目录

[位操作 1](#_Toc464629259)

[Single Number 1](#_Toc464629260)

[Find the Difference 2](#_Toc464629261)

[Single Number III 4](#_Toc464629262)

[Missing Number 7](#_Toc464629263)

[Counting Bits 9](#_Toc464629264)

[Maximum Product of Word Lengths 11](#_Toc464629265)

[Repeated DNA Sequences 15](#_Toc464629266)

[Subsets 书 17](#_Toc464629267)

[Bitwise AND of Numbers Range 18](#_Toc464629268)

[Reverse Bits 19](#_Toc464629269)

[递归和动态规划 22](#_Toc464629270)

[**Triangle** 22](#_Toc464629271)

[**Range Sum Query - Immutable** 25](#_Toc464629272)

[**Range Sum Query 2D - Immutable** 26](#_Toc464629273)

[**Maximum Subarray** 28](#_Toc464629274)

[**Maximum Product Subarray** 29](#_Toc464629275)

[**Longest Increasing Subsequence** 32](#_Toc464629276)

[**Perfect Squares** 36](#_Toc464629277)

[Coin Change 44](#_Toc464629278)

[Maximal Square 47](#_Toc464629279)

[Decode Ways 52](#_Toc464629280)

[Best Time to Buy and Sell Stock 54](#_Toc464629281)

[Best Time to Buy and Sell Stock II（贪心） 56](#_Toc464629282)

[**Best Time to Buy and Sell Stock with Cooldown** 58](#_Toc464629283)

[#1137 : Recruitment 64](#_Toc464629284)

[House Robber 70](#_Toc464629285)

[House Robber II 72](#_Toc464629286)

[Unique Paths 75](#_Toc464629287)

[Unique Paths II 80](#_Toc464629288)

[Minimum Path Sum 85](#_Toc464629289)

[**Ugly Number II** 88](#_Toc464629290)

[Ugly Number （Math题） 94](#_Toc464629291)

[Climbing Stairs 95](#_Toc464629292)

[Guess Number Higher or Lower II 97](#_Toc464629293)

[Count Numbers with Unique Digits 103](#_Toc464629294)

[Partition Equal Subset Sum 104](#_Toc464629295)

[Is Subsequence 107](#_Toc464629296)

[Combination Sum IV 110](#_Toc464629297)

[Integer Break 114](#_Toc464629298)

[总结 118](#_Toc464629299)

[链表 118](#_Toc464629300)

[Palindrome Linked List 118](#_Toc464629301)

[Merge Two Sorted Lists 122](#_Toc464629302)

[Remove Linked List Elements 124](#_Toc464629303)

[Delete Node in a Linked List 126](#_Toc464629304)

[Remove Nth Node From End of List 127](#_Toc464629305)

[Intersection of Two Linked Lists 128](#_Toc464629306)

[Reverse Linked List 132](#_Toc464629307)

[中级 134](#_Toc464629308)

[Odd Even Linked List 134](#_Toc464629309)

[Add Two Numbers 135](#_Toc464629310)

[Swap Nodes in Pairs 138](#_Toc464629311)

[Reverse Nodes in k-Group 139](#_Toc464629312)

[Reverse Linked List II 141](#_Toc464629313)

[Reorder List 144](#_Toc464629314)

[Remove Duplicates from Sorted List 148](#_Toc464629315)

[Remove Duplicates from Sorted List II 149](#_Toc464629316)

[Rotate List 151](#_Toc464629317)

[Insertion Sort List 155](#_Toc464629318)

[Sort List 156](#_Toc464629319)

[Partition List 162](#_Toc464629320)

[Convert Sorted List to Binary Search Tree 164](#_Toc464629321)

[约瑟夫环 170](#_Toc464629322)

[栈和队列 171](#_Toc464629323)

[Min Stack 171](#_Toc464629324)

[Valid Parentheses 175](#_Toc464629325)

[Mini Parser 177](#_Toc464629326)

[树 179](#_Toc464629327)

[EASY 179](#_Toc464629328)

[Path Sum 179](#_Toc464629329)

[Balanced Binary Tree 180](#_Toc464629330)

[Maximum Depth of Binary Tree 181](#_Toc464629331)

[Minimum Depth of Binary Tree 181](#_Toc464629332)

[Same Tree 183](#_Toc464629333)

[Invert Binary Tree 183](#_Toc464629334)

[Binary Tree Level Order Traversal 187](#_Toc464629335)

[Binary Tree Level Order Traversal II 188](#_Toc464629336)

[Symmetric Tree 190](#_Toc464629337)

[Binary Tree Paths 194](#_Toc464629338)

[Lowest Common Ancestor of a Binary Search Tree 196](#_Toc464629339)

[Sum of Left Leaves 199](#_Toc464629340)

[总结 201](#_Toc464629341)

[MEDIUM 201](#_Toc464629342)

[Validate Binary Search Tree 201](#_Toc464629343)

[Binary Tree Inorder Traversal 202](#_Toc464629344)

[Binary Tree Preorder Traversal 204](#_Toc464629345)

[Binary Tree Postorder Traversal 207](#_Toc464629346)

[Binary Search Tree Iterator 209](#_Toc464629347)

[Binary Tree Right Side View 213](#_Toc464629348)

[Populating Next Right Pointers in Each Node 215](#_Toc464629349)

[Populating Next Right Pointers in Each Node II 218](#_Toc464629350)

[Binary Tree Zigzag Level Order Traversal 220](#_Toc464629351)

[Flatten Binary Tree to Linked List 223](#_Toc464629352)

[Unique Binary Search Trees 227](#_Toc464629353)

[Unique Binary Search Trees II 230](#_Toc464629354)

[Count Complete Tree Nodes 234](#_Toc464629355)

[Kth Smallest Element in a BST 239](#_Toc464629356)

[Path Sum II 244](#_Toc464629357)

[Sum Root to Leaf Numbers 247](#_Toc464629358)

[Lowest Common Ancestor of a Binary Tree 250](#_Toc464629359)

[Convert Sorted Array to Binary Search Tree 253](#_Toc464629360)

[Construct Binary Tree from Preorder and Inorder Traversal 255](#_Toc464629361)

[Construct Binary Tree from Inorder and Postorder Traversal 260](#_Toc464629362)

[总结 267](#_Toc464629363)

[字符串 267](#_Toc464629364)

[Compare Version Numbers 267](#_Toc464629365)

[Count and Say 269](#_Toc464629366)

[Add Binary 271](#_Toc464629367)

[Multiply Strings 274](#_Toc464629368)

[Length of Last Word 277](#_Toc464629369)

[Ransom Note 279](#_Toc464629370)

[Basic Calculator 280](#_Toc464629371)

[Basic Calculator II 285](#_Toc464629372)

[Longest Common Prefix 289](#_Toc464629373)

[Longest Palindromic Substring 292](#_Toc464629374)

[数组 297](#_Toc464629375)

[Rotate Array 297](#_Toc464629376)

[**Move Zeroes** 305](#_Toc464629377)

[Remove Element 306](#_Toc464629378)

[Remove Duplicates from Sorted Array 307](#_Toc464629379)

[Remove Duplicates from Sorted Array II 309](#_Toc464629380)

[Plus One 312](#_Toc464629381)

[Majority Element 314](#_Toc464629382)

[Majority Element II 318](#_Toc464629383)

[Contains Duplicate 323](#_Toc464629384)

[Contains Duplicate II 326](#_Toc464629385)

[Contains Duplicate III 二叉查找树 327](#_Toc464629386)

[Pascal's Triangle 331](#_Toc464629387)

[Pascal's Triangle II 333](#_Toc464629388)

[Sort Colors 335](#_Toc464629389)

[Wiggle Sort II 摆动排序 340](#_Toc464629390)

[Product of Array Except Self 350](#_Toc464629391)

[Spiral Matrix 352](#_Toc464629392)

[Combination Sum 353](#_Toc464629393)

[Combination Sum II 356](#_Toc464629394)

[Combination Sum III 359](#_Toc464629395)

[Combinations 362](#_Toc464629396)

[Letter Combinations of a Phone Number 364](#_Toc464629397)

[Generate Parentheses 368](#_Toc464629398)

[Subsets II 375](#_Toc464629399)

[列举所有出栈顺序（微软面试） 379](#_Toc464629400)

[Permutations 386](#_Toc464629401)

[Permutations II 395](#_Toc464629402)

[Next Permutation 400](#_Toc464629403)

[Permutation Sequence 404](#_Toc464629404)

[Hard 408](#_Toc464629405)

[First Missing Positive 408](#_Toc464629406)

[Find the Duplicate Number 414](#_Toc464629407)

[二分查找 417](#_Toc464629408)

[First Bad Version 417](#_Toc464629409)

[Search for a Range 418](#_Toc464629410)

[Guess Number Higher or Lower 422](#_Toc464629411)

[Find Peak Element 425](#_Toc464629412)

[Search Insert Position 432](#_Toc464629413)

[深搜 434](#_Toc464629414)

[Number of Islands 434](#_Toc464629415)

[House Robber III 437](#_Toc464629416)

[Course Schedule 445](#_Toc464629417)

[Course Schedule II 451](#_Toc464629418)

[Reconstruct Itinerary 455](#_Toc464629419)

[Clone Graph 462](#_Toc464629420)

[宽搜 463](#_Toc464629421)

[Surrounded Regions 463](#_Toc464629422)

[Minimum Height Trees 469](#_Toc464629423)

[Word Ladder II 475](#_Toc464629424)

[设计 481](#_Toc464629425)

[Implement Queue using Stacks 481](#_Toc464629426)

[Implement Stack using Queues 484](#_Toc464629427)

[Binary Search Tree Iterator 491](#_Toc464629428)

[Peeking Iterator 496](#_Toc464629429)

[Insert Delete GetRandom O(1) 498](#_Toc464629430)

[Design Twitter 503](#_Toc464629431)

[Trie 511](#_Toc464629432)

[Implement Trie (Prefix Tree) 511](#_Toc464629433)

[Add and Search Word - Data structure design 514](#_Toc464629434)

[贪心 522](#_Toc464629435)

[Patching Array 522](#_Toc464629436)

[Jump Game 525](#_Toc464629437)

[Jump Game II 526](#_Toc464629438)

[Gas Station 528](#_Toc464629439)

[Hard 531](#_Toc464629440)

[Remove Duplicate Letters 531](#_Toc464629441)

[Candy 535](#_Toc464629442)

[Wildcard Matching通配符匹配 537](#_Toc464629443)

[堆 543](#_Toc464629444)

[Kth Largest Element in an Array 543](#_Toc464629445)

[Top K Frequent Elements 551](#_Toc464629446)

[分而治之 557](#_Toc464629447)

[HashMap 557](#_Toc464629448)

[Valid Anagram 557](#_Toc464629449)

[Two Sum 558](#_Toc464629450)

[Fraction to Recurring Decimal 559](#_Toc464629451)

[Happy Number 565](#_Toc464629452)

[Group Anagrams 570](#_Toc464629453)

[Word Pattern 573](#_Toc464629454)

[Isomorphic Strings 576](#_Toc464629455)

[Intersection of Two Arrays 579](#_Toc464629456)

[Intersection of Two Arrays II 582](#_Toc464629457)

[Longest Palindrome 586](#_Toc464629458)

[H-Index 588](#_Toc464629459)

[Valid Sudoku 593](#_Toc464629460)

[腾讯基础研究面试题目 596](#_Toc464629461)

[回溯 596](#_Toc464629462)

[Binary Watch 596](#_Toc464629463)

[Word Search 600](#_Toc464629464)

[Gray Code 604](#_Toc464629465)

[Restore IP Addresses 608](#_Toc464629466)

[Palindrome Partitioning 611](#_Toc464629467)

[N-Queens 615](#_Toc464629468)

[N-Queens II 619](#_Toc464629469)

[双指针 621](#_Toc464629470)

[Two Sum II - Input array is sorted 621](#_Toc464629471)

[3Sum 624](#_Toc464629472)

[3Sum Closest 626](#_Toc464629473)

[4Sum 630](#_Toc464629474)

[Reverse Vowels of a String 637](#_Toc464629475)

[Valid Palindrome 641](#_Toc464629476)

[Reverse String 643](#_Toc464629477)

[Implement strStr()—KMP 646](#_Toc464629478)

[Longest Substring Without Repeating Characters 648](#_Toc464629479)

[Container With Most Water 653](#_Toc464629480)

[Minimum Size Subarray Sum 655](#_Toc464629481)

[Merge Sorted Array 658](#_Toc464629482)

[智力题 659](#_Toc464629483)

[Bulb Switcher 659](#_Toc464629484)

[Nim Game 659](#_Toc464629485)

[二叉排序树 661](#_Toc464629486)

[线段树 661](#_Toc464629487)

[二叉线索树 661](#_Toc464629488)

[Range Sum Query - Mutable 661](#_Toc464629489)

[并查集 667](#_Toc464629490)

[Number of Islands 667](#_Toc464629491)

[排序 670](#_Toc464629492)

[数学 670](#_Toc464629493)

[Count Primes 670](#_Toc464629494)

[Reverse Integer 675](#_Toc464629495)

[数据库 677](#_Toc464629496)

[Employees Earning More Than Their Managers 677](#_Toc464629497)

[Duplicate Emails 678](#_Toc464629498)

[Combine Two Tables 680](#_Toc464629499)

[Customers Who Never Order 681](#_Toc464629500)

[Rising Temperature 682](#_Toc464629501)

[Second Highest Salary 684](#_Toc464629502)

[Delete Duplicate Emails 686](#_Toc464629503)

[中级 689](#_Toc464629504)

[Consecutive Numbers 689](#_Toc464629505)

[Rank Scores 690](#_Toc464629506)

[金典和牛客网编程题 693](#_Toc464629507)

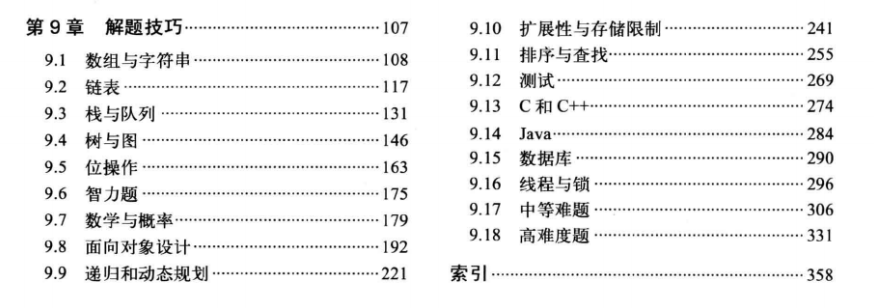
[机器人走方格II 693](#_Toc464629508)

[Java Scanner读取数据 696](#_Toc464629509)

[Java语法技巧 697](#_Toc464629510)

程序员面试金典目录





## 位操作

### Single Number

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/single-number/submissions/)

* Total Accepted: **161721**
* Total Submissions: **311624**
* Difficulty: **Easy**

Given an array of integers, every element appears *twice* except for one. Find that single one.

**Note:**  
Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

public int singleNumber(int[] nums) {

int res=0;

for(int n: nums)

{

res=res^n;

}

return res;

}

### Find the Difference

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/find-the-difference/submissions/)

* Total Accepted: **25589**
* Total Submissions: **51443**
* Difficulty: **Easy**

Given two strings ***s*** and ***t*** which consist of only lowercase letters.

String ***t*** is generated by random shuffling string ***s*** and then add one more letter at a random position.

Find the letter that was added in ***t***.

**Example:**

Input:

s = "abcd"

t = "abcde"

Output:

e

Explanation:

'e' is the letter that was added.

我的做法，同single number

public char findTheDifference(String s, String t) {

int sum=0;

for(int i=0; i<s.length(); i++)

{

char ch=s.charAt(i);

sum^=(ch-'a');

}

for(int i=0; i<t.length(); i++)

{

char ch=t.charAt(i);

sum^=(ch-'a');

}

return (char)(sum+'a');

}

**public** **char** **findTheDifference**(String s, String t) {

**char** c = 0;

**for** (**int** i = 0; i < s.length(); ++i) {

c ^= s.charAt(i);

}

**for** (**int** i = 0; i < t.length(); ++i) {

c ^= t.charAt(i);

}

**return** c;

}

Could be optimized as:

char c = 0;  
for (int i = 0; i < s.length(); ++i) {  
c ^= s.charAt(i);  
c ^= t.charAt(i);  
}  
c ^= t.charAt(t.length() - 1);  
return c;

### Single Number III

[My Submissions](https://leetcode.com/problems/single-number-iii/submissions/)

Question

Total Accepted: **28450** Total Submissions: **66148** Difficulty: **Medium**

Given an array of numbers nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

For example:

Given nums = [1, 2, 1, 3, 2, 5], return [3, 5].

**Note**:

1. The order of the result is not important. So in the above example, [5, 3] is also correct.
2. Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

Once again, we need to use XOR to solve this problem. But this time, we need to do it in two passes:

* In the first pass, we XOR all elements in the array, and get the XOR of the two numbers we need to find. Note that since the two numbers are distinct, so there must be a set bit (that is, the bit with value '1') in the XOR result. Find out an arbitrary set bit (for example, the rightmost set bit).
* In the second pass, we divide all numbers into two groups, one with the aforementioned bit set, another with the aforementinoed bit unset. Two different numbers we need to find must fall into thte two distrinct groups. XOR numbers in each group, we can find a number in either group.

**Complexity:**

* Time: *O* (*n*)
* Space: *O* (1)

**A Corner Case:**

* When diff == numeric\_limits<int>::min(), -diff is alsonumeric\_limits<int>::min(). Therefore, the value of diff after executing diff &= -diff is still numeric\_limits<int>::min(). The answer is still correct.

public class Solution {

public int[] singleNumber(int[] nums) {

// Pass 1 :

// Get the XOR of the two numbers we need to find

int diff = 0;

for (int num : nums) {

diff ^= num;

}

// Get its last set bit

diff &= -diff;

// Pass 2 :

int[] rets = {0, 0}; // this array stores the two numbers we will return

for (int num : nums)

{

if ((num & diff) == 0) // the bit is not set

{

rets[0] ^= num;

}

else // the bit is set

{

rets[1] ^= num;

}

}

return rets;

}

}

x + (~x) = -1

-x = ~x + 1

-(-2147483648) = -(0x80000000) = 0x7FFFFFFF + 1 = 0x80000000 = -2147483648

首先计算nums数组中所有数字的异或，记为xor

令lowbit = xor & -xor，lowbit的含义为xor从低位向高位，第一个非0位所对应的数字

例如假设xor = 6（二进制：0110），则-xor为（二进制：1010，-6的补码，two's complement）

则lowbit = 2（二进制：0010）

根据异或运算的性质，“同0异1”

记只出现一次的两个数字分别为a与b

可知a & lowbit与b & lowbit的结果一定不同

通过这种方式，即可将a与b拆分开来

Python代码：

**class** **Solution**:

# @param {integer[]} nums

# @return {integer[]}

**def** **singleNumber**(self, nums):

xor = reduce(**lambda** x, y : x ^ y, nums)

lowbit = xor & -xor

a = b = 0

**for** num **in** nums:

**if** num & lowbit:

a ^= num

**else**:

b ^= num

**return** [a, b]

数组转换成list

List<Integer> list=new ArrayList<>();

list.addAll(Arrays.asList(new Integer[]{1,2,3}));

|  |  |
| --- | --- |
| static <T> [**List**](file:///E:\docsJava\api\java\util\List.html)<T> | [**asList**](file:///E:\docsJava\api\java\util\Arrays.html#asList-T...-)(T... a)  Returns a fixed-size list backed by the specified array. |

### Missing Number

[My Submissions](https://leetcode.com/problems/missing-number/submissions/)

Question

Total Accepted: **43393** Total Submissions: **109610** Difficulty: **Medium**

Given an array containing *n* distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array.

For example,  
Given *nums* = [0, 1, 3] return 2.

**Note**:  
Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

The basic idea is to use XOR operation. We all know that a^b^b =a, which means two xor operations with the same number will eliminate the number and reveal the original number. In this solution, I apply XOR operation to both the index and value of the array. In a complete array with no missing numbers, the index and value should be perfectly corresponding( nums[index] = index), so in a missing array, what left finally is the missing number.

public **int** missingNumber(**int**[] nums) {

**int** **xor** = 0, i = 0;

**for** (i = 0; i < nums.**length**; i++) {

**xor** = **xor** ^ i ^ nums[i];

}

**return** **xor** ^ i;

}

1.XOR

**public** **int** missingNumber(**int**[] nums) { //xor

**int** res = nums.length;

**for**(**int** i=0; i<nums.length; i++){

res ^= i;

res ^= nums[i];

}

**return** res;

}

2.SUM

**public** **int** missingNumber(**int**[] nums) { //sum

**int** len = nums.length;

**int** sum = (0+len)\*(len+1)/2;

**for**(**int** i=0; i<len; i++)

sum-=nums[i];

**return** sum;

}

3.Binary Search

**public** **int** missingNumber(**int**[] nums) { //binary search

Arrays.sort(nums);

**int** left = 0, right = nums.length, mid= (left + right)/2;

**while**(left<right){

mid = (left + right)/2;

**if**(nums[mid]>mid) right = mid;

**else** left = mid+1;

}

**return** left;

}

Summary:

If the array is in order, I prefer Binary Search method. Otherwise, the XOR method is better.

### Counting Bits

[My Submissions](https://leetcode.com/problems/counting-bits/submissions/)

Question

Total Accepted: **659** Total Submissions: **1120** Difficulty: **Medium**

Given a non negative integer number **num**. For every numbers **i** in the range **0 ≤ i ≤ num** calculate the number of 1's in their binary representation and return them as an array.

**Example:**  
For num = 5 you should return [0,1,1,2,1,2].

**Follow up:**

* It is very easy to come up with a solution with run time **O(n\*sizeof(integer))**. But can you do it in linear time **O(n)** /possibly in a single pass?
* Space complexity should be **O(n)**.
* Can you do it like a boss? Do it without using any builtin function like **\_\_builtin\_popcount** in c++ or in any other language.

**Hint:**

1. You should make use of what you have produced already.
2. Divide the numbers in ranges like [2-3], [4-7], [8-15] and so on. And try to generate new range from previous.
3. Or does the odd/even status of the number help you in calculating the number of 1s?

An easy recurrence for this problem is f[i] = f[i / 2] + i % 2.

1. **public** **int**[] countBits(**int** num) {
2. **int**[] f = **new** **int**[num + 1];
3. **for** (**int** i=1; i<=num; i++) f[i] = f[i >> 1] + (i & 1);
4. **return** f;
5. }

**public** **int**[] countBits(**int** num) {

**int**[] res= **new** **int**[num + 1];

res[0] = 0;

**if** (num == 0) {

**return** res;

}

**for** (**int** i = 1; i <= num; i++) {

//even

**if** ((i & 1) == 0) {

res[i] = res[i >> 1];

}

//odd

**else** {

res[i] = res[i - 1] + 1;

}

}

**return** res;

}

### Maximum Product of Word Lengths

[My Submissions](https://leetcode.com/problems/maximum-product-of-word-lengths/submissions/)

Question

Total Accepted: **15301** Total Submissions: **39331** Difficulty: **Medium**

Given a string array words, find the maximum value of length(word[i]) \* length(word[j]) where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

**Example 1:**

Given ["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]  
Return 16  
The two words can be "abcw", "xtfn".

**Example 2:**

Given ["a", "ab", "abc", "d", "cd", "bcd", "abcd"]  
Return 4  
The two words can be "ab", "cd".

**Example 3:**

Given ["a", "aa", "aaa", "aaaa"]  
Return 0  
No such pair of words.

窗体顶端

public **class** **Solution** {

public int maxProduct(String[] words) {

int max = 0;

Arrays.sort(words, new Comparator<String>(){

public int compare(String a, String b){

**return** b.length() - a.length();

}

});

int[] masks = new int[words.length]; // alphabet masks

**for**(int i = 0; i < masks.length; i++){

**for**(char c: words[i].toCharArray()){

masks[i] |= 1 << (c - 'a');

}

}

**for**(int i = 0; i < masks.length; i++){

**if**(words[i].length() \* words[i].length() <= max) **break**; //prunning

**for**(int j = i + 1; j < masks.length; j++){

**if**((masks[i] & masks[j]) == 0){

max = Math.max(max, words[i].length() \* words[j].length());

**break**; //prunning

}

}

}

**return** max;

}

}

* [solution-sharing](https://leetcode.com/discuss/tag/solution-sharing)

* [java](https://leetcode.com/discuss/tag/java)

* [solution](https://leetcode.com/discuss/tag/solution)

https://www.gravatar.com/avatar/91bc2a00d2a4240ff3b15495e2e19cdc?s=40 [asked](https://leetcode.com/discuss/74589/32ms-java-ac-solution) **Dec 15, 2015** in [**Maximum Product of Word Lengths**](https://leetcode.com/discuss/oj/maximum-product-of-word-lengths) by [**cc9208**](https://leetcode.com/discuss/user/cc9208) (3,190 points)   
edited **Dec 17, 2015** by [**cc9208**](https://leetcode.com/discuss/user/cc9208)

I think there is no need to sort

https://www.gravatar.com/avatar/fdcf419d76ac0b28755f86ad7c5bf34b?s=20 [commented](https://leetcode.com/discuss/74589/32ms-java-ac-solution?show=74868#c74868) Dec 17, 2015 by [**lida2**](https://leetcode.com/discuss/user/lida2)



I think it is necessary for pruning.

https://www.gravatar.com/avatar/15f6ade937615a3d2669d6ec1e141b90?s=20 [commented](https://leetcode.com/discuss/74589/32ms-java-ac-solution?show=74984#c74984) Dec 18, 2015 by [**hidemon**](https://leetcode.com/discuss/user/hidemon)



Since the overall complexity is O(n^2), sorting does not matter too much. Sorting will enable early pruning.

https://www.gravatar.com/avatar/92f53a5c71e5d2b0e23bd786997e53b2?s=20 [commented](https://leetcode.com/discuss/74589/32ms-java-ac-solution?show=75269#c75269) Dec 20, 2015 by [**ZhaoMai**](https://leetcode.com/discuss/user/ZhaoMai)



i'm a little bit confused about that “if(words[i].length() \* words[i].length() <= max) break; //prunning” ？why if the square of words[i].length smaller than max then break?

https://www.gravatar.com/avatar/a39379fc46b8f3de8a56ee80700524ac?s=20 [commented](https://leetcode.com/discuss/74589/32ms-java-ac-solution?show=75276#c75276) Dec 20, 2015 by [**shifenjiandan**](https://leetcode.com/discuss/user/shifenjiandan)



That's what sorting is for. So that you can exit if squaring is not larger than the max. Because if you keep going, the product will not get larger.

https://www.gravatar.com/avatar/f378d294ed82602ebd400a1165df9123?s=20 [commented](https://leetcode.com/discuss/74589/32ms-java-ac-solution?show=75285#c75285) Dec 20, 2015 by [**johnwyz88**](https://leetcode.com/discuss/user/johnwyz88)



And I just tested it without sorting and it's 19 ms. So the nlog(n) could be significant I suppose...

https://www.gravatar.com/avatar/f378d294ed82602ebd400a1165df9123?s=20 [commented](https://leetcode.com/discuss/74589/32ms-java-ac-solution?show=75287#c75287) Dec 20, 2015 by [**johnwyz88**](https://leetcode.com/discuss/user/johnwyz88)



2 Answers

窗体顶端



**0**votes

窗体底端

窗体顶端

How would this work on the following words: "abcde", "fg","hijklmnop" seems that you'll calculate "abcde" \* "fg" and then break before calculating the correct answer. Or perhaps I'm not fully understanding your solution.

https://www.gravatar.com/avatar/d923a0fc9bd985fc35f5e0922c6d7176?s=40 [answered](https://leetcode.com/discuss/74589/32ms-java-ac-solution?show=74610#a74610) **Dec 15, 2015** by [**galster**](https://leetcode.com/discuss/user/galster) (220 points)

1. The array is sorted as {"hijklmnop", "abcde", "fg"}.
2. The "break" only breaks out the inner "for" loop

https://www.gravatar.com/avatar/91bc2a00d2a4240ff3b15495e2e19cdc?s=20 [commented](https://leetcode.com/discuss/74589/32ms-java-ac-solution?show=74611#c74611) Dec 15, 2015 by [**cc9208**](https://leetcode.com/discuss/user/cc9208)



yes, I realized this later. Thanks for responding.

https://www.gravatar.com/avatar/d923a0fc9bd985fc35f5e0922c6d7176?s=20 [commented](https://leetcode.com/discuss/74589/32ms-java-ac-solution?show=74614#c74614) Dec 15, 2015 by [**galster**](https://leetcode.com/discuss/user/galster)



窗体底端

窗体顶端



**0**votes

窗体底端

窗体顶端

A slight optimization is moving the second for loop into the first one, so the first loop can end early with pruning before calculating the masks for words with small lengths

**public** **static** **int** maxProduct(String[] words) {

**if** (words == **null** || words.length == 0)

**return** 0;

**int** len = words.length;

**int**[] **value** = **new** **int**[len];

**for** (**int** i = 0; i < len; i++) {

String tmp = words[i];

**value**[i] = 0;

**for** (**int** j = 0; j < tmp.length(); j++) {

**value**[i] |= 1 << (tmp.charAt(j) - 'a');

}

}

**int** maxProduct = 0;

**for** (**int** i = 0; i < len; i++)

**for** (**int** j = i + 1; j < len; j++) {

**if** ((**value**[i] & **value**[j]) == 0 && (words[i].length() \* words[j].length() > maxProduct))

maxProduct = words[i].length() \* words[j].length();

}

**return** maxProduct;

}

### Repeated DNA Sequences

[My Submissions](https://leetcode.com/problems/repeated-dna-sequences/submissions/)

Question

Total Accepted: **39514** Total Submissions: **160686** Difficulty: **Medium**

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

For example,

Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT",

Return:

["AAAAACCCCC", "CCCCCAAAAA"].

**public** List<String> findRepeatedDnaSequences(String s) {

Set<Integer> words = **new** HashSet<>();

Set<Integer> doubleWords = **new** HashSet<>();

List<String> rv = **new** ArrayList<>();

**char**[] **map** = **new** **char**[26];

//map['A' - 'A'] = 0;

**map**['C' - 'A'] = 1;

**map**['G' - 'A'] = 2;

**map**['T' - 'A'] = 3;

**for**(**int** i = 0; i < s.length() - 9; i++) {

**int** v = 0;

**for**(**int** j = i; j < i + 10; j++) {

v <<= 2;

v |= **map**[s.charAt(j) - 'A'];

}

**if**(!words.add(v) && doubleWords.add(v)) {

rv.add(s.substring(i, i + 10));

}

}

**return** rv;

}

我的做法，数出10位长的字符串。记录相同字符串的出现次数

public List<String> findRepeatedDnaSequences(String s) {

List<String> res=new ArrayList<String>();

HashMap<String, Integer> map=new HashMap<>();

for(int i=0; i<s.length()-9; i++)

{

String sub=s.substring(i, i+10);

if(!map.containsKey(sub))

{

map.put(sub, 1);

}

else

{

map.put(sub, map.get(sub)+1);

}

}

for(String ss: map.keySet())

{

if(map.get(ss)>1) res.add(ss);

}

return res;

}

### Subsets 书

[My Submissions](https://leetcode.com/problems/subsets/submissions/)

Question

Total Accepted: **88959** Total Submissions: **287842** Difficulty: **Medium**

Given a set of distinct integers, *nums*, return all possible subsets.

**Note:**

* Elements in a subset must be in non-descending order.
* 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],

[]

]

**class** Solution {

**public**:

**vector**<**vector**<**int**> > subsets(**vector**<**int**> &S) {

sort (S.begin(), S.end());

**int** elem\_num = S.size();

**int** subset\_num = pow (2, elem\_num);

**vector**<**vector**<**int**> > subset\_set (subset\_num, **vector**<**int**>());

**for** (**int** i = 0; i < elem\_num; i++)

**for** (**int** j = 0; j < subset\_num; j++)

**if** ((j >> i) & 1)

subset\_set[j].push\_back (S[i]);

**return** subset\_set;

}

};

### Bitwise AND of Numbers Range

[My Submissions](https://leetcode.com/problems/bitwise-and-of-numbers-range/submissions/)

Question

Total Accepted: **30644** Total Submissions: **102936** Difficulty: **Medium**

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.

For example, given the range [5, 7], you should return 4.

The idea is very simple:

1. last bit of (odd number & even number) is 0.
2. when m != n, There is at least an odd number and an even number, so the last bit position result is 0.
3. Move m and n rigth a position.

Keep doing step 1,2,3 until m equal to n, use a factor to record the iteration time.

**public** **class** **Solution** {

**public** **int** rangeBitwiseAnd(**int** m, **int** n) {

**if**(m == 0){

**return** 0;

}

**int** moveFactor = 1;

**while**(m != n){

m >>= 1;

n >>= 1;

moveFactor <<= 1;

}

**return** m \* moveFactor;

}

}

**public** **class** **Solution** {

**public** **int** rangeBitwiseAnd(**int** m, **int** n) {

**int** diffBits = 0;

**while** (m != n) {

m >>= 1;

n >>= 1;

diffBits++;

}

**return** n<<diffBits;

}

}

### Reverse Bits

[My Submissions](https://leetcode.com/problems/reverse-bits/submissions/)

QuestionEditorial Solution

Total Accepted: **61394** Total Submissions: **209642** Difficulty: **Easy**

Reverse bits of a given 32 bits unsigned integer.

For example, given input 43261596 (represented in binary as **00000010100101000001111010011100**), return 964176192 (represented in binary as **00111001011110000010100101000000**).

**Follow up**:  
If this function is called many times, how would you optimize it?

Related problem: [Reverse Integer](https://leetcode.com/problems/reverse-integer/)

class Solution {

public:

uint32\_t reverseBits(uint32\_t n) {

n = (n >> 16) | (n <**<** 16);

n = ((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) <**<** 8);

n = ((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) <**<** 4);

n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) <**<** 2);

n = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555555) <**<** 1);

return n;

}

};

for 8 bit binary number abcdefgh, the process is as follow:

abcdefgh -> efghabcd -> ghefcdab -> hgfedcba

The Java solution is straightforward, just bitwise operation:

**public** **int** reverseBits(**int** n) {

**int** result = 0;

**for** (**int** i = 0; i < 32; i++) {

result += n & 1;

n >>>= 1; // CATCH: must do unsigned shift

**if** (i < 31) // CATCH: for last digit, don't shift!

result <<= 1;

}

**return** result;

}

How to optimize if this function is called multiple times? We can divide an int into 4 bytes, and reverse each byte then combine into an int. For each byte, we can use cache to improve performance.

// cache

**private** final Map<Byte, Integer> cache = **new** HashMap<Byte, Integer>();

**public** **int** reverseBits(**int** n) {

**byte**[] bytes = **new** **byte**[4];

**for** (**int** i = 0; i < 4; i++) // convert int into 4 bytes

bytes[i] = (**byte**)((n >>> 8\*i) & 0xFF);

**int** result = 0;

**for** (**int** i = 0; i < 4; i++) {

result += reverseByte(bytes[i]); // reverse per byte

**if** (i < 3)

result <<= 8;

}

**return** result;

}

**private** **int** reverseByte(**byte** b) {

Integer **value** = cache.**get**(b); // first look up from cache

**if** (**value** != **null**)

**return** **value**;

**value** = 0;

// reverse by bit

**for** (**int** i = 0; i < 8; i++) {

**value** += ((b >>> i) & 1);

**if** (i < 7)

**value** <<= 1;

}

cache.put(b, **value**);

**return** **value**;

}

这个写法不错

public int reverseBits(int n) {

int num=0;

int cnt=32;

while(cnt>0)

{

//int temp=((n&1)==1)?2:0;

num=(num<<1)+(n&1);

System.out.println((n&1)+" "+num);

n=n>>>1;

cnt--;

}

return num;

}

总结：

**比较器的编写**

new Comparator<String>()

{

public int compare(String a, String b){…}

}

**HashSet的使用，可以直接用add方法判断元素是否加入**

|  |  |
| --- | --- |
| boolean | [**add**](file:///E:\docsJava\api\java\util\HashSet.html#add-E-)([**E**](file:///E:\docsJava\api\java\util\HashSet.html) e)  Adds the specified element to this set if it is not already present. |

**数组与list的互相转换**

|  |  |
| --- | --- |
| [**Object**](file:///E:\docsJava\api\java\lang\Object.html)[] | [**toArray**](file:///E:\docsJava\api\java\util\ArrayList.html#toArray--)()  Returns an array containing all of the elements in this list in proper sequence  (from first to last element). |

list.toArray();

list.addAll(new Integer[]{});

**Scanner**

## 递归和动态规划

**Triangle**

Total Accepted: **66661** Total Submissions: **223031** Difficulty: **Medium**

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.

This problem is quite well-formed in my opinion. The triangle has a tree-like structure, which would lead people to think about traversal algorithms such as DFS. However, if you look closely, you would notice that the adjacent nodes always share a 'branch'. In other word, there are **overlapping subproblems**. Also, suppose x and y are 'children' of k. Once minimum paths from x and y to the bottom are known, the minimum path starting from k can be decided in O(1), that is **optimal substructure**. Therefore, dynamic programming would be the best solution to this problem in terms of time complexity.

What I like about this problem even more is that the difference between 'top-down' and 'bottom-up' DP can be 'literally' pictured in the input triangle. For 'top-down' DP, starting from the node on the very top, we recursively find the minimum path sum of each node. When a path sum is calculated, we store it in an array (memoization); the next time we need to calculate the path sum of the same node, just retrieve it from the array. However, you will need a cache that is at least the same size as the input triangle itself to store the pathsum, which takes O(N^2) space. With some clever thinking, it might be possible to release some of the memory that will never be used after a particular point, but the order of the nodes being processed is not straightforwardly seen in a recursive solution, so deciding which part of the cache to discard can be a hard job.

'Bottom-up' DP, on the other hand, is very straightforward: we start from the nodes on the bottom row; the min pathsums for these nodes are the values of the nodes themselves. From there, the min pathsum at the ith node on the kth row would be the lesser of the pathsums of its two children plus the value of itself, i.e.:

minpath[k][i] = min( minpath[k+1][i], minpath[k+1][i+1]) + triangle[k][i];

Or even better, since the row minpath[k+1] would be useless after minpath[k] is computed, we can simply set minpath as a 1D array, and iteratively update itself:

For the kth level:

minpath[i] = min( minpath[i], minpath[i+1]) + triangle[k][i];

Thus, we have the following solution

关键：从最后一行的开始元素开始计算。Dp[i]代表任意行，第i个元素（注意不是代表行的意思，当然这里行号和行长一样）

计算每行的dp[i]时，计算其下面一行的邻近两个元素，选择小的，并加上其自身的值，及为该行该位置的最优值。

int minimumTotal(vector<vector<int> > &triangle) {

int n = triangle.size();

vector<int> minlen(triangle.back());

for (int layer = n-2; layer >= 0; layer--) // For each layer

{

for (int i = 0; i <= layer; i++) // Check its every 'node'

{

// Find the lesser of its two children, and sum the current value in the triangle with it.

minlen[i] = min(minlen[i], minlen[i+1]) + triangle[layer][i];

}

}

return minlen[0];

}

public class Solution {

public int minimumTotal(List<List<Integer>> triangle) {

for(int i = triangle.size() - 2; i >= 0; i--)

for(int j = 0; j <= i; j++)

triangle.get(i).set(j, triangle.get(i).get(j) + Math.min(triangle.get(i + 1).get(j), triangle.get(i + 1).get(j + 1)));

return triangle.get(0).get(0);

}

}

The idea is simple.

1) Go from bottom to top.

2) We start form the row above the bottom row [size()-2].

3) Each number add the smaller number of two numbers that below it.

4) And finally we get to the top we the smallest sum.

public int minimumTotal(List<List<Integer>> triangle) {

int[] A = new int[triangle.size()+1];

for(int i=triangle.size()-1;i>=0;i--){

for(int j=0;j<triangle.get(i).size();j++){

A[j] = Math.min(A[j],A[j+1])+triangle.get(i).get(j);

}

}

return A[0];

}

**Range Sum Query - Immutable**

Total Accepted: **25247** Total Submissions: **103528** Difficulty: **Easy**

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *j*), inclusive.

**Example:**

Given nums = [-2, 0, 3, -5, 2, -1]

sumRange(0, 2) -> 1

sumRange(2, 5) -> -1

sumRange(0, 5) -> -3

**Note:**

1. You may assume that the array does not change.
2. There are many calls to *sumRange* function.

**Java simple O(n) init and O(1) query solution**

public class NumArray {

int[] nums;

public NumArray(int[] nums) {

for(int i = 1; i < nums.length; i++)

nums[i] += nums[i - 1];

this.nums = nums;

}

public int sumRange(int i, int j) {

if(i == 0)

return nums[j];

return nums[j] - nums[i - 1];

}

//下面这个写法好

public class NumArray {

private int[] sums;

public NumArray(int[] nums) {

sums= nums;

for(int i=1; i<nums.length; i++){

sums[i] += sums[i-1];//前一步的累加值，和当前值

}

}

public int sumRange(int i, int j) {

return i==0 ? sums[j] : sums[j]-sums[i-1];

}

}

**Range Sum Query 2D - Immutable**

Total Accepted: **10702** Total Submissions: **48455** Difficulty: **Medium**

Given a 2D matrix *matrix*, find the sum of the elements inside the rectangle defined by its upper left corner (*row*1, *col*1) and lower right corner (*row*2, *col*2).

  
The above rectangle (with the red border) is defined by (row1, col1) = **(2, 1)** and (row2, col2) = **(4, 3)**, which contains sum = **8**.

**Example:**

Given matrix = [

[3, 0, 1, 4, 2],

[5, 6, 3, 2, 1],

[1, 2, 0, 1, 5],

[4, 1, 0, 1, 7],

[1, 0, 3, 0, 5]

]

sumRegion(2, 1, 4, 3) -> 8

sumRegion(1, 1, 2, 2) -> 11

sumRegion(1, 2, 2, 4) -> 12

**Note:**

1. You may assume that the matrix does not change.
2. There are many calls to *sumRegion* function.
3. You may assume that *row*1 ≤ *row*2 and *col*1 ≤ *col*2.

我的做法是，第一行和第一列，单独处理，进行累加。

其他情况，arr[i][j]+=arr[i-1][j]+arr[i][j-1]-arr[i-1][j-1];

sumRegion函数返回时考虑行，列是否有为0情况

需要注意初始矩阵为空的一些情况（如下，标红）

private int[][] dp;

public NumMatrix(int[][] matrix) {

if( matrix == null

|| matrix.length == 0

|| matrix[0].length == 0 ){

return;

}

int m = matrix.length;

int n = matrix[0].length;

dp = new int[m + 1][n + 1];

for(int i = 1; i <= m; i++){

for(int j = 1; j <= n; j++){

dp[i][j] = dp[i - 1][j] + dp[i][j - 1] -dp[i - 1][j - 1] + matrix[i - 1][j - 1] ;

}

}

}

public int sumRegion(int row1, int col1, int row2, int col2) {

int iMin = Math.min(row1, row2);

int iMax = Math.max(row1, row2);

int jMin = Math.min(col1, col2);

int jMax = Math.max(col1, col2);

return dp[iMax + 1][jMax + 1] - dp[iMax + 1][jMin] - dp[iMin][jMax + 1] + dp[iMin][jMin];

}

这个写法好，不用考虑行，列是否有为0情况

private int[][] sumRegion;

public NumMatrix(int[][] matrix) {

if (matrix.length != 0) sumRegion = new int[matrix.length + 1][matrix[0].length + 1];

for (int i = 0; i < matrix.length; i++) {

int sum = 0;

for (int j = 0; j < matrix[0].length; j++) {

sum += matrix[i][j];

sumRegion[i + 1][j + 1] = sum + sumRegion[i][j + 1];

}

}

}

public int sumRegion(int row1, int col1, int row2, int col2) {

return sumRegion[row2 + 1][col2 + 1] - sumRegion[row1][col2 + 1] - sumRegion[row2 + 1][col1] + sumRegion[row1][col1];

}

**Maximum Subarray**

Total Accepted: **105672** Total Submissions: **290399** Difficulty: **Medium**

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [−2,1,−3,4,−1,2,1,−5,4],  
the contiguous subarray [4,−1,2,1] has the largest sum = 6.

[click to show more practice.](https://leetcode.com/problems/maximum-subarray/)

**More practice:**

If you have figured out the O(*n*) solution, try coding another solution using the divide and conquer approach, which is more subtle.

书 中等难题章节

关键在于用max来记录最终结果的最大值，用sum来记录尽可能的延伸下的值。

public int maxSubArray(int[] nums) {

int max=Integer.MIN\_VALUE,sum=0;

for(int i=0;i<nums.length;i++)

{

sum+=nums[i];//记录尽可能的延伸

if(max<sum)

{

max=sum;

}

if(sum<0) sum=0;

}

return max;

}

**Maximum Product Subarray**

Total Accepted: **56155** Total Submissions: **255546** Difficulty: **Medium**

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array [2,3,-2,4],  
the contiguous subarray [2,3] has the largest product = 6.

**Sharing my solution: O(1) space, O(n) running time**

public int maxProduct(int[] A) {

if (A.length == 0) {

return 0;

}

int maxherepre = A[0];

int minherepre = A[0];

int maxsofar = A[0];

int maxhere, minhere;

for (int i = 1; i < A.length; i++) {

maxhere = Math.max(Math.max(maxherepre \* A[i], minherepre \* A[i]), A[i]);

minhere = Math.min(Math.min(maxherepre \* A[i], minherepre \* A[i]), A[i]);

maxsofar = Math.max(maxhere, maxsofar);

maxherepre = maxhere;

minherepre = minhere;

}

return maxsofar;

}

Note: There's no need to use O(n) space, as all that you need is a minhere and maxhere. (local max and local min), then you can get maxsofar (which is global max) from them.

There's a chapter in Programming Pearls 2 that discussed the MaxSubArray problem, the idea is similar.

My idea is the same as yours, although I check if A[i] is positive before getting maxhere and minhere

int maxProduct(int A[], int n) {

if (n == 0) return 0;

int maxProduct = A[0];

int minProduct = A[0];

int maxRes = A[0];

for (int i = 1; i < n; i++)

{

if (A[i] >= 0)

{

maxProduct = max(maxProduct \* A[i], A[i]);

minProduct = min(minProduct \* A[i], A[i]);

}

else

{

int temp = maxProduct;

maxProduct = max(minProduct \* A[i], A[i]);

minProduct = min(temp \* A[i], A[i]);

}

maxRes = max(maxRes, maxProduct);

}

return maxRes;

}

下面这个写法比较容易懂

关键是以该数结尾的子数组的，最大，最小数都需要记录。可以使用的小技巧是，当该数是负数时，交换最大最小数。

int maxProduct(int A[], int n) {

// store the result that is the max we have found so far

int r = A[0];

// imax/imin stores the max/min product of

// subarray that ends with the current number A[i]

for (int i = 1, imax = r, imin = r; i < n; i++) {

// multiplied by a negative makes big number smaller, small number bigger

// so we redefine the extremums by swapping them

if (A[i] < 0)

swap(imax, imin);

// max/min product for the current number is either the current number itself

// or the max/min by the previous number times the current one

imax = max(A[i], imax \* A[i]);

imin = min(A[i], imin \* A[i]);

// the newly computed max value is a candidate for our global result

r = max(r, imax);

}

return r;

}

Loop through the array, each time remember the max and min value for the previous product, the most important thing is to update the max and min value: we have to compare among max \* A[i], min \* A[i] as well as A[i], since this is product, a negative \* negative could be positive.

public class Solution {

public int maxProduct(int[] A) {

if (A == null || A.length == 0) {

return 0;

}

int max = A[0], min = A[0], result = A[0];

for (int i = 1; i < A.length; i++) {

int temp = max;

max = Math.max(Math.max(max \* A[i], min \* A[i]), A[i]);

min = Math.min(Math.min(temp \* A[i], min \* A[i]), A[i]);

if (max > result) {

result = max;

}

}

return result;

}

}

**Longest Increasing Subsequence**

Total Accepted: **23868** Total Submissions: **69772** Difficulty: **Medium**

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?

这个方法非常巧妙，通过len来记录最长递增子序列的长度。

当可以将值插入前面构造的递增序列时，返回的i是len，否则，会覆盖前面相应的值

**Short Java solution using DP O(n log n)**

public class Solution {

public int lengthOfLIS(int[] nums) {

int[] dp = new int[nums.length];

int len = 0;

for(int x : nums) {

int i = Arrays.binarySearch(dp, 0, len, x);

if(i < 0) i = -(i + 1);

dp[i] = x;

if(i == len) len++;

}

return len;

}

}

|  |  |
| --- | --- |
| static int | [**binarySearch**](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/util/Arrays.html#binarySearch%28int[],%20int,%20int,%20int%29)(int[] a, int fromIndex, int toIndex, int key)  Searches a range of the specified array of ints for the specified value using the binary search algorithm. |

**Parameters:**

a - the array to be searched

fromIndex - the index of the first element (inclusive) to be searched

toIndex - the index of the last element (exclusive) to be searched

key - the value to be searched for

Returns:

index of the search key, if it is contained in the array within the specified range; otherwise, (-(*insertion point*) - 1). The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element in the range greater than the key, or toIndex if all elements in the range are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

The basic idea is present in the majority of solutions shared for this task, I have only tried to implement it in a manner as concise as possible without damaging the code readability.

The idea is that as you iterate the sequence, you keep track of the minimum value a subsequence of given length might end with, for all so far possible subsequence lengths. So dp[i] is the minimum value a subsequence of length i+1 might end with. Having this info, for each new number we iterate to, we can determine the longest subsequence where it can be appended using binary search. The final answer is the length of the longest subsequence we found so far.

Hello, it's really a nice answer. But I have a question, how did you come up the idea of using binary search. And what if we need to return the subsequence, will binary search still work? I tried to print the subsequence of your answer, the answer is not correct. But the length is correct. For example: [10, 9, 2, 5, 6，3, 7, 101, 18] ＝》 your code's answer: [2, 3, 6，7, 101]， But i think the right answer is [2, 5, 6，7, 101]. Thank you.

@julielong That's because dp[] is not storing the sequence as explained by the author in the reply. dp[i] is storing the smallest number such that the length is i+1.

以下是 机试上的做法，基本的。注意，dp[i]的初值一定要设置为1.

这是以nums[i]结尾的最长递增子序列长度。

This solution is taken from this great guy - https://www.youtube.com/watch?v=CE2b\_-XfVDk

public int lengthOfLIS(int[] nums)

{

// Base case

if(nums.length <= 1)

return nums.length;

// This will be our array to track longest sequence length

int T[] = new int[nums.length];

// Fill each position with value 1 in the array

for(int i=0; i < nums.length; i++)

T[i] = 1;

// Mark one pointer at i. For each i, start from j=0.

for(int i=1; i < nums.length; i++)

{

for(int j=0; j < i; j++)

{

// It means next number contributes to increasing sequence.

if(nums[j] < nums[i])

{

// But increase the value only if it results in a larger value of the sequence than T[i]

// It is possible that T[i] already has larger value from some previous j'th iteration

if(T[j] + 1 > T[i])

{

T[i] = T[j] + 1;

}

}

}

}

// Find the maximum length from the array that we just generated

int longest = 0;

for(int i=0; i < T.length; i++)

longest = Math.max(longest, T[i]);

return longest;

}

public int lengthOfLIS(int[] nums)

{

List<Integer> sequence = new ArrayList();

for(int n : nums) update(sequence, n);

return sequence.size();

}

private void update(List<Integer> seq, int n)

{

if(seq.isEmpty() || seq.get(seq.size() - 1) < n) seq.add(n);

else

{

seq.set(findFirstLargeEqual(seq, n), n);

}

}

private int findFirstLargeEqual(List<Integer> seq, int target)

{

int lo = 0;

int hi = seq.size() - 1;

while(lo < hi)

{

int mid = lo + (hi - lo) / 2;

if(seq.get(mid) < target) lo = mid + 1;

else hi = mid;

}

return lo;

}

**Perfect Squares**

Total Accepted: **30491** Total Submissions: **94665** Difficulty: **Medium**

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.

我的做法是用动态规划的背包问题解决。

下面是我的代码，完全背包，顺序遍历；背包正好装满，初值设为max；价值最大是指数字个数最少

//最少需要几个完全平方数，和才为n

public static int numSquares(int n) {

int max=(int)Math.sqrt(n);

int[] dp=new int[n+1];//和正好为i时需要的最少完全平方数

for(int i=0;i<=n;i++) dp[i]=Integer.MAX\_VALUE;

dp[0]=0;

dp[1]=1;

for(int i=1;i<=max;i++)

{

for(int j=i\*i;j<=n;j++)

{

if(dp[j-i\*i]!=Integer.MAX\_VALUE)

dp[j]=Math.min(dp[j],dp[j-i\*i]+1);

}

}

return dp[n];

}

**Summary of 4 different solutions (BFS, DP, static DP and mathematics)**

Came up with the 2 solutions of breadth-first search and dynamic programming. Also "copied" StefanPochmann's static dynamic programming solution (https://leetcode.com/discuss/56993/static-dp-c-12-ms-python-172-ms-ruby-384-ms) and davidtan1890's mathematical solution (https://leetcode.com/discuss/57066/4ms-c-code-solve-it-mathematically) here with minor style changes and some comments. Thank Stefan and David for posting their nice solutions!

**1.Dynamic Programming:** 440ms

class Solution

{

public:

int numSquares(int n)

{

if (n <= 0)

{

return 0;

}

// cntPerfectSquares[i] = the least number of perfect square numbers

// which sum to i. Note that cntPerfectSquares[0] is 0.

vector<int> cntPerfectSquares(n + 1, INT\_MAX);

cntPerfectSquares[0] = 0;

for (int i = 1; i <= n; i++)

{

// For each i, it must be the sum of some number (i - j\*j) and

// a perfect square number (j\*j).

for (int j = 1; j\*j <= i; j++)

{

cntPerfectSquares[i] =

min(cntPerfectSquares[i], cntPerfectSquares[i - j\*j] + 1);

}

}

return cntPerfectSquares.back();

}

};

**2.Static Dynamic Programming:** 12ms

class Solution

{

public:

int numSquares(int n)

{

if (n <= 0)

{

return 0;

}

// cntPerfectSquares[i] = the least number of perfect square numbers

// which sum to i. Since cntPerfectSquares is a static vector, if

// cntPerfectSquares.size() > n, we have already calculated the result

// during previous function calls and we can just return the result now.

static vector<int> cntPerfectSquares({0});

// While cntPerfectSquares.size() <= n, we need to incrementally

// calculate the next result until we get the result for n.

while (cntPerfectSquares.size() <= n)

{

int m = cntPerfectSquares.size();

int cntSquares = INT\_MAX;

for (int i = 1; i\*i <= m; i++)

{

cntSquares = min(cntSquares, cntPerfectSquares[m - i\*i] + 1);

}

cntPerfectSquares.push\_back(cntSquares);

}

return cntPerfectSquares[n];

}

};

**3.Mathematical Solution:** 4ms

class Solution

{

private:

int is\_square(int n)

{

int sqrt\_n = (int)(sqrt(n));

return (sqrt\_n\*sqrt\_n == n);

}

public:

// Based on Lagrange's Four Square theorem, there

// are only 4 possible results: 1, 2, 3, 4.

int numSquares(int n)

{

// If n is a perfect square, return 1.

if(is\_square(n))

{

return 1;

}

// The result is 4 if and only if n can be written in the

// form of 4^k\*(8\*m + 7). Please refer to

// Legendre's three-square theorem.

while ((n & 3) == 0) // n%4 == 0

{

n >>= 2;

}

if ((n & 7) == 7) // n%8 == 7

{

return 4;

}

// Check whether 2 is the result.

int sqrt\_n = (int)(sqrt(n));

for(int i = 1; i <= sqrt\_n; i++)

{

if (is\_square(n - i\*i))

{

return 2;

}

}

return 3;

}

};

**4.Breadth-First Search:** 80ms

class Solution

{

public:

int numSquares(int n)

{

if (n <= 0)

{

return 0;

}

// perfectSquares contain all perfect square numbers which

// are smaller than or equal to n.

vector<int> perfectSquares;

// cntPerfectSquares[i - 1] = the least number of perfect

// square numbers which sum to i.

vector<int> cntPerfectSquares(n);

// Get all the perfect square numbers which are smaller than

// or equal to n.

for (int i = 1; i\*i <= n; i++)

{

perfectSquares.push\_back(i\*i);

cntPerfectSquares[i\*i - 1] = 1;

}

// If n is a perfect square number, return 1 immediately.

if (perfectSquares.back() == n)

{

return 1;

}

// Consider a graph which consists of number 0, 1,...,n as

// its nodes. Node j is connected to node i via an edge if

// and only if either j = i + (a perfect square number) or

// i = j + (a perfect square number). Starting from node 0,

// do the breadth-first search. If we reach node n at step

// m, then the least number of perfect square numbers which

// sum to n is m. Here since we have already obtained the

// perfect square numbers, we have actually finished the

// search at step 1.

queue<int> searchQ;

for (auto& i : perfectSquares)

{

searchQ.push(i);

}

int currCntPerfectSquares = 1;

while (!searchQ.empty())

{

currCntPerfectSquares++;

int searchQSize = searchQ.size();

for (int i = 0; i < searchQSize; i++)

{

int tmp = searchQ.front();

// Check the neighbors of node tmp which are the sum

// of tmp and a perfect square number.

for (auto& j : perfectSquares)

{

if (tmp + j == n)

{

// We have reached node n.

return currCntPerfectSquares;

}

else if ((tmp + j < n) && (cntPerfectSquares[tmp + j - 1] == 0))

{

// If cntPerfectSquares[tmp + j - 1] > 0, this is not

// the first time that we visit this node and we should

// skip the node (tmp + j).

cntPerfectSquares[tmp + j - 1] = currCntPerfectSquares;

searchQ.push(tmp + j);

}

else if (tmp + j > n)

{

// We don't need to consider the nodes which are greater ]

// than n.

break;

}

}

searchQ.pop();

}

}

return 0;

}

};

dp[n] indicates that the perfect squares count of the given n, and we have:

dp[0] = 0

dp[1] = dp[0]+1 = 1

dp[2] = dp[1]+1 = 2

dp[3] = dp[2]+1 = 3

dp[4] = Min{ dp[4-1\*1]+1, dp[4-2\*2]+1 }

= Min{ dp[3]+1, dp[0]+1 }

= 1

dp[5] = Min{ dp[5-1\*1]+1, dp[5-2\*2]+1 }

= Min{ dp[4]+1, dp[1]+1 }

= 2

.

.

.

dp[13] = Min{ dp[13-1\*1]+1, dp[13-2\*2]+1, dp[13-3\*3]+1 }

= Min{ dp[12]+1, dp[9]+1, dp[4]+1 }

= 2

.

.

.

dp[n] = Min{ dp[n - i\*i] + 1 }, n - i\*i >=0 && i >= 1

and the sample code is like below:

public int numSquares(int n) {

int[] dp = new int[n + 1];

Arrays.fill(dp, Integer.MAX\_VALUE);

dp[0] = 0;

for(int i = 1; i <= n; ++i) {

int min = Integer.MAX\_VALUE;

int j = 1;

while(i - j\*j >= 0) {

min = Math.min(min, dp[i - j\*j] + 1);

++j;

}

dp[i] = min;

}

return dp[n];

}

Hope it can help to understand the DP solution.

Just regular DP. Time Complexity: n \* sqrt(n) Space: O(n)

public class Solution {

public int numSquares(int n) {

int[] dp = new int[n + 1];

Arrays.fill(dp, Integer.MAX\_VALUE);

dp[0] = 0;

for(int i = 0; i <= n; i++){

for(int j = 1; i + j \* j <= n; j++){

dp[i + j \* j] = Math.min(dp[i + j \* j], dp[i] + 1);

}

}

return dp[n];

}

}

### Coin Change

Total Accepted: **17683** Total Submissions: **71468** Difficulty: **Medium**

You are given coins of different denominations and a total amount of money *amount*. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

**Example 1:**  
coins = [1, 2, 5], amount = 11  
return 3 (11 = 5 + 5 + 1)

**Example 2:**  
coins = [2], amount = 3  
return -1.

**Note**:  
You may assume that you have an infinite number of each kind of coin.

我的做法，同上一题。

这是一个完全背包问题。顺序遍历，且背包需要装满，则初值设为max。

public int coinChange(int[] coins, int amount) {

int[] dp=new int[amount+1];

Arrays.fill(dp,Integer.MAX\_VALUE);

dp[0]=0;

for(int i=0;i<coins.length;i++)

{

for(int j=coins[i];j<=amount;j++)

{

if(dp[j-coins[i]]!=Integer.MAX\_VALUE)

dp[j]=Math.min(dp[j-coins[i]]+1,dp[j]);

}

}

return dp[amount]==Integer.MAX\_VALUE?-1:dp[amount];

}

**O(n\*amount) time O(amount) space DP solution**

class Solution {

public:

int coinChange(vector<int>& coins, int amount) {

int Max = amount + 1;

vector<int> dp(amount + 1, Max);

dp[0] = 0;

for (int i = 1; i <= amount; i++) {

for (int j = 0; j < coins.size(); j++) {

if (coins[j] <= i) {

dp[i] = min(dp[i], dp[i - coins[j]] + 1);

}

}

}

return dp[amount] > amount ? -1 : dp[amount];

}

};

**Recursive Method:**

The idea is very classic dynamic programming: think of the last step we take. Suppose we have already found out the best way to sum up to amount a, then for the last step, we can choose any coin type which gives us a remainder r where r = a-coins[i] for all i's. For every remainder, go through exactly the same process as before until either the remainder is 0 or less than 0 (meaning not a valid solution). With this idea, the only remaining detail is to store the minimum number of coins needed to sum up to r so that we don't need to recompute it over and over again.

Code in Java:

public class Solution {

public int coinChange(int[] coins, int amount) {

if(amount<1) return 0;

return helper(coins, amount, new int[amount]);

}

private int helper(int[] coins, int rem, int[] count) { // rem: remaining coins after the last step; count[rem]: minimum number of coins to sum up to rem

if(rem<0) return -1; // not valid

if(rem==0) return 0; // completed

if(count[rem-1] != 0) return count[rem-1]; // already computed, so reuse

int min = Integer.MAX\_VALUE;

for(int coin : coins) {

int res = helper(coins, rem-coin, count);

if(res>=0 && res < min)

min = 1+res;

}

count[rem-1] = (min==Integer.MAX\_VALUE) ? -1 : min;

return count[rem-1];

}

}

**Iterative Method:**

For the iterative solution, we think in bottom-up manner. Suppose we have already computed all the minimum counts up to sum, what would be the minimum count for sum+1?

Code in Java:

public class Solution {

public int coinChange(int[] coins, int amount) {

if(amount<1) return 0;

int[] dp = new int[amount+1];

int sum = 0;

while(++sum<=amount) {

int min = -1;

for(int coin : coins) {

if(sum >= coin && dp[sum-coin]!=-1) {

int temp = dp[sum-coin]+1;

min = min<0 ? temp : (temp < min ? temp : min);

}

}

dp[sum] = min;

}

return dp[amount];

}

}

If you are interested in my other posts, please feel free to check my Github page here: <https://github.com/F-L-A-G/Algorithms-in-Java>

### Maximal Square

Total Accepted: **27173** Total Submissions: **116197** Difficulty: **Medium**

Given a 2D binary matrix filled with 0's and 1's, find the largest square containing all 1's and return its area.

For example, given the following matrix:

1 0 1 0 0

1 0 1 1 1

1 1 1 1 1

1 0 0 1 0

Return 4.

**Easy DP solution in C++ with detailed explanations (8ms, O(n^2) time and O(n) space)**

Well, this problem desires for the use of dynamic programming. They key to any DP problem is to come up with the state equation. In this problem, we define the state to be **the maximal size of the square that can be achieved at point (i, j)**, denoted as P[i][j]. Remember that we use **size** instead of square as the state (square = size^2).

Now let's try to come up with the formula for P[i][j].

First, it is obvious that for the topmost row (i = 0) and the leftmost column (j = 0), P[i][j] = matrix[i][j]. This is easily understood. Let's suppose that the topmost row of matrix is like [1, 0, 0, 1]. Then we can immediately know that the first and last point can be a square of size 1 while the two middle points cannot make any square, giving a size of 0. Thus, P = [1, 0, 0, 1], which is the same as matrix. The case is similar for the leftmost column. Till now, the boundary conditions of this DP problem are solved.

Let's move to the more general case for P[i][j] in which i > 0 and j > 0. First of all, let's see another simple case in which matrix[i][j] = 0. It is obvious that P[i][j] = 0 too. Why? Well, since matrix[i][j] = 0, no square will contain matrix[i][j]. According to our definition of P[i][j], P[i][j] is also 0.

Now we are almost done. The only unsolved case is matrix[i][j] = 1. Let's see an example.

Suppose matrix = [[0, 1], [1, 1]], it is obvious that P[0][0] = 0, P[0][1] = P[1][0] = 1, what about P[1][1]? Well, to give a square of size larger than 1 in P[1][1], all of its three neighbors (left, up, left-up) should be non-zero, right? In this case, the left-up neighbor P[0][0] = 0, so P[1][1] can only be 1, which means that it contains the square of itself.

如果矩阵中出现1，则检查该点的左上，左，上的情况。

选择三者中最小的+1，即为以该点为右下角的最大正方形的边长。

Now you are near the solution. In fact, P[i][j] = min(P[i - 1][j], P[i][j - 1], P[i - 1][j - 1]) + 1 in this case.

Taking all these together, we have the following state equations.

1. P[0][j] = matrix[0][j] (topmost row);
2. P[i][0] = matrix[i][0] (leftmost column);
3. For i > 0 and j > 0: if matrix[i][j] = 0, P[i][j] = 0; if matrix[i][j] = 1, P[i][j] = min(P[i - 1][j], P[i][j - 1], P[i - 1][j - 1]) + 1.

Putting them into codes, and maintain a variable maxsize to record the maximum size of the square we have seen, we have the following (unoptimized) solution.

int maximalSquare(vector<vector<char>>& matrix) {

int m = matrix.size();

if (!m) return 0;

int n = matrix[0].size();

vector<vector<int> > size(m, vector<int>(n, 0));

int maxsize = 0;

for (int j = 0; j < n; j++) {

size[0][j] = matrix[0][j] - '0';

maxsize = max(maxsize, size[0][j]);

}

for (int i = 1; i < m; i++) {

size[i][0] = matrix[i][0] - '0';

maxsize = max(maxsize, size[i][0]);

}

for (int i = 1; i < m; i++) {

for (int j = 1; j < n; j++) {

if (matrix[i][j] == '1') {

size[i][j] = min(size[i - 1][j - 1], min(size[i - 1][j], size[i][j - 1])) + 1;

maxsize = max(maxsize, size[i][j]);

}

}

}

return maxsize \* maxsize;

}

Now let's try to optimize the above solution. As can be seen, each time when we update size[i][j], we only need size[i][j - 1], size[i - 1][j - 1] (at the previous left column) and size[i - 1][j] (at the current column). So we do not need to maintain the full m\*n matrix. In fact, keeping two columns is enough. Now we have the following optimized solution.

int maximalSquare(vector<vector<char>>& matrix) {

int m = matrix.size();

if (!m) return 0;

int n = matrix[0].size();

vector<int> pre(m, 0);

vector<int> cur(m, 0);

int maxsize = 0;

for (int i = 0; i < m; i++) {

pre[i] = matrix[i][0] - '0';

maxsize = max(maxsize, pre[i]);

}

for (int j = 1; j < n; j++) {

cur[0] = matrix[0][j] - '0';

maxsize = max(maxsize, cur[0]);

for (int i = 1; i < m; i++) {

if (matrix[i][j] == '1') {

cur[i] = min(cur[i - 1], min(pre[i - 1], pre[i])) + 1;

maxsize = max(maxsize, cur[i]);

}

}

swap(pre, cur);

fill(cur.begin(), cur.end(), 0);

}

return maxsize \* maxsize;

}

Now you see the solution is finished? In fact, it can still be optimized! In fact, we need not maintain two vectors and one is enough. If you want to explore this idea, please refer to the answers provided by @stellari below. Moreover, in the code above, we distinguish between the 0-th row and other rows since the 0-th row has no row above it. In fact, we can make all the m rows the same by padding a 0 row on the top (in the following code, we pad a 0 on top of dp). Finally, we will have the following short code :) If you find it hard to understand, try to run it using your pen and paper and notice how it realizes what the two-vector solution does using only one vector.

int maximalSquare(vector<vector<char>>& matrix) {

if (matrix.empty()) return 0;

int m = matrix.size(), n = matrix[0].size();

vector<int> dp(m + 1, 0);

int maxsize = 0, pre = 0;

for (int j = 0; j < n; j++) {

for (int i = 1; i <= m; i++) {

int temp = dp[i];

if (matrix[i - 1][j] == '1') {

dp[i] = min(dp[i], min(dp[i - 1], pre)) + 1;

maxsize = max(maxsize, dp[i]);

}

else dp[i] = 0;

pre = temp;

}

}

return maxsize \* maxsize;

}

This solution, since posted, has been suggested various improvements by kind people. For a more comprehensive collection of the solutions, please visit [my technical blog](http://www.cnblogs.com/jcliBlogger/p/4548751.html).

public int maximalSquare(char[][] a) {

if(a.length == 0) return 0;

int m = a.length, n = a[0].length, result = 0;

int[][] b = new int[m+1][n+1];

for (int i = 1 ; i <= m; i++) {

for (int j = 1; j <= n; j++) {

if(a[i-1][j-1] == '1') {

b[i][j] = Math.min(Math.min(b[i][j-1] , b[i-1][j-1]), b[i-1][j]) + 1;

result = Math.max(b[i][j], result); // update result

}

}

}

return result\*result;

}

Logic :

Top, Left, and Top Left decides the size of the square. If all of them are same, then the size of square increases by 1. If they're not same, they can increase by 1 to the minimal square. If you take an example and work it out, it'll be much easier to understand when it comes to dynamic programing. :)

public int maximalSquare(char[][] a) {

if (a == null || a.length == 0 || a[0].length == 0)

return 0;

int max = 0, n = a.length, m = a[0].length;

// dp(i, j) represents the length of the square

// whose lower-right corner is located at (i, j)

// dp(i, j) = min{ dp(i-1, j-1), dp(i-1, j), dp(i, j-1) }

int[][] dp = new int[n + 1][m + 1];

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= m; j++) {

if (a[i - 1][j - 1] == '1') {

dp[i][j] = Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i][j - 1])) + 1;

max = Math.max(max, dp[i][j]);

}

}

}

// return the area

return max \* max;

}

### Decode Ways

Total Accepted: **66217** Total Submissions: **379888** Difficulty: **Medium**

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.

public class Solution {

public int numDecodings(String s) {

int n = s.length();

if (n == 0) return 0;

int[] memo = new int[n+1];

memo[n] = 1;

memo[n-1] = s.charAt(n-1) != '0' ? 1 : 0;

for (int i = n - 2; i >= 0; i--)

if (s.charAt(i) == '0') continue;

else memo[i] = (Integer.parseInt(s.substring(i,i+2))<=26) ? memo[i+1]+memo[i+2] : memo[i+1];

return memo[0];

}

}

int numDecodings(string s) {

if (!s.size() || s.front() == '0') return 0;

// r2: decode ways of s[i-2] , r1: decode ways of s[i-1]

int r1 = 1, r2 = 1;

for (int i = 1; i < s.size(); i++) {

// zero voids ways of the last because zero cannot be used separately

if (s[i] == '0') r1 = 0;

// possible two-digit letter, so new r1 is sum of both while new r2 is the old r1

if (s[i - 1] == '1' || s[i - 1] == '2' && s[i] <= '6') {

r1 = r2 + r1;

r2 = r1 - r2;

}

// one-digit letter, no new way added

else {

r2 = r1;

}

}

return r1;

}

和我的做法类似，不过我没有考虑好10的问题。下面这个处理比较清晰。

在当前数字dp[i]，如果是1到9之间，则可与其前面数字dp[i-1]组成一种解密方式。如果是0，则这种方式不成立。

在当前数字dp[i]，如果与前面一个数字，可组成10到26之间的数字，则可与其前面数字dp[i-2]组成一种解密方式。

I used a dp array of size n + 1 to save subproblem solutions. dp[0] means an empty string will have one way to decode, dp[1] means the way to decode a string of size 1. I then check one digit and two digit combination and save the results along the way. In the end, dp[n] will be the end result.

public class Solution {

public int numDecodings(String s) {

if(s == null || s.length() == 0) {

return 0;

}

int n = s.length();

int[] dp = new int[n+1];

dp[0] = 1;

dp[1] = s.charAt(0) != '0' ? 1 : 0;

for(int i = 2; i <= n; i++) {

int first = Integer.valueOf(s.substring(i-1, i));

int second = Integer.valueOf(s.substring(i-2, i));

if(first >= 1 && first <= 9) {

dp[i] += dp[i-1];

}

if(second >= 10 && second <= 26) {

dp[i] += dp[i-2];

}

}

return dp[n];

}

}

There is 1 way to decoding "", the decoding result is "".

Agree. For those abnormal input, also need to return 0, like '2002', '0123' and '3021'

### Best Time to Buy and Sell Stock

Total Accepted: **93360** Total Submissions: **260141** Difficulty: **Medium**

Say you have an array for which the *i*th element is the price of a given stock on day *i*.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

**我的做法。维护数组到当前为止的最小值，同时不断更新最大值。有点类似于Maximum Subarray**

public int maxProfit(int[] prices) {

if(prices==null || prices.length==0) return 0;

int min=prices[0];

int max=0;

for(int i=1;i<prices.length;i++)

{

max=Math.max(max,prices[i]-min);

min=Math.min(min,prices[i]);

}

return max;

}

int maxProfit(vector<int> &prices) {

int maxPro = 0;

int minPrice = INT\_MAX;

for(int i = 0; i < prices.size(); i++){

minPrice = min(minPrice, prices[i]);

maxPro = max(maxPro, prices[i] - minPrice);

}

return maxPro;

}

minPrice is the minimum price from day 0 to day i. And maxPro is the maximum profit we can get from day 0 to day i.

How to get maxPro? Just get the larger one between current maxPro and prices[i] - minPrice.

**Kadane's Algorithm - Since no one has mentioned about this so far :) (In case if interviewer twists the input)**

The logic to solve this problem is same as "max subarray problem" using Kadane's Algorithm. Since no body has mentioned this so far, I thought it's a good thing for everybody to know.

All the straight forward solution should work, but if the interviewer twists the question slightly by giving the ***difference array of prices***, Ex: for {1, 7, 4, 11}, if he gives {0, 6, -3, 7}, you might end up being confused.

Here, the logic is to calculate the difference (maxCur += prices[i] - prices[i-1]) of the original array, and find a contiguous subarray giving maximum profit. If the difference falls below 0, reset it to zero.

public int maxProfit(int[] prices) {

int maxCur = 0, maxSoFar = 0;

for(int i = 1; i < prices.length; i++) {

maxCur = Math.max(0, maxCur += prices[i] - prices[i-1]);

maxSoFar = Math.max(maxCur, maxSoFar);

}

return maxSoFar;

}

\*maxCur = current maximum value

\*maxSoFar = maximum value found so far

### Best Time to Buy and Sell Stock II（贪心）

Total Accepted: **83043** Total Submissions: **197519** Difficulty: **Medium**

Say you have an array for which the *i*th element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

**我的做法：用sum记录单次上升的最大差值。当数值回落时，将上一轮的sum累加，sum清0.最后再累加sum。（sum没啥必要）**

public int maxProfit(int[] prices) {

if(prices==null || prices.length==0) return 0;

int res=0,sum=0;//sum表示单次的最大差值

for(int i=1;i<prices.length;i++)

{

if(prices[i]>prices[i-1]) sum+=prices[i]-prices[i-1];

else

{

res+=sum;

sum=0;

}

}

res+=sum;

return res;

}

**这个写法不错。**

public class Solution {

public int maxProfit(int[] prices) {

int total = 0;

for (int i=0; i< prices.length-1; i++) {

if (prices[i+1]>prices[i]) total += prices[i+1]-prices[i];

}

return total;

}

A simple code like this. The designer of this question must thought of something too complicated.

This is like accumulating the profit I get every day. Although this alg. by pass the exact days I buy and sell, but it answers the max profit I potentially get

First we post the code here.

int maxProfit(vector<int> &prices) {

int ret = 0;

for (size\_t p = 1; p < prices.size(); ++p)

ret += max(prices[p] - prices[p - 1], 0);

return ret;

}

Second, suppose the first sequence is "a <= b <= c <= d", the profit is "d - a = (b - a) + (c - b) + (d - c)" without a doubt. And suppose another one is "a <= b >= b' <= c <= d", the profit is not difficult to be figured out as "(b - a) + (d - b')". So you just target at monotone sequences.

Hi guys!

The greedy pair-wise approach mentioned in other posts is great for this problem indeed, but if we're not allowed to buy and sell stocks within the same day it can't be applied (logically, of course; the answer will be the same). Actually, the straight-forward way of finding next local minimum and next local maximum is not much more complicated, so, just for the sake of having an alternative I share the code in Java for such case.

public int maxProfit(int[] prices) {

int profit = 0, i = 0;

while (i < prices.length) {

// find next local minimum

while (i < prices.length-1 && prices[i+1] <= prices[i]) i++;

int min = prices[i++]; // need increment to avoid infinite loop for "[1]"

// find next local maximum

while (i < prices.length-1 && prices[i+1] >= prices[i]) i++;

profit += i < prices.length ? prices[i++] - min : 0;

}

return profit;

}

Happy coding!

**Best Time to Buy and Sell Stock with Cooldown**

Total Accepted: **12603** Total Submissions: **34457** Difficulty: **Medium**

Say you have an array for which the *i*th element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

* You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
* After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

**Example:**

prices = [1, 2, 3, 0, 2]

maxProfit = 3

transactions = [buy, sell, cooldown, buy, sell]

The series of problems are typical dp. The key for dp is to find the variables to represent the states and deduce the transition function.

Of course one may come up with a O(1) space solution directly, but I think it is better to be generous when you think and be greedy when you implement.

The natural states for this problem is the **3 possible transactions : buy, sell, rest.** Here rest means no transaction on that day (aka cooldown).

Then the transaction sequences can end with any of these three states.

For each of them we make an array, buy[n], sell[n] and rest[n].

buy[i] means before day i what is the maxProfit for any sequence end with buy.

sell[i] means before day i what is the maxProfit for any sequence end with sell.

rest[i] means before day i what is the maxProfit for any sequence end with rest.

Then we want to deduce the transition functions for buy sell and rest. By definition we have:

buy[i] = max(rest[i-1]-price, buy[i-1])

sell[i] = max(buy[i-1]+price, sell[i-1])

rest[i] = max(sell[i-1], buy[i-1], rest[i-1])

Where price is the price of day i. All of these are very straightforward. They simply represents :

(1) We have to `rest` before we `buy` and

(2) we have to `buy` before we `sell`

One tricky point is how do you make sure you sell before you buy, since from the equations it seems that [buy, rest, buy] is entirely possible.

Well, the answer lies within the fact that buy[i] <= rest[i] which means rest[i] = max(sell[i-1], rest[i-1]). That made sure [buy, rest, buy] is never occurred.

A further observation is that and rest[i] <= sell[i] is also true therefore

rest[i] = sell[i-1]

Substitute this in to buy[i] we now have 2 functions instead of 3:

buy[i] = max(sell[i-2]-price, buy[i-1])

sell[i] = max(buy[i-1]+price, sell[i-1])

This is better than 3, but

**we can do even better**

Since states of day i relies only on i-1 and i-2 we can reduce the O(n) space to O(1). And here we are at our final solution:

**Java**

public int maxProfit(int[] prices) {

int sell = 0, prev\_sell = 0, buy = Integer.MIN\_VALUE, prev\_buy;

for (int price : prices) {

prev\_buy = buy;

buy = Math.max(prev\_sell - price, prev\_buy);

prev\_sell = sell;

sell = Math.max(prev\_buy + price, prev\_sell);

}

return sell;

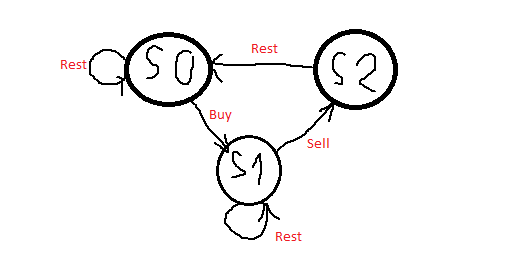
}

Hi,

I just come across this problem, and it's very frustating since I'm bad at DP.

So I just draw all the actions that can be done.

Here is the drawing (Feel like an elementary ...)



There are three states, according to the action that you can take.

Hence, from there, you can now the profit at a state at time i as:

s0[i] = max(s0[i - 1], s2[i - 1]); // Stay at s0, or rest from s2

s1[i] = max(s1[i - 1], s0[i - 1] - prices[i]); // Stay at s1, or buy from s0

s2[i] = s1[i - 1] + prices[i]; // Only one way from s1

Then, you just find the maximum of s0[n] and s2[n], since they will be the maximum profit we need (No one can buy stock and left with more profit that sell right :) )

Define base case:

s0[0] = 0; // At the start, you don't have any stock if you just rest

s1[0] = -prices[0]; // After buy, you should have -prices[0] profit. Be positive!

s2[0] = INT\_MIN; // Lower base case

Here is the code :D

class Solution {

public:

int maxProfit(vector<int>& prices){

if (prices.size() <= 1) return 0;

vector<int> s0(prices.size(), 0);

vector<int> s1(prices.size(), 0);

vector<int> s2(prices.size(), 0);

s1[0] = -prices[0];

s0[0] = 0;

s2[0] = INT\_MIN;

for (int i = 1; i < prices.size(); i++) {

s0[i] = max(s0[i - 1], s2[i - 1]);

s1[i] = max(s1[i - 1], s0[i - 1] - prices[i]);

s2[i] = s1[i - 1] + prices[i];

}

return max(s0[prices.size() - 1], s2[prices.size() - 1]);

}

};

**下面这个比较好理解。确定每天可以有哪些状态结束：卖，买，休息。**

**关键在于卖买之间的关系转移**

Here I share my no brainer weapon when it comes to this kind of problems.

**1. Define States**

To represent the decision at index i:

* buy[i]: Max profit till index i. The series of transaction is ending with a **buy**.
* sell[i]: Max profit till index i. The series of transaction is ending with a **sell**.

To clarify:

* Till index i, the **buy / sell** action must happen and must be the **last action**. It may not happen at index i. It may happen at i - 1, i - 2, ... 0.
* In the end n - 1, return sell[n - 1]. Apparently we cannot finally end up with a buy. In that case, we would rather take a rest at n - 1.
* For special case no transaction at all, classify it as sell[i], so that in the end, we can still return sell[n - 1]. Thanks @alex153 @kennethliaoke @anshu2.

**2. Define Recursion**

* buy[i]: To make a decision whether to buy at i, we either take a rest, by just using the old decision at i - 1, or sell at/before i - 2, then buy at i, We cannot sell at i - 1, then buy at i, because of **cooldown**.
* sell[i]: To make a decision whether to sell at i, we either take a rest, by just using the old decision at i - 1, or buy at/before i - 1, then sell at i.

So we get the following formula:

buy[i] = Math.max(buy[i - 1], sell[i - 2] - prices[i]);

sell[i] = Math.max(sell[i - 1], buy[i - 1] + prices[i]);

**3. Optimize to O(1) Space**

DP solution only depending on i - 1 and i - 2 can be optimized using O(1) space.

* Let b2, b1, b0 represent buy[i - 2], buy[i - 1], buy[i]
* Let s2, s1, s0 represent sell[i - 2], sell[i - 1], sell[i]

Then arrays turn into Fibonacci like recursion:

b0 = Math.max(b1, s2 - prices[i]);

s0 = Math.max(s1, b1 + prices[i]);

**4. Write Code in 5 Minutes**

First we define the initial states at i = 0:

* We can buy. The max profit at i = 0 ending with a **buy** is -prices[0].
* We cannot sell. The max profit at i = 0 ending with a **sell** is 0.

Here is my solution. Hope it helps!

public int maxProfit(int[] prices) {

if(prices == null || prices.length <= 1) return 0;

int b0 = -prices[0], b1 = b0;

int s0 = 0, s1 = 0, s2 = 0;

for(int i = 1; i < prices.length; i++) {

b0 = Math.max(b1, s2 - prices[i]);

s0 = Math.max(s1, b1 + prices[i]);

b1 = b0; s2 = s1; s1 = s0;

}

return s0;

}

Define:

profit1[i] = max profit on day i if I sell

profit2[i] = max profit on day i if I do nothing

How will those profits on day i+1 relate to profits on day i ?

1. profit1[i+1] means I must sell on day i+1, and there are 2 cases:

a. If I just sold on day i, then I have to buy again on day i and sell on day i+1

b. If I did nothing on day i, then I have to buy today and sell today

Taking both cases into account, profit1[i+1] = max(profit1[i]+prices[i+1]-prices[i], profit2[i])

2. profit2[i+1] means I do nothing on day i+1, so it will be max(profit1[i], profit2[i])

And the code:

public int maxProfit(int[] prices) {

int profit1=0, profit2=0;

for(int i=1; i<prices.length; i++){

int copy=profit1;

profit1=Math.max(profit1+prices[i]-prices[i-1], profit2);

profit2=Math.max(copy, profit2);

}

return Math.max(profit1, profit2);

}

### #1137 : Recruitment

时间限制:10000ms

单点时限:1000ms

内存限制:256MB

|  |
| --- |
|  |

A company plans to recruit some new employees. There are N candidates (indexed from 1 to N) have taken the recruitment examination. After the examination, the well-estimated ability value as well as the expected salary per year of each candidate is collected by the Human Resource Department.

Now the company need to choose their new employees according to these data. To maximize the company's benefits, some principles should be followed:

1. There should be exactly X males and Y females.

2. The sum of salaries per year of the chosen candidates should not exceed the given budget B.

3. The sum of ability values of the chosen candidates should be maximum, without breaking the previous principles. Based on this, the sum of the salary per year should be minimum.

4. If there are multiple answers, choose the lexicographically smallest one. In other words, you should minimize the smallest index of the chosen candidates; If there are still multiple answers, then minimize the second smallest index; If still multiple answers, then minimize the third smallest one; ...

Your task is to help the company choose the new employees from those candidates.

输入

The first line contains four integers N, X, Y, and B, separated by a single space. The meanings of all these variables are showed in the description above. 1 <= N <= 100, 0 <= X <= N, 0 <= Y <= N, 1 <= X + Y <= N, 1 <= B <= 1000.

Then follows N lines. The i-th line contains the data of the i-th candidate: a character G, and two integers V and S, separated by a single space. G indicates the gender (either "M" for male, or "F" for female), V is the well-estimated ability value and S is the expected salary per year of this candidate. 1 <= V <= 10000, 0 <= S <= 10.

We assure that there is always at least one possible answer.

输出

On the first line, output the sum of ability values and the sum of salaries per year of the chosen candidates, separated by a single space.

On the second line, output the indexes of the chosen candidates in ascending order, separated by a single space.

样例输入

4 1 1 10

F 2 3

M 7 6

M 3 2

F 9 9

样例输出

9 9

1 2

大意是：

有N个应聘者，分别知道他们的价值V，期望薪水S，以及性别。招聘要求是：男X名，女Y名，总预算B（即期望薪水之和不能超过B），如何使总价值最大？

思路

很容易想到，这题是01背包问题的一个变体。基本思路是分别对男、女集合做01背包。几个难点：

* 男、女分开的预算不知道，只有一个总的预算，怎么办？——把B作为总预算，然而DP的结果其实是包含0~B各个情况的，所以我们可以通过一个B\*B的循环，枚举（0, 0）,（0, 1）……（B, B）所有的预算分配方式，找出价值最大的。
* 男女人数必须为X，Y——这也看作一个限制条件：每个人的代价是1。所以这里的动归其实是三维的。
* 输出的不仅是总价值还要具体方案——这个很复杂，必须回溯。回溯意味着要保存下DP的所有过程，要么直接开三维数组（不压缩），要么用二维数组、但是额外用一个布尔型三维数组保存每次的处理方式（选择该人还是不选）

**DP初始子问题的设定非常重要！！**初始子问题的行（设为0）或不行（设为一个负数，例如-1000000000）将决定整个DP过程，这对所有背包问题都适用。本题中，我们设定：男女人数必须是恰好相等的，所以人数k > 0的初始子问题被初始化为-1000000000；总预算不一定要恰好，因为我们后期反正还会选出总预算最小的。

代码

#include <vector>

#include <algorithm>

#include <iostream>

#include <climits>

using namespace std;

int N, B;

char G;

struct FM {

int n;

int no[100];

int value[100];

int cost[100];

int dp[1024][101];

bool select[1024][1024][101];

int limit;

} mm, gg;

void zero\_one\_pack(struct FM& fm) {

for (int i = 0; i <= B; i++)

for (int j = 0; j <= fm.limit; j++)

fm.dp[i][j] = -1000000000;

fm.dp[0][0] = 0;

for (int i = fm.n - 1; i >= 0; i--) {

for (int j = B; j >= fm.cost[i]; j--) {

for (int k = fm.limit; k >= 1; k--) {

int value = fm.dp[j - fm.cost[i]][k - 1] + fm.value[i];

if (value >= fm.dp[j][k]) {

fm.dp[j][k] = value;

fm.select[i][j][k] = true;

}

}

}

}

}

vector<int> seq;

bool get\_result(struct FM &fm, int s, int k, int i) {

if (s == 0 && k == 0) return true;

for (; i < fm.n; i++) {

if (fm.select[i][s][k]) {

seq.push\_back(fm.no[i]);

if (get\_result(fm, s - fm.cost[i], k - 1, i + 1)) return true;

seq.pop\_back();

}

}

}

vector<int> merge(vector<int> &a, vector<int> &b) {

vector<int> result;

int i = 0, j = 0;

while (i < a.size() && j < b.size()) {

if (a[i] < b[j]) {

result.push\_back(a[i++]);

}

else {

result.push\_back(b[j++]);

}

}

while (i < a.size()) result.push\_back(a[i++]);

while (j < b.size()) result.push\_back(b[j++]);

return result;

}

int main() {

scanf("%d %d %d %d", &N, &gg.limit, &mm.limit, &B);

for (int i = 1; i <= N; i++) {

scanf("%c", &G);

if (G == 'F') {

int j = mm.n++;

scanf("%d %d", &mm.value[j], &mm.cost[j]);

mm.no[j] = i;

}

else if (G == 'M') {

int j = gg.n++;

scanf("%d %d", &gg.value[j], &gg.cost[j]);

gg.no[j] = i;

}

else {

i--;

continue;

}

}

zero\_one\_pack(mm);

zero\_one\_pack(gg);

int max\_value = -1, min\_cost = INT\_MAX;

vector<pair<int, int>> ijs;

for (int i = 0; i <= B; i++) {

for (int j = 0; j <= B - i; j++) {

int value = gg.dp[i][gg.limit] + mm.dp[j][mm.limit];

if (value >= max\_value) {

if (value > max\_value || i + j < min\_cost) {

max\_value = value;

min\_cost = i + j;

ijs.clear();

ijs.push\_back(pair<int, int>(i, j));

}

else if (i + j == min\_cost) {

ijs.push\_back(pair<int, int>(i, j));

}

}

}

}

printf("%d %d\n", max\_value, min\_cost);

vector<int> candidate(1, INT\_MAX);

for (auto &ij : ijs) {

seq.clear();

get\_result(gg, ij.first, gg.limit, 0);

auto rgg = seq;

seq.clear();

get\_result(mm, ij.second, mm.limit, 0);

auto rmm = seq;

auto merged = merge(rgg, rmm);

if (merged < candidate) candidate = merged;

}

for (int i = 0; i < candidate.size(); i++) {

printf("%d%c", candidate[i], i == candidate.size() - 1 ? '\n' : ' ');

}

}

### House Robber

Total Accepted: **63564** Total Submissions: **187097** Difficulty: **Easy**

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**.

**我的做法（只需要考虑该房子是否需要抢，根据前面状态可以推得）🡪空间可以转换成O（1）**

**public int rob(int[] nums) {**

**if(nums==null || nums.length==0) return 0;**

**int[] dp=new int[nums.length];**

**dp[0]=nums[0];**

**if(nums.length>1) dp[1]=Math.max(nums[0],nums[1]);**

**//System.out.println(dp[1]);**

**for(int i=2;i<nums.length;i++)**

**{**

**dp[i]=Math.max(dp[i-2]+nums[i],dp[i-1]);**

**//System.out.println(dp[i]);**

**}**

**return dp[nums.length-1];**

**}**

#define max(a, b) ((a)>(b)?(a):(b))

int rob(int num[], int n) {

int a = 0;

int b = 0;

for (int i=0; i<n; i++)

{

if (i%2==0)

{

a = max(a+num[i], b);

}

else

{

b = max(a, b+num[i]);

}

}

return max(a, b);

}

public int rob(int[] num) {

int[][] dp = new int[num.length + 1][2];

for (int i = 1; i <= num.length; i++) {

dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1]);

dp[i][1] = num[i - 1] + dp[i - 1][0];

}

return Math.max(dp[num.length][0], dp[num.length][1]);

}

dp[i][1] means we rob the current house and dp[i][0] means we don't,

这样的写法好

so it is easy to convert this to O(1) space

public int rob(int[] num) {

int prevNo = 0;

int prevYes = 0;

for (int n : num) {

int temp = prevNo;

prevNo = Math.max(prevNo, prevYes);

prevYes = n + temp;

}

return Math.max(prevNo, prevYes);

}

**JAVA DP Solution, O(n) runtime and O(1) space, with inline comment**

public int rob(int[] num) {

int rob = 0; //max monney can get if rob current house

int notrob = 0; //max money can get if not rob current house

for(int i=0; i<num.length; i++) {

int currob = notrob + num[i]; //if rob current value, previous house must not be robbed

notrob = Math.max(notrob, rob); //if not rob ith house, take the max value of robbed (i-1)th house and not rob (i-1)th house

rob = currob;

}

return Math.max(rob, notrob);

}

### House Robber II

Total Accepted: **25908** Total Submissions: **85272** Difficulty: **Medium**

**Note:** This is an extension of [House Robber](https://leetcode.com/problems/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**.

**我的做法是运用两次搜索。**

**第一种情况，第一个房子抢，则第二个和最后一个房子不能抢，则剩下的房子搜索。**

**第二种情况，第一个房子不抢，则剩下的所有房子搜索（按前一题的方法）**

**其实即，不抢第一个房子和不抢第二个房子两种情况**

**public int rob(int[] nums) {**

**if(nums==null || nums.length==0) return 0;**

**//int[] dp=new int[nums.length];**

**//if(nums.length>1) dp[1]=nums[0];**

**int prevYes=0,prevNo=0;**

**for(int i=2;i<nums.length-1;i++)**

**{**

**int temp=prevNo;**

**prevNo=Math.max(prevNo,prevYes);**

**prevYes=temp+nums[i];**

**}**

**int res=nums[0]+Math.max(prevNo, prevYes);**

**prevYes=0;**

**prevNo=0;**

**for(int i=1;i<nums.length;i++)**

**{**

**int temp=prevNo;**

**prevNo=Math.max(prevNo,prevYes);**

**prevYes=temp+nums[i];**

**}**

**res=Math.max(res,Math.max(prevNo, prevYes));**

**return res;**

**}**

Since this question is a follow-up to House Robber, we can assume we already have a way to solve the simpler question, i.e. given a 1 row of house, we know how to rob them. So we already have such a helper function. We modify it a bit to rob a given range of houses.

private int rob(int[] num, int lo, int hi) {

int include = 0, exclude = 0;

for (int j = lo; j <= hi; j++) {

int i = include, e = exclude;

include = e + num[j];

exclude = Math.max(e, i);

}

return Math.max(include, exclude);

}

Now the question is how to rob a circular row of houses. It is a bit complicated to solve like the simpler question. It is because in the simpler question whether to rob *num[lo]* is entirely our choice. But, it is now constrained by whether *num[hi]* is robbed.

However, since we already have a nice solution to the simpler problem. We do not want to throw it away. Then, it becomes how can we reduce this problem to the simpler one. Actually, extending from the logic that if house i is not robbed, then you are free to choose whether to rob house i + 1, you can break the circle by assuming a house is not robbed.

For example, 1 -> 2 -> 3 -> 1 becomes 2 -> 3 if 1 is not robbed.

Since every house is either robbed or not robbed and at least half of the houses are not robbed, the solution is simply the larger of two cases with consecutive houses, i.e. house i not robbed, break the circle, solve it, or house i + 1 not robbed. Hence, the following solution. I chose i = n and i + 1 = 0 for simpler coding. But, you can choose whichever two consecutive ones.

public int rob(int[] nums) {

if (nums.length == 1) return nums[0];

return Math.max(rob(nums, 0, nums.length - 2), rob(nums, 1, nums.length - 1));

}

This problem is a little tricky at first glance. However, if you have finished the **House Robber** problem, this problem can simply be **decomposed into two House Robber problems**. Suppose there are n houses, since house 0 and n - 1 are now neighbors, we cannot rob them together and thus the solution is now the maximum of

1. Rob houses 0 to n - 2;
2. Rob houses 1 to n - 1.

The code is as follows. Some edge cases (n < 2) are handled explicitly.

class Solution {

public:

int rob(vector<int>& nums) {

int n = nums.size();

if (n < 2) return n ? nums[0] : 0;

return max(robber(nums, 0, n - 2), robber(nums, 1, n - 1));

}

private:

int robber(vector<int>& nums, int l, int r) {

int pre = 0, cur = 0;

for (int i = l; i <= r; i++) {

int temp = max(pre + nums[i], cur);

pre = cur;

cur = temp;

}

return cur;

}

};

**这个写的比较清晰**

**Twice pass:**

1. **not rob nums[n-1]**
2. **not rob nums[0]**

and the other is same as [House Robber](https://leetcode.com/problems/house-robber/).

int rob(vector<int>& nums)

{

if(nums.size() == 0)

return 0;

if(nums.size() == 1)

return nums[0];

int pre1 = 0, cur1 = 0;

for(int i = 0; i < nums.size() - 1; ++ i)

{

int temp = pre1;

pre1 = cur1;

cur1 = max(temp + nums[i], pre1);

}

int pre2 = 0, cur2 = 0;

for(int i = 1; i < nums.size(); ++ i)

{

int temp = pre2;

pre2 = cur2;

cur2 = max(temp + nums[i], pre2);

}

return max(cur1, cur2);

}

### Unique Paths

Total Accepted: **83829** Total Submissions: **232784** Difficulty: **Medium**

A robot is located at the top-left corner of a *m* x *n* grid (marked 'Start' in the diagram below).

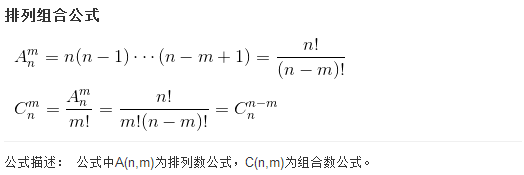
The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?



Above is a 3 x 7 grid. How many possible unique paths are there?

**Note:** *m* and *n* will be at most 100.



**我的做法，注意边界处理时，不用考虑大于边界长（否则0的情况会有问题）**

**public int uniquePaths(int m, int n) {**

**int[][] dp=new int[m][n];**

**for(int i=0;i<m;i++)**

**Arrays.fill(dp[i],Integer.MAX\_VALUE);**

**dp[0][0]=1;**

**return uniquePaths(dp,m-1,n-1);**

**}**

**public int uniquePaths(int[][]dp,int m, int n) {**

**if(m<0 || n<0) return 0;**

**if(dp[m][n]!=Integer.MAX\_VALUE) return dp[m][n];**

**dp[m][n]=uniquePaths(dp,m-1,n)+uniquePaths(dp,m,n-1);**

**//System.out.println(dp[m][n]);**

**return dp[m][n];**

**}**

**My AC solution using formula**

Binomial coefficient:

class Solution {

public:

int uniquePaths(int m, int n) {

int N = n + m - 2;// how much steps we need to do

int k = m - 1; // number of steps that need to go down

double res = 1;

// here we calculate the total possible path number

// Combination(N, k) = n! / (k!(n - k)!)

// reduce the numerator and denominator and get

// C = ( (n - k + 1) \* (n - k + 2) \* ... \* n ) / k!

for (int i = 1; i <= k; i++)

res = res \* (N - k + i) / i;

return (int)res;

}

};

First of all you should understand that we need to do n + m - 2 movements : m - 1 down, n - 1 right, because we start from cell (1, 1).

Secondly, the path it is the sequence of movements( go down / go right), therefore we can say that two paths are different when there is i-th (1 .. m + n - 2) movement in path1 differ i-th movement in path2.

So, how we can build paths. Let's choose (n - 1) movements(number of steps to the right) from (m + n - 2), and rest (m - 1) is (number of steps down).

I think now it is obvious that count of different paths are all combinations (n - 1) movements from (m + n-2).

This is a fundamental DP problem. First of all, let's make some observations.

Since the robot can only move right and down, when it arrives at a point, there are only two possibilities:

1. It arrives at that point from above (moving down to that point);
2. It arrives at that point from left (moving right to that point).

Thus, we have the following state equations: suppose the number of paths to arrive at a point (i, j) is denoted as P[i][j], it is easily concluded that P[i][j] = P[i - 1][j] + P[i][j - 1].

The boundary conditions of the above equation occur at the leftmost column (P[i][j - 1] does not exist) and the uppermost row (P[i - 1][j] does not exist). These conditions can be handled by initialization (pre-processing) --- initialize P[0][j] = 1, P[i][0] = 1 for all valid i, j. Note the initial value is 1 instead of 0!

Now we can write down the following (unoptimized) code.

**这个写法不错**

class Solution {

int uniquePaths(int m, int n) {

vector<vector<int> > path(m, vector<int> (n, 1));

for (int i = 1; i < m; i++)

for (int j = 1; j < n; j++)

path[i][j] = path[i - 1][j] + path[i][j - 1];

return path[m - 1][n - 1];

}

};

As can be seen, the above solution runs in O(n^2) time and costs O(m\*n) space. However, you may have observed that each time when we update path[i][j], we only need path[i - 1][j] (at the same column) and path[i][j - 1] (at the left column). So it is enough to maintain two columns (the current column and the left column) instead of maintaining the full m\*n matrix. Now the code can be optimized to have O(min(m, n)) space complexity.

class Solution {

int uniquePaths(int m, int n) {

if (m > n) return uniquePaths(n, m);

vector<int> pre(m, 1);

vector<int> cur(m, 1);

for (int j = 1; j < n; j++) {

for (int i = 1; i < m; i++)

cur[i] = cur[i - 1] + pre[i];

swap(pre, cur);

}

return pre[m - 1];

}

};

Further inspecting the above code, we find that keeping two columns is used to recover pre[i], which is just cur[i] before its update. So there is even no need to use two vectors and one is just enough. Now the space is further saved and the code also gets much shorter.

class Solution {

int uniquePaths(int m, int n) {

if (m > n) return uniquePaths(n, m);

vector<int> cur(m, 1);

for (int j = 1; j < n; j++)

for (int i = 1; i < m; i++)

cur[i] += cur[i - 1];

return cur[m - 1];

}

};

Well, till now, I guess you may even want to optimize it to O(1) space complexity since the above code seems to rely on only cur[i] and cur[i - 1]. You may think that 2 variables is enough? Well, it is not. Since the whole cur needs to be updated for n - 1 times, it means that all of its values need to be saved for next update and so two variables is not enough.

public class Solution {

public int uniquePaths(int m, int n) {

Integer[][] map = new Integer[m][n];

for(int i = 0; i<m;i++){

map[i][0] = 1;

}

for(int j= 0;j<n;j++){

map[0][j]=1;

}

for(int i = 1;i<m;i++){

for(int j = 1;j<n;j++){

map[i][j] = map[i-1][j]+map[i][j-1];

}

}

return map[m-1][n-1];

}

}

The assumptions are

1. When (n==0||m==0) the function always returns 1 since the robot can't go left or up.
2. For all other cells. The result = uniquePaths(m-1,n)+uniquePaths(m,n-1)

Therefore I populated the edges with 1 first and use DP to get the full 2-D array.

Please give any suggestions on improving the code.

### Unique Paths II

Total Accepted: **63248** Total Submissions: **216307** Difficulty: **Medium**

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 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.

**我的做法在前一题的基础上，加上对障碍的判断**

**public int uniquePathsWithObstacles(int[][] obstacleGrid) {**

**if(obstacleGrid==null ||obstacleGrid.length==0) return 0;**

**int m=obstacleGrid.length;**

**int n=obstacleGrid[0].length;**

**int[][] dp=new int[m][n];**

**for(int i=0;i<m;i++)**

**Arrays.fill(dp[i],Integer.MAX\_VALUE);**

**if(obstacleGrid[0][0]=='1') return 0;**

**dp[0][0]=1;**

**return uniquePaths(obstacleGrid,dp,m-1,n-1);**

**}**

**public int uniquePaths(int[][] obstacleGrid, int[][]dp,int m, int n) {**

**if(m<0 || n<0 ||obstacleGrid[m][n]==1) return 0;**

**if(dp[m][n]!=Integer.MAX\_VALUE) return dp[m][n];**

**dp[m][n]=uniquePaths(obstacleGrid,dp,m-1,n)+uniquePaths(obstacleGrid,dp,m,n-1);**

**System.out.println("m:"+m+" "+"n:"+n+" "+dp[m][n]);**

**return dp[m][n];**

**}**

**my C++ Dp solution , very simple!**

just use dp to find the answer , if there is a obstacle at (i,j), then dp[i][j] = 0. time is O(n*m) , space is O(n*m) . here is my code:

**这个写法不错，注意初值**

class Solution {

public:

int uniquePathsWithObstacles(vector<vector<int> > &obstacleGrid) {

int m = obstacleGrid.size() , n = obstacleGrid[0].size();

vector<vector<int>> dp(m+1,vector<int>(n+1,0));

dp[0][1] = 1;//为求dp[1,1]作准备

for(int i = 1 ; i <= m ; ++i)

for(int j = 1 ; j <= n ; ++j)

if(!obstacleGrid[i-1][j-1])

dp[i][j] = dp[i-1][j]+dp[i][j-1];

return dp[m][n];

}

};

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];

}

Well, this problem is similar to **Unique Paths**. The introduction of obstacles only changes the boundary conditions and make some points unreachable (simply set to 0).

Denote the number of paths to arrive at point (i, j) to be P[i][j], the state equation is P[i][j] = P[i - 1][j] + P[i][j - 1] if obstacleGrid[i][j] != 1 and 0 otherwise.

Now let's finish the boundary conditions. In the **Unique Paths** problem, we initialize P[0][j] = 1, P[i][0] = 1 for all valid i, j. Now, due to obstacles, some boundary points are no longer reachable and need to be initialized to 0. For example, if obstacleGrid is like [0, 0, 1, 0, 0], then the last three points are not reachable and need to be initialized to be 0. The result is [1, 1, 0, 0, 0].

Now we can write down the following (unoptimized) code. Note that we pad the obstacleGrid by 1 and initialize dp[0][1] = 1 to unify the boundary cases.

class Solution {

public:

int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {

int m = obstacleGrid.size(), n = obstacleGrid[0].size();

vector<vector<int> > dp(m + 1, vector<int> (n + 1, 0));

dp[0][1] = 1;

for (int i = 1; i <= m; i++)

for (int j = 1; j <= n; j++)

if (!obstacleGrid[i - 1][j - 1])

dp[i][j] = dp[i - 1][j] + dp[i][j - 1];

return dp[m][n];

}

};

Well, the code is accepted but it has some obvious redundancy. There are two major concerns:

1. Each time when we update path[i][j], we only need path[i - 1][j] (at the same column) and path[i][j - 1] (at the left column), so it is unnecessary to maintain the full m\*n matrix. Maintaining two columns is enough.
2. There are some cases that the loop can be terminated earlier. Suppose obstacleGrid = [[0, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0]], then we can see that it is impossible to reach the bottom-right corner after updating the second column since the number of paths to reach each element in the second column is 0.

Taken these into considerations, we write down the following optimized code.

class Solution {

public:

int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {

int m = obstacleGrid.size();

int n = obstacleGrid[0].size();

vector<int> pre(m, 0);

vector<int> cur(m, 0);

for (int i = 0; i < m; i++) {

if (!obstacleGrid[i][0])

pre[i] = 1;

else break;

}

for (int j = 1; j < n; j++) {

bool flag = false;

if (!obstacleGrid[0][j]) {

cur[0] = pre[0];

if (cur[0]) flag = true;

}

else cur[0] = 0;

for (int i = 1; i < m; i++) {

if (!obstacleGrid[i][j]) {

cur[i] = cur[i - 1] + pre[i];

if (cur[i]) flag = true;

}

else cur[i] = 0;

}

if (!flag) return 0;

swap(pre, cur);

}

return pre[m - 1];

}

};

Further inspecting the above code, keeping two vectors only serve for the purpose of recovering pre[i], which is simply cur[i] before its update. So we can use only one vector and the space is further optimized.

class Solution {

public:

int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {

int m = obstacleGrid.size();

int n = obstacleGrid[0].size();

vector<int> cur(m, 0);

for (int i = 0; i < m; i++) {

if (!obstacleGrid[i][0])

cur[i] = 1;

else break;

}

for (int j = 1; j < n; j++) {

bool flag = false;

if (obstacleGrid[0][j])

cur[0] = 0;

else flag = true;

for (int i = 1; i < m; i++) {

if (!obstacleGrid[i][j]) {

cur[i] += cur[i - 1];

if (cur[i]) flag = true;

}

else cur[i] = 0;

}

if (!flag) return 0;

}

return cur[m - 1];

}

};

public class Solution {

public int uniquePathsWithObstacles(int[][] obstacleGrid) {

int m = obstacleGrid.length;

int n = obstacleGrid[0].length;

for(int i = 0; i < m; i++) {

for(int j = 0; j < n; j++) {

if(obstacleGrid[i][j] == 1) {

obstacleGrid[i][j] = 0;

}

else {

if(i == 0 && j == 0) obstacleGrid[i][j] = 1;

else if(i == 0 && j > 0) obstacleGrid[i][j] = obstacleGrid[i][j-1];

else if(i > 0 && j == 0) obstacleGrid[i][j] = obstacleGrid[i-1][j];

else obstacleGrid[i][j] = obstacleGrid[i-1][j] + obstacleGrid[i][j-1];

}

}

}

return obstacleGrid[m-1][n-1];

}

}

### Minimum Path Sum

Total Accepted: **67895** Total Submissions: **195974** Difficulty: **Medium**

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.

**我的做法，参考前面两题的dp**

**注意初值的设置**

**public int minPathSum(int[][] grid) {**

**int m=grid.length;**

**int n=grid[0].length;**

**int[][] dp=new int[m+1][n+1];**

**for(int i=0;i<=m;i++) Arrays.fill(dp[i],Integer.MAX\_VALUE);**

**dp[0][1]=0;**

**//int max=grid[0][0];**

**for(int i=1;i<=m;i++)**

**{**

**for(int j=1;j<=n;j++)**

**{**

**dp[i][j]=Math.min(dp[i-1][j],dp[i][j-1])+grid[i-1][j-1];**

**//System.out.println("i:"+i+" j:"+j+" "+dp[i][j]);**

**}**

**}**

**return dp[m][n];**

**}**

This is a typical DP problem. Suppose the minimum path sum of arriving at point (i, j) is S[i][j], then the state equation is S[i][j] = min(S[i - 1][j], S[i][j - 1]) + grid[i][j].

Well, some boundary conditions need to be handled. The boundary conditions happen on the topmost row (S[i - 1][j] does not exist) and the leftmost column (S[i][j - 1] does not exist). Suppose grid is like [1, 1, 1, 1], then the minimum sum to arrive at each point is simply an accumulation of previous points and the result is [1, 2, 3, 4].

Now we can write down the following (unoptimized) code.

class Solution {

public:

int minPathSum(vector<vector<int>>& grid) {

int m = grid.size();

int n = grid[0].size();

vector<vector<int> > sum(m, vector<int>(n, grid[0][0]));

for (int i = 1; i < m; i++)

sum[i][0] = sum[i - 1][0] + grid[i][0];

for (int j = 1; j < n; j++)

sum[0][j] = sum[0][j - 1] + grid[0][j];

for (int i = 1; i < m; i++)

for (int j = 1; j < n; j++)

sum[i][j] = min(sum[i - 1][j], sum[i][j - 1]) + grid[i][j];

return sum[m - 1][n - 1];

}

};

As can be seen, each time when we update sum[i][j], we only need sum[i - 1][j] (at the current column) and sum[i][j - 1] (at the left column). So we need not maintain the full m\*n matrix. Maintaining two columns is enough and now we have the following code.

class Solution {

public:

int minPathSum(vector<vector<int>>& grid) {

int m = grid.size();

int n = grid[0].size();

vector<int> pre(m, grid[0][0]);

vector<int> cur(m, 0);

for (int i = 1; i < m; i++)

pre[i] = pre[i - 1] + grid[i][0];

for (int j = 1; j < n; j++) {

cur[0] = pre[0] + grid[0][j];

for (int i = 1; i < m; i++)

cur[i] = min(cur[i - 1], pre[i]) + grid[i][j];

swap(pre, cur);

}

return pre[m - 1];

}

};

Further inspecting the above code, it can be seen that maintaining pre is for recovering pre[i], which is simply cur[i] before its update. So it is enough to use only one vector. Now the space is further optimized and the code also gets shorter.

class Solution {

public:

int minPathSum(vector<vector<int>>& grid) {

int m = grid.size();

int n = grid[0].size();

vector<int> cur(m, grid[0][0]);

for (int i = 1; i < m; i++)

cur[i] = cur[i - 1] + grid[i][0];

for (int j = 1; j < n; j++) {

cur[0] += grid[0][j];

for (int i = 1; i < m; i++)

cur[i] = min(cur[i - 1], cur[i]) + grid[i][j];

}

return cur[m - 1];

}

};

**my java solution using DP and no extra space**

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];

}

You can only reach a cell by going from its left or top neighbor.

class Solution {

public:

int minPathSum(vector<vector<int> > &grid) {

if(!grid.size())return 0;

const int rows=grid.size(),cols=grid[0].size();

// r[i] == min path sum to previous row's column i.

vector<int> r(cols,0);

int i,j;

r[0]=grid[0][0];

for(j=1;j<cols;j++){

r[j]=grid[0][j]+r[j-1];

}

for(i=1;i<rows;i++){

r[0]+=grid[i][0];

for(j=1;j<cols;j++){

r[j]=min(r[j-1],r[j])+grid[i][j];

}

}

return r[cols-1];

}

};

**Ugly Number II**

Total Accepted: **27277** Total Submissions: **99054** Difficulty: **Medium**

Write a program to find the n-th ugly number.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10 ugly numbers.

Note that 1 is typically treated as an ugly number.

**Hint:**

1. The naive approach is to call isUgly for every number until you reach the nth one. Most numbers are *not* ugly. Try to focus your effort on generating only the ugly ones.
2. An ugly number must be multiplied by either 2, 3, or 5 from a smaller ugly number.
3. The key is how to maintain the order of the ugly numbers. Try a similar approach of merging from three sorted lists: L1, L2, and L3.
4. Assume you have Uk, the kth ugly number. Then Uk+1 must be Min(L1 \* 2, L2 \* 3, L3 \* 5).

**我的做法，同书上，建立三个队列。**

We have an array *k* of first n ugly number. We only know, at the beginning, the first one, which is 1. Then

k[1] = min( k[0]x2, k[0]x3, k[0]x5). The answer is k[0]x2. So we move 2's pointer to 1. Then we test:

k[2] = min( k[1]x2, k[0]x3, k[0]x5). And so on. Be careful about the cases such as 6, in which we need to forward both pointers of 2 and 3.

x here is multiplication.

class Solution {

public:

int nthUglyNumber(int n) {

if(n <= 0) return false; // get rid of corner cases

if(n == 1) return true; // base case

int t2 = 0, t3 = 0, t5 = 0; //pointers for 2, 3, 5

vector<int> k(n);

k[0] = 1;

for(int i = 1; i < n ; i ++)

{

k[i] = min(k[t2]\*2,min(k[t3]\*3,k[t5]\*5));

if(k[i] == k[t2]\*2) t2++;

if(k[i] == k[t3]\*3) t3++;

if(k[i] == k[t5]\*5) t5++;

}

return k[n-1];

}

};

struct Solution {

int nthUglyNumber(int n) {

vector <int> results (1,1);

int i = 0, j = 0, k = 0;

while (results.size() < n)

{

results.push\_back(min(results[i] \* 2, min(results[j] \* 3, results[k] \* 5)));

if (results.back() == results[i] \* 2) ++i;

if (results.back() == results[j] \* 3) ++j;

if (results.back() == results[k] \* 5) ++k;

}

return results.back();

}

};

**Explanation:**

The key is to realize each number can be and have to be generated by a former number multiplied by 2, 3 or 5 e.g. 1 2 3 4 5 6 8 9 10 12 15.. what is next? it must be x \* 2 or y \* 3 or z \* 5, where x, y, z is an existing number.

How do we determine x, y, z then? apparently, you can just *traverse the sequence generated by far* from 1 ... 15, until you find such x, y, z that x \* 2, y \* 3, z \* 5 is just bigger than 15. In this case x=8, y=6, z=4. Then you compare x \* 2, y \* 3, z \* 5 so you know next number will be x \* 2 = 8 \* 2 = 16. k, now you have 1,2,3,4,....,15, 16,

Then what is next? You wanna do the same process again to find the new x, y, z, but you realize, wait, do I have to *traverse the sequence generated by far* again?

NO! since you know last time, x=8, y=6, z=4 and x=8 was used to generate 16, so this time, you can immediately know the new*x = 9 (the next number after 8 is 9 in the generated sequence), y=6, z=4. Then you need to compare new*x \* 2, y \* 3, z \* 5. You know next number is 9 \* 2 = 18;

And you also know, the next x will be 10 since new\_x = 9 was used this time. But what is next y? apparently, if y=6, 6\*3 = 18, which is already generated in this round. So you also need to update next y from 6 to 8.

Based on the idea above, you can actually generated x,y,z from very beginning, and update x, y, z accordingly. It ends up with a O(n) solution.

The idea of this solution is from this page:http://www.geeksforgeeks.org/ugly-numbers/

The ugly-number sequence is 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, … because every number can only be divided by 2, 3, 5, one way to look at the sequence is to split the sequence to three groups as below:

(1) 1×2, 2×2, 3×2, 4×2, 5×2, …

(2) 1×3, 2×3, 3×3, 4×3, 5×3, …

(3) 1×5, 2×5, 3×5, 4×5, 5×5, …

We can find that every subsequence is the ugly-sequence itself (1, 2, 3, 4, 5, …) multiply 2, 3, 5.

Then we use similar merge method as merge sort, to get every ugly number from the three subsequence.

Every step we choose the smallest one, and move one step after,including nums with same value.

Thanks for this author about this brilliant idea. Here is my java solution

**这个方法不错**

每次保存因子分别是2,3,5的三个数，从中选出min则为ugly number。同时，加入新的数（要控制每个因子各自的计数）

public class Solution {

public int nthUglyNumber(int n) {

int[] ugly = new int[n];

ugly[0] = 1;

int index2 = 0, index3 = 0, index5 = 0;

int factor2 = 2, factor3 = 3, factor5 = 5;

for(int i=1;i<n;i++){

int min = Math.min(Math.min(factor2,factor3),factor5);

ugly[i] = min;

if(factor2 == min)

factor2 = 2\*ugly[++index2];

if(factor3 == min)

factor3 = 3\*ugly[++index3];

if(factor5 == min)

factor5 = 5\*ugly[++index5];

}

return ugly[n-1];

}

}

The basic idea of this problem is to compute all the ugly numbers in sequence and count to the given number of k ugly numbers. The way I approached this problem is first I have a arraylist to store the ugly numbers in sequence. Then I declared three counter variables: a,b,and c which represent the corresponding index in the arraylist for the multiplier of 2,3,and 5. Since each previous ugly number times one of the multiplier will produce a new ugly number, I start from the starting index 0 and multiply the ugly number at that index with each multiplier and get the smallest product which is the next ugly number from the three. The corresponding multipliers' index will be incremented by one and we do this recursively until we have K ugly numbers. Here is the code implementation in Java:

public class Solution {

public int nthUglyNumber(int n) {

if(n<=0) return 0;

int a=0,b=0,c=0;

List<Integer> table = new ArrayList<Integer>();

table.add(1);

while(table.size()<n)

{

int next\_val = Math.min(table.get(a)\*2,Math.min(table.get(b)\*3,table.get(c)\*5));

table.add(next\_val);

if(table.get(a)\*2==next\_val) a++;

if(table.get(b)\*3==next\_val) b++;

if(table.get(c)\*5==next\_val) c++;

}

return table.get(table.size()-1);

}

}

**两种标准dp方法**

// three lists:

// (1) 1×2, 2×2, 3×2, 4×2, 5×2, …

// (2) 1×3, 2×3, 3×3, 4×3, 5×3, …

// (3) 1×5, 2×5, 3×5, 4×5, 5×5, …

// O(n) time, O(n) space

int nthUglyNumber(int n) {

vector<int> d(n, 0);

d[0] = 1;

int f2 = 2, f3 = 3, f5 = 5; // min values for multipy factor 2, 3, 5

int ix2 = 0, ix3 = 0, ix5 = 0; // indexs for min values of f2, f3, f5

for (int i = 1; i < n; ++i) {

int minV = min(min(f2, f3), f5);

d[i] = minV;

if (minV == f2) f2 = 2 \* d[++ix2];

if (minV == f3) f3 = 3 \* d[++ix3];

if (minV == f5) f5 = 5 \* d[++ix5];

}

return d[n-1];

}

// O(n) (might be more) time, O(3n) space

int nthUglyNumber1(int n) {

queue<int> q1, q2, q3;

q1.push(1), q2.push(1), q3.push(1);

int m = 0;

for (int i = 0; i < n; ++i) {

m = min(min(q1.front(), q2.front()), q3.front());

if (m == q1.front()) q1.pop();

if (m == q2.front()) q2.pop();

if (m == q3.front()) q3.pop();

q1.push(2\*m);

q2.push(3\*m);

q3.push(5\*m);

}

return m;

}

### Ugly Number （Math题）

Total Accepted: **50986** Total Submissions: **139967** Difficulty: **Easy**

Write a program to check whether a given number is an ugly number.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 6, 8 are ugly while 14 is not ugly since it includes another prime factor 7.

Note that 1 is typically treated as an ugly number.

**我的做法，不断去除2,3,5的因子**

**public boolean isUgly(int num) {**

**if(num==0) return false;**

**while(num%2==0) num/=2;**

**while(num%3==0) num/=3;**

**while(num%5==0) num/=5;**

**if(num==1) return true;**

**else return false;**

**}**

Just divide by 2, 3 and 5 as often as possible and then check whether we arrived at 1. Also try divisor 4 if that makes the code simpler / less repetitive.

**Java / C#**

for (int i=2; i<6 && num>0; i++)

while (num % i == 0)

num /= i;

return num == 1;

**General**

Would be a bit cleaner if I did the zero-test outside, and discarding negative numbers right away can speed things up a little, but meh... I don't want to add another line and indentation level :-)

if (num > 0)

for (int i=2; i<6; i++)

while (num % i == 0)

num /= i;

return num == 1;

### Climbing Stairs

Total Accepted: **102815** Total Submissions: **281368** Difficulty: **Easy**

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?

**我的做法🡪可以转成O(1)空间**

**public int climbStairs(int n) {**

**int[] dp=new int[n+1];**

**dp[0]=1;**

**if(n>=1) dp[1]=1;**

**for(int i=2;i<=n;i++)**

**{**

**dp[i]=dp[i-1]+dp[i-2];**

**}**

**return dp[n];**

**}**

**Basically it's a fibonacci.**

The problem seems to be a *dynamic programming* one. **Hint**: the tag also suggests that! Here are the steps to get the solution incrementally.

* Base cases:  
  if n <= 0, then the number of ways should be zero. if n == 1, then there is only way to climb the stair. if n == 2, then there are two ways to climb the stairs. One solution is one step by another; the other one is two steps at one time.
* The key intuition to solve the problem is that given a number of stairs n, if we know the number ways to get to the points [n-1] and [n-2] respectively, denoted as n1 and n2 , then the total ways to get to the point [n] is n1 + n2. Because from the [n-1] point, we can take one single step to reach [n]. And from the [n-2] point, we could take two steps to get there. There is NO overlapping between these two solution sets, because we differ in the final step.

Now given the above intuition, one can construct an array where each node stores the solution for each number n. Or if we look at it closer, it is clear that this is basically a fibonacci number, with the starting numbers as 1 and 2, instead of 1 and 1.

The implementation in Java as follows:

public int climbStairs(int n) {

// base cases

if(n <= 0) return 0;

if(n == 1) return 1;

if(n == 2) return 2;

int one\_step\_before = 2;

int two\_steps\_before = 1;

int all\_ways = 0;

for(int i=2; i<n; i++){

all\_ways = one\_step\_before + two\_steps\_before;

two\_steps\_before = one\_step\_before;

one\_step\_before = all\_ways;

}

return all\_ways;

}

Hi guys, I come up with this arithmetic way. Find the inner logic relations and get the answer.

public class Solution {

public int climbStairs(int n) {

if(n == 0 || n == 1 || n == 2){return n;}

int[] mem = new int[n];

mem[0] = 1;

mem[1] = 2;

for(int i = 2; i < n; i++){

mem[i] = mem[i-1] + mem[i-2];

}

return mem[n-1];

}

Same simple algorithm written in every offered language. Variable a tells you the number of ways to reach the current step, and b tells you the number of ways to reach the next step. So for the situation one step further up, the old b becomes the new a, and the new b is the old a+b, since that new step can be reached by climbing 1 step from what b represented or 2 steps from what a represented.

Ruby wins, and *"the C languages"* all look the same.

**C++** (0 ms)

int climbStairs(int n) {

int a = 1, b = 1;

while (n--)

a = (b += a) - a;

return a;

}

**Java** (208 ms)

public int climbStairs(int n) {

int a = 1, b = 1;

while (n-- > 0)

a = (b += a) - a;

return a;

}

### Guess Number Higher or Lower II

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/guess-number-higher-or-lower-ii/submissions/)

* Total Accepted: **4629**
* Total Submissions: **15403**
* Difficulty: **Medium**

We are playing the Guess Game. The game is as follows:

I pick a number from **1** to **n**. You have to guess which number I picked.

Every time you guess wrong, I'll tell you whether the number I picked is higher or lower.

However, when you guess a particular number x, and you guess wrong, you pay **$x**. You win the game when you guess the number I picked.

**Example:**

n = 10, I pick 8.

First round: You guess 5, I tell you that it's higher. You pay $5.

Second round: You guess 7, I tell you that it's higher. You pay $7.

Third round: You guess 9, I tell you that it's lower. You pay $9.

Game over. 8 is the number I picked.

You end up paying $5 + $7 + $9 = $21.

Given a particular **n ≥ 1**, find out how much money you need to have to guarantee a **win**.

**Hint:**

1. The best strategy to play the game is to minimize the maximum loss you could possibly face. Another strategy is to minimize the expected loss. Here, we are interested in the **first** scenario.
2. Take a small example (n = 3). What do you end up paying in the worst case?
3. Check out [this article](https://en.wikipedia.org/wiki/Minimax) if you're still stuck.
4. The purely recursive implementation of minimax would be worthless for even a small n. You MUST use dynamic programming.
5. As a follow-up, how would you modify your code to solve the problem of minimizing the expected loss, instead of the worst-case loss?

For each number x in range[i~j]  
we do: result\_when\_pick\_x = x + **max**{DP([i~x-1]), DP([x+1, j])}  
--> // the max means whenever you choose a number, the feedback is always bad and therefore leads you to a worse branch.  
then we get DP([i~j]) = **min**{xi, ... ,xj}  
--> // this min makes sure that you are minimizing your cost.

public class Solution {

public int getMoneyAmount(int n) {

int[][] table = new int[n+1][n+1];

return DP(table, 1, n);

}

int DP(int[][] t, int s, int e){

if(s >= e) return 0;

if(t[s][e] != 0) return t[s][e];

int res = Integer.MAX\_VALUE;

for(int x=s; x<=e; x++){

int tmp = x + Math.max(DP(t, s, x-1), DP(t, x+1, e));

res = Math.min(res, tmp);

}

t[s][e] = res;

return res;

}

}

Here is a bottom up solution.

public class Solution {

public int getMoneyAmount(int n) {

int[][] table = new int[n+1][n+1];

for(int j=2; j<=n; j++){

for(int i=j-1; i>0; i--){

int globalMin = Integer.MAX\_VALUE;

for(int k=i+1; k<j; k++){

int localMax = k + Math.max(table[i][k-1], table[k+1][j]);

globalMin = Math.min(globalMin, localMax);

}

table[i][j] = i+1==j?i:globalMin;

}

}

return table[1][n];

}

}

这个解释不错

Definition of dp[i][j]: minimum number of money to guarantee win for subproblem [i, j].

Target: dp[1][n]

Corner case: dp[i][i] = 0 (because the only element must be correct)

Equation: we can choose k (i<=k<=j) as our guess, and pay price k. After our guess, the problem is divided into two subproblems. Notice we do not need to pay the money for both subproblems. We only need to pay the worst case (because the system will tell us which side we should go) to guarantee win. So dp[i][j] = min (i<=k<=j) { k + max(dp[i][k-1], dp[k+1][j]) }

public class Solution {

public int getMoneyAmount(int n) {

if (n == 1) {

return 0;

}

int[][] dp = new int[n + 1][n + 1];

for (int jminusi = 1; jminusi < n; jminusi++) {

for (int i = 0; i + jminusi <= n; i++) {

int j = i + jminusi;

dp[i][j] = Integer.MAX\_VALUE;

for (int k = i; k <= j; k++) {

dp[i][j] = Math.min(dp[i][j], k + Math.max(k - 1 >= i ? dp[i][k - 1] : 0, j >= k + 1 ? dp[k + 1][j] : 0));

}

}

}

return dp[1][n];

}

}

Thanks for your idea, I rewrite the code using your idea:

public int getMoneyAmount(int n) {

int[][] dp = new int[n + 2][n + 2];

for (int l = 1; l < n; l++)

for (int i = 1; i <= n - l; i++) {

int j = i + l;

dp[i][j] = Integer.MAX\_VALUE;

for (int k = i; k <= j; k++)

dp[i][j] = Math.min(dp[i][j], k + Math.max(dp[i][k - 1], dp[k + 1][j]));

}

return dp[1][n];

}

**Big Idea: Given any n, we make a guess k. Then we break the interval [1,n] into [1,k - 1] and [k + 1,n]. The min of worst case cost can be calculated recursively as**

**cost[1,n] = k + max{cost[1,k - 1] + cost[k+1,n]}**  
Also, it takes a while for me to wrap my head around "min of max cost". My understand is that: you strategy is the best, but your luck is the worst. You only guess right when there is no possibilities to guess wrong.

**public** **class** **Solution** {

**public** **int** **getMoneyAmount**(**int** n) {

*// all intervals are inclusive*

*// uninitialized cells are assured to be zero*

*// the zero column and row will be uninitialized*

*// the illegal cells will also be uninitialized*

*// add 1 to the length just to make the index the same as numbers used*

**int**[][] dp = **new** **int**[n + 1][n + 1]; *// dp[i][j] means the min cost in the worst case for numbers (i...j)*

*// iterate the lengths of the intervals since the calculations of longer intervals rely on shorter ones*

**for** (**int** l = 2; l <= n; l++) {

*// iterate all the intervals with length l, the start of which is i. Hence the interval will be [i, i + (l - 1)]*

**for** (**int** i = 1; i <= n - (l - 1); i++) {

dp[i][i + (l - 1)] = Integer.MAX\_VALUE;

*// iterate all the first guesses g*

**for** (**int** g = i; g <= i + (l - 1); g++) {

**int** costForThisGuess;

*// since if g is the last integer, g + 1 does not exist, we have to separate this case*

*// cost for [i, i + (l - 1)]: g (first guess) + max{the cost of left part [i, g - 1], the cost of right part [g + 1, i + (l - 1)]}*

**if** (g == n) {

costForThisGuess = dp[i][g - 1] + g;

} **else** {

costForThisGuess = g + Math.max(dp[i][g - 1], dp[g + 1][i + (l - 1)]);

}

dp[i][i + (l - 1)] = Math.min(dp[i][i + (l - 1)], costForThisGuess); *// keep track of the min cost among all first guesses*

}

}

}

**return** dp[1][n];

}

}

a little improvement :)

这个写法不错

**public** **int** **getMoneyAmount**(**int** n) {

**int**[][] dp = **new** **int**[n+1][n+1];

**for**(**int** len=1;len<n;len++){//增长(i与j直接的间距)

**for**(**int** i=1;i+len<=n;i++){

**int** j=i+len;

**int** min = Integer.MAX\_VALUE;

**for**(**int** k=i;k<j;k++){ //[i,j]

**int** tmp = k+Math.max(dp[i][k-1],dp[k+1][j]);

min = Math.min(min,tmp);

}

dp[i][j] = min;

}

}

**return** dp[1][n];

}

### Count Numbers with Unique Digits

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/count-numbers-with-unique-digits/submissions/)

* Total Accepted: **11815**
* Total Submissions: **27232**
* Difficulty: **Medium**

Given a **non-negative** integer n, count all numbers with unique digits, x, where 0 ≤ x < 10n.

**Example:**  
Given n = 2, return 91. (The answer should be the total numbers in the range of 0 ≤ x < 100, excluding [11,22,33,44,55,66,77,88,99])

**Hint:**

1. A direct way is to use the backtracking approach.
2. Backtracking should contains three states which are (the current number, number of steps to get that number and a bitmask which represent which number is marked as visited so far in the current number). Start with state (0,0,0) and count all valid number till we reach number of steps equals to 10n.
3. This problem can also be solved using a dynamic programming approach and some knowledge of combinatorics.
4. Let f(k) = count of numbers with unique digits with length equals k.
5. f(1) = 10, ..., f(k) = 9 \* 9 \* 8 \* ... (9 - k + 2) [The first factor is 9 because a number cannot start with 0].

**Credits:**  
Special thanks to [@memoryless](https://discuss.leetcode.com/user/memoryless) for adding this problem and creating all test cases.

[Subscribe](https://leetcode.com/subscribe/) to see which companies asked this question

Hide Tags

[Dynamic Programming](https://leetcode.com/tag/dynamic-programming/) [Backtracking](https://leetcode.com/tag/backtracking/) [Math](https://leetcode.com/tag/math/)

窗体顶端

窗体底端

**我的做法，f[k]是长度为k的，每一位都不同的，数字的个数（用hint的方法）**

**public class Solution {**

**public int countNumbersWithUniqueDigits(int n) {**

**int cnt=0;**

**if(n==0) return 1;**

**cnt+=10;**

**int mul=9;**

**for(int k=2;11-k>0 && k<=n;k++)**

**{**

**mul\*=(11-k);**

**cnt+=mul;**

**}**

**return cnt;**

**}**

**}**

### Partition Equal Subset Sum

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/partition-equal-subset-sum/submissions/)

* Total Accepted: **4074**
* Total Submissions: **10607**
* Difficulty: **Medium**
* Contributors: **Admin**

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.

**这个方法很好**

**Java Solution similar to backpack problem - Easy to understand**

**public** **class** **Solution** {

**public** **boolean** **canPartition**(**int**[] nums) {

*// check edge case*

**if** (nums == **null** || nums.length == 0) {

**return** **true**;

}

*// preprocess*

**int** volumn = 0;

**for** (**int** num : nums) {

volumn += num;

}

**if** (volumn % 2 != 0) {

**return** **false**;

}

volumn /= 2;

*// dp def*

**boolean**[] dp = **new** **boolean**[volumn + 1];

*// dp init*

dp[0] = **true**;

*// dp transition*

**for** (**int** i = 1; i <= nums.length; i++) {

**for** (**int** j = volumn; j >= nums[i-1]; j--) {

dp[j] = dp[j] || dp[j - nums[i-1]];

}

}

**return** dp[volumn];

}

}

**dp[i]：使用数组中的某些元素，可以达到值i**

If sum of all the numbers is odd, the surely we cannot reach equal partition.

Using a boolean dp array (limit its max index to sum/2) whose ith entry indicates there is a way to reach the value i using certain subset of the numbers. SO if at any time we can find a subset to reach sum/2 index, we are able to equally partition.

Disclaimer: logic borrowed from <https://chinmaylokesh.wordpress.com/2011/02/10/balanced-partition-problem-finding-the-minimized-sum-between-two-partitions-of-a-set-of-positive-integers/>

**public** **boolean** **canPartition**(**int**[] a) {

**int** sum = 0;

**for**(**int** n:a){

sum += n;

}

**if**(sum%2>0)

**return** **false**;

**boolean** []dp = **new** **boolean**[sum/2+1];

dp[0]=**true**; *// empty array*

**int** max=0;

**for**(**int** n: a){

**if**(n>sum/2)

**return** **false**; *// single number making bigger than sum/2 no way equal partition.*

**for**(**int** j = 0; j<=max; j++){

**if**(dp[j] && ((j+n) <= sum/2) ){

dp[j+n] = **true**;

max = Math.max(max, j+n);

**if**(max==sum/2)

**return** **true**;

}

}

}

**return** dp[sum/2];

}

### Is Subsequence

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/is-subsequence/submissions/)

* Total Accepted: **11965**
* Total Submissions: **27367**
* Difficulty: **Medium**
* Contributors: **Admin**

Given a string **s** and a string **t**, check if **s** is subsequence of **t**.

You may assume that there is only lower case English letters in both **s** and **t**. **t** is potentially a very long (length ~= 500,000) string, and **s** is a short string (<=100).

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ace" is a subsequence of "abcde" while "aec" is not).

**Example 1:**  
**s** = "abc", **t** = "ahbgdc"

Return true.

**Example 2:**  
**s** = "axc", **t** = "ahbgdc"

Return false.

**Follow up:**  
If there are lots of incoming S, say S1, S2, ... , Sk where k >= 1B, and you want to check one by one to see if T has its subsequence. In this scenario, how would you change your code?

**Credits:**  
Special thanks to [@pbrother](https://leetcode.com/pbrother/) for adding this problem and creating all test cases.

[Subscribe](https://leetcode.com/subscribe/) to see which companies asked this question

Hide Tags

[Binary Search](https://leetcode.com/tag/binary-search/) [Dynamic Programming](https://leetcode.com/tag/dynamic-programming/) [Greedy](https://leetcode.com/tag/greedy/)

这个方法很好，使用两个指针，分别指向s和t。当指向的元素相等时，移动s，不然一直移动t的指针。窗体底端

Just use two pointers:

**public** **class** **Solution** {

**public** **boolean** **isSubsequence**(String s, String t) {

**if** (s.length() == 0) **return** **true**;

**int** indexS = 0, indexT = 0;

**while** (indexT < t.length()) {

**if** (t.charAt(indexT) == s.charAt(indexS)) {

indexS++;

**if** (indexS == s.length()) **return** **true**;

}

indexT++;

}

**return** **false**;

}

}

**Java. Only 2ms. Much faster than normal 2 pointers.**

Actually another way of 2 pointers, guess indexOf() performs better.

Tested with other 2-pointers methods in discussion, both took 30ms+.

The one below only takes 2ms.

**public** **class** **Solution**

{

**public** **boolean** **isSubsequence**(String s, String t)

{

**if**(t.length() < s.length()) **return** **false**;

**int** prev = 0;

**for**(**int** i = 0; i < s.length();i++)

{

**char** tempChar = s.charAt(i);

prev = t.indexOf(tempChar,prev);

**if**(prev == -1) **return** **false**;

prev++;

}

**return** **true**;

}

}

Thanks to [@Ankai.Liang](https://discuss.leetcode.com/uid/39025) who looked into both functions and provided us the answer.

In case you guys do not notice, I post Liang Ankai's answer.

[@Ankai.Liang](https://discuss.leetcode.com/uid/39025) said

*Hi, good solution.  
I checked the origin code of func "indexOf" and "charAt". These two solution both traversed the char of String one by one to search the first occurrence specific char.  
The difference is that indexOf only call once function then traversed in "String.value[]" arr, but we used multiple calling function "charAt" to get the value in "String.value[]" arr.  
The time expense of calling function made the difference.*

Re: [Java binary search using TreeSet got TLE](https://discuss.leetcode.com/topic/57994/java-binary-search-using-treeset-got-tle)

I think the Map and TreeSet could be simplified by Array and binarySearch. Since we scan T from beginning to the end (index itself is in increasing order), List will be sufficient. Then we can use binarySearch to replace with TreeSet ability which is a little overkill for this problem. Here is my solution.

*// Follow-up: O(N) time for pre-processing, O(Mlog?) for each S.*

*// Eg-1. s="abc", t="bahbgdca"*

*// idx=[a={1,7}, b={0,3}, c={6}]*

*// i=0 ('a'): prev=1*

*// i=1 ('b'): prev=3*

*// i=2 ('c'): prev=6 (return true)*

*// Eg-2. s="abc", t="bahgdcb"*

*// idx=[a={1}, b={0,6}, c={5}]*

*// i=0 ('a'): prev=1*

*// i=1 ('b'): prev=6*

*// i=2 ('c'): prev=? (return false)*

**public** **boolean** **isSubsequence**(String s, String t) {

List<Integer>[] idx = **new** List[256]; *// Just for clarity*

**for** (**int** i = 0; i < t.length(); i++) {

**if** (idx[t.charAt(i)] == **null**)

idx[t.charAt(i)] = **new** ArrayList<>();

idx[t.charAt(i)].add(i);

}

**int** prev = 0;

**for** (**int** i = 0; i < s.length(); i++) {

**if** (idx[s.charAt(i)] == **null**) **return** **false**; *// Note: char of S does NOT exist in T causing NPE*

**int** j = Collections.binarySearch(idx[s.charAt(i)], prev);

**if** (j < 0) j = -j - 1;

**if** (j == idx[s.charAt(i)].size()) **return** **false**;

prev = idx[s.charAt(i)].get(j) + 1;

}

**return** **true**;

}

### Combination Sum IV

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/combination-sum-iv/submissions/)

* Total Accepted: **15246**
* Total Submissions: **37477**
* Difficulty: **Medium**
* Contributors: **Admin**

Given an integer array with all positive numbers and no duplicates, find the number of possible combinations that add up to a positive integer target.

**Example:**

***nums*** = [1, 2, 3]

***target*** = 4

The possible combination ways are:

(1, 1, 1, 1)

(1, 1, 2)

(1, 2, 1)

(1, 3)

(2, 1, 1)

(2, 2)

(3, 1)

Note that different sequences are counted as different combinations.

Therefore the output is ***7***.

**Follow up:**  
What if negative numbers are allowed in the given array?  
How does it change the problem?  
What limitation we need to add to the question to allow negative numbers?

**题目类似爬楼梯**

Wish to learn better solutions from you guys.

**public** **class** **Solution** {

**public** **int** **combinationSum4**(**int**[] nums, **int** target) {

Arrays.sort(nums);

**int**[] res = **new** **int**[target + 1];

**for** (**int** i = 1; i < res.length; i++) {

**for** (**int** num : nums) {

**if** (num > i)

**break**;

**else** **if** (num == i)

res[i] += 1;

**else**

res[i] += res[i-num];

}

}

**return** res[target];

}

}

[@steve.j.sun](https://discuss.leetcode.com/uid/134) At first I tried the recursive version, but got TLE, then this DP solution came to me. The idea comes from<https://leetcode.com/problems/climbing-stairs/>

**1ms Java DP Solution with Detailed Explanation**

Think about the recurrence relation(递推关系) first. How does the # of combinations of the target related to the # of combinations of numbers that are smaller than the target?

So we know that target is the sum of numbers in the array. Imagine we only need one more number to reach target, this number can be any one in the array, right? So the # of combinations of target, comb[target] = sum(comb[target - nums[i]]), where 0 <= i < nums.length, and target >= nums[i].

In the example given, we can actually find the # of combinations of 4 with the # of combinations of 3(4 - 1), 2(4- 2) and 1(4 - 3). As a result, comb[4] = comb[4-1] + comb[4-2] + comb[4-3] = comb[3] + comb[2] + comb[1].

Then think about the base case. Since if the target is 0, there is only one way to get zero, which is using 0, we can set comb[0] = 1.

EDIT: The problem says that target is a positive integer that makes me feel it's unclear to put it in the above way. Since target == 0only happens when in the previous call, target = nums[i], we know that this is the only combination in this case, so we return 1.

Now we can come up with at least a recursive solution.

**public** **int** **combinationSum4**(**int**[] nums, **int** target) {

**if** (target == 0) {

**return** 1;

}

**int** res = 0;

**for** (**int** i = 0; i < nums.length; i++) {

**if** (target >= nums[i]) {

res += combinationSum4(nums, target - nums[i]);

}

}

**return** res;

}

Now for a DP solution, we just need to figure out a way to store the intermediate results, to avoid the same combination sum being calculated many times. We can use an array to save those results, and check if there is already a result before calculation. We can fill the array with -1 to indicate that the result hasn't been calculated yet. 0 is not a good choice because it means there is no combination sum for the target.

**private** **int**[] dp;

**public** **int** **combinationSum4**(**int**[] nums, **int** target) {

dp = **new** **int**[target + 1];

Arrays.fill(dp, -1);

dp[0] = 1;

**return** helper(nums, target);

}

**private** **int** **helper**(**int**[] nums, **int** target) {

**if** (dp[target] != -1) {

**return** dp[target];

}

**int** res = 0;

**for** (**int** i = 0; i < nums.length; i++) {

**if** (target >= nums[i]) {

res += helper(nums, target - nums[i]);

}

}

dp[target] = res;

**return** res;

}

这个写法不错，类似爬楼梯

EDIT: The above solution is top-down. How about a bottom-up one?

**public** **int** **combinationSum4**(**int**[] nums, **int** target) {

**int**[] comb = **new** **int**[target + 1];

comb[0] = 1;

**for** (**int** i = 1; i < comb.length; i++) {

**for** (**int** j = 0; j < nums.length; j++) {

**if** (i - nums[j] >= 0) {

comb[i] += comb[i - nums[j]];

}

}

}

**return** comb[target];

}

**JAVA recursion solution using HashMap as memory.**

The DP solution goes through every possible sum from 1 to target one by one.  
Using recursion can skip those sums that are not the combinations of the numbers in the given array. Also, there is no need to sort the array first.

**public** **class** Solution {

Map<Integer, Integer> map = **new** HashMap<>();

**public** **int** **combinationSum4**(**int**[] nums, **int** target) {

**int** count = 0;

**if** (nums == null || nums.length ==0 || target < 0 ) **return** 0;

**if** ( target ==0 ) **return** 1;

**if** (map.containsKey(target)) **return** map.get(target);

**for** (**int** num: nums){

count += combinationSum4(nums, target-num);

}

map.put(target, count);

**return** count;

}

}

### Integer Break

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/integer-break/submissions/)

* Total Accepted: **28360**
* Total Submissions: **64413**
* Difficulty: **Medium**
* Contributors: **Admin**

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.

**Hint:**

1. There is a simple O(n) solution to this problem.
2. You may check the breaking results of *n* ranging from 7 to 10 to discover the regularities.

**题目：数字n拆成至少两个数相加，使这些数乘积最大。**

**Why factor 2 or 3? The math behind this problem.**

I saw many solutions were referring to factors of 2 and 3. But why these two magic numbers? Why other factors do not work?  
Let's study the math behind it.

For convenience, say **n** is sufficiently large and can be broken into any smaller real positive numbers. We now try to calculate which real number generates the largest product.  
Assume we break **n** into **(n / x)** **x**'s, then the product will be **xn/x**, and we want to maximize it.

Taking its derivative（导数） gives us **n \* xn/x-2 \* (1 - ln(x))**.  
The derivative is positive when **0 < x < e**, and equal to **0** when **x = e**, then becomes negative when **x > e**,  
which indicates that the product increases as **x** increases, then reaches its maximum when **x = e**, then starts dropping.

This reveals the fact that if **n** is sufficiently large and we are allowed to break **n** into real numbers,  
the best idea is to break it into nearly all **e**'s.  
On the other hand, if **n** is sufficiently large and we can only break **n** into integers, we should choose integers that are closer to **e**.  
The only potential candidates are **2** and **3** since **2 < e < 3**, but we will generally prefer **3** to **2**. Why?

Of course, one can prove it based on the formula above, but there is a more natural way shown as follows.

**6 = 2 + 2 + 2 = 3 + 3**. But **2 \* 2 \* 2 < 3 \* 3**.  
Therefore, if there are three **2**'s in the decomposition, we can replace them by two **3**'s to gain a larger product.

All the analysis above assumes **n** is significantly large. When **n** is small (say **n <= 10**), it may contain flaws.  
For instance, when **n = 4**, we have **2 \* 2 > 3 \* 1**.  
To fix it, we keep breaking **n** into **3**'s until **n** gets smaller than **10**, then solve the problem by brute-force.

but I think  
when y = x^(n/x), y' = x^(n/x) \* n/x^2 \* (1-ln(x))

The first thing we should consider is : What is the max product if we break a number N into two factors?

I use a function to express this product: f=x(N-x)

When x=N/2, we get the maximum of this function.

However, factors should be integers. Thus the maximum is (N/2)\*(N/2) when N is even or (N-1)/2 \*(N+1)/2 when N is odd.

When the maximum of f is larger than N, we should do the break.

(N/2)\*(N/2)>=N, then N>=4

(N-1)/2 \*(N+1)/2>=N, then N>=5

These two expressions mean that factors should be less than 4, otherwise we can do the break and get a better product. The factors in last result should be 1, 2 or 3. Obviously, 1 should be abandoned. Thus, the factors of the perfect product should be 2 or 3.

The reason why we should use 3 as many as possible is

For 6, 3 \* 3>2 \* 2 \* 2. Thus, the optimal product should contain no more than three 2.

Below is my accepted, O(N) solution.

**public** **class** **Solution** {

**public** **int** **integerBreak**(**int** n) {

**if**(n==2) **return** 1;

**if**(n==3) **return** 2;

**int** product = 1;

**while**(n>4){

product\*=3;

n-=3;

}

product\*=n;

**return** product;

}

}

**O(log(n)) Time solution with explanation**

Given a number n lets say we have a possible product P = p1 \* p2 \* ... *pk. Then we notice what would happen if we could break pi up into two more terms lets say one of the terms is 2 we would get the terms pi-2 and 2 so if 2*(pi-2) > pi we would get a bigger product and this happens if pi > 4. since there is one other possible number less then 4 that is not 2 aka 3. Likewise for 3 if we instead breakup the one of the terms into pi-3 and 3 we would get a bigger product if 3\*(pi-3) > pi which happens if pi > 4.5.

Hence we see that all of the terms in the product must be 2's and 3's. So we now just need to write n = a*3 + b*2 such that P = (3^a) \* (2^b) is maximized. Hence we should favor more 3's then 2's in the product then 2's if possible.

So if n = a\*3 then the answer will just be 3^a.

if n = a*3 + 2 then the answer will be 2*(3^a).

and if n = a*3 + 2*2 then the answer will be 2 \* 2 \* 3^a

The above three cover all cases that n can be written as and the Math.pow() function takes O(log n) time to preform hence that is the running time.

**public** **class** Solution {

**public** **int** **integerBreak**(**int** n) {

**if**(n == 2)

**return** 1;

**else** **if**(n == 3)

**return** 2;

**else** **if**(n%3 == 0)

**return** (**int**)Math.pow(3, n/3);

**else** **if**(n%3 == 1)

**return** 2 \* 2 \* (**int**) Math.pow(3, (n - 4) / 3);

**else**

**return** 2 \* (**int**) Math.pow(3, n/3);

}

}

**DP解法**

**public** **int** **integerBreak**(**int** n) {

**int**[] dp = **new** **int**[n + 1];

dp[1] = 1;

**for**(**int** i = 2; i <= n; i ++) {

**for**(**int** j = 1; j < i; j ++) {

dp[i] = Math.max(dp[i], (Math.max(j,dp[j])) \* (Math.max(i - j, dp[i - j])));

}

}

**return** dp[n];

}

**这个写法容易理解**

Basic idea **is** to divide your number into threes **unless** **when** the last number **is** 4

Eg :

7 = 3 \* 2 \* 2

8 = 3 \* 3 \* 2

9 = 3 \* 3 \* 3

10 = 3 \* 3 \* 2 \* 2

11 = 3 \* 3 \* 3 \* 2

12 = 3 \* 3 \* 3 \* 3

13 = 3 \* 3 \* 3 \* 2 \* 2

See, the pattern?

**public** **static** **int** **integerBreak**(**int** n) {

**if**(n==2||n==3) **return** n-1;

**if**(n==4) **return** 4;

**int** temp = n;

**int** sum = 1;

**while**(temp>4){

temp = temp -3;

sum = sum\*3;

}

**return** sum\*temp;

}

### 总结

初值的处理！！

找到递推规律

考虑背包问题

完全背包：顺序遍历（相同物品可重复多次）

0-1背包：逆序遍历（保证每个物品遍历一次）

装满：初值设inf

不装满

for 物品1~n

for 重=物品1.重；重<=背包.V; 重++

dp[j]=min/max{dp[j], dp[j-重]+价值} //前者代表不放入该物品，后者表示放入该物品

## 链表

### Palindrome Linked List

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/palindrome-linked-list/submissions/)

* Total Accepted: **68535**
* Total Submissions: **223909**
* Difficulty: **Easy**

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?

我的做法

public boolean isPalindrome(ListNode head) {

ListNode slow=head, fast=head, prev=null;

int cnt=0;

while(fast!=null && fast.next!=null)

{

fast=fast.next.next; //注意要先用这步

ListNode tmp=slow.next;

slow.next=prev;

prev=slow;

slow=tmp; //保存slow.next,再将slow.next指向prev

}

if(fast!=null) //slow is the mid

{

slow=slow.next;

}

while(slow!=null)

{

if(prev.val!=slow.val) return false;

slow=slow.next;

prev=prev.next;

}

return true;

}

**Share my C++ solution, O(n) time and O(1) memory**

class Solution {

public:

bool isPalindrome(ListNode\* head) {

if(head==NULL||head->next==NULL)

return true;

ListNode\* slow=head;

ListNode\* fast=head;

while(fast->next!=NULL&&fast->next->next!=NULL){

slow=slow->next;

fast=fast->next->next;

}

slow->next=reverseList(slow->next);

slow=slow->next;

while(slow!=NULL){

if(head->val!=slow->val)

return false;

head=head->next;

slow=slow->next;

}

return true;

}

ListNode\* reverseList(ListNode\* head) {

ListNode\* pre=NULL;

ListNode\* next=NULL;

while(head!=NULL){

next=head->next;

head->next=pre;

pre=head;

head=next;

}

return pre;

}

};

**下面这个做法和我的一样，没我写的清晰**

**Easy understand JAVA solution (O(1) space cost)**

*/\*\**

*\* Definition for singly-linked list.*

*\* public class ListNode {*

*\* int val;*

*\* ListNode next;*

*\* ListNode(int x) { val = x; }*

*\* }*

*\*/*

**public** **class** **Solution** {

**public** **boolean** **isPalindrome**(ListNode head) {

**if**(head == **null**) {

**return** **true**;

}

ListNode p1 = head;

ListNode p2 = head;

ListNode p3 = p1.next;

ListNode pre = p1;

*//find mid pointer, and reverse head half part*

**while**(p2.next != **null** && p2.next.next != **null**) {

p2 = p2.next.next;

pre = p1;

p1 = p3;

p3 = p3.next;

p1.next = pre;

}

*//odd number of elements, need left move p1 one step*

**if**(p2.next == **null**) {

p1 = p1.next;

}

**else** { *//even number of elements, do nothing*

}

*//compare from mid to head/tail*

**while**(p3 != **null**) {

**if**(p1.val != p3.val) {

**return** **false**;

}

p1 = p1.next;

p3 = p3.next;

}

**return** **true**;

}

}

### Merge Two Sorted Lists

[My Submissions](https://leetcode.com/problems/merge-two-sorted-lists/submissions/)

Question

Total Accepted: **117312** Total Submissions: **334550** Difficulty: **Easy**

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

这个方法也不错。递归。

class Solution {

public:

ListNode \*mergeTwoLists(ListNode \*l1, ListNode \*l2) {

**if**(l1 == NULL) **return** l2;

**if**(l2 == NULL) **return** l1;

**if**(l1->val < l2->val) {

l1->next = mergeTwoLists(l1->next, l2);

**return** l1;

} **else** {

l2->next = mergeTwoLists(l2->next, l1);

**return** l2;

}

}

};

This solution is not a tail-recursive, the stack will overflow while the list is too long :) <http://en.wikipedia.org/wiki/Tail_call>

这个方法不错。用dummy来确定链表的头部。Tail是新的链表的结尾节点。依次插入两个链表中较小的元素。注意结尾的处理。

class Solution {

public:

ListNode \*mergeTwoLists(ListNode \*l1, ListNode \*l2) {

ListNode dummy(INT\_MIN);

ListNode \*tail = &dummy;

**while** (l1 && l2) {

**if** (l1->val < l2->val) {

tail->next = l1;

l1 = l1->next;

} **else** {

tail->next = l2;

l2 = l2->next;

}

tail = tail->next;

}

tail->next = l1 ? l1 : l2;

**return** dummy.**next**;

}

};

Hello every one, here is my code, simple but works well:

**public** **class** **Solution** {

**public** ListNode mergeTwoLists(ListNode l1, ListNode l2) {

**if**(l1 == **null**){

**return** l2;

}

**if**(l2 == **null**){

**return** l1;

}

ListNode mergeHead;

**if**(l1.val < l2.val){

mergeHead = l1;

mergeHead.next = mergeTwoLists(l1.next, l2);

}

**else**{

mergeHead = l2;

mergeHead.next = mergeTwoLists(l1, l2.next);

}

**return** mergeHead;

}

}

**public** ListNode mergeTwoLists(ListNode l1, ListNode l2) {

**if** (l1 == **null**) **return** l2;

**if** (l2 == **null**) **return** l1;

ListNode head = l1.val < l2.val ? l1 : l2;

ListNode nonHead = l1.val < l2.val ? l2 : l1;

head.next = mergeTwoLists(head.next, nonHead);

**return** head;

}

### Remove Linked List Elements

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/remove-linked-list-elements/submissions/)

* Total Accepted: **82825**
* Total Submissions: **273060**
* Difficulty: **Easy**

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

我的做法

public class Solution {

public ListNode removeElements(ListNode head, int val) {

ListNode dummy=new ListNode(0);

dummy.next=head;

ListNode cur=head, prev=dummy;

while(cur!=null)

{

if(cur.val==val)

{

prev.next=cur.next;

}

else

{

prev=cur;

}

cur=cur.next;

}

return dummy.next;

}

}

这个方法不错

**public** ListNode **removeElements**(ListNode head, **int** val) {

**if** (head == **null**) **return** **null**;

head.next = removeElements(head.next, val);

**return** head.val == val ? head.next : head;

}

**和我的做法一样，使用prev和dummy**

**public** **class** **Solution** {

**public** ListNode **removeElements**(ListNode head, **int** val) {

ListNode fakeHead = **new** ListNode(-1);

fakeHead.next = head;

ListNode curr = head, prev = fakeHead;

**while** (curr != **null**) {

**if** (curr.val == val) {

prev.next = curr.next;

} **else** {

prev = prev.next;

}

curr = curr.next;

}

**return** fakeHead.next;

}

}

### Delete Node in a Linked List

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/delete-node-in-a-linked-list/submissions/)

* Total Accepted: **109582**
* Total Submissions: **244664**
* Difficulty: **Easy**

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 -> 4after calling your function.

同书

public class Solution {

public void deleteNode(ListNode node) {

int data=node.next.val;

node.val=data;

node.next=node.next.next;

}

}

### Remove Nth Node From End of List

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/remove-nth-node-from-end-of-list/submissions/)

* Total Accepted: **134307**
* Total Submissions: **429094**
* Difficulty: **Easy**

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.  
Try to do this in one pass.

我的做法，参考书

书上还有错误检查

public ListNode removeNthFromEnd(ListNode head, int n) {

ListNode slow=head, fast=head, prev=head;

for(int i=0; i<n-1; i++)

{

fast=fast.next;

}

while(fast.next!=null)

{

prev=slow;

slow=slow.next;

fast=fast.next;

}

if(slow==head) return head.next;

prev.next=slow.next;

return head;

}

### Intersection of Two Linked Lists

[My Submissions](https://leetcode.com/problems/intersection-of-two-linked-lists/submissions/)

QuestionEditorial Solution

Total Accepted: **70619** Total Submissions: **233613** Difficulty: **Easy**

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:

A: a1 → a2

↘

c1 → c2 → c3

↗

B: b1 → b2 → b3

begin to intersect at node c1.

**Notes:**

* If the two linked lists have no intersection at all, return null.
* The linked lists must retain their original structure after the function returns.
* You may assume there are no cycles anywhere in the entire linked structure.
* Your code should preferably run in O(n) time and use only O(1) memory.

ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB)

{

ListNode \*p1 = headA;

ListNode \*p2 = headB;

**if** (p1 == **NULL** || p2 == **NULL**) **return** **NULL**;

**while** (p1 != **NULL** && p2 != **NULL** && p1 != p2) {

p1 = p1->next;

p2 = p2->next;

*//*

*// Any time they collide or reach end together without colliding*

*// then return any one of the pointers.*

*//*

**if** (p1 == p2) **return** p1;

*//*

*// If one of them reaches the end earlier then reuse it*

*// by moving it to the beginning of other list.*

*// Once both of them go through reassigning,*

*// they will be equidistant from the collision point.*

*//*

**if** (p1 == **NULL**) p1 = headB;

**if** (p2 == **NULL**) p2 = headA;

}

**return** p1;

}

这个方法很好。双指针在链表上分别移动，相遇则退出。如果没有相遇，但是其中指针为null（会遍历到链表的null）了，则重新调整指针至另一个链表的开头。

设链表1：x+z

链表2：y+z

则指针1需要行进 x+z+y+z

指针2需要行进 y+z+x+z

在z处交汇。

如果两个链表没有重合部分，则两轮迭代后两个指针都为null

**Java solution without knowing the difference in len!**

I found most solutions here preprocess linkedlists to get the difference in len.  
Actually we don't care about the "value" of difference, we just want to make sure two pointers reach the intersection node at the same time.

We can use two iterations to do that. In the first iteration, we will reset the pointer of one linkedlist to the head of another linkedlist after it reaches the tail node. In the second iteration, we will move two pointers until they points to the same node. Our operations in first iteration will help us counteract（抵消；中和；阻碍）the difference. So if two linkedlist intersects, the meeting point in second iteration must be the intersection point. If the two linked lists have no intersection at all, then the meeting pointer in second iteration must be the tail node of both lists, which is null

Below is my commented Java code:

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

//boundary check

if(headA == null || headB == null) return null;

ListNode a = headA;

ListNode b = headB;

//if a & b have different len, then we will stop the loop after second iteration

while( a != b){

//for the end of first iteration, we just reset the pointer to the head of another linkedlist

a = a == null? headB : a.next;

b = b == null? headA : b.next;

}

return a;

}

这个方法也不错。

首先，计算出两个链表的长度。

然后，移动链表指针，使两个链表剩下部分的长度一致（start point找准）

1, Get the length of the two lists.

2, Align them to the same start point.

3, Move them together until finding the intersection point, or the end null

**public** ListNode **getIntersectionNode**(ListNode headA, ListNode headB) {

**int** lenA = length(headA), lenB = length(headB);

*// move headA and headB to the same start point*

**while** (lenA > lenB) {

headA = headA.next;

lenA--;

}

**while** (lenA < lenB) {

headB = headB.next;

lenB--;

}

*// find the intersection until end*

**while** (headA != headB) {

headA = headA.next;

headB = headB.next;

}

**return** headA;

}

**private** **int** **length**(ListNode node) {

**int** length = 0;

**while** (node != **null**) {

node = node.next;

length++;

}

**return** length;

}

1. Scan both lists
2. For each list once it reaches the end, continue scanning the other list
3. Once the two runner equal to each other, return the position

Time O(n+m), space O(1)

**public** ListNode **getIntersectionNode**(ListNode headA, ListNode headB) {

**if**( **null**==headA || **null**==headB )

**return** **null**;

ListNode curA = headA, curB = headB;

**while**( curA!=curB){

curA = curA==**null**?headB:curA.next;

curB = curB==**null**?headA:curB.next;

}

**return** curA;

}

Move cur1 (cur2) forward from headA (headB) and loop back to headB (headA), eventually cur1 and cur2 will meet at the intersection point or nullptr.

ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

ListNode \*cur1 = headA, \*cur2 = headB;

**while**(cur1 != cur2){

cur1 = cur1?cur1->next:headB;

cur2 = cur2?cur2->next:headA;

}

**return** cur1;

}

### Reverse Linked List

[My Submissions](https://leetcode.com/problems/reverse-linked-list/submissions/)

QuestionEditorial Solution

Total Accepted: **101760** Total Submissions: **259181** Difficulty: **Easy**

Reverse a singly linked list.

[click to show more hints.](https://leetcode.com/problems/reverse-linked-list/)

**Hint:**

A linked list can be reversed either iteratively or recursively. Could you implement both?

public static void reverse(ListNode head) //仅反序输出

{

if(head==null) return;

reverse(head.next);

System.out.println("->"+head.val);

}

public ListNode reverseList(ListNode head) { //p是前驱（开始设为null非常巧妙）,q是当前节点，tmp用来保存当前节点的下一个节点

if(head==null) return null;

ListNode p=null,q=head,temp=head;

while(q!=null)

{

temp=q.next;

q.next=p;

p=q;

q=temp;

}

return p;

}

public ListNode reverseList(ListNode head) {

/\* iterative solution \*/

ListNode newHead = null;

**while** (head != null) {

ListNode next = head.next;

head.next = newHead;

newHead = head;

head = next;

}

**return** newHead;

}

public ListNode reverseList(ListNode head) {

/\* recursive solution \*/

**return** reverseListInt(head, null);

}

private ListNode reverseListInt(ListNode head, ListNode newHead) {

**if** (head == null)

**return** newHead;

ListNode next = head.next;

head.next = newHead;

**return** reverseListInt(next, head);

}

### 中级

### Odd Even Linked List

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/odd-even-linked-list/submissions/)

* Total Accepted: **46065**
* Total Submissions: **112860**
* Difficulty: **Medium**

Given a singly linked list, group all odd nodes together followed by the even 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 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 ...

**Simple O(N) time, O(1), space Java solution.**

public **class** **Solution** {

public ListNode oddEvenList(ListNode head) {

**if** (head != null) {

ListNode odd = head, even = head.**next**, evenHead = even;

**while** (even != null && even.**next** != null) {

odd.**next** = odd.**next**.**next**;

even.**next** = even.**next**.**next**;

odd = odd.**next**;

even = even.**next**;

}

odd.**next** = evenHead;

}

**return** head;

}}

public ListNode oddEvenList(ListNode head) {

**if**(head==null||head.**next**==null) **return** head;

ListNode odd=head,ehead=head.**next**,even=ehead;

**while**(even!=null&&even.**next**!=null){

odd.**next**=even.**next**;

odd=odd.**next**;

even.**next**=odd.**next**;

even=even.**next**;

}

odd.**next**=ehead;

**return** head;

}

这个写法好

分别构造奇数点、偶数点的链表，再连接两个链表。

even结点后面如果还有结点，则还需要拼接。

We just need to form a linked list of all odd nodes(X) and another linked list of all even nodes(Y). Afterwards, we link Y to the end of X, and return the head of X.

public ListNode oddEvenList(ListNode head) {

**if**(head == null || head.**next** == null){

**return** head;

}

ListNode odd = head;

ListNode even = head.**next**;

ListNode even\_head = head.**next**;

**while**(even != null && even.**next** != null){

odd.**next** = odd.**next**.**next**;

even.**next** = even.**next**.**next**;

odd = odd.**next**;

even = even.**next**;

}

odd.**next** = even\_head;

**return** head;

}

### Add Two Numbers

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/add-two-numbers/submissions/)

* Total Accepted: **189436**
* Total Submissions: **751750**
* Difficulty: **Medium**

You are given two linked lists representing two non-negative numbers. 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.

**Input:** (2 -> 4 -> 3) + (5 -> 6 -> 4)  
**Output:** 7 -> 0 -> 8

我的做法，同书

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

return add(l1, l2, 0);

}

public ListNode add(ListNode l1, ListNode l2, int carry)

{

if(l1==null && l2==null && carry==0) return null;

int sum = carry;

if(l1!=null) sum += l1.val;

if(l2!=null) sum += l2.val;

ListNode node = new ListNode(sum % 10);

ListNode n1 = l1==null? null: l1.next;

ListNode n2 = l2==null? null: l2.next;

node.next = add(n1, n2, sum/10);

return node;

}

**public** **class** **Solution** {

**public** ListNode **addTwoNumbers**(ListNode l1, ListNode l2) {

ListNode c1 = l1;

ListNode c2 = l2;

ListNode sentinel = **new** ListNode(0);

ListNode d = sentinel;

**int** sum = 0;

**while** (c1 != **null** || c2 != **null**) {

sum /= 10;

**if** (c1 != **null**) {

sum += c1.val;

c1 = c1.next;

}

**if** (c2 != **null**) {

sum += c2.val;

c2 = c2.next;

}

d.next = **new** ListNode(sum % 10);

d = d.next;

}

**if** (sum / 10 == 1)

d.next = **new** ListNode(1);

**return** sentinel.next;

}

}

ListNode \*addTwoNumbers(ListNode \*l1, ListNode \*l2) {

ListNode preHead(0), \*p = &preHead;

int extra = 0;

**while** (l1 || l2 || extra) {

int sum = (l1 ? l1->val : 0) + (l2 ? l2->val : 0) + extra;

extra = sum / 10;

p->next = **new** ListNode(sum % 10);

p = p->next;

l1 = l1 ? l1->next : l1;

l2 = l2 ? l2->next : l2;

}

**return** preHead.next;

}

这个写法不错，迭代

Two things to make the code simple:

1. Whenever one of the two *ListNode* is null, replace it with 0.
2. Keep the while loop going when at least one of the three conditions is met.

Let me know if there is something wrong. Thanks.

**public** **class** **Solution** {

**public** ListNode **addTwoNumbers**(ListNode l1, ListNode l2) {

ListNode prev = **new** ListNode(0);

ListNode head = prev;

**int** carry = 0;

**while** (l1 != **null** || l2 != **null** || carry != 0) {

ListNode cur = **new** ListNode(0);

**int** sum = ((l2 == **null**) ? 0 : l2.val) + ((l1 == **null**) ? 0 : l1.val) + carry;

cur.val = sum % 10;

carry = sum / 10;

prev.next = cur;

prev = cur;

l1 = (l1 == **null**) ? l1 : l1.next;

l2 = (l2 == **null**) ? l2 : l2.next;

}

**return** head.next;

}

}

### Swap Nodes in Pairs

[My Submissions](https://leetcode.com/problems/swap-nodes-in-pairs/submissions/)

Question

Total Accepted: **87062** Total Submissions: **249602** Difficulty: **Medium**

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.

我的做法，递归

public ListNode swapPairs(ListNode head) {

if(head == null) return null;

if(head.next == null) return head;

ListNode tmp = head.next.next;

ListNode dummy = new ListNode(0);

dummy.next = head.next;

head.next.next = head;

head.next = swapPairs(tmp);

return dummy.next;

}

这个方法不错，不需要使用dummy

The recursive stack uses O(n) space

public **class** **Solution** {

public ListNode swapPairs(ListNode head) {

**if** ((head == null)||(head.**next** == null))

**return** head;

ListNode n = head.**next**;

head.**next** = swapPairs(head.**next**.**next**);

n.**next** = head;

**return** n;

}

}

这个方法也不错，迭代使用了dummy

cur指针始终指向需要插入元素的链表的最后

public ListNode swapPairs(ListNode head) {

ListNode dummy = new ListNode(0);

dummy.**next** = head;

ListNode current = dummy;

**while** (current.**next** != null && current.**next**.**next** != null) {

ListNode first = current.**next**;

ListNode second = current.**next**.**next**;

first.**next** = second.**next**;

current.**next** = second;

current.**next**.**next** = first;

current = current.**next**.**next**;

}

**return** dummy.**next**;

}

### Reverse Nodes in k-Group

[My Submissions](https://leetcode.com/problems/reverse-nodes-in-k-group/submissions/)

Question

Total Accepted: **51964** Total Submissions: **194553** Difficulty: **Hard**

Given a linked list, reverse the nodes of a linked list *k* at a time and return its modified 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

**public** **class** **Solution** {

**public** ListNode reverseKGroup(ListNode head, **int** k) {

**if** (head==**null**||head.next==**null**||k<2) **return** head;

ListNode dummy = **new** ListNode(0);

dummy.next = head;

ListNode tail = dummy, prev = dummy,temp;

**int** count;

**while**(**true**){

count =k;

**while**(count>0&&tail!=**null**){

count--;

tail=tail.next;

}

**if** (tail==**null**) **break**;//Has reached the end

head=prev.next;//for next cycle

// prev-->temp-->...--->....--->tail-->....

// Delete @temp and insert to the next position of @tail

// prev-->...-->...-->tail-->head-->...

// Assign @temp to the next node of @prev

// prev-->temp-->...-->tail-->...-->...

// Keep doing until @tail is the next node of @prev

**while**(prev.next!=tail){

temp=prev.next;//Assign

prev.next=temp.next;//Delete

temp.next=tail.next;

tail.next=temp;//Insert

}

tail=head;

prev=head;

}

**return** dummy.next;

}

}

有一个pre指针，保持不动（在一轮反转中），始终是作为翻转部分的前一个节点。

tail指针始终指向一个节点（作为反转后的部分的链表的头节点）

temp指针，每次将temp指针，先从pre指针后面删除，然后插入到tail指针后面（实现反转）

head指针用来控制，作为下一轮开始的前一个指针。

### Reverse Linked List II

[My Submissions](https://leetcode.com/problems/reverse-linked-list-ii/submissions/)

Question

Total Accepted: **67377** Total Submissions: **244212** Difficulty: **Medium**

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.

参考上一题，得出我的做法

public ListNode reverseBetween(ListNode head, int m, int n) {

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode cur = dummy, prev = dummy;

for(int i=0; i<m; i++)

{

prev = cur;

cur = cur.next;

}

ListNode start = cur;

for(int i=m; i<n; i++)

{

cur = cur.next;

}

ListNode tail = cur;

System.out.println(start.val +" "+tail.val);

while(prev.next!=tail)

{

ListNode tmp = start.next;

prev.next = tmp;

start.next = tail.next;

tail.next = start;

start = tmp;

}

return dummy.next;

}

这个写法不错，使用dummy指针、pre（指向待反转链表的前一个结点）、start（待反转链表的第一个结点）、then（start的下一个结点）

使用n-m来end loop

Simply just reverse the list along the way using 4 pointers: dummy, pre, start, then

**public** ListNode reverseBetween(ListNode head, **int** m, **int** n) {

**if**(head == **null**) **return** **null**;

ListNode dummy = **new** ListNode(0); // create a dummy node to mark the head of this list

dummy.next = head;

ListNode pre = dummy; // make a pointer pre as a marker for the node before reversing

**for**(**int** i = 0; i<m-1; i++) pre = pre.next;

ListNode start = pre.next; // a pointer to the beginning of a sub-list that will be reversed

ListNode then = start.next; // a pointer to a node that will be reversed

// 1 - 2 -3 - 4 - 5 ; m=2; n =4 ---> pre = 1, start = 2, then = 3

// dummy-> 1 -> 2 -> 3 -> 4 -> 5

//删去then；将then添加到pre后；更新then

**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, then = 4

// second reversing: dummy->1 - 4 - 3 - 2 - 5; pre = 1, start = 2, then = 5 (finish)

**return** dummy.next;

}

The basic idea is to build a sub-list when we hit Node m by adding the subsequent nodes to the head of the sub-list one by one until we hit Node n. Then connect the nodes before Node m, the sub-list and the nodes following Node n.

public ListNode reverseBetween(ListNode head, int m, int n) {

ListNode dummyhead = new ListNode(0);

dummyhead.next = head;

ListNode sublisthead = new ListNode(0);

ListNode sublisttail = new ListNode(0);

int count = 1;

ListNode pre\_cur = dummyhead, cur = head;

**while**(count <=n){

ListNode temp = cur.next;

**if** (count < m)

pre\_cur = cur;

**else** **if** (count == m){

sublisttail = cur;

sublisthead.next = cur;

}**else** **if** (count > m){

cur.next = sublisthead.next;

sublisthead.next = cur;

}

cur = temp;

++count;

}

pre\_cur.next = sublisthead.next;

sublisttail.next = cur;

**return** dummyhead.next;

}

### Reorder List

[My Submissions](https://leetcode.com/problems/reorder-list/submissions/)

Question

Total Accepted: **61505** Total Submissions: **273008** Difficulty: **Medium**

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}.

This question is a combination of **Reverse a linked list I & II**. It should be pretty straight forward to do it in 3 steps :)

**public** **void** reorderList(ListNode head) {

**if**(head==**null**||head.next==**null**) **return**;

//Find the middle of the list

ListNode p1=head;

ListNode p2=head;

**while**(p2.next!=**null**&&p2.next.next!=**null**){

p1=p1.next;

p2=p2.next.next;

}

//Reverse the half after middle 1->2->3->4->5->6 to 1->2->3->6->5->4

ListNode preMiddle=p1;

ListNode preCurrent=p1.next;

**while**(preCurrent.next!=**null**){

ListNode current=preCurrent.next;

preCurrent.next=current.next;

current.next=preMiddle.next;

preMiddle.next=current;

}

//Start reorder one by one 1->2->3->6->5->4 to 1->6->2->5->3->4

p1=head;

p2=preMiddle.next;

**while**(p1!=preMiddle){

preMiddle.next=p2.next;

p2.next=p1.next;

p1.next=p2;

p1=p2.next;

p2=preMiddle.next;

}

}

My solution is quite similar to @zchen

First, find the second half of List, (slow and fast pointer) reverse it, and make the end of first half point to null

Second, insert second half node into first half of List

For Example: If we have 1 - 2 - 3 - 4 - 5, reverse 4 - 5 to 5 - 4, make the first half end to null

Now We have 1 - 2 - 3 and 5 - 4, Insert each of second list node into the first list between current and next node which gives 1 - 5 - 2 - 4 - 3

**public** **void** reorderList(ListNode head) {

// IMPORTANT: Please reset any member data you declared, as

// the same Solution instance will be reused for each test case.

**if** (head == **null** || head.next == **null**) **return**;

ListNode fast = head;

ListNode slow = head;

**while**(fast != **null** && fast.next != **null**) {

fast = fast.next.next;

slow = slow.next;

}

ListNode reverseHead = slow.next; // find the second half of list

slow.next = **null**; // make first half end point to null

reverseHead = reverse(reverseHead); // reverse second half

ListNode cur = head;

**while**(reverseHead != **null**) { // link together

ListNode tmp = reverseHead.next;

reverseHead.next = cur.next;

cur.next = reverseHead;

cur = cur.next.next;

reverseHead = tmp;

}

}

**private** ListNode reverse(ListNode head) {

**if** (head == **null** || head.next == **null**) **return** head;

ListNode prev = **new** ListNode(0);

prev.next = head;

head = prev;

ListNode cur = prev.next;

**while**(cur.next != **null**) {

ListNode tmp = cur.next;

cur.next = tmp.next;

tmp.next = prev.next;

prev.next = tmp;

}

**return** prev.next;

}

1->2->3->4->5->null

1->2->3->4->…

3<-4<-5 reverse

1->5->2->4->3->null(重要，prev.next=null)

public void reorderList(ListNode head) {

if(head==null || head.next==null) return;

//首先，反转后半段

ListNode p=head,q=head;

while(q!=null && q.next!=null)

{

//System.out.println(p.val+" "+q.val);

p=p.next;

q=q.next.next;

}

ListNode pre=p;

p=p.next;

int cnt=0;

while(p!=null)

{

ListNode next=p.next;

p.next=pre;

pre=p;

p=next;

cnt++;

}

//reorder

ListNode start=head;

while(cnt>0)

{

ListNode next1=start.next;

ListNode next2=pre.next;

start.next=pre;

pre.next=next1;

start=next1;

pre=next2;

cnt--;

}

pre.next=null;

}

从中间节点的下一个节点开始反转。（重要！！！不然不好处理指针为null的问题）

或者，设dummy节点。

### Remove Duplicates from Sorted List

[My Submissions](https://leetcode.com/problems/remove-duplicates-from-sorted-list/submissions/)

Question

Total Accepted: **108105** Total Submissions: **296373** Difficulty: **Easy**

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.

类似题目**Remove Linked List Elements**

This solution is inspired by renzid https://leetcode.com/discuss/33043/3-line-recursive-solution

public ListNode deleteDuplicates(ListNode head) {

**if**(head == null || head.next == null)**return** head;

head.next = deleteDuplicates(head.next);

**return** head.val == head.next.val ? head.next : head;

}

这个写法不错

**public** **class** **Solution** {

**public** ListNode deleteDuplicates(ListNode head) {

ListNode **list** = head;

**while**(**list** != **null**) {

**if** (**list**.next == **null**) {

**break**;

}

**if** (**list**.val == **list**.next.val) {

**list**.next = **list**.next.next;

} **else** {

**list** = **list**.next;

}

}

**return** head;

}

}

//cur指针不动，直到其下一个指针为不同的数，基本同上一种写法

public **class** **Solution** {

public ListNode deleteDuplicates(ListNode head) {

**if** (head == null) **return** head;

ListNode cur = head;

**while**(cur.next != null) {

**if** (cur.val == cur.next.val) {

cur.next = cur.next.next;

}

**else** cur = cur.next;

}

**return** head;

}

}

### Remove Duplicates from Sorted List II

[My Submissions](https://leetcode.com/problems/remove-duplicates-from-sorted-list-ii/submissions/)

Question

Total Accepted: **68459** Total Submissions: **257015** Difficulty: **Medium**

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.

这个写法不错，使用pre指针（相当于tail指针，始终指在链表的尾部。当确定cur指针是唯一值时，pre移动）

public ListNode deleteDuplicates(ListNode head) {

**if**(head==null) **return** null;

ListNode FakeHead=new ListNode(0);

FakeHead.next=head;

ListNode pre=FakeHead;

ListNode cur=head;

**while**(cur!=null){

**while**(cur.next!=null&&cur.val==cur.next.val){

cur=cur.next;

}

**if**(pre.next==cur){ //巧妙。如果pre的next是cur，说明cur指针没有经历上面的移动，则表示其是唯一值。保留该cur。

pre=pre.next;

}

**else**{

pre.next=cur.next;

}

cur=cur.next;

}

**return** FakeHead.next;

}

//我用的方法是标记了是否为重复值,这里用了pre指针

public static ListNode deleteDuplicates(ListNode head) {

if(head==null) return null;

ListNode dummy=new ListNode(0);

dummy.next=head;

ListNode cur=dummy;

boolean flag=false;

while(cur!=null)

{

ListNode next=cur.next;

flag=false;

while(next!=null)

{

next=next.next;

if(next==null) break;

if(next.val==cur.next.val)

{

flag=true;

}

else break;

}

if(flag)

{

cur.next=next;

}

else cur=cur.next;

}

return dummy.next;

}

public ListNode deleteDuplicates(ListNode head) {

**if** (head == null) **return** null;

**if** (head.next != null && head.val == head.next.val) {

**while** (head.next != null && head.val == head.next.val) {

head = head.next;

}

**return** deleteDuplicates(head.next);

} **else** {

head.next = deleteDuplicates(head.next);

}

**return** head;

}

if current node is not unique, return deleteDuplicates with head.next. If current node is unique, link it to the result of next list made by recursive call. Any improvement?

### Rotate List

[My Submissions](https://leetcode.com/problems/rotate-list/submissions/)

Question

Total Accepted: **64783** Total Submissions: **285167** Difficulty: **Medium**

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.

这个写法也不错

Since n may be a large number compared to the length of list. So we need to know the length of linked list.After that, move the list after the (l-n%l )th node to the front to finish the rotation.

Ex: {1,2,3} k=2 Move the list after the 1st node to the front

Ex: {1,2,3} k=5, In this case Move the list after (3-5%3=1)st node to the front.

So the code has three parts.

1) Get the length

2) Move to the (l-n%l)th node

3)Do the rotation

**public** ListNode rotateRight(ListNode head, **int** n) {

**if** (head==**null**||head.next==**null**) **return** head;

ListNode dummy=**new** ListNode(0);

dummy.next=head;

ListNode fast=dummy,slow=dummy;

**int** i;

**for** (i=0;fast.next!=**null**;i++)//Get the total length

fast=fast.next;

**for** (**int** j=i-n%i;j>0;j--) //Get the i-n%i th node

slow=slow.next;

fast.next=dummy.next; //Do the rotation

dummy.next=slow.next;

slow.next=**null**;

**return** dummy.next;

}

Let's start with an example.

Given [0,1,2], rotate 1 steps to the right -> [2,0,1].

Given [0,1,2], rotate 2 steps to the right -> [1,2,0].

Given [0,1,2], rotate 3 steps to the right -> [0,1,2].

Given [0,1,2], rotate 4 steps to the right -> [2,0,1].

So, no matter how big K, the number of steps is, the result is always the same as rotating K % n steps to the right.

The basic idea is to link the tail of the list with the head, make it a cycle. Then count to the rotate point and cut it.

选这个(有很多细节需要注意)

* 1. 移动到链表结尾，同时计算链表的长度。
  2. 将链表结尾与头相连。
  3. 根据k移动head指针指定步数。然后将环断开。
  4. head.next即为新的链表头。

**if** (head == null)

**return** head;

ListNode copyHead = head;

**int** len = 1;

**while** (copyHead.**next** != null) {

copyHead = copyHead.**next**;

len++;

}

copyHead.**next** = head;

**for** (**int** i = len - k % len; i > 1; i--)

head = head.**next**;

copyHead = head.**next**;//巧妙保存链表的开头

head.**next** = null;

**return** copyHead;

}

计算链表的长度，然后将尾节点与头结点相连。并且头节点走（len-K%len）步来，找到分割节点。（注意这里可以通过这个保存真正的头结点，最后将此断开）

注意这里几个初值的设定

我的做法，不太好，有特例需要处理。

public ListNode rotateRight(ListNode head, int k) {

if(head==null || head.next==null) return head;

ListNode slow=head, fast=head, prev=head;

k=k % len(head);

if(k==0) return head;

while(k-1>0)

{

fast=fast.next;

k--;

}

while(fast.next!=null)

{

prev=slow;

slow=slow.next;

fast=fast.next;

}

prev.next=null;

fast.next=head;

return slow;

}

public int len(ListNode head)

{

ListNode node=head;

int cnt=0;

while(node!=null)

{

node=node.next;

cnt++;

}

return cnt;

}

### Insertion Sort List

[My Submissions](https://leetcode.com/problems/insertion-sort-list/submissions/)

Question

Total Accepted: **67204** Total Submissions: **231630** Difficulty: **Medium**

Sort a linked list using insertion sort.

pre指针每次初始化为dummy。从pre.next开始遍历，找可以插入的位置。

**public** ListNode insertionSortList(ListNode head) {

**if**( head == **null** ){

**return** head;

}

ListNode helper = **new** ListNode(0); //new starter of the sorted list

ListNode cur = head; //the node will be inserted

ListNode pre = helper; //insert node between pre and pre.next

ListNode next = **null**; //the next node will be inserted

//not the end of input list

**while**( cur != **null** ){

next = cur.next;

//find the right place to insert

**while**( pre.next != **null** && pre.next.val < cur.val ){

pre = pre.next;

}

//insert between pre and pre.next

cur.next = pre.next;

pre.next = cur;

pre = helper;

cur = next;

}

**return** helper.next;

}

我的解法：一个指针遍历链表。使用dummy，dummy.next=null，然后依次插入dummy后面。（类似上面）

One of the quotes is

*For God's sake, don't try sorting a linked list during the interview*

http://steve-yegge.blogspot.nl/2008/03/get-that-job-at-google.html

So it might be better to actually copy the values into an array and sort them there.

### Sort List

[My Submissions](https://leetcode.com/problems/sort-list/submissions/)

Question

Total Accepted: **67270** Total Submissions: **274374** Difficulty: **Medium**

Sort a linked list in *O*(*n* log *n*) time using constant space complexity.

窗体顶端

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

**public** **class** **Solution** {

**public** ListNode sortList(ListNode head) {

**if** (head == **null** || head.next == **null**)

**return** head;

ListNode f = head.next.next;

ListNode p = head;

**while** (f != **null** && f.next != **null**) {

p = p.next;

f = f.next.next;

}

ListNode h2 = sortList(p.next);

p.next = **null**;

**return** merge(sortList(head), h2);

}

**public** ListNode merge(ListNode h1, ListNode h2) {

ListNode hn = **new** ListNode(Integer.MIN\_VALUE);

ListNode c = hn;

**while** (h1 != **null** && h2 != **null**) {

**if** (h1.val < h2.val) {

c.next = h1;

h1 = h1.next;

}

**else** {

c.next = h2;

h2 = h2.next;

}

c = c.next;

}

**if** (h1 != **null**)

c.next = h1;

**if** (h2 != **null**)

c.next = h2;

**return** hn.next;

}

}

https://www.gravatar.com/avatar/b3aec34fe84f55d1dddb2c261fbacde3?s=40 [asked](https://leetcode.com/discuss/1709/have-pretty-mergesort-method-anyone-speed-reduce-memory-usage) **Dec 30, 2013** in [**Sort List**](https://leetcode.com/discuss/oj/sort-list) by [**potpie**](https://leetcode.com/discuss/user/potpie) (11,300 points)

u r so creative.

https://www.gravatar.com/avatar/5534a813c5662b5ff1f86e76f3e54416?s=20 [commented](https://leetcode.com/discuss/1709/have-pretty-mergesort-method-anyone-speed-reduce-memory-usage?show=4365#c4365) Apr 10, 2014 by [**stephenluu**](https://leetcode.com/discuss/user/stephenluu)



good solution, easy to understand

https://www.gravatar.com/avatar/48874b863dc2be17d45d1b0106fb5a29?s=20 [commented](https://leetcode.com/discuss/1709/have-pretty-mergesort-method-anyone-speed-reduce-memory-usage?show=32487#c32487) Apr 19, 2015 by [**zrythpzhl**](https://leetcode.com/discuss/user/zrythpzhl)



ListNode f = head.next.next;

ListNode p = head;

Why is it that I get a stackoverflow when I initialize both f and p to head?

https://www.gravatar.com/avatar/129f19e04641dc0a493732fdf482d787?s=20 [commented](https://leetcode.com/discuss/1709/have-pretty-mergesort-method-anyone-speed-reduce-memory-usage?show=57271#c57271) Sep 10, 2015 by [**kevinhsu**](https://leetcode.com/discuss/user/kevinhsu)



need to delete hn to avoid memory leak

https://www.gravatar.com/avatar/2cab043b819b9c7e0878179955620be6?s=20 [commented](https://leetcode.com/discuss/1709/have-pretty-mergesort-method-anyone-speed-reduce-memory-usage?show=66804#c66804) Oct 29, 2015 by [**ZekunWang**](https://leetcode.com/discuss/user/ZekunWang)



@kevinhsu same problem here, don't know the difference between

ListNode f = head.next.next;

and

ListNode f = head;

Pls let me know you figured it out.

https://www.gravatar.com/avatar/d015a0a29cf81b5828c5147cb10fdecc?s=20 [commented](https://leetcode.com/discuss/1709/have-pretty-mergesort-method-anyone-speed-reduce-memory-usage?show=75273#c75273) Dec 20, 2015 by [**hankok**](https://leetcode.com/discuss/user/hankok)



I would say it's probably the case where you have only 2 inputs, say 2->1-> null; After the while loop, you end up getting p points to 1. Then you sort p.next (null), which won't do anything. After that you sort 2->1->null again. Stack overflow

https://www.gravatar.com/avatar/b758ed7ee3cd7174eb98806cbc10422f?s=20 [commented](https://leetcode.com/discuss/1709/have-pretty-mergesort-method-anyone-speed-reduce-memory-usage?show=81197#c81197) Jan 19 by [**stupidbird911**](https://leetcode.com/discuss/user/stupidbird911)



窗体底端

窗体顶端



**+2**votes

窗体底端

窗体顶端

Space complexity of the solution is not O(1), but O(nlogn). However, it's very beautiful.

https://www.gravatar.com/avatar/78a436bb9b8e4c7dd21cd6f9700cbb21?s=40 [answered](https://leetcode.com/discuss/1709/have-pretty-mergesort-method-anyone-speed-reduce-memory-usage?show=66354#a66354) **Oct 27, 2015** by [**whuacm**](https://leetcode.com/discuss/user/whuacm) (240 points)

I think space complexity is O(logn), rather than O(nlogn)

https://www.gravatar.com/avatar/57ec65beb20184ac490158ae334a60ff?s=20 [commented](https://leetcode.com/discuss/1709/have-pretty-mergesort-method-anyone-speed-reduce-memory-usage?show=90778#c90778) Mar 6 by [**biwuxia**](https://leetcode.com/discuss/user/biwuxia)



窗体底端

窗体顶端



**+2**votes

窗体底端

窗体顶端

I suggest that by mentioning "Constant space" complexity, we should not use Recursion, which uses stack for each recursively called function.

Thus you can simply use Iteration for merge sort.

https://www.gravatar.com/avatar/04fec545b3e2cb5e08d381f117b4c197?s=40 [answered](https://leetcode.com/discuss/1709/have-pretty-mergesort-method-anyone-speed-reduce-memory-usage?show=85014#a85014) **Feb 5** by [**D\_shaw**](https://leetcode.com/discuss/user/D_shaw) (380 points)

窗体底端

窗体顶端



**0**votes

窗体底端

窗体顶端

Nice solution.

I guess after the loop you can use this to make it even shorter

c.next = (h1 !=null) ? h1 : h2;

Here is my AC solution written in javascript

var sortList = function(head) {

**if** (!head || !head.next) **return** head;

var fast = head.next.next;

var slow = head;

**while** (fast && fast.next) {

fast = fast.next.next;

slow = slow.next;

}

var head1 = sortList(slow.next);

slow.next = null;

var head2 = sortList(head);

**return** mergeList(head1, head2);

};

var mergeList = function(h1, h2) {

var dummy = new ListNode(null);

var tail = dummy;

**while** (h1 && h2) {

**if** (h1.val <= h2.val) {

tail = tail.next = h1;

h1 = h1.next;

} **else** {

tail = tail.next = h2;

h2 = h2.next;

}

}

tail.next = h1 ? h1 : h2;

**return** dummy.next;

};

//我的方法是，归并。当一个元素或者空时直接返回。

用快慢指针将链表分成两个。递归排序两个链表，最后返回合并。（合并函数用的是迭代法，用递归会stack overflow）🡪和下面一样

public **class** **Solution** {

public ListNode sortList(ListNode head) {

**if** (head == null || head.next == null)

**return** head;

// step 1. cut the list to two halves

ListNode prev = null, slow = head, fast = head;

**while** (fast != null && fast.next != null) {

prev = slow;

slow = slow.next;

fast = fast.next.next;

}

prev.next = null;

// 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 l = new ListNode(0), p = l;

**while** (l1 != null && l2 != null) {

**if** (l1.val < l2.val) {

p.next = l1;

l1 = l1.next;

} **else** {

p.next = l2;

l2 = l2.next;

}

p = p.next;

}

**if** (l1 != null)

p.next = l1;

**if** (l2 != null)

p.next = l2;

**return** l.next;

}

}

### Partition List

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/partition-list/submissions/)

* Total Accepted: **79019**
* Total Submissions: **255605**
* Difficulty: **Medium**

Given a linked list and a value *x*, partition it such that all nodes less than *x* come before nodes greater than or equal to *x*.

You should preserve 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.

我的做法，参考书上

public ListNode partition(ListNode head, int x) {

if(head==null) return head;

ListNode dummy1=new ListNode(0);

ListNode dummy2=new ListNode(0);

ListNode tail1=dummy1, tail2=dummy2;

while(head!=null)

{

if(head.val<x)

{

tail1.next=head;

tail1=head;

}

else

{

tail2.next=head;

tail2=head;

}

head=head.next;

}

tail1.next=null;

tail2.next=null; //重要，没有的话会超时

if(dummy1.next==null) return dummy2.next;

tail1.next=dummy2.next;

return dummy1.next;

}

ListNode \*partition(ListNode \*head, int x) {

ListNode node1(0), node2(0);

ListNode \*p1 = &node1, \*p2 = &node2;

**while** (head) {

**if** (head->val < x)

p1 = p1->next = head;

**else**

p2 = p2->next = head;

head = head->next;

}

p2->next = **NULL**;

p1->next = node2.next;

**return** node1.next;

}

这个写法好，比我的简洁

the basic idea is to maintain two queues, the first one stores all nodes with val less than x , and the second queue stores all the rest nodes. Then concat these two queues. Remember to set the tail of second queue a null next, or u will get TLE.

**public** ListNode **partition**(ListNode head, **int** x) {

ListNode dummy1 = **new** ListNode(0), dummy2 = **new** ListNode(0); *//dummy heads of the 1st and 2nd queues*

ListNode curr1 = dummy1, curr2 = dummy2; *//current tails of the two queues;*

**while** (head!=**null**){

**if** (head.val<x) {

curr1.next = head;

curr1 = head;

}**else** {

curr2.next = head;

curr2 = head;

}

head = head.next;

}

curr2.next = **null**; *//important! avoid cycle in linked list. otherwise u will get TLE.*

curr1.next = dummy2.next;

**return** dummy1.next;

}

### Convert Sorted List to Binary Search Tree

[My Submissions](https://leetcode.com/problems/convert-sorted-list-to-binary-search-tree/submissions/)

Question

Total Accepted: **66466** Total Submissions: **220624** Difficulty: **Medium**

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST. 类似书上

我的做法，类似数组的做法

题目要求构造的BST，mid点是中间偏前（并没有！只需要平衡就好）。比如1->2->3->4.按照常规移动指针的方式，中点会指在结点3处。使用dummy则指在结点2处。

如果是奇数点的链表，则两种方法都指在中间结点。

所以这题可以不使用dummy

public TreeNode sortedListToBST(ListNode head) {

if(head==null) return null;

if(head.next==null) return new TreeNode(head.val);

ListNode dummy=new ListNode(0);

dummy.next=head;

ListNode slow=dummy, fast=dummy, prev=dummy;

while(fast!=null && fast.next!=null)

{

prev=slow;

slow=slow.next;

fast=fast.next.next;

}

prev.next=null; // cut the linked list

//slow is the mid

TreeNode node=new TreeNode(slow.val);

node.left=sortedListToBST(dummy.next);

node.right=sortedListToBST(slow.next);

return node;

}

public TreeNode sortedListToBST(ListNode head) {

if(head==null) return null;

if(head.next==null) return new TreeNode(head.val);

ListNode slow=head, fast=head, prev=head;

while(fast!=null && fast.next!=null)

{

prev=slow;

slow=slow.next;

fast=fast.next.next;

}

prev.next=null; // cut the linked list

//slow is the mid

TreeNode node=new TreeNode(slow.val);

node.left=sortedListToBST(head);

node.right=sortedListToBST(slow.next);

return node;

}

**private** ListNode node;

**public** TreeNode sortedListToBST(ListNode head) {

**if**(head == **null**){

**return** **null**;

}

**int** size = 0;

ListNode runner = head;

node = head;

**while**(runner != **null**){

runner = runner.next;

size ++;

}

**return** inorderHelper(0, size - 1);

}

**public** TreeNode inorderHelper(**int** start, **int** end){

**if**(start > end){

**return** **null**;

}

**int** mid = start + (end - start) / 2;

TreeNode left = inorderHelper(start, mid - 1);

TreeNode treenode = **new** TreeNode(node.val);

treenode.left = left;

node = node.next;

TreeNode right = inorderHelper(mid + 1, end);

treenode.right = right;

**return** treenode;

}

count is a function to calculate the size of list.

Key words: inorder traversal.

class Solution {

public:

ListNode \*list;

**int** count(ListNode \*node){

**int** size = 0;

**while** (node) {

++size;

node = node->next;

}

**return** size;

}

TreeNode \*generate(**int** n){

**if** (n == 0)

**return** NULL;

TreeNode \*node = new TreeNode(0);

node->left = generate(n / 2);

node->val = list->val;

list = list->next;

node->right = generate(n - n / 2 - 1);

**return** node;

}

TreeNode \*sortedListToBST(ListNode \*head) {

this->list = head;

**return** generate(count(head));

}

};

Recursive BST construction using slow-fast traversal on linked list

这个写法不错

**public** TreeNode sortedListToBST(ListNode head) {

**if**(head == **null**)

**return** **null**;

ListNode fast = head;

ListNode slow = head;

ListNode prev =**null**;

**while**(fast != **null** && fast.next != **null**)

{

fast = fast.next.next;

prev =slow;

slow=slow.next;

}

TreeNode root = **new** TreeNode(slow.val);

**if**(prev != **null**) //考虑的是只有一个结点的情况

prev.next = **null**;

**else**

head = **null**;

root.left = sortedListToBST(head);

root.right = sortedListToBST(slow.next);

**return** root;

}

Traverse the list to get the middle element and make that the root. left side of the list forms left sub-tree and right side of the middle element forms the right sub-tree.

**class** **Solution** {

**public**:

TreeNode \*sortedListToBST(ListNode \*head)

{

**return** sortedListToBST( head, **NULL** );

}

**private**:

TreeNode \*sortedListToBST(ListNode \*head, ListNode \*tail)

{

**if**( head == tail )

**return** **NULL**;

**if**( head->next == tail ) //

{

TreeNode \*root = **new** TreeNode( head->val );

**return** root;

}

ListNode \*mid = head, \*temp = head;

**while**( temp != tail && temp->next != tail ) // 寻找中间节点

{

mid = mid->next;

temp = temp->next->next;

}

TreeNode \*root = **new** TreeNode( mid->val );

root->left = sortedListToBST( head, mid );

root->right = sortedListToBST( mid->next, tail );

**return** root;

}

};

**class** **Solution** {

**public**:

TreeNode \*sortedListToBST(ListNode \*head)

{

**return** sortedListToBST( head, **NULL** );

}

**private**:

TreeNode \*sortedListToBST(ListNode \*head, ListNode \*tail)

{

**if**( head == tail )

**return** **NULL**;

**if**( head->next == tail ) //

{

TreeNode \*root = **new** TreeNode( head->val );

**return** root;

}

ListNode \*mid = head, \*temp = head;

**while**( temp != tail && temp->next != tail ) // 寻找中间节点

{

mid = mid->next;

temp = temp->next->next;

}

TreeNode \*root = **new** TreeNode( mid->val );

root->left = sortedListToBST( head, mid );

root->right = sortedListToBST( mid->next, tail );

**return** root;

}

};

**public** **class** **Solution** {

**public** TreeNode sortedListToBST(ListNode head) {

**if**(head==**null**)

**return** **null**;

ListNode slow = head;

ListNode fast = head;

ListNode temp=**null**;

//find the mid node

**while**(fast.next!=**null** && fast.next.next!=**null**){

fast = fast.next.next;

temp = slow;

slow = slow.next;

}

**if**(temp!=**null**)

temp.next = **null**; //break the link

**else**

head = **null**;

TreeNode root = **new** TreeNode(slow.val);

root.left = sortedListToBST(head);

root.right = sortedListToBST(slow.next);

**return** root;

}

### 约瑟夫环

约瑟夫问题是一个非常著名的趣题，即由n个人坐成一圈，按顺时针由1开始给他们编号。然后由第一个人开始报数，数到m的人出局。现在需要求的是最后一个出局的人的编号。

给定两个int **n**和**m**，代表游戏的人数。请返回最后一个出局的人的编号。保证n和m小于等于1000。

测试样例：

5 3

返回：4

无论是用链表实现还是用数组实现都有一个共同点：要模拟整个游戏过程，不仅程序写起来比 较烦，而且时间复杂度高达O(nm)，当n，m非常大(例如上百万，上千万)的时候，几乎是没有办法在短时间内出结果的。我们注意到原问题仅仅是要求出最 后的胜利者的序号，而不是要读者模拟整个过程。因此如果要追求效率，就要打破常规，实施一点数学策略。

为了讨论方便，先把问题稍微改变一下，并不影响原意：

问题描述：n个人（编号0~(n-1))，从0开始报数，报到(m-1)的退出，剩下的人继续从0开始报数。求胜利者的编号。

我们知道第一个人(编号一定是m%n-1) 出列之后，剩下的n-1个人组成了一个新的约瑟夫环（以编号为k=m%n的人开始）:  
   k   k+1   k+2   ... n-2, n-1, 0, 1, 2, ... k-2  
并且从k开始报0。

现在我们把他们的编号做一下转换：  
k     --> 0  
k+1 --> 1  
k+2 --> 2  
...  
...  
k-2 --> n-2  
k-1 --> n-1

变换后就完完全全成为了(n-1)个人报数的子问题，假如我们知道这个子问题的解：例如x是最终的胜利者，那么根据上面这个表把这个x变回去不刚好就是n个人情况的解吗？！！变回去的公式很简单，相信大家都可以推出来：x'=(x+k)%n

如何知道(n-1)个人报数的问题的解？对，只要知道(n-2)个人的解就行了。(n-2)个人的解呢？当然是先求(n-3)的情况 ---- 这显然就是一个倒推问题！好了，思路出来了，下面写递推公式：

令f[i]表示i个人玩游戏报m退出最后胜利者的编号，最后的结果自然是f[n]

递推公式  
f[1]=0;  
f[i]=(f[i-1]+m)%i;   (i>1)

有了这个公式，我们要做的就是从1-n顺序算出f[i]的数值，最后结果是f[n]。因为实际生活中编号总是从1开始，我们输出f[n]+1

由于是逐级递推，不需要保存每个f[i]，程序也是异常简单：   
#include <stdio.h>  
int main()  
{  
   int n, m, i, s=0;  
   printf ("N M = "); scanf("%d%d", &n, &m);  
   for (i=2; i<=n; i++) s=(s+m)%i;  
   printf ("The winner is %d\n", s+1);  
}

这个算法的时间复杂度为O(n)，相对于模拟算法已经有了很大的提高。算n，m等于一百万，一千万的情况不是问题了。

## 栈和队列

### Min Stack

[My Submissions](https://leetcode.com/problems/min-stack/submissions/)

QuestionEditorial Solution

Total Accepted: **66860** Total Submissions: **306130** Difficulty: **Easy**

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

* push(x) -- Push element x onto stack.
* pop() -- Removes the element on top of the stack.
* top() -- Get the top element.
* getMin() -- Retrieve the minimum element in the stack.

The question is ask to construct One stack. So I am using one stack.

The idea is to store the gap between the min value and the current value;

The problem for my solution is the cast. I have no idea to avoid the cast. Since the possible gap between the current value and the min value could be Integer.MAXVALUE-Integer.MINVALUE;

**public** **class** MinStack {

**long** min;

Stack<Long> **stack**;

**public** MinStack(){

**stack**=**new** Stack<>();

}

**public** **void** push(**int** x) {

**if** (**stack**.isEmpty()){

**stack**.push(0L);

min=x;

}**else**{

**stack**.push(x-min);//Could be negative if min value needs to change

**if** (x<min) min=x;

}

}

**public** **void** pop() {

**if** (**stack**.isEmpty()) **return**;

**long** pop=**stack**.pop();

**if** (pop<0) min=min-pop;//If negative, increase the min value

}

**public** **int** top() {

**long** top=**stack**.peek();

**if** (top>0){

**return** (**int**)(top+min);

}**else**{

**return** (**int**)(min);

}

}

**public** **int** getMin() {

**return** (**int**)min;

}

}

My idea is directly store the current min value in the stack, below is my code. I think both methods use the same memory space, but my method doesn't need any calculation.

**class** **MinStack**

{

**static** **class** **Element**

{

**final** **int** value;

**final** **int** min;

Element(**final** **int** value, **final** **int** min)

{

**this**.value = value;

**this**.min = min;

}

}

**final** Stack<Element> stack = **new** Stack<>();

**public** **void** push(**int** x) {

**final** **int** min = (stack.empty()) ? x : Math.min(stack.peek().min, x);

stack.push(**new** Element(x, min));

}

**public** **void** pop()

{

stack.pop();

}

**public** **int** top()

{

**return** stack.peek().value;

}

**public** **int** getMin()

{

**return** stack.peek().min;

}

}

**class** MinStack {

**int** min=Integer.MAX\_VALUE;

Stack<Integer> **stack** = **new** Stack<Integer>();

**public** **void** push(**int** x) {

// only push the old minimum value when the current

// minimum value changes after pushing the new value x

**if**(x <= min){

**stack**.push(min);

min=x;

}

**stack**.push(x);

}

**public** **void** pop() {

// if pop operation could result in the changing of the current minimum value,

// pop twice and change the current minimum value to the last minimum value.

**if**(**stack**.peek()==min) {

**stack**.pop();

min=**stack**.peek();

**stack**.pop();

}**else**{

**stack**.pop();

}

**if**(**stack**.empty()){

min=Integer.MAX\_VALUE;

}

}

**public** **int** top() {

**return** **stack**.peek();

}

**public** **int** getMin() {

**return** min;

}

}

### Valid Parentheses

[My Submissions](https://leetcode.com/problems/valid-parentheses/submissions/)

QuestionEditorial Solution

Total Accepted: **101369** Total Submissions: **348349** Difficulty: **Easy**

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.

//和我的方法一样

**public** **class** Solution {

**public** boolean isValid(String s) {

Stack<Character> **stack** = **new** Stack<Character>();

// Iterate through string until empty

**for**(**int** i = 0; i<s.length(); i++) {

// Push any open parentheses onto stack

**if**(s.charAt(i) == '(' || s.charAt(i) == '[' || s.charAt(i) == '{')

**stack**.push(s.charAt(i));

// Check stack for corresponding closing parentheses, false if not valid

**else** **if**(s.charAt(i) == ')' && !**stack**.empty() && **stack**.peek() == '(')

**stack**.pop();

**else** **if**(s.charAt(i) == ']' && !**stack**.empty() && **stack**.peek() == '[')

**stack**.pop();

**else** **if**(s.charAt(i) == '}' && !**stack**.empty() && **stack**.peek() == '{')

**stack**.pop();

**else**

**return** **false**;

}

// return true if no open parentheses left in stack

**return** **stack**.empty();

}

}

public **class** **Solution** {

public boolean isValid(String s) {

int length;

**do** {

length = s.length();

s = s.replace("()", "").replace("{}", "").replace("[]", "");

} **while**(length != s.length());

**return** s.length() == 0;

}

}

In this solution you essentially can remove parentheses that you know are valid until the string is empty. If the string is not empty, that means that the parentheses were malformed.

The code is easy to understand and the idea is good. However, it might not be considered an efficient algorithm, compared with the stack method. Consider the worst case where the string is as follows:

[([([([([()])])])])]

where only one pair of parenthesis can be detected and removed each round, would the running time be O(n^2)?

**Nooooo! Stop upvoting it....**

In an algorithms coding community who appreciates constructing 3\*n/2 separate Stringobjects to validate a single String? Not to mention that each replace call uses regex:

* compiles a Pattern
* creates a Matcher
* builds the replacement using a StringBuffer ("synchronized StringBuilder")

Don't get me wrong, regex is awesome, I love it, and use it very often for parsing complex structures. The difference is that for some tasks there are faster ways. Just because a code has less characters it doesn't mean it's better. My outburst is highly correlated to using a regex without knowing it or knowingly in a loop that isn't pre-compiled. Consider the following code:

**public** **class** **Solution** {

**private** **static** **final** Pattern PARENS = Pattern.compile("\\(\\)|\\[\\]|\\{\\}");

**public** **boolean** isValid(String s) {

**int** length;

do {

length = s.length();

s = PARENS.matcher(s).replaceAll("");

} **while** (length != s.length());

**return** s.isEmpty();

}

}

same result, 1 regex = 1 compile overall, 1 extra string created per iteration; still usingStringBuffer though. I would be happier with this, but there's still a way to do it by iterating over the characters of the input once.

### Mini Parser

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/mini-parser/submissions/)

* Total Accepted: **625**
* Total Submissions: **2252**
* Difficulty: **Medium**

Given a nested list of integers represented as a string, implement a parser to deserialize it.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

**Note:** You may assume that the string is well-formed:

* String is non-empty.
* String does not contain white spaces.
* String contains only digits 0-9, [, - ,, ].

**Example 1:**

Given s = "324",

You should return a NestedInteger object which contains a single integer 324.

**Example 2:**

Given s = "[123,[456,[789]]]",

Return a NestedInteger object containing a nested list with 2 elements:

1. An integer containing value 123.

2. A nested list containing two elements:

i. An integer containing value 456.

ii. A nested list with one element:

a. An integer containing value 789.

## 树

### EASY

### Path Sum

[My Submissions](https://leetcode.com/problems/path-sum/submissions/)

QuestionEditorial Solution

Total Accepted: **97497** Total Submissions: **313025** Difficulty: **Easy**

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.

The basic idea is to subtract the value of current node from sum until it reaches a leaf node and the subtraction equals 0, then we know that we got a hit. Otherwise the subtraction at the end could not be 0.

我的做法用了两个函数

**public** **class** **Solution** {

**public** **boolean** hasPathSum(TreeNode root, **int** sum) {

**if**(root == **null**) **return** **false**;

**if**(root.left == **null** && root.right == **null** && sum - root.val == 0) **return** **true**; //是叶子，且值符合要求

**return** hasPathSum(root.left, sum - root.val) || hasPathSum(root.right, sum - root.val);

}

}

### Balanced Binary Tree

[My Submissions](https://leetcode.com/problems/balanced-binary-tree/submissions/)

QuestionEditorial Solution

Total Accepted: **104917** Total Submissions: **310361** Difficulty: **Easy**

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.

同书本做法

Java solution based on height, check left and right node in every recursion to avoid further useless search

**public** **boolean** isBalanced(TreeNode root) {

**if**(root==**null**){

**return** **true**;

}

**return** height(root)!=-1;

}

**public** **int** height(TreeNode node){

**if**(node==**null**){

**return** 0;

}

**int** lH=height(node.left);

**if**(lH==-1){

**return** -1;

}

**int** rH=height(node.right);

**if**(rH==-1){

**return** -1;

}

**if**(lH-rH<-1 || lH-rH>1){

**return** -1;

}

**return** Math.max(lH,rH)+1;

}

### Maximum Depth of Binary Tree

[My Submissions](https://leetcode.com/problems/maximum-depth-of-binary-tree/submissions/)

QuestionEditorial Solution

Total Accepted: **134653** Total Submissions: **282920** Difficulty: **Easy**

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.

if the node does not exist, simply return 0. Otherwise, return the 1+the longer distance of its subtree.

//就是求树高

**public** **int** maxDepth(TreeNode root) {

**if**(root==**null**){

**return** 0;

}

**return** 1+Math.max(maxDepth(root.left),maxDepth(root.right));

}

### Minimum Depth of Binary Tree

[My Submissions](https://leetcode.com/problems/minimum-depth-of-binary-tree/submissions/)

QuestionEditorial Solution

Total Accepted: **100534** Total Submissions: **329860** Difficulty: **Easy**

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.

这个写法可以，分两种情况讨论。

**public** **class** **Solution** {

**public** **int** minDepth(TreeNode root) {

**if**(root == **null**) **return** 0;

**int** left = minDepth(root.left);

**int** right = minDepth(root.right);

**return** (left == 0 || right == 0) ? left + right + 1: Math.min(left,right) + 1;

}

}

**public** **int** minDepth(TreeNode root) {

**if**(root == **null**) **return** 0;

**if**(root.left == **null** || root.right == **null**)

**return** 1 + Math.max(minDepth(root.left), minDepth(root.right));

**return** 1 + Math.min(minDepth(root.left), minDepth(root.right));

}

**public** **int** minDepth(TreeNode root) {

**if** (root == **null**)

**return** 0;

**if** (root.left != **null** && root.right != **null**)

**return** Math.min(minDepth(root.left), minDepth(root.right))+1;

**else**

**return** Math.max(minDepth(root.left), minDepth(root.right))+1;

}

我的做法

public int minDepth(TreeNode root) {

if(root==null) return 0;

return minDepth(root,1);

}

public int minDepth(TreeNode root, int level) {

if(root==null) return Integer.MAX\_VALUE;

if(root.left==null && root.right==null) return level;

return Math.min(minDepth(root.left,level+1),minDepth(root.right,level+1));

}

### Same Tree

[My Submissions](https://leetcode.com/problems/same-tree/submissions/)

QuestionEditorial Solution

Total Accepted: **120879** Total Submissions: **280447** Difficulty: **Easy**

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 and the nodes have the same value.

做法类似对称树

**public** **boolean** isSameTree(TreeNode p, TreeNode q) {

**if**(p == **null** && q == **null**) **return** **true**;

**if**(p == **null** || q == **null**) **return** **false**;

**if**(p.val == q.val)

**return** isSameTree(p.left, q.left) && isSameTree(p.right, q.right);

**return** **false**;

}

### Invert Binary Tree

[My Submissions](https://leetcode.com/problems/invert-binary-tree/submissions/)

QuestionEditorial Solution

Total Accepted: **81297** Total Submissions: **181160** Difficulty: **Easy**

Invert 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.

Straightforward DFS recursive, iterative, BFS solutions

As in many other cases this problem has more than one possible solutions:

Lets start with straightforward - recursive DFS - it's easy to write and pretty much concise.

**public** **class** **Solution** {

**public** TreeNode invertTree(TreeNode root) {

**if** (root == **null**) {

**return** **null**;

}

**final** TreeNode left = root.left,

right = root.right;

root.left = invertTree(right);

root.right = invertTree(left);

**return** root;

}

}

The above solution is correct, but it is also bound to the application stack, which means that it's no so much scalable - (you can find the problem size that will overflow the stack and crash your application), so more robust solution would be to use stack data structure.

**public** **class** **Solution** {

**public** TreeNode invertTree(TreeNode root) {

**if** (root == **null**) {

**return** **null**;

}

**final** Deque<TreeNode> stack = **new** LinkedList<>();

stack.push(root);

**while**(!stack.isEmpty()) {

**final** TreeNode node = stack.pop();

**final** TreeNode left = node.left;

node.left = node.right;

node.right = left;

**if**(node.left != **null**) {

stack.push(node.left);

}

**if**(node.right != **null**) {

stack.push(node.right);

}

}

**return** root;

}

}

Finally we can easly convert the above solution to BFS - or so called level order traversal.

**public** **class** **Solution** {

**public** TreeNode invertTree(TreeNode root) {

**if** (root == **null**) {

**return** **null**;

}

**final** Queue<TreeNode> queue = **new** LinkedList<>();

queue.offer(root);

**while**(!queue.isEmpty()) {

**final** TreeNode node = queue.poll();

**final** TreeNode left = node.left;

node.left = node.right;

node.right = left;

**if**(node.left != **null**) {

queue.offer(node.left);

}

**if**(node.right != **null**) {

queue.offer(node.right);

}

}

**return** root;

}

}

If I can write this code, does it mean I can get job at Google? ;)

### Binary Tree Level Order Traversal

[My Submissions](https://leetcode.com/problems/binary-tree-level-order-traversal/submissions/)

QuestionEditorial Solution

Total Accepted: **97496** Total Submissions: **300711** Difficulty: **Easy**

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,#,#,15,7},

3

/ \

9 20

/ \

15 7

return its level order traversal as:

[

[3],

[9,20],

[15,7]

]

我是多使用一个队列（按照书上做法）

这里是每层计数

public class Solution {

public List<**List<Integer**>> levelOrder(TreeNode root) {

Queue<**TreeNode**> queue = new LinkedList<**TreeNode**>();

List<**List<Integer**>> wrapList = new LinkedList<**List<Integer**>>();

if(root == null) return wrapList;

queue.offer(root);

while(!queue.isEmpty()){

int levelNum = queue.size();

List<**Integer**> subList = new LinkedList<**Integer**>();

for(int i=0; i<**levelNum;** i++) {

if(queue.peek().left != null) queue.offer(queue.peek().left);

if(queue.peek().right != null) queue.offer(queue.peek().right);

subList.add(queue.poll().val);

}

wrapList.add(subList);

}

return wrapList;

}

}

### Binary Tree Level Order Traversal II

[My Submissions](https://leetcode.com/problems/binary-tree-level-order-traversal-ii/submissions/)

QuestionEditorial Solution

Total Accepted: **75916** Total Submissions: **225865** Difficulty: **Easy**

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,#,#,15,7},

3

/ \

9 20

/ \

15 7

return its bottom-up level order traversal as:

[

[15,7],

[9,20],

[3]

]

我的代码里，Queue<**TreeNode**> queue = new LinkedList<**TreeNode**>();不对？

当编译时类型和运行时类型不同时，编译时类型不支持的方法，在编译的时候是通不过的。

BFS solution:

public class Solution {

public List<**List<Integer**>> levelOrderBottom(TreeNode root) {

Queue<**TreeNode**> queue = new LinkedList<**TreeNode**>();

List<**List<Integer**>> wrapList = new LinkedList<**List<Integer**>>();

if(root == null) return wrapList;

queue.offer(root);

while(!queue.isEmpty()){

int levelNum = queue.size();

List<**Integer**> subList = new LinkedList<**Integer**>();

for(int i=0; i<**levelNum;** i++) {

if(queue.peek().left != null) queue.offer(queue.peek().left);

if(queue.peek().right != null) queue.offer(queue.peek().right);

subList.add(queue.poll().val);

}

wrapList.add(0, subList);

}

return wrapList;

}

}

DFS solution:

**public** **class** **Solution** {

**public** **List**<**List**<Integer>> levelOrderBottom(TreeNode root) {

**List**<**List**<Integer>> wrapList = **new** LinkedList<**List**<Integer>>();

levelMaker(wrapList, root, 0);

**return** wrapList;

}

**public** void levelMaker(**List**<**List**<Integer>> **list**, TreeNode root, int level) {

**if**(root == **null**) **return**;

**if**(level >= **list**.size()) {

**list**.add(0, **new** LinkedList<Integer>());

}

levelMaker(**list**, root.left, level+1);

levelMaker(**list**, root.right, level+1);

**list**.get(**list**.size()-level-1).add(root.val);

}

}

### Symmetric Tree

[My Submissions](https://leetcode.com/problems/symmetric-tree/submissions/)

QuestionEditorial Solution

Total Accepted: **101707** Total Submissions: **301938** Difficulty: **Easy**

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree is symmetric:

1

/ \

2 2

/ \ / \

3 4 4 3

But the following is not:

1

/ \

2 2

\ \

3 3

**Note:**  
Bonus points if you could solve it both recursively and iteratively.

**1ms recursive Java Solution, easy to understand 我的做法同**

**public** **boolean** isSymmetric(TreeNode root) {

**if**(root==**null**) **return** **true**;

**return** isMirror(root.left,root.right);

}

**public** **boolean** isMirror(TreeNode p, TreeNode q) {

**if**(p==**null** && q==**null**) **return** **true**;

**if**(p==**null** || q==**null**) **return** **false**;

**return** (p.val==q.val) && isMirror(p.left,q.right) && isMirror(p.right,q.left);

}

非递归做法

The idea is: 1. level traversal. 2. push nodes onto stack, every 2 consecutive is a pair, and should either be both null or have equal value. repeat until stack is empty.

**public** boolean isSymmetric(TreeNode root) {

**if** (root == null)

**return** **true**;

Stack<TreeNode> **stack** = **new** Stack<TreeNode>();

**stack**.push(root.left);

**stack**.push(root.right);

**while** (!**stack**.isEmpty()) {

TreeNode node1 = **stack**.pop();

TreeNode node2 = **stack**.pop();

**if** (node1 == null && node2 == null)

**continue**;

**if** (node1 == null || node2 == null)

**return** **false**;

**if** (node1.val != node2.val)

**return** **false**;

**stack**.push(node1.left);

**stack**.push(node2.right);

**stack**.push(node1.right);

**stack**.push(node2.left);

}

**return** **true**;

}

**public** **boolean** isSymmetric(TreeNode root) {

Queue<TreeNode> q = **new** LinkedList<TreeNode>();

**if**(root == **null**) **return** **true**;

q.add(root.left);

q.add(root.right);

**while**(q.size() > 1){

TreeNode left = q.poll(),

right = q.poll();

**if**(left== **null**&& right == **null**) **continue**;

**if**(left == **null** ^ right == **null**) **return** **false**;

**if**(left.val != right.val) **return** **false**;

q.add(left.left);

q.add(right.right);

q.add(left.right);

q.add(right.left);

}

**return** **true**;

}

Recursive--400ms:

**public** **boolean** isSymmetric(TreeNode root) {

**return** root==**null** || isSymmetricHelp(root.left, root.right);

}

**private** **boolean** isSymmetricHelp(TreeNode left, TreeNode right){

**if**(left==**null** || right==**null**)

**return** left==right;

**if**(left.val!=right.val)

**return** **false**;

**return** isSymmetricHelp(left.left, right.right) && isSymmetricHelp(left.right, right.left);

}

Non-recursive(use Stack)--460ms:

**public** **boolean** isSymmetric(TreeNode root) {

**if**(root==**null**) **return** **true**;

Stack<TreeNode> stack = **new** Stack<TreeNode>();

TreeNode left, right;

**if**(root.left!=**null**){

**if**(root.right==**null**) **return** **false**;

stack.push(root.left);

stack.push(root.right);

}

**else** **if**(root.right!=**null**){

**return** **false**;

}

**while**(!stack.empty()){

**if**(stack.size()%2!=0) **return** **false**;

right = stack.pop();

left = stack.pop();

**if**(right.val!=left.val) **return** **false**;

**if**(left.left!=**null**){

**if**(right.right==**null**) **return** **false**;

stack.push(left.left);

stack.push(right.right);

}

**else** **if**(right.right!=**null**){

**return** **false**;

}

**if**(left.right!=**null**){

**if**(right.left==**null**) **return** **false**;

stack.push(left.right);

stack.push(right.left);

}

**else** **if**(right.left!=**null**){

**return** **false**;

}

}

**return** **true**;

}

### Binary Tree Paths

[My Submissions](https://leetcode.com/problems/binary-tree-paths/submissions/)

QuestionEditorial Solution

Total Accepted: **41628** Total Submissions: **148557** Difficulty: **Easy**

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"]

**我用数组存了每个数字做处理（需要移除每步的影响），下面这个方法较好。**

**只有当左子树/右子树不空，才继续往下搜索**

**public** **List**<String> binaryTreePaths(TreeNode root) {

**List**<String> answer = **new** ArrayList<String>();

**if** (root != **null**) searchBT(root, "", answer);

**return** answer;

}

**private** void searchBT(TreeNode root, String path, **List**<String> answer) {

**if** (root.left == **null** && root.right == **null**) answer.add(path + root.val);

**if** (root.left != **null**) searchBT(root.left, path + root.val + "->", answer);

**if** (root.right != **null**) searchBT(root.right, path + root.val + "->", answer);

}

Lot of recursive solutions on this forum involves creating a helper recursive function with added parameters. The added parameter which usually is of the type List , carries the supplementary path information. However, the approach below doesn't use such a helper function.

**public** **List**<String> binaryTreePaths(TreeNode root) {

**List**<String> paths = **new** LinkedList<>();

**if**(root == **null**) **return** paths;

**if**(root.left == **null** && root.right == **null**){

paths.add(root.val+"");

**return** paths;

}

**for** (String path : binaryTreePaths(root.left)) {

paths.add(root.val + "->" + path);

}

**for** (String path : binaryTreePaths(root.right)) {

paths.add(root.val + "->" + path);

}

**return** paths;

}

### Lowest Common Ancestor of a Binary Search Tree

[My Submissions](https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-search-tree/submissions/)

QuestionEditorial Solution

Total Accepted: **62564** Total Submissions: **165234** Difficulty: **Easy**

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

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**).”

\_\_\_\_\_\_\_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.

3 lines with O(1) space, 1-Liners, Alternatives

Just walk down from the whole tree's root as long as both p and q are in the same subtree (meaning their values are both smaller or both larger than root's). This walks straight from the root to the LCA, not looking at the rest of the tree, so it's pretty much as fast as it gets. A few ways to do it:

**Iterative, O(1) space**

Python

**def** **lowestCommonAncestor**(self, root, p, q):

**while** (root.val - p.val) \* (root.val - q.val) > 0:

root = (root.left, root.right)[p.val > root.val]

**return** root

Java

**public** TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

**while** ((root.val - p.val) \* (root.val - q.val) > 0)

root = p.val < root.val ? root.left : root.right;

**return** root;

}

(in case of overflow, I'd do (root.val - (long)p.val) \* (root.val - (long)q.val))

Different Python

**def** **lowestCommonAncestor**(self, root, p, q):

a, b = sorted([p.val, q.val])

**while** **not** a <= root.val <= b:

root = (root.left, root.right)[a > root.val]

**return** root

"Long" Python, maybe easiest to understand

**def** **lowestCommonAncestor**(self, root, p, q):

**while** root:

**if** p.val < root.val > q.val:

root = root.left

**elif** p.val > root.val < q.val:

root = root.right

**else**:

**return** root

**Recursive**

Python

**def** **lowestCommonAncestor**(self, root, p, q):

next = p.val < root.val > q.val **and** root.left **or** \

p.val > root.val < q.val **and** root.right

**return** self.lowestCommonAncestor(next, p, q) **if** next **else** root

Python One-Liner

**def** **lowestCommonAncestor**(self, root, p, q):

**return** root **if** (root.val - p.val) \* (root.val - q.val) < 1 **else** \

self.lowestCommonAncestor((root.left, root.right)[p.val > root.val], p, q)

Java One-Liner

**public** TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

**return** (root.val - p.val) \* (root.val - q.val) < 1 ? root :

lowestCommonAncestor(p.val < root.val ? root.left : root.right, p, q);

}

"Long" Python, maybe easiest to understand

**def** **lowestCommonAncestor**(self, root, p, q):

**if** p.val < root.val > q.val:

**return** self.lowestCommonAncestor(root.left, p, q)

**if** p.val > root.val < q.val:

**return** self.lowestCommonAncestor(root.right, p, q)

**return** root

**public** **class** **Solution** {

**public** TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

**if**(root.val > p.val && root.val > q.val){

**return** lowestCommonAncestor(root.left, p, q);

}**else** **if**(root.val < p.val && root.val < q.val){

**return** lowestCommonAncestor(root.right, p, q);

}**else**{

**return** root;

}

}

}

### Sum of Left Leaves

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/sum-of-left-leaves/submissions/)

* Total Accepted: **3274**
* Total Submissions: **6887**
* Difficulty: **Easy**

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**.

我的做法

注意，如果root为叶子，则其不算左、右叶子

public int sumOfLeftLeaves(TreeNode root) {

int[] res=new int[1];

res[0]=0;

sum(root,false,res);

return res[0];

}

public void sum(TreeNode node, boolean flag, int[] res)

{

if(node==null) return;

if(flag && node.left==null && node.right==null)

{

res[0]+=node.val;

return;

}

sum(node.left,true,res);

sum(node.right,false,res);

}

**Java Solution using BFS**

**public** **class** **Solution** {

**public** **int** **sumOfLeftLeaves**(TreeNode root) {

**if**(root == **null** || root.left == **null** && root.right == **null**) **return** 0;

**int** res = 0;

Queue<TreeNode> queue = **new** LinkedList<>();

queue.offer(root);

**while**(!queue.isEmpty()) {

TreeNode curr = queue.poll();

**if**(curr.left != **null** && curr.left.left == **null** && curr.left.right == **null**) res += curr.left.val;

**if**(curr.left != **null**) queue.offer(curr.left);

**if**(curr.right != **null**) queue.offer(curr.right);

}

**return** res;

}

}

**public** **class** **Solution** {

**public** **int** **sumOfLeftLeaves**(TreeNode n) {

**if**(n==**null** ||(n.left==**null** && n.right ==**null**))**return** 0;

**int** l=0,r=0;

**if**(n.left!=**null**)l=(n.left.left==**null** && n.left.right==**null**)?n.left.val:sumOfLeftLeaves(n.left);

**if**(n.right!=**null**)r=sumOfLeftLeaves(n.right);

**return** l+r;

}

}

### 总结

递归处理，递归出口；return左右子树各自分别的情况。Root为空或为叶子之类的。

需要注意有无要考虑倒退回当前状态的变量。（将list加入最终结果时，可以加入list的副本）

转换左右子树时，考虑转换左子树后，左子树就发生变化了，所以需要先保存左子树，才可以用来后面作为参数传入。

在一路往叶子搜索题目时，可以使用if (root.left != null)来跳过不会通往叶子的部分。

层次遍历时可以使用当时队列大小来控制该层个数。

打印一段路径时，可以使用path数组，保存深搜一路的该层的节点值。

### MEDIUM

### Validate Binary Search Tree

[My Submissions](https://leetcode.com/problems/validate-binary-search-tree/submissions/)

QuestionEditorial Solution

Total Accepted: **88547** Total Submissions: **424154** Difficulty: **Medium**

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.

**public** **class** **Solution** {

**public** **boolean** isValidBST(TreeNode root) {

**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);

}

}

Basically what I am doing is recursively iterating over the tree while defining interval <minVal, maxVal> for each node which value must fit in.

### Binary Tree Inorder Traversal

[My Submissions](https://leetcode.com/problems/binary-tree-inorder-traversal/submissions/)

QuestionEditorial Solution

Total Accepted: **118037** Total Submissions: **300157** Difficulty: **Medium**

Given a binary tree, return the *inorder* traversal of its nodes' values.

For example:  
Given binary tree {1,#,2,3},

1

\

2

/

3

return [1,3,2].

**Note:** Recursive solution is trivial, could you do it iteratively?

一路将向左，将结点存入栈中，为空时，pop栈中的数据（pop时说明根和左节点已经处理）

此时将根的右子树进栈。

public List<**Integer**> inorderTraversal(TreeNode root) {

List<**Integer**> result = new ArrayList<>();

Deque<**TreeNode**> stack = new ArrayDeque<>();

TreeNode p = root;

while(!stack.isEmpty() || p != null) {

if(p != null) {

stack.push(p);

p = p.left;

} else {

TreeNode node = stack.pop();

result.add(node.val); // Add after all left children

p = node.right;

}

}

return result;

**Explanation**

The basic idea is referred from [here](https://leetcode.com/discuss/19765/iterative-solution-in-java-simple-and-readable): using stack to simulate the recursion procedure: for each node, travel to its left child until it's left leaf, then pop to left leaf's higher level node A, and switch to A's right branch. Keep the above steps until cur is null and stack is empty. As the following:

**Runtime = O(n)**: As each node is visited once

**Space = O(n)**

public List<**Integer**> inorderTraversal(TreeNode root) {

List<**Integer**> list = new ArrayList<**Integer**>();

Stack<**TreeNode**> stack = new Stack<**TreeNode**>();

TreeNode cur = root;

while(cur!=null || !stack.empty()){

while(cur!=null){

stack.add(cur);

cur = cur.left;

}

cur = stack.pop();

list.add(cur.val);

cur = cur.right;

}

return list;

}

### Binary Tree Preorder Traversal

[My Submissions](https://leetcode.com/problems/binary-tree-preorder-traversal/submissions/)

QuestionEditorial Solution

Total Accepted: **115218** Total Submissions: **291644** Difficulty: **Medium**

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 solution is trivial, could you do it iteratively?

Note that in this solution only right children are stored to stack.

public List<**Integer**> preorderTraversal(TreeNode node) {

List<**Integer**> list = new LinkedList<**Integer**>();

Stack<**TreeNode**> rights = new Stack<**TreeNode**>();

while(node != null) {

list.add(node.val);

if (node.right != null) {

rights.push(node.right);

}

node = node.left;

if (node == null && !rights.isEmpty()) {

node = rights.pop();

}

}

return list;

}

下面这个做法也不错

一路向左，保存及入栈左子树，如果为空，则pop，存入右子树

public List<**Integer**> preorderTraversal(TreeNode root) {

List<**Integer**> result = new ArrayList<>();

Deque<**TreeNode**> stack = new ArrayDeque<>();

TreeNode p = root;

while(!stack.isEmpty() || p != null) {

if(p != null) {

stack.push(p);

result.add(p.val); // Add before going to children

p = p.left;

} else {

TreeNode node = stack.pop();

p = node.right;

}

}

return result;

}

1. Create an empty stack, Push root node to the stack.
2. Do following while stack is not empty.

2.1. pop an item from the stack and print it.

2.2. push the right child of popped item to stack.

2.3. push the left child of popped item to stack.

我的做法是

List<Integer> res=new ArrayList<Integer>();

Stack<TreeNode> stack=new Stack<TreeNode>();

if(root!=null) stack.push(root);

while(!stack.isEmpty())

{

TreeNode temp=stack.pop();

res.add(temp.val);

if(temp.right!=null) stack.push(temp.right);

if(temp.left!=null) stack.push(temp.left);

}

return res;

### Binary Tree Postorder Traversal

[My Submissions](https://leetcode.com/problems/binary-tree-postorder-traversal/submissions/)

QuestionEditorial Solution

Total Accepted: **93592** Total Submissions: **266322** Difficulty: **Hard**

Given a binary tree, return the *postorder* traversal of its nodes' values.

For example:  
Given binary tree {1,#,2,3},

1

\

2

/

3

return [3,2,1].

**Note:** Recursive solution is trivial, could you do it iteratively?

pre-order traversal is **root-left-right**, and post order is **left-right-root**. modify the code for pre-order to make it root-right-left, and then **reverse** the output so that we can get left-right-root .

1. Create an empty stack, Push root node to the stack.
2. Do following while stack is not empty.

2.1. pop an item from the stack and print it.

2.2. push the left child of popped item to stack.

2.3. push the right child of popped item to stack.

1. reverse the ouput.
2. **class** Solution {
3. **public**:
4. **vector**<**int**> postorderTraversal(TreeNode \*root) {
5. **stack**<TreeNode\*> nodeStack;
6. **vector**<**int**> result;
7. //base case
8. **if**(root==NULL)
9. **return** result;
10. nodeStack.push(root);
11. **while**(!nodeStack.empty())
12. {
13. TreeNode\* node= nodeStack.top();
14. result.push\_back(node->val);
15. nodeStack.pop();
16. **if**(node->left)
17. nodeStack.push(node->left);
18. **if**(node->right)
19. nodeStack.push(node->right);
20. }
21. reverse(result.begin(),result.end());
22. **return** result;
23. }

};

**public** List<Integer> postorderTraversal(TreeNode root) {

LinkedList<Integer> result = **new** LinkedList<>();

Deque<TreeNode> **stack** = **new** ArrayDeque<>();

TreeNode p = root;

**while**(!**stack**.isEmpty() || p != null) {

**if**(p != null) {

**stack**.push(p);

result.addFirst(p.val); // Reverse the process of preorder

p = p.right; // Reverse the process of preorder

} **else** {

TreeNode node = **stack**.pop();

p = node.left; // Reverse the process of preorder

}

}

**return** result;

}

我的做法

List<Integer> res=new ArrayList<Integer>();

Stack<TreeNode> stack=new Stack<TreeNode>();

if(root!=null) stack.push(root);

while(!stack.isEmpty())

{

TreeNode temp=stack.pop();

res.add(0,temp.val);

if(temp.left!=null) stack.push(temp.left);

if(temp.right!=null) stack.push(temp.right);

}

return res;

### Binary Search Tree Iterator

[My Submissions](https://leetcode.com/problems/binary-search-tree-iterator/submissions/)

QuestionEditorial Solution

Total Accepted: **45011** Total Submissions: **131710** Difficulty: **Medium**

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling next() will return the next smallest number in the BST.

**Note:**next() and hasNext() should run in average O(1) time and uses O(*h*) memory, where *h* is the height of the tree.

参考树的中序遍历

一路将向左，将结点存入栈中，为空时，pop栈中的数据（pop时说明根和左节点已经处理）

此时将根的右子树进栈。

这个方法很好。每次，将node的所有直接的left结点，存放入栈中。

当需要找最小结点时，pop栈中的值，即为最小。同时将该结点的右子树入栈

I use Stack to store directed left children from root.  
When next() be called, I just pop one element and process its right child as new root.  
The code is pretty straightforward.

So this can satisfy O(h) memory, hasNext() in O(1) time,  
But next() is O(h) time.

I can't find a solution that can satisfy both next() in O(1) time, space in O(h).

Java:

**public** **class** BSTIterator {

**private** Stack<TreeNode> stack = **new** Stack<TreeNode>();

**public** **BSTIterator**(TreeNode root) {

pushAll(root);

}

*/\*\* @return whether we have a next smallest number \*/*

**public** boolean **hasNext**() {

**return** !stack.isEmpty();

}

*/\*\* @return the next smallest number \*/*

**public** **int** **next**() {

TreeNode tmpNode = stack.pop();

pushAll(tmpNode.right);

**return** tmpNode.val;

}

**private** **void** **pushAll**(TreeNode node) {

**for** (; node != null; stack.push(node), node = node.left);

}

}

I think you have the optimal solution. The question said "in average" O(1) time, which means amortized over all the next() calls.

My idea comes from this: My first thought was to use inorder traversal to put every node into an array, and then make an index pointer for the next() and hasNext(). That meets the O(1) run time but not the O(h) memory. O(h) is really much more less than O(n) when the tree is huge.

This means I cannot use a lot of memory, which suggests that I need to make use of the tree structure itself. And also, one thing to notice is the "average O(1) run time". It's weird to say average O(1), because there's nothing below O(1) in run time, which suggests in most cases, I solve it in O(1), while in some cases, I need to solve it in O(n) or O(h). These two limitations are big hints.

Before I come up with this solution, I really draw a lot binary trees and try inorder traversal on them. We all know that, once you get to a TreeNode, in order to get the smallest, you need to go all the way down its left branch. So our first step is to point to pointer to the left most TreeNode. The problem is how to do back trace. Since the TreeNode doesn't have father pointer, we cannot get a TreeNode's father node in O(1) without store it beforehand. Back to the first step, when we are traversal to the left most TreeNode, we store each TreeNode we met ( They are all father nodes for back trace).

After that, I try an example, for next(), I directly return where the pointer pointing at, which should be the left most TreeNode I previously found. What to do next? After returning the smallest TreeNode, I need to point the pointer to the next smallest TreeNode. When the current TreeNode has a right branch (It cannot have left branch, remember we traversal to the left most), we need to jump to its right child first and then traversal to its right child's left most TreeNode. When the current TreeNode doesn't have a right branch, it means there cannot be a node with value smaller than itself father node, point the pointer at its father node.

The overall thinking leads to the structure Stack, which fits my requirement so well.

*/\*\**

*\* Definition for binary tree*

*\* public class TreeNode {*

*\* int val;*

*\* TreeNode left;*

*\* TreeNode right;*

*\* TreeNode(int x) { val = x; }*

*\* }*

*\*/*

**public** **class** BSTIterator {

**private** Stack<TreeNode> stack;

**public** **BSTIterator**(TreeNode root) {

stack = **new** Stack<>();

TreeNode cur = root;

**while**(cur != null){

stack.push(cur);

**if**(cur.left != null)

cur = cur.left;

**else**

**break**;

}

}

*/\*\* @return whether we have a next smallest number \*/*

**public** boolean **hasNext**() {

**return** !stack.isEmpty();

}

*/\*\* @return the next smallest number \*/*

**public** **int** **next**() {

TreeNode node = stack.pop();

TreeNode cur = node;

*// traversal right branch*

**if**(cur.right != null){

cur = cur.right;

**while**(cur != null){

stack.push(cur);

**if**(cur.left != null)

cur = cur.left;

**else**

**break**;

}

}

**return** node.val;

}

}

*/\*\**

*\* Your BSTIterator will be called like this:*

*\* BSTIterator i = new BSTIterator(root);*

*\* while (i.hasNext()) v[f()] = i.next();*

*\*/*

### Binary Tree Right Side View

[My Submissions](https://leetcode.com/problems/binary-tree-right-side-view/submissions/)

QuestionEditorial Solution

Total Accepted: **38680** Total Submissions: **113882** Difficulty: **Medium**

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].

The core idea of this algorithm:

1.Each depth of the tree only select one node.  
2. View depth is current size of result list.

Here is the code:

**public** **class** **Solution** {

**public** **List**<Integer> rightSideView(TreeNode root) {

**List**<Integer> result = **new** ArrayList<Integer>();

rightView(root, result, 0);

**return** result;

}

**public** void rightView(TreeNode curr, **List**<Integer> result, int currDepth){

**if**(curr == **null**){

**return**;

}

**if**(currDepth == result.size()){

result.add(curr.val);

}

rightView(curr.right, result, currDepth + 1);

rightView(curr.left, result, currDepth + 1);

}

}

我的做法，从层次遍历考虑

public static List<Integer> rightSideView(TreeNode root) {

List<Integer> res=new ArrayList<Integer>();

LinkedList<TreeNode> queue=new LinkedList<TreeNode>();

if(root!=null) queue.add(root);

while(!queue.isEmpty())

{

int num=queue.size();

//TreeNode node=queue.pop();

res.add(queue.get(queue.size()-1).val);

for(int i=0;i<num;i++)

{

TreeNode node=queue.pop();

if(node.left!=null) queue.add(node.left);

if(node.right!=null) queue.add(node.right);

}

}

return res;

}

### Populating Next Right Pointers in Each Node

[My Submissions](https://leetcode.com/problems/populating-next-right-pointers-in-each-node/submissions/)

QuestionEditorial Solution

Total Accepted: **83912** Total Submissions: **230079** Difficulty: **Medium**

Given a binary tree

struct TreeLinkNode {

TreeLinkNode \*left;

TreeLinkNode \*right;

TreeLinkNode \*next;

}

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

**Note:**

* You may only use constant extra space.
* You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,  
Given the following perfect binary tree,

1

/ \

2 3

/ \ / \

4 5 6 7

After calling your function, the tree should look like:

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ / \

4->5->6->7 -> NULL

void **connect**(TreeLinkNode \*root) {

**if** (root == NULL) **return**;

TreeLinkNode \*pre = root;

TreeLinkNode \*cur = NULL;

**while**(pre->left) {

cur = pre;

**while**(cur) {

cur->left->next = cur->right;

**if**(cur->next) cur->right->next = cur->next->left;

cur = cur->next;

}

pre = pre->left;

}

}

you need two additional pointer.

cur表示已经构造好的这层；由cur这层去构造其下面那层；用levelStart去保存每一层的开始处（用于联系下一层）。

Java solution with O(1) memory+ O(n) time

**public** **class** **Solution** {

**public** **void** connect(TreeLinkNode root) {

TreeLinkNode level\_start=root;

**while**(level\_start!=**null**){

TreeLinkNode cur=level\_start;

**while**(cur!=**null**){

**if**(cur.left!=**null**) cur.left.next=cur.right;

**if**(cur.right!=**null** && cur.next!=**null**) cur.right.next=cur.next.left;

cur=cur.next;

}

level\_start=level\_start.left;

}

}

}

下面这个思路同第一个解答。左边如果不空，则左边的指针指向root的右边。

如果root右边不空且有下一个节点，则root右子树指向下一个节点的左子树

My recursive solution(Java)

public void **connect**(TreeLinkNode root) {

**if**(root == null)

**return**;

**if**(root.left != null){

root.left.**next** = root.right;

**if**(root.**next** != null)

root.right.**next** = root.**next**.left;

}

**connect**(root.left);

**connect**(root.right);

}

我的做法是递归，每层使用了dummy及一个指针。

### Populating Next Right Pointers in Each Node II

[My Submissions](https://leetcode.com/problems/populating-next-right-pointers-in-each-node-ii/submissions/)

Question

Total Accepted: **54394** Total Submissions: **167140** Difficulty: **Hard**

Follow up for problem "*Populating Next Right Pointers in Each Node*".

What if the given tree could be any binary tree? Would your previous solution still work?

**Note:**

* You may only use constant extra space.

For example,  
Given the following binary tree,

1

/ \

2 3

/ \ \

4 5 7

After calling your function, the tree should look like:

1 -> NULL

/ \

2 -> 3 -> NULL

/ \ \

4-> 5 -> 7 -> NULL

Just share my iterative solution with O(1) space and O(n) Time complexity

**public** **class** **Solution** {

//based on level order traversal

**public** **void** connect(TreeLinkNode root) {

TreeLinkNode head = **null**; //head of the next level

TreeLinkNode prev = **null**; //the leading node on the next level

TreeLinkNode cur = root; //current node of current level

**while** (cur != **null**) {

**while** (cur != **null**) { //iterate on the current level

//left child

**if** (cur.left != **null**) {

**if** (prev != **null**) {

prev.next = cur.left;

} **else** {

head = cur.left;

}

prev = cur.left;

}

//right child

**if** (cur.right != **null**) {

**if** (prev != **null**) {

prev.next = cur.right;

} **else** {

head = cur.right;

}

prev = cur.right;

}

//move to next node

cur = cur.next;

}

//move to next level

cur = head;

head = **null**;

prev = **null**;

}

}

}

The idea is simple: level-order traversal. You can see the following code:

这个方法不错，类似上一题的写法，只是从构造新的层的时候，使用了dummy节点

**public** **class** **Solution** {

**public** **void** connect(TreeLinkNode root) {

**while**(root != **null**){

TreeLinkNode tempChild = **new** TreeLinkNode(0);

TreeLinkNode currentChild = tempChild;

**while**(root!=**null**){

**if**(root.left != **null**) { currentChild.next = root.left; currentChild = currentChild.next;}

**if**(root.right != **null**) { currentChild.next = root.right; currentChild = currentChild.next;}

root = root.next;

}

root = tempChild.next;

}

}

}

### Binary Tree Zigzag Level Order Traversal

[My Submissions](https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/submissions/)

QuestionEditorial Solution

Total Accepted: **57781** Total Submissions: **203363** Difficulty: **Medium**

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,#,#,15,7},

3

/ \

9 20

/ \

15 7

return its zigzag level order traversal as:

[

[3],

[20,9],

[15,7]

]

我的做法是层次遍历。

**public** **class** **Solution** {

**public** **List**<**List**<Integer>> zigzagLevelOrder(TreeNode root)

{

**List**<**List**<Integer>> sol = **new** ArrayList<>();

travel(root, sol, 0);

**return** sol;

}

**private** void travel(TreeNode curr, **List**<**List**<Integer>> sol, int level)

{

**if**(curr == **null**) **return**;

**if**(sol.size() <= level)

{

**List**<Integer> newLevel = **new** LinkedList<>();

sol.add(newLevel);

}

**List**<Integer> collection = sol.get(level);

**if**(level % 2 == 0) collection.add(curr.val);

**else** collection.add(0, curr.val);

travel(curr.left, sol, level + 1);

travel(curr.right, sol, level + 1);

}

}

1. O(n) solution by using LinkedList along with ArrayList. So insertion in the inner list and outer list are both O(1),
2. Using DFS and creating new lists when needed.

should be quite straightforward. any better answer?

public class Solution { public List<List> zigzagLevelOrder(TreeNode root) { List<List> res = new ArrayList<>(); if(root == null) return res;

Queue<**TreeNode**> q = new LinkedList<>();

q.add(root);

boolean order = true;

int size = 1;

while(!q.isEmpty()) {

List<**Integer**> tmp = new ArrayList<>();

for(int i = 0; i < size; ++i) {

TreeNode n = q.poll();

if(order) {

tmp.add(n.val);

} else {

tmp.add(0, n.val);

}

if(n.left != null) q.add(n.left);

if(n.right != null) q.add(n.right);

}

res.add(tmp);

size = q.size();

order = order ? false : true;

}

return res;

}

### Flatten Binary Tree to Linked List

[My Submissions](https://leetcode.com/problems/flatten-binary-tree-to-linked-list/submissions/)

QuestionEditorial Solution

Total Accepted: **78783** Total Submissions: **255331** Difficulty: **Medium**

Given a binary tree, flatten it to a linked list in-place.

For example,  
Given

1

/ \

2 5

/ \ \

3 4 6

The flattened tree should look like:

1

\

2

\

3

\

4

\

5

\

6

[click to show hints.](https://leetcode.com/problems/flatten-binary-tree-to-linked-list/)

**Hints:**

If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order traversal.

我的做法，将前序遍历的结果预先存起来。再重新连接每个节点的右子树。

下面这个方法好。利用深搜的遍历顺序来构造。

My short post order traversal Java solution for share

**private** TreeNode prev = **null**;

**public** **void** flatten(TreeNode root) {

**if** (root == **null**)

**return**;

flatten(root.right);

flatten(root.left);

root.right = prev;

root.left = **null**;

prev = root;

}

class Solution {

public:

void flatten(TreeNode \*root) {

TreeNode\*now = root;

**while** (now)

{

**if**(now->left)

{

//Find current node's prenode that links to current node'**s** right subtree

TreeNode\* pre = now->left;

**while**(pre->right)

{

pre = pre->right;

}

pre->right = now->right;

//Use current node's left subtree to replace its right subtree(original right

//subtree is already linked by current node'**s** prenode

now->right = now->left;

now->left = NULL;

}

now = now->right;

}

}

};

**public** **void** flatten(TreeNode root) {

**if** (root == **null**) **return**;

TreeNode left = root.left;

TreeNode right = root.right;

root.left = **null**;

flatten(left);

flatten(right);

root.right = left;

TreeNode cur = root;

**while** (cur.right != **null**) cur = cur.right;

cur.right = right;

}

void flatten(TreeNode \*root) {

**while** (root) {

**if** (root->left && root->right) {

TreeNode\* t = root->left;

**while** (t->right)

t = t->right;

t->right = root->right;

}

**if**(root->left)

root->right = root->left;

root->left = NULL;

root = root->right;

}

}

This solution is based on recursion. We simply flatten left and right subtree and paste each sublist to the right child of the root. (don't forget to set left child to null)

it is DFS so u need a stack. Dont forget to set the left child to null, or u'll get TLE. (tricky!)

**public** **void** flatten(TreeNode root) {

**if** (root == **null**) **return**;

Stack<TreeNode> stk = **new** Stack<TreeNode>();

stk.push(root);

**while** (!stk.isEmpty()){

TreeNode curr = stk.pop();

**if** (curr.right!=**null**)

stk.push(curr.right);

**if** (curr.left!=**null**)

stk.push(curr.left);

**if** (!stk.isEmpty())

curr.right = stk.peek();

curr.left = **null**; // dont forget this!!

}

}

### Unique Binary Search Trees

[My Submissions](https://leetcode.com/problems/unique-binary-search-trees/submissions/)

QuestionEditorial Solution

Total Accepted: **79512** Total Submissions: **213629** Difficulty: **Medium**

Given *n*, how many structurally unique **BST's** (binary search trees) that store values 1...*n*?

For example,  
Given *n* = 3, there are a total of 5 unique BST's.

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

**这个方法很好。注意G(n)表示长度为n的序列，可以组成不同BST的个数。即{1,2}和{4,5}可以组成的不同BST数目相同。**

**DP Solution in 6 lines with explanation. F(i, n) = G(i-1) \* G(n-i)**

The problem can be solved in a dynamic programming way. I’ll explain the intuition and formulas in the following.

Given a sequence 1…n, to construct a Binary Search Tree (BST) out of the sequence, we could enumerate each number i in the sequence, and use the number as the root, naturally, the subsequence 1…(i-1) on its left side would lay on the left branch of the root, and similarly the right subsequence (i+1)…n lay on the right branch of the root. We then can construct the subtree from the subsequence recursively. Through the above approach, we could ensure that the BST that we construct are all unique, since they have unique roots.

The problem is to calculate the number of unique BST. To do so, we need to define two functions:

G(n): the number of unique BST for a sequence of length n.

F(i, n), 1 <= i <= n: the number of unique BST, where the number i is the root of BST, and the sequence ranges from 1 to n.

As one can see, G(n) is the actual function we need to calculate in order to solve the problem. And G(n) can be derived from F(i, n), which at the end, would recursively refer to G(n).

First of all, given the above definitions, we can see that the total number of unique BST G(n), is the sum of BST F(i) using each number i as a root. i.e.

G(n) = F(1, n) + F(2, n) + ... + F(n, n).

Particularly, the bottom cases, there is only one combination to construct a BST out of a sequence of length 1 (only a root) or 0 (empty tree). i.e.（这里G(0),需要设为1，否则乘积是会变为0）

G(0)=1, G(1)=1.

Given a sequence 1…n, we pick a number i out of the sequence as the root, then the number of unique BST with the specified root F(i), is the cartesian product笛卡尔乘积of the number of BST for its left and right subtrees. For example, F(3, 7): the number of unique BST tree with number 3 as its root. To construct an unique BST out of the entire sequence [1, 2, 3, 4, 5, 6, 7] with 3 as the root, which is to say, we need to construct an unique BST out of its left subsequence [1, 2] and another BST out of the right subsequence [4, 5, 6, 7], and then combine them together (i.e.cartesian product). The tricky part is that we could consider the number of unique BST out of sequence [1,2] as G(2), and the number of of unique BST out of sequence [4, 5, 6, 7] as G(4). Therefore, F(3,7) = G(2) \* G(4).

i.e.

F(i, n) = G(i-1) \* G(n-i) 1 <**=** i <= n

Combining the above two formulas, we obtain the recursive formula for G(n). i.e.

G(n) = G(0) \* G(n-1) + G(1) \* G(n-2) + … + G(n-1) \* G(0)

In terms of calculation, we need to start with the lower number, since the value of G(n) depends on the values of G(0) … G(n-1).

With the above explanation and formulas, here is the implementation in Java.

**public** **int** numTrees(**int** n) {

**int** [] G = **new** **int**[n+1];

G[0] = G[1] = 1;

**for**(**int** i=2; i<=n; ++i) {//长为i的序列，可以构造的唯一的二叉查找树的个数

**for**(**int** j=1; j<=i; ++j) {

G[i] += G[j-1] \* G[i-j];

}

}

**return** G[n];

}

/\*

Hope it will help you to understand :

n = 0; null

count[0] = 1

n = 1; 1

count[1] = 1

n = 2; 1\_\_ \_\_2

\ /

count[1] count[1]

count[2] = 1 + 1 = 2

n = 3; 1\_\_ \_\_2\_\_ \_\_3

\ / \ /

count[2] count[1] count[1] count[2]

count[3] = 2 + 1 + 2 = 5

n = 4; 1\_\_ \_\_2\_\_ \_\_\_3\_\_\_

\ / \ / \

count[3] count[1] count[2] count[2] count[1]

\_\_4

/

count[3]

count[4] = 5 + 2 + 2 + 5 = 14

And so on...

\*/

### Unique Binary Search Trees II

[My Submissions](https://leetcode.com/problems/unique-binary-search-trees-ii/submissions/)

QuestionEditorial Solution

Total Accepted: **52855** Total Submissions: **182605** Difficulty: **Medium**

Given *n*, generate all structurally unique **BST's** (binary search trees) that store values 1...*n*.

For example,  
Given *n* = 3, your program should return all 5 unique BST's shown below.

1 3 3 2 1

\ / / / \ \

3 2 1 1 3 2

/ / \ \

2 1 2 3

I start by noting that 1..n is the in-order traversal for any BST with nodes 1 to n. So if I pick i-th node as my root, the left subtree will contain elements 1 to (i-1), and the right subtree will contain elements (i+1) to n. I use recursive calls to get back all possible trees for left and right subtrees and combine them in all possible ways with the root.

**public** **class** **Solution** {

**public** **List**<TreeNode> generateTrees(int n) {

**return** genTrees(1,n);

}

**public** **List**<TreeNode> genTrees (int start, int end)

{

**List**<TreeNode> **list** = **new** ArrayList<TreeNode>();

**if**(start>end)

{

**list**.add(**null**);

**return** **list**;

}

**if**(start == end){

**list**.add(**new** TreeNode(start));

**return** **list**;

}

**List**<TreeNode> left,right;

**for**(int i=start;i<=end;i++)

{

left = genTrees(start, i-1);

right = genTrees(i+1,end);

**for**(TreeNode lnode: left)

{

**for**(TreeNode rnode: right)

{

TreeNode root = **new** TreeNode(i);

root.left = lnode;

root.right = rnode;

**list**.add(root);

}

}

}

**return** **list**;

}

}

Here is my java solution with DP:

**public** **class** **Solution** {

**public** **static** **List**<TreeNode> generateTrees(int n) {

**List**<TreeNode>[] result = **new** **List**[n+1];

result[0] = **new** ArrayList<TreeNode>();

result[0].add(**null**);

**for**(int len = 1; len <= n; len++){

result[len] = **new** ArrayList<TreeNode>();

**for**(int j=0; j<len; j++){

**for**(TreeNode nodeL : result[j]){

**for**(TreeNode nodeR : result[len-j-1]){

TreeNode node = **new** TreeNode(j+1);

node.left = nodeL;

node.right = **clone**(nodeR, j+1);

result[len].add(node);

}

}

}

}

**return** result[n];

}

**private** **static** TreeNode **clone**(TreeNode n, int offset){

**if**(n == **null**)

**return** **null**;

TreeNode node = **new** TreeNode(n.val + offset);

node.left = **clone**(n.left, offset);

node.right = **clone**(n.right, offset);

**return** node;

}

}

**result[i]** stores the result until length **i**. For the result for length i+1, select the root node j from 0 to i, combine the result from left side and right side. Note for the right side we have to clone the nodes as the value will be offsetted by **j**.

下面这个写法比较好，非常经典的写法

**divide-and-conquer. F(i) = G(i-1) \* G(n-i)**

This problem is a variant of the problem of [Unique Binary Search Trees](https://oj.leetcode.com/problems/unique-binary-search-trees/).

I provided a solution along with explanation for the above problem, in the question ["DP solution in 6 lines with explanation"](https://oj.leetcode.com/discuss/24282/dp-solution-in-6-lines-with-explanation-f-i-g-i-1-g-n-i)

It is intuitive to solve this problem by following the same algorithm. Here is the code in a divide-and-conquer style.

public List<**TreeNode**> generateTrees(int n) {

return generateSubtrees(1, n);

}

private List<**TreeNode**> generateSubtrees(int s, int e) {

List<**TreeNode**> res = new LinkedList<**TreeNode**>();

if (s > e) {

res.add(null); // empty tree

return res;

}

for (int i = s; i <**=** e; ++i) {

List<TreeNode> leftSubtrees = generateSubtrees(s, i - 1);

List<**TreeNode**> rightSubtrees = generateSubtrees(i + 1, e);

for (TreeNode left : leftSubtrees) {

for (TreeNode right : rightSubtrees) {

TreeNode root = new TreeNode(i);

root.left = left;

root.right = right;

res.add(root);

}

}

}

return res;

}

### Count Complete Tree Nodes

[My Submissions](https://leetcode.com/problems/count-complete-tree-nodes/submissions/)

QuestionEditorial Solution

Total Accepted: **30217** Total Submissions: **122136** Difficulty: **Medium**

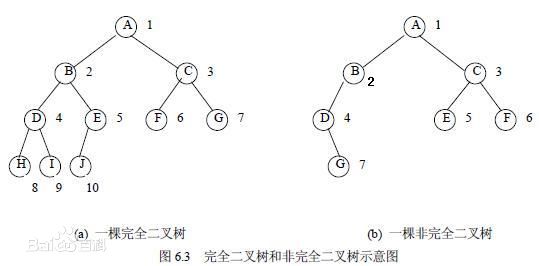
Given a **complete** binary tree, count the number of nodes.

**Definition of a complete binary tree from**[**Wikipedia**](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees)**:**  
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.

若设二叉树的深度为h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。

完全二叉树是由[满二叉树](http://baike.baidu.com/view/427110.htm)而引出来的。对于深度为K的，有n个结点的二叉树，当且仅当其每一个结点都与深度为K的满二叉树中编号从1至n的结点一一对应时称之为完全二叉树。

一棵二叉树至多只有最下面的一层上的结点的度数可以小于2，并且最下层上的结点都集中在该层最左边的若干位置上，则此二叉树成为完全二叉树。



**Concise Java solutions O(log(n)^2)**

**Main Solution** - 572 ms

**class** **Solution** {

**int** height(TreeNode root) {

**return** root == **null** ? -1 : 1 + height(root.left);

}//利用完全二叉树的性质求树高（返回的值是树高-1）

**public** **int** countNodes(TreeNode root) {

**int** h = height(root);

**return** h < 0 ? 0 :

height(root.right) == h-1 ? (1 << h) + countNodes(root.right) //若右子树有结点，说明root的左子树是满的，则左子树+root结点数量为2^(树高-1)

: (1 << h-1) + countNodes(root.left);

}

}

**Explanation**

The height of a tree can be found by just going left. Let a single node tree have height 0. Find the height h of the whole tree. If the whole tree is empty, i.e., has height -1, there are 0 nodes.

Otherwise check whether the height of the right subtree is just one less than that of the whole tree, meaning left and right subtree have the same height.

* If yes, then the last node on the last tree row is in the right subtree and the left subtree is a full tree of height h-1. So we take the 2^h-1 nodes of the left subtree plus the 1 root node plus recursively the number of nodes in the right subtree.
* If no, then the last node on the last tree row is in the left subtree and the right subtree is a full tree of height h-2. So we take the 2^(h-1)-1 nodes of the right subtree plus the 1 root node plus recursively the number of nodes in the left subtree.

Since I halve the tree in every recursive step, I have O(log(n)) steps. Finding a height costs O(log(n)). So overall O(log(n)^2).

**Iterative Version** - 508 ms

Here's an iterative version as well, with the benefit that I don't recompute h in every step.

**class** **Solution** {

**int** height(TreeNode root) {

**return** root == **null** ? -1 : 1 + height(root.left);

}

**public** **int** countNodes(TreeNode root) {

**int** nodes = 0, h = height(root);

**while** (root != **null**) {

**if** (height(root.right) == h - 1) {

nodes += 1 << h;

root = root.right;

} **else** {

nodes += 1 << h-1;

root = root.left;

}

h--;

}

**return** nodes;

}

}

**A Different Solution** - 544 ms

Here's one based on [victorlee's C++ solution](https://leetcode.com/discuss/38899/easy-short-c-recursive-solution).

**class** **Solution** {

**public** **int** countNodes(TreeNode root) {

**if** (root == **null**)

**return** 0;

TreeNode left = root, right = root;

**int** height = 0;

**while** (right != **null**) {

left = left.left;

right = right.right;

height++;

}

**if** (left == **null**)

**return** (1 << height) - 1;

**return** 1 + countNodes(root.left) + countNodes(root.right);

}

}

Note that that's basically this:

**public** **int** countNodes(TreeNode root) {

**if** (root == **null**)

**return** 0;

**return** 1 + countNodes(root.left) + countNodes(root.right)

That would be O(n). But... the actual solution has a gigantic optimization. It first walks all the way left and right to determine the height and whether it's a full tree, meaning the last row is full. If so, then the answer is just 2^height-1. And since always at least one of the two recursive calls is such a full tree, at least one of the two calls immediately stops. Again we have runtime O(log(n)^2).

下面这个方法不错（java版超时？）;第一个方法也不错

class Solution {

public:

**int** countNodes(TreeNode\* root) {

**if**(!root) **return** 0;

**int** hl=0, hr=0;

TreeNode \*l=root, \*r=root;

**while**(l) {hl++;l=l->left;}

**while**(r) {hr++;r=r->right;}

**if**(hl==hr) **return** pow(2,hl)-1; //满二叉树

**return** 1+countNodes(root->left)+countNodes(root->right);

}

};

**class** **Solution** {

**public**:

**int** countNodes(TreeNode\* root) {

**if**(!root) **return** 0;

**int** lh=fTreeHt(root->left);

**int** rh=fTreeHt(root->right);

**if**(lh==rh)

**return** (1<<lh)+countNodes(root->right); /\*1(根节点) + (1<<lh)-1(完全左子树) + # of rightNode \*/

**else**

**return** (1<<rh)+countNodes(root->left); /\*1(根节点) + (1<<rh)-1(完全右子树) + # of leftNode\*/

}

**private**:

**int** fTreeHt(TreeNode \*root){ //get the height of a complete binary tree.

**if**(!root) **return** 0;

**return** 1+fTreeHt(root->left);

}

};

### Kth Smallest Element in a BST

[My Submissions](https://leetcode.com/problems/kth-smallest-element-in-a-bst/submissions/)

QuestionEditorial Solution

Total Accepted: **42139** Total Submissions: **113382** Difficulty: **Medium**

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?

**Hint:**

1. Try to utilize the property of a BST.
2. What if you could modify the BST node's structure?
3. The optimal runtime complexity is O(height of BST).

**3 ways implemented in JAVA: Binary Search, in-order iterative & recursive**

这个方法可以，需要先计算BST总共中有多少节点。

还可以用迭代法（中序遍历），下面的一个方法好。

Binary Search (dfs): most preferable

**public** **int** kthSmallest(TreeNode root, **int** k) {

**int** count = countNodes(root.left);

**if** (k <= count) {

**return** kthSmallest(root.left, k);

} **else** **if** (k > count + 1) {

**return** kthSmallest(root.right, k-1-count); // 1 is counted as current node

}

**return** root.val;

}

**public** **int** countNodes(TreeNode n) {

**if** (n == **null**) **return** 0;

**return** 1 + countNodes(n.left) + countNodes(n.right);

}

DFS in-order recursive:

// better keep these two variables in a wrapper class

**private** **static** **int** number = 0;

**private** **static** **int** count = 0;

**public** **int** kthSmallest(TreeNode root, **int** k) {

count = k;

helper(root);

**return** number;

}

**public** **void** helper(TreeNode n) {

**if** (n.left != **null**) helper(n.left);

count--;

**if** (count == 0) {

number = n.val;

**return**;

}

**if** (n.right != **null**) helper(n.right);

}

DFS in-order iterative:

**public** **int** kthSmallest(TreeNode root, **int** k) {

Stack<TreeNode> st = **new** Stack<>();

**while** (root != **null**) {

st.push(root);

root = root.left;

}

**while** (k != 0) {

TreeNode n = st.pop();

k--;

**if** (k == 0) **return** n.val;

TreeNode right = n.right;

**while** (right != **null**) {

st.push(right);

right = right.left;

}

}

**return** -1; // never hit if k is valid

}

**What if you could modify the BST node's structure?**

If we could add a count field in the BST node class, it will take O(n) time when we calculate the count value for the whole tree, but after that, it will take O(logn) time when insert/delete a node or calculate the kth smallest element.

**public** **class** **Solution** {

**public** **int** kthSmallest(TreeNode root, **int** k) {

TreeNodeWithCount rootWithCount = buildTreeWithCount(root);

**return** kthSmallest(rootWithCount, k);

}

**private** TreeNodeWithCount buildTreeWithCount(TreeNode root) {

**if** (root == **null**) **return** **null**;

TreeNodeWithCount rootWithCount = **new** TreeNodeWithCount(root.val);

rootWithCount.left = buildTreeWithCount(root.left);

rootWithCount.right = buildTreeWithCount(root.right);

**if** (rootWithCount.left != **null**) rootWithCount.count += rootWithCount.left.count;

**if** (rootWithCount.right != **null**) rootWithCount.count += rootWithCount.right.count;

**return** rootWithCount;

}

**private** **int** kthSmallest(TreeNodeWithCount rootWithCount, **int** k) {

**if** (k <= 0 || k > rootWithCount.count) **return** -1;

**if** (rootWithCount.left != **null**) {

**if** (rootWithCount.left.count >= k) **return** kthSmallest(rootWithCount.left, k);

**if** (rootWithCount.left.count == k-1) **return** rootWithCount.val;

**return** kthSmallest(rootWithCount.right, k-1-rootWithCount.left.count);

} **else** {

**if** (k == 1) **return** rootWithCount.val;

**return** kthSmallest(rootWithCount.right, k-1);

}

}

**class** **TreeNodeWithCount** {

**int** val;

**int** count;

TreeNodeWithCount left;

TreeNodeWithCount right;

TreeNodeWithCount(**int** x) {val = x; count = 1;};

}

}

In order traverse for BST gives the natural order of numbers. No need to use array.

Recursive:

**int** count = 0;

**int** result = Integer.MIN\_VALUE;

**public** **int** kthSmallest(TreeNode root, **int** k) {

traverse(root, k);

**return** result;

}

**public** **void** traverse(TreeNode root, **int** k) {

**if**(root == **null**) **return**;

traverse(root.left, k);

count ++;

**if**(count == k) result = root.val;

traverse(root.right, k);

}

这个方法好（基于中序遍历）。我用了优先队列完全没有必要。

Iterative:

**public** **int** kthSmallest(TreeNode root, **int** k) {

Stack<TreeNode> **stack** = **new** Stack<TreeNode>();

TreeNode p = root;

**int** count = 0;

**while**(!**stack**.isEmpty() || p != null) {

**if**(p != null) {

**stack**.push(p); // Just like recursion

p = p.left;

} **else** {

TreeNode node = **stack**.pop();

**if**(++count == k) **return** node.val;

p = node.right;

}

}

**return** Integer.MIN\_VALUE;

}

### Path Sum II

[My Submissions](https://leetcode.com/problems/path-sum-ii/submissions/)

QuestionEditorial Solution

Total Accepted: **77063** Total Submissions: **274611** Difficulty: **Medium**

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]

]

**public** **List**<**List**<Integer>> pathSum(TreeNode root, int sum){

**List**<**List**<Integer>> result = **new** LinkedList<**List**<Integer>>();

**List**<Integer> currentResult = **new** LinkedList<Integer>();

pathSum(root,sum,currentResult,result);

**return** result;

}

**public** void pathSum(TreeNode root, int sum, **List**<Integer> currentResult,

**List**<**List**<Integer>> result) {

**if** (root == **null**)

**return**;

currentResult.add(**new** Integer(root.val));

**if** (root.left == **null** && root.right == **null** && sum == root.val) {

result.add(**new** LinkedList(currentResult));

currentResult.remove(currentResult.size() - 1);//don't forget to remove the last integer

**return**;

} **else** {

pathSum(root.left, sum - root.val, currentResult, result);

pathSum(root.right, sum - root.val, currentResult, result);

}

currentResult.remove(currentResult.size() - 1);

}

Save intermediate result into stack and save the stack into result array once its sum == required sum.

**public** **class** **Solution** {

**private** **List**<**List**<Integer>> resultList = **new** ArrayList<**List**<Integer>>();

**public** void pathSumInner(TreeNode root, int sum, Stack<Integer>path) {

path.push(root.val);

**if**(root.left == **null** && root.right == **null**)

**if**(sum == root.val) resultList.add(**new** ArrayList<Integer>(path));

**if**(root.left!=**null**) pathSumInner(root.left, sum-root.val, path);

**if**(root.right!=**null**)pathSumInner(root.right, sum-root.val, path);

path.pop();

}

**public** **List**<**List**<Integer>> pathSum(TreeNode root, int sum) {

**if**(root==**null**) **return** resultList;

Stack<Integer> path = **new** Stack<Integer>();

pathSumInner(root, sum, path);

**return** resultList;

}

}

**这个写法不错**

**private** **List**<**List**<Integer>> result = **new** ArrayList<**List**<Integer>>();

**public** **List**<**List**<Integer>> pathSum(TreeNode root, int sum) {

helper(**new** ArrayList<Integer>(), root, sum);

**return** result;

}

**private** void helper(**List**<Integer> **list**, TreeNode root, int sum) {

**if** (root == **null**) **return**;

**list**.add(root.val);

sum -= root.val;

**if** (root.left == **null** && root.right == **null**) {

**if** (sum == 0) result.add(**list**);

**return**;

}

helper(**new** ArrayList<Integer>(**list**), root.left, sum);

helper(**new** ArrayList<Integer>(**list**), root.right, sum);

}

### Sum Root to Leaf Numbers

[My Submissions](https://leetcode.com/problems/sum-root-to-leaf-numbers/submissions/)

QuestionEditorial Solution

Total Accepted: **73087** Total Submissions: **225349** Difficulty: **Medium**

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.

**public** **class** **Solution** {

**public** **int** sumNumbers(TreeNode root) {

**if** (root == **null**)

**return** 0;

**return** sumR(root, 0);

}

**public** **int** sumR(TreeNode root, **int** x) {

**if** (root.right == **null** && root.left == **null**)

**return** 10 \* x + root.val;

**int** val = 0;

**if** (root.left != **null**)

val += sumR(root.left, 10 \* x + root.val);

**if** (root.right != **null**)

val += sumR(root.right, 10 \* x + root.val);

**return** val;

}

}

下面这个写法不错

I use recursive solution to solve the problem.

**public** **int** sumNumbers(TreeNode root) {

**return** sum(root, 0);

}

**public** **int** sum(TreeNode n, **int** s){

**if** (n == **null**) **return** 0;

**if** (n.right == **null** && n.left == **null**) **return** s\*10 + n.val;

**return** sum(n.left, s\*10 + n.val) + sum(n.right, s\*10 + n.val);

}

I prefer the iterative method over recursion. I used 2 queues to do a breadth first traversal. both time and space complexity is O(N):

**public** **class** **Solution** {

**public** **int** sumNumbers(TreeNode root) {

**int** total = 0;

LinkedList<TreeNode> q = **new** LinkedList<TreeNode>();

LinkedList<Integer> sumq = **new** LinkedList<Integer>();

**if**(root !=**null**){

q.addLast(root);

sumq.addLast(root.val);

}

**while**(!q.isEmpty()){

TreeNode current = q.removeFirst();

**int** partialSum = sumq.removeFirst();

**if**(current.left == **null** && current.right==**null**){

total+=partialSum;

}**else**{

**if**(current.right !=**null**){

q.addLast(current.right);

sumq.addLast(partialSum\*10+current.right.val);

}

**if**(current.left !=**null**){

q.addLast(current.left);

sumq.addLast(partialSum\*10+current.left.val);

}

}

}

**return** total;

}

我的做法

public int sumNumbers(TreeNode root) {

int[] res=new int[1];

res[0]=0;

sumNumbers(res,root,0);

return res[0];

}

public void sumNumbers(int[] res, TreeNode root,int sum) {

if(root==null) return;

if(root.left==null && root.right==null)//是叶子，则该轮的数可以累加到结果

{

res[0]+=sum\*10+root.val;

}

else

{

sumNumbers(res,root.left,sum\*10+root.val);

sumNumbers(res,root.right,sum\*10+root.val);

}

}

### Lowest Common Ancestor of a Binary Tree

[My Submissions](https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/submissions/)

QuestionEditorial Solution

Total Accepted: **38673** Total Submissions: **134878** Difficulty: **Medium**

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.

我的做法是按照书

Same solution in several languages. It's recursive and expands the meaning of the function. If the current (sub)tree contains both p and q, then the function result is their LCA. If only one of them is in that subtree, then the result is that one of them. If neither are in that subtree, the result is null/None/nil.

Update: I also wrote [two iterative solutions](https://leetcode.com/discuss/45603/iterative-solution) now, one of them being a version of the solution here. They're more complicated than this simple recursive solution, but I do find them interesting

**Java**

**public** TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

**if** (root == **null** || root == p || root == q) **return** root;

TreeNode left = lowestCommonAncestor(root.left, p, q);

TreeNode right = lowestCommonAncestor(root.right, p, q);

**return** left == **null** ? right : right == **null** ? left : root;

}

**public** **class** **Solution** {

**public** TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

**if**(root == **null** || root == p || root == q) **return** root;

TreeNode left = lowestCommonAncestor(root.left, p, q);

TreeNode right = lowestCommonAncestor(root.right, p, q);

**if**(left != **null** && right != **null**) **return** root;

**return** left != **null** ? left : right;

}

}

**Java**

**public** **class** **Solution** {

**public** TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

Map<TreeNode, TreeNode> **parent** = **new** HashMap<>();

Deque<TreeNode> stack = **new** ArrayDeque<>();

**parent**.put(root, **null**);

stack.push(root);

**while** (!**parent**.containsKey(p) || !**parent**.containsKey(q)) {

TreeNode node = stack.pop();

**if** (node.left != **null**) {

**parent**.put(node.left, node);

stack.push(node.left);

}

**if** (node.right != **null**) {

**parent**.put(node.right, node);

stack.push(node.right);

}

}

Set<TreeNode> ancestors = **new** HashSet<>();

**while** (p != **null**) {

ancestors.add(p);

p = **parent**.get(p);

}

**while** (!ancestors.contains(q))

q = **parent**.get(q);

**return** q;

}

}

To find the lowest common ancestor, we need to find where is p and q and a way to track their ancestors. A parent pointer for each node found is good for the job. After we found both p andq, we create a set of p's ancestors. Then we travel through q's ancestors, the first one appears in p's is our answer.

### Convert Sorted Array to Binary Search Tree

[My Submissions](https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/submissions/)

QuestionEditorial Solution

Total Accepted: **72151** Total Submissions: **195367** Difficulty: **Medium**

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

我的做法同下—>书

Hi everyone, this is my accepted recursive Java solution. I get overflow problems at first because I didn't use mid - 1 and mid + 1 as the bound. Hope this helps :)

**public** TreeNode sortedArrayToBST(**int**[] num) {

**if** (num.length == 0) {

**return** **null**;

}

TreeNode head = helper(num, 0, num.length - 1);

**return** head;

}

**public** TreeNode helper(**int**[] num, **int** low, **int** high) {

**if** (low > high) { // Done

**return** **null**;

}

**int** mid = (low + high) / 2;

TreeNode node = **new** TreeNode(num[mid]);

node.left = helper(num, low, mid - 1);

node.right = helper(num, mid + 1, high);

**return** node;

}

I came up with the recursion solution first and tried to translate it into an iterative solution. It is very similar to doing a tree inorder traversal, I use three stacks - nodeStack stores the node I am going to process next, and **leftIndexStack** and **rightIndexStack** store the range where this node need to read from the **nums**.

**public** **class** **Solution** {

**public** TreeNode sortedArrayToBST(**int**[] nums) {

**int** len = nums.length;

**if** ( len == 0 ) { **return** **null**; }

// 0 as a placeholder

TreeNode head = **new** TreeNode(0);

Deque<TreeNode> nodeStack = **new** LinkedList<TreeNode>() {{ push(head); }};

Deque<Integer> leftIndexStack = **new** LinkedList<Integer>() {{ push(0); }};

Deque<Integer> rightIndexStack = **new** LinkedList<Integer>() {{ push(len-1); }};

**while** ( !nodeStack.isEmpty() ) {

TreeNode currNode = nodeStack.pop();

**int** left = leftIndexStack.pop();

**int** right = rightIndexStack.pop();

**int** mid = left + (right-left)/2; // avoid overflow

currNode.val = nums[mid];

**if** ( left <= mid-1 ) {

currNode.left = **new** TreeNode(0);

nodeStack.push(currNode.left);

leftIndexStack.push(left);

rightIndexStack.push(mid-1);

}

**if** ( mid+1 <= right ) {

currNode.right = **new** TreeNode(0);

nodeStack.push(currNode.right);

leftIndexStack.push(mid+1);

rightIndexStack.push(right);

}

}

**return** head;

}

}

### Construct Binary Tree from Preorder and Inorder Traversal

[My Submissions](https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/submissions/)

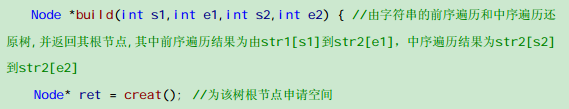
QuestionEditorial Solution

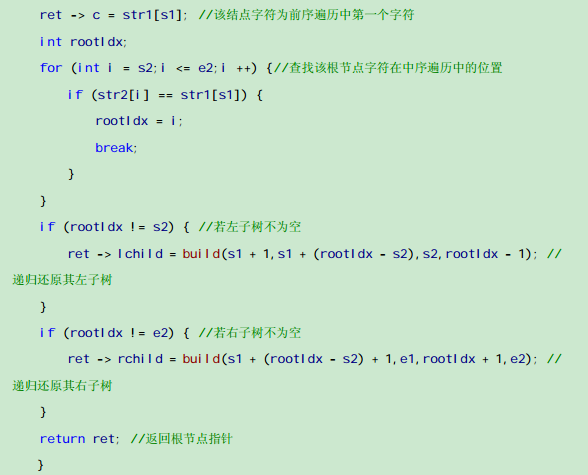
Total Accepted: **60225** Total Submissions: **212107** Difficulty: **Medium**

Given preorder and inorder traversal of a tree, construct the binary tree.

**Note:**  
You may assume that duplicates do not exist in the tree.

王道P63页也有解析





Hi guys, this is my Java solution. I read this [post](http://leetcode.com/2011/04/construct-binary-tree-from-inorder-and-preorder-postorder-traversal.html), which is very helpful.

The basic idea is here: Say we have 2 arrays, PRE and IN. Preorder traversing implies that PRE[0] is the root node. Then we can find this PRE[0] in IN, say it's IN[5]. Now we know that IN[5] is root, so we know that IN[0] - IN[4] is on the left side, IN[6] to the end is on the right side. Recursively doing this on subarrays, we can build a tree out of it :)

Hope this helps.

这个方法同上，可以。（关键在于，设置参数，前序开始指针，中序开始和结束开始指针）。构造法，同Unique Binary Search Trees II

**public** TreeNode buildTree(**int**[] preorder, **int**[] inorder) {

**return** helper(0, 0, inorder.length - 1, preorder, inorder);

}

**public** TreeNode helper(**int** preStart, **int** inStart, **int** inEnd, **int**[] preorder, **int**[] inorder) {

**if** (preStart > preorder.length - 1 || inStart > inEnd) {

**return** **null**;

}

TreeNode root = **new** TreeNode(preorder[preStart]);

**int** inIndex = 0; // Index of current root in inorder

**for** (**int** i = inStart; i <= inEnd; i++) {

**if** (inorder[i] == root.val) {

inIndex = i;

}

}

root.left = helper(preStart + 1, inStart, inIndex - 1, preorder, inorder);

root.right = helper(preStart + inIndex - inStart + 1, inIndex + 1, inEnd, preorder, inorder);

**return** root;

}

Nice solution! One improvement: remember to use **HashMap** to cache the inorder[] position. This can reduce your solution from 20ms to 5ms.

Here is the my Java solution:

**public** TreeNode buildTree(**int**[] preorder, **int**[] inorder) {

Map<Integer, Integer> inMap = **new** HashMap<Integer, Integer>();

**for**(**int** i = 0; i < inorder.length; i++) {

inMap.put(inorder[i], i);

}

TreeNode root = buildTree(preorder, 0, preorder.length - 1, inorder, 0, inorder.length - 1, inMap);

**return** root;

}

**public** TreeNode buildTree(**int**[] preorder, **int** preStart, **int** preEnd, **int**[] inorder, **int** inStart, **int** inEnd, Map<Integer, Integer> inMap) {

**if**(preStart > preEnd || inStart > inEnd) **return** **null**;

TreeNode root = **new** TreeNode(preorder[preStart]);

**int** inRoot = inMap.**get**(root.val);

**int** numsLeft = inRoot - inStart;

root.left = buildTree(preorder, preStart + 1, preStart + numsLeft, inorder, inStart, inRoot - 1, inMap);

root.right = buildTree(preorder, preStart + numsLeft + 1, preEnd, inorder, inRoot + 1, inEnd, inMap);

**return** root;

}

I din't find iterative solutions discussed in the old Discuss. So, I thought, I will add my solution in here.

The idea is as follows:

1) Keep pushing the nodes from the preorder into a stack (and keep making the tree by adding nodes to the left of the previous node) until the top of the stack matches the inorder.

2) At this point, pop the top of the stack until the top does not equal inorder (keep a flag to note that you have made a pop).

3) Repeat 1 and 2 until preorder is empty. The key point is that whenever the flag is set, insert a node to the right and reset the flag.

**class** Solution {

**public**:

TreeNode \*buildTree(**vector**<**int**> &preorder, **vector**<**int**> &inorder) {

**if**(preorder.size()==0)

**return** NULL;

**stack**<**int**> s;

**stack**<TreeNode \*> st;

TreeNode \*t,\*r,\*root;

**int** i,j,f;

f=i=j=0;

s.push(preorder[i]);

root = **new** TreeNode(preorder[i]);

st.push(root);

t = root;

i++;

**while**(i<preorder.size())

{

**if**(!st.empty() && st.top()->val==inorder[j])

{

t = st.top();

st.pop();

s.pop();

f = 1;

j++;

}

**else**

{

**if**(f==0)

{

s.push(preorder[i]);

t -> left = **new** TreeNode(preorder[i]);

t = t -> left;

st.push(t);

i++;

}

**else**

{

f = 0;

s.push(preorder[i]);

t -> right = **new** TreeNode(preorder[i]);

t = t -> right;

st.push(t);

i++;

}

}

}

**return** root;

}

};

### Construct Binary Tree from Inorder and Postorder Traversal

[My Submissions](https://leetcode.com/problems/construct-binary-tree-from-inorder-and-postorder-traversal/submissions/)

QuestionEditorial Solution

Total Accepted: **52695** Total Submissions: **182860** Difficulty: **Medium**

Given inorder and postorder traversal of a tree, construct the binary tree.

**Note:**  
You may assume that duplicates do not exist in the tree.

**My recursive Java code with O(n) time and O(n) space**

The the basic idea is to take the last element in postorder array as the root, find the position of the root in the inorder array; then locate the range for left sub-tree and right sub-tree and do recursion. Use a HashMap to record the index of root in the inorder array.

**public** TreeNode buildTreePostIn(**int**[] inorder, **int**[] postorder) {

**if** (inorder == **null** || postorder == **null** || inorder.length != postorder.length)

**return** **null**;

HashMap<Integer, Integer> hm = **new** HashMap<Integer,Integer>();

**for** (**int** i=0;i<inorder.length;++i)

hm.put(inorder[i], i);

**return** buildTreePostIn(inorder, 0, inorder.length-1, postorder, 0,

postorder.length-1,hm);

}

**private** TreeNode buildTreePostIn(**int**[] inorder, **int** **is**, **int** ie, **int**[] postorder, **int** ps, **int** pe,

HashMap<Integer,Integer> hm){

**if** (ps>pe || **is**>ie) **return** **null**;

TreeNode root = **new** TreeNode(postorder[pe]);

**int** ri = hm.**get**(postorder[pe]);

TreeNode leftchild = buildTreePostIn(inorder, **is**, ri-1, postorder, ps, ps+ri-**is**-1, hm);

TreeNode rightchild = buildTreePostIn(inorder,ri+1, ie, postorder, ps+ri-**is**, pe-1, hm);

root.left = leftchild;

root.right = rightchild;

**return** root;

}

This is my version: similar idea, but no HashMap needed!   
(TreeNode end is the boundary of left subtree.)

**int** pInorder; // index of inorder array

**int** pPostorder; // index of postorder array

**private** TreeNode buildTree(**int**[] inorder, **int**[] postorder, TreeNode end) {

**if** (pPostorder < 0) {

**return** **null**;

}

// create root node

TreeNode n = **new** TreeNode(postorder[pPostorder--]);

// if right node exist, create right subtree

**if** (inorder[pInorder] != n.val) {

n.right = buildTree(inorder, postorder, n);

}

pInorder--;

// if left node exist, create left subtree

**if** ((end == **null**) || (inorder[pInorder] != end.val)) {

n.left = buildTree(inorder, postorder, end);

}

**return** n;

}

**public** TreeNode buildTree(**int**[] inorder, **int**[] postorder) {

pInorder = inorder.length - 1;

pPostorder = postorder.length - 1;

**return** buildTree(inorder, postorder, **null**);

}

Below is the O(n) solution from @hongzhi but that discuss is closed now 'cause @hongzhi says little about his code.

https://oj.leetcode.com/discuss/6334/here-is-my-o-n-solution-is-it-neat

I've modified some of and tried this code and got AC. Just share about some comprehension about his code.

I've modified vtn(vector) to stn(stack) in that **stack** is probably what this algs means and needs.

What matters most is the meaning of *stn*.

Only nodes whoes left side **hasn't been** handled will be pushed into *stn*.

And inorder is organized as (inorder of left) root (inorder of right),

And postorder is as (postorder of left) (postorder of right) root.

So at the very begin, we only have root in stn and we check if *inorder.back() == root->val* and in most cases it's **false**(see Note 1). Then we make this node root's right sub-node and push it into stn.

**Note 1: this is actually *(inorder of right).back() == (postorder of right).back()*, so if only there's no right subtree or the answer will always be false.**

**Note 2: we delete one node from *postorder* as we push one into stn.**

Now we have [root, root's right] as stn and we check *inorder.back() == stn.top()->val* again.

* **true** means *inorder.back()* is the root node and needs handled left case.
* **false** means *inorder.back()* is the next right sub-node

So when we encounter a true, we will cache *stn.top()* as p and **delete both nodes from inorder and stn**.

Then we check inorder.size(), if there's no nodes left, it means p has no left node.

Else the next node in inorder could be *p's left node* or *p's father* which equals to the now*stn.top()* (remember we popped *p* from *stn* above).

If the latter happens, it means *p* has **no left node** and we need to move on to *p's father(stn.top())*.

If the former happens, it means *p* has one left node and it's *postorder.back()*, so we put it to p's left and delete it from the *postorder* and push the left node into *stn* 'cause **it** should be the next check node as the *postorder* is organized as above.

That's all of it. The algs just build a binary tree. :)

Inform me if there's anything vague or wrong, I'm open to any suggestions.

Thank you for sharing your explanation.

Here's a Java version of this solution:

import java.util.Stack;

**public** **class** Solution {

**public** TreeNode buildTree(**int**[] inorder, **int**[] postorder) {

**if** (inorder == null || inorder.length < 1) **return** null;

**int** i = inorder.length - 1;

**int** p = i;

TreeNode node;

TreeNode root = **new** TreeNode(postorder[postorder.length - 1]);

Stack<TreeNode> **stack** = **new** Stack<>();

**stack**.push(root);

p--;

**while** (**true**) {

**if** (inorder[i] == **stack**.peek().val) { // inorder[i] is on top of stack, pop stack to get its parent to get to left side

**if** (--i < 0) **break**;

node = **stack**.pop();

**if** (!**stack**.isEmpty() && inorder[i] == **stack**.peek().val) {// continue pop stack to get to left side

**continue**;

}

node.left = **new** TreeNode(postorder[p]);

**stack**.push(node.left);

} **else** { // inorder[i] is not on top of stack, postorder[p] must be right child

node = **new** TreeNode(postorder[p]);

**stack**.peek().right = node;

**stack**.push(node);

}

p--;

}

**return** root;

}

}

TreeNode \*buildTree(**vector**<**int**> &preorder, **vector**<**int**> &inorder) {

**return** create(preorder, inorder, 0, preorder.size() - 1, 0, inorder.size() - 1);

}

TreeNode\* create(**vector**<**int**>& preorder, **vector**<**int**>& inorder, **int** ps, **int** pe, **int** is, **int** ie){

**if**(ps > pe){

**return** **nullptr**;

}

TreeNode\* node = **new** TreeNode(preorder[ps]);

**int** pos;

**for**(**int** i = is; i <= ie; i++){

**if**(inorder[i] == node->val){

pos = i;

**break**;

}

}

node->left = create(preorder, inorder, ps + 1, ps + pos - is, is, pos - 1);

node->right = create(preorder, inorder, pe - ie + pos + 1, pe, pos + 1, ie);

**return** node;

}

The first element in preorder array can divide inorder array into two parts. Then we can divide preorder array into two parts. Make this element a node. And the left sub-tree of this node is the left part, right sub-tree of this node is the right part. This problem can be solved following this logic.

下面这个方法，容易理解，类似前面一题。

重点在于设置了后序（只需要指定start指针）和中序（反过来的）范围的指针。

倒序遍历

这里，先build左子树还是右子树都可以

注意这里的inStart是从数组的尾部开始。

**public** TreeNode buildTree(**int**[] inorder, **int**[] postorder) {

**return** buildTree(inorder, inorder.length-1, 0, postorder, postorder.length-1);

}

**private** TreeNode buildTree(**int**[] inorder, **int** inStart, **int** inEnd, **int**[] postorder,

**int** postStart) {

**if** (postStart < 0 || inStart < inEnd)

**return** **null**;

//The last element in postorder is the root.

TreeNode root = **new** TreeNode(postorder[postStart]);

//find the index of the root from inorder. Iterating from the end.

**int** rIndex = inStart;

**for** (**int** i = inStart; i >= inEnd; i--) {

**if** (inorder[i] == postorder[postStart]) {

rIndex = i;

**break**;

}

}

//build right and left subtrees. Again, scanning from the end to find the sections.

root.right = buildTree(inorder, inStart, rIndex + 1, postorder, postStart-1);

root.left = buildTree(inorder, rIndex - 1, inEnd, postorder, postStart - (inStart - rIndex) -1);//非常类似于前一题，区别在于这里传入参数时就是从末尾开始

**return** root;

}

### 总结

Integer.MAX\_VALUE不能表示时，可以用Long. MAX\_VALUE

中序遍历时一路向左保存。为空时，才从栈中取出数据，取出数据时就要保存到最后结果里。

构造不同的二叉查找树时，深搜。假设左右两边的子树组已经构造好，然后利用这个，去构造新的二叉查找树。

Root to leaf结果回退的情况需要注意

## 字符串

### Compare Version Numbers

[My Submissions](https://leetcode.com/problems/compare-version-numbers/submissions/)

QuestionEditorial Solution

Total Accepted: **52177** Total Submissions: **298173** Difficulty: **Easy**

Compare two version numbers *version1* and *version2*.  
If *version1* > *version2* return 1, if *version1* < *version2* return -1, otherwise return 0.

You may assume that the version strings are non-empty and contain only digits and the . character.  
The . character does not represent a decimal point and is used to separate number sequences.  
For instance, 2.5 is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.

Here is an example of version numbers ordering:

0.1 < 1.1 < 1.2 < 13.37

有可能版本号中有许多位，如0.0.1；有可能版本号01.xx；1和1.1

我的做法

注意split里是以[\\.来分割，不是直接用](file:///\\.来分割，不是直接用)“.”

用点分割字符串，对字符串数组中的每一位转换成数字进行比较，如果其中一个的数字用完了，就用0补

public int compareVersion(String version1, String version2) {

String[] v1=version1.split("\\.");

String[] v2=version2.split("\\.");

int len1=v1.length;

int len2=v2.length;

int num1=0,num2=0;

for(int i=0;i<len1 || i<len2;i++)

{

num1=0;

num2=0;

if(i<len1) num1=Integer.parseInt(v1[i]);

if(i<len2) num2=Integer.parseInt(v2[i]);

if(num1>num2) return 1;

if(num1<num2) return -1;

}

return 0;

}

这个做法，和我的类似

This code assumes that next level is zero if no mo levels in shorter version number. And than compare levels.

public **int** compareVersion(String version1, String version2) {

String[] levels1 = version1.**split**("\\.");

String[] levels2 = version2.**split**("\\.");

**int** **length** = Math.max(levels1.**length**, levels2.**length**);

**for** (**int** i=0; i<**length**; i++) {

Integer v1 = i < levels1.**length** ? Integer.parseInt(levels1[i]) : 0;

Integer v2 = i < levels2.**length** ? Integer.parseInt(levels2[i]) : 0;

**int** compare = v1.compareTo(v2);

**if** (compare != 0) {

**return** compare;

}

}

**return** 0;

}

### Count and Say

[My Submissions](https://leetcode.com/problems/count-and-say/submissions/)

QuestionEditorial Solution

Total Accepted: **80178** Total Submissions: **275998** Difficulty: **Easy**

The count-and-say sequence is the sequence of integers beginning as follows:  
1, 11, 21, 1211, 111221, ...

1 is read off as "one 1" or 11.  
11 is read off as "two 1s" or 21.  
21 is read off as "one 2, then one 1" or 1211.

Given an integer *n*, generate the *n*th sequence.

Note: The sequence of integers will be represented as a string.

我的做法

常规做法，遍历上一个字符串，记录个数和相应数字

public String countAndSay(int n) {

String str="1";

String res="";

for(int i=1;i<=n-1;i++)

{

char ch=str.charAt(0);

int cnt=1;

res="";

for(int j=1;j<str.length();j++)

{

if(str.charAt(j)==ch)

{

cnt++;

}

else

{

res+=cnt;

res+=ch;

cnt=1;

ch=str.charAt(j);

}

}

res+=cnt;

res+=ch;

str=res;

}

return str;

}

这个写法清晰，思路同上

I found nobody answered this question in Java. Actually I got some trouble even this question is not so hard.

Maybe many other people had some trouble too. So I put my answer here.

public class Solution {

public String countAndSay(int n) {

StringBuilder curr=new StringBuilder("1");

StringBuilder prev;

int count;

char say;

for (int i=1;i<**n;i++){**

prev=curr;

curr=new StringBuilder();

count=1;

say=prev.charAt(0);

for (int j=1,len=prev.length();j<len;j++){

if (prev.charAt(j)!=say){

curr.append(count).append(say);

count=1;

say=prev.charAt(j);

}

else count++;

}

curr.append(count).append(say);

}

return curr.toString();

}

}

@code StringBuilder.append() is the default way to append one string to another. While I have tried String.cancate(),which is not working properly.

Any comment is welcomed.

**string** countAndSay(**int** n) {

**if** (n == 0) **return** "";

**string** res = "1";

**while** (--n) {

**string** cur = "";

**for** (**int** i = 0; i < res.size(); i++) {

**int** count = 1;

**while** ((i + 1 < res.size()) && (res[i] == res[i + 1])){

count++;

i++;

}

cur += to\_string(count) + res[i];

}

res = cur;

}

**return** res;

}

### Add Binary

[My Submissions](https://leetcode.com/problems/add-binary/submissions/)

QuestionEditorial Solution

Total Accepted: **83723** Total Submissions: **304481** Difficulty: **Easy**

Given two binary strings, return their sum (also a binary string).

For example,  
a = "11"  
b = "1"  
Return "100".

我的做法

常规模拟

public String addBinary(String a, String b) {

String res="";

int len1=a.length();

int len2=b.length();

int len=Math.max(len1,len2);

int num1=0,num2=0,carry=0;

for(int i=0;i<len;i++)

{

num1=0;

num2=0;

if(i<len1) num1=a.charAt(len1-1-i)=='1'?1:0;

if(i<len2) num2=b.charAt(len2-1-i)=='1'?1:0;

res=(num1+num2+carry)%2+res;

carry=(num1+num2+carry)/2;

}

if(carry>0) res=carry+res;

return res;

}

**class** Solution

{

**public**:

**string** addBinary(**string** a, **string** b)

{

**string** s = "";

**int** c = 0, i = a.size() - 1, j = b.size() - 1;

**while**(i >= 0 || j >= 0 || c == 1)

{

c += i >= 0 ? a[i --] - '0' : 0;

c += j >= 0 ? b[j --] - '0' : 0;

s = **char**(c % 2 + '0') + s;

c /= 2;

}

**return** s;

}

};

**public** **class** **Solution** {

**public** String addBinary(String a, String b) {

**if**(a == **null** || a.isEmpty()) {

**return** b;

}

**if**(b == **null** || b.isEmpty()) {

**return** a;

}

**char**[] aArray = a.toCharArray();

**char**[] bArray = b.toCharArray();

StringBuilder stb = **new** StringBuilder();

**int** i = aArray.length - 1;

**int** j = bArray.length - 1;

**int** aByte;

**int** bByte;

**int** carry = 0;

**int** result;

**while**(i > -1 || j > -1 || carry == 1) {

aByte = (i > -1) ? Character.getNumericValue(aArray[i--]) : 0;

bByte = (j > -1) ? Character.getNumericValue(bArray[j--]) : 0;

result = aByte ^ bByte ^ carry;

carry = ((aByte + bByte + carry) >= 2) ? 1 : 0;

stb.append(result);

}

**return** stb.reverse().toString();

}

}

这个写法不错

public **class** **Solution** {

public String addBinary(String a, String b) {

StringBuilder sb = new StringBuilder();

int i = a.length() - 1, j = b.length() -1, carry = 0;

**while** (i >= 0 || j >= 0) {

int sum = carry;

**if** (j >= 0) sum += b.charAt(j--) - '0';

**if** (i >= 0) sum += a.charAt(i--) - '0';

sb.append(sum % 2);

carry = sum / 2;

}

**if** (carry != 0) sb.append(carry);

**return** sb.reverse().toString();

}

}

Computation from string usually can be simplified by using a carry as such.

### Multiply Strings

[My Submissions](https://leetcode.com/problems/multiply-strings/submissions/)

QuestionEditorial Solution

Total Accepted: **60999** Total Submissions: **259449** Difficulty: **Medium**

Given two numbers represented as strings, return multiplication of the numbers as a string.

**Note:**

* The numbers can be arbitrarily large and are non-negative.
* Converting the input string to integer is **NOT** allowed.
* You should **NOT** use internal library such as **BigInteger**.

我的做法

注意Corner case：0\*0; 69\*32014; 123\*0

public String multiply(String num1, String num2) {

int len1=num1.length();

int len2=num2.length();

if(len1==0 && len2==0) return "";

String res="";

int carry=0;

String sum="";

int cnt=0;

for(int i=0;i<len1;i++)

{

int n1=num1.charAt(len1-1-i)-'0';

if(n1==0) continue;

sum="";

carry=0;

cnt=i;

for(int j=0;j<len2;j++)

{

int n2=num2.charAt(len2-1-j)-'0';

sum=(n1\*n2+carry)%10+sum;

carry=(n1\*n2+carry)/10;

}

if(carry>0) sum=carry+sum;

while(cnt>0)

{

sum+="0";//increment

cnt--;

}

res=add(sum,res);//remove multi zeroes

}

return res.equals("")?"0":res;

}

public String add(String a, String b) {

String res="";

int len1=a.length();

int len2=b.length();

int len=Math.max(len1,len2);

int num1=0,num2=0,carry=0;

for(int i=0;i<len;i++)

{

num1=0;

num2=0;

if(i<len1) num1=a.charAt(len1-1-i)-'0';

if(i<len2) num2=b.charAt(len2-1-i)-'0';

res=(num1+num2+carry)%10+res;

carry=(num1+num2+carry)/10;

}

if(carry>0) res=carry+res;

if(res.length()>0 && res.charAt(0)=='0') return "0";

return res;

}

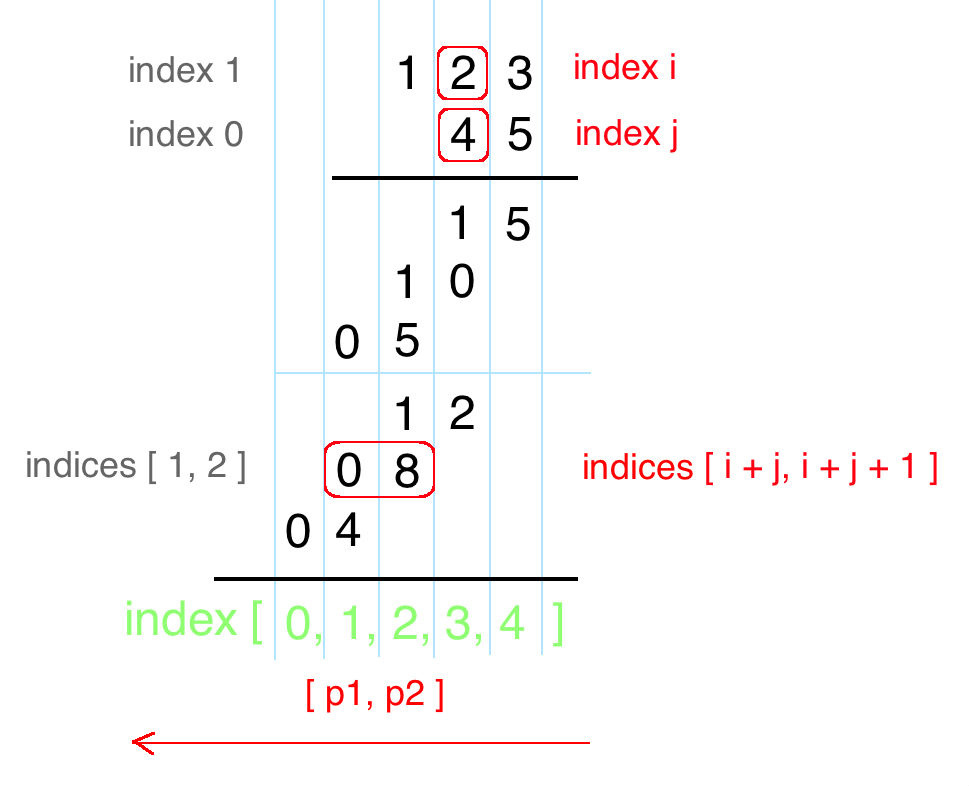
这个方法非常好

N位数乘以M位数，结果最多是N+M位

Remember how we do multiplication?

Start from right to left, perform multiplication on every pair of digits, and add them together. Let's draw the process! From the following draft, we can immediately conclude:

`**num1**[i] \* **num2**[j]` **will** **be** **placed** **at** **indices** `[i + j`, `i + j + 1]`

[](https://drscdn.500px.org/photo/130178585/m=2048/300d71f784f679d5e70fadda8ad7d68f)

Here is my solution. Hope it helps!

public String multiply(String num1, String num2) {

**int** **m** = num1.**length**(), n = num2.**length**();

**int**[] **pos** = new **int**[**m** + n];

**for**(**int** i = **m** - 1; i >= 0; i--) {

**for**(**int** j = n - 1; j >= 0; j--) {

**int** mul = (num1.charAt(i) - '0') \* (num2.charAt(j) - '0');

**int** p1 = i + j, p2 = i + j + 1;

**int** sum = mul + **pos**[p2];

**pos**[p1] += sum / 10;

**pos**[p2] = (sum) % 10;

}

}

StringBuilder sb = new StringBuilder();

**for**(**int** p : **pos**) **if**(!(sb.**length**() == 0 && p == 0)) sb.append(p);

**return** sb.**length**() == 0 ? "0" : sb.toString();

}

### Length of Last Word

[My Submissions](https://leetcode.com/problems/length-of-last-word/submissions/)

QuestionEditorial Solution

Total Accepted: **92354** Total Submissions: **315006** Difficulty: **Easy**

Given a string *s* consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string.

If the last word does not exist, return 0.

**Note:** A word is defined as a character sequence consists of non-space characters only.

For example,   
Given *s* = "Hello World",  
return 5.

我的做法

public int lengthOfLastWord(String s) {

String[] str=s.split("\\s+");

return str.length==0?0:str[str.length-1].length();

}

I've noticed that a lot of solutions use available library functions that return directly the positions of certain characters or do other operations like "split". I personally don't think that's a good idea. Firstly, these functions take some time and usually involve with iteration through the whole string. Secondly, questions like this one is intended to be a practice of detail implementation, not calling other functions. My solution like below uses only the most basic string operations and probably beats many other solutions which call other existing functions.

**int** lengthOfLastWord(const char\* **s**) {

**int** len = 0;

**while** (\*s) {

**if** (\*s++ != ' ')

++len;

**else** **if** (\*s && \*s != ' ')

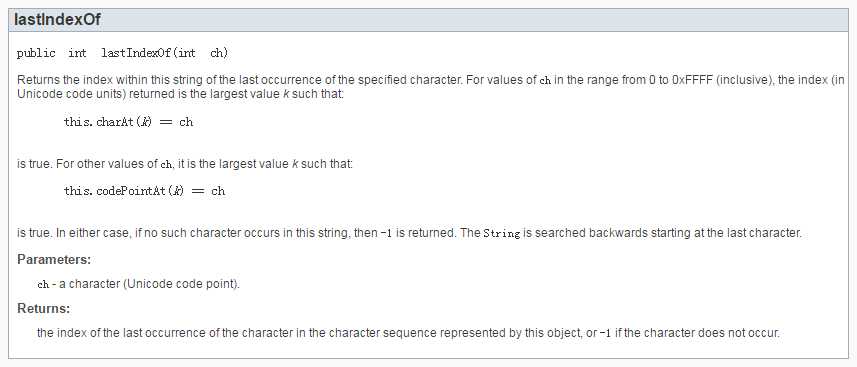
len = 0;

}

**return** len;

}

**String方法**



**public** **int** lengthOfLastWord(String s) {

**return** s.trim().length()-s.trim().lastIndexOf(" ")-1;

}

This one single line solution is just used to compare the time cost with the solution below: (Time cost: 348ms vs. 344ms, almost the same)

**public** **int** lengthOfLastWord(String s) {

**int** len=s.length(), lastLength=0;

**while**(len > 0 && s.charAt(len-1)==' '){

len--;

}

**while**(len > 0 && s.charAt(len-1)!=' '){

lastLength++;

len--;

}

**return** lastLength;

}

### Ransom Note

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/ransom-note/submissions/)

* Total Accepted: **1172**
* Total Submissions: **2443**
* Difficulty: **Easy**

 Given  an  arbitrary  ransom  note  string  and  another  string  containing  letters from  all  the  magazines,  write  a  function  that  will  return  true  if  the  ransom   note  can  be  constructed  from  the  magazines ;  otherwise,  it  will  return  false.

Each  letter  in  the  magazine  string  can  only  be  used  once  in  your  ransom  note.

**Note:**  
You may assume that both strings contain only lowercase letters.

canConstruct("a", "b") -> false

canConstruct("aa", "ab") -> false

canConstruct("aa", "aab") -> true

我的做法，统计下字母数量

public class Solution {

public boolean canConstruct(String ransomNote, String magazine) {

int[] map=new int[26];

for(int i=0; i<magazine.length(); i++)

{

char ch=magazine.charAt(i);

map[ch-'a']++;

}

for(int i=0; i<ransomNote.length();i++)

{

char ch=ransomNote.charAt(i);

if(map[ch-'a']<=0) return false;

map[ch-'a']--;

}

return true;

}

}

### Basic Calculator

[My Submissions](https://leetcode.com/problems/basic-calculator/submissions/)

QuestionEditorial Solution

Total Accepted: **27613** Total Submissions: **123847** Difficulty: **Hard**

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open ( and closing parentheses ), the plus + or minus sign -, **non-negative** integers and empty spaces .

You may assume that the given expression is always valid.

Some examples:

"1 + 1" = 2

" 2-1 + 2 " = 3

"(1+(4+5+2)-3)+(6+8)" = 23

**Note:** **Do not** use the eval built-in library function.

Simple iterative solution by identifying characters one by one. One important thing is that the input is valid, which means the parentheses are always paired and in order. Only 5 possible input we need to pay attention:

1. digit: it should be one digit from the current number
2. '+': number is over, we can add the previous number and start a new number
3. '-': same as above
4. '(': push the previous result and the sign into the stack, set result to 0, just calculate the new result within the parenthesis.
5. ')': pop out the top two numbers from stack, first one is the sign before this pair of parenthesis, second is the temporary result before this pair of parenthesis. We add them together.

Finally if there is only one number, from the above solution, we haven't add the number to the result, so we do a check see if the number is zero.

**public** **int** calculate(String s) {

Stack<Integer> **stack** = **new** Stack<Integer>();

**int** result = 0;

**int** number = 0;

**int** sign = 1;

**for**(**int** i = 0; i < s.length(); i++){

**char** c = s.charAt(i);

**if**(Character.isDigit(c)){

number = 10 \* number + (**int**)(c - '0');

}**else** **if**(c == '+'){

result += sign \* number;

number = 0;

sign = 1;

}**else** **if**(c == '-'){

result += sign \* number;

number = 0;

sign = -1;

}**else** **if**(c == '('){

//we push the result first, then sign;

**stack**.push(result);

**stack**.push(sign);

//reset the sign and result for the value in the parenthesis

sign = 1;

result = 0;

}**else** **if**(c == ')'){

result += sign \* number;

number = 0;

result \*= **stack**.pop(); //stack.pop() is the sign before the parenthesis

result += **stack**.pop(); //stack.pop() now is the result calculated before the parenthesis

}

}

**if**(number != 0) result += sign \* number;

**return** result;

}

Thanks for your answer. The following answer might be more slick (18ms):

**Principle**:

1. (Sign before '+'/'-') = (This context sign);
2. (Sign after '+'/'-') = (This context sign) \* (1 or -1);

**Algorithm:**

1. Start from +1 sign and scan s from left to right;
2. if c == digit: This number = Last digit \* 10 + This digit;
3. if c == '+': Add num to result before this sign; This sign = Last context sign \* 1; clear num;
4. if c == '-': Add num to result before this sign; This sign = Last context sign \* -1; clear num;
5. if c == '(': Push this context sign to stack;
6. if c == ')': Pop this context and we come back to **last context**;
7. Add the last num. This is because we only add number after '+' / '-'.

**Implementation**:

这个方法不错,类似于平时操作的拆括号。首先记录下括号前的运算符。

遇到+/-符号，需要做计算（将上一个运算符\*数累加到result）；同时更新sign

遇到左括号，需要入栈sign（前一个运算符号）

遇到右括号，出栈sign

**public** **int** calculate(String s) {

**if**(s == null) **return** 0;

**int** result = 0;

**int** sign = 1;

**int** num = 0;

Stack<Integer> **stack** = **new** Stack<Integer>();

**stack**.push(sign); //stack中放括号前的符号（遇到左括号时入栈sign）

**for**(**int** i = 0; i < s.length(); i++) {

**char** c = s.charAt(i);

**if**(c >= '0' && c <= '9') {

num = num \* 10 + (c - '0');

} **else** **if**(c == '+' || c == '-') {

result += sign \* num;

sign = **stack**.peek() \* (c == '+' ? 1: -1);

num = 0;

} **else** **if**(c == '(') {

**stack**.push(sign);

} **else** **if**(c == ')') {

**stack**.pop();

}

}

result += sign \* num;

**return** result;

}

**public** **static** **int** calculate(String s) {

**int** len = s.length(), sign = 1, result = 0;

Stack<Integer> **stack** = **new** Stack<Integer>();

**for** (**int** i = 0; i < len; i++) {

**if** (Character.isDigit(s.charAt(i))) {

**int** sum = s.charAt(i) - '0';

**while** (i + 1 < len && Character.isDigit(s.charAt(i + 1))) {

sum = sum \* 10 + s.charAt(i + 1) - '0';

i++;

}

result += sum \* sign;

} **else** **if** (s.charAt(i) == '+')

sign = 1;

**else** **if** (s.charAt(i) == '-')

sign = -1;

**else** **if** (s.charAt(i) == '(') {

**stack**.push(result);

**stack**.push(sign);

result = 0;

sign = 1;

} **else** **if** (s.charAt(i) == ')') {

result = result \* **stack**.pop() + **stack**.pop();

}

}

**return** result;

}

### Basic Calculator II

[My Submissions](https://leetcode.com/problems/basic-calculator-ii/submissions/)

QuestionEditorial Solution

Total Accepted: **21944** Total Submissions: **87515** Difficulty: **Medium**

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only **non-negative** integers, +, -, \*, / operators and empty spaces . The integer division should truncate toward zero.

You may assume that the given expression is always valid.

Some examples:

"3+2\*2" = 7

" 3/2 " = 1

" 3+5 / 2 " = 5

**Note:** **Do not** use the eval built-in library function.

这个方法好

要考虑是最后一位数的情况

如果遇到数字，则记录；

如果遇到非数字和空格，或者已经到字符串的最后一位

（遇到符号，才会有出入栈。根据前一个符号的情况，判断数字的入栈出栈情况）

前一个符号为+，则将数字入栈；

前一个符号为-，则将数字的反，入栈；

前一个符号为\*、/，将栈中元素pop一个与数字相乘（相除），再入栈

num记录当前数字

**public** **class** Solution {

**public** **int** calculate(String s) {

**int** len;

**if**(s==null || (len = s.length())==0) **return** 0;

Stack<Integer> **stack** = **new** Stack<Integer>();

**int** num = 0;

**char** sign = '+';

**for**(**int** i=0;i<len;i++){

**if**(Character.isDigit(s.charAt(i))){

num = num\*10+s.charAt(i)-'0';

}

**if**((!Character.isDigit(s.charAt(i)) &&' '!=s.charAt(i)) || i==len-1){ //不是数字和空格，或者i是最后一个

**if**(sign=='-'){

**stack**.push(-num);

}

**if**(sign=='+'){

**stack**.push(num);

}

**if**(sign=='\*'){

**stack**.push(**stack**.pop()\*num);

}

**if**(sign=='/'){

**stack**.push(**stack**.pop()/num);

}

sign = s.charAt(i);

num = 0;

}

}

**int** re = 0;

**for**(**int** i:**stack**){

re += i;

}

**return** re;

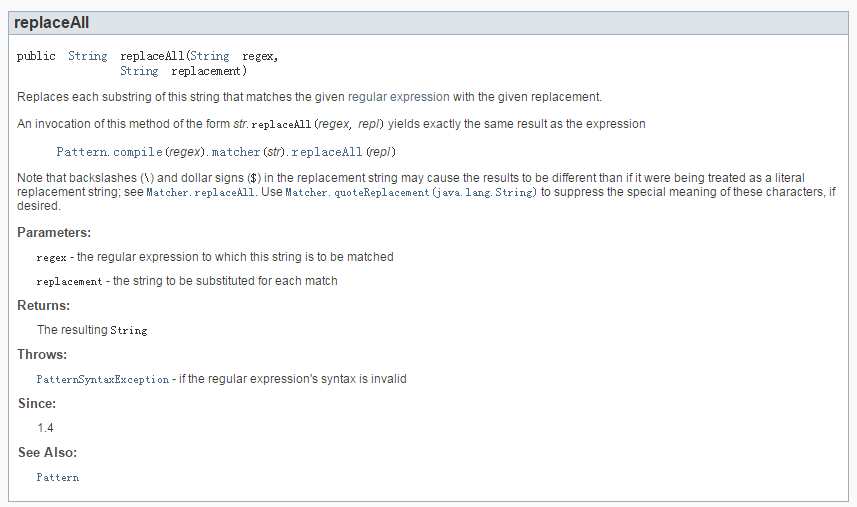
}

}

这个方法也不错，类似上一个方法，只是是O(1)空间

维护栈顶一个元素即可。然后不断累加结果

String方法



public int calculate(String s) {

**if** (s == null) **return** 0;

s = s.trim().replaceAll(" +", "");

int length = s.length();

int res = 0;

long preVal = 0; // initial preVal is 0 //相当于栈顶

char sign = '+'; // initial sign is +

int i = 0;

**while** (i < length) {

//if(s.charAt(i)==' ') {i++;continue;} //不明白replaceAll那句，可以使用这句代替

long curVal = 0;

**while** (i < length && (int)s.charAt(i) <= 57 && (int)s.charAt(i) >= 48) { // int

curVal = curVal\*10 + (s.charAt(i) - '0');

i++;

}

**if** (sign == '+') {

res += preVal; // update res

preVal = curVal;

} **else** **if** (sign == '-') {

res += preVal; // update res

preVal = -curVal;

} **else** **if** (sign == '\*') {

preVal = preVal \* curVal; // **not** update res, combine preVal & curVal **and** keep loop

} **else** **if** (sign == '/') {

preVal = preVal / curVal; // **not** update res, combine preVal & curVal **and** keep loop

}

**if** (i < length) { // getting new sign

sign = s.charAt(i);

i++;

}

}

res += preVal;

**return** res;

}

Share my clean java solution, O(1) space, no stack, beats 99.72%

**public** **class** Solution {

**public** **int** calculate(String s) {

**if**(s == null || s.length() == 0)

**return** 0;

boolean divide = **false**;

**int** result = 0, sign = 1, num = 0, preNum = 0;

**for**(**char** c: s.toCharArray()) {

**if**(c >= '0' && c <= '9')

num = num \* 10 + c -'0';

**else** **if**(c == '+' || c == '-' || c == '\*' || c == '/') {

**if**(divide) {

num = preNum/num;

divide = **false**;

}

//record the temp result, think about case 2 \* 5 / 2

**if**(c == '/') {

divide = **true**;

preNum = num \* sign;

sign = 1;

} **else** **if**(c == '\*'){

sign \*= num;

} **else** {

result += sign \* num;

sign = c == '+' ? 1 : -1;

}

num = 0;

}

}

**if**(num > 0) {

**if**(divide)

num = preNum/num;

result += sign \* num;

}

**return** result;

}

}

### Longest Common Prefix

[My Submissions](https://leetcode.com/problems/longest-common-prefix/submissions/)

QuestionEditorial Solution

Total Accepted: **98928** Total Submissions: **350632** Difficulty: **Easy**

Write a function to find the longest common prefix string amongst an array of strings.

注意题目意思，只需要全部字符串一起比较

public String longestCommonPrefix(String[] strs) {

**if**(strs == null || strs.**length** == 0) **return** "";

String pre = strs[0];

**int** i = 1;

**while**(i < strs.**length**){

**while**(strs[i].indexOf(pre) != 0)

pre = pre.substring(0,pre.**length**()-1);

i++;

}

**return** pre;

}

Rather than building the string for common prefix, you can just record the longest prefix's length, then return the substring from first character till that length, which can save a bit time perhaps.

The code is as below written in javascript:

var longestCommonPrefix = function(strs) {

**if** (strs.**length** === 0) **return** "";

**if** (strs.**length** === 1) **return** strs[0];

**for** (var i = 0; i < strs[0].**length**; i++) {

**for** (var j = 1; j < strs.**length**; j++) {

**if** (strs[j].**length** <= i || strs[j][i] !== strs[0][i]) {

**return** strs[0].substring(0, i);

}

}

}

**return** strs[0];

};

console.**log**(longestCommonPrefix(['this is a test', 'this is also a test']));

console.**log**(longestCommonPrefix(['this is a test', 'this is a test as well']));

public String longestCommonPrefix(String[] strs) {

**if** (strs == null) **return** null;

**if** (strs.length == 0) **return** "";

Arrays.sort(strs);

char[] first = strs[0].toCharArray();

char[] last = strs[strs.length - 1].toCharArray();

int i = 0, len = Math.min(first.length, last.length);

**while** (i < len && first[i] == last[i]) i++;

**return** strs[0].substring(0, i);

}

I think there are two (straightforward) solutions:

1) "Horizontal matching (over strings)". Pick up the first string and compare it with the rest. Return the minimum prefix found among all comparisons.

2) "Vertical matching (over characters)". Compare the characters between all strings from left to right. Stop whenever a mismatch is found.

What is the complexity of these two approaches ?

Is there a better solution ?

The first one is apparently not very optimal. Imagine only the last string is different from all others. You would have wasted so much time comparing the previous ones. The second one is the optimal solution. The question is equivalent to: what is the fastest way to fail? It seems the second one is the only answer.

The following is a possible solution using Java:

这个写法好

public String longestCommonPrefix(String[] strs) {

**if** (strs == null || strs.**length** == 0)

**return** "";

**for** (**int** i = 0; i < strs[0].**length**() ; i++){

char c = strs[0].charAt(i);

**for** (**int** j = 1; j < strs.**length**; j ++) {

**if** (i == strs[j].**length**() || strs[j].charAt(i) != c)

**return** strs[0].substring(0, i);

}

}

**return** strs[0];

}

It seems that it is not to check between pair of strings but on all the strings in the array.

For example:

1. {"a","a","b"} should give "" as there is nothing common in all the 3 strings.
2. {"a", "a"} should give "a" as a is longest common prefix in all the strings.
3. {"abca", "abc"} as abc
4. {"ac", "ac", "a", "a"} as a.

Logic goes something like this:

1. Pick a character at i=0th location and compare it with the character at that location in every string.
2. If anyone doesn't have that just return ""
3. Else append that character in to the result.
4. Increment i and do steps 1-3 till the length of that string.
5. return result.

Make sure proper checks are maintained to avoid index out of bounds error.

### Longest Palindromic Substring

[My Submissions](https://leetcode.com/problems/longest-palindromic-substring/submissions/)

QuestionEditorial Solution

Total Accepted: **107659** Total Submissions: **465599** Difficulty: **Medium**

Given a string *S*, find the longest palindromic substring in *S*. You may assume that the maximum length of *S* is 1000, and there exists one unique longest palindromic substring.

The performance is pretty good, surprisingly.

这个写法可以

以i处开始用两个指针分别向两边扩展，看是否是回文，并记录最大长度及index（回文串长度为奇数时，以i点为对称点）

当回文串是偶数时，从i，i+1处向两边扩展

**public** **class** **Solution** {

**private** **int** lo, maxLen;

**public** String longestPalindrome(String s) {

**int** len = s.length();

**if** (len < 2)

**return** s;

**for** (**int** i = 0; i < len-1; i++) {

extendPalindrome(s, i, i); //assume odd length, try to extend Palindrome as possible

extendPalindrome(s, i, i+1); //assume even length.

}

**return** s.substring(lo, lo + maxLen);

}

**private** **void** extendPalindrome(String s, **int** j, **int** k) {

**while** (j >= 0 && k < s.length() && s.charAt(j) == s.charAt(k)) {

j--;

k++;

}

**if** (maxLen < k - j - 1) {

lo = j + 1;

maxLen = k - j - 1;

}

}}

**string** longestPalindrome(**string** s) {

**if** (s.empty()) **return** "";

**if** (s.size() == 1) **return** s;

**int** min\_start = 0, max\_len = 1;

**for** (**int** i = 0; i < s.size();) {

**if** (s.size() - i <= max\_len / 2) **break**;

**int** j = i, k = i;

**while** (k < s.size()-1 && s[k+1] == s[k]) ++k; // Skip duplicate characters.

i = k+1;

**while** (k < s.size()-1 && j > 0 && s[k + 1] == s[j - 1]) { ++k; --j; } // Expand.

**int** new\_len = k - j + 1;

**if** (new\_len > max\_len) { min\_start = j; max\_len = new\_len; }

}

**return** s.substr(min\_start, max\_len);

}

**Key idea, every time we move to right, we only need to consider whether using this new character as tail could produce new palindrome string of length (current length +1) or (current length +2)**

**public** **class** **Solution** {

**public** String longestPalindrome(String s) {

String res = "";

**int** currLength = 0;

**for**(**int** i=0;i<s.length();i++){

**if**(isPalindrome(s,i-currLength-1,i)){

res = s.substring(i-currLength-1,i+1);

currLength = currLength+2;

}

**else** **if**(isPalindrome(s,i-currLength,i)){

res = s.substring(i-currLength,i+1);

currLength = currLength+1;

}

}

**return** res;

}

**public** **boolean** isPalindrome(String s, **int** begin, **int** end){

**if**(begin<0) **return** **false**;

**while**(begin<end){

**if**(s.charAt(begin++)!=s.charAt(end--)) **return** **false**;

}

**return** **true**;

}

}

For friends who are confused about the key idea to check only new palindrome with length = current length +2 or +1, I add some more explanation here.

Example: "xxxbcbxxxxxa", (x is random character, **not** all x are equal) now we

are dealing with the last character 'a'. The current longest palindrome

is "bcb" with length 3.

1. check "xxxxa" so **if** it is palindrome we could get a new palindrome of length 5.

2. check "xxxa" so **if** it is palindrome we could get a new palindrome of length 4.

3. **do** NOT check "xxa" **or** any shorter string since the length of the new string is

no bigger than current longest length.

4. **do** NOT check "xxxxxa" **or** any longer string because **if** "xxxxxa" is palindrome

**then** "xxxx" got from cutting off the head **and** tail is also palindrom. It has

length > 3 which is impossible.'

Explanation for those interested. For every position i we're interested if there's a palindrome ending at this position (inclusive) longer than the longest palindrome found so far.

For i = 0 the only palindrome ending there is the palindrome of length 1. So the new palindrome has length 1 and therefore the maximum increases by 1.

For every i > 0 we can have many palindromes ending there. We're only interested in those with length >= 2 because we've already found one of length 1. Some of palindromes may have even lengths, some odd. The presence of a palindrome of length len >= 2 ending at iimplies three things:

1. i - len + 1 >= 0 (obviously).
2. s[i - len + 1] == s[i].
3. s[i - len + 2 .. i - 1] is a palindrome. This is the most interesting fact because it means that if we have found a palindrome of length len - 2 ending at i - 1, then we may find another one of length len ending at i. So incrementing i by 1 can lead to increase of the palindrome length by 2. One more important corollary: if we find a palindrome of length len - 2 at i - 1, then we only need to check the conditions (1) and (2) above on the next iteration—we may skip the isPalindrome() call.

Now consider the following loop invariant: just prior to the first iteration, and right after each iteration (after incrementing i) max is the length of the longest palindrome substring within the substring s[0..i) (exclusive). Let's deal with it rigorously, CLR-style:

1. Initialization. This is obviously true for i = 0 if we initialize max = 0. The longest palindrome substring of an empty string s[0..0) is an empty string.
2. Maintenance. Suppose max is the longest palindrome substring of s[0..i). Then fors[0..i+1) we only need to consider palindromes ending at s[i] (inclusive), because we've checked all others already. From the fact (3) above it follows that we can only find palindromes of length max + 1 or max + 2 ending there. Indeed, if we were able to find a palindrome of length max + l (l > 2) ending at i, then it would mean that there is a palindrome of length max + l - 2 > max ending at i - 1, which contradicts the assumption that max is the maximum length of a palindrome found so far.
3. Termination. Since the substring s[0..N) (N = length of s) is the string s itself, thenmax is the length of the longest palindrome substring. If we keep track of starts/ends, we can easily get the substring itself.

My code based on the above is given below. It differs from the solution presented in the question only in that it tries to optimize palindrome checks by using the fact (3) for the case when the previous palindrome is at i - 1. This improves performance by 2 ms (total 7 ms, beating 99%).

public String longestPalindrome(String s) {

int start = 0, **end** = 0;

char[] cs = s.toCharArray();

**for** (int i = 0, max = 0, prev = 0; i < cs.length; ++i) {

**if** (i - max - 1 >= 0 && cs[i - max - 1] == cs[i]

&& (prev == i - 1 || isPalindrome(cs, i - max, i))) {

start = i - max - 1;

**end** = i + 1;

max += 2;

prev = i;

} **else** **if** (isPalindrome(cs, i - max, i + 1)) {

start = i - max;

**end** = i + 1;

++max;

prev = i;

}

}

**return** s.substring(start, **end**);

}

private static boolean isPalindrome(char[] cs, int start, int **end**) {

**for** (int i = start, j = **end** - 1; i < j; ++i, --j) {

**if** (cs[i] != cs[j]) {

**return** **false**;

}

}

**return** **true**;

}

总结：字符串的很多题目，比如要修改字符串，则可以转成字符数组，处理后再转成字符串。

char[] arr=str.toCharArray();

new String(arr);

## 数组

### Rotate Array

[My Submissions](https://leetcode.com/problems/rotate-array/submissions/)

QuestionEditorial Solution

Total Accepted: **69480** Total Submissions: **329596** Difficulty: **Easy**

Rotate an array of *n* elements to the right by *k* steps.

For example, with *n* = 7 and *k* = 3, the array [1,2,3,4,5,6,7] is rotated to [5,6,7,1,2,3,4].

**Note:**  
Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.

[[show hint]](https://leetcode.com/problems/rotate-array/)

**Hint:**  
Could you do it in-place with O(1) extra space?

Related problem: [Reverse Words in a String II](https://leetcode.com/problems/reverse-words-in-a-string-ii/)

The basic idea is that, for example, nums = [1,2,3,4,5,6,7] and k = 3, first we reverse [1,2,3,4], it becomes[4,3,2,1]; then we reverse[5,6,7], it becomes[7,6,5], finally we reverse the array as a whole, it becomes[4,3,2,1,7,6,5] ---> [5,6,7,1,2,3,4].

Reverse is done by using two pointers, one point at the head and the other point at the tail, after switch these two, these two pointers move one position towards the middle.

I really don't like those *something little* line solutions as they are incredibly hard to read. Below is my solution.

public void rotate(**int**[] nums, **int** k) {

k %= nums.**length**;

**reverse**(nums, 0, nums.**length** - 1);

**reverse**(nums, 0, k - 1);

**reverse**(nums, k, nums.**length** - 1);

}

public void **reverse**(**int**[] nums, **int** start, **int** end) {

**while** (start < end) {

**int** temp = nums[start];

nums[start] = nums[end];

nums[end] = temp;

start++;

end--;

}

}

Make an extra copy and then rotate.

Time complexity: O(n). Space complexity: O(n).

**class** Solution

{

**public**:

**void** rotate(**int** nums[], **int** n, **int** k)

{

**if** ((n == 0) || (k <= 0))

{

**return**;

}

// Make a copy of nums

**vector**<**int**> numsCopy(n);

**for** (**int** i = 0; i < n; i++)

{

numsCopy[i] = nums[i];

}

// Rotate the elements.

**for** (**int** i = 0; i < n; i++)

{

nums[(i + k)%n] = numsCopy[i];

}

}

};

Start from one element and keep rotating until we have rotated n different elements.

Time complexity: O(n). Space complexity: O(1).

**class** **Solution**

{

**public**:

**void** rotate(**int** nums[], **int** n, **int** k)

{

**if** ((n == 0) || (k <= 0))

{

**return**;

}

**int** cntRotated = 0;

**int** start = 0;

**int** curr = 0;

**int** numToBeRotated = nums[0];

**int** tmp = 0;

// Keep rotating the elements until we have rotated n

// different elements.

**while** (cntRotated < n)

{

do

{

tmp = nums[(curr + k)%n];

nums[(curr+k)%n] = numToBeRotated;

numToBeRotated = tmp;

curr = (curr + k)%n;

cntRotated++;

} **while** (curr != start);

// Stop rotating the elements when we finish one cycle,

// i.e., we return to start.

// Move to next element to start a new cycle.

start++;

curr = start;

numToBeRotated = nums[curr];

}

}

};

Reverse the first n - k elements, the last k elements, and then all the n elements.

Time complexity: O(n). Space complexity: O(1).

**class** **Solution**

{

**public**:

**void** rotate(**int** nums[], **int** n, **int** k)

{

k = k%n;

// Reverse the first n - k numbers.

// Index i (0 <= i < n - k) becomes n - k - i.

reverse(nums, nums + n - k);

// Reverse tha last k numbers.

// Index n - k + i (0 <= i < k) becomes n - i.

reverse(nums + n - k, nums + n);

// Reverse all the numbers.

// Index i (0 <= i < n - k) becomes n - (n - k - i) = i + k.

// Index n - k + i (0 <= i < k) becomes n - (n - i) = i.

reverse(nums, nums + n);

}

};

4. Swap the last k elements with the first k elements.

Time complexity: O(n). Space complexity: O(1).

**class** **Solution**

{

**public**:

**void** rotate(**int** nums[], **int** n, **int** k)

{

**for** (; k = k%n; n -= k, nums += k)

{

// Swap the last k elements with the first k elements.

// The last k elements will be in the correct positions

// but we need to rotate the remaining (n - k) elements

// to the right by k steps.

**for** (**int** i = 0; i < k; i++)

{

swap(nums[i], nums[n - k + i]);

}

}

}

};

5. Keep swapping two subarrays.

Time complexity: O(n). Space complexity: O(1).

**class** **Solution**

{

**public**:

**void** rotate(**int** nums[], **int** n, **int** k)

{

**if** ((n == 0) || (k <= 0) || (k%n == 0))

{

**return**;

}

k = k%n;

// Rotation to the right by k steps is equivalent to swapping

// the two subarrays nums[0,...,n - k - 1] and nums[n - k,...,n - 1].

**int** start = 0;

**int** tmp = 0;

**while** (k > 0)

{

**if** (n - k >= k)

{

// The left subarray with size n - k is longer than

// the right subarray with size k. Exchange

// nums[n - 2\*k,...,n - k - 1] with nums[n - k,...,n - 1].

**for** (**int** i = 0; i < k; i++)

{

tmp = nums[start + n - 2\*k + i];

nums[start + n - 2\*k + i] = nums[start + n - k + i];

nums[start + n - k + i] = tmp;

}

// nums[n - 2\*k,...,n - k - 1] are in their correct positions now.

// Need to rotate the elements of nums[0,...,n - k - 1] to the right

// by k%n steps.

n = n - k;

k = k%n;

}

**else**

{

// The left subarray with size n - k is shorter than

// the right subarray with size k. Exchange

// nums[0,...,n - k - 1] with nums[n - k,...,2\*(n - k) - 1].

**for** (**int** i = 0; i < n - k; i++)

{

tmp = nums[start + i];

nums[start + i] = nums[start + n - k + i];

nums[start + n - k + i] = tmp;

}

// nums[n - k,...,2\*(n - k) - 1] are in their correct positions now.

// Need to rotate the elements of nums[n - k,...,n - 1] to the right

// by k - (n - k) steps.

tmp = n - k;

n = k;

k -= tmp;

start += tmp;

}

}

}

};

[solution-sharing](https://leetcode.com/discuss/tag/solution-sharing)

**Move Zeroes**

Total Accepted: **74807** Total Submissions: **169974** Difficulty: **Easy**

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.

**Simple O(N) Java Solution Using Insert Index**

// Shift non-zero values as far forward as possible

// Fill remaining space with zeros

public void moveZeroes(int[] nums) {

if (nums == null || nums.length == 0) return;

int insertPos = 0;

for (int num: nums) {

if (num != 0) nums[insertPos++] = num;

}

while (insertPos < nums.length) {

nums[insertPos++] = 0;

}

}

public void moveZeroes(int[] nums) {

int cur = 0;

for (int i = 0; i < nums.length; ++i) {

if (nums[i] != 0) {

int temp = nums[cur];

nums[cur++] = nums[i];

nums[i] = temp;

}

}

}

这个方法很好，只需要记录非0的数，然后直接在最后面补上0.

**A 95.26% beat rate solution**

class Solution {

public:

void moveZeroes(vector<int>& nums) {

int j = 0;

// move all the nonzero elements advance

for (int i = 0; i < nums.size(); i++) {

if (nums[i] != 0) {

nums[j++] = nums[i];

}

}

for (;j < nums.size(); j++) {

nums[j] = 0;

}

}

};

### Remove Element

Total Accepted: **113635** Total Submissions: **337053** Difficulty: **Easy**

Given an array and a value, remove all instances of that value in place and return the new length.

Do not allocate extra space for another array, you must do this in place with constant 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.

**Hint:**

1. Try two pointers.
2. Did you use the property of "the order of elements can be changed"?
3. What happens when the elements to remove are rare?

和上一题类似，只是不需要管数组后面的部分

**int** removeElement(**int** A[], **int** n, **int** elem) {

**int** begin=0;

**for**(**int** i=0;i<n;i++) **if**(A[i]!=elem) A[begin++]=A[i];

**return** begin;

}

### Remove Duplicates from Sorted Array

[My Submissions](https://leetcode.com/problems/remove-duplicates-from-sorted-array/submissions/)

QuestionEditorial Solution

Total Accepted: **126033** Total Submissions: **377161** Difficulty: **Easy**

Given a sorted array, remove the duplicates 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. It doesn't matter what you leave beyond the new length.

我的做法，依然类似上面一题

用cnt指针指向下一个数字可以放置的位子

public int removeDuplicates(int[] nums) {

if(nums==null || nums.length==0) return 0;

int num=nums[0];

int cnt=1;

for(int n:nums)

{

if(n!=num)

{

nums[cnt++]=n;

num=n;

}

}

return cnt;

}

My Solution : Time O(n), Space O(1)

这个写法不错

**class** **Solution** {

**public**:

**int** removeDuplicates(**int** A[], **int** n) {

**if**(n < 2) **return** n;

**int** id = 1;

**for**(**int** i = 1; i < n; ++i)

**if**(A[i] != A[i-1]) A[id++] = A[i];

**return** id;

}

};

**C++ code**

int count = 0;

**for**(int i = 1; i < n; i++){

**if**(A[i] == A[i-1]) count++;

**else** A[i-count] = A[i];

}

**return** n-count;

**Java**

**public** **int** removeDuplicates(**int**[] nums) {

**int** i = 0;

**for** (**int** n : nums)

**if** (i == 0 || n > nums[i-1])

nums[i++] = n;

**return** i;

}

And to not need the i == 0 check in the loop:

**public** **int** removeDuplicates(**int**[] nums) {

**int** i = nums.length > 0 ? 1 : 0;

**for** (**int** n : nums)

**if** (n > nums[i-1])

nums[i++] = n;

**return** i;

}

### Remove Duplicates from Sorted Array II

[My Submissions](https://leetcode.com/problems/remove-duplicates-from-sorted-array-ii/submissions/)

QuestionEditorial Solution

Total Accepted: **73563** Total Submissions: **224757** Difficulty: **Medium**

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.

我的做法，类似上面一题，但是有个地方需要注意，不同于上面一题，这里判断nums[i]是否与前面有重复时，需要使用新生成的nums数组部分，不能使用之前的

比如，1,1,1,2,2,3

当i=3时，数组为1,1,2,2,2,3 cnt=3

当i=4时，数组为1,1,2,2,2,3 cnt=3

当i=5时，数组为1,1,2,3,2,3 cnt=4

public int removeDuplicates(int[] nums) {

int cnt=2;

for(int i=2;i<nums.length;i++)

{

if(nums[i]!=nums[cnt-1] || nums[i]!=nums[cnt-2]) nums[cnt++]=nums[i];

//System.out.println("i:"+i+" "+Arrays.toString(nums)+" cnt"+cnt);

}

return cnt;

}

Same simple solution written in several languages. Just go through the numbers and include those in the result that haven't been included twice already.

**Java**

**这个写法很不错，i是指针（**nums[i-2]是有效数组的倒数第二个数**），当前数字只需要和**有效数组的倒数第二个数进行比较，大于则可以加入数组。和我的做法类似，写法更加简洁

**public** **int** removeDuplicates(**int**[] nums) {

**int** i = 0;

**for** (**int** n : nums)

**if** (i < 2 || n > nums[i-2])

nums[i++] = n;

**return** i;

}

下面这个写法不好，如果需要处理重复数为k时，也只需要修改上面的代码

Share my O(N) time and O(1) solution when duplicates are allowed at most K times

I think both Remove Duplicates from Sorted Array I and II could be solved in a consistent and more general way by allowing the duplicates to appear k times (k = 1 for problem I and k = 2 for problem II). Here is my way: we need a count variable to keep how many times the duplicated element appears, if we encounter a different element, just set counter to 1, if we encounter a duplicated one, we need to check this count, if it is already k, then we need to skip it, otherwise, we can keep this element. The following is the implementation and can pass both OJ:

**int** removeDuplicates(**int** A[], **int** n, **int** k) {

**if** (n <= k) **return** n;

**int** i = 1, j = 1;

**int** cnt = 1;

**while** (j < n) {

**if** (A[j] != A[j-1]) {

cnt = 1;

A[i++] = A[j];

}

**else** {

**if** (cnt < k) {

A[i++] = A[j];

cnt++;

}

}

++j;

}

**return** i;

}

For more details, you can also see this post: [LeetCode Remove Duplicates from Sorted Array I and II: O(N) Time and O(1) Space](http://tech-wonderland.net/blog/leetcode-remove-duplicates-from-sorted-array-i-and-ii.html)

public **int** removeDuplicates(**int**[] nums) {

**if** (nums.**length** == 0) {**return** 0;}

**int** pointer = 0, flag = 0;

**for** (**int** i = 1; i < nums.**length**; i++) {

**if** (nums[i] == nums[i - 1] && flag == 0) {

flag = 1;

pointer++;

} **else** **if** (nums[i] != nums[i - 1]) {

flag = 0;

pointer++;

}

nums[pointer] = nums[i];

}

**return** pointer + 1;

}

The variable flag is to show if this number (nums[i]) has appeared more or equals to third times. The variable pointer is the location that each number (nums[i]) should appeared in. The special case is when the array is empty.

### Plus One

[My Submissions](https://leetcode.com/problems/plus-one/submissions/)

QuestionEditorial Solution

Total Accepted: **96667** Total Submissions: **288773** Difficulty: **Easy**

Given a non-negative number represented as an array of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

**void** plusone(**vector**<**int**> &digits)

{

**int** n = digits.size();

**for** (**int** i = n - 1; i >= 0; --i)

{

**if** (digits[i] == 9)

{

digits[i] = 0;

}

**else**

{

digits[i]++;

**return**;

}

}

digits[0] =1;

digits.push\_back(0);

}

这个方法不错

因为只需要考虑加1的情况

从最低位开始判断，如果不是9则可直接该位加1，返回最终结果。否则还得继续判断其他的位。

如果到循环结束还没有返回结果，则说明是99s的数，则创建新数组，最高位为0

**public** **int**[] plusOne(**int**[] digits) {

**int** n = digits.length;

**for**(**int** i=n-1; i>=0; i--) {

**if**(digits[i] < 9) {

digits[i]++;

**return** digits;

}

digits[i] = 0;

}

**int**[] newNumber = **new** **int** [n+1];

newNumber[0] = 1;

**return** newNumber;

}

The last part of code is only for the case that the whole input array is 9s. For example : 99999-----> 100000 Any other case would return in the loop.

### Majority Element

[My Submissions](https://leetcode.com/problems/majority-element/submissions/)

QuestionEditorial Solution

Total Accepted: **108394** Total Submissions: **263888** Difficulty: **Easy**

Given an array of size *n*, find the majority element. The majority element is the element that appears **more than** ⌊ n/2 ⌋ times.

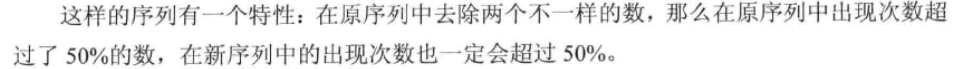
You may assume that the array is non-empty and the majority element always exist in the array.

向下取整的运算称为Floor，用数学符号⌊⌋表示；向上取整的运算称为Ceiling，用数学符号⌈⌉表示。

在C语言中整数除法取的既不是Floor也不是Ceiling，无论操作数是正是负总是把小数部分截掉，在数轴上向零的方向取整（Truncate toward Zero），或者说当操作数为正的时候相当于Floor，当操作数为负的时候相当于Ceiling。

神奇算法的思路：（存在一个计数器count和一个临时存储当前元素的变量now）

1. 如果count==0，则将now的值设置为数组的当前元素，将count赋值为1；
2. 反之，如果now和现在数组元素值相同，则count++，反之count--；
3. 重复上述两步，直到扫描完数组。
4. count赋值为0，再次从头扫描数组，如果数组元素值与now的值相同则count++，直到扫描完数组为止。
5. 如果此时count的值大于等于n/2，则返回now的值，反之则返回-1；





解法：  
声明一个变量count = 0，声明一个常量size等于数组大小。  
假设该数组的第一个元素a(1)为主元素，让其与a(2)进行比较，若相同，则使变量count+1，若不同，则count-1。然后继续比较a(3)。以此类推。  
  
当与a(n)比较后，count = -1时，将count重新归为0，并重新假设a(n+1)为主元素，并继续与a(n+2)作比较。  
  
当count>=(size-m)/2时，此时假设的主元素a(m)即为实际的主元素。  
或遍历完整个数组后，当前假设的主元素为实际主元素。  
  
这个算法的时间复杂度最大才O(N)，看书看到这一段时令我顿时拍案叫绝啊。其核心思想在于：对于这样一个数组，去除掉任意两个不相等的数，剩下的数中，主元素的出现频率仍然大于50%。而使用count来进行加减计数，当count=0时，必然是偶数个数与假设的主元素进行了比较，且其中有一半与假设数相同一半与假设数不同（当count=-1时，加上假设数的集合，也满足该条件）。

/\*\*

**判断输入的数列是否有主元素**

\*/  
private static boolean hasMaster(int data[], int n)  
{  
int count=0; //保存计数  
int seed; //保存参照元素  
  
seed = data[0];  
  
for(int i=1; i  
{  
if(seed == data[i]) //如果数据相同，计数加一  
{  
count++;  
}  
if(seed == data[i])  
{  
if(count>0)  
{  
count--; //如果数据不同，则计数减一  
//相当于删除两个不同的元素  
//不会对主元素造成影响  
}  
else  
{  
seed = data[i]; //计数为零时，seed不可能为主元素  
//读入新数据  
}  
} //end of if  
} //end of for  
  
//因为最终得到的seed元素有可能是序列最末位的两个元素之一  
//因此，这里还需要验证  
count = 0;  
for(int i=0; i  
{  
if (seed == data[i])  
count++;  
}  
  
if(count>(n/2))  
{  
return true;  
}  
  
return false;   
  
}

这个写法不错，核心就是如果两个元素不同，则同时删去，主元素依然是剩下元素中的主元素，迭代可得到。

count为0则不可能是主元素，则需要新赋值

O(n) time O(1) space fastest solution

**public** **class** **Solution** {

**public** **int** majorityElement(**int**[] num) {

**int** major=num[0], count = 1;

**for**(**int** i=1; i<num.length;i++){

**if**(count==0){

count++;

major=num[i];

}**else** **if**(major==num[i]){

count++;

}**else** count--;

}

**return** major;

}

}

**Java solution**

// Sorting

**public** **int** majorityElement1(**int**[] nums) {

Arrays.sort(nums);

**return** nums[nums.length/2];

}

// Hashtable

**public** **int** majorityElement2(**int**[] nums) {

Map<Integer, Integer> myMap = **new** HashMap<Integer, Integer>();

//Hashtable<Integer, Integer> myMap = new Hashtable<Integer, Integer>();

**int** ret=0;

**for** (**int** num: nums) {

**if** (!myMap.containsKey(num))

myMap.put(num, 1);

**else**

myMap.put(num, myMap.**get**(num)+1);

**if** (myMap.**get**(num)>nums.length/2) {

ret = num;

**break**;

}

}

**return** ret;

}

这个写法好

// Moore voting algorithm

**public** **int** majorityElement3(**int**[] nums) {

**int** count=0, ret = 0;

**for** (**int** num: nums) {

**if** (count==0)

ret = num;

**if** (num!=ret)

count--;

**else**

count++;

}

**return** ret;

}

// Bit manipulation

**public** **int** majorityElement(**int**[] nums) {

**int**[] bit = **new** **int**[32];

**for** (**int** num: nums)

**for** (**int** i=0; i<32; i++)

**if** ((num>>(31-i) & 1) == 1)

bit[i]++;

**int** ret=0;

**for** (**int** i=0; i<32; i++) {

bit[i]=bit[i]>nums.length/2?1:0;

ret += bit[i]\*(1<<(31-i));

}

**return** ret;

}

### Majority Element II

[My Submissions](https://leetcode.com/problems/majority-element-ii/submissions/)

QuestionEditorial Solution

Total Accepted: **26382** Total Submissions: **103248** Difficulty: **Medium**

Given an integer array of size *n*, find all elements that appear more than ⌊ n/3 ⌋ times. The algorithm should run in linear time and in O(1) space.

**Hint:**

1. How many majority elements could it possibly have?
2. Do you have a better hint? [Suggest it](mailto:admin@leetcode.com?subject=Hints%20for%20Majority%20Element%20II)!

Boyer-Moore Majority Vote algorithm and my elaboration

For those who aren't familiar with Boyer-Moore Majority Vote algorithm, I found a great article (http://goo.gl/64Nams) that helps me to understand this fantastic algorithm!! Please check it out!

The essential concepts is you keep a counter for the majority number **X**. If you find a number **Y**that is not **X**, the current counter should deduce 1. The reason is that if there is 5 **X** and 4 **Y**, there would be one (5-4) more **X** than **Y**. This could be explained as "4 **X** being paired out by 4**Y**".

And since the requirement is finding the majority for more than ceiling of [n/3], the answer would be less than or equal to two numbers. So we can modify the algorithm to maintain two counters for two majorities.

Followings are my sample Python code:

**class** **Solution**:

# @param {integer[]} nums

# @return {integer[]}

**def** **majorityElement**(self, nums):

**if** **not** nums:

**return** []

count1, count2, candidate1, candidate2 = 0, 0, 0, 1

**for** n **in** nums:

**if** n == candidate1:

count1 += 1

**elif** n == candidate2:

count2 += 1

**elif** count1 == 0:

candidate1, count1 = n, 1

**elif** count2 == 0:

candidate2, count2 = n, 1

**else**:

count1, count2 = count1 - 1, count2 - 1

**return** [n **for** n **in** (candidate1, candidate2)

**if** nums.count(n) > len(nums) // 3]

Boyer-Moore Majority Vote algorithm generalization to elements appear more than floor(n/k) times

**class** Solution {

**public**:

**vector**<**int**> majorityElement(**vector**<**int**> &a) {

**int** y = 0, z = 1, cy = 0, cz = 0;

**for** (**auto** x: a) {

**if** (x == y) cy++;

**else** **if** (x == z) cz++;

**else** **if** (! cy) y = x, cy = 1;

**else** **if** (! cz) z = x, cz = 1;

**else** cy--, cz--;

}

cy = cz = 0;

**for** (**auto** x: a)

**if** (x == y) cy++;

**else** **if** (x == z) cz++;

**vector**<**int**> r;

**if** (cy > a.size()/3) r.push\_back(y);

**if** (cz > a.size()/3) r.push\_back(z);

**return** r;

}

};

**Explanation**

The basic idea is based on Moore's Voting Algorithm, we need two candidates with top 2 frequency. If meeting different number from the candidate, then decrease 1 from its count, or increase 1 on the opposite condition. Once count equals 0, then switch the candidate to the current number. The trick is that we need to count again for the two candidates after the first loop. Finally, output the numbers appearing more than n/3 times.

Thanks for [yanggao](https://leetcode.com/discuss/user/yanggao)'s smart advice!

public List<Integer> majorityElement(int[] nums) {

ArrayList<Integer> res = new ArrayList<Integer>();

**if** (nums.length==0) **return** res;

int count[] = new int[2];

int x[] = new int[2];

x[0] = 0; x[1] = 1;

**for** (int i = 0; i < nums.length; i++) {

**if** (x[0] == nums[i])

count[0]++;

**else** **if** (x[1] == nums[i])

count[1]++;

**else** **if** (count[0] == 0) {

x[0] = nums[i];

count[0] = 1;

} **else** **if** (count[1] == 0) {

x[1] = nums[i];

count[1] = 1;

} **else** {

count[0]--;

count[1]--;

}

}

Arrays.fill(count, 0);

**for** (int i : nums) {// Count again **for** x1, x2

**if** (i == x[0]) count[0]++;

**else** **if** (i == x[1]) count[1]++;

}

**for** (int j = 0; j < 2; j++) {

**if** (count[j] > nums.length/3 && !res.contains(x[j])) res.add(x[j]);

}

**return** res;

}

这个写法好

出现次数多于⌊ n/2 ⌋ 的元素，在数组中最多有1个，因为⌊ n/2 ⌋ X2>n,多于两个则不可能；

出现次数多于⌊ n/3 ⌋ 的元素，在数组中最多有2个，因为⌊ n/3 ⌋ X3>n

**public** **class** **Solution**{

**public** List<Integer> majorityElement(**int**[] nums){

List<Integer> rst = **new** ArrayList<Integer>();

**if**(nums == **null** || nums.length == 0) **return** rst;

**int** count1 = 0, count2 = 0, candidate1 = 0, candidate2 = 1;（候选人初值设为不同即可）

**for**(**int** num : nums){ //这里的步骤基本同⌊ n/2 ⌋的题目，区别在于每次需要考虑两个候选数字；注意这里都是else if（每次loop只会进入一个条件）；else if顺序不能变，主要在于初始判断candidate值要不一样，如果换了else if则不能保证两个数不同

**if**(num == candidate1) count1++;

**else** **if**(num == candidate2) count2++;

**else** **if**(count1 == 0){

candidate1 = num;

count1 = 1;

}

**else** **if**(count2 == 0){

candidate2 = num;

count2 = 1;

}

**else**{

count1--;

count2--;

}

}

count1 = 0; count2 = 0;

**for**(**int** num : nums){//直接计数比较好理解，不采用这种

**if**(num == candidate1) count1+=2;

**else** count1--;

**if**(num == candidate2) count2 += 2;

**else** count2--;

}

**if**(count1 > 0) rst.add(candidate1);

**if**(count2 > 0) rst.add(candidate2);

**return** rst;

}

}

public List<**Integer**> majorityElement(int[] nums) {

if (nums == null || nums.length == 0)

return new ArrayList<**Integer**>();

List<**Integer**> result = new ArrayList<**Integer**>();

int number1 = nums[0], number2 = nums[0], count1 = 0, count2 = 0, len = nums.length;

for (int i = 0; i < len; i++) {

if (nums[i] == number1)

count1++;

else if (nums[i] == number2)

count2++;

else if (count1 == 0) {

number1 = nums[i];

count1 = 1;

} else if (count2 == 0) {

number2 = nums[i];

count2 = 1;

} else {

count1--;

count2--;

}

}

count1 = 0; //直接计数

count2 = 0;

for (int i = 0; i < len; i++) {

if (nums[i] == number1)

count1++;

else if (nums[i] == number2)

count2++;

}

if (count1 > len / 3)

result.add(number1);

if (count2 > len / 3)

result.add(number2);

return result;

}

### Contains Duplicate

[My Submissions](https://leetcode.com/problems/contains-duplicate/submissions/)

QuestionEditorial Solution

Total Accepted: **84541** Total Submissions: **205613** Difficulty: **Easy**

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.

This problem seems trivial, so lets try different approaches to solve it:

Starting from worst time complexity to the best one:

Time complexity: O(N^2), memory: O(1)

The naive approach would be to run a iteration for each element and see whether a duplicate value can be found: this results in O(N^2) time complexity.

**public** **boolean** containsDuplicate(**int**[] nums) {

**for**(**int** i = 0; i < nums.length; i++) {

**for**(**int** j = i + 1; j < nums.length; j++) {

**if**(nums[i] == nums[j]) {

**return** **true**;

}

}

}

**return** **false**;

}

Time complexity: O(N lg N), memory: O(1) - not counting the memory used by sort

Since it is trivial task to find duplicates in sorted array, we can sort it as the first step of the algorithm and then search for consecutive duplicates.

**public** **boolean** containsDuplicate(**int**[] nums) {

Arrays.sort(nums);

**for**(**int** ind = 1; ind < nums.length; ind++) {

**if**(nums[ind] == nums[ind - 1]) {

**return** **true**;

}

}

**return** **false**;

}

Time complexity: O(N), memory: O(N)

Finally we can used a well known data structure hash table that will help us to identify whether an element has been previously encountered in the array.

**public** **boolean** containsDuplicate(**int**[] nums) {

**final** Set<Integer> distinct = **new** HashSet<Integer>();

**for**(**int** num : nums) {

**if**(distinct.contains(num)) {

**return** **true**;

}

distinct.add(num);

}

**return** **false**;

}

This is trivial but quite nice example of space-time tradeoff.

The Set's add method can be used for this situation,because it will return false if the element already exists .

**public** boolean containsDuplicate(**int**[] nums) {

Set<Integer> **set** = **new** HashSet<Integer>();

**for**(**int** i : nums)

**if**(!**set**.add(i))// if there is same

**return** **true**;

**return** **false**;

}

### Contains Duplicate II

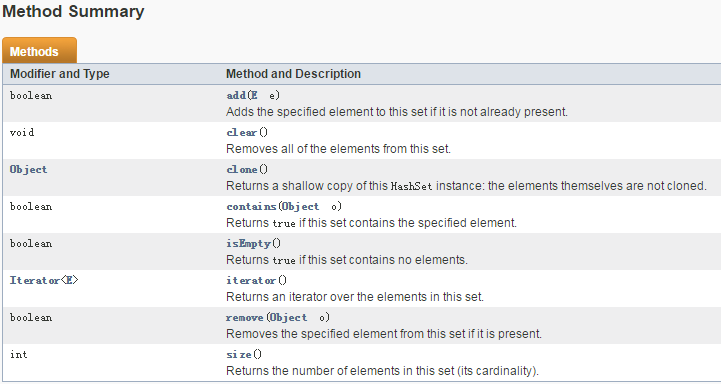
[My Submissions](https://leetcode.com/problems/contains-duplicate-ii/submissions/)

QuestionEditorial Solution

Total Accepted: **54712** Total Submissions: **183416** Difficulty: **Easy**

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 difference between *i* and *j*is at most *k*.

**HashSet**



Explanation: It iterates over the array using a sliding window. The front of the window is at i, the rear of the window is k steps back. The elements within that window are maintained using a set. While adding new element to the set, if add() returns false, it means the element already exists in the set. At that point, we return true. If the control reaches out of for loop, it means that inner return true never executed, meaning no such duplicate element was found.

we are storing at most k elements in hash set at one time. so not hashset, but the sliding window logic is more space efficient then plain hash map method. Running time complexity is similar anyways.

这个写法非常好，滑动窗口

注意HashSet移除数据时是按元素值移除，不可以按下标移除（因为其无序）

**public** boolean containsNearbyDuplicate(**int**[] nums, **int** k) {

Set<Integer> **set** = **new** HashSet<Integer>();

**for**(**int** i = 0; i < nums.length; i++){

**if**(i > k) **set**.remove(nums[i-k-1]);//要加一个元素之前得先清除最早的一个

**if**(!**set**.add(nums[i])) **return** **true**;

}

**return** **false**;

}

我的做法：数组值为key，下标为value（HashMap）

public boolean containsNearbyDuplicate(int[] nums, int k) {

HashMap<Integer,Integer> map=new HashMap<Integer,Integer>();

for(int i=0;i<nums.length;i++)

{

if(map.containsKey(nums[i]))

{

if(i-map.get(nums[i])<=k) return true;

}

map.put(nums[i],i);

}

return false;

}

### Contains Duplicate III 二叉查找树

[My Submissions](https://leetcode.com/problems/contains-duplicate-iii/submissions/)

QuestionEditorial Solution

Total Accepted: **26480** Total Submissions: **143458** Difficulty: **Medium**

Given an array of integers, find out whether there are two distinct indices *i* and *j* in the array such that the difference between **nums[i]** and **nums[j]** is at most *t* and the difference between *i* and *j* is at most *k*.

AC O(N) solution in Java using buckets with explanation

As a followup question, it naturally also requires maintaining a window of size k. When t == 0, it reduces to the previous question so we just reuse the solution.

Since there is now a constraint on the range of the values of the elements to be considered duplicates, it reminds us of doing a range check which is implemented in tree data structure and would take O(LogN) if a balanced tree structure is used, or doing a bucket check which is constant time. We shall just discuss the idea using bucket here.

Bucketing means we map a range of values to the a bucket. For example, if the bucket size is 3, we consider 0, 1, 2 all map to the same bucket. However, if t == 3, (0, 3) is a considered duplicates but does not map to the same bucket. This is fine since we are checking the buckets immediately before and after as well. So, as a rule of thumb, just make sure the size of the bucket is reasonable such that elements having the same bucket is immediately considered duplicates or duplicates must lie within adjacent buckets. So this actually gives us a range of possible bucket size, i.e. t and t + 1. We just choose it to be t and a bucket mapping to be num / t.

Another complication is that negative ints are allowed. A simple num / t just shrinks everything towards 0. Therefore, we can just reposition every element to start from Integer.MIN\_VALUE.

**public** **class** Solution {

**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**;

}

}

Edits:

Actually, we can use t + 1 as the bucket size to get rid of the case when t == 0. It simplifies the code. The above code is therefore the updated version.

Java O(N lg K) solution

This problem requires to maintain a window of size k of the previous values that can be queried for value ranges. The best data structure to do that is Binary Search Tree. As a result maintaining the tree of size k will result in time complexity O(N lg K). In order to check if there exists any value of range abs(nums[i] - nums[j]) to simple queries can be executed both of time complexity O(lg K)

Here is the whole solution using TreeMap.

public **class** **Solution** {

public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {

**if** (nums == null || nums.length == 0 || k <= 0) {

**return** **false**;

}

final TreeSet<Integer> values = new TreeSet<>();

**for** (int ind = 0; ind < nums.length; ind++) {

final Integer floor = values.floor(nums[ind] + t);

final Integer ceil = values.ceiling(nums[ind] - t);

**if** ((floor != null && floor >= nums[ind])

|| (ceil != null && ceil <= nums[ind])) {

**return** **true**;

}

values.add(nums[ind]);

**if** (ind >= k) {

values.remove(nums[ind - k]);

}

}

**return** **false**;

}

}

Java/Python one pass solution, O(n) time O(n) space using buckets

**HashMap**

|  |  |
| --- | --- |
| [**V**](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/util/HashMap.html) | [**remove**](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/util/HashMap.html#remove(java.lang.Object))([**Object**](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/Object.html) key)  Removes the mapping for the specified key from this map if present. |

The idea is like the bucket sort algorithm. Suppose we have consecutive buckets covering the range of nums with each bucket a width of (t+1). If there are two item with difference <= t, one of the two will happen:

(1) the two **in** the same bucket

(2) the two **in** neighbor buckets

**Java**

**这个写法好**

**例子：{-1,-2,-3,-4}标记为-1组；{0,1,2,3}标记为0组；{4,5,6,7}标记为1组；**

**private** **long** getID(**long** i, **long** w) {

**return** i < 0 ? (i + 1) / w - 1 : i / w;

}

**public** boolean containsNearbyAlmostDuplicate(**int**[] nums, **int** k, **int** t) {

**if** (t < 0) **return** **false**;

Map<Long, Long> d = **new** HashMap<>();

**long** w = (**long**)t + 1;

**for** (**int** i = 0; i < nums.length; ++i) {

**long** m = getID(nums[i], w);

**if** (d.containsKey(m))

**return** **true**;

**if** (d.containsKey(m - 1) && Math.abs(nums[i] - d.**get**(m - 1)) < w)

**return** **true**;

**if** (d.containsKey(m + 1) && Math.abs(nums[i] - d.**get**(m + 1)) < w)

**return** **true**;

d.put(m, (**long**)nums[i]);//每个区间只会放一个数，如果该区间有数字则直接返回true，否则去邻近的区间寻找

**if** (i >= k) d.remove(getID(nums[i - k], w));//放在后面写，为下次加入新元素做准备

}

**return** **false**;

}

### Pascal's Triangle

[My Submissions](https://leetcode.com/problems/pascals-triangle/submissions/)

QuestionEditorial Solution

Total Accepted: **81300** Total Submissions: **244243** Difficulty: **Easy**

Given *numRows*, generate the first *numRows* of Pascal's triangle.

For example, given *numRows* = 5,  
Return

[

[1],

[1,1],

[1,2,1],

[1,3,3,1],

[1,4,6,4,1]

]

public class Solution {

public List<**List<Integer**>> generate(int numRows)

{

List<**List<Integer**>> allrows = new ArrayList<**List<Integer**>>();

ArrayList<**Integer**> row = new ArrayList<**Integer**>();

for(int i=0;i<**numRows;i++)**

{

row.add(0, 1);// using row.add(0,1) and j<row.size()-1, you avoid boundary checking.

for(int j=1;j<row.size()-1;j++)//非常有技巧性的处理

row.set(j, row.get(j)+row.get(j+1));

allrows.add(new ArrayList<Integer>(row));// It's a technique to CLONE the row (make a copy of the row). Reason to do that is that the row is going to be modified when i is incremented. You don't want the row that is added to allrows to be modified too, you need to clone it.

}

return allrows;

}

}

two loops, one go through the row, one go through the column

database: pretty straight forward, ArrayList

calculate element value: K(i)(j)=K(i-1)(j-1)+K(i-1)(j) except for the first and last element

public class Solution {

public List<**List<Integer**>> generate(int numRows) {

List<**List<Integer**>> triangle = new ArrayList<**List<Integer**>>();

if (numRows <**=0){**

return triangle;

}

for (int i=0; i<numRows; i++){

List<Integer> row = new ArrayList<**Integer**>();

for (int j=0; j<**i+1;** j++){

if (j==0 || j==i){

row.add(1);

} else {

row.add(triangle.get(i-1).get(j-1)+triangle.get(i-1).get(j));

}

}

triangle.add(row);

}

return triangle;

}

}

public class Solution {

public List<**List<Integer**>> generate(int numRows) {

List<**List<Integer**>> res = new ArrayList<**List<Integer**>>();

List<**Integer**> row, pre = null;

for (int i = 0; i < numRows; ++i) {

row = new ArrayList<Integer>();

for (int j = 0; j <**=** i; ++j)

if (j == 0 || j == i)

row.add(1);

else

row.add(pre.get(j - 1) + pre.get(j));

pre = row;

res.add(row);

}

return res;

}

}

### Pascal's Triangle II

[My Submissions](https://leetcode.com/problems/pascals-triangle-ii/submissions/)

QuestionEditorial Solution

Total Accepted: **73097** Total Submissions: **226989** Difficulty: **Easy**

Given an index *k*, return the *k*th row of the Pascal's triangle.

For example, given *k* = 3,  
Return [1,3,3,1].

**Note:**  
Could you optimize your algorithm to use only *O*(*k*) extra space?

这个做法好

The basic idea is to iteratively update the array from the end to the beginning.

**class** Solution {

**public**:

**vector**<**int**> getRow(**int** rowIndex) {

**vector**<**int**> A(rowIndex+1, 0);

A[0] = 1;

**for**(**int** i=1; i<rowIndex+1; i++)

**for**(**int** j=i; j>=1; j--)

A[j] += A[j-1];

**return** A;

}

};

public List<**Integer**> getRow(int rowIndex) {

List<**Integer**> res = new ArrayList<**Integer**>();

for(int i = 0;i<**rowIndex+1;i++)** {

res.add(1);

for(int j=i-1;j>0;j--) {

res.set(j, res.get(j-1)+res.get(j));

}

}

return res;

}

**public** List<Integer> getRow(**int** rowIndex) {

List<Integer> **list** = **new** ArrayList<Integer>();

**if** (rowIndex < 0)

**return** **list**;

**for** (**int** i = 0; i < rowIndex + 1; i++) {

**list**.add(0, 1);

**for** (**int** j = 1; j < **list**.size() - 1; j++) {

**list**.**set**(j, **list**.get(j) + **list**.get(j + 1));

}

}

**return** **list**;

}

adding elements in ArrayList will involve shifting all following elements, which is O(n). To prevent that, I guess we can update elements from right to left:

for (int i = list.size()-1; i-1 >= 0; i--) list.set(i, list.get(i) + list.get(i-1)); list.add(1);

这个写法好

**public** **class** **Solution** {

**public** List<Integer> getRow(**int** k) {

Integer[] arr = **new** Integer[k + 1];

Arrays.fill(arr, 0);

arr[0] = 1;

**for** (**int** i = 1; i <= k; i++)

**for** (**int** j = i; j > 0; j--)

arr[j] = arr[j] + arr[j - 1];

**return** Arrays.asList(arr);

}

}

### Sort Colors

[My Submissions](https://leetcode.com/problems/sort-colors/submissions/)

QuestionEditorial Solution

Total Accepted: **96365** Total Submissions: **277907** Difficulty: **Medium**

Given an array with *n* objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

**Note:**  
You are not suppose to use the library's sort function for this problem.

[click to show follow up.](https://leetcode.com/problems/sort-colors/)

**Follow up:**  
A rather straight forward solution is a two-pass algorithm using counting sort.  
First, iterate 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.

Could you come up with an one-pass algorithm using only constant space?

我的做法

public void sortColors(int[] nums) {

int a=0,b=0,c=0;

for(int num:nums)

{

if(num==0) a++;

if(num==1) b++;

if(num==2) c++;

}

int cnt=0;

while(a>0)

{

nums[cnt++]=0;

a--;

}

while(b>0)

{

nums[cnt++]=1;

b--;

}

while(c>0)

{

nums[cnt++]=2;

c--;

}

}

这个做法好，需要考虑当前交换数是相等的情况，如果都为2，则右边控制2的范围的指针要一直更改，直到不为2

zero指针前全部都是0，sec指针后全为2

The idea is to sweep all 0s to the left and all 2s to the right, then all 1s are left in the middle.

**class** **Solution** {

**public**:

**void** sortColors(**int** A[], **int** n) {

**int** second=n-1, zero=0;

**for** (**int** i=0; i<=second; i++) {

**while** (A[i]==2 && i<second)

swap(A[i], A[second--]);

**while** (A[i]==0 && i>zero) swap(A[i], A[zero++]);

}

}

};

// two pass O(m+n) space

**void** sortColors(**int** A[], **int** n) {

**int** num0 = 0, num1 = 0, num2 = 0;

**for**(**int** i = 0; i < n; i++) {

**if** (A[i] == 0) ++num0;

**else** **if** (A[i] == 1) ++num1;

**else** **if** (A[i] == 2) ++num2;

}

**for**(**int** i = 0; i < num0; ++i) A[i] = 0;

**for**(**int** i = 0; i < num1; ++i) A[num0+i] = 1;

**for**(**int** i = 0; i < num2; ++i) A[num0+num1+i] = 2;

}

这个方法很不错，用3个计数器分别记录3个color的个数。同时用覆盖的方法对唯一的数组进行赋值。

值为2的是排在最后的，所以当数组值为0,1时它仍然需要计数，来确定最后的2的位置。

// one pass in place solution

**void** sortColors(**int** A[], **int** n) {

**int** n0 = -1, n1 = -1, n2 = -1;

**for** (**int** i = 0; i < n; ++i) {

**if** (A[i] == 0)

{

A[++n2] = 2; A[++n1] = 1; A[++n0] = 0;

}

**else** **if** (A[i] == 1)

{

A[++n2] = 2; A[++n1] = 1;

}

**else** **if** (A[i] == 2)

{

A[++n2] = 2;

}

}

}

// one pass in place solution

**void** sortColors(**int** A[], **int** n) {

**int** j = 0, k = n - 1;

**for** (**int** i = 0; i <= k; ++i){

**if** (A[i] == 0 && i != j)

swap(A[i--], A[j++]); //注意这里的指针变换

**else** **if** (A[i] == 2 && i != k)

swap(A[i--], A[k--]);

}

}

// one pass in place solution

**void** sortColors(**int** A[], **int** n) {

**int** j = 0, k = n-1;

**for** (**int** i=0; i <= k; i++) {

**if** (A[i] == 0)

swap(A[i], A[j++]);

**else** **if** (A[i] == 2)

swap(A[i--], A[k--]);//注意这里的指针变换，i必须回退（关键）否则，i处不能保证不为2

}

}

java solution, both 2-pass and 1-pass

public void sortColors(**int**[] nums) {

// 1-pass

**int** p1 = 0, p2 = nums.**length** - 1, **index** = 0;

**while** (**index** <= p2) {

**if** (nums[**index**] == 0) {

nums[**index**] = nums[p1];

nums[p1] = 0;

p1++;

}

**if** (nums[**index**] == 2) {

nums[**index**] = nums[p2];

nums[p2] = 2;

p2--;

**index**--;

}

**index**++;

}

}

**public** **void** sortColors(**int**[] nums) {

// 2-pass

**int** count0 = 0, count1 = 0, count2 = 0;

**for** (**int** i = 0; i < nums.length; i++) {

**if** (nums[i] == 0) {count0++;}

**if** (nums[i] == 1) {count1++;}

**if** (nums[i] == 2) {count2++;}

}

**for**(**int** i = 0; i < nums.length; i++) {

**if** (i < count0) {nums[i] = 0;}

**else** **if** (i < count0 + count1) {nums[i] = 1;}

**else** {nums[i] = 2;}

}

}

The concept is simple. Maintain two pointer, pointer "one" indicates the begging of all ones and pointer "two" indicates the begging of all twos. When we meet 1, we move 1 to the end of 1 sequence which is begging of two sequence then move begging of 2 forward 1. Doing the same to the 2.

**public** **class** **Solution** {

**public** **void** sortColors(**int**[] A) {

**int** one = 0;

**int** two = 0;

**for**(**int** i=0;i<A.length;i++){

**if**(A[i]==0){

A[i] = A[two];

A[two] = A[one];

A[one] = 0;

one++;

two++;

}

**else** **if**(A[i]==1){

A[i] = A[two];

A[two] = 1;

two++;

}

}

}}

### Wiggle Sort II 摆动排序

[My Submissions](https://leetcode.com/problems/wiggle-sort-ii/submissions/)

QuestionEditorial Solution

Total Accepted: **8802** Total Submissions: **39658** Difficulty: **Medium**

Given an unsorted array nums, reorder it such that nums[0] < nums[1] > nums[2] < nums[3]....

**Example:**  
(1) Given nums = [1, 5, 1, 1, 6, 4], one possible answer is [1, 4, 1, 5, 1, 6].   
(2) Given nums = [1, 3, 2, 2, 3, 1], one possible answer is [2, 3, 1, 3, 1, 2].

**Note:**  
You may assume all input has valid answer.

**Follow Up:**  
Can you do it in O(n) time and/or in-place with O(1) extra space?

这个方法好

数组长度是偶数，则+1；

Index的变换是（1+2i）%(n|1)

关键：1 需要先利用KthMax找出数组的中位数，O(n) time with O(1) extra space

2 将数组的对应index进行改变

3 将大于中位数的数都放在前面，小于中位数的数放在后面，等于中位数的数放中间（三分法）这样利用变换的index就可以达到题目的要求。

O(n)+O(1) after median --- Virtual Indexing

This post is mainly about what I call "virtual indexing" technique (I'm sure I'm not the first who came up with this, but I couldn't find anything about it, so I made up a name as well. If you know better, let me know).

Solution

**void** wiggleSort(**vector**<**int**>& nums) {

**int** n = nums.size();

// Find a median.

**auto** midptr = nums.begin() + n / 2;

nth\_element(nums.begin(), midptr, nums.end());

**int** mid = \*midptr;

// Index-rewiring.

#define A(i) nums[(1+2\*(i)) % (n|1)]

// 3-way-partition-to-wiggly in O(n) time with O(1) space.

**int** i = 0, j = 0, k = n - 1;

**while** (j <= k) {

**if** (A(j) > mid)

swap(A(i++), A(j++));

**else** **if** (A(j) < mid)

swap(A(j), A(k--));//关注指针移动情况

**else**

j++;

}

}

Explanation

First I find a median using nth\_element. That only guarantees O(n) **average** time complexity and I don't know about space complexity. I might write this myself using O(n) time and O(1) space, but that's not what I want to show here.

This post is about what comes **after** that. We can use [three-way partitioning](https://en.wikipedia.org/wiki/Dutch_national_flag_problem#Pseudocode) to arrange the numbers so that those larger than the median come first, then those equal to the median come next, and then those smaller than the median come last.

Ordinarily, you'd then use one more phase to bring the numbers to their final positions to reach the overall wiggle-property. But I don't know a nice O(1) space way for this. Instead, I embed this right into the partitioning algorithm. That algorithm simply works with indexes 0 to n-1 as usual, but sneaky as I am, I rewire those indexes where I want the numbers to actually end up. The partitioning-algorithm doesn't even know that I'm doing that, it just works like normal (it just usesA(x) instead of nums[x]).

Let's say nums is [10,11,...,19]. Then after nth\_element and ordinary partitioning, we might have this (15 is my median):

index: 0 1 2 3 4 5 6 7 8 9

number: 18 17 19 16 15 11 14 10 13 12

I rewire it so that the first spot has index 5, the second spot has index 0, etc, so that I might get this instead:

index: 5 0 6 1 7 2 8 3 9 4

number: 11 18 14 17 10 19 13 16 12 15

And 11 18 14 17 10 19 13 16 12 15 is perfectly wiggly. And the whole partitioning-to-wiggly-arrangement (everything after finding the median) only takes O(n) time and O(1) space.

If the above description is unclear, maybe this explicit listing helps:

Accessing A(0) actually accesses nums[1].  
Accessing A(1) actually accesses nums[3].  
Accessing A(2) actually accesses nums[5].  
Accessing A(3) actually accesses nums[7].  
Accessing A(4) actually accesses nums[9].  
Accessing A(5) actually accesses nums[0].  
Accessing A(6) actually accesses nums[2].  
Accessing A(7) actually accesses nums[4].  
Accessing A(8) actually accesses nums[6].  
Accessing A(9) actually accesses nums[8].

Props to [apolloydy's solution](https://leetcode.com/discuss/77053/time-and-cheating-space-solution-will-there-be-real-solution?show=77054#a77054), I knew the partitioning algorithm already but I didn't know the name. And apolloydy's idea to partition to reverse order happened to make the index rewiring simpler.

**3 lines Python, with Explanation / Proof**

Solution

Roughly speaking I put the smaller half of the numbers on the even indexes and the larger half on the odd indexes.

**def** **wiggleSort**(self, nums):

nums.sort()

half = len(nums[::2])

nums[::2], nums[1::2] = nums[:half][::-1], nums[half:][::-1]

Alternative, maybe nicer, maybe not:

**def** **wiggleSort**(self, nums):

nums.sort()

half = len(nums[::2]) - 1

nums[::2], nums[1::2] = nums[half::-1], nums[:half:-1]

**Explanation / Proof**

I put the smaller half of the numbers on the even indexes and the larger half on the odd indexes, both from right to left:

Example nums = [1,2,...,7] Example nums = [1,2,...,8]

Small half: 4 . 3 . 2 . 1 Small half: 4 . 3 . 2 . 1 .

Large half: . 7 . 6 . 5 . Large half: . 8 . 7 . 6 . 5

-------------------------- --------------------------

Together: 4 7 3 6 2 5 1 Together: 4 8 3 7 2 6 1 5

I want:

* Odd-index numbers are larger than their neighbors.

Since I put the larger numbers on the odd indexes, clearly I already have:

* Odd-index numbers are larger than **or equal to** their neighbors.

Could they be "equal to"? That would require some number M to appear both in the smaller and the larger half. It would be the largest in the smaller half and the smallest in the larger half. Examples again, where S means some number smaller than M and L means some number larger than M.

Small half: M . S . S . S Small half: M . S . S . S .

Large half: . L . L . M . Large half: . L . L . L . M

-------------------------- --------------------------

Together: M L S L S M S Together: M L S L S L S M

You can see the two M are quite far apart. Of course M could appear more than just twice, for example:

Small half: M . M . S . S Small half: M . S . S . S .

Large half: . L . L . M . Large half: . L . M . M . M

-------------------------- --------------------------

Together: M L M L S M S Together: M L S M S M S M

You can see that with seven numbers, three M are no problem. And with eight numbers, four M are no problem. Should be easy to see that in general, with n numbers, floor(n/2) times M is no problem. Now, if there were more M than that, then my method would fail. But... it would also be impossible:

* If n is even, then having more than n/2 times the same number clearly is unsolvable, because you'd have to put two of them next to each other, no matter how you arrange them.
* If n is odd, then the only way to successfully arrange a number appearing more than floor(n/2) times is if it appears exactly floor(n/2)+1 times and you put them on all the even indexes. And to have the wiggle-property, all the other numbers would have to be larger. But then we wouldn't have an M in both the smaller and the larger half.

So if the input has a valid answer at all, then my code will find one.

Step by step explanation of index mapping in Java

The virtual index idea in the post https://leetcode.com/discuss/77133/o-n-o-1-after-median-virtual-indexing is very brilliant! However, it takes me a while to understand why and how it works. There is no 'nth\_element' in Java, but you can use 'findKthLargest' function from "https://leetcode.com/problems/kth-largest-element-in-an-array/" to get the median element in average O(n) time and O(1) space.

Assume your original array is {6,13,5,4,5,2}. After you get median element, the 'nums' is partially sorted such that the first half is larger or equal to the median, the second half is smaller or equal to the median, i.e

13 6 5 5 4 2

M

In the post https://leetcode.com/discuss/76965/3-lines-python-with-explanation-proof, we have learned that , to get wiggle sort, you want to put the number in the following way such that

(1) elements smaller than the 'median' are put into the last even slots

(2) elements larger than the 'median' are put into the first odd slots

(3) the medians are put into the remaining slots.

Index : 0 1 2 3 4 5

Small half: M S S

Large half: L L M

M - Median, S-Small, L-Large. In this example, we want to put {13, 6, 5} in index 1,3,5 and {5,4,2} in index {0,2,4}

The index mapping, (1 + 2\*index) % (n | 1) combined with 'Color sort', will do the job.

After selecting the median element, which is 5 in this example, we continue as the following

Mapped\_idx[Left] denotes the position where the next smaller-than median element will be inserted.

Mapped\_idx[Right] denotes the position where the next larger-than median element will be inserted.

Step 1:

Original idx: 0 1 2 3 4 5

Mapped idx: 1 3 5 0 2 4

Array: 13 6 5 5 4 2

Left

i

Right

nums[Mapped\_idx[i]] = nums[1] = 6 > 5, so it is ok to put 6 **in** the first odd index 1. We increment i **and** left.

Step 2:

Original idx: 0 1 2 3 4 5

Mapped idx: 1 3 5 0 2 4

Array: 13 6 5 5 4 2

Left

i

Right

nums[3] = 5 = 5, so it is ok to put 6 **in** the index 3. We increment i.

Step 3:

Original idx: 0 1 2 3 4 5

Mapped idx: 1 3 5 0 2 4

Array: 13 6 5 5 4 2

Left

i

Right

nums[5] = 2 < 5, so we want to put it to the last even index 4 (pointed by Right). So, we swap nums[Mapped\_idx[i]] with nums[Mapped\_idx[Right]], i.e. nums[5] with nums[4], **and** decrement Right.

Step 4:

Original idx: 0 1 2 3 4 5

Mapped idx: 1 3 5 0 2 4

Array: 13 6 5 5 2 4

Left

i

Right

nums[5] = 4 < 5, so we want to put it to the second last even index 2. So, we swap nums[5] with nums[2], **and** decrement Right.

Step 5:

Original idx: 0 1 2 3 4 5

Mapped idx: 1 3 5 0 2 4

Array: 13 6 4 5 2 5

Left

i

Right

nums[5] = 5 < 5, it is ok to put it there, we increment i.

Step 6:

Original idx: 0 1 2 3 4 5

Mapped idx: 1 3 5 0 2 4

Array: 13 6 4 5 2 5

Left

i

Right

nums[0] = 13 > 5, so, we want to put it to the next odd index which is 3 (pointed by 'Left'). So, we swap nums[0] with nums[3], **and** increment 'Left' **and** 'i'.

Step Final:

Original idx: 0 1 2 3 4 5

Mapped idx: 1 3 5 0 2 4

Array: 5 6 4 13 2 5

Left

i

Right

i > Right, we get the final wiggle array 5 6 4 13 2 5 !

The code is the following:

public void wiggleSort(**int**[] nums) {

**int** median = findKthLargest(nums, (nums.**length** + 1) / 2);

**int** n = nums.**length**;

**int** left = 0, i = 0, right = n - 1;

**while** (i <= right) {

**if** (nums[newIndex(i,n)] > median) {

swap(nums, newIndex(left++,n), newIndex(i++,n));

}

**else** **if** (nums[newIndex(i,n)] < median) {

swap(nums, newIndex(right--,n), newIndex(i,n));

}

**else** {

i++;

}

}

}

private **int** newIndex(**int** **index**, **int** n) {

**return** (1 + 2\*index) % (n | 1);

}

Inspired by this [question](https://leetcode.com/discuss/77122/simple-modulo-solution)

非进阶的解法

奇数位从sort好的数组的末位开始排列；

偶数位从sort好的数组的中间往前开始排列

public void wiggleSort(**int**[] nums) {

Arrays.**sort**(nums);

**int**[] temp = new **int**[nums.**length**];

**int** mid = nums.**length**%2==0?nums.**length**/2-1:nums.**length**/2;

**int** **index** = 0;

**for**(**int** i=0;i<=mid;i++){

temp[**index**] = nums[mid-i];

**if**(**index**+1<nums.**length**)

temp[**index**+1] = nums[nums.**length**-i-1];

**index** = **index**+2;

}

**for**(**int** i=0;i<nums.**length**;i++){

nums[i] = temp[i];

}

}

### Product of Array Except Self

[My Submissions](https://leetcode.com/problems/product-of-array-except-self/submissions/)

QuestionEditorial Solution

Total Accepted: **42934** Total Submissions: **100591** Difficulty: **Medium**

Given an array of *n* integers where *n* > 1, nums, return an array output such that output[i] is equal to the product of all the elements of nums except nums[i].

Solve it **without division** and in O(*n*).

For example, given [1,2,3,4], return [24,12,8,6].

**Follow up:**  
Could you solve it with constant space complexity? (Note: The output array **does not** count as extra space for the purpose of space complexity analysis.)

这题就是需要考虑有0存在的情况，有以下几种情况

当数组中只有1个0时，0的位置是其他非0的位置的元素的乘积

当数组中多于1个0时，结果必是全0

当数组中没有0时

所以只需记录0的个数及index，非0元素的乘积

我的做法

public int[] productExceptSelf(int[] nums) {

int[] res=new int[nums.length];

int mul=1;

int zero=0;

int ind=-1;

for(int i=0;i<nums.length;i++)

{

if(nums[i]==0)

{

zero++;

ind=i;

}

else mul\*=nums[i];

}

if(zero>1)

{

return res;

}

else if(zero==1)

{

res[ind]=mul;

}

else

{

for(int i=0;i<nums.length;i++)

{

res[i]=mul/nums[i];

}

}

return res;

}

这个做法不错，从数组的开始和结束处扫描两次

左边需要考虑第一个数组元素的特例，右边扫描时需要考虑最后一个元素的情况

The idea is simply. The product basically is calculated using the numbers before the current number and the numbers after the current number. Thus, we can scan the array twice. First, we calcuate the running product of the part before the current number. Second, we calculate the running product of the part after the current number through scanning from the end of the array.

**public** **class** **Solution** {

**public** **int**[] productExceptSelf(**int**[] nums) {

**int** n = nums.length;

**int**[] res = **new** **int**[n];

res[0] = 1;

**for** (**int** i = 1; i < n; i++) {

res[i] = res[i - 1] \* nums[i - 1]; //res[i]存放其左边元素的乘积（不包括自己）

}

**int** right = 1;

**for** (**int** i = n - 1; i >= 0; i--) {

res[i] \*= right; //res[i]存放其右边元素的乘积

right \*= nums[i];

}

**return** res;

}

}

Use tmp to store temporary multiply result by two directions. Then fill it into result. Bingo!

public **int**[] productExceptSelf(**int**[] nums) {

**int**[] result = new **int**[nums.**length**];

**for** (**int** i = 0, tmp = 1; i < nums.**length**; i++) {

result[i] = tmp;

tmp \*= nums[i];

}

**for** (**int** i = nums.**length** - 1, tmp = 1; i >= 0; i--) {

result[i] \*= tmp;

tmp \*= nums[i];

}

**return** result;

}

### Spiral Matrix

[My Submissions](https://leetcode.com/problems/spiral-matrix/submissions/)

QuestionEditorial Solution 螺旋

Total Accepted: **57893** Total Submissions: **257485** Difficulty: **Medium**

Given a matrix of *m* x *n* elements (*m* rows, *n* columns), return all elements of the matrix in spiral order.

For example,  
Given the following matrix:

[

[ 1, 2, 3 ],

[ 4, 5, 6 ],

[ 7, 8, 9 ]

]

You should return [1,2,3,6,9,8,7,4,5].

### Combination Sum

[My Submissions](https://leetcode.com/problems/combination-sum/submissions/)

QuestionEditorial Solution

Total Accepted: **89195** Total Submissions: **288928** Difficulty: **Medium**

Given a set of candidate numbers (***C***) 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.
* Elements in a combination (*a*1, *a*2, … , *a*k) must be in non-descending order. (ie, *a*1 ≤ *a*2 ≤ … ≤ *a*k).
* 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]

这题，因为每个数可以取多遍，所以给定数组不会出现重复元素，给定结果也不会出现重复；如果给定数组有多个重复元素，则按照以下解法结果的list中会有重复

这个写法较好

先对数组进行排序，然后用for循环依次将每个数加入

public class Solution {

List<List<Integer>> res=new ArrayList<List<Integer>>();

public List<List<Integer>> combinationSum(int[] candidates, int target) {

List<Integer> cur=new ArrayList<Integer>();

Arrays.sort(candidates);

combine(0,candidates,target,cur);

return res;

}

public void combine(int index,int[] candidates, int target,List<Integer> cur)

{

if(target==0)

{

ArrayList<Integer> tmp=new ArrayList<Integer>(cur);

res.add(tmp);

return;

}

for(int i=index;i<candidates.length;i++)

{

if(target<candidates[i]) return;//mark，不剪枝的话会出现

StackOverflowError

cur.add(candidates[i]);

combine(i,candidates,target-candidates[i],cur);

cur.remove(cur.size()-1);

}

}

}

**public** **class** **Solution** {

**public** **List**<**List**<Integer>> combinationSum(int[] candidates, int target) {

Arrays.sort(candidates);

**List**<**List**<Integer>> result = **new** ArrayList<**List**<Integer>>();

getResult(result, **new** ArrayList<Integer>(), candidates, target, 0);

**return** result;

}

**private** void getResult(**List**<**List**<Integer>> result, **List**<Integer> cur, int candidates[], int target, int start){

**if**(target > 0){

**for**(int i = start; i < candidates.length && target >= candidates[i]; i++){

cur.add(candidates[i]);

getResult(result, cur, candidates, target - candidates[i], i);

cur.remove(cur.size() - 1);

}//for

}//if

**else** **if**(target == 0 ){

result.add(**new** ArrayList<Integer>(cur));

}//else if

}

}

Sort the candidates and we choose from small to large recursively, every time we add a candidate to our possible sub result, we subtract the target to a new smaller one.

public List<**List<Integer**>> combinationSum(int[] candidates, int target) {

List<**List<Integer**>> ret = new LinkedList<**List<Integer**>>();

Arrays.sort(candidates); // sort the candidates

// collect possible candidates from small to large to eliminate duplicates,

recurse(new ArrayList<**Integer**>(), target, candidates, 0, ret);

return ret;

}

// the index here means we are allowed to choose candidates from that index

private void recurse(List<**Integer**> list, int target, int[] candidates, int index, List<**List<Integer**>> ret) {

if (target == 0) {

ret.add(list);

return;

}

for (int i = index; i < candidates.length; i++) {

int newTarget = target - candidates[i];

if (newTarget >= 0) {

List<**Integer**> copy = new ArrayList<**Integer**>(list);

copy.add(candidates[i]);

recurse(copy, newTarget, candidates, i, ret);

} else {

break;

}

}

}

### Combination Sum II

[My Submissions](https://leetcode.com/problems/combination-sum-ii/submissions/)

QuestionEditorial Solution

Total Accepted: **67147** Total Submissions: **243906** Difficulty: **Medium**

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.
* Elements in a combination (*a*1, *a*2, … , *a*k) must be in non-descending order. (ie, *a*1 ≤ *a*2 ≤ … ≤ *a*k).
* The solution set must not contain duplicate combinations.

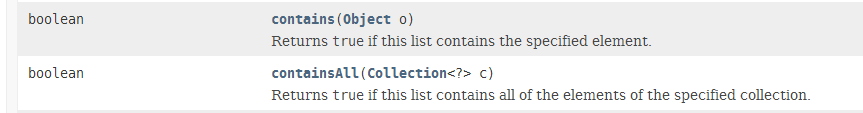
For example, given candidate set 10,1,2,7,6,1,5 and target 8,   
A solution set is:   
[1, 7]   
[1, 2, 5]   
[2, 6]   
[1, 1, 6]

与上一题区别，数组元素可能出现重复（结果list中会存在重复，所以排序后考虑去重情况），但是数组中出现的元素只能使用一次；

考虑什么时候需要去重：从数为index处时一定要考虑（重复也需要考虑）；数大于index，则后面出现重复数时只需要考虑一遍

例如{1,1,1,3}

List的方法



与上一题的区别是标黄处

注意这里需要处理重复的情况

public class Solution {

List<List<Integer>> res=new ArrayList<List<Integer>>();

public List<List<Integer>> combinationSum2(int[] candidates, int target) {

Arrays.sort(candidates);

List<Integer> cur=new ArrayList<Integer>();

combine(0,target,candidates,cur);

return res;

}

public void combine(int index, int target, int[] candidates,List<Integer> cur)

{

if(target==0)

{

List<Integer> tmp=new ArrayList<Integer>(cur);

//if(!res.contains(tmp)) res.add(tmp);

res.add(tmp);

return;

}

for(int i=index;i<candidates.length;i++)

{

if(target<candidates[i]) return;

if(i>index && candidates[i]==candidates[i-1]) continue; //去重

cur.add(candidates[i]);

combine(i+1,target-candidates[i],candidates,cur);

cur.remove(cur.size()-1);

}

}

}

public List<**List<Integer**>> combinationSum2(int[] cand, int target) {

Arrays.sort(cand);

List<**List<Integer**>> res = new ArrayList<**List<Integer**>>();

List<**Integer**> path = new ArrayList<**Integer**>();

dfs\_com(cand, 0, target, path, res);

return res;

}

void dfs\_com(int[] cand, int cur, int target, List<**Integer**> path, List<**List<Integer**>> res) {

if (target == 0) {

res.add(new ArrayList(path));

return ;

}

if (target < 0) return;

for (int i = cur; i < cand.length; i++){

if (i > cur && cand[i] == cand[i-1]) continue; //这样处理不错

path.add(path.size(), cand[i]);

dfs\_com(cand, i+1, target - cand[i], path, res);

path.remove(path.size()-1);

}

}

At the beginning, I stuck on this problem. After careful thought, I think this kind of backtracking contains a iterative component and a resursive component so I'd like to give more details to help beginners save time. The revursive component tries the elements after the current one and also tries duplicate elements. So we can get correct answer for cases like [1 1] 2. The iterative component checks duplicate combinations and skip it if it is. So we can get correct answer for cases like [1 1 1] 2.

**class** Solution {

**public**:

**vector**<**vector**<**int**> > combinationSum2(**vector**<**int**> &num, **int** target)

{

**vector**<**vector**<**int**>> res;

sort(num.begin(),num.end());

**vector**<**int**> local;

findCombination(res, 0, target, local, num);

**return** res;

}

**void** findCombination(**vector**<**vector**<**int**>>& res, **const** **int** order, **const** **int** target, **vector**<**int**>& local, **const** **vector**<**int**>& num)

{

**if**(target==0)

{

res.push\_back(local);

**return**;

}

**else**

{

**for**(**int** i = order;i<num.size();i++) // iterative component

{

**if**(num[i]>target) **return**;

**if**(i&&num[i]==num[i-1]&&i>order) **continue**; // check duplicate combination

local.push\_back(num[i]),

findCombination(res,i+1,target-num[i],local,num); // recursive componenet

local.pop\_back();

}

}

}

};

### Combination Sum III

[My Submissions](https://leetcode.com/problems/combination-sum-iii/submissions/)

QuestionEditorial Solution

Total Accepted: **30697** Total Submissions: **85911** Difficulty: **Medium**

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.

Ensure that numbers within the set are sorted in ascending order.

***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]]

我的做法用了cnt计数，其实用最下面的方法，直接去考虑cur的大小是否为k即可。

public class Solution {

List<List<Integer>> res=new ArrayList<List<Integer>>();

public List<List<Integer>> combinationSum3(int k, int n) {

List<Integer> cur=new ArrayList<Integer>();

combine(1,n,k,cur);

return res;

}

public void combine(int index, int target, int cnt, List<Integer> cur)

{

if(cnt<0) return;

if(target==0 && cnt==0)

{

res.add(new ArrayList<Integer>(cur));

return;

}

for(int i=index;i<10;i++)

{

if(target<i) return;

cur.add(i);

cnt--;

combine(i+1,target-i,cnt,cur);

cnt++;

cur.remove(cur.size()-1);

}

}

}

public List<**List<Integer**>> combinationSum3(int k, int n) {

List<**List<Integer**>> ans = new ArrayList<>();

combination(ans, new ArrayList<**Integer**>(), k, 1, n);

return ans;

}

private void combination(List<**List<Integer**>> ans, List<**Integer**> comb, int k, int start, int n) {

if (comb.size() == k && n == 0) {

List<**Integer**> li = new ArrayList<**Integer**>(comb);

ans.add(li);

return;

}

for (int i = start; i <**=** 9; i++) {

comb.add(i);

combination(ans, comb, k, i+1, n-i);

comb.remove(comb.size() - 1);

}

}

下面这样更好

if you think for loop with multiple conditions is complex, you can add an if condition inside the for loop instead.

private static void combination(List<**List<Integer**>> ans, List<**Integer**> comb, int k, int start, int n) {

if (comb.size() > k) {

return;

}

if (comb.size() == k && n == 0) {

List<**Integer**> li = new ArrayList<**Integer**>(comb);

ans.add(li);

return;

}

for (int i = start; i<**=9;** i++) {

if (n-i >= 0) {

comb.add(i);

combination(ans, comb, k, i+1, n-i);

comb.remove(comb.size() - 1);

}

}

}

### Combinations

[My Submissions](https://leetcode.com/problems/combinations/submissions/)

QuestionEditorial Solution

Total Accepted: **73648** Total Submissions: **214996** Difficulty: **Medium**

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],

]

我的做法

public class Solution {

List<List<Integer>> res=new ArrayList<List<Integer>>();

public List<List<Integer>> combine(int n, int k) {

List<Integer> cur=new ArrayList<Integer>();

combine(1,cur,k,n);

return res;

}

public void combine(int index, List<Integer> cur,int k,int n)

{

if(cur.size()>k) return;

if(cur.size()==k)

{

res.add(new ArrayList<Integer>(cur));

return;

}

for(int i=index;i<=n;i++)

{

cur.add(i);

combine(i+1,cur,k,n);

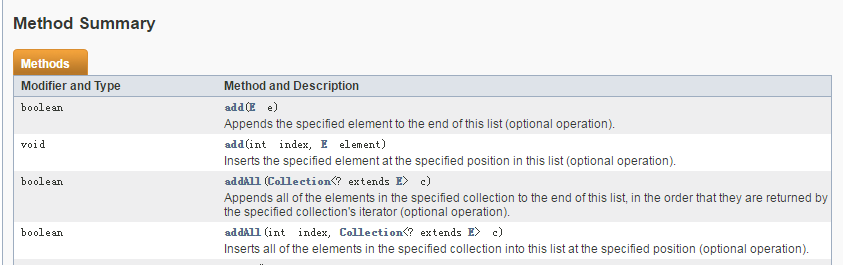
cur.remove(cur.size()-1);

}

}

}

List接口



A short recursive Java solution based on C(n,k)=C(n-1,k-1)+C(n-1,k)

Basically, this solution follows the idea of the mathematical formula C(n,k)=C(n-1,k-1)+C(n-1,k).

Here C(n,k) is divided into two situations. Situation one, number n is selected, so we only need to select k-1 from n-1 next. Situation two, number n is not selected, and the rest job is selecting k from n-1.

**public** **class** **Solution** {

**public** **List**<**List**<Integer>> combine(int n, int k) {

**if** (k == n || k == 0) {

**List**<Integer> row = **new** LinkedList<>();

**for** (int i = 1; i <= k; ++i) {

row.add(i);

}

**return** **new** LinkedList<>(Arrays.asList(row));

}

**List**<**List**<Integer>> result = **this**.combine(n - 1, k - 1);

result.**forEach**(e -> e.add(n));//情形1，n被选中，剩下的k-1个元素从n-1个元素中选取

result.addAll(**this**.combine(n - 1, k));

**return** result;

}

}

这个写法也很不错啊，比使用cur.size()清晰（和我用cnt相比，因为在新的函数中传入k-1，所以这轮的k值不会受影响。一定要注意，新函数中传入的是k-1，不是k--.）

public static List<**List<Integer**>> combine(int n, int k) {

List<**List<Integer**>> combs = new ArrayList<**List<Integer**>>();

combine(combs, new ArrayList<**Integer**>(), 1, n, k);

return combs;

}

public static void combine(List<**List<Integer**>> combs, List<**Integer**> comb, int start, int n, int k) {

if(k==0) {

combs.add(new ArrayList<**Integer**>(comb));

return;

}

for(int i=start;i<**=n;i++)** {

comb.add(i);

combine(combs, comb, i+1, n, k-1);//注意不可以是k--

comb.remove(comb.size()-1);

}

}

### Letter Combinations of a Phone Number

[My Submissions](https://leetcode.com/problems/letter-combinations-of-a-phone-number/submissions/)

QuestionEditorial Solution

Total Accepted: **78079** Total Submissions: **272244** Difficulty: **Medium**

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.



**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.

我的做法

public class Solution {

public List<String> letterCombinations(String digits) {

String[] str={"abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};

List<String> res=new ArrayList<String>();

if(digits.equals("")|| digits==null) return res;

combine(0,res,"",str,digits);

return res;

}

public void combine(int index,List<String> res,String cur,String[] str,String digits)

{

if(index>digits.length()) return;

if(index==digits.length())

{

res.add(cur);

return;

}

int num=Integer.parseInt(digits.charAt(index)+"");

for(int i=0;i<str[num-2].length();i++)

{

char ch=str[num-2].charAt(i);

combine(index+1,res,cur+ch,str,digits);

}

}

}

这个方法不错，运用宽搜（队列处理方式）

My java solution with FIFO queue

public List<**String**> letterCombinations(String digits) {

LinkedList<**String**> ans = new LinkedList<**String**>();

String[] mapping = new String[] {"0", "1", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};

ans.add("");

for(int i =0; i<**digits.length();i++){**

int x = Character.getNumericValue(digits.charAt(i));

while(ans.peek().length()==i){//当队列中存在元素的长度为i时，说明是上一轮还未处理的元素，则仍然需要处理（这里很巧妙）

String t = ans.remove(); //获取并移除list的head

for(char s : mapping[x].toCharArray())

ans.add(t+s); //构造法

}

}

return ans;

}

If you think about the recursive solution similar to a DFS approach, then this would be the equivalent BFS solution (using a queue).

Very good solution!! Thumb up! This is a iterative solution. For each digit added, remove and copy every element in the queue and add the possible letter to each element, then add the updated elements back into queue again. Repeat this procedure until all the digits are iterated.

I did a experiment to compare backtracking(DFS) method and this iterative method. It turns out iterative one is 4 times faster.

One minor bug here. We need to add some code to test whether the input is empty or not. Above ans.add(""); add

**if** (digits.length()==0){

**return** ans;

}

构造法

method **combine** is to add new letters to old list, using 2 for-loop.

for example:

gave digits = "23"

i=0 -> result=combine("abc", [""]) ---> [a,b,c];

i=1 -> result=combine("def", [a,b,c]) ---> [ad,bd,cd, ae,be,ce, af,bf,cf];

public class Solution {

public static List<**String**> letterCombinations(String digits) {

String digitletter[] = {"","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};

List<**String**> result = new ArrayList<**String**>();

if (digits.length()==0) return result;

result.add("");

for (int i=0; i<**digits.length();** i++)

result = combine(digitletter[digits.charAt(i)-'0'],result);

return result;

}

public static List<String> combine(String digit, List<**String**> l) {

List<**String**> result = new ArrayList<**String**>();

for (int i=0; i<**digit.length();** i++)

for (String x : l)

result.add(x+digit.charAt(i));

return result;

}

}

### Generate Parentheses

[My Submissions](https://leetcode.com/problems/generate-parentheses/submissions/)

QuestionEditorial Solution

Total Accepted: **86421** Total Submissions: **233774** Difficulty: **Medium**

Given *n* pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given *n* = 3, a solution set is:

"((()))", "(()())", "(())()", "()(())", "()()()"

|  |  |
| --- | --- |
| static [**String**](file:///E:\docsJava\api\java\lang\String.html) | [**copyValueOf**](file:///E:\docsJava\api\java\lang\String.html#copyValueOf-char:A-)(char[] data)  Equivalent to [**valueOf(char[])**](file:///E:\docsJava\api\java\lang\String.html#valueOf-char:A-). |

书上做法2

从头开始构造字符串（逐一加入左括号，右括号，只要字符串仍然有效）

当左括号个数大于0时，继续添加左括号；当右括号个数大于左括号时才可以添加右括号

注意这个函数的使用 String.copyValueOf(str)

public class Solution {

List<String> res=new ArrayList<String>();

public List<String> generateParenthesis(int n) {

char[] str=new char[n\*2];//记录中间结果

generate(0,str,n,n);

return res;

}

public void generate(int cnt,char[] str,int left, int right)

{

if(left<0 || right<left) return;

if(left==0 && right==0)

{

String tmp=String.copyValueOf(str);

res.add(tmp);

return;

}

if(left>0)

{

str[cnt]='(';

generate(cnt+1,str,left-1,right);

}

if(right>left)

{

str[cnt]=')';

generate(cnt+1,str,left,right-1);

}

}

}

可以写成字符串作为中间结果的形式

public class Solution {

List<String> res=new ArrayList<String>();

public List<String> generateParenthesis(int n) {

//char[] str=new char[n\*2];//记录中间结果

String str="";

generate(0,str,n,n);

return res;

}

public void generate(int cnt,String str,int left, int right)

{

if(left<0 || right<left) return;

if(left==0 && right==0)

{

//String tmp=String.copyValueOf(str);

res.add(str);

return;

}

if(left>0)

{

//str[cnt]='(';

generate(cnt+1,str+"(",left-1,right);

}

if(right>left)

{

//str[cnt]=')';

generate(cnt+1,str+")",left,right-1);

}

}

}

The idea is intuitive. Use two integers to count the remaining left parenthesis (n) and the right parenthesis (m) to be added. At each function call add a left parenthesis if n >0 and add a right parenthesis if m>0. Append the result and terminate recursive calls when both m and n are zero.

**class** Solution {

**public**:

**vector**<**string**> generateParenthesis(**int** n) {

**vector**<**string**> res;

addingpar(res, "", n, 0);

**return** res;

}

**void** addingpar(**vector**<**string**> &v, **string** str, **int** n, **int** m){

**if**(n==0 && m==0) {

v.push\_back(str);

**return**;

}

**if**(m > 0){ addingpar(v, str+")", n, m-1); }

**if**(n > 0){ addingpar(v, str+"(", n-1, m+1); }

}

};

My method is DP. First consider how to get the result f(n) from previous result f(0)...f(n-1). Actually, the result f(n) will be put an extra () pair to f(n-1). Let the "(" always at the first position, to produce a valid result, we can only put ")" in a way that there will be i pairs () inside the extra () and n - 1 - i pairs () outside the extra pair.

Let us consider an example to get clear view:

f(0): ""

f(1): "("f(0)")"

f(2): "("f(0)")"f(1), "("f(1)")"

f(3): "("f(0)")"f(2), "("f(1)")"f(1), "("f(2)")"

So f(n) = "("f(0)")"f(n-1) , "("f(1)")"f(n-2) "("f(2)")"f(n-3) ... "("f(i)")"f(n-1-i) ... "(f(n-1)")"

Below is my code:

**public** **class** **Solution**

{

**public** **List**<String> generateParenthesis(int n)

{

**List**<**List**<String>> lists = **new** ArrayList<>();

lists.add(Collections.singletonList(""));

**for** (int i = 1; i <= n; ++i)

{

**final** **List**<String> **list** = **new** ArrayList<>();

**for** (int j = 0; j < i; ++j)

{

**for** (**final** String first : lists.get(j))

{

**for** (**final** String second : lists.get(i - 1 - j))

{

**list**.add("(" + first + ")" + second);

}

}

}

lists.add(**list**);

}

**return** lists.get(lists.size() - 1);

}

}

I think it's useful to prove this equation.

The equation is equivalent to the following one:

f(n) = (f(0))f(n-1) + (f(1))f(n-2) + ... + (f(n-2))f(1) + (f(n-1))f(0)

First, let f(n) to be a correct solution set when there is n pair of parentheses. This means every combination in f(n) is a valid combination, and any combination which isn't in f(n) is not a valid combination for n. And we can easily get the first three solution sets i.e. f(0) = {""}, f(1) = {"()"} f(2) = {"()()", "(())"}.

For any n > 2, each combination of f(n) can be divided into two parts p0 and p1. p0 and p1 has several properties:

1. Parentheses in both p0 and p1 can match wel
2. p0 should be as short as possible but not empty. This means that p0 belongs to (f(l0-1)) where l0 is the number of pairs in p0. This property can be proved easily. Shortest means the first left parenthesis in this combination always matches the last right parenthesis. So without these two, what left is also a legal combination.

Now, let's reorganize f(n) by p0. Put those combinations with same length of p0 into a same set, and then f(n) is divided into several subsets. Each combination in subset s whose p0 has l0 pair of parentheses also belongs to the set (f(l0-1))f(n-l0). So we can get f(n) belongs to (f(0))f(n-1) + (f(1))f(n-2) + ... + (f(n-2))f(1) + (f(n-1))f(0).

OK, the only thing to prove now is (f(0))f(n-1) + (f(1))f(n-2) + ... + (f(n-2))f(1) + (f(n-1))f(0) also belongs to f(n). Notice that each combination in (f(i))f(n-1-i) is a legal combination for n, and we've declared before that each legal combination for n belongs to f(n). So each combination in the left side of equation belongs to f(n), and the left side as a whole set belongs to f(n).

Prove complete.

public List<String> generateParenthesis(int n) {

List<String> list = new ArrayList<String>();

backtrack(list, "", 0, 0, n);

**return** list;

}

public void backtrack(List<String> list, String str, int open, int close, int max){

**if**(str.length() == max\*2){

list.add(str);

**return**;

}

**if**(open < max)

backtrack(list, str+"(", open+1, close, max);

**if**(close < open)

backtrack(list, str+")", open, close+1, max);

}

The idea here is to only add '(' and ')' that we know will guarantee us a solution (instead of adding 1 too many close). Once we add a '(' we will then discard it and try a ')' which can only close a valid '('. Each of these steps are recursively called.

For 2, it should place one "()" and add another one insert it but none tail it,

'(' f(1) ')' f(0)

or add none insert it but tail it by another one,

'(' f(0) ')' f(1)

Thus for n, we can insert f(i) and tail f(j) and i+j=n-1,

'(' f(i) ')' f(j)

public List<**String**> generateParenthesis(int n) {

List<**String**> result = new ArrayList<**String**>();

if (n == 0) {

result.add("");

} else {

for (int i = n - 1; i >= 0; i--) {

List<**String**> insertSub = generateParenthesis(i);

List<**String**> tailSub = generateParenthesis(n - 1 - i);

for (String insert : insertSub) {

for (String tail : tailSub) {

result.add("(" + insert + ")" + tail);

}

}

}

}

return result;

}

It's better to use dynamic programming to avoid computing result for same i.

### Subsets II

[My Submissions](https://leetcode.com/problems/subsets-ii/submissions/)

QuestionEditorial Solution

Total Accepted: **66157** Total Submissions: **216852** Difficulty: **Medium**

Given a collection of integers that might contain duplicates, ***nums***, return all possible subsets.

**Note:**

* Elements in a subset must be in non-descending order.
* 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],

[]

]

To solve this problem, it is helpful to first think how many subsets are there. If there is no duplicate element, the answer is simply 2^n, where n is the number of elements. This is because you have two choices for each element, either putting it into the subset or not. So all subsets for this no-duplicate set can be easily constructed: num of subset

* (1 to 2^0) empty set is the first subset
* (2^0+1 to 2^1) add the first element into subset from (1)
* (2^1+1 to 2^2) add the second element into subset (1 to 2^1)
* (2^2+1 to 2^3) add the third element into subset (1 to 2^2)
* ....
* (2^(n-1)+1 to 2^n) add the nth element into subset(1 to 2^(n-1))

Then how many subsets are there if there are duplicate elements? We can treat duplicate element as a spacial element. For example, if we have duplicate elements (5, 5), instead of treating them as two elements that are duplicate, we can treat it as one special element 5, but this element has more than two choices: you can either NOT put it into the subset, or put ONE 5 into the subset, or put TWO 5s into the subset. Therefore, we are given an array (a1, a2, a3, ..., an) with each of them appearing (k1, k2, k3, ..., kn) times, the number of subset is (k1+1)*(k2+1)*...(kn+1). We can easily see how to write down all the subsets similar to the approach above.

public class Solution {

public List<**List<Integer**>> subsetsWithDup(int[] num) {

List<**List<Integer**>> result = new ArrayList<**List<Integer**>>();

List<**Integer**> empty = new ArrayList<**Integer**>();

result.add(empty);

Arrays.sort(num);

for (int i = 0; i < num.length; i++) {

int dupCount = 0;

while( ((i+1) < num.length) && num[i+1] == num[i]) {

dupCount++;

i++;

}

int prevNum = result.size();

for (int j = 0; j < prevNum; j++) {

List<Integer> element = new ArrayList<**Integer**>(result.get(j));

for (int t = 0; t <**=** dupCount; t++) {

element.add(num[i]);

result.add(new ArrayList<Integer>(element));

}

}

}

return result;

}

}

I have a question regarding the link of code for(int j=0; j<count; j++) in C++ version. if duplication count is zero, how do we add the new subset to the total set? I am thinking at least we need to jump into for loop at least one time even for the non-duplicate set .

这个方法非常不错,如果当前数与前一个数重复，则只需要从上一轮插入的元素里再操作就行。

If we want to insert an element which is a dup, we can only insert it after the newly inserted elements from last step.

**Java** version of the elegant solution. you are welcome!

public List<**List<Integer**>> subsetsWithDup(int[] num) {

Arrays.sort(num);

List<**List<Integer**>> ret = new ArrayList<>();

ret.add(new ArrayList<**Integer**>());

int size = 0, startIndex;

for(int i = 0; i < num.length; i++) {

startIndex = (i >= 1 && num[i] == num[i - 1]) ? size : 0;//此时的size是上一轮的大小

size = ret.size();

for(int j = startIndex; j < size; j++) {

List<Integer> temp = new ArrayList<>(ret.get(j));

temp.add(num[i]);

ret.add(temp);

}

}

return ret;

}

public List<**List<Integer**>> subsetsWithDup(int[] nums) {

Arrays.sort(nums);

List<**List<Integer**>> res = new ArrayList<>();

List<**Integer**> each = new ArrayList<>();

helper(res, each, 0, nums);

return res;

}

public void helper(List<**List<Integer**>> res, List<**Integer**> each, int pos, int[] n) {

if (pos <**=** n.length) {

res.add(each);

}

int i = pos;

while (i < n.length) {

each.add(n[i]);

helper(res, new ArrayList<>(each), i + 1, n);

each.remove(each.size() - 1);

i++;

while (i < n.length && n[i] == n[i - 1]) {i++;}

}

return;

}

The Basic idea is: use "while (i < n.length && n[i] == n[i - 1]) {i++;}" to avoid the duplicate. For example, the input is 2 2 2 3 4. Consider the helper function. The process is:

* each.add(n[i]); --> add first 2 (index 0)
* helper(res, new ArrayList<>(each), i + 1, n); --> go to recursion part, list each is <2 (index 0)>
* while (i < n.length && n[i] == n[i - 1]) {i++;} --> after this, i == 3, add the element as in subset I

### 列举所有出栈顺序（微软面试）

**这个方法也不错**

**将字符串（入栈序列）不断减短，当剩下的字符串长度为0，进行出栈打印处理。**

**1、算法：**

import java.util.Stack;

public class OrderOut {  
 public static void orderList(Stack<String> stack,String result,String input){  
  //利用对象克隆的方法，先把栈复制下来以免操作后影响下面  
  Stack<String> temp=(Stack<String>)stack.clone();  
  String subStr=input.substring(0,1);  
  input=input.substring(1);  
  temp.push(subStr);  
  if(input.length()==0){  
   while(!temp.isEmpty()){  
    result+=temp.pop().toString();  
   }  
   System.out.println(result);  
  }  
  else{  
   orderList(temp,result,input);  
   while(!temp.isEmpty()){  
    result+=temp.pop();  
    orderList(temp,result,input);  
   }  
  }  
 }  
}

**2、测试：**

import java.util.Stack;  
import xuzhenzhen.datastructure.model.OrderOut;

public class StackOutTest {

 public static void main(String[] args) {  
  //测试列举所有出栈顺序  
  Stack<String> stack=new Stack<String>();  
  OrderOut.orderList(stack, "", "1234");  
 }

}

**3、结果：**

4321  
3421  
3241  
3214  
2431  
2341  
2314  
2143  
2134  
1432  
1342  
1324  
1243  
1234

下面这个方法非常好，类似于打印n对括号

将出栈序列，转换成求0,1序列（有效的左右括号序列）。正确的出栈，前缀子序列中1的个数必大于0的个数。

网上有很多解法，但个人感觉不够清晰。下面本人献丑来写下自己的解法。力求简明易懂。首先这是个卡特兰数，学过组合数学的同学都知道。没学过的可以看下下面这个例子。

有2n个人排成一队进入剧场。入场费5元。其中只有n个人有一张5元钞票，另外n人只有10元钞票，剧院无其它钞票可找零，问有多少中方法使得只要有10元的人买票，售票处就有5元的钞票找零？(将持5元者到达视作将5元入栈，持10元者到达视作使栈中某5元出栈)。

对于这个例子，剧院要想总有零钱可找，那么目前进入剧院的人数中，揣着10元钞票的人数必须少于等于揣着5元钞票的，不然肯定在某个人那出现没零钱找的情况。

现在回到正题上来对于一个给定入栈序列，怎么求它的出栈序列呢？

我们可以把入栈记为1，出栈记为0.那么前缀子序列中1的个数必须大于等于0的个数，即入栈次数要大于等于出栈次数,如1 1 0 1 0 0，它的任意前缀序列中1的个数是大于等于0的个数的。

我们来看个例子:对于1 2 3这个入栈序列，1 1 0 1 0 0就是一个入栈出栈序列，第一个1代表元素1入栈，然后第二个1代表元素2入栈，然后第三个是0，代表出栈，即元素2出栈，然后第四个是1，代表元素3入栈，然后第五个是0，代表出栈，即元素3出栈，然后第六个是0，代表元素1出栈。最后1 1 0 1 0 0就代表了出栈序列2 3 1。

那么现在的问题就转换为如何求出所有符合条件的0 1序列了。其实这和以下问题相同:给定括号对数，输出所有符合要求的序列。如2对括号，输出有()()或者(())两种。1可以看成'('，0可以看成‘)’。

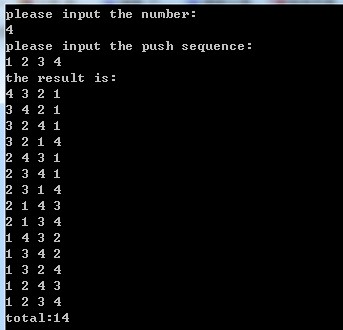
下面贴上本人的程序，并给出详细注释。

#include <iostream>  
#include <vector>  
using namespace std;  
  
  
void func(vector<char>kind,int count[],int n)  
{  
    if(count[0]>=1)  
    {  
        kind.push\_back('(');  
        count[0]--;  
        func(kind,count,n);  
        count[0]++;  
        kind.pop\_back();  
    }  
    if((count[1]>=1) && (count[1]>count[0]))  
    {  
        kind.push\_back(')');  
        count[1]--;  
        func(kind,count,n);  
        count[1]++;  
        kind.pop\_back();  
    }  
    if(kind.size()==2\*n)  
    {  
        vector<char>::iterator iter;  
        for(iter=kind.begin();iter!=kind.end();iter++)  
        {  
            cout<<(\*iter)<<" ";  
        }  
        cout<<endl;  
    }  
}  
  
  
int main()  
{  
    int n;  
    cout << "please input the number of ():" << endl;  
    cin>>n;  
    int count[2]={n-1,n};  
    vector<char>kind;  
    kind.push\_back('(');  
    func(kind,count,n);  
    return 0;  
}

count[0]存着左括号数目，count[1]存着右括号数目。一开始kind中压入左括号，因为第一个肯定是左括号。然后count数组初始化为n-1个左括号，n个右括号。然后我们递归的处理。如果剩余左括号数count[0]大于0，就可以把左括号压栈。而对于右括号，栈中左括号个数必须多于右括号个数，也就是剩余右括号个数大于左括号个数，即count[1]>count[0]时，才能将右括号压栈。如果栈中元素个数达到2n时，就把栈中元素输出。

下面贴出出栈序列代码，几乎和上面相同。

#include <iostream>  
#include <stack>  
#include <vector>  
using namespace std;  
  
  
int number=0;  
void func(vector<int>kind,int count[],int n,int A[])  
{  
    if(count[0]>=1)  
    {  
        kind.push\_back(1);  
        count[0]--;  
        func(kind,count,n,A);  
        count[0]++;  
        kind.pop\_back();  
    }  
    if((count[1]>=1) && (count[1]>count[0]))  
    {  
        kind.push\_back(0);  
        count[1]--;  
        func(kind,count,n,A);  
        count[1]++;  
        kind.pop\_back();  
    }  
    if(kind.size()==2\*n)  
    {  
        vector<int>::iterator iter;  
        stack<int>stk;  
        int j=0;  
        for(iter=kind.begin();iter!=kind.end();iter++)  
        {  
            //cout<<(\*iter)<<" ";  
            if(1==(\*iter))  
            {  
                stk.push(A[j]);  
                j++;  
            }  
            else  
            {  
                cout<<stk.top()<<" ";  
                stk.pop();  
            }  
        }  
        number++;  
        cout<<endl;  
    }  
}  
  
  
int main()  
{  
    int n,i;  
    cout << "please input the number:" << endl;  
    cin>>n;  
    int A[n];  
    cout << "please input the push sequence:" << endl;  
    for(i=0;i<n;i++)  
    {  
        cin>>A[i];  
    }  
    int count[2]={n-1,n};  
    vector<int>kind;  
    kind.push\_back(1);

    cout<<"the result is:"<<endl;  
    func(kind,count,n,A);  
    cout<<"total:"<<number<<endl;  
    return 0;  
}  
  


### Permutations

[My Submissions](https://leetcode.com/problems/permutations/submissions/)

QuestionEditorial Solution

Total Accepted: **98018** Total Submissions: **274245** Difficulty: **Medium**

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], and [3,2,1].

This recursive solution is the my first response for this problem. I was surprised when I found no similar solution posted here. It is much easier to understand than DFS-based ones, at least in my opinion. Please find more explanations [here](http://xiaohuiliucuriosity.blogspot.com/2014/12/permutations.html). All comments are welcome.

**class** Solution {

**public**:

**vector**<**vector**<**int**> > permute(**vector**<**int**> &num) {

**vector**<**vector**<**int**> > result;

permuteRecursive(num, 0, result);

**return** result;

}

// permute num[begin..end]

// invariant: num[0..begin-1] have been fixed/permuted

**void** permuteRecursive(**vector**<**int**> &num, **int** begin, **vector**<**vector**<**int**> > &result) {

**if** (begin >= num.size()) {

// one permutation instance

result.push\_back(num);

**return**;

}

**for** (**int** i = begin; i < num.size(); i++) {

swap(num[begin], num[i]);

permuteRecursive(num, begin + 1, result);

// reset

swap(num[begin], num[i]);

}

}

};

If you don't reset then you will never get those permutation starting with the 3rd number because you are changing the state of the array. for example: 123 -> 132 -> then what ? So you need to reset so that you first get all the permutations starting with 1 then 2 and then 3.

in case of string this would not be the case coz they are immutable.

这个方法也很不错

不断换新的数到位置1，位置1固定后，之后的位置再排列

Thanks for smart code. Following is the Java version.

public class Solution {

public List<**List<Integer**>> permute(int[] num) {

List<**List<Integer**>> result = new ArrayList<**List<Integer**>>();

permute(num,0,result);

return result;}

public void permute(int[] num, int begin, List<**List<Integer**>> result){

if(begin>=num.length){

List<**Integer**> list = new ArrayList<**Integer**>();

for(int i=0;i<**num.length;i++){**

list.add(num[i]);

}

result.add(list);

return;

}

for(int i=begin;i<num.length;i++){

swap(begin,i,num);

permute(num,begin+1,result);

swap(begin,i,num);

}

}

public void swap (int x, int y,int[] num){

int temp = num[x];

num[x]=num[y];

num[y]=temp;

} }

my AC simple iterative java/python solution

the basic idea is, to permute n numbers, we can add the nth number into the resultingList<List<Integer>> from the n-1 numbers, in every possible position.

For example, if the input num[] is {1,2,3}: First, add 1 into the initial List<List<Integer>> (let's call it "answer").

Then, 2 can be added in front or after 1. So we have to copy the List in answer (it's just {1}), add 2 in position 0 of {1}, then copy the original {1} again, and add 2 in position 1. Now we have an answer of {{2,1},{1,2}}. There are 2 lists in the current answer.

Then we have to add 3. first copy {2,1} and {1,2}, add 3 in position 0; then copy {2,1} and {1,2}, and add 3 into position 1, then do the same thing for position 3. Finally we have 2\*3=6 lists in answer, which is what we want.

public List<**List<Integer**>> permute(int[] num) {

List<**List<Integer**>> ans = new ArrayList<**List<Integer**>>();

if (num.length ==0) return ans;

List<**Integer**> l0 = new ArrayList<**Integer**>();

l0.add(num[0]);

ans.add(l0);

for (int i = 1; i< num.length; ++i){

List<List<Integer>> new\_ans = new ArrayList<**List<Integer**>>();

for (int j = 0; j<**=i;** ++j){

for (List<Integer> l : ans){

List<**Integer**> new\_l = new ArrayList<**Integer**>(l);

new\_l.add(j,num[i]);

new\_ans.add(new\_l);

}

}

ans = new\_ans;

}

return ans;

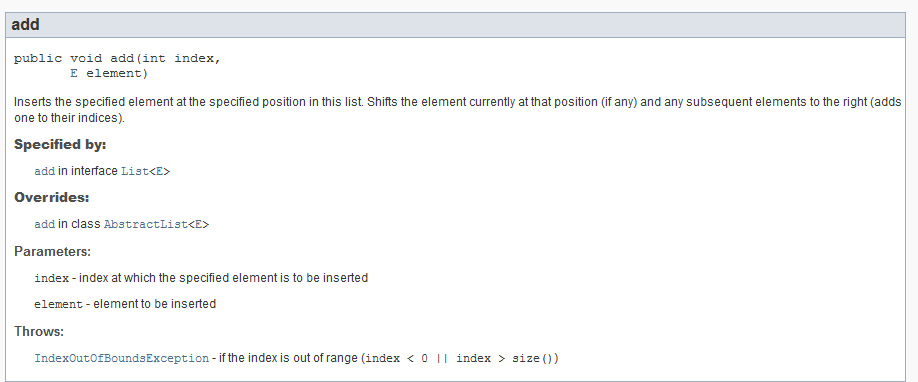
}

A tiny improvement for Java solution, we don't really need the special case for the first element. Justans.add(l0); add the empty list and let the loop starts from 0 for (int i = 0; i< num.length; ++i){ :)

这个递归写法也不错(比迭代容易些)

I used your idea of adding each next value to every possible position of current list, but have done it with recursion.

ArrayList方法



public List<**List<Integer**>> permute(int[] nums) {

List<**List<Integer**>> result = new ArrayList<**List<Integer**>>();

if (nums.length == 0) return result;

backtrack(result, nums, new ArrayList<**Integer**>(), 0);

return result;

}

private void backtrack(List<**List<Integer**>> result, int[] nums, List<**Integer**> currentList, int index) {

if (currentList.size() == nums.length) {

result.add(currentList);

return;

}

int n = nums[index];

for (int i = 0; i <**=** currentList.size(); i++) {

List<Integer> copy = new ArrayList<**Integer**>(currentList);

copy.add(i, n);

backtrack(result, nums, copy, index + 1);

}

}

I guess both solutions have the same complexity

这个写法不错

原先构造一个空的list，将第一个数放入。

放入第二个数时，考虑可以插入到第一个数的哪个位子上。

放入第三个数时，考虑在原先的两种情况下，分别放入第三个数的位子。

有点类似宽搜，层次遍历

public List<**List<Integer**>> permute(int[] num) {

LinkedList<**List<Integer**>> res = new LinkedList<**List<Integer**>>();

res.add(new ArrayList<**Integer**>());

for (int n : num) {

int size = res.size();

for (; size > 0; size--) {

List<**Integer**> r = res.pollFirst();//记得出栈之前元素

for (int i = 0; i <**=** r.size(); i++) {

List<Integer> t = new ArrayList<**Integer**>(r);

t.add(i, n);

res.add(t);

}

}

}

return res;

}

**Bottom up? approach - 280ms**

**public** **class** **Solution** {

**public** **List**<**List**<Integer>> permute(int[] nums) {

**List**<**List**<Integer>> permutations = **new** ArrayList<>();

**if** (nums.length == 0) {

**return** permutations;

}

collectPermutations(nums, 0, **new** ArrayList<>(), permutations);

**return** permutations;

}

**private** void collectPermutations(int[] nums, int start, **List**<Integer> permutation,

**List**<**List**<Integer>> permutations) {

**if** (permutation.size() == nums.length) {

permutations.add(permutation);

**return**;

}

**for** (int i = 0; i <= permutation.size(); i++) {

**List**<Integer> newPermutation = **new** ArrayList<>(permutation);

newPermutation.add(i, nums[start]);

collectPermutations(nums, start + 1, newPermutation, permutations);

}

}

}

***Code flow***

nums = 1,2,3

start = 0, permutation = []

i = 0, newPermutation = [1]

start = 1, permutation = [1]

i = 0, newPermutation = [2, 1]

start = 2, permutation = [2, 1]

i = 0, newPermutation = [3, 2, 1]

i = 1, newPermutation = [2, 3, 1]

i = 2, newPermutation = [2, 1, 3]

i = 1, newPermutation = [1, 2]

start = 2, permutation = [1, 2]

i = 0, newPermutation = [3, 1, 2]

i = 1, newPermutation = [1, 3, 2]

i = 2, newPermutation = [1, 2, 3]

**Base case and build approach - 524ms**

public class Solution {

public List<**List<Integer**>> permute(int[] nums) {

return permute(Arrays.stream(nums).boxed().collect(Collectors.toList()));

}

private List<**List<Integer**>> permute(List<**Integer**> nums) {

List<**List<Integer**>> permutations = new ArrayList<>();

if (nums.size() == 0) {

return permutations;

}

if (nums.size() == 1) {

List<**Integer**> permutation = new ArrayList<>();

permutation.add(nums.get(0));

permutations.add(permutation);

return permutations;

}

List<**List<Integer**>> smallPermutations = permute(nums.subList(1, nums.size()));

int first = nums.get(0);

for(List<**Integer**> permutation : smallPermutations) {

for (int i = 0; i <**=** permutation.size(); i++) {

List<Integer> newPermutation = new ArrayList<>(permutation);

newPermutation.add(i, first);

permutations.add(newPermutation);

}

}

return permutations;

}

}

***Code flow***

nums = 1,2,3

smallPermutations(2, 3)

smallPermutations(3)

**return** [[3]]

first = 2

permutation = [3]

i = 0, newPermutation = [2, 3]

i = 1, newPermutation = [3, 2]

**return** [[2, 3], [3, 2]]

first = 1

permutation = [2, 3]

i = 0, newPermutation = [1, 2, 3]

i = 1, newPermutation = [2, 1, 3]

i = 2, newPermutation = [2, 3, 1]

permutation = [3, 2]

i = 0, newPermutation = [1, 3, 2]

i = 1, newPermutation = [3, 1, 2]

i = 2, newPermutation = [3, 2, 1]

### Permutations II

[My Submissions](https://leetcode.com/problems/permutations-ii/submissions/)

QuestionEditorial Solution

Total Accepted: **69644** Total Submissions: **248784** Difficulty: **Medium**

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

For example,  
[1,1,2] have the following unique permutations:  
[1,1,2], [1,2,1], and [2,1,1].

下面的方法不太理解

**class** Solution {

**public**:

**void** recursion(**vector**<**int**> num, **int** i, **int** j, **vector**<**vector**<**int**> > &res) {

**if** (i == j-1) {

res.push\_back(num);

**return**;

}

**for** (**int** k = i; k < j; k++) {//现在要排位置i，后面的数与i对应处相等时，不用处理

**if** (i != k && num[i] == num[k]) **continue**;

swap(num[i], num[k]);

recursion(num, i+1, j, res);//处理后也不用交换回来

}

}

**vector**<**vector**<**int**> > permuteUnique(**vector**<**int**> &num) {

sort(num.begin(), num.end());

**vector**<**vector**<**int**> >res;

recursion(num, 0, num.size(), res);

**return** res;

}

};

Backtracking is a nightmare for this problem.

The solution of "backtracking" is not efficient in spece since its creating arrays memories for each depth of the recursion function. If you do not pass by value, the swap action will disturb the sorted sequence and you are going to meet repeating answers.

One way to do it is to take advantage of nextPermutaion, which is to find the next larger permutation. And loop the nextPermutation function to find the complete unique permutation sequences.

Another way to solve it is DFS. DFS does not swap the array elements and preserves the sorted property.

For example you have want to solve permuteUnique[3,2,0,3,1,0,1]

Sort it you get

permuteUnique[0,0,1,1,2,3,3] =

0, permuteUnique[0,1,1,2,3,3]

1, permuteUnique[0,0,1,2,3,3]

2, permuteUnique[0,0,1,1,3,3]

3, permuteUnique[0,0,1,1,2,3]

and then you solve the sub question.

Sidenote: make sure in the current-depth DFS you don’t pick multiple duplicate and dig into sub-question.

**public** **class** **Solution** {

**public** **List**<**List**<Integer>> permuteUnique(int[] nums) {

Arrays.sort(nums);

**List**<**List**<Integer>> ans = **new** ArrayList<**List**<Integer>>();

**List**<Integer> cur = **new** ArrayList<Integer>();

boolean[] used = **new** boolean[nums.length];

solve(0, nums, ans, cur, used);

**return** ans;

}

**private** void solve(int depth, int[] nums, **List**<**List**<Integer>> ans, **List**<Integer> cur, boolean[] used) {

**if** (depth == nums.length) {

ans.add(**new** ArrayList<Integer>(cur));

**return**;

}

**for** (int i = 0; i < nums.length; i++) {

**if** (used[i] || (i > 0 && nums[i - 1] == nums[i] && !used[i - 1])) **continue**;

used[i] = **true**; cur.add(nums[i]);

solve(depth + 1, nums, ans, cur, used);

used[i] = **false**; cur.remove(cur.size() - 1);

}

}

}

这个方法也不错

Use an extra boolean array " boolean[] used" to indicate whether the value is added to list.

Sort the array "int[] nums" to make sure we can skip the same value.

when a number has the same value with its previous, we can use this number only if his previous is used

**public** **class** **Solution** {

**public** **List**<**List**<Integer>> permuteUnique(int[] nums) {

**List**<**List**<Integer>> res = **new** ArrayList<**List**<Integer>>();

**if**(nums==**null** || nums.length==0) **return** res;

boolean[] used = **new** boolean[nums.length];

**List**<Integer> **list** = **new** ArrayList<Integer>();

Arrays.sort(nums);

dfs(nums, used, **list**, 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** ArrayList<Integer>(**list**));

**return**;

}

**for**(int i=0;i<nums.length;i++){

**if**(used[i]) **continue**;

**if**(i>0 &&nums[i-1]==nums[i] && !used[i-1]) **continue**;

used[i]=**true**;

**list**.add(nums[i]);

dfs(nums,used,**list**,res);

used[i]=**false**;

**list**.remove(**list**.size()-1);

}

}

}

Hi guys!

Here's an iterative solution which doesn't use nextPermutation helper. It builds the permutations for i-1 first elements of an input array and tries to insert the ith element into all positions of each prebuilt i-1 permutation. I couldn't come up with more effective controling of uniqueness than just using a Set.

See the code below!

public class Solution {

public List<**List<Integer**>> permuteUnique(int[] num) {

LinkedList<**List<Integer**>> res = new LinkedList<>();

res.add(new ArrayList<>());

for (int i = 0; i < num.length; i++) {

Set<String> cache = new HashSet<>();

while (res.peekFirst().size() == i) {

List<**Integer**> l = res.removeFirst();

for (int j = 0; j <**=** l.size(); j++) {

List<Integer> newL = new ArrayList<>(l.subList(0,j));

newL.add(num[i]);

newL.addAll(l.subList(j,l.size()));

if (cache.add(newL.toString())) res.add(newL);

}

}

}

return res;

}

}

**和上一题方法类似，只不过使用一个set集合记录所有不同的数**

**Set来标记在这个位置已经有这些元素了，如果重复，则跳过**

**这个方法不错**

**public** **class** **Solution** {

**public** List<List<Integer>> permuteUnique(**int**[] nums) {

List<List<Integer>> ans = **new** ArrayList<>();

**if** (nums==**null** || nums.length==0) { **return** ans; }

permute(ans, nums, 0);

**return** ans;

}

**private** **void** permute(List<List<Integer>> ans, **int**[] nums, **int** index) {

**if** (index == nums.length) {

List<Integer> temp = **new** ArrayList<>();

**for** (**int** num: nums) { temp.add(num); }

ans.add(temp);

**return**;

}

Set<Integer> appeared = **new** HashSet<>();//每轮排位置使用一个set

**for** (**int** i=index; i<nums.length; ++i) {

**if** (appeared.add(nums[i])) {

swap(nums, index, i);

permute(ans, nums, index+1);

swap(nums, index, i);

}

}

}

**private** **void** swap(**int**[] nums, **int** i, **int** j) {

**int** save = nums[i];

nums[i] = nums[j];

nums[j] = save;

}

}

Since we only need permutations of the array, the actual "content" does not change, we could find each permutation by swapping the elements in the array.

The idea is for each recursion level, swap the **current element at 1st index** with each element that comes after it (including itself). For example, permute[1,2,3]:

At recursion level 0, current element at 1st index is 1, there are 3 possibilities: [1] + permute[2,3], [2] + permute[1,3], [3] + permute[2,1].

Take "2+permute[1,3]" as the example at recursion level 0. At recursion level 1, current elemenet at 1st index is 1, there are 2 possibilities: [2,1] + permute[3], [2,3] + permute[1].

... and so on.

Let's look at another example, permute[1,2,3,4,1].

At recursion level 0, we have [1] + permute[2,3,4,1], [2] + permute[1,3,4,1], [3] + permute[2,1,4,1], [4] + permute[2,3,1,1], **[1] + permute[2,3,4,1]**.

1 has already been at the 1st index of current recursion level, so the last possibility is redundant. We can use a hash set to mark which elements have been at the 1st index of current recursion level, so that if we meet the element again, we can just skip it.

### Next Permutation

[My Submissions](https://leetcode.com/problems/next-permutation/submissions/)

QuestionEditorial Solution

Total Accepted: **64695** Total Submissions: **244159** Difficulty: **Medium**

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.  
1,2,3 → 1,3,2  
3,2,1 → 1,2,3  
1,1,5 → 1,5,1

Share my O(n) time solution

这个方法不错

1 从最后一个元素开始，寻找索引i，使得索引i到n-1逆向有序。

2索引i到n-1的变换不会产生比其大的排列组合，所以我们希望，改变i-1处的值，需要从索引i到n-1的值中找到大于i-1处的值的最小值，两者交换（交换后索引i到n-1的值依然是逆序排列的，即索引i处的值最大，括号里的话有问题吧）

3 逆向排序索引i到n-1的值，使正序

My idea is for an array:

1. Start from its last element, traverse backward to find the first one with index i that satisfy num[i-1] < num[i]. So, elements from num[i] to num[n-1] is reversely sorted.
2. To find the next permutation, we have to swap some numbers at different positions, to minimize the increased amount, we have to make the highest changed position as high as possible. Notice that index larger than or equal to i is not possible as num[i,n-1] is reversely sorted. So, we want to increase the number at index i-1, clearly, swap it with the smallest number between num[i,n-1] that is larger than num[i-1]. For example, original number is 121543321, we want to swap the '1' at position 2 with '2' at position 7.
3. The last step is to make the remaining higher position part as small as possible, we just have to reversely sort the num[i,n-1]

The following is my code:

public void nextPermutation(**int**[] num) {

**int** n=num.**length**;

**if**(n<2)

**return**;

**int** **index**=n-1;

**while**(**index**>0){

**if**(num[**index**-1]<num[**index**])

break;

**index**--;

}

**if**(**index**==0){

reverseSort(num,0,n-1);

**return**;

}

**else**{

**int** val=num[**index**-1];

**int** j=n-1;

**while**(j>=**index**){ //从逆序数里（最小的开始）找大于位置index-1的数

**if**(num[j]>val)

break;

j--;

}

swap(num,j,**index**-1);

reverseSort(num,**index**,n-1);

**return**;

}

}

public void swap(**int**[] num, **int** i, **int** j){

**int** temp=0;

temp=num[i];

num[i]=num[j];

num[j]=temp;

}

public void reverseSort(**int**[] num, **int** start, **int** end){ //记住反转函数

**if**(start>end)

**return**;

**for**(**int** i=start;i<=(end+start)/2;i++)

swap(num,i,start+end-i);

}

Similar Solution, a bit fewer lines of code. Three steps:

1. Reverse find first number which breaks descending order.
2. Exchange this number with the least number that's greater than this number.
3. Reverse sort the numbers after the exchanged number.

**public** **class** **Solution** {

**public** **static** **void** nextPermutation(**int**[] num) {

**int** i = num.length - 2;

**for**(; i >= 0 && num[i] >= num[i+1]; i--)

;

**if**(i >= 0) {

**int** j = i + 1;

**for**(; j<num.length && num[i] < num[j]; j++)

;

exchange(num, i, j-1);

}

i ++ ;

**int** k = num.length - 1;

**for**(; i<k; i++, k--)

exchange(num, i, k);

}

**private** **static** **void** exchange(**int**[] num, **int** i, **int** j) {

**int** t = num[i];

num[i] = num[j];

num[j] = t;

}

}

### Permutation Sequence

[My Submissions](https://leetcode.com/problems/permutation-sequence/submissions/)

QuestionEditorial Solution

Total Accepted: **53519** Total Submissions: **212569** Difficulty: **Medium**

The set [1,2,3,…,*n*] contains a total of *n*! unique permutations.

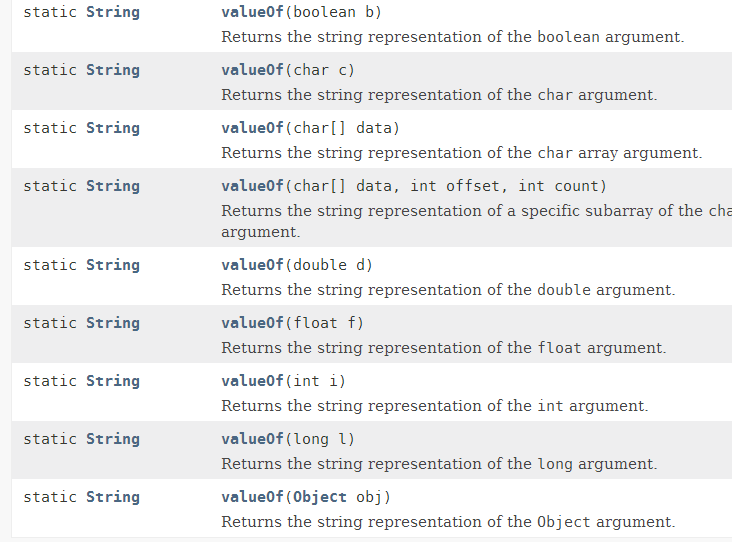
By listing and labeling all of the permutations in order,  
We get the following sequence (ie, for *n* = 3):

1. "123"
2. "132"
3. "213"
4. "231"
5. "312"
6. "321"

Given *n* and *k*, return the *k*th permutation sequence.

**Note:** Given *n* will be between 1 and 9 inclusive.

String valueOf方法（注意最后一条）



这个方法好

1,首先计算好阶乘，方便后续查看，k—

2，将数字放在ArrayList中，如果被使用则需要移除

3, 根据k的值依次除以阶乘数找到对应的index，并更新k

"Explain-like-I'm-five" Java Solution in O(n)

I'm sure somewhere can be simplified so it'd be nice if anyone can let me know. The pattern was that:

say n = 4, you have {1, 2, 3, 4}

If you were to list out all the permutations you have

1 + (permutations of 2, 3, 4)   
2 + (permutations of 1, 3, 4)   
3 + (permutations of 1, 2, 4)   
4 + (permutations of 1, 2, 3)

We know how to calculate the number of permutations of n numbers... n! So each of those with permutations of 3 numbers means there are 6 possible permutations. Meaning there would be a total of 24 permutations in this particular one. So if you were to look for the (k = 14) 14th permutation, it would be in the

3 + (permutations of 1, 2, 4) subset.

To programmatically get that, you take k = 13 (subtract 1 because of things always starting at 0) and divide that by the 6 we got from the factorial, which would give you the index of the number you want. In the array {1, 2, 3, 4}, k/(n-1)! = 13/(4-1)! = 13/3! = 13/6 = 2. The array {1, 2, 3, 4} has a value of 3 at index 2. So the first number is a 3.

Then the problem repeats with less numbers.

The permutations of {1, 2, 4} would be:

1 + (permutations of 2, 4)   
2 + (permutations of 1, 4)   
4 + (permutations of 1, 2)

But our k is no longer the 14th, because in the previous step, we've already eliminated the 12 4-number permutations starting with 1 and 2. So you subtract 12 from k.. which gives you 1. Programmatically that would be...

k = k - (index from previous) \* (n-1)! = k - 2(n-1)! = 13 - 2(3)! = 1

In this second step, permutations of 2 numbers has only 2 possibilities, meaning each of the three permutations listed above a has two possibilities, giving a total of 6. We're looking for the first one, so that would be in the 1 + (permutations of 2, 4) subset.

Meaning: index to get number from is k / (n - 2)! = 1 / (4-2)! = 1 / 2! = 0.. from {1, 2, 4}, index 0 is 1

so the numbers we have so far is 3, 1... and then repeating without explanations.

{2, 4}   
k = k - (index from pervious) \* (n-2)! = k - 0 \* (n - 2)! = 1 - 0 = 1;   
third number's index = k / (n - 3)! = 1 / (4-3)! = 1/ 1! = 1... from {2, 4}, index 1 has 4   
Third number is 4

{2}   
k = k - (index from pervious) \* (n - 3)! = k - 1 \* (4 - 3)! = 1 - 1 = 0;   
third number's index = k / (n - 4)! = 0 / (4-4)! = 0/ 1 = 0... from {2}, index 0 has 2   
Fourth number is 2

Giving us 3142. If you manually list out the permutations using DFS method, it would be 3142. Done! It really was all about pattern finding.

**public** **class** **Solution** {

**public** String getPermutation(**int** n, **int** k) {

**int** pos = 0;

List<Integer> numbers = **new** ArrayList<>(); //巧妙

**int**[] factorial = **new** **int**[n+1];

StringBuilder sb = **new** StringBuilder();

// create an array of factorial lookup

**int** sum = 1;

factorial[0] = 1;

**for**(**int** i=1; i<=n; i++){

sum \*= i;

factorial[i] = sum;

}

// factorial[] = {1, 1, 2, 6, 24, ... n!}

// create a list of numbers to get indices

**for**(**int** i=1; i<=n; i++){

numbers.add(i);

}

// numbers = {1, 2, 3, 4}

k--;

**for**(**int** i = 1; i <= n; i++){

**int** index = k/factorial[n-i];

sb.append(String.valueOf(numbers.get(index)));

numbers.remove(index);

k-=index\*factorial[n-i];

}

**return** String.valueOf(sb);

}

}

Recursion will use more memory, while this problem can be solved by iteration. I solved this problem before, but I didn't realize that using k = k-1 would avoid dealing with case k%(n-1)!==0. Rewrote this code, should be pretty concise now.

Only thing is that I have to use a list to store the remaining numbers, neither linkedlist nor arraylist are very efficient, anyone has a better idea?

The logic is as follows: for n numbers the permutations can be divided to (n-1)! groups, for n-1 numbers can be divided to (n-2)! groups, and so on. Thus k/(n-1)! indicates the index of current number, and k%(n-1)! denotes remaining index for the remaining n-1 numbers. We keep doing this until n reaches 0, then we get n numbers permutations that is kth.

**public** String getPermutation(**int** n, **int** k) {

List<Integer> num = **new** LinkedList<Integer>();

**for** (**int** i = 1; i <= n; i++) num.add(i);

**int**[] fact = **new** **int**[n]; // factorial

fact[0] = 1;

**for** (**int** i = 1; i < n; i++) fact[i] = i\*fact[i-1];

k = k-1;

StringBuilder sb = **new** StringBuilder();

**for** (**int** i = n; i > 0; i--){

**int** ind = k/fact[i-1];

k = k%fact[i-1];

sb.append(num.**get**(ind));

num.remove(ind);

}

**return** sb.toString();

}

### Hard

### First Missing Positive

[My Submissions](https://leetcode.com/problems/first-missing-positive/submissions/)

QuestionEditorial Solution

Total Accepted: **63852** Total Submissions: **267731** Difficulty: **Hard**

Given an unsorted integer array, find the first missing positive integer.

For example,  
Given [1,2,0] return 3,  
and [3,4,-1,1] return 2.

Your algorithm should run in *O*(*n*) time and uses constant space.

这个方法好

遍历数组中的每个数

不断变化位置i处的数字，目的是使其被放置在正确的位置上

A[i]应该放置元素i+1

即1,2,3,…是正确的序列

之后只需要检查一遍数组，看哪个位置元素不是预期的

Put each number in its right place.

For example:

When we find 5, then swap it with A[4].

At last, the first place where its number is not right, return the place + 1.

**class** **Solution**

{

**public**:

**int** firstMissingPositive(**int** A[], **int** n)

{

**for**(**int** i = 0; i < n; ++ i)

**while**(A[i] > 0 && A[i] <= n && A[A[i] - 1] != A[i])

swap(A[i], A[A[i] - 1]);

**for**(**int** i = 0; i < n; ++ i)

**if**(A[i] != i + 1)

**return** i + 1;

**return** n + 1;

}

};

The basic idea is **for any k positive numbers (duplicates allowed), the first missing positive number must be within [1,k+1]**. The reason is like you put k balls into k+1 bins, there must be a bin empty, the empty bin can be viewed as the missing number.

1. Unfortunately, there are 0 and negative numbers in the array, so firstly I think of using partition technique (used in quick sort) to put all positive numbers together in one side. This can be finished in O(n) time, O(1) space.
2. After partition step, you get all the positive numbers lying within A[0,k-1]. Now, According to the basic idea, I infer the first missing number must be within [1,k+1]. I decide to use A[i] (0<=i<=k-1) to indicate whether the number (i+1) exists. But here I still have to main the original information A[i] holds. Fortunately, A[i] are all positive numbers, so I can set them to negative to indicate the existence of (i+1) and I can still use abs(A[i]) to get the original information A[i] holds.
3. After step 2, I can again scan all elements between A[0,k-1] to find the first positive element A[i], that means (i+1) doesn't exist, which is what I want.

**public** **int** firstMissingPositive(**int**[] A) {

**int** n=A.length;

**if**(n==0)

**return** 1;

**int** k=partition(A)+1;

**int** temp=0;

**int** first\_missing\_Index=k;

**for**(**int** i=0;i<k;i++){

temp=Math.abs(A[i]);

**if**(temp<=k)

A[temp-1]=(A[temp-1]<0)?A[temp-1]:-A[temp-1];

}

**for**(**int** i=0;i<k;i++){

**if**(A[i]>0){

first\_missing\_Index=i;

**break**;

}

}

**return** first\_missing\_Index+1;

}

**public** **int** partition(**int**[] A){

**int** n=A.length;

**int** q=-1;

**for**(**int** i=0;i<n;i++){

**if**(A[i]>0){

q++;

swap(A,q,i);

}

}

**return** q;

}

**public** **void** swap(**int**[] A, **int** i, **int** j){

**if**(i!=j){

A[i]^=A[j];

A[j]^=A[i];

A[i]^=A[j];

}

}

The key here is to use swapping to keep constant space and also make use of the length of the array, which means there can be at most n positive integers. So each time we encounter an valid integer, find its correct position and swap. Otherwise we continue.

**public** **class** **Solution** {

**public** **int** firstMissingPositive(**int**[] A) {

**int** i = 0;

**while**(i < A.length){

**if**(A[i] == i+1 || A[i] <= 0 || A[i] > A.length) i++;

**else** **if**(A[A[i]-1] != A[i]) swap(A, i, A[i]-1);

**else** i++;

}

i = 0;

**while**(i < A.length && A[i] == i+1) i++;

**return** i+1;

}

**private** **void** swap(**int**[] A, **int** i, **int** j){

**int** temp = A[i];

A[i] = A[j];

A[j] = temp;

}

}

**class** Solution {

**public**:

**int** firstMissingPositive(**vector**<**int**>& nums) {

**for**(**int** i=0; i<nums.size(); i++){

**if**(i+1==nums[i]) **continue**;

**int** x = nums[i];

**while**(x>=1 && x<=nums.size() && x!=nums[x-1]){

swap(x, nums[x-1]);

}

}

**for**(**int** i=0; i<nums.size(); i++){

**if**(i+1!=nums[i]) **return** i+1;

}

**return** nums.size()+1;

}

};

Since we can not use extra space, so thinking about using the nums vector itself to record a positive number occurred.

The key here is to use swapping to keep constant space and also make use of the length of the array, which means there can be at most n positive integers. So each time we encounter an valid integer, find its correct position and swap. Otherwise we continue.

**public** **class** **Solution** {

**public** **int** firstMissingPositive(**int**[] A) {

**int** i = 0;

**while**(i < A.length){

**if**(A[i] == i+1 || A[i] <= 0 || A[i] > A.length) i++;

**else** **if**(A[A[i]-1] != A[i]) swap(A, i, A[i]-1);

**else** i++;

}

i = 0;

**while**(i < A.length && A[i] == i+1) i++;

**return** i+1;

}

**private** **void** swap(**int**[] A, **int** i, **int** j){

**int** temp = A[i];

A[i] = A[j];

A[j] = temp;

}

}

### Find the Duplicate Number

[My Submissions](https://leetcode.com/problems/find-the-duplicate-number/submissions/)

QuestionEditorial Solution

Total Accepted: **26751** Total Submissions: **68937** Difficulty: **Hard**

Given an array *nums* containing *n* + 1 integers where each integer is between 1 and *n* (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

**Note:**

1. You **must not** modify the array (assume the array is read only).
2. You must use only constant, *O*(1) extra space.
3. Your runtime complexity should be less than O(n2).
4. There is only one duplicate number in the array, but it could be repeated more than once.

这个写法好

关键是初值的设定

元素i，下一跳是在nums[i]处，这样形成链表。重复元素，下一跳会相遇在相同地方，即有重复元素，则会形成循环链表。

整数1~n（包括）直接，有一个重复元素，数组大小是n+1，则nums[n]存在，即，所有元素之间可以进行跳转。

The main idea is the same with problem ***Linked List Cycle II***,*https://leetcode.com/problems/linked-list-cycle-ii/*. Use two pointers the fast and the slow. The fast one goes forward two steps each time, while the slow one goes only step each time. They must meet the same item when slow==fast. In fact, they meet in a circle, the duplicate number must be the entry point of the circle when visiting the array from nums[0]. Next we just need to find the entry point. We use a point(we can use the fast one before) to visit form begining with one step each time, do the same job to slow. When fast==slow, they meet at the entry point of the circle. The easy understood code is as follows.

**int** findDuplicate3(**vector**<**int**>& nums)

{

**if** (nums.size() > 1)

{

**int** slow = nums[0];

**int** fast = nums[nums[0]];

**while** (slow != fast)

{

slow = nums[slow];

fast = nums[nums[fast]];

}

fast = 0;

**while** (fast != slow)

{

fast = nums[fast];

slow = nums[slow];

}

**return** slow;

}

**return** -1;

}

suppose the array is

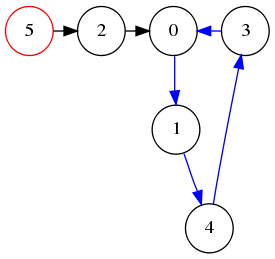
*index: 0 1 2 3 4 5*

*value: 2 5 1 1 4 3*

first subtract 1 from each element in the array, so it is much easy to understand. use the value as pointer. the array becomes:

*index: 0 1 2 3 4 5*

*value: 1 4 0 0 3 2*



Second if the array is

*index: 0 1 2 3 4 5*

*value: 0 1 2 4 2 3*

we must choose the last element as the head of the linked list. If we choose 0, we can not detect the cycle.

Now the problem is the same as find the cycle in linkedlist!

public **int** findDuplicate(**int**[] nums) {

**int** n = nums.**length**;

**for**(**int** i=0;i<nums.**length**;i++) nums[i]--;

**int** slow = n-1;

**int** fast = n-1;

**do**{

slow = nums[slow];

fast = nums[nums[fast]];

}**while**(slow != fast);

slow = n-1;

**while**(slow != fast){

slow = nums[slow];

fast = nums[fast];

}

**return** slow+1;

}

One condition is we cannot modify the array. So the solution is

**public** **int** findDuplicate(**int**[] nums) {

**int** n = nums.length;

**int** slow = n;

**int** fast = n;

**do**{

slow = nums[slow-1];

fast = nums[nums[fast-1]-1];

}**while**(slow != fast);

slow = n;

**while**(slow != fast){

slow = nums[slow-1];

fast = nums[fast-1];

}

**return** slow;

}

## 二分查找

### First Bad Version

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/first-bad-version/submissions/)

* Total Accepted: **66737**
* Total Submissions: **279879**
* Difficulty: **Easy**
* Contributors: **Admin**

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool isBadVersion(version) which will return whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

**The binary search code:**

**public** **int** **firstBadVersion**(**int** n) {

**int** start = 1, end = n;

**while** (start < end) {

**int** mid = start + (end-start) / 2;

**if** (!isBadVersion(mid)) start = mid + 1;//mid版是好的时

**else** end = mid;

}

**return** start;

}

### Search for a Range

[My Submissions](https://leetcode.com/problems/search-for-a-range/submissions/)

QuestionEditorial Solution

Total Accepted: **80546** Total Submissions: **276949** Difficulty: **Medium**

Given a sorted array of integers, find the starting and ending position of a given target value.

Your algorithm's runtime complexity must be in the order of *O*(log *n*).

If the target is not found in the array, return [-1, -1].

For example,  
Given [5, 7, 7, 8, 8, 10] and target value 8,  
return [3, 4].

The problem can be simply broken down as two binary searches for the begining and end of the range, respectively:

First let's find the left boundary of the range. We initialize the range to [i=0, j=n-1]. In each step, calculate the middle element [mid = (i+j)/2]. Now according to the relative value of A[mid] to target, there are three possibilities:

1. If A[mid] < target, then the range must begins on the ***right*** of mid (hence i = mid+1 for the next iteration)
2. If A[mid] > target, it means the range must begins on the ***left*** of mid (j = mid-1)
3. If A[mid] = target, then the range must begins ***on the left of or at*** mid (j= mid)

Since we would move the search range to the same side for case 2 and 3, we might as well merge them as one single case so that less code is needed:

2\*. If A[mid] >= target, j = mid;

Surprisingly, 1 and 2\* are the only logic you need to put in loop while (i < j). When the while loop terminates, the value of i/j is where the start of the range is. Why?

No matter what the sequence originally is, as we narrow down the search range, eventually we will be at a situation where there are only two elements in the search range. Suppose our target is 5, then we have only 7 possible cases:

case 1: [5 7] (A[i] = target < A[j])

case 2: [5 3] (A[i] = target > A[j])

case 3: [5 5] (A[i] = target = A[j])

case 4: [3 5] (A[j] = target > A[i])

case 5: [3 7] (A[i] < target < A[j])

case 6: [3 4] (A[i] < A[j] < target)

case 7: [6 7] (target < A[i] < A[j])

For case 1, 2 and 3, if we follow the above rule, since mid = i => A[mid] = target in these cases, then we would set j = mid. Now the loop terminates and i and j both point to the first 5.

For case 4, since A[mid] < target, then set i = mid+1. The loop terminates and both i and j point to 5.

For all other cases, by the time the loop terminates, A[i] is not equal to 5. So we can easily know 5 is not in the sequence if the comparison fails.

In conclusion, when the loop terminates, if A[i]==target, then i is the left boundary of the range; otherwise, just return -1;

For the right of the range, we can use a similar idea. Again we can come up with several rules:

1. If A[mid] > target, then the range must begins on the ***left*** of mid (j = mid-1)
2. If A[mid] < target, then the range must begins on the ***right*** of mid (hence i = mid+1 for the next iteration)
3. If A[mid] = target, then the range must begins ***on the right of or at*** mid (i= mid)

Again, we can merge condition 2 and 3 into:

2\* If A[mid] <**=** target, then i = mid;

However, the terminate condition on longer works this time. Consider the following case:

[5 7], target = 5

Now A[mid] = 5, then according to rule 2, we set i = mid. This practically does nothing because i is already equal to mid. As a result, the search range is not moved at all!

The solution is by using a small trick: instead of calculating mid as mid = (i+j)/2, we now do:

mid = (i+j)/2+1

Why does this trick work? When we use mid = (i+j)/2, the mid is rounded to the lowest integer. In other words, mid is always *biased* towards the left. This means we could have i == mid when j - i == mid, but we NEVER have j == mid. So in order to keep the search range moving, you must make sure the new i is set to something different than mid, otherwise we are at the risk that i gets stuck. But for the new j, it is okay if we set it to mid, since it was not equal to mid anyways. Our two rules in search of the left boundary happen to satisfy these requirements, so it works perfectly in that situation. Similarly, when we search for the right boundary, we must make sure i won't get stuck when we set the new i to i = mid. The easiest way to achieve this is by making mid *biased* to the right, i.e. mid = (i+j)/2+1.

All this reasoning boils down to the following simple code:

**vector**<**int**> searchRange(**int** A[], **int** n, **int** target) {

**int** i = 0, j = n - 1;

**vector**<**int**> ret(2, -1);

// Search for the left one

**while** (i < j)

{

**int** mid = (i + j) /2;

**if** (A[mid] < target) i = mid + 1;

**else** j = mid;

}

**if** (A[i]!=target) **return** ret;

**else** ret[0] = i;

// Search for the right one

j = n-1; // We don't have to set i to 0 the second time.

**while** (i < j)

{

**int** mid = (i + j) /2 + 1; // Make mid biased to the right

**if** (A[mid] > target) j = mid - 1;

**else** i = mid; // So that this won't make the search range stuck.

}

ret[1] = j;

**return** ret;

}

这个容易理解些

搜索两遍。

记住这个找第一个大于等于目标的数的方法

找到第一个这样的数然后，找大于等于目标加1数

A very simple Java solution, with only one binary search algorithm

**public** **class** **Solution** {

**public** **int**[] searchRange(**int**[] A, **int** target) {

**int** start = Solution.firstGreaterEqual(A, target);

**if** (start == A.length || A[start] != target) {

**return** **new** **int**[]{-1, -1};

}

**return** **new** **int**[]{start, Solution.firstGreaterEqual(A, target + 1) - 1};

}

//find the first number that is greater than or equal to target.

//could return A.length if target is greater than A[A.length-1].

//actually this is the same as lower\_bound in C++ STL.

**private** **static** **int** firstGreaterEqual(**int**[] A, **int** target) {

**int** low = 0, high = A.length;

**while** (low < high) {

**int** mid = low + ((high - low) >> 1);

//low <= mid < high

**if** (A[mid] < target) {

low = mid + 1;

} **else** {

//should not be mid-1 when A[mid]==target.

//could be mid even if A[mid]>target because mid<high.

high = mid;

}

}

**return** low;

}

}

basically it is the same idea with using separate binary search for left and right bounds. The good point here is the lower\_bound and the search for (target+1)

this is a trick of preventing possible overflow. Imagine a case where low = Integer.MAX\_VALUE - 10, high = Integer.MAX\_VALUE – 5

### Guess Number Higher or Lower

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/guess-number-higher-or-lower/submissions/)

* Total Accepted: **14162**
* Total Submissions: **45301**
* Difficulty: **Easy**

We are playing the Guess Game. The game is as follows:

I pick a number from **1** to ***n***. You have to guess which number I picked.

Every time you guess wrong, I'll tell you whether the number is higher or lower.

You call a pre-defined API guess(int num) which returns 3 possible results (-1, 1, or 0):

-1 : My number is lower

1 : My number is higher

0 : Congrats! You got it!

**Example:**

n = 10, I pick 6.

Return 6.

-1 : My number is lower  
1 : My number is higher  
0 : Congrats! You got it!

Here "My" means the number which is given for you to guess not the number you put into **guess(int num).**

**即guess(int num)返回1表示，num小于实际值**

Using binary search to find the smallest number that's not too small.

**def** **guessNumber**(self, n):

**class** **C**: \_\_getitem\_\_ = **lambda** \_, i: -guess(i)

**return** bisect.bisect(C(), -1, 1, n)

Alternatively, without using the library:

**def** **guessNumber**(self, n):

lo, hi = 1, n

**while** lo < hi:

mid = (lo + hi) / 2

**if** guess(mid) == 1:

lo = mid + 1

**else**:

hi = mid

**return** lo

Funny variation:

**def** **guessNumber**(self, n):

lo, hi = 1, n

**while** lo < hi:

mid = (lo + hi) / 2

lo, hi = ((mid, mid), (mid+1, hi), (lo, mid-1))[guess(mid)]

**return** lo

这个方法好，但是要注意mid的写法

This problem is a binary search problem has an O(logn) complexity.

**public** **int** **guessNumber**(**int** n) {

**int** i = 1, j = n;

**while**(i < j) {

**int** mid = i + (j - i) / 2;

**if**(guess(mid) == 0) {

**return** mid;

} **else** **if**(guess(mid) == 1) {

i = mid + 1;

} **else** {

j = mid;

}

}

**return** i;

}

for the testcase:

2126753390（2147483647最大int）  
1702766719

if I use the line "int mid = i + (j - i) / 2", then it can pass

if I replace it with "int mid = (j + i) / 2", which mathematically are the same it will return Time Limit Exceeded error. And this line works fine if the input number is not that big...

注意，重要

"i + (j - i) / 2" is faster than "(i + j) / 2" because the latter one could have integer overflow (becoming negative). that could result infinite loop...

With this test case, due to integer overflow (when j+i becomes a negative number), you get into an infinite loop. So the problem is not that computing "(j + i) / 2" is slower but that it leads to integer overflow. Here is the sequence of 'mids' that you get:

1063376695  
-552418605  
787167393  
-690523256  
718115067  
-725049419  
700851986  
-733680959  
696536216  
-735838844  
695457273  
-736378316  
695187537  
-736513184  
695120103  
-736546901  
695103245  
-736555330  
695099030  
-736557437  
695097977  
-736557964  
695097713  
-736558096  
695097647  
-736558129  
695097631  
-736558137  
695097627  
-736558139  
695097626  
-736558139  
695097626  
and so on...

### Find Peak Element

[My Submissions](https://leetcode.com/problems/find-peak-element/submissions/)

QuestionEditorial Solution

Total Accepted: **62931** Total Submissions: **190894** Difficulty: **Medium**

A peak element is an element that is greater than its neighbors.

Given an input array where num[i] ≠ num[i+1], find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that num[-1] = num[n] = -∞.

For example, in array [1, 2, 3, 1], 3 is a peak element and your function should return the index number 2.

[click to show spoilers.](https://leetcode.com/problems/find-peak-element/)

**Note:**

Your solution should be in logarithmic complexity.

Find the maximum by binary search (recursion and iteration)

Consider that each local maximum is one valid peak. My solution is to find one local maximum with binary search. Binary search satisfies the O(logn) computational complexity.

Binary Search: recursion

**class** Solution {

**public**:

**int** findPeakElement(**const** **vector**<**int**> &num) {

**return** Helper(num, 0, num.size()-1);

}

**int** Helper(**const** **vector**<**int**> &num, **int** low, **int** high)

{

**if**(low == high)

**return** low;

**else**

{

**int** mid1 = (low+high)/2;

**int** mid2 = mid1+1;

**if**(num[mid1] > num[mid2])

**return** Helper(num, low, mid1);

**else**

**return** Helper(num, mid2, high);

}

}

};

Binary Search: iteration

**class** Solution {

**public**:

**int** findPeakElement(**const** **vector**<**int**> &num)

{

**int** low = 0;

**int** high = num.size()-1;

**while**(low < high)

{

**int** mid1 = (low+high)/2;

**int** mid2 = mid1+1;

**if**(num[mid1] < num[mid2])

low = mid2;

**else**

high = mid1;

}

**return** low;

}

};

Sequential Search:

**class** Solution {

**public**:

**int** findPeakElement(**const** **vector**<**int**> &num) {

**for**(**int** i = 1; i < num.size(); i ++)

{

**if**(num[i] < num[i-1])

{// <

**return** i-1;

}

}

**return** num.size()-1;

}

};

This problem is similar to Local Minimum. And according to the given condition, num[i] != num[i+1], there must exist a O(logN) solution. So we use binary search for this problem.

* If num[i-1] < num[i] > num[i+1], then num[i] is peak
* If num[i-1] < num[i] < num[i+1], then num[i+1...n-1] must contains a peak
* If num[i-1] > num[i] > num[i+1], then num[0...i-1] must contains a peak
* If num[i-1] > num[i] < num[i+1], then both sides have peak (n is num.length)

Here is the code

**public** **int** findPeakElement(**int**[] num) {

**return** helper(num,0,num.length-1);

}

**public** **int** helper(**int**[] num,**int** start,**int** end){

**if**(start == end){

**return** start;

}**else** **if**(start+1 == end){

**if**(num[start] > num[end]) **return** start;

**return** end;

}**else**{

**int** m = (start+end)/2;

**if**(num[m] > num[m-1] && num[m] > num[m+1]){

**return** m;

}**else** **if**(num[m-1] > num[m] && num[m] > num[m+1]){

**return** helper(num,start,m-1);

}**else**{

**return** helper(num,m+1,end);

}

}

}

My Iterative code with similar idea.

**public** **class** **Solution** {

**public** **int** findPeakElement(**int**[] nums) {

**int** lo = 0, hi = nums.length-1;

**while**(lo < hi){

**if**(lo +1== hi)

**return** nums[lo] > nums[hi]? lo : hi;

**int** mid = lo + (hi - lo)/2;

**if**(nums[mid] > nums[mid-1] && nums[mid] > nums[mid+1])

**return** mid;

**else** **if**(nums[mid] > nums[mid-1] && nums[mid] < nums[mid+1])

lo = mid+1;

**else**

hi = mid-1;

}

**return** lo;

}

}

I find it useful to reason about binary search problems using invariants. While there are many solutions posted here, neither of them provide (in my opinion) a good explanation about why they work. I just spent some time thinking about this and I thought it might be a good idea to share my thoughts.

Assume we initialize left = 0, right = nums.length - 1. The invariant I'm using is the following:

**nums[left - 1] < nums[left] && nums[right] > nums[right + 1]**

That basically means that in the current interval we're looking, [left, right] the function started increasing to left and will eventually decrease at right. The behavior between [left, right] falls into the following 3 categories:

1) nums[left] > nums[left + 1]. From the invariant, nums[left - 1] < nums[left] => left is a peak

2) The function is increasing from left to right i.e. nums[left] < nums[left + 1] < .. < nums[right - 1] < nums[right]. From the invariant, nums[right] > nums[right + 1] => right is a peak

3) the function increases for a while and then decreases (in which case the point just before it starts decreasing is a peak) e.g. 2 5 6 3 (6 is the point in question)

As shown, if the invariant above holds, there is at least a peak between [left, right]. Now we need to show 2 things:

I) the invariant is initially true. Since left = 0 and right = nums.length - 1 initially and we know that nums[-1] = nums[nums.length] = -oo, this is obviously true

II) At every step of the loop the invariant gets reestablished. If we consider the code in the loop, we have mid = (left + right) / 2 and the following 2 cases:

a) nums[mid] < nums[mid + 1]. It turns out that the interval [mid + 1, right] respects the invariant (nums[mid] < nums[mid + 1] -> part of the cond + nums[right] > nums[right + 1] -> part of the invariant in the previous loop iteration)

b) nums[mid] > nums[mid + 1]. Similarly, [left, mid] respects the invariant (nums[left - 1] < nums[left] -> part of the invariant in the previous loop iteration and nums[mid] > nums[mid + 1] -> part of the cond)

As a result, the invariant gets reestablished and it will also hold when we exit the loop. In that case we have an interval of length 2 i.e. right = left + 1. If nums[left] > nums[right], using the invariant (nums[left - 1] < nums[left]), we get that left is a peak. Otherwise right is the peak (nums[left] < nums[right] and nums[right] < nums[right + 1] from the invariant).

**public** **int** findPeakElement(**int**[] nums) {

**int** N = nums.length;

**if** (N == 1) {

**return** 0;

}

**int** left = 0, right = N - 1;

**while** (right - left > 1) {

**int** mid = left + (right - left) / 2;

**if** (nums[mid] < nums[mid + 1]) {

left = mid + 1;

} **else** {

right = mid;

}

}

**return** (left == N - 1 || nums[left] > nums[left + 1]) ? left : right;

}

I hope this makes things clear despite the long explanation.

**class** Solution {

**public**:

**int** findPeakElement(**const** **vector**<**int**> &num) {

**int** low = 0, high = num.size() - 1;

**while** (low < high - 1) {

**int** mid = (low + high) / 2;

**if** (num[mid] > num[mid - 1] && num[mid] > num[mid + 1])

**return** mid;

**else** **if** (num[mid] > num[mid + 1])

high = mid - 1;

**else**

low = mid + 1;

}

**return** num[low] > num[high] ? low : high;

}

};

这个写法好，但是要注意是否+1，及是否=的细节

**public** **int** findPeakElement(**int**[] a) {

**int** low = 0, mid = 0, high = a.length - 1;

**while**(low < high) {

mid = low + (high-low)/2;

**if**(a[mid] < a[mid+1]) low = mid+1;

**else** high = mid;

}

**return** low;

}

Key point here is that, at each mid, if you move towards the direction of semi-peak (5,7,x - > move right), you'll end up at some peak.

注意题目中，num[-1] = num[n] = -∞，而且只有一个peak点，说明，数组中的最后两个数是下降的，数组最后的开始两个数是上升的。

### Search Insert Position

[My Submissions](https://leetcode.com/problems/search-insert-position/submissions/)

QuestionEditorial Solution

Total Accepted: **103264** Total Submissions: **276894** Difficulty: **Medium**

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You may assume no duplicates in the array.

Here are few examples.  
[1,3,5,6], 5 → 2  
[1,3,5,6], 2 → 1  
[1,3,5,6], 7 → 4  
[1,3,5,6], 0 → 0

我的做法，类似找到第一个比target大的index

public int searchInsert(int[] nums, int target) {

int low=0;

int high=nums.length;

while(low<high)

{

int mid=low+(high-low)/2;

if(nums[mid]<target)

{

low=mid+1;

}

else

{

high=mid;

}

}

return low;

}

这个方法也不错，原来常规解法就可以返回目标不存在时的插入index

**public** **int** searchInsert(**int**[] A, **int** target) {

**int** low = 0, high = A.length-1;

**while**(low<=high){

**int** mid = (low+high)/2; // 或者int mid=low+(high-low)/2;

**if**(A[mid] == target) **return** mid;

**else** **if**(A[mid] > target) high = mid-1;

**else** low = mid+1;

}

**return** low;

}

**C++ O(logn) Binary Search that handles duplicate**

If there are duplicate elements equal to target, my code will always return the one with smallest index.

**class** Solution {

**public**:

**int** searchInsert(**vector**<**int**>& nums, **int** target) {

**int** low = 0, high = nums.size()-1;

// Invariant（不变）: the desired index is between [low, high+1]

**while** (low <= high) {

**int** mid = low + (high-low)/2;

**if** (nums[mid] < target)

low = mid+1;

**else**

high = mid-1;

}

// (1) At this point, low > high. That is, low >= high+1

// (2) From the invariant, we know that the index is between [low, high+1], so low <= high+1. Follwing from (1), now we know low == high+1.

// (3) Following from (2), the index is between [low, high+1] = [low, low], which means that low is the desired index

// Therefore, we return low as the answer. You can also return high+1 as the result, since low == high+1

**return** low;

}

};

## 深搜

### Number of Islands

[My Submissions](https://leetcode.com/problems/number-of-islands/submissions/)

QuestionEditorial Solution

Total Accepted: **42005** Total Submissions: **153005** Difficulty: **Medium**

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:***

11110  
11010  
11000  
00000

Answer: 1

***Example 2:***

11000  
11000  
00100  
00011

Answer: 3

**public** **class** Solution {

**private** **int** n;

**private** **int** m;

**public** **int** numIslands(**char**[][] grid) {

**int** count = 0;

n = grid.length;

**if** (n == 0) **return** 0;

m = grid[0].length;

**for** (**int** i = 0; i < n; i++){

**for** (**int** j = 0; j < m; j++)

**if** (grid[i][j] == '1') {

DFSMarking(grid, i, j);

++count;

}

}

**return** count;

}

**private** **void** DFSMarking(**char**[][] grid, **int** i, **int** j) {

**if** (i < 0 || j < 0 || i >= n || j >= m || grid[i][j] != '1') **return**;

grid[i][j] = '0';

DFSMarking(grid, i + 1, j);

DFSMarking(grid, i - 1, j);

DFSMarking(grid, i, j + 1);

DFSMarking(grid, i, j - 1);

}

**public** **class** **NumberofIslands** {

**static** **int**[] dx = {-1,0,0,1};

**static** **int**[] dy = {0,1,-1,0};

**public** **static** **int** numIslands(**char**[][] grid) {

**if**(grid==**null** || grid.length==0) **return** 0;

**int** islands = 0;

**for**(**int** i=0;i<grid.length;i++) {

**for**(**int** j=0;j<grid[i].length;j++) {

**if**(grid[i][j]=='1') {

explore(grid,i,j);

islands++;

}

}

}

**return** islands;

}

**public** **static** **void** explore(**char**[][] grid, **int** i, **int** j) {

grid[i][j]='x';

**for**(**int** d=0;d<dx.length;d++) {

**if**(i+dy[d]<grid.length && i+dy[d]>=0 && j+dx[d]<grid[0].length && j+dx[d]>=0 && grid[i+dy[d]][j+dx[d]]=='1') {

explore(grid,i+dy[d],j+dx[d]);

}

}

}

}

The algorithm works as follow:

1. Scan each cell in the grid.
2. If the cell value is '1' explore that island.
3. Mark the explored island cells with 'x'.
4. Once finished exploring that island, increment islands counter.

The arrays dx[], dy[] store the possible moves from the current cell. Two land cells ['1'] are considered from the same island if they are horizontally or vertically adjacent (possible moves (-1,0),(0,1),(0,-1),(1,0)). Two '1' diagonally adjacent are not considered from the same island.

这个写法不错

遍历图中的每一个元素，是小岛，则计数，同时，基于这个点进行扩散标记（标记成海）

**public** **class** Solution {

**public** **int** numIslands(**char**[][] grid) {

**int** count = 0;

**for** (**int** i = 0; i < grid.length; i++) {

**for** (**int** j = 0; j < grid[i].length; j++) {

**if** (grid[i][j] == '1') {

count++;

clearRestOfLand(grid, i, j);

}

}

}

**return** count;

}

**private** **void** clearRestOfLand(**char**[][] grid, **int** i, **int** j) {

**if** (i < 0 || j < 0 || i >= grid.length || j >= grid[i].length || grid[i][j] == '0') **return**;

grid[i][j] = '0';

clearRestOfLand(grid, i+1, j);

clearRestOfLand(grid, i-1, j);

clearRestOfLand(grid, i, j+1);

clearRestOfLand(grid, i, j-1);

**return**;

}

}

### House Robber III

[My Submissions](https://leetcode.com/problems/house-robber-iii/submissions/)

QuestionEditorial Solution

Total Accepted: **6679** Total Submissions: **18058** Difficulty: **Medium**

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**.

**Step I -- Think naively**

At first glance, the problem exhibits the feature of "optimal substructure": if we want to "rob" maximum amount of money from current binary tree (rooted at "root"), we surely hope that we can do the same to its left and right subtrees.

So going along this line, let's define the function rob(root) which will return the maximum amount of money that we can rob for the binary tree rooted at "root"; the key now is to construct the solution to the original problem from solutions to its subproblems, i.e., how to get rob(root)from rob(root.left), rob(root.right), ... etc.

Apparently the analyses above suggest a recursive solution. And for recursion, it's always worthwhile to figure out the following two properties:

1. Termination condition: when do we know the answer to rob(root) without any calculation? Of course when the tree is empty -- we've got nothing to rob so the amount of money is zero.
2. Recurrence relation: i.e., how to get rob(root) from rob(root.left), rob(root.right), ... etc. From the point of view of the tree root, there are only two scenarios at the end: "root" is robbed or is not. If it is, due to the constraint that "we cannot rob any two directly-linked houses", the next level of subtrees that are available would be the four "grandchild-subtrees" (root.left.left, root.left.right, root.right.left, root.right.right). However if root is not robbed, the next level of available subtrees would just be the two "child-subtrees" (root.left, root.right). We only need to choose the scenario which yields the larger amount of money.

Here is the program for the ideas above:

**public** **int** rob(TreeNode root) {

**if** (root == **null**) {

**return** 0;

}

**int** val = 0;

**if** (root.left != **null**) {

val += rob(root.left.left) + rob(root.left.right);

}

**if** (root.right != **null**) {

val += rob(root.right.left) + rob(root.right.right);

}

**return** Math.max(val + root.val, rob(root.left) + rob(root.right));

}

However the solution runs very slow (1186 ms) and barely got accepted.

**Step II -- Think one step further**

In step I, we only considered the aspect of "optimal substructure", but think little about the possibilities of overlapping of the subproblems. For example, to obtain rob(root), we needrob(root.left), rob(root.right), rob(root.left.left), rob(root.left.right), rob(root.right.left), rob(root.right.right); but to get rob(root.left), we also needrob(root.left.left), rob(root.left.right), similarly for rob(root.right). The naive solution above computed these subproblems repeatedly, which resulted in bad time performance. Now if you recall the two conditions for dynamic programming: "**optimal substructure**" + "**overlapping of subproblems**", we actually have a DP problem. A naive way to implement DP here is to use a hash map to record the results for visited subtrees.

And here is the improved solution:

**public** **int** rob(TreeNode root) {

Map<TreeNode, Integer> **map** = **new** HashMap<>();

**return** robSub(root, **map**);

}

**private** **int** robSub(TreeNode root, Map<TreeNode, Integer> **map**) {

**if** (root == null) **return** 0;

**if** (**map**.containsKey(root)) **return** **map**.get(root);

**int** val = 0;

**if** (root.left != null) {

val += robSub(root.left.left, **map**) + robSub(root.left.right, **map**);

}

**if** (root.right != null) {

val += robSub(root.right.left, **map**) + robSub(root.right.right, **map**);

}

val = Math.max(val + root.val, robSub(root.left, **map**) + robSub(root.right, **map**));

**map**.put(root, val);

**return** val;

}

The runtime is sharply reduced to 9ms, at the expense of O(n) space cost (n is the total number of nodes; stack cost for recursion is not counted).

**Step III -- Think one step back**

In step I, we defined our problem as rob(root), which will yield the maximum amount of money that can be robbed of the binary tree rooted at "root". This leads to the DP problem summarized in step II.

Now let's take one step back and ask why do we have overlapping subproblems? If you trace all the way back to the beginning, you'll find the answer lies in the way how we have definedrob(root). As I mentioned, for each tree root, there are two scenarios: it is robbed or is not.rob(root) does not distinguish between these two cases, so "information is lost as the recursion goes deeper and deeper", which resulted in repeated subproblems.

If we were able to maintain the information about the two scenarios for each tree root, let's see how it plays out. Redefine rob(root) as a new function which will return an array of two elements, the first element of which denotes the maximum amount of money that can be robbed if "root" is **not robbed**, while the second element signifies the maximum amount of money robbed if root is **robbed**.

Let's relate rob(root) to rob(root.left) and rob(root.right), etc. For the 1st element ofrob(root), we only need to sum up the larger elements of rob(root.left) androb(root.right), respectively, since root is not robbed and we are free to rob the left and right subtrees. For the 2nd element of rob(root), however, we only need to add up the 1st elements of rob(root.left) and rob(root.right), respectively, plus the value robbed from "root" itself, since in this case it's guaranteed that we cannot rob the nodes of root.left and root.right.

As you can see, by keeping track of the information of both scenarios, we decoupled the subproblems and the solution essentially boiled down to a greedy one. Here is the program:

**public** **int** rob(TreeNode root) {

**int**[] res = robSub(root);

**return** Math.max(res[0], res[1]);

}

**private** **int**[] robSub(TreeNode root) {

**if** (root == **null**) {

**return** **new** **int**[2];

}

**int**[] left = robSub(root.left);

**int**[] right = robSub(root.right);

**int**[] res = **new** **int**[2];

res[0] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);

res[1] = root.val + left[0] + right[0];

**return** res;

}

dfs all the nodes of the tree, each node return two number, int[] num, num[0] is the max value while rob this node, num[1] is max value while not rob this value. Current node return value only depend on its children's value. Transform function should be very easy to understand.

**public** **class** **Solution** {

**public** **int** rob(TreeNode root) {

**int**[] num = dfs(root);

**return** Math.max(num[0], num[1]);

}

**private** **int**[] dfs(TreeNode x) {

**if** (x == **null**) **return** **new** **int**[2];

**int**[] left = dfs(x.left);

**int**[] right = dfs(x.right);

**int**[] res = **new** **int**[2];

res[0] = left[1] + right[1] + x.val;

res[1] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);

**return** res;

}

}

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

**class** **Solution** {

**public**:

**int** tryRob(TreeNode\* root, **int**& l, **int**& r) {

**if** (!root)

**return** 0;

**int** ll = 0, lr = 0, rl = 0, rr = 0;

l = tryRob(root->left, ll, lr);

r = tryRob(root->right, rl, rr);

**return** max(root->val + ll + lr + rl + rr, l + r);

}

**int** rob(TreeNode\* root) {

**int** l, r;

**return** tryRob(root, l, r);

}

};

Basically you want to compare which one is bigger between 1) you + sum of your grandchildren and 2) sum of your children. Personally I like my solution better than the most voted solution because I don't need complex data structures like map.

这个解释比较清晰

Let

f1(node) be the value of maximum money we can rob from the subtree with node as root ( we can rob node if necessary).

f2(node) be the value of maximum money we can rob from the subtree with node as root but without robbing node.

Then we have

f2(node) = f1(node.left) + f1(node.right) and

f1(node) = max( f2(node.left)+f2(node.right)+node.value, f2(node) ).

JAVA

**public** **class** **Solution** {

**public** **int** rob(TreeNode root) {

**return** robDFS(root)[1];

}

**int**[] robDFS(TreeNode node){

**int** [] res = **new** **int**[2];

**if**(node==**null**) **return** res;

**int** [] l = robDFS(node.left);

**int** [] r = robDFS(node.right);

res[0] = l[1] + r[1];

res[1] = Math.max(res[0], l[0] + r[0] + node.val);

**return** res;

}

}

**这个写法不错—>超时？**

robInclude 抢；robExclude该点不抢（不同于前面的状态表示，前面f1(node)表示可以抢也可以不抢）

public class Solution {

**public** **int** rob(TreeNode root) {

**if** (root == **null**) **return** 0;

**return** Math.max(robInclude(root), robExclude(root));

}

**public** **int** robInclude(TreeNode node) {

**if**(node == **null**) **return** 0;

**return** robExclude(node.left) + robExclude(node.right) + node.val;

}

**public** **int** robExclude(TreeNode node) {

**if**(node == **null**) **return** 0;

**return** rob(node.left) + rob(node.right);// //其下两个节点随意，可以包含也可以不包含（关键）

}

}

### Course Schedule

[My Submissions](https://leetcode.com/problems/course-schedule/submissions/)

QuestionEditorial Solution

Total Accepted: **36411** Total Submissions: **136566** Difficulty: **Medium**

There are a total of *n* courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

For example:

2, [[1,0]]

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

2, [[1,0],[0,1]]

There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

**Note:**  
The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about [how a graph is represented](https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs).

[click to show more hints.](https://leetcode.com/problems/course-schedule/)

**Hints:**

1. This problem is equivalent to finding if a cycle exists in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. [Topological Sort via DFS](https://class.coursera.org/algo-003/lecture/52) - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via [BFS](http://en.wikipedia.org/wiki/Topological_sorting#Algorithms).

下面这个写法比较清晰

首先，计算节点的入度。将入度为0的节点入队（因为所有入度不为0的节点不可能是第一个节点）

然后，删去以该点为尾的边，并修改相应点的入度

如果图中有环，则最后会剩下点

**public** boolean canFinish(**int** numCourses, **int**[][] prerequisites) {

**int**[][] matrix = **new** **int**[numCourses][numCourses]; // i -> j

**int**[] indegree = **new** **int**[numCourses];

**for** (**int** i=0; i<prerequisites.length; i++) {

**int** ready = prerequisites[i][0];

**int** pre = prerequisites[i][1];//注意这里的顺序

**if** (matrix[pre][ready] == 0)

indegree[ready]++; //duplicate case

matrix[pre][ready] = 1;//注意这里，不包含在if语句中。用于处理重复输入，构建图。

}

**int** count = 0;

Queue<Integer> **queue** = **new** LinkedList();

**for** (**int** i=0; i<indegree.length; i++) {

**if** (indegree[i] == 0) **queue**.offer(i);

}

**while** (!**queue**.isEmpty()) {

**int** course = **queue**.poll();

count++;

**for** (**int** i=0; i<numCourses; i++) {

**if** (matrix[course][i] != 0) {

**if** (--indegree[i] == 0)

**queue**.offer(i);

}

}

}

**return** count == numCourses;

}

I rewrote the code, I think that would be O(V + E)

// O(V + E)

List<Integer>[] matrix = **new** List[numCourses];

**int**[] indegree = **new** **int**[numCourses];

// E part

**for** (**int**[] pre : prerequisites) {

**int** preCourse = pre[1];

**int** readyCourse = pre[0];

List<Integer> **list** = matrix[preCourse];

**if** (**list** == null) {

**list** = **new** LinkedList<>();

matrix[preCourse] = **list**;

}

**list**.add(readyCourse);

indegree[readyCourse]++;

}

Queue<Integer> **queue** = **new** LinkedList<>();

**for** (**int** i=0; i<numCourses; i++) {

**if** (indegree[i] == 0) **queue**.offer(i);

}

**int** count = 0;

// V part

**while** (!**queue**.isEmpty()) {

**int** vertex = **queue**.poll();

count++;

List<Integer> adjacent = matrix[vertex];

**if** (adjacent == null) **continue**;

**for** (**int** neighbor : adjacent) {

indegree[neighbor]--;

**if** (indegree[neighbor] == 0)

**queue**.offer(neighbor);

}

}

**return** count == numCourses;

According to my code test, BFS is much faster than DFS. From my perspective DFS searches more branches. EX: 1->3->4 //1->5->3 the first branch we need search 3's children, in second we still need to do so.

BFS:

**public** **class** Solution {

**public** boolean canFinish(**int** numCourses, **int**[][] prerequisites) {

ArrayList[] graph = **new** ArrayList[numCourses];

**int**[] degree = **new** **int**[numCourses];

Queue **queue** = **new** LinkedList();

**int** count=0;

**for**(**int** i=0;i<numCourses;i++)

graph[i] = **new** ArrayList();

**for**(**int** i=0; i<prerequisites.length;i++){

degree[prerequisites[i][1]]++;

graph[prerequisites[i][0]].add(prerequisites[i][1]);

}

**for**(**int** i=0; i<degree.length;i++){

**if**(degree[i] == 0){

**queue**.add(i);

count++;

}

}

**while**(**queue**.size() != 0){

**int** course = (**int**)**queue**.poll();

**for**(**int** i=0; i<graph[course].size();i++){

**int** pointer = (**int**)graph[course].get(i);

degree[pointer]--;

**if**(degree[pointer] == 0){

**queue**.add(pointer);

count++;

}

}

}

**if**(count == numCourses)

**return** **true**;

**else**

**return** **false**;

}

}

DFS:

**public** **class** **Solution** {

**public** **boolean** canFinish(**int** numCourses, **int**[][] prerequisites) {

ArrayList[] graph = **new** ArrayList[numCourses];

**for**(**int** i=0;i<numCourses;i++)

graph[i] = **new** ArrayList();

**boolean**[] visited = **new** **boolean**[numCourses];

**for**(**int** i=0; i<prerequisites.length;i++){

graph[prerequisites[i][1]].add(prerequisites[i][0]);

}

**for**(**int** i=0; i<numCourses; i++){

**if**(!dfs(graph,visited,i))

**return** **false**;

}

**return** **true**;

}

**private** **boolean** dfs(ArrayList[] graph, **boolean**[] visited, **int** course){

**if**(visited[course])

**return** **false**;

**else**

visited[course] = **true**;;

**for**(**int** i=0; i<graph[course].size();i++){

**if**(!dfs(graph,visited,(**int**)graph[course].get(i)))

**return** **false**;

}

visited[course] = **false**;

**return** **true**;

}

}

### Course Schedule II

[My Submissions](https://leetcode.com/problems/course-schedule-ii/submissions/)

QuestionEditorial Solution

Total Accepted: **24786** Total Submissions: **118464** Difficulty: **Medium**

There are a total of *n* courses you have to take, labeled from 0 to n - 1.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]

Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

For example:

2, [[1,0]]

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1]

4, [[1,0],[2,0],[3,1],[3,2]]

There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is [0,1,2,3]. Another correct ordering is[0,2,1,3].

**Note:**  
The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about [how a graph is represented](https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs).

[click to show more hints.](https://leetcode.com/problems/course-schedule-ii/)

**Hints:**

1. This problem is equivalent to finding the topological order in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. [Topological Sort via DFS](https://class.coursera.org/algo-003/lecture/52) - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via [BFS](http://en.wikipedia.org/wiki/Topological_sorting#Algorithms).

我的做法，和上一题做法一致，用数组存储图，记录中间结果

[**ArrayList**](file:///E:\docsJava\api\java\util\ArrayList.html#ArrayList-int-)(int initialCapacity)

Constructs an empty list with the specified initial capacity.

This question asks for an order in which prerequisite courses must be taken first. This prerequisite relationship reminds one of directed graphs. Then, the problem reduces to find a topological sort order of the courses, which would be a DAG if it has a valid order.

**public** **int**[] findOrder(**int** numCourses, **int**[][] prerequisites) {

**int**[] incLinkCounts = **new** **int**[numCourses];

List<List<Integer>> adjs = **new** ArrayList<>(numCourses);

initialiseGraph(incLinkCounts, adjs, prerequisites);

//return solveByBFS(incLinkCounts, adjs);

**return** solveByDFS(adjs);

}

The first step is to transform it into a directed graph. Since it is likely to be sparse,we use adjacency list graph data structure. 1 -> 2 means 1 must be taken before 2.

**private** **void** initialiseGraph(**int**[] incLinkCounts, List<List<Integer>> adjs, **int**[][] prerequisites){

**int** n = incLinkCounts.length;

**while** (n-- > 0) adjs.add(**new** ArrayList<>());

**for** (**int**[] edge : prerequisites) {

incLinkCounts[edge[0]]++;//该边对应的入度加1

adjs.**get**(edge[1]).add(edge[0]);

}

}

How can we obtain a topological sort order of a DAG?

We observe that if a node has incoming edges, it has prerequisites. Therefore, the first few in the order must be those with no prerequisites, i.e. no incoming edges. Any non-empty DAG must have at least one node without incoming links. You can draw a small graph to convince yourself. If we visit these few and remove all edges attached to them, we are left with a smaller DAG, which is the same problem. This will then give our BFS solution.

**private** **int**[] solveByBFS(**int**[] incLinkCounts, List<List<Integer>> adjs){

**int**[] order = **new** **int**[incLinkCounts.length];

Queue<Integer> toVisit = **new** ArrayDeque<>();

**for** (**int** i = 0; i < incLinkCounts.length; i++) {

**if** (incLinkCounts[i] == 0) toVisit.offer(i);

}

**int** visited = 0;

**while** (!toVisit.isEmpty()) {

**int** **from** = toVisit.poll();

order[visited++] = **from**;

**for** (**int** to : adjs.**get**(**from**)) {

incLinkCounts[to]--;

**if** (incLinkCounts[to] == 0) toVisit.offer(to);

}

}

**return** visited == incLinkCounts.length ? order : **new** **int**[0];

}

Another way to think about it is the last few in the order must be those which are not prerequisites of other courses. Thinking it recursively means if one node has unvisited child node, you should visit them first before you put this node down in the final order array. This sounds like the post-order of a DFS. Since we are putting nodes down in the reverse order, we should reverse it back to correct ordering or use a stack.

**private** **int**[] solveByDFS(List<List<Integer>> adjs) {

BitSet hasCycle = **new** BitSet(1);

BitSet visited = **new** BitSet(adjs.size());

BitSet onStack = **new** BitSet(adjs.size());

Deque<Integer> order = **new** ArrayDeque<>();

**for** (**int** i = adjs.size() - 1; i >= 0; i--) {

**if** (visited.**get**(i) == **false** && hasOrder(i, adjs, visited, onStack, order) == **false**) **return** **new** **int**[0];

}

**int**[] orderArray = **new** **int**[adjs.size()];

**for** (**int** i = 0; !order.isEmpty(); i++) orderArray[i] = order.pop();

**return** orderArray;

}

**private** boolean hasOrder(**int** **from**, List<List<Integer>> adjs, BitSet visited, BitSet onStack, Deque<Integer> order) {

visited.**set**(**from**);

onStack.**set**(**from**);

**for** (**int** to : adjs.**get**(**from**)) {

**if** (visited.**get**(to) == **false**) {

**if** (hasOrder(to, adjs, visited, onStack, order) == **false**) **return** **false**;

} **else** **if** (onStack.**get**(to) == **true**) {

**return** **false**;

}

}

onStack.clear(**from**);

order.push(**from**);

**return** **true**;

}

**public** **class** **Solution** {

**public** **int**[] findOrder(**int** numCourses, **int**[][] prerequisites) {

List<List<Integer>> adj = **new** ArrayList<>(numCourses);

**for** (**int** i = 0; i < numCourses; i++) adj.add(i, **new** ArrayList<>());

**for** (**int** i = 0; i < prerequisites.length; i++) adj.get(prerequisites[i][1]).add(prerequisites[i][0]);

**boolean**[] visited = **new** **boolean**[numCourses];

Stack<Integer> stack = **new** Stack<>();

**for** (**int** i = 0; i < numCourses; i++) {

**if** (!topologicalSort(adj, i, stack, visited, **new** **boolean**[numCourses])) **return** **new** **int**[0];

}

**int** i = 0;

**int**[] result = **new** **int**[numCourses];

**while** (!stack.isEmpty()) {

result[i++] = stack.pop();

}

**return** result;

}

**private** **boolean** topologicalSort(List<List<Integer>> adj, **int** v, Stack<Integer> stack, **boolean**[] visited, **boolean**[] isLoop) {

**if** (visited[v]) **return** **true**;

**if** (isLoop[v]) **return** **false**;

isLoop[v] = **true**;

**for** (Integer u : adj.get(v)) {

**if** (!topologicalSort(adj, u, stack, visited, isLoop)) **return** **false**;

}

visited[v] = **true**;

stack.push(v);

**return** **true**;

}

}

### Reconstruct Itinerary

[My Submissions](https://leetcode.com/problems/reconstruct-itinerary/submissions/)

QuestionEditorial Solution

Total Accepted: **7363** Total Submissions: **31334** Difficulty: **Medium**

Given a list of airline tickets represented by pairs of departure and arrival airports [from, to], reconstruct the itinerary in order. All of the tickets belong to a man who departs from JFK. Thus, the itinerary must begin with JFK.

**Note:**

1. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary ["JFK", "LGA"] has a smaller lexical order than ["JFK", "LGB"].
2. All airports are represented by three capital letters (IATA code).
3. You may assume all tickets form at least one valid itinerary.

**Example 1:**  
tickets = [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]  
Return ["JFK", "MUC", "LHR", "SFO", "SJC"].

**Example 2:**  
tickets = [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]  
Return ["JFK","ATL","JFK","SFO","ATL","SFO"].  
Another possible reconstruction is ["JFK","SFO","ATL","JFK","ATL","SFO"]. But it is larger in lexical order.

Just Eulerian path. Greedy DFS, building the route backwards when retreating.

More explanation and example under the codes.

Iterative versions inspired by [fangyang](https://leetcode.com/discuss/84706/share-solution-java-greedy-stack-15ms-with-explanation) (I had only thought of recursion, d'oh).

**Java**

**用这个写法**

public List<**String**> findItinerary(String[][] tickets) {

for (String[] ticket : tickets)

targets.computeIfAbsent(ticket[0], k -> new PriorityQueue()).add(ticket[1]);

visit("JFK");

return route;

}

Map<**String,** PriorityQueue<String>> targets = new HashMap<>();

List<**String**> route = new LinkedList();

void visit(String airport) {

while(targets.containsKey(airport) && !targets.get(airport).isEmpty())

visit(targets.get(airport).poll());

route.add(0, airport);

}

Iterative version:

**public** List<String> findItinerary(String[][] tickets) {

Map<String, PriorityQueue<String>> targets = **new** HashMap<>();

**for** (String[] ticket : tickets)

targets.computeIfAbsent(ticket[0], k -> **new** PriorityQueue()).add(ticket[1]);

List<String> route = **new** LinkedList();

Stack<String> **stack** = **new** Stack<>();

**stack**.push("JFK");

**while** (!**stack**.empty()) {

**while** (targets.containsKey(**stack**.peek()) && !targets.get(**stack**.peek()).isEmpty())

**stack**.push(targets.get(**stack**.peek()).poll());

route.add(0, **stack**.pop());

}

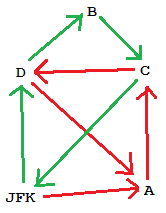
**return** route;

}

**Explanation**

First keep going forward until you get stuck. That's a good main path already. Remaining tickets form cycles which are found on the way back and get merged into that main path. By writing down the path backwards when retreating from recursion, merging the cycles into the main path is easy - the end part of the path has already been written, the start part of the path hasn't been written yet, so just write down the cycle now and then keep backwards-writing the path.

Example:



From JFK we first visit JFK -> A -> C -> D -> A. There we're stuck, so we write down A as the end of the route and retreat back to D. There we see the unused ticket to B and follow it: D -> B -> C -> JFK -> D. Then we're stuck again, retreat and write down the airports while doing so: Write down D before B, then JFK before D, etc. When we're back from our cycle at D, the written route is D -> B -> C -> JFK -> D -> A. Then we retreat further along the original path, prepending C, A and finally JFK to the route, ending up with the route JFK -> A -> C -> D -> B -> C -> JFK -> D -> A.

Noticed some folks are using Hierholzer's algorithm to find a Eulerian path.

My solution is similar, considering this passenger has to be physically in one place before move to another airport, we are considering using up all tickets and choose lexicographically smaller solution if in tie as two constraints.

Thinking as that passenger, the passenger choose his/her flight greedy as the lexicographical order, once he/she figures out go to an airport without departure with more tickets at hand. the passenger will push current ticket in a stack and look at whether it is possible for him/her to travel to other places from the airport on his/her way.

Please let me know if you have any suggestions.

public List<**String**> findItinerary(String[][] tickets) {

List<**String**> ans = new ArrayList<**String**>();

if(tickets == null || tickets.length == 0) return ans;

Map<**String,** PriorityQueue<String>> ticketsMap = new HashMap<>();

for(int i = 0; i < tickets.length; i++) {

if(!ticketsMap.containsKey(tickets[i][0])) ticketsMap.put(tickets[i][0], new PriorityQueue<String>());

ticketsMap.get(tickets[i][0]).add(tickets[i][1]);

}

String curr = "JFK";

Stack<**String**> drawBack = new Stack<**String**>();

for(int i = 0; i < tickets.length; i++) {

while(!ticketsMap.containsKey(curr) || ticketsMap.get(curr).isEmpty()) {

drawBack.push(curr);

curr = ans.remove(ans.size()-1);

}

ans.add(curr);

curr = ticketsMap.get(curr).poll();

}

ans.add(curr);

while(!drawBack.isEmpty()) ans.add(drawBack.pop());

return ans;

}

**public** **class** Solution {

**public** List<String> findItinerary(String[][] tickets) {

ArrayList<String> result = **new** ArrayList<String>();

**if**(tickets == null || tickets.length == 0){

**return** result;

}

**int** total = tickets.length + 1;

HashMap<String, ArrayList<String>> **map** = **new** HashMap<String, ArrayList<String>>();

**for**(**int** i = 0; i < tickets.length; i++){

**if**(**map**.containsKey(tickets[i][0])){

ArrayList<String> tmp = **map**.get(tickets[i][0]);

listAdd(tickets[i][1], tmp);

}

**else**{

ArrayList<String> tmp = **new** ArrayList<String>();

tmp.add(tickets[i][1]);

**map**.put(tickets[i][0], tmp);

}

}

result.add("JFK");

itineraryHelper("JFK", **map**, result, total, 1);

**return** result;

}

**public** boolean itineraryHelper(String current, HashMap<String, ArrayList<String>> **map**, ArrayList<String> result, **int** total, **int** num){

**if**(num >= total){

**return** **true**;

}

**if**(!**map**.containsKey(current) || **map**.get(current).size() == 0){

**return** **false**;

}

ArrayList<String> curList = **map**.get(current);

**int** i = 0;

**while**(i < curList.size()){

String next = curList.remove(i);

result.add(next);

**if**(itineraryHelper(next, **map**, result, total, num + 1)){

**return** **true**;

}

result.remove(result.size() - 1);

listAdd(next, curList);

i++;

}

**return** **false**;

}

**public** **void** listAdd(String value, ArrayList<String> **list**){

**if**(**list**.size() == 0){

**list**.add(value);

**return**;

}

**else**{

**int** i = 0;

**while**(i < **list**.size()){

**if**(value.compareTo(**list**.get(i)) <= 0){

**list**.add(i, value);

**return**;

}

i++;

}

**list**.add(value);

**return**;

}

}

}

解法II 欧拉通路（Eulerian path）：参考链接：https://leetcode.com/discuss/84659/short-ruby-python-java-c

将机场视为顶点，机票视为看做有向边，可以构成一个有向图。

通过图（无向图或有向图）中所有边且每边仅通过一次的通路称为欧拉通路，相应的回路称为欧拉回路。具有欧拉回路的图称为欧拉图（Euler Graph），具有欧拉通路而无欧拉回路的图称为半欧拉图。

因此题目的实质就是从JFK顶点出发寻找欧拉通路，可以利用Hierholzer算法。

### Clone Graph

[My Submissions](https://leetcode.com/problems/clone-graph/submissions/)

QuestionEditorial Solution

Total Accepted: **63961** Total Submissions: **257443** Difficulty: **Medium**

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

**OJ's undirected graph serialization:**

Nodes are labeled uniquely.

We use # as a separator for each node, and , as a separator for node label and each neighbor of the node.

As an example, consider the serialized graph {0,1,2#1,2#2,2}.

The graph has a total of three nodes, and therefore contains three parts as separated by #.

1. First node is labeled as 0. Connect node 0 to both nodes 1 and 2.
2. Second node is labeled as 1. Connect node 1 to node 2.
3. Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle.

Visually, the graph looks like the following:

1

/ \

/ \

0 --- 2

/ \

\\_/

**注意要使用map，同时因为每个数字编号是唯一的。不适用缓存直接做，会导致陷入死循环，堆栈溢出**

**public** **class** **Solution** {

**private** HashMap<Integer, UndirectedGraphNode> map = **new** HashMap<>();

**public** UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

**return** **clone**(node);

}

**private** UndirectedGraphNode **clone**(UndirectedGraphNode node) {

**if** (node == **null**) **return** **null**;

**if** (map.containsKey(node.label)) {

**return** map.get(node.label);

}

UndirectedGraphNode **clone** = **new** UndirectedGraphNode(node.label);

map.put(**clone**.label, **clone**);//需要注意这个的顺序

**for** (UndirectedGraphNode neighbor : node.neighbors) {

**clone**.neighbors.add(**clone**(neighbor));

}

**return** **clone**;

}

}

## 宽搜

### Surrounded Regions

[My Submissions](https://leetcode.com/problems/surrounded-regions/submissions/)

QuestionEditorial Solution

Total Accepted: **51178** Total Submissions: **319242** Difficulty: **Medium**

Given a 2D board containing 'X' and '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

* First, check the four border of the matrix. If there is a element is 'O', alter it and all its neighbor 'O' elements to '1'.
* Then ,alter all the 'O' to 'X'
* At last,alter all the '1' to 'O'

For example:

X X X X X X X X X X X X

X X O X -> X X O X -> X X X X

X O X X X 1 X X X O X X

X O X X X 1 X X X O X X

**class** Solution {

**public**:

**void** solve(**vector**<**vector**<**char**>>& board) {

**int** i,j;

**int** row=board.size();

**if**(!row)

**return**;

**int** col=board[0].size();

**for**(i=0;i<row;i++){

check(board,i,0,row,col);

**if**(col>1)

check(board,i,col-1,row,col);

}

**for**(j=1;j+1<col;j++){

check(board,0,j,row,col);

**if**(row>1)

check(board,row-1,j,row,col);

}

**for**(i=0;i<row;i++)

**for**(j=0;j<col;j++)

**if**(board[i][j]=='O')

board[i][j]='X';

**for**(i=0;i<row;i++)

**for**(j=0;j<col;j++)

**if**(board[i][j]=='1')

board[i][j]='O';

}

**void** check(**vector**<**vector**<**char**> >&vec,**int** i,**int** j,**int** row,**int** col){

**if**(vec[i][j]=='O'){

vec[i][j]='1';

**if**(i>1)

check(vec,i-1,j,row,col);

**if**(j>1)

check(vec,i,j-1,row,col);

**if**(i+1<row)

check(vec,i+1,j,row,col);

**if**(j+1<col)

check(vec,i,j+1,row,col);

}

}

};

The algorithm is quite simple: Use BFS starting from 'O's on the boundary and mark them as 'B', then iterate over the whole board and mark 'O' as 'X' and 'B' as 'O'.

**void** bfsBoundary(**vector**<**vector**<**char**> >& board, **int** w, **int** l)

{

**int** width = board.size();

**int** length = board[0].size();

**deque**<pair<**int**, **int**> > q;

q.push\_back(make\_pair(w, l));

board[w][l] = 'B';

**while** (!q.empty()) {

pair<**int**, **int**> cur = q.front();

q.pop\_front();

pair<**int**, **int**> adjs[4] = {{cur.first-1, cur.second},

{cur.first+1, cur.second},

{cur.first, cur.second-1},

{cur.first, cur.second+1}};

**for** (**int** i = 0; i < 4; ++i)

{

**int** adjW = adjs[i].first;

**int** adjL = adjs[i].second;

**if** ((adjW >= 0) && (adjW < width) && (adjL >= 0)

&& (adjL < length)

&& (board[adjW][adjL] == 'O')) {

q.push\_back(make\_pair(adjW, adjL));

board[adjW][adjL] = 'B';

}

}

}

}

**void** solve(**vector**<**vector**<**char**> > &board) {

**int** width = board.size();

**if** (width == 0) //Add this to prevent run-time error!

**return**;

**int** length = board[0].size();

**if** (length == 0) // Add this to prevent run-time error!

**return**;

**for** (**int** i = 0; i < length; ++i)

{

**if** (board[0][i] == 'O')

bfsBoundary(board, 0, i);

**if** (board[width-1][i] == 'O')

bfsBoundary(board, width-1, i);

}

**for** (**int** i = 0; i < width; ++i)

{

**if** (board[i][0] == 'O')

bfsBoundary(board, i, 0);

**if** (board[i][length-1] == 'O')

bfsBoundary(board, i, length-1);

}

**for** (**int** i = 0; i < width; ++i)

{

**for** (**int** j = 0; j < length; ++j)

{

**if** (board[i][j] == 'O')

board[i][j] = 'X';

**else** **if** (board[i][j] == 'B')

board[i][j] = 'O';

}

}

}

Note that one of the test cases is when the board is empty. So if you don't check it in your code, you will encounter an run-time error.

Java DFS + boundary cell turning solution, simple and clean code, commented.

这个思路和第一种做法一样。

从四周开始判断有无‘O’，如果有，则一定无法被X包围，所以先对其标记为\*

再将剩下的无法延伸到的O标记为X

**public** **void** solve(**char**[][] board) {

**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 '\*';

//如果这个点（i，j）是O，则取判断这个点的上下左右是否为O，如果是则深搜进行标记

**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);

}

### Minimum Height Trees

[My Submissions](https://leetcode.com/problems/minimum-height-trees/submissions/)

QuestionEditorial Solution

Total Accepted: **10357** Total Submissions: **38868** Difficulty: **Medium**

For a undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.

**Format**  
The graph contains n nodes which are labeled from 0 to n - 1. You will be given the number n and a list of undirected edges (each edge is a pair of labels).

You can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0] and thus will not appear together in edges.

**Example 1:**

Given n = 4, edges = [[1, 0], [1, 2], [1, 3]]

0

|

1

/ \

2 3

return [1]

**Example 2:**

Given n = 6, edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]

0 1 2

\ | /

3

|

4

|

5

return [3, 4]

**Hint:**

1. How many MHTs can a graph have at most?

**Note**:

(1) According to the [definition of tree on Wikipedia](https://en.wikipedia.org/wiki/Tree_(graph_theory)): “a tree is an undirected graph in which any two vertices are connected by *exactly* one path. In other words, any connected graph without simple cycles is a tree.”

(2) The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

**See**[**here for a better view**](http://algobox.org/minimum-height-trees/)

First let's review some statement for tree in graph theory:

*(1) A tree is an undirected graph in which any two vertices are connected by exactly one path.*

*(2) Any connected graph who has n nodes with n-1 edges is a tree.*

*(3) The degree of a vertex of a graph is the number of edges incident to the vertex.*

*(4) A leaf is a vertex of degree 1. An internal vertex is a vertex of degree at least 2.*

*(5) A path graph is a tree with two or more vertices that is not branched at all.*

*(6) A tree is called a rooted tree if one vertex has been designated the root.*

*(7) The height of a rooted tree is the number of edges on the longest downward path between root and a leaf.*

OK. Let's stop here and look at our problem.

Our problem want us to find the minimum height trees and return their root labels. First we can think about a simple case -- a path graph.

For a path graph of n nodes, find the minimum height trees is trivial. Just designate the middle point(s) as roots.

Despite its triviality, let design a algorithm to find them.

Suppose we don't know n, nor do we have random access of the nodes. We have to traversal. It is very easy to get the idea of two pointers. One from each end and move at the same speed. When they meet or they are one step away, (depends on the parity of n), we have the roots we want.

This gives us a lot of useful ideas to crack our real problem.

For a tree we can do some thing similar. We start from every end, by end we mean vertex of degree 1 (aka leaves). We let the pointers move the same speed. When two pointers meet, we keep only one of them, until the last two pointers meet or one step away we then find the roots.

It is easy to see that the last two pointers are from the two ends of the longest path in the graph.

The actual implementation is similar to the BFS topological sort. Remove the leaves, update the degrees of inner vertexes. Then remove the new leaves. Doing so level by level until there are 2 or 1 nodes left. What's left is our answer!

The time complexity and space complexity are both O(n).

Note that for a tree we always have V = n, E = n-1.

**Java**

|  |
| --- |
| static <T> [**List**](file:///E:\docsJava\api\java\util\List.html)<T> |
| [singletonList](file:///E:\docsJava\api\java\util\Collections.html#singletonList-T-)(T o)  Returns an immutable list containing only the specified object.  **Set**  boolean  [**remove**](file:///E:\docsJava\api\java\util\Set.html#remove-java.lang.Object-)([**Object**](file:///E:\docsJava\api\java\lang\Object.html) o)  Removes the specified element from this set if it is present (optional operation). |

**List**

E remove(int index)

Removes the element at the specified position in this list (optional operation).

boolean remove(Object o)

Removes the first occurrence of the specified element from this list, if it is present (optional operation).

Interator主要用于遍历Collection集合中的元素

**public** List<Integer> findMinHeightTrees(**int** n, **int**[][] edges) {

**if** (n == 1) **return** Collections.singletonList(0);

List<Set<Integer>> adj = **new** ArrayList<>(n);

**for** (**int** i = 0; i < n; ++i) adj.add(**new** HashSet<>());

**for** (**int**[] edge : edges) {

adj.**get**(edge[0]).add(edge[1]);

adj.**get**(edge[1]).add(edge[0]);

}

List<Integer> leaves = **new** ArrayList<>();

**for** (**int** i = 0; i < n; ++i)

**if** (adj.**get**(i).size() == 1) leaves.add(i);

**while** (n > 2) {

n -= leaves.size();

List<Integer> newLeaves = **new** ArrayList<>();//这个分层的写法也要注意

**for** (**int** i : leaves) {

**int** j = adj.**get**(i).iterator().next();

adj.**get**(j).remove(i);

// int j=list.get(i).get(0); 如果用List不用Set，可以这样处理

//list.get(j).remove((Integer)i);//注意这里，如果不是Integer类型则会应用到移除i索引处的元素

**if** (adj.**get**(j).size() == 1) newLeaves.add(j);

}

leaves = newLeaves;

}

**return** leaves;

}

// Runtime: 53 ms

I am sharing two of my solutions, one is based on the longest path, and the other is related to Tree DP.

**Longest Path**

It is easy to see that the root of an MHT has to be the middle point (or two middle points) of the longest path of the tree. Though multiple longest paths can appear in an unrooted tree, they must share the same middle point(s).

Computing the longest path of a unrooted tree can be done, in O(n) time, by tree dp, or simply 2 tree traversals (dfs or bfs). The following is some thought of the latter.

Randomly select a node x as the root, do a dfs/bfs to find the node y that has the longest distance from x. Then y must be one of the endpoints on some longest path. Let y the new root, and do another dfs/bfs. Find the node z that has the longest distance from y.

Now, the path from y to z is the longest one, and thus its middle point(s) is the answer. [Java Solution](https://github.com/lydxlx1/LeetCode/blob/master/src/_310.java)

**Tree DP**

Alternatively, one can solve this problem directly by tree dp. Let dp[i] be the height of the tree when the tree root is i. We compute dp[0] ... dp[n - 1] by tree dp in a dfs manner.

Arbitrarily pick a node, say node 0, as the root, and do a dfs. When we reach a node u, and let T be the subtree by removing all u's descendant (see the right figure below). We maintain a variable acc that keeps track of the length of the longest path in T with one endpoint being u. Then dp[u] = max(height[u], acc) Note, acc is 0 for the root of the tree.

| |

. .

/|\ /|\

\* u \* \* u \*

/|\

/ | \

\* v \*

. denotes a single node, and \* denotes a subtree (possibly empty).

Now it remains to calculate the new acc for any of u's child, v. It is easy to see that the new acc is the max of the following

1. acc + 1 --- extend the previous path by edge uv;
2. max(height[v'] + 2), where v != v' --- see below for an example.
3. u
4. /|
5. / |
6. v' v
7. |
8. .
9. .
10. .
11. |
12. .

In fact, the second case can be computed in O(1) time instead of spending a time proportional to the degree of u. Otherwise, the runtime can be quadratic when the degree of some node is Omega(n). The trick here is to maintain two heights of each node, the largest height (the conventional height), and the second largest height (the height of the node after removing the branch w.r.t. the largest height).

Therefore, after the dfs, all dp[i]'s are computed, and the problem can be answered trivially. The total runtime is still O(n). [Java Solution](https://github.com/lydxlx1/LeetCode/blob/master/src/_310_1.java)

### Word Ladder II

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/word-ladder-ii/submissions/)

* Total Accepted: **49311**
* Total Submissions: **358978**
* Difficulty: **Hard**

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to*endWord*, such that:

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the word list

For example,

Given:  
*beginWord* = "hit"  
*endWord* = "cog"  
*wordList* = ["hot","dot","dog","lot","log"]

Return

[

["hit","hot","dot","dog","cog"],

["hit","hot","lot","log","cog"]

]

**Note:**

* All words have the same length.
* All words contain only lowercase alphabetic characters.

[Subscribe](https://leetcode.com/subscribe/) to see which companies asked this question

Hide Tags

[Array](https://leetcode.com/tag/array/) [Backtracking](https://leetcode.com/tag/backtracking/) [Breadth-first Search](https://leetcode.com/tag/breadth-first-search/) [String](https://leetcode.com/tag/string/)

窗体顶端

窗体底端

The solution contains two steps 1 Use BFS to construct a graph. 2. Use DFS to construct the paths from end to start.Both solutions got AC within 1s.

The first step BFS is quite important. I summarized three tricks

1) Using a **MAP** to store the min ladder of each word, or use a **SET** to store the words visited in current ladder, when the current ladder was completed, delete the visited words from unvisited. That's why I have two similar solutions.

2) Use **Character iteration** to find all possible paths. Do not compare one word to all the other words and check if they only differ by one character.

3) One word is allowed to be inserted into the queue only **ONCE**. See my comments.

这个写法不错

1，重点是ladder存放单词到int的map。

每一轮，step++；

当出现与end相同的new\_word时，记录min=step。

下一轮当step>min时结束整个循环

2，用map存放单词到list的映射。用来回溯序列。

3，整体采用宽搜，使用队列。

public class Solution {

Map<String,List<String>> map;

List<List<String>> results;

public List<List<String>> findLadders(String start, String end, Set<String> dict) {

results= new ArrayList<List<String>>();

if (dict.size() == 0)

return results;

int min=Integer.MAX\_VALUE;

Queue<String> queue= new ArrayDeque<String>();

queue.add(start);

map = new HashMap<String,List<String>>();

Map<String,Integer> ladder = new HashMap<String,Integer>();

for (String string:dict)

ladder.put(string, Integer.MAX\_VALUE);

ladder.put(start, 0);

dict.add(end);

//BFS: Dijisktra search

while (!queue.isEmpty()) {

String word = queue.poll();

int step = ladder.get(word)+1;//'step' indicates how many steps are needed to travel to one word.

if (step>min) break;

for (int i = 0; i < word.length(); i++){//单词每次只改变一位

StringBuilder builder = new StringBuilder(word);

for (char ch='a'; ch <= 'z'; ch++){

builder.setCharAt(i,ch);

String new\_word=builder.toString();

if (ladder.containsKey(new\_word)) {

if (step>ladder.get(new\_word))//Check if it is the shortest path to one word. 如果曾经访问过这个词，则跳过

continue;

else if (step<ladder.get(new\_word)){

queue.add(new\_word);

ladder.put(new\_word, step);

}else;// It is a KEY line. If one word already appeared in one ladder,

// Do not insert the same word inside the queue twice. Otherwise it gets TLE.

if (map.containsKey(new\_word)) //Build adjacent Graph

map.get(new\_word).add(word);

else{

List<String> list= new LinkedList<String>();

list.add(word);

map.put(new\_word,list);

//It is possible to write three lines in one:

//map.put(new\_word,new LinkedList<String>(Arrays.asList(new String[]{word})));

//Which one is better?

}

if (new\_word.equals(end))

min=step;

}//End if dict contains new\_word

}//End:Iteration from 'a' to 'z'

}//End:Iteration from the first to the last

}//End While

//BackTracking

LinkedList<String> result = new LinkedList<String>();

backTrace(end,start,result);

return results;

}

private void backTrace(String word,String start,List<String> list){

if (word.equals(start)){

list.add(0,start);

results.add(new ArrayList<String>(list));

list.remove(0);

return;

}

list.add(0,word);

if (map.get(word)!=null)

for (String s:map.get(word))

backTrace(s,start,list);

list.remove(0);

}

}

Another solution using two sets. This is similar to the answer in the most viewed thread. While I found my solution more readable and efficient.

public class Solution {

List<List<String>> results;

List<String> list;

Map<String,List<String>> map;

public List<List<String>> findLadders(String start, String end, Set<String> dict) {

results= new ArrayList<List<String>>();

if (dict.size() == 0)

return results;

int curr=1,next=0;

boolean found=false;

list = new LinkedList<String>();

map = new HashMap<String,List<String>>();

Queue<String> queue= new ArrayDeque<String>();

Set<String> unvisited = new HashSet<String>(dict);

Set<String> visited = new HashSet<String>();

queue.add(start);

unvisited.add(end);

unvisited.remove(start);

//BFS

while (!queue.isEmpty()) {

String word = queue.poll();

curr--;

for (int i = 0; i < word.length(); i++){

StringBuilder builder = new StringBuilder(word);

for (char ch='a'; ch <= 'z'; ch++){

builder.setCharAt(i,ch);

String new\_word=builder.toString();

if (unvisited.contains(new\_word)){

//Handle queue

if (visited.add(new\_word)){//Key statement,Avoid Duplicate queue insertion

next++;

queue.add(new\_word);

}

if (map.containsKey(new\_word))//Build Adjacent Graph

map.get(new\_word).add(word);

else{

List<String> l= new LinkedList<String>();

l.add(word);

map.put(new\_word, l);

}

if (new\_word.equals(end)&&!found) found=true;

}

}//End:Iteration from 'a' to 'z'

}//End:Iteration from the first to the last

if (curr==0){

if (found) break;

curr=next;

next=0;

unvisited.removeAll(visited);

visited.clear();

}

}//End While

backTrace(end,start);

return results;

}

private void backTrace(String word,String start){

if (word.equals(start)){

list.add(0,start);

results.add(new ArrayList<String>(list));

list.remove(0);

return;

}

list.add(0,word);

if (map.get(word)!=null)

for (String s:map.get(word))

backTrace(s,start);

list.remove(0);

}

}

For your first solution, in the backTrace function, how do you make sure the paths being put in result are shortest paths?

It is a Directed Graph, all the path starts from the'end' and ends at'start'. All of them are the shortest, otherwise it wll not be added to the graph.

if dict.size()== 0 return an empty list.

Nice answer. But for your first solution can you explain me how the ‘end’ is added to map as a key for latter backtrace? Since the ladder in initialized before the end is added to the dict, In the nested for loops, you check if the new\_word is in ladder and then add it to the map as key. There should be no way for the end to be added because it is not even in the ladder.

I know it work but I just couldn’t figure it out.

Hi, nice solution, but seems that line

dict.add(end);

should be before the for loop, otherwise the graph doesn't contain end at all. It will pass OJ through, probably because in all test cases dict already contains end

Can't agree with you more! It seems like that the OJ doesn't include sample test case.

Hi, thank you for sharing, but I have a little double about your first solution. In your first solution, you use a map : String -> List of Strings to store the shortest path from start to the current word. But what if there are multiple shortest paths out there? Because you use the map to backtrack the final results, I'm afraid you may miss some of the paths due to the reason that you only record one of the shortest paths for every word?

Hi, it seems to me that the map does not really store the shortest path from start to the current word. Instead, as suggested by the code comment, the map stores an adjacency list (to represent a directed graph). Then the directed graph is used to construct the shortest paths.

## 设计

### Implement Queue using Stacks

[My Submissions](https://leetcode.com/problems/implement-queue-using-stacks/submissions/)

QuestionEditorial Solution

Total Accepted: **40159** Total Submissions: **118336** Difficulty: **Easy**

Implement the following operations of a queue using stacks.

* push(x) -- Push element x to the back of queue.
* pop() -- Removes the element from in front of queue.
* peek() -- Get the front element.
* empty() -- Return whether the queue is empty.

**Notes:**

* You must use *only* standard operations of a stack -- which means only push to top, peek/pop from top, size, and is empty operations are valid.
* Depending on your language, stack may not be supported natively. You may simulate a stack by using a list or deque (double-ended queue), as long as you use only standard operations of a stack.
* You may assume that all operations are valid (for example, no pop or peek operations will be called on an empty queue).

书

class MyQueue {

LinkedList<Integer> stack1=new LinkedList<Integer>();

LinkedList<Integer> stack2=new LinkedList<Integer>();

// Push element x to the back of queue.

public void push(int x) {

stack1.push(x);

}

// Removes the element from in front of queue.

public void pop() {

if(stack2.isEmpty())

{

shift();

}

stack2.pop();

}

public void shift()

{

while(!stack1.isEmpty())

{

stack2.push(stack1.pop());

}

}

// Get the front element.

public int peek() {

if(stack2.isEmpty())

{

shift();

}

return stack2.peek();

}

// Return whether the queue is empty.

public boolean empty() {

return stack1.isEmpty() && stack2.isEmpty();

}

}

**Java**

**class** **MyQueue** {

Stack<Integer> input = **new** Stack();

Stack<Integer> output = **new** Stack();

**public** **void** push(**int** x) {

input.push(x);

}

**public** **void** pop() {

peek();

output.pop();

}

**public** **int** peek() {

**if** (output.empty())

**while** (!input.empty())

output.push(input.pop());

**return** output.peek();

}

**public** **boolean** empty() {

**return** input.empty() && output.empty();

}

}

### Implement Stack using Queues

[My Submissions](https://leetcode.com/problems/implement-stack-using-queues/submissions/)

QuestionEditorial Solution

Total Accepted: **36651** Total Submissions: **119904** Difficulty: **Easy**

Implement the following operations of a stack using queues.

* push(x) -- Push element x onto stack.
* pop() -- Removes the element on top of the stack.
* top() -- Get the top element.
* empty() -- Return whether the stack is empty.

**Notes:**

* You must use *only* standard operations of a queue -- which means only push to back, peek/pop from front, size, and is empty operations are valid.
* Depending on your language, queue may not be supported natively. You may simulate a queue by using a list or deque (double-ended queue), as long as you use only standard operations of a queue.
* You may assume that all operations are valid (for example, no pop or top operations will be called on an empty stack).

**Update (2015-06-11):**  
The class name of the **Java** function had been updated to **MyStack** instead of Stack.

每次放入元素时，将队列中的元素依次pop并入队（除新加入元素外）

**class** Stack {

**public**:

**queue**<**int**> que;

// Push element x onto stack.

**void** push(**int** x) {

que.push(x);

**for**(**int** i=0;i<que.size()-1;++i){

que.push(que.front());

que.pop();

}

}

// Removes the element on top of the stack.

**void** pop() {

que.pop();

}

// Get the top element.

**int** top() {

**return** que.front();

}

// Return whether the stack is empty.

**bool** empty() {

**return** que.empty();

}

};

**Only push is O(n), others are O(1). Using one queue. Combination of two shared solutions**

**class** MyStack

{

Queue<Integer> **queue**;

**public** MyStack()

{

**this**.**queue**=**new** LinkedList<Integer>();

}

// Push element x onto stack.

**public** **void** push(**int** x)

{

**queue**.add(x);

**for**(**int** i=0;i<**queue**.size()-1;i++)

{

**queue**.add(**queue**.poll());

}

}

// Removes the element on top of the stack.

**public** **void** pop()

{

**queue**.poll();

}

// Get the top element.

**public** **int** top()

{

**return** **queue**.peek();

}

// Return whether the stack is empty.

**public** boolean empty()

{

**return** **queue**.isEmpty();

}

}

Then, two queue ways.

1 Push method is inefficient.

保持一个队列为空，push时，将新元素放入空队列，再将另一个队列中的元素push进这个队列。

when you push an element, choose one empty queue(whichever when both are empty) to add this element, and then push all elements of the other queue into the chosen queue. After that, the newest element is at the head of the chosen queue so that whenever you want to do pop() or top(), you always get the newest element.

For example,

push(1):

[ , ,1] [ , , ]

push(2):

[ , , ] [ ,1,2]

push(3):

[ 1, 2,3 ] [ , , ]

**class** **MyStack** {

//using two queue. The push is inefficient.

**private** Queue<Integer> q1 = **new** LinkedList<Integer>();

**private** Queue<Integer> q2 = **new** LinkedList<Integer>();

**public** **void** push(**int** x) {

**if**(q1.isEmpty()) {

q1.add(x);

**for**(**int** i = 0; i < q2.size(); i ++)

q1.add(q2.poll());

}**else** {

q2.add(x);

**for**(**int** i = 0; i < q1.size(); i++)

q2.add(q1.poll());

}

}

**public** **void** pop() {

**if**(!q1.isEmpty())

q1.poll();

**else**

q2.poll();

}

**public** **int** top() {

**return** q1.isEmpty() ? q2.peek() : q1.peek();

}

**public** **boolean** empty() {

**return** q1.isEmpty() && q2.isEmpty();

}

}

2 pop() and top() are inefficient

始终保持一个队列是空的

When you push elements, choose a queue which is not empty(whichever when both are empty). When you do pop() or top(), first pop all elements of the queue except the tail into another empty queue, and then pop the tail which is your want.

For example:

push(1):

[ , , 1] [ , , ]

push(2):

[ ,2,1] [ , , ]

top();

[ , , 2] [ , ,1] -> [ , , ] [ ,2,1]

pop():

[ , , 1] [ , ,2] -> [ , ,1] [ , , ]

push(3) :

[ ,3,1] [ , , ]

**class** **MyStack**{

//using two queue. The pop and top are inefficient.

**private** Queue<Integer> q1 = **new** LinkedList<Integer>();

**private** Queue<Integer> q2 = **new** LinkedList<Integer>();

**public** **void** push(**int** x) {

**if**(!q1.isEmpty())

q1.add(x);

**else**

q2.add(x);

}

**public** **void** pop() {

**if**(q1.isEmpty()) {

**int** size = q2.size();

**for**(**int** i = 1; i < size; i ++) {

q1.add(q2.poll());

}

q2.poll();

}**else** {

**int** size = q1.size();

**for**(**int** i = 1; i < size; i ++) {

q2.add(q1.poll());

}

q1.poll();

}

}

**public** **int** top() {

**int** res;

**if**(q1.isEmpty()) {

**int** size = q2.size();

**for**(**int** i = 1; i < size; i ++) {

q1.add(q2.poll());

}

res = q2.poll();

q1.add(res);

}**else** {

**int** size = q1.size();

**for**(**int** i = 1; i < size; i ++) {

q2.add(q1.poll());

}

res = q1.poll();

q2.add(res);

}

**return** res;

}

**public** **boolean** empty() {

**return** q1.isEmpty() && q2.isEmpty();

}

}

class MyStack {

Queue<Integer> q1=new LinkedList<Integer>();

Queue<Integer> q2=new LinkedList<Integer>();

// Push element x onto stack.

public void push(int x) {

if(q1.isEmpty())

{

q1.add(x);

while(!q2.isEmpty())

{

q1.add(q2.poll());

}

}

else

{

q2.add(x);

while(!q1.isEmpty())

{

q2.add(q1.poll());

}

}

}

// Removes the element on top of the stack.

public void pop() {

if(!q1.isEmpty()) q1.poll();

if(!q2.isEmpty()) q2.poll();

}

// Get the top element.

public int top() {

if(!q1.isEmpty()) return q1.peek();

if(!q2.isEmpty()) return q2.peek();

return Integer.MAX\_VALUE;

}

// Return whether the stack is empty.

public boolean empty() {

return q1.isEmpty()&&q2.isEmpty();

}

}

### Binary Search Tree Iterator

[My Submissions](https://leetcode.com/problems/binary-search-tree-iterator/submissions/)

QuestionEditorial Solution

Total Accepted: **46053** Total Submissions: **134033** Difficulty: **Medium**

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling next() will return the next smallest number in the BST.

**Note:**next() and hasNext() should run in average O(1) time and uses O(*h*) memory, where *h* is the height of the tree.

我的做法，就是基于中序遍历，将遍历结果存在list中。（但是不符合空间复杂度要求）

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class BSTIterator {

LinkedList<TreeNode> stack=new LinkedList<TreeNode>();

LinkedList<Integer> res=new LinkedList<>();

public void inorder(TreeNode root)

{

TreeNode p=root;

while(!stack.isEmpty() || p!=null)

{

if(p!=null)

{

stack.push(p);

p=p.left;

}

else

{

p=stack.pop();

res.add(p.val);

p=p.right;

}

}

}

public BSTIterator(TreeNode root) {

inorder(root);

}

/\*\* @return whether we have a next smallest number \*/

public boolean hasNext() {

return !res.isEmpty();

}

/\*\* @return the next smallest number \*/

public int next() {

return res.remove(0);

}

}

/\*\*

\* Your BSTIterator will be called like this:

\* BSTIterator i = new BSTIterator(root);

\* while (i.hasNext()) v[f()] = i.next();

\*/

下面这个方法好

I use Stack to store directed left children from root. When next() be called, I just pop one element and process its right child as new root. The code is pretty straightforward.

So this can satisfy O(h) memory, hasNext() in O(1) time, But next() is O(h) time.

I can't find a solution that can satisfy both next() in O(1) time, space in O(h).

Java:

**public** **class** **BSTIterator** {

**private** Stack<TreeNode> stack = **new** Stack<TreeNode>();

**public** BSTIterator(TreeNode root) {

pushAll(root);

}

/\*\* **@return** whether we have a next smallest number \*/

**public** **boolean** hasNext() {

**return** !stack.isEmpty();

}

/\*\* **@return** the next smallest number \*/

**public** **int** next() {//方法不错

TreeNode tmpNode = stack.pop();

pushAll(tmpNode.right);

**return** tmpNode.val;

}

**private** **void** pushAll(TreeNode node) {

**for** (; node != **null**; stack.push(node), node = node.left);

}

}

My idea comes from this: My first thought was to use inorder traversal to put every node into an array, and then make an index pointer for the next() and hasNext(). That meets the O(1) run time but not the O(h) memory. O(h) is really much more less than O(n) when the tree is huge.

This means I cannot use a lot of memory, which suggests that I need to make use of the tree structure itself. And also, one thing to notice is the "average O(1) run time". It's weird to say average O(1), because there's nothing below O(1) in run time, which suggests in most cases, I solve it in O(1), while in some cases, I need to solve it in O(n) or O(h). These two limitations are big hints.

Before I come up with this solution, I really draw a lot binary trees and try inorder traversal on them. We all know that, once you get to a TreeNode, in order to get the smallest, you need to go all the way down its left branch. So our first step is to point to pointer to the left most TreeNode. The problem is how to do back trace. Since the TreeNode doesn't have father pointer, we cannot get a TreeNode's father node in O(1) without store it beforehand. Back to the first step, when we are traversal to the left most TreeNode, we store each TreeNode we met ( They are all father nodes for back trace).

After that, I try an example, for next(), I directly return where the pointer pointing at, which should be the left most TreeNode I previously found. What to do next? After returning the smallest TreeNode, I need to point the pointer to the next smallest TreeNode. When the current TreeNode has a right branch (It cannot have left branch, remember we traversal to the left most), we need to jump to its right child first and then traversal to its right child's left most TreeNode. When the current TreeNode doesn't have a right branch, it means there cannot be a node with value smaller than itself father node, point the pointer at its father node.

The overall thinking leads to the structure Stack, which fits my requirement so well.

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

**public** **class** **BSTIterator** {

**private** Stack<TreeNode> stack;

**public** BSTIterator(TreeNode root) {

stack = **new** Stack<>();

TreeNode cur = root;

**while**(cur != **null**){

stack.push(cur);

**if**(cur.left != **null**)

cur = cur.left;

**else**

**break**;

}

}

/\*\* **@return** whether we have a next smallest number \*/

**public** **boolean** hasNext() {

**return** !stack.isEmpty();

}

/\*\* **@return** the next smallest number \*/

**public** **int** next() {

TreeNode node = stack.pop();

TreeNode cur = node;

// traversal right branch

**if**(cur.right != **null**){

cur = cur.right;

**while**(cur != **null**){

stack.push(cur);

**if**(cur.left != **null**)

cur = cur.left;

**else**

**break**;

}

}

**return** node.val;

}

}

/\*\*

\* Your BSTIterator will be called like this:

\* BSTIterator i = new BSTIterator(root);

\* while (i.hasNext()) v[f()] = i.next();

\*/

### Peeking Iterator

[My Submissions](https://leetcode.com/problems/peeking-iterator/submissions/)

QuestionEditorial Solution

Total Accepted: **18369** Total Submissions: **54977** Difficulty: **Medium**

Given an Iterator class interface with methods: next() and hasNext(), design and implement a PeekingIterator that support the peek()operation -- it essentially peek() at the element that will be returned by the next call to next().

Here is an example. Assume that the iterator is initialized to the beginning of the list: [1, 2, 3].

Call next() gets you 1, the first element in the list.

Now you call peek() and it returns 2, the next element. Calling next() after that ***still*** return 2.

You call next() the final time and it returns 3, the last element. Calling hasNext() after that should return false.

**Hint:**

1. Think of "looking ahead". You want to cache the next element.
2. Is one variable sufficient? Why or why not?
3. Test your design with call order of peek() before next() vs next() before peek().
4. For a clean implementation, check out [Google's guava library source code](https://github.com/google/guava/blob/703ef758b8621cfbab16814f01ddcc5324bdea33/guava-gwt/src-super/com/google/common/collect/super/com/google/common/collect/Iterators.java#L1125).

**Follow up**: How would you extend your design to be generic and work with all types, not just integer?

关键是next初值的设定

**class** **PeekingIterator** **implements** **Iterator**<**Integer**> {

**private** Integer next = **null**;//存放指向的下一个元素

**private** Iterator<Integer> iter;

**public** PeekingIterator(Iterator<Integer> iterator) {

// initialize any member here.

iter = iterator;

**if** (iter.hasNext())

next = iter.next();

}

// Returns the next element in the iteration without advancing the iterator.

**public** Integer peek() {

**return** next;

}

// hasNext() and next() should behave the same as in the Iterator interface.

// Override them if needed.

@Override

**public** Integer next() {

Integer res = next;

next = iter.hasNext() ? iter.next() : **null**;

**return** res;

}

@Override

**public** **boolean** hasNext() {

**return** next != **null**;

}

}

cache the next element. If next is null, there is no more elements in iterator.

### Insert Delete GetRandom O(1)

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/insert-delete-getrandom-o1/submissions/)

* Total Accepted: **1255**
* Total Submissions: **3671**
* Difficulty: **Medium**

Design a data structure that supports all following operations in *average* **O(1)** time.

1. insert(val): Inserts an item val to the set if not already present.
2. remove(val): Removes an item val from the set if present.
3. getRandom: Returns a random element from current set of elements. Each element must have the**same probability** of being returned.

**Example:**

// Init an empty set.

RandomizedSet randomSet = new RandomizedSet();

// Inserts 1 to the set. Returns true as 1 was inserted successfully.

randomSet.insert(1);

// Returns false as 2 does not exist in the set.

randomSet.remove(2);

// Inserts 2 to the set, returns true. Set now contains [1,2].

randomSet.insert(2);

// getRandom should return either 1 or 2 randomly.

randomSet.getRandom();

// Removes 1 from the set, returns true. Set now contains [2].

randomSet.remove(1);

// 2 was already in the set, so return false.

randomSet.insert(2);

// Since 1 is the only number in the set, getRandom always return 1.

randomSet.getRandom();

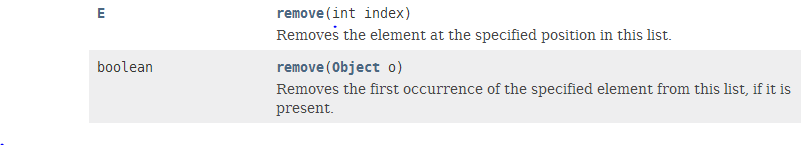
[Subscribe](https://leetcode.com/subscribe/) to see which companies asked this question

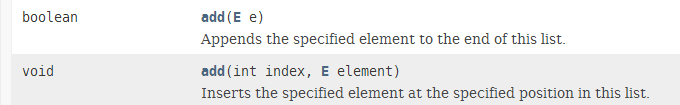
Hide Tags

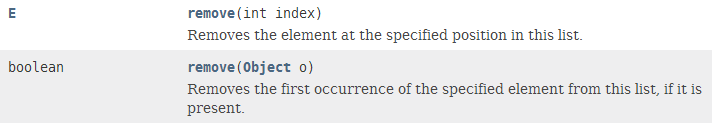
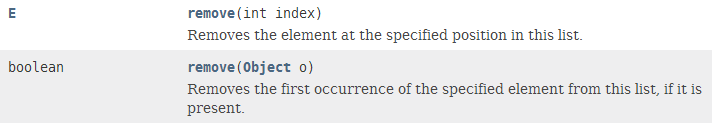
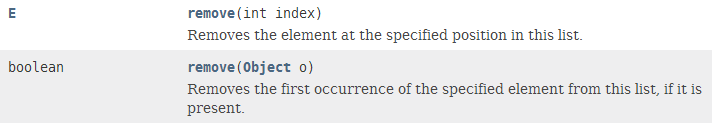
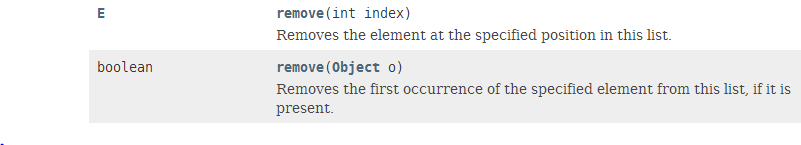
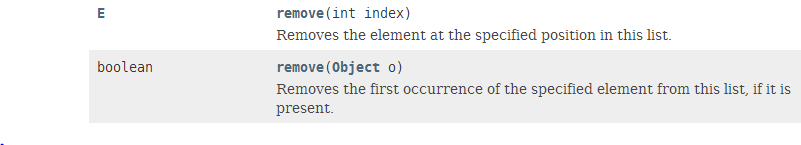
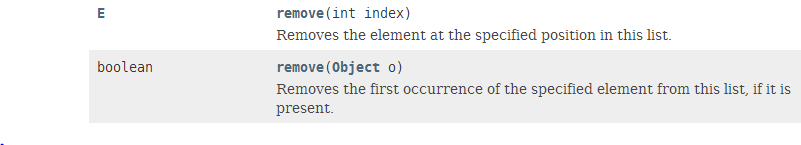
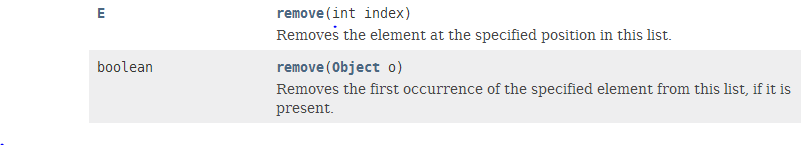
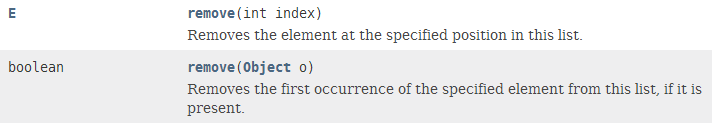
[Array](https://leetcode.com/tag/array/) [Hash Table](https://leetcode.com/tag/hash-table/) [Design](https://leetcode.com/tag/design/)

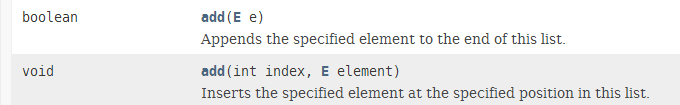
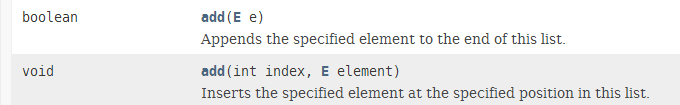
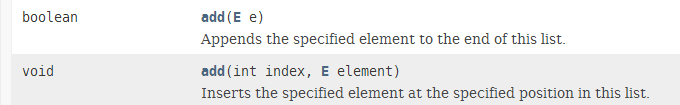
我的做法（因为set记不住放入元素的顺序，获取随机值时是使用随机选择一个索引值得方式，所以这里没有用set，而是使用ArrayList）

注意ArrayList的一些方法





窗体顶端

窗体底端

public class RandomizedSet {

ArrayList<Integer> set;

/\*\* Initialize your data structure here. \*/

public RandomizedSet() {

set=new ArrayList<>();

}

/\*\* Inserts a value to the set. Returns true if the set did not already contain the specified element. \*/

public boolean insert(int val) {

if(set.contains((Integer)val)) return false;

else

{

set.add((Integer)val);

return true;

}

}

/\*\* Removes a value from the set. Returns true if the set contained the specified element. \*/

public boolean remove(int val) {

return set.remove((Integer)val);

}

/\*\* Get a random element from the set. \*/

public int getRandom() {

int j = (int)Math.random() % set.size();

int x = set.get(j);

return x;

}

}

/\*\*

\* Your RandomizedSet object will be instantiated and called as such:

\* RandomizedSet obj = new RandomizedSet();

\* boolean param\_1 = obj.insert(val);

\* boolean param\_2 = obj.remove(val);

\* int param\_3 = obj.getRandom();

\*/

HashMap<放入的元素，当时元素个数>

ArrayList：放入的元素

获得随机数时，从ArrayList中返回随机索引对应的值

删除元素：（不是很理解这么处理的原因）在map中找到要删除的元素的放入位置，如果不是最后一个数，则将最后一个数放到要删除的数的位置。然后删除最后一个数。

用ArrayList来记住存入的顺序？？

**Java solution using a HashMap and a ArrayList. (131 ms)**

I got a similar question for my phone interview. The difference is that the duplicated number is allowed. So, think that it is a follow-up of this question.  
How do you modify your code to allow duplicated number?

**public** **class** **RandomizedSet** {

ArrayList<Integer> nums;

HashMap<Integer, Integer> locs;

java.util.Random rand = **new** java.util.Random();

*/\*\* Initialize your data structure here. \*/*

**public** **RandomizedSet**() {

nums = **new** ArrayList<Integer>();

locs = **new** HashMap<Integer, Integer>();

}

*/\*\* Inserts a value to the set. Returns true if the set did not already contain the specified element. \*/*

**public** boolean **insert**(**int** val) {

boolean contain = locs.containsKey(val);

**if** ( contain ) **return** false;

locs.put( val, nums.size());

nums.add(val);

**return** true;

}

*/\*\* Removes a value from the set. Returns true if the set contained the specified element. \*/*

**public** boolean **remove**(**int** val) {

boolean contain = locs.containsKey(val);

**if** ( ! contain ) **return** false;

**int** loc = locs.**get**(val);

**if** (loc < nums.size() - 1 ) { *// not the last one than swap the last one with this val*

**int** lastone = nums.**get**(nums.size() - 1 );

nums.**set**( loc , lastone );

locs.put(lastone, loc);

}

locs.remove(val);

nums.remove(nums.size() - 1);

**return** true;

}

*/\*\* Get a random element from the set. \*/*

**public** **int** **getRandom**() {

**return** nums.**get**( rand.nextInt(nums.size()) );

}

}

The follow-up: allowing duplications.  
For example, after insert(1), insert(1), insert(2), getRandom() should have 2/3 chance return 1 and 1/3 chance return 2.  
Then, remove(1), 1 and 2 should have an equal chance of being selected by getRandom().

The idea is to add a set to the hashMap to remember all the locations of a duplicated number.

**public** **class** **RandomizedSet** {

ArrayList<Integer> nums;

HashMap<Integer, Set<Integer>> locs;

java.util.Random rand = **new** java.util.Random();

*/\*\* Initialize your data structure here. \*/*

**public** **RandomizedSet**() {

nums = **new** ArrayList<Integer>();

locs = **new** HashMap<Integer, Set<Integer>>();

}

*/\*\* Inserts a value to the set. Returns true if the set did not already contain the specified element. \*/*

**public** boolean **insert**(**int** val) {

boolean contain = locs.containsKey(val);

**if** ( ! contain ) locs.put( val, **new** HashSet<Integer>() );

locs.**get**(val).add(nums.size());

nums.add(val);

**return** ! contain ;

}

*/\*\* Removes a value from the set. Returns true if the set contained the specified element. \*/*

**public** boolean **remove**(**int** val) {

boolean contain = locs.containsKey(val);

**if** ( ! contain ) **return** false;

**int** loc = locs.**get**(val).iterator().next();

**if** (loc < nums.size() - 1 ) {

**int** lastone = nums.**get**(nums.size() - 1 );

nums.**set**( loc , lastone );

locs.**get**(lastone).remove(nums.size() - 1);

locs.**get**(lastone).add(loc);

}

nums.remove(nums.size() - 1);

locs.**get**(val).remove(loc);

**if** (locs.**get**(val).isEmpty()) locs.remove(val);

**return** true;

}

*/\*\* Get a random element from the set. \*/*

**public** **int** **getRandom**() {

**return** nums.**get**( rand.nextInt(nums.size()) );

}

}

### Design Twitter

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/design-twitter/submissions/)

* Total Accepted: **8811**
* Total Submissions: **37086**
* Difficulty: **Medium**

Design a simplified version of Twitter where users can post tweets, follow/unfollow another user and is able to see the 10 most recent tweets in the user's news feed. Your design should support the following methods:

1. **postTweet(userId, tweetId)**: Compose a new tweet.
2. **getNewsFeed(userId)**: Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user herself. Tweets must be ordered from most recent to least recent.
3. **follow(followerId, followeeId)**: Follower follows a followee.
4. **unfollow(followerId, followeeId)**: Follower unfollows a followee.

**Example:**

Twitter twitter = new Twitter();

// User 1 posts a new tweet (id = 5).

twitter.postTweet(1, 5);

// User 1's news feed should return a list with 1 tweet id -> [5].

twitter.getNewsFeed(1);

// User 1 follows user 2.

twitter.follow(1, 2);

// User 2 posts a new tweet (id = 6).

twitter.postTweet(2, 6);

// User 1's news feed should return a list with 2 tweet ids -> [6, 5].

// Tweet id 6 should precede tweet id 5 because it is posted after tweet id 5.

twitter.getNewsFeed(1);

// User 1 unfollows user 2.

twitter.unfollow(1, 2);

// User 1's news feed should return a list with 1 tweet id -> [5],

// since user 1 is no longer following user 2.

twitter.getNewsFeed(1);

[Hash Table](https://leetcode.com/tag/hash-table/) [Heap](https://leetcode.com/tag/heap/) [Design](https://leetcode.com/tag/design/)

下面这个方法比较全面。

1，Tweet（有时间戳字段，初始化tweet后自增）之间用链表的形式连接，且是头插式，这样可以让新的tweet保持在链表的最前方。

2，设user class。

3，getNewsFeed时，需要合并好几人的tweet。合并方法是：使用优先队列（有个构造方法，很不错，记住）；将所有followers的tweet\_head加入优先队列。

因为只需要返回最新的10个推特，所以计数，并依次将选中tweet的next加入到优先列表。窗体顶端

窗体底端

**Java OO Design with most efficient function getNewsFeed**

**public** **class** **Twitter** {

**private** **static** **int** timeStamp=0;

*// easy to find if user exist*

**private** Map<Integer, User> userMap;

*// Tweet link to next Tweet so that we can save a lot of time*

*// when we execute getNewsFeed(userId)*

**private** **class** **Tweet**{

**public** **int** id;

**public** **int** time;

**public** Tweet next;

**public** **Tweet**(**int** id){

**this**.id = id;

time = timeStamp++;

next=null;

}

}

*// OO design so User can follow, unfollow and post itself*

**public** **class** **User**{

**public** **int** id;

**public** Set<Integer> followed;

**public** Tweet tweet\_head;

**public** **User**(**int** id){

**this**.id=id;

followed = **new** HashSet<>();

follow(id); *// first follow itself*

tweet\_head = null;

}

**public** **void** **follow**(**int** id){

followed.add(id);

}

**public** **void** **unfollow**(**int** id){

followed.remove(id);

}

*// everytime user post a new tweet, add it to the head of tweet list.*

**public** **void** **post**(**int** id){

Tweet t = **new** Tweet(id);

t.next=tweet\_head;

tweet\_head=t; //头插式（新发布的消息，放在消息列表的头部）

}

}

*/\*\* Initialize your data structure here. \*/*

**public** **Twitter**() {

userMap = **new** HashMap<Integer, User>();

}

*/\*\* Compose a new tweet. \*/*

**public** **void** **postTweet**(**int** userId, **int** tweetId) {

**if**(!userMap.containsKey(userId)){

User u = **new** User(userId);

userMap.put(userId, u);

}

userMap.**get**(userId).post(tweetId);

}

*// Best part of this.*

*// first get all tweets lists from one user including itself and all people it followed.*

*// Second add all heads into a max heap. Every time we poll a tweet with*

*// largest time stamp from the heap, then we add its next tweet into the heap.*

*// So after adding all heads we only need to add 9 tweets at most into this*

*// heap before we get the 10 most recent tweet.*

**public** List<Integer> **getNewsFeed**(**int** userId) {

List<Integer> res = **new** LinkedList<>();

**if**(!userMap.containsKey(userId)) **return** res;

Set<Integer> users = userMap.**get**(userId).followed;

PriorityQueue<Tweet> q = **new** PriorityQueue<Tweet>(users.size(), (a,b)->(b.time-a.time));

**for**(**int** user: users){

Tweet t = userMap.**get**(user).tweet\_head;

*// very imporant! If we add null to the head we are screwed.*

**if**(t!=null){

q.add(t);

}

}

**int** n=0;

**while**(!q.isEmpty() && n<10){

Tweet t = q.poll();

res.add(t.id);

n++;

**if**(t.next!=null)

q.add(t.next);

}

**return** res;

}

*/\*\* Follower follows a followee. If the operation is invalid, it should be a no-op. \*/*

*//将*followerId、followeeId都放入用户表里

**public** **void** **follow**(**int** followerId, **int** followeeId) {

**if**(!userMap.containsKey(followerId)){

User u = **new** User(followerId);

userMap.put(followerId, u);

}

**if**(!userMap.containsKey(followeeId)){

User u = **new** User(followeeId);

userMap.put(followeeId, u);

}

userMap.**get**(followerId).follow(followeeId);

}

*/\*\* Follower unfollows a followee. If the operation is invalid, it should be a no-op. \*/*

**public** **void** **unfollow**(**int** followerId, **int** followeeId) {

**if**(!userMap.containsKey(followerId) || followerId==followeeId)

**return**;

userMap.**get**(followerId).unfollow(followeeId);

}

}

*/\*\**

*\* Your Twitter object will be instantiated and called as such:*

*\* Twitter obj = new Twitter();*

*\* obj.postTweet(userId,tweetId);*

*\* List<Integer> param\_2 = obj.getNewsFeed(userId);*

*\* obj.follow(followerId,followeeId);*

*\* obj.unfollow(followerId,followeeId);*

*\*/*



I think if you maintain an additional set of newsFeed under User, the function getNewsFeed will be even more efficient.

The idea is upon tweet creation, broadcast it to all followers' newsFeed. Upon follow, add followee's tweets to follower's newsFeed, and delete them upon unfollow. The assumption is that these 3 functions are called much less frequent than getNewsFeed.

**class** **Twitter**(object):

**def** **\_\_init\_\_**(self):

self.timer = itertools.count(step=-1)

self.tweets = collections.defaultdict(collections.deque)

self.followees = collections.defaultdict(set)

**def** **postTweet**(self, userId, tweetId):

self.tweets[userId].appendleft((next(self.timer), tweetId))

**def** **getNewsFeed**(self, userId):

tweets = heapq.merge(\*(self.tweets[u] **for** u **in** self.followees[userId] | {userId}))

**return** [t **for** \_, t **in** itertools.islice(tweets, 10)]

**def** **follow**(self, followerId, followeeId):

self.followees[followerId].add(followeeId)

**def** **unfollow**(self, followerId, followeeId):

self.followees[followerId].discard(followeeId)

**Java Solutions with Two Maps and PriorityQueue**

I use a map to track the tweets for each user. When we need to generate a news feed, I merge the news feed for all the followees and take the most recent 10. This is unlikely to perform, but the code passes the OJ. I'm sure design interviews ask for performance trade-offs and just posting this code in a design interview will not help you get an offer.

**public** **class** **Twitter** {

Map<Integer, Set<Integer>> fans = **new** HashMap<>();

Map<Integer, LinkedList<Tweet>> tweets = **new** HashMap<>();

**int** cnt = 0;

**public** **void** **postTweet**(**int** userId, **int** tweetId) {

**if** (!fans.containsKey(userId)) fans.put(userId, **new** HashSet<>());

fans.get(userId).add(userId);

**if** (!tweets.containsKey(userId)) tweets.put(userId, **new** LinkedList<>());

tweets.get(userId).addFirst(**new** Tweet(cnt++, tweetId));

}

**public** List<Integer> **getNewsFeed**(**int** userId) {

**if** (!fans.containsKey(userId)) **return** **new** LinkedList<>();

PriorityQueue<Tweet> feed = **new** PriorityQueue<>((t1, t2) -> t2.time - t1.time);

fans.get(userId).stream()

.filter(f -> tweets.containsKey(f))

.forEach(f -> tweets.get(f).forEach(feed::add));

List<Integer> res = **new** LinkedList<>();

**while** (feed.size() > 0 && res.size() < 10) res.add(feed.poll().id);

**return** res;

}

**public** **void** **follow**(**int** followerId, **int** followeeId) {

**if** (!fans.containsKey(followerId)) fans.put(followerId, **new** HashSet<>());

fans.get(followerId).add(followeeId);

}

**public** **void** **unfollow**(**int** followerId, **int** followeeId) {

**if** (fans.containsKey(followerId) && followeeId != followerId) fans.get(followerId).remove(followeeId);

}

**class** **Tweet** {

**int** time;

**int** id;

Tweet(**int** time, **int** id) {

**this**.time = time;

**this**.id = id;

}

}

}

Great solution. There are two possible places I think can improve the time complexity as well as space.

A. Since we will get 10 most tweets from each user. We can just save up to 10 tweets for each user. And we can use Dqueue to do that.

B. In getNewsFeed(). we can use minHeap instead so that PQueue only needs to save 10 tweets. And after each iteration if PQueue's size is bigger than 10, take the tweet with the min timestamp out of PQ.

## Trie

Trie;  又称单词查找树;  是一种树形结构;  用于保存大量的字符串。它的优点是：利用字符串的公共前缀来节约存储空间。

<http://www.cnblogs.com/huangxincheng/archive/2012/11/25/2788268.html>

### Implement Trie (Prefix Tree)

[My Submissions](https://leetcode.com/problems/implement-trie-prefix-tree/submissions/)

QuestionEditorial Solution

Total Accepted: **33890** Total Submissions: **133605** Difficulty: **Medium**

Implement a trie with insert, search, and startsWith methods.

**Note:**  
You may assume that all inputs are consist of lowercase letters a-z.

**class** TrieNode

{

**public**:

TrieNode \*next[26];

**bool** is\_word;

// Initialize your data structure here.

TrieNode(**bool** b = **false**)

{

memset(next, 0, **sizeof**(next));

is\_word = b;

}

};

**class** Trie

{

TrieNode \*root;

**public**:

Trie()

{

root = **new** TrieNode();

}

// Inserts a word into the trie.

**void** insert(**string** s)

{

TrieNode \*p = root;

**for**(**int** i = 0; i < s.size(); ++ i)

{

**if**(p -> next[s[i] - 'a'] == NULL)

p -> next[s[i] - 'a'] = **new** TrieNode();

p = p -> next[s[i] - 'a'];

}

p -> is\_word = **true**;

}

// Returns if the word is in the trie.

**bool** search(**string** key)

{

TrieNode \*p = find(key);

**return** p != NULL && p -> is\_word;

}

// Returns if there is any word in the trie

// that starts with the given prefix.

**bool** startsWith(**string** prefix)

{

**return** find(prefix) != NULL;

}

**private**:

TrieNode\* find(**string** key)

{

TrieNode \*p = root;

**for**(**int** i = 0; i < key.size() && p != NULL; ++ i)

p = p -> next[key[i] - 'a'];

**return** p;

}

};

Java

**class** **TrieNode** {

**public** **boolean** isWord;

**public** TrieNode[] children = **new** TrieNode[26];

**public** TrieNode() {}

}

**public** **class** **Trie** {

**private** TrieNode root;

**public** Trie() {

root = **new** TrieNode();

}

**public** **void** insert(String word) {

TrieNode ws = root;

**for**(**int** i = 0; i < word.length(); i++){

**char** c = word.charAt(i);

**if**(ws.children[c - 'a'] == **null**){

ws.children[c - 'a'] = **new** TrieNode();

}

ws = ws.children[c - 'a'];

}

ws.isWord = **true**;

}

**public** **boolean** search(String word) {

TrieNode ws = root;

**for**(**int** i = 0; i < word.length(); i++){

**char** c = word.charAt(i);

**if**(ws.children[c - 'a'] == **null**) **return** **false**;

ws = ws.children[c - 'a'];

}

**return** ws.isWord;

}

**public** **boolean** startsWith(String prefix) {

TrieNode ws = root;

**for**(**int** i = 0; i < prefix.length(); i++){

**char** c = prefix.charAt(i);

**if**(ws.children[c - 'a'] == **null**) **return** **false**;

ws = ws.children[c - 'a'];

}

**return** **true**;

}

}

With my solution I took the simple approach of giving each TrieNode a 26 element array of each possible child node it may have. I only gave 26 children nodes because we are only working with lowercase 'a' - 'z'. If you are uncertain why I made the root of my Trie an empty character this is a standard/typical approach for building out a Trie it is somewhat arbitrary what the root node is.

For insert I used the following algorithm. Loop through each character in the word being inserted check if the character is a child node of the current TrieNode i.e. check if the array has a populated value in the index of this character. If the current character ISN'T a child node of my current node add this character representation to the corresponding index location then set current node equal to the child that was added. However if the current character IS a child of the current node only set current node equal to the child. After evaluating the entire String the Node we left off on is marked as a word this allows our Trie to know which words exist in our "dictionary"

For search I simply navigate through the Trie if I discover the current character isn't in the Trie I return false. After checking each Char in the String I check to see if the Node I left off on was marked as a word returning the result.

Starts with is identical to search except it doesn't matter if the Node I left off was marked as a word or not if entire string evaluated i always return true;

### Add and Search Word - Data structure design

[My Submissions](https://leetcode.com/problems/add-and-search-word-data-structure-design/submissions/)

QuestionEditorial Solution

Total Accepted: **23128** Total Submissions: **114183** Difficulty: **Medium**

Design a data structure that supports the following two operations:

void addWord(word)

bool search(word)

search(word) can search a literal word or a regular expression string containing only letters a-z or .. A . means it can represent any one letter.

For example:

addWord("bad")

addWord("dad")

addWord("mad")

search("pad") -> false

search("bad") -> true

search(".ad") -> true

search("b..") -> true

**Note:**  
You may assume that all words are consist of lowercase letters a-z.

[click to show hint.](https://leetcode.com/problems/add-and-search-word-data-structure-design/)

You should be familiar with how a Trie works. If not, please work on this problem: [Implement Trie (Prefix Tree)](https://leetcode.com/problems/implement-trie-prefix-tree/) first.

我的做法（基于实现Trie的做法）

// Adds a word into the data structure.

public void addWord(String word) {

insert(word);

}

// Returns if the word is in the data structure. A word could

// contain the dot character '.' to represent any one letter.

public boolean search(TrieNode root, String word) {

if(word.length()==0) return root.isWord;

if(word.charAt(0)=='.')

{

for(int i=0;i<26;i++)

{

if(root.children[i]!=null)

{

if(search(root.children[i],word.substring(1)))//如果真

{

return true;

}

}

}

return false;

}

int k = word.charAt(0) - 'a';

if(root.children[k]==null) return false;

String nextWord = word.substring(1);

return search(root.children[k],nextWord);

}

public boolean search(String word)

{

return search(root,word);

}

This problem is an application of the Trie data structure. In the following, it is assumed that you have solved [Implement Trie (Prefix Tree)](https://leetcode.com/problems/implement-trie-prefix-tree/).

Now, let's first look at the TrieNode class. I define it as follows.

**class** **TrieNode** {

**public**:

bool isKey;

TrieNode\* children[26];

TrieNode(): isKey(**false**) {

memset(children, **NULL**, sizeof(TrieNode\*) \* 26);

}

};

The field isKey is to label whether the string comprised of characters starting from root to the current node is a key (word that has been added). In this problem, only lower-case letters a - zneed to be considered, so each TrieNode has at most 26 children. I store it in an array ofTrieNode\*: children[i] corresponds to letter 'a' + i. The remaining code defines the constructor of the TrieNode class.

Adding a word can be done in the same way as in [Implement Trie (Prefix Tree)](https://leetcode.com/problems/implement-trie-prefix-tree/). The basic idea is to create a TrieNode corresponding to each letter in the word. When we are done, label the last node to be a key (set isKey = true). The code is as follows.

**void** addWord(**string** word) {

TrieNode\* run = root;

**for** (**char** c : word) {

**if** (!(run -> children[c - 'a']))

run -> children[c - 'a'] = **new** TrieNode();

run = run -> children[c - 'a'];

}

run -> isKey = **true**;

}

By the way, root is defined as private data of WordDictionary:

**private**:

TrieNode\* root;

And the WordDictionary class has a constructor to initialize root:

WordDictionary() {

root = **new** TrieNode();

}

Now we are left only with search. Let's do it. The basic idea is still the same as typical search operations in a Trie. The critical part is how to deal with the dots .. Well, my solution is very naive in this place. Each time when we reach a ., just traverse all the children of the current node and recursively search the remaining substring in word starting from that children. So I define a helper function query for search that takes in a string and a starting node. And the initial call to query is like query(word, root).

By the way, I pass a char\* instead of string to query and it greatly speeds up the code. So the initial call to query is actually query(word.c\_str(), root).

Now I put all the codes together below. Hope it to be useful!

**class** TrieNode {

**public**:

**bool** isKey;

TrieNode\* children[26];

TrieNode(): isKey(**false**) {

memset(children, NULL, **sizeof**(TrieNode\*) \* 26);

}

};

**class** WordDictionary {

**public**:

WordDictionary() {

root = **new** TrieNode();

}

// Adds a word into the data structure.

**void** addWord(**string** word) {

TrieNode\* run = root;

**for** (**char** c : word) {

**if** (!(run -> children[c - 'a']))

run -> children[c - 'a'] = **new** TrieNode();

run = run -> children[c - 'a'];

}

run -> isKey = **true**;

}

// Returns if the word is in the data structure. A word could

// contain the dot character '.' to represent any one letter.

**bool** search(**string** word) {

**return** query(word.c\_str(), root);

}

**private**:

TrieNode\* root;

**bool** query(**const** **char**\* word, TrieNode\* node) {

TrieNode\* run = node;

**for** (**int** i = 0; word[i]; i++) {

**if** (run && word[i] != '.')

run = run -> children[word[i] - 'a'];

**else** **if** (run && word[i] == '.') {

TrieNode\* tmp = run;

**for** (**int** j = 0; j < 26; j++) {

run = tmp -> children[j];

**if** (query(word + i + 1, run))

**return** **true**;

}

}

**else** **break**;

}

**return** run && run -> isKey;

}

};

// Your WordDictionary object will be instantiated and called as such:

// WordDictionary wordDictionary;

// wordDictionary.addWord("word");

// wordDictionary.search("pattern");

**public** **class** **WordDictionary** {

WordNode root = **new** WordNode();

**public** **void** addWord(String word) {

**char** chars[] = word.toCharArray();

addWord(chars, 0, root);

}

**private** **void** addWord(**char**[] chars, **int** index, WordNode parent) {

**char** c = chars[index];

**int** idx = c-'a';

WordNode node = parent.children[idx];

**if** (node == **null**){

node = **new** WordNode();

parent.children[idx]=node;

}

**if** (chars.length == index+1){

node.isLeaf=**true**;

**return**;

}

addWord(chars, ++index, node);

}

**public** **boolean** search(String word) {

**return** search(word.toCharArray(), 0, root);

}

**private** **boolean** search(**char**[] chars, **int** index, WordNode parent){

**if** (index == chars.length){

**if** (parent.isLeaf){

**return** **true**;

}

**return** **false**;

}

WordNode[] childNodes = parent.children;

**char** c = chars[index];

**if** (c == '.'){

**for** (**int** i=0;i<childNodes.length;i++){

WordNode n = childNodes[i];

**if** (n !=**null**){

**boolean** b = search(chars, index+1, n);

**if** (b){

**return** **true**;

}

}

}

**return** **false**;

}

WordNode node = childNodes[c-'a'];

**if** (node == **null**){

**return** **false**;

}

**return** search(chars, ++index, node);

}

**private** **class** **WordNode**{

**boolean** isLeaf;

WordNode[] children = **new** WordNode[26];

}

}

Using backtrack to check each character of word to search.

**public** **class** **WordDictionary** {

**public** **class** **TrieNode** {

**public** TrieNode[] children = **new** TrieNode[26];

**public** String item = "";

}

**private** TrieNode root = **new** TrieNode();

**public** **void** addWord(String word) {

TrieNode node = root;

**for** (**char** c : word.toCharArray()) {

**if** (node.children[c - 'a'] == **null**) {

node.children[c - 'a'] = **new** TrieNode();

}

node = node.children[c - 'a'];

}

node.item = word;

}

**public** **boolean** search(String word) {

**return** match(word.toCharArray(), 0, root);

}

**private** **boolean** match(**char**[] chs, **int** k, TrieNode node) {

**if** (k == chs.length) **return** !node.item.equals("");

**if** (chs[k] != '.') {

**return** node.children[chs[k] - 'a'] != **null** && match(chs, k + 1, node.children[chs[k] - 'a']);

} **else** {

**for** (**int** i = 0; i < node.children.length; i++) {

**if** (node.children[i] != **null**) {

**if** (match(chs, k + 1, node.children[i])) {

**return** **true**;

}

}

}

}

**return** **false**;

}

}

## 贪心

### Patching Array

[My Submissions](https://leetcode.com/problems/patching-array/submissions/)

QuestionEditorial Solution

Total Accepted: **7403** Total Submissions: **25745** Difficulty: **Medium**

Given a sorted positive integer array *nums* and an integer *n*, add/patch elements to the array such that any number in range [1, n] inclusive can be formed by the sum of some elements in the array. Return the minimum number of patches required.

**Example 1:**  
*nums* = [1, 3], *n* = 6  
Return 1.

Combinations of *nums* are [1], [3], [1,3], which form possible sums of: 1, 3, 4.  
Now if we add/patch 2 to *nums*, the combinations are: [1], [2], [3], [1,3], [2,3], [1,2,3].  
Possible sums are 1, 2, 3, 4, 5, 6, which now covers the range [1, 6].  
So we only need 1 patch.

**Example 2:**  
*nums* = [1, 5, 10], *n* = 20  
Return 2.  
The two patches can be [2, 4].

**Example 3:**  
*nums* = [1, 2, 2], *n* = 5  
Return 0.

这个方法不错

遍历数组，每一轮，确定数组元素0到元素i可以构成的最大数miss（[1,miss）；

如果下一个数组元素值比miss小，则直接构造该轮，加入该数组元素后能构造的最大数miss；

否则，需要打补丁元素miss。然后继续判断，直到到达目标值。

**Solution**

**int** minPatches(**vector**<**int**>& nums, **int** n) {

**long** miss = 1, added = 0, i = 0;//注意miss需要设成long类型

**while** (miss <= n) { //因为miss值取不到，所以miss>n才能退出循环。注意miss的初值设定为1

**if** (i < nums.size() && nums[i] <= miss) {

miss += nums[i++];

} **else** {

miss += miss;

added++;

}

}

**return** added;

}

**Explanation**

Let miss be the smallest sum in [0,n] that we might be missing. Meaning we already know we can build all sums in [0,miss). Then if we have a number num <= miss in the given array, we can add it to those smaller sums to build all sums in [0,miss+num). If we don't, then we must add such a number to the array, and it's best to add miss itself, to maximize the reach.

**Example:** Let's say the input is nums = [1, 2, 4, 13, 43] and n = 100. We need to ensure that all sums in the range [1,100] are possible.

Using the given numbers 1, 2 and 4, we can already build all sums from 0 to 7, i.e., the range [0,8). But we can't build the sum 8, and the next given number (13) is too large. So we insert 8 into the array. Then we can build all sums in [0,16).

Do we need to insert 16 into the array? No! We can already build the sum 3, and adding the given 13 gives us sum 16. We can also add the 13 to the other sums, extending our range to [0,29).

And so on. The given 43 is too large to help with sum 29, so we must insert 29 into our array. This extends our range to [0,58). But then the 43 becomes useful and expands our range to [0,101). At which point we're done.

我写的java版，注意miss<=n，再翻一番后可能会超过int表达的最大数，所以一定要有long

public class Solution {

public int minPatches(int[] nums, int n) {

long miss=1;

int i=0, add=0;

while(miss<=n)

{

if(i<nums.length && nums[i]<=miss)

{

miss+=nums[i];

i++;

}

else

{

miss+=miss;

add++;

}

}

return add;

}

}

**Another implementation**, though I prefer the above one.

**int** minPatches(**vector**<**int**>& nums, **int** n) {

**int** count = 0, i = 0;

**for** (**long** miss=1; miss <= n; count++)

miss += (i < nums.size() && nums[i] <= miss) ? nums[i++] : miss;

**return** count - i;

}

The question asked for the "**minimum** number of patches required". In other words, it asked for an optimal solution. Lots of problems involving optimal solution can be solved by dynamic programming and/or greedy algorithm. I started with greedy algorithm which is conceptually easy to design. Typically, a greedy algorithm needs selection of best moves for a subproblem. So what is our best move?

Think about this example: nums = [1, 2, 3, 9]. We naturally want to iterate through nums from left to right and see what we would discover. After we encountered 1, we know 1...1 is patched completely. After encountered 2, we know 1...3 (1+2) is patched completely. After we encountered 3, we know 1...6 (1+2+3) is patched completely. After we encountered 9, the smallest number we can get is 9. So we must patch a new number here so that we don't miss 7, 8. To have 7, the numbers we can patch is 1, 2, 3 ... 7. Any number greater than 7 won't help here. Patching 8 will not help you get 7. So we have 7 numbers (1...7) to choose from. I hope you can see number 7 works best here because if we chose number 7, we can move all the way up to 1+2+3+7 = 13. (1...13 is patched completely) and it makes us reach n as quickly as possible. After we patched 7 and reach 13, we can consider last element 9 in nums. Having 9 makes us reach 13+9 = 22, which means 1...22 is completely patched. If we still did't reach n, we can then patch 23, which makes 1...45 (22+23) completely patched. We continue until we reach n.

### Jump Game

[My Submissions](https://leetcode.com/problems/jump-game/submissions/)

QuestionEditorial Solution

Total Accepted: **75122** Total Submissions: **266988** Difficulty: **Medium**

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

For example:  
A = [2,3,1,1,4], return true.

A = [3,2,1,0,4], return false.

I just iterate and update the maximal index that I can reach

**bool** canJump(**int** A[], **int** n) {

**int** i = 0;

**for** (**int** reach = 0; i < n && i <= reach; ++i)

reach = max(i + A[i], reach);

**return** i == n;

}

public boolean canJump(int[] A) {

int max = 0;

**for**(int i=0;i<A.length;i++){

**if**(i>max) {**return** **false**;}

max = Math.max(A[i]+i,max);

}

**return** **true**;

}

### Jump Game II

[My Submissions](https://leetcode.com/problems/jump-game-ii/submissions/)

Question

Total Accepted: **57090** Total Submissions: **229702** Difficulty: **Hard**

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

For example:  
Given array A = [2,3,1,1,4]

The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

**Note:**  
You can assume that you can always reach the last index.

I try to change this problem to a BFS problem, where nodes in level i are all the nodes that can be reached in i-1th jump. for example. 2 3 1 1 4 , is 2|| 3 1|| 1 4 ||

clearly, the minimum jump of 4 is 2 since 4 is in level 3. my ac code.

这个方法好，当前位置加上当前索引，可以得出当前能到的最远索引。用i作为全局索引

int jump(int A[], int n) {

**if**(n<2)**return** 0;

int level=0,currentMax=0,i=0,nextMax=0;// //能到的最大索引，当前索引

**while**(currentMax-i+1>0){ //nodes count of current level>0

level++;

**for**(;i<=currentMax;i++){ //traverse current level , **and** update the max reach of next level

nextMax=max(nextMax,A[i]+i);

**if**(nextMax>=n-1)**return** level; // **if** last element is **in** level+1, **then** the min jump=level

}

currentMax=nextMax;

}

**return** 0;

}

Hi All, below is my AC solution:

*public* ***int*** *jump(****int****[] A) {*

***int*** *maxReach = A[0];*

***int*** *edge = 0;*

***int*** *minstep = 0;*

***for****(****int*** *i = 1; i < A.****length****; i++) {*

***if*** *(i > edge) {*

*minstep += 1;*

*edge = maxReach;*

***if****(edge > A.****length*** *- 1)*

***return*** *minstep;*

*}*

*maxReach = Math.max(maxReach, A[i] + i);*

***if*** *(maxReach == i):*

***return*** *-1;*

*}*

***return*** *minstep;*

*}*

When iterate the array, I set an edge for the Search phase, which means that if I exceeds the edge, the minstep must add one and the maxReach will be update. And when the last index is within the range of the edge, output the minstep.

[2, 3, 1, 1, 4]

First, the edge is 0; Second, after start iterate the array, it exceeds the edge 0 when reaching the A[0] and update the edge to 2; Third, after it reach the A[2], it exceeds the edge 2 and update the new edge to the maxReach 4. Finally, end of the array is inside the edge, output the minstep.

### Gas Station

[My Submissions](https://leetcode.com/problems/gas-station/submissions/)

QuestionEditorial Solution

Total Accepted: **59292** Total Submissions: **217675** Difficulty: **Medium**

There are *N* gas stations along a circular route, where the amount of gas at station *i* is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station *i* to its next station (*i*+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

**Note:**  
The solution is guaranteed to be unique.

题目假定车往一个方向开。

如果gas总和大于等于cost总和，则一定可以开完一圈。

用total记录总的油的使用。若为负，则说明，到不了一圈。

用tank记录每次油的使用。用start记录结果。

从i处开始往下走，如果总汽油为负，则不从i处开始走，从它的下一个点开始走，tank要归零。

I have thought for a long time and got two ideas:

* If car starts at A and can not reach B. Any station between A and B can not reach B.(B is the first station that A can not reach.)
* If the total number of gas is bigger than the total number of cost. There must be a solution.
* (Should I prove them?)

Here is my solution based on those ideas:

**class** Solution {

**public**:

**int** canCompleteCircuit(**vector**<**int**> &gas, **vector**<**int**> &cost) {

**int** start(0),total(0),tank(0);

//if car fails at 'start', record the next station

**for**(**int** i=0;i<gas.size();i++)

**if**((tank=tank+gas[i]-cost[i])<0)

{start=i+1;total+=tank;tank=0;}

**return** (total+tank<0)? -1:start;

}

};

I have got one solution to this problem. I am not sure whether somebody has already posted this solution.

**class** Solution {

**public**:

**int** canCompleteCircuit(**vector**<**int**> &gas, **vector**<**int**> &cost) {

**int** start = gas.size()-1;

**int** end = 0;

**int** sum = gas[start] - cost[start];

**while** (start > end) {

**if** (sum >= 0) {

sum += gas[end] - cost[end];

++end;

}

**else** {

--start;

sum += gas[start] - cost[start];

}

}

**return** sum >= 0 ? start : -1;

}

};

**class** Solution {

**public**:

**int** canCompleteCircuit(**vector**<**int**>& gas, **vector**<**int**>& cost) {

**int** size=gas.size();

**int** sum=0;

**int** res=0;

**int** total=0;

**for**(**int** i=0; i<size; ++i){

sum+=gas[i]-cost[i];

**if**(sum<0){

total+=sum;

sum=0;

res=i+1;

}

}

total+=sum;

**return** total<0?-1:res;

}};

The idea is simple.

1. Whenever the sum is negative, reset it and let the car start from next point.
2. In the mean time, add up all of the left gas to total. If it's negative finally, return -1 since it's impossible to finish.
3. If it's non-negative, return the last point saved in res;

### Hard

### Remove Duplicate Letters

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/remove-duplicate-letters/submissions/)

* Total Accepted: **17005**
* Total Submissions: **63797**
* Difficulty: **Hard**

Given a string which contains only lowercase letters, remove duplicate letters so that every letter appear once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

**Example:**

Given "bcabc"  
Return "abc"

Given "cbacdcbc"  
Return "acdb"

**Credits:**  
Special thanks to [@dietpepsi](https://leetcode.com/discuss/user/dietpepsi) for adding this problem and creating all test cases.

[Subscribe](https://leetcode.com/subscribe/) to see which companies asked this question

Hide Tags

[Stack](https://leetcode.com/tag/stack/) [Greedy](https://leetcode.com/tag/greedy/)

这个方法很巧妙

一次递归确定一个字符顺序，下次递归只需要考虑该字符后面的字符串（该字符串需要去除所有出现的该字符）

这个确定的字符是，到出现单个字符为止的pos最小的一个

如果遍历到的字符个数不是为1，则继续，同时记录pos最小的字符。

注意终点，当s长度为0,时直接返回“”

**A short O(n) recursive greedy solution**

Given the string s, the greedy choice (i.e., the leftmost letter in the answer) is the smallest s[i], s.t.  
the suffix s[i .. ] contains all the unique letters. (Note that, when there are more than one smallest s[i]'s, we choose the leftmost one. Why? Simply consider the example: "abcacb".)

After determining the greedy choice s[i], we get a new string s' from s by

1. removing all letters to the left of s[i],
2. removing all s[i]'s from s.

We then recursively solve the problem w.r.t. s'.

The runtime is O(26 \* n) = O(n).

|  |  |
| --- | --- |
| [**String**](file:///E:\docsJava\api\java\lang\String.html) | [**replaceAll**](file:///E:\docsJava\api\java\lang\String.html#replaceAll-java.lang.String-java.lang.String-)([**String**](file:///E:\docsJava\api\java\lang\String.html) regex, [**String**](file:///E:\docsJava\api\java\lang\String.html) replacement)  Replaces each substring of this string that matches the given [**regular expression**](file:///E:\docsJava\api\java\util\regex\Pattern.html#sum)  with the given replacement. |

public class Solution {

public String removeDuplicateLetters(String s) {

int[] cnt = new int[26];

int pos = 0; // the position for the smallest s[i]

for (int i = 0; i < s.length(); i++) cnt[s.charAt(i) - 'a']++; //统计字符串中对应字母数目

for (int i = 0; i < s.length(); i++) {

if (s.charAt(i) < s.charAt(pos)) pos = i;

if (--cnt[s.charAt(i) - 'a'] == 0) break;

}

return s.length() == 0 ? "" : s.charAt(pos) + removeDuplicateLetters(s.substring(pos + 1).replaceAll("" + s.charAt(pos), "")); //去重字母的一个方法

}

}

**Easy to understand iterative Java solution**

The basic idea is to find out the smallest result letter by letter (one letter at a time). Here is the thinking process for input "cbacdcbc":

1. find out the last appeared position for each letter;  
   c - 7  
   b - 6  
   a - 2  
   d - 4
2. find out the smallest index from the map in step 1 (a - 2);
3. the first letter in the final result must be the smallest letter from index 0 to index 2;
4. repeat step 2 to 3 to find out remaining letters.

* the smallest letter from index 0 to index 2: a
* the smallest letter from index 3 to index 4: c
* the smallest letter from index 4 to index 4: d
* the smallest letter from index 5 to index 6: b

so the result is "acdb"

Notes:

* after one letter is determined in step 3, it need to be removed from the "last appeared position map", and the same letter should be ignored in the following steps
* in step 3, the beginning index of the search range should be the index of previous determined letter plus one

**public** **class** **Solution** {

**public** String **removeDuplicateLetters**(String s) {

**if** (s == **null** || s.length() <= 1) **return** s;

Map<Character, Integer> lastPosMap = **new** HashMap<>();

**for** (**int** i = 0; i < s.length(); i++) {

lastPosMap.put(s.charAt(i), i);

}

**char**[] result = **new** **char**[lastPosMap.size()];

**int** begin = 0, end = findMinLastPos(lastPosMap);

**for** (**int** i = 0; i < result.length; i++) {

**char** minChar = 'z' + 1;

**for** (**int** k = begin; k <= end; k++) {

**if** (lastPosMap.containsKey(s.charAt(k)) && s.charAt(k) < minChar) {

minChar = s.charAt(k);

begin = k+1;

}

}

result[i] = minChar;

**if** (i == result.length-1) **break**;

lastPosMap.remove(minChar);

**if** (s.charAt(end) == minChar) end = findMinLastPos(lastPosMap);

}

**return** **new** String(result);

}

**private** **int** **findMinLastPos**(Map<Character, Integer> lastPosMap) {

**if** (lastPosMap == **null** || lastPosMap.isEmpty()) **return** -1;

**int** minLastPos = Integer.MAX\_VALUE;

**for** (**int** lastPos : lastPosMap.values()) {

minLastPos = Math.min(minLastPos, lastPos);

}

**return** minLastPos;

}

}

### Candy

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/candy/submissions/)

* Total Accepted: **55824**
* Total Submissions: **241862**
* Difficulty: **Hard**

There are *N* children standing in a line. Each child is assigned a rating value.

You are giving candies to these children subjected to the following requirements:

* Each child must have at least one candy.
* Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

这个方法不错

1. 初始化所有孩子的糖果数目为1；
2. 顺序遍历rating数组。如果当前rate值大于prev one，则更新当前糖果数目为prevde糖果值+1；
3. 逆序遍历rating数组。如果当前rate值大于prev one（逆序而言的），则更新当前糖果数目为max(prevde糖果值+1,当前糖果值)；（因为得保证满足上一轮的需求）

**int** candy(**vector**<**int**> &ratings)

{

**int** size=ratings.size();

**if**(size<=1)

**return** size;

**vector**<**int**> num(size,1);

**for** (**int** i = 1; i < size; i++)

{

**if**(ratings[i]>ratings[i-1])

num[i]=num[i-1]+1;

}

**for** (**int** i= size-1; i>0 ; i--)

{

**if**(ratings[i-1]>ratings[i])

num[i-1]=max(num[i]+1,num[i-1]);

}

**int** result=0;

**for** (**int** i = 0; i < size; i++)

{

result+=num[i];

// cout<<num[i]<<" ";

}

**return** result;

}

Hi guys!

This solution picks each element from the input array only once. First, we give a candy to the first child. Then for each child we have three cases:

1. His/her rating is equal to the previous one -> give 1 candy.
2. His/her rating is greater than the previous one -> give him (previous + 1) candies.
3. His/her rating is less than the previous one -> don't know what to do yet, let's just count the number of such consequent cases.

When we enter 1 or 2 condition we can check our count from 3. If it's not zero then we know that we were descending before and we have everything to update our total candies amount: number of children in descending sequence of raitings - coundDown, number of candies given at peak - prev (we don't update prev when descending). Total number of candies for "descending" children can be found through arithmetic progression formula (1+2+...+countDown). Plus we need to update our peak child if his number of candies is less then or equal to countDown.

Here's a pretty concise code below.

**public** **class** **Solution** {

**public** **int** candy(**int**[] ratings) {

**if** (ratings == **null** || ratings.length == 0) **return** 0;

**int** total = 1, prev = 1, countDown = 0;

**for** (**int** i = 1; i < ratings.length; i++) {

**if** (ratings[i] >= ratings[i-1]) {

**if** (countDown > 0) {

total += countDown\*(countDown+1)/2; // arithmetic progression

**if** (countDown >= prev) total += countDown - prev + 1;

countDown = 0;

prev = 1;

}

prev = ratings[i] == ratings[i-1] ? 1 : prev+1;

total += prev;

} **else** countDown++;

}

**if** (countDown > 0) { // if we were descending at the end

total += countDown\*(countDown+1)/2;

**if** (countDown >= prev) total += countDown - prev + 1;

}

**return** total;

}

}

### Wildcard Matching通配符匹配

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/wildcard-matching/submissions/)

* Total Accepted: **64164**
* Total Submissions: **355632**
* Difficulty: **Hard**

Implement wildcard pattern matching with support for '?' and '\*'.

'?' Matches any single character.

'\*' Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

The function prototype should be:

bool isMatch(const char \*s, const char \*p)

Some examples:

isMatch("aa","a") → false

isMatch("aa","aa") → true

isMatch("aaa","aa") → false

isMatch("aa", "\*") → true

isMatch("aa", "a\*") → true

isMatch("ab", "?\*") → true

isMatch("aab", "c\*a\*b") → false

[Subscribe](https://leetcode.com/subscribe/) to see which companies asked this question

Hide Tags

[Dynamic Programming](https://leetcode.com/tag/dynamic-programming/) [Backtracking](https://leetcode.com/tag/backtracking/) [Greedy](https://leetcode.com/tag/greedy/) [String](https://leetcode.com/tag/string/)

这个方法不太理解

I found this solution from http://yucoding.blogspot.com/2013/02/leetcode-question-123-wildcard-matching.html

The basic idea is to have one pointer for the string and one pointer for the pattern. This algorithm iterates at most length(string) + length(pattern) times, for each iteration, at least one pointer advance one step.

Here is Yu's elegant solution in C++

bool isMatch(const char \*s, const char \*p) {

const char\* star=NULL;

const char\* ss=s;

while (\*s){

//advancing both pointers when (both characters match) or ('?' found in pattern)

//note that \*p will not advance beyond its length

if ((\*p=='?')||(\*p==\*s)){s++;p++;continue;}

// \* found in pattern, track index of \*, only advancing pattern pointer

if (\*p=='\*'){star=p++; ss=s;continue;}

//current characters didn't match, last pattern pointer was \*, current pattern pointer is not \*

//only advancing pattern pointer

if (star){ p = star+1; s=++ss;continue;}

//current pattern pointer is not star, last patter pointer was not \*

//characters do not match

return false;

}

//check for remaining characters in pattern

while (\*p=='\*'){p++;}

return !\*p;

}

Here is my re-write in Java

﻿﻿﻿boolean comparison(String str, String pattern) {

int s = 0, p = 0, match = 0, starIdx = -1;

while (s < str.length()){

// advancing both pointers

if (p < pattern.length() && (pattern.charAt(p) == '?' || str.charAt(s) == pattern.charAt(p))){

s++;

p++;

}

// \* found, only advancing pattern pointer

else if (p < pattern.length() && pattern.charAt(p) == '\*'){

starIdx = p;

match = s;

p++;

}

// last pattern pointer was \*, advancing string pointer

else if (starIdx != -1){

p = starIdx + 1;

match++;

s = match;

}

//current pattern pointer is not star, last patter pointer was not \*

//characters do not match

else return false;

}

//check for remaining characters in pattern

while (p < pattern.length() && pattern.charAt(p) == '\*')

p++;

return p == pattern.length();

}

public class Solution {

public boolean isMatch(String s, String p) {

boolean[][] match=new boolean[s.length()+1][p.length()+1];

match[s.length()][p.length()]=true;

for(int i=p.length()-1;i>=0;i--){

if(p.charAt(i)!='\*')

break;

else

match[s.length()][i]=true;

}

for(int i=s.length()-1;i>=0;i--){

for(int j=p.length()-1;j>=0;j--){

if(s.charAt(i)==p.charAt(j)||p.charAt(j)=='?')

match[i][j]=match[i+1][j+1];

else if(p.charAt(j)=='\*')

match[i][j]=match[i+1][j]||match[i][j+1];

else

match[i][j]=false;

}

}

return match[0][0];

}

}

The original post has DP 2d array index from high to low, which is not quite intuitive.

Below is another 2D dp solution. Ideal is identical.

dp[i][j] denotes whether s[0....i-1] matches p[0.....j-1],

First, we need to initialize dp[i][0], i= [1,m]. All the dp[i][0] should be false because p has nothing in it.

Then, initialize dp[0][j], j = [1, n]. In this case, s has nothing, to get dp[0][j] = true, p must be '\*', '\*\*', '\*\*\*',etc. Once p.charAt(j-1) != '\*', all the dp[0][j] afterwards will be false.

Then start the typical DP loop.

Though this solution is clear and easy to understand. It is not good enough in the interview. it takes O(mn) time and O(mn) space.

Improvement: 1) optimize 2d dp to 1d dp, this will save space, reduce space complexity to O(N). 2) use iterative 2-pointer.

public boolean isMatch\_2d\_method(String s, String p) {

int m=s.length(), n=p.length();

boolean[][] dp = new boolean[m+1][n+1];

dp[0][0] = true;

for (int i=1; i<=m; i++) {

dp[i][0] = false;

}

for(int j=1; j<=n; j++) {

if(p.charAt(j-1)=='\*'){

dp[0][j] = true;

} else {

break;

}

}

for(int i=1; i<=m; i++) {

for(int j=1; j<=n; j++) {

if (p.charAt(j-1)!='\*') {

dp[i][j] = dp[i-1][j-1] && (s.charAt(i-1)==p.charAt(j-1) || p.charAt(j-1)=='?');

} else {

dp[i][j] = dp[i-1][j] || dp[i][j-1];

}

}

}

return dp[m][n];

}

At first I cannot pass the the long 'aaa...' test case. Then I add more check and pass it.

public class Solution {

public boolean isMatch(String s, String p) {

int m = s.length(), n = p.length();

int count = 0;

for (int i = 0; i < n; i++) {

if (p.charAt(i) == '\*') count++;

}

if (count==0 && m != n) return false;

else if (n - count > m) return false;

boolean[] match = new boolean[m+1];

match[0] = true;

for (int i = 0; i < m; i++) {

match[i+1] = false;

}

for (int i = 0; i < n; i++) {

if (p.charAt(i) == '\*') {

for (int j = 0; j < m; j++) {

match[j+1] = match[j] || match[j+1];

}

} else {

for (int j = m-1; j >= 0; j--) {

match[j+1] = (p.charAt(i) == '?' || p.charAt(i) == s.charAt(j)) && match[j];

}

match[0] = false;

}

}

return match[m];

}

}

窗体顶端

窗体底端

## 堆

### Kth Largest Element in an Array

[My Submissions](https://leetcode.com/problems/kth-largest-element-in-an-array/submissions/)

QuestionEditorial Solution

Total Accepted: **50743** Total Submissions: **154360** Difficulty: **Medium**

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, 1 ≤ k ≤ array's length.

建大小为K的小顶堆，之后的元素如果比堆顶大，则置换堆顶，堆顶元素做下移操作

public int findKthLargest(int[] nums, int k) {

int[] data=nums;

int len=data.length;

for(int i=(k-2)/2;i>=0;i--) //前k个元素建堆（因为lastIndex=k-1）

{

shift\_down(data,k-1,i);

//System.out.println(Arrays.toString(data));

}

for(int i=k;i<len;i++)

{

if(data[i]>data[0])

{

data[0]=data[i];

shift\_down(data,k-1,0);

//System.out.println(Arrays.toString(data));

}

}

return data[0];

}

public void shift\_down(int[] data, int lastIndex, int k){

//如果当前k节点的子节点存在

while(k\*2+1<=lastIndex)

{

int big=2\*k+1;

if(big<lastIndex)

{

if(data[big+1]<data[big]) big++;

}

//比较k节点与其左右子中的max值

if(data[k]>data[big])

{

swap(data,k,big);

k=big;//mark

}

else break;//直接跳出循环，否则，该节点还需要继续向下寻找

}

}

public void swap(int[] data,int i, int j)

{

int tmp=data[i];

data[i]=data[j];

data[j]=tmp;

}

堆排序（感觉下面的代码有问题，下次修改）

**堆的建立：**

**从空堆开始，对数组中的n个元素，连续使用insert操作插入堆中进行构造；（需要用另外一个数组来存放所建造的堆）**

**直接在数组中进行调整。从最后一片树叶，找到它上面的分支，从这个节点开始做下移，一直到根节点。**

**下移操作：使元素A[i]向下移动。下移过程中，将其值与两个儿子节点中较大的进行比较，如果小于，则交换，直到找到其合适的位置。**

**如果根是从0开始标记，则节点i的左子节点为2i+1，右子节点为2i+2**

**精简版代码**

**class HeapSort**

**{**

**public static void sort(int[] data)**

**{**

**int len=data.length;**

**for(int i=0;i<len-1;i++) //每一轮建堆，堆首是最大值，将其放到数组末尾**

**{**

**buildMaxHeap(data,len-1-i);**

**swap(data,0,len-1-i);**

**System.out.println(Arrays.toString(data));**

**}**

**}**

**public static void swap(int[] data,int i, int j)**

**{**

**int tmp=data[i];**

**data[i]=data[j];**

**data[j]=tmp;**

**}**

**public static void buildMaxHeap(int[] data, int lastIndex)**

**{**

**for(int i=(lastIndex-1)/2;i>=0;i--)//从树叶的父亲节点开始**

**{**

**int k=i;**

**//如果当前k节点的子节点存在**

**while(k\*2+1<=lastIndex) //shift\_down操作**

**{**

**int big=2\*k+1;**

**if(big<lastIndex)**

**{**

**if(data[big+1]>data[big]) big++;**

**}**

**//比较k节点与其左右子中的max值**

**if(data[k]<data[big])**

**{**

**swap(data,k,big);**

**k=big;//mark**

**}**

**else break;//直接跳出循环，否则，该节点还需要继续向下寻找**

**}**

**}**

**}**

**public static void main(String[] args)**

**{**

**int[] data={5,7,10,2,6,8};**

**sort(data);**

**}**

**}**

[**PriorityQueue**](file:///E:\docsJava\api\java\util\PriorityQueue.html#PriorityQueue-java.util.Comparator-)([**Comparator**](file:///E:\docsJava\api\java\util\Comparator.html)<? super [**E**](file:///E:\docsJava\api\java\util\PriorityQueue.html)> comparator)

Creates a PriorityQueue with the default initial capacity and whose elements are ordered according to the specified comparator.

|  |  |
| --- | --- |
| int | [**compare**](file:///E:\docsJava\api\java\util\Comparator.html#compare-T-T-)([**T**](file:///E:\docsJava\api\java\util\Comparator.html) o1, [**T**](file:///E:\docsJava\api\java\util\Comparator.html) o2)  Compares its two arguments for order. |

**Parameters:**

o1 - the first object to be compared.

o2 - the second object to be compared.

**Returns:**

a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

public int findKthLargest(int[] nums, int k) {

PriorityQueue<Integer> pq=new PriorityQueue<Integer>(new Comparator<Integer>()

{

public int compare(Integer a, Integer b)

{

return b-a;

}

});

for(int i=0;i<nums.length;i++)

{

pq.add(nums[i]);

}

for(int i=k-1;i>0;i--) pq.poll();

return pq.peek();

}

This problem is well known and quite often can be found in various text books.

You can take a couple of approaches to actually solve it:

* O(N lg N) running time + O(1) memory

The simplest approach is to sort the entire input array and then access the element by it's index (which is O(1)) operation:

**public** **int** findKthLargest(**int**[] nums, **int** k) {

**final** **int** N = nums.length;

Arrays.sort(nums);

**return** nums[N - k];

}

* O(N lg K) running time + O(K) memory

Other possibility is to use a min oriented priority queue that will store the K-th largest values. The algorithm iterates over the whole input and maintains the size of priority queue.

**public** **int** findKthLargest(**int**[] nums, **int** k) {

**final** PriorityQueue<Integer> pq = **new** PriorityQueue<>();

**for**(**int** val : nums) {

pq.offer(val);

**if**(pq.size() > k) {

pq.poll();

}

}

**return** pq.peek();

}

* O(N) best case / O(N^2) worst case running time + O(1) memory

The smart approach for this problem is to use the selection algorithm (based on the partion method - the same one as used in quicksort).

**public** **int** findKthLargest(**int**[] nums, **int** k) {

k = nums.length - k;

**int** lo = 0;

**int** hi = nums.length - 1;

**while** (lo < hi) {

**final** **int** j = partition(nums, lo, hi);

**if**(j < k) {

lo = j + 1;

} **else** **if** (j > k) {

hi = j - 1;

} **else** {

**break**;

}

}

**return** nums[k];

}

**private** **int** partition(**int**[] a, **int** lo, **int** hi) {

**int** i = lo;

**int** j = hi + 1;

**while**(**true**) {

**while**(i < hi && less(a[++i], a[lo]));

**while**(j > lo && less(a[lo], a[--j]));

**if**(i >= j) {

**break**;

}

exch(a, i, j);

}

exch(a, lo, j);

**return** j;

}

**private** **void** exch(**int**[] a, **int** i, **int** j) {

**final** **int** tmp = a[i];

a[i] = a[j];

a[j] = tmp;

}

**private** **boolean** less(**int** v, **int** w) {

**return** v < w;

}

O(N) guaranteed running time + O(1) space

So how can we improve the above solution and make it O(N) guaranteed? The answer is quite simple, we can randomize the input, so that even when the worst case input would be provided the algorithm wouldn't be affected. So all what it is needed to be done is to shuffle the input.

**public** **int** findKthLargest(**int**[] nums, **int** k) {

shuffle(nums);

k = nums.length - k;

**int** lo = 0;

**int** hi = nums.length - 1;

**while** (lo < hi) {

**final** **int** j = partition(nums, lo, hi);

**if**(j < k) {

lo = j + 1;

} **else** **if** (j > k) {

hi = j - 1;

} **else** {

**break**;

}

}

**return** nums[k];

}

**private** **void** shuffle(**int** a[]) {

**final** Random random = **new** Random();

**for**(**int** ind = 1; ind < a.length; ind++) {

**final** **int** r = random.nextInt(ind + 1);

exch(a, ind, r);

}

}

There is also worth mentioning the Blum-Floyd-Pratt-Rivest-Tarjan algorithm that has a guaranteed O(N) running time.

### Top K Frequent Elements

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/top-k-frequent-elements/submissions/)

* Total Accepted: **33858**
* Total Submissions: **76041**
* Difficulty: **Medium**

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.

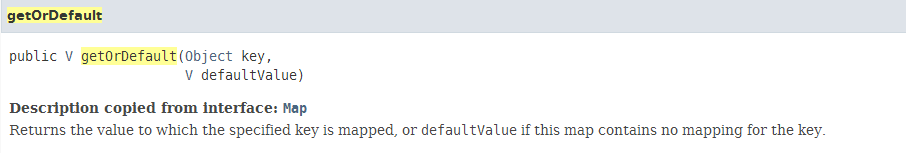
[Subscribe](https://leetcode.com/subscribe/) to see which companies asked this question

Hide Tags

[Hash Table](https://leetcode.com/tag/hash-table/) [Heap](https://leetcode.com/tag/heap/)

Hide Similar Problems

[(M) Word Frequency](https://leetcode.com/problems/word-frequency/) [(M) Kth Largest Element in an Array](https://leetcode.com/problems/kth-largest-element-in-an-array/)



这个方法好，桶排序。

首先对key计数；

按照计数，划分桶；

**Java O(n) Solution - Bucket Sort**

Idea is simple. Build a array of list to be buckets with length 1 to sort.

**public** List<Integer> **topKFrequent**(**int**[] nums, **int** k) {

List<Integer>[] bucket = **new** List[nums.length + 1];

Map<Integer, Integer> frequencyMap = **new** HashMap<Integer, Integer>();

**for** (**int** n : nums) {

frequencyMap.put(n, frequencyMap.getOrDefault(n, 0) + 1);

}

**for** (**int** key : frequencyMap.keySet()) {

**int** frequency = frequencyMap.**get**(key);

**if** (bucket[frequency] == null) {

bucket[frequency] = **new** ArrayList<>();

}

bucket[frequency].add(key);

}

List<Integer> res = **new** ArrayList<>();

**for** (**int** pos = bucket.length - 1; pos >= 0 && res.size() < k; pos--) {

**if** (bucket[pos] != null) {

res.addAll(bucket[pos]);

}

}

**return** res;

}

Nice solution, I initially thought about using PriorityQueue, but build PriorityQueue needs n(logn) time.

Thanks for sharing, only one nitpick:

Think about the case when K=2,  
and you have 1 number that has max frequency, say 10 times.  
and you have 10 numbers that has 2nd max frequency, say 9 times.  
With your algo, the returned list will contain 11 numbers instead of 2.

Any easy fix:  
return res.subList(0,k);

(It seems the above scenario is not covered by the existing test cases.)

**3 ways to solve this problem**

using heap

**class** Solution {

**public**:

vector<**int**> topKFrequent(vector<**int**>& nums, **int** k) {

priority\_queue<pair<**int**, **int**>, vector<pair<**int**, **int**>>, greater<pair<**int**, **int**>>> pq;

unordered\_map<**int**, **int**> cnt;

**for** (**auto** num : nums) cnt[num]++;

**for** (**auto** kv : cnt) {

pq.push({kv.second, kv.first});

**while** (pq.size() > k) pq.pop();

}

vector<**int**> res;

**while** (!pq.empty()) {

res.push\_back(pq.top().second);

pq.pop();

}

**return** res;

}

};

using selection algorithm

**class** Solution {

**public**:

vector<**int**> topKFrequent(vector<**int**>& nums, **int** k) {

vector<**int**> res;

**if** (!nums.size()) **return** res;

unordered\_map<**int**, **int**> cnt;

**for** (**auto** num : nums) cnt[num]++;

vector<pair<**int**, **int**>> num\_with\_cnt;

**for** (**auto** kv : cnt) {

num\_with\_cnt.push\_back({kv.first, kv.second});

}

kselection(num\_with\_cnt, 0, num\_with\_cnt.size()-1, k);

**for** (**int** i = 0; i < k && i < num\_with\_cnt.size(); ++i) {

res.push\_back(num\_with\_cnt[i].first);

}

**return** res;

}

**void** **kselection**(vector<pair<**int**, **int**>>& data, **int** start, **int** end, **int** k) {

**if** (start >= end) **return**;

**auto** pv = data[end];

**int** i = start;

**int** j = start;

**while** (i < end) {

**if** (data[i].second > pv.second) {

swap(data[i++], data[j++]);

} **else** {

++i;

}

}

swap(data[j], data[end]);

**int** num = j - start + 1;

**if** (num == k) **return**;

**else** **if** (num < k) {

kselection(data, j + 1, end, k - num);

} **else** {

kselection(data, start, j - 1, k);

}

}

};

using bucket sort

**class** Solution {

**public**:

vector<**int**> topKFrequent(vector<**int**>& nums, **int** k) {

vector<**int**> res;

**if** (!nums.size()) **return** res;

unordered\_map<**int**, **int**> cnt;

**for** (**auto** num : nums) cnt[num]++;

vector<vector<**int**>> bucket(nums.size() + 1);

**for** (**auto** kv : cnt) {

bucket[kv.second].push\_back(kv.first);

}

**for** (**int** i = bucket.size() - 1; i >= 0; --i) {

**for** (**int** j = 0; j < bucket[i].size(); ++j){

res.push\_back(bucket[i][j]);

**if** (res.size() == k) **return** res;

}

}

**return** res;

}

};

**3 Java Solution using Array, MaxHeap, TreeMap**

*// use an array to save numbers into different bucket whose index is the frequency*

**public** **class** Solution {

**public** List<Integer> topKFrequent(**int**[] nums, **int** k) {

Map<Integer, Integer> map = **new** HashMap<>();

**for**(**int** n: nums){

map.put(n, map.getOrDefault(n,0)+1);

}

*// corner case: if there is only one number in nums, we need the bucket has index 1.*

List<Integer>[] bucket = **new** List[nums.length+1];

**for**(**int** n:map.keySet()){

**int** freq = map.get(n);

**if**(bucket[freq]==null)

bucket[freq] = **new** LinkedList<>();

bucket[freq].add(n);

}

List<Integer> res = **new** LinkedList<>();

**for**(**int** i=bucket.length-1; i>0 && k>0; --i){

**if**(bucket[i]!=null){

List<Integer> list = bucket[i];

res.addAll(list);

k-= list.size();

}

}

**return** res;

}

}

*// use maxHeap. Put entry into maxHeap so we can always poll a number with largest frequency*

**public** **class** Solution {

**public** List<Integer> topKFrequent(**int**[] nums, **int** k) {

Map<Integer, Integer> map = **new** HashMap<>();

**for**(**int** n: nums){

map.put(n, map.getOrDefault(n,0)+1);

}

PriorityQueue<Map.Entry<Integer, Integer>> maxHeap =

**new** PriorityQueue<>((a,b)->(b.getValue()-a.getValue()));

**for**(Map.Entry<Integer,Integer> entry: map.entrySet()){

maxHeap.add(entry);

}

List<Integer> res = **new** ArrayList<>();

**while**(res.size()<k){

Map.Entry<Integer, Integer> entry = maxHeap.poll();

res.add(entry.getKey());

}

**return** res;

}

}

*// use treeMap. Use freqncy as the key so we can get all freqencies in order*

**public** **class** Solution {

**public** List<Integer> topKFrequent(**int**[] nums, **int** k) {

Map<Integer, Integer> map = **new** HashMap<>();

**for**(**int** n: nums){

map.put(n, map.getOrDefault(n,0)+1);

}

TreeMap<Integer, List<Integer>> freqMap = **new** TreeMap<>();

**for**(**int** num : map.keySet()){

**int** freq = map.get(num);

**if**(!freqMap.containsKey(freq)){

freqMap.put(freq, **new** LinkedList<>());

}

freqMap.get(freq).add(num);

}

List<Integer> res = **new** ArrayList<>();

**while**(res.size()<k){

Map.Entry<Integer, List<Integer>> entry = freqMap.pollLastEntry();

res.addAll(entry.getValue());

}

**return** res;

}

}

## 分而治之

## HashMap

窗体顶端

窗体底端

### Valid Anagram

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/valid-anagram/submissions/)

* Total Accepted: **113742**
* Total Submissions: **258815**
* Difficulty: **Easy**

Given two strings *s* and *t*, write a function to determine if *t* is an anagram 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.

**Follow up:**  
What if the inputs contain unicode characters? How would you adapt your solution to such case?

我的做法同书

public boolean isAnagram(String s, String t) {

if(s.length()!=t.length()) return false;

int[] map=new int[256];

for(int i=0; i<s.length(); i++)

{

char ch=s.charAt(i);

map[ch]++;

}

for(int i=0; i<t.length(); i++)

{

char ch=t.charAt(i);

if(--map[ch]<0) return false;

}

return true;

}

### Two Sum

[My Submissions](https://leetcode.com/problems/two-sum/submissions/)

QuestionEditorial Solution

Total Accepted: **218619** Total Submissions: **957295** Difficulty: **Easy**

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

You may assume that each input would have ***exactly*** one solution.

**Example:**

Given nums = [2, 7, 11, 15], target = 9,

Because nums[**0**] + nums[**1**] = 2 + 7 = 9,

return [**0**, **1**].

我的做法

public int[] twoSum(int[] nums, int target) {

HashMap<Integer,Integer> map=new HashMap<Integer,Integer>();

int[] res=new int[2];

for(int i=0;i<nums.length;i++)

{

if(map.containsKey(target-nums[i]))

{

res[0]=map.get(target-nums[i]);

res[1]=i;

}

else

{

map.put(nums[i],i);

}

}

return res;

}

Hi, this is my accepted JAVA solution. It only go through the list once. It's shorter and easier to understand. Hope this can help someone. Please tell me if you know how to make this better :)

**public** **int**[] twoSum(**int**[] numbers, **int** target) {

**int**[] result = **new** **int**[2];

Map<Integer, Integer> **map** = **new** HashMap<Integer, Integer>();

**for** (**int** i = 0; i < numbers.length; i++) {

**if** (**map**.containsKey(target - numbers[i])) {

result[1] = i + 1;

result[0] = **map**.get(target - numbers[i]);

**return** result;

}

**map**.put(numbers[i], i + 1);

}

**return** result;

}

### Fraction to Recurring Decimal

[My Submissions](https://leetcode.com/problems/fraction-to-recurring-decimal/submissions/)

QuestionEditorial Solution

Total Accepted: **29975** Total Submissions: **198258** Difficulty: **Medium**

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

For example,

* Given numerator = 1, denominator = 2, return "0.5".
* Given numerator = 2, denominator = 1, return "2".
* Given numerator = 2, denominator = 3, return "0.(6)".

**Hint:**

1. No scary math, just apply elementary math knowledge. Still remember how to perform a *long division*?
2. Try a long division on 4/9, the repeating part is obvious. Now try 4/333. Do you see a pattern?
3. Be wary of edge cases! List out as many test cases as you can think of and test your code thoroughly.

用HashMap存余数

The important thing is to consider all edge cases while thinking this problem through, including: negative integer, possible overflow, etc.

Use HashMap to store a remainder and its associated index while doing the division so that whenever a same remainder comes up, we know there is a repeating fractional part.

**public** **class** Solution {

**public** String fractionToDecimal(**int** numerator, **int** denominator) {

**if** (numerator == 0) {

**return** "0";

}

StringBuilder res = **new** StringBuilder();

// "+" or "-"

res.append(((numerator > 0) ^ (denominator > 0)) ? "-" : "");

**long** num = Math.abs((**long**)numerator);

**long** den = Math.abs((**long**)denominator);

// integral part

res.append(num / den);

num %= den;

**if** (num == 0) {

**return** res.toString();

}

// fractional part

res.append(".");

HashMap<Long, Integer> **map** = **new** HashMap<Long, Integer>();

**map**.put(num, res.length());

**while** (num != 0) {

num \*= 10;

res.append(num / den);

num %= den;

**if** (**map**.containsKey(num)) {

**int** index = **map**.get(num);

res.insert(index, "(");

res.append(")");

**break**;

}

**else** {

**map**.put(num, res.length());

}

}

**return** res.toString();

}

}

public String fractionToDecimal(int numerator, int denominator) {

StringBuilder result = new StringBuilder();

String sign = (numerator < 0 == denominator < 0 || numerator == 0) ? "" : "-";

long num = Math.abs((long) numerator);

long den = Math.abs((long) denominator);

result.append(sign);

result.append(num / den);

long remainder = num % den;

**if** (remainder == 0)

**return** result.toString();

result.append(".");

HashMap<Long, Integer> hashMap = new HashMap<Long, Integer>();

**while** (!hashMap.containsKey(remainder)) {

hashMap.put(remainder, result.length());

result.append(10 \* remainder / den);

remainder = 10 \* remainder % den;

}

int index = hashMap.get(remainder);

result.insert(index, "(");

result.append(")");

**return** result.toString().replace("(0)", "");

}

【官方Solution】

0.16

6 ) 1.00

0

1 0 <-- Remainder=1, mark 1 as seen at position=0.

- 6

40 <-- Remainder=4, mark 4 as seen at position=1.

- 36

4 <-- Remainder=4 was seen before at position=1, so the fractional part which is 16 starts repeating at position=1 => 1(6).

The key insight here is to notice that once the remainder starts repeating, so does the divided result.

You will need a hash table that maps from the remainder to its position of the fractional part. Once you found a repeating remainder, you may enclose the reoccurring fractional part with parentheses by consulting the position from the table.

The remainder could be zero while doing the division. That means there is no repeating fractional part and you should stop right away.

Just like the question [Divide Two Integers](https://oj.leetcode.com/problems/divide-two-integers/), be wary of edge case such as negative fractions and nasty extreme case such as -2147483648 / -1.

【中文解析】

**难点**：如何识别循环体？

**解决方法**：用一个HashMap记录每一个余数，当出现重复的余数时，那么将会进入循环，两个重复余数之间的部分就是循环体。

**示例**：1/13=0.076923076923076923...，当小数部分第二次出现0时，就意味着开始了循环，那么需要把076923用括号括起来，结果为0.(076923)。

**涉及技巧**：1）在不断相除的过程中，把余数乘以10再进行下一次相除，保证一直是整数相除；2）HashMap的key和value分别是<当前余数, 对应结果下标>，这样获取076923时就可根据value值来找。

注意点1：考虑正负数，先判断符号，然后都转化为正数；

注意点2：考虑溢出，如果输入为Integer.MIN\_VALUE，取绝对值后会溢出。

这个写法不错

ans一直在尾部加上rem/dem的值，rem一直在更新。直到rem出现重复，则添加括号，输出最终结果。

代码中已添加注释，来看代码：

**[java]** [view plaincopy](http://blog.csdn.net/ljiabin/article/details/42025037)

1. **public** **class** Solution {
2. **public** String fractionToDecimal(**int** numerator, **int** denominator) {
3. **if** (numerator == 0) **return** "0";
4. **if** (denominator == 0) **return** "";
6. String ans = "";
8. //如果结果为负数
9. **if** ((numerator < 0) ^ (denominator < 0)) {  //逻辑异或
10. ans += "-";
11. }
13. //下面要把两个数都转为正数，为避免溢出，int转为long
14. **long** num = numerator, den = denominator;
15. num = Math.abs(num);
16. den = Math.abs(den);
18. //结果的整数部分
19. **long** res = num / den;
20. ans += String.valueOf(res);
22. //如果能够整除，返回结果
23. **long** rem = (num % den) \* 10;
24. **if** (rem == 0) **return** ans;
26. //结果的小数部分
27. HashMap<Long, Integer> map = **new** HashMap<Long, Integer>();
28. ans += ".";
29. **while** (rem != 0) {
30. //如果前面已经出现过该余数，那么将会开始循环
31. **if** (map.containsKey(rem)) {
32. **int** beg = map.get(rem); //循环体开始的位置
33. String part1 = ans.substring(0, beg);
34. String part2 = ans.substring(beg, ans.length());
35. ans = part1 + "(" + part2 + ")";
36. **return** ans;
37. }
39. //继续往下除
40. map.put(rem, ans.length());
41. res = rem / den;
42. ans += String.valueOf(res);
43. rem = (rem % den) \* 10;
44. }
46. **return** ans;
47. }
48. }

**Java编程误区**：一定要先把 int 转为 long，然后再取绝对值。如果写成 long num = Math.abs(numerator) 就会有问题，因为这句代码在 numerator=Integer.MIN\_VALUE 时相当于 long num = Math.abs(-2147483648)，这样得到的 num还是 -2147483648。

### Happy Number

[My Submissions](https://leetcode.com/problems/happy-number/submissions/)

QuestionEditorial Solution

Total Accepted: **66950** Total Submissions: **183169** Difficulty: **Easy**

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

我的做法

这题和求循环小数类似，如果不是happy number，sum在几轮计算后会出现重复

这个写法清晰

public boolean isHappy(int n) {

HashMap<Integer,Integer> map=new HashMap<>();

int sum=0;

while(sum!=1)

{

sum=0;

while(n!=0)

{

sum+=(n%10)\*(n%10);

n/=10;

}

n=sum;

if(map.containsKey(sum))

{

return false;

}

map.put(sum,1);

}

return true;

}

I see the majority of those posts use hashset to record values. Actually, we can simply adapt the Floyd Cycle detection algorithm. I believe that many people have seen this in the Linked List Cycle detection problem. The following is my code:

**int** digitSquareSum(**int** n) {

**int** sum = 0, tmp;

**while** (n) {

tmp = n % 10;

sum += tmp \* tmp;

n /= 10;

}

**return** sum;

}

**bool** isHappy(**int** n) {

**int** slow, fast;

slow = fast = n;

**do** {

slow = digitSquareSum(slow);

fast = digitSquareSum(fast);

fast = digitSquareSum(fast);

} **while**(slow != fast);

**if** (slow == 1) **return** 1;

**else** **return** 0;

}

**public** **class** **Solution** {

**public** **boolean** isHappy(**int** n) {

**int** x = n;

**int** y = n;

**while**(x>1){

x = cal(x) ;

**if**(x==1) **return** **true** ;

y = cal(cal(y));

**if**(y==1) **return** **true** ;

**if**(x==y) **return** **false**;

}

**return** **true** ;

}

**public** **int** cal(**int** n){

**int** x = n;

**int** s = 0;

**while**(x>0){

s = s+(x%10)\*(x%10);

x = x/10;

}

**return** s ;

}

}

Beat 90% Fast Easy Understand Java Solution with Brief Explanation

The idea is to use one hash set to record sum of every digit square of every number occurred. Once the current sum cannot be added to set, return false; once the current sum equals 1, return true;

**public** **boolean** isHappy(**int** n) {

Set<Integer> inLoop = **new** HashSet<Integer>();

**int** squareSum,remain;

**while** (inLoop.add(n)) {

squareSum = 0;

**while** (n > 0) {

remain = n%10;

squareSum += remain\*remain;

n /= 10;

}

**if** (squareSum == 1)

**return** **true**;

**else**

n = squareSum;

}

**return** **false**;

}

My Java solution: find 1 or 7 when happy sum is a single digit

In order to find a rule to break out the loop, I start calculating 2 and find a loop at 4, then 3,5,6,9 will all go into that loop. So in 1-9, only 1 and 7 are happy numbers. Also I find all numbers' calculation will goes into a single digit at some time. So what I did is just calculate happy sum and when it is a single digit, check if it is 1 or 7 ^.^

public boolean isHappy(int n) {

**if**(n<=0) **return** **false**;

**while**(**true**){

int sum=0;

**while**(n!=0){

sum+=(n%10)\*(n%10);

n=n/10;

}

**if**(sum/10==0){

**if**(sum==1||sum==7) **return** **true**;

**else** **return** **false**;

}

n=sum;

}

}

Proof to "all numbers' calculation will goes into a single digit at some time". I post this as an answer to have a better format than comment.

The happy sum of a N digit number, Happy(num), will not be larger than Happy(10^(N+1) - 1) = 81N.

Therefore when N > 3, we always have 81N < 100N < (10^2) \* N < 10^N, that is Happy(num) < num / 10, which will eventually reduce the happy sum of any number to smaller or equal to 999, because Happy(9999) < 999.

Happy(999) = 243, therefore any number larger than 99 and smaller or equal to 999 should have a happy sum smaller or equal to 243, which in turn smaller than 299, which should have the largest happy sum among all numbers in [200, 300).

Happy(299) = 166, therefore Happy(num) will get to a number smaller or equal to 166 at some step, which in turn smaller than 199.

1. Happy(199) = 163 < 199.
2. Happy(99) = 162.

Combining 1 and 2 above, we know any numbers larger than 100 will be reduced to smaller than 163 at some step, and any numbers smaller than 100 have a happy sum that is smaller or equal to 162, therefore all numbers will be reduced to smaller or equal to 162 at some step.

Now we only have 162 numbers to deal with, you can simply write a program to verify that their happy sum all get to a single digit at some step.

"Therefore when N > 3, we always have 81N < 100N < (10^2) \* N < 10^N, that is Happy(num) < num / 10, " don't quite understand. think the formula is : Happy(num)<10^N<10^N-1(since we have more than two strictly less than relation whiles.) so Happy(999999)<99999, Happy(99999)<Happy(9999) Happy(9999)<999, so on and so forth. so the largest sum converge to less than 999.

Explanation of why those posted algorithms are mathematically valid

Earlier posts gave the algorithm but did not explain why it is valid mathematically, and this is what this post is about: present a "short" mathematical proof.

First of all, it is easy to argue that starting from a number I, if some value - say a - appears again during the process after k steps, the initial number I cannot be a happy number. Because a will continuously become a after every k steps.

Therefore, as long as we can show that there is a loop after running the process continuously, the number is not a happy number.

There is another detail not clarified yet: For any non-happy number, will it definitely end up with a loop during the process? This is important, because it is possible for a non-happy number to follow the process endlessly while having no loop.

To show that a non-happy number will definitely generate a loop, we only need to show that for any non-happy number, all outcomes during the process are bounded by some large but finite integer N. If all outcomes can only be in a finite set (2,N], and since there are infinitely many outcomes for a non-happy number, there has to be at least one duplicate, meaning a loop!

Suppose after a couple of processes, we end up with a large outcome O1 with D digits where Dis kind of large, say D>=4, i.e., O1 > 999 (If we cannot even reach such a large outcome, it means all outcomes are bounded by 999 ==> loop exists). We can easily see that after processing O1, the new outcome O2 can be at most 9^2\*D < 100D, meaning that O2 can have at most 2+d(D) digits, where d(D) is the number of digits D have. It is obvious that 2+d(D) < D. We can further argue that O1 is the maximum (or boundary) of all outcomes afterwards. This can be shown by contradictory: Suppose after some steps, we reach another large number O3 > O1. This means we process on some number W <= 999 that yields O3. However, this cannot happen because the outcome of W can be at most 9^2\*3 < 300 < O1.

Done.

Please leave your comment if any question or suggestion.

### Group Anagrams

[My Submissions](https://leetcode.com/problems/anagrams/submissions/)

QuestionEditorial Solution

Total Accepted: **74086** Total Submissions: **269143** Difficulty: **Medium**

Given an array of strings, group anagrams(由颠倒字母顺序而构成的字[短语]) together.

For example, given: ["eat", "tea", "tan", "ate", "nat", "bat"],   
Return:

[

["ate", "eat","tea"],

["nat","tan"],

["bat"]

]

**Note:**

1. For the return value, each *inner* list's elements must follow the lexicographic order.
2. All inputs will be in lower-case.

我的做法

使用HashMap，其变位词中字母序第一个为key，变位词为value

最后list加入结果时，对从map中取出来的list按照字典序进行排序

public List<List<String>> groupAnagrams(String[] strs) {

List<List<String>> res=new ArrayList<List<String>>();

HashMap<String,List<String>> map=new HashMap<>();

for(String s:strs)

{

char[] ss=s.toCharArray();

Arrays.sort(ss);

String tmp=new String(ss);

if(map.containsKey(tmp))

{

map.get(tmp).add(s);

}

else

{

List<String> ls=new ArrayList<String>();

ls.add(s);

map.put(tmp,ls);

}

}

for(String s:map.keySet())

{

List<String> ls=map.get(s);

ls.sort(new Comparator<String>()

{

public int compare(String a, String b)

{

return a.compareTo(b);

}

});

res.add(ls);

}

return res;

}

Small suggestion. I found out that its much faster to sort individual hashmap buckets than to sort input array whose size is very large.. Earlier i had not expected this.

Agree with @sreenidhi2. So I modified the code a little bit: instead of sort the string array at the beginning, I sorted the list of each hashmap bucket at the last step.

这个写法不错，比我的更加简洁，思路一样

public List<List<String>> groupAnagrams(String[] strs) {

**if**(strs==null || strs.length == 0){

**return** new ArrayList<List<String>>();

}

HashMap<String, List<String>> map = new HashMap<String, List<String>>();

//Arrays.sort(strs);

**for** (String s:strs) {

char[] ca = s.toCharArray();

Arrays.sort(ca);

String keyStr = String.valueOf(ca);

**if**(!map.containsKey(keyStr))

map.put(keyStr, new ArrayList<String>());

map.get(keyStr).add(s);

}

**for**(String key: map.keySet()) {

Collections.sort(map.get(key));//重要

}

**return** new ArrayList<List<String>>(map.values());

}

Just a small improvement:

As all inputs will be in lower-case, we can implement our own sorting algorithm to sort a string in order to speed up this sorting. I implemented one similar to counting sort as follows, and the running time was decreased to 68ms.

**string** sortLowercase(**string** s) {

**int** charExist[26] = {0};

**for** (**int** i = 0; i < s.size(); i++) {

charExist[s[i] - 'a']++;

}

**string** res;

**int** j = 0;

**while** (j < 26) {

**if** (charExist[j] == 0) {

j++;

}

**else** {

res.push\_back(j + 'a');

charExist[j]--;

}

}

**return** res;

}

### Word Pattern

[My Submissions](https://leetcode.com/problems/word-pattern/submissions/)

QuestionEditorial Solution

Total Accepted: **34765** Total Submissions: **119450** Difficulty: **Easy**

Given a pattern 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 by a single space.

注意两个地方，str中单词个数，需要与pattern中字符个数进行匹配；字符到单词的映射与单词到字符的映射是双向的，即a->dog,b->dog，不成立。

我的做法

这个方法比较简单清晰，也可以用两个map

public boolean wordPattern(String pattern, String str) {

String[] strs=str.split("\\s");

if(pattern.length()!=strs.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(strs[i])) return false;

map.put(pattern.charAt(i),strs[i]);

}

else

{

String s=map.get(pattern.charAt(i));

if(!s.equals(strs[i])) return false;

}

}

return true;

}

public boolean wordPattern(String pattern, String str) {

String[] words = str.split(" ");

**if** (words.length != pattern.length())

**return** **false**;

Map index = new HashMap();

**for** (Integer i=0; i<words.length; ++i)

**if** (index.put(pattern.charAt(i), i) != index.put(words[i], i))

**return** **false**;

**return** **true**;

}

I go through the pattern letters and words in parallel and compare the indexes where they last appeared.

**Edit 1:** Originally I compared the **first** indexes where they appeared, using putIfAbsent instead of put. That was based on [mathsam's solution](https://leetcode.com/discuss/36438/1-liner-in-python?show=39066#a39066) for the old [Isomorphic Strings](https://leetcode.com/problems/isomorphic-strings/) problem. But then[czonzhu's answer](https://leetcode.com/discuss/62374/9-lines-simple-java?show=62383#a62383) below made me realize that put works as well and why.

**Edit 2:** Switched from

**for** (int i=0; i<words.length; ++i)

**if** (!Objects.equals(index.put(pattern.charAt(i), i),

index.put(words[i], i)))

**return** **false**;

to the current version with i being an Integer object, which allows to compare with just !=because there's no autoboxing-same-value-to-different-objects-problem anymore. Thanks to lap\_218 for somewhat pointing that out in the comments.

It is not a good idea to save pattern and words in one map both as keys. The str can contains words like "a", "b", or "c", which will mix with the same characters in pattern. I don't think this solution will work for the input like "abb", "dog a a". Using two maps will easily solve this bug.

@bill.quan Not a bug, it works, and this has already been discussed in previous comments.

public **class** **Solution** {

public boolean wordPattern(String pattern, String str) {

String[] arr= str.split(" ");

HashMap<Character, String> map = new HashMap<Character, String>();

**if**(arr.length!= pattern.length())

**return** **false**;

**for**(int i=0; i<arr.length; i++){

char c = pattern.charAt(i);

**if**(map.containsKey(c)){

**if**(!map.get(c).equals(arr[i]))

**return** **false**;

}**else**{

**if**(map.containsValue(arr[i]))

**return** **false**;

map.put(c, arr[i]);

}

}

**return** **true**;

}

}

这个方法不错

To avoid map.containsValue(arr[i]), you can add an extra map to save <String, Character>, my code as below:

public class Solution { public boolean wordPattern(String pattern, String str) {

String[] strs = str.split(" ");

**if**(pattern.length() != strs.length) **return** **false**;

HashMap<Character, String> hm1 = new HashMap<Character, String>();

HashMap<String, Character> hm2 = new HashMap<String, Character>();

**for**(int i=0; i<pattern.length(); ++i) {

**if**(hm1.containsKey(pattern.charAt(i))) {

**if**(!hm1.get(pattern.charAt(i)).equals(strs[i])) **return** **false**;

}

**else** {

**if**(hm2.containsKey(strs[i])) **return** **false**; //如果该key之前没有出现过，那么该key对应的strs[i]也不该在hm2中出现

**else** {

hm1.put(pattern.charAt(i), strs[i]);

hm2.put(strs[i], pattern.charAt(i));

}

}

}

**return** **true**;

}

}

### Isomorphic Strings

[My Submissions](https://leetcode.com/problems/isomorphic-strings/submissions/)

QuestionEditorial Solution

Total Accepted: **55493** Total Submissions: **187276** Difficulty: **Easy**

Given two strings ***s*** and ***t***, determine if they are isomorphic.

Two strings are isomorphic if the characters in ***s*** can be replaced to get ***t***.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may 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.//不是一一互相对应

**Note:**  
You may assume both ***s*** and ***t*** have the same length.

我的做法，同上一题的两个map的做法

public boolean isIsomorphic(String s, String t) {

HashMap<Character,Character> map1=new HashMap<>();

HashMap<Character,Character> map2=new HashMap<>();

if(s.length()!=t.length()) return false;

for(int i=0;i<s.length();i++)

{

if(map1.containsKey(s.charAt(i)))

{

if(map1.get(s.charAt(i))!=t.charAt(i)) return false;

}

else

{

if(map2.containsKey(t.charAt(i))) return false;

else

{

map1.put(s.charAt(i),t.charAt(i));

map2.put(t.charAt(i),s.charAt(i));

}

}

}

return true;

}

**class** Solution {

**public**:

**bool** isIsomorphic(**string** s, **string** t) {

**int** m1[256] = {0}, m2[256] = {0}, n = s.size();

**for** (**int** i = 0; i < n; ++i) {

**if** (m1[s[i]] != m2[t[i]]) **return** **false**;

m1[s[i]] = i + 1;

m2[t[i]] = i + 1;

}

**return** **true**;

}

};

Hi guys!

The main idea is to store the last seen positions of current (i-th) characters in both strings. If previously stored positions are different then we know that the fact they're occuring in the current i-th position simultaneously is a mistake. We could use a map for storing but as we deal with chars which are basically ints and can be used as indices we can do the whole thing with an array.

Check the code below. Happy coding!

**public** **class** **Solution** {

**public** **boolean** isIsomorphic(String s1, String s2) {

**int**[] m = **new** **int**[512];

**for** (**int** i = 0; i < s1.length(); i++) {

**if** (m[s1.charAt(i)] != m[s2.charAt(i)+256]) **return** **false**;

m[s1.charAt(i)] = m[s2.charAt(i)+256] = i+1;

}

**return** **true**;

}

}

这个写法不错，类似于用两个hashmap

使用两个数组，数组初值为0，如果key和value出现过，则对应map的值不为0

重复key出现时，找之前key出现对应的位置及因为对应value也会设置成相等值；

Key与value必须都没有被设置过，否则就相等才行

Bug with OJ Running Time? My solution beat 98%

**public** **class** **Solution** {

**public** **boolean** isIsomorphic(String s, String t) {

//for(int i = 0; i < Integer.MAX\_VALUE;i ++);

**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**;

}

}

(Edit out the for loop to see for yourself!) For many problems now, redundant for loop will magically speed up the running time.

I found this out because some of the faster solution seem to do more redundant stuff than the top solution which is concise and neater. Both of those solution would follow the same logic. So I just tested out by introducing for loops and found this out.

### Intersection of Two Arrays

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/intersection-of-two-arrays/submissions/)

* Total Accepted: **35295**
* Total Submissions: **79496**
* Difficulty: **Easy**

Given two arrays, write a function to compute their intersection.

**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.

[Subscribe](https://leetcode.com/subscribe/) to see which companies asked this question

[Binary Search](https://leetcode.com/tag/binary-search/) [Hash Table](https://leetcode.com/tag/hash-table/) [Two Pointers](https://leetcode.com/tag/two-pointers/) [Sort](https://leetcode.com/tag/sort/)

**这三个方法都不错Three Java Solutions**

**Three Java Solutions**

Use two hash sets

Time complexity: O(n)

**public** **class** **Solution** {

**public** **int**[] **intersection**(**int**[] nums1, **int**[] nums2) {

Set<Integer> **set** = **new** HashSet<>();

Set<Integer> intersect = **new** HashSet<>();

**for** (**int** i = 0; i < nums1.length; i++) {

**set**.add(nums1[i]);

}

**for** (**int** i = 0; i < nums2.length; i++) {

**if** (**set**.contains(nums2[i])) {

intersect.add(nums2[i]);

}

}

**int**[] result = **new** **int**[intersect.size()];//输出方法

**int** i = 0;

**for** (Integer num : intersect) {

result[i++] = num;

}

**return** result;

}

}

Sort both arrays, use two pointers

Time complexity: O(nlogn)

**public** **class** **Solution** {

**public** **int**[] **intersection**(**int**[] nums1, **int**[] nums2) {

Set<Integer> **set** = **new** HashSet<>();

Arrays.sort(nums1);

Arrays.sort(nums2);

**int** i = 0;

**int** j = 0;

**while** (i < nums1.length && j < nums2.length) {

**if** (nums1[i] < nums2[j]) {

i++;

} **else** **if** (nums1[i] > nums2[j]) {

j++;

} **else** {

**set**.add(nums1[i]);

i++;

j++;

}

}

**int**[] result = **new** **int**[**set**.size()];

**int** k = 0;

**for** (Integer num : **set**) {

result[k++] = num;

}

**return** result;

}

}

Binary search

Time complexity: O(nlogn)

**public** **class** **Solution** {

**public** **int**[] **intersection**(**int**[] nums1, **int**[] nums2) {

Set<Integer> **set** = **new** HashSet<>();

Arrays.sort(nums2);

**for** (Integer num : nums1) {

**if** (binarySearch(nums2, num)) {

**set**.add(num);

}

}

**int** i = 0;

**int**[] result = **new** **int**[**set**.size()];

**for** (Integer num : **set**) {

result[i++] = num;

}

**return** result;

}

**public** boolean **binarySearch**(**int**[] nums, **int** target) {

**int** low = 0;

**int** high = nums.length - 1;

**while** (low <= high) {

**int** mid = low + (high - low) / 2;

**if** (nums[mid] == target) {

**return** true;

}

**if** (nums[mid] > target) {

high = mid - 1;

} **else** {

low = mid + 1;

}

}

**return** false;

}

}

### Intersection of Two Arrays II

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/intersection-of-two-arrays-ii/submissions/)

* Total Accepted: **24782**
* Total Submissions: **59291**
* Difficulty: **Easy**

Given two arrays, write a function to compute their intersection.

**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 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?

**C++ hash table solution and sort + two pointers solution with time and space complexity**

m: nums1.size n: nums2.size

Hash table solution:  
Time: O(m + n) Space: O(m + n)

**class** Solution {

**public**:

vector<**int**> intersect(vector<**int**>& nums1, vector<**int**>& nums2) {

unordered\_map<**int**, **int**> dict;

vector<**int**> res;

**for**(**int** i = 0; i < (**int**)nums1.size(); i++) dict[nums1[i]]++;

**for**(**int** i = 0; i < (**int**)nums2.size(); i++)

**if**(--dict[nums2[i]] >= 0) res.push\_back(nums2[i]);

**return** res;

}

};

Hash table solution2:  
Time: O(m + n) Space: O(m)

**class** Solution {

**public**:

vector<**int**> intersect(vector<**int**>& nums1, vector<**int**>& nums2) {

unordered\_map<**int**, **int**> dict;

vector<**int**> res;

**for**(**int** i = 0; i < (**int**)nums1.size(); i++) dict[nums1[i]]++;

**for**(**int** i = 0; i < (**int**)nums2.size(); i++)

**if**(dict.find(nums2[i]) != dict.end() && --dict[nums2[i]] >= 0) res.push\_back(nums2[i]);

**return** res;

}

};

Sort and two pointers Solution:  
Time: O(max(m, n) log(max(m, n))) Space: O(m + n)

**class** Solution {

**public**:

vector<**int**> intersect(vector<**int**>& nums1, vector<**int**>& nums2) {

sort(nums1.begin(), nums1.end());

sort(nums2.begin(), nums2.end());

**int** n1 = (**int**)nums1.size(), n2 = (**int**)nums2.size();

**int** i1 = 0, i2 = 0;

vector<**int**> res;

**while**(i1 < n1 && i2 < n2){

**if**(nums1[i1] == nums2[i2]) {

res.push\_back(nums1[i1]);

i1++;

i2++;

}

**else** **if**(nums1[i1] > nums2[i2]){

i2++;

}

**else**{

i1++;

}

}

**return** res;

}

};

**AC solution using Java HashMap**

**public** **class** Solution {

**public** **int**[] intersect(**int**[] nums1, **int**[] nums2) {

HashMap<Integer, Integer> map = **new** HashMap<Integer, Integer>();

ArrayList<Integer> result = **new** ArrayList<Integer>();

**for**(**int** i = 0; i < nums1.length; i++)

{

**if**(map.containsKey(nums1[i])) map.put(nums1[i], map.get(nums1[i])+1);

**else** map.put(nums1[i], 1);

}

**for**(**int** i = 0; i < nums2.length; i++)

{

**if**(map.containsKey(nums2[i]) && map.get(nums2[i]) > 0)

{

result.add(nums2[i]);

map.put(nums2[i], map.get(nums2[i])-1);

}

}

**int**[] r = **new** **int**[result.size()];

**for**(**int** i = 0; i < result.size(); i++)

{

r[i] = result.get(i);

}

**return** r;

}

}

下面这个做法和上一题目的区别，是从使用HashSet变成ArrayList

**public** **class** **Solution** {

**public** **int**[] **intersect**(**int**[] nums1, **int**[] nums2) {

List<Integer> res = **new** ArrayList<Integer>();

Arrays.sort(nums1);

Arrays.sort(nums2);

**for**(**int** i = 0, j = 0; i< nums1.length && j<nums2.length;){

**if**(nums1[i]<nums2[j]){

i++;

}

**else** **if**(nums1[i] == nums2[j]){

res.add(nums1[i]);

i++;

j++;

}

**else**{

j++;

}

}

**int**[] result = **new** **int**[res.size()];

**for**(**int** i = 0; i<res.size();i++){

result[i] = res.**get**(i);

}

**return** result;

}

}

*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?*

* If only nums2 cannot fit in memory, put all elements of nums1 into a HashMap, read chunks of array that fit into the memory, and record the intersections.
* If both nums1 and nums2 are so huge that neither fit into the memory, sort them individually (external sort), then read 2 elements from each array at a time in memory, record intersections.

Thanks for the solution. I think the second part of the solution is impractical, if you read 2 elements at a time, this procedure will take forever. In principle, we want minimize the number of disk access during the run-time.

An improvement can be sort them using external sort, read (let's say) 2G of each into memory and then using the 2 pointer technique, then read 2G more from the array that has been exhausted. Repeat this until no more data to read from disk.

But I am not sure this solution is good enough for an interview setting. Maybe the interviewer is expecting some solution using Map-Reduce paradigm.

### Longest Palindrome

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/longest-palindrome/submissions/)

* Total Accepted: **5912**
* Total Submissions: **13060**
* Difficulty: **Easy**

Given a string which consists of lowercase or uppercase letters, find the length of the longest palindromes that can be built with those letters.

This is case sensitive, for example "Aa" is not considered a palindrome here.

**Note:**  
Assume the length of given string will not exceed 1,010.

**Example:**

Input:

"abccccdd"

Output:

7

Explanation:

One longest palindrome that can be built is "dccaccd", whose length is 7.

我的做法，对字母计数。个数为奇数，则sum+=奇数-1；

最后结果看情况是否+1

注意：a的asc码为97；A的asc码为65.注意大小写字母之间不是连续的！

public int longestPalindrome(String s) {

int odd=0;

int even=0;

int[] map=new int[256];

for(int i=0; i<s.length(); i++)

{

char ch=s.charAt(i);

map[ch]++;

}

for(int i=0; i<256; i++)

{

if(map[i]%2==0) even+=map[i];

else

{

odd++;

even+=map[i]-1;

}

}

return odd>0?1+even: even;

}

**Simple HashSet solution Java**

public int longestPalindrome(String s) {

if(s==null || s.length()==0) return 0;

HashSet<Character> hs = new HashSet<Character>();

int count = 0;

for(int i=0; i<s.length(); i++){

if(hs.contains(s.charAt(i))){

hs.remove(s.charAt(i));

count++;

}else{

hs.add(s.charAt(i));

}

}

if(!hs.isEmpty()) return count\*2+1;

return count\*2;

}

这个方法不错，记录有配对的值；如果还有多，则多加一个单个。

Explaination of the previous code: just count the number of same pairs, then this can be used to put in the different direction to consist of palindrome. Then if there exist more chars, we can put one in the middle.

### H-Index

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/h-index/submissions/)

* Total Accepted: **50296**
* Total Submissions: **160575**
* Difficulty: **Medium**

Given an array of citations (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index.

According to the [definition of h-index on Wikipedia](https://en.wikipedia.org/wiki/H-index): "A scientist has index *h* if *h* of his/her *N* papers have **at least** *h*citations each, and the other *N − h* papers have **no more than** *h* citations each."

For example, given citations = [3, 0, 6, 1, 5], which means the researcher has 5 papers in total and each of them had received 3, 0, 6, 1, 5 citations respectively. Since the researcher has 3 papers with **at least** 3citations each and the remaining two with **no more than** 3 citations each, his h-index is 3.

**Note**: If there are several possible values for h, the maximum one is taken as the h-index.

**Hint:**

1. An easy approach is to sort the array first.
2. What are the possible values of h-index?
3. A faster approach is to use extra space.

**Credits:**  
Special thanks to [@jianchao.li.fighter](https://leetcode.com/discuss/user/jianchao.li.fighter) for adding this problem and creating all test cases.

[Subscribe](https://leetcode.com/subscribe/) to see which companies asked this question

Hide Tags

[Hash Table](https://leetcode.com/tag/hash-table/) [Sort](https://leetcode.com/tag/sort/)

Show Similar Problems

我的做法：h-index的值在1到citations.length之间。使用一个map记录每个h-index对应的，满足条件的论文数目。倒序遍历map，找到map[j]>=j的h-index。

public int hIndex(int[] citations) {

int[] map=new int[citations.length+1];

for(int i=0; i<citations.length; i++)

{

for(int j=1; j<=citations.length; j++)

{

if(citations[i]>=j)

{

map[j]++;

}

}

}

for(int j=citations.length; j>=1; j--)

{

if(map[j]>=j) return j;

}

return 0;

}

**My O(n) time solution use Java**

**public** **class** **Solution** {

*// 9.3 70 years diaoZhaTian China jiaYou*

**public** **int** **hIndex**(**int**[] citations) {

**int** length = citations.length;

**if** (length == 0) {

**return** 0;

}

**int**[] array2 = **new** **int**[length + 1];

**for** (**int** i = 0; i < length; i++) {

**if** (citations[i] > length) {

array2[length] += 1;

} **else** {

array2[citations[i]] += 1;

}

}

**int** t = 0;

**int** result = 0;

**for** (**int** i = length; i >= 0; i--) {

t = t + array2[i];

**if** (t >= i) {

**return** i;

}

}

**return** 0;

}

}

这个方法及写法很好，累加的思想。

**public** **int** **hIndex**(**int**[] citations) {

**int** len = citations.length;

**int**[] count = **new** **int**[len + 1];

**for** (**int** c: citations)

**if** (c > len)

count[len]++;

**else**

count[c]++;

**int** total = 0;

**for** (**int** i = len; i >= 0; i--) {

total += count[i];

**if** (total >= i)

**return** i;

}

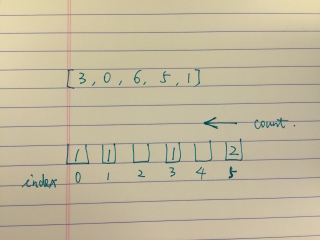
**return** 0;

}

**Java bucket sort O(n) solution with detail explanation**

This type of problems always throw me off, but it just takes some getting used to. The idea behind it is some bucket sort mechanisms. First, you may ask why bucket sort. Well, the h-index is defined as the number of papers with reference greater than the number. So assume n is the total number of papers, if we have n+1 buckets, number from 0 to n, then for any paper with reference corresponding to the index of the bucket, we increment the count for that bucket. The only exception is that for any paper with larger number of reference than n, we put in the n-th bucket.

Then we iterate from the back to the front of the buckets, whenever the total count exceeds the index of the bucket, meaning that we have the index number of papers that have reference greater than or equal to the index. Which will be our h-index result. The reason to scan from the end of the array is that we are looking for the greatest h-index. For example, given array [3,0,6,5,1], we have 6 buckets to contain how many papers have the corresponding index. Hope to image and explanation help.

[](http://i67.tinypic.com/2yvpfv5.jpg)

**public** **int** **hIndex**(**int**[] citations) {

**int** n = citations.length;

**int**[] buckets = **new** **int**[n+1];

**for**(**int** c : citations) {

**if**(c >= n) {

buckets[n]++;

} **else** {

buckets[c]++;

}

}

**int** count = 0;

**for**(**int** i = n; i >= 0; i--) {

count += buckets[i];

**if**(count >= i) {

**return** i;

}

}

**return** 0;

}

The idea is to see that the result can only range from 0 to the length of the array (because we can't have h-index greater than the total papers published). So we create an array "arr" which acts like a HashMap (using pigeon hole principle) and loop backwards from the highest element, then we find "tot" which is the total number of papers that has more than i citations, and we stop when tot>=i (total number of papers with more than i citations >= i). We don't need to keep going because we are trying the biggest i possible, we we stop and return the result.

**public** **class** **Solution** {

**public** **int** **hIndex**(**int**[] citations) {

**int** n = citations.length, tot=0;

**int**[] arr = **new** **int**[n+1];

**for** (**int** i=0; i<n; i++) {

**if** (citations[i]>=n) arr[n]++;

**else** arr[citations[i]]++;

}

**for** (**int** i=n; i>=0; i--) {

tot += arr[i];

**if** (tot>=i) **return** i;

}

**return** 0;

}

### Valid Sudoku

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/valid-sudoku/submissions/)

* Total Accepted: **93603**
* Total Submissions: **284613**
* Difficulty: **Easy**

Determine if a Sudoku is valid, according to: [Sudoku Puzzles - The Rules](http://sudoku.com.au/TheRules.aspx).

The Sudoku board could be partially filled, where empty cells are filled with the character '.'.



A partially filled sudoku which is valid.

**Note:**  
A valid Sudoku board (partially filled) is not necessarily solvable. Only the filled cells need to be validated.

我的做法，注意跳过‘.’时，是判断（i，j）重新组合过对应位置的数的情况。

判断行、列、九格里的是否存在重复。

注意九格遍历的写法，x位置的关系是：3\*(i/3)+j/3. i来控制起点，j控制位移。

y位置的关系是：3\*(i%3)+j%3.

public boolean isValidSudoku(char[][] board) {

for(int i=0; i<9; i++)

{

HashSet<Integer> row=new HashSet<>();

HashSet<Integer> col=new HashSet<>();

HashSet<Integer> grid=new HashSet<>();

for(int j=0; j<9; j++)

{

if(board[i][j]!='.' && !row.add(board[i][j]-'0')) {System.out.println("row"); return false;}// row repeat

if(board[j][i]!='.' && !col.add(board[j][i]-'0')) {System.out.println("col"); return false;}// column repeat

if(board[3\*(i/3)+j/3][3\*(i%3)+j%3]!='.' && !grid.add(board[3\*(i/3)+j/3][3\*(i%3)+j%3]-'0')) {System.out.println("grid"+i+" "+j+" "+board[3\*(i/3)+j/3][3\*(i%3)+j%3]); return false;}// grid repeat

}

}

return true;

}

**My short solution by C++. O(n2)**

Three flags are used to check whether a number appear.

used1: check each row

used2: check each column

used3: check each sub-boxes

**class** Solution

{

**public**:

**bool** **isValidSudoku**(vector<vector<**char**> > &board)

{

**int** used1[9][9] = {0}, used2[9][9] = {0}, used3[9][9] = {0};

**for**(**int** i = 0; i < board.size(); ++ i)

**for**(**int** j = 0; j < board[i].size(); ++ j)

**if**(board[i][j] != '.')

{

**int** num = board[i][j] - '0' - 1, k = i / 3 \* 3 + j / 3;

**if**(used1[i][num] || used2[j][num] || used3[k][num])

**return** false;

used1[i][num] = used2[j][num] = used3[k][num] = 1;

}

**return** true;

}

};

**public** **boolean** **isValidSudoku**(**char**[][] board) {

**for**(**int** i = 0; i<9; i++){

HashSet<Character> rows = **new** HashSet<Character>();

HashSet<Character> columns = **new** HashSet<Character>();

HashSet<Character> cube = **new** HashSet<Character>();

**for** (**int** j = 0; j < 9;j++){

**if**(board[i][j]!='.' && !rows.add(board[i][j]))

**return** **false**;

**if**(board[j][i]!='.' && !columns.add(board[j][i]))

**return** **false**;

**int** RowIndex = 3\*(i/3);

**int** ColIndex = 3\*(i%3);

**if**(board[RowIndex + j/3][ColIndex + j%3]!='.' && !cube.add(board[RowIndex + j/3][ColIndex + j%3]))

**return** **false**;

}

}

**return** **true**;

}

**public** **class** **Solution** {

**public** **boolean** **isValidSudoku**(**char**[][] board) {

**for** (**int** i=0; i<9; i++) {

**if** (!isParticallyValid(board,i,0,i,8)) **return** **false**;

**if** (!isParticallyValid(board,0,i,8,i)) **return** **false**;

}

**for** (**int** i=0;i<3;i++){

**for**(**int** j=0;j<3;j++){

**if** (!isParticallyValid(board,i\*3,j\*3,i\*3+2,j\*3+2)) **return** **false**;

}

}

**return** **true**;

}

**private** **boolean** **isParticallyValid**(**char**[][] board, **int** x1, **int** y1,**int** x2,**int** y2){

Set singleSet = **new** HashSet();

**for** (**int** i= x1; i<=x2; i++){

**for** (**int** j=y1;j<=y2; j++){

**if** (board[i][j]!='.') **if**(!singleSet.add(board[i][j])) **return** **false**;

}

}

**return** **true**;

}

### 腾讯基础研究面试题目

给定一个数组，根据另外一个数组中元素顺序进行排序。剩下的元素按照数组中原始顺序排列。

比如 nums=[2,3,5,6] arr=[5,3]

则按照这样方式排序的数组结果为：nums=[5,3,2,6];[Binary Search](https://leetcode.com/tag/binary-search/) [Hash Table](https://leetcode.com/tag/hash-table/) [Two Pointers](https://leetcode.com/tag/two-pointers/) [Sort](https://leetcode.com/tag/sort/)窗体底端

## 回溯

### Binary Watch

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/binary-watch/submissions/)

* Total Accepted: **7994**
* Total Submissions: **18730**
* Difficulty: **Easy**
* Contributors: **Admin**

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 on the right.



For example, the above binary watch reads "3:25".

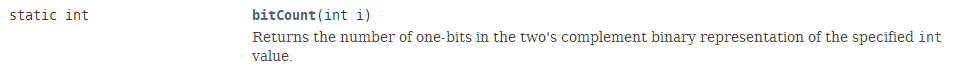
Given a non-negative integer *n* which represents the number of LEDs that are currently on, return all possible times the watch could represent.

**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".



这个方法不错。

Just go through the possible times and collect those with the correct number of one-bits.

Java:

public List<String> readBinaryWatch(**int** num) {

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**;

}

vector<string> readBinaryWatch(int num) {

vector<string> rs;

for (int h = 0; h < 12; h++)

for (int m = 0; m < 60; m++)

if (bitset<10>(h << 6 | m).count() == num)

rs.emplace\_back(to\_string(h) + (m < 10 ? ":0" : ":") + to\_string(m));

return rs;

}

**3ms Java Solution Using Backtracking and Idea of "Permutation and Combination"**

**public** **class** **Solution** {

**public** List<String> **readBinaryWatch**(**int** num) {

List<String> res = **new** ArrayList<>();

**int**[] nums1 = **new** **int**[]{8, 4, 2, 1}, nums2 = **new** **int**[]{32, 16, 8, 4, 2, 1};

**for**(**int** i = 0; i <= num; i++) {

List<Integer> list1 = generateDigit(nums1, i);

List<Integer> list2 = generateDigit(nums2, num - i);

**for**(**int** num1: list1) {

**if**(num1 >= 12) **continue**;

**for**(**int** num2: list2) {

**if**(num2 >= 60) **continue**;

res.add(num1 + ":" + (num2 < 10 ? "0" + num2 : num2));

}

}

}

**return** res;

}

**private** List<Integer> **generateDigit**(**int**[] nums, **int** count) {

List<Integer> res = **new** ArrayList<>();

generateDigitHelper(nums, count, 0, 0, res);

**return** res;

}

**private** **void** **generateDigitHelper**(**int**[] nums, **int** count, **int** pos, **int** sum, List<Integer> res) {

**if**(count == 0) {

res.add(sum);

**return**;

}

**for**(**int** i = pos; i < nums.length; i++) {

generateDigitHelper(nums, count - 1, i + 1, sum + nums[i], res);

}

}

}

public class Solution {

String[][] hour = {{"0"},

{"1", "2", "4", "8"},

{"3", "5", "6", "9", "10"},

{"7", "11"}};

String[][] minute = {{"00"}, //1

{"01", "02", "04", "08", "16", "32"}, //6

{"03", "05", "06", "09", "10", "12", "17", "18", "20", "24", "33", "34", "36", "40", "48"}, //15

{"07", "11", "13", "14", "19", "21", "22", "25", "26", "28", "35", "37", "38", "41", "42", "44", "49", "50", "52", "56"}, //20

{"15", "23", "27", "29", "30", "39", "43", "45", "46", "51", "53", "54", "57", "58"}, //14

{"31", "47", "55", "59"}}; //4

public List<String> readBinaryWatch(int num) {

List<String> ret = new ArrayList();

for (int i = 0; i <= 3 && i <= n; i++) {

if (n - i <= 5) {

for (String str1 : hour[i]) {

for (String str2 : minute[n - i]) {

ret.add(str1 + ":" + str2);

}

}

}

}

return ret;

}

}

### Word Search

[My Submissions](https://leetcode.com/problems/word-search/submissions/)

QuestionEditorial Solution

Total Accepted: **72661** Total Submissions: **317929** Difficulty: **Medium**

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.

Here accepted solution based on recursion. To save memory I decuded to apply bit mask for every visited cell. Please check board[y][x] ^= 256;

**public** **boolean** exist(**char**[][] board, String word) {

**char**[] w = word.toCharArray();

**for** (**int** y=0; y<board.length; y++) {

**for** (**int** x=0; x<board[y].length; x++) {

**if** (exist(board, y, x, w, 0)) **return** **true**;

}

}

**return** **false**;

}

**private** **boolean** exist(**char**[][] board, **int** y, **int** x, **char**[] word, **int** i) {

**if** (i == word.length) **return** **true**;

**if** (y<0 || x<0 || y == board.length || x == board[y].length) **return** **false**;

**if** (board[y][x] != word[i]) **return** **false**;

board[y][x] ^= 256;

**boolean** exist = exist(board, y, x+1, word, i+1)

|| exist(board, y, x-1, word, i+1)

|| exist(board, y+1, x, word, i+1)

|| exist(board, y-1, x, word, i+1);

board[y][x] ^= 256;

**return** exist;

}

这个写法比较容易懂，没有额外开辟空间来保存这个字符是否使用过的信息，而是将其设为一个特殊字符，然后记得恢复。

首先遍历这个矩阵。以矩阵中的每个元素开始去寻找。

如果这个元素不等于要寻找的单词的部分，则直接返回false。

如果等于，则将该元素首先设为特殊字符（表示该元素已经访问过），然后向四周开始搜索。搜索结束后将该元素恢复。

**public** **boolean** exist(**char**[][] board, String word) {

**for** (**int** i = 0; i < board.length; i++) {

**for** (**int** j = 0; j < board[i].length; j++) {

**if**(exist(board, i, j, word, 0)) **return** **true**;

}

}

**return** **false**;

}

**private** **boolean** exist(**char**[][] board, **int** x, **int** y, String word, **int** start) {

**if**(start >= word.length()) **return** **true**;

**if**(x < 0 || x >= board.length || y < 0 || y >= board[0].length) **return** **false**;

**if** (board[x][y] == word.charAt(start++)) {

**char** c = board[x][y];

board[x][y] = '#';

**boolean** res = exist(board, x + 1, y, word, start) || exist(board, x - 1, y, word, start) ||

exist(board, x, y + 1, word, start) || exist(board, x, y - 1, word, start);

board[x][y] = c;

**return** res;

}

**return** **false**;

}

**public** **class** **Solution** {

**static** **boolean**[][] visited;

**public** **boolean** exist(**char**[][] board, String word) {

visited = **new** **boolean**[board.length][board[0].length];

**for**(**int** i = 0; i < board.length; i++){

**for**(**int** j = 0; j < board[i].length; j++){

**if**((word.charAt(0) == board[i][j]) && search(board, word, i, j, 0)){

**return** **true**;

}

}

}

**return** **false**;

}

这个写法不错

**private** **boolean** search(**char**[][]board, String word, **int** i, **int** j, **int** index){

**if**(index == word.length()){

**return** **true**;

}

**if**(i >= board.length || i < 0 || j >= board[i].length || j < 0 || board[i][j] != word.charAt(index) || visited[i][j]){

**return** **false**;

}

visited[i][j] = **true**;

**if**(search(board, word, i-1, j, index+1) ||

search(board, word, i+1, j, index+1) ||

search(board, word, i, j-1, index+1) ||

search(board, word, i, j+1, index+1)){

**return** **true**;

}

visited[i][j] = **false**;

**return** **false**;

}

}

### Gray Code

[My Submissions](https://leetcode.com/problems/gray-code/submissions/)

QuestionEditorial Solution

Total Accepted: **59629** Total Submissions: **163994** Difficulty: **Medium**

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer *n* representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given *n* = 2, return [0,1,3,2]. Its gray code sequence is:

00 - 0

01 - 1

11 - 3

10 - 2

**Note:**  
For a given *n*, a gray code sequence is not uniquely defined.

For example, [0,2,3,1] is also a valid gray code sequence according to the above definition.

For now, the judge is able to judge based on one instance of gray code sequence. Sorry about that.

这个公式不错

public List<**Integer**> grayCode(int n) {

List<**Integer**> result = new LinkedList<>();

for (int i = 0; i < (1<<n); i++) result.add(i ^ (i>>1));

return result;

}

The idea is simple. G(i) = i^ (i/2).

Just added information for those who are interested (all credited to [Wiki gray code](http://en.wikipedia.org/wiki/Gray_code)

/\*

The purpose of this function is to convert an unsigned

binary number to reflected binary Gray code.

The operator >> is shift right. The operator ^ is exclusive or.

\*/

**unsigned** **int** binaryToGray(**unsigned** **int** num)

{

**return** (num >> 1) ^ num;

}

/\*

The purpose of this function is to convert a reflected binary

Gray code number to a binary number.

\*/

**unsigned** **int** grayToBinary(**unsigned** **int** num)

{

**unsigned** **int** mask;

**for** (mask = num >> 1; mask != 0; mask = mask >> 1)

{

num = num ^ mask;

}

**return** num;

}

As we known:

Gi = Bi+1 xor Bi

For example, trans binay '001' to gray code:

tmp = 001 <**<** 1

then,

bin 0 0 0 1

tmp 0 0 1 0

-xor------------

0 0 1 1

and the gray code is:

0 0 1 1 >> 1 (ignore last bit) => 0 0 1

public ArrayList<**Integer**> grayCode(int n) {

ArrayList<**Integer**> arr = new ArrayList<**Integer**>();

arr.add(0);

for(int i=0;i<**n;i++){**

int inc = 1<<i;

for(int j=arr.size()-1;j>=0;j--){

arr.add(arr.get(j)+inc);

}

}

return arr;

}

}

这个方法不错，构造法

My idea is to generate the sequence iteratively. For example, when n=3, we can get the result based on n=2. 00,01,11,10 -> (000,001,011,010 ) (110,111,101,100). The middle two numbers only differ at their highest bit, while the rest numbers of part two are exactly symmetric of part one. It is easy to see its correctness. Code is simple:

public List<**Integer**> grayCode(int n) {

List<**Integer**> rs=new ArrayList<**Integer**>();

rs.add(0);

for(int i=0;i<**n;i++){**

int size=rs.size();

for(int k=size-1;k>=0;k--)//反向处理前面的编码

rs.add(rs.get(k) | (1<**<i));//在高位加1**

}

return rs;

}

这个编码有这样的规律（其中一种）

设n=2的编码已经找到，则n=3的编码，只需要在n=2的编码的第一位都加0，再在逆序的n=2的编码前加1，即可。

I agree with loick. I don't think it's a knowledge base problem. It's also my first time to hear about Gray Code. But after trying some small cases, I still figured out an algorithm for it.

From my intuition, the problem is like Hanoi. If you're able to solve n = 2 case, then you can kind of repeat it twice to achieve n = 3 case. Lets try to extend n = 2 case to n = 3 case first.

When n = 2, the sequence is 00 -> 01 -> 11 -> 10 If you want to extend it to n=3 directly without modifying old part, there are only two possible sequence, and they are not hard to find out.

000 -> 001 -> 011 -> 010 -> 110 -> 111 -> 101 -> 100

000 -> 001 -> 011 -> 010 -> 110 -> 100 -> 101 -> 111

So now, the problem is, which one should we choose. I would choose the first one for two reasons.

1. The last elements have similar form in both n=2 and n=3 case. They are 1 follows bunch of 0's. Since we hope to extend the same algorithm to n=4 n=5... cases. It's good to preserve some properties.
2. If we only look at the last 2 digits, we can see that in the first sequence, the second half is exact the reverse of the first half, that means, we can systematically generate the second half according to the first half.

That's how I figured out the algorithm. Hope that helps!

public static List<**Integer**> grayCode(int n) {

if (n < 0)

return new ArrayList<Integer>();

if (n == 0) {

List<**Integer**> list = new ArrayList<**Integer**>();

list.add(0);

return list;

}

List<**Integer**> tmp = grayCode(n - 1);

List<**Integer**> result = new ArrayList<**Integer**>(tmp);

int addNumber = 1 <**<** (n - 1);

for (int i = tmp.size() - 1; i >= 0; i--) {

result.add(addNumber + tmp.get(i));

}

return result;

}

### Restore IP Addresses

[My Submissions](https://leetcode.com/problems/restore-ip-addresses/submissions/)

QuestionEditorial Solution

Total Accepted: **55286** Total Submissions: **236080** Difficulty: **Medium**

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)

这个题目，需要考虑子串很长时的情况；划分的子串出现01的情况

这个方法也不错

将字符串s分成4个子串，检查每个子串是否有效（无效的几种情况：长度大于3或小于等于0，长度大于1时首位为0，数字大于255）

**public** **class** **Solution** {

**public** List<String> restoreIpAddresses(String s) {

List<String> res = **new** ArrayList<String>();

**int** len = s.length();

**for**(**int** i = 1; i<4 && i<len-2; i++){

**for**(**int** j = i+1; j<i+4 && j<len-1; j++){

**for**(**int** k = j+1; k<j+4 && k<len; k++){

String s1 = s.substring(0,i), s2 = s.substring(i,j), s3 = s.substring(j,k), s4 = s.substring(k,len);

**if**(isValid(s1) && isValid(s2) && isValid(s3) && isValid(s4)){

res.add(s1+"."+s2+"."+s3+"."+s4);

}

}

}

}

**return** res;

}

**public** **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**;

}

}

3-loop divides the string s into 4 substring: s1, s2, s3, s4. Check if each substring is valid. In isValid, strings whose length greater than 3 or equals to 0 is not valid; or if the string's length is longer than 1 and the first letter is '0' then it's invalid; or the string whose integer representation greater than 255 is invalid.

[**my concise AC java code**](https://leetcode.com/discuss/19296/my-concise-ac-java-code)

the basic idea is to make three cuts into the string, separating it into four parts, each part contains 1~3 digits and it must be <255.

**static** List<String> restoreIpAddresses(String s) {

List<String> ans = **new** ArrayList<String>();

**int** len = s.length();

**for** (**int** i = 1; i <=3; ++i){ // first cut

**if** (len-i > 9) **continue**;

**for** (**int** j = i+1; j<=i+3; ++j){ //second cut

**if** (len-j > 6) **continue**;

**for** (**int** k = j+1; k<=j+3 && k<len; ++k){ // third cut

**int** a,b,c,d; // the four int's seperated by "."

a = Integer.parseInt(s.substring(0,i));

b = Integer.parseInt(s.substring(i,j)); // notice that "01" can be parsed into 1. Need to deal with that later.

c = Integer.parseInt(s.substring(j,k));

d = Integer.parseInt(s.substring(k));

**if** (a>255 || b>255 || c>255 || d>255) **continue**;

String ip = a+"."+b+"."+c+"."+d;

**if** (ip.length()<len+3) **continue**; // this is to reject those int's parsed from "01" or "00"-like substrings

ans.add(ip);

}

}

}

**return** ans;

}

Very simple DFS solution

public List<String> restoreIpAddresses(String s) {

List<String> solutions = new ArrayList<String>();

restoreIp(s, solutions, 0, "", 0);

**return** solutions;

}

private void restoreIp(String ip, List<String> solutions, int idx, String restored, int count) {

**if** (count > 4) **return**;

**if** (count == 4 && idx == ip.length()) solutions.add(restored);

**for** (int i=1; i<4; i++) {

**if** (idx+i > ip.length()) **break**;

String s = ip.substring(idx,idx+i);

**if** ((s.startsWith("0") && s.length()>1) || (i==3 && Integer.parseInt(s) >= 256)) continue;

restoreIp(ip, solutions, idx+i, restored+s+(count==3?"" : "."), count+1);

}

}

[**Easy Java code of backtracking within 16 lines**](https://leetcode.com/discuss/48726/easy-java-code-of-backtracking-within-16-lines)

这个方法不错

注意出现010时候的处理

**public** List<String> restoreIpAddresses(String s) {

List<String> res = **new** ArrayList<>();

helper(s,"",res,0);

**return** res;

}

**public** **void** helper(String s, String tmp, List<String> res,**int** n){

**if**(n==4){

**if**(s.length()==0) res.add(tmp.substring(0,tmp.length()-1));

//substring here to get rid of last '.'

**return**; //注意这个return的位置

}

**for**(**int** k=1;k<=3;k++){

**if**(s.length()<k) **continue**;//这句的作用是为了使下一句substring有效

**int** val = Integer.parseInt(s.substring(0,k));

**if**(val>255 || k!=String.valueOf(val).length()) **continue**;

/\*in the case 010 the parseInt will return len=2 where val=10, but k=3, skip this.\*/

helper(s.substring(k),tmp+s.substring(0,k)+".",res,n+1);

}

}

### Palindrome Partitioning

[My Submissions](https://leetcode.com/problems/palindrome-partitioning/submissions/)

QuestionEditorial Solution

Total Accepted: **63816** Total Submissions: **231359** Difficulty: **Medium**

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"]

]

我的做法，较清晰，不断缩短String s

public class Solution {

List<List<String>> res=new ArrayList<List<String>>();

public List<List<String>> partition(String s) {

List<String> cur=new ArrayList<String>();

combine(s,cur);

return res;

}

public void combine(String s,List<String> cur)

{

if(s.length()==0)

{

res.add(new ArrayList<String>(cur));

return;

}

for(int i=1;i<=s.length();i++)

{

String tmp=s.substring(0,i);

if(!isValid(tmp)) continue;

cur.add(tmp);

combine(s.substring(i),cur);

cur.remove(cur.size()-1);

}

}

public boolean isValid(String s)

{

int len=s.length();

for(int i=0;i<len/2;i++)

{

if(s.charAt(i)!=s.charAt(len-1-i)) return false;

}

return true;

}

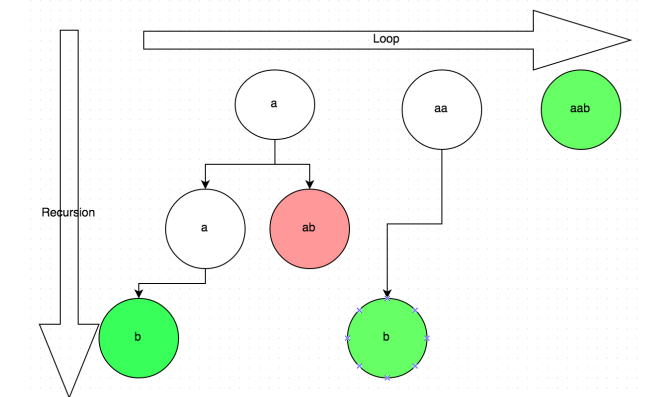
}

Java: Backtracking solution.

if the input is "aab", check if [0,0] "a" is palindrome. then check [0,1] "aa", then [0,2] "aab". While checking [0,0], the rest of string is "ab", use ab as input to make a recursive call.

in this example, in the loop of i=l+1, a recursive call will be made with input = "ab". Every time a recursive call is made, the position of l move right.

How to define a correct answer? Think about DFS, if the current string to be checked (Palindrome) contains the last position, in this case "c", this path is a correct answer, otherwise, it's a false answer.



line 13: is the boundary to check if the current string contains the last element. l>=s.length()

**public** **class** **Solution** {

**List**<**List**<String>> resultLst;

ArrayList<String> currLst;

**public** **List**<**List**<String>> partition(String s) {

resultLst = **new** ArrayList<**List**<String>>();

currLst = **new** ArrayList<String>();

backTrack(s,0);

**return** resultLst;

}

**public** void backTrack(String s, int l){

**if**(currLst.size()>0 //the initial str could be palindrome

&& l>=s.length()){

**List**<String> r = (ArrayList<String>) currLst.**clone**();

resultLst.add(r);

}

**for**(int i=l;i<s.length();i++){

**if**(isPalindrome(s,l,i)){

**if**(l==i)

currLst.add(Character.toString(s.charAt(i)));

**else**

currLst.add(s.substring(l,i+1));

backTrack(s,i+1);

currLst.remove(currLst.size()-1);

}

}

}

**public** 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**;

}

}

My Java DP only solution without recursion. O(n^2)

public class Solution {

public static List<**List<String**>> partition(String s) {

int len = s.length();

List<**List<String**>>[] result = new List[len + 1];

result[0] = new ArrayList<**List<String**>>();

result[0].add(new ArrayList<**String**>());

boolean[][] pair = new boolean[len][len];

for (int i = 0; i < s.length(); i++) {

result[i + 1] = new ArrayList<List<String>>();

for (int left = 0; left <**=** i; left++) {

if (s.charAt(left) == s.charAt(i) && (i-left <= 1 || pair[left + 1][i - 1])) {

pair[left][i] = true;

String str = s.substring(left, i + 1);

for (List<String> r : result[left]) {

List<**String**> ri = new ArrayList<**String**>(r);

ri.add(str);

result[i + 1].add(ri);

}

}

}

}

return result[len];

}

}

Here the **pair** is to mark a range for the substring is a Pal. if pair[i][j] is true, that means sub string from i to j is pal.

The **result[i]**, is to store from beginng until current index i (Non inclusive), all possible partitions. From the past result we can determine current result.

### N-Queens

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/n-queens/submissions/)

* Total Accepted: **64433**
* Total Submissions: **229832**
* Difficulty: **Hard**

The *n*-queens puzzle is the problem of placing *n* queens on an *n*×*n* chessboard such that no two queens attack each other.



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.."]

]

我的做法同书,

1，cols[row]=col表示行为row，列为col处有皇后。

2，有check函数，检查(i, j)位置处是否可以放置皇后。即检查，行号小于i时的列与j的冲突，及是否有对角线冲突。注意，这里row是从行0开始依次往后分配，所以不用检查行冲突。

public class Solution {

public List<List<String>> solveNQueens(int n) {

List<List<String>> res=new ArrayList<List<String>>();

int[] cols=new int[n];

place(0, cols, n, res);

return res;

}

public void place(int row, int[] cols, int n, List<List<String>> res)

{

if(row==n)

{

List<String> list=new ArrayList<String>();

for(int i=0; i<n; i++) //row

{

String ss="";

for(int j=0; j<n; j++)

{

if(cols[i]!=j) ss+=".";

else ss+="Q";

}

list.add(ss);

}

res.add(list);

}

else

{

for(int i=0; i<n; i++)

{

if(check(row, i, cols))

{

cols[row]=i;

place(row+1, cols, n, res);

}

}

}

}

public boolean check(int row, int col, int[] cols)

{

for(int i=0; i<row; i++)

{

int col2=cols[i];

if(col2==col) return false;

int colDis=Math.abs(col2-col);

int rowDis=row-i;

if(colDis==rowDis) return false;

}

return true;

}

}

**public** **class** **Solution** {

**public** List<List<String>> solveNQueens(**int** n) {

**char**[][] board = **new** **char**[n][n];

**for**(**int** i = 0; i < n; i++)

**for**(**int** j = 0; j < n; j++)

board[i][j] = '.';

List<List<String>> res = **new** ArrayList<List<String>>();

dfs(board, 0, res);

**return** res;

}

**private** **void** **dfs**(**char**[][] board, **int** colIndex, List<List<String>> res) {

**if**(colIndex == board.length) {

res.add(construct(board));

**return**;

}

**for**(**int** i = 0; i < board.length; i++) {

**if**(validate(board, i, colIndex)) {

board[i][colIndex] = 'Q';

dfs(board, colIndex + 1, res);

board[i][colIndex] = '.';

}

}

}

**private** **boolean** **validate**(**char**[][] board, **int** x, **int** y) {

**for**(**int** i = 0; i < board.length; i++) {

**for**(**int** j = 0; j < y; j++) {

**if**(board[i][j] == 'Q' && (x + j == y + i || x + y == i + j || x == i))

**return** **false**;

}

}

**return** **true**;

}

**private** List<String> **construct**(**char**[][] board) {

List<String> res = **new** LinkedList<String>();

**for**(**int** i = 0; i < board.length; i++) {

String s = **new** String(board[i]);

res.add(s);

}

**return** res;

}

}

### N-Queens II

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/n-queens-ii/submissions/)

* Total Accepted: **52045**
* Total Submissions: **124853**
* Difficulty: **Hard**

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.



这个方法不错

*/\*\**

*\* don't need to actually place the queen,*

*\* instead, for each row, try to place without violation on*

*\* col/ diagonal1/ diagnol2.*

*\* trick: to detect whether 2 positions sit on the same diagnol:*

*\* if delta(col, row) equals, same diagnol1;*

*\* if sum(col, row) equals, same diagnal2.*

*\*/*

**private** **final** Set<Integer> occupiedCols = **new** HashSet<Integer>();

**private** **final** Set<Integer> occupiedDiag1s = **new** HashSet<Integer>();

**private** **final** Set<Integer> occupiedDiag2s = **new** HashSet<Integer>();

**public** **int** **totalNQueens**(**int** n) {

**return** totalNQueensHelper(0, 0, n);

}

**private** **int** **totalNQueensHelper**(**int** row, **int** count, **int** n) {

**for** (**int** col = 0; col < n; col++) {

**if** (occupiedCols.contains(col))

**continue**;

**int** diag1 = row - col;

**if** (occupiedDiag1s.contains(diag1))

**continue**;

**int** diag2 = row + col;

**if** (occupiedDiag2s.contains(diag2))

**continue**;

*// we can now place a queen here*

**if** (row == n-1)

count++;

**else** {

occupiedCols.add(col);

occupiedDiag1s.add(diag1);

occupiedDiag2s.add(diag2);

count = totalNQueensHelper(row+1, count, n);

*// recover*

occupiedCols.remove(col);

occupiedDiag1s.remove(diag1);

occupiedDiag2s.remove(diag2);

}

}

**return** count;

}

This is a classic backtracking problem.

Start row by row, and loop through columns. At each decision point, skip unsafe positions by using three boolean arrays.

Start going back when we reach row n.

Just FYI, if using HashSet, running time will be at least 3 times slower!

**public** **class** **Solution** {

**int** count = 0;

**public** **int** **totalNQueens**(**int** n) {

**boolean**[] cols = **new** **boolean**[n]; *// columns |*

**boolean**[] d1 = **new** **boolean**[2 \* n]; *// diagonals \*

**boolean**[] d2 = **new** **boolean**[2 \* n]; *// diagonals /*

backtracking(0, cols, d1, d2, n);

**return** count;

}

**public** **void** **backtracking**(**int** row, **boolean**[] cols, **boolean**[] d1, **boolean** []d2, **int** n) {

**if**(row == n) count++;

**for**(**int** col = 0; col < n; col++) {

**int** id1 = col - row + n;

**int** id2 = col + row;

**if**(cols[col] || d1[id1] || d2[id2]) **continue**;

cols[col] = **true**; d1[id1] = **true**; d2[id2] = **true**;

backtracking(row + 1, cols, d1, d2, n);

cols[col] = **false**; d1[id1] = **false**; d2[id2] = **false**;

}

}

}

## 双指针

### Two Sum II - Input array is sorted

QuestionEditorial Solution

[My Submissions](https://leetcode.com/problems/two-sum-ii-input-array-is-sorted/submissions/)

* Total Accepted: **30035**
* Total Submissions: **62278**
* Difficulty: **Medium**

Given an array of integers that is already ***sorted in ascending order***, find two numbers such that they add up to a specific target number.

The function twoSum should return indices 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.

**Input:** numbers={2, 7, 11, 15}, target=9  
**Output:** index1=1, index2=2

我的做法，直接双指针。

public int[] twoSum(int[] numbers, int target) {

int low=0, high=numbers.length-1;

int[] res=new int[2];

while(low<=high)

{

if(numbers[low]+numbers[high]<target)

{

low++;

}

else if(numbers[low]+numbers[high]==target)

{

res[0]=low+1;

res[1]=high+1;

return res;

}

else high--;

}

return res;

}

Without HashMap, just have two pointers, A points to index 0, B points to index len - 1, shrink the scope based on the value and target comparison.

**public** **int**[] **twoSum**(**int**[] num, **int** target) {

**int**[] indice = **new** **int**[2];

**if** (num == null || num.length < 2) **return** indice;

**int** left = 0, right = num.length - 1;

**while** (left < right) {

**int** v = num[left] + num[right];

**if** (v == target) {

indice[0] = left + 1;

indice[1] = right + 1;

**break**;

} **else** **if** (v > target) {

right --;

} **else** {

left ++;

}

}

**return** indice;

}

**A less efficient way (binary search)**

I know that the best solution is using two pointers like what is done in the previous solution sharing. However, I see the tag contains "binary search". I do not know if I misunderstand but is binary search a less efficient way for this problem.

Say, fix the first element A[0] and do binary search on the remaining n-1 elements. If cannot find any element which equals target-A[0], Try A[1]. That is, fix A[1] and do binary search on A[2]~A[n-1]. Continue this process until we have the last two elements A[n-2] and A[n-1].

Does this gives a time complexity lg(n-1) + lg(n-2) + ... + lg(1) ~ O(lg(n!)) ~ O(nlgn). So it is less efficient than the O(n) solution. Am I missing something here?

The code also passes OJ.

vector<**int**> twoSum(vector<**int**> &numbers, **int** target) {

**if**(numbers.empty()) **return** {};

**for**(**int** i=0; i<numbers.size()-1; i++) {

**int** start=i+1, end=numbers.size()-1, gap=target-numbers[i];

**while**(start <= end) {

**int** m = start+(end-start)/2;

**if**(numbers[m] == gap) **return** {i+1,m+1};

**else** **if**(numbers[m] > gap) end=m-1;

**else** start=m+1;

}

}

}

You can add an early exit if target-numbers[i] > numbers[numbers.size()-1]. This can speed up the binary search.

### 3Sum

[My Submissions](https://leetcode.com/problems/3sum/submissions/)

QuestionEditorial Solution

Total Accepted: **112881** Total Submissions: **602879** Difficulty: **Medium**

Given an array *S* of *n* integers, are there elements *a*, *b*, *c* in *S* such that *a* + *b* + *c* = 0? Find all unique triplets in the array which gives the sum of zero.

**Note:**

* Elements in a triplet (*a*,*b*,*c*) must be in non-descending order. (ie, *a* ≤ *b* ≤ *c*)
* The solution set must not contain duplicate triplets.

For example, given array S = {-1 0 1 2 -1 -4},

A solution set is:

(-1, 0, 1)

(-1, -1, 2)

Concise O(N^2) Java solution

这个方法不错

首先将数组排序；

固定第一个数；剩下的数，做双向扫描；

注意跳过重复元素（固定首个数字时；双向扫描时）

Hi guys!

The idea is to sort an input array and then run through all indices of a possible first element of a triplet. For each possible first element we make a standard bi-directional 2Sum sweep of the remaining part of the array. Also we want to skip equal elements to avoid duplicates in the answer without making a set or smth like that.

public List<**List<Integer**>> threeSum(int[] num) {

Arrays.sort(num);

List<**List<Integer**>> res = new LinkedList<>();

for (int i = 0; i < num.length-2; i++) {//首先固定一个数

if (i == 0 || (i > 0 && num[i] != num[i-1])) {

int lo = i+1, hi = num.length-1, sum = 0 - num[i];

while (lo < hi) {

if (num[lo] + num[hi] == sum) {

res.add(Arrays.asList(num[i], num[lo], num[hi]));

while (lo < hi && num[lo] == num[lo+1]) lo++;

while (lo < hi && num[hi] == num[hi-1]) hi--;

lo++; hi--;

} else if (num[lo] + num[hi] < sum) lo++;

else hi--;

}

}

}

return res;

}

Have a nice coding!

Sort the array, iterate through the list, and use another two pointers to approach the target. Runtime: 7ms

**public** List<List<Integer>> threeSum(**int**[] nums) {

List<List<Integer>> result = **new** ArrayList<>();

**if**(nums == **null** || nums.length < 3) **return** result;

Arrays.sort(nums);

**int** len = nums.length;

**for**(**int** i = 0; i < len; i++) {

**if**(i > 0 && nums[i] == nums[i - 1]) **continue**; // Skip same results

**int** target = 0 - nums[i];

**int** j = i + 1, k = len - 1;

**while**(j < k) {

**if**(nums[j] + nums[k] == target) {

result.add(Arrays.asList(nums[i], nums[j], nums[k]));

**while**(j < k && nums[j] == nums[j + 1]) j++; // Skip same results

**while**(j < k && nums[k] == nums[k - 1]) k--; // Skip same results

j++; k--;

} **else** **if**(nums[j] + nums[k] < target) {

j++;

} **else** {

k--;

}

}

}

**return** result;

}

### 3Sum Closest

[My Submissions](https://leetcode.com/problems/3sum-closest/submissions/)

QuestionEditorial Solution

Total Accepted: **76139** Total Submissions: **261980** Difficulty: **Medium**

Given an array *S* of *n* integers, find three integers in *S* such that the sum is closest 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).

Here is a solution in Order(N^2). I got help from this post on [stackoverflow](http://stackoverflow.com/questions/2070359/finding-three-elements-in-an-array-whose-sum-is-closest-to-an-given-number)   
Can we improve this time complexity ?

**int** threeSumClosest(**vector**<**int**> &num, **int** target) {

**vector**<**int**> v(num.begin(), num.end()); // I didn't wanted to disturb original array.

**int** n = 0;

**int** ans = 0;

**int** sum;

sort(v.begin(), v.end());

// If less then 3 elements then return their sum

**while** (v.size() <= 3) {

**return** accumulate(v.begin(), v.end(), 0);

}

n = v.size();

/\* v[0] v[1] v[2] ... v[i] .... v[j] ... v[k] ... v[n-2] v[n-1]

\* v[i] <= v[j] <= v[k] always, because we sorted our array.

\* Now, for each number, v[i] : we look for pairs v[j] & v[k] such that

\* absolute value of (target - (v[i] + v[j] + v[k]) is minimised.

\* if the sum of the triplet is greater then the target it implies

\* we need to reduce our sum, so we do K = K - 1, that is we reduce

\* our sum by taking a smaller number.

\* Simillarly if sum of the triplet is less then the target then we

\* increase out sum by taking a larger number, i.e. J = J + 1.

\*/

ans = v[0] + v[1] + v[2];

**for** (**int** i = 0; i < n-2; i++) {

**int** j = i + 1;

**int** k = n - 1;

**while** (j < k) {

sum = v[i] + v[j] + v[k];

**if** (abs(target - ans) > abs(target - sum)) {

ans = sum;

**if** (ans == target) **return** ans;

}

(sum > target) ? k-- : j++;

}

}

**return** ans;

}

Similar to 3 Sum problem, use 3 pointers to point current element, next element and the last element. If the sum is less than target, it means we have to add a larger element so next element move to the next. If the sum is greater, it means we have to add a smaller element so last element move to the second last element. Keep doing this until the end. Each time compare the difference between sum and target, if it is less than minimum difference so far, then replace result with it, otherwise keep iterating.

public **class** **Solution** {

public int threeSumClosest(int[] num, int target) {

int result = num[0] + num[1] + num[num.length - 1];

Arrays.sort(num);

**for** (int i = 0; i < num.length - 2; i++) {

int start = i + 1, **end** = num.length - 1;

**while** (start < **end**) {

int sum = num[i] + num[start] + num[**end**];

**if** (sum > target) {

**end**--;

} **else** {

start++;

}

**if** (Math.abs(sum - target) < Math.abs(result - target)) {

result = sum;

}

}

}

**return** result;

}

}

这个方法比较完善，会跳过重复，考虑等于目标值的情况

第一个值先固定

后面两个指针，每次移动时，都需要更新最小值

You answer is great. However, we could improve performance a bit by skipping duplicate elements.

**public** **int** threeSumClosest(**int**[] nums, **int** target) {

Arrays.sort(nums);

**int** sum = nums[0] + nums[1] + nums[nums.length - 1];

**int** closestSum = sum;

**for**(**int** i = 0; i < nums.length - 2; i++){

**if**(i==0 || nums[i]!=nums[i-1]){

**int** left = i + 1, right = nums.length - 1;

**while**(left < right){

sum = nums[left] + nums[right] + nums[i];

**if**(sum < target){

//move closer to target sum.

**while**(left<right && nums[left] == nums[left+1]){

left++;

}

left++;//该数与后面一个数不相等，则直接移动

}**else** **if**(sum > target){

//move closer to target sum.

**while**(left<right && nums[right] == nums[right-1]){

right--;

}

right--;

}**else**{

**return** sum;//相等时,直接返回

}

//update the closest sum if needed.

**if**(Math.abs(target - sum) < Math.abs(target - closestSum)){

closestSum = sum;

}

}

}

}

**return** closestSum;

}

### 4Sum

[My Submissions](https://leetcode.com/problems/4sum/submissions/)

QuestionEditorial Solution

Total Accepted: **71143** Total Submissions: **299487** Difficulty: **Medium**

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 in the array which gives the sum of target.

**Note:**

* Elements in a quadruplet (*a*,*b*,*c*,*d*) must be in non-descending order. (ie, *a* ≤ *b* ≤ *c* ≤ *d*)
* 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)

记住这个模式

我的做法，完全类似于3sum，只是先固定前面两个数，注意前面两个数有重复时的处理

当出现sum与target相等时，仍然移动指针，跳过些重复值（为什么3Sum Closest题里将这儿部分处理放到另外地方，怎么判断🡪重点在于等于目标值时一定要相应再移动指针，否则会有重复结果，至于小于大于目标的部分是否要排除重复，都可以）

public List<List<Integer>> fourSum(int[] nums, int target) {

Arrays.sort(nums);

List<List<Integer>> res=new ArrayList<List<Integer>>();

//List<Integer> cur=new ArrayList<Integer>();

for(int i=0;i<nums.length-3;i++)

{

if(i==0 ||(i>0 && nums[i]!=nums[i-1]))

{

for(int j=i+1;j<nums.length-2;j++)

{

if(j==i+1 || (j>i+1 && nums[j]!=nums[j-1]))

{

int low=j+1;

int high=nums.length-1;

while(low<high)

{

int sum=nums[low]+nums[high]+nums[i]+nums[j];

if(sum==target)

{

res.add(Arrays.asList(nums[i],nums[j],nums[low],nums[high]));

while(low<high && nums[low]==nums[low+1]) low++;

while(low<high && nums[high]==nums[high-1]) high--;//这两步用于去除一些重复结果，重要

high--;

low++;//关键，勿忘，否则会陷死循环

}

else if(sum<target)

{

// while(low<high && nums[low]==nums[low+1]) low++; //可以加这步，也可以不加，可能没必要

low++;

}

else

{

//while(low<high && nums[high]==nums[high-1]) high--;

high--;

}

}

}

}

}

}

return res;

}

7ms java code win over 100%

The first time win over 100%. Basic idea is using subfunctions for 3sum and 2sum, and keeping throwing all impossible cases. O(n^3) time complexity, O(1) extra space complexity.

**public** List<List<Integer>> fourSum(**int**[] nums, **int** target) {

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])// avoid duplicate

**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

**if** (i + 3 < len && nums[i + 3] == z)

res.add(Arrays.asList(z, z, z, z));

**break**;

}

threeSumForFourSum(nums, target - z, i + 1, len - 1, res, z);

}

**return** res;

}

/\*

\* Find all possible distinguished three numbers adding up to the target

\* in sorted array nums[] between indices low and high. If there are,

\* add all of them into the ArrayList fourSumList, using

\* fourSumList.add(Arrays.asList(z1, the three numbers))

\*/

**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]) // avoid duplicate

**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**;

}

twoSumForFourSum(nums, target - z, i + 1, high, fourSumList, z1, z);

}

}

/\*

\* Find all possible distinguished two numbers adding up to the target

\* in sorted array nums[] between indices low and high. If there are,

\* add all of them into the ArrayList fourSumList, using

\* fourSumList.add(Arrays.asList(z1, z2, the two numbers))

\*/

**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**;

**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]));

x = nums[i];

**while** (++i < j && x == nums[i]) // avoid duplicate

;

x = nums[j];

**while** (i < --j && x == nums[j]) // avoid duplicate

;

}

**if** (sum < target)

i++;

**if** (sum > target)

j--;

}

**return**;

}

### Reverse Vowels of a String

[My Submissions](https://leetcode.com/problems/reverse-vowels-of-a-string/submissions/)

QuestionEditorial Solution

Total Accepted: **7263** Total Submissions: **20592** Difficulty: **Easy**

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".

这个写法不错，注意细节，start<end

String类的contains方法

|  |  |
| --- | --- |
| boolean | [**contains**](file:///E:\docsJava\api\java\lang\String.html#contains-java.lang.CharSequence-)([**CharSequence**](file:///E:\docsJava\api\java\lang\CharSequence.html) s)  Returns true if and only if this string contains the specified sequence of char values. |

Java Standard Two Pointer Solution

In the inner while loop, don't forget the condition "start less than end" while incrementing start and decrementing end. This is my friend's google phone interview question. Cheers! // update! May use a HashSet to reduce the look up time to O(1)

public **class** **Solution** {

public String reverseVowels(String s) {

**if**(s == null || s.length()==0) **return** s;

String vowels = "aeiouAEIOU";

char[] chars = s.toCharArray();

int start = 0;

int **end** = s.length()-1;

**while**(start<**end**){

**while**(start<**end** && !vowels.contains(chars[start]+"")){

start++;

}

**while**(start<**end** && !vowels.contains(chars[**end**]+"")){

**end**--;

}

char temp = chars[start];

chars[start] = chars[**end**];

chars[**end**] = temp;

start++;

**end**--;

}

**return** new String(chars);

}

}

这个方法也不错，利用栈的性质，先将所有元音入栈，然后第二次遍历字符串，遇到元音，则用出栈元素进行替换。

Try this! But since we traverse the string twice, the running time doubles. Good Luck!

**public** **class** Solution {

**public** String reverseVowels(String s) {

**if**(s == null || s.length()==0){

**return** s;

}

HashSet<Character> vowels = **new** HashSet<>();

vowels.add('a');

vowels.add('e');

vowels.add('i');

vowels.add('o');

vowels.add('u');

vowels.add('A');

vowels.add('E');

vowels.add('I');

vowels.add('O');

vowels.add('U');

// reverse the vowels while popping up

Stack<Character> vStack = **new** Stack<>();

**for**(**char** c : s.toCharArray()){

**if**(vowels.contains(c)){

vStack.push(c);

}

}

StringBuilder sb = **new** StringBuilder();

**for**(**char** c : s.toCharArray()){

**if**(vowels.contains(c)){

sb.append(vStack.pop());

}**else**{

sb.append(c);

}

}

**return** sb.toString();

}

}

Java Solution by Marking Indices of Vovels

I have first iterated through the string to mark the index where vowels are found and stored those indices in an array. I then swap the vowels in the original string using this array of indices.

**public** **class** Solution {

**public** String reverseVowels(String s) {

**char**[] charArray = s.toCharArray();

**int**[] temp = **new** **int**[charArray.length];

**int** j = 0;

HashSet<Character> vovels = **new** HashSet<>();

vovels.add('a');

vovels.add('e');

vovels.add('i');

vovels.add('o');

vovels.add('u');

vovels.add('A');

vovels.add('E');

vovels.add('I');

vovels.add('O');

vovels.add('U');

**for** (**int** i = 0; i < s.length(); i++){

**if** (vovels.contains(s.charAt(i))) temp[j++] = i;

}

**for** (**int** i = 0, k = j-1; i < k; i++, k--){

**char** c = charArray[temp[i]];

charArray[temp[i]] = charArray[temp[k]];

charArray[temp[k]] = c;

}

**return** **new** String(charArray);

}

}

### Valid Palindrome

[My Submissions](https://leetcode.com/problems/valid-palindrome/submissions/)

QuestionEditorial Solution

Total Accepted: **99623** Total Submissions: **418454** Difficulty: **Easy**

Given a string, determine if it is a palindrome, considering only alphanumeric characters 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.

我的做法

public boolean isPalindrome(String s) {

int len=s.length();

if(len==1 ||len==0) return true;

s=s.toLowerCase();

int low=0,high=len-1;

while(low<high)

{

char ch=s.charAt(low);

while(low<high && !((ch>='0' && ch<='9') || (ch>='a' && ch<='z')))

{

low++;

ch=s.charAt(low);

}

ch=s.charAt(high);

while(low<high && !((ch>='0' && ch<='9') || (ch>='a' && ch<='z')))

{

high--;

ch=s.charAt(high);

}

//System.out.println(low+" "+high);

if(s.charAt(low)!=s.charAt(high)) return false;

low++;

high--;//必须要有，否则会陷入死循环

}

return true;

}

bool isPalindrome(string s) {

**for** (int i = 0, j = s.size() - 1; i < j; i++, j--) { // Move 2 pointers from each **end** **until** they collide

**while** (isalnum(s[i]) == **false** && i < j) i++; // Increment left pointer **if** **not** alphanumeric

**while** (isalnum(s[j]) == **false** && i < j) j--; // Decrement right pointer **if** no alphanumeric

**if** (toupper(s[i]) != toupper(s[j])) **return** **false**; // Exit **and** **return** error **if** **not** match

}

**return** **true**;

}

这个写法不错，在while里写if，else。而不是在while里写while。这样可以不用总是判断指针的合理性。

**public** **class** **Solution** {

**public** **boolean** isPalindrome(String s) {

**if** (s.isEmpty()) {

**return** **true**;

}

**int** head = 0, tail = s.length() - 1;

**char** cHead, cTail;

**while**(head <= tail) {

cHead = s.charAt(head);

cTail = s.charAt(tail);

**if** (!Character.isLetterOrDigit(cHead)) {

head++;

} **else** **if**(!Character.isLetterOrDigit(cTail)) {

tail--;

} **else** {

**if** (Character.toLowerCase(cHead) != Character.toLowerCase(cTail)) {

**return** **false**;

}

head++;

tail--;

}

}

**return** **true**;

}

}

**public** **class** **Solution** {

**public** **boolean** isPalindrome(String s) {

String actual = s.replaceAll("[^A-Za-z0-9]", "").toLowerCase();

String rev = **new** StringBuffer(actual).reverse().toString();

**return** actual.equals(rev);

}

}

### Reverse String

[My Submissions](https://leetcode.com/problems/reverse-string/submissions/)

QuestionEditorial Solution

Total Accepted: **11610** Total Submissions: **19848** Difficulty: **Easy**

Write a function that takes a string as input and returns the string reversed.

**Example:**  
Given s = "hello", return "olleh".

|  |  |
| --- | --- |
| [**StringBuilder**](file:///E:\docsJava\api\java\lang\StringBuilder.html) | [**reverse**](file:///E:\docsJava\api\java\lang\StringBuilder.html#reverse--)()  Causes this character sequence to be replaced by the reverse of the sequence. |

String类没有reverse()方法

我的做法

public String reverseString(String s) {

return new StringBuilder(s).reverse().toString();

}

**Cheating Method using StringBuilder**

**public** **class** **Solution** {

**public** String reverseString(String s) {

**return** **new** StringBuilder(s).reverse().toString();

}

}

**Classic Method by swapping first and last**

**public** **class** **Solution** {

**public** String reverseString(String s) {

**char**[] word = s.toCharArray();

**int** i = 0;

**int** j = s.length() - 1;

**while** (i < j) {

**char** temp = word[i];

word[i] = word[j];

word[j] = temp;

i++;

j--;

}

**return** **new** String(word);

}

}

**Same as previous but using byte instead**

**public** **class** **Solution** {

**public** String reverseString(String s) {

**byte**[] bytes = s.getBytes();

**int** i = 0;

**int** j = s.length() - 1;

**while** (i < j) {

**byte** temp = bytes[i];

bytes[i] = bytes[j];

bytes[j] = temp;

i++;

j--;

}

**return** **new** String(bytes);

}

}

**Classic Method by swapping first and last**  
If you don't like temp variable

**public** **class** **Solution** {

**public** String reverseString(String s) {

**byte**[] bytes = s.getBytes();

**int** i = 0;

**int** j = s.length() - 1;

**while** (i < j) {

bytes[i] = (**byte**)(bytes[i] ^ bytes[j]);

bytes[j] = (**byte**)(bytes[i] ^ bytes[j]);

bytes[i] = (**byte**)(bytes[i] ^ bytes[j]);

i++;

j--;

}

**return** **new** String(bytes);

}

}

**Using recursion**

public **class** **Solution** {

public String reverseString(String s) {

int length = s.length();

**if** (length <= 1) **return** s;

String leftStr = s.substring(0, length / 2);

String rightStr = s.substring(length / 2, length);

**return** reverseString(rightStr) + reverseString(leftStr);

}

}

### Implement strStr()—KMP

[My Submissions](https://leetcode.com/problems/implement-strstr/submissions/)

QuestionEditorial Solution

Total Accepted: **104559** Total Submissions: **420143** Difficulty: **Easy**

Implement strStr().

Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

题目意思：返回字符串haystack中第一次出现needle的索引。

我的做法，从haystack中依次提取出与needle长度一样的子串，进行对比

public int strStr(String haystack, String needle) {

int len=needle.length();

for(int i=0;i+len-1<haystack.length();i++)

{

String str=haystack.substring(i,i+len);

if(str.equals(needle)) return i;

}

return -1;

}

a very clean solution, brute-force

**int** strStr(**char** \*haystack, **char** \*needle) {

**if** (!haystack || !needle) **return** -1;

**for** (**int** i = 0; ; ++i) {

**for** (**int** j = 0; ; ++j) {

**if** (needle[j] == 0) **return** i;

**if** (haystack[i + j] == 0) **return** -1;

**if** (haystack[i + j] != needle[j]) **break**;

}

}

}

这个方法也不错

**public** **int** strStr(String haystack, String needle) {

**for** (**int** i = 0; ; i++) {

**for** (**int** j = 0; ; j++) {

**if** (j == needle.length()) **return** i;

**if** (i + j == haystack.length()) **return** -1;

**if** (needle.charAt(j) != haystack.charAt(i + j)) **break**;

}

}

}

Accepted KMP solution in java for reference

public String strStr(String haystack, String needle) {

//KMP algorithms

**if**(needle.equals("")) **return** haystack;

**if**(haystack.equals("")) **return** null;

char[] arr = needle.toCharArray();

**int**[] **next** = makeNext(arr);

**for**(**int** i = 0, j = 0, end = haystack.**length**(); i < end;){

**if**(j == -1 || haystack.charAt(i) == arr[j]){

j++;

i++;

**if**(j == arr.**length**) **return** haystack.substring(i - arr.**length**);

}

**if**(i < end && haystack.charAt(i) != arr[j]) j = **next**[j];

}

**return** null;

}

private **int**[] makeNext(char[] arr){

**int** len = arr.**length**;

**int**[] **next** = new **int**[len];

**next**[0] = -1;

**for**(**int** i = 0, j = -1; i + 1 < len;){

**if**(j == -1 || arr[i] == arr[j]){

**next**[i+1] = j+1;

**if**(arr[i+1] == arr[j+1]) **next**[i+1] = **next**[j+1];

i++;

j++;

}

**if**(arr[i] != arr[j]) j = **next**[j];

}

**return** **next**;

}

### Longest Substring Without Repeating Characters

[My Submissions](https://leetcode.com/problems/longest-substring-without-repeating-characters/submissions/)

QuestionEditorial Solution

Total Accepted: **145845** Total Submissions: **663685** Difficulty: **Medium**

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.

题目意思：给定一个字符串，找到最长的不存在重复字符的子字符串

这个方法不错，有点类似滑动窗口，用两个指针控制窗口大小（即定义最大子串长度）

一个指针i用来遍历字符串，同时更新hashmap。

如果指针i指向的字母出现过，则更新指针j的位置。

11-line simple Java solution, O(n) with explanation

the basic idea is, keep a hashmap which stores the characters in string as keys and their positions as values, and keep two pointers which define the max substring. move the right pointer to scan through the string , and meanwhile update the hashmap. If the character is already in the hashmap, then move the left pointer to the right of the same character last found. Note that the two pointers can only move forward.

public int lengthOfLongestSubstring(String s) {

**if** (s.length()==0) **return** 0;

HashMap<Character, Integer> map = new HashMap<Character, Integer>();

int max=0;

**for** (int i=0, j=0; i<s.length(); ++i){

**if** (map.containsKey(s.charAt(i))){

j = Math.max(j,map.get(s.charAt(i))+1);

}

map.put(s.charAt(i),i);

max = Math.max(max,i-j+1);

}

**return** max;

}

The variable "j" is used to indicate the index of first character of this substring. If the repeated character's index is less than j itself, which means the repeated character in the hash map is no longer available this time

no need for O(n) space. Here is an O(n) time O(1) space solution using Kadane's algorithm

Idea is that, while we traverse form left to right if we see a character at position j is a duplicate of a character at a position i < j on the left then we know that we can't start the substring from i anymore. So, we need to start a new substring from i+1 position. While doing this we also need to update the length of current substring and start of current substring. Important part of this process is to make sure that we always keep the latest position of the characters we have seen so far. Below is a simple O(n) implementation of this logic.

**public** **class** **Solution** {

**public** **int** lengthOfLongestSubstring(String s) {

**int** lastIndices[] = **new** **int**[256];

**for**(**int** i = 0; i<256; i++){

lastIndices[i] = -1;

}

**int** maxLen = 0;

**int** curLen = 0;

**int** start = 0;

**int** bestStart = 0;

**for**(**int** i = 0; i<s.length(); i++){

**char** cur = s.charAt(i);

**if**(lastIndices[cur] < start){

lastIndices[cur] = i;

curLen++;

}

**else**{

**int** lastIndex = lastIndices[cur];

start = lastIndex+1;

curLen = i-start+1;

lastIndices[cur] = i;

}

**if**(curLen > maxLen){

maxLen = curLen;

bestStart = start;

}

}

**return** maxLen;

}

}

这个写法不错，如果新遇到的字符曾经出现过，则更新最左边的指针（指向子串的第一个字符）

Thanks for the intuitive answer. Here is the same algo with int[256] rather than HashMap. Faster than map and shorter code.

**public** **class** **Solution** {

**public** **int** lengthOfLongestSubstring(String s) {

**int** result = 0;

**int**[] cache = **new** **int**[256];

**for** (**int** i = 0, j = 0; i < s.length(); i++) {

j = (cache[s.charAt(i)] > 0) ? Math.max(j,

cache[s.charAt(i)]) : j;

cache[s.charAt(i)] = i + 1;//这里设置很巧妙

result = Math.max(result, i - j + 1);

}

**return** result;

}

}

Shortest O(n) DP solution with explanations

/\*\*

\* Solution (DP, O(n)):

\*

\* Assume L[i] = **s**[**m**...i], denotes the longest substring without repeating

\* characters that ends up at **s**[i], **and** we keep a hashmap **for** every

\* characters between **m** ... i, **while** storing <character, **index**> in the

\* hashmap.

\* We know that **each** character will appear only once.

\* Then to find **s**[i+1]:

\* 1) **if** **s**[i+1] does **not** appear in hashmap

\* we can just add **s**[i+1] to hash **map**. **and** L[i+1] = **s**[**m**...i+1]

\* 2) **if** **s**[i+1] **exists** in hashmap, **and** the hashmap value (the **index**) is k

\* let **m** = max(**m**, k), then L[i+1] = **s**[**m**...i+1], we also need to update

\* entry in hashmap to mark the latest occurency of **s**[i+1].

\*

\* Since we scan the string **for** only once, **and** the 'm' will also move from

\* beginning to end **for** at most once. Overall complexity is O(n).

\*

\* If characters are all in ASCII, we could **use** array to mimic hashmap.

\*/

**int** lengthOfLongestSubstring(string **s**) {

// **for** ASCII char sequence, **use** this as a hashmap

vector<**int**> charIndex(256, -1);

**int** longest = 0, **m** = 0;

**for** (**int** i = 0; i < **s**.**length**(); i++) {

**m** = max(charIndex[**s**[i]] + 1, **m**); // automatically takes care of -1 case

charIndex[**s**[i]] = i;

longest = max(longest, i - **m** + 1);

}

**return** longest;

}

**int** lengthOfLongestSubstring(**string** s) {

**vector**<**int**> dict(256, -1);

**int** maxLen = 0, start = -1;

**for** (**int** i = 0; i != s.length(); i++) {

**if** (dict[s[i]] > start)

start = dict[s[i]];

dict[s[i]] = i;

maxLen = max(maxLen, i - start);

}

**return** maxLen;

}

The idea is use a hash set to track the longest substring without repeating characters so far, use a fast pointer j to see if character j is in the hash set or not, if not, great, add it to the hash set, move j forward and update the max length, otherwise, delete from the head by using a slow pointer i until we can put character j to the hash set.

**public** **int** lengthOfLongestSubstring(String s) {

**int** i = 0, j = 0, max = 0;

Set<Character> **set** = **new** HashSet<>();

**while** (j < s.length()) {

**if** (!**set**.contains(s.charAt(j))) {

**set**.add(s.charAt(j++));

max = Math.max(max, **set**.size());

} **else** {

**set**.remove(s.charAt(i++));

}

}

**return** max;

}

### Container With Most Water

[My Submissions](https://leetcode.com/problems/container-with-most-water/submissions/)

QuestionEditorial Solution

Total Accepted: **76523** Total Submissions: **221142** Difficulty: **Medium**

Given *n* non-negative integers *a1*, *a2*, ..., *an*, where each represents a point at coordinate (*i*, *ai*). *n* vertical lines are drawn such that the two endpoints of line *i* is at (*i*, *ai*) and (*i*, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container.

我的做法

将两个指针分别指向数组两端，然后不断向中心移动。

（因为面积是由指针相距和两个竖线中短的那个为准）当左边指针当前指向的数较小时，则需要推进左边指针，反之，右边指针向中心移动，这样才有可能遇到最大面积值

public int maxArea(int[] height) {

int end=height.length-1;

int start=0;

int sum=0;

while(start<end)

{

sum=Math.max(sum,Math.min(height[start],height[end])\*(end-start));

if(height[start]<height[end]) start++;

else end--;

}

return sum;

}

Start by evaluating the widest container, using the first and the last line. All other possible containers are less wide, so to hold more water, they need to be higher. Thus, after evaluating that widest container, skip lines at both ends that don't support a higher height. Then evaluate that new container we arrived at. Repeat until there are no more possible containers left.

**C++**

**int** maxArea(**vector**<**int**>& height) {

**int** water = 0;

**int** i = 0, j = height.size() - 1;

**while** (i < j) {

**int** h = min(height[i], height[j]);

water = max(water, (j - i) \* h);

**while** (height[i] <= h && i < j) i++;

**while** (height[j] <= h && i < j) j--;

}

**return** water;

}

Easy Concise Java O(N) Solution with Proof and Explanation

AKA, the general idea to find some max is to go through all cases where max value can possibly occur and keep updating the max value. The efficiency of the scan depends on the size of cases you plan to scan. To increase efficiency, all we need to do is to find a smart way of scan to cut off the useless cases and meanwhile 100% guarantee the max value can be reached through the rest of cases.

In this problem, the smart scan way is to set two pointers initialized at both ends of the array. Every time move the smaller value pointer to inner array. Then after the two pointers meet, all possible max cases have been scanned and the max situation is 100% reached somewhere in the scan. Following is a brief prove of this.

Given a1,a2,a3.....an as input array. Lets assume a10 and a20 are the max area situation. We need to prove that a10 can be reached by left pointer and during the time left pointer stays at a10, a20 can be reached by right pointer. That is to say, the core problem is to prove: when left pointer is at a10 and right pointer is at a21, the next move must be right pointer to a20.

Since we are always moving the pointer with the smaller value, i.e. if a10 > a21, we should move pointer at a21 to a20, as we hope. Why a10 >a21? Because if a21>a10, then area of a10 and a20 must be less than area of a10 and a21. Because the area of a10 and a21 is at least height[a10] \* (21-10) while the area of a10 and a20 is at most height[a10] \* (20-10). So there is a contradiction of assumption a10 and a20 has the max area. So, a10 must be greater than a21, then next move a21 has to be move to a20. The max cases must be reached.

和我的做法同

**public** **int** maxArea(**int**[] height) {

**int** left = 0, right = height.length - 1;

**int** maxArea = 0;

**while** (left < right) {

maxArea = Math.max(maxArea, Math.min(height[left], height[right])\* (right - left));

**if** (height[left] < height[right])

left++;

**else**

right--;

}

**return** maxArea;

}

### Minimum Size Subarray Sum

[My Submissions](https://leetcode.com/problems/minimum-size-subarray-sum/submissions/)

QuestionEditorial Solution

Total Accepted: **37857** Total Submissions: **140774** Difficulty: **Medium**

Given an array of **n** positive integers and a positive integer **s**, find the minimal length of a subarray 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.

[click to show more practice.](https://leetcode.com/problems/minimum-size-subarray-sum/)

**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*).

题目意思：给定一个元素都是正的数组和一个正数s，找到最小长度的子数组，使和大于

**public** **class** **Solution** {

**public** **int** minSubArrayLen(**int** s, **int**[] nums) {

**return** solveNLogN(s, nums);

}

**private** **int** solveN(**int** s, **int**[] nums) {

**int** start = 0, end = 0, sum = 0, minLen = Integer.MAX\_VALUE;

**while** (end < nums.length) {

**while** (end < nums.length && sum < s) sum += nums[end++];

**if** (sum < s) **break**;

**while** (start < end && sum >= s) sum -= nums[start++];

**if** (end - start + 1 < minLen) minLen = end - start + 1;

}

**return** minLen == Integer.MAX\_VALUE ? 0 : minLen;

}

**private** **int** solveNLogN(**int** s, **int**[] nums) {

**int**[] sums = **new** **int**[nums.length + 1];

**for** (**int** i = 1; i < sums.length; i++) sums[i] = sums[i - 1] + nums[i - 1];

**int** minLen = Integer.MAX\_VALUE;

**for** (**int** i = 0; i < sums.length; i++) {

**int** end = binarySearch(i + 1, sums.length - 1, sums[i] + s, sums);

**if** (end == sums.length) **break**;

**if** (end - i < minLen) minLen = end - i;

}

**return** minLen == Integer.MAX\_VALUE ? 0 : minLen;

}

**private** **int** binarySearch(**int** lo, **int** hi, **int** key, **int**[] sums) {

**while** (lo <= hi) {

**int** mid = (lo + hi) / 2;

**if** (sums[mid] >= key){

hi = mid - 1;

} **else** {

lo = mid + 1;

}

}

**return** lo;

}

}

Since the given array contains only positive integers, the subarray sum can only increase by including more elements. Therefore, you don't have to include more elements once the current subarray already has a sum large enough. This gives the linear time complexity solution by maintaining a minimum window with a two indices.

As to NLogN solution, logN immediately reminds you of binary search. In this case, you cannot sort as the current order actually matters. How does one get an ordered array then? Since all elements are positive, the cumulative sum must be strictly increasing. Then, a subarray sum can expressed as the difference between two cumulative sum. Hence, given a start index for the cumulative sum array, the other end index can be searched using binary search.

这个写法清晰，一直移动j指针，直到sum大于目标值，才开始移动i，缩小长度差

public int minSubArrayLen(int s, int[] a) {

**if** (a == null || a.length == 0)

**return** 0;

int i = 0, j = 0, sum = 0, min = Integer.MAX\_VALUE;

**while** (j < a.length) {

sum += a[j++];

**while** (sum >= s) {

min = Math.min(min, j - i);//因为j在sum加了以后还会再自增1，所以这里不用+1，注意

sum -= a[i++];

}

}

**return** min == Integer.MAX\_VALUE ? 0 : min;

}

### Merge Sorted Array

[My Submissions](https://leetcode.com/problems/merge-sorted-array/submissions/)

QuestionEditorial Solution

Total Accepted: **98800** Total Submissions: **328777** Difficulty: **Easy**

Given two sorted integer arrays *nums1* and *nums2*, merge *nums2* into *nums1* as one sorted array.

**Note:**  
You may assume that *nums1* has enough space (size that is greater or equal to *m* + *n*) to hold additional elements from *nums2*. The number of elements initialized in *nums1* and *nums2* are *m* and *n* respectively.

和归并排序里面用了辅助数组不同，这里直接在nums1里面放置，所以为了避免原先的元素被覆盖，需要从后往前放数字

**class** **Solution** {

**public**:

**void** merge(**int** A[], **int** m, **int** B[], **int** n) {

**int** i=m-1;

**int** j=n-1;

**int** k = m+n-1;

**while**(i >=0 && j>=0)

{

**if**(A[i] > B[j])

A[k--] = A[i--];

**else**

A[k--] = B[j--];

}

**while**(j>=0)

A[k--] = B[j--];

}

};

## 智力题

### Bulb Switcher

[My Submissions](https://leetcode.com/problems/bulb-switcher/submissions/)

QuestionEditorial Solution

Total Accepted: **16288** Total Submissions: **40474** Difficulty: **Medium**

There are *n* bulbs that are initially off. You first turn on all the bulbs. Then, you turn off every second bulb. On the third round, you toggle every third bulb (turning on if it's off or turning off if it's on). For the *i*th round, you toggle every *i* bulb. For the *n*th round, you only toggle the last bulb. Find how many bulbs are on after*n* rounds.

**Example:**

Given *n* = 3.

At first, the three bulbs are **[off, off, off]**.

After first round, the three bulbs are **[on, on, on]**.

After second round, the three bulbs are **[on, off, on]**.

After third round, the three bulbs are **[on, off, off]**.

So you should return 1, because there is only one bulb is on.

书

public int bulbSwitch(int n) {

int cnt=0;

int num=1;

while(num\*num<=n)

{

cnt++;

num++;

//System.out.println(num);

}

return cnt;

}

### Nim Game

[My Submissions](https://leetcode.com/problems/nim-game/submissions/)

QuestionEditorial Solution

Total Accepted: **62910** Total Submissions: **119538** Difficulty: **Easy**

You are playing the following Nim Game with your friend: There is a heap of stones on the table, each time one of you take turns to remove 1 to 3 stones. The one who removes the last stone will be the winner. You will take the first turn to remove the stones.

Both of you are very clever and have optimal strategies for the game. Write a function to determine whether you can win the game given the number of stones in the heap.

For example, if there are 4 stones in the heap, then you will never win the game: no matter 1, 2, or 3 stones you remove, the last stone will always be removed by your friend.

**Hint:**

1. If there are 5 stones in the heap, could you figure out a way to remove the stones such that you will always be the winner?

努力凑数，使对方剩余的子数为4的倍数（这个情况会输）

博弈论中极为经典的尼姆游戏。有总数为n的石头，每个人可以拿1~m个石头，两个人交替拿，拿到最后一个的人获胜。究竟是先手有利，还是后手有利？ 

1. 1个石子，先手全部拿走；
2. 2个石子，先手全部拿走；
3. 3个石子，先手全部拿走；
4. 4个石子，后手面对的是先手的第1，2，3情况，后手必胜；
5. 5个石子，先手拿走1个让后手面对第4种情况，后手必败；
6. 6个石子，先手拿走2个让后手面对第4种情况，后手必败；
7. ……

容易看出来，只有当出现了4的倍数，先手无可奈何，其余情况先手都可以获胜。

（石子数量为4的倍数）后手的获胜策略十分简单，每次取石子的数量，与上一次先手取石子的数量和为4即可；   
（石子数量不为4的倍数）先手的获胜策略也十分简单，每次都令取之后剩余的石子数量为4的倍数（4\*0=0，直接拿光），他就处于后手的位置上，利用上一行的策略获胜。

## 二叉排序树

## 线段树

## 二叉线索树

### Range Sum Query - Mutable

[My Submissions](https://leetcode.com/problems/range-sum-query-mutable/submissions/)

QuestionEditorial Solution

Total Accepted: **9208** Total Submissions: **53698** Difficulty: **Medium**

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *j*), inclusive.

The *update(i, val)* function modifies *nums* by updating the element at index *i* to *val*.

**Example:**

Given nums = [1, 3, 5]

sumRange(0, 2) -> 9

update(1, 2)

sumRange(0, 2) -> 8

**Note:**

1. The array is only modifiable by the *update* function.
2. You may assume the number of calls to *update* and *sumRange* function is distributed evenly.

17 ms Java solution with segment tree

**1：概述**

**线段树，类似区间树，是一个完全二叉树，它在各个节点保存一条线段（数组中的一段子数组），主要用于高效解决连续区间的动态查询问题，由于二叉结构的特性，它基本能保持每个操作的复杂度为O(lgN)!**

**性质：父亲的区间是[a,b],(c=(a+b)/2)左儿子的区间是[a,c]，右儿子的区间是[c+1,b]，线段树需要的空间为数组大小的四倍**

**2：基本操作（demo用的是查询区间最小值）**

**线段树的主要操作有：**

**(1)：线段树的构造 void build(int node, int begin, int end);**

**主要思想是递归构造，如果当前节点记录的区间只有一个值，则直接赋值，否则递归构造左右子树，最后回溯的时候给当前节点赋值**

public **class** **NumArray** {

**class** **SegmentTreeNode** {

int start, **end**;

SegmentTreeNode left, right;

int sum;

public SegmentTreeNode(int start, int **end**) {

this.start = start;

this.**end** = **end**;

this.left = null;

this.right = null;

this.sum = 0;

}

}

SegmentTreeNode root = null;

public NumArray(int[] nums) {

root = buildTree(nums, 0, nums.length-1);

}

private SegmentTreeNode buildTree(int[] nums, int start, int **end**) {

**if** (start > **end**) {

**return** null;

} **else** {

SegmentTreeNode ret = new SegmentTreeNode(start, **end**);

**if** (start == **end**) {

ret.sum = nums[start];

} **else** {

int mid = start + (**end** - start) / 2;

ret.left = buildTree(nums, start, mid);

ret.right = buildTree(nums, mid + 1, **end**);

ret.sum = ret.left.sum + ret.right.sum;

}

**return** ret;

}

}

void update(int i, int val) {

update(root, i, val);

}

void update(SegmentTreeNode root, int pos, int val) {

**if** (root.start == root.**end**) {

root.sum = val;

} **else** {

int mid = root.start + (root.**end** - root.start) / 2;

**if** (pos <= mid) {

update(root.left, pos, val);

} **else** {

update(root.right, pos, val);

}

root.sum = root.left.sum + root.right.sum;

}

}

public int sumRange(int i, int j) {

**return** sumRange(root, i, j);

}

public int sumRange(SegmentTreeNode root, int start, int **end**) {

**if** (root.**end** == **end** && root.start == start) {

**return** root.sum;

} **else** {

int mid = root.start + (root.**end** - root.start) / 2;

**if** (**end** <= mid) {

**return** sumRange(root.left, start, **end**);

} **else** **if** (start >= mid+1) {

**return** sumRange(root.right, start, **end**);

} **else** {

**return** sumRange(root.right, mid+1, **end**) + sumRange(root.left, start, mid);

}

}

}

}

Java using Binary Indexed Tree with clear explanation

This is to share the explanation of the BIT and the meaning of the bit operations.

**public** **class** **NumArray** {

/\*\*

\* Binary Indexed Trees (BIT or Fenwick tree):

\* https://www.topcoder.com/community/data-science/data-science-

\* tutorials/binary-indexed-trees/

\*

\* Example: given an array a[0]...a[7], we use a array BIT[9] to

\* represent a tree, where index [2] is the parent of [1] and [3], [6]

\* is the parent of [5] and [7], [4] is the parent of [2] and [6], and

\* [8] is the parent of [4]. I.e.,

\*

\* BIT[] as a binary tree:

\* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\*

\* \_\_\_\_\_\_\*

\* \_\_\* \_\_\*

\* \* \* \* \*

\* indices: 0 1 2 3 4 5 6 7 8

\*

\* BIT[i] = ([i] is a left child) ? the partial sum from its left most

\* descendant to itself : the partial sum from its parent (exclusive) to

\* itself. (check the range of "\_\_").

\*

\* Eg. BIT[1]=a[0], BIT[2]=a[1]+BIT[1]=a[1]+a[0], BIT[3]=a[2],

\* BIT[4]=a[3]+BIT[3]+BIT[2]=a[3]+a[2]+a[1]+a[0],

\* BIT[6]=a[5]+BIT[5]=a[5]+a[4],

\* BIT[8]=a[7]+BIT[7]+BIT[6]+BIT[4]=a[7]+a[6]+...+a[0], ...

\*

\* Thus, to update a[1]=BIT[2], we shall update BIT[2], BIT[4], BIT[8],

\* i.e., for current [i], the next update [j] is j=i+(i&-i) //double the

\* last 1-bit from [i].

\*

\* Similarly, to get the partial sum up to a[6]=BIT[7], we shall get the

\* sum of BIT[7], BIT[6], BIT[4], i.e., for current [i], the next

\* summand [j] is j=i-(i&-i) // delete the last 1-bit from [i].

\*

\* To obtain the original value of a[7] (corresponding to index [8] of

\* BIT), we have to subtract BIT[7], BIT[6], BIT[4] from BIT[8], i.e.,

\* starting from [idx-1], for current [i], the next subtrahend [j] is

\* j=i-(i&-i), up to j==idx-(idx&-idx) exclusive. (However, a quicker

\* way but using extra space is to store the original array.)

\*/

**int**[] nums;

**int**[] BIT;

**int** n;

**public** NumArray(**int**[] nums) {

**this**.nums = nums;

n = nums.length;

BIT = **new** **int**[n + 1];

**for** (**int** i = 0; i < n; i++)

init(i, nums[i]);

}

**public** **void** init(**int** i, **int** val) {

i++;

**while** (i <= n) {

BIT[i] += val;

i += (i & -i);

}

}

**void** update(**int** i, **int** val) {

**int** diff = val - nums[i];

nums[i] = val;

init(i, diff);

}

**public** **int** getSum(**int** i) {

**int** sum = 0;

i++;

**while** (i > 0) {

sum += BIT[i];

i -= (i & -i);

}

**return** sum;

}

**public** **int** sumRange(**int** i, **int** j) {

**return** getSum(j) - getSum(i - 1);

}

}

// Your NumArray object will be instantiated and called as such:

// NumArray numArray = new NumArray(nums);

// numArray.sumRange(0, 1);

// numArray.update(1, 10);

// numArray.sumRange(1, 2);

## 并查集

### Number of Islands

[My Submissions](https://leetcode.com/problems/number-of-islands/submissions/)

QuestionEditorial Solution

Total Accepted: **42717** Total Submissions: **155117** Difficulty: **Medium**

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:***

11110  
11010  
11000  
00000

Answer: 1

***Example 2:***

11000  
11000  
00100  
00011

Answer: 3

class UF {

**public** **int** count = 0;

**public** **int**[] id = **null**;

**public** UF(**int** m, **int** n, **char**[][] grid) {

**for**(**int** i = 0; i < m; i++) {

**for**(**int** j = 0; j < n; j++) {

**if**(grid[i][j] == '1') count++;

}

}

id = **new** **int**[m \* n];

**for**(**int** i = 0; i < m \* n; i++) {

id[i] = i;

}

}

**public** **int** find(**int** p) {

**while**(p != id[p]) {

id[p] = id[id[p]];

p = id[p];

}

**return** p;

}

**public** **boolean** isConnected(**int** p, **int** q) {

**int** pRoot = find(p);

**int** qRoot = find(q);

**if**(pRoot != qRoot) **return** **false**;

**else** **return** **true**;

}

**public** **void** union(**int** p, **int** q) {

**int** pRoot = find(p);

**int** qRoot = find(q);

**if**(pRoot == qRoot) **return**;

id[pRoot] = qRoot; //p节点的父亲要改为q节点

count--;

}

}

//首先统计图中有多少1，作为独立个体。然后为图中所有元素建立根。

//遍历图中的每一个元素，如果该元素是1，则向四周合并

//为图中每个元素建立标号（m\*n），利用这个标号来标记父亲

**public** **int** numIslands(**char**[][] grid) {

**if**(grid.length == 0 || grid[0].length == 0) **return** 0;

**int** m = grid.length, n = grid[0].length;

UF uf = **new** UF(m , n, grid);

**for**(**int** i = 0; i < m; i++) {

**for**(**int** j = 0; j < n; j++) {

**if**(grid[i][j] == '0') **continue**;

**int** p = i \* n + j;

**int** q;

**if**(i > 0 && grid[i - 1][j] == '1') {

q = p - n;

uf.**union**(p, q);

}

**if**(i < m - 1 && grid[i + 1][j] == '1') {

q = p + n;

uf.**union**(p, q);

}

**if**(j > 0 && grid[i][j - 1] == '1') {

q = p - 1;

uf.**union**(p, q);

}

**if**(j < n - 1 && grid[i][j + 1] == '1') {

q = p + 1;

uf.**union**(p, q);

}

}

}

**return** uf.count;

}

## 排序

## 数学

### Count Primes

[My Submissions](https://leetcode.com/problems/count-primes/submissions/)

QuestionEditorial Solution

Total Accepted: **60134** Total Submissions: **248798** Difficulty: **Easy**

**Description:**

Count the number of prime numbers less than a non-negative number, ***n***.

**Credits:**  
Special thanks to [@mithmatt](https://leetcode.com/discuss/user/mithmatt) for adding this problem and creating all test cases.

**Hint:**

1. Let's start with a *isPrime* function. To determine if a number is prime, we need to check if it is not divisible by any number less than *n*. The runtime complexity of *isPrime* function would be O(*n*) and hence counting the total prime numbers up to *n* would be O(*n*2). Could we do better?
2. As we know the number must not be divisible by any number > *n* / 2, we can immediately cut the total iterations half by dividing only up to*n* / 2. Could we still do better?
3. Let's write down all of 12's factors:
4. 2 × 6 = 12
5. 3 × 4 = 12
6. 4 × 3 = 12
7. 6 × 2 = 12

As you can see, calculations of 4 × 3 and 6 × 2 are not necessary. Therefore, we only need to consider factors up to √*n* because, if *n* is divisible by some number *p*, then *n* = *p* × *q* and since *p* ≤ *q*, we could derive that *p* ≤ √*n*.

Our total runtime has now improved to O(*n*1.5), which is slightly better. Is there a faster approach?

public int countPrimes(int n) {

int count = 0;

for (int i = 1; i < n; i++) {

if (isPrime(i)) count++;

}

return count;

}

private boolean isPrime(int num) {

if (num <= 1) return false;

// Loop's ending condition is i \* i <= num instead of i <= sqrt(num)

// to avoid repeatedly calling an expensive function sqrt().

for (int i = 2; i \* i <= num; i++) {

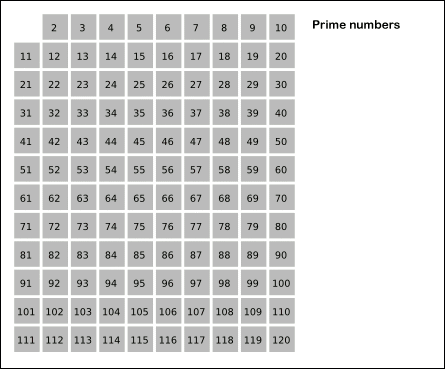
if (num % i == 0) return false;

}

return true;

}

1. The [Sieve of Eratosthenes](http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes) is one of the most efficient ways to find all prime numbers up to *n*. But don't let that name scare you, I promise that the concept is surprisingly simple.

  
Sieve of Eratosthenes: algorithm steps for primes below 121. "[Sieve of Eratosthenes Animation](http://commons.wikimedia.org/wiki/File:Sieve_of_Eratosthenes_animation.gif)" by [SKopp](http://de.wikipedia.org/wiki/Benutzer:SKopp) is licensed under [CC BY 2.0](http://creativecommons.org/licenses/by/2.0/).

We start off with a table of *n* numbers. Let's look at the first number, 2. We know all multiples of 2 must not be primes, so we mark them off as non-primes. Then we look at the next number, 3. Similarly, all multiples of 3 such as 3 × 2 = 6, 3 × 3 = 9, ... must not be primes, so we mark them off as well. Now we look at the next number, 4, which was already marked off. What does this tell you? Should you mark off all multiples of 4 as well?

1. 4 is not a prime because it is divisible by 2, which means all multiples of 4 must also be divisible by 2 and were already marked off. So we can skip 4 immediately and go to the next number, 5. Now, all multiples of 5 such as 5 × 2 = 10, 5 × 3 = 15, 5 × 4 = 20, 5 × 5 = 25, ... can be marked off. There is a slight optimization here, we do not need to start from 5 × 2 = 10. Where should we start marking off?
2. In fact, we can mark off multiples of 5 starting at 5 × 5 = 25, because 5 × 2 = 10 was already marked off by multiple of 2, similarly 5 × 3 = 15 was already marked off by multiple of 3. Therefore, if the current number is *p*, we can always mark off multiples of *p* starting at *p*2, then in increments of *p*: *p*2 + *p*, *p*2 + 2*p*, ... Now what should be the terminating loop condition?
3. It is easy to say that the terminating loop condition is *p* < *n*, which is certainly correct but not efficient. Do you still remember *Hint #3*?
4. Yes, the terminating loop condition can be *p* < √*n*, as all non-primes ≥ √*n* must have already been marked off. When the loop terminates, all the numbers in the table that are non-marked are prime.

The Sieve of Eratosthenes uses an extra O(*n*) memory and its runtime complexity is O(*n* log log *n*). For the more mathematically inclined readers, you can read more about its algorithm complexity on [Wikipedia](http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes#Algorithm_complexity).

public int countPrimes(int n) {

boolean[] isPrime = new boolean[n];

for (int i = 2; i < n; i++) {

isPrime[i] = true;

}

// Loop's ending condition is i \* i < n instead of i < sqrt(n)

// to avoid repeatedly calling an expensive function sqrt().

for (int i = 2; i \* i < n; i++) {

if (!isPrime[i]) continue;

for (int j = i \* i; j < n; j += i) {

isPrime[j] = false;

}

}

int count = 0;

for (int i = 2; i < n; i++) {

if (isPrime[i]) count++;

}

return count;

}

### Reverse Integer

[My Submissions](https://leetcode.com/problems/reverse-integer/submissions/)

QuestionEditorial Solution

Total Accepted: **136895** Total Submissions: **579679** Difficulty: **Easy**

Reverse digits of an integer.

**Example1:** x = 123, return 321  
**Example2:** x = -123, return -321

[click to show spoilers.](https://leetcode.com/problems/reverse-integer/)

**Have you thought about this?**

Here are some good questions to ask before coding. Bonus points for you if you have already thought through this!

If the integer's last digit is 0, what should the output be? ie, cases such as 10, 100.

Did you notice that the reversed integer might overflow? Assume the input is a 32-bit integer, then the reverse of 1000000003 overflows. How should you handle such cases?

For the purpose of this problem, assume that your function returns 0 when the reversed integer overflows.

**Update (2014-11-10):**  
Test cases had been added to test the overflow behavior.

我的做法

public int reverse(int x) {

long n=0;

while(x!=0)

{

n=n\*10+x%10;

x/=10;

}

if(n>Integer.MAX\_VALUE ||n<Integer.MIN\_VALUE) return 0;

return (int)n;

}

Only 15 lines. If overflow exists, the new result will not equal previous one. No flags needed. No hard code like 0xf7777777 needed. Sorry for my bad english.

**public** **int** reverse(**int** x)

{

**int** result = 0;

**while** (x != 0)

{

**int** tail = x % 10;

**int** newResult = result \* 10 + tail;

**if** ((newResult - tail) / 10 != result)

{ **return** 0; }

result = newResult;

x = x / 10;

}

**return** result;

}

Is there any case happen to be equal? The answer is no. the provement is a little tricky. Imagine that "newResult" and "result" are numbers in pure maths(so no overflow happens), then "newResult" is linear with "result" for all values of "result". Now back to computer maths, "newResult" is linear with "result" for values that are not too big(thus no overflow happens). But when "result" is too big, the "newResult" value stored in memory is less than the pure maths' version, thus "(newResult - tail) / 10" can not go back to "result". Hope I wrote what's in my mind correctly.

I got confused by the same question for really long, and know I got the answer. (newResult - tail) / 10 != result is not universal for testing overflow. If the result of result\*10 causes no overflow, but overflow only happens when it plus tail, (newResult - tail) / 10 would still be the last-step result. That's why the method doesn't apply to puzzle 8. However, for this puzzle, such case is not possible. Because the largest number as an int is 2147483647. The only cases that result in above-mentioned overflow are 2147483648 and 2147483649, but both would require original input(their reverse) to be out of int range.

## 数据库

### Employees Earning More Than Their Managers

[My Submissions](https://leetcode.com/problems/employees-earning-more-than-their-managers/submissions/)

QuestionEditorial Solution

Total Accepted: **18460** Total Submissions: **44877** Difficulty: **Easy**

The Employee table holds all employees including their managers. Every employee has an Id, and there is also a column for the manager Id.

+----+-------+--------+-----------+

| Id | Name | Salary | ManagerId |

+----+-------+--------+-----------+

| 1 | Joe | 70000 | 3 |

| 2 | Henry | 80000 | 4 |

| 3 | Sam | 60000 | NULL |

| 4 | Max | 90000 | NULL |

+----+-------+--------+-----------+

Given the Employee table, write a SQL query that finds out employees who earn more than their managers. For the above table, Joe is the only employee who earns more than his manager.

+----------+

| Employee |

+----------+

| Joe |

+----------+

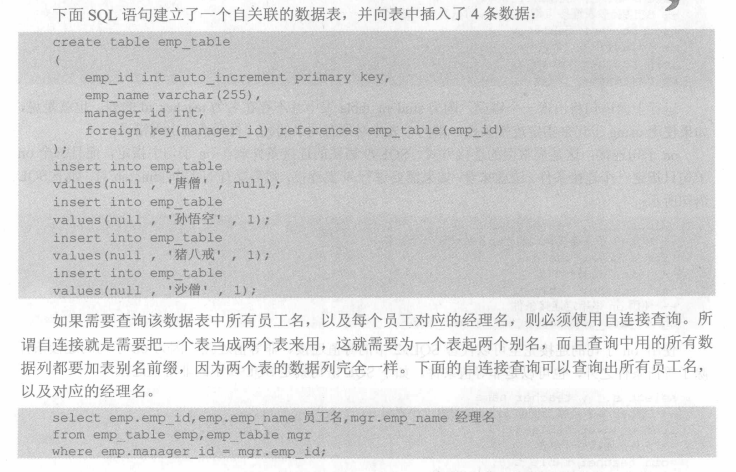
我的做法

select emp.Name from Employee emp,Employee mgr

where emp.ManagerId=mgr.Id **and** emp.Salary>mgr.Salary;

自连接，需要把同一个表当成两个表用，需要为一个表起两个别名，数据列都需要加前缀。





**select** E1.Name

**from** Employee **as** E1, Employee **as** E2

**where** E1.ManagerId = E2.Id **and** E1.Salary > E2.Salary

**SELECT** employer.Name

**FROM** Employee employer **JOIN** Employee manager **ON** (employer.ManagerId = manager.Id )

**WHERE** employer.Salary > manager.Salary ;

### Duplicate Emails

[My Submissions](https://leetcode.com/problems/duplicate-emails/submissions/)

QuestionEditorial Solution

Total Accepted: **17653** Total Submissions: **46205** Difficulty: **Easy**

Write a SQL query to find all duplicate emails in a table named Person.

+----+---------+

| Id | Email |

+----+---------+

| 1 | a@b.com |

| 2 | c@d.com |

| 3 | a@b.com |

+----+---------+

For example, your query should return the following for the above table:

+---------+

| Email |

+---------+

| a@b.com |

+---------+

我的做法

按Email列进行分组，使用having对分组进行过滤，只保留Email个数大于1的分组

select Email

from Person

group by Email

having count(\*)>1;

Your answer is the standard way and should be the most efficient. I also know three other ways as shown below:

1. Use self join.
2. **SELECT** **DISTINCT** a.Email
3. **FROM** Person a **JOIN** Person b
4. **ON** (a.Email = b.Email)
5. **WHERE** a.Id <> b.Id
6. Use subquery with [EXISTS](http://dev.mysql.com/doc/refman/5.0/en/exists-and-not-exists-subqueries.html):
7. **SELECT** **DISTINCT** a.Email
8. **FROM** Person a
9. **WHERE** EXISTS(
10. **SELECT** 1
11. **FROM** Person b
12. **WHERE** a.Email = b.Email
13. LIMIT 1, 1
14. )
15. Basic idea is this query:
16. **SELECT** **DISTINCT** Email **FROM** Person
17. MINUS
18. (**SELECT** Id, Email **FROM** Person **GROUP** **BY** Email)

But since MySQL does not support MINUS, we use LEFT JOIN:

**SELECT** **DISTINCT** a.Email **FROM** Person a

**LEFT** **JOIN** (**SELECT** Id, Email **from** Person **GROUP** **BY** Email) b

**ON** (a.email = b.email) **AND** (a.Id = b.Id)

**WHERE** b.Email **IS** **NULL**

### Combine Two Tables

[My Submissions](https://leetcode.com/problems/combine-two-tables/submissions/)

QuestionEditorial Solution

Total Accepted: **19557** Total Submissions: **56850** Difficulty: **Easy**

Table: Person

+-------------+---------+

| Column Name | Type |

+-------------+---------+

| PersonId | int |

| FirstName | varchar |

| LastName | varchar |

+-------------+---------+

PersonId is the primary key column for this table.

Table: Address

+-------------+---------+

| Column Name | Type |

+-------------+---------+

| AddressId | int |

| PersonId | int |

| City | varchar |

| State | varchar |

+-------------+---------+

AddressId is the primary key column for this table.

Write a SQL query for a report that provides the following information for each person in the Person table, regardless if there is an address for each of those people:

FirstName, LastName, City, State

我的做法

select FirstName, LastName, City, State

from Person left join Address on Person.PersonId=Address.PersonId;

### Customers Who Never Order

[My Submissions](https://leetcode.com/problems/customers-who-never-order/submissions/)

QuestionEditorial Solution

Total Accepted: **17033** Total Submissions: **50561** Difficulty: **Easy**

Suppose that a website contains two tables, the Customers table and the Orders table. Write a SQL query to find all customers who never order anything.

Table: Customers.

+----+-------+

| Id | Name |

+----+-------+

| 1 | Joe |

| 2 | Henry |

| 3 | Sam |

| 4 | Max |

+----+-------+

Table: Orders.

+----+------------+

| Id | CustomerId |

+----+------------+

| 1 | 3 |

| 2 | 1 |

+----+------------+

Using the above tables as example, return the following:

+-----------+

| Customers |

+-----------+

| Henry |

| Max |

+-----------+

**SELECT** A.Name **from** Customers A

**WHERE** **NOT** EXISTS (**SELECT** 1 **FROM** Orders B **WHERE** A.Id = B.CustomerId)

**SELECT** A.Name

**from** Customers A **LEFT** **JOIN** Orders B **on** a.Id = B.CustomerId

**WHERE** b.CustomerId **is** **NULL（不可以用=null）**

**SELECT** A.Name **from** Customers A

**WHERE** A.Id **NOT** **IN** (**SELECT** B.CustomerId **from** Orders B)

The fastest would be (#2 in your comment):

### Rising Temperature

[My Submissions](https://leetcode.com/problems/rising-temperature/submissions/)

QuestionEditorial Solution

Total Accepted: **13800** Total Submissions: **52994** Difficulty: **Easy**

Given a Weather table, write a SQL query to find all dates' Ids with higher temperature compared to its previous (yesterday's) dates.

+---------+------------+------------------+

| Id(INT) | Date(DATE) | Temperature(INT) |

+---------+------------+------------------+

| 1 | 2015-01-01 | 10 |

| 2 | 2015-01-02 | 25 |

| 3 | 2015-01-03 | 20 |

| 4 | 2015-01-04 | 30 |

+---------+------------+------------------+

For example, return the following Ids for the above Weather table:

+----+

| Id |

+----+

| 2 |

| 4 |

+----+

**SELECT** wt1.Id

**FROM** Weather wt1, Weather wt2

**WHERE** wt1.Temperature > wt2.Temperature **AND**

TO\_DAYS(wt1.**DATE**)-TO\_DAYS(wt2.**DATE**)=1;

EXPLANATION:

**TO\_DAYS(wt1.DATE)** return the number of days between from year 0 to date DATE**TODAYS(wt1.DATE)-TODAYS(wt2.DATE)=1** check if wt2.DATE is yesterday respect to wt1.DATE

We select from the joined tables the rows that have

**wt1.Temperature > wt2.Temperature**

and difference between dates in days of 1 (yesterday):

**TODAYS(wt1.DATE)-TODAYS(wt2.DATE)=1;**

**SELECT** a.Id **FROM** Weather **AS** a, Weather **AS** b

**WHERE** DATEDIFF(a.**Date**, b.**Date**)=1 **AND** a.Temperature > b.Temperature

Datediff小写也可以

### Second Highest Salary

[My Submissions](https://leetcode.com/problems/second-highest-salary/submissions/)

QuestionEditorial Solution

Total Accepted: **17834** Total Submissions: **71372** Difficulty: **Easy**

Write a SQL query to get the second highest salary from the Employee table.

+----+--------+

| Id | Salary |

+----+--------+

| 1 | 100 |

| 2 | 200 |

| 3 | 300 |

+----+--------+

For example, given the above Employee table, the second highest salary is 200. If there is no second highest salary, then the query should return null.

SELECT max(Salary)

FROM Employee

WHERE Salary < (SELECT max(Salary) FROM Employee)

Using max() will return a NULL if the value doesn't exist. So there is no need to UNION a NULL. Of course, if the second highest value is guaranteed to exist, using LIMIT 1,1 will be the best answer.

**what if we want to get the third highest ? and fourth …**

select IFNULL( (select distinct e1.salary from Employee e1 where (select count(distinct e2.salary ) from Employee e2 where e2.salary > e1.salary) = 1) , null)

**select** (

**select** **distinct** Salary

**from** Employee

**order** **by** Salary **Desc**

limit 1 offset 1

)**as** **second**

Change the number after 'offset' gives u n-th highest salary

1.Select Distinct 货品编码,数量 From 订单信息  
2.Select Distinct 货品编码 From 订单信息  
第一句和第二句有什么分别，请高手说说Distinct用法

Distinct的是作用是过滤结果集中的重复值。  
  
比如订单信息表中有3条信息：  
货品编码 数量  
001 10  
001 20  
002 10  
  
如果用第一个[SQL语句](https://www.baidu.com/s?wd=SQL%E8%AF%AD%E5%8F%A5&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dBPjR3P103rAcLnvw-rH790ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHb4nHcdP1Td)查询的话，会返回  
货品编码 数量  
001 10  
001 20  
002 10  
  
而用第二个则返回  
货品编码   
001   
002

SQL limit offset

经常用到在数据库中查询中间几条数据的需求

比如下面的sql语句：

① select \* from testtable limit 2,1;

② select \* from testtable limit 2 offset 1;

这两个都是能完成需要，但是他们之间是有区别的：

①是从数据库中第三条开始查询，取一条数据，即第三条数据

而这个SQL，limit后面是从第2条开始读，读取1条信息。

②是从数据库中的第二条数据开始查询两条数据，即第二条和第三条。

比如这个SQL ，limit后面跟的是2条数据，offset后面是从第1条开始读取

SELECT  \*   FROM trom\_data WHERE device\_id=12  ORDER BY time\_stamp DESC LIMIT  1   OFFSET 0

sql语句，其中LIMIT 1 OFFSET 0的意思是说在查询结果中以第0条记录为基准（包括第0条），取1条记录，这样所取得的记录即为第0条记录，也即此表中device\_id=12  且time\_stamp为最大的时间。

LIMIT  所要取的记录数目（以基准点为参考点） OFFSET  基准点

SELECT  \*   FROM trom\_data  LIMIT  nNumRecord   OFFSET nBaseRow

表示从第nBaseRow行(基于0的索引)(包括该行)开始,取其后的nNumRecord  条记录

### Delete Duplicate Emails

[My Submissions](https://leetcode.com/problems/delete-duplicate-emails/submissions/)

QuestionEditorial Solution

Total Accepted: **11872** Total Submissions: **61910** Difficulty: **Easy**

Write a SQL query to delete all duplicate email entries in a table named Person, keeping only unique emails based on its *smallest* **Id**.

+----+------------------+

| Id | Email |

+----+------------------+

| 1 | john@example.com |

| 2 | bob@example.com |

| 3 | john@example.com |

+----+------------------+

Id is the primary key column for this table.

For example, after running your query, the above Person table should have the following rows:

+----+------------------+

| Id | Email |

+----+------------------+

| 1 | john@example.com |

| 2 | bob@example.com |

+----+------------------+

*DELETE p1  
FROM Person p1, Person p2  
WHERE p1.Email = p2.Email AND p1.Id > p2.Id*

EXPLANATION:

* Take the table in the example

**Id | Email**

**1 | john@example.com**

**2 | bob@example.com**

**3 | john@example.com**

* Join the table on itself by the Email and you'll get:

*FROM Person p1, Person p2 WHERE p1.Email = p2.Email*

**p1.Id | p1.Email | p2.Id | p2.Email**

**1 | john@example.com | 1 | john@example.com**

**3 | john@example.com | 1 | john@example.com**

**2 | bob@example.com | 2 | bob@example.com**

**1 | john@example.com | 3 | john@example.com**

**3 | john@example.com | 3 | john@example.com**

* From this results filter the records that have p1.Id>p2.ID, in this case you'll get just one record:

*AND p1.Id > p2.Id*

**p1.Id | p1.Email | p2.Id | p2.Email**

**3 | john@example.com | 1 | john@example.com**

* This is the record we need to delete, and by saying

*DELETE p1*

in this multiple-table syntax, only matching rows from the tables listed before the FROM clause are deleted, in this case just

**p1.Id | p1.Email**

**3 | john@example.com**

will be deleted

delete from Person where Id not in ( select A.Id from (select min(Id) as Id from Person GROUP BY Email) A )

A skillful mysql solution avoid " select and update conflict"

**where we try this clause :**

**delete** **from** Person **where** id **not** **in**(**select** **min**(id) **as** id **from** Person **group** **by** email)

you will be noted " **You can't specify target table 'Person' for update in FROM clause** ", The solution is using a middle table with select clause:

**delete** **from** Person **where** id **not** **in**(

**select** t.id **from** (

**select** **min**(id) **as** id **from** Person **group** **by** email

) t

)

有两个关键点：设子查询的结果为表t，将子查询的结果字段设为id。

### 中级

### Consecutive Numbers

[My Submissions](https://leetcode.com/problems/consecutive-numbers/submissions/)

QuestionEditorial Solution

Total Accepted: **8342** Total Submissions: **31330** Difficulty: **Medium**

Write a SQL query to find all numbers that appear at least three times consecutively.

+----+-----+

| Id | Num |

+----+-----+

| 1 | 1 |

| 2 | 1 |

| 3 | 1 |

| 4 | 2 |

| 5 | 1 |

| 6 | 2 |

| 7 | 2 |

+----+-----+

For example, given the above Logs table, 1 is the only number that appears consecutively for at least three times.

**Select** **DISTINCT** l1.Num **from** Logs l1, Logs l2, Logs l3

**where** l1.Id=l2.Id-1 **and** l2.Id=l3.Id-1

**and** l1.Num=l2.Num **and** l2.Num=l3.Num

这个写法好

select distinct(a.Num) from Logs a, Logs b,Logs c where a.Id=b.Id+1 and a.Num=b.Num and b.Id=c.Id+1 and b.Num=c.Num

满足a表中，第i行记录的数与b表中第i-1行记录的数相同，与c表中第i-2行记录的数相同（即三个连续的数）

### Rank Scores

[My Submissions](https://leetcode.com/problems/rank-scores/submissions/)

QuestionEditorial Solution

Total Accepted: **8909** Total Submissions: **36304** Difficulty: **Medium**

Write a SQL query to rank scores. If there is a tie between two scores, both should have the same ranking. Note that after a tie, the next ranking number should be the next consecutive integer value. In other words, there should be no "holes" between ranks.

+----+-------+

| Id | Score |

+----+-------+

| 1 | 3.50 |

| 2 | 3.65 |

| 3 | 4.00 |

| 4 | 3.85 |

| 5 | 4.00 |

| 6 | 3.65 |

+----+-------+

For example, given the above Scores table, your query should generate the following report (order by highest score):

+-------+------+

| Score | Rank |

+-------+------+

| 4.00 | 1 |

| 4.00 | 1 |

| 3.85 | 2 |

| 3.65 | 3 |

| 3.65 | 3 |

| 3.50 | 4 |

+-------+------+

These are four different solutions.

**With Variables:** 841 ms

First one uses two variables, one for the current rank and one for the previous score.

**SELECT**

Score,

@rank := @rank + (@prev <> (@prev := Score)) Rank

**FROM**

Scores,

(**SELECT** @rank := 0, @prev := -1) init

**ORDER** **BY** Score **desc**

**Always Count:** 1322 ms

This one counts, for each score, the number of distinct greater or equal scores.

**SELECT**

Score,

(**SELECT** **count**(**distinct** Score) **FROM** Scores **WHERE** Score >= s.Score) Rank

**FROM** Scores s

**ORDER** **BY** Score **desc**

**Always Count, Pre-uniqued:** 795 ms

Same as the previous one, but faster because I have a subquery that "uniquifies" the scores first. Not entirely sure *why* it's faster, I'm guessing MySQL makes tmp a temporary table and uses it for every outer Score.

**SELECT**

Score,

(**SELECT** **count**(\*) **FROM** (**SELECT** **distinct** Score s **FROM** Scores) tmp **WHERE** s >= Score) Rank

**FROM** Scores

**ORDER** **BY** Score **desc**

**Filter/count Scores^2:** 1414 ms

Inspired by the attempt in wangkan2001's answer. Finally Id is good for something :-)

**SELECT** s.Score, **count**(**distinct** t.score) Rank

**FROM** Scores s **JOIN** Scores t **ON** s.Score <= t.score

**GROUP** **BY** s.Id

**ORDER** **BY** s.Score **desc**

**SELECT** Scores.Score, **COUNT**(Ranking.Score) **AS** RANK

**FROM** Scores

, (

**SELECT** **DISTINCT** Score

**FROM** Scores

) Ranking

**WHERE** Scores.Score <= Ranking.Score

**GROUP** **BY** Scores.Id, Scores.Score

**ORDER** **BY** Scores.Score **DESC**;

Solution with sub-queries

We just get the rows the distinct rows that are <= that each score, count them and wrap them in an external SELECT for formatting.

**SELECT** Scores.Score, Q3.Rank

**FROM**(

**SELECT** Q1.Score **as** Score, **COUNT**(Q1.Score) **as** Rank

**FROM**

(**SELECT** **DISTINCT** Score **from** Scores) **as** Q1,

(**SELECT** **DISTINCT** Score **from** Scores) **as** Q2

**WHERE** Q1.Score <= Q2.Score

**GROUP** **BY** Q1.Score

) **as** Q3, Scores

**WHERE** Q3.Score = Scores.Score

**ORDER** **BY** Scores.Score **DESC**

## 金典和牛客网编程题

### 机器人走方格II

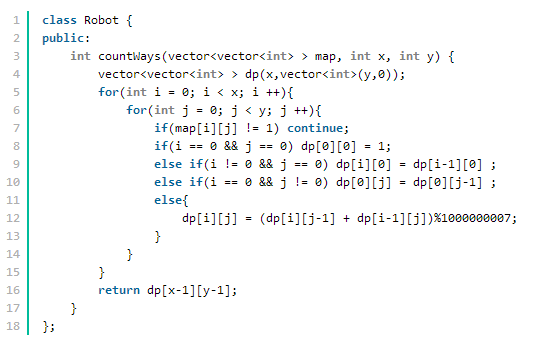
**题目描述**

有一个XxY的网格，一个机器人只能走格点且只能向右或向下走，要从左上角走到右下角。请设计一个算法，计算机器人有多少种走法。注意这次的网格中有些障碍点是不能走的。

给定一个int[][] **map**(C++ 中为vector >),表示网格图，若map[i][j]为1则说明该点不是障碍点，否则则为障碍。另外给定int **x**,int **y**，表示网格的大小。请返回机器人从(0,0)走到(x - 1,y - 1)的走法数，为了防止溢出，请将结果Mod 1000000007。保证x和y均小于等于50

下面的方法好

下面的else if判断可以避免越界的情况



public int countWays(int[][] map, int x, int y) {

    // write code here

    /\*

     \* 1.判断右下角的点以及起点自身是否为障碍点，若是返回0；

     \* 2.若右下角的点非障碍点，判断上面和左边是否为障碍点

     \*  1.若全都为障碍点，返回0

     \*  2.若一个为障碍点，一个不是，则到该点的路径数等于上一个点的路径数（这是递归的思路）

     \* 第2部分可以不用递归，而用循环：

     \*  dp[i-1][j-1]表示从(0,0)到(i,j)的方法数，如果(i,j)非1，则为障碍点，对应dp[i-1][j-1]为0

     \*  其余情况与一般dp相同

     \*/

    if(map == null || map.length != x || map[0].length != y){

        return 0;

    }

    if(map[x-1][y-1] != 1 || map[0][0] != 1){//最后一个点为障碍点

        return 0;

    }

    int dp[][] = new int[x][y];

    dp[0][0] = 1;

    for(int i = 1; i < x; i++){

        if(map[i][0] != 1){

            dp[i][0] = 0;

        }else{

            dp[i][0] = dp[i-1][0];

        }

    }

    for(int i = 1; i < y; i++){

        if(map[0][i] != 1){

            dp[0][i] = 0;

        }else{

            dp[0][i] = dp[0][i-1];

        }

    }

    for(int i = 1; i < x; i++){

        for(int j = 1; j < y; j++){

            if(map[i][j] != 1){

                dp[i][j] = 0;

            }else{

                dp[i][j] = dp[i-1][j]%1000000007 + dp[i][j-1]%1000000007;

            }

        }

    }

    return (dp[x-1][y-1]%1000000007);

}

我的做法，报运行超时:您的程序未能在规定时间内运行结束，请检查是否循环有错或算法复杂度过大。

public class Robot {

public int countWays(int[][] map, int x, int y) {

int[][] f=new int[50][50];

f[0][0]=1;

return count(x,y,x-1,y-1,f,map);

}

public int count(int x, int y, int i, int j, int[][] f,int[][] map)

{

if(i<0 || j<0 || i>=x || j>=y) return 0;

if(map[i][j]!=1) return 0;

if(f[i][j]>0) return f[i][j];

f[i][j]=count(x,y,i-1,j,f,map)+count(x,y,i,j-1,f,map);

return f[i][j];

}

}

[编程题]最大间隔

给定一个递增序列，a1 <a2 <...<an 。定义这个序列的最大间隔为d=max{ai+1 - ai }(1≤i<n),现在要从a2 ,a3..an-1 中删除一个元素。问剩余序列的最大间隔最小是多少？

5  
1 2 3 7 8

输出例子:

4

解题思路：

1.先计算原始数组相邻间隔，并在计算的过程中记录最大相邻间隔maxFull。

2.删除ai(1≤i<n)后所得新数组的最大相邻间隔只会在a[i+1]-a[i-1]与maxFull中取值，也就是Math.max(arr[i+1]-arr[i-1], maxFull)。

3.记录每一次删除ai(1≤i<n)后所得最大相邻间隔的最小值。

|  |  |
| --- | --- |
|  | import java.util.\*;  public class Main {      public static void main(String[] args) {          Scanner in = new Scanner(System.in);          while(in.hasNext()){              int n = in.nextInt(),i;              int arr[] = new int[n];              for (i = 0; i < n; i++) {                  arr[i] = in.nextInt();              }              int maxFull = Integer.MIN\_VALUE,minMaxGap = Integer.MAX\_VALUE;              for (i = 1; i < n; i++) {                  maxFull = Math.max(maxFull, arr[i]-arr[i-1]);              }              for (i = 1; i < n-1; i++) {                  minMaxGap = Math.min(minMaxGap, Math.max(arr[i+1]-arr[i-1], maxFull));              }              System.out.println(minMaxGap);          }          in.close();      }  } |

### Java Scanner读取数据

A simple text scanner which can parse primitive types and strings using regular expressions.

A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.

Scanner将输入按照分隔符分成tokens，默认分隔符是空格。

然后，被分隔开的tokens被转换成不同类型的值。

分隔符可以是任意的.useDelimiter

The scanner can also use delimiters other than whitespace. This example reads several items in from a string:

String input = "1 fish 2 fish red fish blue fish";

Scanner s = new Scanner(input).useDelimiter("\\s\*fish\\s\*");

System.out.println(s.nextInt());

System.out.println(s.nextInt());

System.out.println(s.next());

System.out.println(s.next());

s.close();

prints the following output:

1

2

red

blue

Depending upon the type of delimiting pattern, empty tokens may be returned. For example, the pattern"\\s+" will return no empty tokens since it matches multiple instances of the delimiter. The delimiting pattern "\\s" could return empty tokens since it only passes one space at a time.

File file = new File("F:/input.txt");

Scanner in = new Scanner(file);

Scanner in = new Scanner(System.in);

in.hasNext();

in.nextInt();

in.next ();

in.nextLine();

next()方法是以换行或者空格符为分界线接收下一个String类型变量。

nextInt()方法会读取下一个int型标志的token.但是焦点不会移动到下一行，仍然处在这一行上。当使用nextLine()方法时会读取改行剩余的所有的内容，包括换行符，然后把焦点移动到下一行的开头。所以这样就无法接收到下一行输入的String类型的变量。

**如果不知道什么时候结束输入，怎么处理？**

## Java语法技巧

【打印数组元素】

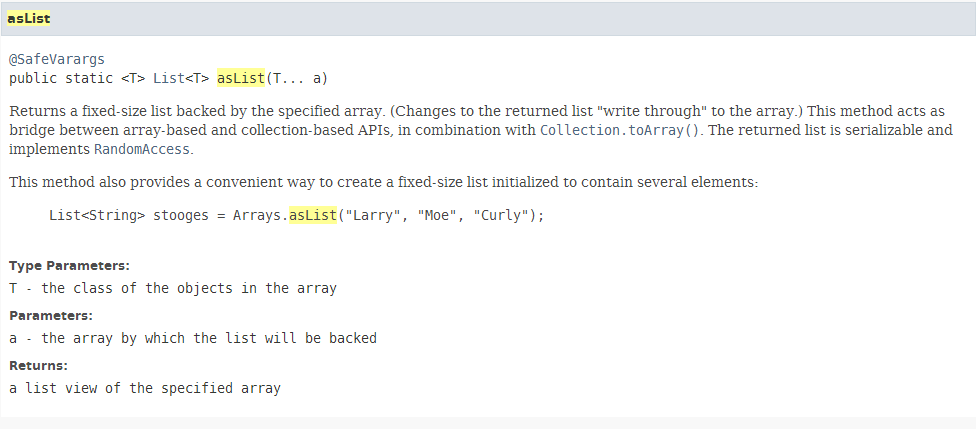
int[] array = {0,1,4,7,2,5,8,3,6,9};

System.out.println(Arrays.toString(array));

【Array和List的转换】

List<**List<Integer**>> res = new LinkedList<>();

res.add(Arrays.asList(num[i], num[lo], num[hi]));



一、集合转化为数组：

       采用集合.toArray()方法，根据转化为数组的类型又可以分为：

       1、转化为Object[]类型数组

             比如：Object[] listArray = list.toArray();

            Object[] setArray = set.toArray();

    2、转化为具体类型数组

  比如：String[] listArray1 = (String[]) list.toArray(**new** String[0])或者String[] listArray2 = (String[]) list.toArray(**new** String[list.size()]);

        String[] setArray1 = (String[]) set.toArray(**new** String[0]);或者String setArray2 = (String[]) set.toArray(**new** String[set.size()]);

二、数组转化为集合：

    采用数组.asList()方法

    List list = new ArrayList();

    list = Arrays.asList(array);   //**注意**：对于int[]数组不能直接这样做，因为asList()方法的参数必须是对象。应该先把int[]转化为Integer[]。

    Set set = new HashSet(Array.asList(array)); //使用list构造set

**【字符串和字符数组的转换】**

**char[] word=str.toCharArray();**

**new String(word)；**

**public List<String> getOneEdit(String str) //str每次只变化一位的字符串集合**

**{**

**List<String> res=new LinkedList<String>();**

**for(int i=0;i<str.length();i++)**

**{**

**char[] word=str.toCharArray();**

**for(char c='A'; c<='Z'; c++)//字符也是可以++**

**{**

**if(c!=str.charAt(i))**

**{**

**word[i]=c;**

**res.add(new String(word));**

**}**

**}**

**}**

**return res;**

**}**

**这样也不错**

for (int i = 0; i < word.length(); i++){

StringBuilder builder = new StringBuilder(word);

for (char ch='a'; ch <= 'z'; ch++){

builder.setCharAt(i,ch);

String new\_word=builder.toString();