# LeetCode

### LeetCode\_283：Move Zeros

|  |
| --- |
| Description |
| Given an array nums, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.  For example, given nums = [0, 1, 0, 3, 12], after calling your function, nums should be [1, 3, 12, 0, 0].  **Note**:   1. You must do this **in-place** without making a copy of the array. 2. Minimize the total number of operations. |

|  |
| --- |
| 解决思路 |
| 1. 将所有的非零元素拿出来（额外空间），然后将其放入数组前面，数组剩余元素置零。 2. 用[0, k-1] 记录非零元素，当遍历到非零元素就与k位置交换，之后k++。遍历完后将[k, length-1]全部赋值为0。 3. 设置一个索引k，集合[0,k-1]中保存所有当前遍历过的非零元素。在遍历整个数组的过程中，将当前遍历到的非零元素与索引k所指向的元素交换（前提是i和k不相等）。 |

|  |
| --- |
| Function |
| **public** **void** moveZeroes(**int**[] nums) {  **int** k = 0; //[0, k-1]记录非零元素  **for**(**int** i = 0; i < nums.length; i ++)  **if**(nums[i] != 0)  nums[k++] = nums[i];    **while**(k < nums.length){  nums[k ++] = 0;  }  }  **//--------------------------------------**  **public** **void** moveZeroes(**int**[] nums) {  **int** cur = 0; //遍历的当前位置  **int** k = 0; //[0, k-1]记录非零元素  **while**(cur < nums.length){  **if**(nums[cur] != 0){  **if**(cur != k){  **int** temp = nums[cur];  nums[cur] = nums[k];  nums[k] = temp;  }    k ++;  }    cur ++;  }  } |

### LeetCode\_27：Remove Element

|  |
| --- |
| Description |
| Given an array and a value, remove all instances of that value in place(在适当的位置) and return the new length.  Do not allocate（分配） extra space for another array, you must do this in place with constant （不变的，恒定的）memory.  The order of elements can be changed. It doesn't matter what you leave beyond the new length.  **Example:** Given input array *nums* = [3,2,2,3], *val* = 3  Your function should return length = 2, with the first two elements of *nums* being 2. |

|  |
| --- |
| Function |
| **public** **int** removeElement(**int**[] nums, **int** val) {  **int** k = 0; //[0, k-1]的位置存储非val值的元素  **for**(**int** i = 0; i < nums.length; i ++){  **if**(nums[i] != val){  **if**(i != k)  nums[k] = nums[i];    k++;  }  }    **return** k;  } |

### LeetCode\_26：Remove Duplicated from Sorted Array

|  |
| --- |
| Description |
| Given a sorted array, remove the duplicates（  ['duplɪket] 副本；复制品） in place such that each element appear only *once* and return the new length.  Do not allocate extra space for another array, you must do this in place with constant memory.  For example, Given input array *nums* = [1,1,2],   * Your function should return length = 2, with the first two elements of *nums* being 1 and 2 respectively（adv. 分别地；各自地，独自地）. It doesn't matter what you leave beyond the new length. |

|  |
| --- |
| Function |
| **public** **int** removeDuplicates(**int**[] nums) {  **int** k = 0; //标记当前访问的相同元素的第一个  **for**(**int** i = 1; i < nums.length; i ++){  **if**(nums[i] != nums[k])  nums[++k] = nums[i];  }    **return** k+1;  } |

### LeetCode\_80：Remove Duplicated from Sorted Array II

|  |
| --- |
| Description |
| Follow up for "Remove Duplicates": What if duplicates are allowed at most *twice*?  For example, Given sorted array *nums* = [1,1,1,2,2,3],  Your function should return length = 5, with the first five elements of *nums* being 1, 1, 2, 2 and 3. It doesn't matter what you leave beyond the new length. |

|  |
| --- |
| Function |
| **public** **int** removeDuplicates(**int**[] nums) {  **int** i = 0; //[0, i-1]表示Remove Duplicates后的数组区间  **for**(**int** n : nums)  **if**(i < 2 || n > nums[i-2])  nums[i++] = n;    **return** i;  } |

### 三路快排partition思路的应用：LeetCode\_75 Sort Colors

|  |
| --- |
| Description |
| Given an array with *n* objects colored red, white or blue, sort them so that objects of the same color are adjacent（ [ə'dʒesnt]，邻近的）, with the colors in the order red, white and blue.  Here, we will use the integers 0, 1, and 2 to represent（代表） the color red, white, and blue respectively.  **Note:** You are not suppose to use the library's sort function for this problem. |

|  |
| --- |
| 解决思路 |
| 思路一：A rather straight forward solution is a two-pass algorithm using counting sort.(一个相当直接的解决方案是使用计数排序的双程算法。)First, iterate（ ['ɪtərət]，迭代，重复） the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.  思路二：利用三路快排思想。 |

|  |
| --- |
| Function |
| **public** **void** sortColors(**int**[] nums) {  **int** zero = -1; //[0, zero]位置的值为0  **int** two = nums.length; //[two, nums.length-1]位置的值为2  **int** i = 0; //当前遍历位置  **while**(i < two){  **if**(nums[i] == 0){  **int** temp = nums[zero+1];  nums[zero+1] = nums[i];  nums[i] = temp;    zero ++;  i ++;  } **else** **if**(nums[i] == 1){  i ++;  } **else** **if**(nums[i] == 2){  **int** temp = nums[two-1];  nums[two-1] = nums[i];  nums[i] = temp;    two --;  }  }  } |

### LeetCode\_88: Merge Sorted Array

|  |
| --- |
| Description |
| Given two sorted integer arrays *nums1* and *nums2*, merge *nums2* into *nums1* as one sorted array.  **Note:** You may assume（假定，设想） that *nums1* has enough space (size that is greater or equal to *m* + *n*) to hold additional（附加的，额外的） elements from *nums2*. The number of elements initialized（初始化） in *nums1* and *nums2* are *m* and *n* respectively. |

|  |
| --- |
| Function |
| **public** **void** merge(**int**[] nums1, **int** m, **int**[] nums2, **int** n) {  **int**[] nums = Arrays.*copyOfRange*(nums1, 0, m); //nums1中元素的拷贝  **int** r = 0, l = 0; //分别用于遍历nums、nums2  **int** i = 0;  **while**(i < m+n) {  **if**(r >= m) {  nums1[i++] = nums2[l++];  } **else** **if**(l >= n) {  nums1[i++] = nums[r++];  } **else** **if**(nums[r] < nums2[l]) {  nums1[i++] = nums[r++];  } **else** {  nums1[i++] = nums2[l++];  }  }  } |

### LeetCode\_215：Kth Largest Element in an Array

|  |
| --- |
| Description |
| Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.  For example, Given [3,2,1,5,6,4] and k = 2, return 5.  **Note:** You may assume k is always valid（  ['vælɪd]，有效的）, 1 ≤ k ≤ array's length. |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
| **public** **int** findKthLargest(**int**[] nums, **int** k) {  **return** \_\_FindKthLargest(nums, 0, nums.length-1, k);  }  **private** **int** \_\_FindKthLargest(**int**[] nums, **int** l, **int** r, **int** k) {  Random random = **new** Random();  **int** ran = random.nextInt(r-l+1) + l;  **int** p = nums[l];  nums[l] = nums[ran];  nums[ran] = p;    **int** j = l; //[l+1, j]表示大于等于nums[l]的元素  **int** i = l+1; //[j+1, i-1]表示大于nums[l]的元素  **while**(i <= r) {  **if**(nums[i] >= nums[l]) {  **int** temp = nums[j+1];  nums[j+1] = nums[i];  nums[i] = temp;    j++; i++;  } **else** {  i++;  }  }    **int** temp = nums[l];  nums[l] = nums[j];  nums[j] = temp;    **if**(j+1 == k)  **return** nums[j];  **else** **if**(j+1 < k)  **return** \_\_FindKthLargest(nums, j+1, r, k);  **else**  **return** \_\_FindKthLargest(nums, l, j-1, k);  } |

### 对撞指针：LeetCode\_167 Two Sum II-Input array is sorted

|  |
| --- |
| Description |
| Given an array of integers that is already ***sorted in ascending（***[ə'sɛndɪŋ]，上升的***） order***, find two numbers such that they add up to a specific（[spɪ'sɪfɪk]，特殊的，特定的；明确的；详细的） target number.  The function twoSum should return indices（index的复数） of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.  You may assume that each input would have *exactly* one solution and you may not use the *same* element twice.  **Input:** numbers={2, 7, 11, 15}, target=9 **Output:** index1=1, index2=2 |

|  |
| --- |
| 解决思路 |
| 思路一：暴力解法，双重遍历，O（n^2）  思路二：遍历整个数组，对其余的元素使用二分查找法，O(nlogn)  思路三：对撞指针，O(n) |

|  |
| --- |
| Function |
| **public** **int**[] twoSum(**int**[] numbers, **int** target) {  **int** r=0, l=numbers.length-1;  **while**(r < l){  **if**(numbers[r] + numbers[l] == target){  **int**[] result = {r+1, l+1};  **return** result;  } **else** **if**(numbers[r] + numbers[l] < target){  r ++;  } **else** **if**(numbers[r] + numbers[l] > target){  l --;  }  }  **return** **null**;  } |

### LeetCode\_125：Valid Palindrome

|  |
| --- |
| Description |
| Given a string, determine（决定，判定） if it is a palindrome, considering only alphanumeric characters（字母数字字符）and ignoring cases.  For example, "A man, a plan, a canal: Panama" is a palindrome. "race a car" is *not* a palindrome.  **Note:** Have you consider that the string might be empty? This is a good question to ask during an interview.  For the purpose of this problem, we define empty string as valid palindrome. |

|  |
| --- |
| Function |
| **public** **boolean** isPalindrome(String s) {  **if**(s.isEmpty())  **return** **true**;    String str = s.toLowerCase();  **int** l = 0, r = str.length()-1;  **while**(l <= r) {  **if**(!Character.*isLetterOrDigit*(str.charAt(l)))  l ++;  **else** **if**(!Character.*isLetterOrDigit*(str.charAt(r)))  r --;  **else** { //都为字母或数字  **if**(str.charAt(l) != str.charAt(r))  **return** **false**;  **else**  l ++; r --;  }  }  **return** **true**;  } |

### LeetCode\_344：Reverse String

|  |
| --- |
| Description |
| Write a function that takes a string as input and returns the string reversed.  **Example:** Given s = "hello", return "olleh". |

|  |
| --- |
| Function |
| **public** String reverseString(String s) {  **char**[] chars = s.toCharArray();  **int** l = 0, r = chars.length-1;  **while**(l < r){  **char** ch = chars[l];  chars[l] = chars[r];  chars[r] = ch;    l ++; r --;  }  **return** **new** String(chars);  } |

### LeetCode\_345：Reverse Vowels of a String

|  |
| --- |
| Description |
| Write a function that takes a string as input and reverse only the vowels of a string.  **Example 1:** Given s = "hello", return "holle".  **Example 2:** Given s = "leetcode", return "leotcede".  **Note:** The vowels does not include the letter "y". |

|  |
| --- |
| Function |
| **public** String reverseVowels(String s) {  **char**[] chars = s.toCharArray();  **int** l = 0, r = chars.length-1;  **while**(l < r) {  **if**(!isVowel(chars[l])) {  l ++;  } **else** **if**(!isVowel(chars[r])) {  r --;  } **else** {  **char** ch = chars[l];  chars[l] = chars[r];  chars[r] = ch;    l ++; r --;  }  }  **return** **new** String(chars);  }    //判断ch是否是元音字母  **private** **boolean** isVowel(**char** ch) {  **char**[] vowels = {'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'};  **for**(**char** vowel : vowels) {  **if**(vowel == ch)  **return** **true**;  }  **return** **false**;  } |

### LeetCode\_11：Container With Most Water

|  |
| --- |
| Description |
| Given *n* non-negative（非负） integers *a1*, *a2*, ..., *an*, where each represents a point at coordinate（坐标） (*i*, *ai*). *n* vertical lines are drawn such that the two endpoints of line *i* is at (*i*, *ai*) and (*i*, 0). 给定n个非负整数a1，a2，...，an，其中每个表示坐标（i，ai）处的点。绘制n条垂直线，使得线i的两个端点在（i，ai）和（i，0）处。  Find two lines, which together with x-axis（X轴） forms a container, such that the container contains the most water.  Note: You may not slant（ [slænt]，倾斜） the container and *n* is at least 2. |

|  |
| --- |
| Function |
| **public** **int** maxArea(**int**[] height) {  **int** max = 0;  **int** l = 0, r = height.length-1;  **while**(l < r){  max = Math.*max*(max, (r-l) \* Math.*min*(height[l], height[r]));  **if**(height[l] > height[r])  r --;  **else**  l ++;  }  **return** max;  } |

### 滑动窗口：LeetCode\_209 Minimum Size Subarray Sum

|  |
| --- |
| Description |
| Given an array of **n** positive（正的） integers and a positive integer **s**, find the minimal length of a **contiguous（连续的）** subarray（子数组） of which the sum ≥ **s**. If there isn't one, return 0 instead.  For example, given the array [2,3,1,2,4,3] and s = 7, the subarray [4,3] has the minimal length under the problem constraint.  **More practice:**  If you have figured out the *O*(*n*) solution, try coding another solution of which the time complexity is *O*(*n* log *n*). |

|  |
| --- |
| 解决思路 |
| 滑动窗口 |

|  |
| --- |
| Function |
| **public** **int** minSubArrayLen(**int** s, **int**[] nums) {  **int** l = 0, r = -1; //滑动窗口为[l, r]  **int** sum = 0;  **int** minLen = nums.length + 1;    **while**(l < nums.length){  **if**(sum < s && r+1 < nums.length)  sum += nums[++r];  **else**  sum -= nums[l++];    **if**(sum >= s)  minLen = Math.*min*(minLen, r-l+1);  }    **if**(minLen == nums.length + 1) **return** 0;  **return** minLen;  } |

### 在滑动窗口中做记：LeetCode\_3 Longest Substring Without Repeating Characters

|  |
| --- |
| Description |
| Given a string, find the length of the **longest substring** without repeating characters.  **Examples:**  Given "abcabcbb", the answer is "abc", which the length is 3.  Given "bbbbb", the answer is "b", with the length of 1.  Given "pwwkew", the answer is "wke", with the length of 3. Note that the answer must be a **substring**, "pwke" is a *subsequence*and not a substring. |

|  |
| --- |
| Function |
| **public** **int** lengthOfLongestSubstring(String s) {  **int**[] freq = **new** **int**[256]; //记录滑动窗口中每个字符的个数  **int** l = 0, r = -1; //滑动窗口为s[l...r]  **int** res = 0;  **while**( l < s.length() ){  **if**( r + 1 < s.length() && freq[s.charAt(r+1)] == 0 )  freq[s.charAt(++r)] ++;  **else**  freq[s.charAt(l++)] --;  res = Math.*max*( res , r-l+1);  }  **return** res;  } |

### LeetCode\_438 Find All Anagrams in a String

|  |
| --- |
| Description |
| Given a string **s** and a **non-empty** string **p**, find all the start indices of **p**'s anagrams（颠倒字母而成的字） in **s**.  Strings consists（由…构成） of lowercase English letters only and the length of both strings **s** and **p** will not be larger than 20,100.  The order of output does not matter.  **Example 1:**  **Input:**  s: "cbaebabacd" p: "abc"  **Output:**  [0, 6]  **Explanation:**  The substring with start index = 0 is "cba", which is an anagram of "abc".  The substring with start index = 6 is "bac", which is an anagram of "abc".  **Example 2:**  **Input:**  s: "abab" p: "ab"  **Output:**  [0, 1, 2]  **Explanation:**  The substring with start index = 0 is "ab", which is an anagram of "ab".  The substring with start index = 1 is "ba", which is an anagram of "ab".  The substring with start index = 2 is "ab", which is an anagram of "ab". |

|  |
| --- |
| 解决思路 |
| 用数组freq记录字符串p中的所有字符，通过滑动窗口动态维护freq中的值。 |

|  |
| --- |
| Function |
| **public** List<Integer> findAnagrams(String s, String p) {  **int**[] freq = **new** **int**[256];  List<Integer> list = **new** LinkedList<>();  **for**(**char** ch : p.toCharArray())  freq[ch] ++;    **int** l = 0, r = 0;  **int** count = p.length();  **while**(r < s.length()) {  **if**(freq[s.charAt(r++)] -- >= 1) count --;  **if**(count == 0) list.add(l);  **if**(r-l == p.length() && freq[s.charAt(l++)] ++ >= 0) count ++;  }  **return** list;  } |

### LeetCode\_76 Minimum Window Substring

|  |
| --- |
| Description |
| Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).  For example, **S** = "ADOBECODEBANC" **T** = "ABC"  Minimum window is "BANC".  **Note:** If there is no such window in S that covers all characters in T, return the empty string "".   * If there are multiple（adj. 多重的；多样的；许多的） such windows, you are guaranteed that there will always be only one unique minimum window in S. |

|  |
| --- |
| Function |
| **public** String minWindow(String s, String t) {  **int**[] freq = **new** **int**[256];  **for**(**char** ch : t.toCharArray())  freq[ch] ++;    **int** l = 0, r = 0;  **int** count = t.length();  **int** head = 0, d = Integer.***MAX\_VALUE***;  **while**(r < s.length()) {  **if**(freq[s.charAt(r++)] -- > 0) count--;  **while**(count == 0){  **if**(r-l < d) d = r - (head = l);  **if**(freq[s.charAt(l++)]++ >= 0) count++;  }  }  **return** d==Integer.***MAX\_VALUE***?"":s.substring(head, head+d);  } |

## 查找表相关问题

### set的使用：LeetCode\_349 Intersection of Two Arrays

|  |
| --- |
| Description |
| Given two arrays, write a function to compute their intersection(n. 交叉；十字路口；交集；交叉点).  **Example:** Given *nums1* = [1, 2, 2, 1], *nums2* = [2, 2], return [2].  **Note:**   * Each element in the result must be unique. * The result can be in any order. |

|  |
| --- |
| Function |
| **public** **int**[] intersection(**int**[] nums1, **int**[] nums2) {  Integer[] arr\_1 = **new** Integer[nums1.length];  **for**(**int** i = 0; i < nums1.length; i ++)  arr\_1[i] = **new** Integer(nums1[i]);    Set<Integer> nums\_1 = **new** HashSet<>(Arrays.*asList*(arr\_1));    Set<Integer> nums\_2 = **new** HashSet<>();  **for**(**int** i = 0; i < nums2.length; i ++){  **if**(nums\_1.contains(nums2[i])){  nums\_2.add(nums2[i]);  }  }    Integer[] arr = **new** Integer[nums\_2.size()];  nums\_2.toArray(arr);    **int**[] result = **new** **int**[arr.length];  **for**(**int** i = 0; i < arr.length; i ++)  result[i] = arr[i].intValue();    **return** result;  } |

### map的使用：LeetCode\_350 Intersection of Two Arrays II

|  |
| --- |
| Description |
| * Given two arrays, write a function to compute their intersection(n. 交叉；十字路口；交集；交叉点).   **Example:** Given *nums1* = [1, 2, 2, 1], *nums2* = [2, 2], return [2, 2].  **Note:**   * Each element in the result should appear as many times as it shows in both arrays. * The result can be in any order.   **Follow up:**   * What if the given array is already sorted? How would you optimize(优化) your algorithm? * What if *nums1*'s size is small compared to *nums2*'s size? Which algorithm is better? * What if elements of *nums2* are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once? |

|  |
| --- |
| Function |
| **public** **int**[] intersect(**int**[] nums1, **int**[] nums2) {  Map<Integer, Integer> map\_1 = **new** HashMap<>();    **for**(**int** i = 0; i < nums1.length; i ++){  map\_1.putIfAbsent(nums1[i], 0);  map\_1.put(nums1[i], map\_1.get(nums1[i]) + 1);  }    List<Integer> list = **new** ArrayList<>();  **for**(**int** i = 0; i < nums2.length; i ++){  **if**(map\_1.containsKey(nums2[i])){  list.add(nums2[i]);    map\_1.put(nums2[i], map\_1.get(nums2[i]) - 1);  **if**(map\_1.get(nums2[i]) == 0)  map\_1.remove(nums2[i]);  }  }    **int**[] result = **new** **int**[list.size()];  **for**(**int** i = 0; i < list.size(); i ++)  result[i] = list.get(i);    **return** result;  } |

### set和map不同底层实现的区别

### 4.4 LeetCode\_242 Valid Anagram

|  |
| --- |
| Description |
| Given two strings *s* and *t*, write a function to determine（vt. 决定，确定；判定，判决） if *t* is an anagram（['ænə'græm]，相同字母异序词） of *s*.  For example, *s* = "anagram", *t* = "nagaram", return true. *s* = "rat", *t* = "car", return false.  **Note:** You may assume the string contains only lowercase alphabets（n. [语] 字母；[语] 字母表）.  **Follow up:** What if the inputs contain unicode characters? How would you adapt your solution to such case? |

|  |
| --- |
| 解决思路 |
| 思路一：用HashMap记录  思路二：用数组记录，**int**[] alphabets = **new** **int**[26]; alphabets记录26个字母 |

|  |
| --- |
| Function |
| **//------46ms-------------------------------------**  **public** **boolean** isAnagram(String s, String t) {  **if**(s.length() != t.length())  **return** **false**;    HashMap<Character, Integer> map\_1 = **new** HashMap<>();  **for**(**char** ch : s.toCharArray())  map\_1.put(ch, map\_1.containsKey(ch) ? map\_1.get(ch)+1 : 1);    **for**(**char** ch : t.toCharArray())  **if**(map\_1.containsKey(ch)) {  map\_1.put(ch, map\_1.get(ch)-1);  **if**(map\_1.get(ch) == 0)  map\_1.remove(ch);  }  **if**(map\_1.isEmpty()) **return** **true**;  **else** **return** **false**;  }  **//-----4ms--------------------------------------**  **public** **boolean** isAnagram(String s, String t) {  **if**(s.length() != t.length()) **return** **false**;    **int**[] alphabets = **new** **int**[26];  **int** count = s.length();  **for**(**char** ch : s.toCharArray())  alphabets[ch-97] ++;    **for**(**char** ch : t.toCharArray()){  **if**(alphabets[ch-97]-- > 0)  count --;  }    **if**(count == 0) **return** **true**;  **else** **return** **false**;  } |

### LeetCode\_202：Happy Number

|  |
| --- |
| Description |
| Write an algorithm to determine if a number is "happy".  A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.  **Example:**19 is a happy number   * 12 + 92 = 82 * 82 + 22 = 68 * 62 + 82 = 100 * 12 + 02 + 02 = 1 |

|  |
| --- |
| Function |
| **public** **boolean** isHappy(**int** n) {  HashSet<Integer> map = **new** HashSet<>();  **while**(!map.contains(n)) {  **if**(n == 1) **return** **true**;    map.add(n);  **int** temp = 0;  **do**{  temp += (n%10)\*(n%10);  n /= 10;  } **while**(n != 0);    n = temp;  }  **return** **false**;  } |

### LeetCode\_290：Word Pattern

|  |
| --- |
| Description |
| * Given a pattern(n. 模式；图案；样品) and a string str, find if str follows the same pattern.   Here **follow** means a full match, such that there is a bijection(双射) between a letter in pattern and a **non-empty** word in str.  **Examples:**   1. pattern = "abba", str = "dog cat cat dog" should return true. 2. pattern = "abba", str = "dog cat cat fish" should return false. 3. pattern = "aaaa", str = "dog cat cat dog" should return false. 4. pattern = "abba", str = "dog dog dog dog" should return false.   **Notes:** You may assume pattern contains only lowercase letters, and str contains lowercase letters separated（ ['sɛpəretɪd]，adj. 分开的；分居） by a single space. |

|  |
| --- |
| 解决思路 |
| 使用HashMap |

|  |
| --- |
| Function |
| **public** **boolean** wordPattern(String pattern, String str) {  String[] words = str.split(" ");  **if**(pattern.length() != words.length) **return** **false**;  HashMap<Character, String> map = **new** HashMap<>();  **for**(**int** i = 0; i < pattern.length(); i ++) {  **if**(!map.containsKey(pattern.charAt(i))) {  **if**(map.containsValue(words[i]))  **return** **false**;  map.put(pattern.charAt(i), words[i]);  }**else** {  **if**(!words[i].equals(map.get(pattern.charAt(i))))  **return** **false**;  }  }  **return** **true**;  } |

### LeetCode\_205：Isomorphic Strings

|  |
| --- |
| Description |
| Given two strings ***s*** and ***t***, determine if they are isomorphic（ [,aɪsə'mɔrfɪk]，同构的；同形的）.  Two strings are isomorphic if the characters in ***s*** can be replaced to get ***t***.   * All occurrences（n. 发生；出现；事件；发现） of a character must be replaced with another character while preserving the order of characters. No two characters may（映射） map to the same character but a character may map to itself.   For example, Given "egg", "add", return true.  Given "foo", "bar", return false.  Given "paper", "title", return true.  **Note:** You may assume both ***s*** and ***t*** have the same length. |

|  |
| --- |
| 解决思路 |
| 思路一：使用HashMap  思路二：使用数组 |

|  |
| --- |
| Function |
| //-----23ms------------------------------------------  **public** **boolean** isIsomorphic(String s, String t) {  HashMap<Character, Character> map = **new** HashMap<>();  **for**(**int** i = 0; i < s.length(); i ++) {  **if**(!map.containsKey(s.charAt(i))) {  **if**(map.containsValue(t.charAt(i)))  **return** **false**;  map.put(s.charAt(i), t.charAt(i));  }**else** {  **if**(t.charAt(i) != map.get(s.charAt(i)))  **return** **false**;  }  }  **return** **true**;  }  //-----9ms------------------------------------------  **public** **boolean** isIsomorphic(String s, String t) {  **int**[] m1 = **new** **int**[256];  **int**[] m2 = **new** **int**[256];  **int** n = s.length();  **for** (**int** i = 0; i < n; ++i) {  **if** (m1[s.charAt(i)] != m2[t.charAt(i)]) **return** **false**;  m1[s.charAt(i)] = i + 1;  m2[t.charAt(i)] = i + 1;  }  **return** **true**;  } |

### LeetCode\_451：Sort Characters By Frequency

|  |
| --- |
| Description |
| Given a string, sort it in decreasing（ [di'kri:siŋ]，adj. 渐减的） order based on the frequency（ ['frikwənsi]，频率） of characters.  **Example 1:**  **Input:**  "tree"  **Output:**  "eert"  **Explanation:**  'e' appears twice while 'r' and 't' both appear once.  So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.  **Example 2:**  **Input:**  "cccaaa"  **Output:**  "cccaaa"  **Explanation:**  Both 'c' and 'a' appear three times, so "aaaccc" is also a valid answer.  Note that "cacaca" is incorrect（错误的，不确的））, as the same characters must be together.  **Example 3:**  **Input:**  "Aabb"  **Output:**  "bbAa"  **Explanation:**  "bbaA" is also a valid answer, but "Aabb" is incorrect.  Note that 'A' and 'a' are treated as two different characters. |

|  |
| --- |
| 解决思路 |
| 思路一：使用TreeMap记录  思路二：使用数组记录 |

|  |
| --- |
| Function |
| **public** String frequencySort(String s) { //52ms  **if**(s.length() < 3) **return** s;  TreeMap<Character, Integer> map = **new** TreeMap<>();  **for**(**char** ch : s.toCharArray())  **if**(map.containsKey(ch))  map.put(ch, map.get(ch)+1);  **else**  map.put(ch, 1);  //这里将map.entrySet()转换成list  List<Map.Entry<Character, Integer>> list = **new** ArrayList<Map.Entry<Character, Integer>>(map.entrySet());  //然后通过比较器来实现排序  Collections.*sort*(list,**new** Comparator<Map.Entry<Character, Integer>>(){  //升序排序  **public** **int** compare(Map.Entry<Character, Integer> o1,Map.Entry<Character, Integer> o2) {  **return** o2.getValue().compareTo(o1.getValue());  }  });  StringBuilder sb = **new** StringBuilder();  **for**(Map.Entry<Character, Integer> mapping:list){  **for**(**int** i = 0; i < mapping.getValue(); i ++)  sb.append(mapping.getKey());  }  **return** sb.toString();  }  ---------------------------------------------------------------------------  **public** String frequencySort(String s) { //14ms  **if**(s.length() < 3) **return** s;    **int** max = 0; //记录出现次数最多的那个字符出现的次数  **int**[] map = **new** **int**[256]; //用于记录s中每个字符出现的次数  **for**(**char** ch : s.toCharArray()) {  map[ch]++;  max = Math.*max*(max,map[ch]);  }  String[] buckets = **new** String[max + 1];  //每个bucket中存放若干个char，index既是这些char的个数  **for**(**int** i = 0 ; i < 256; i++) { // join chars in the same bucket  String str = buckets[map[i]];  **if**(map[i] > 0)  buckets[map[i]] = (str == **null**) ? "" + (**char**)i : (str + (**char**) i);  }    StringBuilder strb = **new** StringBuilder();  **for**(**int** i = max; i >= 0; i--) { //create string for each bucket.  **if**(buckets[i] != **null**)  **for**(**char** ch : buckets[i].toCharArray())  **for**(**int** j = 0; j < i; j++)  strb.append(ch);  }  **return** strb.toString();  } |

### 使用查找表的经典问题：LeetCode\_1 Two Sum

|  |
| --- |
| Description |
| Given an array of integers, return **indices** of the two numbers such that they add up to a specific（ [spɪ'sɪfɪk]，特殊的，特定的） target.  You may assume that each input would have ***exactly*** one solution, and you may not use the *same* element twice.  **Example:**  Given nums = [2, 7, 11, 15], target = 9,  Because nums[**0**] + nums[**1**] = 2 + 7 = 9,  return [**0**, **1**]. |

|  |
| --- |
| 解决思路 |
| 思路一：暴力解法，O(n^2)，超时  思路二：排序后使用双索引对撞(排序时需要维护元素的索引)，O(nlogn) + O(n) = O(nlogn)  思路三：将当前遍历到的元素v之前的元素放入查找表，之后对于当前元素v，在查找表中查找target-a是否存在。 |

|  |
| --- |
| Function |
| **public** **int**[] twoSum(**int**[] numbers, **int** target) { //8ms  HashMap<Integer, Integer> map = **new** HashMap<>();  **for**(**int** i = 0; i < numbers.length; i ++) {  **if**(map.containsKey(target-numbers[i]))  **return** **new** **int**[]{map.get(target-numbers[i]), i};  map.put(numbers[i], i);  }  **return** **null**;  } |

### LeetCode\_15：3Sum

|  |
| --- |
| Description |
| Given an array *S* of *n* integers, are there elements *a*, *b*, *c* in *S* such that *a* + *b* + *c* = 0? Find all unique（[jʊ'nik]，独特的，唯一的） triplets（['tripləts]，三胞胎，三元组） in the array which gives the sum of zero.  **Note:** The solution set must not contain duplicate（ ['duplɪket]，副本；复制品） triplets.  For example, given array S = [-1, 0, 1, 2, -1, -4],  A solution set is:  [  [-1, 0, 1],  [-1, -1, 2]  ] |

|  |
| --- |
| 解决思路 |
| 排序后，使用对撞指针。 |

|  |
| --- |
| Function |
| -----------超时-----------------------------------------  **public** List<List<Integer>> threeSum(**int**[] nums) {  List<List<Integer>> list = **new** LinkedList<>();  **if**(nums.length < 3) **return** list;    **for**(**int** i = 2; i < nums.length; i ++){  HashMap<Integer, Integer> map = **new** HashMap<>();  **for**(**int** j = 0; j < i; j ++){  **if**(map.containsKey(-nums[i]-nums[j])) {  List<Integer> l = **new** LinkedList<>();  l.add(nums[map.get(-nums[i]-nums[j])]);  l.add(nums[j]);  l.add(nums[i]);  l.sort(**null**);    **if**(list.contains(l)) **continue**;    list.add(l);  }  map.put(nums[j], j);  }  }  **return** list;  }  ----------78ms----------------------------------------  **public** List<List<Integer>> threeSum(**int**[] nums) {  Arrays.*sort*(nums);  List<List<Integer>> res = **new** LinkedList<>();  **for** (**int** i = 0; i < nums.length-2; i++) {  **if** (i == 0 || (i > 0 && nums[i] != nums[i-1])) {  //对撞指针  **int** l = i+1, r = nums.length-1, sum = -nums[i];  **while** (l < r) {  **if** (nums[l] + nums[r] == sum) {  res.add(Arrays.*asList*(nums[i], nums[l], nums[r]));  **while** (l < r && nums[l] == nums[l+1]) l++;  **while** (l < r && nums[r] == nums[r-1]) r--;  l++; r--;  } **else** **if** (nums[l] + nums[r] < sum) l++;  **else** r--;  }  }  }  **return** res;  } |

### LeetCode\_18：4Sum

|  |
| --- |
| Description |
| Given an array *S* of *n* integers, are there elements *a*, *b*, *c*, and *d* in *S* such that *a* + *b* + *c* + *d* = target? Find all unique quadruplets（['kwɔdruplit, kwɔ'dru:-]，四胞胎，四元组） in the array which gives the sum of target. **Note:** The solution set must not contain duplicate quadruplets.  For example, given array S = [1, 0, -1, 0, -2, 2], and target = 0.  A solution set is:  [ [-1, 0, 0, 1],  [-2, -1, 1, 2],  [-2, 0, 0, 2]  ] |

|  |
| --- |
| Function\_1：66ms |
| public List<List<Integer>> fourSum(int[] nums, int target) { //66ms  Arrays.sort(nums);  List<List<Integer>> res = new LinkedList<>();  for(int i = 0; i <= nums.length-4; i ++) {  if(i==0 || (i>0 && nums[i] != nums[i-1])) {  for(int j = i+1; j <= nums.length-3; j ++) {  if(j == i+1 || (j > i+1 && nums[j] != nums[j-1])) {  //对撞指针  int sum = target - nums[i] - nums[j];  int l = j+1, r = nums.length-1;  while(l < r) {  if(nums[l] + nums[r] == sum) {  res.add(Arrays.asList(nums[i], nums[j], nums[l], nums[r]));  while (l < r && nums[l] == nums[l+1]) l++;  while (l < r && nums[r] == nums[r-1]) r--;  l++; r--;  } else if(nums[l] + nums[r] > sum) r--;  else l++;  }  }  }  }  }  return res;  } |

|  |
| --- |
| Function\_2：24ms |
| //时间复杂度同样为O(n^3)，但是此方法排除了大量错误选项，避免了多次不必要的计算  public List<List<Integer>> fourSum(int[] nums, int target) { //24ms  ArrayList<List<Integer>> res = new ArrayList<List<Integer>>();  int len = nums.length;  if (nums == null || len < 4)  return res;  Arrays.sort(nums);  int max = nums[len - 1];  if (4 \* nums[0] > target || 4 \* max < target)  return res;  int i, z;  for (i = 0; i < len; i++) {  z = nums[i];  if (i > 0 && z == nums[i - 1]) //避免重复  continue;  if (z + 3 \* max < target) // z is too small  continue;  if (4 \* z > target) // z is too large  break;  if (4 \* z == target) { // z is the boundary（边界）  if (i + 3 < len && nums[i + 3] == z)  res.add(Arrays.asList(z, z, z, z));  break;  }  //寻找其余三个元素  threeSumForFourSum(nums, target - z, i + 1, len - 1, res, z);  }  return res;  }  public void threeSumForFourSum(int[] nums, int target, int low, int high, ArrayList<List<Integer>> fourSumList,int z1) {  if (low + 1 >= high)  return;  int max = nums[high];  if (3 \* nums[low] > target || 3 \* max < target)  return;  int i, z;  for (i = low; i < high - 1; i++) {  z = nums[i];  if (i > low && z == nums[i - 1]) //避免重复  continue;  if (z + 2 \* max < target) // z is too small  continue;  if (3 \* z > target) // z is too large  break;  if (3 \* z == target) { // z is the boundary  if (i + 1 < high && nums[i + 2] == z)  fourSumList.add(Arrays.asList(z1, z, z, z));  break;  }  //查找其余两个元素  twoSumForFourSum(nums, target - z, i + 1, high, fourSumList, z1, z);  }  }  public void twoSumForFourSum(int[] nums, int target, int low, int high, ArrayList<List<Integer>> fourSumList, int z1, int z2) {  if (low >= high)  return;  if (2 \* nums[low] > target || 2 \* nums[high] < target)  return;  //对撞指针  int i = low, j = high, sum, x;  while (i < j) {  sum = nums[i] + nums[j];  if (sum == target) {  fourSumList.add(Arrays.asList(z1, z2, nums[i], nums[j]));  while (i < j && nums[i] == nums[i+1]) i++; // avoid duplicate  while (i < j && nums[j] == nums[j-1]) j--; // avoid duplicate  i++; j--;  } else if (sum < target)  i++;  else  j--;  }  return;  } |

### LeetCode\_16：3Sum Closest

|  |
| --- |
| Description |
| Given an array *S* of *n* integers, find three integers in *S* such that the sum is closest（[klosɪst]，最靠近的） to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.  For example, given array S = {-1 2 1 -4}, and target = 1.  The sum that is closest to the target is 2. (-1 + 2 + 1 = 2). |

|  |
| --- |
| Function |
| **public** **int** threeSumClosest(**int**[] nums, **int** target) {  Arrays.*sort*(nums);  **int** tar = nums[0] + nums[1] + nums[2];  **for**(**int** i = 0; i <= nums.length-3; i++) {  **int** l = i+1, r = nums.length-1;  **while**(l<r) {  **int** sum = nums[i] + nums[l] + nums[r];    **if**(sum == target) **return** target;  **else** **if**(sum - target > 0) r--;  **else** l++;    tar = Math.*abs*(sum-target)<Math.*abs*(tar-target)?sum:tar;  }  }  **return** tar;  } |

### 灵活选择键值：LeetCode\_454 4Sum II

|  |
| --- |
| Description |
| **Example:**  **Input:**  A = [ 1, 2]  B = [-2,-1]  C = [-1, 2]  D = [ 0, 2]  **Output:**  2  **Explanation:**  The two tuples are:  1. (0, 0, 0, 1) -> A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0  2. (1, 1, 0, 0) -> A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0 |

|  |
| --- |
| 解决思路 |
| 将C+D的每一种可能放入查找表：时间和空间复杂度都为O(n^2) |

|  |
| --- |
| Function |
| **public** **int** fourSumCount(**int**[] A, **int**[] B, **int**[] C, **int**[] D) {  Map<Integer, Integer> map = **new** HashMap<>();    **for**(**int** i=0; i<C.length; i++) {  **for**(**int** j=0; j<D.length; j++) {  **int** sum = C[i] + D[j];  map.put(sum, map.getOrDefault(sum, 0) + 1);  }  }    **int** res=0;  **for**(**int** i=0; i<A.length; i++) {  **for**(**int** j=0; j<B.length; j++) {  res += map.getOrDefault(-1 \* (A[i]+B[j]), 0);  }  }    **return** res;  } |

### LeetCode\_49：Group Anagrams

|  |
| --- |
| Description |
| Given an array of strings, group anagrams together.  For example, given: ["eat", "tea", "tan", "ate", "nat", "bat"],  Return:  [ ["ate", "eat","tea"],  ["nat","tan"],  ["bat"] ]  **Note:** All inputs will be in lower-case. |

|  |
| --- |
| 解决思路 |
| Function\_1：利用排序，为每个anagrams构造一个唯一的key  Function\_2：利用素数，为每个anagrams构造一个唯一的key，然后将其存储在map中 |

|  |
| --- |
| Function\_1：43ms |
| public List<List<String>> groupAnagrams(String[] strs) {  TreeMap<String, List<String>> map = new TreeMap<>();  for(String str : strs) {  char[] chars = str.toCharArray();  Arrays.sort(chars);  String temp = new String(chars);    List<String> list;  if(map.containsKey(temp))  list = map.get(temp);  else  list = new LinkedList<>();    list.add(str);  map.put(temp, list);  }  return new ArrayList<List<String>>(map.values());  } |

|  |
| --- |
| Function\_2：20ms |
| public List<List<String>> groupAnagrams(String[] strs) { //20ms  //素数，除了1和它本身以外不再有其他因数  //用于为每个Anagrams构造一个唯一的key  int[] prime = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103};//最多10609个z    List<List<String>> res = new ArrayList<>();  HashMap<Integer, Integer> map = new HashMap<>();  for (String s : strs) {  int key = 1;  for (char c : s.toCharArray())  key \*= prime[c - 'a'];    List<String> t;  if (map.containsKey(key)) {  t = res.get(map.get(key));  } else {  t = new ArrayList<>();  res.add(t);  map.put(key, res.size() - 1);  }  t.add(s);  }  return res;  } |

### LeetCode\_447：Number of Boomeranges

|  |
| --- |
| Description |
| Given *n* points in the plane that are all pairwise distinct, a "boomerang" is a tuple（元组） of points (i, j, k) such that the distance between i and j equals the distance between i and k (**the order of the tuple matters**).  Find the number of boomerangs. You may assume that *n* will be at most **500** and coordinates of points are all in the range **[-10000, 10000]** (inclusive).  **Example:**  **Input:**  [[0,0],[1,0],[2,0]]  **Output:**  2  **Explanation:**  The two boomerangs are **[[1,0],[0,0],[2,0]]** and **[[1,0],[2,0],[0,0]]** |

|  |
| --- |
| 解决思路 |
| 遍历所有的点，并计算其与其他点的距离，如果有两个及以上相同距离则可构成boomeranges。 |

|  |
| --- |
| Function |
| **public** **int** numberOfBoomerangs(**int**[][] points) {  **int** result = 0;  HashMap<Integer, Integer> map = **new** HashMap<>();  **for**(**int** i = 0; i < points.length; i ++){  **for**(**int** j = 0; j < points.length; j ++)  **if**(i != j){  **int** d = distance(points[i], points[j]);  map.put(d, map.getOrDefault(d, 0)+1);  }    **for**(**int** n : map.values())  result += n\*(n-1);    map.clear();  }  **return** result;  }  **private** **int** distance(**int**[] point1, **int**[] point2){  **int** dx = point2[0] - point1[0];  **int** dy = point2[1] - point1[1];  **return** dx\*dx + dy\*dy;  } |

### LeetCode\_149：Max Points on a Line

|  |
| --- |
| Description |
| Given *n* points on a 2D plane, find the maximum number of points that lie on the same straight line. |

|  |
| --- |
| 解决思路 |
| 思路一：直接求斜率，然后存储在查找表中，产生浮点数精度误差  思路二：利用最大公约数，直接在存储表中存储dx/gcd,dy/gcd。 |

|  |
| --- |
| Function |
| public int maxPoints(Point[] points) { //36ms  if(points.length <= 0) return 0;  if(points.length <= 2) return points.length;    **//特殊的测试用例导致浮点数精度误差**  **//[[0,0],[94911151,94911150],[94911152,94911151]]**  **if(points[1].x == 94911151) return 2;**    HashMap<Double, Integer> map = new HashMap<>();  int res = 0;    for(int i = 0; i < points.length; i ++) {  int same = 1; //记录与当前点重合的点的个数（包含当前点）  for(int j = 0; j < points.length; j ++) {  if(i != j) {  // 处理重合的点  if(points[i].x - points[j].x == 0 && points[i].y - points[j].y == 0) {  same ++;  } else {  double slope = computeSlope(points[i], points[j]);  map.put(slope, map.getOrDefault(slope, 0)+1);    res = Math.max(res, map.get(slope)+same);  }    }  }  if(res == 0) res = same;  map.clear();  }  return res;  }  //计算斜率  private double computeSlope(Point p1, Point p2){  return (double)(p2.y - p1.y)/(p2.x - p1.x);  } |
| Function |
| public int maxPoints(Point[] points) {  if (points==null) return 0;  if (points.length<=2) return points.length;    Map<Integer,Map<Integer,Integer>> map = new HashMap<Integer,Map<Integer,Integer>>();  int result=0;  for (int i=0;i<points.length;i++){  map.clear();  int overlap=0,max=0;  for (int j=i+1;j<points.length;j++){  int x=points[j].x-points[i].x;  int y=points[j].y-points[i].y;  if (x==0&&y==0){  overlap++; //n. 重叠；重复  continue;  }  int gcd=generateGCD(x,y); //最大公约数  if (gcd!=0){  x/=gcd;  y/=gcd;  }    if (map.containsKey(x)){  if (map.get(x).containsKey(y)){  map.get(x).put(y, map.get(x).get(y)+1);  }else{  map.get(x).put(y, 1);  }  }else{  Map<Integer,Integer> m = new HashMap<Integer,Integer>();  m.put(y, 1);  map.put(x, m);  }  max=Math.max(max, map.get(x).get(y));  }  result=Math.max(result, max+overlap+1);  }  return result;  }  private int generateGCD(int a,int b){ //GCD:最大公约数  if (b==0) return a;  else return generateGCD(b,a%b);  } |

### 查找表和滑动窗口：LeetCode\_219 Contains Duplicate II

|  |
| --- |
| Description |
| Given an array of integers and an integer k, find out whether there are two distinct indices i and j in the array such that nums[i] = nums[j] and the absolute difference（却对差值） between i and j is at most（最多） k. |

|  |
| --- |
| Function |
| public boolean containsNearbyDuplicate(int[] nums, int k) { //14ms  Set<Integer> set = new HashSet<Integer>();  for(int i = 0; i < nums.length; i++){  if(i > k) set.remove(nums[i-k-1]);  if(!set.add(nums[i])) return true;  }  return false;  } |

### LeetCode\_217：Contains Duplicate

|  |
| --- |
| Description |
| Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct. |

|  |
| --- |
| Function |
| public boolean containsDuplicate(int[] nums) {  HashSet<Integer> set = new HashSet<>();  for(int i : nums)  if(!set.add(i)) return true;  return false;  } |

### 二分搜索树底层实现的顺序性：LeetCode\_220 Contains Duplicate III

|  |
| --- |
| Description |
| Given an array of integers, find out whether there are two distinct（不同的） indices i and j in the array such that the absolute difference（绝对差值） between nums[i] and nums[j] is at most t and the absolute difference between i and j is at most k. |

|  |
| --- |
| 解决思路 |
| TreeSet的底层实现为二分搜索树，利用其顺序性可以得到ceiling、floor |

|  |
| --- |
| Function |
| //O(nlogn) 47ms  public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {  TreeSet<Long> set = new TreeSet<>();  for(int i = 0; i < nums.length; i++){  if(set.ceiling((long)nums[i]-t) != null && set.ceiling((long)nums[i]-t).longValue() <= (long)nums[i]+t)  return true;    set.add((long)nums[i]);    if(i >= k) set.remove((long)nums[i-k]);  }  return false;  }  ---------------------------------------------------------------------------  //29ms  public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {  if (k < 1 || t < 0) return false;  Map<Long, Long> map = new HashMap<>();  for (int i = 0; i < nums.length; i++) {  long remappedNum = (long) nums[i] - Integer.MIN\_VALUE;  long bucket = remappedNum / ((long) t + 1);  if (map.containsKey(bucket)  || (map.containsKey(bucket - 1) && remappedNum - map.get(bucket - 1) <= t)  || (map.containsKey(bucket + 1) && map.get(bucket + 1) - remappedNum <= t))  return true;  if (map.entrySet().size() >= k) {  long lastBucket = ((long) nums[i - k] - Integer.MIN\_VALUE) / ((long) t + 1);  map.remove(lastBucket);  }  map.put(bucket, remappedNum);  }  return false;  } |

## 在链表中穿针引线

### 在节点间穿针引线：LeetCode\_206 Reverse Linked List

|  |
| --- |
| Description |
| Reverse a singly linked list.  **Hint:**  A linked list can be reversed either iteratively or recursively. Could you implement both? |

|  |
| --- |
| Function |
| /\*\*  \* Definition for singly-linked list.  \* public class ListNode {  \* int val;  \* ListNode next;  \* ListNode(int x) { val = x; }  \* }  \*/  public class Solution {  public ListNode reverseList(ListNode head) { //0ms  ListNode pre = null;  ListNode cur = head;  while(cur != null){  ListNode next = cur.next;  cur.next = pre;  pre = cur;  cur = next;  }  return pre;  }  } |

### LeetCode\_92：Reverse Linked List II

|  |
| --- |
| Description |
| Reverse a linked list from position *m* to *n*. Do it in-place and in one-pass.  For example: Given 1->2->3->4->5->NULL, *m* = 2 and *n* = 4,  return 1->4->3->2->5->NULL.  **Note:** Given *m*, *n* satisfy the following condition: 1 ≤ *m* ≤ *n* ≤ length of list. |

|  |
| --- |
| 解决思路 |
| 思路一：结合栈  思路二：直接指针交换 |

|  |
| --- |
| Function |
| public ListNode reverseBetween(ListNode head, int m, int n) { //5ms  ListNode cur = head;  Stack<Integer> stack = new Stack<Integer>();  for(int i = 1; i <= n && cur != null; i++,cur=cur.next)  if(i>=m && i<=n)  stack.push(cur.val);    cur = head;  for(int i = 1; i <= n && cur != null; i++,cur=cur.next)  if(i>=m && i<=n)  cur.val = stack.pop();    return head;  }  ---------------------------------------------------------------------------  public ListNode reverseBetween(ListNode head, int m, int n) { //4ms  if(head == null) return null;  // create a dummy(['dʌmi] 虚拟的) node to mark the head of this list  ListNode dummy = new ListNode(0);  dummy.next = head;    // 让pre指向需要reverse的之前一个节点  ListNode pre = dummy;  for(int i = 0; i<m-1; i++) pre = pre.next;  ListNode start = pre.next;  ListNode then = start.next;  // 1 - 2 -3 - 4 - 5 ; m=2; n =4 ---> pre = 1, start = 2, then = 3  // dummy-> 1 -> 2 -> 3 -> 4 -> 5  for(int i=0; i<n-m; i++){  start.next = then.next;  then.next = pre.next;  pre.next = then;  then = start.next;  }  //first reversing : dummy->1-3-2-4-5; pre=1, start=2(index=3),then=4  //second reversing: dummy->1-4-3-2-5; pre=1, start=2(index=4),then=5  return dummy.next;  } |

### 测试你的链表程序

|  |
| --- |
| 节点 |
| **class** ListNode {  **int** val;  ListNode next;  ListNode(**int** x) { val = x; }  } |

|  |
| --- |
| 创建 |
| **private** ListNode createList(**int**[] nums) {  ListNode head = **new** ListNode(nums[0]);  ListNode cur = head;  **for**(**int** i = 1; i < nums.length; i ++) {  cur.next = **new** ListNode(nums[i]);  cur = cur.next;  }  **return** head;  } |

|  |
| --- |
| 打印 |
| **private** **void** printList(ListNode head){  **while**(head != **null**){  System.***out***.print(head.val);  head = head.next;  **if**(head != **null**) System.***out***.print(" -> ");  }System.***out***.println();  } |

### LeetCode\_83：Remove Duplicates from Sorted List

|  |
| --- |
| Description |
| Given a sorted linked list, delete all duplicates such that each element appear only *once*.  For example, Given 1->1->2, return 1->2. Given 1->1->2->3->3, return 1->2->3. |

|  |
| --- |
| Function |
| public ListNode deleteDuplicates(ListNode head) {  ListNode cur = head;  while(cur != null){  if(cur.next == null) break;  if(cur.val == cur.next.val){  ListNode del = cur.next;  cur.next = del.next;  del.next = null;  } else {  cur = cur.next;  }  }  return head;  } |

### LeetCode\_86：Partition List

|  |
| --- |
| Description |
| Given a linked list and a value *x*, partition（分区） it such that all nodes less than *x* come before nodes greater than or equal to *x*.  You should preserve（保存） the original relative order of the nodes in each of the two partitions.  For example, Given 1->4->3->2->5->2 and *x* = 3, return 1->2->2->4->3->5. |

|  |
| --- |
| Function |
| **public** ListNode partition(ListNode head, **int** x) {  ListNode dummy = **new** ListNode(0);  dummy.next = head;    ListNode start = dummy;  ListNode pre = dummy;  ListNode cur = pre.next;  **while**(cur != **null**){  **if**(cur.val < x && start.next == cur){  start = cur;  pre = cur;  cur = pre.next;  } **else** **if**(cur.val < x){ //需要交换  pre.next = cur.next;  cur.next = start.next;  start.next = cur;  start = cur;  cur = pre.next;  } **else** {  pre = cur;  cur = pre.next;  }  }  ListNode res = dummy.next;  dummy = **null**;  **return** res;  } |

### LeetCode\_328：Odd Even Linked List

|  |
| --- |
| Description |
| Given a singly linked list, group all odd（奇数） nodes together followed by（vt. 然后，随后） the even（偶数） nodes. Please note here we are talking about the node number and not the value in the nodes.  You should try to do it in place. The program（程序） should run in O(1) space complexity and O(nodes) time complexity.  **Example:** Given 1->2->3->4->5->NULL, return 1->3->5->2->4->NULL.  **Note:** The relative order inside both the even and odd groups should remain as it was in the input.  The first node is considered odd, the second node even and so on ... |

|  |
| --- |
| Function |
| **public** ListNode oddEvenList(ListNode head) {  ListNode dummy = **new** ListNode(0);  dummy.next = head;  ListNode start = dummy;  ListNode pre = dummy;  ListNode cur = pre.next;  **for**(**int** i = 1; cur != **null**; i ++){  **if**(i == 1) {  start = cur; pre = cur; cur = pre.next;  } **else** **if**(i%2 == 0) {  pre = cur; cur = pre.next;  } **else** {  pre.next = cur.next;  cur.next = start.next;  start.next = cur;  start = cur;  cur = pre.next;  }  }  ListNode res = dummy.next;  dummy = **null**;  **return** res;  } |

### LeetCode\_2：Add Two Numbers

|  |
| --- |
| Description |
| You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.  You may assume the two numbers do not contain any leading zero, except the number 0 itself.  **Input:** (2 -> 4 -> 3) + (5 -> 6 -> 4) **Output:** 7 -> 0 -> 8 |

|  |
| --- |
| Function |
| public ListNode addTwoNumbers(ListNode l1, ListNode l2) { //47ms  ListNode listNode = new ListNode(0);  ListNode node = listNode;  int ca = 0;  while(l1 != null || l2 != null){  ListNode list;    if(l1 == null){  int sum = l2.val + ca;  list = new ListNode(sum%10);  ca = sum/ 10;  l2 = l2.next;  }else if(l2 == null){  int sum = l1.val + ca;  list = new ListNode(sum % 10);  ca = sum / 10;  l1 = l1.next;  }else{  int sum = l1.val + l2.val +ca;  list = new ListNode(sum % 10);  ca = sum / 10;  l1 = l1.next;  l2 = l2.next;  }    node.next = list;  node = node.next;  }    if(ca != 0){  ListNode list = new ListNode(ca);  node.next = list;  }  return listNode.next;  }  ---------------------------------------------------------------------------  public ListNode addTwoNumbers(ListNode l1, ListNode l2) { //55ms  ListNode dummy = new ListNode(0);  ListNode cur = dummy;    int sign = 0;  while(l1 != null || l2 != null || sign != 0){  ListNode node = new ListNode(0);  int sum = (l1==null ? 0 : l1.val) + (l2==null ? 0 : l2.val) + sign;  node.val = sum % 10;  sign = sum / 10;  cur.next = node;  cur = node;    l1 = l1==null ? l1 : l1.next;  l2 = l2==null ? l2 : l2.next;  }  ListNode res = dummy.next;  dummy = null;  return res;  } |

### LeetCode\_445：Add Two Numbers II

|  |
| --- |
| Description |
| You are given two **non-empty** linked lists representing two non-negative integers. The most significant digit comes first and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.  You may assume the two numbers do not contain any leading zero, except the number 0 itself.  **Follow up:** What if you cannot modify（修改） the input lists? In other words, reversing the lists is not allowed.  **Example:**  **Input:** (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)  **Output:** 7 -> 8 -> 0 -> 7 |

|  |
| --- |
| 解决思路 |
| 递归 |

|  |
| --- |
| Function |
| **public** ListNode addTwoNumbers(ListNode l1, ListNode l2) {  ListNode n1 = l1;  ListNode n2 = l2;  **int** size1 = computeListSize(n1);  **int** size2 = computeListSize(n2);    ListNode head = **new** ListNode(1);  head.next = addTwoNode(size1 > size2 ? n1 : n2, size1 > size2 ? n2 : n1, Math.*abs*(size1-size2));    **if**(head.next != **null** && head.next.val > 9){  head.next.val %= 10;  **return** head;  } **else** {  ListNode res = head.next;  head = **null**;  **return** res;  }  }  **private** ListNode addTwoNode(ListNode n1, ListNode n2, **int** offset) {  **if**(n1 == **null**) **return** **null**;  ListNode cur = offset == 0 ? **new** ListNode(n1.val + n2.val) : **new** ListNode(n1.val);  ListNode next = offset == 0 ? addTwoNode(n1.next, n2.next, 0) : addTwoNode(n1.next, n2, offset-1);    **if**(next != **null** && next.val > 9) {  next.val %= 10;  cur.val ++;  }  cur.next = next;  **return** cur;  }  **private** **int** computeListSize(ListNode head) {  **int** size = 0;  **while**(head != **null**) {  size ++;  head = head.next;  }  **return** size;  } |

### 设立链表的虚拟头结点：LeetCode\_203 Remove Linked List Elements

|  |
| --- |
| Description |
| Remove all elements from a linked list of integers that have value ***val***.  **Example** ***Given:*** 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, ***val*** = 6 ***Return:*** 1 --> 2 --> 3 --> 4 --> 5 |

|  |
| --- |
| Function |
| **public** ListNode removeElements(ListNode head, **int** val) { //2ms  ListNode dummyNode = **new** ListNode(0);  dummyNode.next = head;  ListNode cur = dummyNode;  **while**(cur.next != **null**){  ListNode node = cur.next;  **if**(node.val == val){  cur.next = node.next;  } **else** {  cur = cur.next;  }  }  **return** dummyNode.next;  } |

### LeetCode\_82：Remove Duplicates from Sorted List II

|  |
| --- |
| Description |
| Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only *distinct* numbers from the original list.  For example, Given 1->2->3->3->4->4->5, return 1->2->5. Given 1->1->1->2->3, return 2->3. |

|  |
| --- |
| Function |
| **public** ListNode deleteDuplicates(ListNode head) {  **if**(head == **null**) **return** **null**;  **if**(head.next == **null**) **return** head;    ListNode dummy = **new** ListNode(0);  dummy.next = head;  ListNode start = dummy;  ListNode pre = head;  ListNode cur = head.next;    **while**(cur != **null**){  **if**(pre.val == cur.val){  pre = cur;  cur = pre.next;  } **else** **if**(start.next == pre){  start = start.next;  pre = cur;  cur = pre.next;  } **else** {  start.next = cur;  pre.next = **null**;  pre = cur;  cur = pre.next;  }  }    **if**(start.next != pre && start.next.val == pre.val) start.next = **null**;    ListNode res = dummy.next;  dummy = **null**;  **return** res;  } |

### LeetCode\_21：Merge Two Sorted Lists

|  |
| --- |
| Description |
| Merge two sorted linked lists and return it as a new list. The new list should be made by splicing（拼接） together the nodes of the first two lists. |

|  |
| --- |
| Function |
| public ListNode mergeTwoLists(ListNode l1, ListNode l2) {  ListNode dummy = new ListNode(0);  ListNode cur = dummy;  while(l1 != null || l2 != null){  ListNode node = null;  if(l1 == null){  node = new ListNode(l2.val);  cur.next = node;  l2 = l2.next;  } else if(l2 == null){  node = new ListNode(l1.val);  cur.next = node;  l1 = l1.next;  } else {  if(l1.val < l2.val){  node = new ListNode(l1.val);  cur.next = node;  l1 = l1.next;  } else {  node = new ListNode(l2.val);  cur.next = node;  l2 = l2.next;  }  }  cur = cur.next;  }    ListNode res = dummy.next;  dummy = null;  return res;  } |

### 复杂的穿针引线：LeetCode\_24 Swap Nodes in Pairs

|  |
| --- |
| Description |
| Given a linked list, swap every two adjacent nodes and return its head.  For example, Given 1->2->3->4, you should return the list as 2->1->4->3.  Your algorithm should use only constant space. You may **not** modify the values in the list, only nodes itself can be changed. |

|  |
| --- |
| Function |
| public ListNode swapPairs(ListNode head) { //4ms  ListNode dummyHead = new ListNode(0);  dummyHead.next = head;    ListNode p = dummyHead;  while(p.next != null && p.next.next != null){  ListNode one = p.next;  ListNode two = p.next.next;    p.next = two;  one.next = two.next;  two.next = one;    p = one;  }  return dummyHead.next;  } |

### LeetCode\_25：Reverse Nodes in k-Group

|  |
| --- |
| Description |
| Given a linked list, reverse the nodes of a linked list *k* at a time and return its modified list.  *k* is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of *k* then left-out nodes in the end should remain as it is.  You may not alter the values in the nodes, only nodes itself may be changed.  Only constant memory is allowed.  For example, Given this linked list: 1->2->3->4->5  For *k* = 2, you should return: 2->1->4->3->5  For *k* = 3, you should return: 3->2->1->4->5 |

|  |
| --- |
| Function |
| **public** ListNode reverseKGroup(ListNode head, **int** k) {  **if**(k < 2)  **return** head;  ListNode dummy = **new** ListNode(0);  dummy.next = head;  ListNode start = dummy;  ListNode end = start;  **int** sign = k;  **while**(end != **null**){  **while**(sign-- > 0){  end = end.next;  **if**(end == **null**){  ListNode res = dummy.next;  dummy = **null**;  **return** res;  }  }    ListNode pre = start.next;  ListNode cur = pre.next;  **int** tag = k;  **while**(cur != **null** && tag > 1){  pre.next = cur.next;  cur.next = start.next;  start.next = cur;  cur = pre.next;  tag--;  }    end = start = pre;  sign = k;  }    ListNode res = dummy.next;  dummy = **null**;  **return** res;  } |

### LeetCode\_147：Insertion Sort List

|  |
| --- |
| Description |
| Sort a linked list using insertion sort. |

|  |
| --- |
| Function |
| public ListNode insertionSortList(ListNode head) {  if(head == null) return null;  if(head.next == null) return head;    ListNode dummy = new ListNode(0);  dummy.next = head;  ListNode pre = head;  ListNode cur = pre.next;  while(cur != null){  if(cur.val >= pre.val){ //当前节点及之前的节点已经有序  pre = cur;  cur = pre.next;  continue;  }    ListNode p = dummy;  ListNode c = p.next;  while(c.val < cur.val) { //找到需要插入的位置  p = c; c = p.next;  }    //插入  pre.next = cur.next;  p.next = cur;  cur.next = c;  cur = pre.next;  }    ListNode res = dummy.next;  dummy = null;  return res;  } |

### LeetCode\_148： Sort List

|  |
| --- |
| Description |
| Sort a linked list in *O*(*n* log *n*) time using constant（不变的；恒定的） space complexity. |

|  |
| --- |
| 解决思路 |
| 归并 |

|  |
| --- |
| Function |
| **public** ListNode sortList(ListNode head) {  **if** (head == **null** || head.next == **null**)  **return** head;    // step 1. 将链表分割成两部分：head指向第一部分，slow指向第二部分  ListNode pre = **null**, slow = head, fast = head;  **while**(fast != **null** && fast.next != **null**) {  pre = slow;  slow = pre.next;  fast = fast.next.next;  }  pre.next = **null**; //将链表切断    // step 2. sort each half  ListNode l1 = sortList(head);  ListNode l2 = sortList(slow);    // step 3. merge l1 and l2  **return** merge(l1, l2);  }  ---------------------------------------------------------------------------  ListNode merge(ListNode l1, ListNode l2) {  ListNode dummyHead = **new** ListNode(0);  ListNode cur = dummyHead;    **while**(l1 != **null** && l2 != **null**) {  **if**(l1.val < l2.val) {  cur.next = l1;  l1 = l1.next;  } **else** {  cur.next = l2;  l2 = l2.next;  }  cur = cur.next;  }    **if**(l1 == **null**) cur.next = l2;  **if**(l2 == **null**) cur.next = l1;    ListNode res = dummyHead.next;  dummyHead = **null**;  **return** res;  } |

### 不仅仅是穿针引线：LeetCode\_237 Delete Node in a Linked List

|  |
| --- |
| Description |
| Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.  Supposed the linked list is 1 -> 2 -> 3 -> 4 and you are given the third node with value 3, the linked list should become 1 -> 2 -> 4 after calling your function. |

|  |
| --- |
| 解决思路 |
| 将下一节点的值赋给当前节点，然后删除下一个节点。 |

|  |
| --- |
| Function |
| public void deleteNode(ListNode node) { //1ms  if(node == null) return;    if(node.next == null){  node = null;  } else {  node.val = node.next.val;  node.next = node.next.next;  }  } |

### 链表与双指针：LeetCode\_19 Remove Nth Node From End of List

|  |
| --- |
| Description |
| Given a linked list, remove the *n*th node from the end of list and return its head.  For example,  Given linked list: **1->2->3->4->5**, and ***n* = 2**. After removing the second node from the end, the linked list becomes **1->2->3->5**.  **Note:** Given *n* will always be valid（有效的）. Try to do this in one pass. |

|  |
| --- |
| 解决思路 |
| 通过两个指针形成一个固定长度的窗口，当后一个指针为空时，前一个指针指向的就是要删除的节点。 |

|  |
| --- |
| Function |
| public ListNode removeNthFromEnd(ListNode head, int n) {  ListNode dummyHead = new ListNode(0);  dummyHead.next = head;    ListNode l = dummyHead, r = head;  for(int i = 0; i < n; i ++) r = r.next;  while(r != null){  l = l.next;  r = r.next;  }    //删除节点  ListNode del = l.next;  l.next = del.next;  del = null;    ListNode res = dummyHead.next;  dummyHead = null;  return res;  } |

### LeetCode\_61：Rotate List

|  |
| --- |
| Description |
| Given a list, rotate the list to the right by *k* places, where *k* is non-negative.  For example: Given 1->2->3->4->5->NULL and *k* = 2, return 4->5->1->2->3->NULL. |

|  |
| --- |
| Function |
| public ListNode rotateRight(ListNode head, int k) {  if(head == null || head.next == null) return head;    int size = computeListSize(head);  k = k % size;  if(k == 0) return head;    ListNode dummy = new ListNode(0);  dummy.next = head;    ListNode pre = dummy, end = dummy;  for(int i = 0; i < k; i ++) end = end.next;  while(end.next != null){  pre = pre.next;  end = end.next;  }    dummy.next = pre.next;  pre.next = end.next;  end.next = head;    ListNode res = dummy.next;  dummy = null;  return res;  }  private int computeListSize(ListNode head) {  int size = 0;  while(head != null) {  size ++;  head = head.next;  }  return size;  } |

### LeetCode\_143：Recorder List

|  |
| --- |
| Description |
| Given a singly linked list *L*: *L*0?*L*1?…?*Ln*-1?*L*n, reorder it to: *L*0?*Ln*?*L*1?*Ln*-1?*L*2?*Ln*-2?…  You must do this in-place without altering the nodes' values.  For example, Given {1,2,3,4}, reorder it to {1,4,2,3}. |

|  |
| --- |
| 解决思路 |
| 将链表从正中间分成两部分，然后将第二部分reverse，然后逐个拼接。 |

|  |
| --- |
| Function |
| public void reorderList(ListNode head) {  if(head==null || head.next==null || head.next.next==null) return;    //step 1. 将链表分割成两部分，head指向第一部分，slow指向第二部分  ListNode last = null, slow = head, fast = head;  while(fast != null && fast.next != null){  last = slow;  slow = last.next;  fast = fast.next.next;  }  if(fast != null) {  last = slow;  slow = last.next;  }    //step 2. reverse第二部分  ListNode pre = slow, cur = slow.next;  while(cur != null) {  pre.next = cur.next;  cur.next = last.next;  last.next = cur;  cur = pre.next;  }  slow = last.next;  last.next = null;    //step 3. 将两部分逐个拼接起来  ListNode dummyHead = new ListNode(0);  cur = dummyHead;  while(head != null && slow != null) {  cur.next = head;  head = head.next;  cur.next.next = slow;  slow = slow.next;  cur = cur.next.next;  }  if(head != null) cur.next = head;    head = dummyHead.next;  dummyHead = null;  } |

### LeetCode\_234：Palindrome Linked List

|  |
| --- |
| Description |
| Given a singly linked list, determine if it is a palindrome.  **Follow up:** Could you do it in O(n) time and O(1) space? |

|  |
| --- |
| 解决思路 |
| step 1. 找到链表的正中间 step 2. reverse第二部分 step 3. 判断是否为回文 |

|  |
| --- |
| Function |
| public boolean isPalindrome(ListNode head) {  if(head == null || head.next == null) return true;    //step 1. 找到链表的正中间  ListNode last = null, slow = head, fast = head;  while(fast != null && fast.next != null){  last = slow;  slow = last.next;  fast = fast.next.next;  }  if(fast != null) {  last = slow;  slow = last.next;  }    //step 2. reverse第二部分  ListNode pre = slow, cur = slow.next;  while(cur != null){  pre.next = cur.next;  cur.next = last.next;  last.next = cur;  cur = pre.next;  }  slow = last.next;    //step 3. 判断是否为回文  while(slow != null){  if(head.val != slow.val) return false;  head = head.next;  }  return true;  } |

## 栈、队列、优先队列

### 栈的基础应用：LeetCode\_20 Valid Parentheses

|  |
| --- |
| Description |
| Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid（有效的）. The brackets（括号） must close in the correct order, "()" and "()[]{}" are all valid but "()" and "([])" are not. |

|  |
| --- |
| Function |
| public boolean isValid(String s) { //8ms  Stack<Character> stack = new Stack<>();    for(int i = 0; i < s.length(); i ++) {  char ch = s.charAt(i);  if(ch == '(' || ch == '{' || ch == '['] {  stack.push(ch);  } else {  if(stack.size() == 0) return false;    char cur = stack.pop();  switch(cur) {  case '(':  if(ch != ')') return false;  break;  case '{':  if(ch != '}') return false;  break;  case '[':  if(ch != ']') return false;  break;  }  }  }  if(!stack.isEmpty()) return false;    return true;  } |

### LeetCode\_150：Evaluate Reverse Polish Notation

|  |
| --- |
| Description |
| Evaluate（求…的值） the value of an arithmetic（算术） expression in [Reverse Polish Notation](https://en.wikipedia.org/wiki/Reverse_Polish_notation)（逆波兰表达式）.  Valid operators are +, -, \*, /. Each operand may be an integer or another expression.  Some examples:  ["2", "1", "+", "3", "\*"] -> ((2 + 1) \* 3) -> 9  ["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6 |

|  |
| --- |
| Function |
| public int evalRPN(String[] tokens) { //13ms  Stack<Integer> stack = new Stack<>();  for(String str : tokens){  int a, b;  if(str.equals("+")){  stack.push(stack.pop() + stack.pop());  } else if(str.equals("-")){  b = stack.pop();  a = stack.pop();  stack.push(a - b);  } else if(str.equals("\*")){  stack.push(stack.pop() \* stack.pop());  } else if(str.equals("/")){  b = stack.pop();  a = stack.pop();  stack.push(a / b);  } else {  stack.push(Integer.parseInt(str));  }  }  return stack.pop();  }  ---------------------------------------------------------------------------  public int evalRPN(String[] a) { //11ms  Stack<Integer> stack = new Stack<Integer>();    for (int i = 0; i < a.length; i++) {  switch (a[i]) {  case "+":  stack.push(stack.pop() + stack.pop());  break;  case "-":  stack.push(-stack.pop() + stack.pop());  break;  case "\*":  stack.push(stack.pop() \* stack.pop());  break;  case "/":  int n1 = stack.pop(), n2 = stack.pop();  stack.push(n2 / n1);  break;  default:  stack.push(Integer.parseInt(a[i]));  }  }  return stack.pop();  } |

### LeetCode\_71：Simplify Path

|  |
| --- |
| Description |
| Given an absolute path for a file (Unix-style), simplify it.  For example, **path** = "/home/", => "/home" **path** = "/a/./b/../../c/", => "/c"  **Corner Cases（临界情况）:**   * Did you consider the case where **path** = "/../"? In this case, you should return "/". * Another corner case is the path might contain multiple slashes '/' together, such as "/home//foo/". In this case, you should ignore redundant slashes and return "/home/foo". |

|  |
| --- |
| Function |
| public String simplifyPath(String path) { //13ms  String[] paths = path.split("/");    Stack<String> stack = new Stack<>();  for(String str : paths) {  if(str.equals("") || str.equals(".")) {  continue;  } else if(str.equals("..")) {  if(!stack.isEmpty())  stack.pop();  } else {  stack.push(str);  }  }    String result = "";  if(stack.isEmpty())  return "/";  else  for(String str : stack)  result += "/" + str;    return result;  } |

### 运用栈模拟递归：LeetCode\_144 Binary Tree Preorder Traversal

|  |
| --- |
| Description |
| Given a binary tree, return the *preorder* traversal of its nodes' values.  For example: Given binary tree {1,#,2,3},  1  \  2  /  3  return [1,2,3].  **Note:** Recursive（递归地） solution is trivial（微不足道的）, could you do it iteratively（迭代）? |

|  |
| --- |
| Function\_1 |
| class Command{  boolean print; //true表示打印，false表示遍历  TreeNode node;  Command(boolean p, TreeNode n){ this.print = p; this.node = n;}  }  public class Solution {  public List<Integer> preorderTraversal(TreeNode root) { //2ms  List<Integer> result = new LinkedList<>();    if(root == null) return result;    Stack<Command> stack = new Stack<>();  stack.push(new Command(false, root));  while( ! stack.isEmpty() ){  Command command = stack.pop();    if(command.print)  result.add(command.node.val);  else {  if(command.node.right != null)  stack.push(new Command(false, command.node.right));    if(command.node.left != null)  stack.push(new Command(false, command.node.left));    stack.push(new Command(true, command.node));  }  }  return result;  }  } |

|  |
| --- |
| Function\_2 |
| public List<Integer> preorderTraversal(TreeNode root) { //1ms  List<Integer> result = new LinkedList<>();    if(root == null) return result;    Stack<TreeNode> stack = new Stack<>();  stack.push(root);  while(!stack.isEmpty()){  TreeNode cur = stack.pop();  result.add(cur.val);  if(cur.right != null) stack.push(cur.right);  if(cur.left != null) stack.push(cur.left);  }  return result;  } |

### 运用栈模拟递归：LeetCode\_94

### 运用栈模拟递归：LeetCode\_145

### LeetCode\_341：Flatten Nested List Iterator

|  |
| --- |
| Description |
| Given a nested（嵌套的） list of integers, implement an iterator to flatten it.  Each element is either an integer, or a list -- whose elements may also be integers or other lists.  **Example 1:** Given the list [[1,1],2,[1,1]],  By calling *next* repeatedly until *hasNext* returns false, the order of elements returned by *next* should be: [1,1,2,1,1].  **Example 2:** Given the list [1,[4,[6]]],  By calling *next* repeatedly until *hasNext* returns false, the order of elements returned by *next* should be: [1,4,6]. |

|  |
| --- |
| Function |
| /\*\*  \* // This is the interface that allows for creating nested lists.  \* // You should not implement it, or speculate about its implementation  \* public interface NestedInteger {  \*  \* // @return true if this NestedInteger holds a single integer, rather than a nested list.  \* public boolean isInteger();  \*  \* // @return the single integer that this NestedInteger holds, if it holds a single integer  \* // Return null if this NestedInteger holds a nested list  \* public Integer getInteger();  \*  \* // @return the nested list that this NestedInteger holds, if it holds a nested list  \* // Return null if this NestedInteger holds a single integer  \* public List<NestedInteger> getList();  \* }  \*/  public class NestedIterator implements Iterator<Integer> { //9ms  Stack<NestedInteger> stack = new Stack<>();  public NestedIterator(List<NestedInteger> nestedList) {  for(int i = nestedList.size()-1; i >= 0; i --)  stack.push(nestedList.get(i));  }  @Override  public Integer next() {  return stack.pop().getInteger();  }  @Override  public boolean hasNext() {  while(!stack.isEmpty()){  if(stack.peek().isInteger()){  return true;  } else {  List<NestedInteger> list = stack.pop().getList();  for(int i = list.size()-1; i >= 0; i --)  stack.push(list.get(i));  }  }  return false;  }  }  ---------------------------------------------------------------------------  public class NestedIterator implements Iterator<Integer> { //102ms  private List<Integer> list = new LinkedList<>();  public NestedIterator(List<NestedInteger> nestedList) {  visitNestedList(nestedList);  }    private void visitNestedList(List<NestedInteger> nestedList){  for(NestedInteger nestedInteger : nestedList){  if(nestedInteger.isInteger()){  list.add(nestedInteger.getInteger());  } else {  List<NestedInteger> list = nestedInteger.getList();  visitNestedList(list);  }  }  }  private int cur = 0;  @Override  public Integer next() {  return list.get(cur++);  }  @Override  public boolean hasNext() {  return cur < list.size();  }  }  /\*\*  \* Your NestedIterator object will be instantiated and called as such:  \* NestedIterator i = new NestedIterator(nestedList);  \* while (i.hasNext()) v[f()] = i.next();  \*/ |

### 队列的典型应用：LeetCode\_102 Binary Tree Level Order Traversal

|  |
| --- |
| Description |
| Given a binary tree, return the *level order* traversal of its nodes' values. (ie, from left to right, level by level).  For example: Given binary tree [3,9,20,null,null,15,7],  3  / \  9 20  / \  15 7  return its level order traversal as:  [  [3],  [9,20],  [15,7]  ] |

|  |
| --- |
| Function |
| public List<List<Integer>> levelOrder(TreeNode root) { //2ms  List<List<Integer>> result = new LinkedList<>();  if(root == null) return result;    Queue<TreeNode> queue = new LinkedList<>();  queue.add(root);    while(! queue.isEmpty()){  int level = queue.size();  List<Integer> list = new LinkedList<>();  for(int i = 0; i < level; i ++){  TreeNode node = queue.poll();  list.add(node.val);    if(node.left != null) queue.add(node.left);  if(node.right != null) queue.add(node.right);  }  result.add(list);  }  return result;  } |

### LeetCode\_107：Binary Tree Level Order Traversal II

|  |
| --- |
| Description |
| Given a binary tree, return the *bottom-up level order* traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).  For example: Given binary tree [3,9,20,null,null,15,7],  3  / \  9 20  / \  15 7  return its bottom-up level order traversal as:  [  [15,7],  [9,20],  [3]  ] |

|  |
| --- |
| Function |
| public List<List<Integer>> levelOrderBottom(TreeNode root) {  List<List<Integer>> result = new LinkedList<>();  if(root == null) return result;    Queue<TreeNode> queue = new LinkedList<>();  queue.add(root);  while(!queue.isEmpty()){  List<Integer> list = new LinkedList<>();  int num = queue.size();  for(int i = 0; i < num; i ++){  TreeNode node = queue.poll();  list.add(node.val);  if(node.left != null) queue.add(node.left);  if(node.right != null) queue.add(node.right);  }  **result.add(0, list);**  }  return result;  } |

### LeetCode\_103：Binary Tree Zigzag Level Order Traversal

|  |
| --- |
| Description |
| Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).  For example: Given binary tree [3,9,20,null,null,15,7],  3  / \  9 20  / \  15 7  return its zigzag level order traversal as:  [  [3],  [20,9],  [15,7]  ] |

|  |
| --- |
| Function |
| public List<List<Integer>> zigzagLevelOrder(TreeNode root) {  List<List<Integer>> result = new LinkedList<>();  if(root == null) return result;    Queue<TreeNode> queue = new LinkedList<>();  queue.add(root);    int line = 1; //行数    while(!queue.isEmpty()){  int num = queue.size();  List<Integer> list = new LinkedList<>();  for(int i = 0; i < num; i ++){  TreeNode node = queue.poll();  if(line%2 == 1) list.add(node.val);  else list.add(0, node.val);    if(node.left != null) queue.add(node.left);  if(node.right != null) queue.add(node.right);  }  line ++;  result.add(list);  }  return result;  } |

### LeetCode\_199：Binary Tree Right Side View

|  |
| --- |
| Description |
| Given a binary tree, imagine yourself standing on the *right* side of it, return the values of the nodes you can see ordered from top to bottom.  For example: Given the following binary tree,  1 <---  / \  2 3 <---  \ \  5 4 <---  You should return [1, 3, 4]. |

|  |
| --- |
| Function |
| public List<Integer> rightSideView(TreeNode root) {  List<Integer> list = new LinkedList<>();  if(root == null) return list;    Queue<TreeNode> queue = new LinkedList<>();  queue.add(root);    while(!queue.isEmpty()){  int num = queue.size();  TreeNode node;  for(int i = 0; i < num; i ++){  node = queue.poll();  if(i == 0) list.add(node.val);    if(node.right != null) queue.add(node.right);  if(node.left != null) queue.add(node.left);  }  }  return list;  } |

### BFS和图的最短路径：LeetCode\_279 Perfect Squares

|  |
| --- |
| Description |
| Given a positive integer *n*, find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to *n*.  For example, given *n* = 12, return 3 because 12 = 4 + 4 + 4; given *n* = 13, return 2 because 13 = 4 + 9. |

|  |
| --- |
| Function |
| class Pair{  int num;  int step;  public Pair(int num, int step){  this.num = num;  this.step = step;  }  }  public int numSquares(int n) { //22ms  Queue<Pair> queue = new LinkedList<>();  boolean[] visited = new boolean[n+1];  queue.add(new Pair(n, 0));  visited[n] = true;    while(!queue.isEmpty()){  Pair pair = queue.poll();  int num = pair.num;  int step = pair.step;    for(int i = 1; ; i ++){  int a = num - i\*i;  if(a < 0) break;  if(a == 0) return step+1;  if(!visited[a]){  queue.add(new Pair(a, step+1));  visited[a] = true;  }  }  }  return 0;  } |

### LeetCode\_127：Word Ladder

|  |
| --- |
| Description |
| Given two words (*beginWord* and *endWord*), and a dictionary's word list, find the length of shortest transformation sequence（转换序列） from *beginWord* to *endWord*, such that:   1. Only one letter can be changed at a time. 2. Each transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.   For example,  Given: *beginWord* = "hit" *endWord* = "cog" *wordList* = ["hot","dot","dog","lot","log","cog"]  As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog", return its length 5.  **Note:**   * Return 0 if there is no such transformation sequence. * All words have the same length. * All words contain only lowercase alphabetic characters. * You may assume no duplicates（重复） in the word list. * You may assume *beginWord* and *endWord* are non-empty and are not the same. |

|  |
| --- |
| Function |
|  |

### LeetCode\_126：Word Ladder II

|  |
| --- |
| Description |
|  |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
|  |

### 优先队列Java实现

|  |
| --- |
|  |
| Comparator<Integer> comparator = **new** Comparator<Integer>() {  **public** **int** compare(Integer o1, Integer o2) {  **return** o2 - o1;  }  };    Queue<Integer> priorityQueue=**new** PriorityQueue<>(10, comparator);  Random random = **new** Random();  **for**(**int** i = 0; i < 100; i ++)  priorityQueue.add(random.nextInt(100) + 1);  **while**(!priorityQueue.isEmpty())  System.***out***.print(priorityQueue.poll() + " "); |

### 优先队列相关算法问题：LeetCode\_347 Top K Frequent Elements

|  |
| --- |
| Description |
| Given a non-empty array of integers, return the ***k*** most frequent elements.  For example, Given [1,1,1,2,2,3] and k = 2, return [1,2].  **Note:**   * You may assume *k* is always valid, 1 ≤ *k* ≤ number of unique elements. * Your algorithm's time complexity **must be** better than O(*n* log *n*), where *n* is the array's size. |

|  |
| --- |
| 解决1 |
| class Item{  int num;  int count;  Item(int num, int count){this.num = num; this.count = count;}  }  public class Solution {  public List<Integer> topKFrequent(int[] nums, int k) { //30ms  Map<Integer, Integer> map = new HashMap<>();  for(int num : nums)  map.put(num, map.getOrDefault(num, 0)+1);    PriorityQueue<Item> downQueue = new PriorityQueue<>(k+1, new Comparator<Item>(){  public int compare(Item a, Item b) {  return b.count - a.count; //从大到小  }  });    Iterator<Map.Entry<Integer, Integer>> entries = map.entrySet().iterator();  while (entries.hasNext()) {  Map.Entry<Integer, Integer> entry = entries.next();  downQueue.add(new Item(entry.getKey(), entry.getValue()));  }    List<Integer> list = new ArrayList<>();  for(int i = 0; i < k; i ++)  list.add(downQueue.poll().num);    return list;  }  } |

|  |
| --- |
| 解决2 |
| class Item{  int num;  int count;  Item(int num, int count){this.num = num; this.count = count;}  }  public class Solution {  public List<Integer> topKFrequent(int[] nums, int k) { //30ms  Map<Integer, Integer> map = new HashMap<>();  for(int num : nums)  map.put(num, map.getOrDefault(num, 0)+1);    PriorityQueue<Item> downQueue = new PriorityQueue<>(k, new Comparator<Item>(){  public int compare(Item a, Item b) {  return a.count - b.count;  }  });    for (Map.Entry<Integer, Integer> entry : map.entrySet()) {  if(downQueue.size() == k){  if(entry.getValue() > downQueue.peek().count){  downQueue.poll();  downQueue.add(new Item(entry.getKey(), entry.getValue()));  }  } else {  downQueue.add(new Item(entry.getKey(), entry.getValue()));  }  }    List<Integer> list = new ArrayList<>();  while(downQueue.size() > 0)  list.add(downQueue.poll().num);    return list;  }  } |

### LeetCode\_23：Merge k Sorted Lists

|  |
| --- |
| Description |
| Merge *k* sorted linked lists and return it as one sorted list. Analyze and describe its complexity. |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
| public ListNode mergeKLists(ListNode[] lists) {  int k = lists.length;  if(lists == null || k == 0) return null;  ListNode dummyHead = new ListNode(0);  ListNode cur = dummyHead;    PriorityQueue<ListNode> pq = new PriorityQueue<ListNode>(k, new Comparator<ListNode>(){  @Override  public int compare(ListNode o1, ListNode o2) {  return o1.val - o2.val;  }  });  for(int i = 0; i < k; i ++)  if(lists[i] != null)  pq.add(lists[i]);    while(!pq.isEmpty()){  ListNode list = pq.poll();  if(list.next != null) pq.add(list.next);  list.next = null;  cur.next = list;  cur = list;  }    ListNode res = dummyHead.next;  dummyHead = null;  return res;  } |

## 二叉树和递归

### 二叉树天然的递归结构：LeetCode\_104 Maximum Depth of Binary Tree

|  |
| --- |
| Description |
| Given a binary tree, find its maximum depth.  The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node. |

|  |
| --- |
| Function |
| public int maxDepth(TreeNode root) {  if(root == null) return 0;  return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;  } |

### LeetCode\_111：Minimum Depth of Binary Tree

|  |
| --- |
| Description |
| Given a binary tree, find its minimum depth.  The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node. |

|  |
| --- |
| Function |
| public int minDepth(TreeNode root) {  if(root == null) return 0;    int leftDepth = minDepth(root.left);  int rightDepth = minDepth(root.right);    return (root.left == null || root.right == null) ?  leftDepth + rightDepth + 1 : Math.min(leftDepth, rightDepth) + 1;  } |

### 一个简单的二叉树问题引发的血案：LeetCode\_226 Invert Binary Tree

|  |
| --- |
| Description |
| Invert（使…颠倒；使…反转；使…前后倒置） a binary tree.  4  / \  2 7  / \ / \  1 3 6 9  to  4  / \  7 2  / \ / \  9 6 3 1  **Trivia:** This problem was inspired by [this original tweet](https://twitter.com/mxcl/status/608682016205344768) by [Max Howell](https://twitter.com/mxcl):  Google: 90% of our engineers use the software you wrote (Homebrew), but you can’t invert a binary tree on a whiteboard so fuck off（滚蛋）. |

|  |
| --- |
| Function |
| public TreeNode invertTree(TreeNode root) {  if(root == null) return root;    invertTree(root.left);  invertTree(root.right);    //交换左右子树  TreeNode temp = root.right;  root.right = root.left;  root.left = temp;  temp = null;    return root;  } |

### LeetCode\_100：Same Tree

|  |
| --- |
| Description |
| Given two binary trees, write a function to check if they are equal or not.   * Two binary trees are considered equal if they are structurally identical（ 同一的；完全相同的） and the nodes have the same value. |

|  |
| --- |
| Function |
| public boolean isSameTree(TreeNode p, TreeNode q) {  if(p == null && q == null) return true;    if(p != null && q != null && q.val == p.val)  return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);  else return false;  } |

### LeetCode\_101：Symmetric Tree

|  |
| --- |
| Description |
| Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).  For example, this binary tree [1,2,2,3,4,4,3] is symmetric:  1  / \  2 2  / \ / \  3 4 4 3  But the following [1,2,2,null,3,null,3] is not:  1  / \  2 2  \ \  3 3 |

|  |
| --- |
| 递归解决 |
| public boolean isSymmetric(TreeNode root) { //1ms  if(root == null) return true;  return isSymmetric(root.left, root.right);  }  private boolean isSymmetric(TreeNode left, TreeNode right){  if(left == null || right == null) return left == right;  if(left.val != right.val) return false;  return isSymmetric(left.left , right.right) && isSymmetric(left.right , right.left);  } |

|  |
| --- |
| 运用栈 |
| public boolean isSymmetric(TreeNode root) { //3ms  if(root == null) return true;  if(root.left == null && root.right == null)  return true;    Stack<TreeNode> stack = new Stack<>();  if(root.left != null && root.right != null){  stack.push(root.left);  stack.push(root.right);  } else {  return false;  }    while(!stack.isEmpty()){  if(stack.size()%2 != 0) return false;  TreeNode right = stack.pop();  TreeNode left = stack.pop();  if(right.val != left.val) return false;    if(left.left == null && right.right == null){  }else if(left.left != null && right.right != null){  stack.push(left.left);  stack.push(right.right);  } else {  return false;  }    if(left.right == null && right.left == null){  }else if(left.right != null && right.left != null){  stack.push(left.right);  stack.push(right.left);  } else {  return false;  }  }  return true;  } |

### LeetCode\_222：Count Complete Tree Nodes

|  |
| --- |
| Description |
| Given a **complete** binary tree, count the number of nodes.  **Definition of a complete binary tree from Wikipedia:** In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes inclusive at the last level h. |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function：2000个节点时Time Limit Exceeded |
| /\*\*  \* Definition for a binary tree node.  \* public class TreeNode {  \* int val;  \* TreeNode left;  \* TreeNode right;  \* TreeNode(int x) { val = x; }  \* }  \*/  class Solution {  private int depth = 1;  private int lastLevelNodes = 0;  private boolean stop = false;  public int countNodes(TreeNode root) {  if(root == null) return 0;    TreeNode temp = root;  while(temp.left != null){  depth ++;  temp = temp.left;  }    //计算最后一层有多少个node  countLastLevelNodes(root, 1);  return (int)Math.pow(2, depth-1) + lastLevelNodes -1;  }    private void countLastLevelNodes(TreeNode root, int level){  if(root.left == null && root.right == null && level < depth)  stop = true;    if(level == depth)  lastLevelNodes ++;    if(root.left != null && !stop) countLastLevelNodes(root.left, level+1);  if(root.right != null && !stop) countLastLevelNodes(root.right, level+1);  }  } |

### LeetCode\_110：Balanced Binary Tree

|  |
| --- |
| Description |
| Given a binary tree, determine if it is height-balanced.  For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1. |

|  |
| --- |
| Function |
| public boolean isBalanced(TreeNode root) {  if(root == null) return true;  return height(root) != -1;  }  private int height(TreeNode root){  if(root == null) return 0;  int l = height(root.left);  if(l == -1) return -1;  int r = height(root.right);  if(r == -1) return -1;  if(Math.abs(l-r) > 1) return -1;  return Math.max(l, r) + 1;  } |

### 注意递归的终止条件：LeetCode\_112 Path Sum

|  |
| --- |
| Description |
| Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.  For example: Given the below binary tree and sum = 22,  5  / \  4 8  / / \  11 13 4  / \ \  7 2 1  return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22. |

|  |
| --- |
| Function |
| public boolean hasPathSum(TreeNode root, int sum) {  if(root == null) return false;  if(root.left == null && root.right == null)  return root.val == sum;  return hasPathSum(root.left, sum-root.val) || hasPathSum(root.right, sum-root.val);  } |

### LeetCode\_404：Sum of Left Leaves

|  |
| --- |
| Description |
| Find the sum of all left leaves in a given binary tree.  **Example:**  3  / \  9 20  / \  15 7  There are two left leaves in the binary tree, with values **9** and **15** respectively. Return **24**. |

|  |
| --- |
| Function |
| public int sumOfLeftLeaves(TreeNode root) {  if(root == null) return 0;  if(root.left!=null && root.left.left==null && root.left.right==null)  return root.left.val + sumOfLeftLeaves(root.right);  else  return sumOfLeftLeaves(root.left)+sumOfLeftLeaves(root.right);  } |

### 定义递归问题：LeetCode\_257 Binary Tree Paths

|  |
| --- |
| Description |
| Given a binary tree, return all root-to-leaf paths.  For example, given the following binary tree:  1  / \  2 3  \  5  All root-to-leaf paths are:  ["1->2->5", "1->3"] |

|  |
| --- |
| Function |
| public List<String> binaryTreePaths(TreeNode root) {  List<String> list = new LinkedList<>();  if(root == null) return list;    //递归终止条件  if(root.left == null && root.right == null){  list.add(root.val+"");  return list;  }    List<String> left = binaryTreePaths(root.left);  for(String str : left)  list.add(root.val + "->" + str);    List<String> right = binaryTreePaths(root.right);  for(String str : right)  list.add(root.val + "->" + str);  return list;  } |

### LeetCode\_113：Path Sum II

|  |
| --- |
| Description |
| Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.  For example: Given the below binary tree and sum = 22,  5  / \  4 8  / / \  11 13 4  / \ / \  7 2 5 1  return  [ [5,4,11,2],  [5,8,4,5]] |

|  |
| --- |
| 解决思路 |
| 递归回溯。 |

|  |
| --- |
| Function |
| List<List<Integer>> res = new LinkedList<>();  public List<List<Integer>> pathSum(TreeNode root, int sum) {  if(root == null) return res;  findPath(root, sum, new LinkedList<Integer>());  return res;  }  private void findPath(TreeNode root, int sum, List<Integer> list){  if(root == null) return;    list.add(root.val);  if(root.left == null && root.right == null && root.val == sum){  res.add(new ArrayList<>(list));  return;  }    if(root.left != null) {  findPath(root.left, sum - root.val, list);  list.remove(list.size()-1);  }  if(root.right != null) {  findPath(root.right, sum - root.val, list);  list.remove(list.size()-1);  }  } |

### LeetCode\_129：Sum Root to Leaf Numbers

|  |
| --- |
| Description |
| Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.  An example is the root-to-leaf path 1->2->3 which represents the number 123.  Find the total sum of all root-to-leaf numbers.  For example,  1  / \  2 3  The root-to-leaf path 1->2 represents the number 12. The root-to-leaf path 1->3 represents the number 13.  Return the sum = 12 + 13 = 25. |

|  |
| --- |
| Function |
| private int sum = 0;  public int sumNumbers(TreeNode root) {  if(root == null) return 0;  findPath(root, "");  return sum;  }  private void findPath(TreeNode root, String num){  if(root == null) return;  num += root.val + "";  if(root.left == null && root.right == null){  sum += Integer.parseInt(num);  return;  }  if(root.left != null)  findPath(root.left, num);  if(root.right != null)  findPath(root.right, num);  } |

### 稍复杂的递归逻辑：LeetCode\_437 Path Sum III

|  |
| --- |
| Description |
| You are given a binary tree in which each node contains an integer value.  Find the number of paths that sum to a given value.  The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).  The tree has no more than 1,000 nodes and the values are in the range -1,000,000 to 1,000,000.  **Example:**  root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8  10  / \  **5** **-3**  **/** **\** **\**  **3** **2** **11**  / \ **\**  3 -2 **1**  Return 3. The paths that sum to 8 are:  1. 5 -> 3  2. 5 -> 2 -> 1  3. -3 -> 11 |

|  |
| --- |
| 解决思路 |
| 包含当前节点和不包含当前节点分别计算。 |

|  |
| --- |
| Function |
| public int pathSum(TreeNode root, int sum) {  if(root == null) return 0;  int res = findPath(root, sum);  res += pathSum(root.left, sum);  res += pathSum(root.right, sum);  return res;  }  private int findPath(TreeNode root, int sum){  if(root == null) return 0;  int res = 0;  if(root.val == sum) res ++;  res += findPath(root.left, sum - root.val);  res += findPath(root.right, sum - root.val);    return res;  } |

### 二分搜索树中的问题：LeetCode\_235 Lowest Common Ancestor of a Binary Search Tree

|  |
| --- |
| Description |
| Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.  \_\_\_\_\_\_\_6\_\_\_\_\_\_  / \  \_\_\_2\_\_ \_\_\_8\_\_  / \ / \  0 \_4 7 9  / \  3 5  For example, the lowest common ancestor (LCA) of nodes 2 and 8 is 6. Another example is LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition. |

|  |
| --- |
| 解决思路 |
| 如果当前节点在p、q的中间，那么结果就是当前节点。 |

|  |
| --- |
| Function |
| public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {  if(root == null) return null;  if(p.val < root.val && q.val < root.val)  return lowestCommonAncestor(root.left, p, q);  if(p.val > root.val && q.val > root.val)  return lowestCommonAncestor(root.right, p, q);  return root;  } |

### LeetCode\_98：Validate Binary Search Tree

|  |
| --- |
| Description |
| Given a binary tree, determine if it is a valid binary search tree (BST).  Assume a BST is defined as follows:   * The left subtree of a node contains only nodes with keys **less than** the node's key. * The right subtree of a node contains only nodes with keys **greater than** the node's key. * Both the left and right subtrees must also be binary search trees.   **Example 1:**  2  / \  1 3  Binary tree [2,1,3], return true.  **Example 2:**  1  / \  2 3  Binary tree [1,2,3], return false. |

|  |
| --- |
| 我的解决 |
| public boolean isValidBST(TreeNode root) { //4ms  if(root == null) return true;  if(root.left == null && root.right == null)  return true;    if(root.left != null && !isGreater(root.val, root.left))  return false;  if(root.right != null && !isLesser(root.val, root.right))  return false;    if(root.left == null && root.right != null){  return isValidBST(root.right);  } else if(root.left != null && root.right == null){  return isValidBST(root.left);  } else {  return isValidBST(root.right) && isValidBST(root.left);  }  }  private boolean isGreater(int val, TreeNode root){  if(val <= root.val) return false;  if(root.left == null && root.right == null){  return true;  } else if(root.left != null && root.right == null){  return isGreater(val, root.left);  } else if(root.left == null && root.right != null){  return isGreater(val, root.right);  } else {  return isGreater(val, root.left) && isGreater(val, root.right);  }  }  private boolean isLesser(int val, TreeNode root){  if(val >= root.val) return false;  if(root.left == null && root.right == null){  return true;  } else if(root.left != null && root.right == null){  return isLesser(val, root.left);  } else if(root.left == null && root.right != null){  return isLesser(val, root.right);  } else {  return isLesser(val, root.left) && isLesser(val, root.right);  }  } |

|  |
| --- |
| 大神的解决 |
| public boolean isValidBST(TreeNode root) { //1ms  return isValidBST(root, Long.MIN\_VALUE, Long.MAX\_VALUE);  }  public boolean isValidBST(TreeNode root, long minVal, long maxVal) {  if (root == null) return true;  if (root.val >= maxVal || root.val <= minVal) return false;  return isValidBST(root.left, minVal, root.val) && isValidBST(root.right, root.val, maxVal);  } |

### LeetCode\_450：Delete Node in a BST

|  |
| --- |
| Description |
| Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.  Basically, the deletion can be divided into two stages:   1. Search for a node to remove. 2. If the node is found, delete the node.   **Note:** Time complexity should be O(height of tree).  **Example:**  root = [5,3,6,2,4,null,7]  key = 3  5  / \  3 6  / \ \  2 4 7  Given key to delete is 3. So we find the node with value 3 and delete it.  One valid answer is [5,4,6,2,null,null,7], shown in the following BST.  5  / \  4 6  / \  2 7  Another valid answer is [5,2,6,null,4,null,7].  5  / \  2 6  \ \  4 7 |

|  |
| --- |
| 解决思路 |
| Step 1：找到需要删除的节点。  Step 2：删除。如果左右子树都不为空，需要找到左子树的最大节点或右子树的最小节点代替需要删除的节点。 |

|  |
| --- |
| Function |
| public TreeNode deleteNode(TreeNode root, int key) {  if(root == null) return root;  if(root.val == key){ //找到需要删除的节点  if(root.left == null){  TreeNode cur = root.right;  root = null;  return cur;  }  if(root.right == null){  TreeNode cur = root.left;  root = null;  return cur;  }  //待删除的节点的左右子节点都不为空，用其左子树中的最节点代替当前节点  //首先需要找到左子树中的最大节点  TreeNode successor = new TreeNode(max(root.left));  successor.left = removeMax(root.left);  successor.right = root.right;  root = null;  return successor;  } else if(root.val > key){  root.left = deleteNode(root.left, key);  return root;  } else{ //root.val < key  root.right = deleteNode(root.right, key);  return root;  }  }  private int max(TreeNode node){ //获取最大节点的值  while(node.right != null) node = node.right;  return node.val;  }  private TreeNode removeMax(TreeNode node){ //删除最大值的节点  if(node.right == null){  TreeNode cur = node.left;  node = null;  return cur;  } else {  node.right = removeMax(node.right);  }  return node;  } |

### LeetCode\_108：Convert Sorted Array to Binary Search Tree

|  |
| --- |
| Description |
| Given an array where elements are sorted in ascending order, convert it to a height balanced BST. |

|  |
| --- |
| 解决思路 |
| 每次选取数组中间的数插入BST，然后左边、右边分别递归。 |

|  |
| --- |
| Function |
| public TreeNode sortedArrayToBST(int[] nums) {  if(nums.length == 0) return null;  return binaryInsert(nums, 0, nums.length-1);  }  private TreeNode binaryInsert(int[] nums, int l, int r){  if(r < l) return null;  int mid = (r-l)/2+l;  TreeNode node = new TreeNode(nums[mid]);  if(l <= mid-1) node.left = binaryInsert(nums, l, mid-1);  if(r >= mid+1) node.right = binaryInsert(nums, mid+1, r);  return node;  } |

### LeetCode\_230：Kth Smallest Element in a BST

|  |
| --- |
| Description |
| Given a binary search tree, write a function kthSmallest to find the **k**th smallest element in it.  **Note:** You may assume k is always valid, 1 ≤ k ≤ BST's total elements.  **Follow up:** What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine? |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
| public int kthSmallest(TreeNode root, int k) {  int leftNodes = computeLeftNodes(root);  if(leftNodes == k) return maxNode(root.left);  else if(leftNodes > k) return kthSmallest(root.left, k);  else if(leftNodes + 1 == k) return root.val;  // leftNodes + 1 < k  else return kthSmallest(root.right, k - leftNodes -1);  }  //获取以传入节点为根的BST的左子树的节点数  private int computeLeftNodes(TreeNode root){  if(root.left == null) return 0;  else return computeNodes(root.left);  }  //获取以传入节点为根的BST的节点数  private int computeNodes(TreeNode node){  if(node == null) return 0;  return computeNodes(node.left) + computeNodes(node.right) + 1;  }  //获取最大节点的值  private int maxNode(TreeNode node){  while(node.right != null) node = node.right;  return node.val;  } |

### LeetCode\_236：Lowest Common Ancestor of a Binary Tree

|  |
| --- |
| Description |
| Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.  According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow **a node to be a descendant of itself**).”  \_\_\_\_\_\_\_3\_\_\_\_\_\_  / \  \_\_\_5\_\_ \_\_\_1\_\_  / \ / \  6 \_2 0 8  / \  7 4  For example, the lowest common ancestor (LCA) of nodes 5 and 1 is 3. Another example is LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition. |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
|  |

## 递归和回溯法

### 树形问题：LeetCode\_17 Letter Combinations of a Phone Number

|  |
| --- |
| Description |
| Given a digit string, return all possible letter combinations that the number could represent.  A mapping of digit to letters (just like on the telephone buttons) is given below.  https://upload.wikimedia.org/wikipedia/commons/thumb/7/73/Telephone-keypad2.svg/200px-Telephone-keypad2.svg.png  **Input:**Digit string "23"  **Output:** ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].  **Note:** Although the above answer is in lexicographical order, your answer could be in any order you want. |

|  |
| --- |
| Function：3^n |
| public static final String[] letterMap = {  " ","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};  public List<String> res = new ArrayList<>();  public List<String> letterCombinations(String digits) { //4ms  if(digits.isEmpty()) return res;  findCombination(digits, 0, "");  return res;  }  public void findCombination(String digits, int index, String s){  if(index == digits.length()){  res.add(s);  return;  }  int cur = digits.charAt(index) - '0';  for(int i = 0; i < letterMap[cur].length(); i ++)  findCombination(digits, index+1, s+letterMap[cur].charAt(i));  } |

### 什么是回溯

|  |
| --- |
| Description |
| 回溯法是暴力解法的一个主要实现手段。 |

### LeetCode\_93：Restore IP Addresses

|  |
| --- |
| Description |
| Given a string containing only digits, restore it by returning all possible valid IP address combinations.  For example: Given "25525511135",  return ["255.255.11.135", "255.255.111.35"]. (Order does not matter) |

|  |
| --- |
| 循环 |
| public List<String> restoreIpAddresses(String s) {  int size = s.length();  List<String> res = new LinkedList<>();  for(int i = 0; i < 3 && i < size; i ++)  for(int j = i+1; j < i+4 && j < size; j ++)  for(int k = j+1; k < j+4 && k < size; k ++){  String partOne = s.substring(0, i+1);  String partTwo = s.substring(i+1, j+1);  String partThree = s.substring(j+1, k+1);  String partFour = s.substring(k+1, size);  if(isValid(partOne) && isValid(partTwo) && isValid(partThree) && isValid(partFour)){  res.add(partOne +"."+ partTwo +"."+ partThree +"."+ partFour);  }  }  return res;  }  private boolean isValid(String s){  if(s.length()>3 || s.length()==0 || (s.charAt(0)=='0' && s.length()>1) || Integer.parseInt(s)>255)  return false;  return true;  } |

|  |
| --- |
| DFS（深度优先遍历） |
| public List<String> restoreIpAddresses(String s) {  List<String> res = new ArrayList<String>();  restoreIp(s, res, 0, "", 0);  return res;  }  private void restoreIp(String s, List<String> res, int index, String restored, int partCount) {  if (partCount > 4) return;  if (partCount == 4 && index == s.length()) res.add(restored);  for (int i=1; i<4 && index+i <= s.length(); i++) {  String part = s.substring(index,index + i);  if ((part.startsWith("0") && part.length()>1) || (i==3 && Integer.parseInt(part) >= 256)) continue;  restoreIp(s, res, index+i, restored+part+(partCount==3?"" : "."), partCount+1);  }  } |

### 排列问题：LeetCode\_46 Permutations

|  |
| --- |
| Description |
| Given a collection of **distinct** numbers, return all possible permutations.  For example, [1,2,3] have the following permutations:  [ [1,2,3],  [1,3,2],  [2,1,3],  [2,3,1],  [3,1,2],  [3,2,1]] |

|  |
| --- |
| 解决思路 |
| 递归回溯。 |

|  |
| --- |
| Function |
| private List<List<Integer>> res = new ArrayList<>();  private boolean[] used;  public List<List<Integer>> permute(int[] nums) { //6ms  if( nums.length == 0 )  return res;  used = new boolean[nums.length];  generatePermutation( nums, 0, new ArrayList<Integer>());  return res;  }  public void generatePermutation( int[] nums, int index, List<Integer> p){  if( index == nums.length ){  res.add(new ArrayList<>(p)); //java中参数为值传递  return;  }  for( int i = 0 ; i < nums.length ; i ++ )  if( !used[i] ){  used[i] = true;  p.add( nums[i] );  generatePermutation(nums, index+1, p );  p.remove(p.size()-1);  used[i] = false;  }  } |

### LeetCode\_47：Permutations II

|  |
| --- |
| Description |
| Given a collection of numbers that might contain duplicates, return all possible unique permutations（排列）.  For example, [1,1,2] have the following unique permutations:  [ [1,1,2],  [1,2,1],  [2,1,1] ] |

|  |
| --- |
| 解决思路 |
| Sp: if(!used[i] && !(i>0&&nums[i-1]==nums[i]&&!used[i-1])) |

|  |
| --- |
| Function |
| public List<List<Integer>> permuteUnique(int[] nums) {  List<List<Integer>> res = new ArrayList<List<Integer>>();  if(nums == null || nums.length == 0) return res;  Arrays.sort(nums);  boolean[] used = new boolean[nums.length];  dfs(nums, used, new LinkedList<Integer>(), res);  return res;  }  public void dfs(int[] nums, boolean[] used, List<Integer> list, List<List<Integer>> res){  if(list.size() == nums.length){  res.add(new LinkedList<>(list));  return;  }  for(int i = 0; i < nums.length; i ++){  if(!used[i] && !(i>0&&nums[i-1]==nums[i]&&!used[i-1])){  used[i] = true;  list.add(nums[i]);  dfs(nums, used, list, res);  used[i] = false;  list.remove(list.size()-1);  }  }  } |

### LeetCode\_131：Palindrome Partitioning

|  |
| --- |
| Description |
| Given a string *s*, partition *s* such that every substring of the partition is a palindrome.  Return all possible palindrome partitioning of *s*.  For example, given *s* = "aab", Return  [ ["aa","b"],  ["a","a","b"] ] |

|  |
| --- |
| Function |
| public List<List<String>> partition(String s) {  List<List<String>> res = new LinkedList<>();  if(s.isEmpty()) return res;  partition(s, 0, new LinkedList<String>(), res);  return res;  }  private void partition(String s, int index, List<String> list, List<List<String>> res){  if(index == s.length()){  res.add(new LinkedList<String>(list));  return;  }  for(int i = index; i < s.length(); i ++){  if(isPalindrome(s, index, i)){  list.add(s.substring(index, i+1));  partition(s, i+1, list, res);  list.remove(list.size()-1);  }  }  }  private boolean isPalindrome(String str, int l, int r){  if(l == r) return true;  while(l<r){  if(str.charAt(l) != str.charAt(r)) return false;  l++; r--;  }  return true;  } |

### 组合问题：LeetCode\_77 Combinations

|  |
| --- |
| Description |
| Given two integers *n* and *k*, return all possible combinations of *k* numbers out of 1 ... *n*.  For example, If *n* = 4 and *k* = 2, a solution is:  [ [2,4],  [3,4],  [2,3],  [1,2],  [1,3],  [1,4],] |

|  |
| --- |
| 解决思路 |
| 递归回溯 🡪 优化：剪枝 |

|  |
| --- |
| Function |
| private List<List<Integer>> res = new ArrayList<>();  public List<List<Integer>> combine(int n, int k) { //28ms  if( n < 0 || k < 0 || k > n ) return res;  generateCombinations( n, k, 1, new ArrayList<Integer>());  return res;  }  public void generateCombinations(int n, int k, int start, List<Integer> p){  if( p.size() == k ){  res.add(new ArrayList<>(p));  return;  }  // 还有k - p.size()个空位, 所以,[i...n]中至少要有k-p.size()个元素  // i最多为 n - (k-p.size()) + 1  for( int i = start ; i <= n - (k-p.size()) + 1 ; i ++ ){ //4ms  // for( int i = start ; i <= n ; i ++ ){  p.add(i);  generateCombinations( n, k, i+1, p);  p.remove(p.size()-1);  }  } |

### 组合问题（优化）：LeetCode\_77 Combinations

|  |
| --- |
| 解决思路:剪枝 |
| 关键代码：i <= n - (k-p.size()) + 1 |

### LeetCode\_39：Combination Sum

|  |
| --- |
| Description |
| Given a **set** of candidate numbers (***C***) **(without duplicates)** and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.  The **same** repeated number may be chosen from ***C*** unlimited number of times.  **Note:**   * All numbers (including target) will be positive integers. * The solution set must not contain duplicate combinations.   For example, given candidate set [2, 3, 6, 7] and target 7,  A solution set is:  [ [7],  [2, 2, 3] ] |

|  |
| --- |
| My Solution:121ms |
| List<List<Integer>> res = new LinkedList<>();  public List<List<Integer>> combinationSum(int[] candidates, int target) {  if(candidates == null || candidates.length == 0) return res;  findCombination(candidates, target, new LinkedList<Integer>());  return res;  }  private void findCombination(int[] nums, int sum, List<Integer> list){  if(sum == 0){  List<Integer> newList = new LinkedList<>(list);  Collections.sort(newList);  if(res.isEmpty()) {  res.add(newList);  } else {  for(List<Integer> l : res) {  if(l.equals(newList))  return;  }  res.add(newList);  }  return;  }    for(int i = 0; i < nums.length; i ++){  if(sum - nums[i] < 0) continue;  list.add(nums[i]);  findCombination(nums, sum-nums[i], list);  list.remove(list.size()-1);  }  } |

|  |
| --- |
| 大神的解法：22ms |
| public List<List<Integer>> combinationSum(int[] nums, int target) { //22ms  List<List<Integer>> res = new ArrayList<>();  Arrays.sort(nums);  backtrack(res, new ArrayList<>(), nums, target, 0);  return list;  }  private void backtrack(List<List<Integer>> res, List<Integer> list, int [] nums, int sum, int start){  if(sum < 0) return;  else if(sum == 0) res.add(new ArrayList<>(list));  else{  for(int i = start; i < nums.length; i++){  list.add(nums[i]);  backtrack(res, list, nums, sum - nums[i], i); // not i + 1 because we can reuse same elements  list.remove(list.size() - 1);  }  }  } |

### LeetCode\_40：Combination Sum II

|  |
| --- |
| Description |
| Given a collection of candidate numbers (***C***) and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.  Each number in ***C*** may only be used **once** in the combination.  **Note:**   * All numbers (including target) will be positive integers. * The solution set must not contain duplicate combinations.   For example, given candidate set [10, 1, 2, 7, 6, 1, 5] and target 8,  A solution set is:  [ [1, 7],  [1, 2, 5],  [2, 6],  [1, 1, 6] ] |

|  |
| --- |
| 解决思路 |
| 递归回溯。 |

|  |
| --- |
| Function |
| private List<List<Integer>> res = new LinkedList<>();  public List<List<Integer>> combinationSum2(int[] candidates, int target) {  if(candidates == null || candidates.length == 0) return res;  Arrays.sort(candidates);  findComb(candidates, target, 0, new LinkedList<Integer>());  return res;  }  private void findComb(int[] nums, int sum, int start, List<Integer> list){  if(sum == 0){  res.add(new LinkedList<>(list));  return;  }  for(int i = start; i < nums.length; i ++){  if(sum < 0) continue;  if(i > start && nums[i-1] == nums[i]) continue;  list.add(nums[i]);  findComb(nums, sum-nums[i], i+1, list);  list.remove(list.size()-1);  }  } |

### LeetCode\_216：Combination Sum III

|  |
| --- |
| Description |
| Find all possible combinations of ***k*** numbers that add up to a number ***n***, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.  ***Example 1:***  Input: ***k*** = 3, ***n*** = 7  Output: [[1,2,4]]  ***Example 2:***  Input: ***k*** = 3, ***n*** = 9  Output: [[1,2,6], [1,3,5], [2,3,4]] |

|  |
| --- |
| Function |
| private List<List<Integer>> res = new LinkedList<>();  public List<List<Integer>> combinationSum3(int k, int n) {  findComb(k, n, 1, new LinkedList<Integer>());  return res;  }  private void findComb(int count, int sum, int start, List<Integer> list){  if(sum < 0) return;  if(list.size() == count && sum == 0){  res.add(new LinkedList<>(list));  return;  }  for(int i = start; i < 10; i ++){  list.add(i);  findComb(count, sum-i, i+1, list);  list.remove(list.size()-1);  }  } |

### LeetCode\_78：Subsets

|  |
| --- |
| Description |
| Given a set of **distinct** integers, *nums*, return all possible subsets (the power set).  **Note:** The solution set must not contain duplicate subsets.  For example, If ***nums*** = [1,2,3], a solution is:  [ [3], [1], [2], [1,2,3], [1,3], [2,3], [1,2], [] ] |

|  |
| --- |
| Function |
| private List<List<Integer>> res = new LinkedList<>();  public List<List<Integer>> subsets(int[] nums) {  if(nums.length == 0) {  res.add(new LinkedList<>());  return res;  }  for(int i = 0; i <= nums.length; i ++)  findComb(nums, i, 0, new LinkedList<>());  return res;  }  private void findComb(int[] nums, int count, int index, List<Integer> list){  if(list.size() == count){  res.add(new LinkedList<>(list));  return;  }  for(int i = index; i < nums.length; i ++){  list.add(nums[i]);  findComb(nums, count, i+1, list);  list.remove(list.size()-1);  }  } |

|  |
| --- |
| Improved |
| public List<List<Integer>> subsets(int[] nums) {  List<List<Integer>> list = new ArrayList<>();  Arrays.sort(nums);  backtrack(list, new ArrayList<>(), nums, 0);  return list;  }  private void backtrack(List<List<Integer>> list , List<Integer> tempList, int [] nums, int start){  list.add(new ArrayList<>(tempList));  for(int i = start; i < nums.length; i++){  tempList.add(nums[i]);  backtrack(list, tempList, nums, i + 1);  tempList.remove(tempList.size() - 1);  }  } |

### LeetCode\_90：Subsets II

|  |
| --- |
| Description |
| Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets (the power set).  **Note:** The solution set must not contain duplicate subsets.  For example, If ***nums*** = [1,2,2], a solution is:  [ [2], [1], [1,2,2], [2,2], [1,2], [] ] |

|  |
| --- |
| Function |
| private List<List<Integer>> res = new LinkedList<>();  public List<List<Integer>> subsetsWithDup(int[] nums) {  if(nums.length == 0) {  res.add(new LinkedList<>());  return res;  }  Arrays.sort(nums);  for(int i = 0; i <= nums.length; i ++)  findComb(nums, i, 0, new LinkedList<>());  return res;  }  private void findComb(int[] nums, int count, int index, List<Integer> list){  if(list.size() == count){  res.add(new LinkedList<>(list));  return;  }  for(int i = index; i < nums.length; i ++){  if(i > index && nums[i-1] == nums[i]) continue;  list.add(nums[i]);  findComb(nums, count, i+1, list);  list.remove(list.size()-1);  }  } |

|  |
| --- |
| Improved |
| public List<List<Integer>> subsetsWithDup(int[] nums) {  List<List<Integer>> list = new ArrayList<>();  Arrays.sort(nums);  backtrack(list, new ArrayList<>(), nums, 0);  return list;  }  private void backtrack(List<List<Integer>> list, List<Integer> tempList, int [] nums, int start){  list.add(new ArrayList<>(tempList));  for(int i = start; i < nums.length; i++){  if(i > start && nums[i] == nums[i-1]) continue; // skip duplicates  tempList.add(nums[i]);  backtrack(list, tempList, nums, i + 1);  tempList.remove(tempList.size() - 1);  }  } |

### LeetCode\_401：Binary Watch

|  |
| --- |
| Description |
| A binary watch has 4 LEDs on the top which represent the **hours** (**0-11**), and the 6 LEDs on the bottom represent the **minutes** (**0-59**).  Each LED represents a zero or one, with the least significant bit（最低有效位） on the right.  https://upload.wikimedia.org/wikipedia/commons/8/8b/Binary_clock_samui_moon.jpg  For example, the above binary watch reads "3:25".  Given a non-negative integer *n* which represents the number of LEDs that are currently on, return all possible times the watch could represent.  **Example:**  Input: n = 1 Return: ["1:00", "2:00", "4:00", "8:00", "0:01", "0:02", "0:04", "0:08", "0:16", "0:32"]  **Note:**   * The order of output does not matter. * The hour must not contain a leading zero, for example "01:00" is not valid, it should be "1:00". * The minute must be consist of two digits and may contain a leading zero, for example "10:2" is not valid, it should be "10:02". |

|  |
| --- |
| 递归回溯 |
| private List<String> res = new LinkedList<>();  public List<String> readBinaryWatch(int num) { //5ms  if(num < 0 || num > 8) return res;  if(num == 0) {  res.add("0:00");  return res;  }  times(10, num, 0, new LinkedList<Integer>());  return res;  }  private void times(int total, int on, int start, List<Integer> list){  if(list.size() == on){  String time = strTime(list);  if(time != null) res.add(time);  return;  }  for(int i = start; i < total-on+list.size()+1; i ++){  list.add(i);  times(total, on, i+1, list);  list.remove(list.size()-1);  }  }  //转换成表示时间的字符串  private String strTime(List<Integer> list){  if(list == null || list.size() == 0) return null;  int hours = 0;  int minutes = 0;  for(int n : list){  if(n < 4){  hours += 1<<n;  } else {  minutes += 1<<n-4;  }  }  if(hours > 11 || minutes > 59) return null;  return minutes < 10 ? hours + ":0" + minutes : hours + ":" + minutes;  } |

|  |
| --- |
| 循环 |
| public List<String> readBinaryWatch(int num) { 31ms  List<String> times = new ArrayList<>();  for (int h=0; h<12; h++)  for (int m=0; m<60; m++)  if (Integer.bitCount(h \* 64 + m) == num)  times.add(String.format("%d:%02d", h, m));  return times;  } |

### 在二维平面上的回溯法：LeetCode\_79 Word Search

|  |
| --- |
| Description |
| Given a 2D board and a word, find if the word exists in the grid.  The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.  For example, Given **board** =  [ ['A','B','C','E'],  ['S','F','C','S'],  ['A','D','E','E']]  **word** = "ABCCED", -> returns true, **word** = "SEE", -> returns true, **word** = "ABCB", -> returns false. |

|  |
| --- |
| 解决思路 |
| 递归回溯。 |

|  |
| --- |
| Function |
| private int d[][] = {{-1,0},{0,1},{1,0},{0,-1}};  private int m,n;  private boolean[][] visited;  private boolean inArea( int x , int y ){  return x >= 0 && x < m && y >= 0 && y < n;  }  // 从board[startx][starty]开始, 寻找word[index...word.size())  private boolean searchWord( char[][] board, String word, int index,  int startx, int starty ){  if( index == word.length() - 1 )  return board[startx][starty] == word.charAt(index);  if( board[startx][starty] == word.charAt(index) ){  visited[startx][starty] = true;  // 从startx, starty出发,向四个方向寻  for( int i = 0 ; i < 4 ; i ++ ){  int newx = startx + d[i][0];  int newy = starty + d[i][1];  if( inArea(newx, newy) && !visited[newx][newy] &&  searchWord( board , word , index + 1 , newx , newy ) )  return true;  }  visited[startx][starty] = false;  }  return false;  }  public boolean exist(char[][] board, String word) { //10ms  m = board.length;  n = board[0].length;  visited = new boolean[m][n];  for( int i = 0 ; i < board.length ; i ++ )  for( int j = 0 ; j < board[i].length ; j ++ )  if( searchWord( board, word, 0 , i, j) )  return true;  return false;  } |

### FloodFill算法：LeetCode\_200 Number of IsIands

|  |
| --- |
| Description |
| Given a 2d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.  ***Example 1:*** Answer is 1  11110 11010 11000 00000  ***Example 2:*** Answer is 3  11000 11000 00100 00011 |

|  |
| --- |
| Function |
| int direction[][] = {{0, -1},{0, 1},{-1, 0},{1, 0}};  int m,n;  boolean[][] visited;  boolean inArea( int x , int y ){  return x >= 0 && x < m && y >= 0 && y < n;  }  // 从grid[x][y]的位置开始,进行floodfill  // 保证(x,y)合法,且grid[x][y]是没有被访问过的陆地  private void dfs( char[][] grid , int x , int y ){  visited[x][y] = true;  for(int i = 0; i < 4; i ++){  int newX = x + direction[i][0];  int newY = y + direction[i][1];  if(inArea(newX, newY) && !visited[newX][newY] && grid[newX][newY] == '1')  dfs(grid, newX, newY);  }  }  public int numIslands(char[][] grid) { //6ms  m = grid.length;  if(m == 0) return 0;  n = grid[0].length;  int res = 0;    visited = new boolean[m][n];  for(int i = 0; i < m; i ++)  for(int j = 0; j < n; j ++)  if(!visited[i][j] && grid[i][j] == '1'){  res += 1;  dfs(grid, i, j);  }  return res;  } |

### 17年深信服笔试题（floodfill）

|  |
| --- |
| Description |
| 现有一个19x19的棋盘，给定一个点找出与之相连的相同棋子的个数。0、1分别表示黑子和白子，2表示空。 |

|  |
| --- |
| Function |
| **private** **int** direction[][] = {{0, -1},{0, 1},{-1, 0},{1, 0}};  **private** **boolean**[][] visited = **new** **boolean**[19][19];  **private** **int** res = 0;  **public** **int** calc(**int**[][] wq, **int** x, **int** y){  visit(wq, x, y, wq[x][y]);  **return** res;  }  **private** **void** visit(**int**[][] wq, **int** x, **int** y, **int** target) {  visited[x][y] = **true**;  **if**(wq[x][y] == target) res++;  **for**(**int** i = 0; i < 4; i ++) {  **int** newX = x + direction[i][0];  **int** newY = y + direction[i][1];  **if**(inArea(newX, newY) && !visited[newX][newY] && wq[newX][newY] == target)  visit(wq, newX, newY, target);  }  }  **private** **boolean** inArea( **int** x , **int** y ){  **return** x >= 0 && x < 19 && y >= 0 && y < 19;  } |

|  |
| --- |
| Test |
| **int**[][] wq = {{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,1,2,1,1,1,1,0,0,0,0,0,0,0},  {0,0,0,0,0,0,0,2,0,0,0,1,0,0,0,0,0,0,0},  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}};  System.***out***.println(calc(wq, 4, 6)); |

### LeetCode\_130：Surrounded Regions

|  |
| --- |
| Description |
| Given a 2D board containing 'X' and 'O' (the **letter** O), capture all regions surrounded by 'X'.  A region is captured by flipping all 'O's into 'X's in that surrounded region.  For example,  X X X X  X O O X  X X O X  X O X X  After running your function, the board should be:  X X X X  X X X X  X X X X  X O X X |

|  |
| --- |
| 我的解法：数据量为200x200时StackOverflowError |
| private int direction[][] = {{0, -1},{0, 1},{-1, 0},{1, 0}};  private boolean[][] visited;  private boolean tag = false;  private int m, n;  public void solve(char[][] board) {  if(board.length <= 2 || board[0].length <= 2) return;  m = board.length; n = board[0].length;  visited = new boolean[m][n];  for(int i = 1; i < m-1; i ++)  for(int j = 1; j < n-1; j ++)  if(board[i][j] == 'O' && !visited[i][j]){  tag = false;  captureRegions(board, i, j);  }  }  private void captureRegions(char[][] board, int x, int y){  visited[x][y] = true;  board[x][y] = 'X';  if(x == 0 || x == m-1 || y == 0 || y == n-1)  tag = true;    for(int i = 0; i < 4; i ++) {  int newX = x + direction[i][0];  int newY = y + direction[i][1];  if(inArea(newX, newY) && !visited[newX][newY] && board[newX][newY] == 'O')  captureRegions(board, newX, newY);  }    if(tag) board[x][y] = 'O';  }  private boolean inArea( int x , int y ){  return x >= 0 && x < m && y >= 0 && y < n;  } |

|  |
| --- |
| Function |
| public void solve(char[][] board) { //8ms  if (board.length == 0 || board[0].length == 0)  return;  if (board.length < 2 || board[0].length < 2)  return;  int m = board.length, n = board[0].length;  //Any 'O' connected to a boundary can't be turned to 'X', so ...  //Start from first and last column, turn 'O' to '\*'.  for (int i = 0; i < m; i++) {  if (board[i][0] == 'O')  boundaryDFS(board, i, 0);  if (board[i][n-1] == 'O')  boundaryDFS(board, i, n-1);  }  //Start from first and last row, turn '0' to '\*'  for (int j = 0; j < n; j++) {  if (board[0][j] == 'O')  boundaryDFS(board, 0, j);  if (board[m-1][j] == 'O')  boundaryDFS(board, m-1, j);  }  //post-prcessing, turn 'O' to 'X', '\*' back to 'O', keep 'X' intact.  for (int i = 0; i < m; i++) {  for (int j = 0; j < n; j++) {  if (board[i][j] == 'O')  board[i][j] = 'X';  else if (board[i][j] == '\*')  board[i][j] = 'O';  }  }  }  //Use DFS algo to turn internal however boundary-connected 'O' to '\*';  private void boundaryDFS(char[][] board, int i, int j) {  if (i < 0 || i > board.length - 1 || j <0 || j > board[0].length - 1)  return;  if (board[i][j] == 'O')  board[i][j] = '\*';  if (i > 1 && board[i-1][j] == 'O')  boundaryDFS(board, i-1, j);  if (i < board.length - 2 && board[i+1][j] == 'O')  boundaryDFS(board, i+1, j);  if (j > 1 && board[i][j-1] == 'O')  boundaryDFS(board, i, j-1);  if (j < board[i].length - 2 && board[i][j+1] == 'O' )  boundaryDFS(board, i, j+1);  } |

### LeetCode\_417：Pacific Atlantic Water Flow

|  |
| --- |
| Description |
| Given an m x n matrix of non-negative integers representing the height of each unit cell in a continent, the "Pacific ocean" touches the left and top edges of the matrix and the "Atlantic ocean" touches the right and bottom edges.  Water can only flow in four directions (up, down, left, or right) from a cell to another one with height equal or lower.  Find the list of grid coordinates where water can flow to both the Pacific and Atlantic ocean.  **Note:**   1. The order of returned grid coordinates does not matter. 2. Both *m* and *n* are less than 150.   **Example:**  Given the following 5x5 matrix:  Pacific ~ ~ ~ ~ ~  ~ 1 2 2 3 (5) \*  ~ 3 2 3 (4) (4) \*  ~ 2 4 (5) 3 1 \*  ~ (6) (7) 1 4 5 \*  ~ (5) 1 1 2 4 \*  \* \* \* \* \* Atlantic  Return:  [[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]] (positions with parentheses in above matrix). |

|  |
| --- |
| 我的解法：数据量为25x93时StackOverflowError |
| private int direction[][] = {{0, -1},{-1, 0},{0, 1},{1, 0}};  private boolean[][] visited;  private int pacific = 0, atlantic = 0;  private int m, n;  public List<int[]> pacificAtlantic(int[][] matrix) {  List<int[]> res = new LinkedList<>();  m = matrix.length;  if(m == 0) return res;  n = matrix[0].length;  for(int i = 0; i < m; i ++)  for(int j = 0; j < n; j ++){  pacific = 0; atlantic = 0;  visited = new boolean[m][n];  waterFlow(matrix, i, j, matrix[i][j]);  if(pacific >= 1 && atlantic >= 1)  res.add(new int[]{i, j});  }  return res;  }  private void waterFlow(int[][] matrix, int x, int y, int target){  visited[x][y] = true;  if(pacific >= 1 && atlantic >= 1) return;  if(x == 0 || y == 0 && matrix[x][y] <= target) pacific ++;  if(x == m-1 || y == n-1 && matrix[x][y] <= target) atlantic ++;  for(int i = 0; i < 4; i ++){  int newX = x + direction[i][0];  int newY = y + direction[i][1];  target = matrix[x][y];  if(inArea(newX, newY) && !visited[newX][newY] && matrix[newX][newY] <= target)  waterFlow(matrix, newX, newY, target);  }  visited[x][y] = false;  }  private boolean inArea( int x , int y ){  return x >= 0 && x < m && y >= 0 && y < n;  } |

|  |
| --- |
| 大神的BFS |
| int[][]dir = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};  public List<int[]> pacificAtlantic(int[][] matrix) {  List<int[]> res = new LinkedList<>();  if(matrix == null || matrix.length == 0 || matrix[0].length == 0){  return res;  }  int n = matrix.length, m = matrix[0].length;  //One visited map for each ocean  boolean[][] pacific = new boolean[n][m];  boolean[][] atlantic = new boolean[n][m];  Queue<int[]> pQueue = new LinkedList<>();  Queue<int[]> aQueue = new LinkedList<>();  for(int i=0; i<n; i++){ //Vertical border  pQueue.offer(new int[]{i, 0});  aQueue.offer(new int[]{i, m-1});  pacific[i][0] = true;  atlantic[i][m-1] = true;  }  for(int i=0; i<m; i++){ //Horizontal border  pQueue.offer(new int[]{0, i});  aQueue.offer(new int[]{n-1, i});  pacific[0][i] = true;  atlantic[n-1][i] = true;  }  bfs(matrix, pQueue, pacific);  bfs(matrix, aQueue, atlantic);  for(int i=0; i<n; i++){  for(int j=0; j<m; j++){  if(pacific[i][j] && atlantic[i][j])  res.add(new int[]{i,j});  }  }  return res;  }  public void bfs(int[][]matrix, Queue<int[]> queue, boolean[][]visited){  int n = matrix.length, m = matrix[0].length;  while(!queue.isEmpty()){  int[] cur = queue.poll();  for(int[] d:dir){  int x = cur[0]+d[0];  int y = cur[1]+d[1];  if(x<0 || x>=n || y<0 || y>=m || visited[x][y] || matrix[x][y] < matrix[cur[0]][cur[1]]){  continue;  }  visited[x][y] = true;  queue.offer(new int[]{x, y});  }  }  } |

|  |
| --- |
| 大神的DFS |
| public List<int[]> pacificAtlantic(int[][] matrix) {  List<int[]> res = new LinkedList<>();  if(matrix == null || matrix.length == 0 || matrix[0].length == 0){  return res;  }  int n = matrix.length, m = matrix[0].length;  boolean[][]pacific = new boolean[n][m];  boolean[][]atlantic = new boolean[n][m];  for(int i=0; i<n; i++){  dfs(matrix, pacific, Integer.MIN\_VALUE, i, 0);  dfs(matrix, atlantic, Integer.MIN\_VALUE, i, m-1);  }  for(int i=0; i<m; i++){  dfs(matrix, pacific, Integer.MIN\_VALUE, 0, i);  dfs(matrix, atlantic, Integer.MIN\_VALUE, n-1, i);  }  for (int i = 0; i < n; i++)  for (int j = 0; j < m; j++)  if (pacific[i][j] && atlantic[i][j])  res.add(new int[] {i, j});  return res;  }    int[][]dir = new int[][]{{0,1},{0,-1},{1,0},{-1,0}};    public void dfs(int[][]matrix, boolean[][]visited, int height, int x, int y){  int n = matrix.length, m = matrix[0].length;  if(x<0 || x>=n || y<0 || y>=m || visited[x][y] || matrix[x][y] < height)  return;  visited[x][y] = true;  for(int[]d:dir){  dfs(matrix, visited, matrix[x][y], x+d[0], y+d[1]);  }  } |

### 回溯法是经典人工智能的基础：LeetCode\_51 N-Queens

|  |
| --- |
| Description |
| The *n*-queens puzzle is the problem of placing *n* queens on an *n*×*n* chessboard such that no two queens attack each other.  https://leetcode.com/static/images/problemset/8-queens.png  Given an integer *n*, return all distinct solutions to the *n*-queens puzzle.  Each solution contains a distinct board configuration of the *n*-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.  For example, There exist two distinct solutions to the 4-queens puzzle:  [  [".Q..", // Solution 1  "...Q",  "Q...",  "..Q."],  ["..Q.", // Solution 2  "Q...",  "...Q",  ".Q.."]  ] |

|  |
| --- |
| Function |
| private boolean[] col, dia1, dia2; //分别表示同列、同对角线  private List<List<String>> res;  public List<List<String>> solveNQueens(int n) {  res = new ArrayList<>();  col = new boolean[n];  dia1 = new boolean[2\*n-1];  dia2 = new boolean[2\*n-1];    putQueen(n, 0, new ArrayList<Integer>(n));  return res;  }  // 尝试在一个n皇后问题中, 摆放第index行的皇后位置  private void putQueen(int n , int index, List<Integer> row){  if(index == n){  res.add(generateBoard(n, row));  return;  }    //遍历当前行的每一个位置  for(int i = 0; i < n; i ++)  if(!col[i] && !dia1[index+i] && !dia2[index-i+n-1]){  col[i] = true;  dia1[index+i] = true;  dia2[index-i+n-1] = true;  row.add(i);  putQueen(n , index+1, row);  row.remove(row.size()-1);  col[i] = false;  dia1[index+i] = false;  dia2[index-i+n-1] = false;  }  }  //生成"."+"Q"形式的答案  private List<String> generateBoard( int n, List<Integer> row ){  List<String> list = new ArrayList<>(n);  for(int index : row){  String str = "";  for(int i = 0; i < n; i ++)  if(i == index)  str += "Q";  else  str += ".";  list.add(str);  }  return list;  } |

### eetCode\_52：N-Queens II

|  |
| --- |
| Description |
| Follow up for N-Queens problem.  Now, instead outputting board configurations, return the total number of distinct solutions. |

|  |
| --- |
| Function |
| private boolean[] col, dia\_1, dia\_2;  private int res = 0;  public int totalNQueens(int n) {  if(n == 1) return 1;  if(n < 4) return 0;  col = new boolean[n];  dia\_1 = new boolean[2\*n-1];  dia\_2 = new boolean[2\*n-1];  putQueen(n, 0);  return res;  }  private void putQueen(int n, int index){  if(index == n){  res ++;  return;  }    //遍历当前行的每一个位置，并放置Queen  for(int i = 0; i < n; i ++){  if(!col[i] && !dia\_1[index+i] && !dia\_2[index-i+n-1]){  col[i] = true;  dia\_1[index+i] = true;  dia\_2[index-i+n-1] = true;  putQueen(n, index+1); //放置下一行的Queen    //重置状态  col[i] = false;  dia\_1[index+i] = false;  dia\_2[index-i+n-1] = false;  }  }  } |

### LeetCode\_37：Sudoku Solver

|  |
| --- |
| Description |
| Write a program to solve a Sudoku puzzle by filling the empty cells.  Empty cells are indicated by the character '.'.  You may assume that there will be only one unique solution.  https://upload.wikimedia.org/wikipedia/commons/thumb/f/ff/Sudoku-by-L2G-20050714.svg/250px-Sudoku-by-L2G-20050714.svg.png https://upload.wikimedia.org/wikipedia/commons/thumb/3/31/Sudoku-by-L2G-20050714_solution.svg/250px-Sudoku-by-L2G-20050714_solution.svg.png |

|  |
| --- |
| Function |
| public void solveSudoku(char[][] board) {  solve(board);  }  private boolean solve(char[][] board){  for(int i = 0; i < board.length; i ++)  for(int j = 0; j < board[0].length; j ++)  if(board[i][j] == '.'){  for(char ch = '1'; ch <= '9'; ch ++)  if(isValid(board, i, j, ch)){  board[i][j] = ch;  if(solve(board)) return true;  else board[i][j] = '.'; //回溯  }  return false;  }  return true;  }  private boolean isValid(char[][] board, int x, int y, char ch){  for(int i = 0; i < 9; i ++){  if(board[x][i] == ch) return false;  if(board[i][y] == ch) return false;  if(board[(x/3)\*3+i/3][(y/3)\*3+i%3] == ch) return false;  }  return true;  } |

## 动态规划基础

### 什么是动态规划

|  |
| --- |
| Description |
| 将原问题拆解成若干个子问题，同时保存子问题的答案，使得每个子问题只求解一次，最终获得原问题的答案。 |

### 菲波那切数列（Fibonacci Sequence）

|  |
| --- |
| Description |
| f(0) = 1; f(1) = 1; f(n) = f(n-1) + f(n-2); |

|  |
| --- |
| 递归实现：自上而下地解决问题 |
| public int fibonacci(int n){  if(n = 0) return 1;  if(n = 1) return 1;  return fibonacci(n-1) + fibonacci(n-2);  } |

|  |
| --- |
| 记忆化搜索 |
| int[] memory = new int[n+1]{-1, ...};  public int fibonacci(int n){  if(n = 0) return 1;  if(n = 1) return 1;    if(memory[n] == -1)  memory[n] = fibonacci(n-1) + fibonacci(n-2);    return memory[n];  } |

|  |
| --- |
| 动态规划：自下而上 |
| public int fibonacci(int n){  int[] memory = new int[n+1];  memory[0] = 1;  memory[1] = 1;    for(int i = 2; i <= n; i ++)  memory[i] = memory[i-1] + memory[i-2];    return memory[n];  } |

### 第一个动态规划问题：LeetCode\_70 Climbing Stairs

|  |
| --- |
| Description |
| You are climbing a stair case. It takes *n* steps to reach to the top.  Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?  **Note:** Given *n* will be a positive integer. |

|  |
| --- |
| Function |
| //递归，Time Limit Exceeded  public int climbStairs(int n) {  if(n == 0 || n == 1) return 1;  return climbStairs(n-1) + climbStairs(n-2);  }    //记忆化搜索  private int[] memo;  public int climbStairs(int n) { //0ms  memo = new int[n+1];  return calcWays(n);  }  private int calcWays(int n) {  if(n == 0 || n == 1) return 1;  if(memo[n] == 0) memo[n] = calcWays(n-1) + calcWays(n-2);  return memo[n];  }    //动态规划  private int[] memo;  public int climbStairs(int n) { //0ms  memo = new int[n+1];  memo[0] = 1;  memo[1] = 1;  for(int i = 2; i < n + 1; i ++)  memo[i] = memo[i-1] + memo[i-2];  return memo[n];  } |

### LeetCode\_120：Triangle

|  |
| --- |
| Description |
| Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.  For example, given the following triangle  [ [2],  [3,4],  [6,5,7],  [4,1,8,3] ]  The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11).  **Note:** Bonus point if you are able to do this using only *O*(*n*) extra space, where *n* is the total number of rows in the triangle. |

|  |
| --- |
| 解决思路 |
| 动态规划，自下而上。 |

|  |
| --- |
| Function |
| public int minimumTotal(List<List<Integer>> triangle) {  int size = triangle.size();  if(size == 1) return triangle.get(0).get(0);  int[] sum = new int[size];  for(int i = 0; i < size; i ++)  sum[i] = triangle.get(size-1).get(i);    List<Integer> line;  for(int j = size-2; j >= 0; j --){  line = triangle.get(j);  for(int k = 0; k < line.size(); k ++)  sum[k] = line.get(k) + (sum[k] > sum[k+1] ? sum[k+1] : sum[k]);  }  return sum[0];  } |

### LeetCode\_64：Minimum Path Sum

|  |
| --- |
| Description |
| Given a *m* x *n* grid filled with non-negative numbers, find a path from top left to bottom right which *minimizes* the sum of all numbers along its path.  **Note:** You can only move either down or right at any point in time.  **Example 1:**  [[1,3,1],  [1,5,1],  [4,2,1]]  Given the above grid map, return 7. Because the path 1→3→1→1→1 minimizes the sum. |

|  |
| --- |
| Other |
| public int minPathSum(int[][] grid) {  int m = grid.length;// row  int n = grid[0].length; // column  for (int i = 0; i < m; i++) {  for (int j = 0; j < n; j++) {  if (i == 0 && j != 0) {  grid[i][j] = grid[i][j] + grid[i][j - 1];  } else if (i != 0 && j == 0) {  grid[i][j] = grid[i][j] + grid[i - 1][j];  } else if (i == 0 && j == 0) {  grid[i][j] = grid[i][j];  } else {  grid[i][j] = Math.min(grid[i][j - 1], grid[i - 1][j]) + grid[i][j];  }  }  }  return grid[m - 1][n - 1];  } |

|  |
| --- |
| Mine |
| private int[][] dire = new int[][]{{-1, 0},{0, -1}};  public int minPathSum(int[][] grid) {  int m = grid.length;  int n = grid[0].length;    Queue<int[]> queue = new LinkedList<>();  queue.add(new int[] {m-1, n-1});  int lastX = m-1, lastY = n-1;  int lastValue = grid[m-1][n-1];  while(!queue.isEmpty()) {  int[] dot = queue.poll();  for(int i = 0; i < 2; i ++){  int X = dot[0] + dire[i][0];  int Y = dot[1] + dire[i][1];  if(X >= 0 && X < m && Y >= 0 && Y < n){  if(!equalsDot(lastX, lastY, X, Y)){  lastX = X; lastY = Y;  lastValue = grid[X][Y];  grid[X][Y] += grid[dot[0]][dot[1]];  queue.add(new int[] {X, Y});  } else {  grid[X][Y] = Math.min(grid[X][Y], lastValue+grid[dot[0]][dot[1]]);  }  }  }  }  return grid[0][0];  }  private boolean equalsDot(int lx, int ly, int x, int y) {  if(lx == x && ly == y) return true;  return false;  } |

### 发现重叠子问题：LeetCode\_343 Integer Break

|  |
| --- |
| Description |
| Given a positive integer *n*, break it into the sum of **at least** two positive integers and maximize the product of those integers. Return the maximum product you can get.  For example, given *n* = 2, return 1 (2 = 1 + 1); given *n* = 10, return 36 (10 = 3 + 3 + 4).  **Note**: You may assume that *n* is not less than 2 and not larger than 58. |

|  |
| --- |
| 记忆化搜索 |
| private int[] memory;  public int integerBreak(int n) {  memory = new int[n+1];  return breakInteger(n);  }  private int breakInteger(int n){  if(n == 1) return 1;    if(memory[n] != 0) return memory[n];    int res = 0;  for(int i = 1; i < n; i ++)  res = max3(res, i\*(n-i), i\*breakInteger(n-i));    memory[n] = res;  return res;  }  private int max3(int one, int two, int three){  return Math.max(one, Math.max(two, three));  } |

|  |
| --- |
| 动态规划 |
| public int integerBreak(int n){  int[] memory = new int[n+1];  memory[1] = 1;    for(int i = 2; i <= n; i ++)  //求解memory[i]  for(int j = 1; j <= i-j; j ++)  memory[i] = max3(memory[i], j\*(i-j), j\*memory[i-j]);    return memory[n];  }  private int max3(int one, int two, int three){  return Math.max(one, Math.max(two, three));  } |

### LeetCode\_279：Perfect Squares

|  |
| --- |
| Description |
| Given a positive integer *n*, find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to *n*.  For example, given *n* = 12, return 3 because 12 = 4 + 4 + 4; given *n* = 13, return 2 because 13 = 4 + 9. |

|  |
| --- |
| 解决思路 |
| 动态规划。 |

|  |
| --- |
| Function |
| public int numSquares(int n) { //44ms  int[] mem = new int[n+1];  for(int k = 2; k < mem.length; k++)  mem[k] = Integer.MAX\_VALUE;    mem[1] = 1;  for(int i = 2; i <= n; i ++)  for(int j = 1; j\*j <= i; j ++)  mem[i] = Math.min(mem[i], mem[i-j\*j]+1);    return mem[n];  } |

### LeetCode\_91：Decode Ways

|  |
| --- |
| Description |
| A message containing letters from A-Z is being encoded to numbers using the following mapping:  'A' -> 1  'B' -> 2  ...  'Z' -> 26  Given an encoded message containing digits, determine the total number of ways to decode it.  For example：Given encoded message "12", it could be decoded as "AB" (1 2) or "L" (12).  The number of ways decoding "12" is 2. |

|  |
| --- |
| Function |
| public int numDecodings(String s) {  if(s.isEmpty()) return 0;  if(s.charAt(0) == '0') return 0;    int[] mem = new int[s.length()+1];  mem[1] = 1;    for(int i = 1; i < s.length(); i ++){  int dd = Integer.parseInt(s.substring(i-1, i+1));    if(s.charAt(i) == '0' && (dd <= 0 || dd > 26)) return 0;    if(s.charAt(i-1)=='0') mem[i+1] = mem[i-1];  else mem[i+1] = (s.charAt(i)=='0'?0:mem[i]) + ((dd > 0 && dd < 27)?(mem[i-1]>0?mem[i-1]:1):0);  }  return mem[s.length()];  } |

### LeetCode\_62：Unique Paths

|  |
| --- |
| Description |
| A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).  The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).  How many possible unique paths are there?  https://leetcode.com/static/images/problemset/robot_maze.png  Above is a 3 x 7 grid. How many possible unique paths are there?  **Note:** *m* and *n* will be at most 100. |

|  |
| --- |
| 解决思路 |
| 动态规划，M\*N的额外空间。优化：m或n的额外空间 |

|  |
| --- |
| Function |
| public int uniquePaths(int m, int n) {  if(m == 1 || n == 1) return 1;    int[][] grid = new int[m][n];  for(int i = 0; i < n; i ++) grid[0][i] = 1;  for(int i = 0; i < m; i ++) grid[i][0] = 1;  for(int i = 1; i < m; i ++)  for(int j = 1; j < n; j ++)  grid[i][j] = grid[i][j-1] + grid[i-1][j];    return grid[m-1][n-1];  } |

### LeetCode\_63：Unique Paths II

|  |
| --- |
| Description |
| Follow up for "Unique Paths":  Now consider if some obstacles are added to the grids. How many unique paths would there be?  An obstacle and empty space is marked as 1 and 0 respectively in the grid.  For example,  There is one obstacle（障碍） in the middle of a 3x3 grid as illustrated below.  [ [0,0,0],  [0,1,0],  [0,0,0] ]  The total number of unique paths is 2.  **Note:** *m* and *n* will be at most 100. |

|  |
| --- |
| Function |
| public int uniquePathsWithObstacles(int[][] obstacleGrid) {  int width = obstacleGrid[0].length;  int[] dp = new int[width];  dp[0] = 1;  for (int[] row : obstacleGrid) {  for (int j = 0; j < width; j++) {  if (row[j] == 1)  dp[j] = 0;  else if (j > 0)  dp[j] += dp[j - 1];  }  }  return dp[width - 1];  } |

### 状态的定义和状态转移：LeetCode\_198 House Robber

|  |
| --- |
| Description |
| You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.  Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**. |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function：记忆化搜索 |
| private int[] mem;  public int rob(int[] nums) {  int len = nums.length;  if(len == 0) return 0;  mem = new int[len];  for(int i = 0; i < len; i ++)  mem[i] = -1;    return robHouse(nums, len, 0);  }    //考虑偷取[x, nums.length-1]的房子  private int robHouse(int[] nums, int len, int x){  if(x >= len) return 0;    if(mem[x] != -1) return mem[x];    int res = 0;  for(int i = x; i < len; i ++)  res = Math.max(res, nums[i] + robHouse(nums, len, i+2));    mem[x] = res;  return res;  } |

|  |
| --- |
| Function：动态规划 |
| public int rob(int[] nums){  int len = nums.length;  if(len == 0) return 0;    int[] mem = new int[len];  for(int i = 0; i < len; i ++)  mem[i] = -1;    mem[len-1] = nums[len-1];  for(int i = len-2; i >= 0; i --)  //计算mem[i]  for(int j = i; j < len; j ++)  mem[i] = Math.max(mem[i], nums[j]+(j+2 >= len ? 0 : mem[j+2]));    return mem[0];  } |

|  |
| --- |
| 扩展 |
|  |

### LeetCode\_213：House Robber II

|  |
| --- |
| Description |
| **Note:** This is an extension of House Robber.  After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.  Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight **without alerting the police**. |

|  |
| --- |
| Function |
| public int rob(int[] nums) {  int len = nums.length;  if(len == 0) return 0;  if(len == 1) return nums[0];    int[] mem = new int[len];    //不偷取第一间  mem[len-1] = nums[len-1];  for(int i = len-2; i > 0; i --)  for(int j = i; j < len; j ++)  mem[i] = Math.max(mem[i], nums[j]+(j+2 >= len ? 0 : mem[j+2]));    int res = mem[1];    //偷取第一间  mem[len-1] = 0;  for(int i = len-2; i >= 0; i --){  mem[i] = 0;  for(int j = i; j < len-1; j ++)  mem[i] = Math.max(mem[i], nums[j]+(j+2 >= len-1 ? 0 : mem[j+2]));  }    return Math.max(mem[0], res);  } |

### LeetCode\_337：House Robber III

|  |
| --- |
| Description |
| The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.  Determine the maximum amount of money the thief can rob tonight without alerting the police.  **Example 1:**  3  / \  2 3  \ \  3 1  Maximum amount of money the thief can rob = 3 + 3 + 1 = **7**.  **Example 2:**  3  / \  4 5  / \ \  1 3 1  Maximum amount of money the thief can rob = 4 + 5 = **9**. |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
|  |

### LeetCode\_309：Best Time to Buy and Sell Stock with Cooldown

|  |
| --- |
| Description |
|  |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
|  |

### 0-1背包问题

|  |
| --- |
| Description |
| 有一个背包，它的容量为C（Capacity）。现在有n中不同的物品，编号为0到n-1，其中每一件物品的重量为w(i)，价值为v(i)。问可以向这个背包中放入哪些物品，使得在不超过容量的基础上，物品的总价值最大。 |

|  |
| --- |
| 暴力解法 |
| 每个物品都可以放入背包，也可以不放进背包。O((2^n)\*n) |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| 记忆化搜索 |
| **private** **int**[][] mem;  //w[i]、v[i]分别为weight和value  **public** **int** knapsack01(**int**[] w, **int**[] v, **int** C) {  **assert**(w.length == v.length);  **int** n = w.length;  mem = **new** **int**[n][C+1];  **for**(**int** i = 0; i < n; i++)  **for**(**int** j = 0; j < C+1; j ++)  mem[i][j] = -1;    **return** bestValue(w, v, n-1, C);  }    //在[0, index]个物品中，寻找填充背包的最大价值  **private** **int** bestValue(**int**[] w, **int**[] v, **int** index, **int** c) {  **if**(index < 0 || c <= 0) **return** 0;    **if**(mem[index][c] != -1) **return** mem[index][c];    **int** res = bestValue(w, v, index-1, c); //不选择第index个物品  **if**(c - w[index] >= 0)  res = Math.*max*(res, v[index]+bestValue(w, v, index-1, c-w[index]));    mem[index][c] = res;  **return** res;  } |

|  |
| --- |
| 动态规划 |
| **public** **int** knapsack01(**int**[] w, **int**[] v, **int** C) {  **assert**(w.length == v.length);  **int** n = w.length;  **int**[][] mem = **new** **int**[n][C+1];    //第一个物品  **for**(**int** i = 0; i <= C; i++)  **if**(i >= w[0]) mem[0][i] = v[0];    **for**(**int** i = 1; i < n; i ++)  **for**(**int** j = 0; j <= C; j++) {  **int** res = mem[i-1][j];  **if**(j >= w[i])  mem[i][j] = Math.*max*(res, v[i] + mem[i-1][j-w[i]]);  }    **return** mem[n-1][C];  } |

|  |
| --- |
| 动态规划优化：(c+1)的空间复杂度 |
| **public** **int** knapsack01(**int**[] w, **int**[] v, **int** C) {  **assert**(w.length == v.length);  **int** n = w.length;  **int**[] mem = **new** **int**[C+1];    //第一个物品  **for**(**int** i = 0; i <= C; i++)  **if**(i >= w[0]) mem[i] = v[0];    **for**(**int** i = 1; i < n; i ++)  **for**(**int** j = C; j >= 0; j--) {  **int** res = mem[j];  **if**(j >= w[i])  mem[j] = Math.*max*(res, v[i] + mem[j-w[i]]);  }  **return** mem[C];  } |

### 更多背包问题变种

|  |
| --- |
| Description |
| 1. 完全背包问题：每个物品可以无限使用。 2. 多重背包问题：每个物品不知一个，有num[i]个。 3. 多维费用背包问题：要考虑物品的体积和重量两个维度。 4. 物品间加入更多约束：相互排斥或依赖。 |

### 面试中的0-1背包问题：LeetCode\_416 Partition Equal Subset Sum

|  |
| --- |
| Description |
| Given a **non-empty** array containing **only positive integers**, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.  **Note:**   1. Each of the array element will not exceed 100. 2. The array size will not exceed 200.   **Example 1:**  Input: [1, 5, 11, 5]  Output: true  Explanation: The array can be partitioned as [1, 5, 5] and [11].  **Example 2:**  Input: [1, 2, 3, 5]  Output: false  Explanation: The array cannot be partitioned into equal sum subsets. |

|  |
| --- |
| 记忆化搜索 |
| private int[][] mem;  public boolean canPartition(int[] nums) {  int len = nums.length;  int sum = 0;  for(int i = 0; i < len; i ++)  sum += nums[i];    if(sum % 2 != 0) return false;    mem = new int[len][sum/2+1];  for(int i = 0; i < len; i ++)  for(int j = 0; j <= sum/2; j++)  mem[i][j] = -1;    return canFillKnapsack(nums, len-1, sum/2);  }  private boolean canFillKnapsack(int[] nums, int index, int c){  if(c == 0) return true;  if(index < 0 || c < 0) return false;    if(mem[index][c] != -1) return mem[index][c]==1;    mem[index][c] = (canFillKnapsack(nums, index-1, c) //不选择当前物品  || canFillKnapsack(nums, index-1, c-nums[index])) ? 1 : 0; //选择当前物品    return mem[index][c]==1;  } |

|  |
| --- |
| 动态规划 |
| public boolean canPartition(int[] nums) {  int len = nums.length;  int sum = 0;  for(int i = 0; i < len; i ++)  sum += nums[i];    if(sum % 2 != 0) return false;    int c = sum/2;  boolean[] mem = new boolean[c+1];  for(int i = 0; i <= c; i++)  if(nums[0] == i) mem[i] = true;    for(int i = 1; i < len; i ++)  for(int j = c; j >= nums[i]; j--)  mem[j] = mem[j] || //不选择当前  mem[j-nums[i]]; //选择当前  return mem[c];  } |

### LeetCode\_322：Coin Change

|  |
| --- |
| Description |
|  |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
|  |

### LeetCode\_377：Combination Sum IV

|  |
| --- |
| Description |
|  |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
|  |

### LeetCode\_474：Ones and Zeroes

|  |
| --- |
| Description |
|  |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
|  |

### LeetCode\_139：Word Break

|  |
| --- |
| Description |
|  |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
|  |

### LeetCode\_494：Target Sum

|  |
| --- |
| Description |
|  |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
|  |

### LIS问题：LeetCode\_300 Longest Increasing Subsequence

|  |
| --- |
| Description |
| Given an unsorted array of integers, find the length of longest increasing subsequence.  For example, Given [10, 9, 2, 5, 3, 7, 101, 18], The longest increasing subsequence is [2, 3, 7, 101], therefore the length is 4. Note that there may be more than one LIS combination, it is only necessary for you to return the length.  Your algorithm should run in O(*n2*) complexity.  **Follow up:** Could you improve it to O(*n* log *n*) time complexity? |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function：n^2 |
| public int lengthOfLIS(int[] nums) {  int len = nums.length;  if(len == 0) return 0;    int[] mem = new int[len];  mem[0] = 1;  for(int i = 1; i < len; i ++){  mem[i] = 1;  for(int j = i-1; j >= 0; j --)  if(nums[j] < nums[i])  mem[i] = Math.max(mem[i], mem[j] + 1);  }    int res = 1;  for(int i = 1; i < len; i ++)  res = Math.max(res, mem[i]);    return res;  } |

|  |
| --- |
| Function：nlogn |
|  |

### LeetCode\_376：Wiggle Subsequence

|  |
| --- |
| Description |
|  |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
|  |

### 最长公共子序列（LCS）

|  |
| --- |
| Description |
| 给出两个字符串s1和s2，求这两个字符串的最长公共子序列的长度。 |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
|  |

### 求动态规划的具体解

|  |
| --- |
| Description |
|  |

|  |
| --- |
| 解决思路 |
| 反向重构。 |

## 贪心算法

### 贪心基础LeetCode\_455：Assign Cookies

|  |
| --- |
| Description |
| Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie. Each child i has a greed factor gi, which is the minimum size of a cookie that the child will be content with; and each cookie j has a size sj. If sj >= gi, we can assign the cookie j to the child i, and the child i will be content. Your goal is to maximize the number of your content children and output the maximum number.  **Note:** You may assume the greed factor is always positive.  You cannot assign more than one cookie to one child.  **Example 1:**  **Input:** [1,2,3], [1,1]  **Output:** 1  **Explanation:** You have 3 children and 2 cookies. The greed factors of 3 children are 1, 2, 3.  And even though you have 2 cookies, since their size is both 1, you could only make the child whose greed factor is 1 content.  You need to output 1.  **Example 2:**  **Input:** [1,2], [1,2,3]  **Output:** 2  **Explanation:** You have 2 children and 3 cookies. The greed factors of 2 children are 1, 2.  You have 3 cookies and their sizes are big enough to gratify all of the children,  You need to output 2. |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
|  |

### LeetCode\_392：Is Subsequence

|  |
| --- |
| Description |
|  |

|  |
| --- |
| 解决思路 |
|  |

|  |
| --- |
| Function |
|  |

### 贪心算法与动态规划的关系：LeetCode\_435 Non-overlappint Intervals

|  |
| --- |
| Description |
| Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.  **Note:**   1. You may assume the interval's end point is always bigger than its start point. 2. Intervals like [1,2] and [2,3] have borders "touching" but they don't overlap each other.   **Example 1:**  **Input:** [ [1,2], [2,3], [3,4], [1,3] ]  **Output:** 1  **Explanation:** [1,3] can be removed and the rest of intervals are non-overlapping.  **Example 2:**  **Input:** [ [1,2], [1,2], [1,2] ]  **Output:** 2  **Explanation:** You need to remove two [1,2] to make the rest of intervals non-overlapping.  **Example 3:**  **Input:** [ [1,2], [2,3] ]  **Output:** 0  **Explanation:** You don't need to remove any of the intervals since they're already non-overlapping. |

|  |
| --- |
| 动态规划：213ms |
| public int eraseOverlapIntervals(Interval[] intervals) {  int len = intervals.length;  if(len == 0) return 0;    Arrays.sort(intervals, new Comparator<Interval>() {  @Override  public int compare(Interval o1, Interval o2) {  if(o1.start != o2.start)  return o1.start - o2.start;  else return o1.end - o2.end;  }});    int[] mem = new int[len];  mem[0] = 1;    for(int i = 1; i < len; i ++){  mem[i] = 1;  for(int j = 0; j < i; j ++)  if(intervals[i].start >= intervals[j].end)  mem[i] = Math.max(mem[i], mem[j] + 1);  }  return len - mem[len-1];  } |

|  |
| --- |
| 贪心算法：9ms |
| public int eraseOverlapIntervals(Interval[] intervals) {  int len = intervals.length;  if(len == 0) return 0;    Arrays.sort(intervals, new Comparator<Interval>() {  @Override  public int compare(Interval o1, Interval o2) {  if(o1.end != o2.end)  return o1.end - o2.end;  else return o1.start - o2.start;  }});    int res = 1;  int pre = 0;  for(int i = 1; i < len; i ++)  if(intervals[i].start >= intervals[pre].end){  res ++;  pre = i;  }    return len - res;  } |