**二分法总结：**

|  |
| --- |
| Base Line: 排过序的数组，我们的目标是找到某个特定的元素或者是找到某段区间，**注意有些变形的题**，如：  1: count the occurrence of one element in a sorted array. Like, int []arr,int target  一般的思路，利用二分法找到k,但是怎么处理我们找到了 arr[mid] == k,一般的做法是从这个元素为中心，向两边扩展，但是如果遇到了特殊的情况，如果全部的元素如全部都是k,那么复杂度就是O(n),显然这不是最优解，面试官不会满足。  改进的算法，我们利用两遍二分查找，分别是getFirstIndex(arr,k) 和getLastIndex(num,k), 分别是首个出现的k和最后出现的k, 但是这个二分查找法有些特殊，例如 getFirstIndex, 如果arr[mid] ==k,看他左边是不是k,如果是k,二分查找左边，如果不是，返回mid,  2:find element in a rotated sorted  这题也是二分法的变形，主要分析二分法的每次减半的条件，因为是rotate之后的数组，则这个数组又两个递增的子数组，如[5,6,7,8,1,2,3,4],我们需要判断是在左边的子数组还是右边的数组。  但是特别注意重复的元素，如[9,5,6,7,8,9,9,9,9,9]这种edge case.  此题的另一种情况：查找minimum in rotated sorted array |

**二分法插入的有关总结：向有序的数组内插入元素**

|  |
| --- |
| 在有序的数组内插入元素使用二分法，复杂度是O(logn),例如：  数组a[0],a[1],a[2],a[3]…a[m],a[m+1],其中a[0]~a[m]是有序的，将a[m+1]插入其中，代码的一般的架构是：  int low =0;  int high = m;  int mid;  while(low <= high)  {  mid = (low + high)/2;  if(a[mid] > a[m+1]) high = mid-1;  else low = mid+1;  }  则**最后插入的index 是low,或者是high+1,**分析如下：  1: 最后剩下两个元素a[i],a[i+1] => low =i,high=i+1 ,mid=i  1: a[mid]>a[m] => high=mid-1=i-1,low = i =>插入的index是i  2: a[mid]<=a[m] => low=mid+1=i+1,high=i+1 => 最后剩下一个元素的情况  2: 最后剩下一个元素a[i] => low = i, high =i,mid=i  1: a[mid] >a[m] => high = mid-1=i-1, low=i =>插入的index是i  2: a[mid]<=a[m] => low = mid+1=i+1, high = i => 插入的index是i+1 |

**关于递归的几点小总结**：

|  |
| --- |
| 递归有时候可以很饶人，如：  1: permutation的dfs解法：即将一个字符串的拆分成第一个字符和剩下的字符，  剩下的字符运用递归的思想，但是注意一点的是再执行完当前递归后，要将原对象改成递归前的状态比如这题：  2: 在递归的时候也要考虑传入的值是否是immutable的，如传入的如果是int,Boolean,String… 这样的递归是没有用的，因为java中pass-by-value的原则是的下次递归在改变输入的参数时就重新创建了对象，如balanced binary tree: |
| /\*\*  \* 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: The root of binary tree.  \* @return: True if this Binary tree is Balanced, or false.  \*/  public boolean isBalanced(TreeNode root)  {  // if(root == null) return true;    int height;  return helper(root,height);  }    //此种方法类似与树的后续遍历，为了得到一个节点的高度，我们需要知道左边的树的高度，右边树的高度，  //然后才是当前节点的高度。  public boolean helper(TreeNode root,int height)  {  //edge case  if(root == null)  {  height = 0;  return true;  }      // java中不能这样传递int类型实现递归，因为是pass-by-value.c++可以用指针  int leftHeight;  int rightHeight;  if(helper(root.left,leftHeight) && helper(root.right,rightHeight))  {  if(Math.abs(leftHeight - rightHeight) <= 1)  {  height = Math.max(leftHeight,rightHeight) + 1;  return true;  }else return false;  }  return false;  }  } |

|  |
| --- |
| 但是c++可以用指针实现这种递归，如果要使用java来实现这种效果，我们需要自定义一种数据类型，包括ResultType: |
| class **ResultType** {  public boolean isBalanced;  public int maxDepth;  public ResultType(boolean isBalanced, int maxDepth) {  this.isBalanced = isBalanced;  this.maxDepth = maxDepth;  }  }  public class Solution {  /\*\*  \* @param root: The root of binary tree.  \* @return: True if this Binary tree is Balanced, or false.  \*/  public boolean isBalanced(TreeNode root) {  return helper(root).isBalanced;  }    private ResultType helper(TreeNode root) {  if (root == null) {  return **new ResultType(true, 0)**;  }    ResultType left = helper(root.left);  ResultType right = helper(root.right);    // subtree not balance  if (!left.isBalanced || !right.isBalanced) {  return new ResultType(false, -1);  }    // root not balance  if (Math.abs(left.maxDepth - right.maxDepth) > 1) {  return new ResultType(false, -1);  }    return new ResultType(true, Math.max(left.maxDepth, right.maxDepth) + 1);  }  } |

**关于有序的二位数组的查找问题**：

|  |
| --- |
| 有序的二维数组例如每一行从左到右递增，每一列从上到下递增，这样的二维数组如：  1 2 8 9  1 4 9 12  4 7 10 13  6 8 11 15  这类查找的问题，key是从右上角开始查找并且排除，如果右上角的元素>target,col--,如果<target,row++. |

**关于函数调用栈和堆中栈的比较**：

|  |
| --- |
| 对于一般情况，函数调用栈（递归）的过程就是一个压栈和出栈的过程，但是容易引起**函数调用栈的溢出，**这个时候我们可以用堆中建立stack数据结构的方法来对函数进行调用，因为堆中内存比较大，不容易引起溢出，例如，如果要从尾到头打印一个链表的数值，我们可以用递归，也可以先用stack存储，再依次排出，如果链表很长的话，递归可能溢出，可以考虑用stack。  递归的缺点：  1:由于是函数调用本身，函数调用是有时间和空间的消耗的，如分配函数参数，返回地址和临时变量…往栈里压数据和拿数据都需要时间  2:递归很多计算都是重复的  3:最重要的，容易引起栈溢出 |

**关于位运算的一些总结**：

|  |
| --- |
| 1: 位运算比乘，除这些运算要快上很多  2:移位要特别注意负数的情况和溢出的情况，防止死循环  3: int 的范围是：  正数：0x7FFFFFFF(2^31 – 1= 2147483647);  负数：0x80000000(-2^31=- 2147483648)  0x80000001(-2^31+1=- 2147483647);  例如：统计一个树二进制表达式中1的个数  1:最容易想到的方法：每次右移移位，与1与统计每一位，但是这种情况要防止负数死循环  2: 将原来数减1，然后与原数与，就将右边最后一位1变为了0，然后count++，继续将统计新的数  int numberOf(int n)  {  int count = 0;  while(n != 0)  {  count ++;  n=(n-1)&n;  }  } |

**关于大数运算的一些处理方法和情景**：

|  |
| --- |
| 如果面试题是关于n的整数并且没有限定n的取值范围，或者输入是任意大小的整数，那么这个题目很有可能是需要考虑大数问题的，**字符串**是一个简单的，有效的表示大数的方法：  例如：打印1到最大n位数，如n=3,打印1～999  处理这种大数问题，可以使用字符串，也可以使用int数组来处理，但是字符串实际上是字符数组，比int数组更加省空间，但是用1 byte 来保存0～9还是不够有效率，有什么好的办法吗？  注意其中几个小细节：  1:在打印的时候，我们需要把前面的0去掉，如98而不是098，  2:判断什么时候是最后一位  类似的例子：  定义一个函数，实现两个整数的加法。 |

**关于anagram的一些总结**：

|  |
| --- |
| Anagram的题目一般可以结合hashmap来解决，但是 hashmap的key和value的选择一般是重点，一般key值我们可以使用原string排序之后的新string,这样anagram的字符串就一定会拥有相同的key.  排序的算法可能不是最优的，我们也可以自己写一个hashCode方法，来保证anagrams拥有相同的hashCode. |

**关于hashmap中key的选择问题**：

|  |
| --- |
| 有时候用hashmap 来解答问题十分巧妙，因为可以将一类的类型的数据封装到一起，并且可以在O(1)时间内访问，但是难点就是如何选择key,  1:可以将其转换成string类型，因为string本来就是immutable的，例如leetcode245 Group Shifted Strings 此题，难点就是如何放置shift的pattern：   1. **for**(**int** i = 0; i < strings.length; i++) { 2. StringBuffer sb = **new** StringBuffer(); 3. **for**(**int** j = 0; j < strings[i].length(); j++) { 4. sb.append(Integer.toString(((strings[i].charAt(j) - strings[i].charAt(0)) + 26) % 26)); 5. sb.append(" "); 6. } 7. String shift = sb.toString(); 8. **if**(d.containsKey(shift)) { 9. d.get(shift).add(strings[i]); 10. } **else** { 11. List<String> l = **new** ArrayList<>(); 12. l.add(strings[i]); 13. d.put(shift, l); 14. } 15. } |

**关于LinkedList 的常用的一些操作（单向链表）**：

|  |
| --- |
| 1: 找出中间节点，  如： 1 2 3 4 5 6 => 3  1 2 3 4 5 6 7 =>4  使用快慢指针：  private ListNode findMiddle(ListNode head) {  if (head == null) {  return null;  }  ListNode slow = head, fast = head.next;  //注意终止条件，因为每一次我们不能保证fast指针是否为空，因此需要先判断fast然后再判断fast.next  while (fast != null && fast.next != null) {  slow = slow.next;  fast = fast.next.next;  }  return slow;  } |

|  |
| --- |
| 2: reverse 一个链表  花一个示意图就好理解了，  1 -> 2 -> 3… 当我们处理节点2 时：  1 <- 2 3…  prev cur next  因此我们需要三个节点分别是prev, cur,next分别记录上一个节点，现在要处理的节点以及下一步要处理的节点，终止条件时判断next是否为空,：  private ListNode reverse(ListNode head) {  ListNode prev = null;  while (head != null) {  ListNode temp = head.next;  head.next = prev;  prev = head;  head = temp;  }  return prev;  } |

**关于二叉树的层次遍历(level order travesal)**：

|  |
| --- |
| 1:首先，应该使用queue的数据结构，因为我们是从上到下，从左到右，我们每次poll出新值，就append 相应的left和right节点。  如果需要将每层的结果存储到一个list上，如：  1 <---  / \  2 3 <---  \ \  5 4 <---  [[1],[2,3],[5,4]],  我们需要记录每一层的结束标志位，可以将每一层的最后一个元素后加null,如  [1,null,2,3,null,5,4,null]  类似思想的问题(Binary Tree Right Side View)(leetcode 199):  此题记录每层的最右边的元素，同样使用上述queue的存储方式。  类似的题目还有 invert binary tree 用非递归的方法，都要借助queue来完成  2:第二种方法要考虑hashmap + dfs, key是depth,value是对应的节点或者arraylist,上 |

**关于递归的edge case 问题**：

|  |
| --- |
| 递归的edge case是递归的难点，需要注意，当我们判断edge case 需要对多种情况查询是否满足edge case时，可以使用if语句来对每一个edge case 分别不同的判断，一个典型的例子，**StringMixture** |

**关于**类**字符串转整形的题目**：string->int, string->float…

|  |
| --- |
| 这种题型考验对不同test case的考虑，如：   越界问题？=> 使用double或者long,如果仍然越界，可以考虑用数组或者arraylist   正负号问题？=> 刚开始的时候判断   空格问题？ => 使用trim() 去除头和尾部的空格   精度问题? => 小数点后面的数 |

**关于第k个顺序统计量**：快排的思想

|  |
| --- |
| 一个n个元素组成的集合中，第k个顺序统计量指的是该集合中第k小的元素，我们要讨论的是如何在线性时间里找出一个数组的第k个顺序统计量.  核心步骤：将数组low,high脚标下的数组进行partition, 算法中i代表比pivot小的指针，i 之后是比pivot大的元素。j 是我们遍历数组时的指针，注意返回类型是pivot的index.  int Partition(int A[], int low, int high)  {  int pivot = A[low];  int i = low;  for(int j=low+1; j<=high; ++j)  {  if(A[j] <= pivot)  {  ++i;  swap(A[i], A[j]);  }  }  swap(A[i], A[low]);  return i;  }  采用randomized\_select来选择pivot:  int Randomized\_Partition(int A[], int low, int high)  {  srand(time(NULL));  int i = rand() % (high+1);  swap(A[low], A[i]);  return Partition(A, low, high);  }  最后再利用partition来递归寻找第k小的元素：  int Randomized\_Select(int A[], int p, int q, int i)  {  if(p == q)  return A[p];  int r = Randomized\_Partition(A, p, q);  int k = r-p+1;  if(i == k)  return A[r];  if(i < k)  return Randomized\_Select(A, p, r-1, i);  else  return Randomized\_Select(A, r+1, q, i-k);  }  注意：虽然最坏情况Θ(n^2)出现的概率非常非常小，但是不代表它不会出现。这里就介绍一个非同一般的算法，以保证在最坏情况下也能达到线性时间。  基本步骤：   1. 将输入数组的n个元素划分为n/5（上取整）组，每组5个元素，且至多只有一个组有剩下的n%5个元素组成。 2. 寻找每个组织中中位数。首先对每组中的元素（至多为5个）进行插入排序，然后从排序后的序列中选择出中位数。 3. 对第2步中找出的n/5（上取整）个中位数，递归调用SELECT以找出其中位数x。（如果是偶数取下中位数） 4. 调用PARTITION过程，按照中位数x对输入数组进行划分。确定中位数x的位置k。   如果i=k，则返回x。否则，如果i < k，则在地区间递归调用SELECT以找出第i小的元素，若干i > k，则在高区找第(i-k)个最小元素。 |

**关于三种swap的方法**：

|  |  |  |
| --- | --- | --- |
| 1: 使用暂存空间   |  |  | | --- | --- | |  | int tmp = a;  a = b;  b = tmp |   2: +和-操作  a = a + b;  b= a –b;  a =a-b;  3:XOR  a=a^b;  b=a^b;  a=a^b;  **注意**：如果两个变量对应的同一内存单元，则经过两次加减或异或操作，内存单元的值已经变为了0，因而不能实现变量值交换。**所以当需要交换值的变量可能是同一变量时，必须使用第三变量实现交换**，否则会对变量清零。 |

**关于位图排序(bit map)**：

|  |
| --- |
| 优点：这种排序方法是通过牺牲空间效率来追求时间效率（线性时间）以达到时间-空间折中与双赢的目的  缺点：这种排序方法对输入的数据是有比较严格的要求(数据不能重复，大致知道数据的范围)，而且要求输入的数据需要是均匀的且密集的，不然的话空间效率就很低。  注意：位图排序是使用一个二进制位而不是一个整数来表示0或1，这样可以大大地减少所需要的内存空间。使用位图排序的前提是要知道待排序序列中的最大数。位图排序的缺点是有些数没有出现过，仍要为其保留一个位。故位图排序比较适合关键字密集的序列，例如一个城市的电话号码。  字节位置=数据/32;(采用位运算即右移5位)  位位置=数据%32;(采用位运算即跟0X1F进行与操作)。  http://pic002.cnblogs.com/images/2011/288799/2011101911100137.jpg |

**关于快排(quick sort)**：

|  |
| --- |
| \* 几个注意点：  \* 1: 需要三个函数：  \* quickSort(int [] arr) -> 调用函数，接受排序的数组，返回可以时空，也可以时一个数组  \* sortHelper(int[] arr,int left,int right) -> 排序调用时的递归函数，三个参数，包括左索引和右索引  \* partition(int[] arr,int left,int right) -> 将一个范围分成两部分，注意要返回分完的index,因此返回类型是int |

**关于堆排(heap sort)**：

|  |
| --- |
| 几个注意点：  \* 1: 三个基本步骤  \* 1: 从含有孩子节点开始向上依次建立最大堆(max heap)，即将原数组变成堆  \* 2: 取出堆中最大元素，放在末尾，注意此时堆的大小也改变了饿  \* 3: 当堆的大小变为1时，代表排序结束  \* 2: 注意要维持一个heapSize的变量，因为heapSize 是我们heaplifyDown（）的主要判定条件  \* 3: 在heaplifyDown()方法内，先判断当前节点是否含有左孩子和右孩子，如果有，需要将三者之中的最大值放在parent 节点上 |

**关于归并排序(merge sort)**：

|  |
| --- |
| /\*几个注意点：  \* 1: 可以写成两个函数或者三个函数，以下就是两个函数的版本  \* 2: mergeSort()中的 left和right都是inclusive的，  \* 3:mergePart()里需要创建tmp数组来临时保存合并的结果，因此归并排序空间复杂度是O(n)的  \*/ |