**Microsoft**

Introduction

Top interview questions asked by Microsoft as voted by the community.

We compiled this list thoroughly so you can save time and get well-prepared for a Microsoft interview.

Completing this card should give you a good idea of the type of questions you would encounter in your Microsoft interview.

Arrays and Strings

 Two Sum

 Valid Palindrome

 String to Integer (atoi)

 Reverse String

 Reverse Words in a String

 Reverse Words in a String II

 Valid Parentheses

 Longest Palindromic Substring

 Group Anagrams

 Trapping Rain Water

 Set Matrix Zeroes

 Rotate Image

 Spiral Matrix

Linked Lists

 Reverse Linked List

 Linked List Cycle

 Add Two Numbers

 Add Two Numbers II

 Merge Two Sorted Lists

 Merge k Sorted Lists

 Intersection of Two Linked Lists

 Copy List with Random Pointer

Trees and Graphs

 Validate Binary Search Tree

 Binary Tree Inorder Traversal

 Binary Tree Level Order Traversal

 Binary Tree Zigzag Level Order Traversal

 Populating Next Right Pointers in Each Node

 Populating Next Right Pointers in Each Node II

 Lowest Common Ancestor of a Binary Search Tree

 Lowest Common Ancestor of a Binary Tree

 Construct Binary Tree from Preorder and Inorder Traversal

 Number of Islands

 Clone Graph

Backtracking

 Letter Combinations of a Phone Number

 Word Search II

 Wildcard Matching

 Regular Expression Matching

Sorting and Searching

 Remove Duplicates from Sorted Array

 Merge Sorted Array

 Sort Colors

 Find Minimum in Rotated Sorted Array

 Find Minimum in Rotated Sorted Array II

 Search in Rotated Sorted Array

 Search a 2D Matrix

 Search a 2D Matrix II

 Median of Two Sorted Arrays

Dynamic Programming

 Best Time to Buy and Sell Stock

 Maximum Subarray

 Longest Increasing Subsequence

Design

 Serialize and Deserialize BST

 Serialize and Deserialize Binary Tree

 Implement Trie (Prefix Tree)

 LRU Cache

Others

 Single Number

 Roman to Integer

 Excel Sheet Column Number

 Find the Celebrity

 Integer to English Words

 The Skyline Problem

Arrays and Strings

Microsoft likes to ask simple array questions. Reverse Words in a String is one if the popular questions. Get warmed up by practicing Reverse String first.

**Two Sum**

Given an array of integers nums and an integer target, return *indices of the two numbers such that they add up to target*.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

**Example 1:**

**Input:** nums = [2,7,11,15], target = 9

**Output:** [0,1]

**Output:** Because nums[0] + nums[1] == 9, we return [0, 1].

**Example 2:**

**Input:** nums = [3,2,4], target = 6

**Output:** [1,2]

**Example 3:**

**Input:** nums = [3,3], target = 6

**Output:** [0,1]

**Constraints:**

* 2 <= nums.length <= 103
* -109 <= nums[i] <= 109
* -109 <= target <= 109
* **Only one valid answer exists.**

Solution

Approach 1: Brute Force

The brute force approach is simple. Loop through each element x*x* and find if there is another value that equals to target - x*target*−*x*.

|  |
| --- |
| public int[] twoSum(int[] nums, int target) {  for (int i = 0; i < nums.length; i++) {  for (int j = i + 1; j < nums.length; j++) {  if (nums[j] == target - nums[i]) {  return new int[] { i, j };  }  }  }  throw new IllegalArgumentException("No two sum solution");  } |

**Complexity Analysis**

* Time complexity : O(n^2)*O*(*n*2). For each element, we try to find its complement by looping through the rest of array which takes O(n)*O*(*n*) time. Therefore, the time complexity is O(n^2)*O*(*n*2).
* Space complexity : O(1)*O*(1).

#### **Approach 2: Two-pass Hash Table**

To improve our run time complexity, we need a more efficient way to check if the complement exists in the array. If the complement exists, we need to look up its index. What is the best way to maintain a mapping of each element in the array to its index? A hash table.

We reduce the look up time from O(n)*O*(*n*) to O(1)*O*(1) by trading space for speed. A hash table is built exactly for this purpose, it supports fast look up in near constant time. I say "near" because if a collision occurred, a look up could degenerate to O(n)*O*(*n*) time. But look up in hash table should be amortized O(1)*O*(1) time as long as the hash function was chosen carefully.

A simple implementation uses two iterations. In the first iteration, we add each element's value and its index to the table. Then, in the second iteration we check if each element's complement (target - nums[i]*target*−*nums*[*i*]) exists in the table. Beware that the complement must not be nums[i]*nums*[*i*] itself!

|  |
| --- |
| public int[] twoSum(int[] nums, int target) {  Map<Integer, Integer> map = new HashMap<>();  for (int i = 0; i < nums.length; i++) {  map.put(nums[i], i);  }  for (int i = 0; i < nums.length; i++) {  int complement = target - nums[i];  if (map.containsKey(complement) && map.get(complement) != i) {  return new int[] { i, map.get(complement) };  }  }  throw new IllegalArgumentException("No two sum solution");  } |

**Complexity Analysis:**

* Time complexity : O(n)*O*(*n*). We traverse the list containing n*n* elements exactly twice. Since the hash table reduces the look up time to O(1)*O*(1), the time complexity is O(n)*O*(*n*).
* Space complexity : O(n)*O*(*n*). The extra space required depends on the number of items stored in the hash table, which stores exactly n*n* elements.

#### **Approach 3: One-pass Hash Table**

It turns out we can do it in one-pass. While we iterate and inserting elements into the table, we also look back to check if current element's complement already exists in the table. If it exists, we have found a solution and return immediately.

|  |
| --- |
| public int[] twoSum(int[] nums, int target) {  Map<Integer, Integer> map = new HashMap<>();  for (int i = 0; i < nums.length; i++) {  int complement = target - nums[i];  if (map.containsKey(complement)) {  return new int[] { map.get(complement), i };  }  map.put(nums[i], i);  }  throw new IllegalArgumentException("No two sum solution");  } |

**Complexity Analysis:**

* Time complexity : O(n)*O*(*n*). We traverse the list containing n*n* elements only once. Each look up in the table costs only O(1)*O*(1) time.
* Space complexity : O(n)*O*(*n*). The extra space required depends on the number of items stored in the hash table, which stores at most n*n* elements.

**Valid Palindrome**

Given a string s, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

**Example 1:**

**Input:** s = "A man, a plan, a canal: Panama"

**Output:** true

**Explanation:** "amanaplanacanalpanama" is a palindrome.

**Example 2:**

**Input:** s = "race a car"

**Output:** false

**Explanation:** "raceacar" is not a palindrome.

**Constraints:**

* 1 <= s.length <= 2 \* 105
* s consists only of printable ASCII characters.

FB

**String to Integer (atoi)**

Implement the myAtoi(string s) function, which converts a string to a 32-bit signed integer (similar to C/C++'s atoi function).

The algorithm for myAtoi(string s) is as follows:

1. Read in and ignore any leading whitespace.
2. Check if the next character (if not already at the end of the string) is '-' or '+'. Read this character in if it is either. This determines if the final result is negative or positive respectively. Assume the result is positive if neither is present.
3. Read in next the characters until the next non-digit charcter or the end of the input is reached. The rest of the string is ignored.
4. Convert these digits into an integer (i.e. "123" -> 123, "0032" -> 32). If no digits were read, then the integer is 0. Change the sign as necessary (from step 2).
5. If the integer is out of the 32-bit signed integer range [-231, 231 - 1], then clamp the integer so that it remains in the range. Specifically, integers less than -231 should be clamped to -231, and integers greater than 231 - 1 should be clamped to 231 - 1.
6. Return the integer as the final result.

**Note:**

* Only the space character ' ' is considered a whitespace character.
* **Do not ignore** any characters other than the leading whitespace or the rest of the string after the digits.

**Example 1:**

**Input:** s = "42"

**Output:** 42

**Explanation:** The underlined characters are what is read in, the caret is the current reader position.

Step 1: "42" (no characters read because there is no leading whitespace)

^

Step 2: "42" (no characters read because there is neither a '-' nor '+')

^

Step 3: "42" ("42" is read in)

^

The parsed integer is 42.

Since 42 is in the range [-231, 231 - 1], the final result is 42.

**Example 2:**

**Input:** s = " -42"

**Output:** -42

**Explanation:**

Step 1: " -42" (leading whitespace is read and ignored)

^

Step 2: " -42" ('-' is read, so the result should be negative)

^

Step 3: " -42" ("42" is read in)

^

The parsed integer is -42.

Since -42 is in the range [-231, 231 - 1], the final result is -42.

**Example 3:**

**Input:** s = "4193 with words"

**Output:** 4193

**Explanation:**

Step 1: "4193 with words" (no characters read because there is no leading whitespace)

^

Step 2: "4193 with words" (no characters read because there is neither a '-' nor '+')

^

Step 3: "4193 with words" ("4193" is read in; reading stops because the next character is a non-digit)

^

The parsed integer is 4193.

Since 4193 is in the range [-231, 231 - 1], the final result is 4193.

**Example 4:**

**Input:** s = "words and 987"

**Output:** 0

**Explanation:**

Step 1: "words and 987" (no characters read because there is no leading whitespace)

^

Step 2: "words and 987" (no characters read because there is neither a '-' nor '+')

^

Step 3: "words and 987" (reading stops immediately because there is a non-digit 'w')

^

The parsed integer is 0 because no digits were read.

Since 0 is in the range [-231, 231 - 1], the final result is 0.

**Example 5:**

**Input:** s = "-91283472332"

**Output:** -2147483648

**Explanation:**

Step 1: "-91283472332" (no characters read because there is no leading whitespace)

^

Step 2: "-91283472332" ('-' is read, so the result should be negative)

^

Step 3: "-91283472332" ("91283472332" is read in)

^

The parsed integer is -91283472332.

Since -91283472332 is less than the lower bound of the range [-231, 231 - 1], the final result is clamped to -231 = -2147483648.

**Constraints:**

* 0 <= s.length <= 200
* s consists of English letters (lower-case and upper-case), digits (0-9), ' ', '+', '-', and '.'.

EZ FB

**Reverse String**

Write a function that reverses a string. The input string is given as an array of characters char[].

Do not allocate extra space for another array, you must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

You may assume all the characters consist of [printable ascii characters](https://en.wikipedia.org/wiki/ASCII#Printable_characters).

**Example 1:**

**Input:** ["h","e","l","l","o"]

**Output:** ["o","l","l","e","h"]

**Example 2:**

**Input:** ["H","a","n","n","a","h"]

**Output:** ["h","a","n","n","a","H"]

EZ

**Reverse Words in a String**

Given an input string s, reverse the order of the **words**.

A **word** is defined as a sequence of non-space characters. The **words** in s will be separated by at least one space.

Return a string of the words in reverse order concatenated by a single space.

**Note** that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

**Example 1:**

**Input:** s = "the sky is blue"

**Output:** "blue is sky the"

**Example 2:**

**Input:** s = " hello world "

**Output:** "world hello"

**Explanation:** Your reversed string should not contain leading or trailing spaces.

**Example 3:**

**Input:** s = "a good example"

**Output:** "example good a"

**Explanation:** You need to reduce multiple spaces between two words to a single space in the reversed string.

**Example 4:**

**Input:** s = " Bob Loves Alice "

**Output:** "Alice Loves Bob"

**Example 5:**

**Input:** s = "Alice does not even like bob"

**Output:** "bob like even not does Alice"

**Constraints:**

* 1 <= s.length <= 104
* s contains English letters (upper-case and lower-case), digits, and spaces ' '.
* There is **at least one** word in s.

**Follow up:**

* Could you solve it **in-place** with O(1) extra space?

Array

**Reverse Words in a String II**

Given an input string, reverse the string word by word.

**Example:**

**Input:** ["t","h","e"," ","s","k","y"," ","i","s"," ","b","l","u","e"]

**Output:** ["b","l","u","e"," ","i","s"," ","s","k","y"," ","t","h","e"]

**Note:**

* A word is defined as a sequence of non-space characters.
* The input string does not contain leading or trailing spaces.
* The words are always separated by a single space.

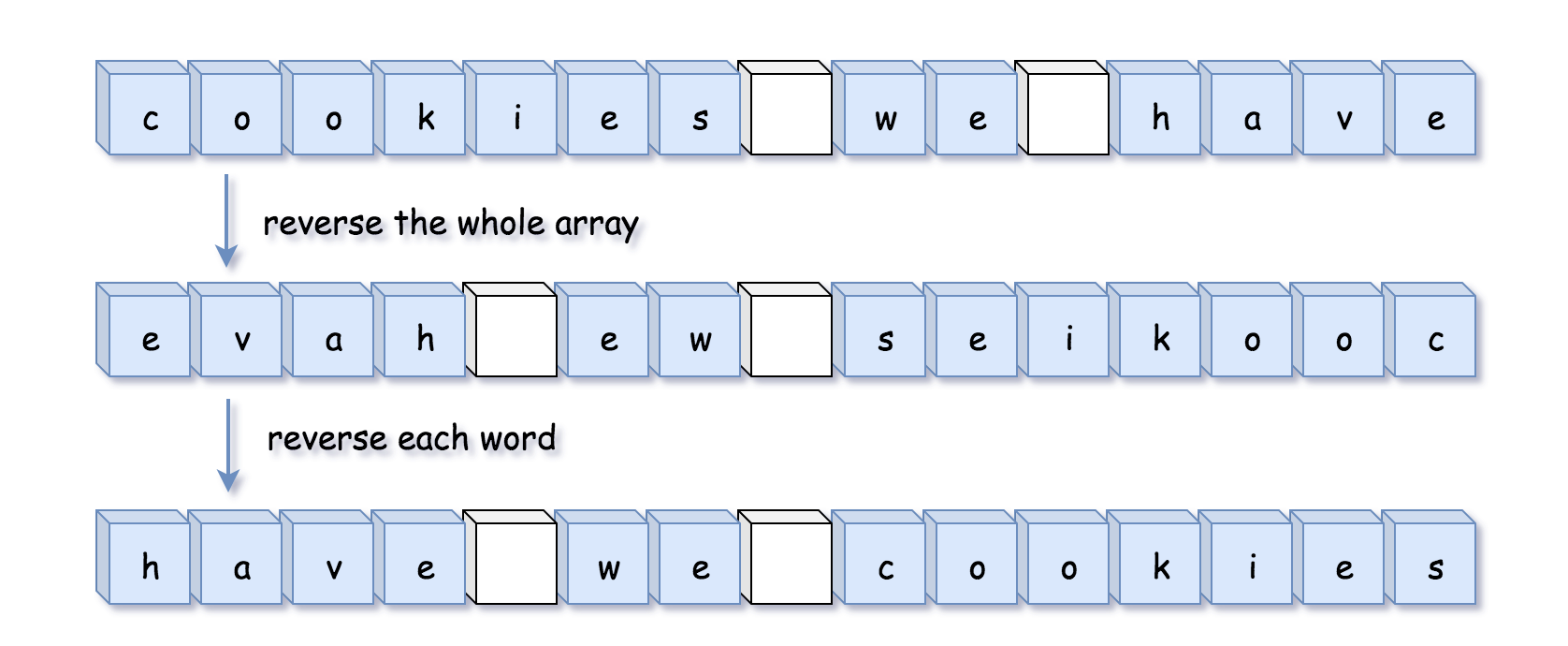
**Follow up:**Could you do it *in-place* without allocating extra space?

## Solution

#### **Approach 1: Reverse the Whole String and Then Reverse Each Word**

To have this problem in Amazon interview is a good situation, since input is a mutable structure and hence one could aim \mathcal{O}(1)O(1) space solution without any technical difficulties.

The idea is simple: reverse the whole string and then reverse each word.



**Algorithm**

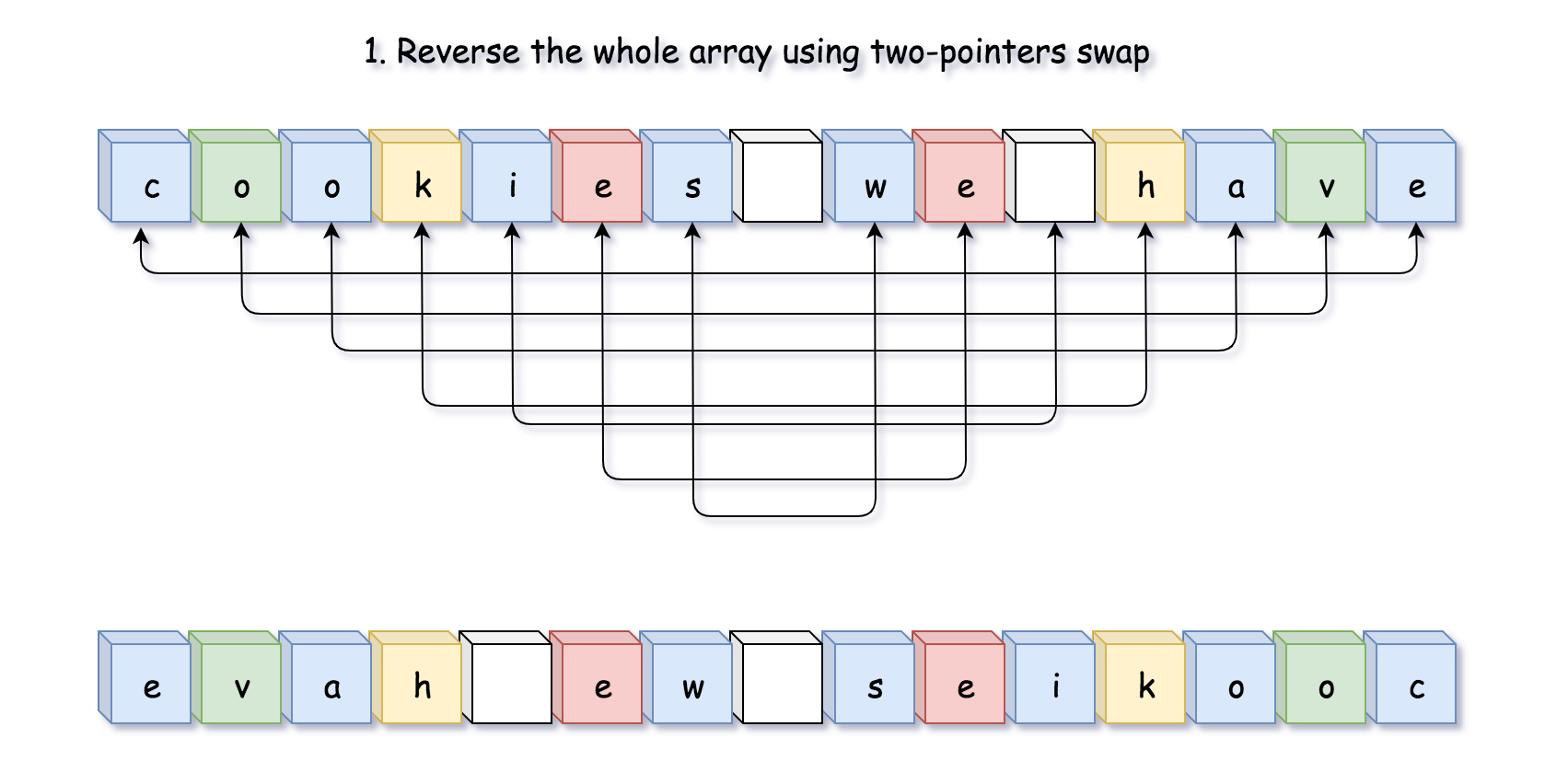
Let's first implement two functions:

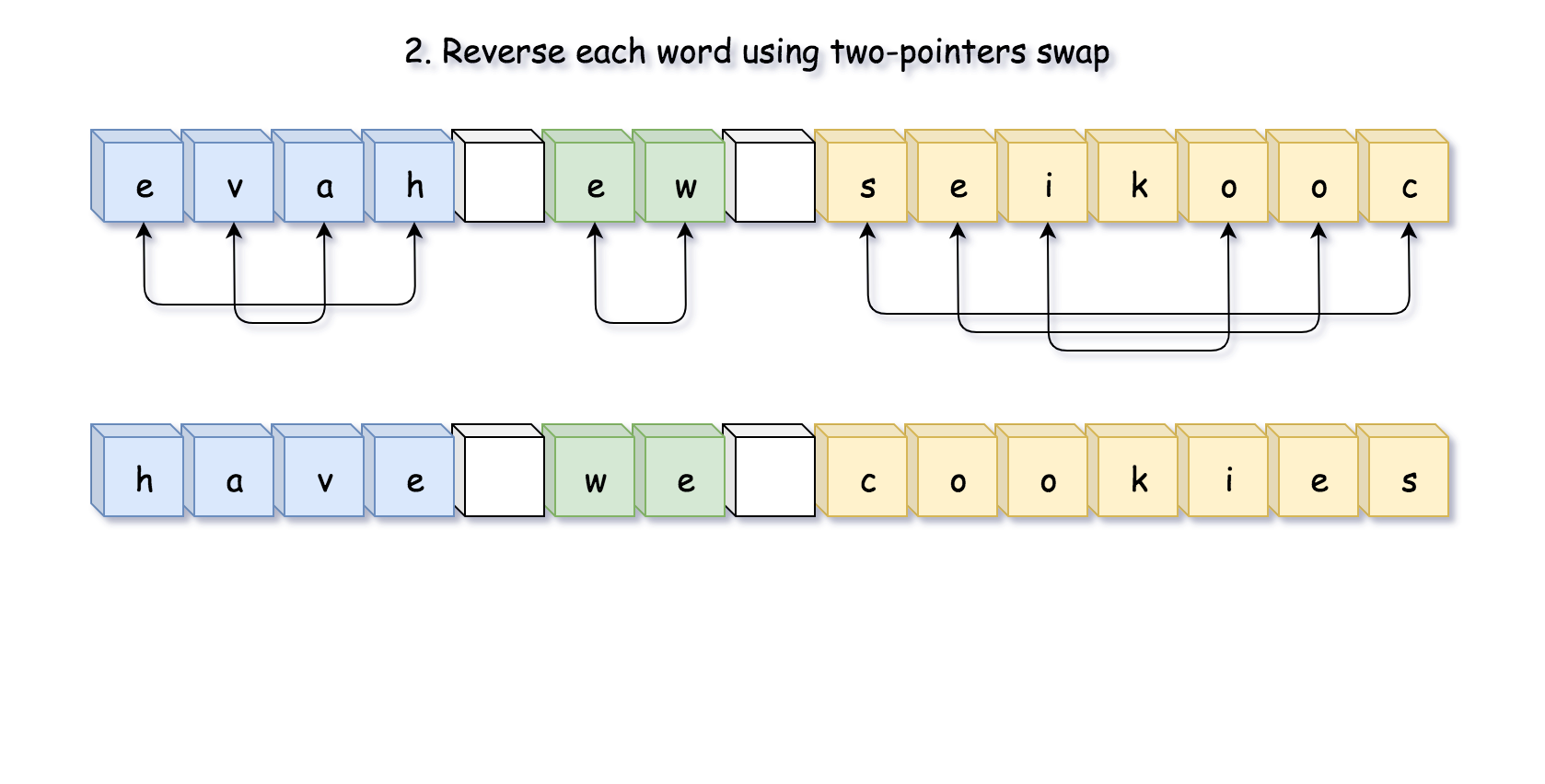
* reverse(l: list, left: int, right: int), which reverses array characters between left and right pointers. C++ users could directly use built-in std::reverse.
* reverse\_each\_word(l: list), which uses two pointers to mark the boundaries of each word and previous function to reverse it.

Now reverseWords(s: List[str]) implementation is straightforward:

* Reverse the whole string: reverse(s, 0, len(s) - 1).
* Reverse each word: reverse\_each\_word(s).

**Implementation**





|  |
| --- |
| class Solution {  public void reverse(char[] s, int left, int right) {  while (left < right) {  char tmp = s[left];  s[left++] = s[right];  s[right--] = tmp;  }  }  public void reverseEachWord(char[] s) {  int n = s.length;  int start = 0, end = 0;  while (start < n) {  // go to the end of the word  while (end < n && s[end] != ' ') ++end;  // reverse the word  reverse(s, start, end - 1);  // move to the next word  start = end + 1;  ++end;  }  }  public void reverseWords(char[] s) {  // reverse the whole string  reverse(s, 0, s.length - 1);  // reverse each word  reverseEachWord(s);  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*), it's two passes along the string.
* Space complexity: \mathcal{O}(1)O(1), it's a constant space solution.

**Valid Parentheses**

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid. An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

**Example 1:**

**Input:** s = "()"

**Output:** true

**Example 2:**

**Input:** s = "()[]{}"

**Output:** true

**Example 3:**

**Input:** s = "(]"

**Output:** false

**Example 4:**

**Input:** s = "([)]"

**Output:** false

**Example 5:**

**Input:** s = "{[]}"

**Output:** true

**Constraints:**

* 1 <= s.length <= 104
* s consists of parentheses only '()[]{}'.

Queue Stack

**Longest Palindromic Substring**

Given a string s, return *the longest palindromic substring* in s.

**Example 1:**

**Input:** s = "babad"

**Output:** "bab"

**Note:** "aba" is also a valid answer.

**Example 2:**

**Input:** s = "cbbd"

**Output:** "bb"

**Example 3:**

**Input:** s = "a"

**Output:** "a"

**Example 4:**

**Input:** s = "ac"

**Output:** "a"

**Constraints:**

* 1 <= s.length <= 1000
* s consist of only digits and English letters (lower-case and/or upper-case),

Medium

**Group Anagrams**

**Solution**

Given an array of strings strs, group **the anagrams** together. You can return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Example 1:**

**Input:** strs = ["eat","tea","tan","ate","nat","bat"]

**Output:** [["bat"],["nat","tan"],["ate","eat","tea"]]

**Example 2:**

**Input:** strs = [""]

**Output:** [[""]]

**Example 3:**

**Input:** strs = ["a"]

**Output:** [["a"]]

**Constraints:**

* 1 <= strs.length <= 104
* 0 <= strs[i].length <= 100
* strs[i] consists of lower-case English letters.

Medium

**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 can trap after raining.

**Example 1:**



**Input:** height = [0,1,0,2,1,0,1,3,2,1,2,1]

**Output:** 6

**Explanation:** The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

**Example 2:**

**Input:** height = [4,2,0,3,2,5]

**Output:** 9

**Constraints:**

* n == height.length
* 0 <= n <= 3 \* 104
* 0 <= height[i] <= 105

Hard

**Set Matrix Zeroes**

Given an *m* x *n* matrix. If an element is **0**, set its entire row and column to **0**. Do it [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm).

**Follow up:**

* 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?

**Example 1:**



**Input:** matrix = [[1,1,1],[1,0,1],[1,1,1]]

**Output:** [[1,0,1],[0,0,0],[1,0,1]]

**Example 2:**



**Input:** matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]

**Output:** [[0,0,0,0],[0,4,5,0],[0,3,1,0]]

**Constraints:**

* m == matrix.length
* n == matrix[0].length
* 1 <= m, n <= 200
* -231 <= matrix[i][j] <= 231 - 1

Medium

**Rotate Image**

You are given an *n* x *n* 2D matrix representing an image, rotate the image by 90 degrees (clockwise).

You have to rotate the image [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm), which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

**Example 1:**



**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

**Output:** [[7,4,1],[8,5,2],[9,6,3]]

**Example 2:**



**Input:** matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]

**Output:** [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

**Example 3:**

**Input:** matrix = [[1]]

**Output:** [[1]]

**Example 4:**

**Input:** matrix = [[1,2],[3,4]]

**Output:** [[3,1],[4,2]]

**Constraints:**

* matrix.length == n
* matrix[i].length == n
* 1 <= n <= 20
* -1000 <= matrix[i][j] <= 1000

EZ Google Amazon

**Spiral Matrix**

Given an m x n matrix, return *all elements of the* matrix *in spiral order*.

**Example 1:**



**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

**Output:** [1,2,3,6,9,8,7,4,5]

**Example 2:**



**Input:** matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]

**Output:** [1,2,3,4,8,12,11,10,9,5,6,7]

**Constraints:**

* m == matrix.length
* n == matrix[i].length
* 1 <= m, n <= 10
* -100 <= matrix[i][j] <= 100

Hard

## Linked Lists

These are some popular Linked List questions being asked by Microsoft. We recommend Reverse Linked List, Linked List Cycle, Merge k Sorted Lists, Intersection of Two Linked Lists and Copy List with Random Pointers.

**Reverse Linked List**

Given the head of a singly linked list, reverse the list, and return *the reversed list*.

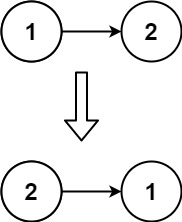
**Example 1:**



**Input:** head = [1,2,3,4,5]

**Output:** [5,4,3,2,1]

**Example 2:**



**Input:** head = [1,2]

**Output:** [2,1]

**Example 3:**

**Input:** head = []

**Output:** []

**Constraints:**

* The number of nodes in the list is the range [0, 5000].
* -5000 <= Node.val <= 5000

**Follow up:** A linked list can be reversed either iteratively or recursively. Could you implement both?

EZ MS

**Linked List Cycle**

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter**.

Return true*if there is a cycle in the linked list*. Otherwise, return false.

**Example 1:**



**Input:** head = [3,2,0,-4], pos = 1

**Output:** true

**Explanation:** There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

**Example 2:**



**Input:** head = [1,2], pos = 0

**Output:** true

**Explanation:** There is a cycle in the linked list, where the tail connects to the 0th node.

**Example 3:**



**Input:** head = [1], pos = -1

**Output:** false

**Explanation:** There is no cycle in the linked list.

**Constraints:**

* The number of the nodes in the list is in the range [0, 104].
* -105 <= Node.val <= 105
* pos is -1 or a **valid index** in the linked-list.

**Follow up:** Can you solve it using O(1) (i.e. constant) memory?

LinkedList EZ

**Add Two Numbers**

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order**, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

**Example 1:**



**Input:** l1 = [2,4,3], l2 = [5,6,4]

**Output:** [7,0,8]

**Explanation:** 342 + 465 = 807.

**Example 2:**

**Input:** l1 = [0], l2 = [0]

**Output:** [0]

**Example 3:**

**Input:** l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]

**Output:** [8,9,9,9,0,0,0,1]

**Constraints:**

* The number of nodes in each linked list is in the range [1, 100].
* 0 <= Node.val <= 9
* It is guaranteed that the list represents a number that does not have leading zeros.

LinkedList

**Add Two Numbers II**

You are given two **non-empty** linked lists representing two non-negative integers. The most significant digit comes first and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

**Follow up:**  
What if you cannot modify the input lists? In other words, reversing the lists is not allowed.

**Example:**

**Input:** (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)

**Output:** 7 -> 8 -> 0 -> 7

Solution

Overview

**Prerequisites**

The problem is a combination of three basic problems:

* [Reverse Linked List](https://leetcode.com/problems/reverse-linked-list/solution/).
* [Add Strings](https://leetcode.com/problems/add-strings/solution/) - the good problem to refresh textbook digit-by-digit addition algorithm.
* [Add Two Numbers](https://leetcode.com/problems/add-two-numbers/) - the same problem as the current one, but the digits are stored in *reverse order*.

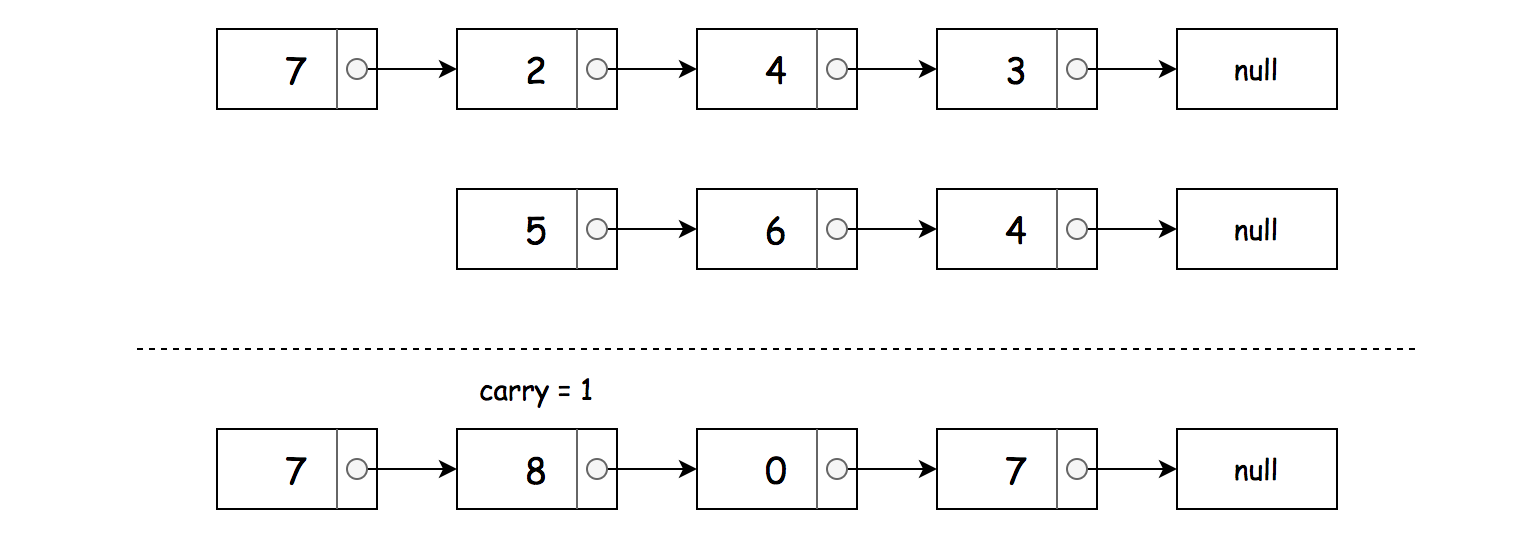
**Time and Space Complexity To Target**

Each list should be parsed at least once, hence the best time complexity we could have is \mathcal{O}(N\_1 + N\_2)O(*N*1​+*N*2​), where N\_1*N*1​ and N\_2*N*2​ are the numbers of elements in the lists.

Space complexity is more interesting. It's relatively standard for linked list problems not to allocate any data structure but the output list. This way, one could target \mathcal{O}(1)O(1) space complexity without taking the output list into account.

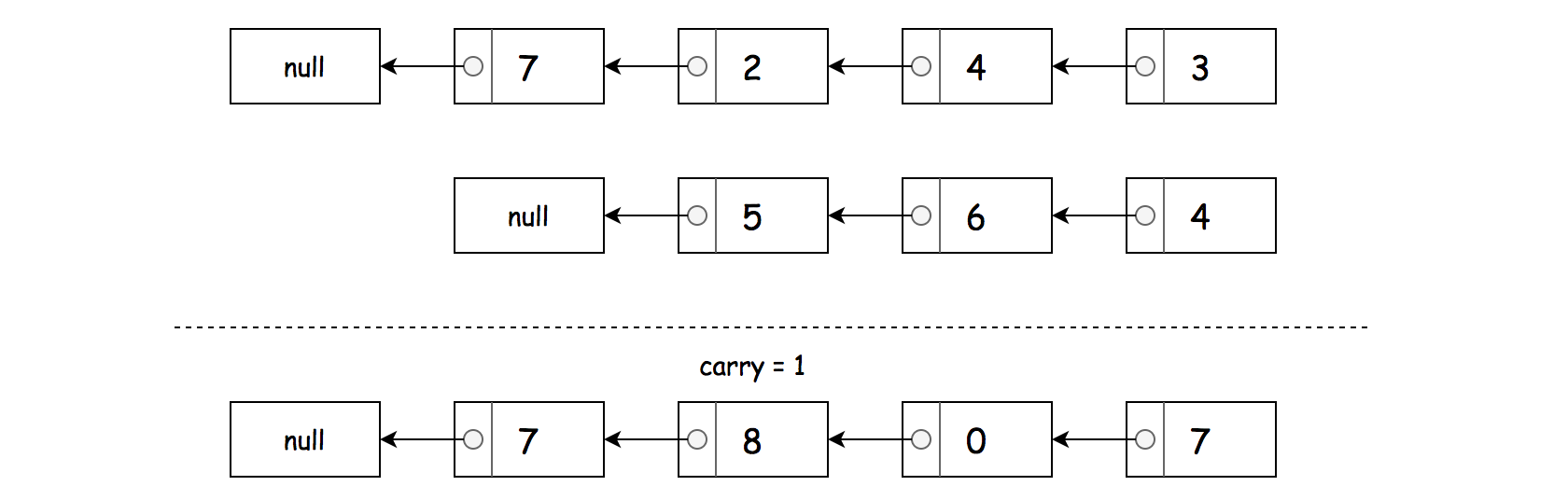
**How to Solve: Reverse Input vs. Reverse Output**

The standard textbook addition algorithm begins by summing the least-significant digits. Each digit is in the range from 0 to 9, and the "overflows" are managed by moving the carry into the next range.

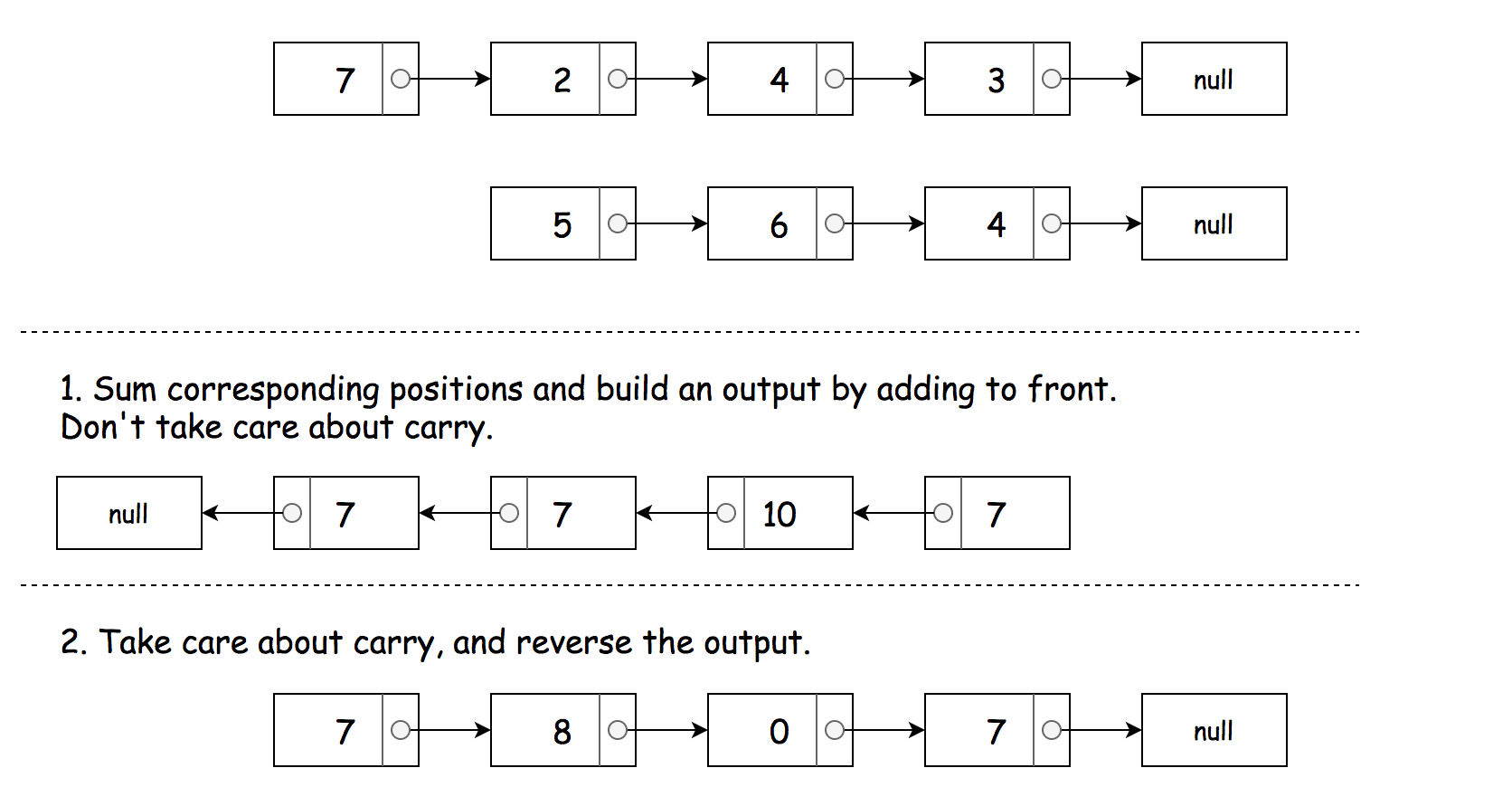
 *Figure 1. Textbook Addition.*

That is quite convenient for the problem [Add Two Numbers](https://leetcode.com/problems/add-two-numbers/), where the digits are stored in *reverse order*. In this problem, the digits are stored in *direct order*. There are two ways to proceed:

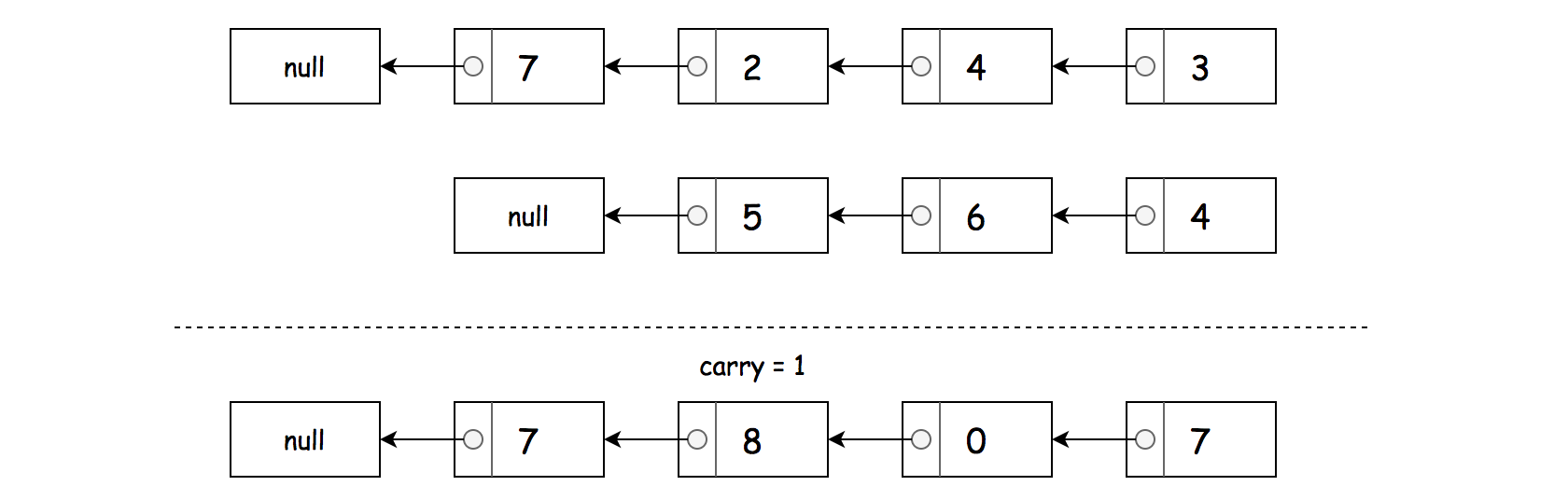
* Approach 1. The idea is to adapt input by reversing the input lists. This way, one could parse reverse lists starting from the head and use the textbook addition algorithm.

 *Figure 2. Approach 1: Reverse Input + Construct Output by Adding to Front.*

* Approach 2. The idea is to adapt the addition algorithm.

 *Figure 3. Approach 2: Adapt Addition Algorithm + Reverse Output.*

Approach 1: Reverse Input + Construct Output by Adding to Front

 *Figure 4. Reverse Input + Construct Output by Adding to Front.*

**Algorithm**

* [Implement reverseList function](https://leetcode.com/problems/reverse-linked-list/solution/).
* Reverse both input lists: l1 = reverseList(l1), l2 = reverseList(l2).
* Initialize the result list: head = None.
* Initialize the carry: carry = 0.
* Loop through lists l1 and l2 until you reach both ends.
  + Set x1 = l1.val if l1 is not finished yet, and x1 = 0 otherwise.
  + Set x2 = l2.val if l2 is not finished yet, and x2 = 0 otherwise.
  + Compute the current value: val = (carry + x1 + x2) % 10, and the current carry: carry = (carry + x1 + x2) / 10.
  + Update the result by adding the current value to front.
  + Move to the next elements in the lists.
* If the carry is not equal to zero, append it to frond of the result list.
* Return the result list: return head.

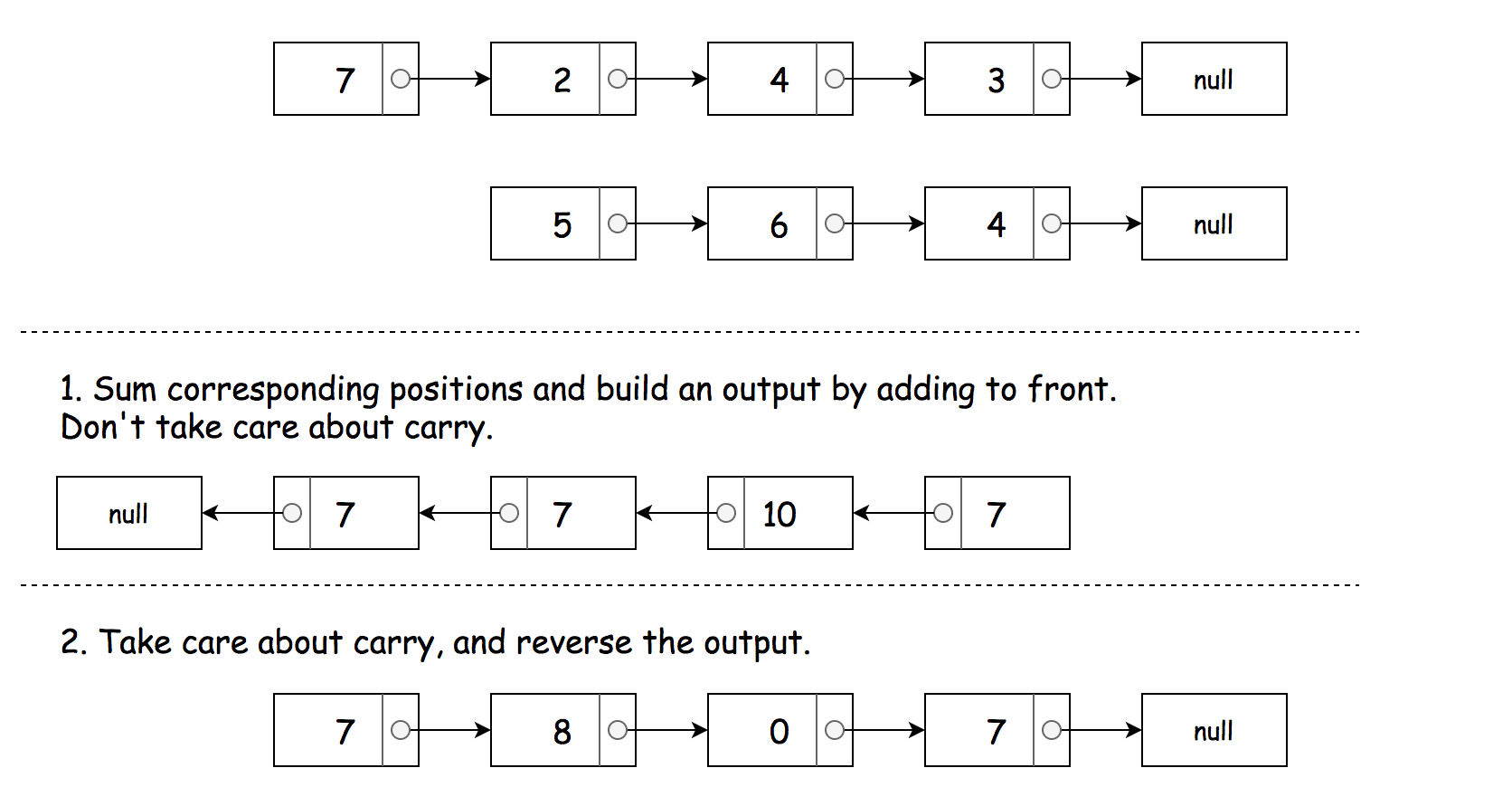
**Implementation**

|  |
| --- |
| class Solution {  public ListNode reverseList(ListNode head) {  ListNode last = null;  while (head != null) {  // keep the next node  ListNode tmp = head.next;  // reverse the link  head.next = last;  // update the last node and the current node  last = head;  head = tmp;  }  return last;  }    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {  // reverse lists  l1 = reverseList(l1);  l2 = reverseList(l2);    ListNode head = null;  int carry = 0;  while (l1 != null || l2 != null) {  // get the current values  int x1 = l1 != null ? l1.val : 0;  int x2 = l2 != null ? l2.val : 0;    // current sum and carry  int val = (carry + x1 + x2) % 10;  carry = (carry + x1 + x2) / 10;    // update the result: add to front  ListNode curr = new ListNode(val);  curr.next = head;  head = curr;    // move to the next elements in the lists  l1 = l1 != null ? l1.next : null;  l2 = l2 != null ? l2.next : null;  }  if (carry != 0) {  ListNode curr = new ListNode(carry);  curr.next = head;  head = curr;  }  return head;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N\_1 + N\_2)O(*N*1​+*N*2​), where N\_1 + N\_2*N*1​+*N*2​ is a number of elements in both lists.
* Space complexity: \mathcal{O}(1)O(1) space complexity without taking the output list into account, and \mathcal{O}(\max(N\_1, N\_2))O(max(*N*1​,*N*2​)) to store the output list.

#### **Approach 2: Follow Up: Do not Reverse Input.**

 Figure 5. Adapt Addition Algorithm + Reverse Output.

**Algorithm**

* Find the length of both lists: n1 and n2.
* Parse both lists and sum the corresponding positions without taking carry into account, i.e. convert 3->3->3 + 7->7 into 3->10->10 and then into 10->10->3.
  + To sum the corresponding positions, do the following:
    - If n1 >= n2, add to the current value the value of the node from the first list, and decrease the number of elements to parse: n1 -= 1.
    - If n1 < n2, add to the current value the value of the node from the second list and decrease the number of elements to parse: n2 -= 1.
  + Update the result by adding the current value to the front.
* Now it's time to take care about the carry, to limit each node value by 9, i.e. to convert 10->10->3 into 0->1->4 and then into 4->1->0:
  + Initialize the carry carry = 0.
  + Re-initialize the current list: curr1 = head and the output list: head = None.
  + Parse the current list curr1:
    - Normalize the current value to be less than 10: val = (curr1.val + carry) % 10, and keep the carry: carry = (curr1.val + carry) // 10.
    - Update the result by adding the current value to front.
    - Move to the next element in the list: curr1 = curr1.next.
* If the carry is not equal to zero, append it to frond of the result list.
* Return the result list: return head.

**Implementation**

|  |
| --- |
| class Solution {  public ListNode addTwoNumbers(ListNode l1, ListNode l2) {  // find the length of both lists  int n1 = 0, n2 = 0;  ListNode curr1 = l1, curr2 = l2;  while (curr1 != null) {  curr1 = curr1.next;  ++n1;  }  while (curr2 != null) {  curr2 = curr2.next;  ++n2;  }    // parse both lists  // and sum the corresponding positions  // without taking carry into account  // 3->3->3 + 7->7 --> 3->10->10--> 10->10->3  curr1 = l1;  curr2 = l2;  ListNode head = null;  while (n1 > 0 && n2 > 0) {  int val = 0;  if (n1 >= n2) {  val += curr1.val;  curr1 = curr1.next;  --n1;  }  if (n1 < n2) {  val += curr2.val;  curr2 = curr2.next;  --n2;  }    // update the result: add to front  ListNode curr = new ListNode(val);  curr.next = head;  head = curr;  }  // take the carry into account  // to have all elements to be less than 10  // 10->10->3 --> 0->1->4 --> 4->1->0  curr1 = head;  head = null;  int carry = 0;  while (curr1 != null) {  // current sum and carry  int val = (curr1.val + carry) % 10;  carry = (curr1.val + carry) / 10;    // update the result: add to front  ListNode curr = new ListNode(val);  curr.next = head;  head = curr;  // move to the next elements in the list  curr1 = curr1.next;  }    // add the last carry  if (carry != 0) {  ListNode curr = new ListNode(carry);  curr.next = head;  head = curr;  }  return head;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N\_1 + N\_2)O(*N*1​+*N*2​), where N\_1 + N\_2*N*1​+*N*2​ is a number of elements in both lists.
* Space complexity: \mathcal{O}(1)O(1) space complexity without taking the output list into account, and \mathcal{O}(\max(N\_1, N\_2))O(max(*N*1​,*N*2​)) to store the output list.

**Merge Two Sorted Lists**

Merge two sorted linked lists and return it as a **sorted** list. The list should be made by splicing together the nodes of the first two lists.

**Example 1:**



**Input:** l1 = [1,2,4], l2 = [1,3,4]

**Output:** [1,1,2,3,4,4]

**Example 2:**

**Input:** l1 = [], l2 = []

**Output:** []

**Example 3:**

**Input:** l1 = [], l2 = [0]

**Output:** [0]

**Constraints:**

* The number of nodes in both lists is in the range [0, 50].
* -100 <= Node.val <= 100
* Both l1 and l2 are sorted in **non-decreasing** order.

LinkedList

**Merge k Sorted Lists**

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order. *Merge all the linked-lists into one sorted linked-list and return it.*  
**Example 1:**

**Input:** lists = [[1,4,5],[1,3,4],[2,6]]

**Output:** [1,1,2,3,4,4,5,6]

**Explanation:** The linked-lists are:

[

1->4->5,

1->3->4,

2->6

]

merging them into one sorted list:

1->1->2->3->4->4->5->6

**Example 2:**

**Input:** lists = []

**Output:** []

**Example 3:**

**Input:** lists = [[]]

**Output:** []

**Constraints:**

* k == lists.length
* 0 <= k <= 10^4
* 0 <= lists[i].length <= 500
* -10^4 <= lists[i][j] <= 10^4
* lists[i] is sorted in **ascending order**.
* The sum of lists[i].length won't exceed 10^4.  
  Hard

**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:



begin to intersect at node c1.

**Example 1:**



**Input:** intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipB = 3

**Output:** Reference of the node with value = 8

**Input Explanation:** The intersected node's value is 8 (note that this must not be 0 if the two lists intersect). From the head of A, it reads as [4,1,8,4,5]. From the head of B, it reads as [5,6,1,8,4,5]. There are 2 nodes before the intersected node in A; There are 3 nodes before the intersected node in B.

**Example 2:**



**Input:** intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1

**Output:** Reference of the node with value = 2

**Input Explanation:** The intersected node's value is 2 (note that this must not be 0 if the two lists intersect). From the head of A, it reads as [1,9,1,2,4]. From the head of B, it reads as [3,2,4]. There are 3 nodes before the intersected node in A; There are 1 node before the intersected node in B.

**Example 3:**



**Input:** intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2

**Output:** null

**Input Explanation:** From the head of A, it reads as [2,6,4]. From the head of B, it reads as [1,5]. Since the two lists do not intersect, intersectVal must be 0, while skipA and skipB can be arbitrary values.

**Explanation:** The two lists do not intersect, so return null.

**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.
* Each value on each linked list is in the range [1, 10^9].
* Your code should preferably run in O(n) time and use only O(1) memory.

Medium

**Copy List with Random Pointer**

A linked list of length n is given such that each node contains an additional random pointer, which could point to any node in the list, or null.

Construct a [**deep copy**](https://en.wikipedia.org/wiki/Object_copying#Deep_copy) of the list. The deep copy should consist of exactly n **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list**.

For example, if there are two nodes X and Y in the original list, where X.random --> Y, then for the corresponding two nodes x and y in the copied list, x.random --> y.

Return *the head of the copied linked list*.

The linked list is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random\_index] where:

* val: an integer representing Node.val
* random\_index: the index of the node (range from 0 to n-1) that the random pointer points to, or null if it does not point to any node.

Your code will **only** be given the head of the original linked list.

**Example 1:**



**Input:** head = [[7,null],[13,0],[11,4],[10,2],[1,0]]

**Output:** [[7,null],[13,0],[11,4],[10,2],[1,0]]

**Example 2:**



**Input:** head = [[1,1],[2,1]]

**Output:** [[1,1],[2,1]]

**Example 3:**

****

**Input:** head = [[3,null],[3,0],[3,null]]

**Output:** [[3,null],[3,0],[3,null]]

**Example 4:**

**Input:** head = []

**Output:** []

**Explanation:** The given linked list is empty (null pointer), so return null.

**Constraints:**

* 0 <= n <= 1000
* -10000 <= Node.val <= 10000
* Node.random is null or is pointing to some node in the linked list.

Hard Google

## Trees and Graphs

Validate Binary Tree, Binary Tree Level Order Traversal and Binary Tree Inorder Traversal are frequently asked by Microsoft. Microsoft does not ask a lot of graph type questions but you should still know the basics of graphs.

**Validate Binary Search Tree**

Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

* The left subtree of a node contains only nodes with keys **less than** the node's key.
* The right subtree of a node contains only nodes with keys **greater than** the node's key.
* Both the left and right subtrees must also be binary search trees.

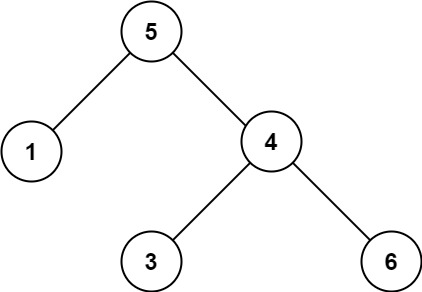
**Example 1:**



**Input:** root = [2,1,3]

**Output:** true

**Example 2:**



**Input:** root = [5,1,4,null,null,3,6]

**Output:** false

**Explanation:** The root node's value is 5 but its right child's value is 4.

**Constraints:**

* The number of nodes in the tree is in the range [1, 104].
* -231 <= Node.val <= 231 - 1

EZ FB

**Binary Tree Inorder Traversal**

Given the root of a binary tree, return *the inorder traversal of its nodes' values*.

**Example 1:**



**Input:** root = [1,null,2,3]

**Output:** [1,3,2]

**Example 2:**

**Input:** root = []

**Output:** []

**Example 3:**

**Input:** root = [1]

**Output:** [1]

**Example 4:**



**Input:** root = [1,2]

**Output:** [2,1]

**Example 5:**



**Input:** root = [1,null,2]

**Output:** [1,2]

**Constraints:**

* The number of nodes in the tree is in the range [0, 100].
* -100 <= Node.val <= 100

**Follow up:**

Recursive solution is trivial, could you do it iteratively?

Binary Search Medium

**Binary Tree Level Order Traversal**

Given the root of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

**Example 1:**



**Input:** root = [3,9,20,null,null,15,7]

**Output:** [[3],[9,20],[15,7]]

**Example 2:**

**Input:** root = [1]

**Output:** [[1]]

**Example 3:**

**Input:** root = []

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 2000].
* -1000 <= Node.val <= 1000

Binary Search EZ

**Binary Tree Zigzag Level Order Traversal**

Given the root of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

**Example 1:**



**Input:** root = [3,9,20,null,null,15,7]

**Output:** [[3],[20,9],[15,7]]

**Example 2:**

**Input:** root = [1]

**Output:** [[1]]

**Example 3:**

**Input:** root = []

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 2000].
* -100 <= Node.val <= 100

Medium

**Populating Next Right Pointers in Each Node**

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

struct Node {

int val;

Node \*left;

Node \*right;

Node \*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.

**Follow up:**

* You may only use constant extra space.
* Recursive approach is fine, you may assume implicit stack space does not count as extra space for this problem.

**Example 1:**



**Input:** root = [1,2,3,4,5,6,7]

**Output:** [1,#,2,3,#,4,5,6,7,#]

**Explanation:** Given the above perfect binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.

**Constraints:**

* The number of nodes in the given tree is less than 4096.
* -1000 <= node.val <= 1000

Medium Binary Tree

**Populating Next Right Pointers in Each Node II**

Given a binary tree

struct Node {

int val;

Node \*left;

Node \*right;

Node \*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.

**Follow up:**

* You may only use constant extra space.
* Recursive approach is fine, you may assume implicit stack space does not count as extra space for this problem.

**Example 1:**



**Input:** root = [1,2,3,4,5,null,7]

**Output:** [1,#,2,3,#,4,5,7,#]

**Explanation:** Given the above binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.

**Constraints:**

* The number of nodes in the given tree is less than 6000.
* -100 <= node.val <= 100

Binary Tree

**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 p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

**Example 1:**



**Input:** root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

**Output:** 6

**Explanation:** The LCA of nodes 2 and 8 is 6.

**Example 2:**



**Input:** root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4

**Output:** 2

**Explanation:** The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

**Example 3:**

**Input:** root = [2,1], p = 2, q = 1

**Output:** 2

**Constraints:**

* The number of nodes in the tree is in the range [2, 105].
* -109 <= Node.val <= 109
* All Node.val are **unique**.
* p != q
* p and q will exist in the BST.

Binary Search Tree

**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 p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

**Example 1:**



**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

**Output:** 3

**Explanation:** The LCA of nodes 5 and 1 is 3.

**Example 2:**



**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

**Output:** 5

**Explanation:** The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

**Example 3:**

**Input:** root = [1,2], p = 1, q = 2

**Output:** 1

**Constraints:**

* The number of nodes in the tree is in the range [2, 105].
* -109 <= Node.val <= 109
* All Node.val are **unique**.
* p != q
* p and q will exist in the tree.

Hard FB

**Construct Binary Tree from Preorder and Inorder Traversal**

Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return *the binary tree*.

**Example 1:**



**Input:** preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]

**Output:** [3,9,20,null,null,15,7]

**Example 2:**

**Input:** preorder = [-1], inorder = [-1]

**Output:** [-1]

**Constraints:**

* 1 <= preorder.length <= 3000
* inorder.length == preorder.length
* -3000 <= preorder[i], inorder[i] <= 3000
* preorder and inorder consist of **unique** values.
* Each value of inorder also appears in preorder.
* preorder is **guaranteed** to be the preorder traversal of the tree.
* inorder is **guaranteed** to be the inorder traversal of the tree.

Binary Tree Medium

**Number of Islands**

Given an m x n 2d grid map of '1's (land) and '0's (water), return *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:**

**Input:** grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

["0","0","0","0","0"]

]

**Output:** 1

**Example 2:**

**Input:** grid = [

["1","1","0","0","0"],

["1","1","0","0","0"],

["0","0","1","0","0"],

["0","0","0","1","1"]

]

**Output:** 3

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 300
* grid[i][j] is '0' or '1'.

Queue Stack Medium

**Clone Graph**

Given a reference of a node in a [**connected**](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)#Connected_graph) undirected graph.

Return a [**deep copy**](https://en.wikipedia.org/wiki/Object_copying#Deep_copy) (clone) of the graph.

Each node in the graph contains a val (int) and a list (List[Node]) of its neighbors.

class Node {

public int val;

public List<Node> neighbors;

}

**Test case format:**

For simplicity sake, each node's value is the same as the node's index (1-indexed). For example, the first node with val = 1, the second node with val = 2, and so on. The graph is represented in the test case using an adjacency list.

**Adjacency list** is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with val = 1. You must return the **copy of the given node** as a reference to the cloned graph.

**Example 1:**



**Input:** adjList = [[2,4],[1,3],[2,4],[1,3]]

**Output:** [[2,4],[1,3],[2,4],[1,3]]

**Explanation:** There are 4 nodes in the graph.

1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

**Example 2:**



**Input:** adjList = [[]]

**Output:** [[]]

**Explanation:** Note that the input contains one empty list. The graph consists of only one node with val = 1 and it does not have any neighbors.

**Example 3:**

**Input:** adjList = []

**Output:** []

**Explanation:** This an empty graph, it does not have any nodes.

**Example 4:**



**Input:** adjList = [[2],[1]]

**Output:** [[2],[1]]

**Constraints:**

* 1 <= Node.val <= 100
* Node.val is unique for each node.
* Number of Nodes will not exceed 100.
* There is no repeated edges and no self-loops in the graph.
* The Graph is connected and all nodes can be visited starting from the given node.

Queue Stack FB

## Backtracking

We recommend Letter Combinations of a Phone Number and Word Search II.

**Letter Combinations of a Phone Number**

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**.

A mapping of digit to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.



**Example 1:**

**Input:** digits = "23"

**Output:** ["ad","ae","af","bd","be","bf","cd","ce","cf"]

**Example 2:**

**Input:** digits = ""

**Output:** []

**Example 3:**

**Input:** digits = "2"

**Output:** ["a","b","c"]

**Constraints:**

* 0 <= digits.length <= 4
* digits[i] is a digit in the range ['2', '9'].

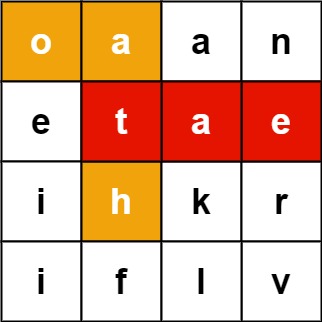
Recursion 2 Binary Search Medium. Google FB MS

**Word Search II**

Given an m x n board of characters and a list of strings words, return *all words on the board*.

Each word must be constructed from letters of sequentially adjacent cells, where **adjacent cells** are horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

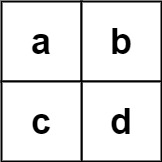
**Example 1:**



**Input:** board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","pea","eat","rain"]

**Output:** ["eat","oath"]

**Example 2:**



**Input:** board = [["a","b"],["c","d"]], words = ["abcb"]

**Output:** []

**Constraints:**

* m == board.length
* n == board[i].length
* 1 <= m, n <= 12
* board[i][j] is a lowercase English letter.
* 1 <= words.length <= 3 \* 104
* 1 <= words[i].length <= 10
* words[i] consists of lowercase English letters.
* All the strings of words are unique.

Tries Google Hard Amazon

**Wildcard Matching**

Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '\*' where:

* '?' Matches any single character.
* '\*' Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

**Example 1:**

**Input:** s = "aa", p = "a"

**Output:** false

**Explanation:** "a" does not match the entire string "aa".

**Example 2:**

**Input:** s = "aa", p = "\*"

**Output:** true

**Explanation:** '\*' matches any sequence.

**Example 3:**

**Input:** s = "cb", p = "?a"

**Output:** false

**Explanation:** '?' matches 'c', but the second letter is 'a', which does not match 'b'.

**Example 4:**

**Input:** s = "adceb", p = "\*a\*b"

**Output:** true

**Explanation:** The first '\*' matches the empty sequence, while the second '\*' matches the substring "dce".

**Example 5:**

**Input:** s = "acdcb", p = "a\*c?b"

**Output:** false

**Constraints:**

* 0 <= s.length, p.length <= 2000
* s contains only lowercase English letters.
* p contains only lowercase English letters, '?' or '\*'.

Hard

**Regular Expression Matching**

Given an input string (s) and a pattern (p), implement regular expression matching with support for '.' and '\*' where:

* '.' Matches any single character.​​​​
* '\*' Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial).

**Example 1:**

**Input:** s = "aa", p = "a"

**Output:** false

**Explanation:** "a" does not match the entire string "aa".

**Example 2:**

**Input:** s = "aa", p = "a\*"

**Output:** true

**Explanation:** '\*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

**Example 3:**

**Input:** s = "ab", p = ".\*"

**Output:** true

**Explanation:** ".\*" means "zero or more (\*) of any character (.)".

**Example 4:**

**Input:** s = "aab", p = "c\*a\*b"

**Output:** true

**Explanation:** c can be repeated 0 times, a can be repeated 1 time. Therefore, it matches "aab".

**Example 5:**

**Input:** s = "mississippi", p = "mis\*is\*p\*."

**Output:** false

**Constraints:**

* 0 <= s.length <= 20
* 0 <= p.length <= 30
* s contains only lowercase English letters.
* p contains only lowercase English letters, '.', and '\*'.
* It is guaranteed for each appearance of the character '\*', there will be a previous valid character to match.

Hard FB Yelp

**Remove Duplicates from Sorted Array**

Given a sorted array *nums*, remove the duplicates [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) such that each element appears only *once* and returns the new length.

Do not allocate extra space for another array, you must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

**Clarification:**

Confused why the returned value is an integer but your answer is an array?

Note that the input array is passed in by **reference**, which means a modification to the input array will be known to the caller as well.

Internally you can think of this:

// **nums** is passed in by reference. (i.e., without making a copy)

int len = removeDuplicates(nums);

// any modification to **nums** in your function would be known by the caller.

// using the length returned by your function, it prints the first **len** elements.

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

    print(nums[i]);

}

**Example 1:**

**Input:** nums = [1,1,2]

**Output:** 2, nums = [1,2]

**Explanation:** Your function should return length = **2**, with the first two elements of *nums* being **1** and **2** respectively. It doesn't matter what you leave beyond the returned length.

**Example 2:**

**Input:** nums = [0,0,1,1,1,2,2,3,3,4]

**Output:** 5, nums = [0,1,2,3,4]

**Explanation:** Your function should return length = **5**, with the first five elements of *nums* being modified to **0**, **1**, **2**, **3**, and **4** respectively. It doesn't matter what values are set beyond the returned length.

**Constraints:**

* 0 <= nums.length <= 3 \* 104
* -104 <= nums[i] <= 104
* nums is sorted in ascending order.

Array 101

Array

EZ

**Merge Sorted Array**

Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.

The number of elements initialized in nums1 and nums2 are m and n respectively. You may assume that nums1 has a size equal to m + n such that it has enough space to hold additional elements from nums2.

**Example 1:**

**Input:** nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3

**Output:** [1,2,2,3,5,6]

**Example 2:**

**Input:** nums1 = [1], m = 1, nums2 = [], n = 0

**Output:** [1]

**Constraints:**

* nums1.length == m + n
* nums2.length == n
* 0 <= m, n <= 200
* 1 <= m + n <= 200
* -109 <= nums1[i], nums2[i] <= 109

Array 101

EZ

**Sort Colors**

Given an array nums with n objects colored red, white, or blue, sort them [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm)so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

**Example 1:**

**Input:** nums = [2,0,2,1,1,0]

**Output:** [0,0,1,1,2,2]

**Example 2:**

**Input:** nums = [2,0,1]

**Output:** [0,1,2]

**Example 3:**

**Input:** nums = [0]

**Output:** [0]

**Example 4:**

**Input:** nums = [1]

**Output:** [1]

**Constraints:**

* n == nums.length
* 1 <= n <= 300
* nums[i] is 0, 1, or 2.

**Follow up:**

* Could you solve this problem without using the library's sort function?
* Could you come up with a one-pass algorithm using only O(1) constant space?

Medium

**Find Minimum in Rotated Sorted Array**

Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

* [4,5,6,7,0,1,2] if it was rotated 4 times.
* [0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that **rotating** an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums, return *the minimum element of this array*.

**Example 1:**

**Input:** nums = [3,4,5,1,2]

**Output:** 1

**Explanation:** The original array was [1,2,3,4,5] rotated 3 times.

**Example 2:**

**Input:** nums = [4,5,6,7,0,1,2]

**Output:** 0

**Explanation:** The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.

**Example 3:**

**Input:** nums = [11,13,15,17]

**Output:** 11

**Explanation:** The original array was [11,13,15,17] and it was rotated 4 times.

**Constraints:**

* n == nums.length
* 1 <= n <= 5000
* -5000 <= nums[i] <= 5000
* All the integers of nums are **unique**.
* nums is sorted and rotated between 1 and n times.

Hide Hint #1

Array was originally in ascending order. Now that the array is rotated, there would be a point in the array where there is a small deflection from the increasing sequence. eg. The array would be something like [4, 5, 6, 7, 0, 1, 2].

   Hide Hint #2

You can divide the search space into two and see which direction to go. Can you think of an algorithm which has O(logN) search complexity?

   Hide Hint #3

1. All the elements to the left of inflection point > first element of the array.
2. All the elements to the right of inflection point < first element of the array.

## Solution

#### **Approach 1: Binary Search**

**Intuition**

A very brute way of solving this question is to search the entire array and find the minimum element. The time complexity for that would be O(N)*O*(*N*) given that N is the size of the array.

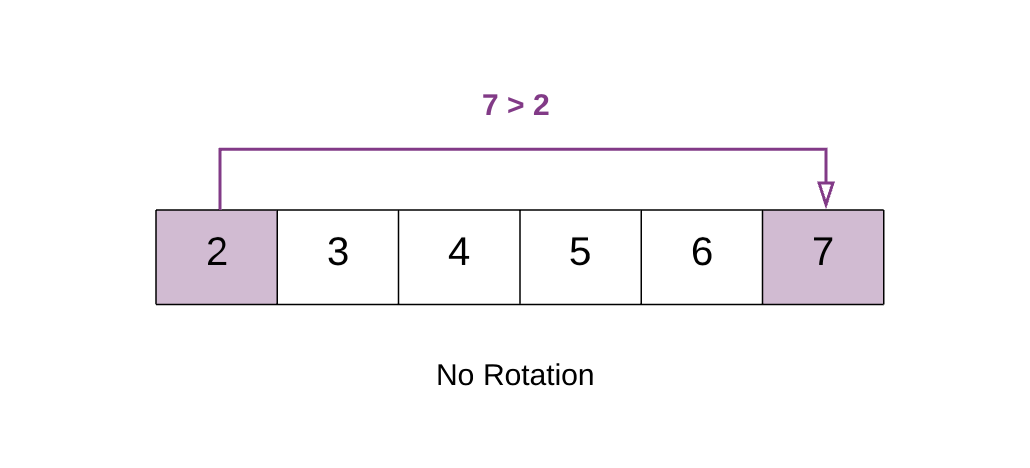
A very cool way of solving this problem is using the Binary Search algorithm. In binary search we find out the mid point and decide to either search on the left or right depending on some condition.

Since the given array is sorted, we can make use of binary search. However, the array is rotated. So simply applying the binary search won't work here.

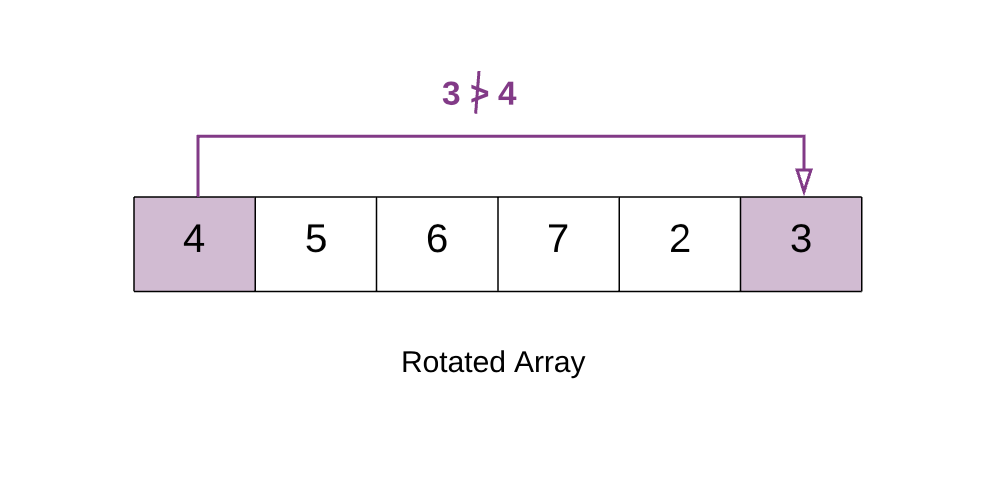
In this question we would essentially apply a modified version of binary search where the condition that decides the search direction would be different than in a standard binary search.

We want to find the smallest element in a rotated sorted array. What if the array is not rotated? How do we check that?

If the array is not rotated and the array is in ascending order, then last element > first element.

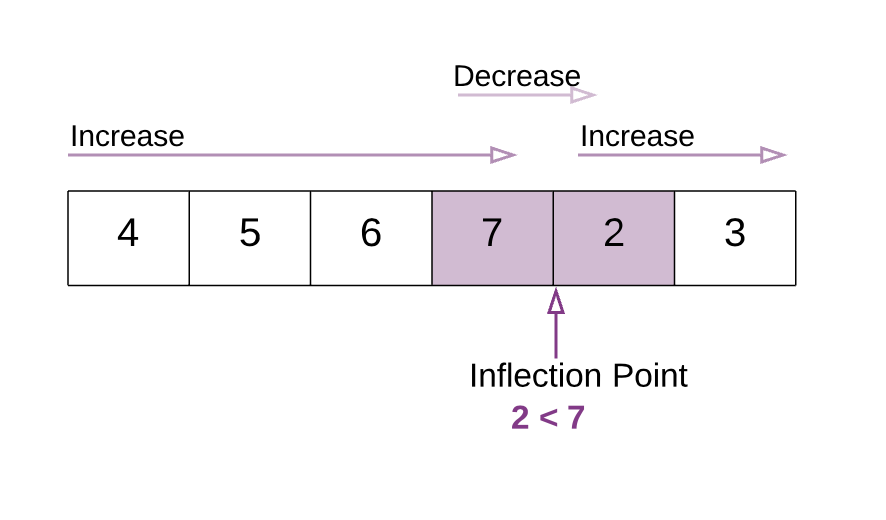


In the above example 7 > 2. This means that the array is still sorted and has no rotation.



In the above example 3 < 4. Hence the array is rotated. This happens because the array was initially [2, 3 ,4 ,5 ,6 ,7]. But after the rotation the smaller elements[2,3] go at the back. i.e. [4, 5, 6, 7, 2, 3]. Because of this the first element [4] in the rotated array becomes greater than the last element.

This means there is a point in the array at which you would notice a change. This is the point which would help us in this question. We call this the Inflection Point.

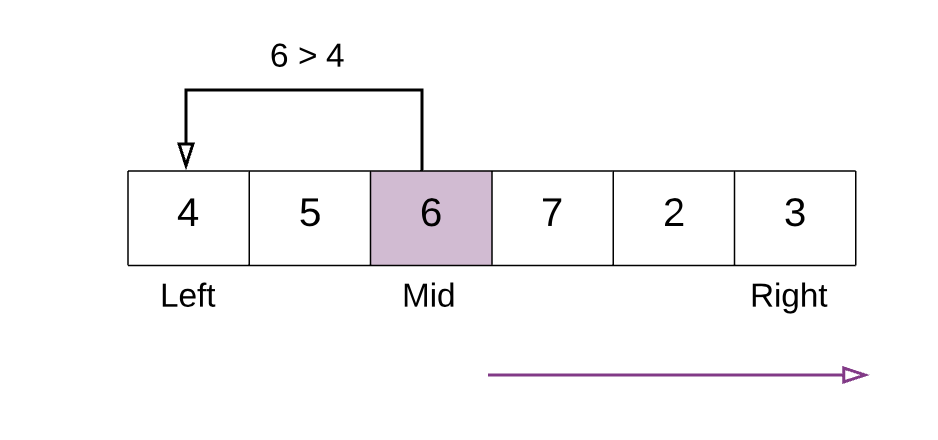


In this modified version of binary search algorithm, we are looking for this point. In the above example notice the Inflection Point .

All the elements to the left of inflection point > first element of the array.  
All the elements to the right of inflection point < first element of the array.

**Algorithm**

1. Find the mid element of the array.
2. If mid element > first element of array this means that we need to look for the inflection point on the right of mid.
3. If mid element < first element of array this that we need to look for the inflection point on the left of mid.

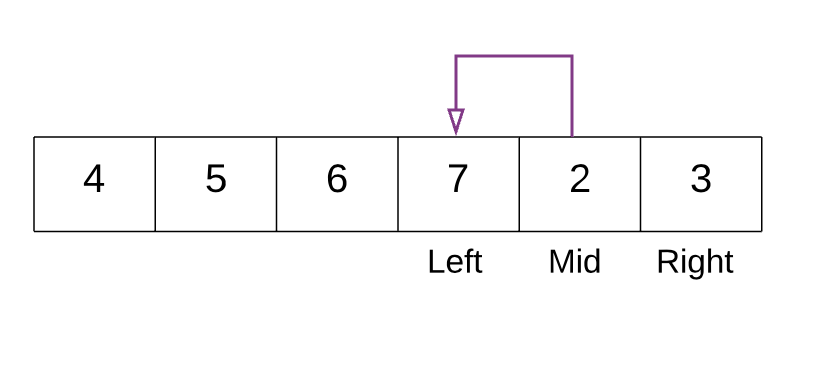


In the above example mid element 6 is greater than first element 4. Hence we continue our search for the inflection point to the right of mid.

4 . We stop our search when we find the inflection point, when either of the two conditions is satisfied:

nums[mid] > nums[mid + 1] Hence, **mid+1** is the smallest.

nums[mid - 1] > nums[mid] Hence, **mid** is the smallest.



In the above example. With the marked left and right pointers. The mid element is 2. The element just before 2 is 7 and 7>2 i.e. nums[mid - 1] > nums[mid]. Thus we have found the point of inflection and 2 is the smallest element.

|  |
| --- |
| class Solution {  public int findMin(int[] nums) {  // If the list has just one element then return that element.  if (nums.length == 1) {  return nums[0];  }  // initializing left and right pointers.  int left = 0, right = nums.length - 1;  // if the last element is greater than the first element then there is no rotation.  // e.g. 1 < 2 < 3 < 4 < 5 < 7. Already sorted array.  // Hence the smallest element is first element. A[0]  if (nums[right] > nums[0]) {  return nums[0];  }    // Binary search way  while (right >= left) {  // Find the mid element  int mid = left + (right - left) / 2;  // if the mid element is greater than its next element then mid+1 element is the smallest  // This point would be the point of change. From higher to lower value.  if (nums[mid] > nums[mid + 1]) {  return nums[mid + 1];  }  // if the mid element is lesser than its previous element then mid element is the smallest  if (nums[mid - 1] > nums[mid]) {  return nums[mid];  }  // if the mid elements value is greater than the 0th element this means  // the least value is still somewhere to the right as we are still dealing with elements  // greater than nums[0]  if (nums[mid] > nums[0]) {  left = mid + 1;  } else {  // if nums[0] is greater than the mid value then this means the smallest value is somewhere to  // the left  right = mid - 1;  }  }  return -1;  }  } |

**Complexity Analysis**

* Time Complexity : Same as Binary Search O(\log N)*O*(log*N*)
* Space Complexity : O(1)*O*(1)

**Find Minimum in Rotated Sorted Array II**

Suppose an array sorted in ascending order 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.

**Example 1:**

**Input:** [1,3,5]

**Output:** 1

**Example 2:**

**Input:** [2,2,2,0,1]

**Output:** 0

**Note:**

* This is a follow up problem to [Find Minimum in Rotated Sorted Array](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/description/).
* Would allow duplicates affect the run-time complexity? How and why?

## Solution

#### **Approach 1: Variant of Binary Search**

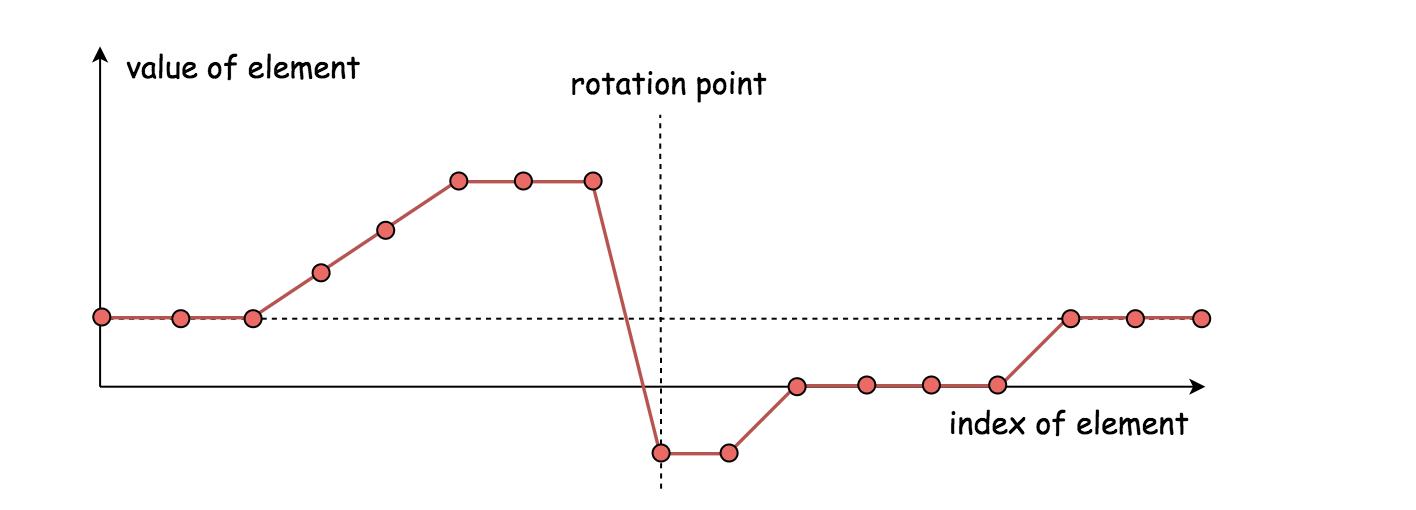
**Intuition**

Given a sorted array in ascending order (denoted as L[i]), the array is then rotated over certain unknown pivot, (denoted as L'[i]). We are asked to find the minimum value of this sorted and rotated array, which is to find the value of the first element in the original array, i.e. L[0].

The problem resembles a common problem of finding a given value from a sorted array, to which problem one could apply the **binary search** algorithm. Intuitively, one might wonder if we could apply a variant of binary search algorithm to solve our problem here.

Indeed, this is the right intuition, though the tricky part is to figure out a ***concise solution*** that could work for all cases.

To illustrate the algorithm, we draw the array in a 2D dimension in the following graph, where the X axis indicates the index of each element in the array and the Y axis indicates the value of the element.



The main structure of our algorithm remains the same as the classical binary search algorithm. As a reminder, we summarize it briefly as follows:

* We keep two pointers, i.e. low, high which point to the lowest and highest boundary of our search scope.
* We then reduce the search scope by moving either of pointers, according to various situations. Usually we shift one of pointers to the mid point between low and high, (i.e. pivot = (low+high)/2), which reduces the search scope down to half. This is also where the name of the algorithm comes from.
* The reduction of the search scope would stop, either we find the desired element or the two pointers converge (i.e. low == high).

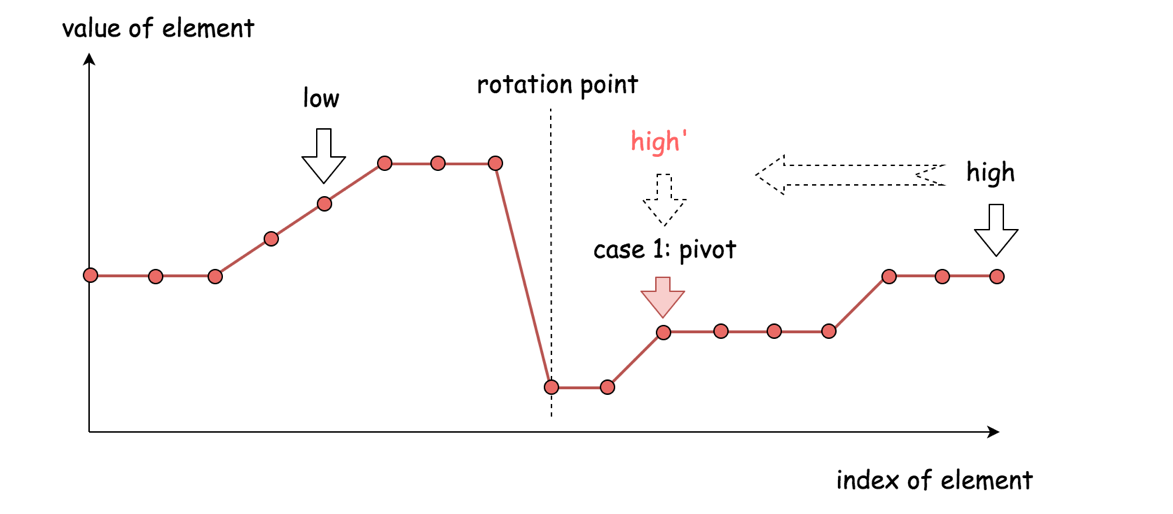
**Algorithm**

In the classical binary search algorithm, we would compare the pivot element (i.e. nums[pivot]) with the value that we would like to locate. In our case, however, we would compare the pivot element to the element pointed by the upper bound pointer (i.e. nums[high]).

Following the structure of the binary search algorithm, the essential part remained is to design the cases on how to update the two pointers.

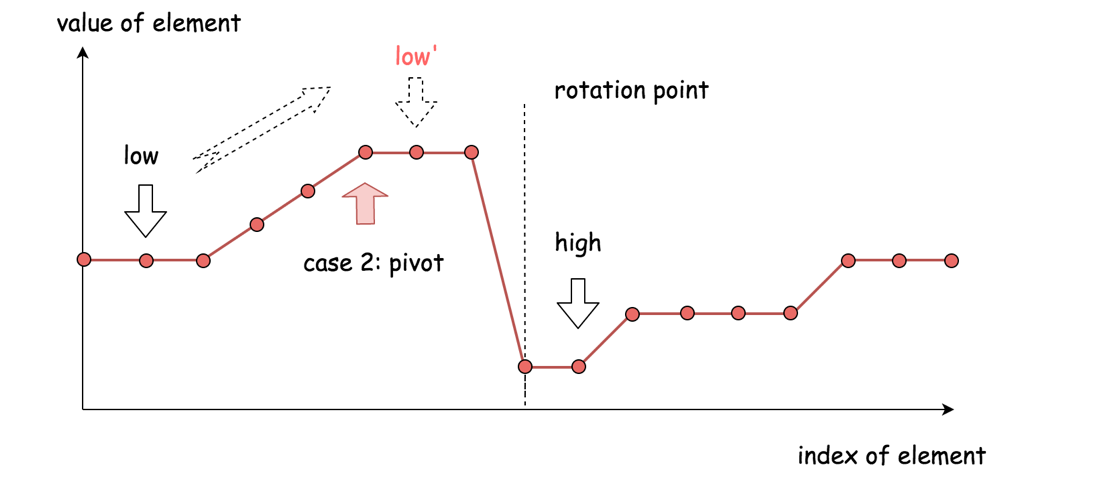
Here we give one example on how we can break it down ***concisely*** into three cases. Note that given the array, we consider the element pointed by the low index to be on the left-hand side of the array, and the element pointed by the high index to be on the right-hand side.

Case 1). nums[pivot] < nums[high]



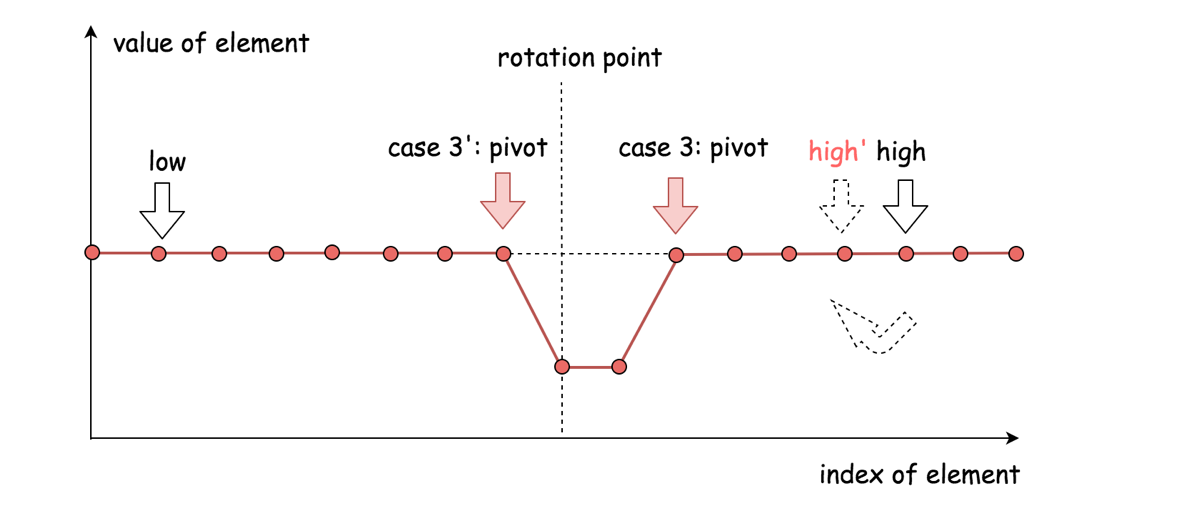
* The pivot element resides in the same half as the upper bound element.
* Therefore, the desired minimum element should reside to the **left-hand side** of pivot element. As a result, we then move the upper bound down to the pivot index, i.e. high = pivot.

Case 2). nums[pivot] > nums[high]



* The pivot element resides in the different half of array as the upper bound element.
* Therefore, the desired minium element should reside to the **right-hand side** of the pivot element. As a result, we then move the lower bound up next to the pivot index, i.e. low = pivot + 1.

Case 3). nums[pivot] == nums[high]



* In this case, we are not sure which side of the pivot that the desired minimum element would reside.
* To further reduce the search scope, a safe measure would be to reduce the upper bound by one (i.e. high = high - 1), rather than moving aggressively to the pivot point.
* The above strategy would prevent the algorithm from stagnating (i.e. endless loop). More importantly, it maintains the **correctness** of the procedure, i.e. we would not end up with skipping the desired element.

To summarize, this algorithm differs to the classical binary search algorithm in two parts:

* We use the upper bound of search scope as the reference for the comparison with the pivot element, while in the classical binary search the reference would be the desired value.
* When the result of comparison is equal (i.e. Case #3), we further move the upper bound, while in the classical binary search normally we would return the value immediately.

Here are some sample implementations based on the above algorithm. Note: the idea is inspired by the post from [sheehan](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array-ii/discuss/48808/My-pretty-simple-code-to-solve-it) in the discussion forum.

|  |
| --- |
| class Solution {  public int findMin(int[] nums) {  int low = 0, high = nums.length - 1;  while (low < high) {  int pivot = low + (high - low) / 2;  if (nums[pivot] < nums[high])  high = pivot;  else if (nums[pivot] > nums[high])  low = pivot + 1;  else  high -= 1;  }  return nums[low];  }  } |

**Complexity Analysis**

* Time complexity: on average \mathcal{O}(\log\_{2}{N})O(log2​*N*) where N*N* is the length of the array, since in general it is a binary search algorithm. However, in the worst case where the array contains identical elements (*i.e.* case #3 nums[pivot]==nums[high]), the algorithm would deteriorate to iterating each element, as a result, the time complexity becomes \mathcal{O}(N)O(*N*).
* Space complexity : \mathcal{O}(1)O(1), it's a constant space solution.

**Discussion**

The problem is a follow-up to the problem of [153. Find Minimum in Rotated Sorted Array](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/). The difference is that in this problem the array can contain duplicates. *So the question is "Would allow duplicates affect the run-time complexity? How and why?"*

First of all, the problem of [153. Find Minimum in Rotated Sorted Array](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/) can be considered as a specific case of this problem, where it just happens that the array does not contain any duplicate. As a result, the very solutions of this problem would work for the problem of [#153](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/) as well. It is just that we would never come cross the case #3 (*i.e.* nums[pivot] == nums[high]) in the problem of [#153](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/).

It is due to the fact that there might exist some duplicates in the array, that we come up the case #3 which eventually render the time complexity of the algorithm to be linear \mathcal{O}(N)O(*N*), rather than \mathcal{O}(\log\_{2}{N})O(log2​*N*).

One might wonder that whether it works in case #3 if we move the lower boundary (*i.e.* low += 1), rather than the upper boundary (*i.e.* high -= 1).

The short answer is that it could work for some cases, but not for all. For instance, given the input [1, 3, 3], by moving the lower boundary, we would skip the correct answer.

While we do low = pivot + 1 to reduce the search scope, then why not do high = pivot - 1 instead of high = pivot? Or a similar question would be *"why don't we do check of low <= high rather than low < high"?*

As a matter of fact, the binary search algorithm has several [forms of implementation](https://en.wikipedia.org/wiki/Binary_search_algorithm), regarding how we set the boundaries and the loop conditions. One can refer to the [Explore card of Binary Search](https://leetcode.com/explore/learn/card/binary-search/) in LeetCode for more details. As simple as the idea of binary search might seem to be, it is tricky to make it work for all cases.

As one would discover from the card, the above implementation of binary search complies with the [template II](https://leetcode.com/explore/learn/card/binary-search/126/template-ii/937/) of binary search. And by replacing high = pivot with high = pivot - 1, the algorithm will not work.

As subtle as it looks like, the update of the pointers should be consistent with the conditions of the loop. As a rule of thumb, it is advised to stick with one form of binary search, and not to mix them up.

One might notice that we are calculating the pivot with the formula of pivot = low + (high-low)/2, rather than the more intuitive term pivot = (high+low)/2.

Actually, this is done intentionally to prevent the numeric overflow issue, since the sum of two integers could exceed the limit of the integer number. As a fun fact, the above mistake prevails in many implementations of binary search, as revealed from a post titled [*"Nearly All Binary Searches and Mergesorts are Broken"*](https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html) from googleblog in 2006.

**Search in Rotated Sorted Array**

There is an integer array nums sorted in ascending order (with **distinct** values).

Prior to being passed to your function, nums is **rotated** at an unknown pivot index k (0 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (**0-indexed**). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums **after** the rotation and an integer target, return *the index of*target*if it is in*nums*, or*-1*if it is not in*nums.

**Example 1:**

**Input:** nums = [4,5,6,7,0,1,2], target = 0

**Output:** 4

**Example 2:**

**Input:** nums = [4,5,6,7,0,1,2], target = 3

**Output:** -1

**Example 3:**

**Input:** nums = [1], target = 0

**Output:** -1

**Constraints:**

* 1 <= nums.length <= 5000
* -104 <= nums[i] <= 104
* All values of nums are **unique**.
* nums is guaranteed to be rotated at some pivot.
* -104 <= target <= 104

**Follow up:** Can you achieve this in O(log n) time complexity?

Medium FB

**Search a 2D Matrix**

**Solution**

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.

**Example 1:**



**Input:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

**Output:** true

**Example 2:**



**Input:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13

**Output:** false

**Constraints:**

* m == matrix.length
* n == matrix[i].length
* 1 <= m, n <= 100
* -104 <= matrix[i][j], target <= 104

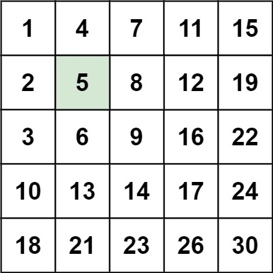
Recursion 2

**Search a 2D Matrix II**

Write an efficient algorithm that searches for a target value in an m x n integer matrix. The 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.

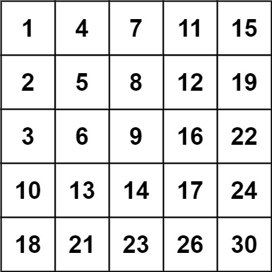
**Example 1:**



**Input:** 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]], target = 5

**Output:** true

**Example 2:**



**Input:** 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]], target = 20

**Output:** false

**Constraints:**

* m == matrix.length
* n == matrix[i].length
* 1 <= n, m <= 300
* -109 <= matix[i][j] <= 109
* All the integers in each row are **sorted** in ascending order.
* All the integers in each column are **sorted** in ascending order.

Recursion 2

**Median of Two Sorted Arrays**

Given two sorted arrays nums1 and nums2 of size m and n respectively, return **the median** of the two sorted arrays.

**Follow up:** The overall run time complexity should be O(log (m+n)).

**Example 1:**

**Input:** nums1 = [1,3], nums2 = [2]

**Output:** 2.00000

**Explanation:** merged array = [1,2,3] and median is 2.

**Example 2:**

**Input:** nums1 = [1,2], nums2 = [3,4]

**Output:** 2.50000

**Explanation:** merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.

**Example 3:**

**Input:** nums1 = [0,0], nums2 = [0,0]

**Output:** 0.00000

**Example 4:**

**Input:** nums1 = [], nums2 = [1]

**Output:** 1.00000

**Example 5:**

**Input:** nums1 = [2], nums2 = []

**Output:** 2.00000

**Constraints:**

* nums1.length == m
* nums2.length == n

Hard

## Dynamic Programming

Microsoft does not ask a lot of Dynamic Programming type questions, but question such as Longest Increasing Subsequence has been asked in a Microsoft onsite interviews.

**Best Time to Buy and Sell Stock**

You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

**Example 1:**

**Input:** prices = [7,1,5,3,6,4]

**Output:** 5

**Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

**Example 2:**

**Input:** prices = [7,6,4,3,1]

**Output:** 0

**Explanation:** In this case, no transactions are done and the max profit = 0.

**Constraints:**

* 1 <= prices.length <= 105
* 0 <= prices[i] <= 104

EZ Hard

**Maximum Subarray**

Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

**Example 1:**

**Input:** nums = [-2,1,-3,4,-1,2,1,-5,4]

**Output:** 6

**Explanation:** [4,-1,2,1] has the largest sum = 6.

**Example 2:**

**Input:** nums = [1]

**Output:** 1

**Example 3:**

**Input:** nums = [0]

**Output:** 0

**Example 4:**

**Input:** nums = [-1]

**Output:** -1

**Example 5:**

**Input:** nums = [-100000]

**Output:** -100000

**Constraints:**

* 1 <= nums.length <= 3 \* 104
* -105 <= nums[i] <= 105

EZ

**Longest Increasing Subsequence**

Given an integer array nums, return the length of the longest strictly increasing subsequence.

A **subsequence** is a sequence that can be derived from an array by deleting some or no elements without changing the order of the remaining elements. For example, [3,6,2,7] is a subsequence of the array [0,3,1,6,2,2,7].

**Example 1:**

**Input:** nums = [10,9,2,5,3,7,101,18]

**Output:** 4

**Explanation:** The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

**Example 2:**

**Input:** nums = [0,1,0,3,2,3]

**Output:** 4

**Example 3:**

**Input:** nums = [7,7,7,7,7,7,7]

**Output:** 1

**Constraints:**

* 1 <= nums.length <= 2500
* -104 <= nums[i] <= 104

**Follow up:**

* Could you come up with the O(n2) solution?
* Could you improve it to O(n log(n)) time complexity?

Medium

## Design

These are some design questions for you to practice for your Microsoft interview. You should practice both Serialize and Deserialize Binary Tree/BST. Make sure for the BST variant that you use the special properties of BST for the serialization and deserialization.

Also make sure you know how to implement a prefix tree (Trie), which is frequently asked by Microsoft.

**Serialize and Deserialize BST**

Serialization is converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a **binary search tree**. There is no restriction on how your serialization/deserialization algorithm should work. You need to ensure that a binary search tree can be serialized to a string, and this string can be deserialized to the original tree structure.

**The encoded string should be as compact as possible.**

**Example 1:**

**Input:** root = [2,1,3]

**Output:** [2,1,3]

**Example 2:**

**Input:** root = []

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* 0 <= Node.val <= 104
* The input tree is **guaranteed** to be a binary search tree.

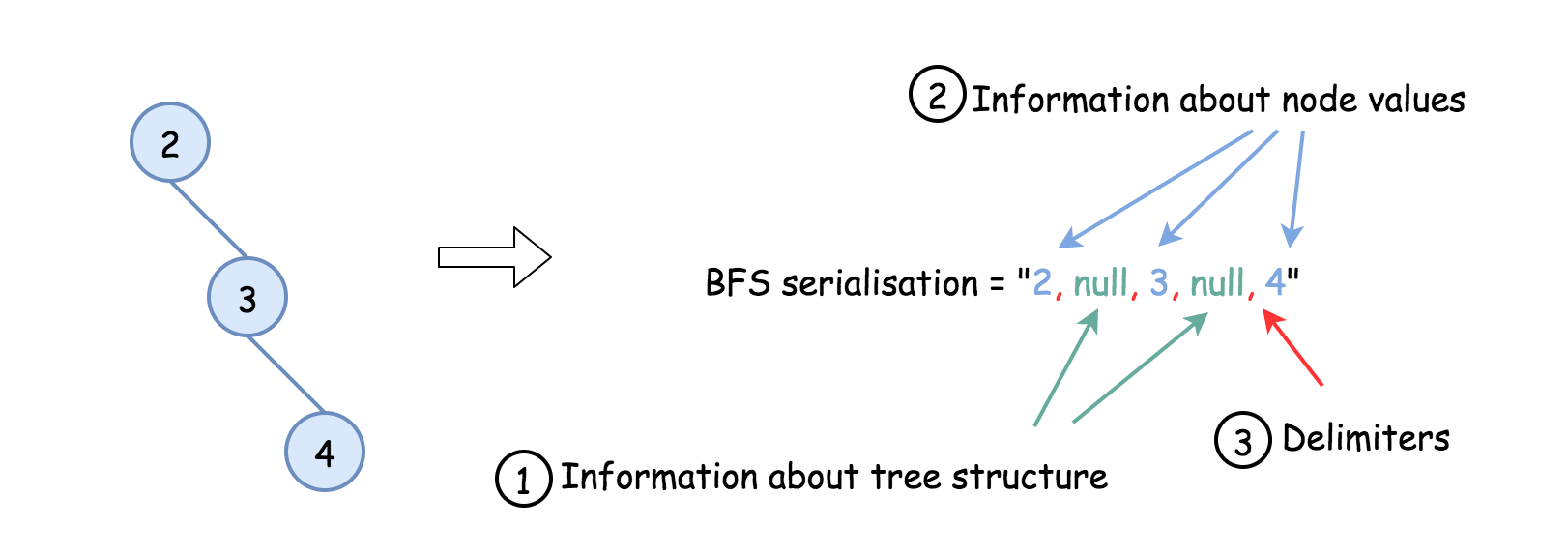
Solution

How to make the encoded string as compact as possible

This question is similar to the [Google interview question discussed last week](https://leetcode.com/discuss/interview-experience/297576/google-onsite-interview-sde1-new-grad-mountain-view-ca).

[To serialize](https://en.wikipedia.org/wiki/Serialization) a binary tree means to

* Encode tree structure.
* Encode node values.
* Choose delimiters to separate the values in the encoded string.



Hence there are three axes of optimisation here.

Approach 1: Postorder traversal to optimise space for the tree structure.

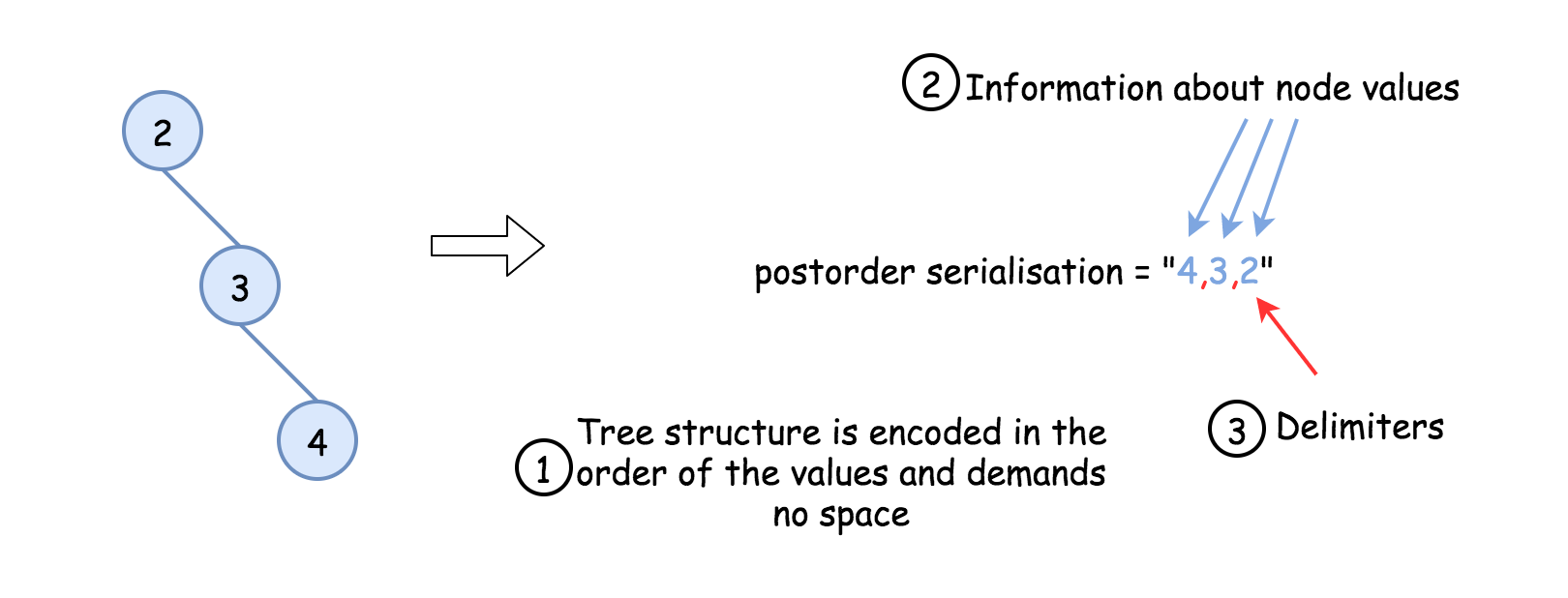
**Intuition**

Let's use here the fact that BST could be constructed from preorder or postorder traversal only. Please [check this article](https://leetcode.com/problems/construct-binary-search-tree-from-preorder-traversal/solution/) for the detailed discussion. In brief, it's a consequence of two facts:

* [Binary tree could be constructed from preorder/postorder and inorder traversal](https://leetcode.com/articles/construct-binary-tree-from-postorder-and-inorder-t/).
* [Inorder traversal of BST is an array sorted in the ascending order: inorder = sorted(preorder)](https://leetcode.com/articles/delete-node-in-a-bst/).

That means that BST structure is already encoded in the preorder or postorder traversal and hence they are both suitable for the compact serialization.

Serialization could be easily implemented with both strategies, but for optimal deserialization better to choose the postorder traversal because member/global/static variables are not allowed here.



**Implementation**

|  |
| --- |
| public class Codec {  public StringBuilder postorder(TreeNode root, StringBuilder sb) {  if (root == null)  return sb;  postorder(root.left, sb);  postorder(root.right, sb);  sb.append(root.val);  sb.append(' ');  return sb;  }  // Encodes a tree to a single string.  public String serialize(TreeNode root) {  StringBuilder sb = postorder(root, new StringBuilder());  if (sb.length() > 0)  sb.deleteCharAt(sb.length() - 1);  return sb.toString();  }  public TreeNode helper(Integer lower, Integer upper, ArrayDeque<Integer> nums) {  if (nums.isEmpty())  return null;  int val = nums.getLast();  if (val < lower || val > upper)  return null;  nums.removeLast();  TreeNode root = new TreeNode(val);  root.right = helper(val, upper, nums);  root.left = helper(lower, val, nums);  return root;  }  // Decodes your encoded data to tree.  public TreeNode deserialize(String data) {  if (data.isEmpty())  return null;  ArrayDeque<Integer> nums = new ArrayDeque<Integer>();  for (String s : data.split("\\s+"))  nums.add(Integer.valueOf(s));  return helper(Integer.MIN\_VALUE, Integer.MAX\_VALUE, nums);  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) both for serialization and deserialization. Let's compute the solution with the help of [master theorem](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)) T(N) = aT\left(\frac{b}{N}\right) + \Theta(N^d)*T*(*N*)=*aT*(*Nb*​)+Θ(*Nd*). The equation represents dividing the problem up into a*a* subproblems of size \frac{N}{b}*bN*​ in \Theta(N^d)Θ(*Nd*) time. Here one divides the problem in two subproblemes a = 2, the size of each subproblem (to compute left and right subtree) is a half of initial problem b = 2, and all this happens in a constant time d = 0. That means that \log\_b(a) > dlog*b*​(*a*)>*d* and hence we're dealing with [case 1](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)#Case_1_example) that means \mathcal{O}(N^{\log\_b(a)}) = \mathcal{O}(N)O(*N*log*b*​(*a*))=O(*N*) time complexity.
* Space complexity : \mathcal{O}(N)O(*N*), since we store the entire tree. Encoded string: one needs to store (N - 1)(*N*−1) delimiters, and N*N* node values in the encoded string. Tree structure is encoded in the order of values and uses no space.

#### **Approach 2: Convert int to 4-bytes string to optimise space for node values.**

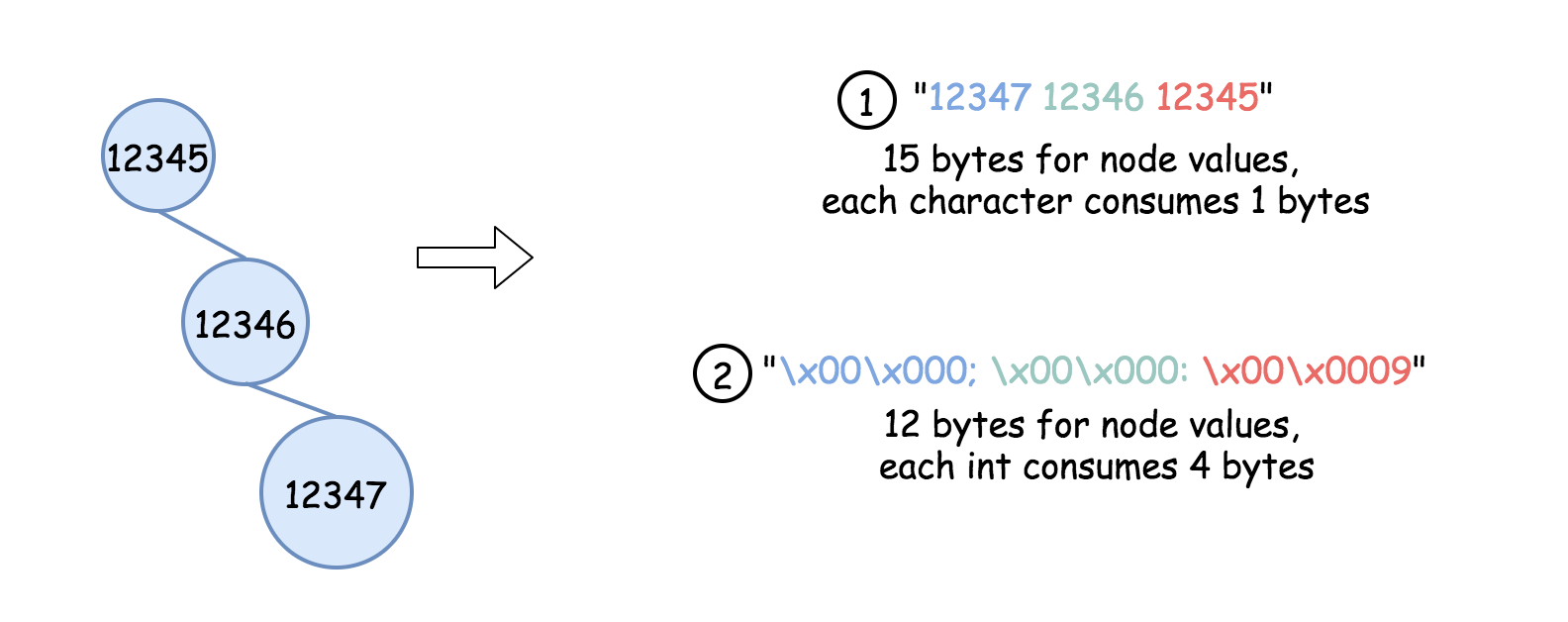
**Intuition**

Approach 1 works fine with the small node values but starts to consume more and more space in the case of large ones.

For example, the tree [2,null,3,null,4] is encoded as a string "4 3 2" which uses 5 bytes to store the values and delimiters, 1 byte per value or delimiter. So far everything is fine.

Let's consider now the tree [12345,null,12346,null,12347] which is encoded as "12347 12346 12345" and consumes 17 bytes to store 3 integers and 2 delimiters, 15 bytes for node values only. At the same time it's known that 4 bytes is enough to store an int value, i.e. 12 bytes should be enough for 3 integers. 15 > 12 and hence the storage of values could be optimised.

How to do it? Convert each integer into 4-bytes string.



**Implementation**

|  |
| --- |
| class Codec:  def postorder(self, root):  return self.postorder(root.left) + self.postorder(root.right) + [root.val] if root else []    def int\_to\_str(self, x):  """  Encodes integer to bytes string.  """  bytes = [chr(x >> (i \* 8) & 0xff) for i in range(4)]  bytes.reverse()  bytes\_str = ''.join(bytes)  return bytes\_str    def serialize(self, root):  """  Encodes a tree to a single string.  """  lst = self.postorder(root)  lst = [self.int\_to\_str(x) for x in lst]  return 'ç'.join(map(str, lst))    def str\_to\_int(self, bytes\_str):  """  Decodes bytes string to integer.  """  result = 0  for ch in bytes\_str:  result = result \* 256 + ord(ch)  return result    def deserialize(self, data):  """  Decodes your encoded data to tree.  """  def helper(lower = float('-inf'), upper = float('inf')):  if not data or data[-1] < lower or data[-1] > upper:  return None    val = data.pop()  root = TreeNode(val)  root.right = helper(val, upper)  root.left = helper(lower, val)  return root    data = [self.str\_to\_int(x) for x in data.split('ç') if x]  return helper() |

**Complexity Analysis**

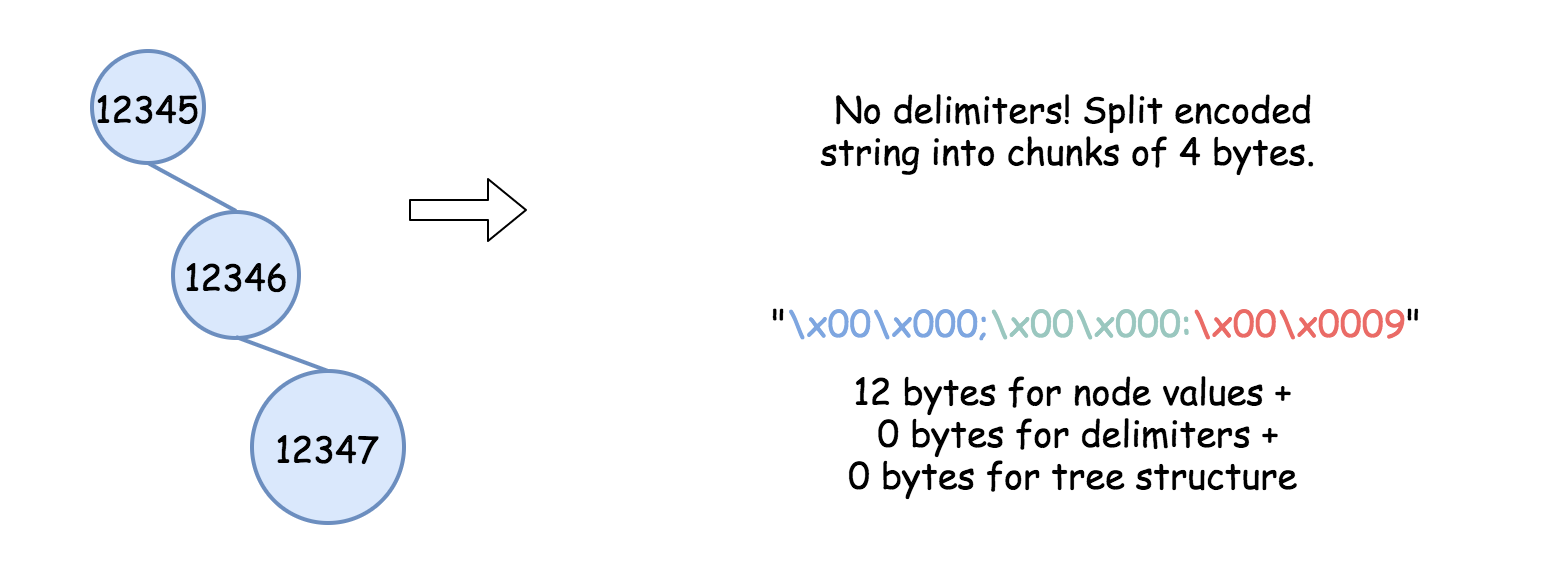
* Time complexity : \mathcal{O}(N)O(*N*) both for serialization and deserialization.
* Space complexity : \mathcal{O}(N)O(*N*), since we store the entire tree. Encoded string: one needs 2(N - 1)2(*N*−1) bytes for the delimiters, and 4 N4*N* bytes for the node values in the encoded string. Tree structure is encoded in the order of node values and uses no space.

#### **Approach 3: Get rid of delimiters.**

**Intuition**

Approach 2 works well except for delimiter usage.

Since all node values are now encoded as 4-bytes strings, one could just split the encoded string into 4-bytes chunks, convert each chunk back to the integer and proceed further.



**Implementation**

|  |
| --- |
| public class Codec {  // Encodes a tree to a list.  public void postorder(TreeNode root, StringBuilder sb) {  if (root == null)  return;  postorder(root.left, sb);  postorder(root.right, sb);  sb.append(intToString(root.val));  }  // Encodes integer to bytes string  public String intToString(int x) {  char[] bytes = new char[4];  for (int i = 3; i > -1; --i) {  bytes[3 - i] = (char) (x >> (i \* 8) & 0xff);  }  return new String(bytes);  }  // Encodes a tree to a single string.  public String serialize(TreeNode root) {  StringBuilder sb = new StringBuilder();  postorder(root, sb);  return sb.toString();  }  // Decodes list to tree.  public TreeNode helper(Integer lower, Integer upper, ArrayDeque<Integer> nums) {  if (nums.isEmpty())  return null;  int val = nums.getLast();  if (val < lower || val > upper)  return null;  nums.removeLast();  TreeNode root = new TreeNode(val);  root.right = helper(val, upper, nums);  root.left = helper(lower, val, nums);  return root;  }  // Decodes bytes string to integer  public int stringToInt(String bytesStr) {  int result = 0;  for (char b : bytesStr.toCharArray()) {  result = (result << 8) + (int) b;  }  return result;  }  // Decodes your encoded data to tree.  public TreeNode deserialize(String data) {  ArrayDeque<Integer> nums = new ArrayDeque<Integer>();  int n = data.length();  for (int i = 0; i < (int) (n / 4); ++i) {  nums.add(stringToInt(data.substring(4 \* i, 4 \* i + 4)));  }  return helper(Integer.MIN\_VALUE, Integer.MAX\_VALUE, nums);  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) both for serialization and deserialization.
* Space complexity : \mathcal{O}(N)O(*N*), since we store the entire tree. Encoded string: no delimiters, no additional space for the tree structure, just 4 N4*N* bytes for the node values in the encoded string.

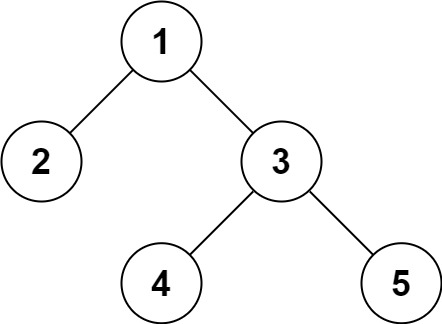
**Serialize and Deserialize Binary Tree**

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

**Clarification:** The input/output format is the same as [how LeetCode serializes a binary tree](https://leetcode.com/faq/#binary-tree). You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

**Example 1:**



**Input:** root = [1,2,3,null,null,4,5]

**Output:** [1,2,3,null,null,4,5]

**Example 2:**

**Input:** root = []

**Output:** []

**Example 3:**

**Input:** root = [1]

**Output:** [1]

**Example 4:**

**Input:** root = [1,2]

**Output:** [1,2]

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* -1000 <= Node.val <= 1000

Medium Google FB MS

**Implement Trie (Prefix Tree)**

Implement a trie with insert, search, and startsWith methods.

**Example:**

Trie trie = new Trie();

trie.insert("apple");

trie.search("apple"); // returns true

trie.search("app"); // returns false

trie.startsWith("app"); // returns true

trie.insert("app");

trie.search("app"); // returns true

**Note:**

* You may assume that all inputs are consist of lowercase letters a-z.
* All inputs are guaranteed to be non-empty strings.

Tries

**LRU Cache**

Design a data structure that follows the constraints of a [**Least Recently Used (LRU) cache**](https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU).

Implement the LRUCache class:

* LRUCache(int capacity) Initialize the LRU cache with **positive** size capacity.
* int get(int key) Return the value of the key if the key exists, otherwise return -1.
* void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, **evict** the least recently used key.

**Follow up:**  
Could you do get and put in O(1) time complexity?

**Example 1:**

**Input**

["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]

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

**Output**

[null, null, null, 1, null, -1, null, -1, 3, 4]

**Explanation**

LRUCache lRUCache = new LRUCache(2);

lRUCache.put(1, 1); // cache is {1=1}

lRUCache.put(2, 2); // cache is {1=1, 2=2}

lRUCache.get(1); // return 1

lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}

lRUCache.get(2); // returns -1 (not found)

lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}

lRUCache.get(1); // return -1 (not found)

lRUCache.get(3); // return 3

lRUCache.get(4); // return 4

**Constraints:**

* 1 <= capacity <= 3000
* 0 <= key <= 3000
* 0 <= value <= 104
* At most 3 \* 104 calls will be made to get and put.

Hard. Google FB

## Others

Here are some other questions for you to practice which are not in the previous chapters, so you can prepare for your Microsoft interview. We recommend Find the Celebrity, Excel Sheet Column Number and The Skyline Problem.

**Single Number**

Given a **non-empty** array of integers nums, every element appears *twice* except for one. Find that single one.

**Follow up:** Could you implement a solution with a linear runtime complexity and without using extra memory?

**Example 1:**

**Input:** nums = [2,2,1]

**Output:** 1

**Example 2:**

**Input:** nums = [4,1,2,1,2]

**Output:** 4

**Example 3:**

**Input:** nums = [1]

**Output:** 1

**Constraints:**

* 1 <= nums.length <= 3 \* 104
* -3 \* 104 <= nums[i] <= 3 \* 104
* Each element in the array appears twice except for one element which appears only once.

HashTable

**Roman to Integer**

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

For example, 2 is written as II in Roman numeral, just two one's added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

**Example 1:**

**Input:** s = "III"

**Output:** 3

**Example 2:**

**Input:** s = "IV"

**Output:** 4

**Example 3:**

**Input:** s = "IX"

**Output:** 9

**Example 4:**

**Input:** s = "LVIII"

**Output:** 58

**Explanation:** L = 50, V= 5, III = 3.

**Example 5:**

**Input:** s = "MCMXCIV"

**Output:** 1994

**Explanation:** M = 1000, CM = 900, XC = 90 and IV = 4.

**Constraints:**

* 1 <= s.length <= 15
* s contains only the characters ('I', 'V', 'X', 'L', 'C', 'D', 'M').
* It is **guaranteed** that s is a valid roman numeral in the range [1, 3999].

EZ

**Excel Sheet Column Number**

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

...

**Example 1:**

**Input:** "A"

**Output:** 1

**Example 2:**

**Input:** "AB"

**Output:** 28

**Example 3:**

**Input:** "ZY"

**Output:** 701

**Constraints:**

* 1 <= s.length <= 7
* s consists only of uppercase English letters.
* s is between "A" and "FXSHRXW".

Medium

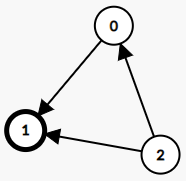
**Find the Celebrity**

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 - 1 people 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 about 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). 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.

**Example 1:**

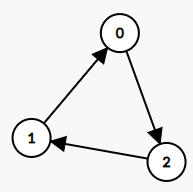


**Input:** graph = [[1,1,0],[0,1,0],[1,1,1]]

**Output:** 1

**Explanation:** There are three persons labeled with 0, 1 and 2. graph[i][j] = 1 means person i knows person j, otherwise graph[i][j] = 0 means person i does not know person j. The celebrity is the person labeled as 1 because both 0 and 2 know him but 1 does not know anybody.

**Example 2:**



**Input:** graph = [[1,0,1],[1,1,0],[0,1,1]]

**Output:** -1

**Explanation:** There is no celebrity.

**Constraints:**

* n == graph.length
* n == graph[i].length
* 2 <= n <= 100
* graph[i][j] is 0 or 1.
* graph[i][i] == 1

**Follow up:** If the maximum number of allowed calls to the API knows is 3 \* n, could you find a solution without exceeding the maximum number of calls?

Medium

**Integer to English Words**

Convert a non-negative integer num to its English words representation.

**Example 1:**

**Input:** num = 123

**Output:** "One Hundred Twenty Three"

**Example 2:**

**Input:** num = 12345

**Output:** "Twelve Thousand Three Hundred Forty Five"

**Example 3:**

**Input:** num = 1234567

**Output:** "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"

**Example 4:**

**Input:** num = 1234567891

**Output:** "One Billion Two Hundred Thirty Four Million Five Hundred Sixty Seven Thousand Eight Hundred Ninety One"

**Constraints:**

* 0 <= num <= 231 - 1

FB/MS/AMZ/Yelp

**The Skyline Problem**

A city's **skyline** is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Given the locations and heights of all the buildings, return *the****skyline****formed by these buildings collectively*.

The geometric information of each building is given in the array buildings where buildings[i] = [lefti, righti, heighti]:

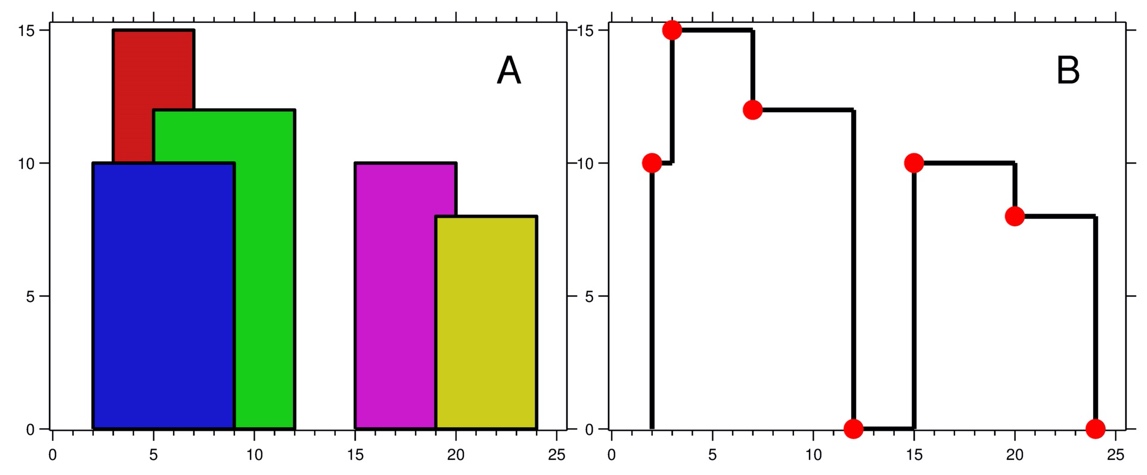
* lefti is the x coordinate of the left edge of the ith building.
* righti is the x coordinate of the right edge of the ith building.
* heighti is the height of the ith building.

You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

The **skyline** should be represented as a list of "key points" **sorted by their x-coordinate** in the form [[x1,y1],[x2,y2],...]. Each key point is the left endpoint of some horizontal segment in the skyline except the last point in the list, which always has a y-coordinate 0 and is used to mark the skyline's termination where the rightmost building ends. Any ground between the leftmost and rightmost buildings should be part of the skyline's contour.

**Note:** There must be no consecutive horizontal lines of equal height in the output skyline. For instance, [...,[2 3],[4 5],[7 5],[11 5],[12 7],...] is not acceptable; the three lines of height 5 should be merged into one in the final output as such: [...,[2 3],[4 5],[12 7],...]

**Example 1:**



**Input:** buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]

**Output:** [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]

**Explanation:**

Figure A shows the buildings of the input.

Figure B shows the skyline formed by those buildings. The red points in figure B represent the key points in the output list.

**Example 2:**

**Input:** buildings = [[0,2,3],[2,5,3]]

**Output:** [[0,3],[5,0]]

**Constraints:**

* 1 <= buildings.length <= 104
* 0 <= lefti < righti <= 231 - 1
* 1 <= heighti <= 231 - 1
* buildings is sorted by lefti in non-decreasing order.

Recursion 2

Given employees and thier schedules find the common free time.  
Table:  
Emp\_ID Start\_time End\_Time  
1 1 2  
1 4 5  
2 3 6  
3 1 3

Segment ?