算法 2

常见的面试时间复杂度和对应的算法

- O(logN)二分法较多
- O(根号N)分解质因数(极少)
- O(N)双指针，单调栈，枚举法
- O(NlogN)排序，O(N*logN*的数据结构上的操作
- O(N^2),O(N^3)，动态规划等
- O(2^N)，组合类搜索问题
- O(N!)，排列类搜索问题

如何刷题以达到事半功倍的效果

面试官眼中的求职者

HR主要从衣着，谈吐和外貌看

技术面试官从代码来看

- 变量命名
- 多重嵌套循环，应将其变为子函数

- 有工程化的意识去优化自己的代码，尽可能提高代码的可读性
- 代码要定期进行codeReview
- 项目经验：耦合度高、重复代码、全局变量

- 用含义清晰的变量名命名+简单易读的处理逻辑>>用注释去解释让人看不懂的代码
- 通过适当的子函数化的代码包装，多加空行，虽然代码更长了，但是能让你的代码：
- 通过子函数化、避免全局变量等手段可以让你出BUG的概率大大降低

面试评价体系

逻辑思维能力

- 是否能很快的写出Work Solution
- 是否能够在面试官点出问题后优化自己的Solution

代码质量

- 代码到底写完没有
- 代码风格好不好
    - 可读性
    - 变量名、函数名命名
    - 空格和空行的正确使用
- 异常检测
- Bug Free

沟通能力

把面试官当作Co-worker而不是考官，让面试官愿意和你一起工作。

几个面试沟通法则

- 做一个题之前，先沟通清楚，得到面试官肯定，再开始写代码，写完之后再解释。
  - 不要闷头写
  - 也不要一边写一边解释太多（容易写不完）
- 可以要提示，经过提示做出来的题，也是可以拿到Hire的
  - 但是先自己努力想一下，别太容易放弃，容易让人觉得不会主动思考问题
- 别和面试官吵架
  - 面试官带着答案来面试你的
  - 不同意见在大部分情况下，都是你自己想错了
- 会就会，不会就不会，不要遮遮掩掩，坦诚很重要
  - 容易让人觉得和你沟通“不顺畅”
  - 做过的题就说做过，不要故意说没做过
  - 因为他既然已经怀疑你做过了，即使你说没有，他也无法打消这个顾虑，还不如让他换题。

重点考察方法

1. 拓扑排序算法、二分法、哈希表、二叉查找树
2. 动态规划
3. 分治法、堆

4. 字典树、并查集、最小生成树算法、贪心法

典型题目

题目：最长回文子串

//动态规划

//枚举法

代码质量Code Quality很重要

好的代码质量包括

- Bug Free
- 好的代码风格，包括变量名命名规范有意义，合理的使用空格，善用空行
- 容易让人读懂的逻辑
- 没有冗余代码
- 有边界检测和异常处理

如何快速优化自己的codingQuality

Coding Style 相关

- 二元运算符两边加空格，单元运算符两边不加
- 花括号和for、if之间要加空格， 圆括号和if之间要加空格
- 用空行分隔开不同的逻辑块

- 逗号后面加空格

Readability相关

- 函数名和变量名用一到两个单词来作为名称
- 确保一个函数内部不超过三层缩进
- 多用子函数来减少入口函数的代码量
- 多用continue，少用if

```
//将
for (条件){
    if (条件1){
        做一些处理
        if (条件2) {
            做一些处理
            做一些处理
        }
    }
}

//替换成

for (条件){
    if (!条件2) {
        continue
    }
    做一些处理

    if (!条件2) {
        continue
    }
    做一些处理
    做一些处理
    做一些处理
    做一些处理
```

Bug Free相关

- 不管有没有可能出问题，入口函数的参数都要进行异常检测
- 访问一个下标的时候，必须确保这个下标不越界
- 访问一个对象的属性或者方法时，一定要确保这个对象不是空
- 不用全局变量

题目：Implement strStr

查询一个字符串中另一个字符串第一次出现的位置

```
//hashFuction O(n+m) Robin Kcarp算法
```

算法复杂度理论

1. 时间复杂度 - 核心考察点
2. 空间复杂度 - 次要考察点
3. 编程复杂度 - 能看得懂
4. 思维复杂度 - 能想得出 （可能会对时间和空间做一定的妥协）

时间复杂度

P问题 Polynomial

O(n) ,O(n^2) ,O(n^3)

O(n+m) ,O(√n) ,O(1)

O(logn), O(nlogn)

NP问题 Nondeterministic Polynomial

O(2^n), O(n^n), O(n!)

时间复杂度为O(N)的算法有哪些?

双指针算法、打擂台算法（本质上是枚举）、单调栈算法、单调队列算法等等

双指针算法

三种

- 相向双指针（判断回文串）
- 背向双指针（最长回文串）出现频率极低
- 同向双指针

相向双指针

Reverse型

- 翻转字符串
- 判断回文串

Two Sum型

- 两数之和
- 三数之和

Partition型

- 快速排序
- 颜色排序

题目

Valid Palindrome

//双指针，通常大部分双指针是因为避免使用额外空间

Valid Palindrome II

//贪心法

Two Sum

//HashMap

//排序+双指针 时间复杂度主要受限于排序

快速排序

```
//先整体有序，再局部有序
//小于等于p的放在左边，大于等于p的放在右边
//之所以这么做是避免所有数都一样的极端情况
public class Solution {
    /**
     * @param A: an integer array
     * @return: nothing
     */
    public void sortIntegers(int[] A) {
        // write your code here
        //特殊情况
       if (A == null || A.length == 0){
           return;
       }
        //分治法
       quickSort(A, 0, A.length - 1);
    }

    private void quickSort(int[] A, int start, int end){
        //递归的结束条件
        if (start >= end){
            return;
        }
        int left = start, right = end;
        int pivot = A[(start + end) / 2];
```

```
        //1.pivot = A[start] // pivot = A[end]; 很容易造成极端情况;
        //get value not index\


        //left <= right
        //循环条件能保证即便两边的数字个数不同，也能完成一次整体有序的操作
        while (left <= right){
        //A[left] < pivot not <=
            while (left <= right && A[left] < pivot){
                left++;
            }
        //左指针找一个不属于左侧的
            while (left <= right && A[right] > pivot){
                right--;
            }
            if (left <= right){
                int temp = A[left];
                A[left] = A[right];
                A[right] = temp;
                left++;
                right--;
            }
        //交换左右的两个部分
        }

        quickSort(A, start, right);
        quickSort(A, left, end);
    }
}
```

思路：

先整体有序，再分别局部有序。如此递归

递归排序

思路：先递归到两个数的排序，然后这两个数的排序结果又和它右边的（1、2）个数排序成的结果再排序。

双指针分别从两个排序结果的左边往右读就行

```
public class Solution {
```

```
    /**
     * @param a: an integer array
     * @return: nothing
     */
        public void sortIntegers2(int[] A) {
            // write your code here
        if (A == null || A.length == 0){
            return;
        }
        //排序过程中要用到的辅助数组，因为存在辅助数组的操作，归并排序的时间复杂度会比快排的logn大一些。（快排不极端的情况下）
        int[] temp = new int[A.length];
        //分治法
        mergerSort(A, 0,A.length - 1,temp);
        }

        private void mergerSort(int[] A, int start, int end,int[] temp){
            //递归的退出条件
            if (start >= end){
                return;
            }
            //分治，暴力从中间砍一刀
            mergerSort(A, start, (start + end) >> 1, temp);
            mergerSort(A, ((start + end) >> 1) + 1, end, temp);
            //真正的排序过程
            merge(A, start, end, temp);
        }

        private void merge(int[] A, int start, int end, int[] temp){
         int middle = (start + end) >> 1;
         int leftStart = start;
         int rightStart = ((start + end) >> 1) + 1;
         int index = leftStart;
         while (leftStart <= middle && rightStart <= end){
            if (A[leftStart] < A[rightStart]){
                temp[index++] = A[leftStart++];
            }
            else{
                temp[index++] = A[rightStart++];
            }
```

```
        }

        while (leftStart <= middle){
            temp[index++] = A[leftStart++];
        }
        while (rightStart <= end){
            temp[index++] = A[rightStart++];
        }

        for (int i = start; i <= end; i++){
            A[i] = temp[i];
        }
    }
}
```

Merge是比较严格的nlogn，而quickSort 在最坏情况下可能会达到n^2。

但merge会需要O(n)的额外空间复杂度

Quick无法做到稳定排序

Merge可以做到稳定

这两个都是分治算法(DC)，需要把它分成两个部分，然后去做同样的事情。

Quicksort是先整体有序，再局部有序

mergeSort是中间分开，左边先排然后右边再排。得到两个排好序的数组。然后再merge到一起。

先局部有序，再整体有序。

递归是怎样工作的

斐波那契数列

类似于在推导数学公式时将函数不断展开

递归的缺点

会很耗费空间，当n比较大的时候，会出现栈溢出(stackoverflow)

用盗梦空间来进行理解

```
public class Solution {
    /**
     * @param n: an integer
     * @return: an ineger f(n)
     */
    //1. 递归的定义：函数接受什么样的参数，返回什么样的值，代表什么样的意思
    public int fibonacci(int n) {
        // write your code here
        //3. 递归的出口
        if (n <= 2){
            return n - 1;
        }
        //2. 递归的拆解
        return fibonacci(n-1) + fibonacci(n - 2);
/**     int[] arr = new int[n+1];
        arr[2] = 1;
        for (int i = 3; i <= n; i++){
            arr[i] = arr[i-1] + arr[i-2];
        }
        return arr[n];
**/
    }
}
```

用递归实现二分法（log n），会很少造成时间超时

```
//用递归实现二分法
public class Solution {
    /**
     * @param nums: An integer array sorted in ascending order
     * @param target: An integer
     * @return: An integer
     */
    public int findPosition(int[] nums, int target) {
        // write your code here
        return binarySearch(nums, 0, nums.length - 1, target);
```

```
    }
    private int binarySearch(int[] nums, int start, int end, int
target){
        if (start > end){
            return -1;
        }

        int mid = (start + end) >> 1;

        if (nums[mid] == target){
            return mid;
        }
        if (nums[mid] < target){
            return binarySearch(nums, mid + 1, end, target);
        }

        return binarySearch(nums, start, mid - 1, target);

    }
}
```

二分法

基本原理

二分法相比hash表的优势是使用额外空间较小

步骤

1. 确定区间
2. 找到中点

二分法学习的四重境界

- 高效的查找算法
- 时间复杂度为O(logN)，空间复杂度O(1)
- 一般情况下，二分法用于在某个有序数组上寻找target值
- 二分法利用了减治的算法思想，不属于分治

第一境界

1. 写出不会死循环的二分法

```
start + 1 < end
start + (end -start) / 2
A[mid] ==, <, >
A[start]  A[end] ? target
```

常见痛点:

While循环的退出条件

```
start <= end
start < end
start + 1 < end
```

指针变化到底是哪个

```
start = mid;
start = mid + 1;
start = mid - 1;
```

模板

```
//自写版
public int binarySearch(int[] nums, int target){
    if (nums == null || nums.length == 0){
    return -1;
```

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


//九章提供 带注释
public int binarySearch(int[] nums, int target){
    if (nums == null || nums.length == 0){
    return -1;
    }
    int start = 0, end = nums.length - 1;
    /** 用start + 1 < end 而不是 start < end 的目的是为了避免死循环
        在first position of target 的情况下不会出现死循环
        但是在 last position of target 的情况下会出现死循环
        样例: nums = [1, 1] target = 1;
        为了统一模板，我们就都采用start + 1 < end，就保证不会出现死循环
*/

    while (start + 1 < end){
        //防止越界
        int mid = start + (end - start) / 2;
        // >, <, =，的逻辑先分开写，然后再看看 = 的情况能否合并到其他分支
里
        if (nums[mid] < target){
            //写作 start = mid + 1 也是正确的
            //只是可以偷懒不写，因为不写也没问题，不会影响时间复杂度
            //不写的好处是，万一你不小心写成了 mid - 1 你就错了
            start = mid;
        }else if (nums[mid] == target){
```

```
                end = mid;
            }else{
                //写作 end = mid - 1 也是正确的
                //只是可以偷懒不写，因为不写也没问题，不会影响时间复杂度
                //不写的好处是，万一你不小心写成了 mid + 1 你就错了
                end = mid;
            }
        }
        //因为上面的循环退出条件是 start + 1 < end
        //因此这里循环结束的时候， start 和 end 的关系是相邻关系（1和2，3和4这
种
        //因此想需要再单独判断 start 和 end 这两个数谁是我们想要的答案
        //如果是找 first position of target 就先看 start ，否则就先看end

        if (nums[start] == target){
            return start;
        }
        if (nums[end] == target){
            return end;
        }
        return -1;
    }
```

2. 递归与非递归的权衡
3. 二分的三大痛点
4. 通用的二分模版

第二境界

在排序的数据集上进行二分，一般会给你一个数组，让你找数组中第一个/最后一个满足条件的位置。

常见有序输入集合中的二分法问题（以5为例）

小于5的最大*Index*（或者*value*），插入5 的*Index*

大于等于5 的最小*Index*（或*value*）

任意一个出现5的*Index*

小于等于5 的最大*Index*（或者*value*）

大于5的最小*Indesx*（或*value*），插入5的*Index*

[在大数组中查找](#)

```
//倍增法
//使用到倍增思想的场景(ArrayList in Java)
//动态数组，网络重试
//时间复杂度O(logK)，空间复杂度O(1)
public class Solution {
    /**
     * @param reader: An instance of ArrayReader.
     * @param target: An integer
     * @return: An integer which is the first index of target.
     */
    public int searchBigSortedArray(ArrayReader reader, int
target) {
        // write your code here
        //初始化查找范围为1，代表在前1个数中查找
        //倍增法：如果target在查找范围之外，查找范围翻倍，时间复杂度
 (logK)
        int rangeTotal = 1;
        //start 也可以是rangeTotal / 2，但是我比较习惯保守的写法
        //因为写为 0 也不会影响时间复杂度
        //如果 target 在t前 total 中，则Index范围为[0, rangeTotal -
1]
        //时间复杂度为O(logK)
        while (reader.get(rangeTotal - 1) < target){
            rangeTotal = rangeTotal * 2;
        }

        int start = 0, end = rangeTotal - 1;
        while (start + 1 < end){
            int mid = start + (end - start) / 2;
        //如果(中点值 < target)，target 在右边，丢弃左边
            if (reader.get(mid) < target){
                start = mid;
            }
        //如果(target <= 中点值)，丢弃右边，去左边
        //为什么target = 中点值 的时候不直接返回?
        //因为左边可能存在更靠左的target值
```

```
            else{
                end = mid;
            }
        }
        //先查start再查end的原因是要找到第一次出现的Index
        if (reader.get(start) == target){
            return start;
        }
        if (reader.get(end) == target){
            return end;
        }
        return -1;
    }
}
```

[在排序数组中找最接近的K个数]

给一个目标数 target, 一个非负整数 k, 一个按照升序排列的数组 A。

在A中找与target最接近的k个整数。

返回这k个数并按照与target的接近程度从小到大排序，如果接近程度相当，那么小的数排在前面。

```
public class Solution {
    /**
     * @param A: an integer array
     * @param target: An integer
     * @param k: An integer
     * @return: an integer array
     */
    public int[] kClosestNumbers(int[] A, int target, int k) {
        // write your code here
        if (k == 0){
            return new int[0];
        }
        return sort(A, k, target, findUppperClosest(A, target));

    }

    private int findUppperClosest(int[] A, int target){
        int start = 0, end = A.length - 1;
        while (start + 1 < end){
```

```
            int mid = start + (end - start) / 2;
            if (A[mid] > target){
                end = mid;
            }
            else if (A[mid] < target){
                start = mid;
            }
            else {
                return mid;
            }
        }
        if (A[start] >= target){
            return start;
        }
        if (A[end] >= target){
            return end;
        }
        //可以合并
        return end;
    }

    private int[] sort(int[] A, int k, int target, int index){
        System.out.print("A[index] == "+target);
        int[] temp = new int[k];
        int left = index - 1, right = index;
        int i = 0;
        while (left >= 0 && right < A.length && i < k){
            if (target - A[left] > A[right] - target){
                temp[i++] = A[right++];
            }
            else{
                temp[i++] = A[left--];
            }
        }
        while (i < k && left < 0){
                temp[i++] = A[right++];
            }

        while (i < k && right >= A.length){
                temp[i++] = A[left--];
        }
        return temp;
```

```
    }

}


public class Solution {
    /**
     * @param A: an integer array
     * @param target: An integer
     * @param k: An integer
     * @return: an integer array
     */
    public int[] kClosestNumbers(int[] A, int target, int k) {
        // write your code here
        int left = findLowerClosest(A, target);
        int right = left + 1;

        int[] results = new int[k];
        for(int i  = 0; i < k; i++){
            //如果左边更接近，选左边
            if (isLeftCloser(A, target, left, right)){
                results[i] = A[left];
                left--;
            }else{
                results[i] = A[right];
                right++;
            }
        }
        return results;

    }

    private boolean isLeftCloser(int[] A, int target, int left,
int right){
        //如果左边已耗尽，返回false
        if (left < 0){
            return false;
        }
        //如果右边已耗尽，返回true
        if (right >= A.length){
            return true;
```

```
        }
        //为什么有等号？如果左右距离相等，选左边
        return target - A[left] <= A[right] - target;
    }
        //找到比target小的最又一个数
    private int findLowerClosest(int[] A, int target){
        int start = 0, end = A.length - 1;
        while (start + 1 < end){
            int mid = start + (end - start) / 2;
            //如果 mid < target，答案在右边，丢掉左边。
            if (A[mid] < target){
                start = mid;
            }
            //如果 mid >= target，  答案在左边，丢掉右边
            else{
                end = mid;
            }

        }
        //因为要找最右数，所以这里要先判断end
        if (A[end] < target){
            return end;
        }
        //如果end不行，再判断左边的start

        if (A[start] < target){
            return start;
        }
        //找不到，说明所有的数都 >= target
        return -1;
    }
}
```

小结 --在有序的输入集合中的二分查找

本质

在有序的数组中寻找一个跟Target有关的Index或值

时间复杂度

一次二分法是 O(logN)

关键词

array, target, equal or close to target的词(e.g.，小于n的最大值），O(NlogN)

套路

如果两个数和的target，固定一个值，另一个值二分查找

如果没有排序，可以先排序再求解

第三境界 -- 在未排序的数据集上进行二分

并无法找到一个条件，形成XXOO的模型

但可以根据判断，保留下有解的 那一半或者去掉无解的一半

[山脉序列中的最大值]

给 n 个整数的山脉数组，即先增后减的序列，找到山顶（最大值）。

```
public class Solution {
    /**
     * @param nums: a mountain sequence which increase firstly
and then decrease
     * @return: then mountain top
     */
    public int mountainSequence(int[] nums) {
        // write your code here
        if (nums == null || nums.length == 0){
            return -1;
        }
        int start = 0, end = nums.length - 1;
        while (start + 1 < end){
            int mid = start + (end - start) / 2;
            //mid + 1一定不会越界
            if (nums[mid] > nums[mid+1]){
                end = mid;
            }
            else {
                start = mid;
            }
        }
        return Math.max(nums[start], nums[end]);
```

```
    }
}
```

寻找旋转排序数组中的最小值

```
public class Solution {
    /**
     * @param nums: a rotated sorted array
     * @return: the minimum number in the array
     */
    public int findMin(int[] nums) {
        // write your code here
        if (nums == null || nums.length == 0){
            return -1;
        }
        int start = 0, end = nums.length - 1;
        while (start + 1 < end){
            int mid = start + (end - start) / 2;
            if (nums[mid] > nums[end]){
                start = mid;
            }
            else if (nums[mid] < nums[end]){
                end = mid;
            }
            else{
                return nums[end];
            }
        }
        return Math.min(nums[start], nums[end]);
    }
}
```

搜索旋转排序数组

给定一个有序数组，但是数组以某个元素作为支点进行了旋转(比如，0 1 2 4 5 6 7 可能成为4 5 6 7 0 1 2)。

给定一个目标值target进行搜索，如果在数组中找到目标值返回数组中的索引位置，否则返回-1。

你可以假设数组中不存在重复的元素。

```
public class Solution {
    /**
     * @param A: an integer rotated sorted array
     * @param target: an integer to be searched
     * @return: an integer
     */
    public int search(int[] A, int target) {
        // write your code here
        if (A == null || A.length == 0){
            return -1;
        }
        int start = 0, end = A.length - 1;
        while (start + 1 < end){
            int mid = start + (end - start) / 2;
            if (A[mid] > A[end]){
                start = mid;
            }
            else{
                end = mid;
            }
        }
        int minIndex = A[start] > A[end] ? end : start;

        if (target <= A[A.length - 1]){
            start = minIndex;
            end = A.length - 1;
            System.out.println(" target = " + target + " minIndex
= " + minIndex);
            while (start + 1 < end){
                int mid = start + (end - start) / 2;
                if (target > A[mid]){
                    start = mid;
                }
                else{
                    System.out.println("1 start = " + start + "
end = " + end);
                    end = mid;
                }
            }
            System.out.println("2 start = " + start + " end = " +
end);
            if (A[start] == target){
```

```
                return start;
            }
            if (A[end] == target){
                return end;
            }
        }

        if (target >= A[A.length - 1]){
            start = 0;
            end = minIndex;
            while (start + 1 < end){
                int mid = start + (end - start) / 2;
                if (A[mid] < target){
                    start = mid;
                }
                else{
                    end = mid;
                }
            }
            if (A[start] == target){
                return start;
            }
            if (A[end] == target){
                return end;
            }
        }
        return -1;
    }
}
```

寻找峰值

给定一个整数数组(size为n)，其具有以下特点：

相邻位置的数字是不同的

A[0] < A[1] 并且 A[n - 2] > A[n - 1]

假定P是峰值的位置则满足A[P] > A[P-1]且A[P] > A[P+1]，返回数组中任意一个峰值的位置。

```
public class Solution {
    /**
```

```
     * @param A: An integers array.
     * @return: return any of peek positions.
     */
    public int findPeak(int[] A) {
        // write your code here
        //peak不可能在两端，所以在[1, A.lenght - 1]范围内寻找
        int start = 1, end = A.length - 2;
        while (start + 1 < end){
            int mid = start + (end - start) / 2;
            //如果mid向左上方倾斜，选左半边
            if (A[mid] < A[mid - 1]){
                end = mid;
            }
            //如果mid向右上方倾斜，选右半边
            else if (A[mid] < A[mid + 1]){
                start = mid;
            }

            //如果mid为peak，返回
            else {
                return mid;
            }
        }
        return A[start] < A[end] ? end : start;

    }
}
```

第四境界 -- 在答案集上进行二分

[木材加工]

有一些原木，现在想把这些木头切割成一些长度相同的小段木头，需要得到的小段的数目至少为 k。

给定L和k，你需要计算能够得到的小段木头的最大长度。

```
public class Solution {
    /**
     * @param L: Given n pieces of wood with length L[i]
     * @param k: An integer
```

```
     * @return: The maximum length of the small pieces
     */
    public int woodCut(int[] L, int k) {
        // write your code here
        //特殊情况处理
        if (L == null || L.length == 0){
            return 0;
        }
        //初始化start和end, end = min{max in L, sum L / k}
        int start = 1, end = 0, maxLen = 0;
        long sum = 0;
        for (int l : L){
            end = Math.max(end, l);
            sum += l;
        }
        end = (int)Math.min(end, sum / k);

        //如果end < 1, 则不可能完成任务, 返回0
        if (end < 1){
            return 0;
        }
        while (start + 1 < end){
            int mid = start + (end - start) / 2;
            //长度为mid的木头总数 >= 目标总数, 继续增长木头长度, 选右边
            if (getCount(L, mid) >= k){
                start = mid;
            }
            //长度为mid的木头总数 < 目标总数, 继续缩短木头长度, 选左边
            else {
                end = mid;
            }
        }
        //因为之前排除了无解的情况, 所以这里一定有解, 非start即end
        //如果end符合要求, 首选end (因为end更长) , 否则选start
        return (getCount(L, end) >= k) ? end :start;
    }

    private int getCount(int[] L, int len){
        int cnt = 0;
        for (int item : L){
            cnt += item / len;
        }
```

```
            return cnt;
        }
}
```

小结 -- 在答案集上进行二分

本质： 求满足某条件的最大值或者最小值

最终

结果是个有限的集合

每个结果都有一个对应的映射

结果集合跟映射集合正相关或者负相关

可以通过在映射集合上进行二分，从而实现对结果集合的二分

关键词： Array，有限可能解，映射，正负相关，O(NlogN)

BFS的三种实现方法

单队列

```
/**
 * Definition of TreeNode:
 * public class TreeNode {
 *     public int val;
 *     public TreeNode left, right;
 *     public TreeNode(int val) {
 *         this.val = val;
 *         this.left = this.right = null;
 *     }
 * }
 */

public class Solution {
    /**
```

```
     * @param root: A Tree
     * @return: Level order a list of lists of integer
     */
    public List<List<Integer>> levelOrder(TreeNode root) {
        // write your code here
        List result = new ArrayList<>();

        if (root == null){
            return result;
        }

        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.offer(root);

        while (!queue.isEmpty()){
            ArrayList<Integer> Level = new ArrayList<Integer>;
            int size = queue.size();
            for (int i = 0; i < size; i++){
                TreeNode head = queue.poll();
                level.add(head.val);
                if (head.left != null){
                    queue.offer(head.left);
                }
                if (head.right != null){
                    queue.offer(head.right);
                }
            }
            result.add(level);
        }
        return result;
    }
}
```

2022.4.5

双队列

DummyNode