**May 2020 Challenge**

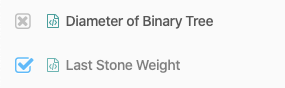
Introduction

This Challenge is beginner-friendly and available to both Premium and non-Premium users. It consists of 31 daily problems over the month of May. A problem is added here each day, and you have 24 hours to make a valid submission for it in order to be eligible for rewards.

Rules

In order to be eligible for prizes, you must comply with all the Challenge rules.

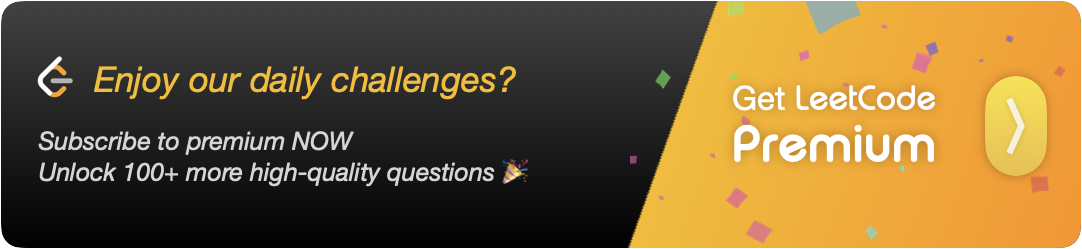
1. **Valid submission:**only problems where a valid submission is made will be eligible for rewards. This means the submission was:
   1. within the **valid time period** for that problem; this is before 23:59 PM PDT on the day it appeared.
   2. made **within this Explore Card**. If you have completed the question previously, you will need to re-submit an answer within this Explore Card.
   3. given the judgement **"Accepted"** when submitted. Please note that if your submission receives a different judgement, including, but not limited to, Wrong Answer or Time Limit Exceeded, you are allowed to resubmit, as long as it's still within the valid time period.

After making a valid submission for a problem, a **blue checkmark** will appear next to that problem in the left bar. A grey cross means that a valid submission was not made on time for that problem, and so, unfortunately, you're no longer eligible for prizes for that problem.  


1. **Late or invalid submissions:**will not be accepted under **any** circumstances. It is ***your responsibility*** to make a valid submission for a problem within its valid time period, and to confirm that the blue checkmark has appeared.

1. **Rewards:**only valid submissions will be eligible for rewards. For more information about the rewards, [check out the Discuss post](https://leetcode.com/discuss/general-discussion/595334/May-LeetCoding-Challenge).

1. **Disclaimer:** LeetCode reserves the right, in our sole discretion, to disqualify any entries where we believe a user undermines the fairness of this May's LeetCoding Challenge event, which includes, but is not limited to, copying and pasting solutions from other places directly into your submission.

[](https://leetcode.com/subscribe/?ref=ex_dc)

Week 1: May 1st–May 7th

~~+10~~

 First Bad Version

~~+10~~

 Jewels and Stones

~~+10~~

**Ransom Note**

~~+10~~

**Number Complement**

~~+10~~

 First Unique Character in a String

~~+10~~

 Majority Element

~~+10~~

**Cousins in Binary Tree**

Week 2: May 8th–May 14th

~~+10~~

**Check If It Is a Straight Line**

~~+10~~

**Valid Perfect Square**

~~+10~~

**Find the Town Judge**

~~+10~~

 Flood Fill

~~+10~~

**Single Element in a Sorted Array**

~~+10~~

**Remove K Digits**

~~+10~~

 Implement Trie (Prefix Tree)

Week 3: May 15th–May 21st

~~+10~~

**Maximum Sum Circular Subarray**

~~+10~~

**Odd Even Linked List**

~~+10~~

**Find All Anagrams in a String**

~~+10~~

 Permutation in String

~~+10~~

**Online Stock Span**

~~+10~~

 Kth Smallest Element in a BST

~~+10~~

**Count Square Submatrices with All Ones**

Week 4: May 22nd–May 28th

~~+10~~

**Sort Characters By Frequency**

~~+10~~

**Interval List Intersections**

~~+10~~

 Construct Binary Search Tree from Preorder Traversal

~~+10~~

**Uncrossed Lines**

~~+10~~

 Contiguous Array

~~+10~~

**Possible Bipartition**

~~+10~~

 Counting Bits

Week 5: May 29th–May 31st

~~+10~~

 Course Schedule

~~+10~~

 K Closest Points to Origin

~~+10~~

**Ransom Note**

Given an arbitrary ransom note string and another string containing letters from all the magazines, write a function that will return true if the ransom note can be constructed from the magazines ; otherwise, it will return false.

Each letter in the magazine string can only be used once in your ransom note.

**Example 1:**

**Input:** ransomNote = "a", magazine = "b"

**Output:** false

**Example 2:**

**Input:** ransomNote = "aa", magazine = "ab"

**Output:** false

**Example 3:**

**Input:** ransomNote = "aa", magazine = "aab"

**Output:** true

**Constraints:**

* You may assume that both strings contain only lowercase letters.

## Solution

Something you might notice when you run code for this problem here on Leetcode is that Approach 1 passes, and is the fastest. This is because all the testcases are very small. For huge test cases though, the other approaches would beat it, and Approach 1 would be far too slow.

In an interview, it's unlikely that Approach 1 would be sufficient to get you the job. Interviewers will expect to see an optimized approach such as Approach 2 or 3.

#### **Approach 1: Simulation**

**Intuition**

To create our ransom note, for every character we have in the note, we need to take a copy of that character out of the magazine so that it can go into the note.

If a character we need isn't in the magazine, then we should stop and return False. Otherwise, if we manage to get all the characters we need to complete the note, then we should return True.

For each char in ransomNote:

Find that letter in magazine.

If it is in magazine:

Remove it from magazine.

Else:

Return False

Return True

Note that there's no need to explicitly build up the ransom note; we only need to return whether or not it's possible. This can be determined simply by removing the characters we need from the magazine.

This is the most straightforward approach, but as we'll see soon, although it does pass here on Leetcode, it's not very efficient and is not likely to get you a job at a top company.

**Algorithm**

Strings are an **immutable** type. This means that they can't be modified, and so don't have "insert" and "delete" operations. For this reason, we instead need to repeatedly replace the magazine with a new String, that doesn't have the character we wanted to remove.

|  |
| --- |
| public boolean canConstruct(String ransomNote, String magazine) {  // For each character, c, in the ransom note.  for (char c : ransomNote.toCharArray()) {  // Find the index of the first occurrence of c in the magazine.  int index = magazine.indexOf(c);  // If there are none of c left in the String, return False.  if (index == -1) {  return false;  }  // Use substring to make a new string with the characters  // before "index" (but not including), and the characters  // after "index".  magazine = magazine.substring(0, index) + magazine.substring(index + 1);  }  // If we got this far, we can successfully build the note.  return true;  } |

**Complexity Analysis**

We'll say m*m* is the length of the **m**agazine, and n*n* is the length of the ransom **n**ote.

* Time Complexity : O(m \cdot n)*O*(*m*⋅*n*).

Finding the letter we need in the magazine has a cost of O(m)*O*(*m*). This is because we need to perform a linear search of the magazine. Removing the letter we need from the magazine is also O(m)*O*(*m*). This is because we need to make a new string to represent it. O(m) + O(m) = O(2 \cdot m) = O(m)*O*(*m*)+*O*(*m*)=*O*(2⋅*m*)=*O*(*m*) because we drop constants in big-o analysis.

So, how many times are we performing this O(m)*O*(*m*) operation? Well, we are looping through each of the n*n* characters in the ransom note and performing it once for each letter. This is a total of n*n* times, and so we get n \cdot O(m) = O(m \cdot n)*n*⋅*O*(*m*)=*O*(*m*⋅*n*).

* Space Complexity : O(m)*O*(*m*).

Creating a new magazine with one letter less requires auxillary space the length of the magazine; O(m)*O*(*m*).

#### **Approach 2: Two HashMaps**

**Intuition**

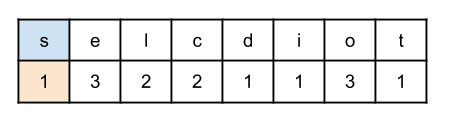
Remember that we decided the length of the ransom **n**ote is n*n*, and the length of the **m**agazine is m*m*.

In an interview, you might start by describing the previous approach and determining its time complexity, but not actually implementing it. Your next goal would be to reason carefully about the implementation and its time complexity, to identify parts that could be made more efficient.

Removing the n*n* factor from the time complexity is going to be impossible, because we need to at least look at each character in the ransom note. Otherwise, how could we possibly know whether or not we have the characters we need to make it? We might be able to avoid the need for an O(m)*O*(*m*) operation for every one of the n*n* characters in the ransom note though.

As an example, notice that if there's three 'a''s in the ransom note, then there needs to be at least three 'a's in the magazine. This should be fairly intuitive, as you'd encounter it if trying to make a note out of a magazine for real. The same idea applies for all the other unique characters too.

Therefore, a better way of solving the problem would be to count up how many of each letter are in both the magazine and the ransom note. We can represent the counts with a HashMap that has characters as keys, and counts as values. For example, the string "leetcode is cool" is represented as follows.



We can make two HashMaps; one for the magazine, and the other for the ransom note. Here is the pseudocode for making one of these "counts" HashMaps.

define function makeCountsMap(string):

counts = a new HashMap

for each char in string:

if char not in counts:

counts.put(char, 1)

else:

old\_count = counts.get(char)

counts.put(char, old\_count + 1)

return counts

Then, to actually check whether or not the ransom note can be made using the magazine, we should loop over each character of the ransom note, checking how many of it we need, and checking that at least that many exist in the magazine, by looking it up in the magazine HashMap. We need to be careful of the case where the character we need isn't in the magazine at all; in this case we should return False as the number of them in the magazine is definitely smaller than the number we need. If we manage to check all the characters without False being returned, then we know that we must have had enough characters to complete the note, and can therefore return True. Here is some pseudocode for that algorithm.

noteCounts = makeCountsMap(ransomNote)

magazineCounts = makeCountsMap(magazine)

for each (char, count) in noteCounts:

if char is not in magazineCounts:

return False

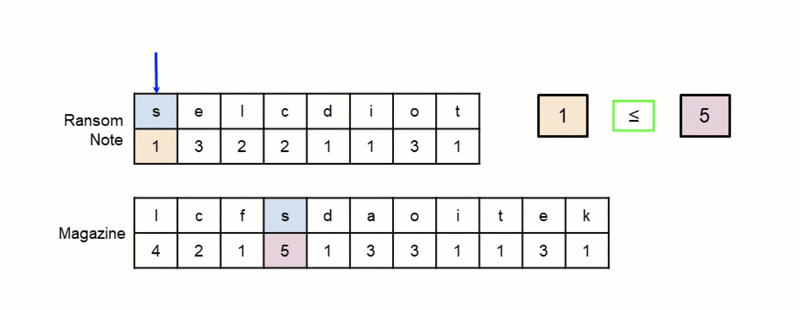
countInMagazine = magazineCounts.get(char)

if countInMagazine < count:

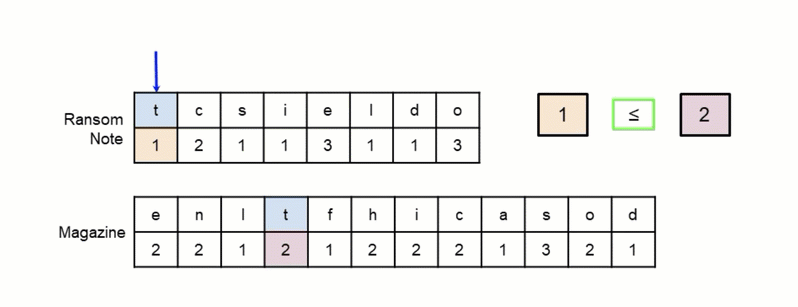
return False

return True

Here is an animation showing the above algorithm in action with the ransom note "leetcode is cool" and the magazine "close call as fools take sides".



And here is another example, with the same ransom note, but the magazine "cats close in on the fish".



There's one more optimization we can make. Notice that if the length of the ransom note is *longer* than the length of the magazine, then its impossible for there to be enough characters in the magazine.

**Algorithm**

|  |
| --- |
| class Solution {    // Takes a String, and returns a HashMap with counts of  // each character.  private Map<Character, Integer> makeCountsMap(String s) {  Map<Character, Integer> counts = new HashMap<>();  for (char c : s.toCharArray()) {  int currentCount = counts.getOrDefault(c, 0);  counts.put(c, currentCount + 1);  }  return counts;  }      public boolean canConstruct(String ransomNote, String magazine) {    // Check for obvious fail case.  if (ransomNote.length() > magazine.length()) {  return false;  }  // Make the count maps.  Map<Character, Integer> ransomNoteCounts = makeCountsMap(ransomNote);  Map<Character, Integer> magazineCounts = makeCountsMap(magazine);    // For each unique character, c, in the ransom note:  for (char c : ransomNoteCounts.keySet()) {  // Check that the count of char in the magazine is equal  // or higher than the count in the ransom note.  int countInMagazine = magazineCounts.getOrDefault(c, 0);  int countInRansomNote = ransomNoteCounts.get(c);  if (countInMagazine < countInRansomNote) {  return false;  }  }    // If we got this far, we can successfully build the note.  return true;  }  } |

**Complexity Analysis**

We'll say m*m* is the length of the **m**agazine, and n*n* is the length of the ransom **n**ote.

Also, let k*k* be the number of unique characters across both the ransom note and magazine. While this is never more than 2626, we'll treat it as a variable for a more accurate complexity analysis.

The basic HashMap operations, get(...) and put(...), are O(1)*O*(1) time complexity.

* Time Complexity : O(m)*O*(*m*).

When m < n*m*<*n*, we immediately return false. Therefore, the worst case occurs when m ≥ n*m*≥*n*.

Creating a HashMap of counts for the magazine is O(m)*O*(*m*), as each insertion/ count update is is O(1)*O*(1), and is done for each of the m*m* characters.

Likewise, creating the HashMap of counts for the ransom note is O(n)*O*(*n*).

We then iterate over the ransom note HashMap, which contains at most n*n* unique values, looking up their counterparts in the magazine `HashMap. This is, therefore, at worst O(n)*O*(*n*).

This gives us O(n) + O(n) + O(m)*O*(*n*)+*O*(*n*)+*O*(*m*). Now, remember how we said m ≥ n*m*≥*n*? This means that we can simplify it to O(m) + O(m) + O(m) = 3 \cdot O(m) = O(m)*O*(*m*)+*O*(*m*)+*O*(*m*)=3⋅*O*(*m*)=*O*(*m*), dropping the constant of 33.

* Space Complexity : O(k)*O*(*k*) / O(1)*O*(1).

We build two HashMaps of counts; each with up to k*k* characters in them. This means that they take up O(k)*O*(*k*) space.

For this problem, because k*k* is never more than 2626, which is a constant, it'd be reasonable to say that this algorithm requires O(1)*O*(1) space.

#### **Approach 3: One HashMap**

**Intuition**

In the previous approach, we used two HashMaps. You might have noticed a slightly better way though; we can simply put the magazine into a HashMap, and then subtract characters from the ransom note from it. Here is the pseudocode, using our makeCountsMap(...) function from above.

magazineCounts = makeCountsMap(magazine)

for each char in ransomNote:

countInMagazine = magazineCounts.get(char)

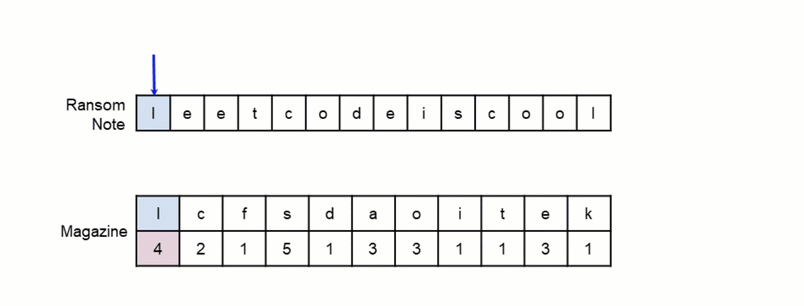
if countInMagazine == 0:

return False

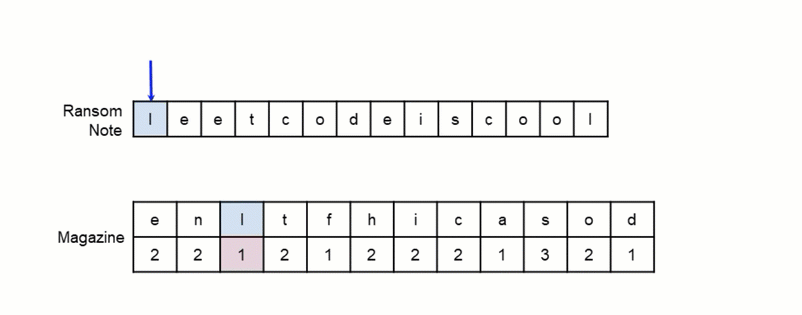
magazineCounts.put(char, countInMagazine - 1)

return True

Here is an animation of the algorithm on our "true" case from before.



And here's one on the "false" case.



|  |
| --- |
| class Solution {    // Takes a String, and returns a HashMap with counts of  // each character.  private Map<Character, Integer> makeCountsMap(String s) {  Map<Character, Integer> counts = new HashMap<>();  for (char c : s.toCharArray()) {  int currentCount = counts.getOrDefault(c, 0);  counts.put(c, currentCount + 1);  }  return counts;  }      public boolean canConstruct(String ransomNote, String magazine) {    // Check for obvious fail case.  if (ransomNote.length() > magazine.length()) {  return false;  }  // Make a counts map for the magazine.  Map<Character, Integer> magazineCounts = makeCountsMap(magazine);    // For each character in the ransom note:  for (char c : ransomNote.toCharArray()) {  // Get the current count for c in the magazine.  int countInMagazine = magazineCounts.getOrDefault(c, 0);  // If there are none of c left, return false.  if (countInMagazine == 0) {  return false;  }  // Put the updated count for c back into magazineCounts.  magazineCounts.put(c, countInMagazine - 1);  }    // If we got this far, we can successfully build the note.  return true;  }  } |

**Complexity Analysis**

We'll say m*m* is the length of the **m**agazine, and n*n* is the length of the ransom **n**ote.

Also, let k*k* be the number of unique characters across both the ransom note and magazine. While this is never more than 2626, we'll treat it as a variable for a more accurate complexity analysis.

The basic HashMap operations, get(...) and put(...), are O(1)*O*(1) time complexity.

* Time Complexity : O(m)*O*(*m*).

When m < n*m*<*n*, we immediately return false. Therefore, the worst case occurs when m ≥ n*m*≥*n*.

Creating a HashMap of counts for the magazine is O(m)*O*(*m*), as each insertion/ count update is is O(1)*O*(1), and is done for each of the m*m* characters.

We then iterate over the ransom note, performing an O(1)*O*(1) operation for each character in it. This has a cost of O(n)*O*(*n*).

Becuase we know that m ≥ n*m*≥*n*, again this simplifies to O(m)*O*(*m*).

* Space Complexity : O(k)*O*(*k*) / O(1)*O*(1).

Same as above.

For this problem, because k*k* is never more than 2626, which is a constant, it'd be reasonable to say that this algorithm requires O(1)*O*(1) space.

#### **Approach 4: Sorting and Stacks**

**Intuition**

This approach isn't needed for an interview, and is better than Approach 1, but worse than Approach 2 and 3. I've included it because it's still very cool and might give you additional creative ideas for when tackling related problems! :)

Another, completely different, way of solving the problem is to start by converting each string into an Array of characters, and then reverse sorting them by alphabetical order. It's not actually necessary to reverse sort, but it will make things easier for the rest of the algorithm. For example, here's the sorted characters for the ransom note leetcode is cool and the magazine close call as fools take sides.



Now, convert each array into a stack.

Compare the tops of the stacks. There are three possibilities.

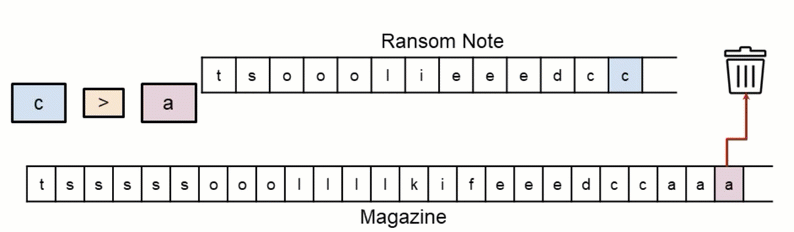
1. The characters are the same.
2. The ransom note character is earlier in the alphabet than the magazine character.
3. The ransom note character is later in the alphabet than the magazine character.

For the first possibility, we've found a copy of the letter we need in the magazine, for a letter in our ransom note. So pop the top off each stack.

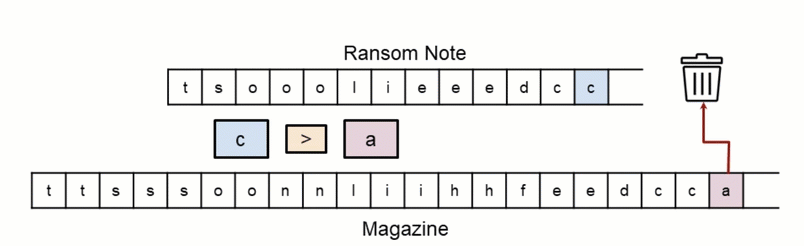
For the second possibility, we know that the letter we need can't be on the magazine stack. This is because all the other characters on the magazine must be even later than the top, and we needed an earlier letter. Therefore, we can return false now.

For the third possibility, we know that the letter on the top of the magazine stack will never be needed, as all the characters on the ransom note stack must be later than it, so we pop the top off just the magazine stack.

Here is an animation of the algorithm on our "true" case from before.



And here's one on the "false" case.



|  |
| --- |
| class Solution {    // Please, if there's a nicer way of doing this, without getting tangled in  // primitives vs Java's generics let me know in the article comments :-)  private Stack<Character> sortedCharacterStack(String s) {  char[] charArray = s.toCharArray();  Arrays.sort(charArray);  Stack<Character> stack = new Stack<>();  for (int i = s.length() - 1; i >= 0; i--) {  stack.push(charArray[i]);  }  return stack;  }      public boolean canConstruct(String ransomNote, String magazine) {    // Check for obvious fail case.  if (ransomNote.length() > magazine.length()) {  return false;  }    // Reverse sort the characters of the note and magazine, and then  // put them into stacks.  Stack<Character> magazineStack = sortedCharacterStack(magazine);  Stack<Character> ransomNoteStack = sortedCharacterStack(ransomNote);    // And now process the stacks, while both have letters remaining.  while (!magazineStack.isEmpty() && !ransomNoteStack.isEmpty()) {  // If the tops are the same, pop both because we have found a match.  if (magazineStack.peek().equals(ransomNoteStack.peek())) {  ransomNoteStack.pop();  magazineStack.pop();  }  // If magazine's top is earlier in the alphabet, we should remove that  // character of magazine as we definitely won't need that letter.  else if (magazineStack.peek() < ransomNoteStack.peek()) {  magazineStack.pop();  }  // Otherwise, it's impossible for top of ransomNote to be in magazine.  else {  return false;  }  }    // Return true iff the entire ransomNote was built.  return ransomNoteStack.isEmpty();    }  } |

**Complexity Analysis**

We'll say m*m* is the length of the **m**agazine, and n*n* is the length of the ransom **n**ote.

* Time Complexity : O(m \, \log \, m)*O*(*m*log*m*).

When m < n*m*<*n*, we immediately return false. Therefore, the worst case occurs when m ≥ n*m*≥*n*.

Sorting the magazine is O(m \, \log \, m)*O*(*m*log*m*). Inserting the contents into the stack is O(m)*O*(*m*), which is insignificant. This, therefore, gives us O(m \, \log \, m)*O*(*m*log*m*) for creating the magazine stack.

Likewise, creating the ransom note stack is O(n \, \log \, n)*O*(*n*log*n*).

In total, the stacks contain n + m*n*+*m* characters. For each iteration of the loop, we are either immediately returning false, or removing at least one character from the stacks. This means that the stack processing loop has to use at most O(n + m)*O*(*n*+*m*) time.

This gives us  *O*(*m*log*m*)+*O*(*n*log*n*)+*O*(*n*+*m*). Now, remembering that m ≥ n*m*≥*n* it simplifies down to *O*(*m*log*m*)+*O*(*m*log*m*)+*O*(*m*+*m*)=2⋅*O*(*m*log*m*)+*O*(2⋅*m*)=*O*(*m*log*m*).

* Space Complexity : O(m)*O*(*m*).

The magazine stack requires O(m)*O*(*m*) space, and the ransom note stack requires O(n)*O*(*n*) space. Because m ≥ n*m*≥*n*, this simplifies down to O(m)*O*(*m*).

**Number Complement**

Given a **positive** integer num, output its complement number. The complement strategy is to flip the bits of its binary representation.

**Example 1:**

**Input:** num = 5

**Output:** 2

**Explanation:** The binary representation of 5 is 101 (no leading zero bits), and its complement is 010. So you need to output 2.

**Example 2:**

**Input:** num = 1

**Output:** 0

**Explanation:** The binary representation of 1 is 1 (no leading zero bits), and its complement is 0. So you need to output 0.

**Constraints:**

* The given integer num is guaranteed to fit within the range of a 32-bit signed integer.
* num >= 1
* You could assume no leading zero bit in the integer’s binary representation.
* This question is the same as 1009: <https://leetcode.com/problems/complement-of-base-10-integer/>

## Solution

#### **Prerequisites**

**XOR**

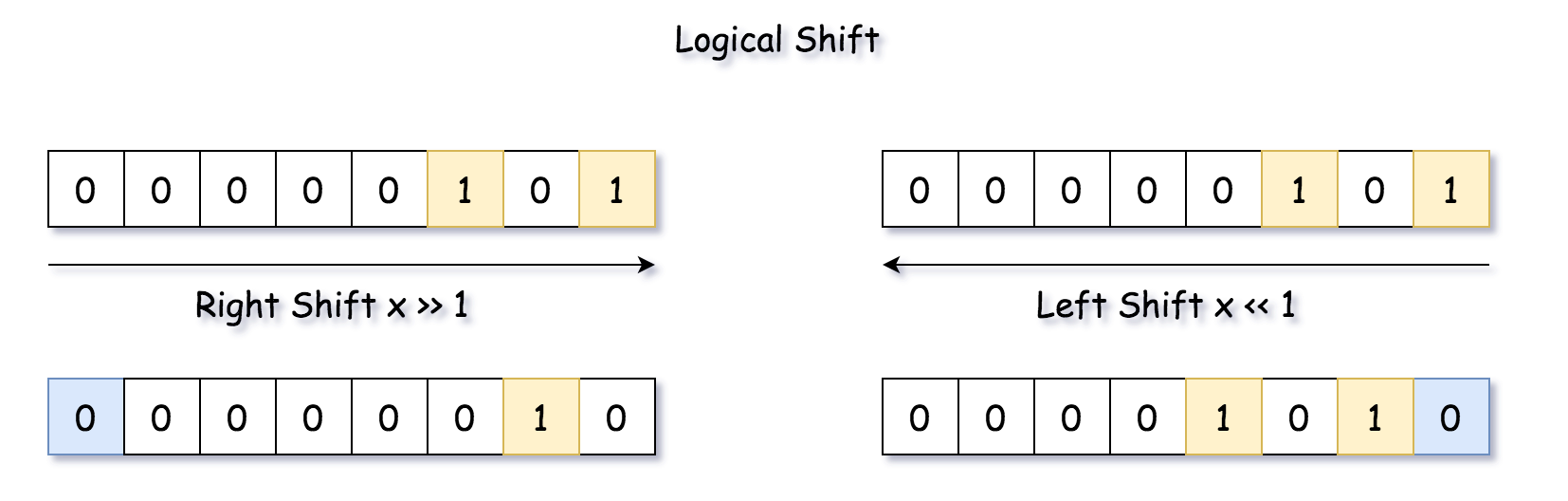
[XOR](https://en.wikipedia.org/wiki/Exclusive_or) of zero and a bit results in that bit

0 \oplus x = x0⊕*x*=*x*

XOR of one and a bit flips that bit

1 \oplus x = 1 - x1⊕*x*=1−*x*

**Right Shift and Left Shift**



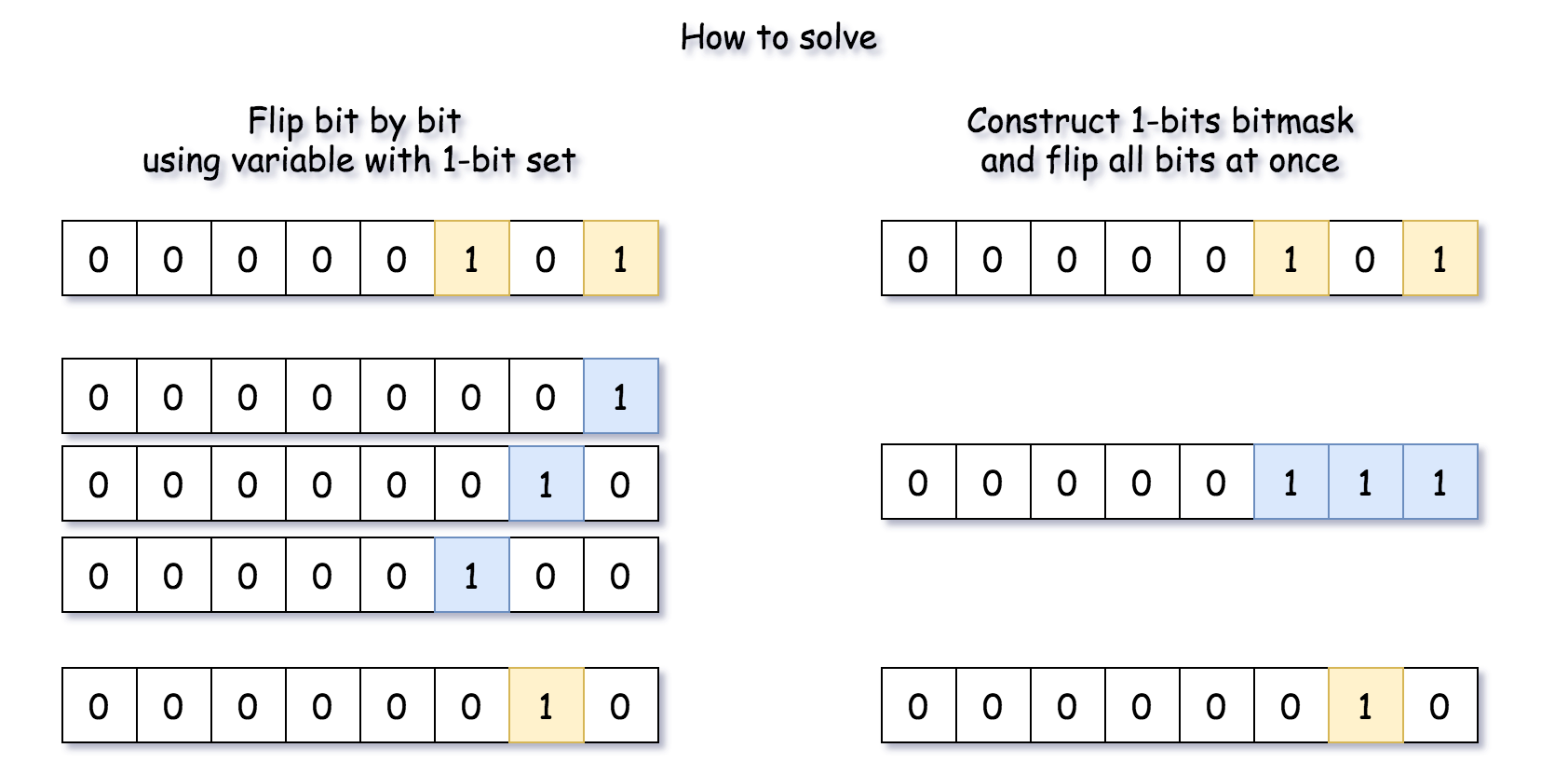
#### **Overview**

The article is long, and the best approach is the one number 4. In the case of limited time, you could jump to it directly.

There are two standard ways to solve the problem:

* To move along the number and flip bit by bit.
* To construct 1-bits bitmask which has the same length as the input number, and to get the answer as bitmask - num or bitmask ^ num.

For example, for \textrm{num} = 5 = (101)\_2num=5=(101)2​ the bitmask is \textrm{bitmask} = (111)\_2bitmask=(111)2​, and the complement number is \textrm{bitmask} \oplus \textrm{num} = (010)\_2 = 2bitmask⊕num=(010)2​=2.



#### **Approach 1: Flip Bit by Bit**

**Algorithm**

* Initiate 1-bit variable which will be used to flip bits one by one. Set it to the smallest register bit = 1.
* Initiate the marker variable which will be used to stop the loop over the bits todo = num.
* Loop over the bits. While todo != 0:
  + Flip the current bit: num = num ^ bit.
  + Prepare for the next run. Shift flip variable to the left and todo variable to the right.
* Return num.



**Implementation**

|  |
| --- |
| class Solution {  public int findComplement(int num) {  int todo = num, bit = 1;  while (todo != 0) {  // flip current bit  num = num ^ bit;  // prepare for the next run  bit = bit << 1;  todo = todo >> 1;  }  return num;  }  } |

**Complexity**

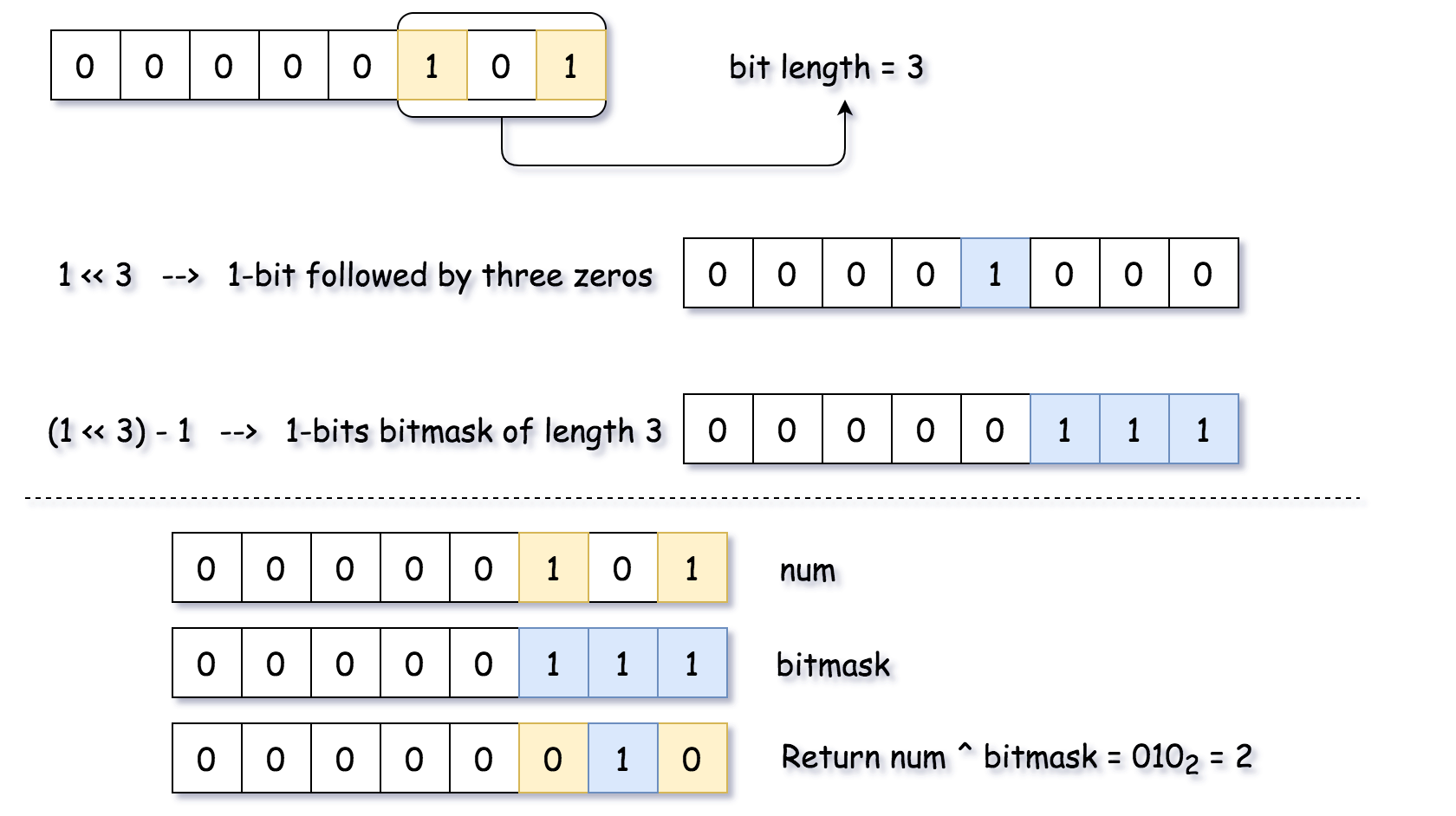
* Time Complexity: \mathcal{O}(1)O(1), since we're doing not more than 32 iterations here.
* Space Complexity: \mathcal{O}(1)O(1).

#### **Approach 2: Compute Bit Length and Construct 1-bits Bitmask**

Instead of flipping bits one by one, let's construct 1-bits bitmask and flip all the bits at once.

There are many ways to do it, let's start from the simplest one:

* Compute bit length of the input number l = [\log\_2 \textrm{num}] + 1*l*=[log2​num]+1.
* Compute 1-bits bitmask of length l*l*: \textrm{bitmask} = (1 << l) - 1bitmask=(1<<*l*)−1.
* Return num ^ bitmask.



**Implementation**

|  |
| --- |
| class Solution {  public int findComplement(int num) {  // n is a length of num in binary representation  int n = (int)( Math.log(num) / Math.log(2) ) + 1;  // bitmask has the same length as num and contains only ones 1...1  int bitmask = (1 << n) - 1;  // flip all bits  return bitmask ^ num;  }  } |

**Complexity**

* Time Complexity: \mathcal{O}(1)O(1).
* Space Complexity: \mathcal{O}(1)O(1).

#### **Approach 3: Built-in Functions to Construct 1-bits Bitmask**

Approach 2 could be rewritten with the help of built-in functions: bit\_length in Python and highestOneBit in Java. The first one is trivial, and Integer.highestOneBit(int x) method in Java returns int with leftmost bit set in x, i.e. Integer.highestOneBit(3) = 2.

**Implementation**

|  |
| --- |
| class Solution {  public int findComplement(int num) {  return (Integer.highestOneBit(num) << 1) - num - 1;  }  } |

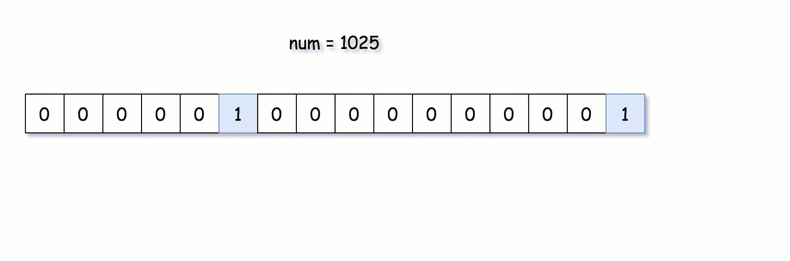
**Complexity**

* Time Complexity: \mathcal{O}(1)O(1) because one deals here with integers of not more than 32 bits.
* Space Complexity: \mathcal{O}(1)O(1).

#### **Approach 4: highestOneBit OpenJDK algorithm from Hacker's Delight**

The best algorithm for this task is an implementation of highestOneBit in OpenJDK. [This implementation is taken from "Hacker's Delight" book](http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/lang/Integer.java#l40).

The idea is to create the same 1-bits bitmask by propagating the highest 1-bit into the lower ones.



|  |
| --- |
| class Solution {  public int findComplement(int num) {  // bitmask has the same length as num and contains only ones 1...1  int bitmask = num;  bitmask |= (bitmask >> 1);  bitmask |= (bitmask >> 2);  bitmask |= (bitmask >> 4);  bitmask |= (bitmask >> 8);  bitmask |= (bitmask >> 16);  // flip all bits  return bitmask ^ num;  }  } |

**Complexity**

* Time Complexity: \mathcal{O}(1)O(1).
* Space Complexity: \mathcal{O}(1)O(1).

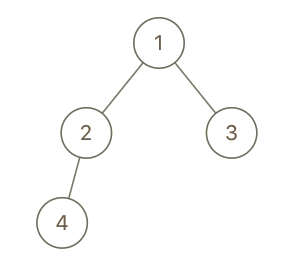
**Cousins in Binary Tree**

In a binary tree, the root node is at depth 0, and children of each depth k node are at depth k+1.

Two nodes of a binary tree are cousins if they have the same depth, but have **different parents**.

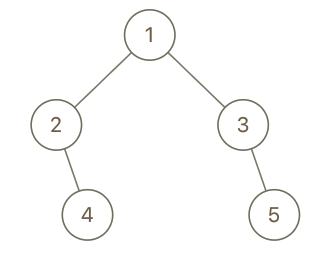
We are given the root of a binary tree with unique values, and the values x and y of two different nodes in the tree.

Return true if and only if the nodes corresponding to the values x and y are cousins.

**Example 1:  
**

**Input:** root = [1,2,3,4], x = 4, y = 3

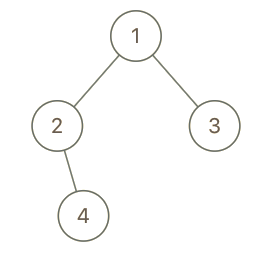
**Output:** false

**Example 2:  
**

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

**Output:** true

**Example 3:**

****

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

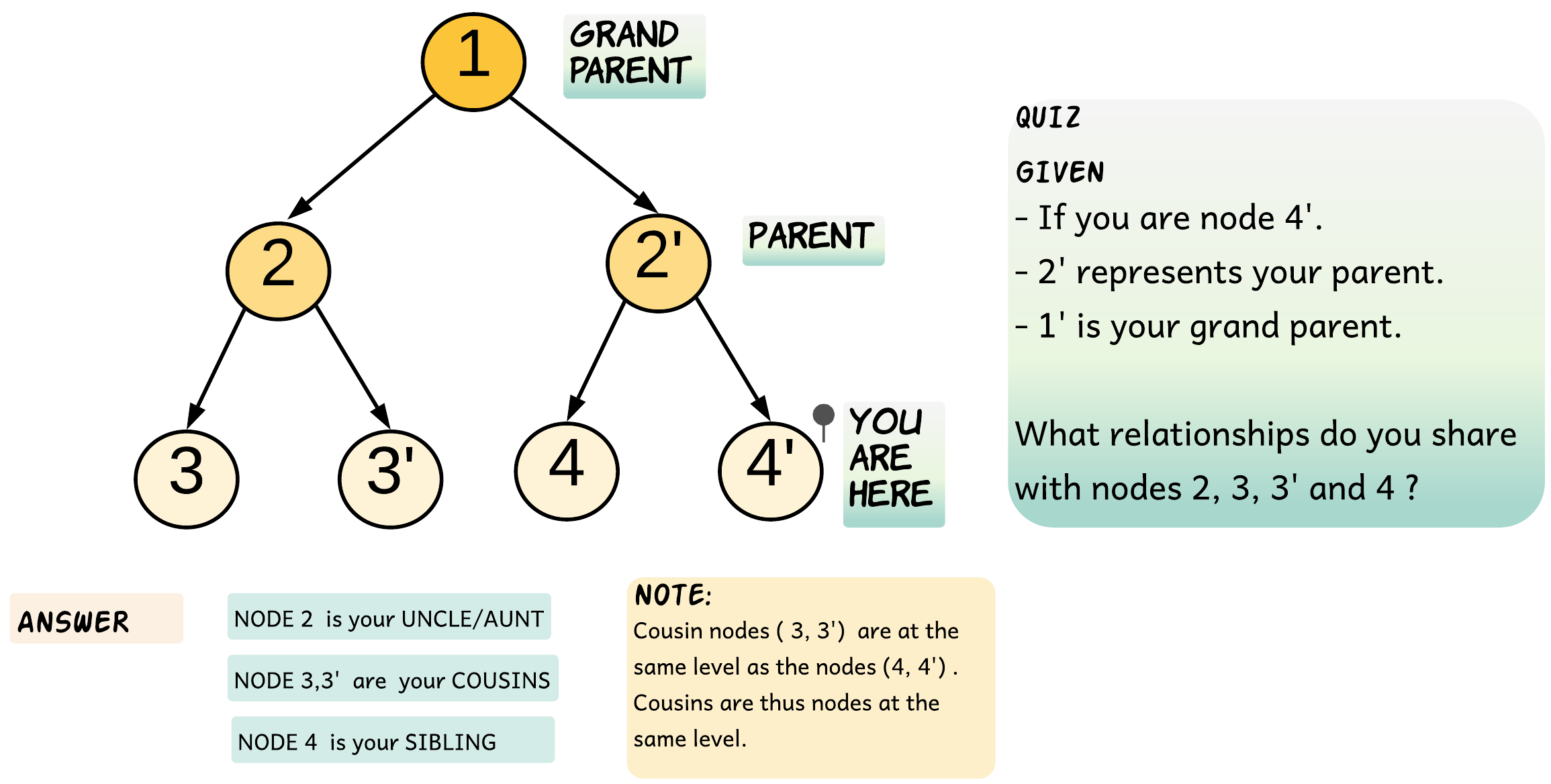
**Output:** false

**Constraints:**

* The number of nodes in the tree will be between 2 and 100.
* Each node has a unique integer value from 1 to 100.

## Solution

Let's first take a quiz and see how good your aptitude is. This quiz is mainly to help you understand the question better. If you understand the question well, then just hop on to the approaches.



Let me admit this, I was really bad at visualizing family trees and hence would perform badly in family tree questions.

The best I could do was draw a family tree and visualize it and then come to some conclusion. That's why this aptitude question was thrown at you and that explanation was needed :P

This was an easy one. What if we are given a family tree with a depth of 25 and now the same question is posed? Maybe you can answer this question only if it involves your cousins, or maybe not.

#### **Approach 1: Depth First Search with Branch Pruning**

**Intuition**

We can do a depth-first traversal and find the depth and parent for each node. Once we know the depth and parent for each node we can easily find out if they are cousins. Let's look at the pseudo-code for this before we try to optimize it a bit.

// This pseudo-code recursively traverses the tree and

// records the depth and parent for each node.

function dfs(node, parentNode = None) {

if (node != null) {

depth[node.val] = 1 + depth[parentNode.val]

parent[node.val] = parentNode

dfs(node.left, node)

dfs(node.right, node)

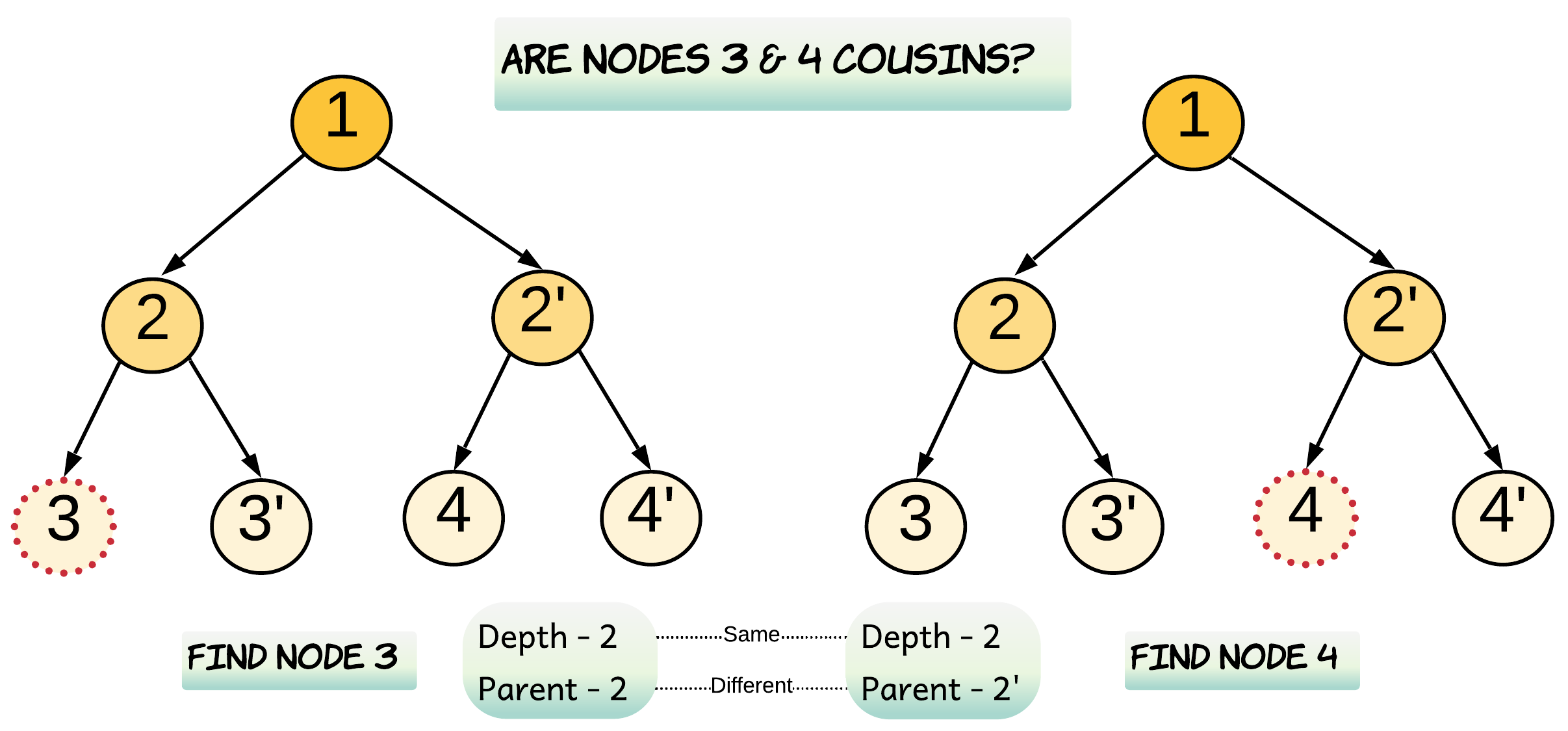
}

}

The above pseudo-code would give us the depth and parent for each node. To find out whether or not x and y are cousins is just one step away.

// If x and y are at same depth but have different parents.

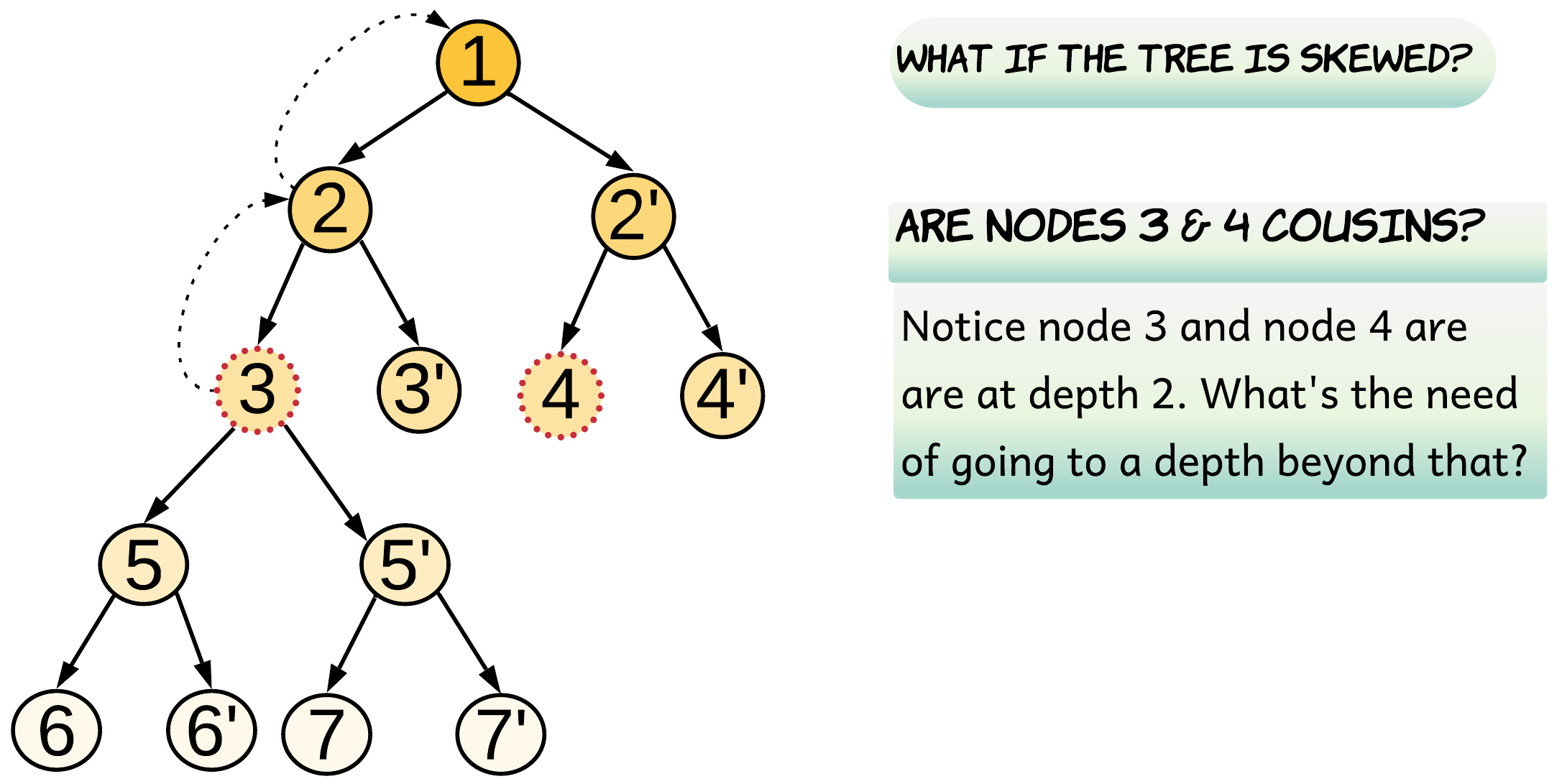
depth[x] == depth[y] and parent[x] != parent[y]



Now let's see if this brute-force recursive approach can be optimized for some scenarios.

If Node x or Node y is lying very shallow in the tree, then does it make any sense to iterate down the entire tree?

In the above example, Node 3 and Node 4 are both cousins and hence at the same depth. What if we find one of the nodes very early on during the traversal? How would that help us?



The diagram above shows that we encounter Node 3 very early on. This would help us to restrict our search space for the other node i.e. Node 4. For the second node, we do not need to go beyond the depth at which the first node was found, thus saving traversal of the subtree below node 3.

We can search for the desired nodes in the tree recursively. Whenever either of the given nodes is found, we record its parent and depth.

**Algorithm**

1. Start traversing the tree from the root node. Look for Node x and Node y.
2. Record the depth when the first node i.e. either of x or y is found and return true.
3. Once one of the nodes is discovered, for every other recursive call after this discovery, we return false if the current depth is more than the recorded depth. This basically means we didn't find the other node at the same depth and there is no point going beyond. This step of pruning helps to speed up the recursion by reducing the number of recursive calls.
4. Return true when the other node is discovered and has the same depth as the recorded depth.
5. Recurse the left and the right subtree of the current node. If both left and right recursions return true and the current node is not their immediate parent, then Node x and Node y are cousins. Thus, isCousin is set to value true.

|  |
| --- |
| /\*\*  \* Definition for a binary tree node.  \* public class TreeNode {  \* int val;  \* TreeNode left;  \* TreeNode right;  \* TreeNode(int x) { val = x; }  \* }  \*/  class Solution {  // To save the depth of the first node.  int recordedDepth = -1;  boolean isCousin = false;  private boolean dfs(TreeNode node, int depth, int x, int y) {  if (node == null) {  return false;  }  // Don't go beyond the depth restricted by the first node found.  if (this.recordedDepth != -1 && depth > this.recordedDepth) {  return false;  }  if (node.val == x || node.val == y) {  if (this.recordedDepth == -1) {  // Save depth for the first node found.  this.recordedDepth = depth;  }  // Return true, if the second node is found at the same depth.  return this.recordedDepth == depth;  }  boolean left = dfs(node.left, depth + 1, x, y);  boolean right = dfs(node.right, depth + 1, x, y);  // this.recordedDepth != depth + 1 would ensure node x and y are not  // immediate child nodes, otherwise they would become siblings.  if (left && right && this.recordedDepth != depth + 1) {  this.isCousin = true;  }  return left || right;  }  public boolean isCousins(TreeNode root, int x, int y) {  // Recurse the tree to find x and y  dfs(root, 0, x, y);  return this.isCousin;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the number of nodes in the binary tree. In the worst case, we might have to visit all the nodes of the binary tree.

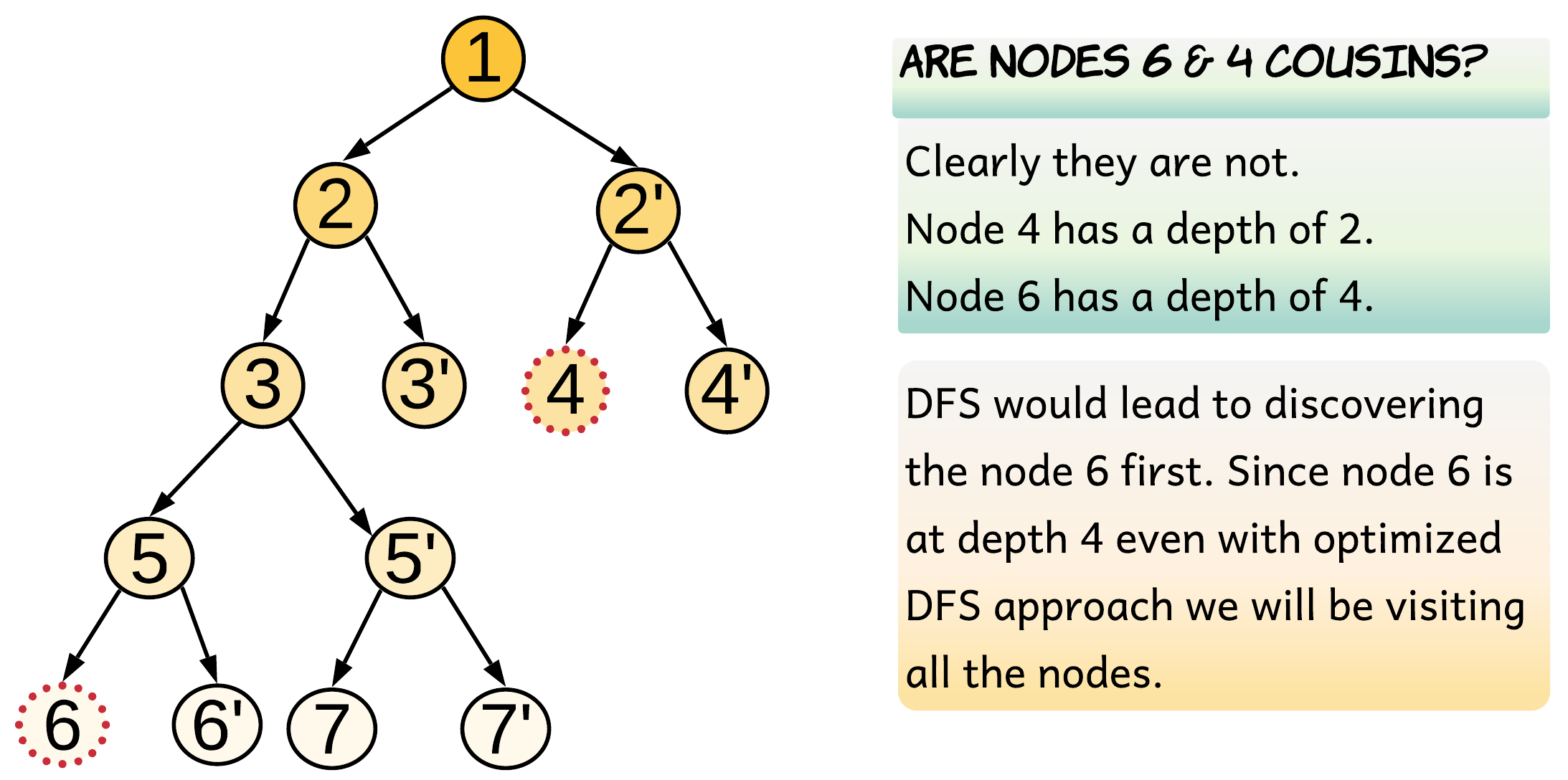
Let's look into one such scenario. When both Node x and Node y are the leaf nodes and at the last level of the tree, the algorithm has no reasons to prune the recursion. It can only come to a conclusion once it visits both the nodes. If one of these nodes is the last node to be discovered the algorithm inevitably goes through each and every node in the tree.

* Space Complexity: O(N)*O*(*N*). This is because the maximum amount of space utilized by the recursion stack would be N*N*, as the height of a skewed binary tree could be, at worst, N*N*. For a left skewed or a right skewed binary tree, where the desired nodes are lying at the maximum depth possible, the algorithm would have to maintain a recursion stack of the height of the tree.

#### **Approach 2: Breadth First Search with Early Stopping**

I will repeat my question

If Node x or Node y is lying very shallow in the tree, then does it make any sense to iterate down the entire tree?



Since this problem is about finding cousins, i.e. nodes lying at the same level/depth, it seems more natural to do a level order traversal of the tree.

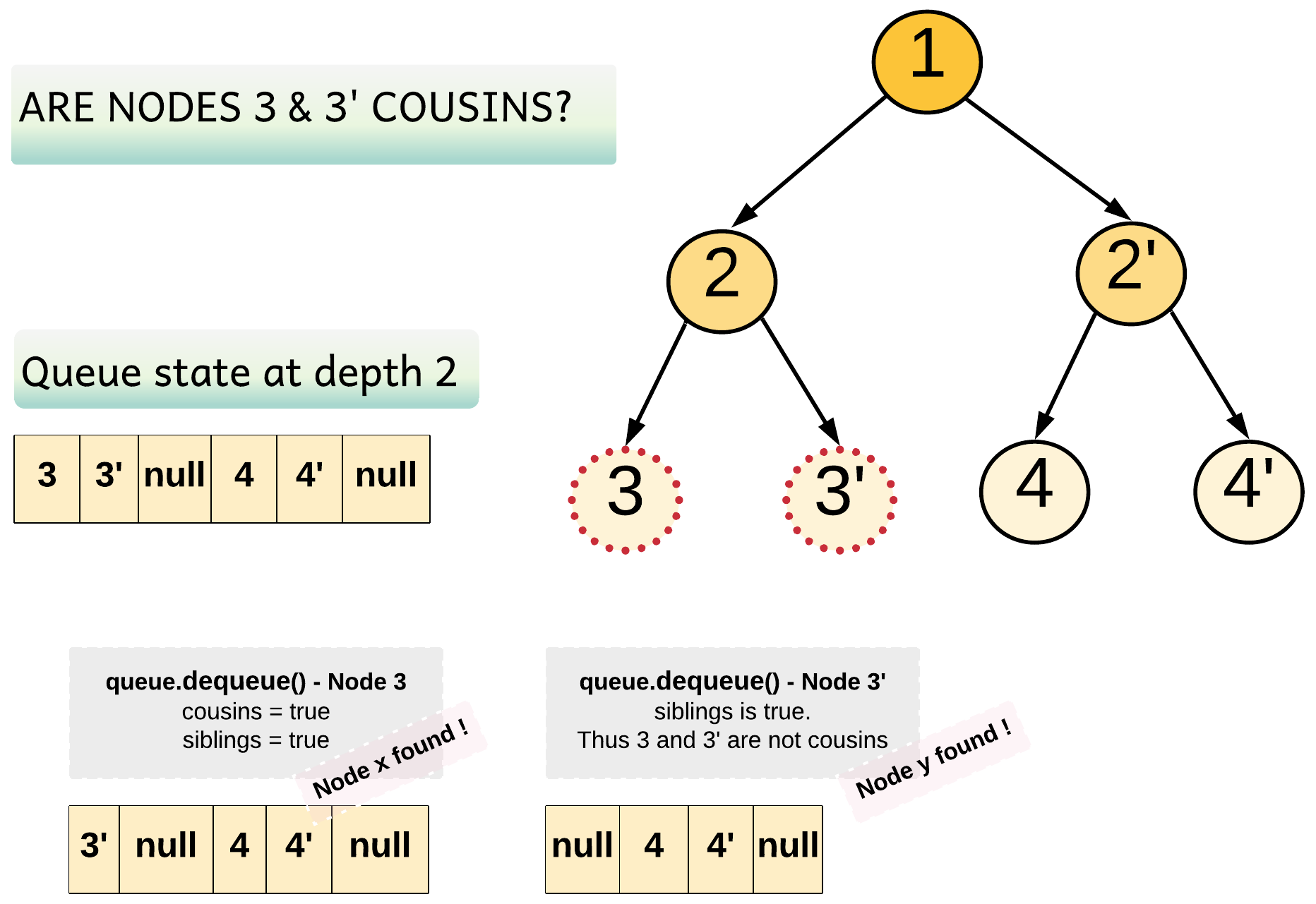
If we do a level order traversal for the aforementioned example, we would only traverse until depth 2. At depth 2, we discover Node 4, but we do not find Node 6 at the same level. Hence we can just stop our traversal and conclude that the nodes are not cousins.

Note, if the nodes are cousins, we would find both the nodes at the same depth. However, this is also true for siblings. We need to figure out how to determine when two nodes are siblings. One way to find out that they are siblings is when we are adding the nodes to the queue. If Node x and Node y are left and right children of a node, this would mean that they are siblings. Therefore, we would return false.

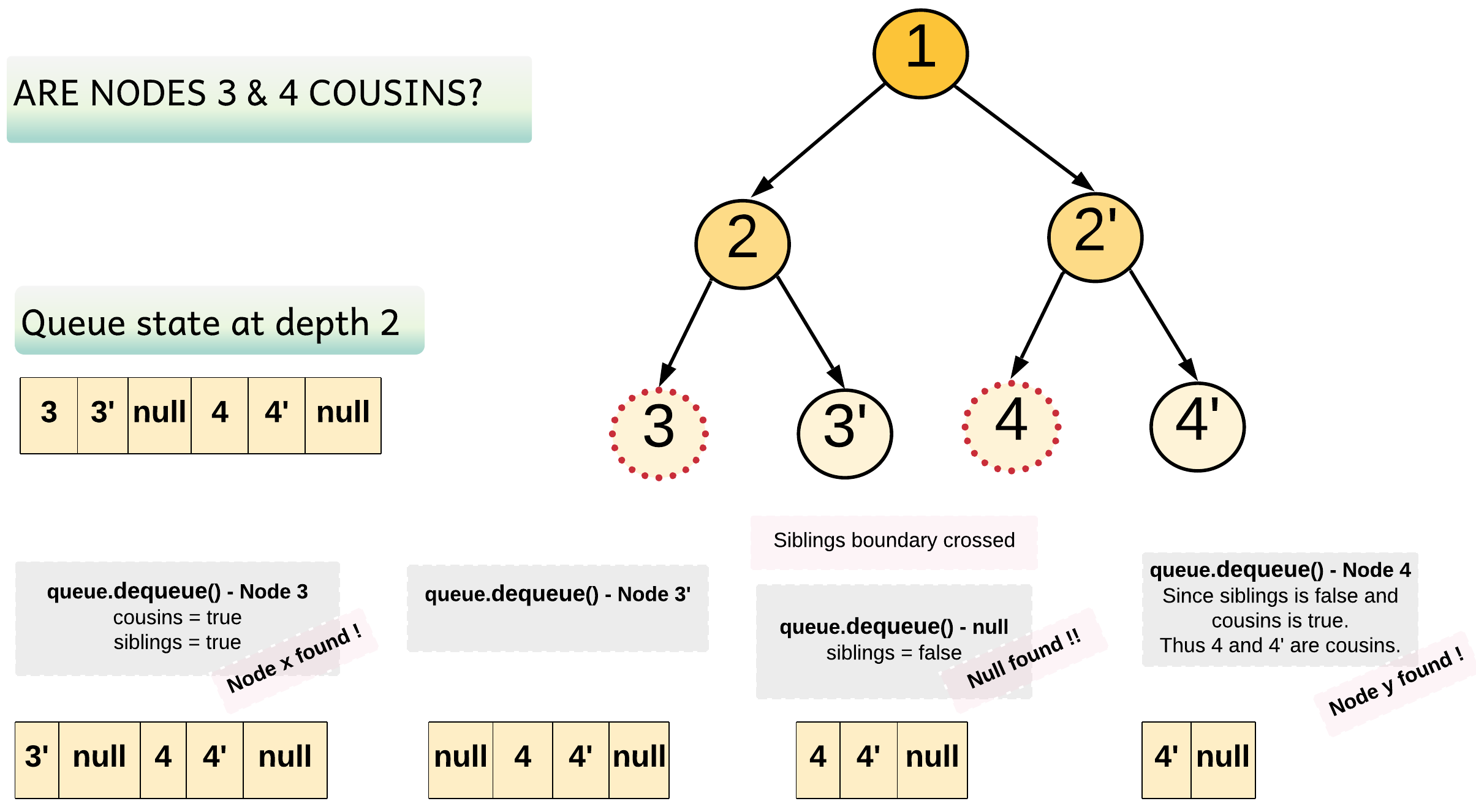
There is a cleaner implementation for the level order traversal for this problem, though. For each node, we can add a delimiter to the queue after its children are added. These delimiters help us define boundaries for each parent and the siblings that are confined within those. This implementation was shared by [@votrubac](https://leetcode.com/votrubac/). You can refer to his C++ implementation in the [discussion section](https://leetcode.com/problems/cousins-in-binary-tree/discuss/238624/C++-level-order-traversal)

**Algorithm**

1. Do a level order traversal of the tree using a queue.
2. For every node that is popped off the queue, check if the node is either Node x or Node y. If it is, then for the first time, set both siblings and cousins flags as true. The flags are set as true to mark the possibility of siblings or cousins.
3. To distinguish siblings from cousins, we insert markers in the queue. After inserting the children for each node, we also insert a null marker. This marker defines a boundary for each set of siblings and hence helps us to differentiate a pair of siblings from cousins.
4. Whenever we encounter the null marker during our traversal, we set the siblings flag to false. This is because the marker marks the end of the siblings territory.
5. The second time we encounter a node which is equal to Node x or Node y we will have clarity about whether or not we are still within the siblings boundary. If we are within the siblings boundary, i.e. if the siblings flag is still true, then we return false. Otherwise, we return true.



In the above diagram, Node 3 and Node 3' are children of the same parent. Hence the siblings flag remains true.



Clearly, Node 3 and Node 4 have different parents. Hence, we do encounter a null marker after Node 3'. The null marker marks the end of siblings for Node 3, and hence when Node 4 is found, we know it is the cousin of Node 3.

|  |
| --- |
| /\*\*  \* Definition for a binary tree node.  \* public class TreeNode {  \* int val;  \* TreeNode left;  \* TreeNode right;  \* TreeNode(int x) { val = x; }  \* }  \*/  class Solution {  public boolean isCousins(TreeNode root, int x, int y) {  // Queue for BFS  Queue <TreeNode> queue = new LinkedList <> ();  queue.add(root);  while (!queue.isEmpty()) {  boolean siblings = false;  boolean cousins = false;  int nodesAtDepth = queue.size();  for (int i = 0; i < nodesAtDepth; i++) {  // FIFO  TreeNode node = queue.remove();  // Encountered the marker.  // Siblings should be set to false as we are crossing the boundary.  if (node == null) {  siblings = false;  } else {  if (node.val == x || node.val == y) {  // Set both the siblings and cousins flag to true  // for a potential first sibling/cousin found.  if (!cousins) {  siblings = cousins = true;  } else {  // If the siblings flag is still true this means we are still  // within the siblings boundary and hence the nodes are not cousins.  return !siblings;  }  }  if (node.left != null) queue.add(node.left);  if (node.right != null) queue.add(node.right);  // Adding the null marker for the siblings  queue.add(null);  }  }  // After the end of a level if `cousins` is set to true  // This means we found only one node at this level  if (cousins) return false;  }  return false;  }  } |

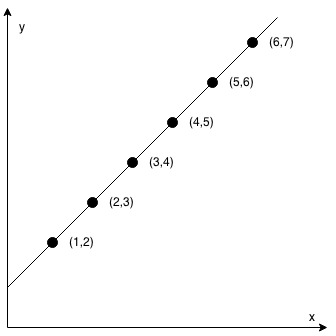
**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the number of nodes in the binary tree. In the worst case, we might have to visit all the nodes of the binary tree. Similar to approach 1 this approach would also have a complexity of O(N)*O*(*N*) when the Node x and Node y are present at the last level of the binary tree. The algorithm would follow the standard BFS approach and end up in checking each node before discovering the desired nodes.
* Space Complexity: O(N)*O*(*N*). In the worst case, we need to store all the nodes of the last level in the queue. The last level of a binary tree can have a maximum of \frac{N}{2}2*N*​ nodes. Not to forget we would also need space for \frac{N}{4}4*N*​ null markers, one for each pair of siblings. That results in a space complexity of O(\frac{3N}{4})*O*(43*N*​) = O(N)*O*(*N*) (You are right Big-O notation doesn't care about constants).

**Check If It Is a Straight Line**

You are given an array coordinates, coordinates[i] = [x, y], where [x, y] represents the coordinate of a point. Check if these points make a straight line in the XY plane.

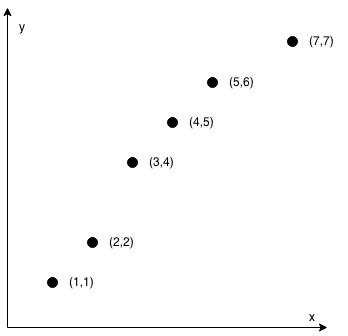
**Example 1:**



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

**Output:** true

**Example 2:**

****

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

**Output:** false

**Constraints:**

* 2 <= coordinates.length <= 1000
* coordinates[i].length == 2
* -10^4 <= coordinates[i][0], coordinates[i][1] <= 10^4
* coordinates contains no duplicate point.

 Hide Hint #1

If there're only 2 points, return true.

   Hide Hint #2

Check if all other points lie on the line defined by the first 2 points.

   Hide Hint #3

Use cross product to check collinearity.

**Valid Perfect Square**

Given a **positive** integer *num*, write a function which returns True if *num* is a perfect square else False.

**Follow up:** **Do not** use any built-in library function such as sqrt.

**Example 1:**

**Input:** num = 16

**Output:** true

**Example 2:**

**Input:** num = 14

**Output:** false

**Constraints:**

* 1 <= num <= 2^31 - 1

## Solution

#### **Overview**

Square root related problems usually could be solved in logarithmic time. There are three standard logarithmic time approaches, listed here from the worst to the best:

* Recursion. The slowest one.
* Binary Search. The simplest one.
* Newton's Method. The fastest one, and therefore widely used in dynamical simulations.

The last two algorithms are interesting ones, let's discuss them in details.

These solutions have the same starting point. If one knows an [integer square](https://en.wikipedia.org/wiki/Integer_square_root) x*x* of num, the answer is straightforward: num is a perfect square if x \* x == \textrm{num}*x*∗*x*==num. Hence the problem is to compute this integer square.

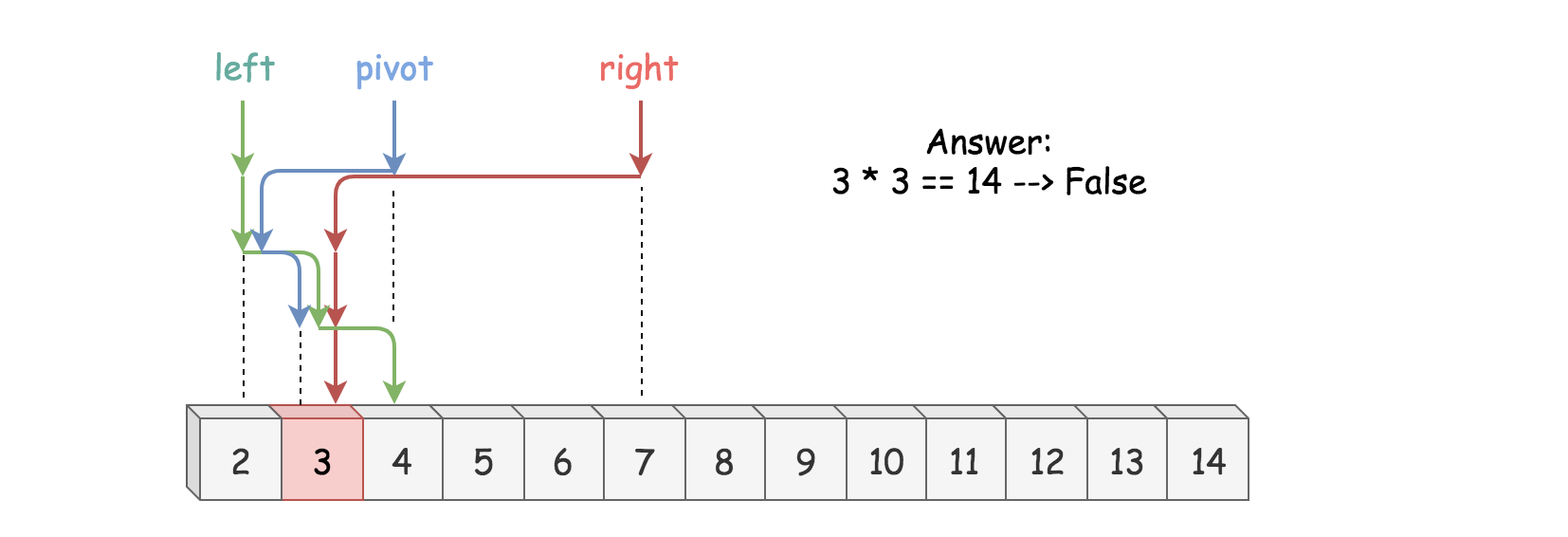
#### **Approach 1: Binary Search**

For \textrm{num} > 2num>2 the square root a*a* is always less than \textrm{num} / 2num/2 and greater than 1: 1 < x < \textrm{num} / 21<*x*<num/2. Since x*x* is an integer, the problem goes down to the search in the sorted set of integer numbers. Binary search is a standard way to proceed in such a situation.

**Algorithm**

* If num < 2, return True.
* Set the left boundary to 2, and the right boundary to num / 2.
* While left <= right:
  + Take x = (left + right) / 2 as a guess. Compute guess\_squared = x \* x and compare it with num:
    - If guess\_squared == num, then the perfect square is right here, return True.
    - If guess\_squared > num, move the right boundary right = x - 1.
    - Otherwise, move the left boundary left = x + 1.
* If we're here, the number is not a prefect square. Return False.

**Implementation**



|  |
| --- |
| class Solution {  public boolean isPerfectSquare(int num) {  if (num < 2) {  return true;  }  long left = 2, right = num / 2, x, guessSquared;  while (left <= right) {  x = left + (right - left) / 2;  guessSquared = x \* x;  if (guessSquared == num) {  return true;  }  if (guessSquared > num) {  right = x - 1;  } else {  left = x + 1;  }  }  return false;  }  } |

**Complexity Analysis**

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

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).

#### **Approach 2: Newton's Method**

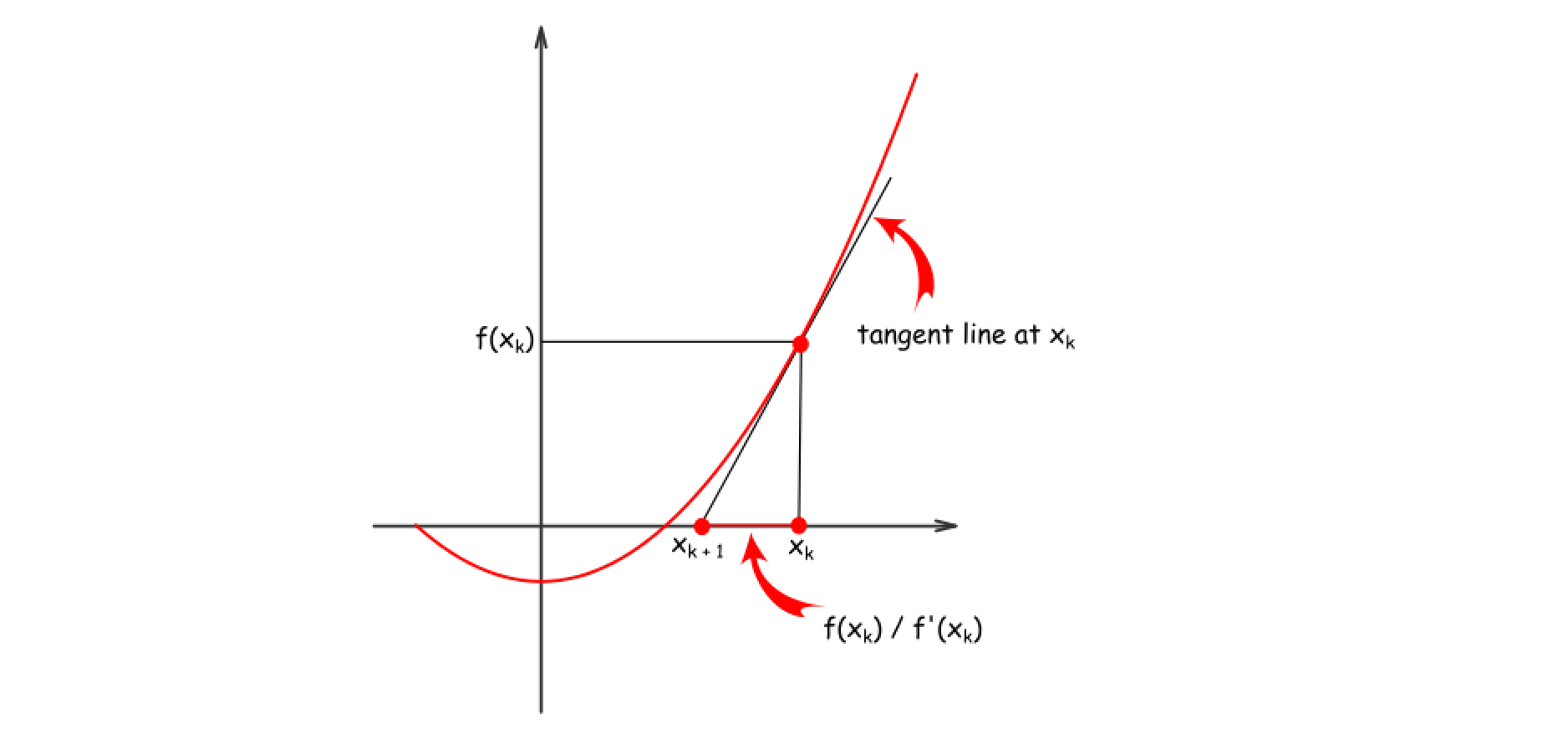
**Newton's Algorithm: How to Figure out the Formula**

Let's do a very rough derivation of Newton's sequence which could be done in two minutes during the interview. Please note that it's more a way to memorize than a strict mathematical proof.

The problem is to find a root of

f(x) = x^2 - \textrm{num} = 0*f*(*x*)=*x*2−num=0

The idea of Newton's algorithm is to start from a seed (= initial guess) and then to compute a root as a sequence of improved guesses.



For example, there is a guess x\_k*xk*​. To compute next guess x\_{k + 1}*xk*+1​, let's approximate f(x\_k)*f*(*xk*​) by its tangent line, that would result in

x\_{k + 1} = x\_k - \frac{f(x\_k)}{f'(x\_k)}*xk*+1​=*xk*​−*f*′(*xk*​)*f*(*xk*​)​

Now use f(x\_k) = x\_k^2 - \textrm{num}*f*(*xk*​)=*xk*2​−num and f'(x\_k) = 2x\_k*f*′(*xk*​)=2*xk*​, and voila the result

x\_{k + 1} = \frac{1}{2}\left(x\_k + \frac{\textrm{num}}{x\_k}\right)*xk*+1​=21​(*xk*​+*xk*​num​)

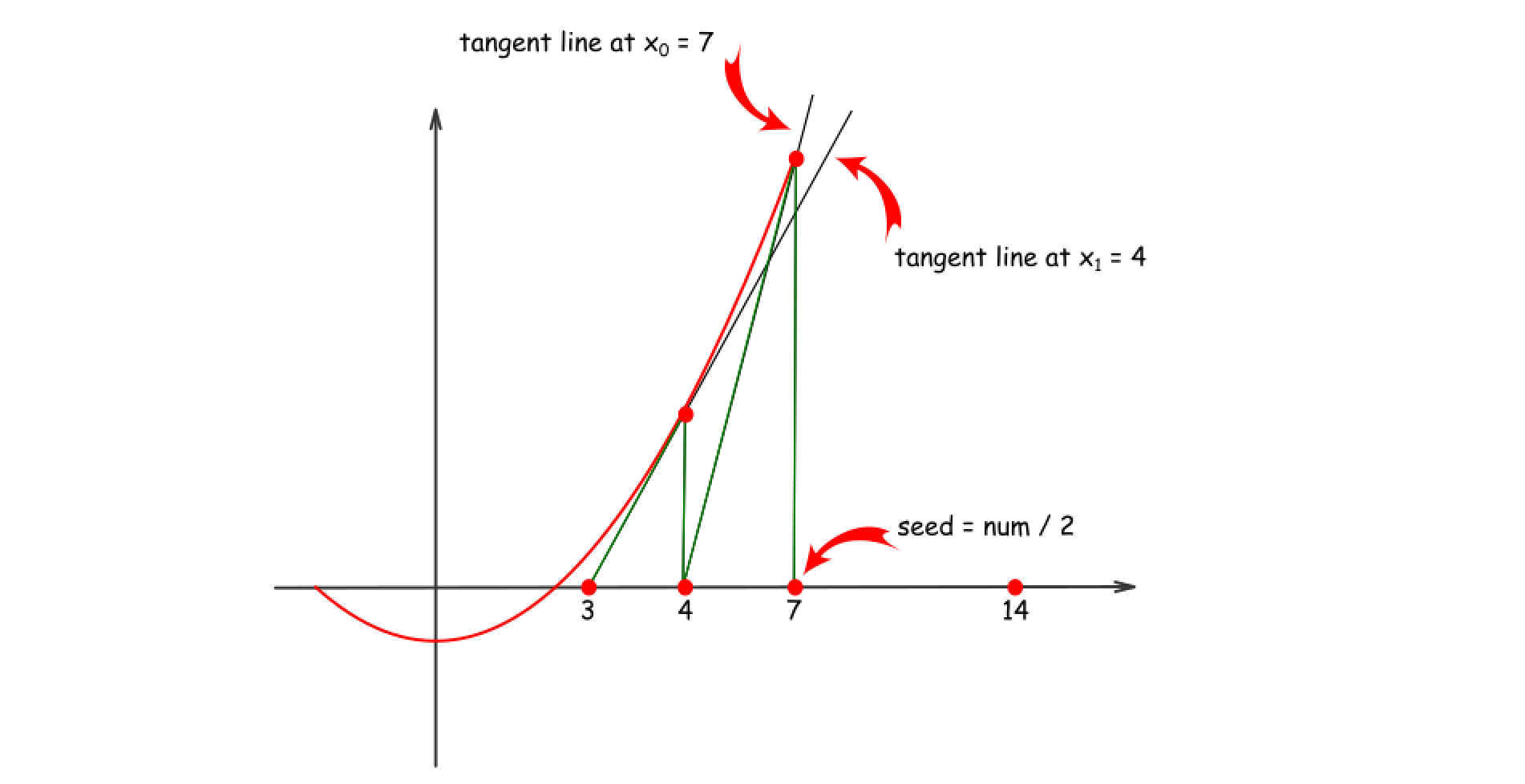
**Choose a seed**

How to choose a seed? Since the function f(x) = x^2 - \textrm{num}*f*(*x*)=*x*2−num is monotonous, the smaller seed the better, so let's take \textrm{num}/2num/2.

**Algorithm**

* Take num / 2 as a seed.
* While x \* x > num, compute the next guess using Newton's method: x = \frac{1}{2}\left(x + \frac{\textrm{num}}{x}\right)*x*=21​(*x*+*x*num​).
* Return x \* x == num

**Implementation**



|  |
| --- |
| class Solution {  public boolean isPerfectSquare(int num) {  if (num < 2) return true;  long x = num / 2;  while (x \* x > num) {  x = (x + num / x) / 2;  }  return (x \* x == num);  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(\log N)O(log*N*) because [guess sequence converges quadratically](https://en.wikipedia.org/wiki/Newton%27s_method#Proof_of_quadratic_convergence_for_Newton's_iterative_method).
* Space complexity : \mathcal{O}(1)O(1).

**Find the Town Judge**

In a town, there are N people labelled from 1 to N.  There is a rumor that one of these people is secretly the town judge.

If the town judge exists, then:

1. The town judge trusts nobody.
2. Everybody (except for the town judge) trusts the town judge.
3. There is exactly one person that satisfies properties 1 and 2.

You are given trust, an array of pairs trust[i] = [a, b] representing that the person labelled a trusts the person labelled b.

If the town judge exists and can be identified, return the label of the town judge.  Otherwise, return -1.

**Example 1:**

**Input:** N = 2, trust = [[1,2]]

**Output:** 2

**Example 2:**

**Input:** N = 3, trust = [[1,3],[2,3]]

**Output:** 3

**Example 3:**

**Input:** N = 3, trust = [[1,3],[2,3],[3,1]]

**Output:** -1

**Example 4:**

**Input:** N = 3, trust = [[1,2],[2,3]]

**Output:** -1

**Example 5:**

**Input:** N = 4, trust = [[1,3],[1,4],[2,3],[2,4],[4,3]]

**Output:** 3

**Constraints:**

* 1 <= N <= 1000
* 0 <= trust.length <= 10^4
* trust[i].length == 2
* trust[i] are all different
* trust[i][0] != trust[i][1]
* 1 <= trust[i][0], trust[i][1] <= N

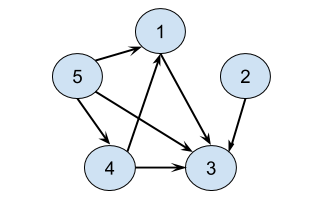
## Solution

#### **Approach 1: Two Arrays**

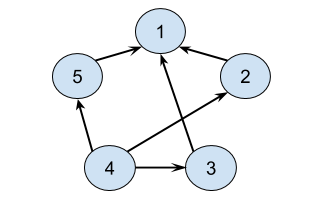
**Intuition**

The trust relationships form a graph. Each trust pair, [a, b] represents a **directed edge** going from a to b.

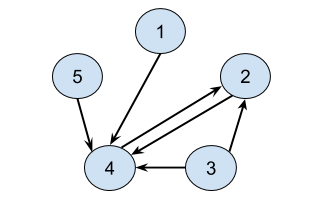
For example, with N = 5 and trust = [[1,3],[2,3],[4,3],[4,1],[5,3],[5,1],[5,4]], we get the following graph. Who is the town judge?



What about this example, with trust = [[2,1],[3,1],[4,2],[4,3],[4,5],[5,1]]?



And what about this example, with trust = [[1,4],[2,4],[3,2],[3,4],[4,2],[5,4]]?



For the first example, the town judge is 3, because they are trusted by all four other people; 1, 2, 4, and 5, but they don't trust anybody themselves.

For the second example, there is no town judge. Nobody is trusted by everybody else.

For the third example, there is also no town judge. While 4 is trusted by everybody, 4 themselves trusts 2. Therefore, 4 can't be the town judge.

Some people would be tempted to launch straight into converting the input into a standard graph format, for example an adjacency list (or worse, a complicated linked structure), as soon as they make the observation that this has something to do with graphs. Then, they'll go back to trying to solve the actual problem. But as we'll see, it's better to start by looking really closely at the problem, as there's a way we can solve it without making a graph.

In graph theory, we say the **outdegree** of a vertex (person) is the number of directed edges going out of it. For this graph, the outdegree of the vertex represents the number of other people that person trusts.

Likewise, we say that the **indegree** of a vertex (person) is the number of directed edges going into it. So here, it represents the number of people trusted by that person.

We can define the town judge in terms of **indegree** and **outdegree**.

The town judge has an outdegree of 0 and an indegree of N - 1 because they trust nobody, and everybody trusts them (except themselves).

Therefore, this problem simplifies to calculating the **indegree** and **outdegree** for each person and then checking whether or not any of them meet the criteria of the town judge.

We can calculate the indegrees and outdegrees for everybody, using a single loop over the input trust array. We'll write the results into two arrays.

int[] indegrees = new int[N + 1];

int[] outdegrees = new int[N + 1];

for (int[] relation : trust) {

outdegrees[relation[0]]++;

indegrees[relation[1]]++;

}

Then, we can simply loop over the people (numbered from 1 to N) and check whether or not they meet the town judge criteria.

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

if (indegrees[i] == N - 1 && outdegrees[i] == 0) {

return i;

}

return -1;

}

One optimization we can make is to observe that it is impossible for there to be a town judge if there are not at least N - 1 edges in the trust array. This is because a town judge must have N - 1 in-going edges, and so if there aren't at least N - 1 edges in total, then it is impossible to meet this requirement. This observation will also be very useful when we're reasoning about the time complexity.

If trust.length < N - 1, then we can immediately return -1.

**Algorithm**

|  |
| --- |
| public int findJudge(int N, int[][] trust) {    if (trust.length < N - 1) {  return -1;  }    int[] indegrees = new int[N + 1];  int[] outdegrees = new int[N + 1];  for (int[] relation : trust) {  outdegrees[relation[0]]++;  indegrees[relation[1]]++;  }  for (int i = 1; i <= N; i++) {  if (indegrees[i] == N - 1 && outdegrees[i] == 0) {  return i;  }  }  return -1;  } |

**Complexity Analysis**

Let N*N* be the number of people, and E*E* be the number of edges (trust relationships).

* Time Complexity : O(E)*O*(*E*).

We loop over the trust list once. The cost of doing this is O(E)*O*(*E*).

We then loop over the people. The cost of doing this is O(N)*O*(*N*).

Going by this, it now looks this is one those many graph problems where the cost is O(\max(N, E) = O(N + E)*O*(max(*N*,*E*)=*O*(*N*+*E*). After all, we don't know whether E*E* or N*N* is the bigger one, right?

However, remember how we terminate early if E < N - 1*E*<*N*−1? This means that in the best case, the time complexity is O(1)*O*(1). And in the worst case, we know that E ≥ N - 1*E*≥*N*−1. For the purpose of big-oh notation, we ignore the constant of 11. Therefore, in the worst case, E*E* has to be bigger, and so we can simply drop the N*N*, leaving O(E)*O*(*E*).

* Space Complexity : O(N)*O*(*N*).

We allocated 2 arrays; one for the indegrees and the other for the outdegrees. Each was of length N + 1. Because in big-oh notation we drop constants, this leaves us with O(N)*O*(*N*).

***This last note is provided more for interest than for interview preparation.*** A variant of the approach is to use a HashMaps instead of Arrays. That way, you'll only need to store indegrees and outdegrees that are non-zero. This will have no impact on the time complexity, because we still need to look at the entire input Array. It also has no impact on the worst case space complexity, because when a town judge exists, all the other N - 1 people have an outdegree of at least 1 (from their trust of the town judge). In some cases where E ≥ N - 1*E*≥*N*−1 but there is no town judge, some memory might be saved, with a best case of O(\sqrt{E}\,)*O*(*E*​). This represents the situation of the number of unique people present in the trust Array being minimized (beyond an easy-level question interview, don't panic!). With the overhead of a HashMap, there's probably no gain of using one over an Array for this problem.

#### **Approach 2: One Array**

**Intuition**

Just to be clear, there's nothing wrong with Approach 1. If you got it, you're doing great! Approach 2 is a little more subtle. Coming up with these kinds of approaches is something you'll learn to do with experience.

We don't need separate arrays for indegree and outdegree. We can instead build a single Array with the result of indegree - outdegree for each person. In other words, we'll +1 to their "score" for each person they are trusted by, and -1 from their "score" for each person they trust. Therefore, for a person to maximize their "score", they should be trusted by as many people as possible, and trust as few people as possible.

The maximum indegree is N - 1. This represents everybody trusting the person (except for themselves, they cannot trust themselves). The minimum indegree is 0. This represents not trusting anybody. Therefore, the maximum value for indegree - outdegree is (N - 1) - 0 = N - 1. These values also happen to be the definition of the town judge!

The town judge is the only person who could possibly have indegree - outdegree equal to N - 1.

**Algorithm**

Each person gains 1 "point" for each person they are trusted by, and loses 1 "point" for each person they trust. Then at the end, the town judge, if they exist, must be the person with N - 1 "points".

|  |
| --- |
| public int findJudge(int N, int[][] trust) {    if (trust.length < N - 1) {  return -1;  }    int[] trustScores = new int[N + 1];  for (int[] relation : trust) {  trustScores[relation[0]]--;  trustScores[relation[1]]++;  }    for (int i = 1; i <= N; i++) {  if (trustScores[i] == N - 1) {  return i;  }  }  return -1;  } |

**Complexity Analysis**

Recall that N*N* is the number of people, and E*E* is the number of edges (trust relationships).

* Time Complexity : O(E)*O*(*E*).

Same as above. We still need to loop through the E*E* edges in trust, and the argument about the relationship between N*N* and E*E* still applies.

* Space Complexity : O(N)*O*(*N*).

Same as above. We're still allocating an array of length N*N*.

#### **Bonus**

**Can There Be More Than One Town Judge?**

In the problem description, we're told that iff there is a town judge, there'll only be one town judge.

It's likely that not all interviewers would tell you directly that there can only be one town judge. If you asked them whether or not there could be more than one town judge, they might ask you if there could be. And the answer is... it's impossible!

If there were two town judges, then they would have to trust each other, otherwise we'd have a town judge not trusted by everybody. But this doesn't work, because town judges aren't supposed to trust anybody. Therefore, we know there can be at most one town judge.

**A Related Question**

Secondly, for premium members, there is a similar question on Leetcode, called [Find the Celebrity](https://leetcode.com/articles/find-the-celebrity/). You need to do the same thing—find a person who has an indegree of N - 1 and an outdegree of 0. However, the input format is a bit different.

It's well worth a look at. A seemingly small difference (the input format) completely changes what the optimal algorithm to solve it is. Interestingly though, the optimal algorithm for that problem can also be used here. The only difference is that there, it has a cost of O(N)*O*(*N*), but here it has a cost of O(E)*O*(*E*). Try and figure out why once you've solved both problems. It's a really nice example of cost analysis with graphs.

**Single Element in a Sorted Array**

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once. Find this single element that appears only once.

**Follow up:** Your solution should run in O(log n) time and O(1) space.

**Example 1:**

**Input:** nums = [1,1,2,3,3,4,4,8,8]

**Output:** 2

**Example 2:**

**Input:** nums = [3,3,7,7,10,11,11]

**Output:** 10

**Constraints:**

* 1 <= nums.length <= 10^5
* 0 <= nums[i] <= 10^5

## Solution

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

**Intuition**

We can use a linear search to check every element in the array until we find the single element.

**Algorithm**

Starting with the first element, we iterate over every 2nd element, checking whether or not the next element is the same as the current. If it's not, then we know this must be the single element.

If we get as far as the last element, we know that it must be the single element. We need to treat it as a special case after the loop, because otherwise we'll be going over the end of the array.

|  |
| --- |
| class Solution {  public int singleNonDuplicate(int[] nums) {  for (int i = 0; i < nums.length - 1; i+=2) {  if (nums[i] != nums[i + 1]) {  return nums[i];  }  }  return nums[nums.length - 1];  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). For linear search, we are looking at every element in the array once.
* Space complexity : O(1)*O*(1). We are only using constant extra space.

While this approach will work, the question tells us we need a O(\log n)*O*(log*n*) solution. Therefore, this solution isn't good enough.

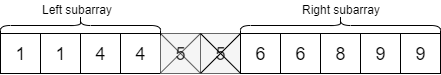
#### **Approach 2: Binary Search**

**Intuition**

It makes sense to try and convert the linear search into a binary search. In order to use binary search, we need to be able to look at the middle item and then determine whether the solution is the middle item, or to the left, or to the right. The key observation to make is that the starting array must always have an odd number of elements (be odd-lengthed), because it has one element appearing once, and all the other elements appearing twice.

An array with the elements 1, 1, 4, 4, 5, 5, 6, 6, 8, 9, 9

Here is what happens when we remove a pair from the center. We are left with a left subarray and a right subarray.



Like the original array, the subarray containing the single element must be odd-lengthed. The subarray not containing it must be even-lengthed. So by taking a pair out of the middle and then calculating which side is now odd-lengthed, we have the information needed for binary search.

**Algorithm**

We start by setting lo and hi to be the lowest and highest index (inclusive) of the array, and then iteratively halve the array until we find the single element or until there is only one element left. We know that if there is only one element in the search space, it must be the single element, so should terminate the search.

On each loop iteration, we find mid, and determine the odd/ evenness of the sides and save it in a variable called halvesAreEven. By then looking at which half the middle element's partner is in (either last element in the left subarray or first element in the right subarray), we can decide which side is now (or remained) odd-lengthed and set lo and hi to cover the part of the array we now know the single element must be in.

The trickiest part is ensuring we update lo and hi correctly based on the values of mid and halvesAreEven. These diagrams should help you understand the cases. When solving problems like this, it's often good to draw a diagram and think really carefully about it to avoid off-by-one errors. Avoid using a guess and check approach.

Case 1: Mid’s partner is to the right, and the halves were originally even.

The right side becomes odd-lengthed because we removed mid's partner from it. We need to set lo to mid + 2 so that the remaining array is the part above mid's partner.

fig

Case 2: Mid’s partner is to the right, and the halves were originally odd.

The left side remains odd-lengthed. We need to set hi to mid - 1 so that the remaining array is the part below mid.

fig

Case 3: Mid’s partner is to the left, and the halves were originally even.

The left side becomes odd-lengthed because we removed mid's partner from it. We need to set hi to mid - 2 so that the remaining array is the part below mid's partner.

fig

Case 4: Mid’s partner is to the left, and the halves were originally odd.

The right side remains odd-lengthed. We need to set lo to mid + 1 so that the remaining array is the part above mid.

fig

|  |
| --- |
| class Solution {  public int singleNonDuplicate(int[] nums) {  int lo = 0;  int hi = nums.length - 1;  while (lo < hi) {  int mid = lo + (hi - lo) / 2;  boolean halvesAreEven = (hi - mid) % 2 == 0;  if (nums[mid + 1] == nums[mid]) {  if (halvesAreEven) {  lo = mid + 2;  } else {  hi = mid - 1;  }  } else if (nums[mid - 1] == nums[mid]) {  if (halvesAreEven) {  hi = mid - 2;  } else {  lo = mid + 1;  }  } else {  return nums[mid];  }  }  return nums[lo];  }  } |

Another interesting observation you might have made is that this algorithm will still work even if the array isn't fully sorted. As long as pairs are always grouped together in the array (for example, [10, 10, 4, 4, 7, 11, 11, 12, 12, 2, 2]), it doesn't matter what order they're in. Binary search worked for this problem because we knew the subarray with the single number is always odd-lengthed, not because the array was fully sorted numerically. We commonly call this an invariant, something that is always true (i.e. "The array containing the single element is always odd-lengthed"). Be on the lookout for invariants like this when solving array problems, as binary search is very flexibile!

**Complexity Analysis**

* Time complexity : O(\log n)*O*(log*n*). On each iteration of the loop, we're halving the number of items we still need to search.
* Space complexity : O(1)*O*(1). We are only using constant space to keep track of where we are in the search.

#### **Approach 3: Binary Search on Evens Indexes Only**

It turns out that we only need to binary search on the even indexes. This approach is more elegant than the last, although both are good solutions.

**Intuition**

The single element is at the first even index not followed by its pair. We used this property in the linear search algorithm, where we iterated over all of the even indexes until we encountered the first one not followed by its pair.

Instead of linear searching for this index though, we can binary search for it.

After the single element, the pattern changes to being odd indexes followed by their pair. This means that the single element (an even index) and all elements after it are even indexes not followed by their pair. Therefore, given any even index in the array, we can easily determine whether the single element is to the left or to the right.

**Algorithm**

We need to set up the binary search variables and loop so that we are only considering even indexes. The last index of an odd-lengthed array is always even, so we can set lo and hi to be the start and end of the array.

We need to make sure our mid index is even. We can do this by dividing lo and hi in the usual way, but then decrementing it by 1 if it is odd. This also ensures that if we have an even number of even indexes to search, that we are getting the lower middle (incrementing by 1 here would not work, it'd lead to an infinite loop as the search space would not be reduced in some cases).

Then we check whether or not the mid index is the same as the one after it.

* If it is, then we know that mid is not the single element, and that the single element must be at an even index after mid. Therefore, we set lo to be mid + 2. It is +2 rather than the usual +1 because we want it to point at an even index.
* If it is not, then we know that the single element is either at mid, or at some index before mid. Therefore, we set hi to be mid.

Once lo == hi, the search space is down to 1 element, and this must be the single element, so we return it.

|  |
| --- |
| class Solution {  public int singleNonDuplicate(int[] nums) {  int lo = 0;  int hi = nums.length - 1;  while (lo < hi) {  int mid = lo + (hi - lo) / 2;  if (mid % 2 == 1) mid--;  if (nums[mid] == nums[mid + 1]) {  lo = mid + 2;  } else {  hi = mid;  }  }  return nums[lo];  }  } |

**Complexity Analysis**

* Time complexity : O(\log \frac{n}{2}) = O(\log n)*O*(log2*n*​)=*O*(log*n*). Same as the binary search above, except we are only binary searching half the elements, rather than all of them.
* Space complexity : O(1)*O*(1). Same as the other approaches. We are only using constant space to keep track of where we are in the search.

**Remove K Digits**

Given a non-negative integer *num* represented as a string, remove *k* digits from the number so that the new number is the smallest possible.

**Note:**

* The length of *num* is less than 10002 and will be ≥ *k*.
* The given *num* does not contain any leading zero.

**Example 1:**

Input: num = "1432219", k = 3

Output: "1219"

Explanation: Remove the three digits 4, 3, and 2 to form the new number 1219 which is the smallest.

**Example 2:**

Input: num = "10200", k = 1

Output: "200"

Explanation: Remove the leading 1 and the number is 200. Note that the output must not contain leading zeroes.

**Example 3:**

Input: num = "10", k = 2

Output: "0"

Explanation: Remove all the digits from the number and it is left with nothing which is 0.

## Solution

#### **Approach 1: Brute-force [Time Limit Exceeded]**

**Intuition**

At the first glance, one of the first intuitions that might come to one's mind is to enumerate all the possible combinations and find the minimal number among them, i.e. brute-force.

Though after a small moment of reflection, we would easily rule it out. We could name a few reasons. The major caveat is that the algorithm would have an **exponential time complexity**, since we need to enumerate the combinations of selecting k*k* numbers out of a list of n*n*, i.e. C\_{n}^{k}*Cnk*​. Even for a trial example, the algorithm could run out of the time limit.

Apart from the complexity issue, another technical issue that one needs to solve in the above brute-force approach, is to compare the values of two digit strings. As naive as it sounds, we could convert the digit string to the numerical value. Soon one would realize that this method does not scale. For an unsigned 32 bit-integer, the maximum value it can hold is a number with 10 digits (i.e. 4,294,967,295). We can expect there are plenty of test cases with strings of hundreds of digits.

One would argue that for comparison, we don't need to convert the digit string to its numeric value, but simply compare the sequence of digits one by one from left to right. Indeed, it would work.

But then, if we look at the overall problem again, it seems that there should be some ***deterministic way*** to construct the solution, without the need of exhausting all possible solutions.

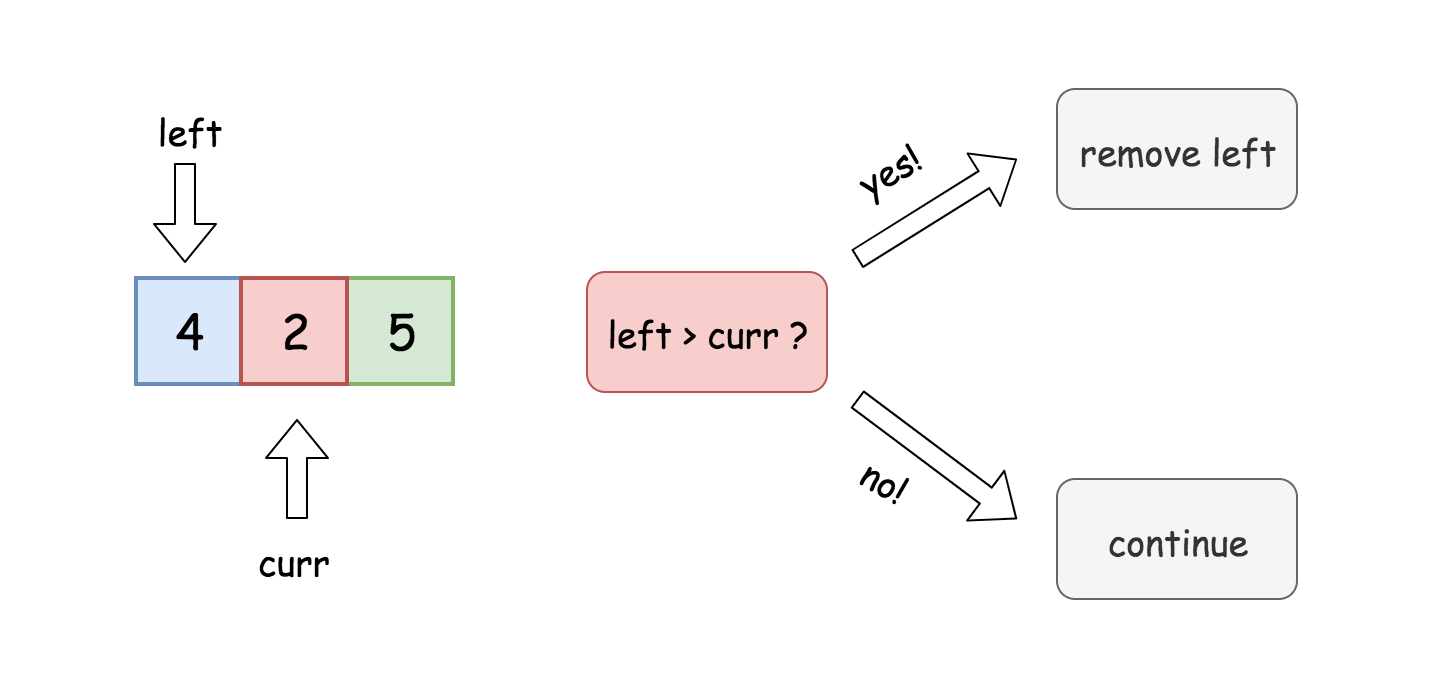
#### **Approach 2: Greedy with Stack**

**Intuition**

We've got a hint while entertaining the idea of brute-force, that given two sequences of digit of the same length, it is the **leftmost** **distinct** digits that determine the superior of the two numbers, e.g. for A = 1axxx, B = 1bxxx, if the digits a > b, then A > B.

With this insight, the first intuition we got for our problem is that we should iterate from the left to right, when removing the digits. The more a digit to the left-hand side, the more weight it carries.

Now that we fix on the order of the iteration, it is critical to come up some **criteria** on how we eliminate digits, in order to obtain the minimum value.



Let us start from a simple example. Given a sequence of digits, e.g. 425, if we are asked to remove just one digit, then from left to right, we have the candidates as 4, 2 and 5. And we compare each digit with its left neighbor. Starting from 2, which is less than its left neighbor 4. At this very moment, we are sure that we should remove the digit *4*. Because the consequence of not doing so is that we won't obtain the minimum number, no matter what we do subsequently.

Imagine if we keep the digit 4, then all the possible solutions are lead with the digit 4 (i.e. 42, 45). While in one of the opposite cases, e.g. removing 4 and keeping 2, we have solutions lead with 2 (i.e. 25), which is obviously less than any of the solutions of keeping the digit 4.

We could summarize the above scenario of removing a digit, as a rule below:

Given a sequence of digits [D\_1D\_2D\_3...D\_n][*D*1​*D*2​*D*3​...*Dn*​], if the digit D\_2*D*2​ is less than its left neighbor D\_1*D*1​, then we should remove the left neighbor (D\_1*D*1​) in order to obtain the minimum result.

**Algorithm**

Believe it or not, the above rule is the only key needed to solve the problem. It clearly defines a condition on which we can remove a digit without a doubt. By removing the digits one by one, we are steadily approaching the optimal solution step by step. Now, it might ring a bell, to one of the popular algorithmic paradigms -− **Greedy**.

Indeed, the problem could be solved with the greedy algorithm. The above rule clarifies the essential logic on how we can approach the final solution. Once we remove a digit from the sequence, the remaining digits forms a new problem where we can continue to apply the rule.

One might notice that, there could be some cases where the condition to apply the rule does not hold for any of the digits. To put it in another word, in those cases, we would have a ***monotonic increasing sequence***, i.e. each digit is bigger than its previous digit. In this scenario, we simply remove the pending large digits, again, ***greedily***. We skip the rigorous proof here, and claim that the solution obtained by the above measure is indeed the optimal one.

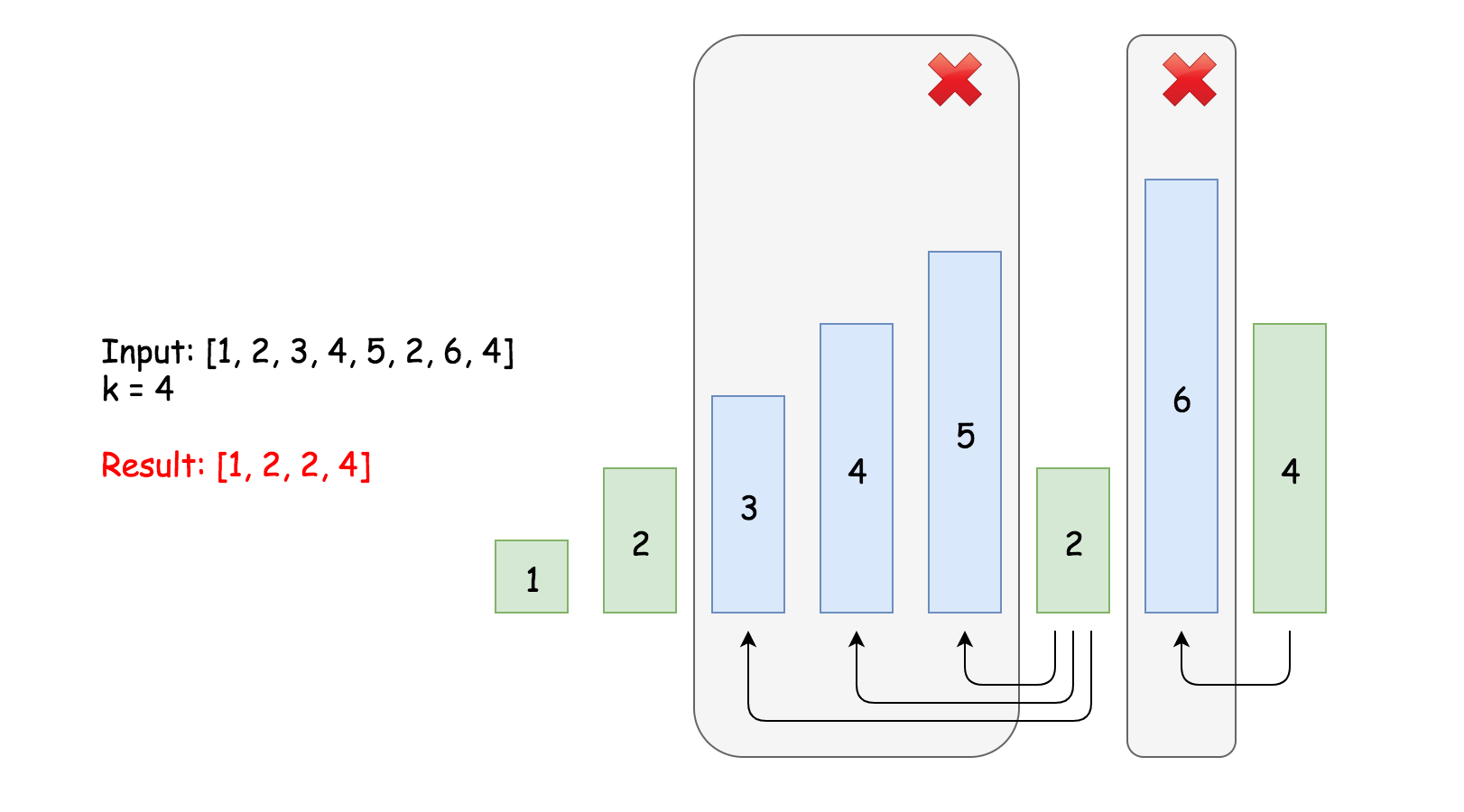
On the other hand, we did provide a **proof by contradiction**, with the simple example of 425 in the Intuition section, that by repeatedly applying the rule we would obtain the optimal solution.

**Implementation**

One could implement the above algorithm with the help of the **stack** data structure. We use a stack to hold the digits that we would keep at the end.

Iterating the sequence of digits from left to right, the main loop can be broken down as follows:

* 1). for each digit, if the digit is less than the top of the stack, i.e. the left neighbor of the digit, then we pop the stack, i.e. removing the left neighbor. At the end, we push the digit to the stack.
* 2). we repeat the above step (1) until any of the conditions does not hold any more, e.g. the stack is empty (no more digits left). or in another case, we have already removed k digits, therefore mission accomplished.



We demonstrate how the algorithm works in the above graph. Given the input sequence [1, 2, 3, 4, 5, 2, 6, 4] and the number of digits to remove k=4, the rule is triggered for the first time at the digit of 5. Once we remove the digit 5, the rule is triggered again at the digit 4 till the digit 3. And then later, at the digit 6, the rule is triggered as well.

Out of the above main loop, we need to handle some corner cases to make the solution more complete.

* case 1). when we get out of the main loop, we removed m digits, which is less than asked, i.e.(m < k). In the extreme case, we would not remove any digit for the monotonic increasing sequence in the loop, i.e. m==0. In this case, we just need to remove the additional k-m digits from the tail of the sequence.
* case 2). once we remove all the k digits from the sequence, there could be some leading zeros left. To format the final number, we need to strip off those leading zeros.
* case 3). we might end up removing all numbers from the sequence. In this case, we should return zero, instead of empty string.

Here are some sample implementations.

|  |
| --- |
| class Solution {  public String removeKdigits(String num, int k) {  LinkedList<Character> stack = new LinkedList<Character>();    for(char digit : num.toCharArray()) {  while(stack.size() > 0 && k > 0 && stack.peekLast() > digit) {  stack.removeLast();  k -= 1;  }  stack.addLast(digit);  }    /\* remove the remaining digits from the tail. \*/  for(int i=0; i<k; ++i) {  stack.removeLast();  }    // build the final string, while removing the leading zeros.  StringBuilder ret = new StringBuilder();  boolean leadingZero = true;  for(char digit: stack) {  if(leadingZero && digit == '0') continue;  leadingZero = false;  ret.append(digit);  }    /\* return the final string \*/  if (ret.length() == 0) return "0";  return ret.toString();  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*). Although there are nested loops, the inner loop is bounded to be run at most k*k* times globally. Together with the outer loop, we have the exact (N + k)(*N*+*k*) number of operations. Since 0 < k \le N0<*k*≤*N*, the time complexity of the main loop is bounded within 2N2*N*. For the logic outside the main loop, it is clear to see that their time complexity is \mathcal{O}(N)O(*N*). As a result, the overall time complexity of the algorithm is \mathcal{O}(N)O(*N*).
* Space complexity : \mathcal{O}(N)O(*N*). We have a stack which would hold all the input digits in the worst case.

**Maximum Sum Circular Subarray**

Given a **circular array** **C** of integers represented by A, find the maximum possible sum of a non-empty subarray of **C**.

Here, a circular array means the end of the array connects to the beginning of the array.  (Formally, C[i] = A[i] when 0 <= i < A.length, and C[i+A.length] = C[i] when i >= 0.)

Also, a subarray may only include each element of the fixed buffer A at most once.  (Formally, for a subarray C[i], C[i+1], ..., C[j], there does not exist i <= k1, k2 <= j with k1 % A.length = k2 % A.length.)

**Example 1:**

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

**Output:** 3

**Explanation:** Subarray [3] has maximum sum 3

**Example 2:**

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

**Output:** 10

**Explanation:** Subarray [5,5] has maximum sum 5 + 5 = 10

**Example 3:**

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

**Output:** 4

**Explanation:** Subarray [2,-1,3] has maximum sum 2 + (-1) + 3 = 4

**Example 4:**

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

**Output:** 3

**Explanation:** Subarray [3] and [3,-2,2] both have maximum sum 3

**Example 5:**

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

**Output:** -1

**Explanation:** Subarray [-1] has maximum sum -1

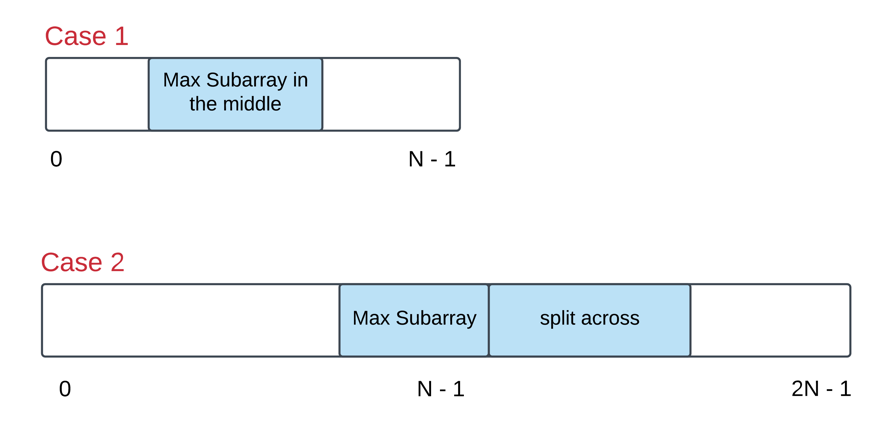
**Note:**

1. -30000 <= A[i] <= 30000
2. 1 <= A.length <= 30000

 Hide Hint #1

For those of you who are familiar with the **Kadane's algorithm**, think in terms of that. For the newbies, Kadane's algorithm is used to finding the maximum sum subarray from a given array. This problem is a twist on that idea and it is advisable to read up on that algorithm first before starting this problem. Unless you already have a great algorithm brewing up in your mind in which case, go right ahead!

   Hide Hint #2

What is an alternate way of representing a circular array so that it appears to be a straight array? Essentially, there are two cases of this problem that we need to take care of. Let's look at the figure below to understand those two cases:  


   Hide Hint #3

The first case can be handled by the good old Kadane's algorithm. However, is there a smarter way of going about handling the second case as well?

## Solution

#### **Notes and A Primer on Kadane's Algorithm**

**About the Approaches**

In both Approach 1 and Approach 2, "grindy" solutions are presented that require less insight, but may be more intuitive to those with a solid grasp of the techniques in those approaches. Without prior experience, these approaches would be very challenging to emulate.

Approaches 3 and 4 are much easier to implement, but require some insight.

**Explanation of Kadane's Algorithm**

To understand the solutions in this article, we need some familiarity with Kadane's algorithm. In this section, we will explain the core idea behind it.

For a given array A, Kadane's algorithm can be used to find the maximum sum of the subarrays of A. Here, we only consider non-empty subarrays.

Kadane's algorithm is based on dynamic programming. Let dp[j] be the maximum sum of a subarray that ends in A[j]. That is,

\text{dp}[j] = \max\limits\_i (A[i] + A[i+1] + \cdots + A[j])dp[*j*]=*i*max​(*A*[*i*]+*A*[*i*+1]+⋯+*A*[*j*])

Then, a subarray ending in j+1 (such as A[i], A[i+1] + ... + A[j+1]) maximizes the A[i] + ... + A[j] part of the sum by being equal to dp[j] if it is non-empty, and 0 if it is. Thus, we have the recurrence:

\text{dp}[j+1] = A[j+1] + \max(\text{dp}[j], 0)dp[*j*+1]=*A*[*j*+1]+max(dp[*j*],0)

Since a subarray must end somewhere, \max\limits\_j dp[j]*j*max​*dp*[*j*] must be the desired answer.

To compute dp efficiently, Kadane's algorithm is usually written in the form that reduces space complexity. We maintain two variables: ans as \max\limits\_j dp[j]*j*max​*dp*[*j*], and cur as dp[j]*dp*[*j*]; and update them as j*j* iterates from 00 to A\text{.length} - 1*A*.length−1.

Then, Kadane's algorithm is given by the following psuedocode:

#Kadane's algorithm

ans = cur = None

for x in A:

cur = x + max(cur, 0)

ans = max(ans, cur)

return ans

#### **Approach 1: Next Array**

**Intuition and Algorithm**

Subarrays of circular arrays can be classified as either as one-interval subarrays, or two-interval subarrays, depending on how many intervals of the fixed-size buffer A are required to represent them.

For example, if A = [0, 1, 2, 3, 4, 5, 6] is the underlying buffer of our circular array, we could represent the subarray [2, 3, 4] as one interval [2, 4][2,4], but we would represent the subarray [5, 6, 0, 1] as two intervals [5, 6], [0, 1][5,6],[0,1].

Using Kadane's algorithm, we know how to get the maximum of one-interval subarrays, so it only remains to consider two-interval subarrays.

Let's say the intervals are [0, i], [j, A\text{.length} - 1][0,*i*],[*j*,*A*.length−1]. Let's try to compute the i-th candidate: the largest possible sum of a two-interval subarray for a given i*i*. Computing the [0, i][0,*i*] part of the sum is easy. Let's write

T\_j = A[j] + A[j+1] + \cdots + A[A\text{.length} - 1]*Tj*​=*A*[*j*]+*A*[*j*+1]+⋯+*A*[*A*.length−1]

and

R\_j = \max\limits\_{k \geq j} T\_k*Rj*​=*k*≥*j*max​*Tk*​

so that the desired i-th candidate is:

(A[0] + A[1] + \cdots + A[i]) + R\_{i+2}(*A*[0]+*A*[1]+⋯+*A*[*i*])+*Ri*+2​

Since we can compute T\_j*Tj*​ and R\_j*Rj*​ in linear time, the answer is straightforward after this setup.

|  |
| --- |
| class Solution {  public int maxSubarraySumCircular(int[] A) {  int N = A.length;  int ans = A[0], cur = A[0];  for (int i = 1; i < N; ++i) {  cur = A[i] + Math.max(cur, 0);  ans = Math.max(ans, cur);  }  // ans is the answer for 1-interval subarrays.  // Now, let's consider all 2-interval subarrays.  // For each i, we want to know  // the maximum of sum(A[j:]) with j >= i+2  // rightsums[i] = A[i] + A[i+1] + ... + A[N-1]  int[] rightsums = new int[N];  rightsums[N-1] = A[N-1];  for (int i = N-2; i >= 0; --i)  rightsums[i] = rightsums[i+1] + A[i];  // maxright[i] = max\_{j >= i} rightsums[j]  int[] maxright = new int[N];  maxright[N-1] = A[N-1];  for (int i = N-2; i >= 0; --i)  maxright[i] = Math.max(maxright[i+1], rightsums[i]);  int leftsum = 0;  for (int i = 0; i < N-2; ++i) {  leftsum += A[i];  ans = Math.max(ans, leftsum + maxright[i+2]);  }  return ans;  }  } |

**Complexity Analysis**

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

#### **Approach 2: Prefix Sums + Monoqueue**

**Intuition**

First, we can frame the problem as a problem on a fixed array.

We can consider any subarray of the circular array with buffer A, to be a subarray of the fixed array A+A.

For example, if A = [0,1,2,3,4,5] represents a circular array, then the subarray [4,5,0,1] is also a subarray of fixed array [0,1,2,3,4,5,0,1,2,3,4,5]. Let B = A+A be this fixed array.

Now say N = A\text{.length}*N*=*A*.length, and consider the prefix sums

P\_k = B[0] + B[1] + \cdots + B[k-1]*Pk*​=*B*[0]+*B*[1]+⋯+*B*[*k*−1]

Then, we want the largest P\_j - P\_i*Pj*​−*Pi*​ where j - i \leq N*j*−*i*≤*N*.

Now, consider the j-th candidate answer: the best possible P\_j - P\_i*Pj*​−*Pi*​ for a fixed j*j*. We want the i*i* so that P\_i*Pi*​ is smallest, with j - N \leq i < j*j*−*N*≤*i*<*j*. Let's call this the optimal i for the j-th candidate answer. We can use a monoqueue to manage this.

**Algorithm**

Iterate forwards through j*j*, computing the j*j*-th candidate answer at each step. We'll maintain a queue of potentially optimal i*i*'s.

The main idea is that if i\_1 < i\_2*i*1​<*i*2​ and P\_{i\_1} \geq P\_{i\_2}*Pi*1​​≥*Pi*2​​, then we don't need to remember i\_1*i*1​ anymore.

Please see the inline comments for more algorithmic details about managing the queue.

|  |
| --- |
| class Solution {  public int maxSubarraySumCircular(int[] A) {  int N = A.length;  // Compute P[j] = B[0] + B[1] + ... + B[j-1]  // for fixed array B = A+A  int[] P = new int[2\*N+1];  for (int i = 0; i < 2\*N; ++i)  P[i+1] = P[i] + A[i % N];  // Want largest P[j] - P[i] with 1 <= j-i <= N  // For each j, want smallest P[i] with i >= j-N  int ans = A[0];  // deque: i's, increasing by P[i]  Deque<Integer> deque = new ArrayDeque();  deque.offer(0);  for (int j = 1; j <= 2\*N; ++j) {  // If the smallest i is too small, remove it.  if (deque.peekFirst() < j-N)  deque.pollFirst();  // The optimal i is deque[0], for cand. answer P[j] - P[i].  ans = Math.max(ans, P[j] - P[deque.peekFirst()]);  // Remove any i1's with P[i2] <= P[i1].  while (!deque.isEmpty() && P[j] <= P[deque.peekLast()])  deque.pollLast();  deque.offerLast(j);  }  return ans;  }  } |

**Complexity Analysis**

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

#### **Approach 3: Kadane's (Sign Variant)**

**Intuition and Algorithm**

As in Approach 1, subarrays of circular arrays can be classified as either as one-interval subarrays, or two-interval subarrays.

Using Kadane's algorithm kadane for finding the maximum sum of non-empty subarrays, the answer for one-interval subarrays is kadane(A).

Now, let N = A\text{.length}*N*=*A*.length. For a two-interval subarray like:

(A\_0 + A\_1 + \cdots + A\_i) + (A\_j + A\_{j+1} + \cdots + A\_{N - 1})(*A*0​+*A*1​+⋯+*Ai*​)+(*Aj*​+*Aj*+1​+⋯+*AN*−1​)

we can write this as

(\sum\_{k=0}^{N-1} A\_k) - (A\_{i+1} + A\_{i+2} + \cdots + A\_{j-1})(∑*k*=0*N*−1​*Ak*​)−(*Ai*+1​+*Ai*+2​+⋯+*Aj*−1​)

For two-interval subarrays, let B*B* be the array A*A* with each element multiplied by -1−1. Then the answer for two-interval subarrays is \text{sum}(A) + \text{kadane}(B)sum(*A*)+kadane(*B*).

Except, this isn't quite true, as if the subarray of B*B* we choose is the entire array, the resulting two interval subarray [0, i] + [j, N-1][0,*i*]+[*j*,*N*−1] would be empty.

We can remedy this problem by doing Kadane twice: once on B*B* with the first element removed, and once on B*B* with the last element removed.

|  |
| --- |
| class Solution {  public int maxSubarraySumCircular(int[] A) {  int S = 0; // S = sum(A)  for (int x: A)  S += x;  int ans1 = kadane(A, 0, A.length-1, 1);  int ans2 = S + kadane(A, 1, A.length-1, -1);  int ans3 = S + kadane(A, 0, A.length-2, -1);  return Math.max(ans1, Math.max(ans2, ans3));  }  public int kadane(int[] A, int i, int j, int sign) {  // The maximum non-empty subarray for array  // [sign \* A[i], sign \* A[i+1], ..., sign \* A[j]]  int ans = Integer.MIN\_VALUE;  int cur = Integer.MIN\_VALUE;  for (int k = i; k <= j; ++k) {  cur = sign \* A[k] + Math.max(cur, 0);  ans = Math.max(ans, cur);  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of A.
* Space Complexity: O(1)*O*(1) in additional space complexity.

#### **Approach 4: Kadane's (Min Variant)**

**Intuition and Algorithm**

As in Approach 3, subarrays of circular arrays can be classified as either as one-interval subarrays (which we can use Kadane's algorithm), or two-interval subarrays.

We can modify Kadane's algorithm to use min instead of max. All the math in our explanation of Kadane's algorithm remains the same, but the algorithm lets us find the minimum sum of a subarray instead.

For a two interval subarray written as (\sum\_{k=0}^{N-1} A\_k) - (\sum\_{k=i+1}^{j-1} A\_k)(∑*k*=0*N*−1​*Ak*​)−(∑*k*=*i*+1*j*−1​*Ak*​), we can use our kadane-min algorithm to minimize the "interior" (\sum\_{k=i+1}^{j-1} A\_k)(∑*k*=*i*+1*j*−1​*Ak*​) part of the sum.

Again, because the interior [i+1, j-1][*i*+1,*j*−1] must be non-empty, we can break up our search into a search on A[1:] and on A[:-1].

|  |
| --- |
| class Solution {  public int maxSubarraySumCircular(int[] A) {  // S: sum of A  int S = 0;  for (int x: A)  S += x;  // ans1: answer for one-interval subarray  int ans1 = Integer.MIN\_VALUE;  int cur = Integer.MIN\_VALUE;  for (int x: A) {  cur = x + Math.max(cur, 0);  ans1 = Math.max(ans1, cur);  }  // ans2: answer for two-interval subarray, interior in A[1:]  int ans2 = Integer.MAX\_VALUE;  cur = Integer.MAX\_VALUE;  for (int i = 1; i < A.length; ++i) {  cur = A[i] + Math.min(cur, 0);  ans2 = Math.min(ans2, cur);  }  ans2 = S - ans2;  // ans3: answer for two-interval subarray, interior in A[:-1]  int ans3 = Integer.MAX\_VALUE;  cur = Integer.MAX\_VALUE;  for (int i = 0; i < A.length - 1; ++i) {  cur = A[i] + Math.min(cur, 0);  ans3 = Math.min(ans3, cur);  }  return Math.max(ans1, Math.max(ans2, ans3));  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of A.
* Space Complexity: O(1)*O*(1) in additional space complexity.

**Odd Even Linked List**

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.

You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.

**Example 1:**

**Input:** 1->2->3->4->5->NULL

**Output:** 1->3->5->2->4->NULL

**Example 2:**

**Input:** 2->1->3->5->6->4->7->NULL

**Output:** 2->3->6->7->1->5->4->NULL

**Constraints:**

* The relative order inside both the even and odd groups should remain as it was in the input.
* The first node is considered odd, the second node even and so on ...
* The length of the linked list is between [0, 10^4].

## Solution

**Intuition**

Put the odd nodes in a linked list and the even nodes in another. Then link the evenList to the tail of the oddList.

**Algorithm**

The solution is very intuitive. But it is not trivial to write a concise and bug-free code.

A well-formed LinkedList need two pointers head and tail to support operations at both ends. The variables head and odd are the head pointer and tail pointer of one LinkedList we call oddList; the variables evenHead and even are the head pointer and tail pointer of another LinkedList we call evenList. The algorithm traverses the original LinkedList and put the odd nodes into the oddList and the even nodes into the evenList. To traverse a LinkedList we need at least one pointer as an iterator for the current node. But here the pointers odd and even not only serve as the tail pointers but also act as the iterators of the original list.

The best way of solving any linked list problem is to visualize it either in your mind or on a piece of paper. An illustration of our algorithm is following:

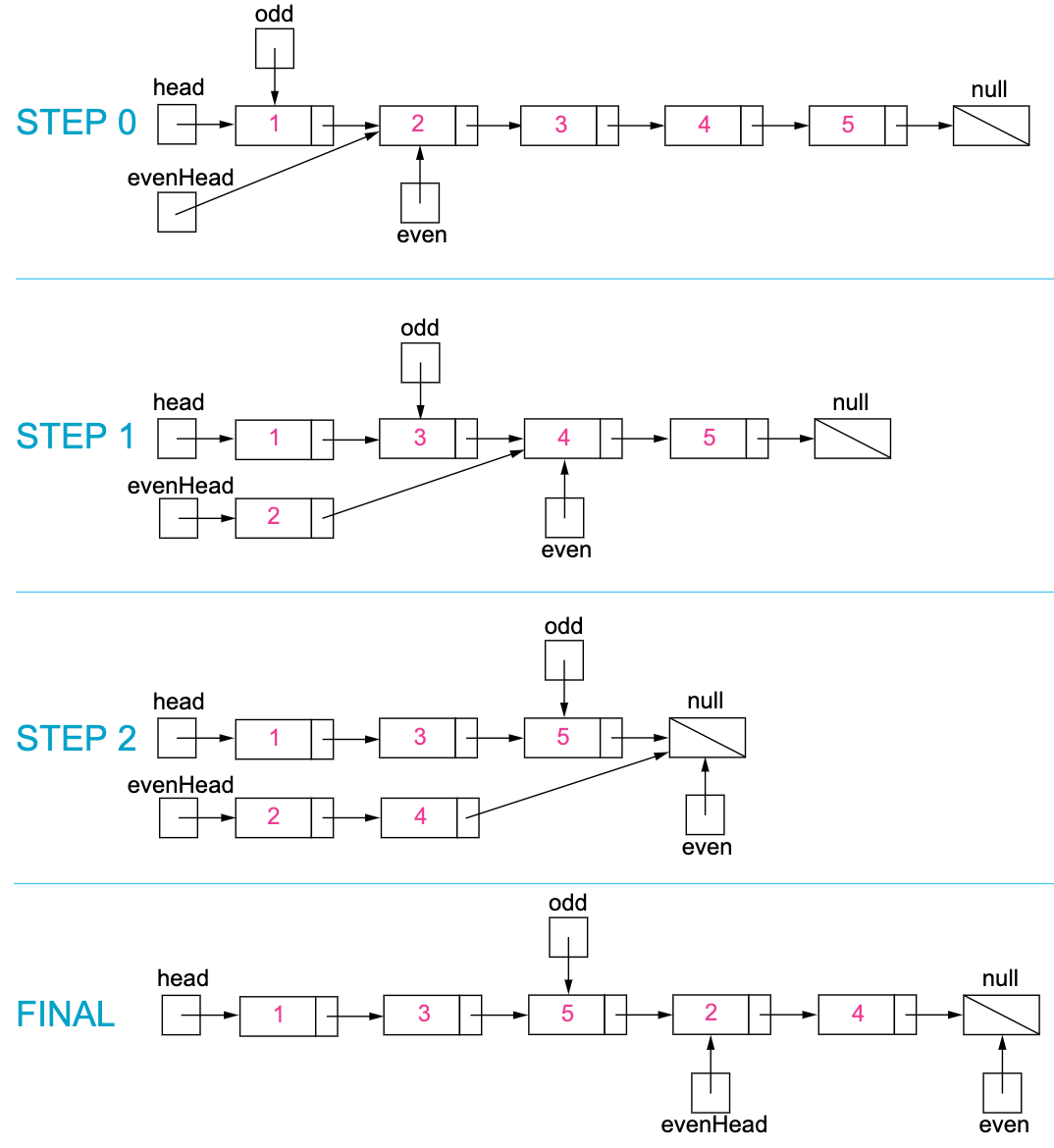


Figure 1. Step by step example of the odd and even linked list.

|  |
| --- |
| public class Solution {  public ListNode oddEvenList(ListNode head) {  if (head == null) return null;  ListNode odd = head, even = head.next, evenHead = even;  while (even != null && even.next != null) {  odd.next = even.next;  odd = odd.next;  even.next = odd.next;  even = even.next;  }  odd.next = evenHead;  return head;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). There are total n*n* nodes and we visit each node once.
* Space complexity : O(1)*O*(1). All we need is the four pointers.

**Find All Anagrams in a String**

Given a string **s** and a **non-empty** string **p**, find all the start indices of **p**'s anagrams in **s**.

Strings consists of lowercase English letters only and the length of both strings **s** and **p** will not be larger than 20,100.

The order of output does not matter.

**Example 1:**

**Input:**

s: "cbaebabacd" p: "abc"

**Output:**

[0, 6]

**Explanation:**

The substring with start index = 0 is "cba", which is an anagram of "abc".

The substring with start index = 6 is "bac", which is an anagram of "abc".

**Example 2:**

**Input:**

s: "abab" p: "ab"

**Output:**

[0, 1, 2]

**Explanation:**

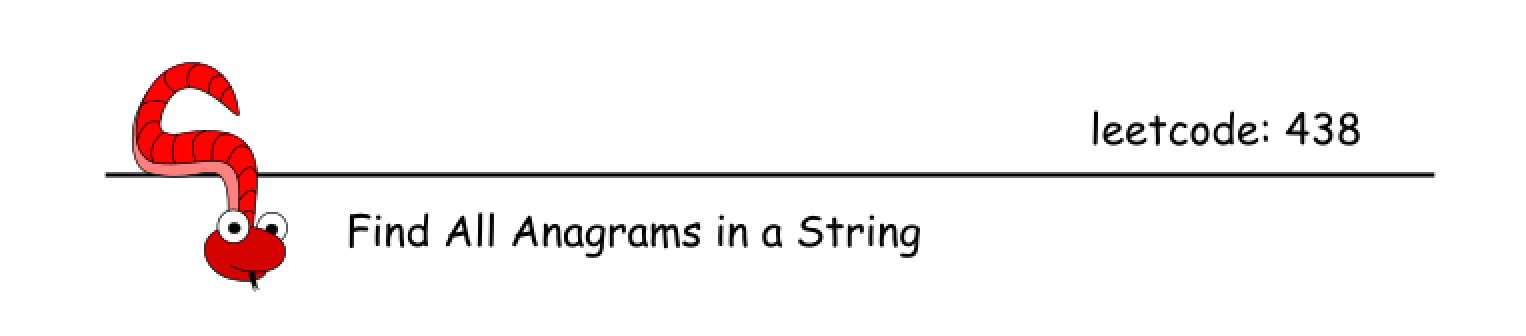
The substring with start index = 0 is "ab", which is an anagram of "ab".

The substring with start index = 1 is "ba", which is an anagram of "ab".

The substring with start index = 2 is "ab", which is an anagram of "ab".

## Solution

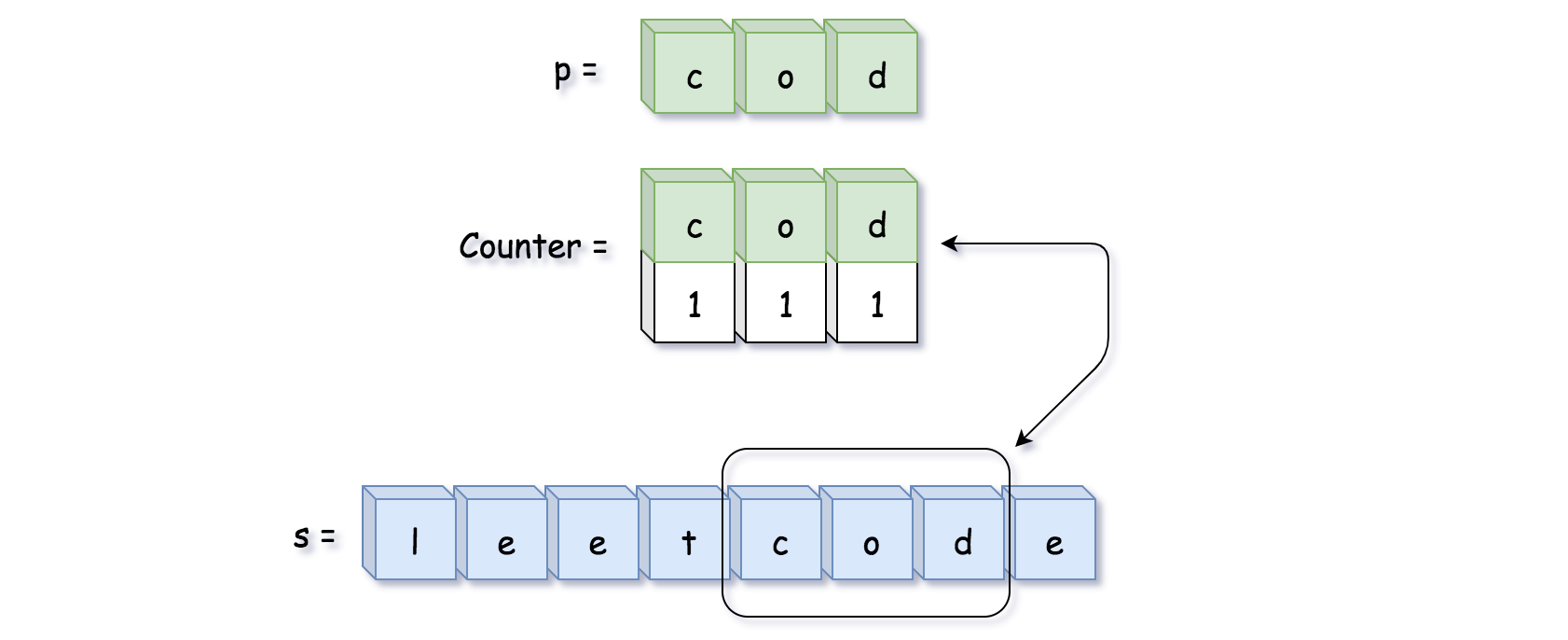
#### **Solution Template**



This is a problem of multiple pattern search in a string. All such problems usually could be solved by sliding window approach in a linear time. The challenge here is how to implement constant-time slice to fit into this linear time.

If the patterns are not known in advance, i.e. it's "find duplicates" problem, one could use one of two ways to implement constant-time slice: Bitmasks or Rabin-Karp. Please check article [Repeated DNA Sequences](https://leetcode.com/articles/repeated-dna-sequences/) for the detailed comparison of these two algorithms.

Here the situation is more simple: patterns are known in advance, and the set of characters in the patterns is very limited as well: 26 lowercase English letters. Hence one could allocate array or hashmap with 26 elements and use it as a letter counter in the sliding window.



#### **Approach 1: Sliding Window with HashMap**

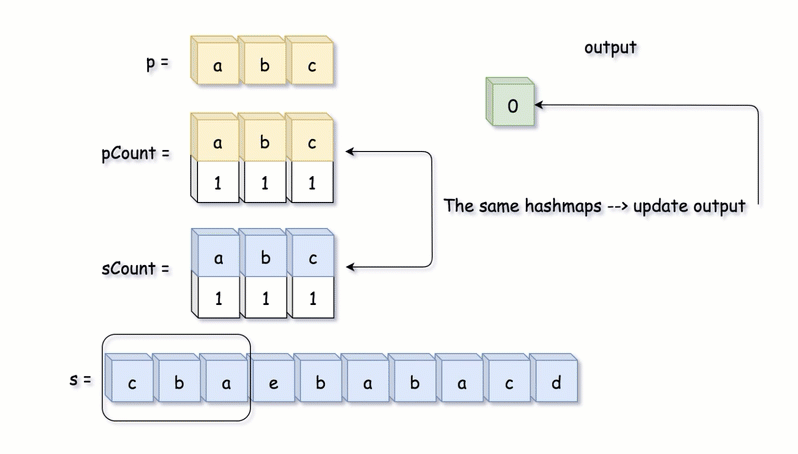
Let's start from the simplest approach: sliding window + two counter hashmaps letter -> its count. The first hashmap is a reference counter pCount for string p, and the second one is a counter sCount for string in the sliding window.

The idea is to move sliding window along the string s, recompute the second hashmap sCount in a constant time and compare it with the first hashmap pCount. If sCount == pCount, then the string in the sliding window is a permutation of string p, and one could add its start position in the output list.

**Algorithm**

* Build reference counter pCount for string p.
* Move sliding window along the string s:
  + Recompute sliding window counter sCount at each step by adding one letter on the right and removing one letter on the left.
  + If sCount == pCount, update the output list.
* Return output list.

**Implementation**



|  |
| --- |
| class Solution {  public List<Integer> findAnagrams(String s, String p) {  int ns = s.length(), np = p.length();  if (ns < np) return new ArrayList();  Map<Character, Integer> pCount = new HashMap();  Map<Character, Integer> sCount = new HashMap();  // build reference hashmap using string p  for (char ch : p.toCharArray()) {  if (pCount.containsKey(ch)) {  pCount.put(ch, pCount.get(ch) + 1);  }  else {  pCount.put(ch, 1);  }  }  List<Integer> output = new ArrayList();  // sliding window on the string s  for (int i = 0; i < ns; ++i) {  // add one more letter  // on the right side of the window  char ch = s.charAt(i);  if (sCount.containsKey(ch)) {  sCount.put(ch, sCount.get(ch) + 1);  }  else {  sCount.put(ch, 1);  }  // remove one letter  // from the left side of the window  if (i >= np) {  ch = s.charAt(i - np);  if (sCount.get(ch) == 1) {  sCount.remove(ch);  }  else {  sCount.put(ch, sCount.get(ch) - 1);  }  }  // compare hashmap in the sliding window  // with the reference hashmap  if (pCount.equals(sCount)) {  output.add(i - np + 1);  }  }  return output;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N\_s + N\_p)O(*Ns*​+*Np*​) since it's one pass along both strings.
* Space complexity: \mathcal{O}(1)O(1), because pCount and sCount contain not more than 26 elements.

#### **Approach 2: Sliding Window with Array**

**Algorithm**

Hashmap is quite complex structure, [with known performance issues in Java](https://github.com/vavr-io/vavr/issues/571). Let's implement approach 1 using 26-elements array instead of hashmap:

* Element number 0 contains count of letter a.
* Element number 1 contains count of letter b.
* ...
* Element number 26 contains count of letter z.

**Algorithm**

* Build reference array pCount for string p.
* Move sliding window along the string s:
  + Recompute sliding window array sCount at each step by adding one letter on the right and removing one letter on the left.
  + If sCount == pCount, update the output list.
* Return output list.

**Implementation**

|  |
| --- |
| class Solution {  public List<Integer> findAnagrams(String s, String p) {  int ns = s.length(), np = p.length();  if (ns < np) return new ArrayList();  int [] pCount = new int[26];  int [] sCount = new int[26];  // build reference array using string p  for (char ch : p.toCharArray()) {  pCount[(int)(ch - 'a')]++;  }  List<Integer> output = new ArrayList();  // sliding window on the string s  for (int i = 0; i < ns; ++i) {  // add one more letter  // on the right side of the window  sCount[(int)(s.charAt(i) - 'a')]++;  // remove one letter  // from the left side of the window  if (i >= np) {  sCount[(int)(s.charAt(i - np) - 'a')]--;  }  // compare array in the sliding window  // with the reference array  if (Arrays.equals(pCount, sCount)) {  output.add(i - np + 1);  }  }  return output;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N\_s + N\_p)O(*Ns*​+*Np*​) since it's one pass along both strings.
* Space complexity: \mathcal{O}(1)O(1), because pCount and sCount contain 26 elements each.

**Online Stock Span**

Write a class StockSpanner which collects daily price quotes for some stock, and returns the span of that stock's price for the current day.

The span of the stock's price today is defined as the maximum number of consecutive days (starting from today and going backwards) for which the price of the stock was less than or equal to today's price.

For example, if the price of a stock over the next 7 days were [100, 80, 60, 70, 60, 75, 85], then the stock spans would be [1, 1, 1, 2, 1, 4, 6].

**Example 1:**

**Input:** ["StockSpanner","next","next","next","next","next","next","next"], [[],[100],[80],[60],[70],[60],[75],[85]]

**Output:** [null,1,1,1,2,1,4,6]

**Explanation:**

First, S = StockSpanner() is initialized. Then:

S.next(100) is called and returns 1,

S.next(80) is called and returns 1,

S.next(60) is called and returns 1,

S.next(70) is called and returns 2,

S.next(60) is called and returns 1,

S.next(75) is called and returns 4,

S.next(85) is called and returns 6.

Note that (for example) S.next(75) returned 4, because the last 4 prices

(including today's price of 75) were less than or equal to today's price.

**Note:**

1. Calls to StockSpanner.next(int price) will have 1 <= price <= 10^5.
2. There will be at most 10000 calls to StockSpanner.next per test case.
3. There will be at most 150000 calls to StockSpanner.next across all test cases.
4. The total time limit for this problem has been reduced by 75% for C++, and 50% for all other languages.

## Solution

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

**Intuition**

Clearly, we need to focus on how to make each query faster than a linear scan. In a typical case, we get a new element like 7, and there are some previous elements like 11, 3, 9, 5, 6, 4. Let's try to create some relationship between this query and the next query.

If (after getting 7) we get an element like 2, then the answer is 1. So in general, whenever we get a smaller element, the answer is 1.

If we get an element like 8, the answer is 1 plus the previous answer (for 7), as the 8 "stops" on the same value that 7 does (namely, 9).

If we get an element like 10, the answer is 1 plus the previous answer, plus the answer for 9.

Notice throughout this evaluation, we only care about elements that occur in increasing order - we "shortcut" to them. That is, from adding an element like 10, we cut to 7 [with "weight" 4], then to 9 [with weight 2], then cut to 11 [with weight 1].

A stack is the ideal data structure to maintain what we care about efficiently.

**Algorithm**

Let's maintain a weighted stack of decreasing elements. The size of the weight will be the total number of elements skipped. For example, 11, 3, 9, 5, 6, 4, 7 will be (11, weight=1), (9, weight=2), (7, weight=4).

When we get a new element like 10, this helps us count the previous values faster by popping weighted elements off the stack. The new stack at the end will look like (11, weight=1), (10, weight=7).

|  |
| --- |
| class StockSpanner {  Stack<Integer> prices, weights;  public StockSpanner() {  prices = new Stack();  weights = new Stack();  }  public int next(int price) {  int w = 1;  while (!prices.isEmpty() && prices.peek() <= price) {  prices.pop();  w += weights.pop();  }  prices.push(price);  weights.push(w);  return w;  }  } |

**Complexity Analysis**

* Time Complexity: O(Q)*O*(*Q*), where Q*Q* is the number of calls to StockSpanner.next. In total, there are Q*Q* pushes to the stack, and at most Q*Q* pops.
* Space Complexity: O(Q)*O*(*Q*).

**Count Square Submatrices with All Ones**

Given a m \* n matrix of ones and zeros, return how many **square** submatrices have all ones.

**Example 1:**

**Input:** matrix =

[

  [0,1,1,1],

  [1,1,1,1],

  [0,1,1,1]

]

**Output:** 15

**Explanation:**

There are **10** squares of side 1.

There are **4** squares of side 2.

There is **1** square of side 3.

Total number of squares = 10 + 4 + 1 = **15**.

**Example 2:**

**Input:** matrix =

[

[1,0,1],

[1,1,0],

[1,1,0]

]

**Output:** 7

**Explanation:**

There are **6** squares of side 1.

There is **1** square of side 2.

Total number of squares = 6 + 1 = **7**.

**Constraints:**

* 1 <= arr.length <= 300
* 1 <= arr[0].length <= 300
* 0 <= arr[i][j] <= 1

   Hide Hint #1

Create an additive table that counts the sum of elements of submatrix with the superior corner at (0,0).

   Hide Hint #2

Loop over all subsquares in O(n^3) and check if the sum make the whole array to be ones, if it checks then add 1 to the answer.

**Sort Characters By Frequency**

Given a string, sort it in decreasing order based on the frequency of characters.

**Example 1:**

**Input:**

"tree"

**Output:**

"eert"

**Explanation:**

'e' appears twice while 'r' and 't' both appear once.

So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.

**Example 2:**

**Input:**

"cccaaa"

**Output:**

"cccaaa"

**Explanation:**

Both 'c' and 'a' appear three times, so "aaaccc" is also a valid answer.

Note that "cacaca" is incorrect, as the same characters must be together.

**Example 3:**

**Input:**

"Aabb"

**Output:**

"bbAa"

**Explanation:**

"bbaA" is also a valid answer, but "Aabb" is incorrect.

Note that 'A' and 'a' are treated as two different characters.

## Solution

#### **Remember, Strings are Immutable!**

The input type for this question is a String. When dealing with Strings, we need to be careful to not inadvertently convert what should have been an O(n)*O*(*n*) algorithm into an O(n^2)*O*(*n*2) one.

Strings in most programming languages are **immutable**. This means that once a String is created, we cannot modify it. We can only create a new String. Consider the following Java code.

String a = "Hello ";

a += "Leetcode";

This code creates a String called a with the value "Hello ". It then sets a to be a new String, made with the letters from the old a and the additional letters "Leetcode". It then assigns this new String to the variable a, throwing away the reference to the old one. It does NOT actually "modify" a.

For the most part, we don't run into problems with Strings being treated like this. But consider this code for reversing a String.

String s = "Hello There";

String reversedString = "";

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

reversedString += s.charAt(i);

}

System.out.println(reversedString);

Each time a character is added to reverseString, a new String is created. Creating a new String has a cost of n*n*, where n*n* is the length of the String. The result? Simply reversing a String has a cost of O(n^2)*O*(*n*2) using the above algorithm.

The solution is to use a StringBuilder. A StringBuilder collects up the characters that will be converted into a String so that only one String needs to be created—once all the characters are ready to go. Recall that inserting an item at the end of an Array has a cost of O(1)*O*(1), and so the total cost of inserting the n*n* characters into the StringBuilder is O(n)*O*(*n*), and it is also O(n)*O*(*n*) to then convert that StringBuilder into a String, giving a total of O(n)*O*(*n*).

String s = "Hello There";

StringBuilder sb = new StringBuilder();

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

sb.append(s.charAt(i));

}

String reversedString = sb.toString();

System.out.println(reversedString);

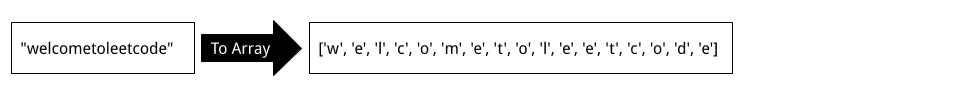
If you're unsure what to do for your particular programming language, it shouldn't be too difficult to find using a web search. The algorithms provided in the solutions here all do string building efficiently.

#### **Approach 1: Arrays and Sorting**

**Intuition**

In order to sort the characters by frequency, we firstly need to know how many of each there are. One way to do this is to sort the characters by their numbers so that identical characters are side-by-side (all characters in a programming language are identified by a unique number). Then, knowing how many times each appears will be a lot easier.

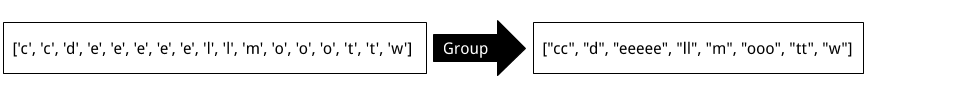
Because Strings are **immutable** though, we cannot sort the String directly. Therefore, we'll need to start by converting it from a String to an Array of characters.



Now that we have an Array, we can sort it, which will make all identical characters side-by-side.

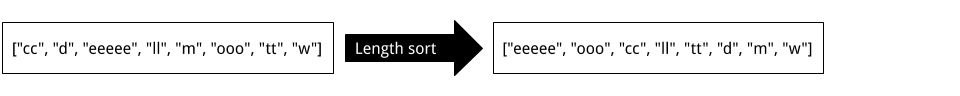


There are a few different ways we can go from here. One easy-to-understand way is to create a new Array of Strings. Each String in the list will consist of one of the unique characters from the sorted characters Array.

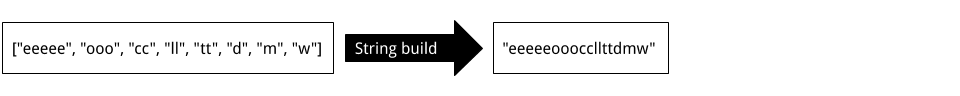


Remember: do this process using StringBuilders, not naïve String appending! (See the first section of this article if you're confused).

The next step is to sort this Array of Strings by length. To do this, we'll need to implement a suitable **Comparator**. Recall that there is no requirement for characters of the same frequency to appear in a specific order.



Finally, we can convert this Array of Strings into a single String. In Java, this can be done by passing the Array into a StringBuilder and then calling .toString(...) on it.



**Algorithm**

|  |
| --- |
| public String frequencySort(String s) {    if (s == null || s.isEmpty()) return s;    // Create a sorted Array of chars.  char[] chars = s.toCharArray();  Arrays.sort(chars);    // Convert identical chars into single Strings.  List<String> charStrings = new ArrayList<String>();  StringBuilder currentString = new StringBuilder();  currentString.append(chars[0]);  for (int i = 1; i < chars.length; i++) {  if (chars[i] != chars[i - 1]) {  charStrings.add(currentString.toString());  currentString = new StringBuilder();  }  currentString.append(chars[i]);  }  charStrings.add(currentString.toString());    // Our comparator is (a, b) -> b.length() - a.length().  // If a is longer than b, then a negative number will be returned  // telling the sort algorithm to place a first. Otherwise, a positive  // number will be returned, telling it to place a second.  // This results in a longest to shortest sorted list of the strings.  Collections.sort(charStrings, (a, b) -> b.length() - a.length());    // Use StringBuilder to build the String to return.  StringBuilder sb = new StringBuilder();  for (String str : charStrings) sb.append(str);  return sb.toString();  } |

**Complexity Analysis**

Let n*n* be the length of the input String.

* Time Complexity : O(n \, \log \, n)*O*(*n*log*n*).

The first part of the algorithm, converting the String to a List of characters, has a cost of O(n)*O*(*n*), because we are adding n*n* characters to the end of a List.

The second part of the algorithm, sorting the List of characters, has a cost of O(n \, \log \, n)*O*(*n*log*n*).

The third part of the algorithm, grouping the characters into Strings of similar characters, has a cost of O(n)*O*(*n*) because each character is being inserted once into a StringBuilder and once converted into a String.

Finally, the fourth part of the algorithm, sorting the Strings by length, has a worst case cost of O(n)*O*(*n*), which occurs when all the characters in the input String are unique.

Because we drop constants and insignificant terms, we get O(n \, \log \, n) + 3 \cdot O(n) = O(n \, \log \, n)*O*(*n*log*n*)+3⋅*O*(*n*)=*O*(*n*log*n*).

Be careful with your own implementation—if you didn't do the string building process in a sensible way, then your solution could potentially be O(n^2)*O*(*n*2).

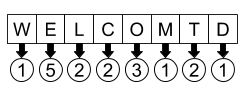
* Space Complexity : O(n)*O*(*n*).

It is impossible to do better with the space complexity, because Strings are immutable. The List of characters, List of Strings, and the final output String, are all of length n*n*, so we have a space complexity of O(n)*O*(*n*).

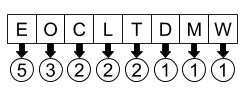
#### **Approach 2: HashMap and Sort**

**Intuition**

Another approach is to use a HashMap to count how many times each character occurs in the String; the keys are characters and the values are frequencies.



Next, extract a copy of the keys from the HashMap and sort them by frequency using a Comparator that looks at the HashMap values to make its decisions.



Finally, initialise a new StringBuilder and then iterate over the list of sorted characters (sorted by frequency). Look up the values in the HashMap to know how many of each character to append to the StringBuilder.

**Algorithm**

|  |
| --- |
| public String frequencySort(String s) {  // Count up the occurances.  Map<Character, Integer> counts = new HashMap<>();  for (char c : s.toCharArray()) {  counts.put(c, counts.getOrDefault(c, 0) + 1);  }    // Make a list of the keys, sorted by frequency.  List<Character> characters = new ArrayList<>(counts.keySet());  Collections.sort(characters, (a, b) -> counts.get(b) - counts.get(a));  // Convert the counts into a string with a sb.  StringBuilder sb = new StringBuilder();  for (char c : characters) {  int copies = counts.get(c);  for (int i = 0; i < copies; i++) {  sb.append(c);  }  }  return sb.toString();  } |

**Complexity Analysis**

Let n*n* be the length of the input String and k*k* be the number of unique characters in the String.

We know that k ≤ n*k*≤*n*, because there can't be more unique characters than there are characters in the String. We also know that k*k* is somewhat bounded by the fact that there's only a finite number of characters in Unicode (or ASCII, which I suspect is all we need to worry about for this question).

There are two ways of approaching the complexity analysis for this question.

1. We can disregard k*k*, and consider that in the worst case, *k = n*.
2. We can consider k*k*, recognising that the number of unique characters is not infinite. This is more accurate for real world purposes.

I've provided analysis for both ways of approaching it. I choose not to bring it up for the previous approach, because it made no difference there.

* Time Complexity : O(n \, \log \, n)*O*(*n*log*n*) OR O(n + k \, \log \, k)*O*(*n*+*k*log*k*).

Putting the characterss into the HashMap has a cost of O(n)*O*(*n*), because each of the n*n* characterss must be put in, and putting each in is an O(1)*O*(1) operation.

Sorting the HashMap keys has a cost of O(k \, \log \, k)*O*(*k*log*k*), because there are k*k* keys, and this is the standard cost for sorting. If only using n*n*, then it's O(n \, \log \, n)*O*(*n*log*n*). For the previous question, the sort was carried out on n*n* items, not k*k*, so was possibly a lot worse.

Traversing over the sorted keys and building the String has a cost of O(n)*O*(*n*), as n*n* characters must be inserted.

Therefore, if we're only considering n*n*, then the final cost is O(n \, \log \, n)*O*(*n*log*n*).

Considering k*k* as well gives us O(n + k \, \log \, k)*O*(*n*+*k*log*k*), because we don't know which is largest out of n*n* and k \, \log \, k*k*log*k*. We do, however, know that in total this is less than or equal to O(n \, \log \, n)*O*(*n*log*n*).

* Space Complexity : O(n)*O*(*n*).

The HashMap uses O(k)*O*(*k*) space.

However, the StringBuilder at the end dominates the space complexity, pushing it up to O(n)*O*(*n*), as every character from the input String must go into it. Like was said above, it's impossible to do better with the space complexity here.

What's interesting here is that if we only consider n*n*, the time complexity is the same as the previous approach. But by considering k*k*, we can see that the difference is potentially substantial.

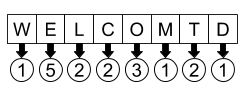
#### **Approach 3: Multiset and Bucket Sort**

**Intuition**

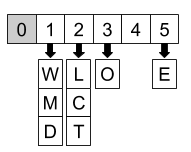
While the second approach is probably adequate for an interview, there is actually a way of solving this problem with a time complexity of O(n)*O*(*n*).

Firstly, observe that because all of the characters came out of a String of length n*n*, the maximum frequency for any one character is n*n*. This means that once we've determined all the letter frequencies using a HashMap, we can sort them in O(n)*O*(*n*) time using **Bucket Sort**. Recall that for our previous approaches, we used comparison-based sorts, which have a cost of O(n \, \log \, n)*O*(*n*log*n*).

This was the HashMap from earlier.



Recall that **Bucket Sort** is the sorting algorithm where items are placed at Array indexes based on their values (the indexes are called "buckets"). For this problem, we'll need to have a List of characters at each index. For example, here is how our String from before goes into the buckets.



While we could simply make our bucket Array length n*n*, we're best to just look for the maximum value (frequency) in the HashMap. That way, we only use as much space as we need, and won't need to iterate over heaps of empty buckets during the next phase.

Finally, we need to iterate over the buckets, starting with the largest and ending with the smallest, building up the string in much the same way as we did before.

**Algorithm**

|  |
| --- |
| public String frequencySort(String s) {    if (s == null || s.isEmpty()) return s;    // Count up the occurances.  Map<Character, Integer> counts = new HashMap<>();  for (char c : s.toCharArray()) {  counts.put(c, counts.getOrDefault(c, 0) + 1);  }    int maximumFrequency = Collections.max(counts.values());    // Make the list of buckets and apply bucket sort.  List<List<Character>> buckets = new ArrayList<>();  for (int i = 0; i <= maximumFrequency; i++) {  buckets.add(new ArrayList<Character>());  }  for (Character key : counts.keySet()) {  int freq = counts.get(key);  buckets.get(freq).add(key);  }  // Build up the string.  StringBuilder sb = new StringBuilder();  for (int i = buckets.size() - 1; i >= 1; i--) {  for (Character c : buckets.get(i)) {  for (int j = 0; j < i; j++) {  sb.append(c);  }  }  }  return sb.toString();  } |

**Complexity Analysis**

Let n*n* be the length of the input String. The k*k* (number of unique characters in the input String that we considered for the last approach makes no difference this time).

* Time Complexity : O(n)*O*(*n*).

Like before, the HashMap building has a cost of O(n)*O*(*n*).

The bucket sorting is O(n)*O*(*n*), because inserting items has a cost of O(k)*O*(*k*) (each entry from the HashMap), and building the buckets initially has a worst case of O(n)*O*(*n*) (which occurs when k = 1*k*=1). Because k ≤ n*k*≤*n*, we're left with O(n)*O*(*n*).

So in total, we have O(n)*O*(*n*).

It'd be impossible to do better than this, because we need to look at each of the n*n* characters in the input String at least once.

* Space Complexity : O(n)*O*(*n*).

Same as above. The bucket Array also uses O(n)*O*(*n*) space, because its length is at most n*n*, and there are k*k* items across all the buckets.

**Interval List Intersections**

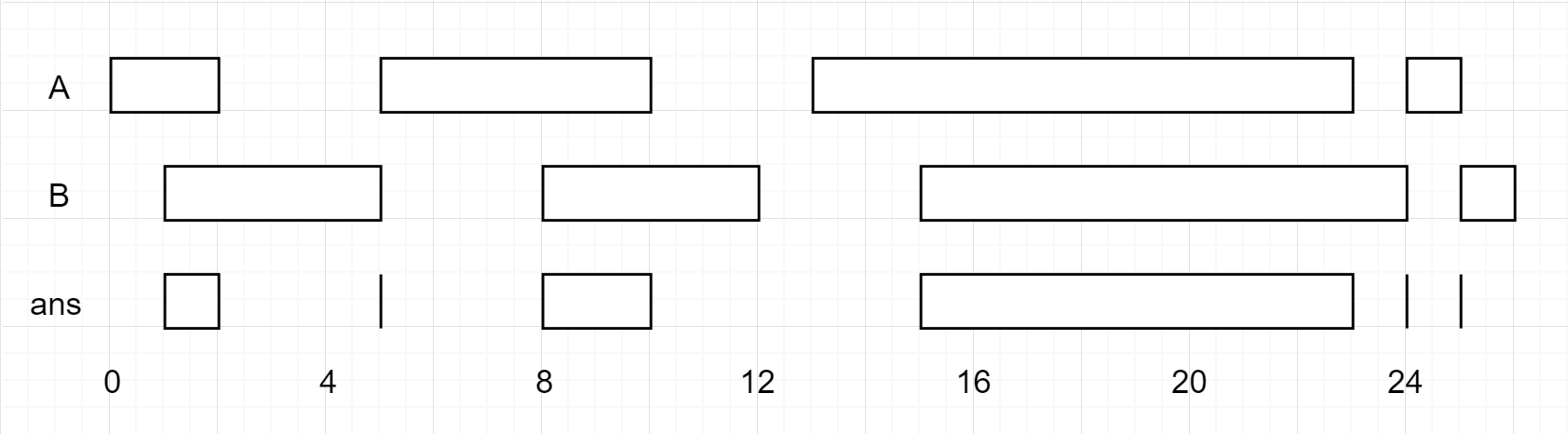
You are given two lists of closed intervals, firstList and secondList, where firstList[i] = [starti, endi] and secondList[j] = [startj, endj]. Each list of intervals is pairwise **disjoint** and in **sorted order**.

Return the intersection of these two interval lists.

A **closed interval** [a, b] (with a < b) denotes the set of real numbers x with a <= x <= b.

The **intersection** of two closed intervals is a set of real numbers that are either empty or represented as a closed interval. For example, the intersection of [1, 3] and [2, 4] is [2, 3].

**Example 1:**



**Input:** firstList = [[0,2],[5,10],[13,23],[24,25]], secondList = [[1,5],[8,12],[15,24],[25,26]]

**Output:** [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]

**Example 2:**

**Input:** firstList = [[1,3],[5,9]], secondList = []

**Output:** []

**Example 3:**

**Input:** firstList = [], secondList = [[4,8],[10,12]]

**Output:** []

**Example 4:**

**Input:** firstList = [[1,7]], secondList = [[3,10]]

**Output:** [[3,7]]

**Constraints:**

* 0 <= firstList.length, secondList.length <= 1000
* firstList.length + secondList.length >= 1
* 0 <= starti < endi <= 109
* endi < starti+1
* 0 <= startj < endj <= 109
* endj < startj+1

## Solution

#### **Approach 1: Merge Intervals**

**Intuition**

In an interval [a, b], call b the "endpoint".

Among the given intervals, consider the interval A[0] with the smallest endpoint. (Without loss of generality, this interval occurs in array A.)

Then, among the intervals in array B, A[0] can only intersect one such interval in array B. (If two intervals in B intersect A[0], then they both share the endpoint of A[0] -- but intervals in B are disjoint, which is a contradiction.)

**Algorithm**

If A[0] has the smallest endpoint, it can only intersect B[0]. After, we can discard A[0] since it cannot intersect anything else.

Similarly, if B[0] has the smallest endpoint, it can only intersect A[0], and we can discard B[0] after since it cannot intersect anything else.

We use two pointers, i and j, to virtually manage "discarding" A[0] or B[0] repeatedly.

|  |
| --- |
| class Solution {  public int[][] intervalIntersection(int[][] A, int[][] B) {  List<int[]> ans = new ArrayList();  int i = 0, j = 0;  while (i < A.length && j < B.length) {  // Let's check if A[i] intersects B[j].  // lo - the startpoint of the intersection  // hi - the endpoint of the intersection  int lo = Math.max(A[i][0], B[j][0]);  int hi = Math.min(A[i][1], B[j][1]);  if (lo <= hi)  ans.add(new int[]{lo, hi});  // Remove the interval with the smallest endpoint  if (A[i][1] < B[j][1])  i++;  else  j++;  }  return ans.toArray(new int[ans.size()][]);  }  } |

**Complexity Analysis**

* Time Complexity: O(M + N)*O*(*M*+*N*), where M, N*M*,*N* are the lengths of A and B respectively.
* Space Complexity: O(M + N)*O*(*M*+*N*), the maximum size of the answer.

**Uncrossed Lines**

We write the integers of A and B (in the order they are given) on two separate horizontal lines.

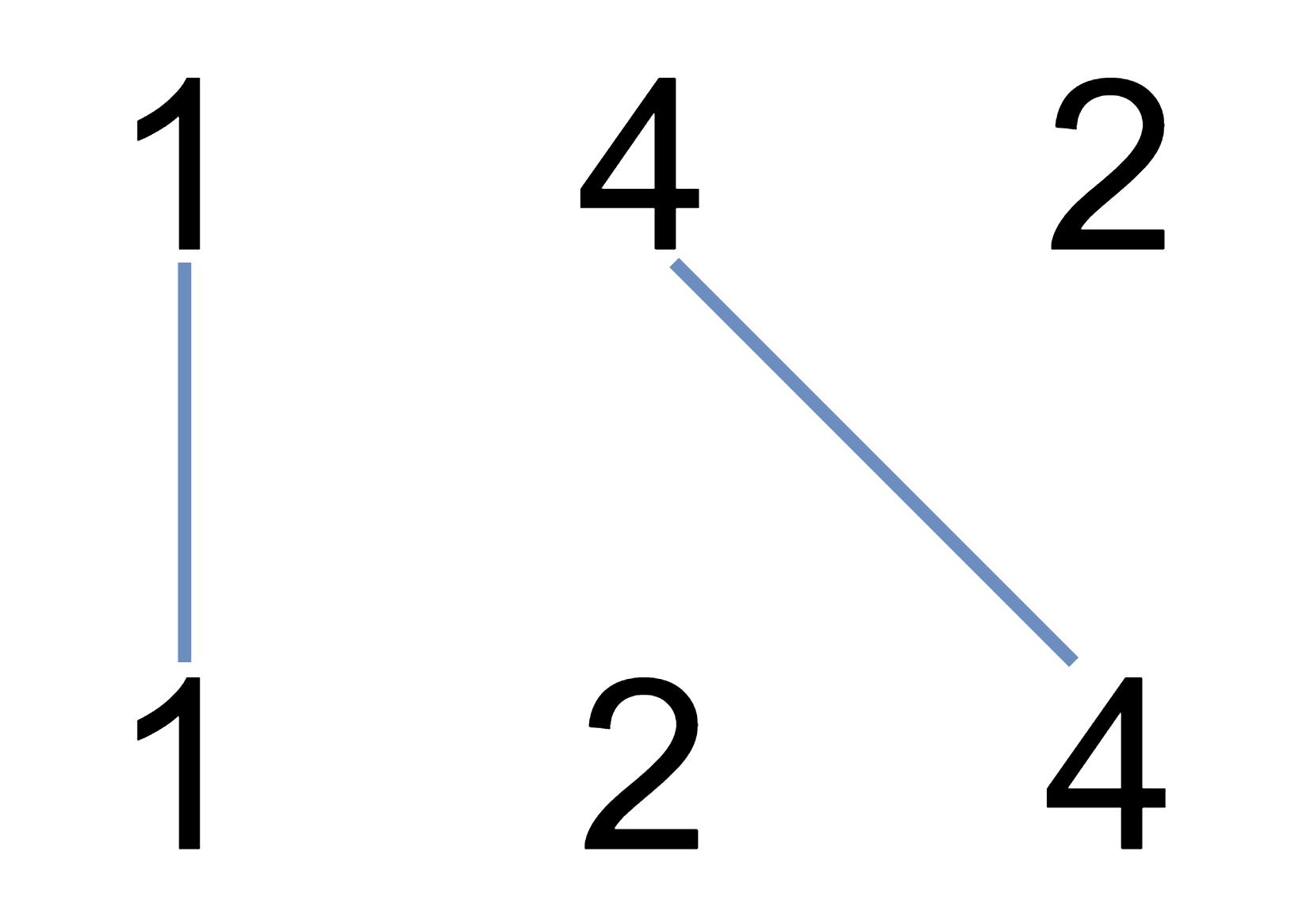
Now, we may draw connecting lines: a straight line connecting two numbers A[i] and B[j] such that:

* A[i] == B[j];
* The line we draw does not intersect any other connecting (non-horizontal) line.

Note that a connecting lines cannot intersect even at the endpoints: each number can only belong to one connecting line.

Return the maximum number of connecting lines we can draw in this way.

**Example 1:**



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

**Output:** 2

**Explanation:** We can draw 2 uncrossed lines as in the diagram.

We cannot draw 3 uncrossed lines, because the line from A[1]=4 to B[2]=4 will intersect the line from A[2]=2 to B[1]=2.

**Example 2:**

**Input:** A = [2,5,1,2,5], B = [10,5,2,1,5,2]

**Output:** 3

**Example 3:**

**Input:** A = [1,3,7,1,7,5], B = [1,9,2,5,1]

**Output:** 2

**Note:**

1. 1 <= A.length <= 500
2. 1 <= B.length <= 500
3. 1 <= A[i], B[i] <= 2000

   Hide Hint #1

Think dynamic programming. Given an oracle dp(i,j) that tells us how many lines A[i:], B[j:] [the sequence A[i], A[i+1], ... and B[j], B[j+1], ...] are uncrossed, can we write this as a recursion?

**Possible Bipartition**

Given a set of N people (numbered 1, 2, ..., N), we would like to split everyone into two groups of **any** size.

Each person may dislike some other people, and they should not go into the same group.

Formally, if dislikes[i] = [a, b], it means it is not allowed to put the people numbered a and b into the same group.

Return true if and only if it is possible to split everyone into two groups in this way.

**Example 1:**

**Input:** N = 4, dislikes = [[1,2],[1,3],[2,4]]

**Output:** true

**Explanation**: group1 [1,4], group2 [2,3]

**Example 2:**

**Input:** N = 3, dislikes = [[1,2],[1,3],[2,3]]

**Output:** false

**Example 3:**

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

**Output:** false

**Constraints:**

* 1 <= N <= 2000
* 0 <= dislikes.length <= 10000
* dislikes[i].length == 2
* 1 <= dislikes[i][j] <= N
* dislikes[i][0] < dislikes[i][1]
* There does not exist i != j for which dislikes[i] == dislikes[j].

## Solution

#### **Approach 1: Depth-First Search**

**Intuition**

It's natural to try to assign everyone to a group. Let's say people in the first group are red, and people in the second group are blue.

If the first person is red, anyone disliked by this person must be blue. Then, anyone disliked by a blue person is red, then anyone disliked by a red person is blue, and so on.

If at any point there is a conflict, the task is impossible, as every step logically follows from the first step. If there isn't a conflict, then the coloring was valid, so the answer would be true.

**Algorithm**

Consider the graph on N people formed by the given "dislike" edges. We want to check that each connected component of this graph is bipartite.

For each connected component, we can check whether it is bipartite by just trying to coloring it with two colors. How to do this is as follows: color any node red, then all of it's neighbors blue, then all of those neighbors red, and so on. If we ever color a red node blue (or a blue node red), then we've reached a conflict.

|  |
| --- |
| class Solution {  ArrayList<Integer>[] graph;  Map<Integer, Integer> color;  public boolean possibleBipartition(int N, int[][] dislikes) {  graph = new ArrayList[N+1];  for (int i = 1; i <= N; ++i)  graph[i] = new ArrayList();  for (int[] edge: dislikes) {  graph[edge[0]].add(edge[1]);  graph[edge[1]].add(edge[0]);  }  color = new HashMap();  for (int node = 1; node <= N; ++node)  if (!color.containsKey(node) && !dfs(node, 0))  return false;  return true;  }  public boolean dfs(int node, int c) {  if (color.containsKey(node))  return color.get(node) == c;  color.put(node, c);  for (int nei: graph[node])  if (!dfs(nei, c ^ 1))  return false;  return true;  }  } |

**Complexity Analysis**

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

**Counting Bits**

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

**Example 1:**

**Input:** 2

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

**Example 2:**

**Input:** 5

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

**Follow up:**

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

   Hide Hint #1

You should make use of what you have produced already.

   Hide Hint #2

Divide the numbers in ranges like [2-3], [4-7], [8-15] and so on. And try to generate new range from previous.

   Hide Hint #3

Or does the odd/even status of the number help you in calculating the number of 1s?

## Summary

This article is for intermediate readers. It relates to the following ideas: Pop Count, Most Significant Bit, Least Significant Bit, Last Set Bit and Dynamic Programming.

## Solutions

#### **Approach #1 Pop Count [Accepted]**

**Intuition**

Solve the problem for one number and applies that for all.

**Algorithm**

This problem can be seen as a follow-up of the [Problem 191 The number of 1 bits](https://leetcode.com/problems/number-of-1-bits/). It counts the bits for an unsigned integer. The number is often called pop count or [Hamming weight](https://en.wikipedia.org/wiki/Hamming_weight). See the editorial of [Problem 191 The number of 1 bits](https://leetcode.com/problems/number-of-1-bits/) for a detailed explanation of different approaches.

Now we just take that for granted. And suppose we have the function int popcount(int x) which will return the count of the bits for a given non-negative integer. We just loop through the numbers in range [0, num] and put the results in a list.

|  |
| --- |
| public class Solution {  public int[] countBits(int num) {  int[] ans = new int[num + 1];  for (int i = 0; i <= num; ++i)  ans[i] = popcount(i);  return ans;  }  private int popcount(int x) {  int count;  for (count = 0; x != 0; ++count)  x &= x - 1; //zeroing out the least significant nonzero bit  return count;  }  } |

**Complexity Analysis**

* Time complexity : O(nk)*O*(*nk*). For each integer x*x*, we need O(k)*O*(*k*) operations where k*k* is the number of bits in x*x*.
* Space complexity : O(n)*O*(*n*). We need O(n)*O*(*n*) space to store the count results. If we exclude that, it costs only constant space.

#### **Approach #2 DP + Most Significant Bit [Accepted]**

**Intuition**

Use previous count results to generate the count for a new integer.

**Algorithm**

Suppose we have an integer:

x = (1001011101)\_2 = (605)\_{10}*x*=(1001011101)2​=(605)10​

and we already calculated and stored all the results of 00 to x - 1*x*−1.

Then we know that x*x* is differ by one bit with a number we already calculated:

x' = (1011101)\_2 = (93)\_{10}*x*′=(1011101)2​=(93)10​

They are different only in the most significant bit.

Let's exam the range [0, 3][0,3] in the binary form:

(0) = (0)\_2(0)=(0)2​

(1) = (1)\_2(1)=(1)2​

(2) = (10)\_2(2)=(10)2​

(3) = (11)\_2(3)=(11)2​

One can see that the binary form of 2 and 3 can be generated by adding 1 bit in front of 0 and 1. Thus, they are different only by 1 regarding pop count.

Similarly, we can generate the results for [4, 7][4,7] using [0, 3][0,3] as blueprints.

In general, we have the following transition function for popcount P(x)*P*(*x*):

P(x + b) = P(x) + 1, b = 2^m > x*P*(*x*+*b*)=*P*(*x*)+1,*b*=2*m*>*x*

With this transition function, we can then apply Dynamic Programming to generate all the pop counts starting from 00.

public class Solution {

public int[] countBits(int num) {

int[] ans = new int[num + 1];

int i = 0, b = 1;

// [0, b) is calculated

while (b <= num) {

// generate [b, 2b) or [b, num) from [0, b)

while(i < b && i + b <= num){

ans[i + b] = ans[i] + 1;

++i;

}

i = 0; // reset i

b <<= 1; // b = 2b

}

return ans;

}

}

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). For each integer x*x* we need constant operations which do not depend on the number of bits in x*x*.
* Space complexity : O(n)*O*(*n*). We need O(n)*O*(*n*) space to store the count results. If we exclude that, it costs only constant space.

#### **Approach #3 DP + Least Significant Bit [Accepted]**

**Intuition**

We can have different transition functions, as long as x'*x*′ is smaller than x*x* and their pop counts have a function.

**Algorithm**

Following the same principle of the previous approach, we can also have a transition function by playing with the least significant bit.

Let look at the relation between x*x* and x' = x / 2*x*′=*x*/2

*x*=(1001011101)2​=(605)10​

*x*′=(100101110)2​=(302)10​

We can see that x'*x*′ is differ than x*x* by one bit, because x'*x*′ can be considered as the result of removing the least significant bit of x*x*.

Thus, we have the following transition function of pop count P(x)*P*(*x*):

*P*(*x*)=*P*(*x*/2)+(*x*mod2)

|  |
| --- |
| public class Solution {  public int[] countBits(int num) {  int[] ans = new int[num + 1];  for (int i = 1; i <= num; ++i)  ans[i] = ans[i >> 1] + (i & 1); // x / 2 is x >> 1 and x % 2 is x & 1  return ans;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). For each integer x*x* we need constant operations which do not depend on the number of bits in x*x*.
* Space complexity : O(n)*O*(*n*). Same as approach #2.

#### **Approach #4 DP + Last Set Bit [Accepted]**

**Algorithm**

With the same logic as previous approaches, we can also manipulate the last set bit.

Last set bit is the rightmost set bit. Setting that bit to zero with the bit trick, x &= x - 1, leads to the following transition function:

*P*(*x*)=*P*(*x*&(*x*−1))+1;

|  |
| --- |
| public class Solution {  public int[] countBits(int num) {  int[] ans = new int[num + 1];  for (int i = 1; i <= num; ++i)  ans[i] = ans[i & (i - 1)] + 1;  return ans;  }  } |

**Complexity Analysis**

* Time complexity : O(n)*O*(*n*). Same as approach #3.
* Space complexity : O(n)*O*(*n*). Same as approach #3.