**Google Interview**

Introduction

Google tech interviews are notoriously difficult and quite challenging. To get a phone screen, you will need to submit your resume to their online application system or via an internal referral from a Googler.

Assuming you passed their resume screen, a recruiter will reach out to you. Usually there will be two phone screens, and if you do well, you'll be invited to onsite interviews.

Since Google operates at a large scale, be prepared to answer lots of follow up questions on how to scale the algorithm you wrote for multiple machines. Some examples are: [Number of Islands](https://leetcode.com/problems/number-of-islands) and [Intersection of Two Arrays II](https://leetcode.com/problems/intersection-of-two-arrays-ii/description/).

Interview Process

 Unique Email Addresses

 Odd Even Jump

 License Key Formatting

 Fruit Into Baskets

 Google Phone Interview

 Google Onsite Interview

 Google Hiring Committee

 Google Offer Review

Arrays and Strings

 Longest Substring Without Repeating Characters

 Container With Most Water

 3Sum

 Next Permutation

 Multiply Strings

 Rotate Image

 Jump Game

 Plus One

 Minimum Window Substring

 Read N Characters Given Read4 II - Call multiple times

 Longest Substring with At Most Two Distinct Characters

 Missing Ranges

 Next Closest Time

 Expressive Words

 Find And Replace in String

 Maximize Distance to Closest Person

 Valid Parentheses

 Merge k Sorted Lists

 Trapping Rain Water

 Kth Largest Element in an Array

 Meeting Rooms II

 Backspace String Compare

 Minimum Cost to Hire K Workers

 K Closest Points to Origin

Linked Lists

 Add Two Numbers

 Remove Nth Node From End of List

 Merge Two Sorted Lists

 Copy List with Random Pointer

Trees and Graphs

 Binary Tree Maximum Path Sum

 Word Ladder

 Number of Islands

 Course Schedule II

 Count Complete Tree Nodes

 Longest Increasing Path in a Matrix

 Decode String

 Evaluate Division

 Diameter of Binary Tree

 Cracking the Safe

 Robot Room Cleaner

 Most Stones Removed with Same Row or Column

 Flip Equivalent Binary Trees

Recursion

 Word Squares

 Strobogrammatic Number II

 Word Search II

 Android Unlock Patterns

 Letter Combinations of a Phone Number

 Generate Parentheses

Sorting and Searching

 Median of Two Sorted Arrays

 Find First and Last Position of Element in Sorted Array

 Merge Intervals

 Insert Interval

 Valid Anagram

 Count of Smaller Numbers After Self

 Peak Index in a Mountain Array

Dynamic Programming

 Longest Palindromic Substring

 Maximum Subarray

 Best Time to Buy and Sell Stock

 Maximum Product Subarray

 Coin Change

 Split Array Largest Sum

Design

 LRU Cache

 Min Stack

 Serialize and Deserialize Binary Tree

 Logger Rate Limiter

 Insert Delete GetRandom O(1)

 Design Search Autocomplete System

Others

 Reverse Integer

 Candy

 Isomorphic Strings

 Strobogrammatic Number

 Bulls and Cows

 Range Sum Query 2D - Mutable

 My Calendar II

 Jewels and Stones

 Swap Adjacent in LR String

 Guess the Word

 Minimum Area Rectangle

## Interview Process

You may receive an online assessment link as your first step of interview process. The assessment will expire within 7 days and contains two coding questions to be completed within an hour. Below are some Online Assessment questions for you to practice.

Near the end of this chapter we provide more details of the different stages of a Google interview.

**Unique Email Addresses**

Every email consists of a local name and a domain name, separated by the @ sign.

For example, in alice@leetcode.com, alice is the local name, and leetcode.com is the domain name.

Besides lowercase letters, these emails may contain '.'s or '+'s.

If you add periods ('.') between some characters in the **local name** part of an email address, mail sent there will be forwarded to the same address without dots in the local name.  For example, "alice.z@leetcode.com" and "alicez@leetcode.com" forward to the same email address.  (Note that this rule does not apply for domain names.)

If you add a plus ('+') in the **local name**, everything after the first plus sign will be **ignored**. This allows certain emails to be filtered, for example m.y+name@email.com will be forwarded to my@email.com.  (Again, this rule does not apply for domain names.)

It is possible to use both of these rules at the same time.

Given a list of emails, we send one email to each address in the list.  How many different addresses actually receive mails?

**Example 1:**

**Input:** ["test.email+alex@leetcode.com","test.e.mail+bob.cathy@leetcode.com","testemail+david@lee.tcode.com"]

**Output:** 2

**Explanation:** "testemail@leetcode.com" and "testemail@lee.tcode.com" actually receive mails

**Note:**

* 1 <= emails[i].length <= 100
* 1 <= emails.length <= 100
* Each emails[i] contains exactly one '@' character.
* All local and domain names are non-empty.
* Local names do not start with a '+' character.

## Solution

#### **Approach 1: Canonical Form**

**Intuition and Algorithm**

For each email address, convert it to the canonical address that actually receives the mail. This involves a few steps:

* Separate the email address into a local part and the rest of the address.
* If the local part has a '+' character, remove it and everything beyond it from the local part.
* Remove all the dots from the local part.
* The canonical address is local + rest.

After, we can count the number of unique canonical addresses with a Set structure.

|  |
| --- |
| class Solution {  public int numUniqueEmails(String[] emails) {  Set<String> seen = new HashSet();  for (String email : emails) {  int i = email.indexOf('@');  String local = email.substring(0, i);  String rest = email.substring(i);  if (local.contains("+")) {  local = local.substring(0, local.indexOf('+'));  }  // Note: one should escape the specific character '.',  // since it is treated as a regex expression.  local = local.replaceAll("\\.", "");  seen.add(local + rest);  }  return seen.size();  }  } |

**Complexity Analysis**

* Time Complexity: O(\mathcal{C})*O*(C), where \mathcal{C}C is the total content of emails.
* Space Complexity: O(\mathcal{C})*O*(C).

**Odd Even Jump**

You are given an integer array A. From some starting index, you can make a series of jumps. The (1st, 3rd, 5th, ...) jumps in the series are called **odd-numbered jumps**, and the (2nd, 4th, 6th, ...) jumps in the series are called **even-numbered jumps**. Note that the **jumps** are numbered, not the indices.

You may jump forward from index i to index j (with i < j) in the following way:

* During **odd-numbered jumps** (i.e., jumps 1, 3, 5, ...), you jump to the index j such that A[i] <= A[j] and A[j] is the smallest possible value. If there are multiple such indices j, you can only jump to the **smallest** such index j.
* During **even-numbered jumps** (i.e., jumps 2, 4, 6, ...), you jump to the index j such that A[i] >= A[j] and A[j] is the largest possible value. If there are multiple such indices j, you can only jump to the **smallest** such index j.
* It may be the case that for some index i, there are no legal jumps.

A starting index is **good** if, starting from that index, you can reach the end of the array (index A.length - 1) by jumping some number of times (possibly 0 or more than once).

Return *the number of****good****starting indices*.

**Example 1:**

**Input:** A = [10,13,12,14,15]

**Output:** 2

**Explanation:**

From starting index i = 0, we can make our 1st jump to i = 2 (since A[2] is the smallest among A[1], A[2], A[3],

A[4] that is greater or equal to A[0]), then we cannot jump any more.

From starting index i = 1 and i = 2, we can make our 1st jump to i = 3, then we cannot jump any more.

From starting index i = 3, we can make our 1st jump to i = 4, so we have reached the end.

From starting index i = 4, we have reached the end already.

In total, there are 2 different starting indices i = 3 and i = 4, where we can reach the end with some number of

jumps.

**Example 2:**

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

**Output:** 3

**Explanation:**

From starting index i = 0, we make jumps to i = 1, i = 2, i = 3:

During our 1st jump (odd-numbered), we first jump to i = 1 because A[1] is the smallest value in [A[1], A[2],

A[3], A[4]] that is greater than or equal to A[0].

During our 2nd jump (even-numbered), we jump from i = 1 to i = 2 because A[2] is the largest value in [A[2], A[3],

A[4]] that is less than or equal to A[1]. A[3] is also the largest value, but 2 is a smaller index, so we can

only jump to i = 2 and not i = 3

During our 3rd jump (odd-numbered), we jump from i = 2 to i = 3 because A[3] is the smallest value in [A[3], A[4]]

that is greater than or equal to A[2].

We can't jump from i = 3 to i = 4, so the starting index i = 0 is not good.

In a similar manner, we can deduce that:

From starting index i = 1, we jump to i = 4, so we reach the end.

From starting index i = 2, we jump to i = 3, and then we can't jump anymore.

From starting index i = 3, we jump to i = 4, so we reach the end.

From starting index i = 4, we are already at the end.

In total, there are 3 different starting indices i = 1, i = 3, and i = 4, where we can reach the end with some

number of jumps.

**Example 3:**

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

**Output:** 3

**Explanation:**

We can reach the end from starting indices 1, 2, and 4.

**Constraints:**

* 1 <= A.length <= 2 \* 104
* 0 <= A[i] < 105

## Solution

#### **Approach 1: Monotonic Stack**

**Intuition**

First, we notice that where you jump to is determined only by the state of your current index and the jump number parity.

For each state, there is exactly one state you could jump to (or you can't jump.) If we somehow knew these jumps, we could solve the problem by a simple traversal.

So the problem reduces to solving this question: for some index i during an odd numbered jump, what index do we jump to (if any)? The question for even-numbered jumps is similar.

**Algorithm**

Let's figure out where index i jumps to, assuming this is an odd-numbered jump.

Let's consider each value of A in order from smallest to largest. When we consider a value A[j] = v, we search the values we have already processed (which are <= v) from largest to smallest. If we find that we have already processed some value v0 = A[i] with i < j, then we know i jumps to j.

Naively this is a little slow, but we can speed this up with a common trick for harder problems: a monotonic stack. (For another example of this technique, please see the solution to this problem: [(Article - Sum of Subarray Minimums)](https://leetcode.com/articles/sum-of-subarray-minimums/))

Let's store the indices i of the processed values v0 = A[i] in a stack, and maintain the invariant that this is monotone decreasing. When we add a new index j, we pop all the smaller indices i < j from the stack, which all jump to j.

Afterwards, we know oddnext[i], the index where i jumps to if this is an odd numbered jump. Similarly, we know evennext[i]. We can use this information to quickly build out all reachable states using dynamic programming.

|  |
| --- |
| class Solution(object):  def oddEvenJumps(self, A):  N = len(A)  def make(B):  ans = [None] \* N  stack = [] # invariant: stack is decreasing  for i in B:  while stack and i > stack[-1]:  ans[stack.pop()] = i  stack.append(i)  return ans  B = sorted(range(N), key = lambda i: A[i])  oddnext = make(B)  B.sort(key = lambda i: -A[i])  evennext = make(B)  odd = [False] \* N  even = [False] \* N  odd[N-1] = even[N-1] = True  for i in xrange(N-2, -1, -1):  if oddnext[i] is not None:  odd[i] = even[oddnext[i]]  if evennext[i] is not None:  even[i] = odd[evennext[i]]  return sum(odd) |

**Complexity Analysis**

* Time Complexity: O(N \log N)*O*(*N*log*N*), where N*N* is the length of A.
* Space Complexity: O(N)*O*(*N*).

#### **Approach 2: Tree Map**

**Intuition**

As in Approach 1, the problem reduces to solving this question: for some index i during an odd numbered jump, what index do we jump to (if any)?

**Algorithm**

We can use a TreeMap, which is an excellent structure for maintaining sorted data. Our map vals will map values v = A[i] to indices i.

Iterating from i = N-2 to i = 0, we have some value v = A[i] and we want to know what the next largest or next smallest value is. The TreeMap.lowerKey and TreeMap.higherKey functions do this for us.

With this in mind, the rest of the solution is straightforward: we use dynamic programming to maintain odd[i] and even[i]: whether the state of being at index i on an odd or even numbered jump is possible to reach.

|  |
| --- |
| class Solution {  public int oddEvenJumps(int[] A) {  int N = A.length;  if (N <= 1) return N;  boolean[] odd = new boolean[N];  boolean[] even = new boolean[N];  odd[N-1] = even[N-1] = true;  TreeMap<Integer, Integer> vals = new TreeMap();  vals.put(A[N-1], N-1);  for (int i = N-2; i >= 0; --i) {  int v = A[i];  if (vals.containsKey(v)) {  odd[i] = even[vals.get(v)];  even[i] = odd[vals.get(v)];  } else {  Integer lower = vals.lowerKey(v);  Integer higher = vals.higherKey(v);  if (lower != null)  even[i] = odd[vals.get(lower)];  if (higher != null) {  odd[i] = even[vals.get(higher)];  }  }  vals.put(v, i);  }  int ans = 0;  for (boolean b: odd)  if (b) ans++;  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N \log N)*O*(*N*log*N*), where N*N* is the length of A.
* Space Complexity: O(N)*O*(*N*).

**License Key Formatting**

You are given a license key represented as a string S which consists only alphanumeric character and dashes. The string is separated into N+1 groups by N dashes.

Given a number K, we would want to reformat the strings such that each group contains *exactly* K characters, except for the first group which could be shorter than K, but still must contain at least one character. Furthermore, there must be a dash inserted between two groups and all lowercase letters should be converted to uppercase.

Given a non-empty string S and a number K, format the string according to the rules described above.

**Example 1:**

**Input:** S = "5F3Z-2e-9-w", K = 4

**Output:** "5F3Z-2E9W"

**Explanation:** The string S has been split into two parts, each part has 4 characters.

Note that the two extra dashes are not needed and can be removed.

**Example 2:**

**Input:** S = "2-5g-3-J", K = 2

**Output:** "2-5G-3J"

**Explanation:** The string S has been split into three parts, each part has 2 characters except the first part as it could be shorter as mentioned above.

**Note:**

1. The length of string S will not exceed 12,000, and K is a positive integer.
2. String S consists only of alphanumerical characters (a-z and/or A-Z and/or 0-9) and dashes(-).
3. String S is non-empty.

**Fruit Into Baskets**

In a row of trees, the i-th tree produces fruit with type tree[i].

You **start at any tree of your choice**, then repeatedly perform the following steps:

1. Add one piece of fruit from this tree to your baskets.  If you cannot, stop.
2. Move to the next tree to the right of the current tree.  If there is no tree to the right, stop.

Note that you do not have any choice after the initial choice of starting tree: you must perform step 1, then step 2, then back to step 1, then step 2, and so on until you stop.

You have two baskets, and each basket can carry any quantity of fruit, but you want each basket to only carry one type of fruit each.

What is the total amount of fruit you can collect with this procedure?

**Example 1:**

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

**Output:** 3

**Explanation:** We can collect [1,2,1].

**Example 2:**

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

**Output:** 3

**Explanation:** We can collect [1,2,2].

If we started at the first tree, we would only collect [0, 1].

**Example 3:**

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

**Output:** 4

**Explanation:** We can collect [2,3,2,2].

If we started at the first tree, we would only collect [1, 2].

**Example 4:**

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

**Output:** 5

**Explanation:** We can collect [1,2,1,1,2].

If we started at the first tree or the eighth tree, we would only collect 4 fruits.

**Note:**

1. 1 <= tree.length <= 40000
2. 0 <= tree[i] < tree.length

## Solution

#### **Approach 1: Scan Through Blocks**

**Intuition**

Equivalently, we want the longest subarray with at most two "types" (values of tree[i]).

Instead of considering each element individually, we can consider blocks of adjacent elements of the same type.

For example, instead of tree = [1, 1, 1, 1, 2, 2, 3, 3, 3], we can say this is blocks = [(1, weight = 4), (2, weight = 2), (3, weight = 3)].

Now say we brute forced, scanning from left to right. We'll have something like blocks = [1, \_2\_, 1, 2, 1, 2, \_1\_, 3, ...] (with various weights).

The key insight is that when we encounter a 3, we do not need to start from the second element 2 (marked \_2\_ for convenience); we can start from the first element (\_1\_) before the 3. This is because if we started two or more elements before, the sequence must have types 1 and 2, and that sequence is going to end at the 3, and thus be shorter than anything we've already considered.

Since every starting point (that is the left-most index of a block) was considered, this solution is correct.

**Algorithm**

As the notation and strategy around implementing this differs between Python and Java, please see the inline comments for more details.

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of tree.
* Space Complexity: O(N)*O*(*N*).

#### **Approach 2: Sliding Window**

**Intuition**

As in Approach 1, we want the longest subarray with at most two different "types" (values of tree[i]). Call these subarrays valid.

Say we consider all valid subarrays that end at index j. There must be one with the smallest possible starting index i: lets say opt(j) = i.

Now the key idea is that opt(j) is a monotone increasing function. This is because any subarray of a valid subarray is valid.

**Algorithm**

Let's perform a sliding window, keeping the loop invariant that i will be the smallest index for which [i, j] is a valid subarray.

We'll maintain count, the count of all the elements in the subarray. This allows us to quickly query whether there are 3 types in the subarray or not.

|  |
| --- |
| class Solution {  public int totalFruit(int[] tree) {  int ans = 0, i = 0;  Counter count = new Counter();  for (int j = 0; j < tree.length; ++j) {  count.add(tree[j], 1);  while (count.size() >= 3) {  count.add(tree[i], -1);  if (count.get(tree[i]) == 0)  count.remove(tree[i]);  i++;  }  ans = Math.max(ans, j - i + 1);  }  return ans;  }  }  class Counter extends HashMap<Integer, Integer> {  public int get(int k) {  return containsKey(k) ? super.get(k) : 0;  }  public void add(int k, int v) {  put(k, get(k) + v);  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of tree.
* Space Complexity: O(N)*O*(*N*).

**Google Phone Interview**

The phone screen usually last between 30 and 60 minutes.

Your phone interview will cover data structures and algorithms. Be prepared to write around 20-30 lines of code in your strongest language. This should be clean, rich, robust code.

Your interviewer will ask one or two coding questions with lots of follow up questions. Be prepared to talk through your thought process while writing code in a Google Doc that you’ll share with your interviewer. We recommend using a hands-free headset or speakerphone so you can type freely.

1. You will be asked an open ended question. Ask clarifying questions, devise requirements.
2. You will be asked to explain it in an algorithm.
3. Convert it to a workable code.  
   (Hint: Don't worry about getting it perfect because time is limited. Write what comes but then refine it later. Also make sure you consider corner cases and edge cases, production ready.)
4. Optimize the code, follow it with test cases and find any bugs.

**Google Onsite Interview**

This is it! You've passed the phone interviews and now you have received an onsite interview invitation. Onsite interview is the most important process of a Google interview, and your ability to receive an offer will largely depend on your performance during onsite interviews. Traditionally, the percentage of candidates who will receive an offer after onsite interview is about [1 of 5 to 1 of 7](https://www.quora.com/What-percentage-of-candidates-convert-a-Google-onsite-interview-into-an-offer).

You'll usually meet with five Googlers for about 30 to 45 minutes each, and one of them will be your lunch interviewer who does not submit interview feedback, so take this opportunity to freely ask questions.

All candidates will have the chance to highlight strengths in four different areas:

* General cognitive ability: Open-ended questions are asked to learn how you approach and solve problems. And there’s no one right answer—your ability to explain your thought process and how you use data to inform decisions is what’s most important.
* Leadership: Be prepared to discuss how you have used your communication and decision-making skills to mobilize others. This might be by stepping up to a leadership role at work or with an organization, or by helping a team succeed even when you weren’t officially the leader.
* Role-related knowledge: Google is interested in how your individual strengths combine with your experience to drive impact. Google don’t just look for how you can contribute today, but how you can grow into different roles—including ones that haven’t even been invented yet.
* Googleyness: Share how you work individually and on a team, how you help others, how you navigate ambiguity, and how you push yourself to grow outside of your comfort zone.

Note that Google put heavy emphasis on your analytical ability *regardless* of your job experience. Be prepared to showcase solid understanding of data structures and algorithms. If you have years of experiences in the industry and are rusty in your Computer Science fundamentals, invest more time in brushing up your basic CS fundamental skills. Otherwise, you may [end up being rejected like Max Howell](https://www.quora.com/Whats-the-logic-behind-Google-rejecting-Max-Howell-the-author-of-Homebrew-for-not-being-able-to-invert-a-binary-tree/answer/Max-Howell) for not able to [invert a binary tree](https://leetcode.com/problems/invert-binary-tree).

Google wants to understand your coding skills and technical areas of expertise, including tools or programming languages and general knowledge on topics like data structures and algorithms. There's generally some back and forth in these discussions, just like there is on the job to push each other's thinking and learn about different approaches. So be prepared to talk through your solutions in depth. Push your own boundaries and find the best answer—that’s probably how you work anyway.

Technical onsite interviews at Google were historically conducted on whiteboards, but to provide a more authentic coding experience that’s less time-consuming, Google has started to offer laptops for coding interviews in some sites. These chromebooks have an interview app that let's you choose a coding language of your preference.

Throughout the interview process, feel free to ask your interviewers for clarification to make sure you fully understand their questions.

**Google Hiring Committee**

After the onsite interview, each of your interviewers (except the lunch interviewer) will have to submit detailed interview feedback. Each interviewer's feedback is kept hidden from other interviewers to ensure one's feedback does not influence the others. Each interviewer will assign a score and make a Hire / No Hire decision.

Assume the scores from the feedback are good enough, your whole packet will be moved forward to the hiring committee for review.

The hiring committee will review all submitted feedback thoroughly and make a hire / no hire recommendation. For more details about this process, you can [read here](https://www.quora.com/What-happens-during-a-Google-hiring-committee-meeting).

**Google Offer Review**

If Hiring Committee recommends hire, congratulations! You have arrived at the final stage of the process. There are about [10-12% candidates not extended offers](https://www.quora.com/My-Google-recruiter-has-asked-me-about-my-current-compensation-and-external-references-Whats-the-probability-of-not-getting-an-offer-from-this-point/answer/Bob-See) at this stage. Recruiter requests for External References and Compensation History, conducts reference checks, and prepares a packet for the Offer Review Process.

**The Offer Review process:**

1. Pre-Review Committee
2. Compensation Committee Review
3. Senior VP Review

There is a small percentage chance of [the offer not getting approved at the final SVP review](https://www.quora.com/Why-would-one-be-rejected-at-the-final-stage-at-Google/answer/Bob-See).

## Arrays and Strings

String manipulation problems are in the same category as arrays, because internally, a string is represented as an array of characters. Array problems usually do not require knowledge of advanced data structures, so just basic data structures such as [Hash Tables](https://leetcode.com/articles/hash-table/) and basic techniques like [Two Pointers](https://leetcode.com/articles/two-pointer-technique/) should suffice.

Google likes to test your ability to think at large scale by asking variation of problems represented in a [data stream model](https://en.wikipedia.org/wiki/Streaming_algorithm#Data_stream_model). For example, instead of giving you an integer array, you are given a stream of integers and all integers are too large to fit in memory. A great example of such problem, which can be represented in a data stream model, is the [Longest Substring with At Most K Distinct Characters](https://leetcode.com/problems/longest-substring-with-at-most-k-distinct-characters).

**Longest Substring Without Repeating Characters**

Given a string s, find the length of the **longest substring** without repeating characters.

**Example 1:**

**Input:** s = "abcabcbb"

**Output:** 3

**Explanation:** The answer is "abc", with the length of 3.

**Example 2:**

**Input:** s = "bbbbb"

**Output:** 1

**Explanation:** The answer is "b", with the length of 1.

**Example 3:**

**Input:** s = "pwwkew"

**Output:** 3

**Explanation:** The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

**Example 4:**

**Input:** s = ""

**Output:** 0

**Constraints:**

* 0 <= s.length <= 5 \* 104
* s consists of English letters, digits, symbols and spaces.

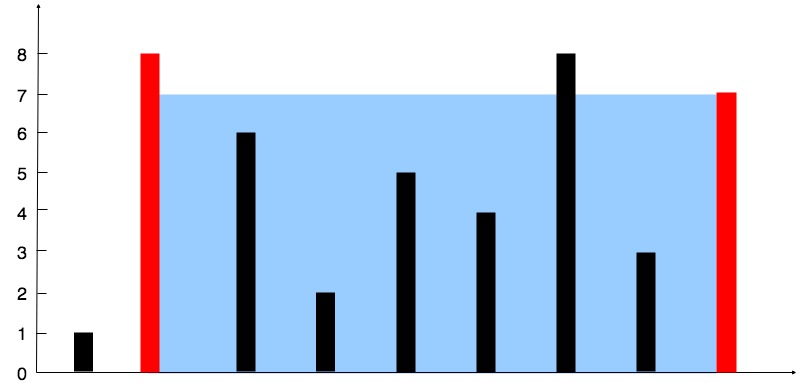
--- check HashTable

**Container With Most Water**

Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of the line i is at (i, ai) and (i, 0). Find two lines, which, together with the x-axis forms a container, such that the container contains the most water.

**Notice** that you may not slant the container.

**Example 1:**



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

**Output:** 49

**Explanation:** The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

**Example 2:**

**Input:** height = [1,1]

**Output:** 1

**Example 3:**

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

**Output:** 16

**Example 4:**

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

**Output:** 2

-- check hard question

**3Sum**

Given an array nums of *n* integers, are there elements *a*, *b*, *c* in nums such that *a* + *b* + *c* = 0? Find all unique triplets in the array which gives the sum of zero.

Notice that the solution set must not contain duplicate triplets.

**Example 1:**

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

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

**Example 2:**

**Input:** nums = []

**Output:** []

**Example 3:**

**Input:** nums = [0]

**Output:** []

**Constraints:**

* 0 <= nums.length <= 3000
* -105 <= nums[i] <= 105

Check hard

**Next Permutation**

Implement **next permutation**, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such an arrangement is not possible, it must rearrange it as the lowest possible order (i.e., sorted in ascending order).

The replacement must be [**in place**](http://en.wikipedia.org/wiki/In-place_algorithm) and use only constant extra memory.

**Example 1:**

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

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

**Example 2:**

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

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

**Example 3:**

**Input:** nums = [1,1,5]

**Output:** [1,5,1]

**Example 4:**

**Input:** nums = [1]

**Output:** [1]

**Constraints:**

* 1 <= nums.length <= 100
* 0 <= nums[i] <= 100

## Summary

We need to find the next lexicographic permutation of the given list of numbers than the number formed by the given array.

## Solution

#### **Approach 1: Brute Force**

**Algorithm**

In this approach, we find out every possible permutation of list formed by the elements of the given array and find out the permutation which is just larger than the given one. But this one will be a very naive approach, since it requires us to find out every possible permutation which will take really long time and the implementation is complex. Thus, this approach is not acceptable at all. Hence, we move on directly to the correct approach.

**Complexity Analysis**

* Time complexity : O(n!)*O*(*n*!). Total possible permutations is n!*n*!.
* Space complexity : O(n)*O*(*n*). Since an array will be used to store the permutations.

#### **Approach 2: Single Pass Approach**

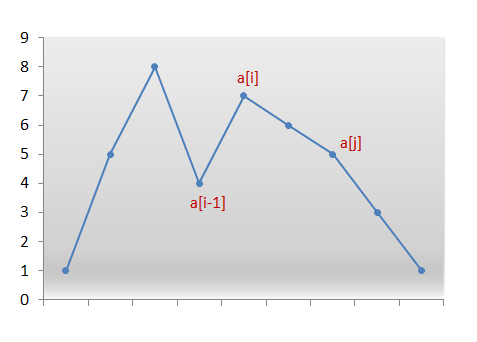
**Algorithm**

First, we observe that for any given sequence that is in descending order, no next larger permutation is possible. For example, no next permutation is possible for the following array:

[9, 5, 4, 3, 1]

We need to find the first pair of two successive numbers a[i]*a*[*i*] and a[i-1]*a*[*i*−1], from the right, which satisfy a[i] > a[i-1]*a*[*i*]>*a*[*i*−1]. Now, no rearrangements to the right of a[i-1]*a*[*i*−1] can create a larger permutation since that subarray consists of numbers in descending order. Thus, we need to rearrange the numbers to the right of a[i-1]*a*[*i*−1] including itself.

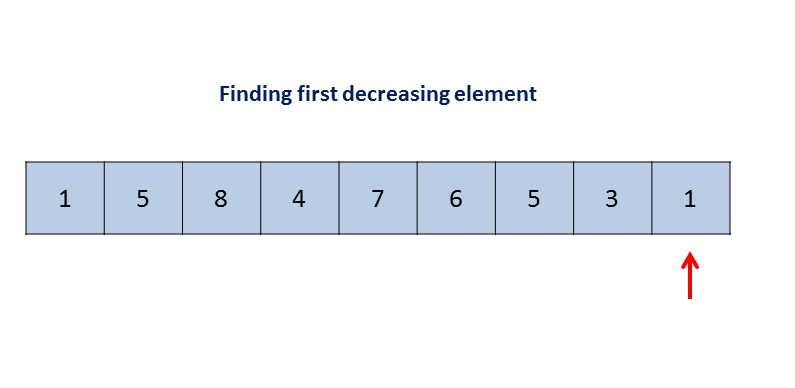
Now, what kind of rearrangement will produce the next larger number? We want to create the permutation just larger than the current one. Therefore, we need to replace the number a[i-1]*a*[*i*−1] with the number which is just larger than itself among the numbers lying to its right section, say a[j]*a*[*j*].



We swap the numbers a[i-1]*a*[*i*−1] and a[j]*a*[*j*]. We now have the correct number at index i-1*i*−1. But still the current permutation isn't the permutation that we are looking for. We need the smallest permutation that can be formed by using the numbers only to the right of a[i-1]*a*[*i*−1]. Therefore, we need to place those numbers in ascending order to get their smallest permutation.

But, recall that while scanning the numbers from the right, we simply kept decrementing the index until we found the pair a[i]*a*[*i*] and a[i-1]*a*[*i*−1] where, a[i] > a[i-1]*a*[*i*]>*a*[*i*−1]. Thus, all numbers to the right of a[i-1]*a*[*i*−1] were already sorted in descending order. Furthermore, swapping a[i-1]*a*[*i*−1] and a[j]*a*[*j*] didn't change that order. Therefore, we simply need to reverse the numbers following a[i-1]*a*[*i*−1] to get the next smallest lexicographic permutation.

The following animation will make things clearer:



|  |
| --- |
| public class Solution {  public void nextPermutation(int[] nums) {  int i = nums.length - 2;  while (i >= 0 && nums[i + 1] <= nums[i]) {  i--;  }  if (i >= 0) {  int j = nums.length - 1;  while (j >= 0 && nums[j] <= nums[i]) {  j--;  }  swap(nums, i, j);  }  reverse(nums, i + 1);  }  private void reverse(int[] nums, int start) {  int i = start, j = nums.length - 1;  while (i < j) {  swap(nums, i, j);  i++;  j--;  }  }  private void swap(int[] nums, int i, int j) {  int temp = nums[i];  nums[i] = nums[j];  nums[j] = temp;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). In worst case, only two scans of the whole array are needed.
* Space complexity : O(1)*O*(1). No extra space is used. In place replacements are done.

**Multiply Strings**

Given two non-negative integers num1 and num2 represented as strings, return the product of num1 and num2, also represented as a string.

**Note:** You must not use any built-in BigInteger library or convert the inputs to integer directly.

**Example 1:**

**Input:** num1 = "2", num2 = "3"

**Output:** "6"

**Example 2:**

**Input:** num1 = "123", num2 = "456"

**Output:** "56088"

**Constraints:**

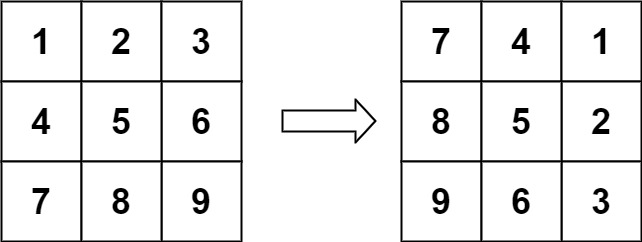
* 1 <= num1.length, num2.length <= 200
* num1 and num2 consist of digits only.
* Both num1 and num2 do not contain any leading zero, except the number 0 itself.

**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.

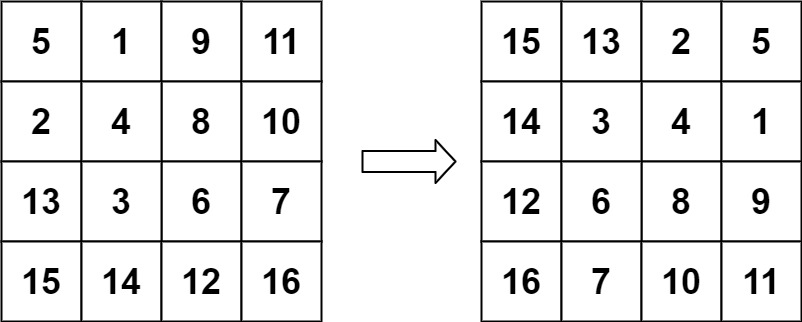
**Example 1:**



**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:**

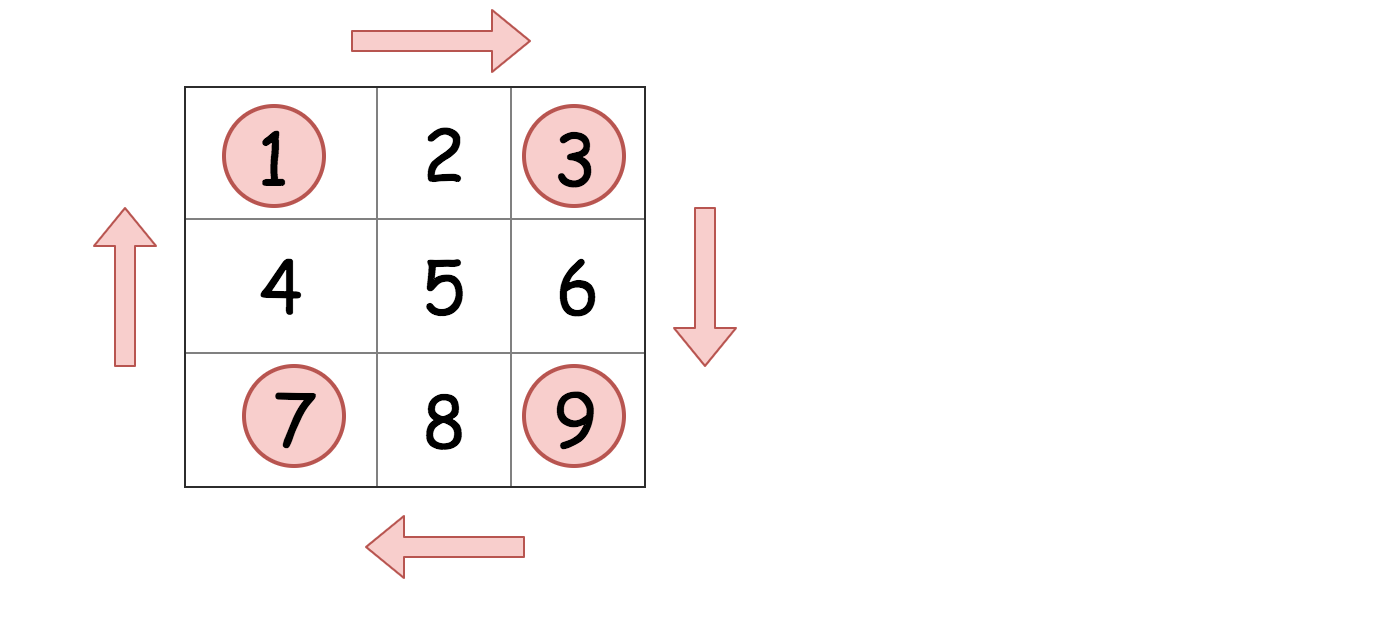
* matrix.length == n
* matrix[i].length == n
* 1 <= n <= 20
* -1000 <= matrix[i][j] <= 1000

## Solution

#### **Approach 1: Rotate Groups of Four Cells**

**Intuition**

Observe how the cells move in groups when we rotate the image.



We can iterate over each group of four cells and rotate them.

**Implementation**

Here is a visualization of the algorithm in action.

|  |
| --- |
| class Solution {  public void rotate(int[][] matrix) {  int n = matrix.length;  for (int i = 0; i < (n + 1) / 2; i ++) {  for (int j = 0; j < n / 2; j++) {  int temp = matrix[n - 1 - j][i];  matrix[n - 1 - j][i] = matrix[n - 1 - i][n - j - 1];  matrix[n - 1 - i][n - j - 1] = matrix[j][n - 1 -i];  matrix[j][n - 1 - i] = matrix[i][j];  matrix[i][j] = temp;  }  }  }  } |

**Complexity Analysis**

Let M*M* be the number of cells in the matrix.

* Time complexity : \mathcal{O}(M)O(*M*), as each cell is getting read once and written once.
* Space complexity : \mathcal{O}(1)O(1) because we do not use any other additional data structures.

#### **Approach 2: Reverse on Diagonal and then Reverse Left to Right**

**Intuition**

The most elegant solution for rotating the matrix is to firstly reverse the matrix around the main diagonal, and then reverse it from left to right. These operations are called **transpose** and **reflect** in linear algebra.

Here is a visualization to help you see why this works.

**Bonus Question:** What would happen if you reflect and *then* transpose? Would you still get the correct answer?

Even though this approach does twice as many reads and writes as approach 1, most people would consider it a better approach because the code is simpler, and it is built with standard matrix operations that can be found in any matrix library.

**Implementation**

|  |
| --- |
| class Solution {  public void rotate(int[][] matrix) {  transpose(matrix);  reflect(matrix);  }    public void transpose(int[][] matrix) {  int n = matrix.length;  for (int i = 0; i < n; i++) {  for (int j = i; j < n; j++) {  int tmp = matrix[j][i];  matrix[j][i] = matrix[i][j];  matrix[i][j] = tmp;  }  }  }    public void reflect(int[][] matrix) {  int n = matrix.length;  for (int i = 0; i < n; i++) {  for (int j = 0; j < n / 2; j++) {  int tmp = matrix[i][j];  matrix[i][j] = matrix[i][n - j - 1];  matrix[i][n - j - 1] = tmp;  }  }  }  } |

**Complexity Analysis**

Let M*M* be the number of cells in the grid.

* Time complexity : \mathcal{O}(M)O(*M*). We perform two steps; transposing the matrix, and then reversing each row. Transposing the matrix has a cost of \mathcal{O}(M)O(*M*) because we're moving the value of each cell once. Reversing each row also has a cost of \mathcal{O}(M)O(*M*), because again we're moving the value of each cell once.
* Space complexity : \mathcal{O}(1)O(1) because we do not use any other additional data structures.

**Bonus Question:** If you're not too confident with matrices and linear algebra, get some more practice by also coding a method that transposes the matrix on the *other* diagonal, and another that reflects from top to bottom. You can test your functions by printing out the matrix before and after each operation. Finally, use your functions to find *three more solutions to this problem*. Each solution uses two matrix operations.

**Interview Tip**: Terrified of being asked this question in an interview? Many people are: it can be intimidating due to the fiddly logic. Unfortunately, if you do a lot of interviewing, the probability of seeing it at least once is high, and some people have claimed to have seen it multiple times! This is one of the few questions where I recommend practicing until you can confidently code it and explain it without thinking too much.

<https://youtu.be/fMSJSS7eO1w>

**Jump Game**

Given an array of non-negative integers nums, you are initially positioned at the **first index** of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

**Example 1:**

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

**Output:** true

**Explanation:** Jump 1 step from index 0 to 1, then 3 steps to the last index.

**Example 2:**

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

**Output:** false

**Explanation:** You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.

**Constraints:**

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

Check medium

**Plus One**

Given a **non-empty** array of decimal digits representing a non-negative integer, increment one to the integer.

The digits are stored such that the most significant digit is at the head of the list, and each element in the array contains a single digit.

You may assume the integer does not contain any leading zero, except the number 0 itself.

**Example 1:**

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

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

**Explanation:** The array represents the integer 123.

**Example 2:**

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

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

**Explanation:** The array represents the integer 4321.

**Example 3:**

**Input:** digits = [0]

**Output:** [1]

**Constraints:**

* 1 <= digits.length <= 100
* 0 <= digits[i] <= 9

Check Array

**Minimum Window Substring**

Given two strings s and t, return *the minimum window in s which will contain all the characters in t*. If there is no such window in s that covers all characters in t, return *the empty string ""*.

**Note** that If there is such a window, it is guaranteed that there will always be only one unique minimum window in s.

**Example 1:**

**Input:** s = "ADOBECODEBANC", t = "ABC"

**Output:** "BANC"

**Example 2:**

**Input:** s = "a", t = "a"

**Output:** "a"

**Constraints:**

* 1 <= s.length, t.length <= 105
* s and t consist of English letters.

**Follow up:** Could you find an algorithm that runs in O(n) time?

## Solution

#### **Approach 1: Sliding Window**

**Intuition**

The question asks us to return the minimum window from the string S*S* which has all the characters of the string T*T*. Let us call a window desirable if it has all the characters from T*T*.

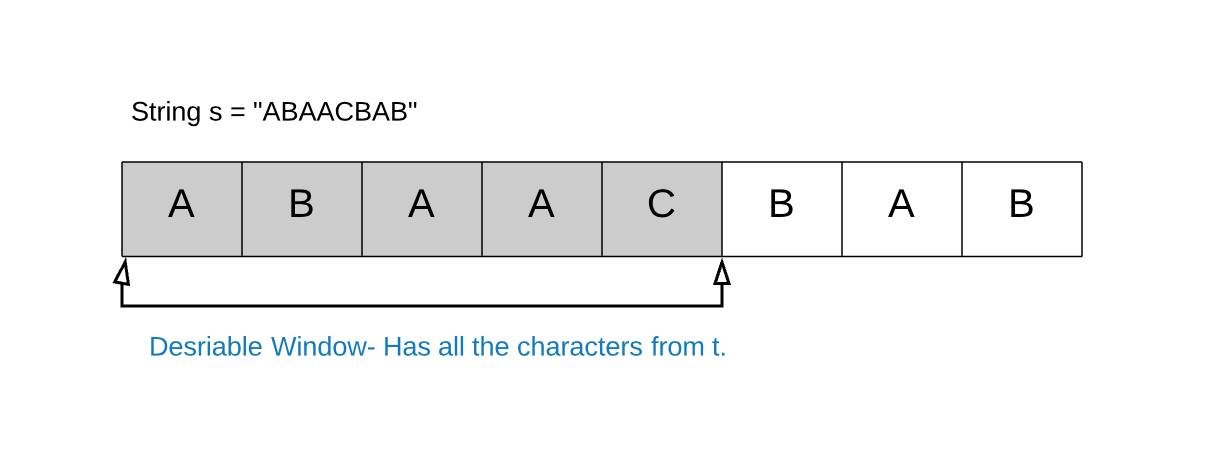
We can use a simple sliding window approach to solve this problem.

In any sliding window based problem we have two pointers. One right*right* pointer whose job is to expand the current window and then we have the left*left* pointer whose job is to contract a given window. At any point in time only one of these pointers move and the other one remains fixed.

The solution is pretty intuitive. We keep expanding the window by moving the right pointer. When the window has all the desired characters, we contract (if possible) and save the smallest window till now.

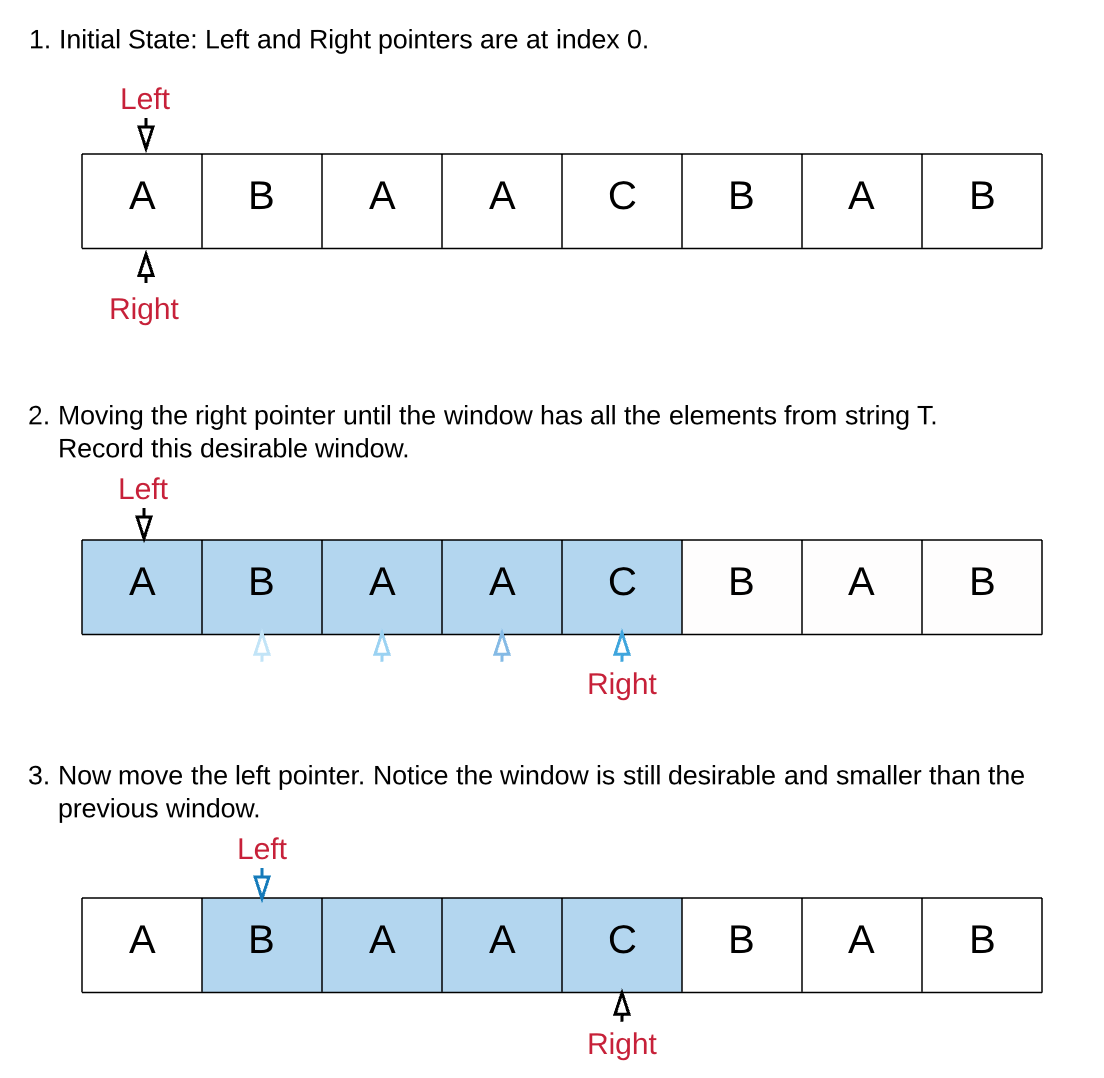
The answer is the smallest desirable window.

For eg. S = "ABAACBAB" T = "ABC". Then our answer window is "ACB" and shown below is one of the possible desirable windows.

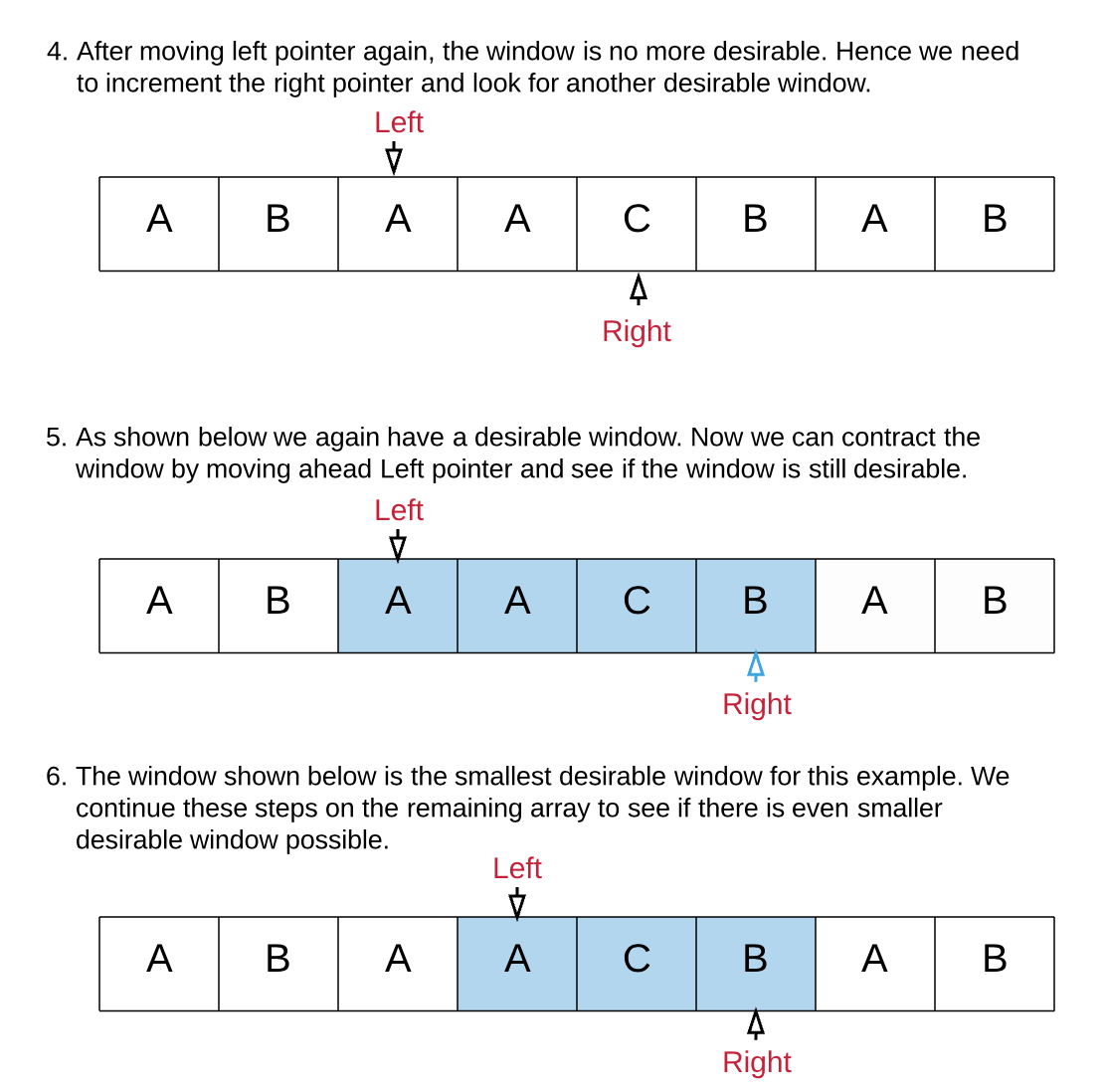


**Algorithm**

1. We start with two pointers, left*left* and right*right* initially pointing to the first element of the string S*S*.
2. We use the right*right* pointer to expand the window until we get a desirable window i.e. a window that contains all of the characters of T*T*.
3. Once we have a window with all the characters, we can move the left pointer ahead one by one. If the window is still a desirable one we keep on updating the minimum window size.
4. If the window is not desirable any more, we repeat step \; 2*step*2 onwards.



The above steps are repeated until we have looked at all the windows. The smallest window is returned.



|  |
| --- |
| class Solution {  public String minWindow(String s, String t) {  if (s.length() == 0 || t.length() == 0) {  return "";  }  // Dictionary which keeps a count of all the unique characters in t.  Map<Character, Integer> dictT = new HashMap<Character, Integer>();  for (int i = 0; i < t.length(); i++) {  int count = dictT.getOrDefault(t.charAt(i), 0);  dictT.put(t.charAt(i), count + 1);  }  // Number of unique characters in t, which need to be present in the desired window.  int required = dictT.size();  // Left and Right pointer  int l = 0, r = 0;  // formed is used to keep track of how many unique characters in t  // are present in the current window in its desired frequency.  // e.g. if t is "AABC" then the window must have two A's, one B and one C.  // Thus formed would be = 3 when all these conditions are met.  int formed = 0;  // Dictionary which keeps a count of all the unique characters in the current window.  Map<Character, Integer> windowCounts = new HashMap<Character, Integer>();  // ans list of the form (window length, left, right)  int[] ans = {-1, 0, 0};  while (r < s.length()) {  // Add one character from the right to the window  char c = s.charAt(r);  int count = windowCounts.getOrDefault(c, 0);  windowCounts.put(c, count + 1);  // If the frequency of the current character added equals to the  // desired count in t then increment the formed count by 1.  if (dictT.containsKey(c) && windowCounts.get(c).intValue() == dictT.get(c).intValue()) {  formed++;  }  // Try and contract the window till the point where it ceases to be 'desirable'.  while (l <= r && formed == required) {  c = s.charAt(l);  // Save the smallest window until now.  if (ans[0] == -1 || r - l + 1 < ans[0]) {  ans[0] = r - l + 1;  ans[1] = l;  ans[2] = r;  }  // The character at the position pointed by the  // `Left` pointer is no longer a part of the window.  windowCounts.put(c, windowCounts.get(c) - 1);  if (dictT.containsKey(c) && windowCounts.get(c).intValue() < dictT.get(c).intValue()) {  formed--;  }  // Move the left pointer ahead, this would help to look for a new window.  l++;  }  // Keep expanding the window once we are done contracting.  r++;  }  return ans[0] == -1 ? "" : s.substring(ans[1], ans[2] + 1);  }  } |

**Complexity Analysis**

* Time Complexity: O(|S| + |T|)*O*(∣*S*∣+∣*T*∣) where |S| and |T| represent the lengths of strings S*S* and T*T*. In the worst case we might end up visiting every element of string S*S* twice, once by left pointer and once by right pointer. |T|∣*T*∣ represents the length of string T*T*.
* Space Complexity: O(|S| + |T|)*O*(∣*S*∣+∣*T*∣). |S|∣*S*∣ when the window size is equal to the entire string S*S*. |T|∣*T*∣ when T*T* has all unique characters.

#### **Approach 2: Optimized Sliding Window**

**Intuition**

A small improvement to the above approach can reduce the time complexity of the algorithm to O(2\*|filtered\\_S| + |S| + |T|)*O*(2∗∣*filtered*\_*S*∣+∣*S*∣+∣*T*∣), where filtered\\_S*filtered*\_*S* is the string formed from S by removing all the elements not present in T*T*.

This complexity reduction is evident when |filtered\\_S| <<< |S|∣*filtered*\_*S*∣<<<∣*S*∣.

This kind of scenario might happen when length of string T*T* is way too small than the length of string S*S* and string S*S* consists of numerous characters which are not present in T*T*.

**Algorithm**

We create a list called filtered\\_S*filtered*\_*S* which has all the characters from string S*S* along with their indices in S*S*, but these characters should be present in T*T*.

S = "ABCDDDDDDEEAFFBC" T = "ABC"

filtered\_S = [(0, 'A'), (1, 'B'), (2, 'C'), (11, 'A'), (14, 'B'), (15, 'C')]

Here (0, 'A') means in string S character A is at index 0.

We can now follow our sliding window approach on the smaller string filtered\\_S*filtered*\_*S*.

|  |
| --- |
| class Solution {  public String minWindow(String s, String t) {  if (s.length() == 0 || t.length() == 0) {  return "";  }  Map<Character, Integer> dictT = new HashMap<Character, Integer>();  for (int i = 0; i < t.length(); i++) {  int count = dictT.getOrDefault(t.charAt(i), 0);  dictT.put(t.charAt(i), count + 1);  }  int required = dictT.size();  // Filter all the characters from s into a new list along with their index.  // The filtering criteria is that the character should be present in t.  List<Pair<Integer, Character>> filteredS = new ArrayList<Pair<Integer, Character>>();  for (int i = 0; i < s.length(); i++) {  char c = s.charAt(i);  if (dictT.containsKey(c)) {  filteredS.add(new Pair<Integer, Character>(i, c));  }  }  int l = 0, r = 0, formed = 0;  Map<Character, Integer> windowCounts = new HashMap<Character, Integer>();  int[] ans = {-1, 0, 0};  // Look for the characters only in the filtered list instead of entire s.  // This helps to reduce our search.  // Hence, we follow the sliding window approach on as small list.  while (r < filteredS.size()) {  char c = filteredS.get(r).getValue();  int count = windowCounts.getOrDefault(c, 0);  windowCounts.put(c, count + 1);  if (dictT.containsKey(c) && windowCounts.get(c).intValue() == dictT.get(c).intValue()) {  formed++;  }  // Try and contract the window till the point where it ceases to be 'desirable'.  while (l <= r && formed == required) {  c = filteredS.get(l).getValue();  // Save the smallest window until now.  int end = filteredS.get(r).getKey();  int start = filteredS.get(l).getKey();  if (ans[0] == -1 || end - start + 1 < ans[0]) {  ans[0] = end - start + 1;  ans[1] = start;  ans[2] = end;  }  windowCounts.put(c, windowCounts.get(c) - 1);  if (dictT.containsKey(c) && windowCounts.get(c).intValue() < dictT.get(c).intValue()) {  formed--;  }  l++;  }  r++;  }  return ans[0] == -1 ? "" : s.substring(ans[1], ans[2] + 1);  }  } |

**Complexity Analysis**

* Time Complexity : O(|S| + |T|)*O*(∣*S*∣+∣*T*∣) where |S| and |T| represent the lengths of strings S*S* and T*T*. The complexity is same as the previous approach. But in certain cases where |filtered\\_S|∣*filtered*\_*S*∣ <<< |S|∣*S*∣, the complexity would reduce because the number of iterations would be 2\*|filtered\\_S| + |S| + |T|2∗∣*filtered*\_*S*∣+∣*S*∣+∣*T*∣.
* Space Complexity : O(|S| + |T|)*O*(∣*S*∣+∣*T*∣).

**Read N Characters Given Read4 II - Call multiple times**

Given a file and assume that you can only read the file using a given method read4, implement a method read to read *n* characters. **Your method read may be called multiple times.**

**Method read4:**

The API read4 reads 4 consecutive characters from the file, then writes those characters into the buffer array buf4.

The return value is the number of actual characters read.

Note that read4() has its own file pointer, much like FILE \*fp in C.

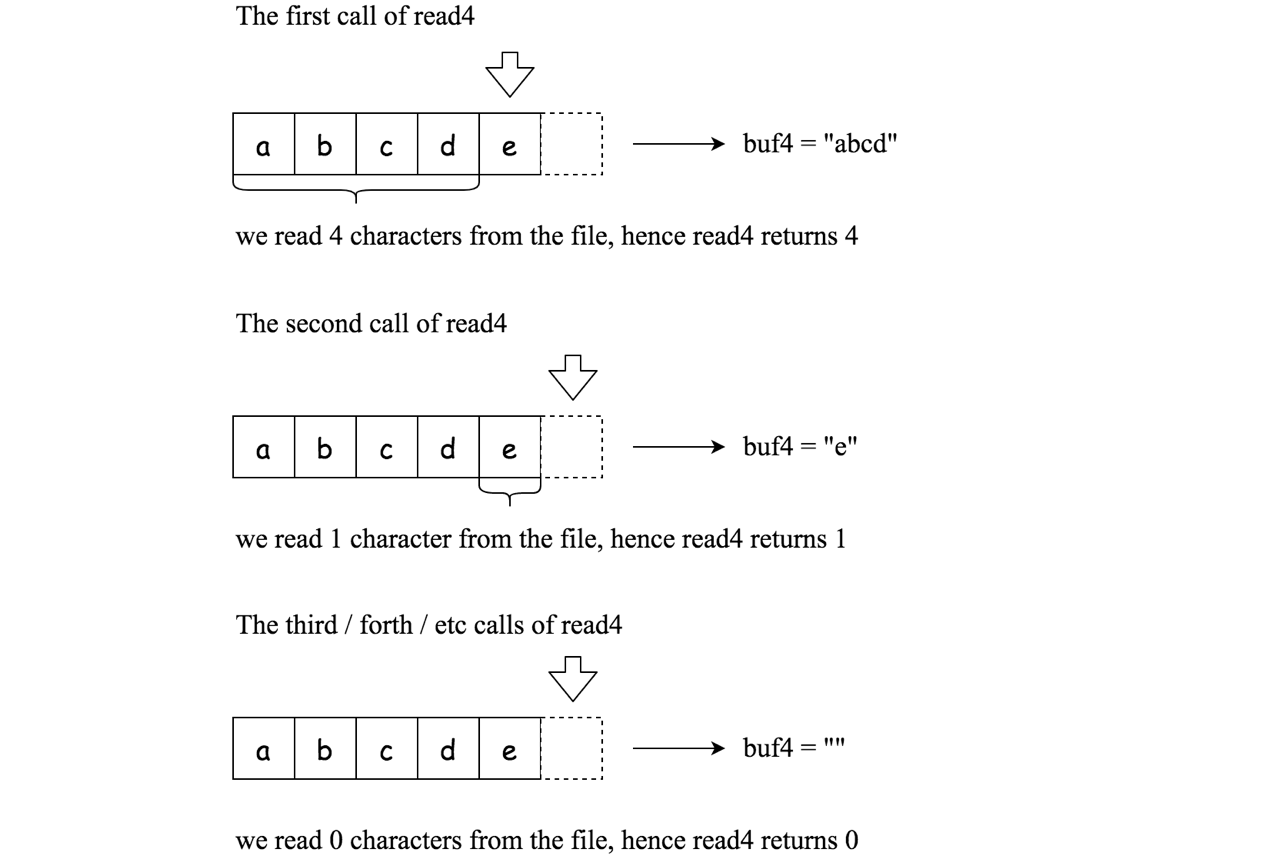
**Definition of read4:**

Parameter: char[] buf4

Returns: int

Note: buf4[] is destination not source, the results from read4 will be copied to buf4[]

Below is a high level example of how read4 works:



File file("abcde"); // File is "abcde", initially file pointer (fp) points to 'a'

char[] buf4 = new char[4]; // Create buffer with enough space to store characters

read4(buf4); // read4 returns 4. Now buf4 = "abcd", fp points to 'e'

read4(buf4); // read4 returns 1. Now buf4 = "e", fp points to end of file

read4(buf4); // read4 returns 0. Now buf4 = "", fp points to end of file

**Method read:**

By using the read4 method, implement the method read that reads *n* characters from the file and store it in the buffer array buf. Consider that you **cannot** manipulate the file directly.

The return value is the number of actual characters read.

**Definition of read:**

Parameters: char[] buf, int n

Returns: int

Note: buf[] is destination not source, you will need to write the results to buf[]

**Example 1:**

File file("abc");

Solution sol;

// Assume buf is allocated and guaranteed to have enough space for storing all characters from the file.

sol.read(buf, 1); // After calling your read method, buf should contain "a". We read a total of 1 character from the file, so return 1.

sol.read(buf, 2); // Now buf should contain "bc". We read a total of 2 characters from the file, so return 2.

sol.read(buf, 1); // We have reached the end of file, no more characters can be read. So return 0.

**Example 2:**

File file("abc");

Solution sol;

sol.read(buf, 4); // After calling your read method, buf should contain "abc". We read a total of 3 characters from the file, so return 3.

sol.read(buf, 1); // We have reached the end of file, no more characters can be read. So return 0.

**Note:**

* Consider that you **cannot** manipulate the file directly, the file is only accesible for read4 but **not** for read.
* The read function may be called **multiple times**.
* Please remember to **RESET** your class variables declared in Solution, as static/class variables are **persisted across multiple test cases**. Please see [here](https://leetcode.com/faq/) for more details.
* You may assume the destination buffer array, buf, is guaranteed to have enough space for storing *n* characters.
* It is guaranteed that in a given test case the same buffer buf is called by read.

**Longest Substring with At Most Two Distinct Characters**

Given a string ***s*** , find the length of the longest substring ***t***that contains **at most**2 distinct characters.

**Example 1:**

**Input:** "eceba"

**Output:** 3

**Explanation: *t***is "ece" which its length is 3.

**Example 2:**

**Input:** "ccaabbb"

**Output:** 5

**Explanation: *t***is "aabbb" which its length is 5.

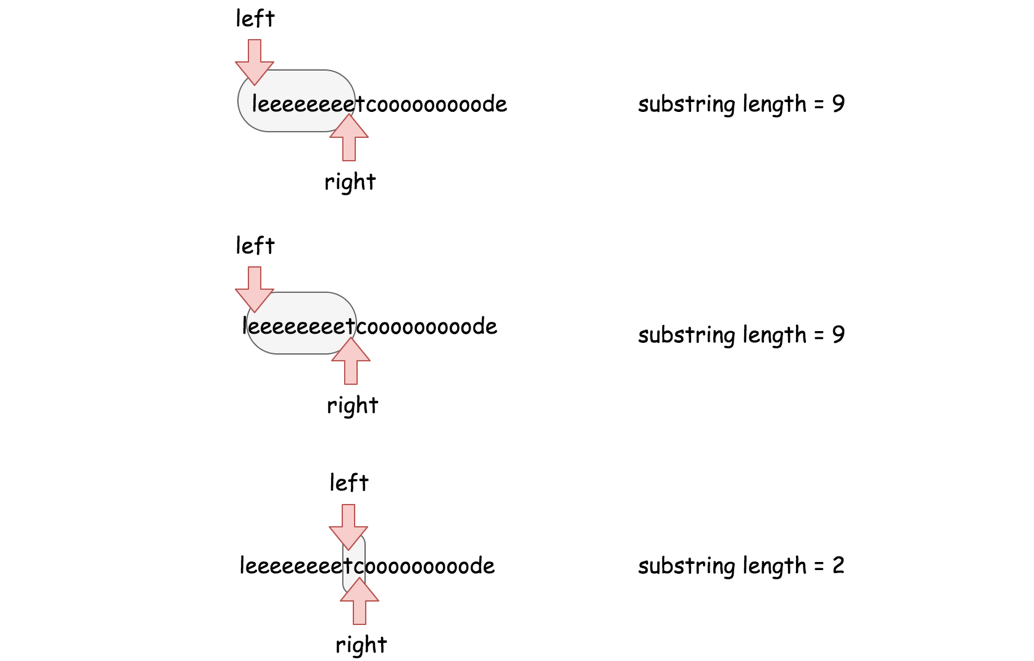
## Solution Article

#### **Approach 1: Sliding Window**

**Intuition**

To solve the problem in one pass let's use here sliding window approach with two set pointers left and right serving as the window boundaries.

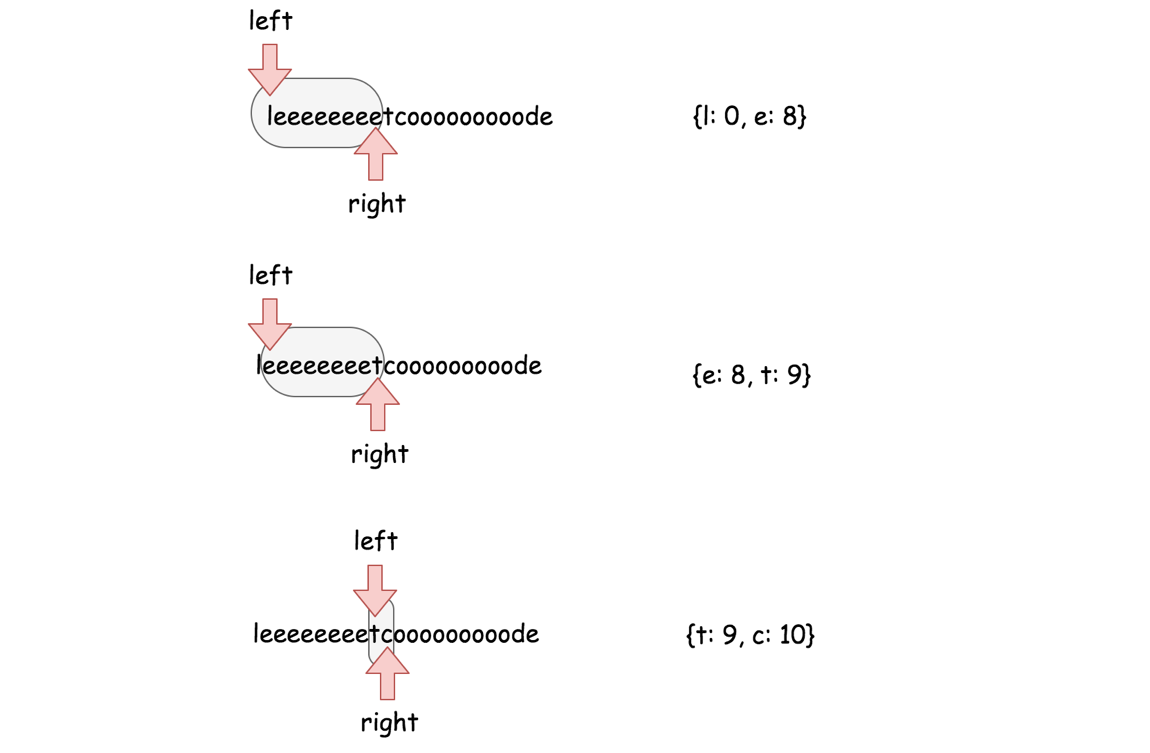
The idea is to set both pointers in the position 0 and then move right pointer to the right while the window contains not more than two distinct characters. If at some point we've got 3 distinct characters, let's move left pointer to keep not more than 2 distinct characters in the window.



Basically that's the algorithm : to move sliding window along the string, to keep not more than 2 distinct characters in the window, and to update max substring length at each step.

There is just one more question to reply - how to move the left pointer to keep only 2 distinct characters in the string?

Let's use for this purpose hashmap containing all characters in the sliding window as keys and their rightmost positions as values. At each moment, this hashmap could contain not more than 3 elements.



For example, using this hashmap one knows that the rightmost position of character e in "eeeeeeeet" window is 8 and so one has to move left pointer in the position 8 + 1 = 9 to exclude the character e from the sliding window.

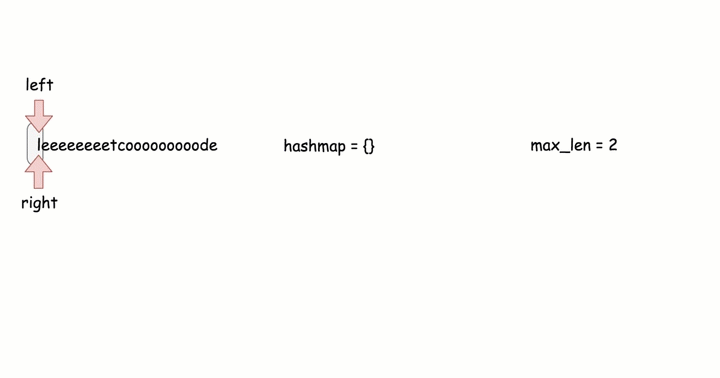
Do we have here the best possible time complexity? Yes, we do - it's the only one pass along the string with N characters and the time complexity is \mathcal{O}(N)O(*N*).

**Algorithm**

Now one could write down the algortihm.

* Return N if the string length N is smaller than 3.
* Set both set pointers in the beginning of the string left = 0 and right = 0 and init max substring length max\_len = 2.
* While right pointer is less than N:
  + If hashmap contains less than 3 distinct characters, add the current character s[right] in the hashmap and move right pointer to the right.
  + If hashmap contains 3 distinct characters, remove the leftmost character from the hashmap and move the left pointer so that sliding window contains again 2 distinct characters only.
  + Update max\_len.

**Implementation**



|  |
| --- |
| class Solution {  public int lengthOfLongestSubstringTwoDistinct(String s) {  int n = s.length();  if (n < 3) return n;  // sliding window left and right pointers  int left = 0;  int right = 0;  // hashmap character -> its rightmost position  // in the sliding window  HashMap<Character, Integer> hashmap = new HashMap<Character, Integer>();  int max\_len = 2;  while (right < n) {  // when the slidewindow contains less than 3 characters  hashmap.put(s.charAt(right), right++);  // slidewindow contains 3 characters  if (hashmap.size() == 3) {  // delete the leftmost character  int del\_idx = Collections.min(hashmap.values());  hashmap.remove(s.charAt(del\_idx));  // move left pointer of the slidewindow  left = del\_idx + 1;  }  max\_len = Math.max(max\_len, right - left);  }  return max\_len;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) where N is a number of characters in the input string.
* Space complexity : \mathcal{O}(1)O(1) since additional space is used only for a hashmap with at most 3 elements.

**Problem generalization**

The same sliding window approach could be used to solve the generalized problem :

[Longest Substring with At Most K Distinct Characters](https://leetcode.com/problems/longest-substring-with-at-most-k-distinct-characters/)

**Missing Ranges**

You are given an inclusive range [lower, upper] and a **sorted unique** integer array nums, where all elements are in the inclusive range.

A number x is considered **missing** if x is in the range [lower, upper] and x is not in nums.

Return *the****smallest sorted****list of ranges that****cover every missing number exactly***. That is, no element of nums is in any of the ranges, and each missing number is in one of the ranges.

Each range [a,b] in the list should be output as:

* "a->b" if a != b
* "a" if a == b

**Example 1:**

**Input:** nums = [0,1,3,50,75], lower = 0, upper = 99

**Output:** ["2","4->49","51->74","76->99"]

**Explanation:** The ranges are:

[2,2] --> "2"

[4,49] --> "4->49"

[51,74] --> "51->74"

[76,99] --> "76->99"

**Example 2:**

**Input:** nums = [], lower = 1, upper = 1

**Output:** ["1"]

**Explanation:** The only missing range is [1,1], which becomes "1".

**Example 3:**

**Input:** nums = [], lower = -3, upper = -1

**Output:** ["-3->-1"]

**Explanation:** The only missing range is [-3,-1], which becomes "-3->-1".

**Example 4:**

**Input:** nums = [-1], lower = -1, upper = -1

**Output:** []

**Explanation:** There are no missing ranges since there are no missing numbers.

**Example 5:**

**Input:** nums = [-1], lower = -2, upper = -1

**Output:** ["-2"]

**Constraints:**

* -109 <= lower <= upper <= 109
* 0 <= nums.length <= 100
* lower <= nums[i] <= upper
* All the values of nums are **unique**.

Check medium

**Next Closest Time**

Given a time represented in the format "HH:MM", form the next closest time by reusing the current digits. There is no limit on how many times a digit can be reused.

You may assume the given input string is always valid. For example, "01:34", "12:09" are all valid. "1:34", "12:9" are all invalid.

**Example 1:**

**Input:** time = "19:34"

**Output:** "19:39"

**Explanation:** The next closest time choosing from digits **1**, **9**, **3**, **4**, is **19:39**, which occurs 5 minutes later.

It is not **19:33**, because this occurs 23 hours and 59 minutes later.

**Example 2:**

**Input:** time = "23:59"

**Output:** "22:22"

**Explanation:** The next closest time choosing from digits **2**, **3**, **5**, **9**, is **22:22**.

It may be assumed that the returned time is next day's time since it is smaller than the input time numerically.

**Constraints:**

* time.length == 5
* time is a valid time in the form "HH:MM".
* 0 <= HH < 24
* 0 <= MM < 60

#### **Approach #1: Simulation [Accepted]**

**Intuition and Algorithm**

Simulate the clock going forward by one minute. Each time it moves forward, if all the digits are allowed, then return the current time.

The natural way to represent the time is as an integer t in the range 0 <= t < 24 \* 60. Then the hours are t / 60, the minutes are t % 60, and each digit of the hours and minutes can be found by hours / 10, hours % 10 etc.

|  |
| --- |
| class Solution {  public String nextClosestTime(String time) {  int cur = 60 \* Integer.parseInt(time.substring(0, 2));  cur += Integer.parseInt(time.substring(3));  Set<Integer> allowed = new HashSet();  for (char c: time.toCharArray()) if (c != ':') {  allowed.add(c - '0');  }  while (true) {  cur = (cur + 1) % (24 \* 60);  int[] digits = new int[]{cur / 60 / 10, cur / 60 % 10, cur % 60 / 10, cur % 60 % 10};  search : {  for (int d: digits) if (!allowed.contains(d)) break search;  return String.format("%02d:%02d", cur / 60, cur % 60);  }  }  }  } |

**Complexity Analysis**

* Time Complexity: O(1)*O*(1). We try up to 24 \* 6024∗60 possible times until we find the correct time.
* Space Complexity: O(1)*O*(1).

#### **Approach #2: Build From Allowed Digits [Accepted]**

**Intuition and Algorithm**

We have up to 4 different allowed digits, which naively gives us 4 \* 4 \* 4 \* 4 possible times. For each possible time, let's check that it can be displayed on a clock: ie., hours < 24 and mins < 60. The best possible time != start is the one with the smallest cand\_elapsed = (time - start) % (24 \* 60), as this represents the time that has elapsed since start, and where the modulo operation is taken to be always non-negative.

For example, if we have start = 720 (ie. noon), then times like 12:05 = 725 means that (725 - 720) % (24 \* 60) = 5 minutes have elapsed; while times like 00:10 = 10 means that (10 - 720) % (24 \* 60) = -710 % (24 \* 60) = 730 seconds have elapsed.

Also, we should make sure to handle cand\_elapsed carefully. When our current candidate time cur is equal to the given starting time, then cand\_elapsed will be 0 and we should handle this case appropriately.

|  |
| --- |
| class Solution {  public String nextClosestTime(String time) {  int start = 60 \* Integer.parseInt(time.substring(0, 2));  start += Integer.parseInt(time.substring(3));  int ans = start;  int elapsed = 24 \* 60;  Set<Integer> allowed = new HashSet();  for (char c: time.toCharArray()) if (c != ':') {  allowed.add(c - '0');  }  for (int h1: allowed) for (int h2: allowed) if (h1 \* 10 + h2 < 24) {  for (int m1: allowed) for (int m2: allowed) if (m1 \* 10 + m2 < 60) {  int cur = 60 \* (h1 \* 10 + h2) + (m1 \* 10 + m2);  int candElapsed = Math.floorMod(cur - start, 24 \* 60);  if (0 < candElapsed && candElapsed < elapsed) {  ans = cur;  elapsed = candElapsed;  }  }  }  return String.format("%02d:%02d", ans / 60, ans % 60);  }  } |

**Complexity Analysis**

* Time Complexity: O(1)*O*(1). We all 4^444 possible times and take the best one.
* Space Complexity: O(1)*O*(1).

**Expressive Words**

Sometimes people repeat letters to represent extra feeling, such as "hello" -> "heeellooo", "hi" -> "hiiii".  In these strings like "heeellooo", we have *groups* of adjacent letters that are all the same:  "h", "eee", "ll", "ooo".

For some given string S, a query word is *stretchy* if it can be made to be equal to S by any number of applications of the following *extension* operation: choose a group consisting of characters c, and add some number of characters c to the group so that the size of the group is 3 or more.

For example, starting with "hello", we could do an extension on the group "o" to get "hellooo", but we cannot get "helloo" since the group "oo" has size less than 3.  Also, we could do another extension like "ll" -> "lllll" to get "helllllooo".  If S = "helllllooo", then the query word "hello" would be stretchy because of these two extension operations: query = "hello" -> "hellooo" -> "helllllooo" = S.

Given a list of query words, return the number of words that are stretchy.

**Example:**

**Input:**

S = "heeellooo"

words = ["hello", "hi", "helo"]

**Output:** 1

**Explanation:**

We can extend "e" and "o" in the word "hello" to get "heeellooo".

We can't extend "helo" to get "heeellooo" because the group "ll" is not size 3 or more.

**Constraints:**

* 0 <= len(S) <= 100.
* 0 <= len(words) <= 100.
* 0 <= len(words[i]) <= 100.
* S and all words in words consist only of lowercase letters

#### **Approach #1: Run Length Encoding**

**Intuition**

For some word, write the head character of every group, and the count of that group. For example, for "abbcccddddaaaaa", we'll write the "key" of "abcda", and the "count" [1,2,3,4,5].

Let's see if a word is stretchy. Evidently, it needs to have the same key as S.

Now, let's say we have individual counts c1 = S.count[i] and c2 = word.count[i].

* If c1 < c2, then we can't make the ith group of word equal to the ith word of S by adding characters.
* If c1 >= 3, then we can add letters to the ith group of word to match the ith group of S, as the latter is extended.
* Else, if c1 < 3, then we must have c2 == c1 for the ith groups of word and S to match.

|  |
| --- |
| class Solution {  public int expressiveWords(String S, String[] words) {  RLE R = new RLE(S);  int ans = 0;  search: for (String word: words) {  RLE R2 = new RLE(word);  if (!R.key.equals(R2.key)) continue;  for (int i = 0; i < R.counts.size(); ++i) {  int c1 = R.counts.get(i);  int c2 = R2.counts.get(i);  if (c1 < 3 && c1 != c2 || c1 < c2)  continue search;  }  ans++;  }  return ans;  }  }  class RLE {  String key;  List<Integer> counts;  public RLE(String S) {  StringBuilder sb = new StringBuilder();  counts = new ArrayList();  char[] ca = S.toCharArray();  int N = ca.length;  int prev = -1;  for (int i = 0; i < N; ++i) {  if (i == N-1 || ca[i] != ca[i+1]) {  sb.append(ca[i]);  counts.add(i - prev);  prev = i;  }  }  key = sb.toString();  }  } |

**Complexity Analysis**

* Time Complexity: O(QK)*O*(*QK*), where Q*Q* is the length of words (at least 1), and K*K* is the maximum length of a word.
* Space Complexity: O(K)*O*(*K*).

**Find And Replace in String**

To some string S, we will perform some replacement operations that replace groups of letters with new ones (not necessarily the same size).

Each replacement operation has 3 parameters: a starting index i, a source word x and a target word y.  The rule is that if x starts at position i in the **original** **string** **S**, then we will replace that occurrence of x with y.  If not, we do nothing.

For example, if we have S = "abcd" and we have some replacement operation i = 2, x = "cd", y = "ffff", then because "cd" starts at position 2 in the original string S, we will replace it with "ffff".

Using another example on S = "abcd", if we have both the replacement operation i = 0, x = "ab", y = "eee", as well as another replacement operation i = 2, x = "ec", y = "ffff", this second operation does nothing because in the original string S[2] = 'c', which doesn't match x[0] = 'e'.

All these operations occur simultaneously.  It's guaranteed that there won't be any overlap in replacement: for example, S = "abc", indexes = [0, 1], sources = ["ab","bc"] is not a valid test case.

**Example 1:**

**Input:** S = "abcd", indexes = [0, 2], sources = ["a", "cd"], targets = ["eee", "ffff"]

**Output:** "eeebffff"

**Explanation:**

"a" starts at index 0 in S, so it's replaced by "eee".

"cd" starts at index 2 in S, so it's replaced by "ffff".

**Example 2:**

**Input:** S = "abcd", indexes = [0, 2], sources = ["ab","ec"], targets = ["eee","ffff"]

**Output:** "eeecd"

**Explanation:**

"ab" starts at index 0 in S, so it's replaced by "eee".

"ec" doesn't starts at index 2 in the **original** S, so we do nothing.

**Constraints:**

* 0 <= S.length <= 1000
* S consists of only lowercase English letters.
* 0 <= indexes.length <= 100
* 0 <= indexes[i] < S.length
* sources.length == indexes.length
* targets.length == indexes.length
* 1 <= sources[i].length, targets[i].length <= 50
* sources[i] and targets[i] consist of only lowercase English letters.

#### **Approach #1: Direct [Accepted]**

**Intuition and Algorithm**

We showcase two different approaches. In both approaches, we build some answer string ans, that starts as S. Our main motivation in these approaches is to be able to identify and handle when a given replacement operation does nothing.

In Java, the idea is to build an array match that tells us match[ix] = j whenever S[ix] is the head of a successful replacement operation j: that is, whenever S[ix:].startswith(sources[j]).

After, we build the answer using this match array. For each index ix in S, we can use match to check whether S[ix] is being replaced or not. We repeatedly either write the next character S[ix], or groups of characters targets[match[ix]], depending on the value of match[ix].

In Python, we sort our replacement jobs (i, x, y) in reverse order. If S[i:].startswith(x), then we can replace that section S[i:i+len(x)] with the target y. We used a reverse order so that edits to S do not interfere with the rest of the queries.

|  |
| --- |
| class Solution {  public String findReplaceString(String S, int[] indexes, String[] sources, String[] targets) {  int N = S.length();  int[] match = new int[N];  Arrays.fill(match, -1);  for (int i = 0; i < indexes.length; ++i) {  int ix = indexes[i];  if (S.substring(ix, ix + sources[i].length()).equals(sources[i]))  match[ix] = i;  }  StringBuilder ans = new StringBuilder();  int ix = 0;  while (ix < N) {  if (match[ix] >= 0) {  ans.append(targets[match[ix]]);  ix += sources[match[ix]].length();  } else {  ans.append(S.charAt(ix++));  }  }  return ans.toString();  }  } |

**Complexity Analysis**

* Time Complexity: O(NQ)*O*(*NQ*), where N*N* is the length of S, and we have Q*Q* replacement operations. (Our complexity could be faster with a more accurate implementation, but it isn't necessary.)
* Space Complexity: O(N)*O*(*N*), if we consider targets[i].length <= 100 as a constant bound.

**Maximize Distance to Closest Person**

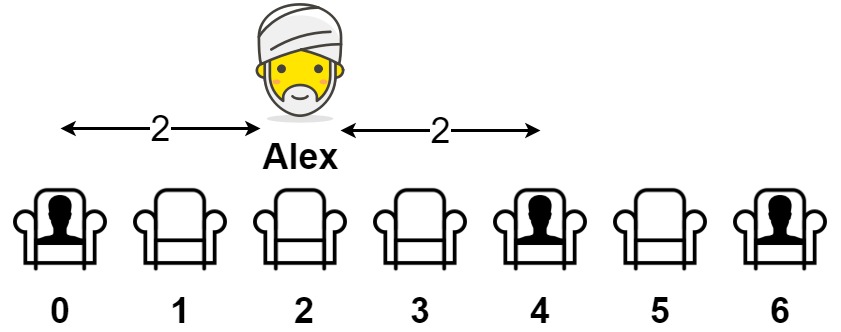
You are given an array representing a row of seats where seats[i] = 1 represents a person sitting in the ith seat, and seats[i] = 0 represents that the ith seat is empty **(0-indexed)**.

There is at least one empty seat, and at least one person sitting.

Alex wants to sit in the seat such that the distance between him and the closest person to him is maximized.

Return *that maximum distance to the closest person*.

**Example 1:**



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

**Output:** 2

**Explanation:**

If Alex sits in the second open seat (i.e. seats[2]), then the closest person has distance 2.

If Alex sits in any other open seat, the closest person has distance 1.

Thus, the maximum distance to the closest person is 2.

**Example 2:**

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

**Output:** 3

**Explanation:**

If Alex sits in the last seat (i.e. seats[3]), the closest person is 3 seats away.

This is the maximum distance possible, so the answer is 3.

**Example 3:**

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

**Output:** 1

**Constraints:**

* 2 <= seats.length <= 2 \* 104
* seats[i] is 0 or 1.
* At least one seat is **empty**.
* At least one seat is **occupied**.

#### **Approach #1: Next Array [Accepted]**

**Intuition**

Let left[i] be the distance from seat i to the closest person sitting to the left of i. Similarly, let right[i] be the distance to the closest person sitting to the right of i. This is motivated by the idea that the closest person in seat i sits a distance min(left[i], right[i]) away.

**Algorithm**

To construct left[i], notice it is either left[i-1] + 1 if the seat is empty, or 0 if it is full. right[i] is constructed in a similar way.

|  |
| --- |
| class Solution {  public int maxDistToClosest(int[] seats) {  int N = seats.length;  int[] left = new int[N], right = new int[N];  Arrays.fill(left, N);  Arrays.fill(right, N);  for (int i = 0; i < N; ++i) {  if (seats[i] == 1) left[i] = 0;  else if (i > 0) left[i] = left[i-1] + 1;  }  for (int i = N-1; i >= 0; --i) {  if (seats[i] == 1) right[i] = 0;  else if (i < N-1) right[i] = right[i+1] + 1;  }  int ans = 0;  for (int i = 0; i < N; ++i)  if (seats[i] == 0)  ans = Math.max(ans, Math.min(left[i], right[i]));  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of seats.
* Space Complexity: O(N)*O*(*N*), the space used by left and right.

#### **Approach #2: Two Pointer [Accepted]**

**Intuition**

As we iterate through seats, we'll update the closest person sitting to our left, and closest person sitting to our right.

**Algorithm**

Keep track of prev, the filled seat at or to the left of i, and future, the filled seat at or to the right of i.

Then at seat i, the closest person is min(i - prev, future - i), with one exception. i - prev should be considered infinite if there is no person to the left of seat i, and similarly future - i is infinite if there is no one to the right of seat i.

|  |
| --- |
| class Solution {  public int maxDistToClosest(int[] seats) {  int N = seats.length;  int prev = -1, future = 0;  int ans = 0;  for (int i = 0; i < N; ++i) {  if (seats[i] == 1) {  prev = i;  } else {  while (future < N && seats[future] == 0 || future < i)  future++;  int left = prev == -1 ? N : i - prev;  int right = future == N ? N : future - i;  ans = Math.max(ans, Math.min(left, right));  }  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of seats.
* Space Complexity: O(1)*O*(1).

#### **Approach #3: Group by Zero [Accepted]**

**Intuition**

In a group of K adjacent empty seats between two people, the answer is (K+1) / 2.

**Algorithm**

For each group of K empty seats between two people, we can take into account the candidate answer (K+1) / 2.

For groups of empty seats between the edge of the row and one other person, the answer is K, and we should take into account those answers too.

|  |
| --- |
| class Solution {  public int maxDistToClosest(int[] seats) {  int N = seats.length;  int K = 0; //current longest group of empty seats  int ans = 0;  for (int i = 0; i < N; ++i) {  if (seats[i] == 1) {  K = 0;  } else {  K++;  ans = Math.max(ans, (K + 1) / 2);  }  }  for (int i = 0; i < N; ++i) if (seats[i] == 1) {  ans = Math.max(ans, i);  break;  }  for (int i = N-1; i >= 0; --i) if (seats[i] == 1) {  ans = Math.max(ans, N - 1 - i);  break;  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of seats.
* Space Complexity: O(1)*O*(1).

**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 and Stack

**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.

Check hard

**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

Check hard

**Kth Largest Element in an Array**

Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

**Example 1:**

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

**Output:** 5

**Example 2:**

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

**Output:** 4

**Note:**  
You may assume k is always valid, 1 ≤ k ≤ array's length.

Check medium

**Meeting Rooms II**

Given an array of meeting time intervals intervals where intervals[i] = [starti, endi], return *the minimum number of conference rooms required*.

**Example 1:**

**Input:** intervals = [[0,30],[5,10],[15,20]]

**Output:** 2

**Example 2:**

**Input:** intervals = [[7,10],[2,4]]

**Output:** 1

**Constraints:**

* 1 <= intervals.length <= 104
* 0 <= starti < endi <= 106

Check medium

**Backspace String Compare**

Given two strings S and T, return if they are equal when both are typed into empty text editors. # means a backspace character.

Note that after backspacing an empty text, the text will continue empty.

**Example 1:**

**Input:** S = "ab#c", T = "ad#c"

**Output:** true

**Explanation**: Both S and T become "ac".

**Example 2:**

**Input:** S = "ab##", T = "c#d#"

**Output:** true

**Explanation**: Both S and T become "".

**Example 3:**

**Input:** S = "a##c", T = "#a#c"

**Output:** true

**Explanation**: Both S and T become "c".

**Example 4:**

**Input:** S = "a#c", T = "b"

**Output:** false

**Explanation**: S becomes "c" while T becomes "b".

**Note**:

* 1 <= S.length <= 200
* 1 <= T.length <= 200
* S and T only contain lowercase letters and '#' characters.

**Follow up:**

* Can you solve it in O(N) time and O(1) space?

#### **Approach #1: Build String [Accepted]**

**Intuition**

Let's individually build the result of each string (build(S) and build(T)), then compare if they are equal.

**Algorithm**

To build the result of a string build(S), we'll use a stack based approach, simulating the result of each keystroke.

|  |
| --- |
| class Solution {  public boolean backspaceCompare(String S, String T) {  return build(S).equals(build(T));  }  public String build(String S) {  Stack<Character> ans = new Stack();  for (char c: S.toCharArray()) {  if (c != '#')  ans.push(c);  else if (!ans.empty())  ans.pop();  }  return String.valueOf(ans);  }  } |

**Complexity Analysis**

* Time Complexity: O(M + N)*O*(*M*+*N*), where M, N*M*,*N* are the lengths of S and T respectively.
* Space Complexity: O(M + N)*O*(*M*+*N*).

#### **Approach #2: Two Pointer [Accepted]**

**Intuition**

When writing a character, it may or may not be part of the final string depending on how many backspace keystrokes occur in the future.

If instead we iterate through the string in reverse, then we will know how many backspace characters we have seen, and therefore whether the result includes our character.

**Algorithm**

Iterate through the string in reverse. If we see a backspace character, the next non-backspace character is skipped. If a character isn't skipped, it is part of the final answer.

See the comments in the code for more details.

|  |
| --- |
| class Solution {  public boolean backspaceCompare(String S, String T) {  int i = S.length() - 1, j = T.length() - 1;  int skipS = 0, skipT = 0;  while (i >= 0 || j >= 0) { // While there may be chars in build(S) or build (T)  while (i >= 0) { // Find position of next possible char in build(S)  if (S.charAt(i) == '#') {skipS++; i--;}  else if (skipS > 0) {skipS--; i--;}  else break;  }  while (j >= 0) { // Find position of next possible char in build(T)  if (T.charAt(j) == '#') {skipT++; j--;}  else if (skipT > 0) {skipT--; j--;}  else break;  }  // If two actual characters are different  if (i >= 0 && j >= 0 && S.charAt(i) != T.charAt(j))  return false;  // If expecting to compare char vs nothing  if ((i >= 0) != (j >= 0))  return false;  i--; j--;  }  return true;  }  } |

**Complexity Analysis**

* Time Complexity: O(M + N)*O*(*M*+*N*), where M, N*M*,*N* are the lengths of S and T respectively.
* Space Complexity: O(1)*O*(1).

**Minimum Cost to Hire K Workers**

There are N workers.  The i-th worker has a quality[i] and a minimum wage expectation wage[i].

Now we want to hire exactly K workers to form a *paid group*.  When hiring a group of K workers, we must pay them according to the following rules:

1. Every worker in the paid group should be paid in the ratio of their quality compared to other workers in the paid group.
2. Every worker in the paid group must be paid at least their minimum wage expectation.

Return the least amount of money needed to form a paid group satisfying the above conditions.

**Example 1:**

**Input:** quality = [10,20,5], wage = [70,50,30], K = 2

**Output:** 105.00000

**Explanation**: We pay 70 to 0-th worker and 35 to 2-th worker.

**Example 2:**

**Input:** quality = [3,1,10,10,1], wage = [4,8,2,2,7], K = 3

**Output:** 30.66667

**Explanation**: We pay 4 to 0-th worker, 13.33333 to 2-th and 3-th workers seperately.

**Note:**

1. 1 <= K <= N <= 10000, where N = quality.length = wage.length
2. 1 <= quality[i] <= 10000
3. 1 <= wage[i] <= 10000
4. Answers within 10^-5 of the correct answer will be considered correct.

## Solution

#### **Approach 1: Greedy**

**Intuition**

At least one worker will be paid their minimum wage expectation. If not, we could scale all payments down by some factor and still keep everyone earning more than their wage expectation.

**Algorithm**

For each captain worker that will be paid their minimum wage expectation, let's calculate the cost of hiring K workers where each point of quality is worth wage[captain] / quality[captain] dollars. With this approach, the remaining implementation is straightforward.

Note that this algorithm would not be efficient enough to pass larger test cases.

|  |
| --- |
| class Solution {  public double mincostToHireWorkers(int[] quality, int[] wage, int K) {  int N = quality.length;  double ans = 1e9;  for (int captain = 0; captain < N; ++captain) {  // Must pay at least wage[captain] / quality[captain] per qual  double factor = (double) wage[captain] / quality[captain];  double prices[] = new double[N];  int t = 0;  for (int worker = 0; worker < N; ++worker) {  double price = factor \* quality[worker];  if (price < wage[worker]) continue;  prices[t++] = price;  }  if (t < K) continue;  Arrays.sort(prices, 0, t);  double cand = 0;  for (int i = 0; i < K; ++i)  cand += prices[i];  ans = Math.min(ans, cand);  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N^2 \log N)*O*(*N*2log*N*), where N*N* is the number of workers.
* Space Complexity: O(N)*O*(*N*).

#### **Approach 2: Heap**

**Intuition**

As in Approach #1, at least one worker is paid their minimum wage expectation.

Additionally, every worker has some minimum ratio of dollars to quality that they demand. For example, if wage[0] = 100 and quality[0] = 20, then the ratio for worker 0 is 5.0.

The key insight is to iterate over the ratio. Let's say we hire workers with a ratio R or lower. Then, we would want to know the K workers with the lowest quality, and the sum of that quality. We can use a heap to maintain these variables.

**Algorithm**

Maintain a max heap of quality. (We're using a minheap, with negative values.) We'll also maintain sumq, the sum of this heap.

For each worker in order of ratio, we know all currently considered workers have lower ratio. (This worker will be the 'captain', as described in Approach #1.) We calculate the candidate answer as this ratio times the sum of the smallest K workers in quality.

|  |
| --- |
| class Solution {  public double mincostToHireWorkers(int[] quality, int[] wage, int K) {  int N = quality.length;  Worker[] workers = new Worker[N];  for (int i = 0; i < N; ++i)  workers[i] = new Worker(quality[i], wage[i]);  Arrays.sort(workers);  double ans = 1e9;  int sumq = 0;  PriorityQueue<Integer> pool = new PriorityQueue();  for (Worker worker: workers) {  pool.offer(-worker.quality);  sumq += worker.quality;  if (pool.size() > K)  sumq += pool.poll();  if (pool.size() == K)  ans = Math.min(ans, sumq \* worker.ratio());  }  return ans;  }  }  class Worker implements Comparable<Worker> {  public int quality, wage;  public Worker(int q, int w) {  quality = q;  wage = w;  }  public double ratio() {  return (double) wage / quality;  }  public int compareTo(Worker other) {  return Double.compare(ratio(), other.ratio());  }  } |

**Complexity Analysis**

* Time Complexity: O(N \log N)*O*(*N*log*N*), where N*N* is the number of workers.
* Space Complexity: O(N)*O*(*N*).

**K Closest Points to Origin**

We have a list of points on the plane.  Find the K closest points to the origin (0, 0).

(Here, the distance between two points on a plane is the Euclidean distance.)

You may return the answer in any order.  The answer is guaranteed to be unique (except for the order that it is in.)

**Example 1:**

**Input:** points = [[1,3],[-2,2]], K = 1

**Output:** [[-2,2]]

**Explanation:**

The distance between (1, 3) and the origin is sqrt(10).

The distance between (-2, 2) and the origin is sqrt(8).

Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.

We only want the closest K = 1 points from the origin, so the answer is just [[-2,2]].

**Example 2:**

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

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

(The answer [[-2,4],[3,3]] would also be accepted.)

**Note:**

1. 1 <= K <= points.length <= 10000
2. -10000 < points[i][0] < 10000
3. -10000 < points[i][1] < 10000

## Solution

#### **Approach 1: Sort**

**Intuition**

Sort the points by distance, then take the closest K points.

**Algorithm**

There are two variants.

In Java, we find the K-th distance by creating an array of distances and then sorting them. After, we select all the points with distance less than or equal to this K-th distance.

In Python, we sort by a custom key function - namely, the distance to the origin. Afterwards, we return the first K elements of the list.

|  |
| --- |
| class Solution {  public int[][] kClosest(int[][] points, int K) {  int N = points.length;  int[] dists = new int[N];  for (int i = 0; i < N; ++i)  dists[i] = dist(points[i]);  Arrays.sort(dists);  int distK = dists[K-1];  int[][] ans = new int[K][2];  int t = 0;  for (int i = 0; i < N; ++i)  if (dist(points[i]) <= distK)  ans[t++] = points[i];  return ans;  }  public int dist(int[] point) {  return point[0] \* point[0] + point[1] \* point[1];  }  } |

**Complexity Analysis**

* Time Complexity: O(N \log N)*O*(*N*log*N*), where N*N* is the length of points.
* Space Complexity: O(N)*O*(*N*).

#### **Approach 2: Divide and Conquer**

**Intuition**

We want an algorithm faster than N \log N*N*log*N*. Clearly, the only way to do this is to use the fact that the K elements returned can be in any order -- otherwise we would be sorting which is at least N \log N*N*log*N*.

Say we choose some random element x = A[i] and split the array into two buckets: one bucket of all the elements less than x, and another bucket of all the elements greater than or equal to x. This is known as "quickselecting by a pivot x".

The idea is that if we quickselect by some pivot, on average in linear time we'll reduce the problem to a problem of half the size.

**Algorithm**

Let's do the work(i, j, K) of partially sorting the subarray (points[i], points[i+1], ..., points[j]) so that the smallest K elements of this subarray occur in the first K positions (i, i+1, ..., i+K-1).

First, we quickselect by a random pivot element from the subarray. To do this in place, we have two pointers i and j, and move these pointers to the elements that are in the wrong bucket -- then, we swap these elements.

After, we have two buckets [oi, i] and [i+1, oj], where (oi, oj) are the original (i, j) values when calling work(i, j, K). Say the first bucket has 10 items and the second bucket has 15 items. If we were trying to partially sort say, K = 5 items, then we only need to partially sort the first bucket: work(oi, i, 5). Otherwise, if we were trying to partially sort say, K = 17 items, then the first 10 items are already partially sorted, and we only need to partially sort the next 7 items: work(i+1, oj, 7).

|  |
| --- |
| import java.util.concurrent.ThreadLocalRandom;  class Solution {  int[][] points;  public int[][] kClosest(int[][] points, int K) {  this.points = points;  sort(0, points.length - 1, K);  return Arrays.copyOfRange(points, 0, K);  }  public void sort(int i, int j, int K) {  if (i >= j) return;  int k = ThreadLocalRandom.current().nextInt(i, j);  swap(i, k);  int mid = partition(i, j);  int leftLength = mid - i + 1;  if (K < leftLength)  sort(i, mid - 1, K);  else if (K > leftLength)  sort(mid + 1, j, K - leftLength);  }  public int partition(int i, int j) {  int oi = i;  int pivot = dist(i);  i++;  while (true) {  while (i < j && dist(i) < pivot)  i++;  while (i <= j && dist(j) > pivot)  j--;  if (i >= j) break;  swap(i, j);  }  swap(oi, j);  return j;  }  public int dist(int i) {  return points[i][0] \* points[i][0] + points[i][1] \* points[i][1];  }  public void swap(int i, int j) {  int t0 = points[i][0], t1 = points[i][1];  points[i][0] = points[j][0];  points[i][1] = points[j][1];  points[j][0] = t0;  points[j][1] = t1;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*) in *average case* and O(N^2)*O*(*N*2) in the worst case, where N*N* is the length of points.
* Space Complexity: O(N)*O*(*N*).

## Linked Lists

According to our user survey data, Linked List problems are not asked frequently at Google. Perhaps, most linked list problems are not that complex and it is harder to ask follow up and complexity analysis questions

Nonetheless, we strongly recommend you to still practice classic Linked List interview questions such as: [Linked List Cycle](https://leetcode.com/problems/linked-list-cycle/), [Intersection of Two Linked Lists](https://leetcode.com/problems/intersection-of-two-linked-lists/), and [Copy List with Random Pointers](https://leetcode.com/problems/copy-list-with-random-pointer/). These problems are really fun and they teach you how to think outside of the box.

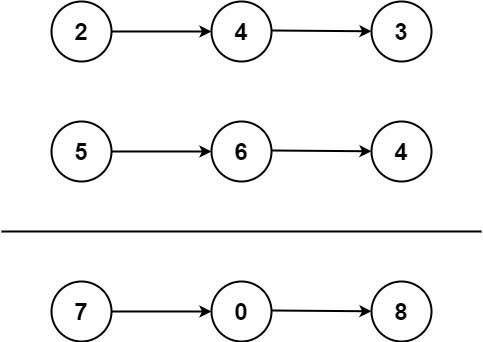
Of course, [Merge k Sorted Lists](https://leetcode.com/problems/merge-k-sorted-lists/) is one of our all-time favorite interview questions, and Google seems to love this question as well. Make sure you understand how to analyze the time complexity! This is a common follow up question for this problem.

**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:**

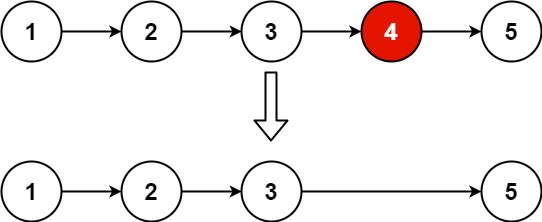
* The number of nodes in each linked list is in the range [1, 100].
* 0 <= Node.val <= 9
* It is guaranteed that the list represents a number that does not have leading zeros.

**Remove Nth Node From End of List**

Given the head of a linked list, remove the nth node from the end of the list and return its head.

**Follow up:** Could you do this in one pass?

**Example 1:**



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

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

**Example 2:**

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

**Output:** []

**Example 3:**

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

**Output:** [1]

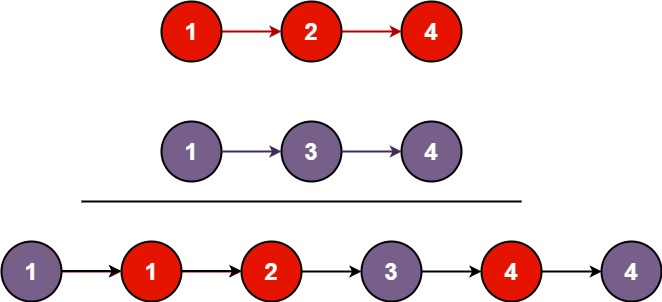
**Constraints:**

* The number of nodes in the list is sz.
* 1 <= sz <= 30
* 0 <= Node.val <= 100
* 1 <= n <= sz

**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.

**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.

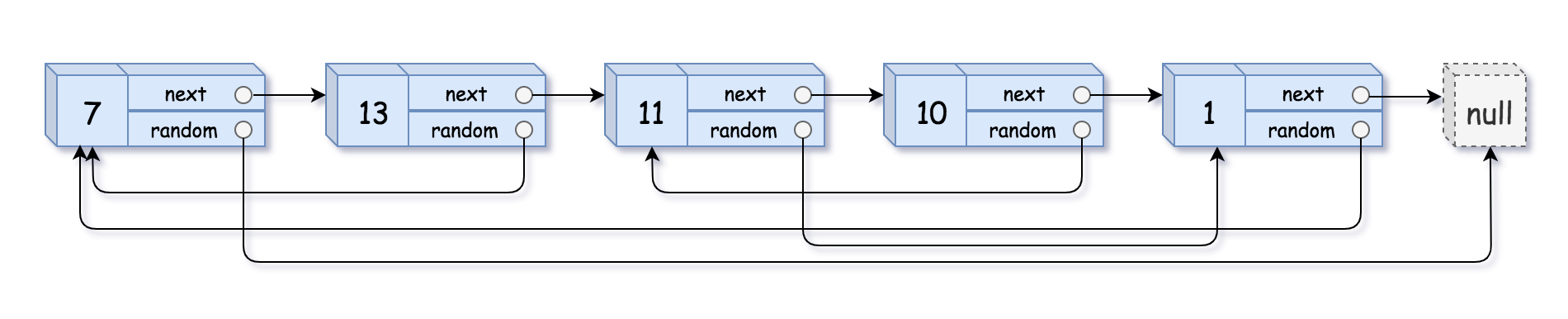
Return *the head of the copied linked list*.

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.

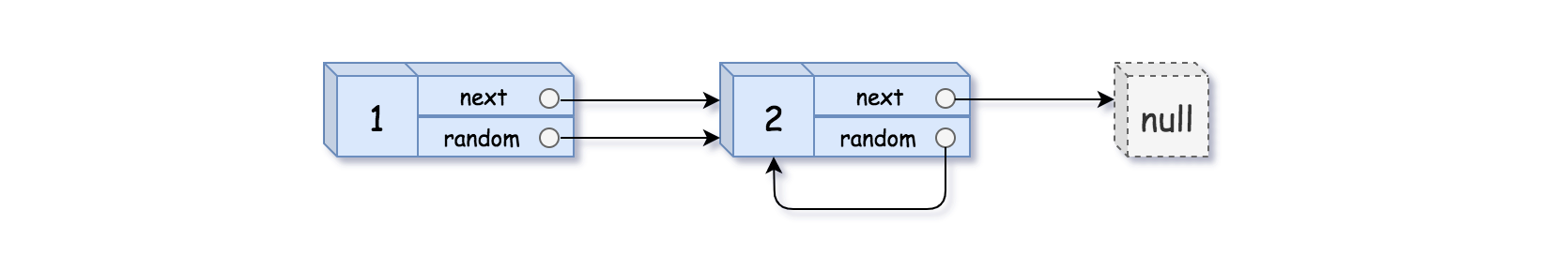
**Example 1:**



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

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

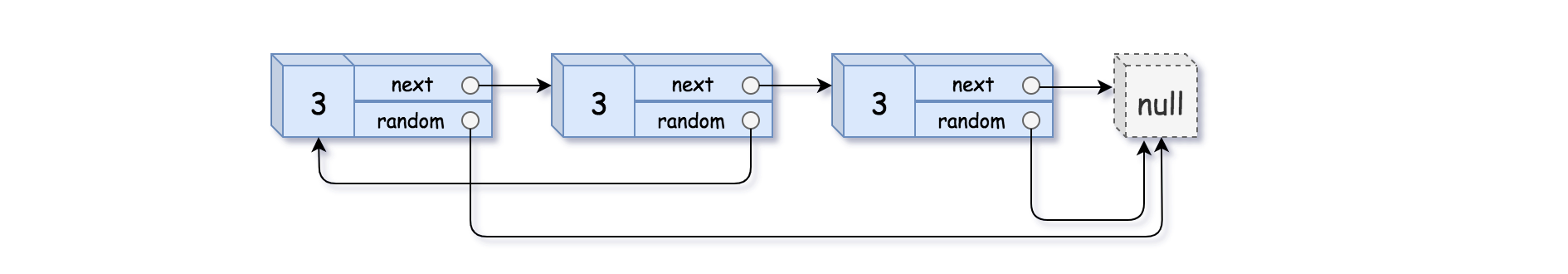
**Example 2:**



**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.

## Trees and Graphs

Tree is just a special case of graph. To understand the difference between trees and graphs, you can work on [Graph Valid Tree](https://leetcode.com/problems/graph-valid-tree/description/).

Graphs are generally breath-first search or depth-first search. The same applies to Trees, but trees never contain cycles.

Graphs are generally more complex than trees. Similarly, trees are generally more complex than linear data structures, such as arrays or linked lists.

Prepping your knowledge in Graphs is essential for Google interviews as you would most likely encounter a tree or a graph question. A great way to brush up your skills in this area is to implement a tree or graph by coding it from scratch in the [Playground](https://leetcode.com/playground/).

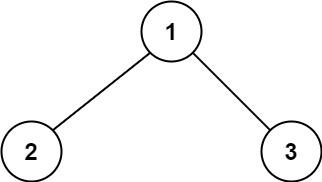
**Binary Tree Maximum Path Sum**

A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

Given the root of a binary tree, return *the maximum****path sum****of any path*.

**Example 1:**

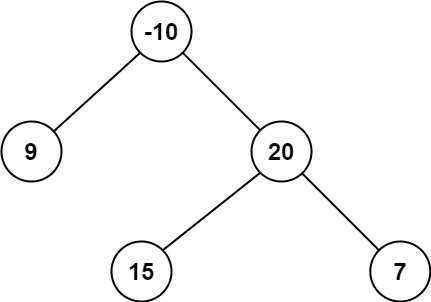


**Input:** root = [1,2,3]

**Output:** 6

**Explanation:** The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.

**Example 2:**



**Input:** root = [-10,9,20,null,null,15,7]

**Output:** 42

**Explanation:** The optimal path is 15 -> 20 -> 7 with a path sum of 15 + 20 + 7 = 42.

**Constraints:**

* The number of nodes in the tree is in the range [1, 3 \* 104].
* -1000 <= Node.val <= 1000

hard

**Word Ladder**

A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words such that:

* The first word in the sequence is beginWord.
* The last word in the sequence is endWord.
* Only one letter is different between each adjacent pair of words in the sequence.
* Every word in the sequence is in wordList.

Given two words, beginWord and endWord, and a dictionary wordList, return *the****number of words****in the****shortest transformation sequence****from* beginWord *to* endWord*, or*0*if no such sequence exists.*

**Example 1:**

**Input:** beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

**Output:** 5

**Explanation:** One shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog" with 5 words.

**Example 2:**

**Input:** beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]

**Output:** 0

**Explanation:** The endWord "cog" is not in wordList, therefore there is no possible transformation.

**Constraints:**

* 1 <= beginWord.length <= 10
* endWord.length == beginWord.length
* 1 <= wordList.length <= 5000
* wordList[i].length == beginWord.length
* beginWord, endWord, and wordList[i] consist of lowercase English letters.
* beginWord != endWord
* All the strings in wordList are **unique**.

Hard

**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

**Course Schedule II**

There are a total of n courses you have to take labelled from 0 to n - 1.

Some courses may have prerequisites, for example, if prerequisites[i] = [ai, bi] this means you must take the course bi before the course ai.

Given the total number of courses numCourses and a list of the prerequisite pairs, return the ordering of courses you should take to finish all courses.

If there are many valid answers, return **any** of them. If it is impossible to finish all courses, return **an empty array**.

**Example 1:**

**Input:** numCourses = 2, prerequisites = [[1,0]]

**Output:** [0,1]

**Explanation:** There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is [0,1].

**Example 2:**

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

**Output:** [0,2,1,3]

**Explanation:** There are a total of 4 courses to take. To take course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0.

So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3].

**Example 3:**

**Input:** numCourses = 1, prerequisites = []

**Output:** [0]

**Constraints:**

* 1 <= numCourses <= 2000
* 0 <= prerequisites.length <= numCourses \* (numCourses - 1)
* prerequisites[i].length == 2
* 0 <= ai, bi < numCourses

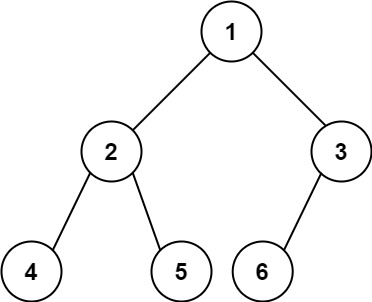
Hard

**Count Complete Tree Nodes**

Given the root of a **complete** binary tree, return the number of the nodes in the tree.

According to [**Wikipedia**](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees), every level, except possibly the last, is completely filled in a complete binary tree, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes inclusive at the last level h.

**Example 1:**



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

**Output:** 6

**Example 2:**

**Input:** root = []

**Output:** 0

**Example 3:**

**Input:** root = [1]

**Output:** 1

**Constraints:**

* The number of nodes in the tree is in the range [0, 5 \* 104].
* 0 <= Node.val <= 5 \* 104
* The tree is guaranteed to be **complete**.

**Follow up:** Traversing the tree to count the number of nodes in the tree is an easy solution but with O(n) complexity. Could you find a faster algorithm?

## Solution

#### **Approach 1: Linear Time**

**Intuition**

This problem is quite popular at Google during the last year. The naive solution here is a linear time one-liner which counts nodes recursively one by one.

**Implementation**

|  |
| --- |
| class Solution {  public int countNodes(TreeNode root) {  return root != null ? 1 + countNodes(root.right) + countNodes(root.left) : 0;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*).
* Space complexity : \mathcal{O}(d) = \mathcal{O}(\log N)O(*d*)=O(log*N*) to keep the recursion stack, where d is a tree depth.

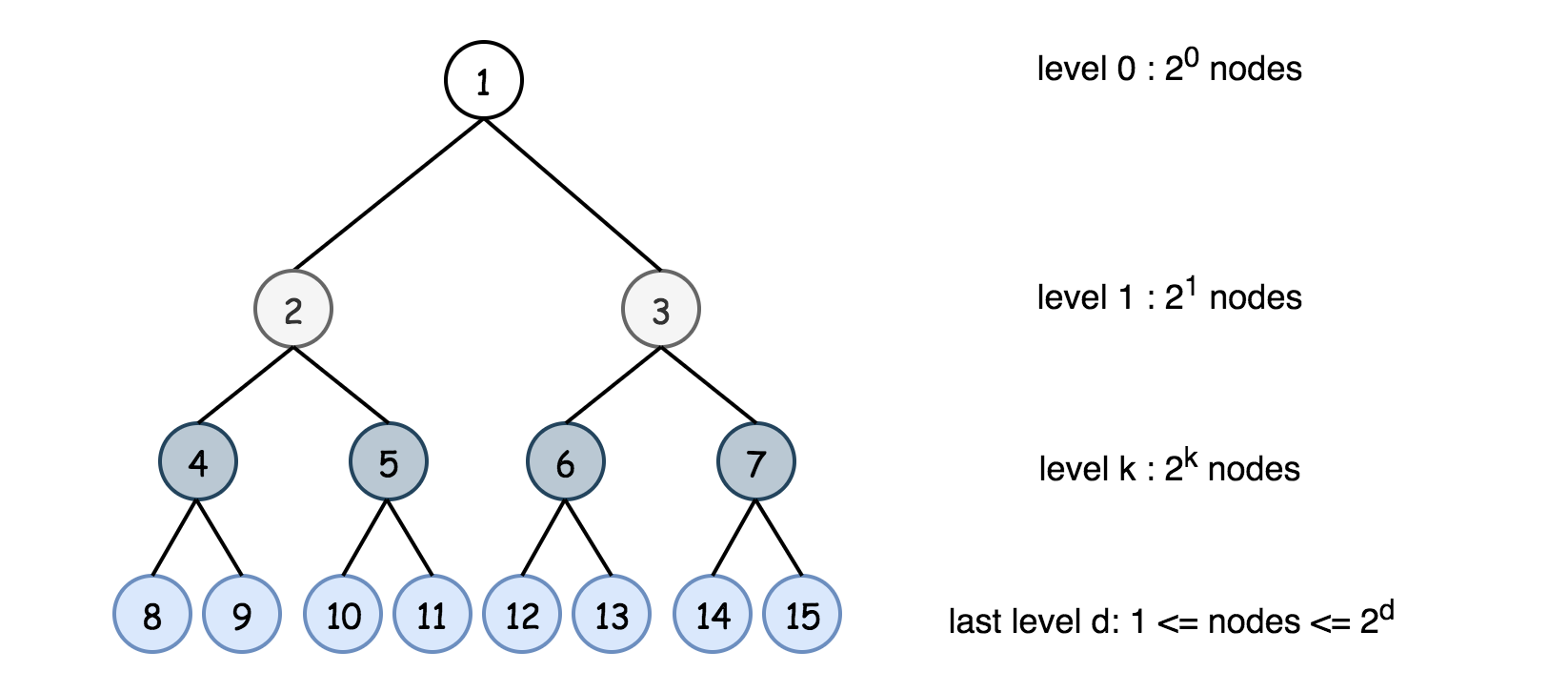
#### **Approach 2: Binary search**

**Intuition**

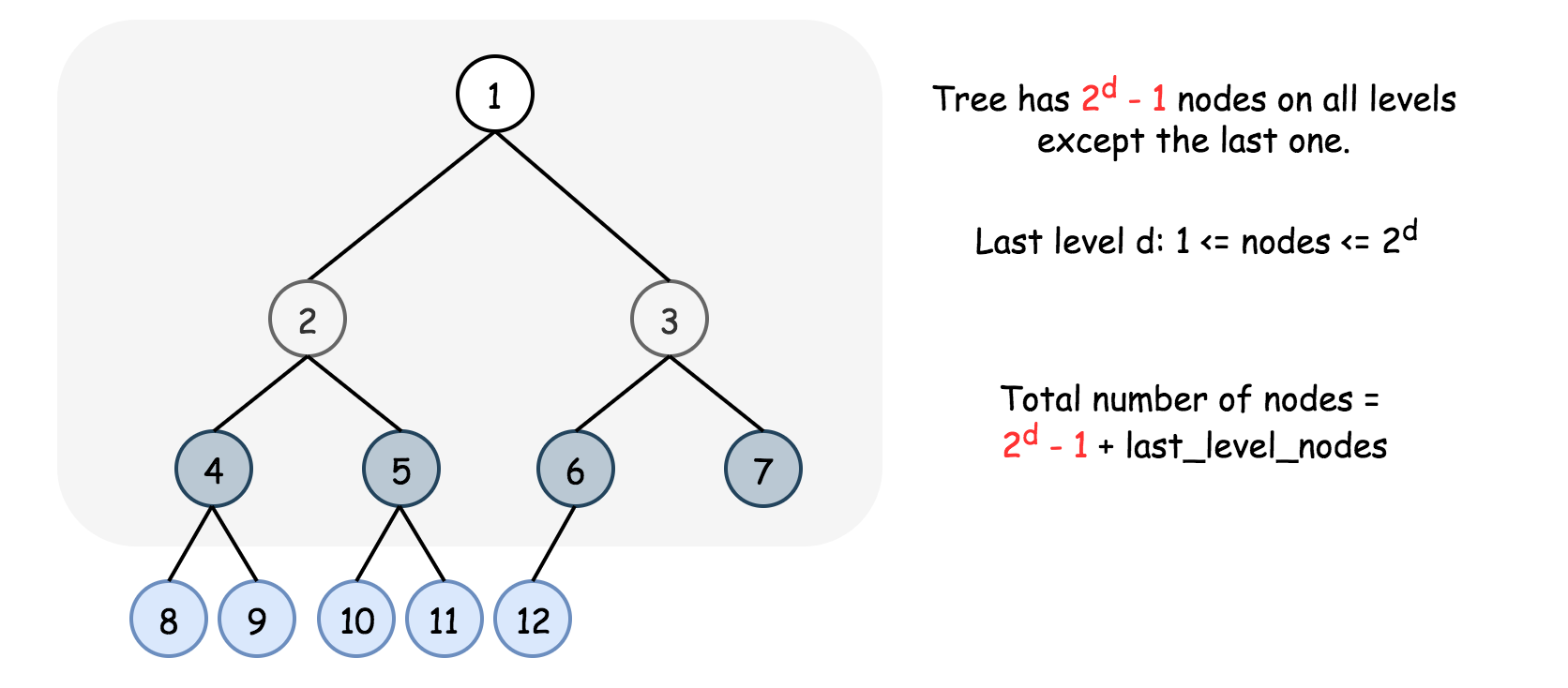
Approach 1 doesn't profit from the fact that the tree is a complete one.

In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

That means that complete tree has 2^k2*k* nodes in the kth level if the kth level is not the last one. The last level may be not filled completely, and hence in the last level the number of nodes could vary from 1 to 2^d2*d*, where d is a tree depth.



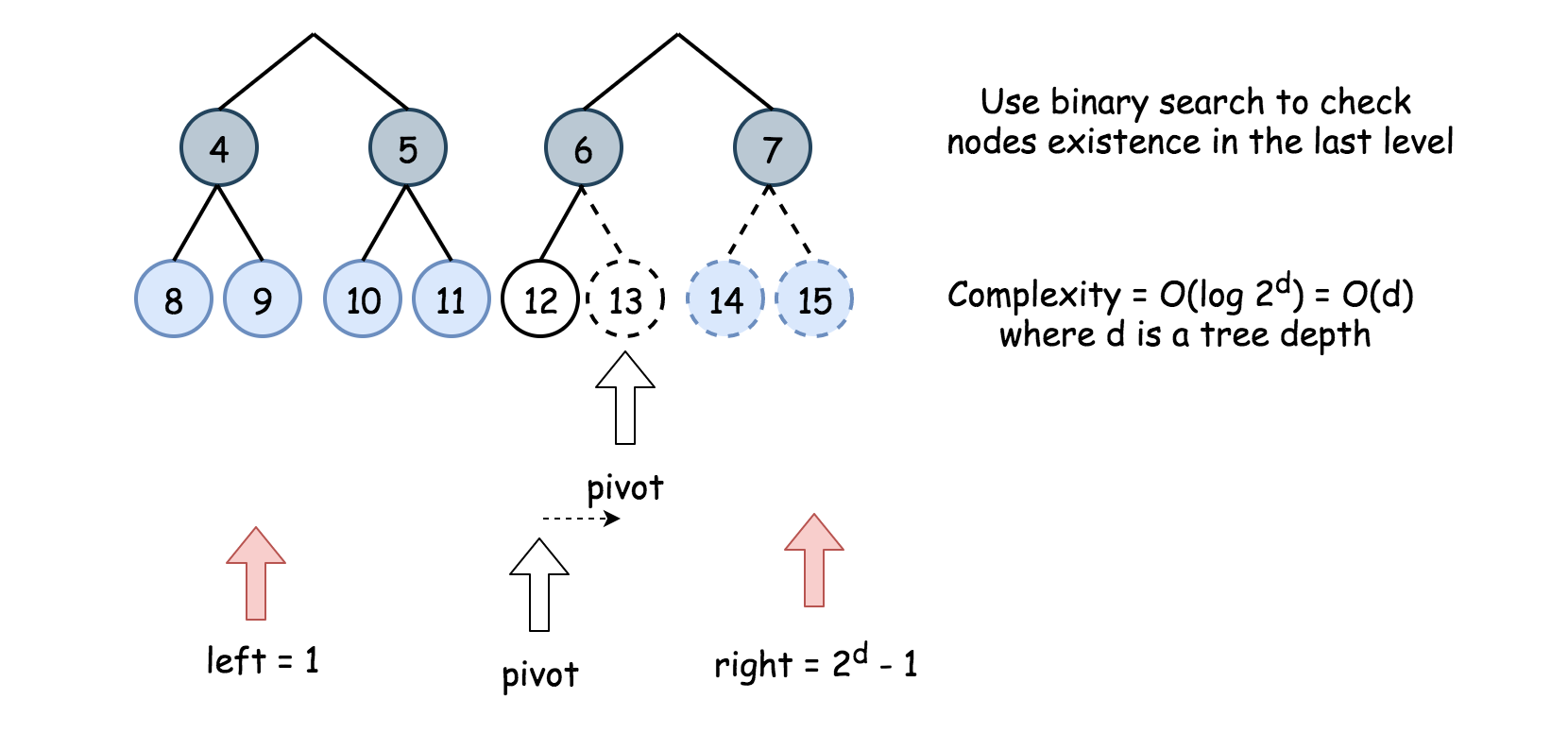
Now one could compute the number of nodes in all levels but the last one: \sum\_{k = 0}^{k = d - 1}{2^k} = 2^d - 1∑*k*=0*k*=*d*−1​2*k*=2*d*−1. That reduces the problem to the simple check of how many nodes the tree has in the last level.



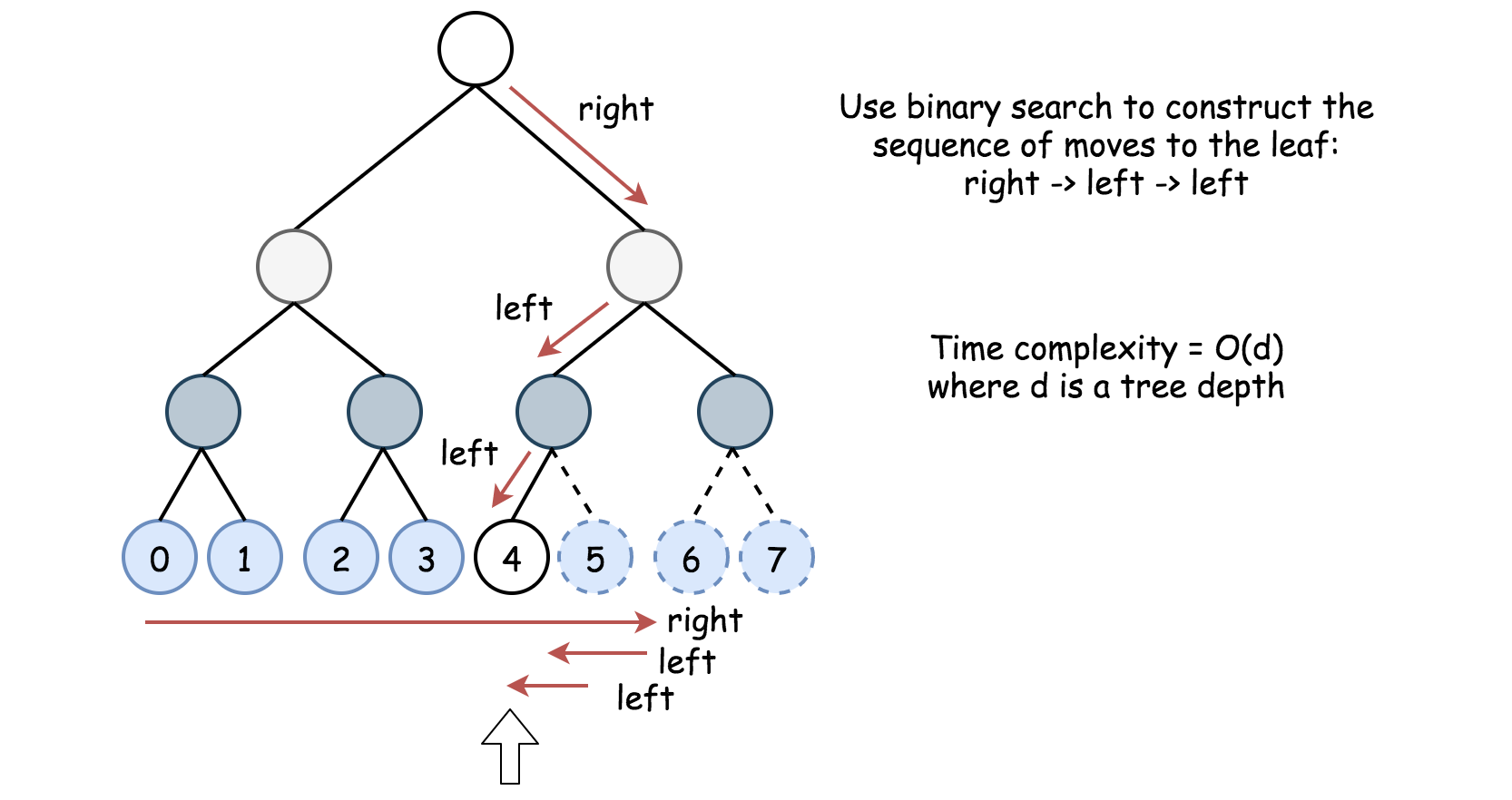
Now there are two questions:

1. How many nodes in the last level have to be checked?
2. What is the best time performance for such a check?

Let's start from the first question. It's a complete tree, and hence all nodes in the last level are as far left as possible. That means that instead of checking the existence of all 2^d2*d* possible leafs, one could use binary search and check \log(2^d) = dlog(2*d*)=*d* leafs only.



Let's move to the second question, and enumerate potential nodes in the last level from 0 to 2^d - 12*d*−1. How to check if the node number idx exists? Let's use binary search again to reconstruct the sequence of moves from root to idx node. For example, idx = 4. idx is in the second half of nodes 0,1,2,3,4,5,6,7 and hence the first move is to the right. Then idx is in the first half of nodes 4,5,6,7 and hence the second move is to the left. The idx is in the first half of nodes 4,5 and hence the next move is to the left. The time complexity for one check is \mathcal{O}(d)O(*d*).



1 and 2 together result in \mathcal{O}(d)O(*d*) checks, each check at a price of \mathcal{O}(d)O(*d*). That means that the overall time complexity would be \mathcal{O}(d^2)O(*d*2).

**Algorithm**

* Return 0 if the tree is empty.
* Compute the tree depth d.
* Return 1 if d == 0.
* The number of nodes in all levels but the last one is 2^d - 12*d*−1. The number of nodes in the last level could vary from 1 to 2^d2*d*. Enumerate potential nodes from 0 to 2^d - 12*d*−1 and perform the binary search by the node index to check how many nodes are in the last level. Use the function exists(idx, d, root) to check if the node with index idx exists.
* Use binary search to implement exists(idx, d, root) as well.
* Return 2^d - 12*d*−1 + the number of nodes in the last level.

**Implementation**

|  |
| --- |
| class Solution {  // Return tree depth in O(d) time.  public int computeDepth(TreeNode node) {  int d = 0;  while (node.left != null) {  node = node.left;  ++d;  }  return d;  }  // Last level nodes are enumerated from 0 to 2\*\*d - 1 (left -> right).  // Return True if last level node idx exists.  // Binary search with O(d) complexity.  public boolean exists(int idx, int d, TreeNode node) {  int left = 0, right = (int)Math.pow(2, d) - 1;  int pivot;  for(int i = 0; i < d; ++i) {  pivot = left + (right - left) / 2;  if (idx <= pivot) {  node = node.left;  right = pivot;  }  else {  node = node.right;  left = pivot + 1;  }  }  return node != null;  }  public int countNodes(TreeNode root) {  // if the tree is empty  if (root == null) return 0;  int d = computeDepth(root);  // if the tree contains 1 node  if (d == 0) return 1;  // Last level nodes are enumerated from 0 to 2\*\*d - 1 (left -> right).  // Perform binary search to check how many nodes exist.  int left = 1, right = (int)Math.pow(2, d) - 1;  int pivot;  while (left <= right) {  pivot = left + (right - left) / 2;  if (exists(pivot, d, root)) left = pivot + 1;  else right = pivot - 1;  }  // The tree contains 2\*\*d - 1 nodes on the first (d - 1) levels  // and left nodes on the last level.  return (int)Math.pow(2, d) - 1 + left;  }  } |

**Complexity Analysis**

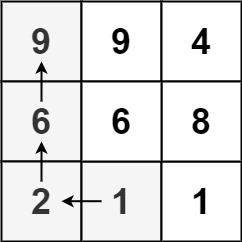
* Time complexity : \mathcal{O}(d^2) = \mathcal{O}(\log^2 N)O(*d*2)=O(log2*N*), where d*d* is a tree depth.
* Space complexity : \mathcal{O}(1)O(1).

**Longest Increasing Path in a Matrix**

Given an m x n integers matrix, return *the length of the longest increasing path in*matrix.

From each cell, you can either move in four directions: left, right, up, or down. You **may not** move **diagonally** or move **outside the boundary** (i.e., wrap-around is not allowed).

**Example 1:**

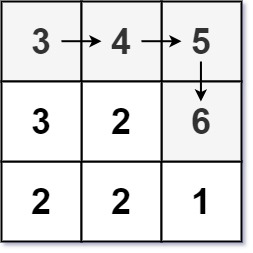


**Input:** matrix = [[9,9,4],[6,6,8],[2,1,1]]

**Output:** 4

**Explanation:** The longest increasing path is [1, 2, 6, 9].

**Example 2:**



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

**Output:** 4

**Explanation:** The longest increasing path is [3, 4, 5, 6]. Moving diagonally is not allowed.

**Example 3:**

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

**Output:** 1

Hard

**Decode String**

Given an encoded string, return its decoded string.

The encoding rule is: k[encoded\_string], where the encoded\_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; No extra white spaces, square brackets are well-formed, etc.

Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there won't be input like 3a or 2[4].

**Example 1:**

**Input:** s = "3[a]2[bc]"

**Output:** "aaabcbc"

**Example 2:**

**Input:** s = "3[a2[c]]"

**Output:** "accaccacc"

**Example 3:**

**Input:** s = "2[abc]3[cd]ef"

**Output:** "abcabccdcdcdef"

**Example 4:**

**Input:** s = "abc3[cd]xyz"

**Output:** "abccdcdcdxyz"

**Constraints:**

* 1 <= s.length <= 30
* s consists of lowercase English letters, digits, and square brackets '[]'.
* s is guaranteed to be **a valid** input.
* All the integers in s are in the range [1, 300].

Queue Stack

**Evaluate Division**

You are given an array of variable pairs equations and an array of real numbers values, where equations[i] = [Ai, Bi] and values[i] represent the equation Ai / Bi = values[i]. Each Ai or Bi is a string that represents a single variable.

You are also given some queries, where queries[j] = [Cj, Dj] represents the jth query where you must find the answer for Cj / Dj = ?.

Return *the answers to all queries*. If a single answer cannot be determined, return -1.0.

**Note:** The input is always valid. You may assume that evaluating the queries will not result in division by zero and that there is no contradiction.

**Example 1:**

**Input:** equations = [["a","b"],["b","c"]], values = [2.0,3.0], queries = [["a","c"],["b","a"],["a","e"],["a","a"],["x","x"]]

**Output:** [6.00000,0.50000,-1.00000,1.00000,-1.00000]

**Explanation:**

Given: *a / b = 2.0*, *b / c = 3.0*

queries are: *a / c = ?*, *b / a = ?*, *a / e = ?*, *a / a = ?*, *x / x = ?*

return: [6.0, 0.5, -1.0, 1.0, -1.0 ]

**Example 2:**

**Input:** equations = [["a","b"],["b","c"],["bc","cd"]], values = [1.5,2.5,5.0], queries = [["a","c"],["c","b"],["bc","cd"],["cd","bc"]]

**Output:** [3.75000,0.40000,5.00000,0.20000]

**Example 3:**

**Input:** equations = [["a","b"]], values = [0.5], queries = [["a","b"],["b","a"],["a","c"],["x","y"]]

**Output:** [0.50000,2.00000,-1.00000,-1.00000]

**Constraints:**

* 1 <= equations.length <= 20
* equations[i].length == 2
* 1 <= Ai.length, Bi.length <= 5
* values.length == equations.length
* 0.0 < values[i] <= 20.0
* 1 <= queries.length <= 20
* queries[i].length == 2
* 1 <= Cj.length, Dj.length <= 5
* Ai, Bi, Cj, Dj consist of lower case English letters and digits.

Do you recognize this as a graph problem?

## Solution

#### **Overview**

As revealed by the hints, the problem can be solved with two important data structures, namely Graph and Union-Find.

In the following sections, we will explain how to solve the problem respectively with regards to the data structures.

#### **Approach 1: Path Search in Graph**

**Intuition**

First, let us look at the example given in the problem description. Given two equations, namely \frac{a}{b} = 2, \space \frac{b}{c} = 3*ba*​=2, *cb*​=3, we could derive the following equations:

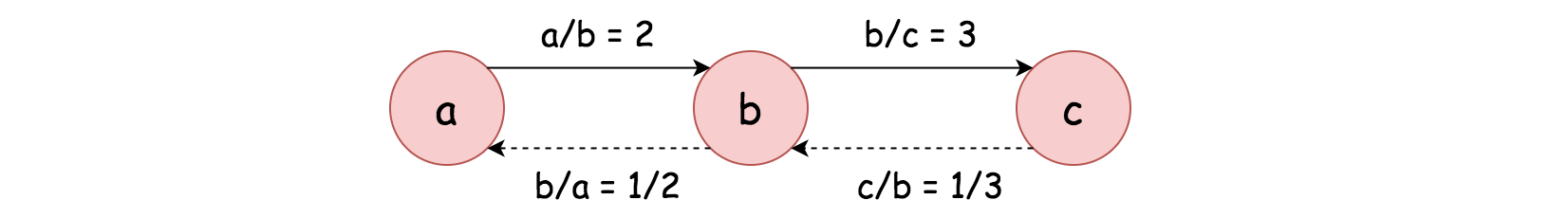
* 1). \frac{b}{a} = \frac{1}{2}, \space \frac{c}{b} = \frac{1}{3}*ab*​=21​, *bc*​=31​
* 2). \frac{a}{c} = \frac{a}{b} \cdot \frac{b}{c} = 2 \cdot 3 = 6*ca*​=*ba*​⋅*cb*​=2⋅3=6

Each division implies the reverse of the division, which is how we derive the equations in **(1)**. While by **chaining up** equations, we could obtain new equations in **(2)**.

We could reformulate the equations with the graph data structure, where each variable can be represented as a **node** in the graph, and the division relationship between variables can be modeled as **edge** with direction and weight.

The direction of edge indicates the order of division, and the weight of edge indicates the result of division.

With the above formulation, we then can convert the initial equations into the following graph:



To evaluate a query (e.g. \frac{a}{c}=?*ca*​=?) is equivalent to performing two tasks on the graph: 1). find if there exists a path between the two entities. 2). if so, calculate the cumulative products along the paths.

In the above example (\frac{a}{c}=?*ca*​=?), we could find a path between them, and the cumulative products are 66. As a result, we can conclude that the result of \frac{a}{c}*ca*​ is 2 \cdot 3 = 62⋅3=6.

**Algorithm**

As one can see, we just transform the problem into a **path searching** problem in a graph.

More precisely, we can reinterpret the problem as "given two nodes, we are asked to check if there exists a path between them. If so, we should return the cumulative products along the path as the result.

Given the above problem statement, it seems intuitive that one could apply the backtracking algorithm, or sometimes people might call it DFS (Depth-First Search).

Essentially, we can break down the algorithm into two steps overall:

* Step 1). we build the graph out of the list of input equations.
  + Each equation corresponds to two edges in the graph.
* Step 2). once the graph is built, we then can evaluate the query one by one.
  + The evaluation of the query is done via searching the path between the given two variables.
  + Other than the above searching operation, we need to handle two exceptional cases as follows:
  + Case 1): if either of the nodes does not exist in the graph, i.e. the variables did not appear in any of the input equations, then we can assert that no path exists.
  + Case 2): if the origin and the destination are the same node, i.e. \frac{a}{a}*aa*​, we can assume that there exists an invisible self-loop path for each node and the result is one.

Here we give one sample implementation on the backtracking algorithm.

|  |
| --- |
| class Solution {  public double[] calcEquation(List<List<String>> equations, double[] values,  List<List<String>> queries) {  HashMap<String, HashMap<String, Double>> graph = new HashMap<>();  // Step 1). build the graph from the equations  for (int i = 0; i < equations.size(); i++) {  List<String> equation = equations.get(i);  String dividend = equation.get(0), divisor = equation.get(1);  double quotient = values[i];  if (!graph.containsKey(dividend))  graph.put(dividend, new HashMap<String, Double>());  if (!graph.containsKey(divisor))  graph.put(divisor, new HashMap<String, Double>());  graph.get(dividend).put(divisor, quotient);  graph.get(divisor).put(dividend, 1 / quotient);  }  // Step 2). Evaluate each query via bactracking (DFS)  // by verifying if there exists a path from dividend to divisor  double[] results = new double[queries.size()];  for (int i = 0; i < queries.size(); i++) {  List<String> query = queries.get(i);  String dividend = query.get(0), divisor = query.get(1);  if (!graph.containsKey(dividend) || !graph.containsKey(divisor))  results[i] = -1.0;  else if (dividend == divisor)  results[i] = 1.0;  else {  HashSet<String> visited = new HashSet<>();  results[i] = backtrackEvaluate(graph, dividend, divisor, 1, visited);  }  }  return results;  }  private double backtrackEvaluate(HashMap<String, HashMap<String, Double>> graph, String currNode, String targetNode, double accProduct, Set<String> visited) {  // mark the visit  visited.add(currNode);  double ret = -1.0;  Map<String, Double> neighbors = graph.get(currNode);  if (neighbors.containsKey(targetNode))  ret = accProduct \* neighbors.get(targetNode);  else {  for (Map.Entry<String, Double> pair : neighbors.entrySet()) {  String nextNode = pair.getKey();  if (visited.contains(nextNode))  continue;  ret = backtrackEvaluate(graph, nextNode, targetNode,  accProduct \* pair.getValue(), visited);  if (ret != -1.0)  break;  }  }  // unmark the visit, for the next backtracking  visited.remove(currNode);  return ret;  }  } |

Note: with the built graph, one could also apply the **BFS** (Breadth-First Search) algorithm, as opposed to the DFS algorithm we employed.

However, the essence of the solution remains the same, i.e. we are searching for a path in a graph.

**Complexity Analysis**

Let N*N* be the number of input equations and M*M* be the number of queries.

* Time Complexity: \mathcal{O}(M \cdot N)O(*M*⋅*N*)
  + First of all, we iterate through the equations to build a graph. Each equation takes \mathcal{O}(1)O(1) time to process. Therefore, this step will take \mathcal{O}(N)O(*N*) time in total.
  + For each query, we need to traverse the graph. In the worst case, we might need to traverse the entire graph, which could take \mathcal{O}(N)O(*N*). Hence, in total, the evaluation of queries could take M \cdot \mathcal{O}(N) = \mathcal{O}(M \cdot N)*M*⋅O(*N*)=O(*M*⋅*N*).
  + To sum up, the overall time complexity of the algorithm is \mathcal{O}(N) + \mathcal{O}(M \cdot N) = \mathcal{O}(M \cdot N)O(*N*)+O(*M*⋅*N*)=O(*M*⋅*N*)
* Space Complexity: \mathcal{O}(N)O(*N*)
  + We build a graph out the equations. In the worst case where there is no overlapping among the equations, we would have N*N* edges and 2N2*N* nodes in the graph. Therefore, the sapce complexity of the graph is \mathcal{O}(N + 2N) = \mathcal{O}(3N) = \mathcal{O}(N)O(*N*+2*N*)=O(3*N*)=O(*N*).
  + Since we employ the recursion in the backtracking, we would consume additional memory in the function call stack, which could amount to \mathcal{O}(N)O(*N*) space.
  + In addition, we used a set visited to keep track of the nodes we visited during the backtracking. The space complexity of the visited set would be \mathcal{O}(N)O(*N*).
  + To sum up, the overall space complexity of the algorithm is \mathcal{O}(N) + \mathcal{O}(N) + \mathcal{O}(N) = \mathcal{O}(N)O(*N*)+O(*N*)+O(*N*)=O(*N*).
  + Note that we did not take into account the space needed to hold the results. Otherwise, the total space complexity would be \mathcal{O}(N + M)O(*N*+*M*).

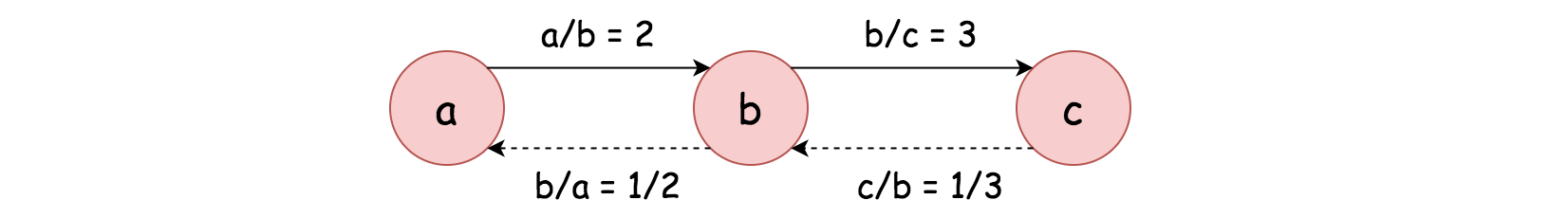
#### **Approach 2: Union-Find with Weights**

**Intuition**

As we mentioned before, the problem can also be solved with the **Union-Find** data structure and algorithm.

As a reminder, the [Union-Find](https://en.wikipedia.org/wiki/Disjoint-set_data_structure) data structure, also known as Disjoint Set, is used to track a set of elements partitioned into a number of disjoint (non-overlapping) subsets. The Union-Find data structure is often applied to solve the **graph partition** problem, where we partition a graph into a set of inter-connected subgraph.

Given the above description, it is not immediately evident that how we can apply the algorithm in this problem. To get familiar with the Union-Find data structure, we would recommend one to check out another problem called [Largest Component Size by Common Factor](https://leetcode.com/problems/largest-component-size-by-common-factor/), where we can apply the Union-Find algorithm in a more **canonical** way.



One thing is clear though. Thanks to the characteristic of the Union-Find data structure, we can easily determine whether the nodes of a and c belong to the same group in the above graph, which accomplishes the first task that we need to perform, i.e. determining if there exists a path between two nodes.

However, one important task is missing, which is how can we calculate the cumulative product along the path, with the Union-Find data structure.

As a spoiler alert, it suffices to adapt the Union-Find data structure and algorithm a little bit.

**Customized Union-Find Data Structure**

The name of Union-Find data structure is originated from the fact that it mainly consists of two operations: Union() and Find() defined as follows:

* Find(x): get the identity of the group that the element x belongs to.
* Union(x, y): merge the two groups that the two elements belong to respectively.

Now, here are the adaptions that we will do. Or more precisely, here are a few **behaviors** that our customized Union-Find data structure would possess at the end.

First of all, essentially we will build a table which contains an entry for each node in the graph, with the help of Union-Find.

The entry is defined as key -> (group\_id, weight). For example, initially, given a node a, its entry in the table would look like 'a' -> ('a', 1), where the first 'a'indicates the id of the node, the second 'a' indicates the id of the group that the node belongs to, and finally the value 1 indicates the weight of the node.

With the above definitions, the tasks become simple. Given two nodes (variables a and b) with entries as (a\_group\_id, a\_weight) and (b\_group\_id, b\_weight) respectively, to evaluate the query of \frac{a}{b} = ?*ba*​=?, we only need to perform the following two calculations:

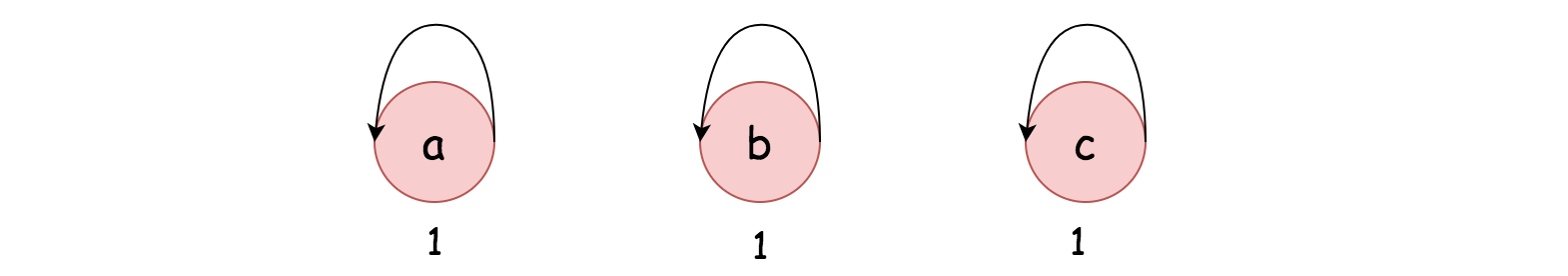
* a\_group\_id == b\_group\_id: If so, they belong to the same group, i.e. there exists a path between them.
* a\_weight / b\_weight: If the above condition holds, by dividing over the **relative** weights that are assigned to the variables, we then can obtain the result of \frac{a}{b}*ba*​ at the end.

Now it all boils down to how we can build the above table with the help of Union-Find algorithm.

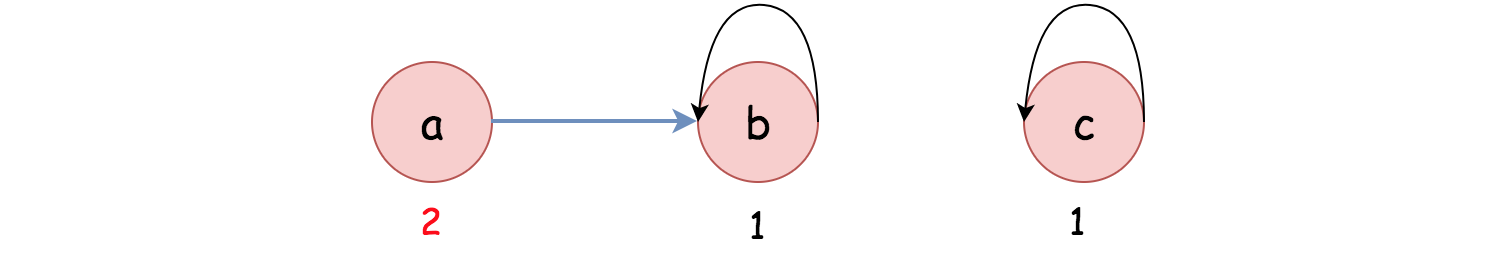
Again, let us look at the same example we presented before.

We have two equations as input, namely \frac{a}{b} = 2, \space \frac{b}{c} = 3*ba*​=2, *cb*​=3.

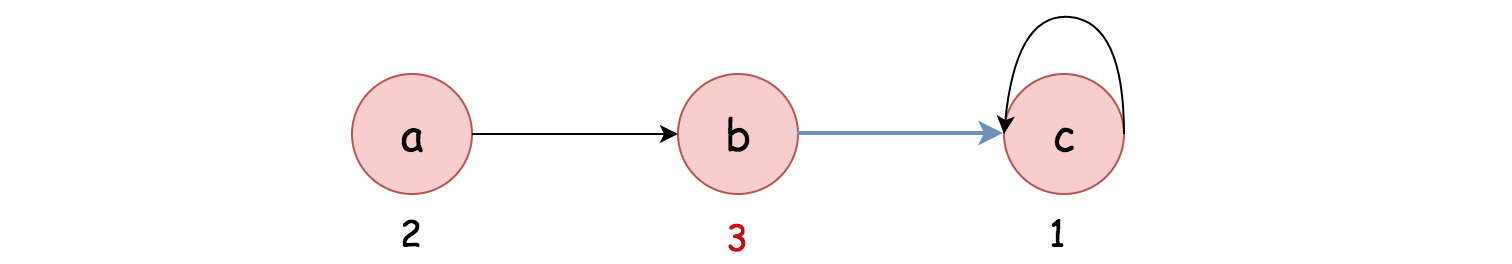
* Initially, the entries for each variable would look like the following, where the group\_id of each variable is the variable itself and the weight of each variable is 1. Each variable forms a group on its own, since there is no relationship among them at the moment.



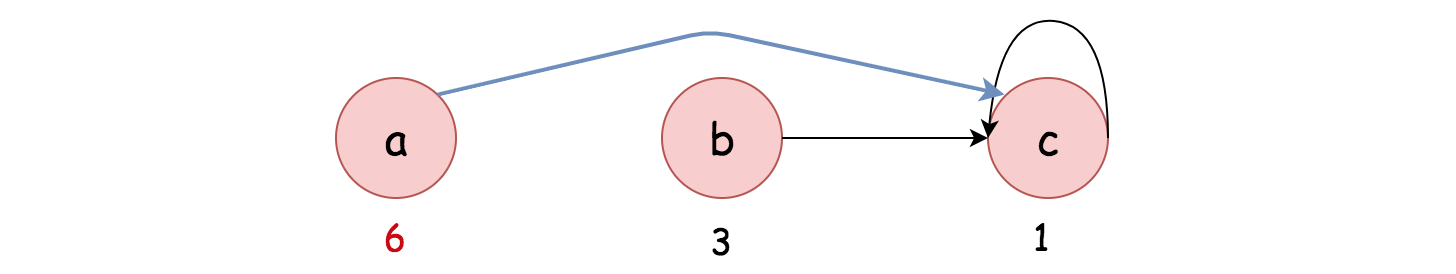
* Now if we process the equation \frac{a}{b}=2*ba*​=2, by joining (**Union** operation) the two groups that the variables a and b belong to, we would obtain the results as shown in the following graph. More precisely, we attach the group of dividend a to the one of the divisor b. Meanwhile, we would also update the relative weight of the group a to reflect the ratio between the two variables.



* Similarly, we continue to process the equation of \frac{b}{c}=3*cb*​=3, by joining (**Union** operation) the groups of b and c together. Similarly, we attach the group of dividend b to the one of divisor c. And also we update the weight of the group b to reflect the ratio between the two variables.



* As one might notice, there is some inconsistency in the above graph, i.e. the group\_id of the variable of a should then be c and the weight of the variable a should be 6 rather than 2. Indeed, these inconsistencies are expected. The **magic** happens when we invoke the **Find** operation on the variable a, where a chain reaction would be triggered to update the group\_id and weight along the chain, as follows:



Once the **lazy** evaluation of find() is triggered, the states of the nodes along the chain would then be updated, and eventually they become consistent.

The mechanism of update is fairly similar with the DFS traversal, as one will see more in detail in the implementation later.

**Algorithm**

Now that we have defined the behaviors for the desired Union-Find data structure, let us put them down into implementation.

The overall interfaces of our Union-Find data structure remain the same. We will implement two functions: find(variable) and union(dividend, divisor, quotient).

* find(variable): the function will return the group\_id that the variable belongs to. Moreover, the function will update the states of variables along the chain, if there is any discrepancy.
* union(dividend, divisor, quotient): this function will attach the group of dividend to that of the divisor, if they are not already the same group. In addition, it needs to update the weight of the dividend variable accordingly, so that the ratio between the dividend and divisor is respected.

We present a sample implementation of the above two functions in the later section, which is inspired from the post of [WangQiuc](https://leetcode.com/problems/evaluate-division/discuss/270993/Python-BFS-and-UF(detailed-explanation)) in the discussion forum. Concise the implementation might be, it might be tricky to wrap one's head around it. One might want to refer to the step-wise example we showed before.

Once we implement the above two functions, we then solve the problem in two steps:

* Step 1): we iterate through each input equation, and invoke the union(dividend, divisor, quotient) on each of them, in order to build the Union-Find data structure.
* Step 2): we evaluate the query one by one. The evaluation is just as intuitive as our first approach, which can be broken down into the following cases:
  + case 1): Either of the variables did not appear in the input equations. The query is not valid. We then return -1.0 as the result.
  + case 2): If both variables are valid, we then apply the find(variable) to obtain the tuple of (group\_id, weight) for each variable. If they are not of the same group, i.e. there is no chain of division between them, we then return -1.0 as the result.
  + case 3): Finally if both variables are of the same group, then we simply perform the division between their weights as the result.

|  |
| --- |
| class Solution {  public double[] calcEquation(List<List<String>> equations, double[] values,  List<List<String>> queries) {  HashMap<String, Pair<String, Double>> gidWeight = new HashMap<>();  // Step 1). build the union groups  for (int i = 0; i < equations.size(); i++) {  List<String> equation = equations.get(i);  String dividend = equation.get(0), divisor = equation.get(1);  double quotient = values[i];  union(gidWeight, dividend, divisor, quotient);  }  // Step 2). run the evaluation, with "lazy" updates in find() function  double[] results = new double[queries.size()];  for (int i = 0; i < queries.size(); i++) {  List<String> query = queries.get(i);  String dividend = query.get(0), divisor = query.get(1);  if (!gidWeight.containsKey(dividend) || !gidWeight.containsKey(divisor))  // case 1). at least one variable did not appear before  results[i] = -1.0;  else {  Pair<String, Double> dividendEntry = find(gidWeight, dividend);  Pair<String, Double> divisorEntry = find(gidWeight, divisor);  String dividendGid = dividendEntry.getKey();  String divisorGid = divisorEntry.getKey();  Double dividendWeight = dividendEntry.getValue();  Double divisorWeight = divisorEntry.getValue();  if (!dividendGid.equals(divisorGid))  // case 2). the variables do not belong to the same chain/group  results[i] = -1.0;  else  // case 3). there is a chain/path between the variables  results[i] = dividendWeight / divisorWeight;  }  }  return results;  }  private Pair<String, Double> find(HashMap<String, Pair<String, Double>> gidWeight, String nodeId) {  if (!gidWeight.containsKey(nodeId))  gidWeight.put(nodeId, new Pair<String, Double>(nodeId, 1.0));  Pair<String, Double> entry = gidWeight.get(nodeId);  // found inconsistency, trigger chain update  if (!entry.getKey().equals(nodeId)) {  Pair<String, Double> newEntry = find(gidWeight, entry.getKey());  gidWeight.put(nodeId, new Pair<String, Double>(  newEntry.getKey(), entry.getValue() \* newEntry.getValue()));  }  return gidWeight.get(nodeId);  }  private void union(HashMap<String, Pair<String, Double>> gidWeight, String dividend, String divisor, Double value) {  Pair<String, Double> dividendEntry = find(gidWeight, dividend);  Pair<String, Double> divisorEntry = find(gidWeight, divisor);  String dividendGid = dividendEntry.getKey();  String divisorGid = divisorEntry.getKey();  Double dividendWeight = dividendEntry.getValue();  Double divisorWeight = divisorEntry.getValue();  // merge the two groups together,  // by attaching the dividend group to the one of divisor  if (!dividendGid.equals(divisorGid)) {  gidWeight.put(dividendGid, new Pair<String, Double>(divisorGid,  divisorWeight \* value / dividendWeight));  }  }  } |

**Complexity Analysis**

Since we applied the Union-Find data structure in our algorithm, we would like to start with a statement on the time complexity of the data structure, as follows:

**Statement**: If M*M* operations, either Union or Find, are applied to N*N* elements, the total run time is \mathcal{O}(M \cdot \log^{\*}{N})O(*M*⋅log∗*N*), where \log^{\*}log∗ is the [iterated logarithm](https://en.wikipedia.org/wiki/Iterated_logarithm).

One can refer to the [proof of Union-Find complexity](https://en.wikipedia.org/wiki/Proof_of_O(log*n)_time_complexity_of_union%E2%80%93find) for more details.

In our case, the maximum number of elements in the Union-Find data structure is equal to twice of the number of equations, *i.e.* each equation introduces two new variables.

Let N*N* be the number of input equations and M*M* be the number of queries.

* Time Complexity: \mathcal{O}\big( (M+N) \cdot \log^{\*}N\big)O((*M*+*N*)⋅log∗*N*).
  + First of all, we iterate through each input equation and invoke union() upon it. As a result, the overall time complexity of this step is \mathcal{O}\big(N \cdot \log^{\*}N\big)O(*N*⋅log∗*N*).
  + As the second step, we then evaluate the query one by one. For each evaluation, we invoke the find() function at most twice. Therefore, the overall time complexity of this step is \mathcal{O}\big(M \cdot \log^{\*}N\big)O(*M*⋅log∗*N*).
  + To sum up, the total time complexity of the algorithm is \mathcal{O}\big( (M+N) \cdot \log^{\*}N\big)O((*M*+*N*)⋅log∗*N*).
  + Note, as compared to the DFS/BFS search approach, Union-Find data structure is more **efficient** for the repetitive/redundant query scenario.
  + Once we evaluate a query with Union-Find, all the subsequent repetitive queries or any query that has the overlapping with the previous query in terms of variable group could be evaluated in **constant time**. For instance, in the above example, once the query of \frac{a}{c}*ca*​ is evaluated, if later we want to evaluate \frac{a}{b}*ba*​, we could instantly obtain the states of variables a and b without triggering the chain update again. While for DFS/BFS approaches, the cost of evaluating each query is independent for each other.
* Space Complexity: \mathcal{O}(N)O(*N*)
  + The space complexity of our Union-Find data structure is \mathcal{O}(N)O(*N*) where we maintain a state for each variable.
  + Since the find() function is implemented with recursion, there would be some additional memory consumption in function call stack, which could amount to \mathcal{O}(N)O(*N*).
  + To sum up, the total space complexity of the algorithm is \mathcal{O}(N) + \mathcal{O}(N) = \mathcal{O}(N)O(*N*)+O(*N*)=O(*N*).
  + Again, we did not take into account the space needed to hold the results. Otherwise, the total space complexity would be \mathcal{O}(N + M)O(*N*+*M*).

**Diameter of Binary Tree**

Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the **longest** path between any two nodes in a tree. This path may or may not pass through the root.

**Example:**  
Given a binary tree

1

/ \

2 3

/ \

4 5

Return **3**, which is the length of the path [4,2,1,3] or [5,2,1,3].

**Note:** The length of path between two nodes is represented by the number of edges between them.

#### **Approach #1: Depth-First Search [Accepted]**

**Intuition**

Any path can be written as two arrows (in different directions) from some node, where an arrow is a path that starts at some node and only travels down to child nodes.

If we knew the maximum length arrows L, R for each child, then the best path touches L + R + 1 nodes.

**Algorithm**

Let's calculate the depth of a node in the usual way: max(depth of node.left, depth of node.right) + 1. While we do, a path "through" this node uses 1 + (depth of node.left) + (depth of node.right) nodes. Let's search each node and remember the highest number of nodes used in some path. The desired length is 1 minus this number.

|  |
| --- |
| class Solution {  int ans;  public int diameterOfBinaryTree(TreeNode root) {  ans = 1;  depth(root);  return ans - 1;  }  public int depth(TreeNode node) {  if (node == null) return 0;  int L = depth(node.left);  int R = depth(node.right);  ans = Math.max(ans, L+R+1);  return Math.max(L, R) + 1;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*). We visit every node once.
* Space Complexity: O(N)*O*(*N*), the size of our implicit call stack during our depth-first search.

**Cracking the Safe**

There is a box protected by a password. The password is a sequence of n digits where each digit can be one of the first k digits 0, 1, ..., k-1.

While entering a password, the last n digits entered will automatically be matched against the correct password.

For example, assuming the correct password is "345", if you type "012345", the box will open because the correct password matches the suffix of the entered password.

Return any password of **minimum length** that is guaranteed to open the box at some point of entering it.

**Example 1:**

**Input:** n = 1, k = 2

**Output:** "01"

**Note:** "10" will be accepted too.

**Example 2:**

**Input:** n = 2, k = 2

**Output:** "00110"

**Note:** "01100", "10011", "11001" will be accepted too.

**Note:**

1. n will be in the range [1, 4].
2. k will be in the range [1, 10].
3. k^n will be at most 4096.

We can think of this problem as the problem of finding an Euler path (a path visiting every edge exactly once) on the following graph: there are $$k^{n-1}$$ nodes with each node having $$k$$ edges. It turns out this graph always has an Eulerian circuit (path starting where it ends.) We should visit each node in "post-order" so as to not get stuck in the graph prematurely.

#### **Approach #1: Hierholzer's Algorithm [Accepted]**

**Intuition**

We can think of this problem as the problem of finding an Euler path (a path visiting every edge exactly once) on the following graph: there are k^{n-1}*kn*−1 nodes with each node having k*k* edges.

For example, when k = 4, n = 3, the nodes are '00', '01', '02', ..., '32', '33' and each node has 4 edges '0', '1', '2', '3'. A node plus edge represents a complete edge and viewing that substring in our answer.

Any connected directed graph where all nodes have equal in-degree and out-degree has an Euler circuit (an Euler path ending where it started.) Because our graph is highly connected and symmetric, we should expect intuitively that taking any path greedily in some order will probably result in an Euler path.

This intuition is called Hierholzer's algorithm: whenever there is an Euler cycle, we can construct it greedily. The algorithm goes as follows:

* Starting from a vertex u, we walk through (unwalked) edges until we get stuck. Because the in-degrees and out-degrees of each node are equal, we can only get stuck at u, which forms a cycle.
* Now, for any node v we had visited that has unwalked edges, we start a new cycle from v with the same procedure as above, and then merge the cycles together to form a new cycle u \rightarrow \dots \rightarrow v \rightarrow \dots \rightarrow v \rightarrow \dots \rightarrow u*u*→⋯→*v*→⋯→*v*→⋯→*u*.

**Algorithm**

We will modify our standard depth-first search: instead of keeping track of nodes, we keep track of (complete) edges: seen records if an edge has been visited.

Also, we'll need to visit in a sort of "post-order", recording the answer after visiting the edge. This is to prevent getting stuck. For example, with k = 2, n = 2, we have the nodes '0', '1'. If we greedily visit complete edges '00', '01', '10', we will be stuck at the node '0' prematurely. However, if we visit in post-order, we'll end up visiting '00', '01', '11', '10' correctly.

In general, during our Hierholzer walk, we will record the results of other subcycles first, before recording the main cycle we started from, just as in our first description of the algorithm. Technically, we are recording backwards, as we exit the nodes.

For example, we will walk (in the "original cycle") until we get stuck, then record the node as we exit. (Every edge walked is always marked immediately so that it can no longer be used.) Then in the penultimate node of our original cycle, we will do a Hierholzer walk and then record this node; then in the third-last node of our original cycle we will do a Hierholzer walk and then record this node, and so on.

|  |
| --- |
| class Solution {  Set<String> seen;  StringBuilder ans;  public String crackSafe(int n, int k) {  if (n == 1 && k == 1) return "0";  seen = new HashSet();  ans = new StringBuilder();  StringBuilder sb = new StringBuilder();  for (int i = 0; i < n-1; ++i)  sb.append("0");  String start = sb.toString();  dfs(start, k);  ans.append(start);  return new String(ans);  }  public void dfs(String node, int k) {  for (int x = 0; x < k; ++x) {  String nei = node + x;  if (!seen.contains(nei)) {  seen.add(nei);  dfs(nei.substring(1), k);  ans.append(x);  }  }  }  } |

**Complexity Analysis**

* Time Complexity: O(n \* k^n)*O*(*n*∗*kn*). We visit every edge once in our depth-first search, and nodes take O(n)*O*(*n*) space.
* Space Complexity: O(n \* k^n)*O*(*n*∗*kn*), the size of seen.

#### **Approach #2: Inverse Burrows-Wheeler Transform [Accepted]**

**Explanation**

If we are familiar with the theory of combinatorics on words, recall that a Lyndon Word L is a word that is the unique minimum of it's rotations.

One important mathematical result (due to [Fredericksen and Maiorana](http://www-igm.univ-mlv.fr/~perrin/Recherche/Publications/Articles/debruijnRevised3.pdf)), is that the concatenation in lexicographic order of Lyndon words with length dividing n, forms a de Bruijin sequence: a sequence where every every word (from the k^n*kn* available) appears as a substring of length n (where we are allowed to wrap around.)

For example, when n = 6, k = 2, all the Lyndon words with length dividing n in lexicographic order are (spaces for convenience): 0 000001 000011 000101 000111 001 001011 001101 001111 01 010111 011 011111 1. It turns out this is the smallest de Bruijin sequence.

We can use the Inverse Burrows-Wheeler Transform (IBWT) to generate these Lyndon words. Consider two sequences: S is the alphabet repeated k^{n-1}*kn*−1 times: S = 0123...0123...0123...., and S' is the alphabet repeated k^{n-1}*kn*−1 times for each letter: S' = 00...0011...1122.... We can think of S' and S as defining a permutation, where the j-th occurrence of each letter of the alphabet in S' maps to the corresponding j-th occurrence in S. The cycles of this permutation turn out to be the corresponding smallest de Bruijin sequence ([link](http://www.macs.hw.ac.uk/~markl/Higgins.pdf)).

Under this view, the permutation S' \rightarrow S*S*′→*S* [mapping permutation indices (i \* k^{n-1} + q) \rightarrow (q \* k + i)(*i*∗*kn*−1+*q*)→(*q*∗*k*+*i*)] form the desired Lyndon words.

|  |
| --- |
| class Solution {  public String crackSafe(int n, int k) {  int M = (int) Math.pow(k, n-1);  int[] P = new int[M \* k];  for (int i = 0; i < k; ++i)  for (int q = 0; q < M; ++q)  P[i\*M + q] = q\*k + i;  StringBuilder ans = new StringBuilder();  for (int i = 0; i < M\*k; ++i) {  int j = i;  while (P[j] >= 0) {  ans.append(String.valueOf(j / M));  int v = P[j];  P[j] = -1;  j = v;  }  }  for (int i = 0; i < n-1; ++i)  ans.append("0");  return new String(ans);  }  } |

**Complexity Analysis**

* Time Complexity: O(k^n)*O*(*kn*). We loop through every possible substring.
* Space Complexity: O(k^n)*O*(*kn*), the size of P and ans.

**Robot Room Cleaner**

Given a robot cleaner in a room modeled as a grid.

Each cell in the grid can be empty or blocked.

The robot cleaner with 4 given APIs can move forward, turn left or turn right. Each turn it made is 90 degrees.

When it tries to move into a blocked cell, its bumper sensor detects the obstacle and it stays on the current cell.

Design an algorithm to clean the entire room using only the 4 given APIs shown below.

interface Robot {

  // returns true if next cell is open and robot moves into the cell.

  // returns false if next cell is obstacle and robot stays on the current cell.

  boolean move();

// Robot will stay on the same cell after calling turnLeft/turnRight.

  // Each turn will be 90 degrees.

  void turnLeft();

  void turnRight();

// Clean the current cell.

void clean();

}

**Example:**

**Input:**

room = [

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

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

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

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

[1,1,1,1,1,1,1,1]

],

row = 1,

col = 3

**Explanation:**

All grids in the room are marked by either 0 or 1.

0 means the cell is blocked, while 1 means the cell is accessible.

The robot initially starts at the position of row=1, col=3.

From the top left corner, its position is one row below and three columns right.

**Notes:**

1. The input is only given to initialize the room and the robot's position internally. You must solve this problem "blindfolded". In other words, you must control the robot using only the mentioned 4 APIs, without knowing the room layout and the initial robot's position.
2. The robot's initial position will always be in an accessible cell.
3. The initial direction of the robot will be facing up.
4. All accessible cells are connected, which means the all cells marked as 1 will be accessible by the robot.
5. Assume all four edges of the grid are all surrounded by wall.

Recursion 2

**Most Stones Removed with Same Row or Column**

On a 2D plane, we place n stones at some integer coordinate points. Each coordinate point may have at most one stone.

A stone can be removed if it shares either **the same row or the same column** as another stone that has not been removed.

Given an array stones of length n where stones[i] = [xi, yi] represents the location of the ith stone, return *the largest possible number of stones that can be removed*.

**Example 1:**

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

**Output:** 5

**Explanation:** One way to remove 5 stones is as follows:

1. Remove stone [2,2] because it shares the same row as [2,1].

2. Remove stone [2,1] because it shares the same column as [0,1].

3. Remove stone [1,2] because it shares the same row as [1,0].

4. Remove stone [1,0] because it shares the same column as [0,0].

5. Remove stone [0,1] because it shares the same row as [0,0].

Stone [0,0] cannot be removed since it does not share a row/column with another stone still on the plane.

**Example 2:**

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

**Output:** 3

**Explanation:** One way to make 3 moves is as follows:

1. Remove stone [2,2] because it shares the same row as [2,0].

2. Remove stone [2,0] because it shares the same column as [0,0].

3. Remove stone [0,2] because it shares the same row as [0,0].

Stones [0,0] and [1,1] cannot be removed since they do not share a row/column with another stone still on the plane.

**Example 3:**

**Input:** stones = [[0,0]]

**Output:** 0

**Explanation:** [0,0] is the only stone on the plane, so you cannot remove it.

**Constraints:**

* 1 <= stones.length <= 1000
* 0 <= xi, yi <= 104
* No two stones are at the same coordinate point.

Solution

Approach 1: Depth-First Search

**Intuition**

Let's say two stones are connected by an edge if they share a row or column, and define a connected component in the usual way for graphs: a subset of stones so that there doesn't exist an edge from a stone in the subset to a stone not in the subset. For convenience, we refer to a *component* as meaning a connected component.

The main insight is that we can always make moves that reduce the number of stones in each component to 1.

Firstly, every stone belongs to exactly one component, and moves in one component do not affect another component.

Now, consider a spanning tree of our component. We can make moves repeatedly from the leaves of this tree until there is one stone left.

**Algorithm**

To count connected components of the above graph, we will use depth-first search.

For every stone not yet visited, we will visit it and any stone in the same connected component. Our depth-first search traverses each node in the component.

For each component, the answer changes by -1 + component.size.

|  |
| --- |
| class Solution {  public int removeStones(int[][] stones) {  int N = stones.length;  // graph[i][0] = the length of the 'list' graph[i][1:]  int[][] graph = new int[N][N];  for (int i = 0; i < N; ++i)  for (int j = i+1; j < N; ++j)  if (stones[i][0] == stones[j][0] || stones[i][1] == stones[j][1]) {  graph[i][++graph[i][0]] = j;  graph[j][++graph[j][0]] = i;  }  int ans = 0;  boolean[] seen = new boolean[N];  for (int i = 0; i < N; ++i) if (!seen[i]) {  Stack<Integer> stack = new Stack();  stack.push(i);  seen[i] = true;  ans--;  while (!stack.isEmpty()) {  int node = stack.pop();  ans++;  for (int k = 1; k <= graph[node][0]; ++k) {  int nei = graph[node][k];  if (!seen[nei]) {  stack.push(nei);  seen[nei] = true;  }  }  }  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N^2)*O*(*N*2), where N*N* is the length of stones.
* Space Complexity: O(N^2)*O*(*N*2).

#### **Approach 2: Union-Find**

**Intuition**

As in Approach 1, we will need to consider components of an underlying graph. A "Disjoint Set Union" (DSU) data structure is ideal for this.

We will skip the explanation of how a DSU structure is implemented. Please refer to <https://leetcode.com/problems/redundant-connection/solution/> for a tutorial on DSU.

**Algorithm**

Let's connect row i to column j, which will be represented by j+10000. The answer is the number of components after making all the connections.

Note that for brevity, our DSU implementation does not use union-by-rank. This makes the asymptotic time complexity larger.

|  |
| --- |
| class Solution {  public int removeStones(int[][] stones) {  int N = stones.length;  DSU dsu = new DSU(20000);  for (int[] stone: stones)  dsu.union(stone[0], stone[1] + 10000);  Set<Integer> seen = new HashSet();  for (int[] stone: stones)  seen.add(dsu.find(stone[0]));  return N - seen.size();  }  }  class DSU {  int[] parent;  public DSU(int N) {  parent = new int[N];  for (int i = 0; i < N; ++i)  parent[i] = i;  }  public int find(int x) {  if (parent[x] != x) parent[x] = find(parent[x]);  return parent[x];  }  public void union(int x, int y) {  parent[find(x)] = find(y);  }  } |

**Complexity Analysis**

* Time Complexity: O(N \log N)*O*(*N*log*N*), where N*N* is the length of stones. (If we used union-by-rank, this can be O(N \* \alpha(N))*O*(*N*∗*α*(*N*)), where \alpha*α* is the Inverse-Ackermann function.)
* Space Complexity: O(N)*O*(*N*).

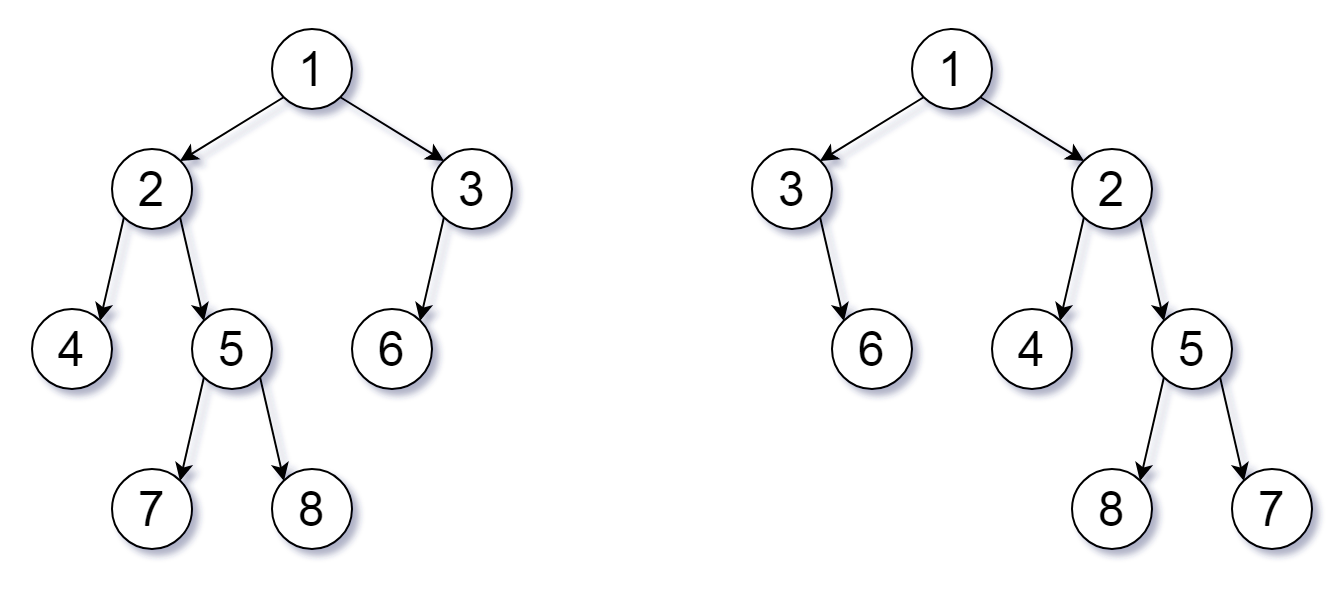
**Flip Equivalent Binary Trees**

For a binary tree **T**, we can define a **flip operation** as follows: choose any node, and swap the left and right child subtrees.

A binary tree **X** is *flip equivalent* to a binary tree **Y** if and only if we can make **X** equal to **Y** after some number of flip operations.

Given the roots of two binary trees root1 and root2, return true if the two trees are flip equivelent or false otherwise.

**Example 1:**



**Input:** root1 = [1,2,3,4,5,6,null,null,null,7,8], root2 = [1,3,2,null,6,4,5,null,null,null,null,8,7]

**Output:** true

**Explanation:** We flipped at nodes with values 1, 3, and 5.

**Example 2:**

**Input:** root1 = [], root2 = []

**Output:** true

**Example 3:**

**Input:** root1 = [], root2 = [1]

**Output:** false

**Example 4:**

**Input:** root1 = [0,null,1], root2 = []

**Output:** false

**Example 5:**

**Input:** root1 = [0,null,1], root2 = [0,1]

**Output:** true

**Constraints:**

* The number of nodes in each tree is in the range [0, 100].
* Each tree will have **unique node values** in the range [0, 99].

## Solution

#### **Approach 1: Recursion**

**Intuition**

If root1 and root2 have the same root value, then we only need to check if their children are equal (up to ordering.)

**Algorithm**

There are 3 cases:

* If root1 or root2 is null, then they are equivalent if and only if they are both null.
* Else, if root1 and root2 have different values, they aren't equivalent.
* Else, let's check whether the children of root1 are equivalent to the children of root2. There are two different ways to pair these children.

|  |
| --- |
| class Solution {  public boolean flipEquiv(TreeNode root1, TreeNode root2) {  if (root1 == root2)  return true;  if (root1 == null || root2 == null || root1.val != root2.val)  return false;  return (flipEquiv(root1.left, root2.left) && flipEquiv(root1.right, root2.right) ||  flipEquiv(root1.left, root2.right) && flipEquiv(root1.right, root2.left));  }  } |

**Complexity Analysis**

* Time Complexity: O(min(N\_1, N\_2))*O*(*min*(*N*1​,*N*2​)), where N\_1, N\_2*N*1​,*N*2​ are the lengths of root1 and root2.
* Space Complexity: O(min(H\_1, H\_2))*O*(*min*(*H*1​,*H*2​)), where H\_1, H\_2*H*1​,*H*2​ are the heights of the trees of root1 and root2.

#### **Approach 2: Canonical Traversal**

**Intuition**

Flip each node so that the left child is smaller than the right, and call this the canonical representation. All equivalent trees have exactly one canonical representation.

**Algorithm**

We can use a depth-first search to compare the canonical representation of each tree. If the traversals are the same, the representations are equal.

When traversing, we should be careful to encode both when we enter or leave a node.

|  |
| --- |
| class Solution {  public boolean flipEquiv(TreeNode root1, TreeNode root2) {  List<Integer> vals1 = new ArrayList();  List<Integer> vals2 = new ArrayList();  dfs(root1, vals1);  dfs(root2, vals2);  return vals1.equals(vals2);  }  public void dfs(TreeNode node, List<Integer> vals) {  if (node != null) {  vals.add(node.val);  int L = node.left != null ? node.left.val : -1;  int R = node.right != null ? node.right.val : -1;  if (L < R) {  dfs(node.left, vals);  dfs(node.right, vals);  } else {  dfs(node.right, vals);  dfs(node.left, vals);  }  vals.add(null);  }  }  } |

**Complexity Analysis**

* Time Complexity: O(N\_1 + N\_2)*O*(*N*1​+*N*2​), where N\_1, N\_2*N*1​,*N*2​ are the lengths of root1 and root2. (In Python, this is \min(N\_1, N\_2)min(*N*1​,*N*2​).)
* Space Complexity: O(N\_1 + N\_2)*O*(*N*1​+*N*2​). (In Python, this is \min(H\_1, H\_2)min(*H*1​,*H*2​), where H\_1, H\_2*H*1​,*H*2​ are the heights of the trees of root1 and root2.)

## Recursion

Recursion usually involves some kind of backtracking to enumerate all possibilities.

Note that [Recursion](https://leetcode.com/explore/learn/card/recursion-i/) is a more general purpose algorithm. [Depth-First search](http://en.wikipedia.org/wiki/Depth-first_search) is a specific form of backtracking related to searching tree data structures. Therefore we categorize those problems in "Trees and Graphs", even though they involve recursion.

For a great introduction of how backtracking works, please check out LeetCode's [Recursion II card](https://leetcode.com/explore/learn/card/recursion-ii/). A great example is "Word Search II" (aka the Boggle solver), which uses a data structure to optimize the search.

**Word Squares**

Given a set of words **(without duplicates)**, find all [word squares](https://en.wikipedia.org/wiki/Word_square) you can build from them.

A sequence of words forms a valid word square if the *k*th row and column read the exact same string, where 0 ≤ *k* < max(numRows, numColumns).

For example, the word sequence ["ball","area","lead","lady"] forms a word square because each word reads the same both horizontally and vertically.

b a l l

a r e a

l e a d

l a d y

**Note:**

1. There are at least 1 and at most 1000 words.
2. All words will have the exact same length.
3. Word length is at least 1 and at most 5.
4. Each word contains only lowercase English alphabet a-z.

**Example 1:**

**Input:**

["area","lead","wall","lady","ball"]

**Output:**

[

[ "wall",

"area",

"lead",

"lady"

],

[ "ball",

"area",

"lead",

"lady"

]

]

**Explanation:**

The output consists of two word squares. The order of output does not matter (just the order of words in each word square matters).

**Example 2:**

**Input:**

["abat","baba","atan","atal"]

**Output:**

[

[ "baba",

"abat",

"baba",

"atan"

],

[ "baba",

"abat",

"baba",

"atal"

]

]

**Explanation:**

The output consists of two word squares. The order of output does not matter (just the order of words in each word square matters).

Tries

**Strobogrammatic Number II**

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Find all strobogrammatic numbers that are of length = n.

**Example:**

**Input:** n = 2

**Output:** ["11","69","88","96"]

   Hide Hint #1

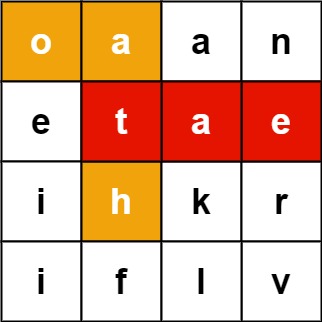
Try to use recursion and notice that it should recurse with *n* - 2 instead of *n* - 1.

**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.

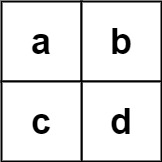
**Example 1:**



**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

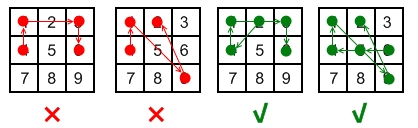
Hard

**Android Unlock Patterns**

Android devices have a special lock screen with a 3 x 3 grid of dots. Users can set an "unlock pattern" by connecting the dots in a specific sequence, forming a series of joined line segments where each segment's endpoints are two consecutive dots in the sequence. A sequence of k dots is a **valid** unlock pattern if both of the following are true:

* All the dots in the sequence are **distinct**.
* If the line segment connecting two consecutive dots in the sequence passes through any other dot, the other dot **must have previously appeared** in the sequence. No jumps through non-selected dots are allowed.

Here are some example valid and invalid unlock patterns:



* The 1st pattern [4,1,3,6] is invalid because the line connecting dots 1 and 3 pass through dot 2, but dot 2 did not previously appear in the sequence.
* The 2nd pattern [4,1,9,2] is invalid because the line connecting dots 1 and 9 pass through dot 5, but dot 5 did not previously appear in the sequence.
* The 3rd pattern [2,4,1,3,6] is valid because it follows the conditions. The line connecting dots 1 and 3 meets the condition because dot 2 previously appeared in the sequence.
* The 4th pattern [6,5,4,1,9,2] is valid because it follows the conditions. The line connecting dots 1 and 9 meets the condition because dot 5 previously appeared in the sequence.

Given two integers m and n, return *the****number of unique and valid unlock patterns****of the Android grid lock screen that consist of****at least***m*keys and****at most***n*keys.*

Two unlock patterns are considered **unique** if there is a dot in one sequence that is not in the other, or the order of the dots is different.

**Example 1:**

**Input:** m = 1, n = 1

**Output:** 9

**Example 2:**

**Input:** m = 1, n = 2

**Output:** 65

**Constraints:**

* 1 <= m, n <= 9

## Summary

After Android launched its "unlock pattern" system to protect our smart phones from unauthorized access, the most common question that comes to one's mind is: How secure exactly are these patterns? The current article gives an answer to this question, as presenting an algorithm, which computes the number of all valid pattern combinations. It is intended for intermediate users and introduces the following ideas: Backtracking, Arrays.

## Solution

#### **Approach #1: (Backtracking) [Accepted]**

**Algorithm**

The algorithm uses backtracking technique to enumerate all possible k*k* combinations of numbers [1\dots 9][1…9] where m \leq k \leq n*m*≤*k*≤*n*. During the generation of the recursive solution tree, the algorithm cuts all the branches which lead to patterns which doesn't satisfy the rules and counts only the valid patterns. In order to compute a valid pattern, the algorithm performs the following steps:

* Select a digit i*i* which is not used in the pattern till this moment. This is done with the help of a used*used* array which stores all available digits.
* We need to keep last inserted digit last*last*. The algorithm makes a check whether one of the following conditions is valid.
  + There is a knight move (as in chess) from last*last* towards i*i* or last*last* and i*i* are adjacent digits in a row, in a column. In this case the sum of both digits should be an odd number.
  + The middle element mid*mid* in the line which connects i*i* and last*last* was previously selected. In case i*i* and last*last* are positioned at both ends of the diagonal, digit mid*mid* = 5 should be previously selected.
  + last*last* and i*i* are adjacent digits in a diagonal

In case one of the conditions above is satisfied, digit i*i* becomes part of partially generated valid pattern and the algorithm continues with the next candidate digit till the pattern is fully generated. Then it counts it. In case none of the conditions are satisfied, the algorithm rejects the current digit i*i*, backtracks and continues to search for other valid digits among the unused ones.

|  |
| --- |
| public class Solution {  private boolean used[] = new boolean[9];  public int numberOfPatterns(int m, int n) {  int res = 0;  for (int len = m; len <= n; len++) {  res += calcPatterns(-1, len);  for (int i = 0; i < 9; i++) {  used[i] = false;  }  }  return res;  }  private boolean isValid(int index, int last) {  if (used[index])  return false;  // first digit of the pattern  if (last == -1)  return true;  // knight moves or adjacent cells (in a row or in a column)  if ((index + last) % 2 == 1)  return true;  // indexes are at both end of the diagonals for example 0,0, and 8,8  int mid = (index + last)/2;  if (mid == 4)  return used[mid];  // adjacent cells on diagonal - for example 0,0 and 1,0 or 2,0 and //1,1  if ((index%3 != last%3) && (index/3 != last/3)) {  return true;  }  // all other cells which are not adjacent  return used[mid];  }  private int calcPatterns(int last, int len) {  if (len == 0)  return 1;  int sum = 0;  for (int i = 0; i < 9; i++) {  if (isValid(i, last)) {  used[i] = true;  sum += calcPatterns(i, len - 1);  used[i] = false;  }  }  return sum;  }  } |

\*\*Complexity Analysis\*\*

* Time complexity : O( n!)*O*(*n*!), where n*n* is maximum pattern length

The algorithm computes each pattern once and no element can appear in the pattern twice. The time complexity is proportional to the number of the computed patterns. One upper bound of the number of all possible combinations is :

\sum\_{i=m}^{n} {\_9}P\_i = \sum\_{i=m}^{n} \frac{9!}{(9 - i)!}∑*i*=*mn*​9​*Pi*​=∑*i*=*mn*​(9−*i*)!9!​

* Space complexity : O(n)*O*(*n*), where n*n* is maximum pattern length In the worst case the maximum depth of recursion is n*n*. Therefore we need O( n)*O*(*n*) space used by the system recursive stack

## Further Thoughts

The algorithm above could be optimized if we consider the symmetry property of the problem. We notice that the number of valid patterns with first digit 1, 3, 7, 9 are the same. A similar observation is true for patterns which starts with digit 2, 4, 6, 8. Hence we only need to calculate one among each group and multiply by 4.

You can find the optimized solution [here](https://leetcode.com/problems/android-unlock-patterns/discuss/82463/Java-DFS-solution-with-clear-explanations-and-optimization-beats-97.61-(12ms)).

Analysis written by: @elmirap.

**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'].

Medium

**Generate Parentheses**

Given n pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

**Example 1:**

**Input:** n = 3

**Output:** ["((()))","(()())","(())()","()(())","()()()"]

**Example 2:**

**Input:** n = 1

**Output:** ["()"]

**Constraints:**

* 1 <= n <= 8

Medium

## Sorting and Searching

Interval related problems are quite often asked at Google interviews.

Similar to "Arrays and Strings", interval related problems can be asked in the context of [data stream](https://en.wikipedia.org/wiki/Streaming_algorithm#Data_stream_model).

**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

**Hard**

**Find First and Last Position of Element in Sorted Array**

Given an array of integers nums sorted in ascending order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

**Follow up:** Could you write an algorithm with O(log n) runtime complexity?

**Example 1:**

**Input:** nums = [5,7,7,8,8,10], target = 8

**Output:** [3,4]

**Example 2:**

**Input:** nums = [5,7,7,8,8,10], target = 6

**Output:** [-1,-1]

**Example 3:**

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

**Output:** [-1,-1]

**Constraints:**

* 0 <= nums.length <= 105
* -109 <= nums[i] <= 109
* nums is a non-decreasing array.
* -109 <= target <= 109

#### **Approach 1: Linear Scan**

**Intuition**

Checking every index for target exhausts the search space, so it must work.

**Algorithm**

First, we do a linear scan of nums from the left, breaking when we find an instance of target. If we never break, then target is not present, so we can return the "error code" of [-1, -1] early. Given that we did find a valid left index, we can do a second linear scan, but this time from the right. In this case, the first instance of target encountered will be the rightmost one (and because a leftmost one exists, there is guaranteed to also be a rightmost one). We then simply return a list containing the two located indices.

|  |
| --- |
| class Solution {  public int[] searchRange(int[] nums, int target) {  int[] targetRange = {-1, -1};  // find the index of the leftmost appearance of `target`.  for (int i = 0; i < nums.length; i++) {  if (nums[i] == target) {  targetRange[0] = i;  break;  }  }  // if the last loop did not find any index, then there is no valid range  // and we return [-1, -1].  if (targetRange[0] == -1) {  return targetRange;  }  // find the index of the rightmost appearance of `target` (by reverse  // iteration). it is guaranteed to appear.  for (int j = nums.length-1; j >= 0; j--) {  if (nums[j] == target) {  targetRange[1] = j;  break;  }  }  return targetRange;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*)

This brute-force approach examines each of the n elements of nums exactly twice, so the overall runtime is linear.

* Space complexity : O(1)*O*(1)

The linear scan method allocates a fixed-size array and a few integers, so it has a constant-size memory footprint.

#### **Approach 2: Binary Search**

**Intuition**

Because the array is sorted, we can use binary search to locate the left and rightmost indices.

**Algorithm**

The overall algorithm works fairly similarly to the linear scan approach, except for the subroutine used to find the left and rightmost indices themselves. Here, we use a modified binary search to search a sorted array, with a few minor adjustments. First, because we are locating the leftmost (or rightmost) index containing target (rather than returning true iff we find target), the algorithm does not terminate as soon as we find a match. Instead, we continue to search until lo == hi and they contain some index at which target can be found.

The other change is the introduction of the left parameter, which is a boolean indicating what to do in the event that target == nums[mid]; if left is true, then we "recurse" on the left subarray on ties. Otherwise, we go right. To see why this is correct, consider the situation where we find target at index i. The leftmost target cannot occur at any index greater than i, so we never need to consider the right subarray. The same argument applies to the rightmost index.

The first animation below shows the process for finding the leftmost index, and the second shows the process for finding the index right of the rightmost index.

|  |
| --- |
| class Solution {  // returns leftmost (or rightmost) index at which `target` should be  // inserted in sorted array `nums` via binary search.  private int extremeInsertionIndex(int[] nums, int target, boolean left) {  int lo = 0;  int hi = nums.length;  while (lo < hi) {  int mid = (lo + hi) / 2;  if (nums[mid] > target || (left && target == nums[mid])) {  hi = mid;  }  else {  lo = mid+1;  }  }  return lo;  }  public int[] searchRange(int[] nums, int target) {  int[] targetRange = {-1, -1};  int leftIdx = extremeInsertionIndex(nums, target, true);  // assert that `leftIdx` is within the array bounds and that `target`  // is actually in `nums`.  if (leftIdx == nums.length || nums[leftIdx] != target) {  return targetRange;  }  targetRange[0] = leftIdx;  targetRange[1] = extremeInsertionIndex(nums, target, false)-1;  return targetRange;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(\log N)O(log*N*)

Here we invoke binary search twice. Let's compute time complexity with the help of [master theorem](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)) T(N) = aT\left(\frac{N}{b}\right) + \Theta(N^d)*T*(*N*)=*aT*(*bN*​)+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 at step there is only one subproblem a = 1, its size is a half of the 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 2](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)#Case_2_example) that results in \mathcal{O}(n^{\log\_b{a}} \log^{d + 1} N)O(*n*log*b*​*a*log*d*+1*N*) = \mathcal{O}(\log N)O(log*N*) time complexity.

* Space complexity : \mathcal{O}(1)O(1)

All work is done in place, so the overall memory usage is constant.

**Merge Intervals**

Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

**Example 1:**

**Input:** intervals = [[1,3],[2,6],[8,10],[15,18]]

**Output:** [[1,6],[8,10],[15,18]]

**Explanation:** Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].

**Example 2:**

**Input:** intervals = [[1,4],[4,5]]

**Output:** [[1,5]]

**Explanation:** Intervals [1,4] and [4,5] are considered overlapping.

**Constraints:**

* 1 <= intervals.length <= 104
* intervals[i].length == 2
* 0 <= starti <= endi <= 104

## Solution Article

#### **Approach 1: Connected Components**

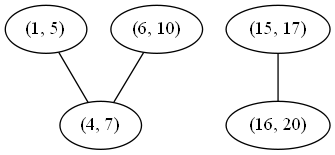
**Intuition**

If we draw a graph (with intervals as nodes) that contains undirected edges between all pairs of intervals that overlap, then all intervals in each connected component of the graph can be merged into a single interval.

**Algorithm**

With the above intuition in mind, we can represent the graph as an adjacency list, inserting directed edges in both directions to simulate undirected edges. Then, to determine which connected component each node is it, we perform graph traversals from arbitrary unvisited nodes until all nodes have been visited. To do this efficiently, we store visited nodes in a Set, allowing for constant time containment checks and insertion. Finally, we consider each connected component, merging all of its intervals by constructing a new Interval with start equal to the minimum start among them and end equal to the maximum end.

This algorithm is correct simply because it is basically the brute force solution. We compare every interval to every other interval, so we know exactly which intervals overlap. The reason for the connected component search is that two intervals may not directly overlap, but might overlap indirectly via a third interval. See the example below to see this more clearly.



Although (1, 5) and (6, 10) do not directly overlap, either would overlap with the other if first merged with (4, 7). There are two connected components, so if we merge their nodes, we expect to get the following two merged intervals:

(1, 10), (15, 20)

|  |
| --- |
| class Solution {  private Map<int[], List<int[]>> graph;  private Map<Integer, List<int[]>> nodesInComp;  private Set<int[]> visited;  // return whether two intervals overlap (inclusive)  private boolean overlap(int[] a, int[] b) {  return a[0] <= b[1] && b[0] <= a[1];  }  // build a graph where an undirected edge between intervals u and v exists  // iff u and v overlap.  private void buildGraph(int[][] intervals) {  graph = new HashMap<>();  for (int[] interval : intervals) {  graph.put(interval, new LinkedList<>());  }  for (int[] interval1 : intervals) {  for (int[] interval2 : intervals) {  if (overlap(interval1, interval2)) {  graph.get(interval1).add(interval2);  graph.get(interval2).add(interval1);  }  }  }  }  // merges all of the nodes in this connected component into one interval.  private int[] mergeNodes(List<int[]> nodes) {  int minStart = nodes.get(0)[0];  for (int[] node : nodes) {  minStart = Math.min(minStart, node[0]);  }  int maxEnd = nodes.get(0)[1];  for (int[] node : nodes) {  maxEnd = Math.max(maxEnd, node[1]);  }  return new int[] {minStart, maxEnd};  }  // use depth-first search to mark all nodes in the same connected component  // with the same integer.  private void markComponentDFS(int[] start, int compNumber) {  Stack<int[]> stack = new Stack<>();  stack.add(start);  while (!stack.isEmpty()) {  int[] node = stack.pop();  if (!visited.contains(node)) {  visited.add(node);  if (nodesInComp.get(compNumber) == null) {  nodesInComp.put(compNumber, new LinkedList<>());  }  nodesInComp.get(compNumber).add(node);  for (int[] child : graph.get(node)) {  stack.add(child);  }  }  }  }  // gets the connected components of the interval overlap graph.  private void buildComponents(int[][] intervals) {  nodesInComp = new HashMap<>();  visited = new HashSet<>();  int compNumber = 0;  for (int[] interval : intervals) {  if (!visited.contains(interval)) {  markComponentDFS(interval, compNumber);  compNumber++;  }  }  }  public int[][] merge(int[][] intervals) {  buildGraph(intervals);  buildComponents(intervals);  // for each component, merge all intervals into one interval.  List<int[]> merged = new LinkedList<>();  for (int comp = 0; comp < nodesInComp.size(); comp++) {  merged.add(mergeNodes(nodesInComp.get(comp)));  }  return merged.toArray(new int[merged.size()][]);  }  } |

**Complexity Analysis**

* Time complexity : O(n^2)*O*(*n*2)

Building the graph costs O(V + E) = O(V) + O(E) = O(n) + O(n^2) = O(n^2)*O*(*V*+*E*)=*O*(*V*)+*O*(*E*)=*O*(*n*)+*O*(*n*2)=*O*(*n*2) time, as in the worst case all intervals are mutually overlapping. Traversing the graph has the same cost (although it might appear higher at first) because our visited set guarantees that each node will be visited exactly once. Finally, because each node is part of exactly one component, the merge step costs O(V) = O(n)*O*(*V*)=*O*(*n*) time. This all adds up as follows:

O(n^2) + O(n^2) + O(n) = O(n^2)*O*(*n*2)+*O*(*n*2)+*O*(*n*)=*O*(*n*2)

* Space complexity : O(n^2)*O*(*n*2)

As previously mentioned, in the worst case, all intervals are mutually overlapping, so there will be an edge for every pair of intervals. Therefore, the memory footprint is quadratic in the input size.

#### **Approach 2: Sorting**

**Intuition**

If we sort the intervals by their start value, then each set of intervals that can be merged will appear as a contiguous "run" in the sorted list.

**Algorithm**

First, we sort the list as described. Then, we insert the first interval into our merged list and continue considering each interval in turn as follows: If the current interval begins after the previous interval ends, then they do not overlap and we can append the current interval to merged. Otherwise, they do overlap, and we merge them by updating the end of the previous interval if it is less than the end of the current interval.

A simple proof by contradiction shows that this algorithm always produces the correct answer. First, suppose that the algorithm at some point fails to merge two intervals that should be merged. This would imply that there exists some triple of indices i*i*, j*j*, and k*k* in a list of intervals \text{ints}ints such that i < j < k*i*<*j*<*k* and (\text{ints[i]}ints[i], \text{ints[k]}ints[k]) can be merged, but neither (\text{ints[i]}ints[i], \text{ints[j]}ints[j]) nor (\text{ints[j]}ints[j], \text{ints[k]}ints[k]) can be merged. From this scenario follow several inequalities:

\begin{aligned} \text{ints[i].end} < \text{ints[j].start} \\ \text{ints[j].end} < \text{ints[k].start} \\ \text{ints[i].end} \geq \text{ints[k].start} \\ \end{aligned}ints[i].end<ints[j].startints[j].end<ints[k].startints[i].end≥ints[k].start​

We can chain these inequalities (along with the following inequality, implied by the well-formedness of the intervals: \text{ints[j].start} \leq \text{ints[j].end}ints[j].start≤ints[j].end) to demonstrate a contradiction:

\begin{aligned} \text{ints[i].end} < \text{ints[j].start} \leq \text{ints[j].end} < \text{ints[k].start} \\ \text{ints[i].end} \geq \text{ints[k].start} \end{aligned}ints[i].end<ints[j].start≤ints[j].end<ints[k].startints[i].end≥ints[k].start​

Therefore, all mergeable intervals must occur in a contiguous run of the sorted list.



Consider the example above, where the intervals are sorted, and then all mergeable intervals form contiguous blocks.

|  |
| --- |
| class Solution {  public int[][] merge(int[][] intervals) {  Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));  LinkedList<int[]> merged = new LinkedList<>();  for (int[] interval : intervals) {  // if the list of merged intervals is empty or if the current  // interval does not overlap with the previous, simply append it.  if (merged.isEmpty() || merged.getLast()[1] < interval[0]) {  merged.add(interval);  }  // otherwise, there is overlap, so we merge the current and previous  // intervals.  else {  merged.getLast()[1] = Math.max(merged.getLast()[1], interval[1]);  }  }  return merged.toArray(new int[merged.size()][]);  }  } |

**Complexity Analysis**

* Time complexity : O(n\log{}n)*O*(*n*log*n*)

Other than the sort invocation, we do a simple linear scan of the list, so the runtime is dominated by the O(n\log{}n)*O*(*n*log*n*) complexity of sorting.

* Space complexity : O(\log N)*O*(log*N*) (or O(n)*O*(*n*))

If we can sort intervals in place, we do not need more than constant additional space, although the sorting itself takes O(\log n)*O*(log*n*) space. Otherwise, we must allocate linear space to store a copy of intervals and sort that.

**Insert Interval**

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

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

**Example 1:**

**Input:** intervals = [[1,3],[6,9]], newInterval = [2,5]

**Output:** [[1,5],[6,9]]

**Example 2:**

**Input:** intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]

**Output:** [[1,2],[3,10],[12,16]]

**Explanation:** Because the new interval [4,8] overlaps with [3,5],[6,7],[8,10].

**Example 3:**

**Input:** intervals = [], newInterval = [5,7]

**Output:** [[5,7]]

**Example 4:**

**Input:** intervals = [[1,5]], newInterval = [2,3]

**Output:** [[1,5]]

**Example 5:**

**Input:** intervals = [[1,5]], newInterval = [2,7]

**Output:** [[1,7]]

**Constraints:**

* 0 <= intervals.length <= 104
* intervals[i].length == 2
* 0 <= intervals[i][0] <= intervals[i][1] <= 105
* intervals is sorted by intervals[i][0] in **ascending** order.
* newInterval.length == 2
* 0 <= newInterval[0] <= newInterval[1] <= 105

## Solution

#### **Approach 1: Greedy.**

**Greedy algorithms**

Greedy problems usually look like "Find minimum number of something to do something" or "Find maximum number of something to fit in some conditions", and typically propose an unsorted input.

The idea of greedy algorithm is to pick the locally optimal move at each step, that will lead to the globally optimal solution.

The standard solution has \mathcal{O}(N \log N)O(*N*log*N*) time complexity and consists of two parts:

* Figure out how to sort the input data (\mathcal{O}(N \log N)O(*N*log*N*) time). That could be done directly by a sorting or indirectly by a heap usage. Typically sort is better than the heap usage because of gain in space.
* Parse the sorted input to have a solution (\mathcal{O}(N)O(*N*) time).

Please notice that in case of well-sorted input one doesn't need the first part and the greedy solution could have \mathcal{O}(N)O(*N*) time complexity, [here is an example](https://leetcode.com/articles/gas-station/).

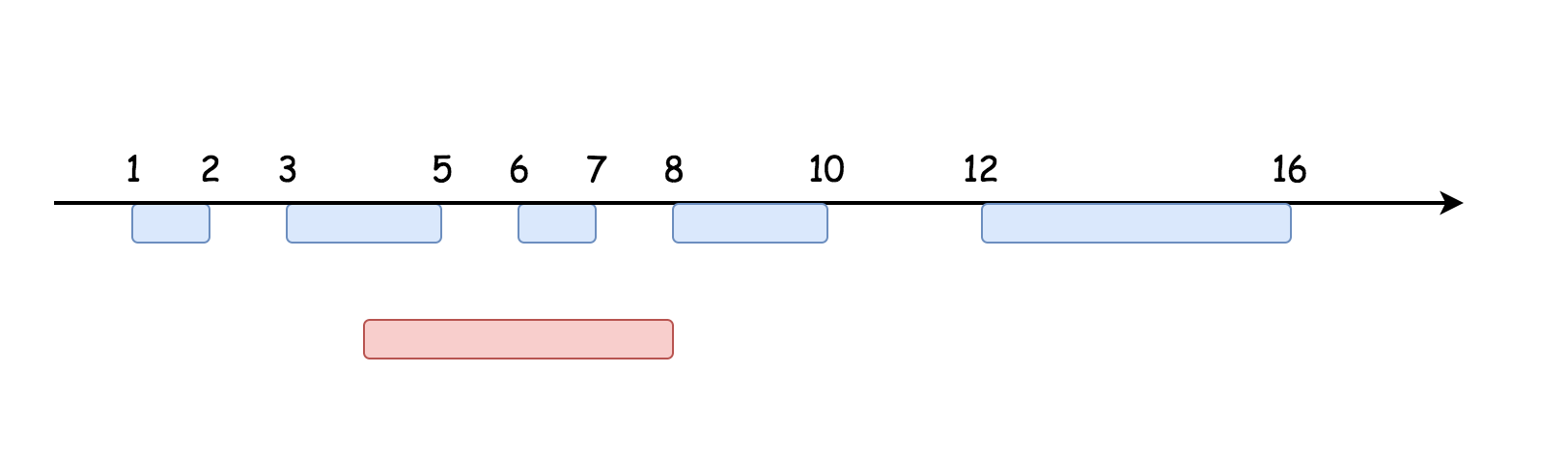
How to prove that your greedy algorithm provides globally optimal solution?

Usually you could use the [proof by contradiction](https://en.wikipedia.org/wiki/Proof_by_contradiction).

**Intuition**

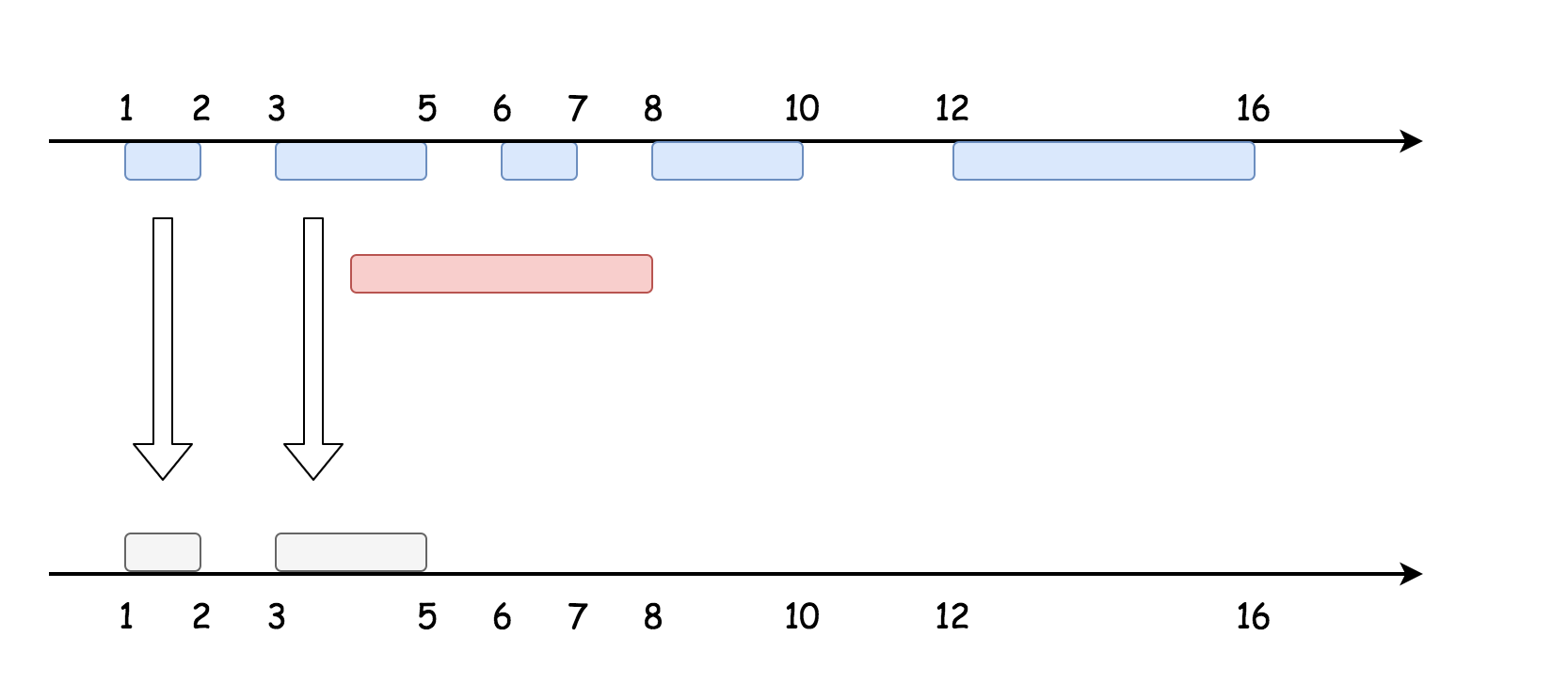
Here we have an example of a greedy problem with a well-sorted input, and hence the algorithm time complexity should be \mathcal{O}(N)O(*N*).

Let's consider the following intervals

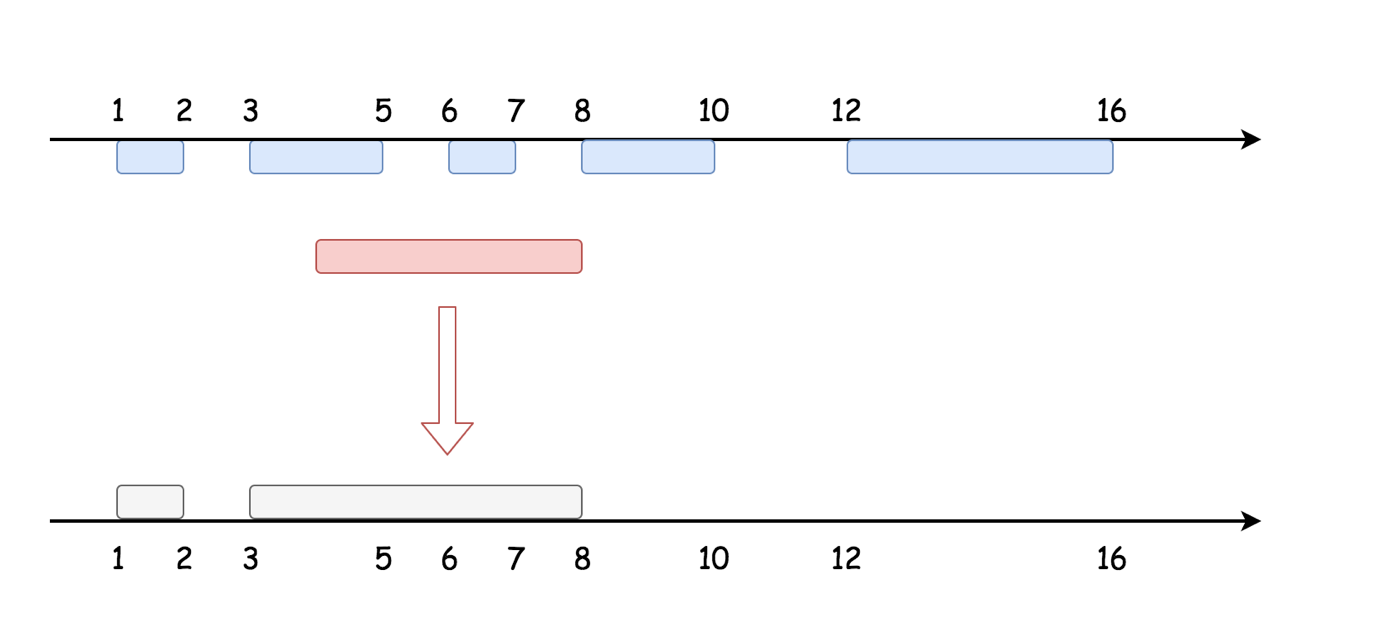


The straightforward one-pass strategy could be implemented in three steps.

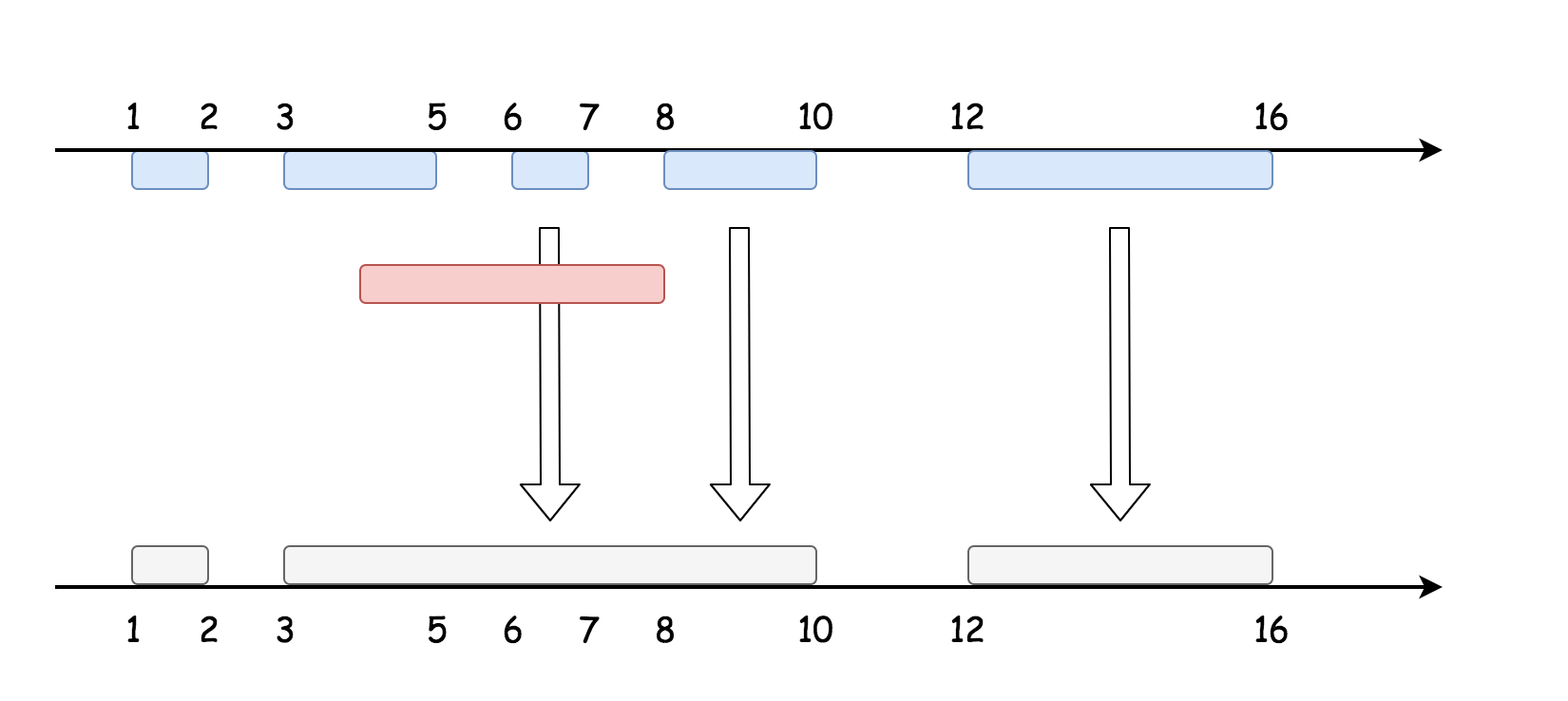
1 . Add to the output all the intervals starting before newInterval.



2 . Add to the output newInterval, merge it with the last added interval if needed.



3 . Add the next intervals one by one, merge if needed.



Basically, the same strategy [as here](https://leetcode.com/articles/merge-intervals/), with an additional care to add the new interval in its proper position in order not to destroy the well-sorted input.

**Algorithm**

Here is the algorithm :

* Add to the output all the intervals starting before newInterval.
* Add to the output newInterval. Merge it with the last added interval if newInterval starts before the last added interval.
* Add the next intervals one by one. Merge with the last added interval if the current interval starts before the last added interval.

**Implementation**

|  |
| --- |
| class Solution {  public int[][] insert(int[][] intervals, int[] newInterval) {  // init data  int newStart = newInterval[0], newEnd = newInterval[1];  int idx = 0, n = intervals.length;  LinkedList<int[]> output = new LinkedList<int[]>();  // add all intervals starting before newInterval  while (idx < n && newStart > intervals[idx][0])  output.add(intervals[idx++]);  // add newInterval  int[] interval = new int[2];  // if there is no overlap, just add the interval  if (output.isEmpty() || output.getLast()[1] < newStart)  output.add(newInterval);  // if there is an overlap, merge with the last interval  else {  interval = output.removeLast();  interval[1] = Math.max(interval[1], newEnd);  output.add(interval);  }  // add next intervals, merge with newInterval if needed  while (idx < n) {  interval = intervals[idx++];  int start = interval[0], end = interval[1];  // if there is no overlap, just add an interval  if (output.getLast()[1] < start) output.add(interval);  // if there is an overlap, merge with the last interval  else {  interval = output.removeLast();  interval[1] = Math.max(interval[1], end);  output.add(interval);  }  }  return output.toArray(new int[output.size()][2]);  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) since it's one pass along the input array.
* Space complexity : \mathcal{O}(N)O(*N*) to keep the output.

**Valid Anagram**

Given two strings s and t , write a function to determine if t is an anagram of s.

**Example 1:**

**Input:** s = "anagram", t = "nagaram"

**Output:** true

**Example 2:**

**Input:** s = "rat", t = "car"

**Output:** false

**Note:**  
You may assume the string contains only lowercase alphabets.

**Follow up:**  
What if the inputs contain unicode characters? How would you adapt your solution to such case?

EZ

**Count of Smaller Numbers After Self**

You are given an integer array nums and you have to return a new counts array. The counts array has the property where counts[i] is the number of smaller elements to the right of nums[i].

**Example 1:**

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

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

**Explanation:**

To the right of 5 there are **2** smaller elements (2 and 1).

To the right of 2 there is only **1** smaller element (1).

To the right of 6 there is **1** smaller element (1).

To the right of 1 there is **0** smaller element.

**Example 2:**

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

**Output:** [0]

**Example 3:**

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

**Output:** [0,0]

**Constraints:**

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

Hard -no answer

**Peak Index in a Mountain Array**

Let's call an array arr a **mountain** if the following properties hold:

* arr.length >= 3
* There exists some i with 0 < i < arr.length - 1 such that:
  + arr[0] < arr[1] < ... arr[i-1] < arr[i]
  + arr[i] > arr[i+1] > ... > arr[arr.length - 1]

Given an integer array arr that is **guaranteed** to be a mountain, return any i such that arr[0] < arr[1] < ... arr[i - 1] < arr[i] > arr[i + 1] > ... > arr[arr.length - 1].

**Example 1:**

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

**Output:** 1

**Example 2:**

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

**Output:** 1

**Example 3:**

**Input:** arr = [0,10,5,2]

**Output:** 1

**Example 4:**

**Input:** arr = [3,4,5,1]

**Output:** 2

**Example 5:**

**Input:** arr = [24,69,100,99,79,78,67,36,26,19]

**Output:** 2

**Constraints:**

* 3 <= arr.length <= 104
* 0 <= arr[i] <= 106
* arr is **guaranteed** to be a mountain array.

**Follow up:** Finding the O(n) is straightforward, could you find an O(log(n)) solution?

#### **Approach 1: Linear Scan**

**Intuition and Algorithm**

The mountain increases until it doesn't. The point at which it stops increasing is the peak.

|  |
| --- |
| class Solution {  public int peakIndexInMountainArray(int[] A) {  int i = 0;  while (A[i] < A[i+1]) i++;  return i;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of A.
* Space Complexity: O(1)*O*(1).

#### **Approach 2: Binary Search**

**Intuition and Algorithm**

The comparison A[i] < A[i+1] in a mountain array looks like [True, True, True, ..., True, False, False, ..., False]: 1 or more boolean Trues, followed by 1 or more boolean False. For example, in the mountain array [1, 2, 3, 4, 1], the comparisons A[i] < A[i+1] would be True, True, True, False.

We can binary search over this array of comparisons, to find the largest index i such that A[i] < A[i+1]. For more on binary search, see the [LeetCode explore topic here.](https://leetcode.com/explore/learn/card/binary-search/)

|  |
| --- |
| class Solution {  public int peakIndexInMountainArray(int[] A) {  int lo = 0, hi = A.length - 1;  while (lo < hi) {  int mi = lo + (hi - lo) / 2;  if (A[mi] < A[mi + 1])  lo = mi + 1;  else  hi = mi;  }  return lo;  }  } |

**Complexity Analysis**

* Time Complexity: O(\log N)*O*(log*N*), where N*N* is the length of A.
* Space Complexity: O(1)*O*(1).

## Dynamic Programming

It can be tricky to identify the subproblems and connect them, which is essential in solving Dynamic Programming problems. Dynamic programming is not that scary as you might think, and you can improve your dynamic programming skills by practicing a lot of these problems.

According to our user survey, one of the most frequently asked Dynamic Programming Google interview questions is [Split Array Largest Sum](https://leetcode.com/explore/interview/card/google/64/dynamic-programming-4/3089/).

**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

**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**

**Best Time to Buy and Sell Stock**

Ez

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

**Maximum Product Subarray**

Given an integer array nums, find the contiguous subarray within an array (containing at least one number) which has the largest product.

**Example 1:**

**Input:** [2,3,-2,4]

**Output:** 6

**Explanation:** [2,3] has the largest product 6.

**Example 2:**

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

**Output:** 0

**Explanation:** The result cannot be 2, because [-2,-1] is not a subarray.

Hard

**Coin Change**

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

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

**Example 1:**

**Input:** coins = [1,2,5], amount = 11

**Output:** 3

**Explanation:** 11 = 5 + 5 + 1

**Example 2:**

**Input:** coins = [2], amount = 3

**Output:** -1

**Example 3:**

**Input:** coins = [1], amount = 0

**Output:** 0

**Example 4:**

**Input:** coins = [1], amount = 1

**Output:** 1

**Example 5:**

**Input:** coins = [1], amount = 2

**Output:** 2

**Constraints:**

Medium

**Split Array Largest Sum**

Given an array nums which consists of non-negative integers and an integer m, you can split the array into m non-empty continuous subarrays.

Write an algorithm to minimize the largest sum among these m subarrays.

**Example 1:**

**Input:** nums = [7,2,5,10,8], m = 2

**Output:** 18

**Explanation:**

There are four ways to split nums into two subarrays.

The best way is to split it into [7,2,5] and [10,8],

where the largest sum among the two subarrays is only 18.

**Example 2:**

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

**Output:** 9

**Example 3:**

**Input:** nums = [1,4,4], m = 3

**Output:** 4

**Constraints:**

* 1 <= nums.length <= 1000
* 0 <= nums[i] <= 106
* 1 <= m <= min(50, nums.length)

#### **Approach #1 Brute Force [Time Limit Exceeded]**

**Intuition**

Check all possible splitting plans to find the minimum largest value for subarrays.

**Algorithm**

We can use depth-first search to generate all possible splitting plan. For each element in the array, we can choose to append it to the previous subarray or start a new subarray starting with that element (if the number of subarrays does not exceed m). The sum of the current subarray can be updated at the same time.

|  |
| --- |
| class Solution {  private int ans;  private int n, m;  private void dfs(int[] nums, int i, int cntSubarrays, int curSum, int curMax) {  if (i == n && cntSubarrays == m) {  ans = Math.min(ans, curMax);  return;  }  if (i == n) {  return;  }  if (i > 0) {  dfs(nums, i + 1, cntSubarrays, curSum + nums[i], Math.max(curMax, curSum + nums[i]));  }  if (cntSubarrays < m) {  dfs(nums, i + 1, cntSubarrays + 1, nums[i], Math.max(curMax, nums[i]));  }  }  public int splitArray(int[] nums, int M) {  ans = Integer.MAX\_VALUE;  n = nums.length;  m = M;  dfs(nums, 0, 0, 0, 0);  return ans;  }  } |

**Complexity Analysis**

* Time complexity : O(n^m)*O*(*nm*). To split n elements into m parts, we can have \binom{n - 1}{m - 1}(*m*−1*n*−1​) different solutions. This is equivalent to n ^ m*nm*.
* Space complexity : O(n)*O*(*n*). The dfs helper function adds a frame to the run-time stack for every element in the list, of which there are n*n*.

#### **Approach #2 Dynamic Programming [Accepted]**

**Intuition**

The problem satisfies the non-aftereffect property. We can try to use dynamic programming to solve it.

The non-aftereffect property means, once the state of a certain stage is determined, it is not affected by the state in the future. In this problem, if we get the largest subarray sum for splitting nums[0..i] into j parts, this value will not be affected by how we split the remaining part of nums.

To know more about non-aftereffect property, this link may be helpful : <http://www.programering.com/a/MDOzUzMwATM.html>

**Algorithm**

Let's define f[i][j] to be the minimum largest subarray sum for splitting nums[0..i] into j parts.

Consider the jth subarray. We can split the array from a smaller index k to i to form it. Thus f[i][j] can be derived from max(f[k][j - 1], nums[k + 1] + ... + nums[i]). For all valid index k, f[i][j] should choose the minimum value of the above formula.

The final answer should be f[n][m], where n is the size of the array.

For corner situations, all the invalid f[i][j] should be assigned with INFINITY, and f[0][0] should be initialized with 0.

|  |
| --- |
| class Solution {  public int splitArray(int[] nums, int m) {  int n = nums.length;  int[][] f = new int[n + 1][m + 1];  int[] sub = new int[n + 1];  for (int i = 0; i <= n; i++) {  for (int j = 0; j <= m; j++) {  f[i][j] = Integer.MAX\_VALUE;  }  }  for (int i = 0; i < n; i++) {  sub[i + 1] = sub[i] + nums[i];  }  f[0][0] = 0;  for (int i = 1; i <= n; i++) {  for (int j = 1; j <= m; j++) {  for (int k = 0; k < i; k++) {  f[i][j] = Math.min(f[i][j], Math.max(f[k][j - 1], sub[i] - sub[k]));  }  }  }  return f[n][m];  }  } |

**Complexity Analysis**

* Time complexity : O(n^2 \* m)*O*(*n*2∗*m*). The total number of states is O(n \* m)*O*(*n*∗*m*). To compute each state f[i][j], we need to go through the whole array to find the optimum k. This requires another O(n)*O*(*n*) loop. So the total time complexity is O(n ^ 2 \* m)*O*(*n*2∗*m*).
* Space complexity : O(n \* m)*O*(*n*∗*m*). The space complexity is equivalent to the number of states, which is O(n \* m)*O*(*n*∗*m*).

#### **Approach #3 Binary Search + Greedy [Accepted]**

**Intuition**

We can easily find a property for the answer:

If we can find a splitting method that ensures the maximum largest subarray sum will not exceed a value x, then we can also find a splitting method that ensures the maximum largest subarray sum will not exceed any value y that is greater than x.

Lets define this property as F(x) for the value x. F(x) is true means we can find a splitting method that ensures the maximum largest subarray sum will not exceed x.

From the discussion above, we can find out that for x ranging from -INFINITY to INFINITY, F(x) will start with false, then from a specific value x0, F(x) will turn to true and stay true forever.

Obviously, the specific value x0 is our answer.

**Algorithm**

We can use Binary search to find the value x0. Keeping a value mid = (left + right) / 2. If F(mid) is false, then we will search the range [mid + 1, right]; If F(mid) is true, then we will search [left, mid - 1].

For a given x, we can get the answer of F(x) using a greedy approach. Using an accumulator sum to store the sum of the current processing subarray and a counter cnt to count the number of existing subarrays. We will process the elements in the array one by one. For each element num, if sum + num <= x, it means we can add num to the current subarray without exceeding the limit. Otherwise, we need to make a cut here, start a new subarray with the current element num. This leads to an increment in the number of subarrays.

After we have finished the whole process, we need to compare the value cnt to the size limit of subarrays m. If cnt <= m, it means we can find a splitting method that ensures the maximum largest subarray sum will not exceed x. Otherwise, F(x) should be false.

|  |
| --- |
| class Solution {  public int splitArray(int[] nums, int m) {  long l = 0;  long r = 0;  int n = nums.length;  for (int i = 0; i < n; i++) {  r += nums[i];  if (l < nums[i]) {  l = nums[i];  }  }  long ans = r;  while (l <= r) {  long mid = (l + r) >> 1;  long sum = 0;  int cnt = 1;  for (int i = 0; i < n; i++) {  if (sum + nums[i] > mid) {  cnt ++;  sum = nums[i];  } else {  sum += nums[i];  }  }  if (cnt <= m) {  ans = Math.min(ans, mid);  r = mid - 1;  } else {  l = mid + 1;  }  }  return (int)ans;  }  } |

\*\*Complexity Analysis\*\*

* Time complexity : O(n \* log(sum of array))*O*(*n*∗*log*(*sumofarray*)). The binary search costs O(log(sum of array))*O*(*log*(*sumofarray*)), where sum of array is the sum of elements in nums. For each computation of F(x), the time complexity is O(n)*O*(*n*) since we only need to go through the whole array.
* Space complexity : O(1)*O*(1) space complexity without taking the output list into account, and O(n)*O*(*n*) to store the output list.

## Design

Google loves to ask lots of question variations based on the [Iterator pattern](https://en.wikipedia.org/wiki/Iterator_pattern), so make sure you are familiar with the concept of iterators and how iterators work in principle. A good way to learn is to read the open source code and try to code it yourself. For example, here is [Google's guava implementation of Peeking Iterator](https://github.com/google/guava/blob/6f22af40e1526b8c194e9e36d457bcd37680c6a3/guava/src/com/google/common/collect/Iterators.java#L1121).

**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

Hard

**Min Stack**

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

* push(x) -- Push element x onto stack.
* pop() -- Removes the element on top of the stack.
* top() -- Get the top element.
* getMin() -- Retrieve the minimum element in the stack.

**Example 1:**

**Input**

["MinStack","push","push","push","getMin","pop","top","getMin"]

[[],[-2],[0],[-3],[],[],[],[]]

**Output**

[null,null,null,null,-3,null,0,-2]

**Explanation**

MinStack minStack = new MinStack();

minStack.push(-2);

minStack.push(0);

minStack.push(-3);

minStack.getMin(); // return -3

minStack.pop();

minStack.top(); // return 0

minStack.getMin(); // return -2

**Constraints:**

* Methods pop, top and getMin operations will always be called on **non-empty** stacks.

Queue Stack

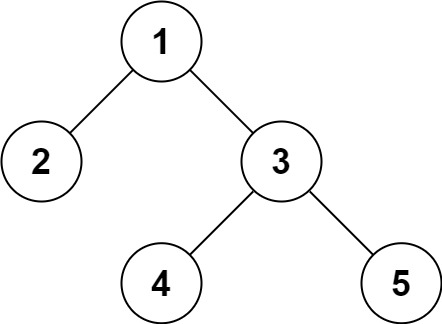
**Serialize and Deserialize Binary Tree**

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]

Binary Tree

**Logger Rate Limiter**

Design a logger system that receives a stream of messages along with their timestamps. Each **unique** message should only be printed **at most every 10 seconds** (i.e. a message printed at timestamp t will prevent other identical messages from being printed until timestamp t + 10).

All messages will come in chronological order. Several messages may arrive at the same timestamp.

Implement the Logger class:

* Logger() Initializes the logger object.
* bool shouldPrintMessage(int timestamp, string message) Returns true if the message should be printed in the given timestamp, otherwise returns false.

**Example 1:**

**Input**

["Logger", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage"]

[[], [1, "foo"], [2, "bar"], [3, "foo"], [8, "bar"], [10, "foo"], [11, "foo"]]

**Output**

[null, true, true, false, false, false, true]

**Explanation**

Logger logger = new Logger();

logger.shouldPrintMessage(1, "foo"); // return true, next allowed timestamp for "foo" is 1 + 10 = 11

logger.shouldPrintMessage(2, "bar"); // return true, next allowed timestamp for "bar" is 2 + 10 = 12

logger.shouldPrintMessage(3, "foo"); // 3 < 11, return false

logger.shouldPrintMessage(8, "bar"); // 8 < 12, return false

logger.shouldPrintMessage(10, "foo"); // 10 < 11, return false

logger.shouldPrintMessage(11, "foo"); // 11 >= 11, return true, next allowed timestamp for "foo" is

// 11 + 10 = 21

 HashTable

**Insert Delete GetRandom O(1)**

Implement the RandomizedSet class:

* bool insert(int val) Inserts an item val into the set if not present. Returns true if the item was not present, false otherwise.
* bool remove(int val) Removes an item val from the set if present. Returns true if the item was present, false otherwise.
* int getRandom() Returns a random element from the current set of elements (it's guaranteed that at least one element exists when this method is called). Each element must have the **same probability** of being returned.

**Follow up:** Could you implement the functions of the class with each function works in **average** O(1) time?

**Example 1:**

**Input**

["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert", "getRandom"]

[[], [1], [2], [2], [], [1], [2], []]

**Output**

[null, true, false, true, 2, true, false, 2]

**Explanation**

RandomizedSet randomizedSet = new RandomizedSet();

randomizedSet.insert(1); // Inserts 1 to the set. Returns true as 1 was inserted successfully.

randomizedSet.remove(2); // Returns false as 2 does not exist in the set.

randomizedSet.insert(2); // Inserts 2 to the set, returns true. Set now contains [1,2].

randomizedSet.getRandom(); // getRandom() should return either 1 or 2 randomly.

randomizedSet.remove(1); // Removes 1 from the set, returns true. Set now contains [2].

randomizedSet.insert(2); // 2 was already in the set, so return false.

randomizedSet.getRandom(); // Since 2 is the only number in the set, getRandom() will always return 2.

 HashTable

**Design Search Autocomplete System**

Design a search autocomplete system for a search engine. Users may input a sentence (at least one word and end with a special character '#'). For **each character** they type **except '#'**, you need to return the **top 3** historical hot sentences that have prefix the same as the part of sentence already typed. Here are the specific rules:

1. The hot degree for a sentence is defined as the number of times a user typed the exactly same sentence before.
2. The returned top 3 hot sentences should be sorted by hot degree (The first is the hottest one). If several sentences have the same degree of hot, you need to use ASCII-code order (smaller one appears first).
3. If less than 3 hot sentences exist, then just return as many as you can.
4. When the input is a special character, it means the sentence ends, and in this case, you need to return an empty list.

Your job is to implement the following functions:

The constructor function:

AutocompleteSystem(String[] sentences, int[] times): This is the constructor. The input is **historical data**. Sentences is a string array consists of previously typed sentences. Times is the corresponding times a sentence has been typed. Your system should record these historical data.

Now, the user wants to input a new sentence. The following function will provide the next character the user types:

List<String> input(char c): The input c is the next character typed by the user. The character will only be lower-case letters ('a' to 'z'), blank space (' ') or a special character ('#'). Also, the previously typed sentence should be recorded in your system. The output will be the **top 3** historical hot sentences that have prefix the same as the part of sentence already typed.

**Example:**  
**Operation:** AutocompleteSystem(["i love you", "island","ironman", "i love leetcode"], [5,3,2,2])  
The system have already tracked down the following sentences and their corresponding times:  
"i love you" : 5 times  
"island" : 3 times  
"ironman" : 2 times  
"i love leetcode" : 2 times  
Now, the user begins another search:  
  
**Operation:** input('i')  
**Output:** ["i love you", "island","i love leetcode"]  
**Explanation:**  
There are four sentences that have prefix "i". Among them, "ironman" and "i love leetcode" have same hot degree. Since ' ' has ASCII code 32 and 'r' has ASCII code 114, "i love leetcode" should be in front of "ironman". Also we only need to output top 3 hot sentences, so "ironman" will be ignored.  
  
**Operation:** input(' ')  
**Output:** ["i love you","i love leetcode"]  
**Explanation:**  
There are only two sentences that have prefix "i ".  
  
**Operation:** input('a')  
**Output:** []  
**Explanation:**  
There are no sentences that have prefix "i a".  
  
**Operation:** input('#')  
**Output:** []  
**Explanation:**  
The user finished the input, the sentence "i a" should be saved as a historical sentence in system. And the following input will be counted as a new search.

**Note:**

1. The input sentence will always start with a letter and end with '#', and only one blank space will exist between two words.
2. The number of **complete sentences** that to be searched won't exceed 100. The length of each sentence including those in the historical data won't exceed 100.
3. Please use double-quote instead of single-quote when you write test cases even for a character input.
4. Please remember to **RESET** your class variables declared in class AutocompleteSystem, as static/class variables are **persisted across multiple test cases**. Please see [here](https://leetcode.com/faq/#different-output) for more details.

Tries

## Others

Here are other type of problems you may encounter in a Google interview, such as Bit Manipulation.

**Reverse Integer**

Given a signed 32-bit integer x, return x with its digits reversed. If reversing x causes the value to go outside the signed 32-bit integer range [-231, 231 - 1], then return 0.

**Assume the environment does not allow you to store 64-bit integers (signed or unsigned).**

**Example 1:**

**Input:** x = 123

**Output:** 321

**Example 2:**

**Input:** x = -123

**Output:** -321

**Example 3:**

**Input:** x = 120

**Output:** 21

**Example 4:**

**Input:** x = 0

**Output:** 0

**Constraints:**

* -231 <= x <= 231 - 1

EZ

**Candy**

There are n children standing in a line. Each child is assigned a rating value given in the integer array ratings.

You are giving candies to these children subjected to the following requirements:

* Each child must have at least one candy.
* Children with a higher rating get more candies than their neighbors.

Return the minimum number of candies you need to have to distribute the candies to the children.

**Example 1:**

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

**Output:** 5

**Explanation:** You can allocate to the first, second and third child with 2, 1, 2 candies respectively.

**Example 2:**

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

**Output:** 4

**Explanation:** You can allocate to the first, second and third child with 1, 2, 1 candies respectively.

The third child gets 1 candy because it satisfies the above two conditions.

**Constraints:**

* n == ratings.length
* 1 <= n <= 2 \* 104
* 1 <= ratings[i] <= 2 \* 104

Solution

Approach 1: Brute Force

The simplest approach makes use of a 1d array, candies to keep a track of the candies given to the students. Firstly, we give 1 candy to each student. Then, we start scanning the array from left-to-right. At every element encountered, firstly, if the current element's rating, ratings[i], is larger than the previous element ratings[i - 1] and additionally candies[i] <= candies[i - 1], then we update candies[i] as candies[i] = candies[i - 1] + 1. Thus, now the candy distribution for these two elements candies[i - 1] and candies[i] becomes correct for the time being (locally). In the same step, we also check if the current element's ratings, ratings[i], is larger than the next element's ratings, i.e. ratings[i] > ratings[i + 1]. If so, we again update candies[i] = candies[i + 1] + 1. We continue this process for the whole ratings array. If in any traversal, no update of the candies array occurs, it means that the array now contains the final distribution of candies, and so we should stop. We use boolean hasChanged to keep track of whether or not we made any changes in the last traversal.

At the end, we can sum up all the elements of the candies array to obtain the required minimum number of candies.

|  |
| --- |
| public class Solution {  public int candy(int[] ratings) {  int[] candies = new int[ratings.length];  Arrays.fill(candies, 1);  boolean hasChanged = true;  while (hasChanged) {  hasChanged = false;  for (int i = 0; i < ratings.length; i++) {  if (i != ratings.length - 1 && ratings[i] > ratings[i + 1] && candies[i] <= candies[i + 1]) {  candies[i] = candies[i + 1] + 1;  hasChanged = true;  }  if (i > 0 && ratings[i] > ratings[i - 1] && candies[i] <= candies[i - 1]) {  candies[i] = candies[i - 1] + 1;  hasChanged = true;  }  }  }  int sum = 0;  for (int candy : candies) {  sum += candy;  }  return sum;  }  } |

**Complexity Analysis**

* Time complexity : O(n^2)*O*(*n*2). We need to traverse the array at most n*n* times. This is because at most, a child will get n*n* candies, and their candy count will be update once on each traversal.
* Space complexity : O(n)*O*(*n*). One array, candies array of size n*n* is used.

#### **Approach 2: Using two arrays**

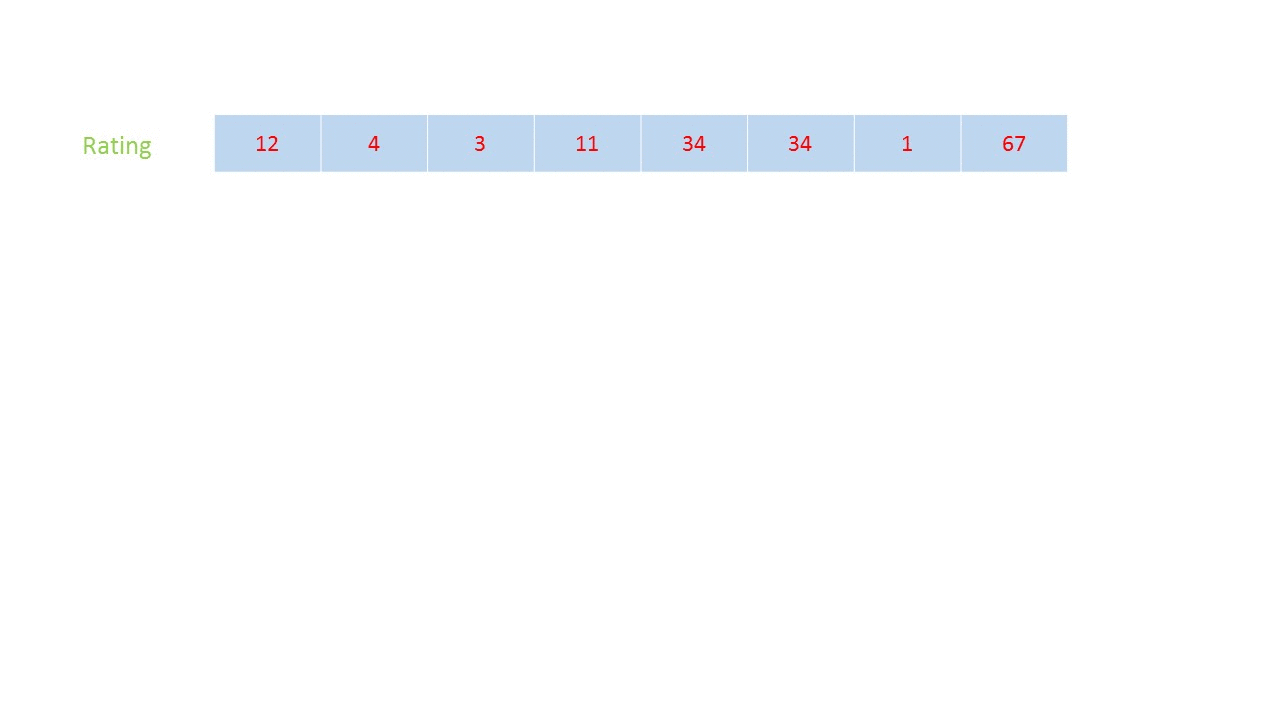
**Algorithm**

In this approach, we make use of two 1-d arrays left2right and right2left. The left2right array is used to store the number of candies required by the current student taking care of the distribution relative to the left neighbors only. i.e. Assuming the distribution rule is: The student with a higher rating than their left neighbor should always get more candies than its left neighbor. Similarly, the right2left array is used to store the number of candies required by the current student taking care of the distribution relative to the right neighbors only. i.e. Assuming the distribution rule to be: The student with a higher rating than their right neighbor should always get more candies than their right neighbor. To do so, firstly we assign 1 candy to each student in both the left2right and right2left arrays. Then, we traverse the array from left-to-right and whenever the current element's ratings is larger than the left neighbor we update the current element's candies in the left2right array as left2right[i] = left2right[i-1] + 1, since the current element's candies are always less than or equal candies than its left neighbor before updating. After the forward traversal, we traverse the array from left-to-right and update right2left[i] as right2left[i] = right2left[i + 1] + 1, whenever the current i'th element has a higher ratings than the right i+1'th element.

Now, for the i'th student in the array, we need to give max(left2right[i], right2left[i]) to them, in order to satisfy both the left and the right neighbor relationship. Thus, at the end, we obtain the minimum number of candies required as:

\text{minimum\\_candies}=\sum\_{i=0}^{n-1} \text{max}(\text{left2right}[i], \text{right2left}[i]) \\ \text{where } n = \text{length of the ratings array.}minimum\_candies=∑*i*=0*n*−1​max(left2right[*i*],right2left[*i*])where *n*=length of the ratings array.

The following animation illustrates the method:



|  |
| --- |
| public class Solution {  public int candy(int[] ratings) {  int sum = 0;  int[] left2right = new int[ratings.length];  int[] right2left = new int[ratings.length];  Arrays.fill(left2right, 1);  Arrays.fill(right2left, 1);  for (int i = 1; i < ratings.length; i++) {  if (ratings[i] > ratings[i - 1]) {  left2right[i] = left2right[i - 1] + 1;  }  }  for (int i = ratings.length - 2; i >= 0; i--) {  if (ratings[i] > ratings[i + 1]) {  right2left[i] = right2left[i + 1] + 1;  }  }  for (int i = 0; i < ratings.length; i++) {  sum += Math.max(left2right[i], right2left[i]);  }  return sum;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). left2right and right2left arrays are traversed thrice.
* Space complexity : O(n)*O*(*n*). Two arrays left2right and right2left of size n*n* are used.

#### **Approach 3: Using one array**

**Algorithm**

In the previous approach, we used two arrays to keep track of the left neighbor and the right neighbor relation individually and later on combined these two. Instead of this, we can make use of a single array candies to keep the count of the number of candies to be allocated to the current student. In order to do so, firstly we assign 1 candy to each student. Then, we traverse the array from left-to-right and distribute the candies following only the left neighbor relation i.e. whenever the current element's ratings is larger than the left neighbor and has less than or equal candies than its left neighbor, we update the current element's candies in the candies array as candies[i] = candies[i-1] + 1. While updating we need not compare candies[i] and candies[i - 1], since candies[i] <= candies[i - 1] before updating. After this, we traverse the array from right-to-left. Now, we need to update the i'th element's candies in order to satisfy both the left neighbor and the right neighbor relation. Now, during the backward traversal, if ratings[i] > ratings[i + 1], considering only the right neighbor criteria, we could've updated candies[i] as candies[i] = candies[i + 1] + 1. But, this time we need to update candies[i] only if candies[i] <= candies[i + 1]. This happens because this time we've already altered the candies array during the forward traversal and thus candies[i] isn't necessarily less than or equal to candies[i + 1]. Thus, if ratings[i] > ratings[i + 1], we can update candies[i] as candies[i] = max(candies[i], candies[i + 1] + 1), which makes candies[i] satisfy both the left neighbor and the right neighbor criteria.

Again, we need to sum up all the elements of the candies array to obtain the required result.

\text{minimum\\_candies} = \sum\_{i=0}^{n-1} candies[i], \\ \text{where } n = \text{length of the ratings array.}minimum\_candies=∑*i*=0*n*−1​*candies*[*i*],where *n*=length of the ratings array.

|  |
| --- |
| public class Solution {  public int candy(int[] ratings) {  int[] candies = new int[ratings.length];  Arrays.fill(candies, 1);  for (int i = 1; i < ratings.length; i++) {  if (ratings[i] > ratings[i - 1]) {  candies[i] = candies[i - 1] + 1;  }  }  int sum = candies[ratings.length - 1];  for (int i = ratings.length - 2; i >= 0; i--) {  if (ratings[i] > ratings[i + 1]) {  candies[i] = Math.max(candies[i], candies[i + 1] + 1);  }  sum += candies[i];  }  return sum;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). The array candies of size n*n* is traversed thrice.
* Space complexity : O(n)*O*(*n*). An array candies of size n*n* is used.

#### **Approach 4: Single Pass Approach with Constant Space**

**Algorithm**

This approach relies on the observation (as demonstrated in the figure below as well) that in order to distribute the candies as per the given criteria using the minimum number of candies, the candies are always distributed in terms of increments of 1. Further, while distributing the candies, the local minimum number of candies given to a student is 1. Thus, the sub-distributions always take the following form: \text{1, 2, 3, ..., n}1, 2, 3, ..., n or \text{n,..., 2, 1}n,..., 2, 1. Which, can simply be added using the formula n(n+1)/2*n*(*n*+1)/2.

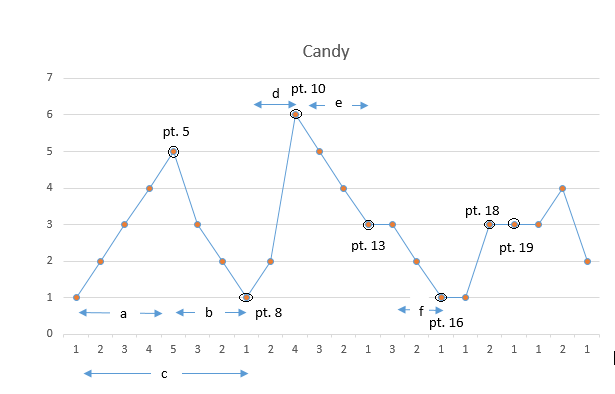
Now, we can view the given rankings as some rising and falling slopes. Whenever the slope is rising, the distribution takes the form: \text{1, 2, 3, ..., m}1, 2, 3, ..., m. Similarly, a falling slope takes the form: \text{k,..., 2, 1}k,..., 2, 1. A challenge that arises now is that the local peak point can be included in only one of the slopes. Should we include the local peak point, n, in the rising slope or the in falling slope?

In order to decide, we can observe that in order to satisfy both the right neighbor and the left neighbor criteria, the peak point's count needs to be the max. of the counts determined by the rising and the falling slopes. Thus, in order to determine the number of candies required, the peak point needs to be included in the slope which contains more number of points. The local valley point can also be included in only one of the slopes, but this issue can be resolved easily, since the local valley point will always be assigned a candy count of 1 (which can be subtracted from the next slope's count calculations).

Coming to the implementation, we maintain two variables oldSlope and newSlope to determine the occurrence of a peak or a valley. We also use up and down variables to keep a track of the count of elements on the rising slope and on the falling slope respectively (without including the peak element). We always update the total count of candies at the end of a falling slope following a rising slope (or a mountain). The leveling of the points in rankings also works as the end of a mountain. At the end of the mountain, we determine whether to include the peak point in the rising slope or in the falling slope by comparing the up and down variables up to that point. Thus, the count assigned to the peak element becomes: max(up, down) + 1. At this point, we can reset the up and down variables indicating the start of a new mountain.

The following figure shows the cases that need to be handled for this example:

rankings: [1 2 3 4 5 3 2 1 2 6 5 4 3 3 2 1 1 3 3 3 4 2]



From this figure, we can see that the candy distributions in the sub-regions always take the form \text{1, 2, ...n}1, 2, ...n or \text{n, ..., 2, 1}n, ..., 2, 1. For the first mountain comprised by the regions a and b, while assigning candies to the local peak point (point 5), it needs to be included in a to satisfy the left neighbor criteria. The local valley point at the end of region b (point 8) marks the end of the first mountain (region c). While performing the calculations, we can include this point in either the current or the following mountain. Then, point 13 marks the end of the second mountain due to leveling of the point 13 and point 14. Since region e has more points than region d, the local peak (point 10) needs to be included in region e to satisfy the right neighbor criteria. Now, the third mountain f can be considered as a mountain with no rising slope (as up = 0) but only a falling slope. Similarly, points 16, 18, and 19 also act as the mountain ends due to the leveling of the points.

|  |
| --- |
| public class Solution {  public int count(int n) {  return (n \* (n + 1)) / 2;  }  public int candy(int[] ratings) {  if (ratings.length <= 1) {  return ratings.length;  }  int candies = 0;  int up = 0;  int down = 0;  int oldSlope = 0;  for (int i = 1; i < ratings.length; i++) {  int newSlope = (ratings[i] > ratings[i - 1]) ? 1  : (ratings[i] < ratings[i - 1] ? -1  : 0);  if ((oldSlope > 0 && newSlope == 0) || (oldSlope < 0 && newSlope >= 0)) {  candies += count(up) + count(down) + Math.max(up, down);  up = 0;  down = 0;  }  if (newSlope > 0) {  up++;  } else if (newSlope < 0) {  down++;  } else {  candies++;  }  oldSlope = newSlope;  }  candies += count(up) + count(down) + Math.max(up, down) + 1;  return candies;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). We traverse the rankings array once only.
* Space complexity : O(1)*O*(1). Constant Extra Space is used.

**Isomorphic Strings**

Given two strings s and t, *determine if they are isomorphic*.

Two strings s and t are isomorphic if the characters in s can be replaced to get t.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

**Example 1:**

**Input:** s = "egg", t = "add"

**Output:** true

**Example 2:**

**Input:** s = "foo", t = "bar"

**Output:** false

**Example 3:**

**Input:** s = "paper", t = "title"

**Output:** true

**Constraints:**

* 1 <= s.length <= 5 \* 104
* t.length == s.length
* s and t consist of any valid ascii character.

Sort then compare

**Strobogrammatic Number**

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to determine if a number is strobogrammatic. The number is represented as a string.

**Example 1:**

**Input:** num = "69"

**Output:** true

**Example 2:**

**Input:** num = "88"

**Output:** true

**Example 3:**

**Input:** num = "962"

**Output:** false

**Example 4:**

**Input:** num = "1"

**Output:** true

## Solution

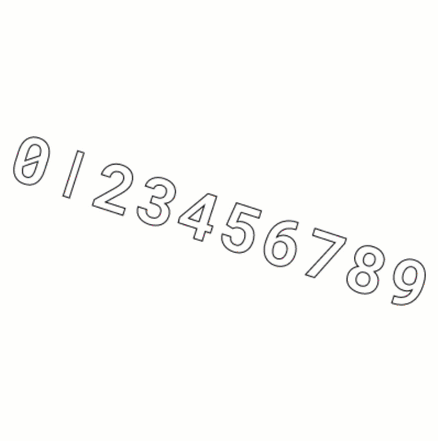
#### **Overview**

Given a number represented as a string, we need to return true if it is strobogrammatic and false if it is not. Determining whether or not a number is strobogrammatic requires looking at whether or not it stays the same when rotated by 180 degrees.

We aren't told very much about which digits count as strobogrammatic. While some people have found it frustrating to not be told everything, this level of uncertainty is intentionally typical of real interviews. A big part of solving this problem is clarifying the requirements, and there is a great way of doing that here on LeetCode, which we'll talk about.

**Rotating a number by 180 degrees**

A good first step is to think carefully about what it means to rotate a number by 180 degrees. Some people find it easy to rotate numbers in their head, whereas others struggle to do so. If you're in the latter group, write some numbers down on a scrap of paper and rotate the paper.



Observe that rotating a number by 180 degrees reverses the order of the digits and rotates each digit upside-down in its new position. It isn't guaranteed that the rotation will form a valid number, though; some digits will become invalid when rotated in this way.

**Investigating the rotation of each digit**

The next step is to determine what each digit becomes when rotated by 180 degrees. There are three possibilities for each digit:

1. it becomes invalid
2. it stays the same
3. it becomes a different digit

We'll consider a *digit* to be **rotatable** if, and only if, that digit becomes a valid digit when rotated. For example, 9 becomes 6, and 8 remains as 8. On the other hand, 4 becomes ߈, which is clearly no longer valid. So, we can say that 6, 9, and 8 are rotatable, whereas 4 is *not*. Numbers such as 1 might be rotatable, but it is font dependent.

So, which digits are rotatable? Your interviewer is unlikely to just give you the answer; you'll probably be expected to figure this out by asking good questions and stating your assumptions and conclusions. The main challenge is that the rotatable-ness of a number is somewhat dependent on the font being used.

Remember, somewhat vague requirements are standard in interviews, and the first step you should take in any interview is to analyze the question, ensure you understood it, clarify the requirements, and state your assumptions. The interviewer *expects* you to ask questions; they don’t want you to *guess* the requirements!

A good idea would be to quickly go through each of the ten digits in order, giving an initial thought on whether or not the digit is rotatable (it's okay if your initial thoughts are different from mine; we're not committed to this yet).

* 0 is probably considered rotatable. Most people draw it as a circle without a "decoration" in the middle.
* 1 may be considered rotatable. On one hand, it's commonly drawn as simply a vertical bar, but on the other, sometimes decorations are added, and most fonts put it off-center.
* 2 may be considered rotatable. It looks a lot like itself when rotated, although there is a distinctive difference we need to be wary of (see the pictures above).
* 3 is *not* rotatable.
* 4 is *not* rotatable.
* 5 may be considered rotatable, for the same reason 2 might be.
* 6 is rotatable; we were shown that it becomes 9.
* 7 is *not* rotatable.
* 8 is probably considered rotatable. Most people draw the two "circles" the same size.
* 9 is rotatable; we were shown it becomes 6.

The next step is dependent on how your interviewer responds to your initial thoughts. They might give you a more precise definition, ask you to make your own judgment call, or they might even ask you to come up with some specific test cases whose answers would clarify the definition. We're going to go with this latter possibility, as it's what you should do on LeetCode.

Before you read any further, have a think about which test cases you could use.

The general rule of thumb for interviews is that you should *state* your assumptions (and be ready to justify them if asked) and *ask* clarifying questions for anything you’re unsure about. Keep in mind that your interviewer is more likely to give you useful answers to specific questions, as opposed to general questions. For example, asking them to list all 1 and 2 digit strobogrammatic numbers might seem like it would be very helpful to you, but they probably won’t be keen on doing your work for you. Instead, you should, for example, ask whether or not 1 and 22 are strobogrammatic numbers. You might need to briefly explain why the cases are unclear (e.g., font).

Here are the simplest test cases you could use. We're going to assume that 3, 4, and 7 are *not* rotatable.

* 0: Will return true if 0 is rotatable.
* 1: Will return true if 1 is rotatable.
* 8: Will return true if 8 is rotatable.
* 2: Will return true if 2 is rotatable.
* 5: Will return true if 5 is rotatable.
* 69: Will return true if 6 and 9 are rotatable and become each other (we already know this from the example given in the problem, but you might not be given it in an interview).

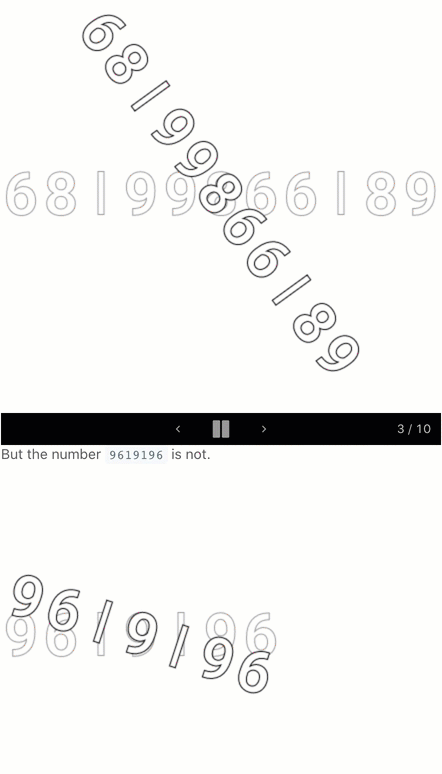
**On LeetCode, you can simulate the clarification process using the Run Code button!**. Note that this process will **not** affect your solution acceptance statistics. You can use the *Run Code* button as much as you want. If you're not familiar with the trick, here's how you would check whether or not 2 is considered to be strobogrammatic.

1. If you haven’t yet written any code, you’ll first need to modify the template function to return a dummy value (e.g. return false).
2. Type 2 into the *Testcase* panel.
3. Click the *Run Code* button.
4. Check what the ***Expected*** output is. In this case, it is false, allowing us to conclude that 2 is **not** a strobogrammatic number.

After checking test cases, we can refine our thoughts from above.

8, 0, and 1 are themselves upside-down. 6 and 9 are each other upside-down. All other numbers, including 2 and 5, are not themselves up-side-down, nor are they any other number up-side-down.

Therefore, the number 68199866189 is rotatable.



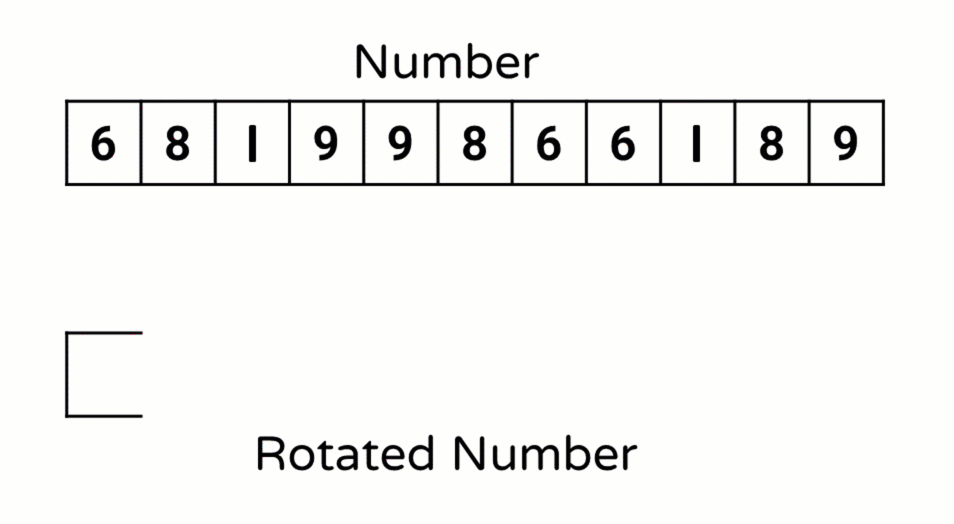
Learning to deal with vague requirements in a calm and methodical way is an often overlooked but very important soft skill for both interviews and day-to-day work as a software engineer. If you fail an interview due to misunderstanding the requirements, then this is a sign that you need to work on your requirement clarification skills and not a sign that the interviewer just asked you a "stupid" question.

From here, we're going to look at a couple of different ways of solving this problem.

#### **Approach 1: Make a Rotated Copy**

**Intuition**

A straightforward solution is to make a copy of the input that is rotated by 180 degrees. If the rotated copy is identical to the original input, then the input has to be strobogrammatic.



**Algorithm**

Recall that when a number is rotated by 180 degrees, *the order of the digits reverses* and *each digit is rotated upside-down in its new position*. As such, we could build a new rotated string by looping through the original string backward (to reverse it) and rotating + appending each digit to the new string.

Recall from above that the rules for rotating a character are as follows:

* 0 ⟶ 0
* 1 ⟶ 1
* 6 ⟶ 9
* 8 ⟶ 8
* 9 ⟶ 6

The digits 2, 3, 4, 5, and 7 are not rotatable. Their presence immediately signifies that the input number couldn't possibly be rotatable, and therefore not strobogrammatic.

The simplest way of doing the rotations in code is to use if statements.

define function isStrobogrammatic(num\_string):

rotated\_string = an empty string

for each character in num\_string, going in reverse:

if character is 0, 1, or 8:

append character to rotated\_string

else if character is 6:

append 9 to rotated\_string

else if character is 9:

append 6 to rotated\_string

else (character is invalid):

return false

if rotated\_string is the same as num\_string:

return true

return false

Alternatively, you could use a hash map, or even an array, to store the rules for flipping. I've provided code for these alternate approaches in the next section.

**Code**

Recall that we should always use a **string builder** to build a new string, and *not* repeated string concatenations.

In most programming languages, the built-in string type is *immutable*. This means that if the string is modified, then a new string object must be made. Appending N*N* characters to a string would, therefore, have a cost of O(N^2)*O*(*N*2), as for each of the N*N* characters, a new string of length N / 2*N*/2 (on average) has to be created. This problem is solved by using string builders. A string builder is a list-like data structure that characters are inserted into. Once all characters have been inserted, the characters in the string builder are converted to a string object.

Here is the code that uses if statements (based on the algorithm from above). Catching the "invalid digit" case in the else is best, as it avoids the need for five statements within a single condition.

|  |
| --- |
| class Solution {  public boolean isStrobogrammatic(String num) {    // Note that using a String here and appending to it would be  // poor programming practice.  StringBuilder rotatedStringBuilder = new StringBuilder();    // Remember that we want to loop backwards through the string  for (int i = num.length() - 1; i >= 0; i--) {  char c = num.charAt(i);  if (c == '0' || c == '1' || c == '8') {  rotatedStringBuilder.append(c);  } else if (c == '6') {  rotatedStringBuilder.append('9');  } else if (c == '9') {  rotatedStringBuilder.append('6');  } else { // This must be an invalid digit.  return false;  }  }    String rotatedString = rotatedStringBuilder.toString();  return num.equals(rotatedString);  }  } |

Here is the code using a Hash Map to avoid the need for a complex conditional statement.

|  |
| --- |
| class Solution {  public boolean isStrobogrammatic(String num) {    // Initialise a map with the five-digit rotation rules  Map<Character, Character> rotatedDigits = new HashMap<> (  Map.of('0', '0', '1', '1', '6', '9', '8', '8', '9', '6'));    StringBuilder rotatedStringBuilder = new StringBuilder();    // Remember that we want to loop backwards through the string  for (int i = num.length() - 1; i >= 0; i--) {  char c = num.charAt(i);  if (!rotatedDigits.containsKey(c)) {  return false; // This must be an invalid digit.  }  rotatedStringBuilder.append(rotatedDigits.get(c));  }    String rotatedString = rotatedStringBuilder.toString();  return num.equals(rotatedString);  }  } |

Alternatively, we could use an Array instead of a Hash Map; the indexes act as keys. It is simplest in code to map the non-rotatable characters to empty strings instead of explicitly checking for them during the string building process. If they were present, then the rotated string will be of a different length to the original, and therefore would be correctly flagged as not strobogrammatic in the final check. You could also check for the invalid characters in the same way we did for the hash map approach

This approach is nice in that the code is very compact—the conditional inside the loop has been eliminated—but not so nice in that it is somewhat confusing to understand. The Hash Map approach is probably safer in an interview.

|  |
| --- |
| class Solution {  public boolean isStrobogrammatic(String num) {    // In Java, we need to put '\0' to represent an empty character  char[] rotatedDigits = new char[]{'0', '1', '\0', '\0', '\0', '\0', '9', '\0', '8', '6'};  StringBuilder rotatedStringBuilder = new StringBuilder();    // Remember that we want to loop backwards through the string  for (int i = num.length() - 1; i >= 0; i--) {  char c = num.charAt(i);  int charIndex = Character.getNumericValue(c);  rotatedStringBuilder.append(rotatedDigits[charIndex]);  }    String rotatedString = rotatedStringBuilder.toString();  return num.equals(rotatedString);  }  } |

**Complexity Analysis**

Let N*N* be the length of the input string.

* Time complexity : O(N)*O*(*N*).

In the worst case, all digits in the string will be rotatable. We're going to assume this for the time complexity analysis.

For each of the N*N* digits in the string, we're looking up (with a hash map, array, or cascading if) the rotation of that digit. For all three sub-implementations, this has a cost of O(1)*O*(1). Appending to the end of a string builder is also O(1)*O*(1). Therefore, building the 180-degree rotation of a string has a cost of O(N)*O*(*N*).

In the final step, we're comparing two strings of length N*N*. This also has a cost of O(N)*O*(*N*).

This gives us O(N) + O(N)*O*(*N*)+*O*(*N*). In Big-O notation, we treat this as simply O(N)*O*(*N*).

* Space complexity : O(N)*O*(*N*).

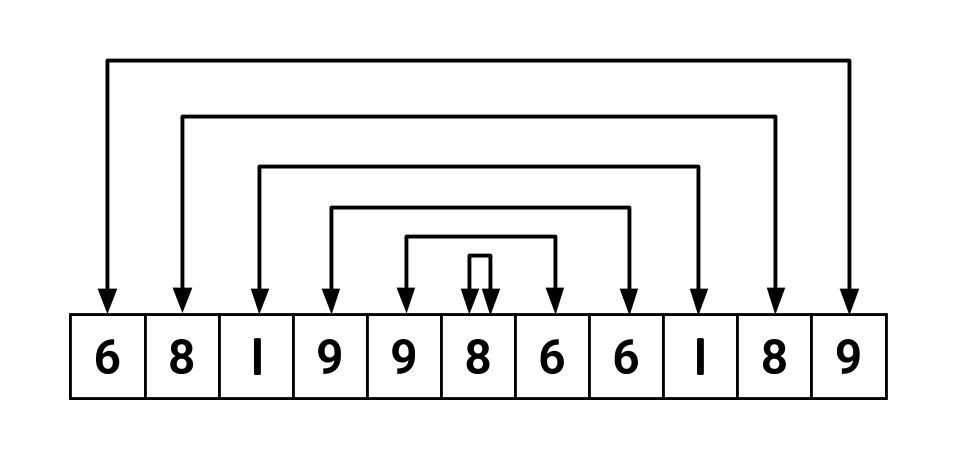
The string builder requires O(N)*O*(*N*) space.

Note that if you didn't implement your string building sensibly (e.g., if you used string concatenation), then your solution will have a time complexity of O(N^2)*O*(*N*2).

#### **Approach 2: Two Pointers**

**Intuition**

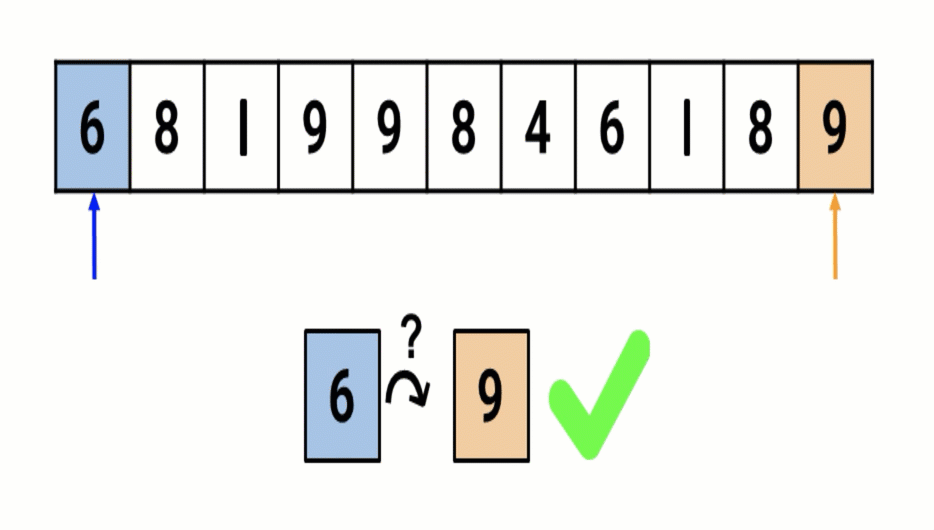
You might have observed a pattern in how digits move in the rotation: the first and last swap, the second and the second-to-last swap, etc.



For the number to be strobogrammatic, we have to write the **same** number back into each index. As we deduced earlier, there are only five valid pairs of numbers for this to work.

1. 0 and 0.
2. 1 and 1.
3. 6 and 9.
4. 8 and 8.
5. 9 and 6.

Therefore, we can check each pair that would swap in the reversal for whether or not it is one of the five pairs listed above. If all pairs are on the list, then the number is strobogrammatic. For odd-lengthed numbers, the middle digit has to be 0, 1, or 8.



**Algorithm**

We initialize two pointers; left and right. We then iterate both pointers towards the middle at each step, ensuring that the digits at left and right correspond to one of the five valid pairs. An elegant way of doing this is to define a hash map of valid left -> right mappings (like what we did in the hash map variant of approach 1).

define function isStrobogrammatic(num):

rotations = a new hash map

add to rotations: '0' -> '0', '1' -> '1', '8' -> '8', '6' -> '9', and '9' -> '6'

left = 0

right = num.length - 1

while left <= right:

if left is not in rotations:

return false

expected\_rotation = get rotation for character at position left in num

if expected\_rotation is not the same as the character at right in num:

return false

add 1 to left

subtract 1 from right

return true

If no invalid pairs are found, then the number must be strobogrammatic. Note that the middle-digit-of-an-odd-number case is handled correctly; the final iteration will have left = right. If they are both pointing to the same 0, 1, or 8, then the condition will be false, and true returned at the end. If they are both pointing at a 6, then the condition will be true, and false will be returned, as expected\_rotation will be 9, and num[right] will be 6.

Be careful of that middle value in an odd-lengthed number. While the number 8 9 0 6 8 is strobogrammatic, the number 8 9 4 6 8 is not. And nor is 8 9 9 6 8 (this last example is one that is particularly likely to catch some people out, as the middle digit is rotatable, but it doesn't become itself).

**Code**

|  |
| --- |
| class Solution {  public boolean isStrobogrammatic(String num) {    Map<Character, Character> rotatedDigits = new HashMap<> (  Map.of('0', '0', '1', '1', '6', '9', '8', '8', '9', '6'));    // Java allows us to have more than one iteration variable.  for (int left = 0, right = num.length() - 1; left <= right; left++, right--) {  char leftChar = num.charAt(left);  char rightChar = num.charAt(right);  if (!rotatedDigits.containsKey(leftChar) || rotatedDigits.get(leftChar) != rightChar) {  return false;  }  }  return true;    }  } |

**Complexity Analysis**

Let N*N* be the length of the input string.

* Time complexity : O(N)*O*(*N*).

For each of the N*N* digits in the string, we're doing a single lookup and comparison.

* Space complexity : O(1)*O*(1).

We are only using constant extra space. This is an in-place algorithm.

**Bulls and Cows**

you are playing the [**Bulls and Cows**](https://en.wikipedia.org/wiki/Bulls_and_Cows) game with your friend.

You write down a secret number and ask your friend to guess what the number is. When your friend makes a guess, you provide a hint with the following info:

* The number of "bulls", which are digits in the guess that are in the correct position.
* The number of "cows", which are digits in the guess that are in your secret number but are located in the wrong position. Specifically, the non-bull digits in the guess that could be rearranged such that they become bulls.

Given the secret number secret and your friend's guess guess, return the hint for your friend's guess.

The hint should be formatted as "xAyB", where x is the number of bulls and y is the number of cows. Note that both secret and guess may contain duplicate digits.

**Example 1:**

**Input:** secret = "1807", guess = "7810"

**Output:** "1A3B"

**Explanation:** Bulls are connected with a '|' and cows are underlined:

"1807"

|

"7810"

**Example 2:**

**Input:** secret = "1123", guess = "0111"

**Output:** "1A1B"

**Explanation:** Bulls are connected with a '|' and cows are underlined:

"1123" "1123"

| or |

"0111" "0111"

Note that only one of the two unmatched 1s is counted as a cow since the non-bull digits can only be rearranged to allow one 1 to be a bull.

**Example 3:**

**Input:** secret = "1", guess = "0"

**Output:** "0A0B"

**Example 4:**

**Input:** secret = "1", guess = "1"

**Output:** "1A0B"

**Constraints:**

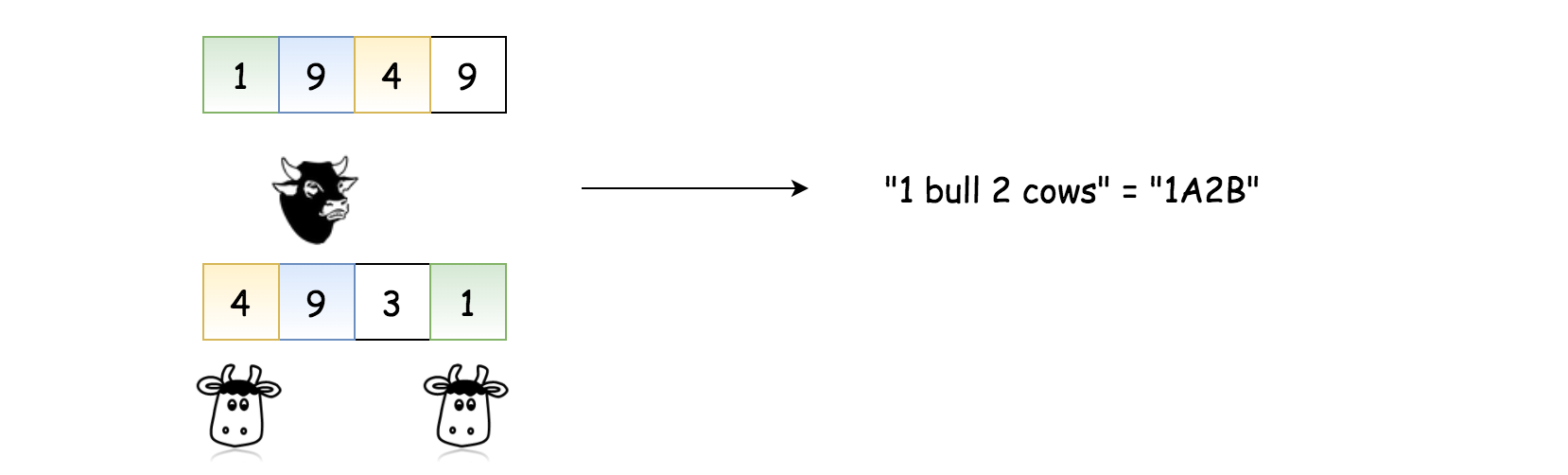
* 1 <= secret.length, guess.length <= 1000
* secret.length == guess.length
* secret and guess consist of digits only.

## Solution

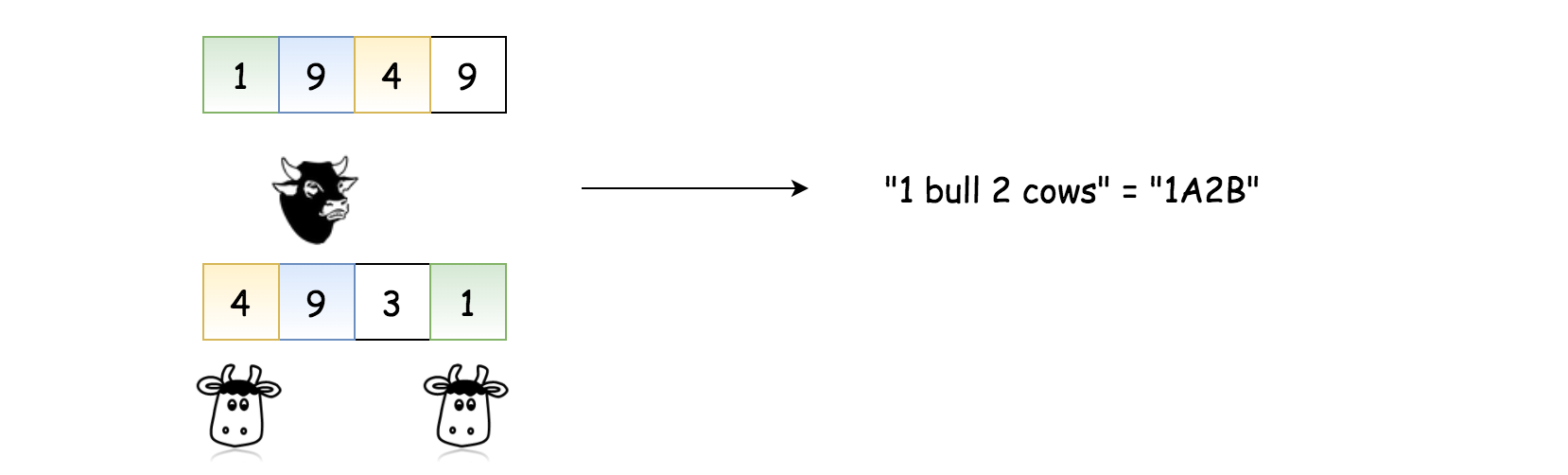
#### **Overview**

At first glance, this popular Google question seems to be easy. The only difficulty here is to define how to count.

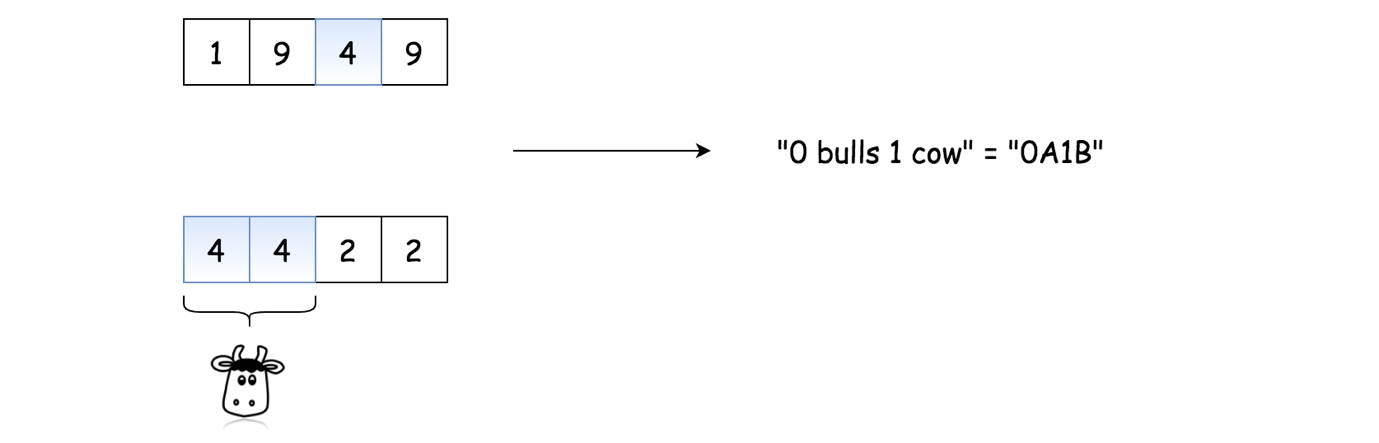
* To count bulls, one could parse both strings and count the matches, and that's a correct way to do things.

 Figure 1. Count bulls.

* To count cows, one could parse guess string and count the characters which are present in the string secret but located in the wrong positions. Though the cows are more complicated creatures than bulls. For some test cases, it might work.

 Figure 2. Count cows.

* Although, in general, one has to count the characters as well. For example, if secret contains just one digit "4" then the maximum number of 4-cows is 1. Even if guess contains many "4"s. In the following example, only one of two "4"s should be counted as a cow.

 Figure 3. Only one of two "4"s should be counted as "4-cow".

To figure out these three points is enough to solve the problem.

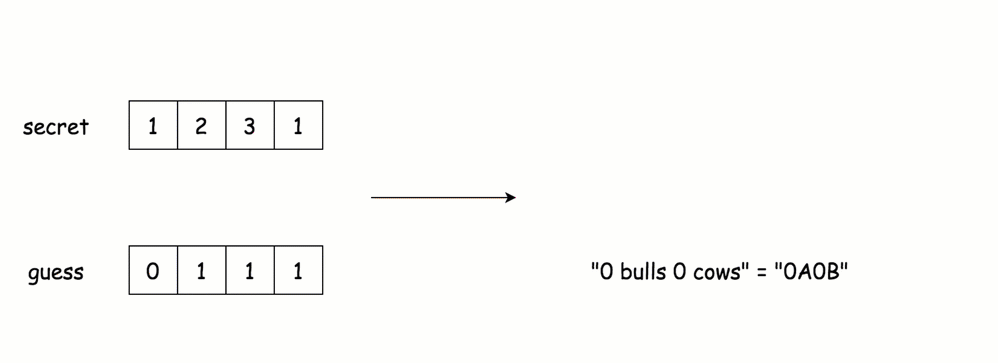
In this article, we start from a straightforward two-pass solution which already has the best time (\mathcal{O}(N)O(*N*)) and space complexity (\mathcal{O}(1)O(1)).

As a follow-up, we implement a one-pass solution and discuss some Java-related optimizations in the approach 2.

#### **Approach 1: HashMap: Two Passes**

**Algorithm**

* Initialize the number of bulls and cows to zero.
* Initialize the hashmap character -> its frequency for the string secret. This hashmap could be later used during the iteration over the string guess to keep the available characters.
* It's time to iterate over the string guess.
  + If the current character ch of the string guess is in the string secret: if ch in h, then there could be two situations.
    - The corresponding characters of two strings match: ch == secret[idx].
      * Then it's time to update the bulls: bulls += 1.
      * The update of the cows is needed if the count for the current character in the hashmap is negative or equal to zero. That means that before it was already used for cows, and the cows counter should be decreased: cows -= int(h[ch] <= 0).
    - The corresponding characters of two strings don't match: ch != secret[idx]. Then increase the cows counter: cows += int(h[ch] > 0).
    - In both cases, one has to update hashmap, marking the current character as used: h[ch] -= 1.
* Return the number of bulls and cows.



|  |
| --- |
| class Solution {  public String getHint(String secret, String guess) {  HashMap<Character, Integer> h = new HashMap();  for (char s : secret.toCharArray()) {  h.put(s, h.getOrDefault(s, 0) + 1);  }    int bulls = 0, cows = 0;  int n = guess.length();  for (int idx = 0; idx < n; ++idx) {  char ch = guess.charAt(idx);  if (h.containsKey(ch)) {  // corresponding characters match  if (ch == secret.charAt(idx)) {  // update the bulls  bulls++;  // update the cows  // if all ch characters from secret  // were used up  if (h.get(ch) <= 0)  cows--;  // corresponding characters don't match  } else {  // update the cows  if (h.get(ch) > 0)  cows++;  }  // ch character was used  h.put(ch, h.get(ch) - 1);  }  }    return Integer.toString(bulls) + "A" + Integer.toString(cows) + "B";  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*), we pass over the strings two times.
* Space complexity: \mathcal{O}(1)O(1) to keep hashmap h which contains at most 10 elements.

#### **Approach 2: One Pass**

**Intuition**

Let's optimize approach 1 by building the hashmap during the strings' parsing. That would allow us to reduce the number of passes to one.

**Algorithm**

* Initialize the number of bulls and cows to zero.
* Initialize the hashmap to count characters. During the iteration, secret string gives a positive contribution, and guess - negative contribution.
* Iterate over the strings: s is the current character in the string secret and g - the current character in the string guess.
  + If s == g, update bulls counter: bulls += 1.
  + Otherwise, if s != g:
    - Update cows by adding 1 if so far guess contains more s characters than secret: h[s] < 0.
    - Update cows by adding 1 if so far secret contains more g characters than guess: h[g] > 0.
    - Update the hashmap by marking the presence of s character in the string secret: h[s] += 1.
    - Update the hashmap by marking the presence of g character in the string guess: h[g] -= 1.
* Return the number of bulls and cows.

**Implementation**

|  |
| --- |
| class Solution {  public String getHint(String secret, String guess) {  HashMap<Character, Integer> h = new HashMap();    int bulls = 0, cows = 0;  int n = guess.length();  for (int idx = 0; idx < n; ++idx) {  char s = secret.charAt(idx);  char g = guess.charAt(idx);  if (s == g) {  bulls++;  } else {  if (h.getOrDefault(s, 0) < 0)  cows++;  if (h.getOrDefault(g, 0) > 0)  cows++;  h.put(s, h.getOrDefault(s, 0) + 1);  h.put(g, h.getOrDefault(g, 0) - 1);  }  }    return Integer.toString(bulls) + "A" + Integer.toString(cows) + "B";  }  } |

To further optimize the Java solution, one could use an array instead of hashmap because there are known problems with [Java HashMap performance](https://github.com/vavr-io/vavr/issues/571). Another small improvement is to replace string concatenation by a StringBuilder.

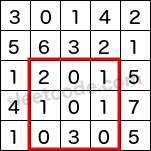
|  |
| --- |
| class Solution {  public String getHint(String secret, String guess) {  int[] h = new int[10];    int bulls = 0, cows = 0;  int n = guess.length();  for (int idx = 0; idx < n; ++idx) {  char s = secret.charAt(idx);  char g = guess.charAt(idx);  if (s == g) {  bulls++;  } else {  if (h[s - '0'] < 0)  cows++;  if (h[g - '0'] > 0)  cows++;  h[s - '0']++;  h[g - '0']--;  }  }    StringBuilder sb = new StringBuilder();  sb.append(bulls);  sb.append("A");  sb.append(cows);  sb.append("B");  return sb.toString();  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*), we do one pass over the strings.
* Space complexity: \mathcal{O}(1)O(1) to keep hashmap (or array) h which contains at most 10 elements.

**Range Sum Query 2D - Mutable**

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

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

**Example:**

Given matrix = [

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

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

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

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

[1, 0, 3, 0, 5]

]

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

update(3, 2, 2)

sumRegion(2, 1, 4, 3) -> 10

**Note:**

1. The matrix is only modifiable by the *update* function.
2. You may assume the number of calls to *update* and *sumRegion* function is distributed evenly.
3. You may assume that *row*1 ≤ *row*2 and *col*1 ≤ *col*2.

Hard

**My Calendar II**

Implement a MyCalendarTwo class to store your events. A new event can be added if adding the event will not cause a **triple** booking.

Your class will have one method, book(int start, int end). Formally, this represents a booking on the half open interval [start, end), the range of real numbers x such that start <= x < end.

A *triple booking* happens when **three** events have some non-empty intersection (ie., there is some time that is common to all 3 events.)

For each call to the method MyCalendar.book, return true if the event can be added to the calendar successfully without causing a **triple** booking. Otherwise, return false and do not add the event to the calendar.

Your class will be called like this: MyCalendar cal = new MyCalendar(); MyCalendar.book(start, end)

**Example 1:**

MyCalendar();

MyCalendar.book(10, 20); // returns true

MyCalendar.book(50, 60); // returns true

MyCalendar.book(10, 40); // returns true

MyCalendar.book(5, 15); // returns false

MyCalendar.book(5, 10); // returns true

MyCalendar.book(25, 55); // returns true

**Explanation:**

The first two events can be booked. The third event can be double booked.

The fourth event (5, 15) can't be booked, because it would result in a triple booking.

The fifth event (5, 10) can be booked, as it does not use time 10 which is already double booked.

The sixth event (25, 55) can be booked, as the time in [25, 40) will be double booked with the third event;

the time [40, 50) will be single booked, and the time [50, 55) will be double booked with the second event.

**Note:**

* The number of calls to MyCalendar.book per test case will be at most 1000.
* In calls to MyCalendar.book(start, end), start and end are integers in the range [0, 10^9].

#### **Approach #1: Brute Force [Accepted]**

**Intuition**

Maintain a list of bookings and a list of double bookings. When booking a new event [start, end), if it conflicts with a double booking, it will have a triple booking and be invalid. Otherwise, parts that overlap the calendar will be a double booking.

**Algorithm**

Evidently, two events [s1, e1) and [s2, e2) do not conflict if and only if one of them starts after the other one ends: either e1 <= s2 OR e2 <= s1. By De Morgan's laws, this means the events conflict when s1 < e2 AND s2 < e1.

If our event conflicts with a double booking, it's invalid. Otherwise, we add conflicts with the calendar to our double bookings, and add the event to our calendar.

|  |
| --- |
| public class MyCalendarTwo {  List<int[]> calendar;  List<int[]> overlaps;  MyCalendarTwo() {  calendar = new ArrayList();  }  public boolean book(int start, int end) {  for (int[] iv: overlaps) {  if (iv[0] < end && start < iv[1]) return false;  }  for (int[] iv: calendar) {  if (iv[0] < end && start < iv[1])  overlaps.add(new int[]{Math.max(start, iv[0]), Math.min(end, iv[1])});  }  calendar.add(new int[]{start, end});  return true;  }  } |

**Complexity Analysis**

* Time Complexity: O(N^2)*O*(*N*2), where N*N* is the number of events booked. For each new event, we process every previous event to decide whether the new event can be booked. This leads to \sum\_k^N O(k) = O(N^2)∑*kN*​*O*(*k*)=*O*(*N*2) complexity.
* Space Complexity: O(N)*O*(*N*), the size of the calendar.

#### **Approach #2: Boundary Count [Accepted]**

**Intuition and Algorithm**

When booking a new event [start, end), count delta[start]++ and delta[end]--. When processing the values of delta in sorted order of their keys, the running sum active is the number of events open at that time. If the sum is 3 or more, that time is (at least) triple booked.

A Python implementation was not included for this approach because there is no analog to TreeMap available.

|  |
| --- |
| class MyCalendarTwo {  TreeMap<Integer, Integer> delta;  public MyCalendarTwo() {  delta = new TreeMap();  }  public boolean book(int start, int end) {  delta.put(start, delta.getOrDefault(start, 0) + 1);  delta.put(end, delta.getOrDefault(end, 0) - 1);  int active = 0;  for (int d: delta.values()) {  active += d;  if (active >= 3) {  delta.put(start, delta.get(start) - 1);  delta.put(end, delta.get(end) + 1);  if (delta.get(start) == 0)  delta.remove(start);  return false;  }  }  return true;  }  } |

**Complexity Analysis**

* Time Complexity: O(N^2)*O*(*N*2), where N*N* is the number of events booked. For each new event, we traverse delta in O(N)*O*(*N*) time.
* Space Complexity: O(N)*O*(*N*), the size of delta.

**Jewels and Stones**

You're given strings jewels representing the types of stones that are jewels, and stones representing the stones you have. Each character in stones is a type of stone you have. You want to know how many of the stones you have are also jewels.

Letters are case sensitive, so "a" is considered a different type of stone from "A".

**Example 1:**

**Input:** jewels = "aA", stones = "aAAbbbb"

**Output:** 3

**Example 2:**

**Input:** jewels = "z", stones = "ZZ"

**Output:** 0

**Constraints:**

* 1 <= jewels.length, stones.length <= 50
* jewels and stones consist of only English letters.
* All the characters of jewels are **unique**.

HashTable

**Swap Adjacent in LR String**

In a string composed of 'L', 'R', and 'X' characters, like "RXXLRXRXL", a move consists of either replacing one occurrence of "XL" with "LX", or replacing one occurrence of "RX" with "XR". Given the starting string start and the ending string end, return True if and only if there exists a sequence of moves to transform one string to the other.

**Example 1:**

**Input:** start = "RXXLRXRXL", end = "XRLXXRRLX"

**Output:** true

**Explanation:** We can transform start to end following these steps:

RXXLRXRXL ->

XRXLRXRXL ->

XRLXRXRXL ->

XRLXXRRXL ->

XRLXXRRLX

**Example 2:**

**Input:** start = "X", end = "L"

**Output:** false

**Example 3:**

**Input:** start = "LLR", end = "RRL"

**Output:** false

**Example 4:**

**Input:** start = "XL", end = "LX"

**Output:** true

**Example 5:**

**Input:** start = "XLLR", end = "LXLX"

**Output:** false

**Constraints:**

* 1 <= start.length <= 104
* start.length == end.length
* Both start and end will only consist of characters in 'L', 'R', and 'X'.

#### **Approach 1: Invariant**

**Intuition**

Let's think of 'L' and 'R' as people facing left and right standing on one line, and 'X' as an empty space on that line.

We can ask: what invariants (conditions that remain true after making any move) there are. This is natural for any question that involves transforming some state and asking whether a final state is possible.

**Algorithm**

One invariant is that 'L' and 'R' characters in the string can never cross each other - people walking on the line cannot pass through each other. That means the starting and target strings must be the same when reading only the 'L' and 'R's. With respect to some starting string, let's call such a target string "solid" (since the people don't pass through each other).

Additionally, the n-th 'L' can never go to the right of it's original position, and similarly the n-th 'R' can never go to the left of it's original position. We'll call this "accessibility". Formally, if (i\_1, i\_2, \cdots )(*i*1​,*i*2​,⋯) and (i^{'}\_1, i^{'}\_2, \cdots )(*i*1′​,*i*2′​,⋯) are the positions of each 'L' character in the source and target string, and similarly for (j\_i \cdots ), (j^{'}\_1 \cdots )(*ji*​⋯),(*j*1′​⋯) and positions of 'R' characters, then we will say a target string is accessible if i\_k \geq i^{'}\_k*ik*​≥*ik*′​ and j\_k \leq j^{'}\_k*jk*​≤*jk*′​.

This shows being solid and accessible are necessary conditions. With an induction on the number of people, we can show that they are sufficient conditions. The basic idea of the proof is this: A person on the end either walks away from the others, or walks towards. If they walk away, then let them walk first and it is true; if they walk towards, then let them walk last and it is true (by virtue of the target being solid).

With these ideas in hand, we'll investigate the two properties individually. If they are both true, then the answer is True, otherwise it is False.

|  |
| --- |
| class Solution {  public boolean canTransform(String start, String end) {  if (!start.replace("X", "").equals(end.replace("X", "")))  return false;  int t = 0;  for (int i = 0; i < start.length(); ++i)  if (start.charAt(i) == 'L') {  while (end.charAt(t) != 'L') t++;  if (i < t++) return false;  }  t = 0;  for (int i = 0; i < start.length(); ++i)  if (start.charAt(i) == 'R') {  while (end.charAt(t) != 'R') t++;  if (i > t++) return false;  }  return true;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of start and end.
* Space Complexity: O(N)*O*(*N*). The replacement operation is O(N)*O*(*N*), while the remaining operations use O(1)*O*(1) additional space. We could amend the replace part of our algorithm to use pointers so as to reduce the total complexity to O(1)*O*(1).

#### **Approach 2: Two Pointers**

**Intuition and Algorithm**

Following the explanation in Approach #1, the target string must be solid and accessible.

We use two pointers to solve it. Each pointer i, j points to an index of start, end with start[i] != 'X', end[j] != 'X'.

Then, if start[i] != end[j], the target string isn't solid. Also, if start[i] == 'L' and i < j, (or start[i] == 'R' and i > j), the string is not accessible.

In our Python implementation, we use generators to handle moving i, j to the next index where start[i] != 'X', end[j] != 'X'.

|  |
| --- |
| class Solution {  public boolean canTransform(String start, String end) {  int n = start.length();    // count X in start and end --> should be the same  int count = 0;  for (int i = 0; i < n; ++i) {  if (start.charAt(i) == 'X') count++;  if (end.charAt(i) == 'X') count--;  }  if (count != 0) return false;  int i = 0, j = 0;  while (i < n && j < n) {  while (i < n && start.charAt(i) == 'X') i++;  while (j < n && end.charAt(j) == 'X') j++;  // i and j are the indices representing the next  // occurrences of non-X characters  if (i == n || j == n)  return i == n && j == n;  if (start.charAt(i) != end.charAt(j)) return false;  if (start.charAt(i) == 'L' && i < j) return false;  if (start.charAt(i) == 'R' && i > j) return false;    i++;  j++;  }    return true;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of start and end.
* Space Complexity: O(1)*O*(1).

**Guess the Word**

This problem is an **interactive problem** new to the LeetCode platform.

We are given a word list of unique words, each word is 6 letters long, and one word in this list is chosen as **secret**.

You may call master.guess(word) to guess a word.  The guessed word should have type string and must be from the original list with 6 lowercase letters.

This function returns an integer type, representing the number of exact matches (value and position) of your guess to the **secret word**.  Also, if your guess is not in the given wordlist, it will return -1 instead.

For each test case, you have 10 guesses to guess the word. At the end of any number of calls, if you have made 10 or less calls to master.guess and at least one of these guesses was the **secret**, you pass the testcase.

Besides the example test case below, there will be 5 additional test cases, each with 100 words in the word list.  The letters of each word in those testcases were chosen independently at random from 'a' to 'z', such that every word in the given word lists is unique.

**Example 1:**

**Input:** secret = "acckzz", wordlist = ["acckzz","ccbazz","eiowzz","abcczz"]

**Explanation:**

master.guess("aaaaaa") returns -1, because "aaaaaa" is not in wordlist.

master.guess("acckzz") returns 6, because "acckzz" is secret and has all 6 matches.

master.guess("ccbazz") returns 3, because "ccbazz" has 3 matches.

master.guess("eiowzz") returns 2, because "eiowzz" has 2 matches.

master.guess("abcczz") returns 4, because "abcczz" has 4 matches.

We made 5 calls to master.guess and one of them was the secret, so we pass the test case.

**Note:**  Any solutions that attempt to circumvent the judge will result in disqualification.

Approach #1: Minimax with Heuristic [Accepted]

**Intuition**

We can guess that having less words in the word list is generally better. If the data is random, we can reason this is often the case.

Now let's use the strategy of making the guess that minimizes the maximum possible size of the resulting word list. If we started with N*N* words in our word list, we can iterate through all possibilities for what the secret could be.

**Algorithm**

Store H[i][j] as the number of matches of wordlist[i] and wordlist[j]. For each guess that hasn't been guessed before, do a minimax as described above, taking the guess that gives us the smallest group that might occur.

|  |
| --- |
| class Solution {  int[][] H;  public void findSecretWord(String[] wordlist, Master master) {  int N = wordlist.length;  H = new int[N][N];  for (int i = 0; i < N; ++i)  for (int j = i; j < N; ++j) {  int match = 0;  for (int k = 0; k < 6; ++k)  if (wordlist[i].charAt(k) == wordlist[j].charAt(k))  match++;  H[i][j] = H[j][i] = match;  }  List<Integer> possible = new ArrayList();  List<Integer> path = new ArrayList();  for (int i = 0; i < N; ++i) possible.add(i);  while (!possible.isEmpty()) {  int guess = solve(possible, path);  int matches = master.guess(wordlist[guess]);  if (matches == wordlist[0].length()) return;  List<Integer> possible2 = new ArrayList();  for (Integer j: possible) if (H[guess][j] == matches) possible2.add(j);  possible = possible2;  path.add(guess);  }  }  public int solve(List<Integer> possible, List<Integer> path) {  if (possible.size() <= 2) return possible.get(0);  List<Integer> ansgrp = possible;  int ansguess = -1;  for (int guess = 0; guess < H.length; ++guess) {  if (!path.contains(guess)) {  ArrayList<Integer>[] groups = new ArrayList[7];  for (int i = 0; i < 7; ++i) groups[i] = new ArrayList<Integer>();  for (Integer j: possible) if (j != guess) {  groups[H[guess][j]].add(j);  }  ArrayList<Integer> maxgroup = groups[0];  for (int i = 0; i < 7; ++i)  if (groups[i].size() > maxgroup.size())  maxgroup = groups[i];  if (maxgroup.size() < ansgrp.size()) {  ansgrp = maxgroup;  ansguess = guess;  }  }  }  return ansguess;  }  } |

**Complexity Analysis**

* Time Complexity: O(N^2 \log N)*O*(*N*2log*N*), where N*N* is the number of words, and assuming their length is O(1)*O*(1). Each call to solve is O(N^2)*O*(*N*2), and the number of calls is bounded by O(\log N)*O*(log*N*).
* Space Complexity: O(N^2)*O*(*N*2).

**Minimum Area Rectangle**

Given a set of points in the xy-plane, determine the minimum area of a rectangle formed from these points, with sides parallel to the x and y axes.

If there isn't any rectangle, return 0.

**Example 1:**

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

**Output:** 4

**Example 2:**

**Input:** [[1,1],[1,3],[3,1],[3,3],[4,1],[4,3]]

**Output:** 2

**Note**:

1. 1 <= points.length <= 500
2. 0 <= points[i][0] <= 40000
3. 0 <= points[i][1] <= 40000
4. All points are distinct.

Approach 1: Sort by Column

**Intuition**

Count each rectangle by right-most edge.

**Algorithm**

Group the points by x coordinates, so that we have columns of points. Then, for every pair of points in a column (with coordinates (x,y1) and (x,y2)), check for the smallest rectangle with this pair of points as the rightmost edge. We can do this by keeping memory of what pairs of points we've seen before.

|  |
| --- |
| class Solution {  public int minAreaRect(int[][] points) {  Map<Integer, List<Integer>> rows = new TreeMap();  for (int[] point: points) {  int x = point[0], y = point[1];  rows.computeIfAbsent(x, z-> new ArrayList()).add(y);  }  int ans = Integer.MAX\_VALUE;  Map<Integer, Integer> lastx = new HashMap();  for (int x: rows.keySet()) {  List<Integer> row = rows.get(x);  Collections.sort(row);  for (int i = 0; i < row.size(); ++i)  for (int j = i+1; j < row.size(); ++j) {  int y1 = row.get(i), y2 = row.get(j);  int code = 40001 \* y1 + y2;  if (lastx.containsKey(code))  ans = Math.min(ans, (x - lastx.get(code)) \* (y2-y1));  lastx.put(code, x);  }  }  return ans < Integer.MAX\_VALUE ? ans : 0;  }  } |

**Complexity Analysis**

* Time Complexity: O(N^2)*O*(*N*2), where N*N* is the length of points.
* Space Complexity: O(N)*O*(*N*).

#### **Approach 2: Count by Diagonal**

**Intuition**

For each pair of points in the array, consider them to be the long diagonal of a potential rectangle. We can check if all 4 points are there using a Set.

For example, if the points are (1, 1) and (5, 5), we check if we also have (1, 5) and (5, 1). If we do, we have a candidate rectangle.

**Algorithm**

Put all the points in a set. For each pair of points, if the associated rectangle are 4 distinct points all in the set, then take the area of this rectangle as a candidate answer.

|  |
| --- |
| class Solution {  public int minAreaRect(int[][] points) {  Set<Integer> pointSet = new HashSet();  for (int[] point: points)  pointSet.add(40001 \* point[0] + point[1]);  int ans = Integer.MAX\_VALUE;  for (int i = 0; i < points.length; ++i)  for (int j = i+1; j < points.length; ++j) {  if (points[i][0] != points[j][0] && points[i][1] != points[j][1]) {  if (pointSet.contains(40001 \* points[i][0] + points[j][1]) &&  pointSet.contains(40001 \* points[j][0] + points[i][1])) {  ans = Math.min(ans, Math.abs(points[j][0] - points[i][0]) \*  Math.abs(points[j][1] - points[i][1]));  }  }  }  return ans < Integer.MAX\_VALUE ? ans : 0;  }  } |

**Complexity Analysis**

* Time Complexity: O(N^2)*O*(*N*2), where N*N* is the length of points.
* Space Complexity: O(N)*O*(*N*).

Here's a list of topics I found helpful for Google. These are based on questions I found tagged with Google across LeetCode and conversations with recruiters/interview-experiences.

1. Topological sort
2. DFS/BFS
3. 2-D Matrix Prefix Sum
4. Binary Search over functions
5. Quick sort/Quick select
6. Merge sort
7. Dijkstra/Bellman Ford/ Floyd Warshall/ Kruskals Minimum Spanning Tree
8. UnionFind
9. Trie
10. Min-max Games
11. Cycle Detection
12. Intervals
13. Segment Tree
14. KMP
15. Bridges/Articulation point/Tarjan SCC
16. Euler's Path/Circuit/Hierholzer's Algorithm

Note that this is more topics than you would expect for other companies. Google really does try to emphasise on DSA questions more than most other companies. Still topics 13 to 16 are (somewhat) optional with it being rarer to see direct questions.

Amazon relies a lot more on design and leadership for SDE-2 onwards.  
Facebook favours implementation speed in DSA with 2 LeetCode Mediums (ideally, both solved with code!) rather than harder questions.  
Google asks LeetCode Mediums and occasionally Hards. It's less rushed than Facebook but expect follow-ups.

A lot of the more obscure topics above like Segment Tree, Bridges, KMP, you can expect them to ask you in follow-up rather than the core questioning absolutely requiring them. So they are sort of optional but definitely leave a good impression in follow-ups.

And they typically don't expect as much (sometimes, zero) code for the follow-ups.

Google also has points for code style so do spend a few hours going through their style code: <https://google.github.io/styleguide/javaguide.html>  
Again, it's optional brownie points and doesn't matter if you can't solve the base questions. But it's a good guide to follow during work/LeetCode too :)