Leetcode

### Two Sum

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**].

**UPDATE (2016/2/13):**  
The return format had been changed to **zero-based** indices. Please read the above updated description carefully.

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public int[] twoSum(int[] nums, int target) {

if (nums == null || nums.length < 2) {

return null;

}

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

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

int val = nums[i];

if (map.containsKey(target-val)) {

return new int[] {map.get(target-val), i};

}

map.put(val, i);

}

return null;

}

}

### Add Two Numbers

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

**Solution 1 - Time O (n+m) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

if (l1 == null && l2 == null) {

return null;

}

ListNode ret = new ListNode(0);

ListNode cur = ret;

int carry = 0;

while (l1 != null && l2 != null) {

int val = l1.val + l2.val + carry;

carry = val / 10;

cur.next = new ListNode(val % 10);

l1 = l1.next;

l2 = l2.next;

cur = cur.next;

}

while (l1 != null) {

int val = l1.val + carry;

carry = val / 10;

cur.next = new ListNode(val % 10);

l1 = l1.next;

cur = cur.next;

}

while (l2 != null) {

int val = l2.val + carry;

carry = val / 10;

cur.next = new ListNode(val % 10);

l2 = l2.next;

cur = cur.next;

}

if (carry != 0) {

cur.next = new ListNode(carry);

}

return ret.next;

}

}

### Longest Substring Without Repeating Characters

Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3. For "bbbbb" the longest substring is "b", with the length of 1.

**Solution 1 – Time O (n) Space O (1) single iteration**

public class Solution {

public int lengthOfLongestSubstring(String s) {

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

return 0;

}

int[] arr = new int[256];

Arrays.fill(arr, -1);

int maxLen = 0;

int l = 0;

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

int index = s.charAt(r);

if (arr[index] >= l) {

l = arr[index] + 1;

}

arr[index] = r;

maxLen = Math.max(r - l + 1, maxLen);

}

return maxLen;

}

}

**Solution 2 - Time O (n) Space O (n)**

public class Solution {

public int lengthOfLongestSubstring(String s) {

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

return 0;

}

Set<Character> set = new HashSet<Character>();

int maxLen = 0;

int l = 0;

int r = 0;

while (r < s.length()) {

if (set.contains(s.charAt(r))) {

while (s.charAt(l) != s.charAt(r)) {

set.remove(s.charAt(l));

l++;

}

set.remove(s.charAt(l));

l++;

}

set.add(s.charAt(r));

maxLen = Math.max(r - l + 1, maxLen);

r++;

}

return maxLen;

}

}

### Median of Two Sorted Arrays

There are two sorted arrays A and B of size m and n respectively. Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

**Solution 1 – Time O (log (m + n)) Space O (log (m + n))**

public class Solution {

public double findMedianSortedArrays(int A[], int B[]) {

int lenA = A.length;

int lenB = B.length;

int totalLen = lenA + lenB;

if (totalLen % 2 == 0) {

return (findKth(A, 0, lenA - 1, B, 0, lenB - 1, totalLen / 2) + findKth(

A, 0, lenA - 1, B, 0, lenB - 1, totalLen / 2 + 1)) / 2.0;

} else {

return findKth(A, 0, lenA - 1, B, 0, lenB - 1, totalLen / 2 + 1);

}

}

public double findKth(int A[], int a1, int a2, int B[], int b1, int b2,

int k) {

int a = a2 - a1 + 1;

int b = b2 - b1 + 1;

if (a > b) {

return findKth(B, b1, b2, A, a1, a2, k);

}

if (a == 0) {

return B[b1 + k - 1];

}

if (k == 1) {

return Math.min(A[a1], B[b1]);

}

int posA = Math.min(k / 2, a);

int posB = k - posA;

if (A[a1 + posA - 1] == B[b1 + posB - 1]) {

return A[a1 + posA - 1];

} else if (A[a1 + posA - 1] < B[b1 + posB - 1]) {

return findKth(A, a1 + posA, a2, B, b1, b1 + posB - 1, posB);

} else {

return findKth(A, a1, a1 + posA - 1, B, b1 + posB, b2, posA);

}

}

}

### Longest Palindromic Substring

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.

**Solution 1 – Time O (n^2) Space O (1)**

public class Solution {

public String longestPalindrome(String s) {

int start = 0, end = 0;

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

int len1 = expandAroundCenter(s, i, i);

int len2 = expandAroundCenter(s, i, i + 1);

int len = Math.max(len1, len2);

if (len > end - start) {

start = i - (len - 1) / 2;

end = i + len / 2;

}

}

return s.substring(start, end + 1);

}

private int expandAroundCenter(String s, int left, int right) {

int L = left, R = right;

while (L >= 0 && R < s.length() && s.charAt(L) == s.charAt(R)) {

L--;

R++;

}

return R - L - 1;

}

}

**Solution 2 – Time O (n^2) Space O (1)**

public class Solution {

public String longestPalindrome(String s) {

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

return "";

}

int len = s.length();

int maxLen = 0;

int maxLeft = 0;

int maxRight = 0;

int curLen = 0;

int prev;

int next;

for (int i = 0; i < len \* 2 - 1; i++) {

if (i % 2 == 0) {

curLen = 1;

prev = (i / 2 - 1);

next = (i / 2 + 1);

} else {

curLen = 0;

prev = (i / 2);

next = (i / 2 + 1);

}

for (; prev >= 0 && next < len; prev--, next++) {

if (s.charAt(prev) == s.charAt(next)) {

curLen += 2;

}

else {

break;

}

}

if (curLen > maxLen) {

maxLen = curLen;

maxLeft = prev + 1;

maxRight = next;

}

}

return s.substring(maxLeft, maxRight);

}

}

**Solution 3 – Time O (n^2) Space O (n^2)**

//DP

public class Solution {

public String longestPalindrome(String s) {

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

return "";

}

boolean[][] palin = new boolean[s.length()][s.length()];

String res = "";

int maxLen = 0;

for (int i = s.length() - 1; i >= 0; i--) {

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

if (s.charAt(i) == s.charAt(j)

&& (j - i <= 2 || palin[i + 1][j - 1])) {

palin[i][j] = true;

if (maxLen < j - i + 1) {

maxLen = j - i + 1;

res = s.substring(i, j + 1);

}

}

}

}

return res;

}

}

### ZigZag Conversion

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

P A H N

A P L S I I G

Y I R

And then read line by line: "PAHNAPLSIIGYIR"

Write the code that will take a string and make this conversion given a number of rows:

string convert(string text, int nRows);

convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".

**Solution 1 – Time O (n) Space O (1)**

public String convert(String s, int nRows) {

if (s == null || s.length() == 0 || nRows < 1) {

return "";

}

if (nRows == 1) {

return s;

}

StringBuilder ret = new StringBuilder();

int len = s.length();

int size = 2 \* nRows - 2;

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

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

ret.append(s.charAt(j));

if (i != 0 && i != (nRows - 1) && (j + size - 2 \* i) < len) {

ret.append(s.charAt(j + size - 2 \* i));

}

}

}

return ret.toString();

}

### Reverse Integer

Reverse digits of an integer.

**Example1:** x = 123, return 321  
**Example2:** x = -123, return -321

[click to show spoilers.](https://oj.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.

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public int reverse(int x) {

if (x == Integer.MIN\_VALUE) {

return 0;

}

int num = Math.abs(x);

int ret = 0;

while (num != 0) {

if (ret > (Integer.MAX\_VALUE - num % 10) / 10) {

return 0;

}

ret = ret \* 10 + num % 10;

num = num / 10;

}

return x > 0 ? ret : -ret;

}

}

### String to Integer (atoi)

Total Accepted: **36372** Total Submissions: **267566**

Implement atoi to convert a string to an integer.

**Hint:** Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.

**Notes:** It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.

**Update (2015-02-10):**  
The signature of the C++ function had been updated. If you still see your function signature accepts a const char \* argument, please click the reload button to reset your code definition.

[spoilers alert... click to show requirements for atoi.](https://oj.leetcode.com/problems/string-to-integer-atoi/)

**Requirements for atoi:**

The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, INT\_MAX (2147483647) or INT\_MIN (-2147483648) is returned.

/\*\*

\* 这种题的考察重点并不在于问题本身，而是要注意corner

\* case的处理，整数一般有两点，一个是正负符号问题，另一个是整数越界问题。思路比较简单，就是先去掉多余的空格字符

\* ，然后读符号（注意正负号都有可能，也有可能没有符号

\* ），接下来按顺序读数字，结束条件有三种情况：（1）异常字符出现（按照C语言的标准是把异常字符起的后面全部截去

\* ，保留前面的部分作为结果）；（2）数字越界（返回最接近的整数）；（3）字符串结束。

\*/

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int myAtoi(String str) {

int i = 0, n = str.length();

while (i < n && Character.isWhitespace(str.charAt(i)))

i++;

int sign = 1;

if (i < n && str.charAt(i) == '+') {

i++;

} else if (i < n && str.charAt(i) == '-') {

sign = -1;

i++;

}

int num = 0;

while (i < n && Character.isDigit(str.charAt(i))) {

int digit = Character.getNumericValue(str.charAt(i));

if (sign == -1 && (Integer.MIN\_VALUE + digit) / 10 > -num) {

return Integer.MIN\_VALUE;

}

if (sign == 1 && (Integer.MAX\_VALUE - digit) / 10 < num) {

return Integer.MAX\_VALUE;

}

num = num \* 10 + digit;

i++;

}

return sign \* num;

}

}

**Solution 2 – Time O (n) Space O (1)**

public class Solution {

public int atoi(String str) {

if (str == null) {

return 0;

}

str = str.trim();

if (str.length() == 0) {

return 0;

}

int i = 0;

boolean isNeg = false;

if (str.charAt(0) == '+' || str.charAt(0) == '-') {

i++;

if (str.charAt(0) == '-') {

isNeg = true;

}

}

int len = str.length();

int ret = 0;

while (i < len) {

if (str.charAt(i) > '9' || str.charAt(i) < '0') {

break;

}

int num = str.charAt(i) - '0';

if (isNeg && (Integer.MIN\_VALUE + num) / 10 > -ret) {

return Integer.MIN\_VALUE;

}

if (!isNeg && (Integer.MAX\_VALUE - num) / 10 < ret) {

ret = Integer.MAX\_VALUE;

break;

}

ret = ret \* 10 + num;

i++;

}

return isNeg ? -ret : ret;

}

}

### Palindrome Number

Determine whether an integer is a palindrome. Do this without extra space.

[click to show spoilers.](https://oj.leetcode.com/problems/palindrome-number/)

**Some hints:**

Could negative integers be palindromes? (ie, -1)

If you are thinking of converting the integer to string, note the restriction of using extra space.

You could also try reversing an integer. However, if you have solved the problem "Reverse Integer", you know that the reversed integer might overflow. How would you handle such case?

There is a more generic way of solving this problem.

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public boolean isPalindrome(int x) {

if (x < 0) {

return false;

}

int d = 1;

while (x / d >= 10) {

d \*= 10;

}

while (x > 0) {

int right = x % 10;

int left = x / d;

if (left != right) {

return false;

}

x = x % d / 10;

d /= 100;

}

return true;

}

}

### Regular Expression Matching

Implement regular expression matching with support for '.' and '\*'.

'.' Matches any single character.

'\*' Matches zero or more of the preceding element.

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", "a\*") → true

isMatch("aa", ".\*") → true

isMatch("ab", ".\*") → true

isMatch("aab", "c\*a\*b") → true

**Solution 1 – Time O (n^n) Space (n^n)**

public class Solution {

public boolean isMatch(String s, String p) {

return helper(s, p, 0, 0);

}

private boolean helper(String s, String p, int i, int j) {

if (j == p.length())

return i == s.length();

if (j == p.length() - 1 || p.charAt(j + 1) != '\*') {

if (i == s.length() || s.charAt(i) != p.charAt(j)

&& p.charAt(j) != '.')

return false;

else

return helper(s, p, i + 1, j + 1);

}

// p.charAt(j+1)=='\*'

while (i < s.length()

&& (p.charAt(j) == '.' || s.charAt(i) == p.charAt(j))) {

if (helper(s, p, i, j + 2))

return true;

i++;

}

return helper(s, p, i, j + 2);

}

}

**Solution 2 – Time O (n^2) Space (n^2)**

public class Solution {

public boolean isMatch(String s, String p) {

if (s.length() == 0 && p.length() == 0)

return true;

if (p.length() == 0)

return false;

boolean[][] res = new boolean[s.length() + 1][p.length() + 1];

res[0][0] = true;

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

if (p.charAt(j) == '\*') {

if (j > 0 && res[0][j - 1])

res[0][j + 1] = true;

if (j < 1)

continue;

if (p.charAt(j - 1) != '.') {

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

if (res[i + 1][j] || j > 0 && res[i + 1][j - 1]

|| i > 0 && j > 0 && res[i][j + 1]

&& s.charAt(i) == s.charAt(i - 1)

&& s.charAt(i - 1) == p.charAt(j - 1))

res[i + 1][j + 1] = true;

}

} else {

int i = 0;

while (j > 0 && i < s.length() && !res[i + 1][j - 1]

&& !res[i + 1][j])

i++;

for (; i < s.length(); i++) {

res[i + 1][j + 1] = true;

}

}

} else {

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

if (s.charAt(i) == p.charAt(j) || p.charAt(j) == '.')

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

}

}

}

return res[s.length()][p.length()];

}

}

### Container With Most Water

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.

**Solution 1 – Time O (n) Space O (1)**

package problems;

public class ContainerWithMostWater {

public int maxArea(int[] height) {

if (height == null || height.length < 2) {

return 0;

}

int l = 0;

int r = height.length - 1;

int ret = 0;

while (l < r) {

int area = Math.min(height[l], height[r]) \* (r - l);

ret = Math.max(ret, area);

if (height[l] < height[r]) {

l++;

} else {

r--;

}

}

return ret;

}

}

### Integer to Roman

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public String intToRoman(int num) {

int[] val = new int[] { 1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5,

4, 1 };

String[] str = new String[] { "M", "CM", "D", "CD", "C", "XC", "L",

"XL", "X", "IX", "V", "IV", "I" };

int i = 0;

StringBuilder ret = new StringBuilder();

while (num > 0) {

if (num >= val[i]) {

ret.append(str[i]);

num = num - val[i];

} else {

i++;

}

}

return ret.toString();

}

}

### Roman to Integer

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int romanToInt(String s) {

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

return 0;

}

int ret = map(s.charAt(0));

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

if (map(s.charAt(i)) > map(s.charAt(i - 1))) {

ret -= map(s.charAt(i - 1)) \* 2;

}

ret += map(s.charAt(i));

}

return ret;

}

public int map(char c) {

switch (c) {

case 'I':

return 1;

case 'V':

return 5;

case 'X':

return 10;

case 'L':

return 50;

case 'C':

return 100;

case 'D':

return 500;

case 'M':

return 1000;

}

return 0;

}

}

### Longest Common Prefix

Write a function to find the longest common prefix string amongst an array of strings.

**Solution 1 – Time O (m\*n) Space O (1)**

public class Solution {

public String longestCommonPrefix(String[] strs) {

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

return "";

}

for (int index = 0; index < strs[0].length(); index++) {

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

if (strs[i].length() <= index

|| strs[i].charAt(index) != strs[0].charAt(index)) {

return strs[0].substring(0, index);

}

}

}

return strs[0];

}

}

### 3Sum

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)

**Solution 1 – Time O (n^2) Space (n)**

public class Solution {

public List<List<Integer>> threeSum(int[] num) {

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

if (num == null || num.length <= 2) {

return ret; // ret is []

}

Arrays.sort(num);

for (int i = num.length - 1; i >= 2; i--) {

if ((i < num.length - 1) && (num[i] == num[i + 1])) {

continue;

}

List<List<Integer>> curRet = twoSum(num, i - 1, -num[i]);

ret.addAll(curRet);

}

return ret;

}

public List<List<Integer>> twoSum(int[] num, int right, int target) {

int left = 0;

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

while (left < right) {

if (num[left] + num[right] == target) {

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

list.add(num[left]);

list.add(num[right]);

list.add(-target);

curRet.add(list);

left++;

right--;

while (left < right && num[left] == num[left - 1]) {

left++;

}

while (left < right && num[right] == num[right + 1]) {

right--;

}

} else if (num[left] + num[right] < target) {

left++;

} else {

right--;

}

}

return curRet;

}

}

### 3Sum Closest

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).

**Solution 1 – Time O(n^2) Space(1)**

public class Solution {

public int threeSumClosest(int[] num, int target) {

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

return 0;

}

Arrays.sort(num);

int closet = num[0] + num[1] + num[2] - target;

for (int i = 0; i < num.length - 2; i++) {

int cur = twoSum(num, i + 1, target - num[i]);

if (Math.abs(cur) < Math.abs(closet)) {

closet = cur;

}

}

return closet + target;

}

public int twoSum(int[] num, int left, int target) {

int right = num.length - 1;

int closet = num[left] + num[right] - target;

while (left < right) {

if (num[left] + num[right] == target) {

return 0;

}

int cur = num[left] + num[right] - target;

if (Math.abs(cur) < Math.abs(closet)) {

closet = cur;

}

if (num[left] + num[right] < target) {

left++;

} else {

right--;

}

}

return closet;

}

}

### Letter Combinations of a Phone Number

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.

**Solution 1 – Time O (4^n) Space O (1)**

public class Solution {

public List<String> letterCombinations(String digits) {

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

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

return ret;

}

ret.add("");

int digitsLen = digits.length();

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

String letter = getLetters(digits.charAt(i));

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

int letterLen = letter.length();

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

for (int k = 0; k < letterLen; k++) {

cur.add(ret.get(j) + letter.charAt(k));

}

}

ret = cur;

}

return ret;

}

public String getLetters(char digit) {

switch (digit) {

case '2':

return "abc";

case '3':

return "def";

case '4':

return "ghi";

case '5':

return "jkl";

case '6':

return "mno";

case '7':

return "pqrs";

case '8':

return "tuv";

case '9':

return "wxyz";

case '0':

return "";

case '1':

return "";

default:

return "";

}

}

}

### 4Sum

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)

**Solution 1 – Time O (n^3) Space O (n)**

public class Solution {

public List<List<Integer>> fourSum(int[] num, int target) {

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

if (num == null || num.length <= 3) {

return ret; // ret is []

}

Arrays.sort(num);

for (int i = num.length - 1; i >= 3; i--) {

if (i < num.length - 1 && num[i] == num[i + 1]) {

continue;

}

List<List<Integer>> curRet = threeSum(num, i - 1, target - num[i]);

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

curRet.get(j).add(num[i]);

}

ret.addAll(curRet);

}

return ret;

}

public List<List<Integer>> threeSum(int[] num, int right, int target) {

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

for (int i = right; i >= 2; i--) {

if (i < right && num[i] == num[i + 1]) {

continue;

}

List<List<Integer>> curRet = twoSum(num, i - 1, target - num[i]);

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

curRet.get(j).add(num[i]);

}

ret.addAll(curRet);

}

return ret;

}

public List<List<Integer>> twoSum(int[] num, int right, int target) {

int left = 0;

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

while (left < right) {

if (num[left] + num[right] == target) {

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

list.add(num[left]);

list.add(num[right]);

curRet.add(list);

left++;

right--;

while (left < right && num[left] == num[left - 1]) {

left++;

}

while (left < right && num[right] == num[right + 1]) {

right--;

}

} else if (num[left] + num[right] < target) {

left++;

} else {

right--;

}

}

return curRet;

}

}

### Remove Nth Node From End of List

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.

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode removeNthFromEnd(ListNode head, int n) {

ListNode ret = new ListNode(0);

ret.next = head;

ListNode cur = head;

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

if (cur != null) {

cur = cur.next;

}

}

ListNode prev = ret;

while (cur != null) {

prev = prev.next;

cur = cur.next;

}

prev.next = prev.next.next;

return ret.next;

}

}

### Valid Parentheses

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.

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public boolean isValid(String s) {

if (s == null) {

return true;

}

int len = s.length();

Stack<Character> stack = new Stack<Character>();

String left = "([{";

String right = ")]}";

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

if (left.indexOf(s.charAt(i)) != -1) {

stack.push(s.charAt(i));

} else if (stack.empty()

|| stack.pop() != left.charAt(right.indexOf(s.charAt(i)))) {

return false;

}

}

return stack.empty();

}

}

### Merge Two Sorted Lists

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.

**Solution 1 – Time O (n+m) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode mergeTwoLists(ListNode l1, ListNode l2) {

ListNode head = new ListNode(0);

ListNode cur = head;

while (l1 != null && l2 != null) {

if (l1.val < l2.val) {

cur.next = l1;

l1 = l1.next;

} else {

cur.next = l2;

l2 = l2.next;

}

cur = cur.next;

}

if (l1 != null) {

cur.next = l1;

} else {

cur.next = l2;

}

return head.next;

}

}

### Generate Parentheses

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:

"((()))", "(()())", "(())()", "()(())", "()()()"

**Solution 1 – Time O (Catalan numbers) Space O (Catalan numbers)**

public class Solution {

public List<String> generateParenthesis(int n) {

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

if (n <= 0) {

return ret;

}

generateParenthesis(n, n, "", ret);

return ret;

}

public void generateParenthesis(int l, int r, String item, List<String> ret) {

if (l == 0 && r == 0) {

ret.add(item);

return;

}

if (l > 0) {

generateParenthesis(l - 1, r, item + "(", ret);

}

if (l < r) {

generateParenthesis(l, r - 1, item + ")", ret);

}

}

}

### Merge k Sorted Lists

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

**Solution 1 – Time O (nklogk) Space O (logk)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode mergeKLists(List<ListNode> lists) {

if (lists == null || lists.size() == 0) {

return null;

}

return helper(lists, 0, lists.size() - 1);

}

public ListNode helper(List<ListNode> lists, int l, int r) {

if (l < r) {

int m = (l + r) / 2;

return merge(helper(lists, l, m), helper(lists, m + 1, r));

}

return lists.get(l);

}

public ListNode merge(ListNode l1, ListNode l2) {

ListNode head = new ListNode(0);

ListNode cur = head;

while (l1 != null && l2 != null) {

if (l1.val < l2.val) {

cur.next = l1;

l1 = l1.next;

} else {

cur.next = l2;

l2 = l2.next;

}

cur = cur.next;

}

if (l1 != null) {

cur.next = l1;

}

if (l2 != null) {

cur.next = l2;

}

return head.next;

}

}

**Solution 2 – Time O (nklogk) Space O (k)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode mergeKLists(List<ListNode> lists) {

if (lists == null || lists.size() == 0) {

return null;

}

Comparator<ListNode> comparator = new QueueComparator();

PriorityQueue<ListNode> queue = new PriorityQueue<ListNode>(10,

comparator);

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

ListNode node = lists.get(i);

if (node != null) {

queue.add(node);

}

}

ListNode head = new ListNode(0);

ListNode cur = head;

while (queue.size() != 0) {

ListNode top = queue.poll();

cur.next = top;

cur = cur.next;

if (top.next != null) {

queue.add(top.next);

}

}

return head.next;

}

class QueueComparator implements Comparator<ListNode> {

@Override

public int compare(ListNode a, ListNode b) {

return a.val - b.val;

}

}

}

### Swap Nodes in Pairs

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.

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode swapPairs(ListNode head) {

ListNode prev = new ListNode(0);

prev.next = head;

ListNode ret = prev;

while (prev.next != null && prev.next.next != null) {

ListNode first = prev.next;

ListNode second = first.next;

prev.next = second;

first.next = second.next;

second.next = first;

prev = first;

}

return ret.next;

}

}

### Reverse Nodes in k-Group

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

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode reverseKGroup(ListNode head, int k) {

if (head == null || head.next == null || k <= 1) {

return head;

}

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode prev = dummy;

int count = 0;

ListNode cur = head;

while (cur != null) {

count++;

cur = cur.next;

if (count == k) {

prev = reverseNode(prev, cur);

count = 0;

}

}

return dummy.next;

}

/\*

\* the real list need to reverse is from (prev, end), both prev and end are

\* exclusive

\*/

public ListNode reverseNode(ListNode prev, ListNode end) {

ListNode head = prev.next;

ListNode cur = head.next;

while (cur != end) {

ListNode next = cur.next;

cur.next = prev.next;

prev.next = cur;

cur = next;

}

head.next = end;

return head;

}

}

### Remove Duplicates from Sorted Array

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 A = [1,1,2],

Your function should return length = 2, and A is now [1,2].

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int removeDuplicates(int[] A) {

if (A == null || A.length < 1) {

return 0;

}

int count = 0;

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

if (A[i] != A[count]) {

count++;

A[count] = A[i];

}

}

return count + 1;

}

}

### Remove Element

Given an array and a value, remove all instances of that value in place and return the new length.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int removeElement(int[] A, int elem) {

if (A == null) {

return 0;

}

int count = 0;

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

if (A[i] != elem) {

A[count++] = A[i];

}

}

return count;

}

}

### Implement strStr()

Implement strStr().

Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

**Update (2014-11-02):**  
The signature of the function had been updated to return the *index* instead of the pointer. If you still see your function signature returns a char \* or String, please click the reload button to reset your code definition.

**Solution 1 – Time O (n\*m) Space O (1)**

public class Solution {

public int strStr(String haystack, String needle) {

if (needle == null || needle == null) {

return 0;

}

if (needle.length() == 0) {

return 0;

}

for (int i = 0; i <= haystack.length() - needle.length(); i++) {

boolean flag = true;

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

if (haystack.charAt(i + j) != needle.charAt(j)) {

flag = false;

break;

}

}

if (flag == true) {

return i;

}

}

return -1;

}

}

**Solution 2 – Time O (n) Space O (1)**

public class Solution {

public int strStr(String haystack, String needle) {

if (needle == null || needle == null) {

return 0;

}

if (needle.length() == 0) {

return 0;

}

if (haystack.length() < needle.length()) {

return -1;

}

int base = 31;

long highBase = 1;

long needleHash = 0;

long hayHash = 0;

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

needleHash = base \* needleHash + needle.charAt(i);

hayHash = base \* hayHash + haystack.charAt(i);

highBase \*= base;

}

highBase = highBase / base;

if (needleHash == hayHash) {

return 0;

}

for (int i = needle.length(); i < haystack.length(); i++) {

hayHash = (hayHash - highBase

\* haystack.charAt(i - needle.length()))

\* base + haystack.charAt(i);

if (needleHash == hayHash) {

return i - needle.length() + 1;

}

}

return -1;

}

}

**Solution 3 – Time O (nm) Space O (1)**

public class Solution {

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;

}

}

}

}

}

### Divide Two Integers

Divide two integers without using multiplication, division and mod operator.

If it is overflow, return MAX\_INT.

**Solution 1 – Time O (logn) Space O (1)**

public class Solution {

public int divide(int dividend, int divisor) {

if (divisor == 0) {

return Integer.MAX\_VALUE;

}

if (dividend == Integer.MIN\_VALUE && divisor == -1) {

return Integer.MAX\_VALUE;

}

if (dividend == Integer.MIN\_VALUE && divisor == Integer.MIN\_VALUE) {

return 1;

}

if (divisor == Integer.MIN\_VALUE) {

return 0;

}

boolean sign = ((dividend ^ divisor) >>> 31 != 1) ? true : false;

int ret = 0;

if (dividend == Integer.MIN\_VALUE) {

dividend = dividend + Math.abs(divisor);

ret++;

}

dividend = Math.abs(dividend);

divisor = Math.abs(divisor);

int factor = divisor;

int count = 0;

while (dividend >> 1 >= factor) {

factor = factor << 1;

count++;

}

while (dividend >= divisor) {

if (dividend >= factor) {

ret += 1 << count;

dividend -= factor;

}

factor = factor >> 1;

count--;

}

return sign ? ret : -ret;

}

}

### Substring with Concatenation of All Words

You are given a string, **S**, and a list of words, **L**, that are all of the same length. Find all starting indices of substring(s) in S that is a concatenation of each word in L exactly once and without any intervening characters.

For example, given:  
**S**: "barfoothefoobarman"  
**L**: ["foo", "bar"]

You should return the indices: [0,9].  
(order does not matter).

**Solution 1 – Time O (n) Space O (m\*l)**

public class Solution {

public List<Integer> findSubstring(String S, String[] L) {

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

if (S == null || S.length() == 0 || L == null || L.length == 0) {

return ret;

}

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

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

if (map.containsKey(L[i])) {

map.put(L[i], map.get(L[i]) + 1);

} else {

map.put(L[i], 1);

}

}

int len = L[0].length();

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

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

int count = 0;

int left = i;

for (int j = i; j <= S.length() - len; j += len) {

String str = S.substring(j, j + len);

if (map.containsKey(str)) {

if (curMap.containsKey(str)) {

curMap.put(str, curMap.get(str) + 1);

} else {

curMap.put(str, 1);

}

if (curMap.get(str) <= map.get(str)) {

count++;

} else {

while (curMap.get(str) > map.get(str)) {

String temp = S.substring(left, left + len);

if (curMap.containsKey(temp)) {

curMap.put(temp, curMap.get(temp) - 1);

if (curMap.get(temp) < map.get(temp)) {

count--;

}

left += len;

}

}

}

if (count == L.length) {

ret.add(left);

String temp = S.substring(left, left + len);

if (curMap.containsKey(temp)) {

curMap.put(temp, curMap.get(temp) - 1);

}

count--;

left += len;

}

} else {

curMap.clear();

count = 0;

left = j + len;

}

}

}

return ret;

}

}

### Next Permutation

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

**Solution 1 Time O (n) Space O (1)**

public class Solution {

public void nextPermutation(int[] num) {

if (num == null || num.length <= 1) {

return;

}

int i;

for (i = num.length - 2; i >= 0; i--) {

if (num[i] < num[i + 1]) {

break;

}

}

if (i < 0) {

reverse(num, 0);

return;

}

for (int j = num.length - 1; j >= i; j--) {

if (num[j] > num[i]) {

int temp = num[i];

num[i] = num[j];

num[j] = temp;

break;

}

}

reverse(num, i + 1);

}

public void reverse(int[] num, int index) {

int right = num.length - 1;

while (index < right) {

int temp = num[index];

num[index++] = num[right];

num[right--] = temp;

}

}

}

### Longest Valid Parentheses

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2.

Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public int longestValidParentheses(String s) {

if (s == null || s.length() < 2) {

return 0;

}

int ret = 0;

int start = 0;

Stack<Integer> stack = new Stack<Integer>();

int len = s.length();

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

if (s.charAt(i) == '(') {

stack.push(i);

} else {

if (stack.empty()) {

start = i + 1;

} else {

stack.pop();

if (stack.empty()) {

ret = max(ret, i - start + 1);

} else {

ret = max(ret, i - stack.peek());

}

}

}

}

return ret;

}

public int max(int a, int b) {

return a > b ? a : b;

}

}

### Search in Rotated Sorted Array

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public int search(int[] A, int target) {

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

return -1;

}

int l = 0, r = A.length - 1;

while (l <= r) {

int mid = (l + r) / 2;

if (target == A[mid]) {

return mid;

}

if (A[l] <= A[mid]) {

if (target < A[mid] && target >= A[l]) {

r = mid - 1;

} else {

l = mid + 1;

}

} else {

if (target <= A[r] && target > A[mid]) {

l = mid + 1;

} else {

r = mid - 1;

}

}

}

return -1;

}

}

### Search for a Range

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].

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public int[] searchRange(int[] A, int target) {

int[] ret = new int[] { -1, -1 };

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

return ret;

}

int l = 0;

int r = A.length - 1;

int m = (l + r) / 2;

while (l <= r) {

m = (l + r) / 2;

if (A[m] == target) {

ret[0] = m;

ret[1] = m;

break;

} else if (A[m] < target) {

l = m + 1;

} else {

r = m - 1;

}

}

if (A[m] != target) {

return ret;

}

l = m;

r = A.length - 1;

while (l <= r) {

m = (l + r) / 2;

if (A[m] == target) {

l = m + 1;

} else {

r = m - 1;

}

}

ret[1] = r;

l = 0;

r = ret[0];

while (l <= r) {

m = (l + r) / 2;

if (A[m] == target) {

r = m - 1;

} else {

l = m + 1;

}

}

ret[0] = l;

return ret;

}

}

**Solution 2 – Time O (log n) Space O (1)**

public class Solution {

public int[] searchRange(int[] A, int target) {

int[] res = { -1, -1 };

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

return res;

}

int ll = 0;

int lr = A.length - 1;

while (ll <= lr) {

int m = (ll + lr) / 2;

if (A[m] < target) {

ll = m + 1;

} else {

lr = m - 1;

}

}

int rl = 0;

int rr = A.length - 1;

while (rl <= rr) {

int m = (rl + rr) / 2;

if (A[m] <= target) {

rl = m + 1;

} else {

rr = m - 1;

}

}

if (ll <= rr) {

res[0] = ll;

res[1] = rr;

}

return res;

}

}

### Search Insert Position

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

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public int searchInsert(int[] A, int target) {

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

return 0;

}

int l = 0;

int r = A.length - 1;

int m;

while (l <= r) {

m = (l + r) / 2;

if (A[m] == target) {

return m;

} else if (A[m] < target) {

l = m + 1;

} else {

r = m - 1;

}

}

return l;

}

}

### Valid Sudoku

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.

**Solution 1 – Time O (3\*9\*9) Space (9)**

public class Solution {

public boolean isValidSudoku(char[][] board) {

if (board == null || board.length != 9 || board[0] == null

|| board[0].length != 9) {

return false;

}

int n = 9;

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

boolean[] map = new boolean[n];

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

if (board[i][j] != '.') {

if (map[board[i][j] - '1']) {

return false;

}

map[board[i][j] - '1'] = true;

}

}

}

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

boolean[] map = new boolean[n];

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

if (board[i][j] != '.') {

if (map[board[i][j] - '1']) {

return false;

}

map[board[i][j] - '1'] = true;

}

}

}

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

boolean[] map = new boolean[n];

for (int i = block / 3 \* 3; i < block / 3 \* 3 + 3; i++) {

for (int j = block % 3 \* 3; j < block % 3 \* 3 + 3; j++) {

if (board[i][j] != '.') {

if (map[board[i][j] - '1']) {

return false;

}

map[board[i][j] - '1'] = true;

}

}

}

}

return true;

}

}

### Sudoku Solver

Total Accepted: **23747** Total Submissions: **110660**

Write a program to solve a Sudoku puzzle by filling the empty cells.

Empty cells are indicated by the character '.'.

You may assume that there will be only one unique solution.



A sudoku puzzle...



**Solution 1 – Time O (9^4) Space O (1)**

public class Solution {

public void solveSudoku(char[][] board) {

if (board == null || board.length != 9 || board[0].length != 9) {

return;

}

helper(board, 0, 0);

}

public boolean helper(char[][] board, int i, int j) {

if (j >= 9) {

return helper(board, i + 1, 0);

}

if (i >= 9) {

return true;

}

if (board[i][j] == '.') {

for (int k = 1; k <= 9; k++) {

board[i][j] = (char) (k + '0');

if (isValid(board, i, j)) {

if (helper(board, i, j + 1)) {

return true;

}

}

board[i][j] = '.';

}

} else {

return helper(board, i, j + 1);

}

return false;

}

public boolean isValid(char[][] board, int i, int j) {

for (int k = 0; k < 9; k++) {

if (k != j && board[i][k] == board[i][j]) {

return false;

}

}

for (int k = 0; k < 9; k++) {

if (k != i && board[k][j] == board[i][j]) {

return false;

}

}

for (int m = i / 3 \* 3; m < i / 3 \* 3 + 3; m++) {

for (int n = j / 3 \* 3; n < j / 3 \* 3 + 3; n++) {

if (m != i && n != j && board[m][n] == board[i][j]) {

return false;

}

}

}

return true;

}

}

### Count and Say

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.

**Solution 1 – Time O (n \* StringLength) Space O (StringLength)**

public class Solution {

public String countAndSay(int n) {

if (n < 1) {

return "";

}

String ret = "1";

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

int count = 1;

String cur = new String();

for (int j = 1; j < ret.length(); j++) {

if (ret.charAt(j) == ret.charAt(j - 1)) {

count++;

} else {

cur += count;

cur += ret.charAt(j - 1);

count = 1;

}

}

cur += count;

cur += ret.charAt(ret.length() - 1);

ret = cur;

}

return ret;

}

}

### Combination Sum

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]

**Solution 1 – Time O (NP) Space O (NP)**

public class Solution {

public List<List<Integer>> combinationSum(int[] candidates, int target) {

List<List<Integer>> ret = new LinkedList<List<Integer>>();

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

return ret;

}

Arrays.sort(candidates);

helper(ret, new LinkedList<Integer>(), candidates, 0, target);

return ret;

}

public void helper(List<List<Integer>> ret, List<Integer> tmp,

int[] candidates, int index, int target) {

if (target == 0) {

ret.add(new LinkedList<Integer>(tmp));

return;

}

for (int i = index; i < candidates.length; i++) {

if (candidates[i] <= target) {

tmp.add(candidates[i]);

helper(ret, tmp, candidates, i, target - candidates[i]);

tmp.remove(tmp.size() - 1);

}

}

}

}

### Combination Sum II

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]

**Solution 1 – Time O (NP) Space O (NP)**

public class Solution {

public List<List<Integer>> combinationSum2(int[] candidates, int target) {

List<List<Integer>> ret = new LinkedList<List<Integer>>();

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

return ret;

}

Arrays.sort(candidates);

helper(ret, new LinkedList<Integer>(), candidates, 0, target);

return ret;

}

public void helper(List<List<Integer>> ret, List<Integer> tmp,

int[] candidates, int index, int target) {

if (target == 0) {

ret.add(new LinkedList<Integer>(tmp));

return;

}

for (int i = index; i < candidates.length; i++) {

if (i > index && candidates[i] == candidates[i - 1]) {

continue;

}

if (candidates[i] <= target) {

tmp.add(candidates[i]);

helper(ret, tmp, candidates, i + 1, target - candidates[i]);

tmp.remove(tmp.size() - 1);

}

}

}

}

### First Missing Positive

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.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int firstMissingPositive(int[] A) {

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

return 1;

}

int tmp;

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

if ((A[i] <= 0) || (A[i] > A.length) || (A[i] == A[A[i] - 1])) {

continue;

}

tmp = A[A[i] - 1];

A[A[i] - 1] = A[i];

A[i] = tmp;

i--;

}

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

if (A[i] != i + 1) {

return i + 1;

}

}

return A.length + 1;

}

}

### Trapping Rain Water

Given *n* non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example,   
Given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.



**Solution 1 – Time O (n) Space (n)**

public class Solution {

public int trap(int[] A) {

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

return 0;

}

int[] left = new int[A.length];

int[] right = new int[A.length];

left[0] = 0;

right[A.length - 1] = 0;

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

left[i] = Math.max(left[i - 1], A[i - 1]);

right[A.length - 1 - i] = Math.max(right[A.length - i], A[A.length

- i]);

}

int ret = 0;

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

int height = Math.min(left[i], right[i]);

if (height > A[i]) {

ret += height - A[i];

}

}

return ret;

}

}

**Solution 2 – Time O (n) Space O (n)**

public class Solution {

public int trap(int[] A) {

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

return 0;

}

int l = 0;

int r = A.length - 1;

int ret = 0;

while (l < r) {

int min = Math.min(A[l], A[r]);

if (A[l] == min) {

l++;

while (A[l] < min) {

ret += min - A[l];

l++;

}

} else {

r--;

while (A[r] < min) {

ret += min - A[r];

r--;

}

}

}

return ret;

}

}

### Multiply Strings

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.

**Solution 1 – Time O ((m + n) ^2) Space O (1)**

public class Solution {

public String multiply(String num1, String num2) {

if (num1 == null || num1.length() == 0 || num2 == null

|| num2.length() == 0) {

return "";

}

if (num1.charAt(0) == '0' || num2.charAt(0) == '0') {

return "0";

}

int l1 = num1.length();

int l2 = num2.length();

StringBuilder ret = new StringBuilder();

int carry = 0;

for (int i = 0; i < l1 + l2 - 1; i++) {

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

int k = i - j;

if (j < l1 && k < l2) {

int v1 = (int) (num1.charAt(l1 - 1 - j) - '0');

int v2 = (int) (num2.charAt(l2 - 1 - k) - '0');

carry += v1 \* v2;

}

}

ret.append(carry % 10);

carry = carry / 10;

}

if (carry != 0) {

ret.append(carry);

}

return ret.reverse().toString();

}

}

### Wildcard Matching

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

**Solution 1 – Time O (n\*m) Space (n)**

public class Solution {

public boolean isMatch(String s, String p) {

if (p.length() == 0)

return s.length() == 0;

if(s.length()>300 && p.charAt(0)=='\*' && p.charAt(p.length()-1)=='\*')

return false;

boolean[] ret = new boolean[s.length() + 1];

ret[0] = true;

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

if (p.charAt(j) != '\*') {

for (int i = s.length() - 1; i >= 0; i--) {

ret[i + 1] = ret[i]

&& (p.charAt(j) == '?' || s.charAt(i) == p

.charAt(j));

}

} else {

int i = 0;

while (i <= s.length() && !ret[i])

i++;

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

ret[i] = true;

}

}

ret[0] = ret[0] && p.charAt(j) == '\*';

}

return ret[s.length()];

}

}

### Jump Game II

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.)

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int jump(int[] A) {

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

return -1;

}

int step = 0;

int reach = 0;

int lastReach = 0;

for (int i = 0; i < A.length && i <= reach; i++) {

if (i > lastReach) {

step++;

lastReach = reach;

}

reach = Math.max(reach, A[i]+i);

}

if (reach >= A.length - 1) {

return step;

}

else {

return -1;

}

}

}

### Permutations

Given a collection of 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].

**Solution 1 – Time O (NP) Space (?)**

public class Solution {

public List<List<Integer>> permute(int[] num) {

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

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

return ret;

}

helper(ret, new ArrayList<Integer>(), new boolean[num.length], num);

return ret;

}

public void helper(List<List<Integer>> ret, List<Integer> item,

boolean[] used, int[] num) {

if (item.size() == num.length) {

ret.add(new ArrayList<Integer>(item));

return;

}

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

if (used[i] == false) {

item.add(num[i]);

used[i] = true;

helper(ret, item, used, num);

used[i] = false;

item.remove(item.size() - 1);

}

}

}

}

**Solution 2 – Time O (NP) Space (?)**

public class Solution {

public List<List<Integer>> permute(int[] num) {

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

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

return ret;

}

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

first.add(num[0]);

ret.add(first);

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

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

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

List<Integer> cur = ret.get(j);

for (int k = 0; k < cur.size()+1; k++) {

List<Integer> item = new ArrayList<Integer>(cur);

item.add(k, num[i]);

curRet.add(item);

}

}

ret = curRet;

}

return ret;

}

}

### Permutations II

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].

**Solution 1 – Time O (NP) Space (?)**

public class Solution {

public List<List<Integer>> permuteUnique(int[] num) {

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

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

return ret;

}

Arrays.sort(num);

helper(ret, new ArrayList<Integer>(), new boolean[num.length], num);

return ret;

}

public void helper(List<List<Integer>> ret, List<Integer> item,

boolean[] used, int[] num) {

if (item.size() == num.length) {

ret.add(new ArrayList<Integer>(item));

return;

}

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

if (i > 0 && used[i - 1] == false && num[i] == num[i - 1]) {

continue;

}

if (used[i] == false) {

used[i] = true;

item.add(num[i]);

helper(ret, item, used, num);

used[i] = false;

item.remove(item.size() - 1);

}

}

}

}

### Rotate Image

You are given an *n* x *n* 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up:  
Could you do this in-place?

**Solution 1 – Time O (n\*n) Space O (1)**

public class Solution {

public void rotate(int[][] matrix) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {

return;

}

int n = matrix.length;

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

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

int tmp = matrix[i][j];

matrix[i][j] = matrix[j][i];

matrix[j][i] = tmp;

}

}

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

for (int j = 0; j < n / 2; j++) {

int tmp = matrix[i][j];

matrix[i][j] = matrix[i][n - 1 - j];

matrix[i][n - 1 - j] = tmp;

}

}

}

}

**Solution 2 – Time O (n\*n) Space O (1)**

public class Solution {

public void rotate(int[][] matrix) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {

return;

}

int level = matrix.length / 2;

int n = matrix.length;

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

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

int tmp = matrix[i][j];

matrix[i][j] = matrix[n-1-j][i];

matrix[n-1-j][i] = matrix[n-1-i][n-1-j];

matrix[n-1-i][n-1-j] = matrix[j][n-1-i];

matrix[j][n-1-i] = tmp;

}

}

}

}

### Group Anagrams

Given an array of strings, group anagrams together.

For example, given: ["eat", "tea", "tan", "ate", "nat", "bat"],   
Return:

[

["ate", "eat","tea"],

["nat","tan"],

["bat"]

]

**Note:**

1. For the return value, each *inner* list's elements must follow the lexicographic order.
2. All inputs will be in lower-case.

**Solution 1 – Time O (nklogk if ignore the inner must follow order) Space O (nk)**

public class Solution {

public List<List<String>> groupAnagrams(String[] strs) {

List<List<String>> ret = new LinkedList<List<String>>();

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

return ret;

}

Map<String, List<String>> map = new HashMap<String, List<String>>();

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

char[] c = strs[i].toCharArray();

Arrays.sort(c);

String key = new String(c);

if (map.containsKey(key)) {

map.get(key).add(strs[i]);

} else {

List<String> list = new LinkedList<String>();

list.add(strs[i]);

map.put(key, list);

}

}

for (List<String> value : map.values()) {

Collections.sort(value);

ret.add(value);

}

return ret;

}

}

### Pow(x, n)

Implement pow(*x*, *n*).

**Solution 1 – Time O (lgn) Space O (lgn)**

// assume no overflow

public class Solution {

public double pow(double x, int n) {

if (n == 0) {

return 1.0;

}

double half = pow(x, n / 2);

if (n % 2 == 0) {

return half \* half;

}

if (n > 0) {

return half \* half \* x;

} else {

return half \* half / x;

}

}

}

**Solution 2 – Time O (lgn) Space O (1)**

public class Solution {

public double pow(double x, int n) {

if (n == 0) {

return 1.0;

}

double ret = 1.0;

if (n < 0) {

if (x >= 1.0 / Double.MAX\_VALUE || x <= 1.0 / -Double.MAX\_VALUE) {

x = 1 / x;

} else {

return Double.MAX\_VALUE;

}

if (n == Integer.MIN\_VALUE) {

ret \*= x;

n++;

}

}

n = Math.abs(n);

boolean isNeg = false;

if (n % 2 == 1 && x < 0) {

isNeg = true;

}

x = Math.abs(x);

while (n > 0) {

if (n % 2 == 1) {

if (ret > Double.MAX\_VALUE / x) {

return Double.MAX\_VALUE;

}

ret \*= x;

}

x \*= x;

n = n >> 1;

}

return isNeg ? -ret : ret;

}

}

**Solution 3 – Time O(lgn) Space O(1)**

// assume no overflow

public class Solution {

public double pow(double x, int n) {

if (n == 0) {

return 1.0;

}

double ret = 1.0;

if (n < 0) {

x = 1 / x;

}

n = Math.abs(n);

boolean isNeg = false;

if (n % 2 == 1 && x < 0) {

isNeg = true;

}

x = Math.abs(x);

while (n > 0) {

if (n % 2 == 1) {

ret \*= x;

}

x \*= x;

n = n >> 1;

}

return isNeg ? -ret : ret;

}

}

**Solution 4 – Time O (lg n) Space O (1)**

// Non recursive, very simple

public class Solution {

public double myPow(double x, int n) {

double result = 1.0;

for (int i = n; i != 0; i /= 2, x \*= x) {

if (i % 2 != 0) {

result \*= x;

}

}

return n < 0 ? 1.0 / result : result;

}

}

### N-Queens

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.."]

]

**Solution 1 – Time O (NP) Space O(n)**

public class Solution {

public List<String[]> solveNQueens(int n) {

List<String[]> ret = new ArrayList<String[]>();

if (n <= 0) {

return ret;

}

helper(n, 0, new int[n], ret);

return ret;

}

private void helper(int n, int row, int[] columnForRow, List<String[]> ret) {

if (row == n) {

String[] str = new String[n];

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

str[i] = "";

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

if (columnForRow[i] != j) {

str[i] += '.';

} else {

str[i] += 'Q';

}

}

}

ret.add(str);

return;

}

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

columnForRow[row] = i;

if (check(row, columnForRow)) {

helper(n, row + 1, columnForRow, ret);

}

}

}

private boolean check(int row, int[] columnForRow) {

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

if ((row - i == Math.abs(columnForRow[row] - columnForRow[i]))

|| (columnForRow[row] == columnForRow[i])) {

return false;

}

}

return true;

}

}

### N-Queens II

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.



**Solution 1 – Time O (NP) Space O (n)**

public class Solution {

public int totalNQueens(int n) {

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

;

ret.add(0);

if (n <= 0) {

return 0;

}

helper(n, 0, new int[n], ret);

return ret.get(0);

}

private void helper(int n, int row, int[] columnForRow, List<Integer> ret) {

if (row == n) {

ret.set(0, ret.get(0) + 1);

return;

}

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

columnForRow[row] = i;

if (check(row, columnForRow)) {

helper(n, row + 1, columnForRow, ret);

}

}

}

private boolean check(int row, int[] columnForRow) {

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

if ((row - i == Math.abs(columnForRow[row] - columnForRow[i]))

|| (columnForRow[row] == columnForRow[i])) {

return false;

}

}

return true;

}

}

### Maximum Subarray

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.](http://oj.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.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int maxSubArray(int[] nums) {

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

return 0;

}

int ret = Integer.MIN\_VALUE;

int cur = 0;

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

cur += nums[i];

ret = Math.max(ret, cur);

if (cur < 0) {

cur = 0;

}

}

return ret;

}

}

### Spiral Matrix

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].

**Solution 1 – Time O (m\*n) Space O (1)**

public class Solution {

public List<Integer> spiralOrder(int[][] matrix) {

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

if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {

return ret;

}

int r = matrix.length;

int c = matrix[0].length;

int min = Math.min(r, c);

for (int level = 0; level < min / 2; level++) {

for (int i = level; i < c - 1 - level; i++) {

ret.add(matrix[level][i]);

}

for (int i = level; i < r - 1 - level; i++) {

ret.add(matrix[i][c - 1 - level]);

}

for (int i = c - 1 - level; i > level; i--) {

ret.add(matrix[r - 1 - level][i]);

}

for (int i = r - 1 - level; i > level; i--) {

ret.add(matrix[i][level]);

}

}

if (min % 2 == 1) {

if (r < c) {

for (int i = min / 2; i < c - min / 2; i++) {

ret.add(matrix[r / 2][i]);

}

} else {

for (int i = min / 2; i < r - min / 2; i++) {

ret.add(matrix[i][c / 2]);

}

}

}

return ret;

}

}

**Solution 2 – Time O (m\*n) Space O (1)**

public class Solution {

public List<Integer> spiralOrder(int[][] matrix) {

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

if (matrix.length == 0)

return elements;

int m = matrix.length, n = matrix[0].length;

int row = 0, col = -1;

while (true) {

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

elements.add(matrix[row][++col]);

}

if (--m == 0)

break;

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

elements.add(matrix[++row][col]);

}

if (--n == 0)

break;

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

elements.add(matrix[row][--col]);

}

if (--m == 0)

break;

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

elements.add(matrix[--row][col]);

}

if (--n == 0)

break;

}

return elements;

}

}

### Jump Game

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.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public boolean canJump(int[] nums) {

int ret = 0;

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

return false;

}

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

if (ret < i) {

return false;

}

ret = Math.max(nums[i] + i, ret);

}

return ret >= nums.length - 1;

}

}

### Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

For example,  
Given [1,3],[2,6],[8,10],[15,18],  
return [1,6],[8,10],[15,18].

**Solution 1 – Time O (nlgn) Space O (1)**

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

public class Solution {

public List<Interval> merge(List<Interval> intervals) {

List<Interval> ret = new ArrayList<Interval>();

if (intervals == null || intervals.size() < 2) {

return intervals;

}

Comparator<Interval> comp = new Comparator<Interval>() {

public int compare(Interval i1, Interval i2) {

if (i1.start == i2.start) {

return i1.end - i2.end;

}

return i1.start - i2.start;

}

};

Collections.sort(intervals, comp);

ret.add(intervals.get(0));

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

int end = ret.get(ret.size() - 1).end;

int start = intervals.get(i).start;

if (start <= end) {

ret.get(ret.size() - 1).end = Math.max(end,

intervals.get(i).end);

} else {

ret.add(intervals.get(i));

}

}

return ret;

}

}

### Insert Interval

Given a set of *non-overlapping* intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

**Example 1:**  
Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].

**Example 2:**  
Given [1,2],[3,5],[6,7],[8,10],[12,16], insert and merge [4,9] in as [1,2],[3,10],[12,16].

This is because the new interval [4,9] overlaps with [3,5],[6,7],[8,10].

**Solution 1 – Time O (n) Space O (n)**

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

public class Solution {

public List<Interval> insert(List<Interval> intervals, Interval newInterval) {

List<Interval> ret = new ArrayList<Interval>();

if (intervals == null || intervals.size() == 0) {

ret.add(newInterval);

return ret;

}

int i = 0;

while (i < intervals.size() && intervals.get(i).end < newInterval.start) {

ret.add(intervals.get(i));

i++;

}

if (i == intervals.size()) {

ret.add(newInterval);

return ret;

}

newInterval.start = Math.min(intervals.get(i).start, newInterval.start);

ret.add(newInterval);

while (i < intervals.size() && newInterval.end >= intervals.get(i).start) {

newInterval.end = Math.max(intervals.get(i).end, newInterval.end);

i++;

}

while (i < intervals.size()) {

ret.add(intervals.get(i));

i++;

}

return ret;

}

}

### Length of Last Word

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.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int lengthOfLastWord(String s) {

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

return 0;

}

int ret = 0;

int i = s.length() - 1;

while (i >= 0 && s.charAt(i) == ' ') {

i--;

}

while (i >= 0 && s.charAt(i) != ' ') {

ret++;

i--;

}

return ret;

}

}

### Spiral Matrix II

Given an integer *n*, generate a square matrix filled with elements from 1 to *n*2 in spiral order.

For example,  
Given *n* = 3,

You should return the following matrix:

[

[ 1, 2, 3 ],

[ 8, 9, 4 ],

[ 7, 6, 5 ]

]

**Solution 1 – Time O (n\*n) Space O (1)**

public class Solution {

public int[][] generateMatrix(int n) {

if (n < 0) {

return null;

}

int[][] ret = new int[n][n];

int level = n / 2;

int num = 1;

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

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

ret[i][j] = num++;

}

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

ret[j][n - 1 - i] = num++;

}

for (int j = n - 1 - i; j >= i + 1; j--) {

ret[n - 1 - i][j] = num++;

}

for (int j = n - 1 - i; j >= i + 1; j--) {

ret[j][i] = num++;

}

}

if (n % 2 != 0) {

ret[level][level] = num;

}

return ret;

}

}

### Permutation Sequence

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.

**Solution 1 – Time O (n^2) Space O (n)**

public class Solution {

public String getPermutation(int n, int k) {

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

StringBuilder ret = new StringBuilder();

int factor = 1;

k--;

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

factor \*= i;

}

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

num.add(i);

}

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

int index = k / factor;

k = k % factor;

ret.append(num.get(index));

num.remove(index);

if (i > 0) {

factor = factor / i;

}

}

return ret.toString();

}

}

### Rotate List

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.

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode rotateRight(ListNode head, int k) {

if (head == null) {

return null;

}

ListNode cur = head;

int len = 1;

while (cur.next != null) {

len++;

cur = cur.next;

}

cur.next = head;

k = len - k % len;

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

cur = cur.next;

}

head = cur.next;

cur.next = null;

return head;

}

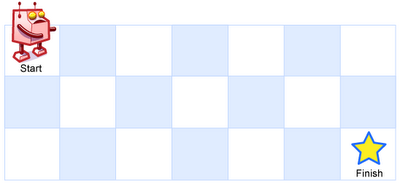
}

### Unique Paths

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.

**Solution 1 – Time O (min(m, n)) Space O (1)**

// Math method

public class Solution {

public int uniquePaths(int m, int n) {

return combination(m + n - 2, Math.max(m - 1, n - 1));

}

public int combination(int n, int k) {

return (int) (factor(k + 1, n) / factor(1, n - k));

}

public long factor(int start, int end) {

long ret = 1;

for (int i = start; i <= end; i++) {

ret \*= i;

}

return ret;

}

}

**Solution 2 – Time O (result) Space O (result)**

// Recursive: (May out of memory)

public class Solution {

public int uniquePaths(int m, int n) {

if (m < 1 || n < 1) {

return 0;

}

if (m == 1 || n == 1) {

return 1;

}

return uniquePaths(m - 1, n) + uniquePaths(m, n - 1);

}

}

**Solution 3 – Time O (n\*m) Space O (n)**

// Dynamic programming

public class Solution {

public int uniquePaths(int m, int n) {

if (m < 1 || n < 1) {

return 0;

}

int ret[] = new int[n];

ret[0] = 1;

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

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

ret[j] = ret[j - 1] + ret[j];

}

}

return ret[n - 1];

}

}

### Unique Paths II

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.

**Solution 1 – Time O (m\*n), Space O (n)**

public class Solution {

public int uniquePathsWithObstacles(int[][] obstacleGrid) {

if (obstacleGrid == null || obstacleGrid.length == 0

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

return 0;

}

int[] ret = new int[obstacleGrid[0].length];

ret[0] = 1;

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

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

if (obstacleGrid[i][j] == 1) {

ret[j] = 0;

} else {

if (j > 0) {

ret[j] = ret[j - 1] + ret[j];

}

}

}

}

return ret[obstacleGrid[0].length - 1];

}

}

### Minimum Path Sum

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.

**Solution 1 – Time O (m \* n) Space O (n)**

public class Solution {

public int minPathSum(int[][] grid) {

if (grid == null || grid.length == 0 || grid[0].length == 0) {

return 0;

}

int[] ret = new int[grid[0].length];

ret[0] = grid[0][0];

for (int j = 1; j < grid[0].length; j++) {

ret[j] = ret[j - 1] + grid[0][j];

}

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

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

if (j == 0) {

ret[j] = ret[j] + grid[i][j];

} else {

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

}

}

}

return ret[grid[0].length - 1];

}

}

### Valid Number

Validate if a given string is numeric.

Some examples:  
"0" => true  
" 0.1 " => true  
"abc" => false  
"1 a" => false  
"2e10" => true

**Note:** It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.

**Update (2015-02-10):**  
The signature of the C++ function had been updated. If you still see your function signature accepts a const char \* argument, please click the reload button to reset your code definition.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public boolean isNumber(String s) {

int i = 0, n = s.length();

while (i < n && Character.isWhitespace(s.charAt(i))) {

i++;

}

if (i < n && (s.charAt(i) == '+' || s.charAt(i) == '-')) {

i++;

}

boolean isNumeric = false;

while (i < n && Character.isDigit(s.charAt(i))) {

i++;

isNumeric = true;

}

if (i < n && s.charAt(i) == '.') {

i++;

while (i < n && Character.isDigit(s.charAt(i))) {

i++;

isNumeric = true;

}

}

if (isNumeric && i < n && s.charAt(i) == 'e') {

i++;

isNumeric = false;

if (i < n && (s.charAt(i) == '+' || s.charAt(i) == '-')) {

i++;

}

while (i < n && Character.isDigit(s.charAt(i))) {

i++;

isNumeric = true;

}

}

while (i < n && Character.isWhitespace(s.charAt(i))) {

i++;

}

return isNumeric && i == n;

}

}

**Solution 2 – Time O (n) Space O (1)**

public class Solution {

public boolean isNumber(String s) {

if (s == null) {

return false;

}

s = s.trim();

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

return false;

}

boolean dotFlag = false;

boolean eFlag = false;

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

switch (s.charAt(i)) {

case '.':

if (dotFlag || eFlag) {

return false;

}

if (((i == 0 || !(s.charAt(i - 1) >= '0' && s.charAt(i - 1) <= '9')) && (i == s

.length() - 1 || !(s.charAt(i + 1) >= '0' && s

.charAt(i + 1) <= '9'))))

return false;

dotFlag = true;

break;

case '+':

case '-':

if (i > 0 && (s.charAt(i - 1) != 'e' && s.charAt(i - 1) != 'E')) {

return false;

}

if (i == s.length() - 1

|| !(s.charAt(i + 1) == '.' || (s.charAt(i + 1) >= '0' && s

.charAt(i + 1) <= '9'))) {

return false;

}

break;

case 'E':

case 'e':

if (eFlag || i == 0 || i == s.length() - 1) {

return false;

}

eFlag = true;

break;

case '0':

case '1':

case '2':

case '3':

case '4':

case '5':

case '6':

case '7':

case '8':

case '9':

break;

default:

return false;

}

}

return true;

}

}

### Plus One

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.

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public int[] plusOne(int[] digits) {

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

return new int[] { 1 };

}

int carry = 1;

for (int i = digits.length - 1; i >= 0; i--) {

int digit = (digits[i] + carry) % 10;

carry = (digits[i] + carry) / 10;

digits[i] = digit;

if (carry == 0) {

return digits;

}

}

int[] ret = new int[digits.length + 1];

ret[0] = 1;

return ret;

}

}

### Add Binary

Given two binary strings, return their sum (also a binary string).

For example,  
a = "11"  
b = "1"  
Return "100".

**Solution 1 – Time O (max(m, n)) Space O (1)**

public class Solution {

public String addBinary(String a, String b) {

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

return b;

}

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

return a;

}

StringBuilder ret = new StringBuilder();

int carry = 0;

int i = a.length() - 1;

int j = b.length() - 1;

int num;

while (i >= 0 && j >= 0) {

num = a.charAt(i) - '0' + b.charAt(j) - '0' + carry;

ret.append(num % 2);

carry = num / 2;

i--;

j--;

}

while (i >= 0) {

num = a.charAt(i) - '0' + carry;

ret.append(num % 2);

carry = num / 2;

i--;

}

while (j >= 0) {

num = b.charAt(j) - '0' + carry;

ret.append(num % 2);

carry = num / 2;

j--;

}

if (carry > 0) {

ret.append(carry);

}

return ret.reverse().toString();

}

}

### Text Justification

Given an array of words and a length *L*, format the text such that each line has exactly *L* characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly *L* characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

For example,  
**words**: ["This", "is", "an", "example", "of", "text", "justification."]  
**L**: 16.

Return the formatted lines as:

[

"This is an",

"example of text",

"justification. "

]

**Note:** Each word is guaranteed not to exceed *L* in length.

**Solution 1 – Time O (n) Space O (r \* L)**

public class Solution {

public List<String> fullJustify(String[] words, int L) {

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

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

return ret;

}

int last = 0;

int count = 0;

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

int spaceNum = 0;

int extraNum = 0;

if (count + words[i].length() + (i - last) > L) {

if (i - last > 1) {

spaceNum = (L - count) / (i - last - 1);

extraNum = (L - count) % (i - last - 1);

}

StringBuilder str = new StringBuilder();

for (int j = last; j < i; j++) {

str.append(words[j]);

if (j < i - 1) {

for (int k = 0; k < spaceNum; k++) {

str.append(" ");

}

if (extraNum > 0) {

str.append(" ");

}

extraNum--;

}

}

for (int j = str.length(); j < L; j++) {

str.append(" ");

}

ret.add(str.toString());

last = i;

count = 0;

}

count += words[i].length();

}

StringBuilder str = new StringBuilder();

for (int i = last; i < words.length; i++) {

str.append(words[i]);

if (str.length() < L) {

str.append(" ");

}

}

for (int i = str.length(); i < L; i++) {

str.append(" ");

}

ret.add(str.toString());

return ret;

}

}

### Sqrt(x)

Implement int sqrt(int x).

Compute and return the square root of *x*.

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public int sqrt(int x) {

if (x < 0) {

return -1;

}

if (x == 0) {

return 0;

}

int l = 1;

int r = x;

int m = 0;

while (l <= r) {

m = (l + r) / 2;

if (x / m >= m && x / (m + 1) < m + 1) {

return m;

}

if (x / m < m) {

r = m - 1;

} else {

l = m + 1;

}

}

return m;

}

}

**Solution 2 – Time O (Log n) Space O (1)**

public class Solution {

public int sqrt(int x) {

if (x < 0) {

return -1;

}

if (x == 0) {

return 0;

}

double yLast = 0;

double y = 1;

while (y != yLast) {

yLast = y;

y = (y + x / y) / 2;

}

return (int) y;

}

}

### Climbing Stairs

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?

**Solution 1 – Time O (n) Space O (n)**

// Recursive, may out of memory

public class Solution {

public int climbStairs(int n) {

if (n < 1) {

return 0;

}

if (n == 1) {

return 1;

} else {

return climbStairs(n - 1) + climbStairs(n - 2);

}

}

}

**Solution 2 – Time O (n) Space O (1)**

public class Solution {

public int climbStairs(int n) {

if (n < 1) {

return 0;

}

if (n < 3) {

return n;

}

int first = 1;

int second = 2;

for (int i = 3; i <= n; i++) {

int tmp = first + second;

first = second;

second = tmp;

}

return second;

}

}

### Simplify Path

Given an absolute path for a file (Unix-style), simplify it.

For example,  
**path** = "/home/", => "/home"  
**path** = "/a/./b/../../c/", => "/c"

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public String simplifyPath(String path) {

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

return "";

}

Deque<String> stack = new LinkedList<String>();

String[] list = path.split("/");

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

if (list[i].equals(".") || list[i].equals("")) {

continue;

}

if (stack.isEmpty() && list[i].equals("..")) {

continue;

}

if (!stack.isEmpty() && list[i].equals("..")) {

stack.pop();

} else {

stack.push(list[i]);

}

}

String ret = "";

if (stack.isEmpty()) {

return "/";

}

while (!stack.isEmpty()) {

String cur = stack.pop();

ret = "/" + cur + ret;

}

return ret;

}

}

**Solution 2 – Time O (n) Space O (n)**

public class Solution {

public String simplifyPath(String path) {

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

return "";

}

LinkedList<String> stack = new LinkedList<String>();

StringBuilder ret = new StringBuilder();

int i = 0;

while (i < path.length()) {

int index = i;

StringBuilder tmp = new StringBuilder();

while (i < path.length() && path.charAt(i) != '/') {

tmp.append(path.charAt(i));

i++;

}

if (index != i) {

String str = tmp.toString();

if (str.equals("..")) {

if (!stack.isEmpty()) {

stack.pop();

}

} else if (!str.equals(".")) {

stack.push(str);

}

}

i++;

}

if (!stack.isEmpty()) {

String[] strs = stack.toArray(new String[stack.size()]);

for (int j = strs.length - 1; j >= 0; j--) {

ret.append("/" + strs[j]);

}

}

if (ret.length() == 0) {

return "/";

}

return ret.toString();

}

}

### Edit Distance

Given two words *word1* and *word2*, find the minimum number of steps required to convert *word1* to *word2*. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

a) Insert a character  
b) Delete a character  
c) Replace a character

**Solution 1 – Time O (m \* n) Space O (n)**

public class Solution {

public int minDistance(String word1, String word2) {

if (word1.length() == 0) {

return word2.length();

}

if (word2.length() == 0) {

return word1.length();

}

int n = word1.length();

int m = word2.length();

int[] ret = new int[m + 1];

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

ret[i] = i;

}

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

int[] cur = new int[m + 1];

cur[0] = i + 1;

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

if (word1.charAt(i) == word2.charAt(j)) {

cur[j + 1] = ret[j];

} else {

cur[j + 1] = Math.min(cur[j], Math.min(ret[j], ret[j + 1])) + 1;

}

}

ret = cur;

}

return ret[m];

}

}

### Set Matrix Zeroes

Given a *m* x *n* matrix, if an element is 0, set its entire row and column to 0. Do it in place.

[click to show follow up.](http://oj.leetcode.com/problems/set-matrix-zeroes/)

**Follow up:**

Did you use extra space?  
A straight forward solution using O(*mn*) space is probably a bad idea.  
A simple improvement uses O(*m* + *n*) space, but still not the best solution.  
Could you devise a constant space solution?

**Solution 1 – Time O (mn) Space O (1)**

public class Solution {

public void setZeroes(int[][] matrix) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {

return;

}

boolean rowFlag = false;

boolean columnFlga = false;

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

if (matrix[0][i] == 0) {

rowFlag = true;

break;

}

}

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

if (matrix[i][0] == 0) {

columnFlga = true;

break;

}

}

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

for (int j = 1; j < matrix[0].length; j++) {

if (matrix[i][j] == 0) {

matrix[0][j] = 0;

matrix[i][0] = 0;

}

}

}

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

for (int j = 1; j < matrix[0].length; j++) {

if (matrix[0][j] == 0 || matrix[i][0] == 0) {

matrix[i][j] = 0;

}

}

}

if (rowFlag) {

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

matrix[0][i] = 0;

}

}

if (columnFlga) {

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

matrix[i][0] = 0;

}

}

}

}

### Search a 2D Matrix

Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:

* Integers in each row are sorted from left to right.
* The first integer of each row is greater than the last integer of the previous row.

For example,

Consider the following matrix:

[

[1, 3, 5, 7],

[10, 11, 16, 20],

[23, 30, 34, 50]

]

Given **target** = 3, return true.

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public boolean searchMatrix(int[][] matrix, int target) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {

return false;

}

int r = matrix.length;

int c = matrix[0].length;

int left = 0;

int right = r \* c - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (matrix[mid / c][mid % c] == target) {

return true;

} else if (matrix[mid / c][mid % c] < target) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return false;

}

}

**Solution 2 – Time O (log m + log n) Space O (1)**

public class Solution {

public boolean searchMatrix(int[][] matrix, int target) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {

return false;

}

int l = 0;

int r = matrix.length - 1;

boolean findRow = false;

while (l <= r) {

int mid = (l + r) / 2;

if (matrix[mid][0] <= target

&& matrix[mid][matrix[0].length - 1] >= target) {

findRow = true;

break;

}

if (matrix[mid][0] > target) {

r = mid - 1;

} else {

l = mid + 1;

}

}

int row;

if (findRow) {

row = (l + r) / 2;

} else {

return false;

}

l = 0;

r = matrix[0].length - 1;

while (l <= r) {

int mid = (l + r) / 2;

if (matrix[row][mid] == target) {

return true;

} else if (matrix[row][mid] > target) {

r = mid - 1;

} else {

l = mid + 1;

}

}

return false;

}

}

### Sort Colors

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.](http://oj.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?

**Solution 1 – Time O (3n) Space O (n)**

public class Solution {

public void sortColors(int[] A) {

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

return;

}

int[] count = new int[3];

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

count[A[i]]++;

}

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

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

}

int[] ret = new int[A.length];

for (int i = A.length - 1; i >= 0; i--) {

ret[count[A[i]] - 1] = A[i];

count[A[i]]--;

}

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

A[i] = ret[i];

}

}

}

**Solution 2 – Time O (n) Space O (1)**

public class Solution {

public void sortColors(int[] A) {

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

return;

}

int idx0 = 0;

int idx1 = 0;

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

if (A[i] == 0) {

A[i] = 2;

A[idx1++] = 1;

A[idx0++] = 0;

} else if (A[i] == 1) {

A[i] = 2;

A[idx1++] = 1;

}

}

}

}

**Solution 3 – Time O (n) Space O (1)**

public class Solution {

public void sortColors(int[] nums) {

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

return;

}

int red = 0;

int blue = nums.length - 1;

int i = 0;

while (i <= blue) {

if (nums[i] == 0) {

int tmp = nums[i];

nums[i] = nums[red];

nums[red++] = tmp;

i++;

} else if (nums[i] == 1) {

i++;

} else {

int tmp = nums[i];

nums[i] = nums[blue];

nums[blue--] = tmp;

}

}

}

}

### Minimum Window Substring

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

For example,  
**S** = "ADOBECODEBANC"  
**T** = "ABC"

Minimum window is "BANC".

**Note:**  
If there is no such window in S that covers all characters in T, return the emtpy string "".

If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

**Solution 1 – Time O (n) Space O (size of dictionary)**

public class Solution {

public String minWindow(String S, String T) {

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

return "";

}

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

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

if (map.containsKey(T.charAt(i))) {

map.put(T.charAt(i), map.get(T.charAt(i)) + 1);

} else {

map.put(T.charAt(i), 1);

}

}

int left = 0;

int count = 0;

int minLen = S.length() + 1;

int minStart = 0;

for (int right = 0; right < S.length(); right++) {

if (map.containsKey(S.charAt(right))) {

map.put(S.charAt(right), map.get(S.charAt(right)) - 1);

if (map.get(S.charAt(right)) >= 0) {

count++;

}

while (count == T.length()) {

if (right - left + 1 < minLen) {

minLen = right - left + 1;

minStart = left;

}

if (map.containsKey(S.charAt(left))) {

map.put(S.charAt(left), map.get(S.charAt(left)) + 1);

if (map.get(S.charAt(left)) > 0) {

count--;

}

}

left++;

}

}

}

if (minLen > S.length()) {

return "";

}

return S.substring(minStart, minStart + minLen);

}

}

### Combinations

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],

]

**Solution 1 – Tim e O (NP) Space O (?)**

// Recursive

public class Solution {

public List<List<Integer>> combine(int n, int k) {

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

if (n < k || n < 1) {

return ret;

}

helper(ret, new ArrayList<Integer>(), 1, n, k);

return ret;

}

public void helper(List<List<Integer>> ret, List<Integer> item, int start,

int n, int k) {

if (item.size() == k) {

ret.add(new ArrayList<Integer>(item));

return;

}

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

item.add(i);

helper(ret, item, i + 1, n, k);

item.remove(item.size() - 1);

}

}

}

**Solution 2 – Time O (NP) Space O (?)**

public class Solution {

public List<List<Integer>> combine(int n, int k) {

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

if (n < k || n < 1) {

return ret;

}

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

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

ArrayList<Integer> b = new ArrayList<Integer>();

b.add(i);

prev.add(b);

}

ret = new ArrayList<List<Integer>>(prev);

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

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

for (List<Integer> temp : ret) {

int a = temp.get(temp.size() - 1);

for (int j = a + 1; j <= n - k + 1 + i; j++) {

ArrayList<Integer> b = new ArrayList<Integer>(temp);

b.add(j);

prev.add(b);

}

}

ret = new ArrayList<List<Integer>>(prev);

}

return ret;

}

}

### Subsets

Given a set of distinct integers, *S*, 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 ***S*** = [1,2,3], a solution is:

**Solution 1 – Time O (2^n) Space O (2^n)**

// Iterative

public class Solution {

public List<List<Integer>> subsets(int[] S) {

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

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

return ret;

}

ret.add(new ArrayList<Integer>());

Arrays.sort(S);

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

int size = ret.size();

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

List<Integer> item = new ArrayList<Integer>(ret.get(j));

item.add(S[i]);

ret.add(item);

}

}

return ret;

}

}

**Solution 2 – Time O (NP) Space O (n)**

//Recursive

public class Solution {

public List<List<Integer>> subsets(int[] S) {

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

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

return ret;

}

ret.add(new ArrayList<Integer>());

Arrays.sort(S);

helper(ret, S, 0);

return ret;

}

public void helper(List<List<Integer>> ret, int[] num, int index) {

if (index == num.length) {

return;

}

int size = ret.size();

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

ArrayList<Integer> item = new ArrayList<Integer>(ret.get(i));

item.add(num[index]);

ret.add(item);

}

helper(ret, num, index + 1);

}

}

### Word Search

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** =

[

["ABCE"],

["SFCS"],

["ADEE"]

]

**word** = "ABCCED", -> returns true,  
**word** = "SEE", -> returns true,  
**word** = "ABCB", -> returns false.

**Solution 1 – Time O (m^2 \* n^2) Space O (m\*n)**

public class Solution {

public boolean exist(char[][] board, String word) {

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

return true;

}

if (board == null || board.length == 0 || board[0].length == 0) {

return false;

}

boolean[][] used = new boolean[board.length][board[0].length];

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

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

if (search(board, word, 0, i, j, used))

return true;

}

}

return false;

}

private boolean search(char[][] board, String word, int index, int i,

int j, boolean[][] used) {

if (index == word.length()) {

return true;

}

if (i < 0 || j < 0 || i >= board.length || j >= board[0].length

|| used[i][j] || board[i][j] != word.charAt(index)) {

return false;

}

used[i][j] = true;

boolean ret = search(board, word, index + 1, i - 1, j, used)

|| search(board, word, index + 1, i + 1, j, used)

|| search(board, word, index + 1, i, j - 1, used)

|| search(board, word, index + 1, i, j + 1, used);

used[i][j] = false;

return ret;

}

}

### Remove Duplicates from Sorted Array II

Follow up for "Remove Duplicates":  
What if duplicates are allowed at most *twice*?

For example,  
Given sorted array A = [1,1,1,2,2,3],

Your function should return length = 5, and A is now [1,1,2,2,3].

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int removeDuplicates(int[] A) {

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

return 0;

}

if (A.length <= 2) {

return A.length;

}

int count = 2;

for (int i = 2; i < A.length; i++) {

if (A[i] == A[count - 1] && A[count - 1] == A[count - 2]) {

continue;

} else {

A[count] = A[i];

count++;

}

}

return count;

}

}

### Search in Rotated Sorted Array II

Follow up for "Search in Rotated Sorted Array":  
What if *duplicates* are allowed?

Would this affect the run-time complexity? How and why?

Write a function to determine if a given target is in the array.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public boolean search(int[] A, int target) {

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

return false;

}

int l = 0;

int r = A.length - 1;

while (l <= r) {

int mid = (l + r) / 2;

if (A[mid] == target) {

return true;

} else if (A[l] < A[mid]) {

if (target >= A[l] && target < A[mid]) {

r = mid - 1;

} else {

l = mid + 1;

}

} else if (A[mid] < A[l]) {

if (target > A[mid] && target <= A[r]) {

l = mid + 1;

} else {

r = mid - 1;

}

} else {

l++;

}

}

return false;

}

}

### Remove Duplicates from Sorted List II

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.

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode deleteDuplicates(ListNode head) {

if (head == null) {

return head;

}

ListNode ret = new ListNode(0);

ret.next = head;

ListNode prev = ret;

ListNode cur = head;

boolean isSame = false;

while (cur.next != null) {

if (cur.next.val == cur.val) {

cur.next = cur.next.next;

isSame = true;

} else {

if (isSame) {

prev.next = cur.next;

isSame = false;

} else {

prev = prev.next;

}

cur = cur.next;

}

}

if (isSame) {

prev.next = cur.next;

}

return ret.next;

}

}

### Remove Duplicates from Sorted List

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.

Solution 1 – Time O (n) Space O (1)

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode deleteDuplicates(ListNode head) {

if (head == null)

return head;

ListNode pre = head;

ListNode cur = head.next;

while (cur != null) {

if (cur.val == pre.val) {

pre.next = cur.next;

} else {

pre = cur;

}

cur = cur.next;

}

return head;

}

}

### Largest Rectangle in Histogram

Given *n* non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.



Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].



The largest rectangle is shown in the shaded area, which has area = 10 unit.

For example,  
Given height = [2,1,5,6,2,3],  
return 10.

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public int largestRectangleArea(int[] height) {

if (height == null || height.length == 0)

return 0;

int max = 0;

LinkedList<Integer> stack = new LinkedList<Integer>();

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

while (!stack.isEmpty() && height[i] <= height[stack.peek()]) {

int index = stack.pop();

int curArea = stack.isEmpty() ? i \* height[index] : (i

- stack.peek() - 1)

\* height[index];

max = Math.max(max, curArea);

}

stack.push(i);

}

while (!stack.isEmpty()) {

int index = stack.pop();

int curArea = stack.isEmpty() ? height.length \* height[index]

: (height.length - stack.peek() - 1) \* height[index];

max = Math.max(max, curArea);

}

return max;

}

}

### Maximal Rectangle

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

Solution 1 – Time O (n^2) Space O (n)

public class Solution {

public int maximalRectangle(char[][] matrix) {

if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {

return 0;

}

int m = matrix.length;

int n = matrix[0].length;

int[] left = new int[n];

int[] right = new int[n];

int[] height = new int[n];

Arrays.fill(right, n);

int max = 0;

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

int curLeft = 0;

int curRight = n;

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

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

height[j]++;

} else {

height[j] = 0;

}

}

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

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

left[j] = Math.max(left[j], curLeft);

} else {

left[j] = 0;

curLeft = j + 1;

}

}

for (int j = n - 1; j >= 0; j--) {

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

right[j] = Math.min(right[j], curRight);

} else {

right[j] = n;

curRight = j;

}

}

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

max = Math.max(max, (right[j] - left[j]) \* height[j]);

}

}

return max;

}

}

### Partition List

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.

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode partition(ListNode head, int x) {

if (head == null) {

return head;

}

ListNode l1 = new ListNode(0);

ListNode l2 = new ListNode(0);

ListNode p1 = l1;

ListNode p2 = l2;

while (head != null) {

if (head.val < x) {

p1.next = head;

p1 = p1.next;

}

else {

p2.next = head;

p2 = p2.next;

}

head = head.next;

}

p1.next = l2.next;

p2.next = null;

return l1.next;

}

}

**Solution 2 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode partition(ListNode head, int x) {

if (head == null) {

return head;

}

ListNode ret = new ListNode(0);

ret.next = head;

ListNode prev = ret;

ListNode cur = ret;

while (cur.next != null) {

if (cur.next.val < x) {

if (prev == cur) {

prev = prev.next;

cur = cur.next;

} else {

ListNode tmp = cur.next.next;

cur.next.next = prev.next;

prev.next = cur.next;

cur.next = tmp;

prev = prev.next;

}

} else {

cur = cur.next;

}

}

return ret.next;

}

}

### Scramble String

Given a string *s1*, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Below is one possible representation of *s1* = "great":

great

/ \

gr eat

/ \ / \

g r e at

/ \

a t

To scramble the string, we may choose any non-leaf node and swap its two children.

For example, if we choose the node "gr" and swap its two children, it produces a scrambled string "rgeat".

rgeat

/ \

rg eat

/ \ / \

r g e at

/ \

a t

We say that "rgeat" is a scrambled string of "great".

Similarly, if we continue to swap the children of nodes "eat" and "at", it produces a scrambled string "rgtae".

rgtae

/ \

rg tae

/ \ / \

r g ta e

/ \

t a

We say that "rgtae" is a scrambled string of "great".

Given two strings *s1* and *s2* of the same length, determine if *s2* is a scrambled string of *s1*.

**Solution 1 – Time O (n^4) Space O (n^3)**

public class Solution {

public boolean isScramble(String s1, String s2) {

if (s1 == null || s2 == null || s1.length() != s2.length())

return false;

if (s1.length() == 0)

return true;

boolean[][][] res = new boolean[s1.length()][s2.length()][s1.length() + 1];

for (int i = 0; i < s1.length(); i++) {

for (int j = 0; j < s2.length(); j++) {

res[i][j][1] = s1.charAt(i) == s2.charAt(j);

}

}

for (int len = 2; len <= s1.length(); len++) {

for (int i = 0; i < s1.length() - len + 1; i++) {

for (int j = 0; j < s2.length() - len + 1; j++) {

for (int k = 1; k < len; k++) {

res[i][j][len] |= res[i][j][k]

&& res[i + k][j + k][len - k]

|| res[i][j + len - k][k]

&& res[i + k][j][len - k];

}

}

}

}

return res[0][0][s1.length()];

}

}

### Merge Sorted Array

Given two sorted integer arrays A and B, merge B into A as one sorted array.

**Note:**  
You may assume that A has enough space (size that is greater or equal to *m* + *n*) to hold additional elements from B. The number of elements initialized in A and B are *m* and *n* respectively.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public void merge(int A[], int m, int B[], int n) {

if (A == null || B == null) {

return;

}

int idx1 = m - 1;

int idx2 = n - 1;

int idx3 = m + n - 1;

while (idx1 >= 0 && idx2 >= 0) {

if (A[idx1] > B[idx2]) {

A[idx3--] = A[idx1];

idx1--;

} else {

A[idx3--] = B[idx2];

idx2--;

}

}

while (idx2 >= 0) {

A[idx3--] = B[idx2--];

}

}

}

### Gray Code

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.

**Solution 1 – Time O (2^n) Space O (2^n)**

public class Solution {

public List<Integer> grayCode(int n) {

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

if (n < 0) {

return ret;

}

if (n == 0) {

ret.add(0);

return ret;

}

ret.add(0);

ret.add(1);

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

int size = ret.size();

for (int j = size - 1; j >= 0; j--) {

ret.add(ret.get(j) + (1 << (i - 1)));

}

}

return ret;

}

}

### Subsets II

Given a collection of integers that might contain duplicates, *S*, 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 ***S*** = [1,2,2], a solution is:

[

[2],

[1],

[1,2,2],

[2,2],

[1,2],

[]

]

**Solution 1 – Time O (2^n) Space O (2^n)**

// Iterative

public class Solution {

public List<List<Integer>> subsetsWithDup(int[] num) {

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

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

return ret;

}

ret.add(new ArrayList<Integer>());

Arrays.sort(num);

int start = 0;

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

int size = ret.size();

for (int j = start; j < size; j++) {

List<Integer> item = new ArrayList<Integer>(ret.get(j));

item.add(num[i]);

ret.add(item);

}

if (i < num.length - 1 && num[i] == num[i + 1]) {

start = size;

} else {

start = 0;

}

}

return ret;

}

}

### Decode Ways

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.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int numDecodings(String s) {

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

return 0;

}

int num1 = 1;

int num2 = 1;

int num3;

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

if (s.charAt(i) == '0') {

if (s.charAt(i - 1) == '1' || s.charAt(i - 1) == '2') {

num3 = num1;

} else {

return 0;

}

} else {

if (s.charAt(i - 1) == '0' || s.charAt(i - 1) >= '3') {

num3 = num2;

} else {

if (s.charAt(i - 1) == '2' && s.charAt(i) >= '7'

&& s.charAt(i) <= '9') {

num3 = num2;

} else {

num3 = num1 + num2;

}

}

}

num1 = num2;

num2 = num3;

}

return num2;

}

}

### Reverse Linked List II

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.

**Solution – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode reverseBetween(ListNode head, int m, int n) {

if (head == null) {

return head;

}

ListNode ret = new ListNode(0);

ListNode prev = ret;

ret.next = head;

for (int i = 0; i < m - 1; i++) {

prev = prev.next;

}

ListNode cur = prev.next;

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

ListNode node = cur.next;

cur.next = node.next;

node.next = prev.next;

prev.next = node;

}

return ret.next;

}

}

### Restore IP Addresses

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)

**Solution 1 – Time O (NP) Space O (NP) – recursive**

public class Solution {

public List<String> restoreIpAddresses(String s) {

List<String> ret = new LinkedList<String>();

if (s == null || s.length() < 4 || s.length() > 12) {

return ret;

}

helper(ret, s, new String(), 0, 0);

return ret;

}

public void helper(List<String> ret, String s, String tmp, int start, int section) {

if (start == s.length() && section == 4) {

ret.add(new String(tmp.substring(0, tmp.length() - 1)));

return;

}

for (int i = start; i < start + 3; i++) {

if (i >= s.length()) {

return;

}

if (s.charAt(start) == '0' && i > start) {

return;

}

int val = 0;

for (int j = start; j <= i; j++) {

val = val \* 10 + (s.charAt(j) - '0');

}

if (val <= 255) {

helper(ret, s, tmp + Integer.toString(val) + '.', i + 1,

section + 1);

}

}

}

}

**Solution 2 – Time O (NP) Space O (NP) - iterative**

// Iterative

public class Solution {

public List<String> restoreIpAddresses(String s) {

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

if (s == null || s.length() < 4 || s.length() > 12) {

return ret;

}

for (int i = 1; i < 4; i++) {

String first = s.substring(0, i);

if (!isValid(first)) {

continue;

}

for (int j = 1; j < 4 && (i + j) < s.length(); j++) {

String second = s.substring(i, i + j);

if (!isValid(second)) {

continue;

}

for (int k = 1; k < 4 && (i + j + k) < s.length(); k++) {

String third = s.substring(i + j, i + j + k);

String fourth = s.substring(i + j + k);

if (!isValid(third) || !isValid(fourth)) {

continue;

}

String tmp = first + '.' + second + '.' + third + '.'

+ fourth;

ret.add(tmp);

}

}

}

return ret;

}

public boolean isValid(String str) {

if (str.length() > 1 && str.charAt(0) == '0') {

return false;

}

if (str.length() > 3) {

return false;

}

int num = 0;

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

num = num \* 10 + str.charAt(i) - '0';

}

if (num > 255) {

return false;

}

return true;

}

}

**Solution 2 – Time O (NP) Space O (NP) - recursive**

// Recursive

public class Solution {

public List<String> restoreIpAddresses(String s) {

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

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

return ret;

}

helper(ret, s, "", 0, 1);

return ret;

}

public void helper(List<String> ret, String s, String item, int index,

int segment) {

if (index >= s.length()) {

return;

}

if (segment == 4) {

String str = s.substring(index);

if (isValid(str)) {

ret.add(item + "." + str);

}

return;

}

for (int i = 1; i < 4 && (index + i <= s.length()); i++) {

String str = s.substring(index, index + i);

if (isValid(str)) {

if (segment == 1) {

helper(ret, s, str, index + i, segment + 1);

} else {

helper(ret, s, item + "." + str, index + i, segment + 1);

}

}

}

}

private boolean isValid(String str) {

if (str == null || str.length() > 3) {

return false;

}

if (str.charAt(0) == '0' && str.length() > 1) {

return false;

}

int num = Integer.parseInt(str);

if (num >= 0 && num <= 255) {

return true;

}

return false;

}

}

### Binary Tree Inorder Traversal

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?

confused what "{1,#,2,3}" means? [> read more on how binary tree is serialized on OJ.](http://oj.leetcode.com/problems/binary-tree-inorder-traversal/)

**OJ's Binary Tree Serialization:**

The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

1

/ \

2 3

/

4

\

5

The above binary tree is serialized as "{1,2,3,#,#,4,#,#,5}".

**Solution 1 – Time O (n) Space O (logn)**

// Recursive

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> inorderTraversal(TreeNode root) {

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

helper(ret, root);

return ret;

}

private void helper(List<Integer> ret, TreeNode root) {

if (root == null) {

return;

}

helper(ret, root.left);

ret.add(root.val);

helper(ret, root.right);

}

}

**Solution 2– Time O (n) Space O (logn)**

// Iterative

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> inorderTraversal(TreeNode root) {

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

Stack<TreeNode> s = new Stack<TreeNode>();

while (root != null || !s.isEmpty()) {

if (root != null) {

s.push(root);

root = root.left;

} else {

root = s.pop();

ret.add(root.val);

root = root.right;

}

}

return ret;

}

}

**Solution 3 – Time O (n) Space O (1)**

// Morris Algorithm

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> inorderTraversal(TreeNode root) {

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

TreeNode cur = root;

TreeNode pre = null;

while (cur != null) {

if (cur.left == null) {

ret.add(cur.val);

cur = cur.right;

} else {

pre = cur.left;

while (pre.right != null && pre.right != cur) {

pre = pre.right;

}

if (pre.right == null) {

pre.right = cur;

cur = cur.left;

} else {

pre.right = null;

ret.add(cur.val);

cur = cur.right;

}

}

}

return ret;

}

}

### Unique Binary Search Trees II

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

confused what "{1,#,2,3}" means? [> read more on how binary tree is serialized on OJ.](https://leetcode.com/problems/unique-binary-search-trees-ii/)

**Solution 1 – Time O (NP) Space O (NP)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; left = null; right = null; }

\* }

\*/

public class Solution {

public List<TreeNode> generateTrees(int n) {

return helper(1, n);

}

private List<TreeNode> helper(int left, int right) {

List<TreeNode> ret = new ArrayList<TreeNode>();

if (left > right) {

ret.add(null);

return ret;

}

for (int i = left; i <= right; i++) {

List<TreeNode> leftList = helper(left, i - 1);

List<TreeNode> rightList = helper(i + 1, right);

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

for (int k = 0; k < rightList.size(); k++) {

TreeNode root = new TreeNode(i);

root.left = leftList.get(j);

root.right = rightList.get(k);

ret.add(root);

}

}

}

return ret;

}

}

### Unique Binary Search Trees

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

**Solution 1 – Time O (n^2) Space O (n)**

public class Solution {

public int numTrees(int n) {

if (n <= 0) {

return 0;

}

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

ret[0] = 1;

ret[1] = 1;

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

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

ret[i] += ret[j] \* ret[i - 1 - j];

}

}

return ret[n];

}

}

### Interleaving String

Given *s1*, *s2*, *s3*, find whether *s3* is formed by the interleaving of *s1* and *s2*.

For example,  
Given:  
*s1* = "aabcc",  
*s2* = "dbbca",

When *s3* = "aadbbcbcac", return true.  
When *s3* = "aadbbbaccc", return false.

**Solution 1 – Time O (m \* n) Space O (min (m, n))**

public class Solution {

public boolean isInterleave(String s1, String s2, String s3) {

if (s1.length() + s2.length() != s3.length()) {

return false;

}

String minWord = s1.length() > s2.length() ? s2 : s1;

String maxWord = s1.length() > s2.length() ? s1 : s2;

boolean[] ret = new boolean[minWord.length() + 1];

ret[0] = true;

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

ret[i + 1] = ret[i] && minWord.charAt(i) == s3.charAt(i);

}

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

ret[0] = ret[0] && maxWord.charAt(i) == s3.charAt(i);

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

ret[j + 1] = ret[j + 1]

&& maxWord.charAt(i) == s3.charAt(i + j + 1) || ret[j]

&& minWord.charAt(j) == s3.charAt(i + j + 1);

}

}

return ret[minWord.length()];

}

}

**Solution 2 – O (n^2) Space O (n^2)**

public class Solution {

public boolean isInterleave(String s1, String s2, String s3) {

int l1 = s1.length();

int l2 = s2.length();

int l3 = s3.length();

if (l3 != l1 + l2) {

return false;

}

boolean[][] ret = new boolean[l1 + 1][l2 + 1];

ret[0][0] = true;

for (int j = 0; j < l2; j++) {

ret[0][j + 1] = ret[0][j] && s2.charAt(j) == s3.charAt(j);

}

for (int i = 0; i < l1; i++) {

ret[i + 1][0] = ret[i][0] && s1.charAt(i) == s3.charAt(i);

}

for (int i = 0; i < l1; i++) {

for (int j = 0; j < l2; j++) {

ret[i + 1][j + 1] = (ret[i + 1][j] && s2.charAt(j) == s3

.charAt(i + j + 1))

|| (ret[i][j + 1] && s1.charAt(i) == s3.charAt(i + j

+ 1));

}

}

return ret[l1][l2];

}

}

### Validate Binary Search Tree

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.

confused what "{1,#,2,3}" means? [> read more on how binary tree is serialized on OJ.](https://leetcode.com/problems/validate-binary-search-tree/)

**Solution 1 – Time O (n) Space O (log n)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isValidBST(TreeNode root) {

return helper(root, null, null);

}

private boolean helper(TreeNode root, Integer minValue, Integer maxValue) {

if (root == null) {

return true;

}

if ((minValue != null && root.val <= minValue)

|| (maxValue != null && root.val >= maxValue)) {

return false;

}

return helper(root.left, minValue, root.val)

&& helper(root.right, root.val, maxValue);

}

}

**Solution 2 – Time O (n) Space O (log n)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isValidBST(TreeNode root) {

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

return helper(root, ret);

}

private boolean helper(TreeNode root, List<Integer> ret) {

if (root == null) {

return true;

}

boolean left = helper(root.left, ret);

ret.add(root.val);

if (ret.size() != 1 && ret.get(ret.size() - 2) >= root.val) {

return false;

}

return left && helper(root.right, ret);

}

}

### Recover Binary Search Tree

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

**Note:**  
A solution using O(*n*) space is pretty straight forward. Could you devise a constant space solution?

confused what "{1,#,2,3}" means? [> read more on how binary tree is serialized on OJ.](https://leetcode.com/problems/recover-binary-search-tree/)

**Solution 1 – Time O (n) Space O (n)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public void recoverTree(TreeNode root) {

if (root == null)

return;

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

List<TreeNode> ret = new ArrayList<TreeNode>();

helper(root, list, ret);

if (ret.size() > 0) {

int temp = ret.get(0).val;

ret.get(0).val = ret.get(1).val;

ret.get(1).val = temp;

}

}

private void helper(TreeNode root, List<TreeNode> list, List<TreeNode> ret) {

if (root == null) {

return;

}

helper(root.left, list, ret);

list.add(root);

if (list.size() != 1 && list.get(list.size() - 2).val >= root.val) {

if (ret.size() == 0) {

ret.add(list.get(list.size() - 2));

ret.add(root);

} else {

ret.set(1, root);

return;

}

}

helper(root.right, list, ret);

}

}

**Solution 2 – Time O (n) Space O (logn)**

// Use a class member to simply code

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

TreeNode prev = null;

public void recoverTree(TreeNode root) {

if (root == null) {

return;

}

List<TreeNode> ret = new ArrayList<TreeNode>();

helper(root, ret);

if (ret.size() > 0) {

int tmp = ret.get(0).val;

ret.get(0).val = ret.get(1).val;

ret.get(1).val = tmp;

}

}

public void helper(TreeNode root, List<TreeNode> ret) {

if (root == null) {

return;

}

helper(root.left, ret);

if (prev != null && prev.val > root.val) {

if (ret.size() == 0) {

ret.add(prev);

ret.add(root);

} else {

ret.set(1, root);

}

}

prev = root;

helper(root.right, ret);

}

}

### Same Tree

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.

**Solution 1 – Time O (n) Space O (log n)**

// Recursive

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isSameTree(TreeNode p, TreeNode q) {

if (p == null && q == null) {

return true;

}

if (p == null || q == null) {

return false;

}

return p.val == q.val && isSameTree(p.left, q.left)

&& isSameTree(p.right, q.right);

}

}

**Solution 2 – Time O (n) Space O (log n)**

// Iterative

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isSameTree(TreeNode p, TreeNode q) {

Stack<TreeNode> s = new Stack<TreeNode>();

s.push(p);

s.push(q);

while (!s.isEmpty()) {

p = s.pop();

q = s.pop();

if (p == null && q == null) {

continue;

}

if (p == null || q == null) {

return false;

}

if (p.val != q.val) {

return false;

}

s.push(p.left);

s.push(q.left);

s.push(p.right);

s.push(q.right);

}

return true;

}

}

### Symmetric Tree

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.

confused what "{1,#,2,3}" means? [> read more on how binary tree is serialized on OJ.](http://oj.leetcode.com/problems/symmetric-tree/)

**OJ's Binary Tree Serialization:**

The serialization of a binary tree follows a level order traversal, where '#' signifies a path terminator where no node exists below.

Here's an example:

1

/ \

2 3

/

4

\

5

The above binary tree is serialized as "{1,2,3,#,#,4,#,#,5}".

**Solution 1 – Time O (n) Space O (log n)**

// Recursive

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isSymmetric(TreeNode root) {

if (root == null) {

return true;

}

return helper(root.left, root.right);

}

private boolean helper(TreeNode left, TreeNode right) {

if (left == null && right == null) {

return true;

}

if (left == null || right == null) {

return false;

}

return left.val == right.val && helper(left.left, right.right)

&& helper(left.right, right.left);

}

}

**Solution 2 – Time O (n) Space O (log n)**

// Iterative

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isSymmetric(TreeNode root) {

if (root == null) {

return true;

}

Stack<TreeNode> s = new Stack<TreeNode>();

s.push(root.left);

s.push(root.right);

while (!s.isEmpty()) {

TreeNode p = s.pop();

TreeNode q = s.pop();

if (p == null && q == null) {

continue;

}

if (p == null || q == null) {

return false;

}

if (p.val != q.val) {

return false;

}

s.push(p.left);

s.push(q.right);

s.push(p.right);

s.push(q.left);

}

return true;

}

}

### Binary Tree Level Order Traversal

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]

]

**Solution 1 – Time O (n) Space O (n)**

// Recursive

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> ret = new LinkedList<List<Integer>>();

helper(ret, root, 0);

return ret;

}

private void helper(List<List<Integer>> ret, TreeNode root, int level) {

if (root == null) {

return;

}

if (level >= ret.size()) {

ret.add(new ArrayList<Integer>());

}

ret.get(level).add(root.val);

helper(ret, root.left, level + 1);

helper(ret, root.right, level + 1);

}

}

Solution 2 – Time O (n) Space O (n)

// Iterative

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> ret = new LinkedList<List<Integer>>();

if (root == null) {

return ret;

}

Queue<TreeNode> q = new LinkedList<TreeNode>();

q.offer(root);

int curNum = 0;

int preNum = 1;

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

while (!q.isEmpty()) {

TreeNode cur = q.poll();

preNum--;

list.add(cur.val);

if (cur.left != null) {

q.offer(cur.left);

curNum++;

}

if (cur.right != null) {

q.offer(cur.right);

curNum++;

}

if (preNum == 0) {

preNum = curNum;

curNum = 0;

ret.add(list);

list = new LinkedList<Integer>();

}

}

return ret;

}

}

### Binary Tree Zigzag Level Order Traversal

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]

]

confused what "{1,#,2,3}" means? [> read more on how binary tree is serialized on OJ.](https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/)

**Solution 1 – Time O (n) Space O (n)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

List<List<Integer>> ret = new LinkedList<List<Integer>>();

if (root == null) {

return ret;

}

Deque<TreeNode> pre = new LinkedList<TreeNode>();

pre.push(root);

int level = 1;

while (!pre.isEmpty()) {

Deque<TreeNode> cur = new LinkedList<TreeNode>();

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

while (!pre.isEmpty()) {

TreeNode node = pre.pop();

list.add(node.val);

if (level % 2 == 0) {

if (node.right != null) {

cur.push(node.right);

}

if (node.left != null) {

cur.push(node.left);

}

} else {

if (node.left != null) {

cur.push(node.left);

}

if (node.right != null) {

cur.push(node.right);

}

}

}

level++;

pre = cur;

ret.add(list);

}

return ret;

}

}

### Maximum Depth of Binary Tree

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.

**Solution 1 – Time O (n) Space (log n)**

// Recursive

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int maxDepth(TreeNode root) {

if (root == null) {

return 0;

}

return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;

}

}

**Solution 2 – Time O (n) Space O (log n)**

// Iterative

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int maxDepth(TreeNode root) {

if (root == null) {

return 0;

}

Queue<TreeNode> q = new LinkedList<TreeNode>();

q.offer(root);

int curNum = 0;

int preNum = 1;

int level = 0;

while (!q.isEmpty()) {

TreeNode cur = q.poll();

preNum--;

if (cur.left != null) {

q.offer(cur.left);

curNum++;

}

if (cur.right != null) {

q.offer(cur.right);

curNum++;

}

if (preNum == 0) {

level++;

preNum = curNum;

curNum = 0;

}

}

return level;

}

}

### Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

**Note:**  
You may assume that duplicates do not exist in the tree.

**Solution 1 – Time O (n) Space O (n)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode buildTree(int[] preorder, int[] inorder) {

if (preorder == null || inorder == null) {

return null;

}

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

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

map.put(inorder[i], i);

}

return helper(preorder, 0, preorder.length - 1, inorder, 0,

inorder.length - 1, map);

}

private TreeNode helper(int[] preorder, int preL, int preR, int[] inorder,

int inL, int inR, HashMap<Integer, Integer> map) {

if (preL > preR || inL > inR)

return null;

TreeNode root = new TreeNode(preorder[preL]);

int index = map.get(root.val);

root.left = helper(preorder, preL + 1, index - inL + preL, inorder,

inL, index - 1, map);

root.right = helper(preorder, preL + index - inL + 1, preR, inorder,

index + 1, inR, map);

return root;

}

}

### Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

**Note:**  
You may assume that duplicates do not exist in the tree.

**Solution 1- Time O (n) Space O (n)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode buildTree(int[] inorder, int[] postorder) {

if (postorder == null || inorder == null) {

return null;

}

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

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

map.put(inorder[i], i);

}

return helper(postorder, 0, postorder.length - 1, inorder, 0,

inorder.length - 1, map);

}

private TreeNode helper(int[] postorder, int postL, int postR,

int[] inorder, int inL, int inR, HashMap<Integer, Integer> map) {

if (postL > postR || inL > inR)

return null;

TreeNode root = new TreeNode(postorder[postR]);

int index = map.get(root.val);

root.left = helper(postorder, postL, index - inL + postL - 1, inorder,

inL, index - 1, map);

root.right = helper(postorder, postL + index - inL, postR - 1, inorder,

index + 1, inR, map);

return root;

}

}

### Binary Tree Level Order Traversal II

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],

]

**Solution 1 – Time O (n) Space O (n)**

// Recursive

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<List<Integer>> levelOrderBottom(TreeNode root) {

List<List<Integer>> ret = new LinkedList<List<Integer>>();

helper(ret, root, 0);

Collections.reverse(ret);

return ret;

}

private void helper(List<List<Integer>> ret, TreeNode root, int level) {

if (root == null) {

return;

}

if (level >= ret.size()) {

ret.add(new ArrayList<Integer>());

}

ret.get(level).add(root.val);

helper(ret, root.left, level + 1);

helper(ret, root.right, level + 1);

}

}

**Solution 2 – Time O (n) Space O (n)**

// Iterative

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<List<Integer>> levelOrderBottom(TreeNode root) {

List<List<Integer>> ret = new LinkedList<List<Integer>>();

if (root == null) {

return ret;

}

Queue<TreeNode> q = new LinkedList<TreeNode>();

q.offer(root);

int curNum = 0;

int preNum = 1;

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

while (!q.isEmpty()) {

TreeNode cur = q.poll();

preNum--;

list.add(cur.val);

if (cur.left != null) {

q.offer(cur.left);

curNum++;

}

if (cur.right != null) {

q.offer(cur.right);

curNum++;

}

if (preNum == 0) {

preNum = curNum;

curNum = 0;

ret.add(list);

list = new LinkedList<Integer>();

}

}

Collections.reverse(ret);

return ret;

}

}

### Convert Sorted Array to Binary Search Tree

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

**Solution 1 – Time O (n) Space O (n)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode sortedArrayToBST(int[] num) {

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

return null;

}

return helper(num, 0, num.length - 1);

}

private TreeNode helper(int[] num, int l, int r) {

if (l > r) {

return null;

}

int mid = (l + r) / 2;

TreeNode root = new TreeNode(num[mid]);

root.left = helper(num, l, mid - 1);

root.right = helper(num, mid + 1, r);

return root;

}

}

### Convert Sorted List to Binary Search Tree

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

**Solution 1 – Time O (n) Space O (log n)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; next = null; }

\* }

\*/

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode sortedListToBST(ListNode head) {

if (head == null) {

return null;

}

ListNode cur = head;

int count = 0;

while (cur != null) {

cur = cur.next;

count++;

}

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

list.add(head);

return helper(list, 0, count - 1);

}

private TreeNode helper(List<ListNode> list, int l, int r) {

if (l > r) {

return null;

}

int m = (l + r) / 2;

TreeNode left = helper(list, l, m - 1);

TreeNode root = new TreeNode(list.get(0).val);

root.left = left;

list.set(0, list.get(0).next);

root.right = helper(list, m + 1, r);

return root;

}

}

### Balanced Binary Tree

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.

**Solution 1 – Time O (n) Space O (log n)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isBalanced(TreeNode root) {

return helper(root) >= 0;

}

private int helper(TreeNode root) {

if (root == null) {

return 0;

}

int left = helper(root.left);

int right = helper(root.right);

if (left < 0 || right < 0) {

return -1;

}

if (Math.abs(left - right) > 1) {

return -1;

}

return Math.max(left, right) + 1;

}

}

**Solution 2 – Time O (n^2) Space O (n)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean isBalanced(TreeNode root) {

if (root == null)

return true;

return Math.abs(maxDepth(root.left) - maxDepth(root.right)) <= 1

&& isBalanced(root.left) && isBalanced(root.right);

}

public int maxDepth(TreeNode root) {

if (root == null)

return 0;

return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;

}

}

### Minimum Depth of Binary Tree

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.

**Solution 1 – Time O (log n), Space O (log n)**

// Recursive

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int minDepth(TreeNode root) {

if (root == null) {

return 0;

}

if (root.left == null) {

return minDepth(root.right) + 1;

}

if (root.right == null) {

return minDepth(root.left) + 1;

}

return Math.min(minDepth(root.left), minDepth(root.right)) + 1;

}

}

**Solution 2 – Time O (log n), Space O (log n)**

// Iterative

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int minDepth(TreeNode root) {

if (root == null) {

return 0;

}

Queue<TreeNode> q = new LinkedList<TreeNode>();

q.offer(root);

int curNum = 0;

int preNum = 1;

int level = 0;

while (!q.isEmpty()) {

TreeNode cur = q.poll();

preNum--;

if (cur.left == null && cur.right == null) {

return level + 1;

}

if (cur.left != null) {

q.offer(cur.left);

curNum++;

}

if (cur.right != null) {

q.offer(cur.right);

curNum++;

}

if (preNum == 0) {

level++;

preNum = curNum;

curNum = 0;

}

}

return level;

}

}

### Path Sum

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.

**Solution 1 – Time O (n) Space O (log n)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public boolean hasPathSum(TreeNode root, int sum) {

if (root == null) {

return false;

}

if (root.left == null && root.right == null && root.val == sum) {

return true;

}

return hasPathSum(root.left, sum - root.val)

|| hasPathSum(root.right, sum - root.val);

}

}

### Path Sum II

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]

]

**Solution 1 – Time O (n) Space O (k log n)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<List<Integer>> pathSum(TreeNode root, int sum) {

List<List<Integer>> ret = new LinkedList<List<Integer>>();

helper(ret, new LinkedList<Integer>(), root, sum);

return ret;

}

public void helper(List<List<Integer>> ret, List<Integer> cur, TreeNode root, int sum) {

if (root == null) {

return;

}

cur.add(root.val);

if (root.left == null && root.right == null && root.val == sum) {

ret.add(new LinkedList<Integer>(cur));

}

helper(ret, cur, root.left, sum - root.val);

helper(ret, cur, root.right, sum - root.val);

cur.remove(cur.size() - 1);

}

}

**Solution 2 – Time O (n) Space O (k log n)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<List<Integer>> pathSum(TreeNode root, int sum) {

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

if (root == null) {

return ret;

}

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

item.add(root.val);

helper(root, sum - root.val, item, ret);

return ret;

}

private void helper(TreeNode root, int sum, List<Integer> item,

List<List<Integer>> ret) {

if (root.left == null && root.right == null && sum == 0) {

ret.add(new ArrayList<Integer>(item));

return;

}

if (root.left != null) {

item.add(root.left.val);

helper(root.left, sum - root.left.val, item, ret);

item.remove(item.size() - 1);

}

if (root.right != null) {

item.add(root.right.val);

helper(root.right, sum - root.right.val, item, ret);

item.remove(item.size() - 1);

}

}

}

### Flatten Binary Tree to Linked List

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.

**Solution 1 – Time O (n) Space O (log n)**

// Recursive

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public void flatten(TreeNode root) {

List<TreeNode> ret = new ArrayList<TreeNode>();

ret.add(null);

helper(ret, root);

}

private void helper(List<TreeNode> ret, TreeNode root) {

if (root == null) {

return;

}

TreeNode right = root.right;

if (ret.get(0) != null) {

ret.get(0).left = null;

ret.get(0).right = root;

}

ret.set(0, root);

root.right = right;

helper(ret, root.left);

helper(ret, right);

}

}

**Solution 2 – Time O (n) Space O (log n)**

// Iterative

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public void flatten(TreeNode root) {

if (root == null) {

return;

}

Stack<TreeNode> s = new Stack<TreeNode>();

s.push(root);

while (!s.isEmpty()) {

TreeNode p = s.pop();

if (p.right != null) {

s.push(p.right);

}

if (p.left != null) {

s.push(p.left);

}

p.left = null;

if (!s.isEmpty()) {

p.right = s.peek();

}

}

}

}

### Distinct Subsequences

Given a string **S** and a string **T**, count the number of distinct subsequences of **T** in **S**.

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).

Here is an example:  
**S** = "rabbbit", **T** = "rabbit"

Return 3.

**Solution 1 – Time O (m \* n) Space O (m)**

public class Solution {

public int numDistinct(String s, String t) {

if (s == null || t == null) {

return 0;

}

if (t.length() == 0) {

return 1;

}

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

return 0;

}

int[] ret = new int[t.length() + 1];

ret[0] = 1;

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

for (int j = t.length() - 1; j >= 0; j--) {

if (s.charAt(i) == t.charAt(j)) {

ret[j + 1] = ret[j + 1] + ret[j];

} else {

ret[j + 1] = ret[j + 1];

}

}

}

return ret[t.length()];

}

}

### Populating Next Right Pointers in Each Node

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

**Solution 1 – Time O (n) Space O (log n)**

// Recursive

/\*\*

\* Definition for binary tree with next pointer.

\* public class TreeLinkNode {

\* int val;

\* TreeLinkNode left, right, next;

\* TreeLinkNode(int x) { val = x; }

\* }

\*/

public class Solution {

public void connect(TreeLinkNode root) {

connect(root, null);

}

public void connect(TreeLinkNode root, TreeLinkNode sibling) {

if (root == null) {

return;

} else {

root.next = sibling;

}

connect(root.left, root.right);

if (sibling == null) {

connect(root.right, null);

} else {

connect(root.right, sibling.left);

}

}

}

**Solution 2 – Time O (n) Space O (1)**

// Iterative

/\*\*

\* Definition for binary tree with next pointer.

\* public class TreeLinkNode {

\* int val;

\* TreeLinkNode left, right, next;

\* TreeLinkNode(int x) { val = x; }

\* }

\*/

public class Solution {

public void connect(TreeLinkNode root) {

if (root == null) {

return;

}

TreeLinkNode cur;

while (root.left != null) {

cur = root;

while (cur != null) {

cur.left.next = cur.right;

if (cur.next != null) {

cur.right.next = cur.next.left;

}

cur = cur.next;

}

root = root.left;

}

}

}

### Populating Next Right Pointers in Each Node II

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

**Solution 1 – Time O (n) Space O(1)**

// Level order

/\*\*

\* Definition for binary tree with next pointer.

\* public class TreeLinkNode {

\* int val;

\* TreeLinkNode left, right, next;

\* TreeLinkNode(int x) { val = x; }

\* }

\*/

public class Solution {

public void connect(TreeLinkNode root) {

if (root == null) {

return;

}

Queue<TreeLinkNode> q = new LinkedList<TreeLinkNode>();

q.offer(root);

int preNum = 1;

int curNum = 0;

while (!q.isEmpty()) {

TreeLinkNode node = q.poll();

preNum--;

if (node.left != null) {

q.offer(node.left);

curNum++;

}

if (node.right != null) {

q.offer(node.right);

curNum++;

}

if (preNum == 0) {

node.next = null;

preNum = curNum;

curNum = 0;

} else {

node.next = q.peek();

}

}

}

}

**Solution 2 – Time O (n) Space O(1)**

/\*\*

\* Definition for binary tree with next pointer.

\* public class TreeLinkNode {

\* int val;

\* TreeLinkNode left, right, next;

\* TreeLinkNode(int x) { val = x; }

\* }

\*/

public class Solution {

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;

}

}

}

### Pascal's Triangle

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]

]

**Solution 1 - Time O (n^2) Space O (1)**

public class Solution {

public List<List<Integer>> generate(int numRows) {

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

if (numRows < 1) {

return ret;

}

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

preList.add(1);

ret.add(preList);

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

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

curList.add(1);

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

curList.add(preList.get(j - 1) + preList.get(j));

}

curList.add(1);

ret.add(curList);

preList = curList;

}

return ret;

}

}

### Pascal's Triangle II

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?

**Solution 1 – Time O (n^2) Space O (1)**

public class Solution {

public List<Integer> getRow(int rowIndex) {

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

if (rowIndex < 0) {

return ret;

}

ret.add(1);

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

for (int j = i - 1; j > 0; j--) {

ret.set(j, ret.get(j) + ret.get(j - 1));

}

ret.add(1);

}

return ret;

}

}

### Triangle

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.

**Solution 1 – Time O (n^2) Space O (n)**

public class Solution {

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

if (triangle.size() <= 0) {

return 0;

}

int size = triangle.size();

int[] ret = new int[size];

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

ret[i] = triangle.get(size - 1).get(i);

}

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

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

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

}

}

return ret[0];

}

}

### Best Time to Buy and Sell Stock

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.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int maxProfit(int[] prices) {

if (prices == null || prices.length < 2) {

return 0;

}

int min = prices[0];

int profit = 0;

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

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

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

}

return profit;

}

}

### Best Time to Buy and Sell Stock II

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).

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int maxProfit(int[] prices) {

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

return 0;

}

int profit = 0;

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

if (prices[i] - prices[i - 1] > 0) {

profit += prices[i] - prices[i - 1];

}

}

return profit;

}

}

### Best Time to Buy and Sell Stock III

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 at most *two* transactions.

**Note:**  
You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

**Solution 1 – Time O (n\*k) Space O (k)**

public class Solution {

public int maxProfit(int[] prices) {

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

return 0;

}

int[] local = new int[3];

int[] global = new int[3];

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

int diff = prices[i + 1] - prices[i];

for (int j = 2; j >= 1; j--) {

local[j] = Math.max(global[j - 1] + (diff > 0 ? diff : 0),

local[j] + diff);

global[j] = Math.max(local[j], global[j]);

}

}

return global[2];

}

}

### Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

The path may start and end at any node in the tree.

For example:  
Given the below binary tree,

1

/ \

2 3

Return 6.

**Solution 1 – Time O (n) Space O (log n)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int maxPathSum(TreeNode root) {

if (root == null) {

return 0;

}

ArrayList<Integer> ret = new ArrayList<Integer>();

ret.add(Integer.MIN\_VALUE);

helper(root, ret);

return ret.get(0);

}

private int helper(TreeNode root, ArrayList<Integer> ret) {

if (root == null)

return 0;

int left = helper(root.left, ret);

int right = helper(root.right, ret);

int cur = root.val + (left > 0 ? left : 0) + (right > 0 ? right : 0);

if (cur > ret.get(0)) {

ret.set(0, cur);

}

return root.val + Math.max(left, Math.max(right, 0));

}

}

### Valid Palindrome

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.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public boolean isPalindrome(String s) {

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

return true;

}

int l = 0;

int r = s.length() - 1;

while (l < r) {

while (l < r && !Character.isLetterOrDigit(s.charAt(l))) {

l++;

}

while (l < r && !Character.isLetterOrDigit(s.charAt(r))) {

r--;

}

if (Character.toLowerCase(s.charAt(l)) != Character.toLowerCase(s

.charAt(r))) {

return false;

}

l++;

r--;

}

return true;

}

}

**Solution 2 – Time O (n) Space O (1)**

public class Solution {

public boolean isPalindrome(String s) {

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

return true;

}

int l = 0;

int r = s.length() - 1;

while (l < r) {

if (!isValid(s.charAt(l))) {

l++;

continue;

}

if (!isValid(s.charAt(r))) {

r--;

continue;

}

if (!isSame(s.charAt(l), s.charAt(r))) {

return false;

}

l++;

r--;

}

return true;

}

private boolean isValid(char c) {

if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z' || c >= '0'

&& c <= '9') {

return true;

} else {

return false;

}

}

private boolean isSame(char c1, char c2) {

if (c1 >= 'A' && c1 <= 'Z') {

c1 = (char) (c1 - 'A' + 'a');

}

if (c2 >= 'A' && c2 <= 'Z') {

c2 = (char) (c2 - 'A' + 'a');

}

return c1 == c2;

}

}

### Word Ladder II

Given two words (*start* and *end*), and a dictionary, find all shortest transformation sequence(s) from *start* to *end*, such that:

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the dictionary

For example,

Given:  
*start* = "hit"  
*end* = "cog"  
*dict* = ["hot","dot","dog","lot","log"]

Return

[

["hit","hot","dot","dog","cog"],

["hit","hot","lot","log","cog"]

]

**Solution 1 – Time O (?) Space O (?)**

public class Solution {

public List<List<String>> findLadders(String start, String end,

Set<String> dict) {

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

Set<String> unvisitedSet = new HashSet<String>();

unvisitedSet.addAll(dict);

unvisitedSet.add(start);

unvisitedSet.remove(end);

Map<String, List<String>> nextMap = new HashMap<String, List<String>>();

for (String e : unvisitedSet) {

nextMap.put(e, new ArrayList<String>());

}

Queue<StringWithLevel> queue = new LinkedList<StringWithLevel>();

queue.add(new StringWithLevel(end, 0));

boolean found = false;

int finalLevel = Integer.MAX\_VALUE;

int curLevel = 0;

int preLevel = 0;

Set<String> visitedCurLevel = new HashSet<String>();

while (!queue.isEmpty()) {

StringWithLevel cur = queue.poll();

String curStr = cur.str;

curLevel = cur.level;

if (found && curLevel > finalLevel) {

break;

}

if (curLevel > preLevel) {

unvisitedSet.removeAll(visitedCurLevel);

}

preLevel = curLevel;

char[] curStrCharArray = curStr.toCharArray();

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

char originalChar = curStrCharArray[i];

boolean foundCurCycle = false;

for (char c = 'a'; c <= 'z'; ++c) {

curStrCharArray[i] = c;

String newStr = new String(curStrCharArray);

if (c != originalChar && unvisitedSet.contains(newStr)) {

nextMap.get(newStr).add(curStr);

if (newStr.equals(start)) {

found = true;

finalLevel = curLevel;

foundCurCycle = true;

break;

}

if (visitedCurLevel.add(newStr)) {

queue.add(new StringWithLevel(newStr, curLevel + 1));

}

}

}

if (foundCurCycle) {

break;

}

curStrCharArray[i] = originalChar;

}

}

if (found) {

ArrayList<String> list = new ArrayList<String>();

list.add(start);

getPaths(start, end, list, finalLevel + 1, nextMap, ret);

}

return ret;

}

private void getPaths(String cur, String end, List<String> list, int level,

Map<String, List<String>> nextMap, List<List<String>> ret) {

if (cur.equals(end)) {

ret.add(new ArrayList<String>(list));

} else if (level > 0) {

List<String> parentsSet = nextMap.get(cur);

for (String parent : parentsSet) {

list.add(parent);

getPaths(parent, end, list, level - 1, nextMap, ret);

list.remove(list.size() - 1);

}

}

}

}

class StringWithLevel {

String str;

int level;

public StringWithLevel(String str, int level) {

this.str = str;

this.level = level;

}

}

### Word Ladder

Given two words (*beginWord* and *endWord*), and a dictionary, find the length of shortest transformation sequence from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the dictionary

For example,

Given:  
*start* = "hit"  
*end* = "cog"  
*dict* = ["hot","dot","dog","lot","log"]

As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",  
return its length 5.

**Note:**

* Return 0 if there is no such transformation sequence.
* All words have the same length.
* All words contain only lowercase alphabetic characters.

**Solution 1 – Time O (min (26^L, size (dict)) Space O (min (26^L, size (dict))**

public class Solution {

public int ladderLength(String beginWord, String endWord,

Set<String> wordDict) {

if (beginWord == null || endWord == null || beginWord.length() == 0

|| endWord.length() == 0

|| beginWord.length() != endWord.length()) {

return 0;

}

Queue<String> queue = new LinkedList<String>();

Set<String> visited = new HashSet<String>();

int level = 1;

int lastNum = 1;

int curNum = 0;

queue.offer(beginWord);

visited.add(beginWord);

while (!queue.isEmpty()) {

String cur = queue.poll();

lastNum--;

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

char[] charCur = cur.toCharArray();

for (char c = 'a'; c <= 'z'; c++) {

charCur[i] = c;

String temp = new String(charCur);

if (temp.equals(endWord)) {

return level + 1;

}

if (wordDict.contains(temp) && !visited.contains(temp)) {

curNum++;

queue.offer(temp);

visited.add(temp);

}

}

}

if (lastNum == 0) {

lastNum = curNum;

curNum = 0;

level++;

}

}

return 0;

}

}

### Longest Consecutive Sequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example,  
Given [100, 4, 200, 1, 3, 2],  
The longest consecutive elements sequence is [1, 2, 3, 4]. Return its length: 4.

Your algorithm should run in O(*n*) complexity.

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public int longestConsecutive(int[] nums) {

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

return 0;

}

HashSet<Integer> set = new HashSet<Integer>();

int ret = 1;

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

set.add(nums[i]);

}

while (!set.isEmpty()) {

Iterator<Integer> iter = set.iterator();

int item = iter.next();

set.remove(item);

int len = 1;

int i = item - 1;

while (set.contains(i)) {

set.remove(i--);

len++;

}

i = item + 1;

while (set.contains(i)) {

set.remove(i++);

len++;

}

if (len > ret) {

ret = len;

}

}

return ret;

}

}

### Sum Root to Leaf Numbers

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.

**Solution 1 – Time O (n) Space O (log n)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int sumNumbers(TreeNode root) {

return helper(root, 0);

}

int helper(TreeNode root, int sum) {

if (root == null) {

return 0;

}

if (root.left == null && root.right == null) {

return sum \* 10 + root.val;

}

return helper(root.left, sum \* 10 + root.val)

+ helper(root.right, sum \* 10 + root.val);

}

}

### Surrounded Regions

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

**Solution 1 – Time O (m\*n) Space O (m+n)**

public class Solution {

public void solve(char[][] board) {

if (board == null || board.length <= 1 || board[0].length <= 1) {

return;

}

for (int i = 0; i < board[0].length; i++) {

fill(board, 0, i);

fill(board, board.length - 1, i);

}

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

fill(board, i, 0);

fill(board, i, board[0].length - 1);

}

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

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

if (board[i][j] == 'O') {

board[i][j] = 'X';

} else if (board[i][j] == '#') {

board[i][j] = 'O';

}

}

}

}

void fill(char[][] board, int i, int j) {

if (board[i][j] != 'O') {

return;

}

board[i][j] = '#';

Queue<Integer> queue = new LinkedList<Integer>();

int code = i \* board[0].length + j;

queue.offer(code);

while (!queue.isEmpty()) {

code = queue.poll();

int row = code / board[0].length;

int col = code % board[0].length;

if (row > 0 && board[row - 1][col] == 'O') {

queue.offer((row - 1) \* board[0].length + col);

board[row - 1][col] = '#';

}

if (row < board.length - 1 && board[row + 1][col] == 'O') {

queue.offer((row + 1) \* board[0].length + col);

board[row + 1][col] = '#';

}

if (col > 0 && board[row][col - 1] == 'O') {

queue.offer(row \* board[0].length + col - 1);

board[row][col - 1] = '#';

}

if (col < board[0].length - 1 && board[row][col + 1] == 'O') {

queue.offer(row \* board[0].length + col + 1);

board[row][col + 1] = '#';

}

}

}

}

### Palindrome Partitioning

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"]

]

**Solution 1 – Time O (2^n) Space O (n^2)**

public class Solution {

public List<List<String>> partition(String s) {

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

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

return ret;

helper(s, getDict(s), 0, new ArrayList<String>(), ret);

return ret;

}

private void helper(String s, boolean[][] dict, int start,

List<String> item, List<List<String>> ret) {

if (start == s.length()) {

ret.add(new ArrayList<String>(item));

return;

}

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

if (dict[start][i]) {

item.add(s.substring(start, i + 1));

helper(s, dict, i + 1, item, ret);

item.remove(item.size() - 1);

}

}

}

private boolean[][] getDict(String s) {

boolean[][] dict = new boolean[s.length()][s.length()];

for (int i = s.length() - 1; i >= 0; i--) {

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

if (s.charAt(i) == s.charAt(j)

&& ((j - i < 2) || dict[i + 1][j - 1])) {

dict[i][j] = true;

}

}

}

return dict;

}

}

### Palindrome Partitioning II

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of *s*.

For example, given *s* = "aab",  
Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.

**Solution 1 – Time O (n^2) Space O (n^2)**

public class Solution {

public int minCut(String s) {

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

return 0;

}

boolean[][] dict = getDict(s);

int[] res = new int[s.length() + 1];

res[0] = 0;

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

res[i + 1] = i + 1;

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

if (dict[j][i]) {

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

}

}

}

return res[s.length()] - 1;

}

private boolean[][] getDict(String s) {

boolean[][] dict = new boolean[s.length()][s.length()];

for (int i = s.length() - 1; i >= 0; i--) {

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

if (s.charAt(i) == s.charAt(j)

&& ((j - i < 2) || dict[i + 1][j - 1])) {

dict[i][j] = true;

}

}

}

return dict;

}

}

### Clone Graph

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

/ \

\\_/

**Solution 1 – Time O (n) Space O (n)**

// BFS

/\*\*

\* Definition for undirected graph.

\* class UndirectedGraphNode {

\* int label;

\* List<UndirectedGraphNode> neighbors;

\* UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }

\* };

\*/

public class Solution {

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

if (node == null) {

return null;

}

Queue<UndirectedGraphNode> queue = new LinkedList<UndirectedGraphNode>();

Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<UndirectedGraphNode, UndirectedGraphNode>();

UndirectedGraphNode copy = new UndirectedGraphNode(node.label);

map.put(node, copy);

queue.offer(node);

while (!queue.isEmpty()) {

UndirectedGraphNode cur = queue.poll();

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

if (!map.containsKey(cur.neighbors.get(i))) {

copy = new UndirectedGraphNode(cur.neighbors.get(i).label);

map.put(cur.neighbors.get(i), copy);

queue.offer(cur.neighbors.get(i));

}

map.get(cur).neighbors.add(map.get(cur.neighbors.get(i)));

}

}

return map.get(node);

}

}

**Solution 2 – Time O (n) Space O (n)**

// DFS iterative

/\*\*

\* Definition for undirected graph.

\* class UndirectedGraphNode {

\* int label;

\* List<UndirectedGraphNode> neighbors;

\* UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }

\* };

\*/

public class Solution {

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

if (node == null) {

return null;

}

Stack<UndirectedGraphNode> stack = new Stack<UndirectedGraphNode>();

Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<UndirectedGraphNode, UndirectedGraphNode>();

stack.push(node);

UndirectedGraphNode copy = new UndirectedGraphNode(node.label);

map.put(node, copy);

while (!stack.isEmpty()) {

UndirectedGraphNode cur = stack.pop();

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

if (!map.containsKey(cur.neighbors.get(i))) {

copy = new UndirectedGraphNode(cur.neighbors.get(i).label);

map.put(cur.neighbors.get(i), copy);

stack.push(cur.neighbors.get(i));

}

map.get(cur).neighbors.add(map.get(cur.neighbors.get(i)));

}

}

return map.get(node);

}

}

**Solution 3 – Time O (n) Space O (n)**

// DFS recursive 1

/\*\*

\* Definition for undirected graph.

\* class UndirectedGraphNode {

\* int label;

\* List<UndirectedGraphNode> neighbors;

\* UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }

\* };

\*/

public class Solution {

public UndirectedGraphNode cloneGraph(UndirectedGraphNode graph) {

if (graph == null)

return null;

Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<>();

return DFS(graph, map);

}

private UndirectedGraphNode DFS(UndirectedGraphNode graph,

Map<UndirectedGraphNode, UndirectedGraphNode> map) {

if (map.containsKey(graph)) {

return map.get(graph);

}

UndirectedGraphNode graphCopy = new UndirectedGraphNode(graph.label);

map.put(graph, graphCopy);

for (UndirectedGraphNode neighbor : graph.neighbors) {

graphCopy.neighbors.add(DFS(neighbor, map));

}

return graphCopy;

}

}

**Solution 4 – Time O (n) Space O (n)**

// DFS recursive 2

/\*\*

\* Definition for undirected graph.

\* class UndirectedGraphNode {

\* int label;

\* List<UndirectedGraphNode> neighbors;

\* UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }

\* };

\*/

public class Solution {

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

if (node == null) {

return null;

}

HashMap<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<UndirectedGraphNode, UndirectedGraphNode>();

UndirectedGraphNode copy = new UndirectedGraphNode(node.label);

map.put(node, copy);

helper(node, map);

return map.get(node);

}

private void helper(UndirectedGraphNode node,

HashMap<UndirectedGraphNode, UndirectedGraphNode> map) {

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

UndirectedGraphNode cur = node.neighbors.get(i);

if (!map.containsKey(cur)) {

UndirectedGraphNode copy = new UndirectedGraphNode(cur.label);

map.put(cur, copy);

helper(cur, map);

}

map.get(node).neighbors.add(map.get(cur));

}

}

}

### Gas Station

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.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int canCompleteCircuit(int[] gas, int[] cost) {

if (gas == null || cost == null || gas.length == 0 || cost.length == 0

|| gas.length != cost.length) {

return -1;

}

int sum = 0;

int total = 0;

int start = -1;

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

int diff = gas[i] - cost[i];

sum += diff;

total += diff;

if (sum < 0) {

sum = 0;

start = i;

}

}

return total >= 0 ? start + 1 : -1;

}

}

### Candy

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?

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public int candy(int[] ratings) {

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

return 0;

}

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

nums[0] = 1;

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

if (ratings[i] > ratings[i - 1]) {

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

} else {

nums[i] = 1;

}

}

int ret = nums[ratings.length - 1];

for (int i = ratings.length - 2; i >= 0; i--) {

int cur = 1;

if (ratings[i] > ratings[i + 1]) {

cur = nums[i + 1] + 1;

}

ret += Math.max(cur, nums[i]);

nums[i] = Math.max(cur, nums[i]);

}

return ret;

}

}

### Single Number

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?

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int singleNumber(int[] nums) {

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

return 0;

}

int ret = 0;

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

ret ^= nums[i];

}

return ret;

}

}

### Single Number II

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

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

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int singleNumber(int[] nums) {

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

return 0;

}

int ret = 0;

int[] digits = new int[32];

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

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

digits[i] += (nums[j] >> i) & 1;

}

ret += (digits[i] % 3) << i;

}

return ret;

}

}

**Solution 2 – Time O (n) Space O (1)**

public class Solution {

public int singleNumber(int[] nums) {

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

return 0;

}

int one = 0, two = 0, three = 0;

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

two |= one & nums[i];

one ^= nums[i];

three = two & one;

one &= ~three;

two &= ~three;

}

return one;

}

}

### Copy List with Random Pointer

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a deep copy of the list.

**Solution 1 – Time O (n) Space O (n)**

/\*\*

\* Definition for singly-linked list with a random pointer.

\* class RandomListNode {

\* int label;

\* RandomListNode next, random;

\* RandomListNode(int x) { this.label = x; }

\* };

\*/

public class Solution {

public RandomListNode copyRandomList(RandomListNode head) {

if (head == null) {

return head;

}

Map<RandomListNode, RandomListNode> map = new HashMap<RandomListNode, RandomListNode>();

RandomListNode newHead = new RandomListNode(head.label);

map.put(head, newHead);

RandomListNode pre = newHead;

RandomListNode node = head.next;

while (node != null) {

RandomListNode newNode = new RandomListNode(node.label);

map.put(node, newNode);

pre.next = newNode;

pre = newNode;

node = node.next;

}

node = head;

RandomListNode copyNode = newHead;

while (node != null) {

copyNode.random = map.get(node.random);

copyNode = copyNode.next;

node = node.next;

}

return newHead;

}

}

**Solution 2 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list with a random pointer.

\* class RandomListNode {

\* int label;

\* RandomListNode next, random;

\* RandomListNode(int x) { this.label = x; }

\* };

\*/

public class Solution {

public RandomListNode copyRandomList(RandomListNode head) {

if (head == null) {

return head;

}

RandomListNode node = head;

while (node != null) {

RandomListNode newNode = new RandomListNode(node.label);

newNode.next = node.next;

node.next = newNode;

node = newNode.next;

}

node = head;

while (node != null) {

if (node.random != null) {

node.next.random = node.random.next;

}

node = node.next.next;

}

RandomListNode newHead = head.next;

node = head;

while (node != null) {

RandomListNode newNode = node.next;

node.next = newNode.next;

if (newNode.next != null)

newNode.next = newNode.next.next;

node = node.next;

}

return newHead;

}

}

### Word Break

Given a string *s* and a dictionary of words *dict*, determine if *s* can be segmented into a space-separated sequence of one or more dictionary words.

For example, given  
*s* = "leetcode",  
*dict* = ["leet", "code"].

Return true because "leetcode" can be segmented as "leet code".

**Solution 1 – Time O (n^2) Space O (n)**

public class Solution {

public boolean wordBreak(String s, Set<String> wordDict) {

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

return true;

}

boolean[] ret = new boolean[s.length() + 1];

ret[0] = true;

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

StringBuilder str = new StringBuilder(s.substring(0, i + 1));

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

if (ret[j] && wordDict.contains(str.toString())) {

ret[i + 1] = true;

break;

}

str.deleteCharAt(0);

}

}

return ret[s.length()];

}

}

### Word Break II

Given a string *s* and a dictionary of words *dict*, add spaces in *s* to construct a sentence where each word is a valid dictionary word.

Return all such possible sentences.

For example, given  
*s* = "catsanddog",  
*dict* = ["cat", "cats", "and", "sand", "dog"].

A solution is ["cats and dog", "cat sand dog"].

**Solution 1 – Time O (n^2) Space O (n^2)**

public class Solution {

public List<String> wordBreak(String s, Set<String> wordDict) {

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

for (int j = s.length() - 1; j >= 0; j--) {

if (wordDict.contains(s.substring(j)))

break;

else {

if (j == 0)

return ret;

}

}

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

if (wordDict.contains(s.substring(0, i + 1))) {

List<String> strs = wordBreak(s.substring(i + 1, s.length()),

wordDict);

if (strs.size() != 0)

for (Iterator<String> it = strs.iterator(); it.hasNext();) {

ret.add(s.substring(0, i + 1) + " " + it.next());

}

}

}

if (wordDict.contains(s)) {

ret.add(s);

}

return ret;

}

}

### Linked List Cycle

Given a linked list, determine if it has a cycle in it.

Follow up:  
Can you solve it without using extra space?

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public boolean hasCycle(ListNode head) {

if (head == null) {

return false;

}

ListNode slow = head;

ListNode fast = head;

while (fast.next != null && fast.next.next != null) {

fast = fast.next.next;

slow = slow.next;

if (fast == slow) {

return true;

}

}

return false;

}

}

### Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

Follow up:  
Can you solve it without using extra space?

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode detectCycle(ListNode head) {

if (head == null) {

return null;

}

ListNode slow = head;

ListNode fast = head;

while (fast.next != null && fast.next.next != null) {

fast = fast.next.next;

slow = slow.next;

if (fast == slow) {

break;

}

}

if (fast.next == null || fast.next.next == null) {

return null;

}

slow = head;

while (fast != slow) {

fast = fast.next;

slow = slow.next;

}

return fast;

}

}

### Reorder List

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}.

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public void reorderList(ListNode head) {

if (head == null || head.next == null) {

return;

}

ListNode walker = head;

ListNode runner = head;

while (runner.next != null && runner.next.next != null) {

walker = walker.next;

runner = runner.next.next;

}

ListNode head1 = head;

ListNode head2 = walker.next;

walker.next = null;

head2 = reverse(head2);

while (head1 != null && head2 != null) {

ListNode next = head2.next;

head2.next = head1.next;

head1.next = head2;

head1 = head2.next;

head2 = next;

}

}

private ListNode reverse(ListNode head) {

ListNode pre = null;

ListNode cur = head;

while (cur != null) {

ListNode next = cur.next;

cur.next = pre;

pre = cur;

cur = next;

}

return pre;

}

public ListNode recursive\_reverse(ListNode head) {

if (head == null || head.next == null)

return head;

return recursive\_reverse(head, head.next);

}

private ListNode recursive\_reverse(ListNode current, ListNode next) {

if (next == null)

return current;

ListNode newHead = recursive\_reverse(current.next, next.next);

next.next = current;

current.next = null;

return newHead;

}

}

### Binary Tree Preorder Traversal

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?

**Solution 1 – Time O (n) Space O (log n)**

// Recursive

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> preorderTraversal(TreeNode root) {

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

preorderTraversalRecursive(root, ret);

return ret;

}

private void preorderTraversalRecursive(TreeNode root, List<Integer> ret) {

if (root == null) {

return;

}

ret.add(root.val);

preorderTraversalRecursive(root.left, ret);

preorderTraversalRecursive(root.right, ret);

}

}

**Solution 2 – Time O (n) Space O (log n)**

// Iterative 1 – easiest

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> preorderTraversal(TreeNode root) {

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

Stack<TreeNode> s = new Stack<TreeNode>();

if (root == null) {

return ret;

}

s.push(root);

while (!s.isEmpty()) {

TreeNode node = s.pop();

ret.add(node.val);

if (node.right != null) {

s.push(node.right);

}

if (node.left != null) {

s.push(node.left);

}

}

return ret;

}

}

**Solution 3 – Time O (n) Space O (log n)**

// Iterative 2 – similar to inorder

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> preorderTraversal(TreeNode root) {

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

Stack<TreeNode> s = new Stack<TreeNode>();

while (root != null || !s.isEmpty()) {

if (root != null) {

s.push(root);

ret.add(root.val);

root = root.left;

} else {

root = s.pop();

root = root.right;

}

}

return ret;

}

}

**Solution 4 – Time O (n) Space O (log n)**

// Iterative 3 – from Wiki

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> preorderTraversal(TreeNode root) {

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

Stack<TreeNode> s = new Stack<TreeNode>();

while (root != null || !s.isEmpty()) {

if (root != null) {

ret.add(root.val);

if (root.right != null) {

s.push(root.right);

}

root = root.left;

} else {

root = s.pop();

}

}

return ret;

}

}

### Binary Tree Postorder Traversal

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?

**Solution 1 – Time O (n) Space O (log n)**

// Recursive

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> postorderTraversal(TreeNode root) {

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

postorderTraversalRecursive(root, ret);

return ret;

}

private void postorderTraversalRecursive(TreeNode root, List<Integer> ret) {

if (root == null) {

return;

}

postorderTraversalRecursive(root.left, ret);

postorderTraversalRecursive(root.right, ret);

ret.add(root.val);

}

}

**Solution 2 – Time O (n) Space O (log n)**

// Iterative 1 – easiest

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> postorderTraversal(TreeNode root) {

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

Stack<TreeNode> s = new Stack<TreeNode>();

if (root == null) {

return ret;

}

s.push(root);

s.push(root);

while (!s.isEmpty()) {

TreeNode node = s.pop();

if (!s.isEmpty() && node == s.peek()) {

if (node.right != null) {

s.push(node.right);

s.push(node.right);

}

if (node.left != null) {

s.push(node.left);

s.push(node.left);

}

} else {

ret.add(node.val);

}

}

return ret;

}

}

**Solution 3 – Time O (n) Space O (log n)**

// Iterative 2 – From wiki

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> postorderTraversal(TreeNode root) {

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

Stack<TreeNode> s = new Stack<TreeNode>();

TreeNode pre = null;

while (root != null || !s.isEmpty()) {

if (root != null) {

s.push(root);

root = root.left;

} else {

TreeNode peekNode = s.peek();

if (peekNode.right != null && pre != peekNode.right) {

root = peekNode.right;

} else {

s.pop();

ret.add(peekNode.val);

pre = peekNode;

}

}

}

return ret;

}

}

### LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.  
set(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

**Solution 1 – Time O (1) Space O (1)**

public class LRUCache {

class Node {

Node pre, next;

int key;

int val;

public Node(int key, int value) {

this.key = key;

this.val = value;

}

}

private int capacity;

private int num;

private HashMap<Integer, Node> map;

private Node first, last;

public LRUCache(int capacity) {

this.capacity = capacity;

num = 0;

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

first = null;

last = null;

}

public int get(int key) {

Node node = map.get(key);

if (node == null)

return -1;

else if (node != last) {

if (node == first) {

first = first.next;

} else {

node.pre.next = node.next;

}

node.next.pre = node.pre;

last.next = node;

node.pre = last;

node.next = null;

last = node;

}

return node.val;

}

public void set(int key, int value) {

Node node = map.get(key);

if (node != null) {

node.val = value;

if (node != last) {

if (node == first) {

first = first.next;

} else {

node.pre.next = node.next;

}

node.next.pre = node.pre;

last.next = node;

node.pre = last;

node.next = null;

last = node;

}

} else {

Node newNode = new Node(key, value);

if (num >= capacity) {

map.remove(first.key);

first = first.next;

if (first != null)

first.pre = null;

else

last = null;

num--;

}

if (first == null || last == null) {

first = newNode;

} else {

last.next = newNode;

}

newNode.pre = last;

last = newNode;

map.put(key, newNode);

num++;

}

}

}

### Insertion Sort List

Sort a linked list using insertion sort.

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode insertionSortList(ListNode head) {

if (head == null)

return null;

ListNode helper = new ListNode(0);

ListNode pre = helper;

ListNode cur = head;

while (cur != null) {

ListNode next = cur.next;

pre = helper;

while (pre.next != null && pre.next.val <= cur.val) {

pre = pre.next;

}

cur.next = pre.next;

pre.next = cur;

cur = next;

}

return helper.next;

}

}

### Sort List

Sort a linked list in *O*(*n* log *n*) time using constant space complexity.

**Solution 1 – Time O (n log n) Space O (n log n)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode sortList(ListNode head) {

return mergeSort(head);

}

private ListNode mergeSort(ListNode head) {

if (head == null || head.next == null) {

return head;

}

ListNode slow = head;

ListNode fast = head;

while (fast.next != null && fast.next.next != null) {

slow = slow.next;

fast = fast.next.next;

}

ListNode head2 = slow.next;

slow.next = null;

ListNode head1 = head;

head1 = mergeSort(head1);

head2 = mergeSort(head2);

return merge(head1, head2);

}

private ListNode merge(ListNode l1, ListNode l2) {

if (l1 == null) {

return l2;

}

if (l2 == null) {

return l1;

}

ListNode head = new ListNode(0);

ListNode cur = head;

while (l1 != null && l2 != null) {

if (l1.val < l2.val) {

cur.next = l1;

l1 = l1.next;

} else {

cur.next = l2;

l2 = l2.next;

}

cur = cur.next;

}

if (l1 != null) {

cur.next = l1;

} else {

cur.next = l2;

}

return head.next;

}

}

### Max Points on a Line

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

**Solution 1 – Time O (n^2) Space O (n)**

/\*\*

\* Definition for a point.

\* class Point {

\* int x;

\* int y;

\* Point() { x = 0; y = 0; }

\* Point(int a, int b) { x = a; y = b; }

\* }

\*/

public class Solution {

public int maxPoints(Point[] points) {

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

return 0;

}

int max = 1;

double ratio = 0.0;

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

Map<Double, Integer> map = new HashMap<Double, Integer>();

int numofSame = 0;

int localMax = 1;

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

if (points[j].x == points[i].x && points[j].y == points[i].y) {

numofSame++;

continue;

} else if (points[j].x == points[i].x) {

ratio = (double) Integer.MAX\_VALUE;

// this is very import, ratio may be 0.0 or -0.0

} else if (points[j].y == points[i].y) {

ratio = 0.0;

} else {

ratio = (double) (points[j].y - points[i].y)

/ (double) (points[j].x - points[i].x);

}

if (map.containsKey(ratio)) {

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

} else {

map.put(ratio, 2);

}

}

for (Integer value : map.values()) {

localMax = Math.max(localMax, value);

}

localMax += numofSame;

max = Math.max(max, localMax);

}

return max;

}

}

### Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in [Reverse Polish Notation](http://en.wikipedia.org/wiki/Reverse_Polish_notation).

Valid operators are +, -, \*, /. Each operand may be an integer or another expression.

Some examples:

["2", "1", "+", "3", "\*"] -> ((2 + 1) \* 3) -> 9

["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int evalRPN(String[] tokens) {

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

return 0;

}

Stack<Integer> stack = new Stack<Integer>();

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

if (tokens[i].equals("+")) {

stack.push(stack.pop() + stack.pop());

} else if (tokens[i].equals("-")) {

stack.push(-stack.pop() + stack.pop());

} else if (tokens[i].equals("\*")) {

stack.push(stack.pop() \* stack.pop());

} else if (tokens[i].equals("/")) {

int num1 = stack.pop();

int num2 = stack.pop();

stack.push(num2 / num1);

} else {

stack.push(Integer.parseInt(tokens[i]));

}

}

return stack.pop();

}

}

### Reverse Words in a String

Given an input string, reverse the string word by word.

For example,  
Given s = "the sky is blue",  
return "blue is sky the".

**Update (2015-02-12):**  
For C programmers: Try to solve it *in-place* in *O*(1) space.

[click to show clarification.](https://leetcode.com/problems/reverse-words-in-a-string/)

**Clarification:**

* What constitutes a word?  
  A sequence of non-space characters constitutes a word.
* Could the input string contain leading or trailing spaces?  
  Yes. However, your reversed string should not contain leading or trailing spaces.
* How about multiple spaces between two words?  
  Reduce them to a single space in the reversed string.

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public String reverseWords(String s) {

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

return s;

}

char[] arr = s.trim().toCharArray();

int len = 0;

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

if (i < arr.length - 1 && arr[i] == arr[i + 1] && arr[i] == ' ') {

continue;

} else {

arr[len++] = arr[i];

}

}

reverse(arr, 0, len - 1);

int l = 0;

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

if (arr[i] == ' ') {

reverse(arr, l, i - 1);

l = i + 1;

}

}

reverse(arr, l, len - 1);

return new String(arr, 0, len);

}

public void reverse(char[] arr, int l, int r) {

while (l < r) {

char tmp = arr[l];

arr[l++] = arr[r];

arr[r--] = tmp;

}

}

}

**Solution 2 – Time O (n) Space O (n)**

public class Solution {

public String reverseWords(String s) {

StringBuilder reversed = new StringBuilder();

int j = s.length();

for (int i = s.length() - 1; i >= 0; i--) {

if (s.charAt(i) == ' ') {

j = i;

} else if (i == 0 || s.charAt(i - 1) == ' ') {

if (reversed.length() != 0) {

reversed.append(' ');

}

reversed.append(s.substring(i, j));

}

}

return reversed.toString();

}

}

**Solution 3 – Time O (n) Space O (n)**

public class Solution {

public String reverseWords(String s) {

if (s == null) {

return null;

}

s = s.trim();

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

return "";

}

StringBuilder ret = new StringBuilder();

for (int i = s.length() - 1; i >= 0; i--) {

if (i != s.length() - 1 && s.charAt(i) == ' '

&& s.charAt(i) == s.charAt(i + 1)) {

continue;

}

ret.append(s.charAt(i));

}

int left = 0;

int right = 0;

while (right < ret.length()) {

while (right < ret.length() && ret.charAt(right) != ' ') {

right++;

}

int next = right + 1;

right = right - 1;

while (left < right) {

char temp = ret.charAt(left);

ret.setCharAt(left++, ret.charAt(right));

ret.setCharAt(right--, temp);

}

left = next;

right = next;

}

return ret.toString();

}

}

### Maximum Product Subarray

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.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int maxProduct(int[] nums) {

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

return 0;

}

if (nums.length == 1) {

return nums[0];

}

int maxLocal = nums[0];

int minLocal = nums[0];

int global = nums[0];

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

int maxCopy = maxLocal;

maxLocal = Math.max(Math.max(nums[i] \* maxLocal, nums[i]), nums[i]

\* minLocal);

minLocal = Math.min(Math.min(maxCopy \* nums[i], nums[i]), nums[i]

\* minLocal);

global = Math.max(global, maxLocal);

}

return global;

}

}

### Find Minimum in Rotated Sorted Array

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

You may assume no duplicate exists in the array.

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public int findMin(int[] A) {

int L = 0, R = A.length - 1;

while (L < R && A[L] >= A[R]) {

int M = (L + R) / 2;

if (A[M] > A[R]) {

L = M + 1;

} else {

R = M;

}

}

return A[L];

}

}

**Solution 2 – Time O (log n) Space O (1)**

public class Solution {

public int findMin(int[] nums) {

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

return 0;

}

int start = 0;

int end = nums.length - 1;

if (nums[start] < nums[end]) {

return nums[0];

}

while (end - start > 1) {

int mid = (end + start) / 2;

if (nums[start] < nums[mid]) {

start = mid;

} else {

end = mid;

}

}

return nums[end];

}

}

### Find Minimum in Rotated Sorted Array II

*Follow up* for "Find Minimum in Rotated Sorted Array":  
What if *duplicates* are allowed?

Would this affect the run-time complexity? How and why?

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.

The array may contain duplicates.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int findMin(int[] nums) {

int L = 0, R = nums.length - 1;

while (L < R && nums[L] >= nums[R]) {

int M = (L + R) / 2;

if (nums[M] > nums[R]) {

L = M + 1;

} else if (nums[M] < nums[L]) {

R = M;

} else { // A[L] == A[M] == A[R]

L = L + 1;

}

}

return nums[L];

}

}

**Solution 2 – Time O (n) Space O (1)**

public class Solution {

public int findMin(int[] nums) {

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

return 0;

}

int l = 0;

int r = nums.length - 1;

while (l < r) {

int mid = (l + r) / 2;

if (nums[mid] == nums[r]) {

r--;

} else if (nums[mid] > nums[r]) {

l = mid + 1;

} else {

r = mid;

}

}

return nums[l];

}

}

### Min Stack

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.

**Solution 1 – Time O (1) Space O (n)**

class MinStack {

Stack<Integer> s = new Stack<Integer>();

Stack<Integer> min = new Stack<Integer>();

public void push(int x) {

if (min.isEmpty() || x <= min.peek()) {

min.push(x);

}

s.push(x);

}

public void pop() {

if (!s.isEmpty()) {

int val = s.pop();

if (val == min.peek()) {

min.pop();

}

}

}

public int top() {

if (!s.isEmpty()) {

return s.peek();

} else {

return 0;

}

}

public int getMin() {

if (!min.isEmpty()) {

return min.peek();

} else {

return 0;

}

}

}

### Binary Tree Upside Down

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left

node that shares the same parent node) or empty, flip it upside down and turn it into a tree

where the original right nodes turned into left leaf nodes. Return the new root.

For example:  
Given a binary tree {1,2,3,4,5},  
1  
/ \  
2 3  
/ \  
4 5

return the root of the binary tree [4,5,2,#,#,3,1].  
4  
/ \  
5 2  
  / \  
 3 1

**Solution 1 – Time O (Log n) Space O (1)**

public class Solution {

public TreeNode UpsideDownBinaryTree(TreeNode root) {

TreeNode p = root, parent = null, parentRight = null;

while (p != null) {

TreeNode left = p.left;

p.left = parentRight;

parentRight = p.right;

p.right = parent;

parent = p;

p = left;

}

return parent;

}

}

**Solution 2 – Time O (Log n) Space O (log n)**

public class Solution {

public TreeNode UpsideDownBinaryTree(TreeNode root) {

if (root == null || root.left == null) {

return root;

} else {

TreeNode left = root.left;

TreeNode right = root.right;

TreeNode newNode = UpsideDownBinaryTree(root.left);

left.left = right;

left.right = root;

root.left = null;

root.right = null;

return newNode;

}

}

}

### Read N Characters Given Read4

The API: *int read4(char \*buf)* reads 4 characters at a time from a file.

The return value is the actual number of characters read. For example, it returns 3 if there

is only 3 characters left in the file.

By using the *read4* API, implement the function *int read(char \*buf, int n)* that reads *n*

characters from the file.

Note: The *read* function will only be called once for each test case.

**Solution 1 – Time O (n) Space O (1)**

public class ReadNCharactersGivenRead4 extends Reader4 {

public int read(char[] buf, int n) {

char[] buffer = new char[4];

int readBytes = 0;

boolean eof = false;

while (!eof && readBytes < n) {

int sz = read4(buffer);

if (sz < 4)

eof = true;

int bytes = Math.min(n - readBytes, sz);

System.arraycopy(buffer /\* src \*/, 0 /\* srcPos \*/, buf /\* dest \*/,

readBytes /\* destPos \*/, bytes /\* length \*/);

readBytes += bytes;

}

return readBytes;

}

}

class Reader4 {

int read4(char[] buf) {

// some dummy return

return 0;

}

}

### Read N Characters Given Read4 – Call Multiple Times

Similar to Question [Read N Characters Given Read4], but the read function may be

called multiple times.

**Solution 1 – Time O (n) Space O (1)**

public class ReadNCharactersGivenRead4CallMultipleTimes extends Reader4 {

private char[] buffer = new char[4];

int offset = 0, bufsize = 0;

/\*\*

\* @param buf

\* Destination buffer

\* @param n

\* Maximum number of characters to read

\* @return The number of characters read

\*/

public int read(char[] buf, int n) {

int readBytes = 0;

boolean eof = false;

while (!eof && readBytes < n) {

int sz = (bufsize > 0) ? bufsize : read4(buffer);

if (bufsize == 0 && sz < 4)

eof = true;

int bytes = Math.min(n - readBytes, sz);

System.arraycopy(buffer /\* src \*/, offset /\* srcPos \*/, buf /\* dest \*/,

readBytes /\* destPos \*/, bytes /\* length \*/);

offset = (offset + bytes) % 4;

bufsize = sz - bytes;

readBytes += bytes;

}

return readBytes;

}

}

class Reader4 {

int read4(char[] buf) {

// some dummy return

return 0;

}

}

### Longest Substring with At Most Two Distinct Characters

Given a string *S*, find the length of the longest substring *T* that contains at most two

distinct characters.

For example,

Given *S* = “eceba”,

*T* is "ece" which its length is 3.

**Solution 1 – Time O (n) Space O (1)**

public class LongestSubstringWithAtMostTwoDistinctCharacters {

public int lengthOfLongestSubstringTwoDistinct(String s) {

int i = 0, j = -1, maxLen = 0;

for (int k = 1; k < s.length(); k++) {

if (s.charAt(k) == s.charAt(k - 1))

continue;

if (j >= 0 && s.charAt(j) != s.charAt(k)) {

maxLen = Math.max(k - i, maxLen);

i = j + 1;

}

j = k - 1;

}

return Math.max(s.length() - i, maxLen);

}

}

**Solution 2 – Time O (n) Space O (1)**

public class LongestSubstringWithAtMostTwoDistinctCharacters {

public int lengthOfLongestSubstringTwoDistinct(String s) {

int[] count = new int[256];

int i = 0, numDistinct = 0, maxLen = 0;

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

if (count[s.charAt(j)] == 0) {

numDistinct++;

}

count[s.charAt(j)]++;

while (numDistinct > 2) {

count[s.charAt(i)]--;

if (count[s.charAt(i)] == 0) {

numDistinct--;

}

i++;

}

maxLen = Math.max(j - i + 1, maxLen);

}

return maxLen;

}

}

### Intersection of Two Linked Lists

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.

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

if (headA == null || headB == null) {

return null;

}

ListNode pA = headA;

ListNode pB = headB;

while (pA != null && pB != null) {

pA = pA.next;

pB = pB.next;

}

if (pA == null) {

pA = headB;

} else {

pB = headA;

}

while (pA != null && pB != null) {

pA = pA.next;

pB = pB.next;

}

if (pA == null) {

pA = headB;

} else {

pB = headA;

}

while (pA != null && pB != null) {

if (pA == pB) {

return pA;

}

pA = pA.next;

pB = pB.next;

}

return null;

}

}

**Solution 2 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) {

\* val = x;

\* next = null;

\* }

\* }

\*/

public class Solution {

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

if (headA == null || headB == null) {

return null;

}

int lenA = getLen(headA);

int lenB = getLen(headB);

if (lenA > lenB) {

while (lenA > lenB) {

headA = headA.next;

lenA--;

}

} else {

while (lenA < lenB) {

headB = headB.next;

lenB--;

}

}

while (headA != null) {

if (headA == headB) {

return headA;

}

headA = headA.next;

headB = headB.next;

}

return null;

}

public int getLen(ListNode node) {

int len = 0;

while (node != null) {

len++;

node = node.next;

}

return len;

}

}

### One Edit Distance

Given two strings *S* and *T*, determine if they are both one edit distance apart.

**Hint:**

1. If | n – m | is greater than 1, we know immediately both are not one-edit distance

apart.

2. It might help if you consider these cases separately, m == n and m ≠ n.

3. Assume that *m* is always ≤ *n*, which greatly simplifies the conditional statements.

If m > n, we could just simply swap *S* and *T*.

4. If *m* == *n*, it becomes finding if there is *exactly* one modified operation. If m ≠ n,

you do not have to consider the delete operation. Just consider the insert operation

in *T*.

**Solution 1 – Time O (n) Space O (1)**

public class OneEditDistance {

public boolean isOneEditDistance(String s, String t) {

int m = s.length(), n = t.length();

if (m > n) {

return isOneEditDistance(t, s);

}

if (n - m > 1) {

return false;

}

int i = 0, shift = n - m;

while (i < m && s.charAt(i) == t.charAt(i)) {

i++;

}

if (i == m) {

return shift > 0;

}

if (shift == 0) {

i++;

}

while (i < m && s.charAt(i) == t.charAt(i + shift)) {

i++;

}

return i == m;

}

}

### Find Peak Element

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.

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public int findPeakElement(int[] nums) {

int low = 0;

int high = nums.length - 1;

while (low < high) {

int mid1 = (low + high) / 2;

int mid2 = mid1 + 1;

if (nums[mid1] < nums[mid2]) {

low = mid2;

} else {

high = mid1;

}

}

return low;

}

}

### Missing Ranges

Given a sorted integer array where the range of elements are [0, 99] inclusive, return its

missing ranges.

For example, given [0, 1, 3, 50, 75], return [“2”, “4->49”, “51->74”, “76->99”]

**Solution 1 – Time O (n) Space O (1)**

public class MissingRanges {

public List<String> findMissingRanges(int[] vals, int start, int end) {

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

int prev = start - 1;

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

int curr = (i == vals.length) ? end + 1 : vals[i];

if (curr - prev >= 2) {

ranges.add(getRange(prev + 1, curr - 1));

}

prev = curr;

}

return ranges;

}

private String getRange(int from, int to) {

return (from == to) ? String.valueOf(from) : from + "->" + to;

}

}

### Maximum Gap

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Try to solve it in linear time/space.

Return 0 if the array contains less than 2 elements.

You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

class Bucket {

int low;

int high;

public Bucket() {

low = -1;

high = -1;

}

}

public int maximumGap(int[] num) {

if (num == null || num.length < 2) {

return 0;

}

int max = num[0];

int min = num[0];

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

max = Math.max(max, num[i]);

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

}

// initialize an array of buckets

Bucket[] buckets = new Bucket[num.length + 1]; // project to (0 - n)

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

buckets[i] = new Bucket();

}

double interval = (double) num.length / (max - min);

// distribute every number to a bucket array

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

int index = (int) ((num[i] - min) \* interval);

if (buckets[index].low == -1) {

buckets[index].low = num[i];

buckets[index].high = num[i];

} else {

buckets[index].low = Math.min(buckets[index].low, num[i]);

buckets[index].high = Math.max(buckets[index].high, num[i]);

}

}

// scan buckets to find maximum gap

int result = 0;

int prev = buckets[0].high;

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

if (buckets[i].low != -1) {

result = Math.max(result, buckets[i].low - prev);

prev = buckets[i].high;

}

}

return result;

}

}

### Compare Version Numbers

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

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

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;

}

}

### Fraction to Recurring Decimal

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)".

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public String fractionToDecimal(int numerator, int denominator) {

if (denominator == 0) {

return "0";

}

String ret = "";

if ((numerator > 0 && denominator < 0)

|| (numerator < 0 && denominator > 0)) {

ret += "-";

}

long num = Math.abs((long) numerator);

long denom = Math.abs((long) denominator);

ret += (num / denom);

long reminder = num % denom;

if (reminder == 0) {

return ret;

}

ret += ".";

reminder = reminder \* 10;

Map<Long, Integer> map = new HashMap<Long, Integer>();

while (reminder != 0) {

map.put(reminder, ret.length());

ret += reminder / denom;

reminder = (reminder % denom) \* 10;

if (map.containsKey(reminder)) {

int idx1 = map.get(reminder);

int idx2 = ret.length();

String part1 = ret.substring(0, idx1);

String part2 = ret.substring(idx1, idx2);

return part1 + "(" + part2 + ")";

}

}

return ret;

}

}

### Two Sum II – Input Array is Sorted

Similar to Question [Two Sum], except that the input array is already sorted in

ascending order.

**Solution 1 – Time O (n) Space O (1)**

public class TwoSumIIInputArrayIsSorted {

public int[] twoSum(int[] numbers, int target) {

// Assume input is already sorted.

int i = 0, j = numbers.length - 1;

while (i < j) {

int sum = numbers[i] + numbers[j];

if (sum < target) {

i++;

} else if (sum > target) {

j--;

} else {

return new int[] { i + 1, j + 1 };

}

}

throw new IllegalArgumentException("No two sum solution");

}

}

### Excel Sheet Column Title

Given a positive integer, return its corresponding column title as appear in an Excel sheet.

For example:

1 -> A

2 -> B

3 -> C

...

26 -> Z

27 -> AA

28 -> AB

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public String convertToTitle(int n) {

if (n <= 0) {

throw new IllegalArgumentException("Input is not valid!");

}

StringBuilder sb = new StringBuilder();

while (n > 0) {

n--;

char ch = (char) (n % 26 + 'A');

n /= 26;

sb.append(ch);

}

sb.reverse();

return sb.toString();

}

}

### Majority Element

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.

**Solution 1 – Time O (n) Space O (1)**

// Moore's voting algorithm

public class Solution {

public int majorityElement(int[] nums) {

int majorityIndex = 0;

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

if (nums[majorityIndex] == nums[i]) {

count++;

} else {

count--;

}

if (count == 0) {

majorityIndex = i;

count = 1;

}

}

return nums[majorityIndex];

}

}

**Solution 2 – Time O (n) Space O (1)**

public class Solution {

public int majorityElement(int[] num) {

int ret = 0;

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

int ones = 0, zeros = 0;

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

if ((num[j] & (1 << i)) != 0) {

++ones;

} else

++zeros;

}

if (ones > zeros)

ret |= (1 << i);

}

return ret;

}

}

**Solution 3 – Time O (n log n) Space O (1)**

public class Solution {

public int majorityElement(int[] nums) {

if (nums.length == 1) {

return nums[0];

}

Arrays.sort(nums);

return nums[nums.length / 2];

}

}

### Two Sum III – Data Structure Design

Design and implement a *TwoSum* class. It should support the following operations: *add*

and *find*.

*add*(*input*) – Add the number *input* to an internal data structure.

*find*(*value*) – Find if there exists any pair of numbers which sum is equal to the *value*.

For example,

*add*(1); *add*(3); *add*(5); *find*(4) true; *find*(7) false

**Solution 1 – Add Time O (1) Find Time O (n) Space O (n)**

public class TwoSumIIIDataStructureDesign {

private Map<Integer, Integer> table = new HashMap<>();

public void add(int input) {

int count = table.containsKey(input) ? table.get(input) : 0;

table.put(input, count + 1);

}

public boolean find(int val) {

for (Map.Entry<Integer, Integer> entry : table.entrySet()) {

int num = entry.getKey();

int y = val - num;

if (y == num) {

// For duplicates, ensure there are at least two individual

// numbers.

if (entry.getValue() >= 2)

return true;

} else if (table.containsKey(y)) {

return true;

}

}

return false;

}

}

### Excel Sheet Column Number

Related to question [Excel Sheet Column Title](https://oj.leetcode.com/problems/excel-sheet-column-title/)

Given a column title as appear in an Excel sheet, return its corresponding column number.

For example:

A -> 1

B -> 2

C -> 3

...

Z -> 26

AA -> 27

AB -> 28

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int titleToNumber(String s) {

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

return 0;

}

int digit = s.length();

int result = 0;

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

result = result \* 26 + ((int) s.charAt(i) % 65 + 1);

}

return result;

}

}

### Factorial Trailing Zeroes

Given an integer *n*, return the number of trailing zeroes in *n*!.

**Note:** Your solution should be in logarithmic time complexity.

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public int trailingZeroes(int n) {

if (n < 0)

return -1;

int count = 0;

for (long i = 5; n / i >= 1; i \*= 5) {

count += n / i;

}

return count;

}

}

### Binary Search Tree Iterator

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.

**Solution 1 – Time O (1) Space O (h)**

/\*\*

\* Definition for binary tree

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class BSTIterator {

Deque<TreeNode> s;

TreeNode node;

public BSTIterator(TreeNode root) {

s = new LinkedList<TreeNode>();

node = root;

}

/\*\* @return whether we have a next smallest number \*/

public boolean hasNext() {

return !s.isEmpty() || node != null;

}

/\*\* @return the next smallest number \*/

public int next() {

int ret;

while (!s.isEmpty() || node != null) {

if (node != null) {

s.push(node);

node = node.left;

} else {

node = s.pop();

ret = node.val;

node = node.right;

return ret;

}

}

return 0;

}

}

/\*\*

\* Your BSTIterator will be called like this:

\* BSTIterator i = new BSTIterator(root);

\* while (i.hasNext()) v[f()] = i.next();

\*/

### Dungeon Game

The demons had captured the princess (**P**) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid. Our valiant knight (**K**) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (*negative* integers) upon entering these rooms; other rooms are either empty (*0's*) or contain magic orbs that increase the knight's health (*positive* integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

**Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.**

For example, given the dungeon below, the initial health of the knight must be at least **7** if he follows the optimal path RIGHT-> RIGHT -> DOWN -> DOWN.

|  |  |  |
| --- | --- | --- |
| -2 (K) | -3 | 3 |
| -5 | -10 | 1 |
| 10 | 30 | -5 (P) |

**Notes:**

* The knight's health has no upper bound.
* Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

**Solution 1 – Time O (n \* m) Space O (n \* m)**

public class Solution {

public int calculateMinimumHP(int[][] dungeon) {

int m = dungeon.length;

int n = dungeon[0].length;

// init dp table

int[][] h = new int[m][n];

h[m - 1][n - 1] = Math.max(1 - dungeon[m - 1][n - 1], 1);

// init last col

for (int i = m - 2; i >= 0; i--) {

h[i][n - 1] = Math.max(h[i + 1][n - 1] - dungeon[i][n - 1], 1);

}

// init last row

for (int j = n - 2; j >= 0; j--) {

h[m - 1][j] = Math.max(h[m - 1][j + 1] - dungeon[m - 1][j], 1);

}

// calculate dp table

for (int i = m - 2; i >= 0; i--) {

for (int j = n - 2; j >= 0; j--) {

int down = Math.max(h[i + 1][j] - dungeon[i][j], 1);

int right = Math.max(h[i][j + 1] - dungeon[i][j], 1);

h[i][j] = Math.min(right, down);

}

}

return h[0][0];

}

}

### Largest Number

Given a list of non negative integers, arrange them such that they form the largest number.

For example, given [3, 30, 34, 5, 9], the largest formed number is 9534330.

Note: The result may be very large, so you need to return a string instead of an integer.

**Solution 1 – Time O (n) Space O (1)**

class NumbersComparator implements Comparator<String> {

@Override

public int compare(String s1, String s2) {

return (s2 + s1).compareTo(s1 + s2);

}

}

public class Solution {

public String largestNumber(int[] nums) {

String[] strs = new String[nums.length];

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

strs[i] = Integer.toString(nums[i]);

}

Arrays.sort(strs, new NumbersComparator());

StringBuilder sb = new StringBuilder();

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

sb.append(strs[i]);

}

String result = sb.toString();

int index = 0;

while (index < result.length() && result.charAt(index) == '0') {

index++;

}

if (index == result.length()) {

return "0";

}

return result.substring(index);

}

}

### Reverse Words in a String II

Similar to Question [Reverse Words in a String], but with the following constraints:

“The input string does not contain leading or trailing spaces and the words are always

separated by a single space.”

Could you do it *in-place* without allocating extra space?

**Solution 1 – Time O (n) Space O (1)**

public class ReverseWordsInAStringII {

public void reverseWords(char[] s) {

reverse(s, 0, s.length);

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

if (j == s.length || s[j] == ' ') {

reverse(s, i, j);

i = j + 1;

}

}

}

private void reverse(char[] s, int begin, int end) {

for (int i = 0; i < (end - begin) / 2; i++) {

char temp = s[begin + i];

s[begin + i] = s[end - i - 1];

s[end - i - 1] = temp;

}

}

}

### Repeated DNA Sequences

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"].

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public List<String> findRepeatedDnaSequences(String s) {

List<String> ret = new LinkedList<String>();

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

return ret;

}

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

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

Map<Character, Integer> map = new HashMap<Character, Integer>();

map.put('A', 0);

map.put('C', 1);

map.put('G', 2);

map.put('T', 3);

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

int v = 0;

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

v = v << 2;

v += map.get(s.charAt(j));

}

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

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

}

}

return ret;

}

}

### Best Time to Buy and Sell Stock IV

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 at most **k** transactions.

**Note:**  
You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

**Solution 1 – Time O (n \* k) Space O (k)**

public class Solution {

public int maxProfit(int k, int[] prices) {

if (prices.length < 2 || k <= 0)

return 0;

// pass leetcode online judge (can be ignored)

if (k == 1000000000)

return 1648961;

int[] local = new int[k + 1];

int[] global = new int[k + 1];

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

int diff = prices[i + 1] - prices[i];

for (int j = k; j >= 1; j--) {

local[j] = Math.max(global[j - 1] + Math.max(diff, 0), local[j]

+ diff);

global[j] = Math.max(local[j], global[j]);

}

}

return global[k];

}

}

### Rotate Array

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/)

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public void rotate(int[] nums, int k) {

if (nums == null || nums.length <= 1) {

return;

}

k = k % nums.length;

int l = 0;

int r = nums.length - 1;

reverse(nums, l, r);

l = 0;

r = k - 1;

reverse(nums, l, r);

l = k;

r = nums.length - 1;

reverse(nums, l, r);

}

public void reverse(int[] nums, int l, int r) {

while (l < r) {

int tmp = nums[l];

nums[l++] = nums[r];

nums[r--] = tmp;

}

}

}

### Reverse Bits

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/)

**Solution 1 – Time O (1) Space O (1)**

public class Solution {

// you need treat n as an unsigned value

public int reverseBits(int n) {

int ret = n;

ret = ret >>> 16 | ret << 16;

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

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

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

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

return ret;

}

}

**Solution 2 – Time O (n) Space O (1)**

public class Solution {

// you need treat n as an unsigned value

public int reverseBits(int n) {

for (int i = 0; i < 16; i++) {

n = swapBits(n, i, 32 - i - 1);

}

return n;

}

public int swapBits(int n, int i, int j) {

int a = (n >> i) & 1;

int b = (n >> j) & 1;

if ((a ^ b) != 0) {

return n ^= (1 << i) | (1 << j);

}

return n;

}

}

### Number of 1 Bits

Write a function that takes an unsigned integer and returns the number of ’1' bits it has (also known as the [Hamming weight](http://en.wikipedia.org/wiki/Hamming_weight)).

For example, the 32-bit integer ’11' has binary representation 00000000000000000000000000001011, so the function should return 3.

**Solution 1 – Time O (n) Space O (1) – n is number of bits**

public class Solution {

// you need to treat n as an unsigned value

public int hammingWeight(int n) {

int count = 0;

while (n != 0) {

n = n & (n - 1);

count++;

}

return count;

}

}

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

// you need to treat n as an unsigned value

public int hammingWeight(int n) {

int count = 0;

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

if (getBit(n, i) == true) {

count++;

}

}

return count;

}

public boolean getBit(int n, int i) {

return (n & (1 << i)) != 0;

}

}

### House Robber

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**.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int rob(int[] nums) {

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

return 0;

}

int ret = 0;

int prepre = 0;

int pre = 0;

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

ret = Math.max(prepre + nums[i], pre);

prepre = pre;

pre = ret;

}

return ret;

}

}

### Binary Tree Right Side View

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].

**Solution 1 – Time O (n) Space O (log n)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<Integer> rightSideView(TreeNode root) {

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

if (root == null)

return result;

Queue<TreeNode> queue = new LinkedList<TreeNode>();

queue.add(root);

while (queue.size() > 0) {

// get size here

int size = queue.size();

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

TreeNode top = queue.remove();

// the first element in the queue (right-most of the tree)

if (i == 0) {

result.add(top.val);

}

// add right first

if (top.right != null) {

queue.add(top.right);

}

// add left

if (top.left != null) {

queue.add(top.left);

}

}

}

return result;

}

}

### Number of Islands

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

**Solution 1 – Time O (n^2) Space O (n)**

// Iterative

public class Solution {

public int numIslands(char[][] grid) {

if (grid == null || grid.length == 0 || grid[0] == null

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

return 0;

}

int rLen = grid.length;

int cLen = grid[0].length;

int ret = 0;

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

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

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

ret++;

helper(grid, i, j);

}

}

}

return ret;

}

public void helper(char[][] grid, int i, int j) {

int index = i \* grid[0].length + j;

Queue<Integer> q = new LinkedList<Integer>();

q.offer(index);

grid[i][j] = '2';

while (!q.isEmpty()) {

index = q.poll();

int r = index / grid[0].length;

int c = index % grid[0].length;

if (r > 0 && grid[r - 1][c] == '1') {

grid[r - 1][c] = '2';

q.offer((r - 1) \* grid[0].length + c);

}

if (r < grid.length - 1 && grid[r + 1][c] == '1') {

grid[r + 1][c] = '2';

q.offer((r + 1) \* grid[0].length + c);

}

if (c > 0 && grid[r][c - 1] == '1') {

grid[r][c - 1] = '2';

q.offer(r \* grid[0].length + c - 1);

}

if (c < grid[0].length - 1 && grid[r][c + 1] == '1') {

grid[r][c + 1] = '2';

q.offer(r \* grid[0].length + c + 1);

}

}

}

}

**Solution 2 – Time O (n^2) Space O (n)**

// Recursive

public class Solution {

public int numIslands(char[][] grid) {

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

return 0;

int count = 0;

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

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

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

count++;

merge(grid, i, j);

}

}

}

return count;

}

public void merge(char[][] grid, int i, int j) {

// validity checking

if (i < 0 || j < 0 || i > grid.length - 1 || j > grid[0].length - 1)

return;

// if current cell is water or visited

if (grid[i][j] != '1')

return;

// set visited cell to '2'

grid[i][j] = '2';

// merge all adjacent land

merge(grid, i - 1, j);

merge(grid, i + 1, j);

merge(grid, i, j - 1);

merge(grid, i, j + 1);

}

}

### Bitwise AND of Numbers Range

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.

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public int rangeBitwiseAnd(int m, int n) {

while (n > m) {

n = n & n - 1;

}

return n;

}

}

### Happy Number

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

**Solution 1 – Time O (?) Space O (1)**

public class Solution {

public boolean isHappy(int n) {

int slow = n;

int fast = n;

do {

slow = digitSquareSum(slow);

fast = digitSquareSum(fast);

fast = digitSquareSum(fast);

} while (slow != fast);

if (slow == 1) {

return true;

} else {

return false;

}

}

int digitSquareSum(int n) {

int sum = 0, tmp;

while (n != 0) {

tmp = n % 10;

sum += tmp \* tmp;

n /= 10;

}

return sum;

}

}

**Solution 2 – Time O (?) Space O (?)**

public class Solution {

public boolean isHappy(int n) {

Set<Integer> set = new HashSet<Integer>();

while (n != 1) {

if (set.contains(n)) {

return false;

}

set.add(n);

n = digitSquareSum(n);

}

return true;

}

int digitSquareSum(int n) {

int sum = 0, tmp;

while (n != 0) {

tmp = n % 10;

sum += tmp \* tmp;

n /= 10;

}

return sum;

}

}

### Remove Linked List Elements

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

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode removeElements(ListNode head, int val) {

ListNode helper = new ListNode(0);

helper.next = head;

ListNode p = helper;

while (p.next != null) {

if (p.next.val == val) {

p.next = p.next.next;

} else {

p = p.next;

}

}

return helper.next;

}

}

### Count Primes

**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++) {

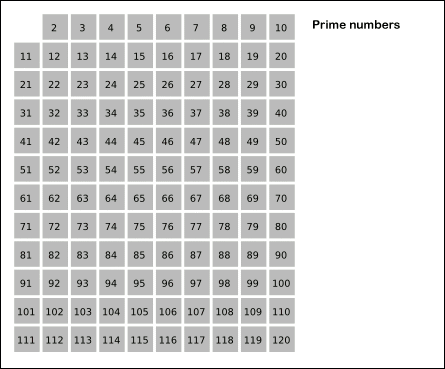
if (num % i == 0) return false;

}

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;

}

**Solution 1 – Time O (n) Space O (*n* log log *n*)**

public class Solution {

public int countPrimes(int n) {

if (n <= 2)

return 0;

// init an array to track prime numbers

boolean[] primes = new boolean[n];

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

primes[i] = true;

for (int i = 2; i <= Math.sqrt(n - 1); i++) {

// or for (int i = 2; i <= n-1; i++) {

if (primes[i]) {

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

primes[j] = false;

}

}

int count = 0;

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

if (primes[i])

count++;

}

return count;

}

}

### Isomorphic Strings

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.

**Note:**  
You may assume both ***s*** and ***t*** have the same length.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public boolean isIsomorphic(String s, String t) {

int len = s.length();

int[] m1 = new int[256];

int[] m2 = new int[256];

for (int i = 0; i < 256; i++) {

m1[i] = m2[i] = -1;

}

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

if (m1[s.charAt(i)] != m2[t.charAt(i)]) {

return false;

}

m1[s.charAt(i)] = m2[t.charAt(i)] = i;

}

return true;

}

}

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public boolean isIsomorphic(String s, String t) {

if (s == null || t == null) {

return false;

}

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

return true;

}

Map<Character, Character> map = new HashMap<Character, Character>();

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

char sc = s.charAt(i);

char tc = t.charAt(i);

if (map.containsKey(sc)) {

if (map.get(sc) != tc) {

return false;

}

} else {

if (map.containsValue(tc)) {

return false;

} else {

map.put(sc, tc);

}

}

}

return true;

}

}

### Reverse Linked List

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?

**Solution 1 – Time O (n) Space O (1)**

// Iterative

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode reverseList(ListNode head) {

ListNode prev = null;

ListNode curr = head;

while (curr != null) {

ListNode nextTemp = curr.next;

curr.next = prev;

prev = curr;

curr = nextTemp;

}

return prev;

}

}

**Solution 2 – Time O (n) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public ListNode reverseList(ListNode head) {

if (head == null || head.next == null)

return head;

// get second node

ListNode second = head.next;

// set first's next to be null

head.next = null;

ListNode rest = reverseList(second);

second.next = head;

return rest;

}

}

### Course Schedule

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).

**Solution 1 – Time O (V+E) Space O (V^V)**

// BFS - adjacency matrix

public class Solution {

public boolean canFinish(int numCourses, int[][] prerequisites) {

int[][] matrix = new int[numCourses][numCourses]; // i->j

int[] pCount = new int[numCourses];

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

int pre = prerequisites[i][1];

int ready = prerequisites[i][0];

if (matrix[pre][ready] == 0) { // duplicate case

pCount[ready]++;

}

matrix[pre][ready] = 1;

}

Queue<Integer> q = new LinkedList<Integer>();

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

if (pCount[i] == 0) {

q.offer(i);

}

}

int count = 0;

while (!q.isEmpty()) {

int val = q.poll();

count++;

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

if (matrix[val][i] != 0) {

if (--pCount[i] == 0) {

q.offer(i);

}

}

}

}

return count == numCourses;

}

}

**Solution 2 – Time O (V+E) Space O (V)**

// DFS

public class Solution {

public boolean canFinish(int numCourses, int[][] prerequisites) {

if (prerequisites == null) {

throw new IllegalArgumentException("illegal prerequisites array");

}

int len = prerequisites.length;

if (numCourses == 0 || len == 0) {

return true;

}

// track visited courses

int[] visit = new int[numCourses];

// use the map to store what courses depend on a course

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

for (int[] a : prerequisites) {

if (map.containsKey(a[1])) {

map.get(a[1]).add(a[0]);

} else {

ArrayList<Integer> l = new ArrayList<Integer>();

l.add(a[0]);

map.put(a[1], l);

}

}

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

if (!canFinishDFS(map, visit, i))

return false;

}

return true;

}

private boolean canFinishDFS(HashMap<Integer, ArrayList<Integer>> map,

int[] visit, int i) {

if (visit[i] == -1)

return false;

if (visit[i] == 1)

return true;

visit[i] = -1;

if (map.containsKey(i)) {

for (int j : map.get(i)) {

if (!canFinishDFS(map, visit, j))

return false;

}

}

visit[i] = 1;

return true;

}

}

### Implement Trie (Prefix Tree)

Implement a trie with insert, search, and startsWith methods.

**Note:**  
You may assume that all inputs are consist of lowercase letters a-z.

**Solution 1 – Time O (n) Space O (n)**

class TrieNode {

// Initialize your data structure here.

Map<Character, TrieNode> map;

boolean isLeaf;

public TrieNode() {

map = new HashMap<Character, TrieNode>();

isLeaf = false;

}

}

public class Trie {

private TrieNode root;

public Trie() {

root = new TrieNode();

}

// Inserts a word into the trie.

public void insert(String word) {

TrieNode cur = root;

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

char ch = word.charAt(i);

if (!cur.map.containsKey(ch)) {

cur.map.put(ch, new TrieNode());

}

cur = cur.map.get(ch);

}

cur.isLeaf = true;

}

// Returns if the word is in the trie.

public boolean search(String word) {

TrieNode cur = root;

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

char ch = word.charAt(i);

if (!cur.map.containsKey(ch)) {

return false;

}

cur = cur.map.get(ch);

}

return cur.isLeaf;

}

// Returns if there is any word in the trie

// that starts with the given prefix.

public boolean startsWith(String prefix) {

TrieNode cur = root;

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

char ch = prefix.charAt(i);

if (!cur.map.containsKey(ch)) {

return false;

}

cur = cur.map.get(ch);

}

return true;

}

}

// Your Trie object will be instantiated and called as such:

// Trie trie = new Trie();

// trie.insert("somestring");

// trie.search("key");

### Minimum Size Subarray Sum

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*).

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int minSubArrayLen(int s, int[] nums) {

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

return 0;

}

int l = 0;

int r = 0;

int sum = 0;

int ret = Integer.MAX\_VALUE;

while (r < nums.length) {

sum += nums[r++];

while (sum >= s) {

ret = Math.min(ret, r - l);

sum -= nums[l++];

}

}

return ret == Integer.MAX\_VALUE ? 0 : ret;

}

}

### Course Schedule II

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).

**Solution 1 – Time O (V+E) Space O (V+E)**

// BFS - adjacency list

public class Solution {

public int[] findOrder(int numCourses, int[][] prerequisites) {

int[] pCount = new int[numCourses];

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

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

list.add(new ArrayList<Integer>());

}

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

int pre = prerequisites[i][1];

int ready = prerequisites[i][0];

list.get(pre).add(ready);

pCount[ready]++;

}

int count = 0;

int[] ret = new int[numCourses];

Queue<Integer> q = new LinkedList<Integer>();

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

if (pCount[i] == 0) {

q.offer(i);

}

}

while (!q.isEmpty()) {

int val = q.poll();

ret[count++] = val;

for (int i = 0; i < list.get(val).size(); i++) {

if (--pCount[list.get(val).get(i)] == 0) {

q.offer(list.get(val).get(i));

}

}

}

if (count == numCourses) {

return ret;

} else {

return new int[0];

}

}

}

### Add and Search Word - Data structure design

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.

**Solution 1 – Time O (n) Space O (?)**

public class WordDictionary {

class TrieNode {

HashMap<Character, TrieNode> map;

boolean isEnd;

TrieNode() {

map = new HashMap<Character, TrieNode>();

isEnd = false;

}

}

TrieNode root;

WordDictionary() {

root = new TrieNode();

}

// Adds a word into the data structure.

public void addWord(String word) {

TrieNode node = root;

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

char ch = word.charAt(i);

if (!node.map.containsKey(ch)) {

node.map.put(ch, new TrieNode());

}

node = node.map.get(ch);

}

node.isEnd = true;

}

// Returns if the word is in the data structure. A word could

// contain the dot character '.' to represent any one letter.

public boolean search(String word) {

return search(word, root, 0);

}

public boolean search(String word, TrieNode node, int index) {

if (index == word.length()) {

return node.isEnd;

}

char ch = word.charAt(index);

if (ch == '.') {

for (TrieNode n : node.map.values()) {

if (search(word, n, index + 1)) {

return true;

}

}

return false;

}

if (!node.map.containsKey(ch)) {

return false;

}

return search(word, node.map.get(ch), index + 1);

}

}

// Your WordDictionary object will be instantiated and called as such:

// WordDictionary wordDictionary = new WordDictionary();

// wordDictionary.addWord("word");

// wordDictionary.search("pattern");

### House Robber II

**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**.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int rob(int[] nums) {

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

return 0;

}

if (nums.length == 1) {

return nums[0];

}

return Math.max(helper(nums, 0, nums.length - 2),

helper(nums, 1, nums.length - 1));

}

public int helper(int[] nums, int l, int r) {

int v1 = 0;

int v2 = 0;

int ret = 0;

for (int i = l; i <= r; i++) {

ret = Math.max(v1 + nums[i], v2);

v1 = v2;

v2 = ret;

}

return ret;

}

}

### Kth Largest Element in an Array

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.

**Solution 1 – Time O (N log K) Space O (k)**

public class Solution {

public int findKthLargest(int[] nums, int k) {

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

return 0;

}

PriorityQueue<Integer> q = new PriorityQueue<Integer>();

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

q.offer(nums[i]);

}

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

if (nums[i] > q.peek()) {

q.poll();

q.add(nums[i]);

}

}

return q.peek();

}

}

**Solution 2 – Time O (average N) Space O (1)**

public class Solution {

public int findKthLargest(int[] nums, int k) {

k = nums.length - k;

int l = 0;

int r = nums.length - 1;

while (l < r) {

int j = partition(nums, l, r);

if (j == k) {

break;

} else if (j < k) {

l = j + 1;

} else {

r = j - 1;

}

}

return nums[k];

}

private int partition(int[] nums, int l, int r) {

int pivot = nums[r];

int j = l - 1;

for (int i = l; i < r; i++) {

if (nums[i] <= pivot) {

j++;

swap(nums, i, j);

}

}

swap(nums, j + 1, r);

return j + 1;

}

public void swap(int[] a, int i, int j) {

final int tmp = a[i];

a[i] = a[j];

a[j] = tmp;

}

}

### Combination Sum III

Find all possible combinations of ***k*** numbers that add up to a number ***n***, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

***Example 1:***

Input: ***k*** = 3, ***n*** = 7

Output:

[[1,2,4]]

***Example 2:***

Input: ***k*** = 3, ***n*** = 9

Output:

[[1,2,6], [1,3,5], [2,3,4]]

**Solution 1 – Time O (NP) Space O (NP)**

public class Solution {

public List<List<Integer>> combinationSum3(int k, int n) {

List<List<Integer>> ret = new LinkedList<List<Integer>>();

if (k > n || n < 1) {

return ret;

}

helper(ret, new LinkedList<Integer>(), k, n, 1);

return ret;

}

public void helper(List<List<Integer>> ret, List<Integer> tmp, int len,

int sum, int start) {

if (len == 0 && sum == 0) {

ret.add(new LinkedList<Integer>(tmp));

return;

}

if (len == 0) {

return;

}

for (int i = start; i <= 9; i++) {

if (i <= sum) {

tmp.add(i);

helper(ret, tmp, len - 1, sum - i, i + 1);

tmp.remove(tmp.size() - 1);

}

}

}

}

### Contains Duplicate

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.

**Solution 1 - Time O (n) Space O (n)**

public class Solution {

public boolean containsDuplicate(int[] nums) {

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

return false;

}

HashSet<Integer> set = new HashSet<Integer>();

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

if (set.contains(nums[i])) {

return true;

}

set.add(nums[i]);

}

return false;

}

}

### Contains Duplicate II

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*.

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public boolean containsNearbyDuplicate(int[] nums, int k) {

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

return false;

}

Map<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

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*.

**Solution 1 – Time O (n) Space O (t)**

public class Solution {

public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {

if (nums == null || nums.length == 0 || k < 1 || t < 0) {

return false;

}

long bucketSize = (long) t + 1; // to handle t == 0

Map<Long, Long> map = new HashMap<Long, Long>();

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

long posNum = (long) nums[i] - Integer.MIN\_VALUE;

long bucket = posNum / bucketSize;

if (map.containsKey(bucket)

|| (map.containsKey(bucket - 1) && posNum

- map.get(bucket - 1) <= t)

|| (map.containsKey(bucket + 1) && map.get(bucket + 1)

- posNum <= t)) {

return true;

}

if (i >= k) {

map.remove(((long) nums[i - k] - Integer.MIN\_VALUE)

/ bucketSize);

}

map.put(bucket, posNum);

}

return false;

}

}

**Solution 2 – Time O (n lgk) Space O (k)**

public class Solution {

public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {

if (nums == null || nums.length == 0 || k < 1 || t < 0) {

return false;

}

TreeSet<Integer> set = new TreeSet<Integer>();

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

Integer floor = set.floor(nums[i] + t);

Integer ceil = set.ceiling(nums[i] - t);

while ((floor != null && floor >= nums[i])

|| (ceil != null && ceil <= nums[i])) {

return true;

}

if (i >= k) {

set.remove(nums[i - k]);

}

set.add(nums[i]);

}

return false;

}

}

### Maximal Square

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.

**Solution 1 – Time O (N^2) Space O (N^2)**

public class Solution {

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

if (matrix == null || matrix.length == 0 || matrix[0] == null

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

return 0;

}

int m = matrix.length;

int n = matrix[0].length;

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

int ret = 0;

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

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

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

edge[i][j] = Math.min(edge[i - 1][j],

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

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

}

}

}

return ret \* ret;

}

}

### Count Complete Tree Nodes

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 2hnodes inclusive at the last level h.

**Solution 1 – Time O (logn \* log n) Space O (log n)**  
/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int countNodes(TreeNode root) {

int ret = 0;

int h = height(root);

while (root != null) {

if (height(root.right) == h - 1) {

ret += 1 << (h - 1);

root = root.right;

} else {

ret += 1 << (h - 2);

root = root.left;

}

h--;

}

return ret;

}

public int height(TreeNode root) {

if (root == null) {

return 0;

}

return height(root.left) + 1;

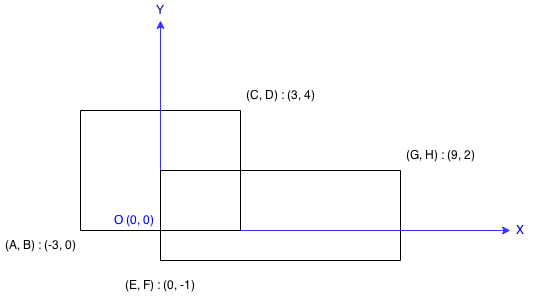
}

}

### Rectangle Area

Find the total area covered by two **rectilinear** rectangles in a **2D** plane.

Each rectangle is defined by its bottom left corner and top right corner as shown in the figure.



Assume that the total area is never beyond the maximum possible value of **int**.

**Solution 1 – Time O (1) Space O (1)**

public class Solution {

public int computeArea(int A, int B, int C, int D, int E, int F, int G,

int H) {

int areaA = (C - A) \* (D - B);

int areaB = (G - E) \* (H - F);

int lx = Math.max(A, E);

int rx = Math.min(C, G);

int ly = Math.max(B, F);

int ry = Math.min(D, H);

int overlap = 0;

if (rx > lx && ry > ly) {

overlap = (rx - lx) \* (ry - ly);

}

return areaA + areaB - overlap;

}

}

### Implement Stack using Queues

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).

**Solution 1 – Time O (push: O(n), other O(1)) Space O (n)**

class MyStack {

Queue<Integer> q = new LinkedList<Integer>();

// Push element x onto stack.

public void push(int x) {

q.offer(x);

for (int i = 0; i < q.size() - 1; i++) {

q.offer(q.poll());

}

}

// Removes the element on top of the stack.

public void pop() {

q.poll();

}

// Get the top element.

public int top() {

return q.peek();

}

// Return whether the stack is empty.

public boolean empty() {

return q.isEmpty();

}

}

### Invert Binary Tree

Invert a binary tree.

4

/ \

2 7

/ \ / \

1 3 6 9

to

4

/ \

7 2

/ \ / \

9 6 3 1

**Solution 1 – Time O (n) Space O (log n)**

// Iterative

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode invertTree(TreeNode root) {

if (root == null) {

return root;

}

Queue<TreeNode> q = new LinkedList<TreeNode>();

q.offer(root);

while (!q.isEmpty()) {

TreeNode n = q.poll();

TreeNode left = n.left;

n.left = n.right;

n.right = left;

if (n.left != null) {

q.offer(n.left);

}

if (n.right != null) {

q.offer(n.right);

}

}

return root;

}

}

**Solution 2 – Time O (n) Space O (log n)**

// Recursive

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode invertTree(TreeNode root) {

if (root == null) {

return null;

}

TreeNode left = invertTree(root.left);

TreeNode right = invertTree(root.right);

root.left = right;

root.right = left;

return root;

}

}

### Basic Calculator II

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.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int calculate(String s) {

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

return 0;

}

int ret = 0;

int pre = 0;

char sign = '+';

int cur = 0;

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

if (s.charAt(i) == ' ') {

continue;

}

else if (s.charAt(i) <= '9' && s.charAt(i) >= '0') {

cur = cur \* 10 + (s.charAt(i) - '0');

}

else {

if (sign == '+') {

ret += pre;

pre = cur;

}

else if (sign == '-') {

ret += pre;

pre = -cur;

}

else if (sign == '\*') {

pre \*= cur;

}

else if (sign == '/') {

pre /= cur;

}

sign = s.charAt(i);

cur = 0;

}

}

if (sign == '+') {

ret += pre;

pre = cur;

}

else if (sign == '-') {

ret += pre;

pre = -cur;

}

else if (sign == '\*') {

pre \*= cur;

}

else if (sign == '/') {

pre /= cur;

}

return ret + pre;

}

}

### Summary Ranges

Given a sorted integer array without duplicates, return the summary of its ranges.

For example, given [0,1,2,4,5,7], return ["0->2","4->5","7"].

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public List<String> summaryRanges(int[] nums) {

List<String> ret = new LinkedList<String>();

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

return ret;

}

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

int num = nums[i];

while (i < nums.length - 1 && nums[i] + 1 == nums[i + 1]) {

i++;

}

if (num != nums[i]) {

ret.add(num + "->" + nums[i]);

} else {

ret.add(Integer.toString(num));

}

}

return ret;

}

}

### Majority Element II

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)!

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public List<Integer> majorityElement(int[] nums) {

List<Integer> ret = new LinkedList<Integer>();

int c1 = 0;

int c2 = 0;

int m1 = 0;

int m2 = 1;

for (int n : nums) {

if (m1 == n) {

c1++;

} else if (m2 == n) {

c2++;

} else if (c1 == 0) {

m1 = n;

c1 = 1;

} else if (c2 == 0) {

m2 = n;

c2 = 1;

} else {

c1--;

c2--;

}

}

c1 = 0;

c2 = 0;

for (int n : nums) {

if (n == m1) {

c1++;

}

if (n == m2) {

c2++;

}

}

if (c1 > nums.length / 3) {

ret.add(m1);

}

if (c2 > nums.length / 3) {

ret.add(m2);

}

return ret;

}

}

### Kth Smallest Element in a BST

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).

**Solution 1 – Time O (n) Space O (n)**

// Not modify Tree structure

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public int kthSmallest(TreeNode root, int k) {

Deque<TreeNode> s = new LinkedList<TreeNode>();

while (!s.isEmpty() || root != null) {

if (root != null) {

s.push(root);

root = root.left;

} else {

TreeNode n = s.pop();

k--;

if (k == 0) {

return n.val;

}

root = n.right;

}

}

return 0;

}

}

**Solution 2 – Time O (first time O(n), afterwards O(log n)) Space O (n)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

class TreeNodeWithCount {

int val;

int count;

TreeNodeWithCount left;

TreeNodeWithCount right;

TreeNodeWithCount(int x) {

val = x;

count = 1;

}

}

public int kthSmallest(TreeNode root, int k) {

TreeNodeWithCount rootWithCount = buildTreeWithCount(root);

return kthSmallest(rootWithCount, k);

}

public 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;

}

public int kthSmallest(TreeNodeWithCount rootWithCount, int k) {

if (k <= 0 || k > rootWithCount.count) {

return -1;

}

if (rootWithCount.left != null) {

if (rootWithCount.left.count == k - 1) {

return rootWithCount.val;

} else if (rootWithCount.left.count > k - 1) {

return kthSmallest(rootWithCount.left, k);

} else {

return kthSmallest(rootWithCount.right, k - 1

- rootWithCount.left.count);

}

} else {

if (k == 1) {

return rootWithCount.val;

} else {

return kthSmallest(rootWithCount.right, k - 1);

}

}

}

}

### Power of Two

Given an integer, write a function to determine if it is a power of two.

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public boolean isPowerOfTwo(int n) {

if (n <= 0) {

return false;

}

return (n & (n - 1)) == 0;

}

}

### Implement Queue using Stacks

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).

**Solution 1 – Time O (pop/peek O(n), other O(1)) Space O (n)**

class MyQueue {

Deque<Integer> s1 = new LinkedList<Integer>();

Deque<Integer> s2 = new LinkedList<Integer>();

// Push element x to the back of queue.

public void push(int x) {

s1.push(x);

}

// Removes the element from in front of queue.

public void pop() {

if (s2.isEmpty()) {

while (!s1.isEmpty()) {

s2.push(s1.pop());

}

}

s2.pop();

}

// Get the front element.

public int peek() {

if (s2.isEmpty()) {

while (!s1.isEmpty()) {

s2.push(s1.pop());

}

}

return s2.peek();

}

// Return whether the queue is empty.

public boolean empty() {

return s1.isEmpty() && s2.isEmpty();

}

}

### Palindrome Linked List

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?

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* 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) {

ListNode fast = head;

ListNode slow = head;

while (fast != null && fast.next != null) {

fast = fast.next.next;

slow = slow.next;

}

if (fast != null) { // odd number

slow = slow.next;

}

ListNode reverse = null;

while (slow != null) {

ListNode n = slow;

slow = slow.next;

n.next = reverse;

reverse = n;

}

while (reverse != null) {

if (reverse.val != head.val) {

return false;

}

head = head.next;

reverse = reverse.next;

}

return true;

}

}

### Lowest Common Ancestor of a Binary Search Tree

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.

**Solution 1 – Time O (n) Space O (1)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

while (root != null) {

if (root.val > p.val && root.val > q.val)

root = root.left;

else if (root.val < p.val && root.val < q.val)

root = root.right;

else

return root;

}

return null;

}

}

### Lowest Common Ancestor of a Binary Tree

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.

**Solution 1 – Time O (n) Space O (n)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

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) {

return right;

} else if (right == null) {

return left;

} else {

return root;

}

}

}

### Delete Node in a Linked List

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Supposed the linked list is 1 -> 2 -> 3 -> 4 and you are given the third node with value 3, the linked list should become 1 -> 2 -> 4 after calling your function.

**Solution 1 – Time O (1) Space O (1)**

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

public class Solution {

public void deleteNode(ListNode node) {

node.val = node.next.val;

node.next = node.next.next;

}

}

### Product of Array Except Self

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.)

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

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

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

return new int[0];

}

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

output[0] = nums[0];

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

output[i] = output[i - 1] \* nums[i];

}

for (int i = nums.length - 2; i >= 0; i--) {

nums[i] = nums[i + 1] \* nums[i];

}

output[nums.length - 1] = output[nums.length - 2];

for (int i = nums.length - 2; i >= 1; i--) {

output[i] = output[i - 1] \* nums[i + 1];

}

output[0] = nums[1];

return output;

}

}

### Search a 2D Matrix II

Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:

* Integers in each row are sorted in ascending from left to right.
* Integers in each column are sorted in ascending from top to bottom.

For example,

Consider the following matrix:

[

[1, 4, 7, 11, 15],

[2, 5, 8, 12, 19],

[3, 6, 9, 16, 22],

[10, 13, 14, 17, 24],

[18, 21, 23, 26, 30]

]

Given **target** = 5, return true.

Given **target** = 20, return false.

**Solution 1 – Time O (n + m) Space O (1)**

public class Solution {

public boolean searchMatrix(int[][] matrix, int target) {

if (matrix == null || matrix.length == 0 || matrix[0] == null

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

return false;

}

int m = matrix.length;

int n = matrix[0].length;

int r = 0;

int c = n - 1;

while (c >= 0 && r < m) {

if (target == matrix[r][c]) {

return true;

} else if (target > matrix[r][c]) {

r++;

} else {

c--;

}

}

return false;

}

}

### Different Ways to Add Parentheses

Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are+, - and \*.

**Example 1**

Input: "2-1-1".

((2-1)-1) = 0

(2-(1-1)) = 2

Output: [0, 2]

**Example 2**

Input: "2\*3-4\*5"

(2\*(3-(4\*5))) = -34

((2\*3)-(4\*5)) = -14

((2\*(3-4))\*5) = -10

(2\*((3-4)\*5)) = -10

(((2\*3)-4)\*5) = 10

Output: [-34, -14, -10, -10, 10]

**Solution 1 – Time O (NP) Space O (NP)**

public class Solution {

public List<Integer> diffWaysToCompute(String input) {

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

return helper(input, map);

}

public List<Integer> helper(String input, Map<String, List<Integer>> map) {

List<Integer> ret = new LinkedList<Integer>();

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

if (isSign(input.charAt(i))) {

String s1 = input.substring(0, i);

String s2 = input.substring(i + 1);

List<Integer> l1;

if (map.containsKey(s1)) {

l1 = map.get(s1);

} else {

l1 = helper(s1, map);

}

List<Integer> l2;

if (map.containsKey(s2)) {

l2 = map.get(s2);

} else {

l2 = helper(s2, map);

}

for (Integer i1 : l1) {

for (Integer i2 : l2) {

int r = 0;

switch (input.charAt(i)) {

case '+':

r = i1 + i2;

break;

case '-':

r = i1 - i2;

break;

case '\*':

r = i1 \* i2;

break;

}

ret.add(r);

}

}

}

}

if (ret.size() == 0) {

ret.add(Integer.parseInt(input));

}

map.put(input, ret);

return ret;

}

public boolean isSign(char c) {

return c == '+' || c == '-' || c == '\*';

}

}

### Valid Anagram

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.

**Solution 1 – Time O (n) Space O (1)**

// Assume only low case character

public class Solution {

public boolean isAnagram(String s, String t) {

if (s == null || t == null || s.length() != t.length()) {

return false;

}

int[] count = new int[26];

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

count[s.charAt(i) - 'a']++;

}

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

count[t.charAt(i) - 'a']--;

if (count[t.charAt(i) - 'a'] < 0) {

return false;

}

}

return true;

}

}

**Solution 2 – Time O (n) Space O (n)**

// Use hashmap for Unicode character

public class Solution {

public boolean isAnagram(String s, String t) {

if (s == null || t == null) {

return false;

}

Map<Character, Integer> map = new HashMap<Character, Integer>();

if (s.length() != t.length()) {

return false;

}

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

if (map.containsKey(s.charAt(i))) {

map.put(s.charAt(i), map.get(s.charAt(i)) + 1);

} else {

map.put(s.charAt(i), 1);

}

}

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

if (!map.containsKey(t.charAt(i)) || map.get(t.charAt(i)) < 1) {

return false;

}

map.put(t.charAt(i), map.get(t.charAt(i)) - 1);

}

return true;

}

}

### Shortest Word Distance

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = “coding”, word2 = “practice”, return 3. Given word1 = "makes", word2 = "coding", return 1.

Note: You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.

**Solution 1 – Time O (n+m) Space O (1)**

public class Solution {

public int shortestDistance(String[] words, String word1, String word2) {

int p1 = -1, p2 = -1, min = Integer.MAX\_VALUE;

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

if (words[i].equals(word1)) {

p1 = i;

}

if (words[i].equals(word2)) {

p2 = i;

}

if (p1 != -1 && p2 != -1) {

min = Math.min(min, Math.abs(p1 - p2));

}

}

return min;

}

}

### Shortest Word Distance II

This is a follow up of Shortest Word Distance. The only difference is now you are given the list of words and your method will be called repeatedly many times with different parameters. How would you optimize it?

Design a class which receives a list of words in the constructor, and implements a method that takes two words word1 and word2 and return the shortest distance between these two words in the list.

For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = “coding”, word2 = “practice”, return 3. Given word1 = "makes", word2 = "coding", return 1.

Note: You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.

**Solution 1 – Time O (n + m) Space O (n + m)**

public class Solution {

private Map<String, List<Integer>> map;

public void WordDistance(String[] words) {

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

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

String w = words[i];

if (map.containsKey(w)) {

map.get(w).add(i);

} else {

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

list.add(i);

map.put(w, list);

}

}

}

public int shortest(String word1, String word2) {

List<Integer> list1 = map.get(word1);

List<Integer> list2 = map.get(word2);

int ret = Integer.MAX\_VALUE;

for (int i = 0, j = 0; i < list1.size() && j < list2.size();) {

int index1 = list1.get(i), index2 = list2.get(j);

if (index1 < index2) {

ret = Math.min(ret, index2 - index1);

i++;

} else {

ret = Math.min(ret, index1 - index2);

j++;

}

}

return ret;

}

}

### Shortest Word Distance III

This is a follow up of Shortest Word Distance. The only difference is now word1 could be the same as word2.

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

word1 and word2 may be the same and they represent two individual words in the list.

For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = “makes”, word2 = “coding”, return 1. Given word1 = "makes", word2 = "makes", return 3.

Note: You may assume word1 and word2 are both in the list.

**Solution 1 – Time O (n+m) Space O (1)**

public class Solution {

public int shortestDistance(String[] words, String word1, String word2) {

int p1 = -1, p2 = -1, min = Integer.MAX\_VALUE;

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

if (words[i].equals(word1)) {

p1 = i;

}

if (words[i].equals(word2)) {

if (word1.equals(word2)) {

p1 = p2;

}

p2 = i;

}

if (p1 != -1 && p2 != -1) {

min = Math.min(min, Math.abs(p1 - p2));

}

}

return min;

}

}

### Strobogrammatic Number

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to determine if a number is strobogrammatic. The number is represented as a string.

For example, the numbers "69", "88", and "818" are all strobogrammatic.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public boolean isStrobogrammatic(String num) {

Map<Character, Character> map = new HashMap<Character, Character>();

map.put('6', '9');

map.put('9', '6');

map.put('0', '0');

map.put('1', '1');

map.put('8', '8');

int l = 0, r = num.length() - 1;

while (l <= r) {

if (!map.containsKey(num.charAt(l)))

return false;

if (map.get(num.charAt(l)) != num.charAt(r))

return false;

l++;

r--;

}

return true;

}

}

### Strobogrammatic Number II

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Find all strobogrammatic numbers that are of length = n.

For example, Given n = 2, return ["11","69","88","96"].

**Solution 1 – Time O (n^5) Space O (n^5)**

public class Solution {

public List<String> findStrobogrammatic(int n) {

List<String> one = Arrays.asList("0", "1", "8");

List<String> two = Arrays.asList("");

List<String> ret;

if (n % 2 == 1) {

ret = one;

} else {

ret = two;

}

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

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

for (String str : ret) {

if (i != n) {

newList.add("0" + str + "0");

}

newList.add("1" + str + "1");

newList.add("6" + str + "9");

newList.add("8" + str + "8");

newList.add("9" + str + "6");

}

ret = newList;

}

return ret;

}

}

### Group Shifted Strings

Given a string, we can "shift" each of its letter to its successive letter, for example: "abc" -> "bcd". We can keep "shifting" which forms the sequence:

"abc" -> "bcd" -> ... -> "xyz"

Given a list of strings which contains only lowercase alphabets, group all strings that belong to the same shifting sequence.

For example, given: ["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"],   
Return:

[

["abc","bcd","xyz"],

["az","ba"],

["acef"],

["a","z"]

]

**Note:** For the return value, each inner list's elements must follow the lexicographic order.

**Solution 1 – Time O (?) Space O (?)**

public class Solution {

public List<List<String>> groupStrings(String[] strings) {

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

Map<String, List<String>> map = new HashMap<String, List<String>>();

for (String str : strings) {

int offset = str.charAt(0) - 'a';

String key = "";

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

char c = (char) (str.charAt(i) - offset);

if (c < 'a') {

c += 26;

}

key += c;

}

if (!map.containsKey(key)) {

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

map.put(key, list);

}

map.get(key).add(str);

}

for (String key : map.keySet()) {

List<String> list = map.get(key);

Collections.sort(list);

result.add(list);

}

return result;

}

}

### Count Univalue Subtrees

Given a binary tree, count the number of uni-value subtrees.

A Uni-value subtree means all nodes of the subtree have the same value.

For example:  
Given binary tree,

5

/ \

1 5

/ \ \

5 5 5

return 4.

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public int countUnivalSubtrees(TreeNode root) {

int[] count = new int[1];

helper(root, count);

return count[0];

}

private boolean helper(TreeNode node, int[] count) {

if (node == null) {

return true;

}

boolean left = helper(node.left, count);

boolean right = helper(node.right, count);

if (left && right) {

if (node.left != null && node.val != node.left.val) {

return false;

}

if (node.right != null && node.val != node.right.val) {

return false;

}

count[0]++;

return true;

}

return false;

}

}

### Flatten 2D Vector

Implement an iterator to flatten a 2d vector.

For example, Given 2d vector =

[

[1,2],

[3],

[4,5,6]

]

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,2,3,4,5,6].

**Solution 1 – Time O (1) Space O (1)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Flatten2DVector {

private Iterator<List<Integer>> i;

private Iterator<Integer> j;

public Flatten2DVector(List<List<Integer>> vec2d) {

i = vec2d.iterator();

}

public int next() {

hasNext();

return j.next();

}

public boolean hasNext() {

while ((j == null || !j.hasNext()) && i.hasNext()) {

j = i.next().iterator();

}

return j != null && j.hasNext();

}

}

### Meeting Rooms

Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] (si < ei), determine if a person could attend all meetings.

For example, Given [[0, 30],[5, 10],[15, 20]], return false.

**Solution 1 – Time O (n log n) Space O (1)**

public class Solution {

public boolean canAttendMeetings(Interval[] intervals) {

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

return false;

}

// Sort the intervals by start time

Arrays.sort(intervals, new Comparator<Interval>() {

public int compare(Interval a, Interval b) {

return a.start - b.start;

}

});

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

if (intervals[i].start < intervals[i - 1].end) {

return false;

}

return true;

}

}

### Meeting Rooms II

Given an array of meeting time intervals consisting of start and end times [[s1,e1],[s2,e2],...] (si < ei), find the minimum number of conference rooms required.

For example, Given [[0, 30],[5, 10],[15, 20]], return 2.

**Solution 1 – Time O (n log n) Space O (1)**

public class Solution {

public int minMeetingRooms(Interval[] intervals) {

int[] starts = new int[intervals.length];

int[] ends = new int[intervals.length];

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

starts[i] = intervals[i].start;

ends[i] = intervals[i].end;

}

Arrays.sort(starts);

Arrays.sort(ends);

int rooms = 0;

int endsItr = 0;

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

if (starts[i] < ends[endsItr]) {

rooms++;

} else {

endsItr++;

}

}

return rooms;

}

}

**Solution 2 – Time O (n log n) Space O (1)**

public class Solution {

public int minMeetingRooms(Interval[] intervals) {

if (intervals == null || intervals.length == 0)

return 0;

Arrays.sort(intervals, new Comparator<Interval>() {

public int compare(Interval i1, Interval i2) {

return i1.start - i2.start;

}

});

PriorityQueue<Integer> endTimes = new PriorityQueue<Integer>();

endTimes.offer(intervals[0].end);

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

if (intervals[i].start >= endTimes.peek()) {

endTimes.poll();

}

endTimes.offer(intervals[i].end);

}

return endTimes.size();

}

}

### Factor Combinations

Numbers can be regarded as product of its factors. For example,

8 = 2 x 2 x 2;

= 2 x 4.

Write a function that takes an integer *n* and return all possible combinations of its factors.

**Note:**

1. Each combination's factors must be sorted ascending, for example: The factors of 2 and 6 is [2, 6], not [6, 2].
2. You may assume that *n* is always positive.
3. Factors should be greater than 1 and less than *n*.

**Examples:**  
input: 1  
output:

[]

input: 37  
output: 

[]

input: 12  
output:

[

[2, 6],

[2, 2, 3],

[3, 4]

]

input: 32  
output:

[

[2, 16],

[2, 2, 8],

[2, 2, 2, 4],

[2, 2, 2, 2, 2],

[2, 4, 4],

[4, 8]

]

**Solution 1 – Time O (NP) Space O (NP)**

public class Solution {

public List<List<Integer>> getFactors(int n) {

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

helper(result, new ArrayList<Integer>(), n, 2);

return result;

}

public void helper(List<List<Integer>> result, List<Integer> item, int n,

int start) {

if (n <= 1) {

if (item.size() > 1) {

result.add(new ArrayList<Integer>(item));

}

return;

}

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

if (n % i == 0) {

item.add(i);

helper(result, item, n / i, i);

item.remove(item.size() - 1);

}

}

}

}

### Verify Preorder Sequence in Binary Search Tree

Given an array of numbers, verify whether it is the correct preorder traversal sequence of a binary search tree.

You may assume each number in the sequence is unique.

Follow up: Could you do it using only constant space complexity?

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public boolean verifyPreorder(int[] preorder) {

int low = Integer.MIN\_VALUE, i = -1;

for (int p : preorder) {

if (p < low)

return false;

while (i >= 0 && p > preorder[i])

low = preorder[i--];

preorder[++i] = p;

}

return true;

}

}

**Solution 2 – Time O (n) Space O (n)**

public class Solution {

public boolean verifyPreorder(int[] preorder) {

int low = Integer.MIN\_VALUE;

Deque<Integer> path = new LinkedList<Integer>();

for (int p : preorder) {

if (p < low)

return false;

while (!path.isEmpty() && p > path.peek())

low = path.pop();

path.push(p);

}

return true;

}

}

### Paint House

There are a row of n houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a n x 3 cost matrix. For example, costs0 is the cost of painting house 0 with color red; costs1 is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

Note: All costs are positive integers.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int minCost(int[][] costs) {

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

return 0;

}

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

costs[i][0] += Math.min(costs[i - 1][1], costs[i - 1][2]);

costs[i][1] += Math.min(costs[i - 1][0], costs[i - 1][2]);

costs[i][2] += Math.min(costs[i - 1][1], costs[i - 1][0]);

}

int n = costs.length - 1;

return Math.min(Math.min(costs[n][0], costs[n][1]), costs[n][2]);

}

}

### Binary Tree Paths

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"]

Solution 1 – Time O (n) Space O (n)

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

public class Solution {

public List<String> binaryTreePaths(TreeNode root) {

List<String> ret = new LinkedList<String>();

if (root == null) {

return ret;

}

if (root.left == null && root.right == null) {

ret.add(Integer.toString(root.val));

return ret;

}

if (root.left != null) {

List<String> left = binaryTreePaths(root.left);

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

String str = root.val + "->" + left.get(i);

ret.add(str);

}

}

if (root.right != null) {

List<String> right = binaryTreePaths(root.right);

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

String str = root.val + "->" + right.get(i);

ret.add(str);

}

}

return ret;

}

}

### Add Digits

Given a non-negative integer num, repeatedly add all its digits until the result has only one digit.

For example:

Given num = 38, the process is like: 3 + 8 = 11, 1 + 1 = 2. Since 2 has only one digit, return it.

**Follow up:**  
Could you do it without any loop/recursion in O(1) runtime?

**Hint:**

1. A naive implementation of the above process is trivial. Could you come up with other methods?
2. What are all the possible results?
3. How do they occur, periodically or randomly?
4. You may find this [Wikipedia article](https://en.wikipedia.org/wiki/Digital_root) useful.

**Solution 1 – Time O (1) Space O (1)**

public class Solution {

public int addDigits(int num) {

return 1 + ((num - 1) % 9);

}

}

**Solution 2 – Time O (1) Space O (1)**

public int addDigits(int num) {

if (num == 0) {

return 0;

}

if (num % 9 == 0) {

return 9;

}

else {

return num % 9;

}

}

**Solution 3 – Time O (n^2) Space O (1)**

public class Solution {

public int addDigits(int num) {

while (num >= 10) {

int ret = 0;

while (num != 0) {

ret += num % 10;

num /= 10;

}

num = ret;

}

return num;

}

}

### 3Sum Smaller

Given an array of *n* integers *nums* and a *target*, find the number of index triplets i, j, k with 0 <= i < j < k < n that satisfy the condition nums[i] + nums[j] + nums[k] < target.

For example, given *nums* = [-2, 0, 1, 3], and *target* = 2.

Return 2. Because there are two triplets which sums are less than 2:

[-2, 0, 1]

[-2, 0, 3]

**Solution 1 – Time O (n ^ 2) Space O (1)**

public class Solution {

public int threeSumSmaller(int[] nums, int target) {

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

return 0;

}

Arrays.sort(nums);

int count = 0;

for (int i = 0; i < nums.length - 2; i++) {

int left = i + 1;

int right = nums.length - 1;

while (left < right) {

if (nums[i] + nums[left] + nums[right] < target) {

count += right - left;

left++;

}

if (nums[i] + nums[left] + nums[right] >= target) {

right--;

}

}

}

return count;

}

}

### Single Number III

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?

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

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

int xor = 0;

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

xor = xor ^ nums[i];

}

// find the right most 1

xor = xor & (~(xor - 1));

// separate into two different groups

int[] ret = new int[2];

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

if ((nums[i] & xor) == 0) {

ret[0] ^= nums[i];

} else {

ret[1] ^= nums[i];

}

}

return ret;

}

}

### Graph Valid Tree

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

For example:

Given n = 5 and edges = [[0, 1], [0, 2], [0, 3], [1, 4]], return true.

Given n = 5 and edges = [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]], return false.

Hint:

Given n = 5 and edges = [[0, 1], [1, 2], [3, 4]], what should your return? Is this case a valid tree? Show More Hint Note: 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.

<http://www.geeksforgeeks.org/union-find/>

**Solution 1 – Time O (N \* M) Space O (N)**

public class Solution {

public boolean validTree(int n, int[][] edges) {

// initialize n isolated islands

int[] nums = new int[n];

Arrays.fill(nums, -1);

// perform union find

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

int x = find(nums, edges[i][0]);

int y = find(nums, edges[i][1]);

// if two vertices happen to be in the same set

// then there's a cycle

if (x == y)

return false;

// union

nums[x] = y;

}

return edges.length == n - 1;

}

int find(int nums[], int i) {

if (nums[i] == -1)

return i;

return find(nums, nums[i]);

}

}

### Ugly Number

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.

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public boolean isUgly(int num) {

if (num < 1) {

return false;

}

while (num % 2 == 0) {

num = num / 2;

}

while (num % 3 == 0) {

num = num / 3;

}

while (num % 5 == 0) {

num = num / 5;

}

return num == 1;

}

}

### Ugly Number II

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).

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public int nthUglyNumber(int n) {

int[] ret = new int[n];

ret[0] = 1;

int p2 = 0, p3 = 0, p5 = 0;

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

ret[i] = Math.min(Math.min(ret[p2] \* 2, ret[p3] \* 3), ret[p5] \* 5);

// Be careful about the cases such as 6, in which we need to forward

// both pointers of 2 and 3.

if (ret[i] == ret[p2] \* 2) {

p2++;

}

if (ret[i] == ret[p3] \* 3) {

p3++;

}

if (ret[i] == ret[p5] \* 5) {

p5++;

}

}

return ret[n - 1];

}

}

**Solution 2 – Time O (n) Space O (n)**

// Use queue, but it may exceed integer range, so have to use long type

public class Solution {

public int nthUglyNumber(int n) {

long ret = 0;

if (n == 1) {

return 1;

}

Queue<Long> q2 = new LinkedList<Long>();

Queue<Long> q3 = new LinkedList<Long>();

Queue<Long> q5 = new LinkedList<Long>();

q2.offer(2L);

q3.offer(3L);

q5.offer(5L);

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

long v2 = q2.peek();

long v3 = q3.peek();

long v5 = q5.peek();

if (v2 < v3 && v2 < v5) {

ret = q2.poll();

q2.offer(v2 \* 2);

q3.offer(v2 \* 3);

q5.offer(v2 \* 5);

} else if (v3 < v5) {

ret = q3.poll();

q3.offer(v3 \* 3);

q5.offer(v3 \* 5);

} else {

ret = q5.poll();

q5.offer(v5 \* 5);

}

}

return (int) ret;

}

}

### Palindrome Permutation

Given a string, determine if a permutation of the string could form a palindrome.

For example, "code" -> False, "aab" -> True, "carerac" -> True.

Hint:

Consider the palindromes of odd vs even length. What difference do you notice? Count the frequency of each character. If each character occurs even number of times, then it must be a palindrome. How about character which occurs odd number of times?

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

public boolean canPermutePalindrome(String s) {

Set<Character> set = new HashSet<Character>();

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

if (!set.contains(s.charAt(i))) {

set.add(s.charAt(i));

} else {

set.remove(s.charAt(i));

}

}

return set.size() == 0 || set.size() == 1;

}

}

### Palindrome Permutation II

Given a string s, return all the palindromic permutations (without duplicates) of it. Return an empty list if no palindromic permutation could be form.

For example:

Given s = "aabb", return ["abba", "baab"].

Given s = "abc", return [].

**Hint:**

1. If a palindromic permutation exists, we just need to generate the first half of the string.
2. To generate all distinct permutations of a (half of) string, use a similar approach from: [Permutations II](https://leetcode.com/problems/permutations-ii) or[Next Permutation](https://leetcode.com/problems/next-permutation).

**Solution 1 – Time O (NP) Space O (NP)**

public class Solution {

public List<String> generatePalindromes(String s) {

int odd = 0;

String mid = "";

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

List<Character> list = new LinkedList<Character>();

Map<Character, Integer> map = new HashMap<Character, Integer>();

// step 1. build character count map and count odds

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

char c = s.charAt(i);

map.put(c, map.containsKey(c) ? map.get(c) + 1 : 1);

odd += map.get(c) % 2 != 0 ? 1 : -1;

}

// cannot form any palindromic string

if (odd > 1)

return res;

// step 2. add half count of each character to list

for (Map.Entry<Character, Integer> entry : map.entrySet()) {

char key = entry.getKey();

int val = entry.getValue();

if (val % 2 != 0)

mid += key;

for (int i = 0; i < val / 2; i++)

list.add(key);

}

// step 3. generate all the permutations

getPerm(list, mid, new boolean[list.size()], new StringBuilder(), res);

return res;

}

// generate all unique permutation from list

void getPerm(List<Character> list, String mid, boolean[] used,

StringBuilder sb, List<String> res) {

if (sb.length() == list.size()) {

// form the palindromic string

res.add(sb.toString() + mid + sb.reverse().toString());

sb.reverse();

return;

}

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

// avoid duplication

if (i > 0 && list.get(i) == list.get(i - 1) && !used[i - 1])

continue;

if (!used[i]) {

used[i] = true;

sb.append(list.get(i));

// recursion

getPerm(list, mid, used, sb, res);

// backtracking

used[i] = false;

sb.deleteCharAt(sb.length() - 1);

}

}

}

}

### Missing Number

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?

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

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;

}

}

**Solution 2 – Time O (n) Space O (1)**

public class Solution {

public int missingNumber(int[] nums) {

if (nums == null) {

return 0;

}

int ret = 0;

for (int j = 0; j < 32; j++) {

int mask = 1 << j;

int zero = 0;

int one = 0;

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

if (j > 0 && (nums[i] & (0xffffffffL >>> (32 - j))) != ret) {

continue;

}

if ((nums[i] & mask) == 0) {

zero++;

} else {

one++;

}

}

if (zero > one) {

ret |= mask;

}

}

return ret;

}

}

### Closest Binary Search Tree Value

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

Note: Given target value is a floating point. You are guaranteed to have only one unique value in the BST that is closest to the target.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int closestValue(TreeNode root, double target) {

int ret = root.val;

while (root != null) {

if (Math.abs(target - root.val) < Math.abs(target - ret)) {

ret = root.val;

}

root = root.val > target ? root.left : root.right;

}

return ret;

}

}

### Encode and Decode Strings

Design an algorithm to encode a list of strings to a string. The encoded string is then sent over the network and is decoded back to the original list of strings.

Machine 1 (sender) has the function:  
string encode(vector<string> strs) { // ... your code return encoded\_string; }   
Machine 2 (receiver) has the function:   
vector<string> decode(string s) { //... your code return strs; }

So Machine 1 does:  
string encoded\_string = encode(strs);   
and Machine 2 does:  
vector<string> strs2 = decode(encoded\_string);   
strs2 in Machine 2 should be the same as strs in Machine 1.

Implement the encode and decode methods.

Note: The string may contain any possible characters out of 256 valid ascii characters. Your algorithm should be generalized enough to work on any possible characters. Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless. Do not rely on any library method such as eval or serialize methods. You should implement your own encode/decode algorithm.

**Solution 1 – Time O (n) Space O (n)**

public class Solution {

// Encodes a list of strings to a single string.

public String encode(List<String> strs) {

StringBuilder sb = new StringBuilder();

for (String s : strs) {

sb.append(s.length()).append('/').append(s);

}

return sb.toString();

}

// Decodes a single string to a list of strings.

public List<String> decode(String s) {

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

int i = 0;

while (i < s.length()) {

int slash = s.indexOf('/', i);

int size = Integer.valueOf(s.substring(i, slash));

ret.add(s.substring(slash + 1, slash + size + 1));

i = slash + size + 1;

}

return ret;

}

}

### Integer to English Words

Convert a non-negative integer to its english words representation. Given input is guaranteed to be less than 231 - 1.

For example,

123 -> "One Hundred Twenty Three"

12345 -> "Twelve Thousand Three Hundred Forty Five"

1234567 -> "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"

**Hint:**

1. Did you see a pattern in dividing the number into chunk of words? For example, 123 and 123000.
2. Group the number by thousands (3 digits). You can write a helper function that takes a number less than 1000 and convert just that chunk to words.
3. There are many edge cases. What are some good test cases? Does your code work with input such as 0? Or 1000010? (middle chunk is zero and should not be printed out)

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

private final String[] belowTen = new String[] { "", "One", "Two", "Three",

"Four", "Five", "Six", "Seven", "Eight", "Nine" };

private final String[] belowTwenty = new String[] { "Ten", "Eleven",

"Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen",

"Seventeen", "Eighteen", "Nineteen" };

private final String[] belowHundred = new String[] { "", "Ten", "Twenty",

"Thirty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety" };

public String numberToWords(int num) {

if (num == 0) {

return "Zero";

}

return helper(num);

}

private String helper(int num) {

String result = new String();

if (num < 10) {

result = belowTen[num];

} else if (num < 20) {

result = belowTwenty[num - 10];

} else if (num < 100) {

result = belowHundred[num / 10] + " " + helper(num % 10);

} else if (num < 1000) {

result = helper(num / 100) + " Hundred " + helper(num % 100);

} else if (num < 1000000) {

result = helper(num / 1000) + " Thousand " + helper(num % 1000);

} else if (num < 1000000000) {

result = helper(num / 1000000) + " Million "

+ helper(num % 1000000);

} else {

result = helper(num / 1000000000) + " Billion "

+ helper(num % 1000000000);

}

return result.trim();

}

}

**Solution 2 – Time O (n) Space O (1)**

public class Solution {

public String numberToWords(int num) {

StringBuilder ret = new StringBuilder();

if (num == 0) {

return "Zero";

}

int chunk = 0;

if (num >= 1e9) {

chunk = num / 1000000000;

ret.append(myHelper(chunk)).append(" Billion");

num = num % 1000000000;

}

if (num >= 1e6) {

if (ret.length() != 0) {

ret.append(" ");

}

chunk = num / 1000000;

ret.append(myHelper(chunk)).append(" Million");

num = num % 1000000;

}

if (num >= 1e3) {

if (ret.length() != 0) {

ret.append(" ");

}

chunk = num / 1000;

ret.append(myHelper(chunk)).append(" Thousand");

num = num % 1000;

}

if (num > 0) {

if (ret.length() != 0) {

ret.append(" ");

}

chunk = num;

ret.append(myHelper(chunk));

}

return ret.toString();

}

public String myHelper(int chunk) {

StringBuilder ret = new StringBuilder();

if (chunk >= 100) {

ret.append(map(chunk / 100)).append(" Hundred");

chunk = chunk % 100;

}

if (ret.length() != 0 && chunk != 0) {

ret.append(" ");

}

return ret.append(map(chunk)).toString();

}

public String map(int num) {

switch (num) {

case 0:

return "";

case 1:

return "One";

case 2:

return "Two";

case 3:

return "Three";

case 4:

return "Four";

case 5:

return "Five";

case 6:

return "Six";

case 7:

return "Seven";

case 8:

return "Eight";

case 9:

return "Nine";

case 10:

return "Ten";

case 11:

return "Eleven";

case 12:

return "Twelve";

case 13:

return "Thirteen";

case 15:

return "Fifteen";

case 18:

return "Eighteen";

case 14:

case 16:

case 17:

case 19:

return map(num - 10) + "teen";

case 20:

return "Twenty";

case 30:

return "Thirty";

case 40:

return "Forty";

case 50:

return "Fifty";

case 60:

case 70:

case 90:

return myHelper(num / 10) + "ty";

case 80:

return "Eighty";

default:

return map(num - num % 10) + " " + map(num % 10);

}

}

}

### H-Index

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** 3 citations 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.

**Solution 1 – Time O () Space O ()**

public class Solution {

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;

}

}

**Solution 2 – Time O (nlogn) Space O (1)**

public class Solution {

public int hIndex(int[] citations) {

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

return 0;

}

Arrays.sort(citations);

for (int i = citations.length - 1; i >= 0; i--) {

int h = citations.length - i;

if (citations[i] >= h && (i == 0 || citations[i - 1] <= h)) {

return h;

}

}

return 0;

}

}

### H-Index II

**Follow up** for [H-Index](https://leetcode.com/problems/h-index/): What if the citations array is sorted in ascending order? Could you optimize your algorithm?

**Hint:**

1. Expected runtime complexity is in *O*(log *n*) and the input is sorted.

**Solution 1 – Time O (log n) Space O (1)**

public class Solution {

public int hIndex(int[] citations) {

int len = citations.length;

int l = 0;

int r = len - 1;

while (l <= r) {

int m = (l + r) / 2;

if (citations[m] < len - m) {

l = m + 1;

} else {

r = m - 1;

}

}

return len - l;

}

}

### Paint Fence

There is a fence with n posts, each post can be painted with one of the k colors.

You have to paint all the posts such that no more than two adjacent fence posts have the same color.

Return the total number of ways you can paint the fence.

Note: n and k are non-negative integers.

Solution 1- Time O (n) Space O (1)

public class Solution {

public int numWays(int n, int k) {

if (n == 0)

return 0;

else if (n == 1)

return k;

int diffColorCounts = k \* (k - 1);

int sameColorCounts = k;

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

int temp = diffColorCounts;

diffColorCounts = (diffColorCounts + sameColorCounts) \* (k - 1);

sameColorCounts = temp;

}

return diffColorCounts + sameColorCounts;

}

}

### Find the Celebrity

Suppose you are at a party with n people (labeled from 0 to n - 1) and among them, there may exist one celebrity. The definition of a celebrity is that all the other n - 1people know him/her but he/she does not know any of them.

Now you want to find out who the celebrity is or verify that there is not one. The only thing you are allowed to do is to ask questions like: "Hi, A. Do you know B?" to get information of whether A knows B. You need to find out the celebrity (or verify there is not one) by asking as few questions as possible (in the asymptotic sense).

You are given a helper function bool knows(a, b) which tells you whether A knows B. Implement a function int findCelebrity(n), your function should minimize the number of calls to knows.

Note: There will be exactly one celebrity if he/she is in the party. Return the celebrity's label if there is a celebrity in the party. If there is no celebrity, return -1.

**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public int findCelebrity(int n) {

int candidate = 0;

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

if (knows(candidate, i))

candidate = i;

}

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

if (i != candidate && (knows(candidate, i) || !knows(i, candidate)))

return -1;

}

return candidate;

}

private boolean knows(int candidate, int i) {

return false;

}

}

### First Bad Version

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.

**Solution 1 – Time O (log n) Space O (1)**

/\* The isBadVersion API is defined in the parent class VersionControl.

boolean isBadVersion(int version); \*/

public class Solution extends VersionControl {

public int firstBadVersion(int n) {

int l = 1;

int r = n;

while (l <= r) {

int m = l + (r - l) / 2;

if (isBadVersion(m)) {

r = m - 1;

} else {

l = m + 1;

}

}

return l;

}

}

### Perfect Squares

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.

**Solution 1 – Time O (n log n) Space O (n)**

public class Solution {

public int numSquares(int n) {

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

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];

}

}

### Wiggle Sort

Given an unsorted array nums, reorder it in-place such that nums[0] <= nums[1] >= nums[2] <= nums[3]....

For example, given nums = [3, 5, 2, 1, 6, 4], one possible answer is [1, 6, 2, 5, 3, 4].  
**Solution 1 – Time O (n) Space O (1)**

public class Solution {

public void wiggleSort(int[] nums) {

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

if (i % 2 == 1 && (nums[i - 1] > nums[i])) {

swap(nums, i);

} else if (i != 0 && nums[i - 1] < nums[i]) {

swap(nums, i);

}

}

void swap(int[] nums, int i) {

int temp = nums[i];

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

nums[i - 1] = temp;

}

}

### ------------BreakBreak-------------