 3 * 4，2 * 4

寻找数组的中心索引

数组中某一个下标，左右两边的元素之后相等，该下标即为中心索引

思路：先统计出整个数组的总和，然后从第一个元素开始叠加

总和递减当前元素，叠加递增当前元素，知道两个值相等

```
public static int pivotIndex(int[] nums) {
    int sum1 = Arrays.stream(nums).sum();
    int sum2 = 0;
    for(int i = 0; i<nums.length; i++){
        sum2 += nums[i];
        if(sum1 == sum2){
            return i;
        }
        sum1 = sum1 - nums[i];
    }
    return -1;
}
```

删除排序数组中的重复项

一个有序数组 nums ，原地删除重复出现的元素，使每个元素只出现一次 ，返回删除后数组的新长度。

不要使用额外的数组空间，必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。

双指针算法：

数组完成排序后，我们可以放置两个指针 i 和 j，其中 i 是慢指针，而 j 是快指针。只要 nums[i]=nums[j]，我们就增加 j 以跳过重复项。

当遇到 nums[j] ！ = nums[i]时，跳过重复项的运行已经结束，必须把nums[j]）的值复制到 nums[i + 1]。然后递增 i，接着将再次重复相同的过程，直到 j 到达数组的末尾为止。

```java
public int removeDuplicates(int[] nums) {
    if (nums.length == 0) return 0;
    int i = 0;
    for (int j = 1; j < nums.length; j++) {
        if (nums[j] != nums[i]) {
            i++;
            nums[i] = nums[j];
        }
    }
    return i + 1;
}
```

x的平方根

在不使用 sqrt(x) 函数的情况下，得到 x的平方根的整数部分

解法一：二分查找

x的平方根肯定在0到x之间，使用二分查找定位该数字，该数字的平方一定是最接近x的，m平方值如果大于x、则往左边找，如果小于等于x则往右边找

找到0和X的最中间的数m，

如果m * m > x，则m取x/2到x的中间数字，直到m * m < x，m则为平方根的整数部分

如果m * m <= x，则取0到x/2的中间值，知道两边的界限重合，找到最大的整数，则为x平方根的整数部分

时间复杂度：O(logN)

```java
public static int binarySearch(int x) {
    int l = 0, r = x, index = -1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if ((long) mid * mid <= x) {
            index = mid;
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    return index;
}
```

解法二：牛顿迭代

假设平方根是 i ，则 i 和 x/i 必然都是x的因子，而 x/i 必然等于 i ，推导出 i + x / i = 2 * i，得出 i = (i + x / i) / 2

由此得出解法，i 可以任选一个值，只要上述公式成立，i 必然就是x的平方根，如果不成立，(i + x / i) / 2得出的值进行递归，直至得出正确解

```
public static int newton(int x) {
    if(x==0) return 0;
    return ((int)(sqrts(x,x)));
}

public static double sqrts(double i,int x){
    double res = (i + x / i) / 2;
    if (res == i) {
        return i;
    } else {
        return sqrts(res,x);
    }
}
```

三个数的最大乘积

一个整型数组 `nums` ，在数组中找出由三个数字组成的最大乘积，并输出这个乘积。

乘积不会越界

如果数组中全是非负数，则排序后最大的三个数相乘即为最大乘积；如果全是非正数，则最大的三个数相乘同样也为最大乘积。

如果数组中有正数有负数，则最大乘积既可能是三个最大正数的乘积，也可能是两个最小负数（即绝对值最大）与最大正数的乘积。

分别求出三个最大正数的乘积，以及两个最小负数与最大正数的乘积，二者之间的最大值即为所求答案。

解法一：排序

```
public static int sort(int[] nums) {
    Arrays.sort(nums);
    int n = nums.length;
    return Math.max(nums[0] * nums[1] * nums[n - 1], nums[n - 3] * nums[n - 2] *
nums[n - 1]);
}
```

解法二：线性扫描

```
public static int getMaxMin(int[] nums) {
    // 最小的和第二小的
    int min1 = 0, min2 = 0;
    // 最大的、第二大的和第三大的
    int max1 = 0, max2 = 0, max3 = 0;
```

```
    for (int x : nums) {
        if (x < min1) {
            min2 = min1;
            min1 = x;
        } else if (x < min2) {
            min2 = x;
        }

        if (x > max1) {
            max3 = max2;
            max2 = max1;
            max1 = x;
        } else if (x > max2) {
            max3 = max2;
            max2 = x;
        } else if (x > max3) {
            max3 = x;
        }
    }

    return Math.max(min1 * min2 * max1, max1 * max2 * max3);
}
```

两数之和

给定一个升序排列的整数数组 numbers ，从数组中找出两个数满足相加之和等于目标数 target 。

假设每个输入只对应唯一的答案，而且不可以重复使用相同的元素。

返回两数的下标值，以数组形式返回

暴力解法

```
public int[] twoSum(int[] nums, int target) {
    int n = nums.length;
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            if (nums[i] + nums[j] == target) {
                return new int[]{i, j};
            }
        }
    }
    return new int[0];
}
```

时间复杂度：O(N的平方)

空间复杂度：O(1)

哈希表：将数组的值作为key存入map，target - num作为key

```
public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<Integer, Integer>();
    for (int i = 0; i < nums.length; ++i) {
        if (map.containsKey(target - nums[i])) {
            return new int[]{map.get(target - nums[i]), i};
        }
        map.put(nums[i], i);
    }
    return new int[0];
}
```

时间复杂度：O(N)

空间复杂度：O(N)

解法一：二分查找

先固定一个值(从下标0开始)，再用二分查找查另外一个值，找不到则固定值向右移动，继续二分查找

```
public int[] twoSearch(int[] numbers, int target) {
    for (int i = 0; i < numbers.length; ++i) {
        int low = i, high = numbers.length -1;
        while (low <= high) {
            int mid = (high - low) / 2 + low;
            if (numbers[mid] == target - numbers[i]) {
                return new int[]{i, mid};
            } else if (numbers[mid] > target - numbers[i]) {
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }
    }
}
```

时间复杂度：O(N * logN)

空间复杂度：O(1)

解法二：双指针

左指针指向数组head，右指针指向数组tail，head+tail > target 则tail 左移，否则head右移

```
public int[] twoPoint(int[] numbers, int target) {
    int low = 0, high = numbers.length - 1;
    while (low < high) {
        int sum = numbers[low] + numbers[high];
        if (sum == target) {
            return new int[]{low + 1, high + 1};
        } else if (sum < target) {
            ++low;
```

```
            } else {
                --high;
            }
        }
        return new int[]{-1, -1};
    }
```

时间复杂度：O(N)

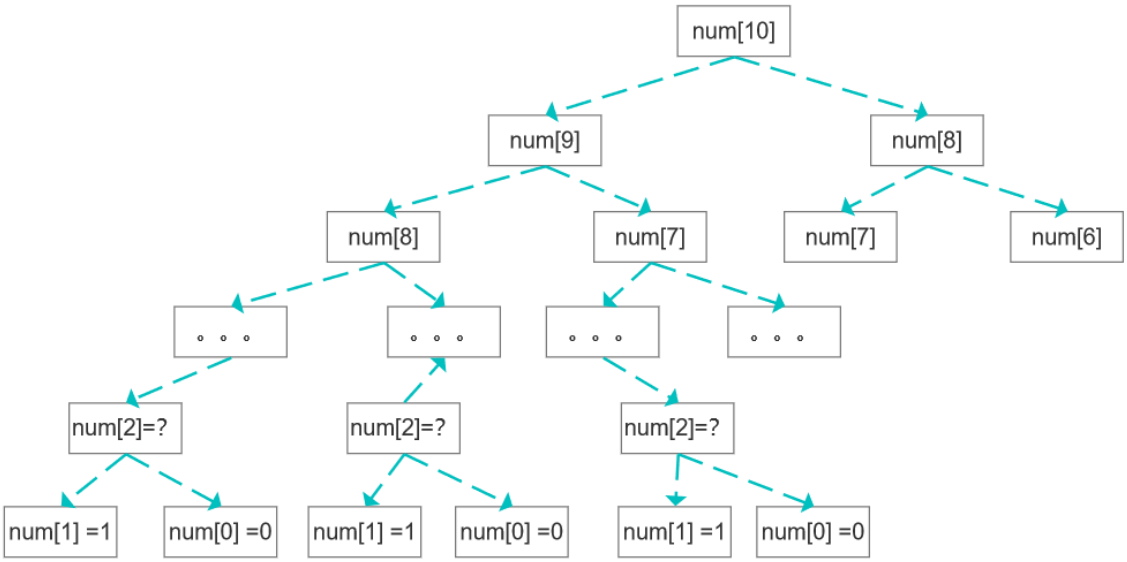
空间复杂度：O(1)

斐波那契数列

求取斐波那契数列第N位的值。

斐波那契数列：每一位的值等于他前两位数字之和。前两位固定 0，1,1,2,3,5,8。。。。

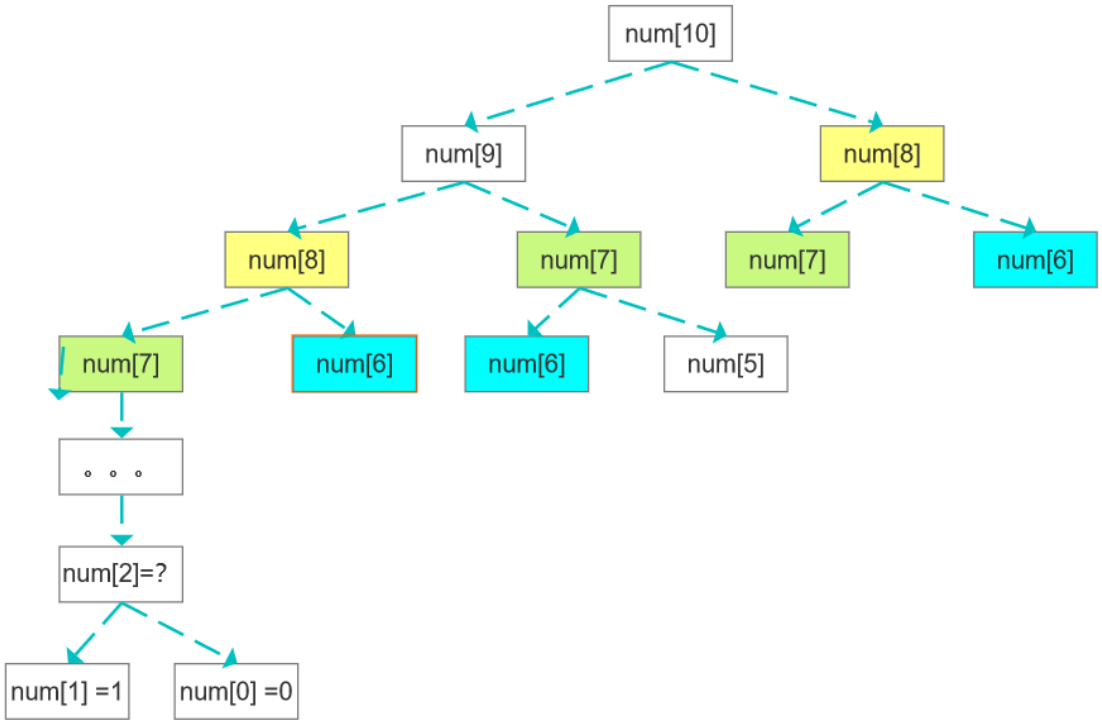
解法一：暴力递归




```
public static int calculate(int num){
    if(num == 0 ){
        return 0;
    }
    if(num == 1){
        return 1;
    }
    return calculate(num-1) + calculate(num-2);
}
```

解法二：去重递归

递归得出具体数值之后、存储到一个集合(下标与数列下标一致)，后面递归之前先到该集合查询一次，如果查到则无需递归、直接取值。查不到再进行递归计算

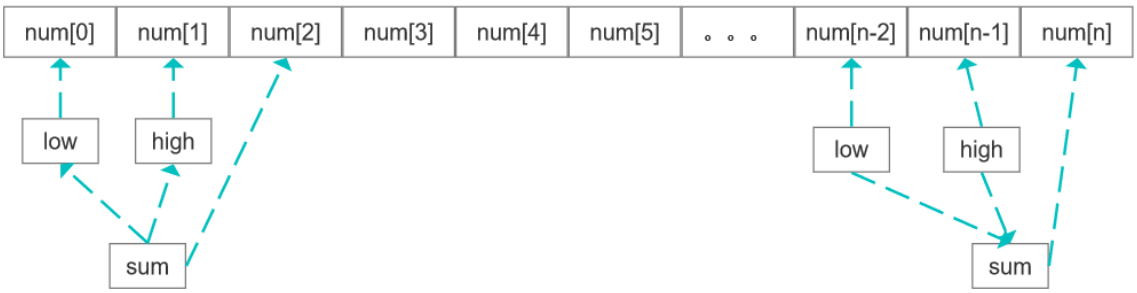



```
public static int calculate2(int num){
    int[] arr = new int[num+1];
    return recurse(arr,num);
}

private static int recurse(int[] arr, int num) {
    if(num == 0 ){
        return 0;
    }
    if(num == 1){
        return 1;
    }
    if(arr[num] != 0){
        return arr[num];
    }
    arr[num] = recurse(arr,num-1) + recurse(arr,num-2);
    return arr[num];
}
```

解法三：双指针迭代

基于去重递归优化，集合没有必要保存每一个下标值，只需保存前两位即可，向后遍历，得出N的值




```
public static int iterate(int num){
    if(num == 0 ){
        return 0;
    }
    if(num == 1){
        return 1;
    }
    int low = 0,high = 1;
    for(int i=2; i<= num; i++){
        int sum = low + high;
        low = high;
        high = sum;
    }
    return high;
}
```

环形链表

给定一个链表，判断链表中是否有环。
如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达该节点，则链表中存在环
如果链表中存在环，则返回 true 。 否则，返回 false 。

解法一：哈希表

```
public static boolean hasCycle(ListNode head) {
    Set<ListNode> seen = new HashSet<ListNode>();
    while (head != null) {
        if (!seen.add(head)) {
            return true;
        }
        head = head.next;
    }
    return false;
}
```

解法二：双指针

```
public static boolean hasCycle2(ListNode head) {
```

```
    if (head == null || head.next == null) {
        return false;
    }
    ListNode slow = head;
    ListNode fast = head.next;
    while (slow != fast) {
        if (fast == null || fast.next == null) {
            return false;
        }
        slow = slow.next;
        fast = fast.next.next;
    }
    return true;
}
```

排列硬币

总共有 n 枚硬币，将它们摆成一个阶梯形状，第 k 行就必须正好有 k 枚硬币。
给定一个数字 n，找出可形成完整阶梯行的总行数。
n 是一个非负整数，并且在32位有符号整型的范围内

解法一：迭代

从第一行开始排列，排完一列、计算剩余硬币数，排第二列，直至剩余硬币数小于或等于行数

```
public static int arrangeCoins(int n) {
    for(int i=1; i<=n; i++){
        n = n-i;
        if (n <= i){
            return i;
        }
    }
    return 0;
}
```

解法二：二分查找

假设能排 n 行，计算 n 行需要多少硬币数，如果大于 n，则排 n/2行，再计算硬币数和 n 的大小关系

```
public static int arrangeCoins2(int n) {
    int low = 0, high = n;
    while (low <= high) {
        long mid = (high - low) / 2 + low;
        long cost = ((mid + 1) * mid) / 2;
```

```
            if (cost == n) {
                return (int)mid;
            } else if (cost > n) {
                high = (int)mid - 1;
            } else {
                low = (int)mid + 1;
            }
        }
        return high;
    }
```

解法三：牛顿迭代

使用牛顿迭代求平方根，(x + n/x)/2

假设能排 x 行 则 1 + 2 + 3 + ...+ x = n，即 x(x+1)/2 = n 推导出 x = 2n - x

```
public static double sqrts(double x,int n){
    double res = (x + (2*n-x) / x) / 2;
    if (res == x) {
        return x;
    } else {
        return sqrts(res,n);
    }
}
```

合并两个有序数组

两个有序整数数组 nums1 和 nums2，将 nums2 合并到 nums1 中，使 nums1 成为一个有序数组。

初始化 nums1 和 nums2 的元素数量分别为 m 和 n 。假设 nums1 的空间大小等于 m + n，这样它就有足够的空间保存来自 nums2 的元素。

解法一：合并后排序

```
public void merge(int[] nums1, int m, int[] nums2, int n) {
    System.arraycopy(nums2, 0, nums1, m, n);
    Arrays.sort(nums1);
}
```

- 时间复杂度 : O((n+m)log(n+m))。
- 空间复杂度 : O(1)。

解法二：双指针 从前往后

将两个数组按顺序进行比较，放入新的数组

```
```

```
public void merge(int[] nums1, int m, int[] nums2, int n) {
    int [] nums1_copy = new int[m];
    System.arraycopy(nums1, 0, nums1_copy, 0, m);//拷贝数组1

    int p1 = 0;//指向数组1的拷贝
    int p2 = 0;//指向数组2

    int p = 0;//指向数组1

    //将数组1当成空数组，比较数组1的拷贝和数组2，将较小的放入空数组
    while ((p1 < m) && (p2 < n))
        nums1[p++] = (nums1_copy[p1] < nums2[p2]) ? nums1_copy[p1++] :
nums2[p2++];

    //数组2和数组1不等长，将多出的元素拷贝
    if (p1 < m)
        System.arraycopy(nums1_copy, p1, nums1, p1 + p2, m + n - p1 - p2);
    if (p2 < n)
        System.arraycopy(nums2, p2, nums1, p1 + p2, m + n - p1 - p2);
}
```

- 时间复杂度 : O(n + m)。
- 空间复杂度 : O(m)。

解法三：双指针优化

从后往前

```
public void merge(int[] nums1, int m, int[] nums2, int n) {
    int p1 = m - 1;
    int p2 = n - 1;
    int p = m + n - 1;

    while ((p1 >= 0) && (p2 >= 0))
        nums1[p--] = (nums1[p1] < nums2[p2]) ? nums2[p2--] : nums1[p1--];

    System.arraycopy(nums2, 0, nums1, 0, p2 + 1);
}
```

- 时间复杂度 : O(n + m)。
- 空间复杂度 : O(1)。

子数组最大平均数

给一个整数数组，找出平均数最大且长度为 `k` 的下标连续的子数组，并输出该最大平均数。

滑动窗口：

窗口移动时，窗口内的和等于sum加上新加进来的值，减去出去的值

```
public double findMaxAverage(int[] nums, int k) {
    int sum = 0;
    int n = nums.length;
    for (int i = 0; i < k; i++) {
        sum += nums[i];
    }
    int maxSum = sum;
    for (int i = k; i < n; i++) {
        sum = sum - nums[i - k] + nums[i];
        maxSum = Math.max(maxSum, sum);
    }
    return 1.0 * maxSum / k;
}
```

二叉树的最小深度

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

解法一：深度优先

遍历整颗数，找到每一个叶子节点，从叶子节点往上开始计算，左右子节点都为空则记录深度为1

左右子节点只有一边，深度记录为子节点深度+1

左右两边都有子节点，则记录左右子节点的深度较小值+1

```
public int minDepth(TreeNode root) {
    if (root == null) {
        return 0;
```

```
    }

    if (root.left == null && root.right == null) {
        return 1;
    }

    int min_depth = Integer.MAX_VALUE;
    if (root.left != null) {
        min_depth = Math.min(minDepth(root.left), min_depth);
    }
    if (root.right != null) {
        min_depth = Math.min(minDepth(root.right), min_depth);
    }

    return min_depth + 1;
}
```

时间复杂度：O(N)

空间复杂度：$O(\log N)$ 取决于树的高度

解法二：广度优先

从上往下，找到一个节点时，标记这个节点的深度。查看该节点是否为叶子节点，如果是直接返回深度

如果不是叶子节点，将其子节点标记深度(在父节点深度的基础上加1)，再判断该节点是否为叶子节点

```
class QueueNode {
    TreeNode node;
    int depth;

    public QueueNode(TreeNode node, int depth) {
        this.node = node;
        this.depth = depth;
    }
}

public int minDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }

    Queue<QueueNode> queue = new LinkedList<QueueNode>();
    queue.offer(new QueueNode(root, 1));
    while (!queue.isEmpty()) {
        QueueNode nodeDepth = queue.poll();
        TreeNode node = nodeDepth.node;
        int depth = nodeDepth.depth;
        if (node.left == null && node.right == null) {
            return depth;
        }
        if (node.left != null) {
            queue.offer(new QueueNode(node.left, depth + 1));
        }
        if (node.right != null) {
            queue.offer(new QueueNode(node.right, depth + 1));
        }
```

```
    }
    return 0;
}
```

时间复杂度：O(N)

空间复杂度：O(N)

最长连续递增序列

给定一个未经排序的整数数组，找到最长且连续递增的子序列，并返回该序列的长度。

序列的下标是连续的

贪心算法

从0开始寻找递增序列，并将长度记录，记录递增序列的最后一个下标，然后从该下标继续寻找，记录长度，取长度最大的即可

```
public static int findLength(int[] nums) {
    int ans = 0;
    int start = 0;
    for (int i = 0; i < nums.length; i++) {
        if (i > 0 && nums[i] <= nums[i - 1]) {
            start = i;
        }
        ans = Math.max(ans, i - start + 1);
    }
    return ans;
}
```

柠檬水找零

在柠檬水摊上，每一杯柠檬水的售价为 5 美元。

顾客排队购买你的产品，一次购买一杯。

每位顾客只买一杯柠檬水，然后向你付 5 美元、10 美元或 20 美元。必须给每个顾客正确找零

注意，一开始你手头没有任何零钱。

如果你能给每位顾客正确找零，返回 true ，否则返回 false 。

```
输入：[5,5,5,10,20]
输出：true

输入：[10,10]
输出：false
```

贪心：

```java
public boolean lemonadeChange(int[] bills) {
    int five = 0, ten = 0;
    for (int bill : bills) {
        if (bill == 5) {
            five++;
        } else if (bill == 10) {
            if (five == 0) {
                return false;
            }
            five--;
            ten++;
        } else {
            if (five > 0 && ten > 0) {
                five--;
                ten--;
            } else if (five >= 3) {
                five -= 3;
            } else {
                return false;
            }
        }
    }
    return true;
}
```

三角形的最大周长

给定由一些正数（代表长度）组成的数组 `A`，返回由其中三个长度组成的、面积不为零的三角形的最大周长。

如果不能形成任何面积不为零的三角形，返回 `0`。

贪心：

先小到大排序，假设最长边是最后下标，另外两条边是倒数第二和第三下标，则此时三角形周长最大

n < (n-1) + (n-2)，如果不成立，意味着该数组中不可能有另外两个值之和大于n，此时将n左移，重新计算

```
public int largestPerimeter(int[] A) {
    Arrays.sort(A);
    for (int i = A.length - 1; i >= 2; --i) {
        if (A[i - 2] + A[i - 1] > A[i]) {
            return A[i - 2] + A[i - 1] + A[i];
        }
    }
    return 0;
}
```

二叉树遍历

从根节点往下查找，先找左子树、直至左子树为空(左子节点逐个入栈、直至左子节点为空)，再找右子树(出栈找右子节点)

前序遍历：根左右，第一次经过节点即打印，直到打印null，往回溯，打印右子树

中序遍历：左根右，第二次经过该节点时进行打印，即左边回溯时

后序遍历：左右根，第三次经过该节点时进行打印，即右边回溯时

层序遍历：按照层级，从上往下，从左到右。使用广度优先搜索算法。

递归遍历：

```
public static void preorder(TreeNode root) {
    if (root == null) {
        return;
    }
    //System.out.println(root.val);//前序    第一次成为栈顶
    preorder(root.left);
    System.out.println(root.val);//中序    第二次成为栈顶
    preorder(root.right);
    //System.out.println(root.val);//后序    第三次成为栈顶
}
```

迭代遍历：

```
//前序：使用stack记录递归路径，左子节点后添加保证先出栈
public static void preOrder2(TreeNode head) {
    if (head != null) {
        Stack<TreeNode> stack = new Stack<TreeNode>();
        stack.add(head);
        while (!stack.isEmpty()) {
            head = stack.pop();
            if(head != null){
                System.out.println(head.val);
```

```
                stack.push(head.right);
                stack.push(head.left);
            }
        }
    }
}

//中序：将左子节点入栈，出栈打印值，然后添加右子节点
public static void preOrder3(TreeNode head) {
    if (head != null) {
        Stack<TreeNode> stack = new Stack<TreeNode>();
        while (!stack.isEmpty() || head != null) {
            if (head != null) {
                stack.push(head);
                head = head.left;
            } else {
                head = stack.pop();
                System.out.println(head.val);
                head = head.right;
            }
        }
    }
}

//后序：
public static void postorderTraversal(TreeNode root) {
    if (root == null) {
        return ;
    }

    Deque<TreeNode> stack = new LinkedList<TreeNode>();
    TreeNode prev = null;
    while (root != null || !stack.isEmpty()) {
        while (root != null) {
            stack.push(root);
            root = root.left;
        }
        root = stack.pop();//root的左子节点为null
        if (root.right == null || root.right == prev) {//右子节点为null，或者右子节
点已打印
            System.out.println(root.val);
            prev = root;
            root = null;
        } else {//右子节点有值，重新入栈
            stack.push(root);
            root = root.right;
        }
    }
}
```

层序遍历：

```
public static void levelTraversal(Node root) {
        Queue<Node> q = new LinkedList<>();
        q.add(root);
```

```
        while (!q.isEmpty()) {
            Node temp = q.poll();
            if (temp != null) {
                System.out.print(temp.value + "  ");
                q.add(temp.left);
                q.add(temp.right);
            }
        }
    }

public static void deepOrder(TreeNode root) {
    if (root == null) {
        return ;
    }

    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        for (int i = 1; i <= queue.size(); ++i) {
            TreeNode node = queue.poll();
            System.out.println(node.val);
            if (node.left != null) {
                queue.offer(node.left);
            }
            if (node.right != null) {
                queue.offer(node.right);
            }
        }
    }
}

private static List order(TreeNode root, int i, ArrayList list) {
    if (root == null) {
        return null;
    }
    int length = list.size();
    if(length <= i){
        for(int j=0; j<= i-length; j++){
            list.add(length+j,null);
        }
    }
    list.set(i,root.val);
    order(root.left, 2 * i,list);
    order(root.right, 2 * i + 1,list);
    return list;
}
```

线索二叉树：

在N个节点的二叉树中，每个节点有2个指针，所以一共有2N个指针，除了根节点以外，每一个节点都有一个指针从它的父节点指向它，所以一共使用了N-1个指针，所以剩下2N-(N-1)也就是N+1个空指针；

如果能利用这些空指针域来存放指向该节点的直接前驱或是直接后继的指针，则可由此信息直接找到在该遍历次序下的前驱节点或后继节点，从而比递归遍历提高了遍历速度，节省了建立系统递归栈所使用的存储空间；

这些被重新利用起来的空指针就被称为线索（Thread），加上了线索的二叉树就是线索二叉树

实现思路：按某种次序遍历二叉树，在遍历过程中用线索取代空指针即可。以中序遍历为例，首先找到中序遍历的开始节点，然后利用线索依次查找后继节点即可。

由于它充分利用了空指针域的空间（等于节省了空间），又保证了创建时的一次遍历就可以终生受用前驱、后继的信息（这意味着节省了时间），所以在实际问题中，如果所使用的二叉树需要经常遍历或查找节点时需要某种遍历中的前驱和后继，那么采用线索二叉链表的存储结构就是不错的选择

morris遍历：构建中序线索二叉树的过程中，如果发现前驱节点的右指针指向自身，则将指针（线索）删除

```
public static void morrisPre(Node cur) {
    if(head == null){
        return;
    }
    Node mostRight = null;
    while (cur != null){
        // cur表示当前节点，mostRight表示cur的左孩子的最右节点
        mostRight = cur.left;
        if(mostRight != null){
            // cur有左孩子，找到cur左子树最右节点
            while (mostRight.right !=null && mostRight.right != cur){
                mostRight = mostRight.right;
            }
            // mostRight的右孩子指向空，让其指向cur，cur向左移动
            if(mostRight.right == null){
                mostRight.right = cur;
                System.out.print(cur.value+" ");
                cur = cur.left;
                continue;
            }else {
                // mostRight的右孩子指向cur，让其指向空，cur向右移动
                mostRight.right = null;
            }
        }else {
            System.out.print(cur.value + " ");
        }
        cur = cur.right;
    }
}


public static void morrisIn(Node cur) {
    if(head == null){
        return;
    }
    Node mostRight = null;
    while (cur != null){
        mostRight = cur.left;
        if(mostRight != null){
            while (mostRight.right !=null && mostRight.right != cur){
                mostRight = mostRight.right;
            }
```

```
            if(mostRight.right == null){
                mostRight.right = cur;
                cur = cur.left;
                continue;
            }else {
                mostRight.right = null;
            }
        }
        System.out.print(cur.value+" ");
        cur = cur.right;
    }
}


public static void morrisPos(TreeNode cur) {
    if (cur == null) {
        return;
    }
    TreeNode head = cur;
    TreeNode mostRight = null;
    while (cur != null) {
        mostRight = cur.left;
        if (mostRight != null) {
            while (mostRight.right != null && mostRight.right != cur) {
                mostRight = mostRight.right;
            }
            if (mostRight.right == null) {
                mostRight.right = cur;
                cur = cur.left;
                continue;
            } else {
                mostRight.right = null;
                printEdge(cur.left);
            }
        }
        cur = cur.right;
    }
    printEdge(head);
    System.out.println();
}


public static void printEdge(TreeNode head) {
    TreeNode tail = reverseEdge(head);
    TreeNode cur = tail;
    while (cur != null) {
        System.out.print(cur.val + " ");
        cur = cur.right;
    }
    reverseEdge(tail);
}


public static TreeNode reverseEdge(TreeNode from) {
    TreeNode pre = null;
    TreeNode next = null;
    while (from != null) {
        next = from.right;
```

```
            from.right = pre;
            pre = from;
            from = next;
        }
        return pre;
    }


    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<Integer>();
        if (root == null) {
            return res;
        }

        Deque<TreeNode> stack = new LinkedList<TreeNode>();
        TreeNode prev = null;
        while (root != null || !stack.isEmpty()) {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            if (root.right == null || root.right == prev) {
                res.add(root.val);
                prev = root;
                root = null;
            } else {
                stack.push(root);
                root = root.right;
            }
        }
        return res;
    }
```

省份数量

有 n 个城市，其中一些彼此相连，另一些没有相连。如果城市 a 与城市 b 直接相连，且城市 b 与城市 c 直接相连，那么城市 a 与城市 c 间接相连。

省份 是一组直接或间接相连的城市，组内不含其他没有相连的城市。

给你一个 n x n 的矩阵 isConnected ，其中 isConnected[i][j] = 1 表示第 i 个城市和第 j 个城市直接相连，而 isConnected[i][j] = 0 表示二者不直接相连。

返回矩阵中 省份 的数量。

亲戚问题、朋友圈问题

解法一：深度优先

获取一个城市，通过递归找到离该城市最远的城市，标记为已访问，然后逐个向内进行标记

```
public int findCircleNum(int[][] isConnected) {
    int provinces = isConnected.length;
```

```
    boolean[] visited = new boolean[provinces];
    int circles = 0;
    for (int i = 0; i < provinces; i++) {
        if (!visited[i]) {
            dfs(isConnected, visited, provinces, i);
            circles++;
        }
    }
    return circles;
}

public void dfs(int[][] isConnected, boolean[] visited, int provinces, int i) {
    for (int j = 0; j < provinces; j++) {
        if (isConnected[i][j] == 1 && !visited[j]) {
            visited[j] = true;
            dfs(isConnected, visited, provinces, j);
        }
    }
}
```

解法二：广度优先

获取一个城市，先标记与该城市直连的城市(最近的)，然后逐步向外扩散寻找

```
public int bfs(int[][] isConnected) {
    int provinces = isConnected.length;
    boolean[] visited = new boolean[provinces];
    int circles = 0;
    Queue<Integer> queue = new LinkedList<Integer>();
    for (int i = 0; i < provinces; i++) {
        if (!visited[i]) {
            queue.offer(i);
            while (!queue.isEmpty()) {
                int j = queue.poll();
                visited[j] = true;
                for (int k = 0; k < provinces; k++) {
                    if (isConnected[j][k] == 1 && !visited[k]) {
                        queue.offer(k);
                    }
                }
            }
            circles++;
        }
    }
    return circles;
}
```

解法三：并查集

将每个城市看成一个节点，如果两个城市相连，则建立树关系，选出其中一个为head，

如果两个树中的节点也相连，则将其中一个head设置为另一个树的head

两个方法：一个寻找head节点，一个合并树

```
static int mergeFind(int[][] isConnected){
    int provinces = isConnected.length;
    int[] head = new int[provinces];
    int[] level = new int[provinces];
    for (int i = 0; i < provinces; i++) {
        head[i] = i;
        level[i] = 1;

    }
    for (int i = 0; i < provinces; i++) {
        for (int j = i + 1; j < provinces; j++) {
            if (isConnected[i][j] == 1) {
                merge(i, j,head,level);
            }
        }
    }
    int count = 0;
    //找出所有的head
    for (int i = 0; i < provinces; i++) {
        if (head[i] == i) {
            count++;
        }
    }
    return count;
}
//查找head节点
static int find(int x, int[] arr) {
    if(arr[x] == x)
        return x;
    else
        arr[x] = find(arr[x],arr);//路径压缩，每一个节点直接能找到head
    return arr[x];
}

static void merge(int x, int y,int[] arr,int[] level) {
    int i = find(x,arr);
    int j = find(y,arr);
    //深度比较短的树的head往深度大的树上挂，使合并后的深度尽量小
    if(i == j){
        return;
    }
    if(level[i] <= level[j]){
        arr[i] = j;
    }else{
        arr[j] = i;
    }
    //深度加1
    level[j]++;

}
```

预测赢家

给定一个表示分数的非负整数数组。 玩家 1 从数组任意一端拿取一个分数，随后玩家 2 继续从剩余数组任意一端拿取分数，然后玩家 1 拿，…… 。每次一个玩家只能拿取一个分数，分数被拿取之后不再可取。直到没有剩余分数可取时游戏结束。最终获得分数总和最多的玩家获胜。

给定一个表示分数的数组，预测玩家1是否会成为赢家。你可以假设每个玩家的玩法都会使他的分数最大化。

两个值的时候必然是取较大的，三个值，取一个能使自己分数和最大的，后手必然留较小的给先手，因此先手选一个值加上该较小值最大化

```
static int maxScore(int[] nums, int l, int r) {
    //剩下一个值，只能取该值
    if (l == r) {
        return nums[l];
    }
    int selectLeft =0, selectRight=nums.length-1;
    //剩下大于两个值，先手选一边(使自己得分最高的一边)，后手则选使对手得分最低的一边
    if ((r - l) >= 2) {
        selectLeft = nums[l] +Math.min(maxScore(nums, l+2, r),maxScore(nums,
l+1, r-1));
        selectRight = nums[r] +Math.min(maxScore(nums, l+1, r-1),maxScore(nums,
l, r-2));
    }
    //剩下两个值，取较大的
    if ((r - l) == 1) {
        selectLeft = nums[l];
        selectRight = nums[r];
    }

    return Math.max(selectLeft, selectRight);
}

int getScore(int[] nums, int start, int end) {
    int selectLeft, selectRight;
    int gap = end - start;
    if (gap == 0) {
        return nums[start];
    } else if (gap == 1) { // 此时直接取左右的值就可以
        selectLeft = nums[start];
        selectRight = nums[end];
    } else if (gap >= 2) { // 如果gap大于2，递归计算selectLeft和selectRight
        // 计算的过程为什么用min，因为要按照对手也是最聪明的来计算。
        int num = getScore(nums, start + 1, end - 1);
        selectLeft = nums[start] + min(getScore(nums, start + 2, end), num);
        selectRight = nums[end] + min(num, getScore(nums, start, end - 2));
    }
    return max(selectLeft, selectRight);
}


bool PredictTheWinner(int[] nums) {
    int sum = 0;
    for (int i : nums) {
        sum += i;
```

```
    }
    int player1 = getScore(nums, 0, nums.size() - 1);
    int player2 = sum - player1;
    // 如果最终两个玩家的分数相等，那么玩家 1 仍为赢家，所以是大于等于。
    return player1 >= player2;

    //return getScore(nums, 0, nums.size() - 1) >=0 ;
}

//差值
int getScore(int[] nums, int start, int end) {
    if (end == start) {
        return nums[start];
    }
    int selectLeft = nums[start] - getScore(nums, start + 1, end);
    int selectRight = nums[end] - getScore(nums, start, end - 1);
    return max(selectLeft, selectRight);
}
```

动态规划：使用二维数组存储差值

```
public boolean PredictTheWinner(int[] nums) {
    int length = nums.length;
    int[][] dp = new int[length][length];
    for (int i = 0; i < length; i++) {
        dp[i][i] = nums[i];
    }
    for (int i = length - 2; i >= 0; i--) {
        for (int j = i + 1; j < length; j++) {
            //j = i +1 因此可以优化为一维数组,下标位置相同才有值，据此推导其他的值
            //Math.max(nums[i] - dp[j][j], nums[j] - dp[j - 1][j - 1]);
            dp[i][j] = Math.max(nums[i] - dp[i + 1][j], nums[j] - dp[i][j - 1]);
        }
    }
    return dp[0][length - 1] >= 0;
}
```

香槟塔

我们把玻璃杯摆成金字塔的形状，其中第一层有1个玻璃杯，第二层有2个，依次类推到第100层，每个玻璃杯(250ml)将盛有香槟。

从顶层的第一个玻璃杯开始倾倒一些香槟，当顶层的杯子满了，任何溢出的香槟都会立刻等流量的流向左右两侧的玻璃杯。当左右两边的杯子也满了，就会等流量的流向它们左右两边的杯子，依次类推。（当最底层的玻璃杯满了，香槟会流到地板上）

例如，在倾倒一杯香槟后，最顶层的玻璃杯满了。倾倒了两杯香槟后，第二层的两个玻璃杯各自盛放一半的香槟。在倒三杯香槟后，第二层的香槟满了 - 此时总共有三个满的玻璃杯。在倒第四杯后，第三层中间的玻璃杯盛放了一半的香槟，他两边的玻璃杯各自盛放了四分之一的香槟

现在当倾倒了非负整数杯香槟后，返回第 i 行 j 个玻璃杯所盛放的香槟占玻璃杯容积的比例（i 和 j都从0开始）。

```
public double champagneTower(int poured, int query_row, int query_glass) {
    double[][] A = new double[102][102];
    A[0][0] = (double) poured;
    for (int r = 0; r <= query_row; ++r) {
        for (int c = 0; c <= r; ++c) {
            double q = (A[r][c] - 1.0) / 2.0;
            if (q > 0) {
                A[r+1][c] += q;
                A[r+1][c+1] += q;
            }
        }
    }

    return Math.min(1, A[query_row][query_glass]);
}
```

井字游戏

用字符串数组作为井字游戏的游戏板 board，判断该游戏板有没有可能最终形成
游戏板是一个 3 x 3 数组，由字符 " "，"X" 和 "O" 组成。字符 " " 代表一个空位。
两个玩家轮流将字符放入空位，一个玩家执X棋，另一个玩家执O棋
“X” 和 “O” 只允许放置在空位中，不允许对已放有字符的位置进行填充。
当有 3 个相同（且非空）的字符填充任何行、列或对角线时，游戏结束，board生成

分类讨论

```
public static boolean validBoard(String[] board) {
    int xCount = 0, oCount = 0;
    for (String row: board)
        for (char c: row.toCharArray()) {
            if (c == 'X') xCount++;
            if (c == 'O') oCount++;
        }
    //X与O    一样多，或者X比O多一个(X赢则X多一个，O赢则一样多)
    if (oCount != xCount && oCount != xCount - 1) return false;
    if (win(board, "XXX") && oCount != xCount - 1) return false;
    if (win(board, "OOO") && oCount != xCount) return false;
    return true;
}

public static boolean win(String[] board, String flag) {
    for (int i = 0; i < 3; ++i) {
        //纵向3连
        if (flag.equals("" + board[i].charAt(0) + board[i].charAt(1) +
board[i].charAt(2)))
            return true;
        //横向3连
```

```
        if (flag.equals(board[i]))
            return true;
    }
    //  \向3连
    if (flag.equals(""+
board[0].charAt(0)+board[1].charAt(1)+board[2].charAt(2)))
        return true;
    //   /向3连
    if
(flag.equals(""+board[0].charAt(2)+board[1].charAt(1)+board[2].charAt(0)))
        return true;
    return false;
}
```

打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。
给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下 ， 一夜之内能够偷窃到的最高金额。

输入：[1,2,3,1]　输出：4
输入：[2,7,9,3,1]　输出：12

```
static int maxMoney(int[] nums,int index){
    if (nums == null || index < 0) {
        return 0;
    }
    if (index == 0) {
        return nums[0];
    }
    return Math.max(maxMoney(nums,index - 2) + nums[index], maxMoney(nums,index
- 1));
}


static int maxMoney(int[] nums){
    if (nums == null || nums.length == 0) {
        return 0;
    }
    int length = nums.length;
    if (length == 1) {
        return nums[0];
    }

    /*int[] dp = new int[length];
        dp[0] = nums[0];
        dp[1] = Math.max(nums[0], nums[1]);
        for (int i = 2; i < length; i++) {
            dp[i] = Math.max(dp[i - 2] + nums[i], dp[i - 1]);
        }
```

```
        return dp[length - 1];*/
    int first = nums[0], second = Math.max(nums[0], nums[1]);
    for (int i = 2; i < length; i++) {
        int temp = second;
        second = Math.max(first + nums[i], second);
        first = temp;
    }
    return second;
}
```

如果房子首尾相连：

```
public int rob(int[] nums) {
    int length = nums.length;
    if (length == 1) {
        return nums[0];
    } else if (length == 2) {
        return Math.max(nums[0], nums[1]);
    }
    return Math.max(robRange(nums, 0, length - 2), robRange(nums, 1, length -
1));
}

public int robRange(int[] nums, int start, int end) {
    int first = nums[start], second = Math.max(nums[start], nums[start + 1]);
    for (int i = start + 2; i <= end; i++) {
        int temp = second;
        second = Math.max(first + nums[i], second);
        first = temp;
    }
    return second;
}
```

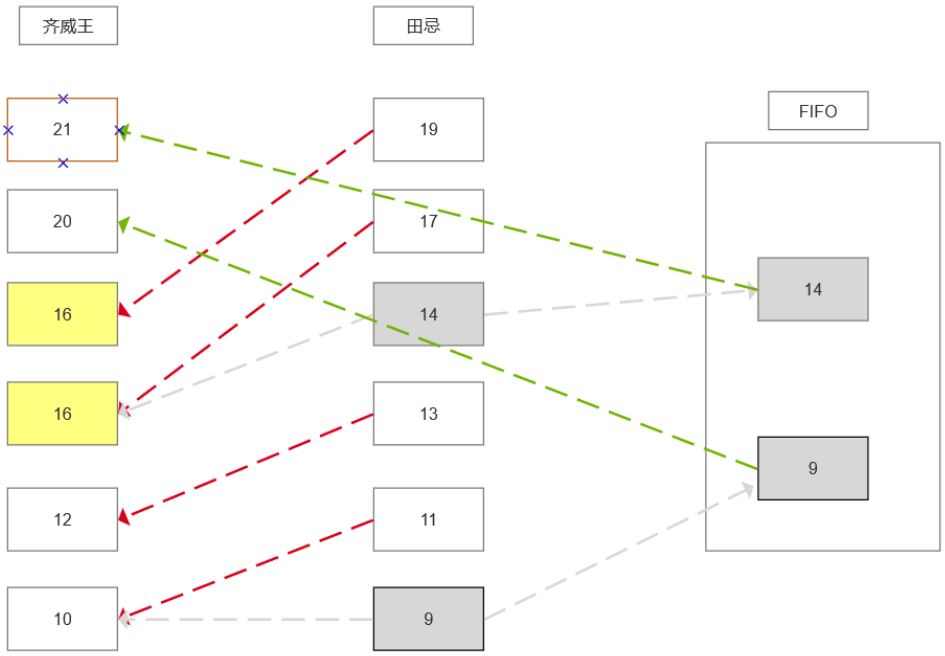
```
public int rob(TreeNode root) {
    int[] rootStatus = dfs(root);
    return Math.max(rootStatus[0], rootStatus[1]);
}

public int[] dfs(TreeNode node) {
    if (node == null) {
        return new int[]{0, 0};
    }
    int[] l = dfs(node.left);
    int[] r = dfs(node.right);
    int selected = node.val + l[1] + r[1];
    int notSelected = Math.max(l[0], l[1]) + Math.max(r[0], r[1]);
    return new int[]{selected, notSelected};
}
```

优势洗牌

给定两个大小相等的数组 A 和 B，A 相对于 B 的优势可以用满足 A[i] > B[i] 的索引 i 的数目来描述。

返回 A 的任意排列，使其相对于 B 的优势最大化。




```
public static int[] advantageCount(int[] A, int[] B) {
    int[] sortedA = A.clone();
    Arrays.sort(sortedA);//找一个代价最小的去匹配B中的，比B大，在A中又是最小的
    int[] sortedB = B.clone();
    Arrays.sort(sortedB);//避免比较时，A每次都要重头遍历

    Map<Integer, Deque<Integer>> assigned = new HashMap();
    for (int b: B)
        assigned.put(b, new LinkedList());

    Deque<Integer> remaining = new LinkedList();

    int j = 0;
    for (int a: sortedA) {
        if (a > sortedB[j]) {
            assigned.get(sortedB[j++]).add(a);
        } else {
            remaining.add(a);
        }
    }

    int[] ans = new int[B.length];
    for (int i = 0; i < B.length; ++i) {
        if (assigned.get(B[i]).size() > 0)
            ans[i] = assigned.get(B[i]).removeLast();
        else
```

```
            ans[i] = remaining.removeLast();
        }
        return ans;
    }
```

- 时间复杂度：O(N log N)，其中 N 是 `A` 和 `B` 的长度。
- 空间复杂度：O(N)。

Dota2参议院

Dota2 的世界里有两个阵营：Radiant(天辉)和 Dire(夜魇)

Dota2 参议院由来自两派的参议员组成。现在参议院希望对一个 Dota2 游戏里的改变作出决定。他们以一个基于轮为过程的投票进行。在每一轮中，每一位参议员都可以行使两项权利中的一项：

禁止一名参议员的权利：参议员可以让另一位参议员在这一轮和随后的几轮中丧失所有的权利。

宣布胜利： 如果参议员发现有权利投票的参议员都是同一个阵营的，他可以宣布胜利并决定在游戏中的有关变化。

给定一个字符串代表每个参议员的阵营。字母 “R” 和 “D” 分别代表了 Radiant（天辉）和 Dire（夜魇）。然后，如果有 n 个参议员，给定字符串的大小将是 n。

以轮为基础的过程从给定顺序的第一个参议员开始到最后一个参议员结束。这一过程将持续到投票结束。所有失去权利的参议员将在过程中被跳过。

假设每一位参议员都足够聪明，会为自己的政党做出最好的策略，你需要预测哪一方最终会宣布胜利并在 Dota2 游戏中决定改变。输出应该是 Radiant 或 Dire。

```
public String predictPartyVictory(String senate) {
    int n = senate.length();
    Queue<Integer> radiant = new LinkedList<Integer>();
    Queue<Integer> dire = new LinkedList<Integer>();
    for (int i = 0; i < n; ++i) {
        if (senate.charAt(i) == 'R') {
            radiant.offer(i);
        } else {
            dire.offer(i);
        }
    }
    while (!radiant.isEmpty() && !dire.isEmpty()) {
        int radiantIndex = radiant.poll(), direIndex = dire.poll();
        if (radiantIndex < direIndex) {
            radiant.offer(radiantIndex + n);
        } else {
            dire.offer(direIndex + n);
        }
    }
    return !radiant.isEmpty() ? "Radiant" : "Dire";
}
```

时间和空间：O(n)