作者：星\_\_尘  
链接：<https://www.nowcoder.com/discuss/428158>  
来源：牛客网

**前几天发了字节面试算法100题，这一篇给牛友们整理了答案，楼主整理过程中又对题目做了归类，所以和之前的题目顺序不完全一致，这一篇包含了100题中动态规划和贪心，数组，排序，栈+队列+链表，二叉树相关的算法题题解。一些不易分类的算法题，场景题和智力题答案下一篇发出来。**

整理不易，感觉有用的话点个赞呀，亲！😁😁（牛客网的编辑器插入代码必须一个一个来，楼主从自己的笔记中一点一点复制过来，眼睛要瞎了😭😭😭）

很多题目的题解参照了牛客和力扣中的优秀题解，一些题目中也贴出了题解链接。楼主能力一般，水平有限，难免有错误和遗漏，有问题的地方欢迎牛友们在评论区指正，有更好的解法也可在评论区分享，万分感谢。

**算法题**

**动态规划和贪心**

* **算法题：买卖股票的最佳时机（只能有一次买卖，可以最多两次买卖，不限次数）**

股票问题通用解法详解参考：<https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock/solution/yi-ge-fang-fa-tuan-mie-6-dao-gu-piao-wen-ti-by-l-3/>

最复杂的情况是限制交易k次，状态转移方程无法化简，其他情况均可化简为二维或一维动态规划。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | //一次买卖  public class Solution {      public int buy(int[] price) {          int n = price.length;          int ans = 0;          int[][] dp = new int[n][2];          dp[0][0] = 0;          dp[0][1] = Integer.MIN\_VALUE;          for(int i = 1;i < n; i++){              dp[i][0] = Math.max(dp[i - 1][0],dp[i - 1][1] + price[i]);              dp[i][1] = Math.max(dp[i - 1][1],- price[i]);          }          return dp[i][0];      }  }  优化：只与相邻的一个状态有关，那么可以只记录前一个状态，不使用数组，空间降到O(1)  public class Solution {      public int buy(int[] price) {          int n = price.length;          int ans = 0;          int dp\_i\_0 = 0;          int dp\_i\_1 = Integer.MIN\_VALUE;          for(int i = 1;i < n; i++){              dp\_i\_0 = Math.max(dp\_i\_0,dp\_i\_1 + price[i]);              dp\_i\_1 = Math.max(dp\_i\_1,- price[i]);          }          return dp\_i\_0;      }  }  //不限制次数：  dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])  dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])    //有一天冷却期  dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])  dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i])  解释：第 i 天选择 buy 的时候，要从 i-2 的状态转移，而不是 i-1 。    //有服务费  dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])  dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i] - fee)  解释：相当于买入股票的价格升高了。  在第一个式子里减也是一样的，相当于卖出股票的价格减小了。    //2次买卖  原始的动态转移方程，没有可化简的地方  dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])  dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])    int max\_k = 2;  int[][][] dp = new int[n][max\_k + 1][2];  for (int i = 0; i < n; i++) {      for (int k = max\_k; k >= 1; k--) {          if (i - 1 == -1) {              /\* 处理 base case \*/              dp[i][k][0] = 0;              dp[i][k][1] = -prices[i];              continue;          }          dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);          dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);      }  }  // 穷举了 n × max\_k × 2 个状态，正确。  return dp[n - 1][max\_k][0]; |

* **剑指原题，剪绳子。**

1.动态规划：每次剪一刀之后，剩余部分还可以继续剪，那么就是计算出所有可能的情况，取最大值。自底向上改善递归从上而下重复计算的问题。

时间复杂度O(N^2),空间复杂度O（N）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | public class Solution {      public int cutRope(int target) {          if(target == 2)              return 1;          if(target == 3)              return 2;          if(target == 4)              return 4;          int[] dp = new int[target + 1];            /\*          下面3行是n>=4的情况，跟n<=3不同，4可以分很多段，比如分成1、3，          这里的3可以不需要再分了，因为3分段最大才2，不分就是3。记录最大的。           \*/          dp[1]=1;          dp[2]=2;          dp[3]=3;          //用来记录最大值          int res = 0;          for(int i = 4;i <= target; i++){              for(int j = 1;j <= i/2;j++){                  res = Math.max(res,dp[j]\*dp[i - j]);              }              dp[i] = res;          }          return dp[target];      }  } |

2.贪心算法：可以证明，每段长度为3是最大乘积。

证明过程请参照《剑指offer》或牛客剑指offer题解中的回答：<https://www.nowcoder.com/questionTerminal/57d85990ba5b440ab888fc72b0751bf8?f=discussion>

时间复杂度O(logN),因为乘方运算的时间复杂度是logN。当数据特别大时，只能使用贪心算法，因为动态规划枚举每个状态需要大量的空间。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class Solution {      public int cutRope(int target) {          if(target==2){              return 1;          }else if(target==3){              return 2;          }      //pow(n,m)是求n的m次方          if(target%3==0){              return (int)Math.pow(3,target/3);          }else if(target%3==1){              return 4\*(int)Math.pow(3,target/3-1);          }else {              return 2\*(int)Math.pow(3,target/3);          }      }  } |

* **算法：接雨水（leetcode 42）**

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。



上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

示例:

输入: [0,1,0,2,1,0,1,3,2,1,2,1]

输出: 6

思路：

对于每一列来说，他能存的雨水量是他左边最高墙和右边最高墙中较低的那堵墙的高度减去自身墙的高度。所以可以用数组记录每列左右最高墙的高度，然后计算每一列可以存的雨水量。

动态规划：时间复杂度O(N)，空间复杂度O(N)。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class Solution {      public int trap(int[] height) {          int len = height.length;          if(len == 0 || len == 1) return 0;          int[] left = new int[len];          int[] right = new int[len];          left[0] = height[0];          right[len - 2] = height[len - 1];          for(int i = 1;i < len - 1;i++){              left[i] = Math.max(height[i - 1],left[i - 1]);          }          for(int i = len - 2;i >= 0;i--){              right[i] = Math.max(height[i + 1],right[i + 1]);          }          int sum = 0;          for(int i = 1; i < len - 1;i++){              int min = Math.min(right[i],left[i]);              if(min > height[i])                  sum = sum + (min - height[i]);          }          return sum;      }  } |

动态规划空间优化：

因为left和right数组中每个元素我们只使用一次，所以没必要保存整个数组，左右只保存一个值就可以。由于left和right遍历时方向不一致，left和计算时的遍历方向一致，很容易更新，而right依然要使用数组来存储，这样空间复杂度仍为O(n)。

如果当前列左侧最高墙的高度比右侧最高墙的高度小，那么能装多少水由左侧决定，也就是说min变量就等于左侧最高墙的高度，反之，右侧最高墙的高度小，那么min就等于右侧最高墙的高度。那么我们在遍历过程中就可以先比较左右最高墙的高度，选两侧的较小值计算雨水量加入sum。（这里我可能说的不太清楚，如果无法理解请参照力扣中的题解：<https://leetcode-cn.com/problems/trapping-rain-water/solution/>）

时间复杂度O(N)，空间复杂度O(1)。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class Solution {      public int trap(int[] height) {          int len = height.length;          if(len == 0 || len == 1) return 0;          int maxLeft = 0;          int maxRight = 0;          //加左右指针          int left = 1;          int right = len - 2;          int sum = 0;          for(int i = 1; i < len - 1;i++){              if(height[left - 1] < height[right + 1]){                  maxLeft = Math.max(maxLeft,height[left - 1]);                  if(maxLeft > height[left])                      sum +=  maxLeft - height[left];                  left++;              }else{                  maxRight = Math.max(maxRight,height[right + 1]);                  if(maxRight > height[right])                      sum += maxRight - height[right];                  right--;              }          }          return sum;      }  } |

* **柠檬水找零（LeetCode860）**

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，（按账单 bills 支付的顺序）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。你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 5 美元。

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 。

思路：尽可能有大面值找零，也就是能用10元找零就不用2个5元。是贪心算法思想的体现。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class Solution {      public boolean lemonadeChange(int[] bills) {          int five = 0,ten = 0;          for(int value : bills){              if(value == 5){                  five++;              }else if(value == 10){                  if(five == 0) return false;                  five--;                  ten++;              }else{                  if(ten >= 1 && five >= 1){                      ten--;                      five--;                  }else if(five >= 3){                      five = five - 3;                  }else{                      return false;                  }              }          }          return true;      }  } |

**数组**

**双指针遍历：解决有序数组的问题**

* **排序数组，平方后，数组当中有多少不同的数字（相同算一个）。**

如果不是排序数组，可以使用hashset来保存数字的平方，重复就存不进去，那么最后就可以直接返回set的大小size即可。时间空间复杂度都是O（n)。

双指针遍历：这里是排序数组，既然有重复，肯定是有负数，0,1这些数字。平方后两头大，中间小，可以用首尾指针共同向中间扫描，扫描时去掉重复元素，同时用一个sum来记录有多少个不同数字。

时间复杂度O(N)，空间复杂度O(1)。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public class Solution {      public int diffSquareNum(int nums[]) {          int n = nums.length;          if(n == 0 || nums == null){              return 0;          }          int sum = 0;          int left = 0;          int right = n - 1;          while(left <= right){              if(nums[left] + nums[right] == 0){                  sum++;                  int temp = nums[left];                  //这里开始去掉后面重复的数字                  while(left <= right && nums[left] == temp)                      left++;                  while(left <= right && nums[right] == -temp)                      right--;              }              else if(nums[left] + nums[right] < 0){                  sum++;                  int temp = nums[left];                  while(left <= right && nums[left] == temp)                      left++;              }              else {                  sum++;                  int temp = nums[right];                  while(left <= right && nums[right] == temp)                      right--;              }          }          return sum;      }  } |

* **一个数据先递增再递减，找出数组不重复的个数，比如 [1, 3, 9, 1]，结果为3，不能使用额外空间，复杂度o(n)**

与上面一题一模一样，双指针从两头向中间逼近。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | public class Solution {      public int diffnum(int[] nums){          int n = nums.length;          if(n == 0 || nums == null){              return 0;          }          int left = 0;          int right = n - 1;          int sum = 0;          while(left <= right){              if(nums[left] == nums[right]){                  sum++;                  int temp = nums[left];                  while(left <= right && nums[right] == temp)                      right--;                  while(left <= right && nums[left] == temp)                      left++;              }              else if(nums[left] < nums[right]){                  sum++;                  int temp = nums[left];                  while(left <= right && nums[left] == temp)                      left++;              }              else{                  sum++;                  int temp = nums[right];                  while(left <= right && nums[right] == temp)                      right--;              }          }          return sum;      }  } |

* **递增数组，找出和为k的数对**

双指针遍历：用头尾两个指针，分别开始遍历，两个数字和大于k时，右指针向前移动，小于k时左指针向后移动。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class Solution{      public ArrayList findPair(int[] nums,int k){          int n = nums.length;          int i = 0;          int j = n - 1;          ArrayList<Integer> list = new ArrayList<>();          while(i < j){              if(nums[i] + nums[j] < k){                  i++;              }else if(num[i] + nums[j] > k){                  j--;              }else{                  list.add(nums[i]);                  list.add(nums[j]);              }          }          return list;      }  } |

* **给出一个数组nums，一个值k，找出数组中的两个下标 i，j 使得 nums[i] + nums[j] = k.**

这个题目跟上面一题的区别就是不是有序数组，那么解题思路就可以是排序+双指针遍历，时间复杂度就因为排序升为O(NlogN)。

对于这个无序数组另一种解决办法是使用HashMap，数字作为键，下标作为值存入hashmap，然后遍历map查找符合条件的数对，map可以实现O(1)的查找，所以时间复杂度和空间复杂度都是O(N)。（这里需要注意，如果数组中有重复元素，那么这个方法就不能用，因为重复元素存入map时，后面的值（下标）会覆盖掉前面的值。）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //这种实现方式无法去掉重复的pair，如结果中会同时有（3,6）和（6,3）。  public class Solution{      public ArrayList<Pair<Integer, Integer>> findPair(int[] nums, int k) {          ArrayList<Pair<Integer, Integer>> pairs = new ArrayList<>();          HashMap<Integer, Integer> map = new HashMap<>();          for (int i = 0; i < nums.length; i++) {              map.put(nums[i], i);          }          for (Map.Entry<Integer, Integer> set : map.entrySet()) {              int target = k - set.getKey();              //第二个判断条件是为了去除（2，2）这种情况，即同一数字即做加数又做被加数的情况              if (map.containsKey(target) && !map.get(target).equals(set.getValue())) {                  pairs.add(new Pair<>(set.getValue(), map.get(target)));              }          }          return pairs;      }  } |

**滑动窗口：解决连续序列问题**

* **和为s的连续正整数序列（剑指offer57-II)**

输入一个正整数 target ，输出所有和为 target 的**连续正整数**序列（至少含有两个数）。序列内的数字由小到大排列，不同序列按照首个数字从小到大排列。

示例 1：

输入：target = 9

输出：[[2,3,4],[4,5]]

滑动窗口：窗口左右两端都只能向右移动，当和小于sum时，j++，和大于sum时，i++，和等于sum就记录下窗口中i —j 中的序列，然后窗口继续后移，查找下一个满足条件的序列。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | class Solution {      public int[][] findContinuousSequence(int target) {          int i = 1;          int j = 1;          int sum = 0;          List<int[]> list = new ArrayList<>();          //序列是由小到大排列，所以如果i>target/2，那么i+i+1肯定大于target          while(i <= target/2){              if(sum < target){                  sum += j;                  j++;              }else if(sum > target){                  sum -= i;                  i++;              }else{                  int[] res = new int[j - i];                  for(int z = i; z < j;z++){                      res[z - i] = z;                  }                  list.add(res);                  // 左边界向右移动                  sum -= i;                  i++;              }          }          return list.toArray(new int[list.size()][]);      }  } |

* **某一个大文件被拆成了N个小文件，每个小文件编号从0至N-1，相应大小分别记为S(i)。给定磁盘空间为C，试实现一个函数从N个文件中连续选出若干个文件拷贝到磁盘中，使得磁盘剩余空间最小。**

滑动窗口：每次记录窗口内的总和，和小于C，记录剩余空间，再窗口右端右移，和大于C，就窗口左端右移，小于C情况下比较剩余空间取最小值。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class Solution {      public int[] findMin(int[] s,int c){          int i = 0;          int j = 0;          int minValue = Integer.MAX\_VALUE;          int sum = 0;          int left = 0;          int right = 0;          while(j <= s.length){              if(sum <= c){                 j++;                 sum += s[j];                 minValue = Math.min(minValue,c - sum);                 if(minValue == c - sum){                     left = i;                     right = j;                 }              }else{                  i++;                  sum -= s[i];              }          }          int nums = new int[right - left];          for(int k = left;k < right;k++){              nums[k - left] = s[k];          }          return nums;      }  } |

* **给定m个不重复的字符 [a, b, c, d]，以及一个长度为n的字符串tbcacbdata，问能否在这个字符串中找到一个长度为m的连续子串，使得这个子串刚好由上面m个字符组成，顺序无所谓，返回任意满足条件的一个子串的起始位置，未找到返回-1。比如上面这个例子，acbd，3。 （2）**

类似leetcode 567：<https://leetcode-cn.com/problems/permutation-in-string/>

分析：

本题的子串需要满足长度为m，字符不重复，可以使用长为m的滑动窗口遍历字符串，窗口内每个字符都要出现一次，如果符合条件，就返回窗口起始位置。

如果这里不允许使用substring,indexOf函数，可以将字符串中每个字符出现的次数存入长度为26的数组中进行比较每次数组中对应位置数量是否相等，具体可参照上面LeetCode 567题。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class Solution {      public int checkInclusion(char[] ch,String s) {          if(ch.length > s.length()){              return -1;          }          for(int i = 0; i < s.length() - ch.length; i++){              //每次匹配长度为m的子串              if(matchs(ch,s.substring(i,i+ch.length)))                  return i;          }          return -1;      }      private static boolean matchs(char[] ch,String s){          for(int i = 0; i < s.length();i++){              //返回-1说明字符串中不包含这个字符              if(s.indexOf(ch[i]) == -1)                  return false;          }          return true;      }  } |

* **有序有重复数组，给定target确定范围**

使用滑动窗口。

* **算法题，一个有序数组，从随机一位截断，把前段放在后边，如 4 5 6 7 1 2 3求中位数**

用排序解决，问了下一般排序算法的复杂度，O(nlogn)

扫一遍，找出分界点，O(n)

提示我还有更快的方法，那肯定只有二分了，想了好久mid的判断条件后写了个解法，给面试官叙述了一遍跑样例的过程。

**哈希表辅助解决数组问题**

* **算法：求数组的最长连续递增数列，如：4， 200， 3， 1， 100， 2。结果是1 2 3 4，也就是说顺序可以打乱。（leetcode 128）**

思路一：排序，然后再遍历一遍找最长连续递增数列，时间复杂度O(NlogN)。

思路二：使用HashMap或者HashSet，将数组元素存入set中，然后遍历set，先找到每个序列的开头元素，即num没有num-1的情况，这时从num开始循环找num+1,记录次数，通过比较找出最大长度。时间和空间复杂度都是O（n)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | //这里用set，是因为set可以自动去重，用map也不影响结果  class Solution {      public int longestConsecutive(int[] nums) {          if (nums.length == 0) return 0;          HashSet<Integer> set = new HashSet<>();          for(int value : nums)              set.add(value);            int longestLength = 0;          for (int num: set) {              if (!set.contains(num - 1)){                  int currentNum = num;                  int currentLength = 1;                  while (set.contains(currentNum + 1)){                      currentNum += 1;                      currentLength += 1;                  }                  longestLength = Math.max(longestLength,currentLength);              }          }          return longestLength;      }  } |

* **一个无序数组，从小到大找到第一个缺的数，比如[8 2 4 3 6 9 7 11 12],第一个缺的就是5 （2）**

1.排序，时间复杂度O(NlogN)

2.用数组作为哈希表，将数字i放入数组中的i索引处，然后找中间没有存入数字的位置。时间和空间复杂度都是O(N)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public  class Solution {      public int findLost(int[] nums){          int min = nums[0];          int max = nums[0];          for(int value : nums){              min = Math.min(min,value);              max = Math.max(max,value);          }          int[] res = new int[max - min + 1];          for(int i = 0; i < res.length;i++){              res[i] = min - 1;          }          for(int value : nums){              res[value] = value;          }          int i = 0;          while(i < res.length){              if(res[i] = min - 1)                  break;          }          return i;      }  } |

* **输出给定数字下一个比它大的数字，比如输入：1234， 输出 1243。**

<https://leetcode-cn.com/problems/next-permutation/solution/xia-yi-ge-pai-lie-suan-fa-xiang-jie-si-lu-tui-dao-/>

首先从后往前遍历找到一个数字nums[i]比num[i+1]小，也就是找到递减到递增的转折点，然后再与递减序列中最小的大于num[i]的数字交换，这样交换完之后后面仍然是递减序列，将其转换为递增序列，这样可以保证数字增大，并且增大的幅度最小。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | class Solution {      public void nextPermutation(int[] nums) {          //从倒数第二个数字开始比较          int i = nums.length - 2;          //找nums[i] < nums[i + 1]的i          while(i >= 0 && nums[i] >= nums[i + 1]){              i--;          }          if(i >= 0){              int j = nums.length - 1;              while(j > 0 && nums[i] >= nums[j]){                  j--;              }              swap(nums,i,j);          }          reverse(nums,i + 1);      }        public void swap(int[] nums,int i, int j){          int temp = nums[i];          nums[i] = nums[j];          nums[j] = temp;      }        public void reverse(int[] nums,int start){          int j = nums.length - 1;          int i = start;          //因为是递减序列，所以可以这样交换成递增的          while(i < j){              swap(nums,i,j);              i++;              j--;          }      }  } |

* **算法题(leetcode55题)：给一个数组，例如[1,2,3,4,5]，a[i]表示在该位置可以向前行走的最大距离，判断是否可以到达数组的最后一个元素**

思路：

每次走最大的距离，如果能超过末尾元素，说明只要调小步伐就一定可以达到最后位置。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class Solution {      public boolean canJump(int[] nums) {          int n = nums.length;          int rightmost = 0;          for (int i = 0; i < n; ++i) {              if (i <= rightmost) {                  rightmost = Math.max(rightmost, i + nums[i]);                  if (rightmost >= n - 1) {                      return true;                  }              }          }          return false;      }  } |

* **AB两个排序数组，原地合并数组。（A当中穿插一些无效数字怎么处理？）**

思路：因为要原地合并数组，如果从前往后遍历，数组原来的值会被覆盖，所以只能从后往前遍历，将值从后往前存入。存入时比较当前两个指针指向的数字大小，选较大的存入，然后往前移动指针。

（有无效数字的情况楼主没有想到好的解决办法，小伙伴们如果有思路请在评论区解答，感谢！）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Solution {      public void merge(int[] A, int m, int[] B, int n) {          int i = m - 1,j = n - 1,k = m + n - 1;          while(j >= 0){              if(i < 0 || B[j] >= A[i]){                  A[k--] = B[j--];              }else {                  A[k--] = A[i--];              }          }      }  } |

**排序相关**

* **快排（3)**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public void quickSort(int[] arr,int L,int R){      if (arr.length == 0) return;      int i = L;      int j = R;      int key = arr[(i + j)/2];      while (i <= j){          while (arr[i] < key)              i++;          while (arr[j] > key)              j--;          if (i <= j){              int temp = arr[i];              arr[i] = arr[j];              arr[j] = temp;              i++;              j--;          }      }      //上面一个while保证了第一趟排序支点的左边比支点小，支点的右边比支点大了。      //“左边”再做排序，直到左边剩下一个数(递归出口)      if (L < j)          quickSort(arr,L,j);      //“右边”再做排序，直到右边剩下一个数(递归出口)      if(i < R)          quickSort(arr,i,R);  } |

* **高考成绩2000万数据，分数0-750，如何快速知道你的排名，如何知道任一分数排名 --->桶排序 （3）**

桶排序 (Bucket sort)的工作的原理：假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排）。

算法描述：

设置一个定量的数组当作空桶；

遍历输入数据，并且把数据一个一个放到对应的桶里去；

对每个不是空的桶进行排序；

从不是空的桶里把排好序的数据拼接起来。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | public class Solution {      public ArrayList<Integer> bucketSort(int[] scores){        //先确定最大最小值，来确定范围        int max = Integer.MIN\_VALUE;        int min = Integer.MAX\_VALUE;        for(int i = 0; i < scores.length;i++){            max = Math.max(max,scores[i]);            min = Math.min(min,scores[i]);        }        //计算出桶数        //int bucketNum = (max - min)/scores.length + 1;        //这里直接给出751个桶        int bucketNum = 751;        ArrayList<ArrayList<Integer>> list = new ArrayList<>(bucketNum);        for(int i = 0; i < bucketNum; i++){            list.add(new ArrayList<Integer>());        }          //将每个元素放入桶        for(int i = 0; i < scores.length;i++){            //本题中这里放元素也可以简化            //list.get((scores[i] - min)/bucketNum).add(scores[i]);            list.get(scores[i]).add(scores[i]);        }          //桶内排序,本题中可以省略这一步        for(int i = 0; i< list.size();i++){            Collections.sort(list.get(i));        }        return list;      }  } |

* **数组中第K大的数（LeetCode215题）**

首先想到最简单的排序，然后返回第k大的元素，时间复杂度O(NlogN)。

**快排的patition思想**：

时间复杂度O（n)，空间复杂度O(1)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | public class Solution {      public int findKthLargest(int[] nums, int k) {          int len = nums.length;          int left = 0;          int right = len - 1;          // 转换一下，第 k 大元素的索引是 len - k          int target = len - k;            while (true) {              int index = partition(nums, left, right);              if (index == target) {                  return nums[index];              } else if (index < target) {                  left = index + 1;              } else {                  right = index - 1;              }          }      }        /\*\*       \* 在数组 nums 的子区间 [left, right]       执行 partition 操作，返回 nums[left] 排序以后应该在的位置       \* 在遍历过程中保持循环不变量的语义       \* 1、[left + 1, j] < nums[left]       \* 2、(j, i] >= nums[left]       \*/      public int partition(int[] nums, int left, int right) {          int pivot = nums[left];          int j = left;          for (int i = left + 1; i <= right; i++) {              if (nums[i] < pivot) {                  // 小于 pivot 的元素都被交换到前面                  j++;                  swap(nums, j, i);              }          }          // 在之前遍历的过程中，满足 [left + 1, j] < pivot，          //并且 (j, i] >= pivot          swap(nums, j, left);          // 交换以后 [left, j - 1] < pivot, nums[j] = pivot,          // [j + 1, right] >= pivot          return j;      }        private void swap(int[] nums, int index1, int index2) {          int temp = nums[index1];          nums[index1] = nums[index2];          nums[index2] = temp;      }  } |

* **top k，返回前K大的元素**

这里可以使用和上题一样的快排patition。

也可以用堆排序，可以取前k个元素构造小根堆，然后遍历后面的元素，如果读取到的元素比堆顶下，直接丢弃，如果比堆顶大，就替换掉堆顶元素，然后调整堆。时间复杂度O(NlogK)。（堆实现参考下一题）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | public class Solution {      public ArrayList<Integer> topK(int[] nums){          if(nums.length <= k) return nums;          int n = nums.length;          int left = 0;          int right = n - 1;          int target = n - k;          int index = 0;          while(true){              index = partition(nums,left,right)              if(index == target){                  break;              }else if(index < target{                  left = index + 1;              }else{                  right = index - 1;              }          }          ArrayList<Integer> list = new ArrayList<>();          for(int i = target;i < n;i++){              list.add(nums[i]);          }          return list;      }      public int pattition(int[] nums,int left,int right){          int pivot = nums[left];          int j =left;         for(int i = left + 1;i <= right;i++){             if(nums[i] > pivot){                 j++;                 swap(nums,i,j);             }         }         swap(nums,left,j);         return j;      }      public void swap(int nums,int i,int j){          int temp = nums[i];          nums[i] = nums[j];          nums[j] =temp;      }  } |

* **10亿个数字，取最小的100个数**

(说思路 最大堆， partition 两者的时间复杂度，写伪代码)

1.最大堆：

先取出前100个数，维护一个100个数的最大堆，遍历一遍剩余的元素，在此过程中维护这个最大堆就可以了。

具体步骤如下：

step1：取前m个元素（例如m=100），建立一个大根堆。保持一个大根堆得性质的步骤，运行时间为O（logm);建立一个大根堆运行时间为mO(logm）=O(m logm);

step2:顺序读取后续元素，直到结束。每次读取一个元素，如果该元素比堆顶元素大，直接丢弃；如果小于堆顶元素，则用该元素替换堆顶元素，然后保持最大堆性质。最坏情况是每次都需要替换掉堆顶的最大元素，因此需要维护堆的代价为(N-m)O(lgm); 最后这个堆中的元素就是前最小的100个。时间复杂度为O(N lgm）。

时间复杂度为O(10亿\*lg100)。

**首先构造大根堆**：

堆的定义：n个关键字序列array[0，...，n-1]，当且仅当满足下列要求：

(0 <= i <= (n-1)/2)

① array[i] <= array[2\*i + 1] 且 array[i] <= array[2\*i + 2]； 称为小根堆；

② array[i] >= array[2\*i + 1] 且 array[i] >= array[2\*i + 2]； 称为大根堆；

n个节点的完全二叉树array[0，...，n-1]，最后一个节点n-1是第(n-1-1)/2个节点的孩子。对第(n-1-1)/2个节点为根的子树调整，使该子树称为堆。

对于大根堆，**调整方法**为：若【根节点的关键字】小于【左右子女中关键字较大者】，则交换。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | //构建大根堆：将array看成完全二叉树的顺序存储结构  private int[] buildMaxHeap(int[] array) {      //从最后一个节点array.length-1的父节点（array.length-1-1）/2开始，      //直到根节点0，反复调整堆      for (int i = (array.length - 2) / 2; i >= 0; i--) {          adjustDownToUp(array, i, array.length);      }      return array;  }  //将元素array[k]自下往上逐步调整树形结构  private void adjustDownToUp(int[] array, int k, int length) {      int temp = array[k];      for (int i = 2 \* k + 1; i < length - 1; i = 2 \* i + 1) {      //i为初始化为节点k的左孩子，沿节点较大的子节点向下调整          if (i < length && array[i] < array[i + 1]) {          //取节点较大的子节点的下标              i++;   //如果节点的右孩子>左孩子，则取右孩子节点的下标          }          if (temp >= array[i]) {  //根节点 >=左右子女中关键字较大者，调整结束              break;          } else {   //根节点 <左右子女中关键字较大者              array[k] = array[i]; //将左右子结点中较大值array[i]调整到双亲节点上              k = i; //【关键】修改k值，以便继续向下调整          }      }      array[k] = temp;  //被调整的结点的值放人最终位置  } |

堆排序(大根堆)：

①将存放在array[0，...，n-1]中的n个元素建成初始堆；

②将堆顶元素与堆底元素进行交换，则序列的最大值即已放到正确的位置；

③但此时堆被破坏，将堆顶元素向下调整使其继续保持大根堆的性质，再重复第②③步，直到堆中仅剩下一个元素为止。

堆排序算法的性能分析：

空间复杂度:o(1)；

时间复杂度:建堆：o(n)，每次调整o(log n)，故最好、最坏、平均情况下：o(n\*logn);

稳定性：不稳定

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | //堆排序  public int[] heapSort(int[] array) {      array = buildMaxHeap(array); //初始建堆，array[0]为第一趟值最大的元素      for (int i = array.length - 1; i > 1; i--) {          int temp = array[0];  //将堆顶元素和堆底元素交换，即得到当前最大元素正确的排序位置          array[0] = array[i];          array[i] = temp;          adjustDownToUp(array, 0, i);  //整理，将剩余的元素整理成堆      }      return array;  } |

本题中：构造完堆之后将新的元素与大根堆的堆顶元素比较，如果新元素比堆顶元素大，直接舍弃，如果小就跟堆顶元素交换，然后调整大根堆。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public int[] findMin(int[] array,int[] all){      array = buildMaxHeap(array);      //前面100个数字（0-99号）已经取出了      for(int i = 100;i <  all.length - 1; i++){          if(all[i] >= array[0]){              continue;          } else {              array[0] = all[i];              adjustDownToUp(array,0,array.length);          }      }      return array;  } |

2.快排划分的思想：

每次分割之后只考虑比轴小的一部分，直到比轴小的一部分数量在100多个的时候，采用传统排序算法排序，取前100个。

step1：递归对所有数据分成[a,b)，(b,d]两个区间，[a,b)区间内的数都是小于(b,d]区间内的数。

step2：对[a,b)重复 step1操作，直到最左边的区间个数小于100个。注意(b,d]区间不用划分

step3：返回上一个区间，并返回此区间的数字数目。接着方法仍然是对上一区间的左边进行划分，分为[a2,b2)，(b2,d2]两个区间，取(a2,b2]区间。如果个数不够，继续 step3操作，如果个数超过100的就重复 step1操作，直到最后右边只有100个数为止。

复杂度为O(10亿\*100)

* **算法：手写jdk中的优先级队列 PriorityQueue（最小堆）**

参照上题中堆的实现。

**栈，队列和链表**

* **最大栈（最小栈）**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | 使用辅助栈：  public class MaxStack{      private Stack<Integer> stack;      private Stack<Integer> helper;      public MaxStack(){          stack = new Stack<>();          helper = new Stack<>();      }      public void push(int x) {          if(helper.isEmpty() || helper.peek() <= x){              helper.push(x);          }          stack.push(x);      }      public void pop(){          if(stack.peek() == helper.peek()){              helper.pop();          }          stack.pop();      }      public int peek(){          if(!helper.isEmpty()){              return stack.peek();          }          throw new RuntimeException("栈中元素为空");        }      public int getMax(){          if(!helper.isEmpty()){              return helper.peek();          }          throw new RuntimeException("最大值栈中元素为空");      }  }  用最大值标记，存入数据栈中，空间复杂度降到O(1)  public class MaxStack {      private Stack<Integer> stack;      public MaxStack(){          stack = new Stack<>();      }      int max = Integer.MIN\_VALUE;      public void push(int x){          if(max <= x){              if(!stack.isEmpty()){                  stack.push(max);              }              max = x;          }          stack.push(x);      }      public void pop(){          if(stack.peek() == max){              stack.pop();              max = stack.pop();          }else{              stack.pop();          }      }      public int getMax(){          return max;      }  } |

* **两个栈实现一个队列**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | public class StackToList{      private Stack<Integer> stack;      private Stack<Integer> helper;        public StackToList{          stack = new Stack<>();          helper = new helper<>();      }      public void push(int x){          stack.push(x);      }      public Integer poll() {          if(stack.isEmpty() && helper.isEmpty())              throw new RuntimeException("队列为空");          dataRemove();          return helper.pop();      }      public Integer peek() {          if(stack.isEmpty() && helper.isEmpty())              throw new RuntimeException("队列为空");          dataRemove();          return helper.peek();      }      public void dataRemove(){         if(helper.isEmpty()){              while(!stack.isEmpty())                  helper.push(stack.pop());          }      }  } |

**两个队列实现一个栈：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | /\*\*       \* @Description: 仅用队列来实现栈       \* 采用两个队列，一个数据队列queue 一个辅助队列 help       \* 入队时全部入队queue 出队时，将queue队列中数据入队help 将最后一个数 返回 并queue指针和help交换       \*/      public static class QueueToStack{          private Queue<Integer> queue;          private Queue<Integer> help;          public QueueToStack(){              queue = new LinkedList<Integer>();              help = new LinkedList<Integer>();          }            public void push(int pushInt){              queue.add(pushInt);          }          /\*\*           \* @Description: 弹栈操作操作           \* 弹栈时，queue队列所有数据迁移至 help 返回最后一个数 并交换指针           \*/          public Integer pop(){              if (queue.isEmpty())                  throw new RuntimeException("栈空！");              while (queue.size()>1){                  help.add(queue.poll());              }              int temp = queue.poll();              swap();              return temp;          }            /\*\*           \* @Description: 栈的peek操作 只返回栈顶元素           \* 原理同pop操作，但是在最后的一个元素要继续入队 help 因为peek只是返回栈顶 并非弹出           \*/          public Integer peek(){              if (queue.isEmpty())                  throw new RuntimeException("栈空");              while (queue.size()>1){                  help.add(queue.poll());              }              int temp=queue.poll();              help.add(temp);              swap();              return temp;          }            private void swap() {              Queue<Integer> temp = queue;              queue = help;              help = temp;          }      } |

* **链表实现一个栈**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public class ListNode{      int val;      ListNode next;      ListNode(int val){          this.val =val;      }  }  public class ListToStack{      public ListToStack(){          ListNode head;      }      public void push(int x){          ListNode node = new ListNode(x);          node.next = head.next;          head.next = node;      }      public int pop(){          ListNode node = head.next;          head.next = node.next;          node.next = null;          return node.val;      }      public int peek(){          return head.next.val;      }  } |

* **两个链表，可能相交，找出相交的节点，给出证明（2)**

1.若两个单链表一个为有环,一个无环. 那么肯定不能相交.

2.若二者都没有环, 问题就转化为 两个无环单链表是否相交,是否能找到第一个相交的节点，方法就是 快慢指针 。

3.若二者都有环,那么问题变成了两个有环单链表是否相交.

第一,先找到二者是否相交.

第二,若相交则需要遍历一遍找到相交点.

* + 如果无环：问题转化为如何判断两个无环链表是否相交？
    - 要想判断是否相交，肯定要看是否能找到相交的点，如果能让两个指针同时走到相交点，也就可以确定两个链表相交。因为相交后两个链表到终点的距离相等，那么只要两个指针能够消除两个链表之间的长度差就可以到达终点的距离相等。假设链表A比B长，pB到达结尾后指向headA，然后继续向前遍历，当pA到达结尾后指向headB，此时两者的长度差就消除了。继续遍历如果两者指向同一节点，返回节点，如果两者都走到结尾依然没有相交，就返回null。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | //无环的情况  public class ListNode {      int val;      ListNode next;      ListNode(int val){          this.val = val;          next = null;      }  }    public class Solution {      public ListNode getListNode(ListNode headA,ListNode headB) {          if(headA == null || headB == null){              return null;          }          ListNode pA = headA,pB = headB;          while(pA != pB){              if(pA == null){                  pA = headB;              }else{                  pA = pA.next;              }              if(pB == null){                  pB = headA;              }else{                  pB = pB.next;              }          }          return pA;      }  }  简化版：  public class Solution {      public ListNode getIntersectionNode(ListNode headA, ListNode headB) {          if (headA == null || headB == null) return null;          ListNode pA = headA, pB = headB;          //如果不相交，pA=pB=null          while (pA != pB) {              pA = pA == null ? headB : pA.next;              pB = pB == null ? headA : pB.next;          }          return pA;      }  } |

* + 如果有环，就转化为如何判断两个有环链表是否相交？
    - 两个有环链表相交分为两种情况，一种是两个链表在进入环之前相交，另一种是两个链表在环内相交，如果不是在同一个节点进入，就是两个入口在两个节点。
      * 第一种情况：可以将环在环的入口处断开，这样就与无环链表相交一样，可以根据无环链表的方法判断；
      * 第二种情况：两者相交于环内，而且入环口不同。pA的入口节点（标记位loopA），pB的入环口标记为loopB,当他们都遍历一圈停在自己的入口节点时，pA往后遍历，pB不动，pA回到loopA之前，一定会碰到pB,如果碰不到，说明两者不在一个环里。
      * 如果两种都不是，则是没有相交。

**找有环链表的入口节点：**

快慢指针：设置快慢指针，都从链表头出发，快指针每次走两步，慢指针一次走一步，假如有环，一定相遇于环中某点(结论1)。接着让两个指针分别从相遇点和链表头出发，两者都改为每次走一步，最终相遇于环入口(结论2)。

证明结论1：设置快慢指针fast和low，fast每次走两步，low每次走一步。假如有环，两者一定会相遇（因为low一旦进环，可看作fast在后面追赶low的过程，每次两者都接近一步，最后一定能追上）。

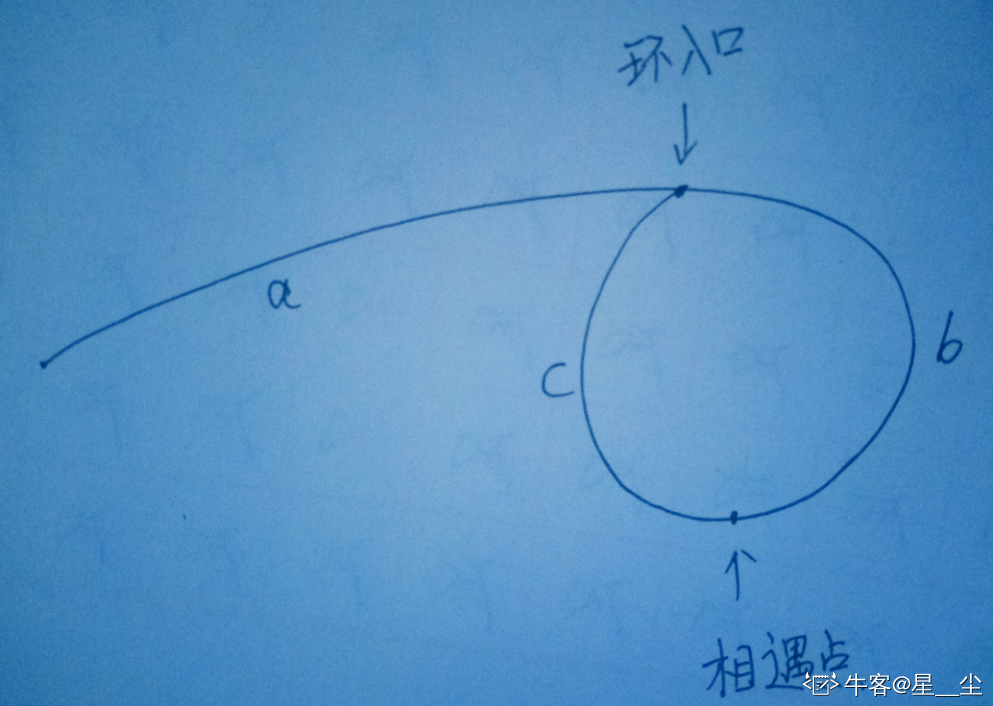
证明结论2：

设：

链表头到环入口长度为--a

环入口到相遇点长度为--b

相遇点到环入口长度为--c



则：相遇时

快指针路程=a+(b+c)k+b ，k>=1 其中b+c为环的长度，k为绕环的圈数（k>=1,即最少一圈，不能是0圈，不然和慢指针走的一样长，矛盾）。

慢指针路程=a+b

快指针走的路程是慢指针的两倍，所以：

（a+b）\*2=a+(b+c)k+b

化简可得：

a=(k-1)(b+c)+c 这个式子的意思是： 链表头到环入口的距离=相遇点到环入口的距离+（k-1）圈环长度。其中k>=1,所以k-1>=0圈。所以两个指针分别从链表头和相遇点出发，最后一定相遇于环入口。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | pulic class ListNode{      int val;      ListNode next = null;      ListNode(int val){          this.val = val;      }  }  public class Solution {      public ListNode EntryNodeOfLoop(ListNode pHead){          ListNode fast = pHead;          ListNode slow = pHead;          if(fast == null || fast.next == null){              return null;          }          while(fast != null && fast.next != null){              fast = fast.next.next;              slow = slow.next;              if(fast == slow){                  break;              }          }          slow = pHead;          while(fast != slow){              fast = fast.next;              slow = slow.next;          }          return fast;      }  } |

* **反转链表 ---->反转链表升级版（每k个反转一下）（4）**

普通版：

方法一:递归

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class Listnode{      int val;      ListNode next;      ListNode pre;      ListNode (int val){          this.val = val;      }  }  //递归  public class Solution{      public ListNode revoseList(ListNode head){          if(head == null || head.next == null){              return head;          }          ListNode cur = revoseList(head.next);          head.next.next = head;          head.next = null;          return cur;      }  } |

方法二：双指针迭代

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class Solution{      public ListNode revoseList(ListNode head){          if(head == null || head.next == null){              return head;          }          ListNode pre = null;          ListNode cur = head;          ListNode temp = null;          while(cur != null){              //标记下一个节点，以免丢失              temp = cur.next;              //实现反转              cur.next = pre;              pre = cur;              cur = temp;          }          return pre;      }  } |

**K个一组反转链表（LeetCode25题）：**

<https://leetcode-cn.com/problems/reverse-nodes-in-k-group/solution/tu-jie-kge-yi-zu-fan-zhuan-lian-biao-by-user7208t/>

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | /\*\*   \* Definition for singly-linked list.   \* public class ListNode {   \*     int val;   \*     ListNode next;   \*     ListNode(int x) { val = x; }   \* }   \*/  class Solution {      public ListNode reverseKGroup(ListNode head, int k) {          //定义一个头指针，用来将后面所有节点统一处理          ListNode dum = new ListNode();          dum.next = head;          //用来标记每一次反转部分的前一个结点          ListNode pre = dum;          //运动指针，扫描要反转的部分          ListNode end = dum;            while(end.next != null){              //每次扫描完要反转的部分，如果end为空说明达到尾部，直接break              for(int i = 0; i < k && end != null;i++) end = end.next;              if(end == null) break;              //标记要反转部分的开头节点              ListNode start = pre.next;              //标记反转链表后面部分的第一个节点，避免丢失              ListNode next = end.next;              //将要反转部分向后的指针断开，准备反转              end.next = null;              //反转完的链表返回反转后的头结点，接到pre的后面              pre.next = reverse(start);              //反转后start指针应该指向反转完成部分的末尾              start.next = next;              //让pre和end指针继续指向待反转链表的前一个节点              pre = start;              end = pre;          }          return dum.next;      }      public ListNode reverse(ListNode start){          ListNode cur = start;          ListNode pre1 = null;          while(cur != null){              ListNode temp = cur.next;              cur.next = pre1;              pre1 = cur;              cur = temp;          }          return pre1;      }  } |

* **leetcode 链表求和**

给定两个用链表表示的整数，每个节点包含一个数位。这些数位是反向存放的，也就是个位排在链表首部。编写函数对这两个整数求和，并用链表形式返回结果。

示例：

输入：(7 -> 1 -> 6) + (5 -> 9 -> 2)，即617 + 295

输出：2 -> 1 -> 9，即912

（如果链表存储数据的顺序反过来怎么加？这个楼主不会，大家有思路请在评论区交流，十分感谢！）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | /\*\*   \* Definition for singly-linked list.   \* public class ListNode {   \*     int val;   \*     ListNode next;   \*     ListNode(int x) { val = x; }   \* }   \*/  class Solution {      public ListNode addTwoNumbers(ListNode l1, ListNode l2) {          ListNode head = new ListNode(0);          ListNode cur = head;          int carry = 0;//用以计算进位          //只要有一个不为空或者最后有进位都要继续循环          while(l1 != null || l2 != null || carry != 0){              int num1 = l1 == null ? 0 : l1.val;              int num2 = l2 == null ? 0 : l2.val;              int sum = num1 + num2 + carry;              carry = sum / 10;              cur.next = new ListNode(sum % 10);              if(l1 != null) l1 = l1.next;              if(l2 != null) l2 = l2.next;              cur = cur.next;          }          return head.next;      }  } |

**二叉树**

* **二叉树的前序遍历非递归**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | public class TreeNode{      int val;      TreeNode left;      TreeNode right;      TreeNode(int val){          this.val = val;      }  }  //非递归  public void preOrder(TreeNode root){      if(root == null) return;      Stack<TreeNode> stack = new Stack<>();      stack.push(root);      while(!stack.isEmpty()){          TreeNode node = stack.pop();          System.out.print(node.val);          //因为栈是先进后出，所以先压右孩子，再压左孩子          if(node.right != null)              stack.push(node.right);          if(node.left != null)              stack.push(node.left);      }  }  //递归  public void preOrder(TreeNode root){      if(root == null) return;      System.out.print(root.val);      preOrder(root.left);      preOrder(root.right);  } |

* **二叉树的后序遍历非递归(2)**

由于压栈顺序是从上而下，先左后右，出栈时就是从下而上，先右后左，这时可以用一个辅助栈，将弹出的值压入栈中，最后统一弹出，顺序就是后序遍历的先左后右，从下而上。（也可以用LinKedList，弹出后加入链表头，这样也可以逆序）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public void postOrder(TreeNode root){      if(root == null) return;      Stack<TreeNode> s1 = new Stack<>();      Stack<TreeNode> s2 = new Stack<>();      s1.push(root);      while(!s1.isEmpty()){          TreeNode node = s1.pop();          s2.push(node);          if(node.left != null)             s1.push(node.left);         if(node.right != null)             s1.push(node.right);      }      while(!s2.isEmpty())          System.out.print(s2.pop().val + " ");  } |

* **中序遍历非递归**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | //中序遍历非递归  class Solution {      public List<Integer> inorderTraversal(TreeNode root) {          Stack<TreeNode> stack = new Stack<>();          TreeNode p =root;          ArrayList<Integer> list = new ArrayList<>();          while(p != null || !stack.isEmpty()){              //一直走到最左边的左孩子              while(p != null){                  stack.push(p);                  p = p.left;              }              p = stack.pop();              list.add(p.val);              p = p.right;          }          return list;      }  } |

* **剑指offer 从上往下打印二叉树（层序遍历）（2)**

与后序遍历类似，可使用**队列**，将左右节点节点先后加入**队列尾**，每次从**队列头**取出节点进入输出链表。这样可以保证每次取出的元素都是本层现存的第一个元素。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class TreeNode {      int val;      TreeNode left = null;      TreeNode right = null;      public TreeNode(int val){          this.val = val;      }  }    public class Solution {      public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {          ArrayList<Integer> result = new ArrayList<>();          if(root == null)              return result;          LinkedList<Integer> queue = new LinkedList<>();          queue.add(root);          while(!queue.isEmpty()){              TreeNode node = queue.poll();              result.add(node.val);              if(node.left != null){                  queue.add(node.left);              }              if(node.right != null){                  queue.add(node.right);              }          }          return result;      }  } |

* **蛇形遍历二叉树(2)**

与层序遍历的区别是当层数为奇数就从左往右遍历，层数是偶数，就从右往左。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | public class Solution {      public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {          ArrayList<Integer> result = new ArrayList<>();          if(root == null)              return result;          LinkedList<Integer> queue = new LinkedList<>();          queue.add(root);          int level = 1;//认为根节点在第一层          while(!queue.isEmpty()){              TreeNode node = queue.poll();              result.add(node.val);              level++;              if(level % 2 == 1){                  if(node.left != null){                      queue.add(node.left);                  }                  if(node.right != null){                      queue.add(node.right);                  }              }else {                  if(node.right != null){                      queue.add(node.right);                  }                  if(node.left != null){                      queue.add(node.left);                  }              }            }          return result;      }  } |

* **二叉树右视图（2）**

就是找层序遍历每层最后一个节点。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | /\*\*   \* Definition for a binary tree node.   \* public class TreeNode {   \*     int val;   \*     TreeNode left;   \*     TreeNode right;   \*     TreeNode(int x) { val = x; }   \* }   \*/  class Solution {      public List<Integer> rightSideView(TreeNode root) {          ArrayList<Integer> res = new ArrayList<>();          if(root == null) return res;          int curCount = 1;          int nextCount = 0;          LinkedList<TreeNode> list = new LinkedList<>();          list.add(root);          while(!list.isEmpty()){              TreeNode node = list.poll();              curCount --;              if(node.left != null){                  list.add(node.left);                  nextCount++;              }              if(node.right != null){                  list.add(node.right);                  nextCount++;              }              if(curCount == 0){                  res.add(node.val);                  curCount = nextCount;                  nextCount = 0;              }          }          return res;      }  } |

* **二叉树各层节点数，递归、非递归，时间、空间复杂度**

思路：借助队列实现层序遍历，在遍历的过程中，level记录当前层数，curCount当前层的节点数，nextCount下一层的节点数。每次弹出节点，将当前层节点减一，然后将当前节点的左右子节点加入队列，同时计数。当前层的节点数等于0时，说明这一层遍历完了，同时记录的下一层节点数就是下一层总的节点，直接保存。遍历到队列为空为止。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | //非递归，时间复杂度O(N),空间复杂度O(N)  public Map<Integer,Integer> levelNodeCount(TreeNode root){      HashMap<Integer,Integer> map = new HashMap<>();      if(root == null) return map;      int level = 1;      int curCount = 1;      int nextCount = 0;      map.put(level,curCount);      LinkedList<TreeNode> list = new LinkedList<>();      list.add(root);      while(!list.isEmpty()){          TreeNode node = list.poll();          curCount--;          if(node.left != null){              list.add(node.left);              nextCount++;          }          if(node.right != null){              list.add(node.right);              nextCount++;          }           if(curCount == 0){               level++;               map.put(level,nextCount);               curCount = nextCount;               nextCout = 0;           }      }      return map;  }  递归：时间复杂度O(N),空间复杂度O(N)  这里保存了每一层的每一个节点  class Solution {      private List<List<Integer>> res = new ArrayList<>();      public List<List<Integer>> levelOrder(TreeNode root) {          if(root == null)              return res;          helper(root,0);          return res;      }      public void helper(TreeNode root,int level){          //说明level已经比现存的层数(res.size() - 1)多了，需要新建一个新的          if(res.size() == level){              res.add(new ArrayList<Integer>());          }          //存入当前层的列表里，递归过程中每次都会往一个列表里面加入数字，每次都          //加到不同的列表里。          res.get(level).add(root.val);          if(root.left != null)              helper(root.left,level + 1);          if(root.right != null)              helper(root.right,level + 1);      }  } |

* **二叉搜索树的第K大的节点（剑指offer第54题）**

二叉搜索树（二叉排序树）满足：根节点大于左子树，小于右子树。那么二叉树的中序遍历序列就是一个递增序列。为方便了找第K大的节点，我们可以调换左右子树的遍历顺序，这样遍历序列是一个递减序列，这样遍历到第K个元素就是我们要找的第K大的节点。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | //迭代：  class Solution {      public int kthLargest(TreeNode root, int k) {          Stack<TreeNode> stack = new Stack<>();          TreeNode p = root;          int count = 0;          while(p != null || !stack.isEmpty()){              while(p != null){                  stack.push(p);                  p = p.right;              }              p = stack.pop();              count++;              if(count == k){                  return p.val;              }              p = p.left;          }          return 0;      }  }  //递归  class Solution {      int res,index;      public int kthLargest(TreeNode root, int k) {          index = k;          dfs(root);          return res;      }      public void dfs(TreeNode root){          if(root == null) return;          dfs(root.right);          if(index == 0) return;          index--;          if(index == 0) res = root.val;          dfs(root.left);      }  } |

* **求完全二叉树的节点个数，小于O(n)，并分析复杂度**

完全二叉树的定义如下：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第 h 层，则该层包含 1~2^h个节点。

* 分析：首先统计二叉树左右子树的层数，有以下两种情况：
  + left == right。这说明，左子树一定是满二叉树，因为节点已经填充到右子树了，左子树必定已经填满了。所以左子树的节点总数我们可以直接得到，是2^left - 1，加上当前这个root节点，则正好是2^left。再对右子树进行递归统计。
  + left != right。说明此时最后一层不满，但倒数第二层已经满了，可以直接得到右子树的节点个数。同理，右子树节点+root节点，总数为2^right。再对左子树进行递归查找。
* 时间复杂度为O(logN\*logN) < O(n)。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class Solution {      public int countNodes(TreeNode root) {          if(root == null)              return 0;          int left = countlevel(root.left);          int right = countlevel(root.right);          if(left == right){              //移位运算，1向左移动left次，相当于1\*2^n              return countNodes(root.right) + (1<<left);          }else{              return countNodes(root.left) + (1<<right);          }      }      private int countlevel(TreeNode root){          int level = 0;          while(root != null){              level++;              root = root.left;          }          return level;      }  } |

* **编程题：求二叉树根节点到叶子结点的路径和的最小值（leetcode）**

递归：只有左子树，就只计算左子树，只有右子树，就只计算右子树，两个都有，就计算左右子树取最小值。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public int minPath(TreeNode root){      if(root == null) return 0;      if(root.left != null && root.right == null){          return 1+minPath(root.left);      }      if(root.left == null && root.right != null){          return 1 + minPath(root.right);      }      return 1 + Math.min(minPath(root.left),minPath(root.right));  } |

* **二叉树中的最大路径和 leetcode 124**

给定一个**非空**二叉树，返回其最大路径和。

本题中，路径被定义为一条从树中任意节点出发，达到任意节点的序列。该路径**至少包含一个**节点，且不一定经过根节点。

思路：递归计算单边值，如果单边值小于0就不算，单边值大于0就加到总的路径和里面。**单边最大和**等于Math.max(root, root+left, root+right)。计算单边最大和的过程中记录总的最大和。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | int max = Integer.MIN\_VALUE;      public int maxPathSum(TreeNode root) {          if (root == null) {              return 0;          }          dfs(root);          return max;      }        /\*\*       \* 返回经过root的单边分支最大和， 即Math.max(root, root+left, root+right)       \* @param root       \* <a href="/profile/547241" data-card-uid="547241" class="js-nc-card" target="\_blank">@return \*/      public int dfs(TreeNode root) {          if (root == null) {              return 0;          }          //计算左边分支最大值，左边分支如果为负数还不如不选择          int leftMax = Math.max(0, dfs(root.left));          //计算右边分支最大值，右边分支如果为负数还不如不选择          int rightMax = Math.max(0, dfs(root.right));          //left->root->right 作为路径与历史最大值做比较          max = Math.max(max, root.val + leftMax + rightMax);          // 返回经过root的单边最大分支给上游          return root.val + Math.max(leftMax, rightMax);      }    </a> |

* **写代码： 二叉树的最近公共祖先 leetcode 236 稍有不同，原题的2个节点，面试是多个节点，算法的时间复杂度 (2)**

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

两个节点:递归

当我们用递归去做这个题时不要被题目误导，应该要明确一点

这个函数的功能有三个：给定两个节点 p 和 q

如果 p 和 q 都存在，则返回它们的公共祖先；

如果只存在一个，则返回存在的一个；

如果 p 和 q 都不存在，则返回NULL

本题说给定的两个节点都存在，那自然还是能用上面的函数来解决

具体思路：

（1） 如果当前结点 root 等于NULL，则直接返回NULL

（2） 如果root 等于 p 或者 q ，那这棵树一定返回 p 或者 q

（3） 然后递归左右子树，因为是递归，使用函数后可认为左右子树已经算出结果，用 left 和 right 表示

（4） 此时若left为空，那最终结果只要看 right；若 right 为空，那最终结果只要看 left

（5） 如果 left 和 right 都非空，因为只给了 p 和 q 两个结点，都非空，说明一边一个，因此 root 是他们的最近公共祖先

（6） 如果 left 和right 都为空，则返回空（其实已经包含在前面的情况中了）

时间复杂度是O(n)：每个结点最多遍历一次，空间复杂度是O(n)：需要系统栈空间

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Solution {      public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {          if(root == null) return null;          if(root == p || root == q) return root;          TreeNode left = lowestCommonAncestor(root.left,p,q);          TreeNode right = lowestCommonAncestor(root.right,p,q);          if(left == null) return right;          if(right == null) return left;          if(right != null && left != null) return root;          return null;      }  } |

多个节点的情况类似，多加一个判断条件就好了，依然是看左右子树是否存在某个节点。

* **路径总和 leetcode 112及其延伸**

给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | //递归  class Solution {      public boolean hasPathSum(TreeNode root, int sum) {          if(root == null) return false;          sum -= root.val;          if(root.left == null && root.right == null){              return sum==0;          }else{              return (hasPathSum(root.left,sum) || hasPathSum(root.right,sum));          }      }  }  //迭代：需要同时记录当前剩余的sum和当前节点  class Solution {      public boolean hasPathSum(TreeNode root, int sum) {          if(root == null) return false;          Stack<TreeNode> nodeStack = new Stack<>();          Stack<Integer> numStack = new Stack<>();            nodeStack.push(root);          numStack.push(sum - root.val);            while(!nodeStack.isEmpty()){              TreeNode node = nodeStack.pop();              Integer value = numStack.pop();                if(node.left == null && node.right == null && value == 0){                  return true;              }                if(node.left != null){                  nodeStack.push(node.left);                  numStack.push(value - node.left.val);              }              if(node.right != null){                  nodeStack.push(node.right);                  numStack.push(value - node.right.val);              }          }          return false;      }  } |

**延伸：（LeetCode113题）**

给定一个二叉树和一个目标和，找到所有从根节点到叶子节点路径总和等于给定目标和的路径。

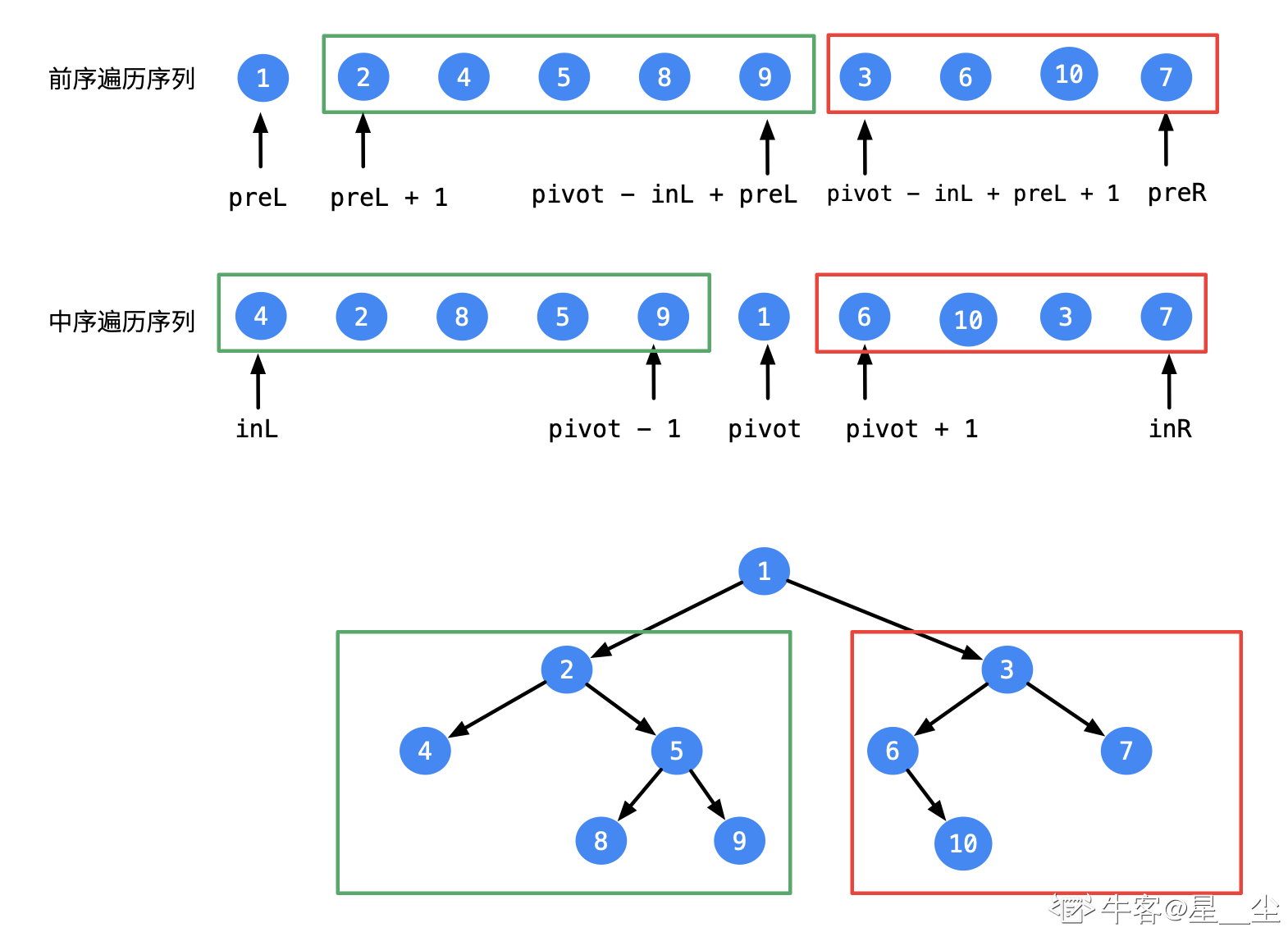
递归+回溯：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class Solution {      List<List<Integer>> res;      public List<List<Integer>> pathSum(TreeNode root, int sum) {          res = new ArrayList<>();          if(root == null)              return res;          backTrack(root, sum, 0, new ArrayList<>());          return res;      }        private void backTrack(TreeNode x, int sum, int curSum, List<Integer> vals){          vals.add(x.val);          curSum += x.val;          if(x.left == null && x.right == null){              if(curSum == sum){                  res.add(new ArrayList<>(vals));              }              //到达叶子节点后，无论是否符合要求都要              //回退到上一个节点，看上一个节点的右子树是否符合要求              vals.remove(vals.size() - 1);              return;          }          if(x.left != null)              backTrack(x.left, sum, curSum, vals);          if(x.right != null)              backTrack(x.right, sum, curSum, vals);         //回溯，结果不同就回溯到上一个节点判断另一条路可不可以          vals.remove(vals.size() - 1);      }  } |

* **重建二叉树（剑指offer第7题）**

输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。

<https://leetcode-cn.com/problems/zhong-jian-er-cha-shu-lcof/solution/er-cha-shu-de-qian-xu-bian-li-fen-zhi-si-xiang-by-/>



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class Solution {      private HashMap<Integer,Integer> map = new HashMap<>();      private int[] preorder;      public TreeNode buildTree(int[] preorder, int[] inorder) {          int length = preorder.length;          this.preorder = preorder;          for(int i = 0;i < length;i++){              map.put(inorder[i],i);          }          return buildTree(0,length - 1,0,length - 1);      }        private TreeNode buildTree(int preL,int preR,int inL,int inR){          if(preL > preR || inL > inR){              return null;          }            int privot = preorder[preL];          TreeNode root = new TreeNode(privot);            int privotIndex = map.get(privot);            root.left = buildTree(preL + 1,preL + (privotIndex - inL),inL,privotIndex -1);          root.right = buildTree(preL + (privotIndex - inL) + 1,preR,privotIndex + 1,inR);          return root;      }  } |