**September 2020 Challenge**

Week 1: September 1st - September 7th

~~+35~~

 Read N Characters Given Read4

~~+10~~

**Largest Time for Given Digits**

~~+10~~

 Contains Duplicate III

~~+10~~

**Repeated Substring Pattern**

~~+10~~

 Partition Labels

~~+10~~

**All Elements in Two Binary Search Trees**

~~+10~~

**Image Overlap**

~~+10~~

**Word Pattern**

Week 2: September 8th - September 14th

~~+35~~

 Moving Average from Data Stream

~~+10~~

**Sum of Root To Leaf Binary Numbers**

~~+10~~

 Compare Version Numbers

~~+10~~

 Bulls and Cows

~~+10~~

 Maximum Product Subarray

~~+10~~

**Combination Sum III**

~~+10~~

 Insert Interval

~~+10~~

 House Robber

Week 3: September 15th - September 21st

~~+35~~

**Inorder Successor in BST II**

~~+10~~

**Length of Last Word**

~~+10~~

 Maximum XOR of Two Numbers in an Array

~~+10~~

**Robot Bounded In Circle**

~~+10~~

 Best Time to Buy and Sell Stock

~~+10~~

**Sequential Digits**

~~+10~~

**Unique Paths III**

~~+10~~

**Car Pooling**

Week 4: September 22nd - September 28th

~~+35~~

**Insert into a Sorted Circular Linked List**

~~+10~~

 Majority Element II

~~+10~~

**Gas Station**

~~+10~~

**Find the Difference**

~~+10~~

 Largest Number

~~+10~~

**Teemo Attacking**

~~+10~~

 Evaluate Division

~~+10~~

**Subarray Product Less Than K**

Week 5: September 29th - September 30th

~~+35~~

 Word Squares

~~+10~~

 Word Break

~~+10~~

 First Missing Positive

**Largest Time for Given Digits**

Given an array arr of 4 digits, find the latest 24-hour time that can be made using each digit **exactly once**.

24-hour times are formatted as "HH:MM", where HH is between 00 and 23, and MM is between 00 and 59. The earliest 24-hour time is 00:00, and the latest is 23:59.

Return *the latest 24-hour time in "HH:MM" format*.  If no valid time can be made, return an empty string.

**Example 1:**

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

**Output:** "23:41"

**Explanation:** The valid 24-hour times are "12:34", "12:43", "13:24", "13:42", "14:23", "14:32", "21:34", "21:43", "23:14", and "23:41". Of these times, "23:41" is the latest.

**Example 2:**

**Input:** A = [5,5,5,5]

**Output:** ""

**Explanation:** There are no valid 24-hour times as "55:55" is not valid.

**Example 3:**

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

**Output:** "00:00"

**Example 4:**

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

**Output:** "10:00"

**Constraints:**

* arr.length == 4
* 0 <= arr[i] <= 9

Solution

Overview

If we add one statement in the problem description that one could use the utility function in their preferred programming language to generate all permutations from a given list, *i.e.* itertools.permutations(list) in Python, then many of you would agree that this is definitely an **easy** problem.

Indeed, if we have all the permutations from the given list of digits, we could simply ***enumerate*** each of the permutations to see if we could build a valid time.

At the end of the enumeration, it would be easy to get the maximum of all valid times.

In the following sections, we would first present a solution with the built-in ***permutation*** utility.

Then, in case the interviewer insists that one should not use the permutation utility, we would present another solution where we ***hand-craft*** the permutations via enumeration.

At the end, we would also present a more *generic* and *efficient* algorithm to generate permutations via **backtracking**.

Approach 1: Enumerate the Permutations

**Intuition**

As we stated before, once we have the permutations at our disposal, the idea is simple: we iterate through all possible permutations of the given 4 digits, and for each permutation, we check if we could build a time out of it in the 24H format (*i.e.* HH:MM).

There are two conditions that we should meet, in order to construct a valid time format:

* HH < 24: The first two digits, *i.e.* the hour, should be less than 24.
* MM < 60: The last two digits, *i.e.* the minute, should be less than 60.

**Algorithm**

* The algorithm can be implemented in a single loop over all the possible permutations for the given 4 digits.
* At each iteration, we check if we could build a valid time based on the conditions we presented before.
* Meanwhile, we use a variable (*i.e.*max\_time) to keep track of the maximum valid time that we've seen during the iteration.

|  |
| --- |
| class Solution {  public:  string largestTimeFromDigits(vector<int>& A) {  int max\_time = -1;  // prepare for the generation of permutations next.  std::sort(A.begin(), A.end());  do {  int hour = A[0] \* 10 + A[1];  int minute = A[2] \* 10 + A[3];  if (hour < 24 && minute < 60) {  int new\_time = hour \* 60 + minute;  max\_time = new\_time > max\_time ? new\_time : max\_time;  }  } while(next\_permutation(A.begin(), A.end()));  if (max\_time == -1) {  return "";  } else {  std::ostringstream strstream;  strstream << std::setw(2) << std::setfill('0') << max\_time / 60  << ":" << std::setw(2) << std::setfill('0') << max\_time % 60;  return strstream.str();  }  }  }; |

**Note:**

* We did not provide a solution in Java, since in Java we don't have a built-in function that can do the permutation.
* Both the [itertools.permutations](https://docs.python.org/3/library/itertools.html#itertools.permutations) API in Python and the [next\_permutation()](https://en.cppreference.com/w/cpp/algorithm/next_permutation) in C++ can generate the permutations in ***lexicographic*** ordering. As a result, one can order the input array in descending order, rather than iterating all possible permutations, one can have an **early stop** as soon as we find the first valid time, which would also be the largest one, since the permutations are generated in lexicographic ordering.

**Complexity Analysis**

* Time Complexity: \mathcal{O}(1)O(1)
  + For an array of length N*N*, the number of permutations would be N!*N*!. In our case, the input is an array of 4 digits. Hence, the number of permutations would be 4! = 4 \* 3 \* 2 \* 1 = 244!=4∗3∗2∗1=24.
  + Since the length of the input array is fixed, it would take the same constant time to generate its permutations, regardless the content of the array. Therefore, the time complexity to generate the permutations would be \mathcal{O}(1)O(1).
  + In the above program, each iteration takes a constant time to process. Since the total number of permutations is fixed (constant), the time complexity of the loop in the algorithm is constant as well, i.e. 24 \cdot \mathcal{O}(1) = \mathcal{O}(1)24⋅O(1)=O(1).
  + To sum up, the overall time complexity of the algorithm would be \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)O(1)+O(1)=O(1).
* Space Complexity: \mathcal{O}(1)O(1)
  + In the algorithm, we keep the permutations for the input digits, which are in total 24, i.e. a constant number regardless the input.

#### Approach 2: Permutation via Enumeration

**Intuition**

We are asked to generate permutations of four digits, i.e. D\_1D\_2D\_3D\_4*D*1​*D*2​*D*3​*D*4​ from a given array e.g. A=[1, 2, 3, 4].

Each of the digit in the permutation, say D\_1*D*1​, can come from any of the elements in the input array, e.g. D\_1 = 1*D*1​=1 or D\_1 = 2*D*1​=2 etc.

An intuitive idea would be that we can run nested loops to generate combination of digits, one loop per digit. At the end of loops, we **filter** out those invalid combinations, i.e. combinations that contains duplicate elements from the input array. The remaining combinations are actually permutations by definition.

**Algorithm**

* Normally, we should have 4 nested loops, one loop per digit. But once we choose 3 non-duplicate elements from the input array, the last digit of the permutation is then fixed. As a result, we could reduce the loops down to 3-level nested loops, rather than 4.
* At the end of loops, we check if we could build a valid time out of the permutation we generate.
* Meanwhile, we use a variable (i.e.max\_time) to keep track of the maximum valid time that we've seen during the iteration.

|  |
| --- |
| class Solution {  private int max\_time = -1;  public String largestTimeFromDigits(int[] A) {  this.max\_time = -1;  for (int i1 = 0; i1 < A.length; ++i1)  for (int i2 = 0; i2 < A.length; ++i2)  for (int i3 = 0; i3 < A.length; ++i3) {  // skip duplicate elements  if (i1 == i2 || i2 == i3 || i1 == i3)  continue;  // the total sum of indices is 0 + 1 + 2 + 3 = 6.  int i4 = 6 - i1 - i2 - i3;  int [] perm = {A[i1], A[i2], A[i3], A[i4]};  // check if the permutation can form a time  validateTime(perm);  }  if (this.max\_time == -1)  return "";  else  return String.format("%02d:%02d", max\_time / 60, max\_time % 60);  }  protected void validateTime(int[] perm) {  int hour = perm[0] \* 10 + perm[1];  int minute = perm[2] \* 10 + perm[3];  if (hour < 24 && minute < 60)  this.max\_time = Math.max(this.max\_time, hour\*60 + minute);  }  } |

**Complexity Analysis**

* Time Complexity: \mathcal{O}(1)O(1)
  + We have a 3-level nested loops, each loop would have 4 iterations. As a result, the total number of iterations is 4 \* 4 \* 4 = 644∗4∗4=64.
  + Since the length of the input array is fixed, it would take the same constant time to generate its permutations, regardless the content of the array. Therefore, the time complexity to generate the permutations would be \mathcal{O}(1)O(1).
  + Note that the total number of permutations is 4! = 4 \* 3 \* 2 \* 1 = 244!=4∗3∗2∗1=24. Yet, it takes us 64 iterations to generate the permutations, which is not the most efficient algorithm as one can see. As the size of array grows, this discrepancy would grow exponentially.
* Space Complexity: \mathcal{O}(1)O(1)
  + In the algorithm, we keep a variable to keep track of the maximum time, as well as some intermediates variables for the function. Since the size of the input array is fixed, the total size of the local variables are bounded as well.

#### Approach 3: Permutation via Backtracking

**Intuition**

As we discussed before, the **hard** part of the problem is not enumerating over the permutations, but actually constructing the permutations itself. For practice, one can implement the permutation algorithms on these two problem: [permutations](https://leetcode.com/problems/permutations/) and [next permutation](https://leetcode.com/problems/next-permutation/).

In the previous approach, we've presented a naive way to implement the permutation, which is not the most efficient algorithm obviously.

There have been several classic algorithms to generate the permutations. For instance, B.R. Heap proposed an algorithm (named [Heap's algorithm](https://en.wikipedia.org/wiki/Heap%27s_algorithm)) in 1963, which minimizes the movements of elements. It was still considered as the most efficient algorithm later in 1977.

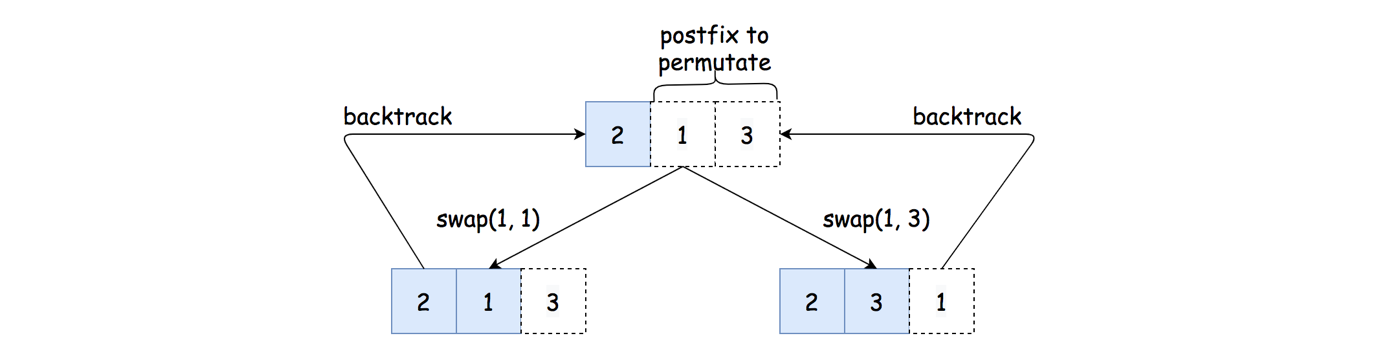
Here we present an algorithm, which might not be the most efficient one but arguably more intuitive.

It is based on the ideas of **divide-and-conquer**, **swapping** and **backtracking**.

* First of all, the algorithm follows the paradigm of **divide and conquer**. Given an array A[0:n], once we fix on the arrangements of the prefix subarray A[0:i], we then reduce the problem down to a subproblem, i.e. generating the permutations for the postfix subarray A[i:n].
* In order to fix on a prefix subarray, we apply the operation of **swapping**, where we swap the elements between a fixed position and an alternative position.



* Finally, once we explore the permutations after a swapping operation, we then revert the choice (i.e. **backtracking**) by performing the same swapping, so that we could have a clean slate to start all over again.

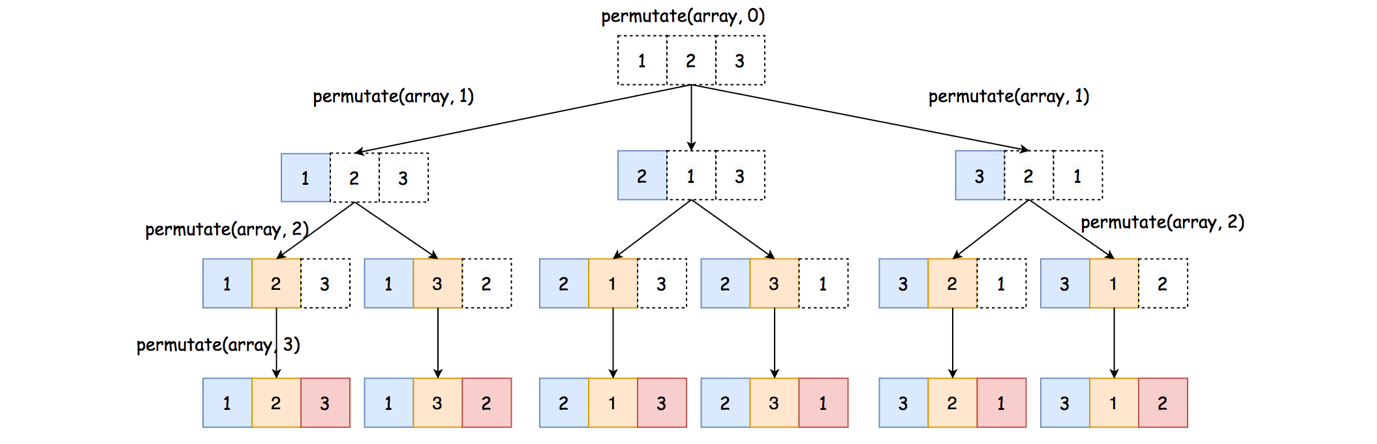


**Algorithm**

Now we can put together all the ideas that we presented before, and implement the permutation algorithm.

Here we implement the permutation algorithm as the function permutate(array, start) which generates the permutations for the postfix subarray of array[start:len(array)]. Once we implement the function, we invocate it as permutate(array, 0) to generate all the permutations from the array.

As a preview, once implemented, the function will unfold itself as in the following example.



For instance, starting from the root node, first we try to fix on the first element in the final combination, which we try to switch the element between the first position in the array and each of the positions in the array. Since there are 3 possible candidates, we branch out in 3 directions from the root node.

The function can be implemented in **recursion**, due to its nature of divide-and-conquer and backtracking.

* The base case of the function would be start == len(array), where we've fixed all the prefixes and reached the end of the combination. In this case, we simply add the current array as one of the results of combination.
* When we still have some postfix that need to be permutated, i.e. start < len(array), we then apply backtracking to try out all possible permutations for the postfixes, i.e. permutate(array, start+1). More importantly, we need to swap the start element with each of the elements following the start index (including the start element). The goal is two-fold: 1). we generate different prefixes for the final combination; 2). we generate different lists of candidates in the postfixes, so that the permutations generated from the postfixes would vary as well.
* At the end of backtracking, we will swap the start element back to its original position, so that we can try out other alternatives.
* For each permutation, we apply the same logic as in the previous approach, i.e. check if the permutation is of valid time and update the maximum time.

|  |
| --- |
| class Solution {  private int max\_time = -1;  public String largestTimeFromDigits(int[] A) {  this.max\_time = -1;  permutate(A, 0);  if (this.max\_time == -1)  return "";  else  return String.format("%02d:%02d", max\_time / 60, max\_time % 60);  }  protected void permutate(int[] array, int start) {  if (start == array.length) {  this.build\_time(array);  return;  }  for (int i = start; i < array.length; ++i) {  this.swap(array, i, start);  this.permutate(array, start + 1);  this.swap(array, i, start);  }  }  protected void build\_time(int[] perm) {  int hour = perm[0] \* 10 + perm[1];  int minute = perm[2] \* 10 + perm[3];  if (hour < 24 && minute < 60)  this.max\_time = Math.max(this.max\_time, hour \* 60 + minute);  }  protected void swap(int[] array, int i, int j) {  if (i != j) {  int temp = array[i];  array[i] = array[j];  array[j] = temp;  }  }  } |

**Complexity Analysis**

* Time Complexity: \mathcal{O}(1)O(1)
  + Since the length of the input array is fixed, it would take the same constant time to generate its permutations, regardless the content of the array. Therefore, the time complexity to generate the permutations would be \mathcal{O}(1)O(1).
  + Therefore, same as the previous approach, the overall time complexity of the algorithm would be \mathcal{O}(1)O(1).
* Space Complexity: \mathcal{O}(1)O(1)
  + In the algorithm, we keep the permutations for the input digits, which are in total 24, i.e. a constant number regardless the input.
  + Although the recursion in the algorithm could incur additional memory consumption in the function call stack, the maximal number of recursion is bounded by the size of the combination. Hence, the space overhead for the recursion in this problem is constant.

**Repeated Substring Pattern**

Given a string s, check if it can be constructed by taking a substring of it and appending multiple copies of the substring together.

**Example 1:**

**Input:** s = "abab"

**Output:** true

**Explanation:** It is the substring "ab" twice.

**Example 2:**

**Input:** s = "aba"

**Output:** false

**Example 3:**

**Input:** s = "abcabcabcabc"

**Output:** true

**Explanation:** It is the substring "abc" four times or the substring "abcabc" twice.

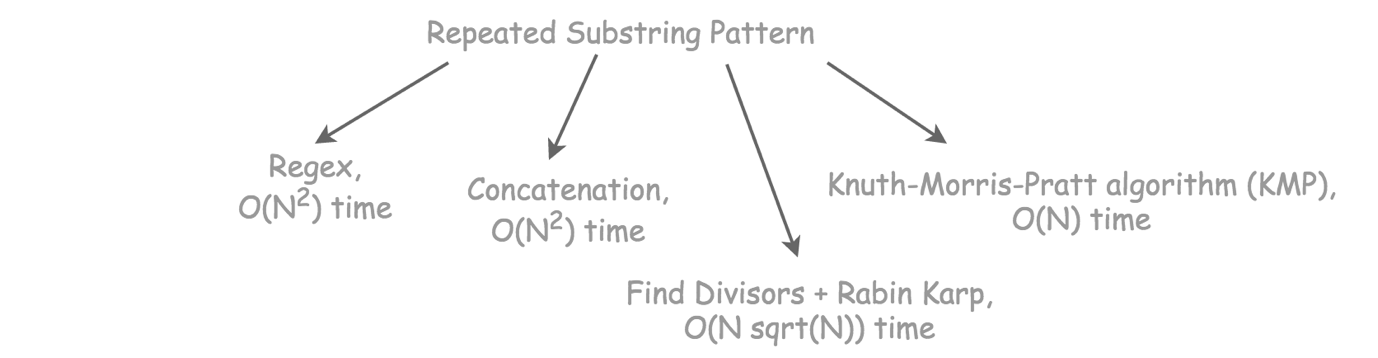
**Constraints:**

* 1 <= s.length <= 104
* s consists of lowercase English letters.

## Solution

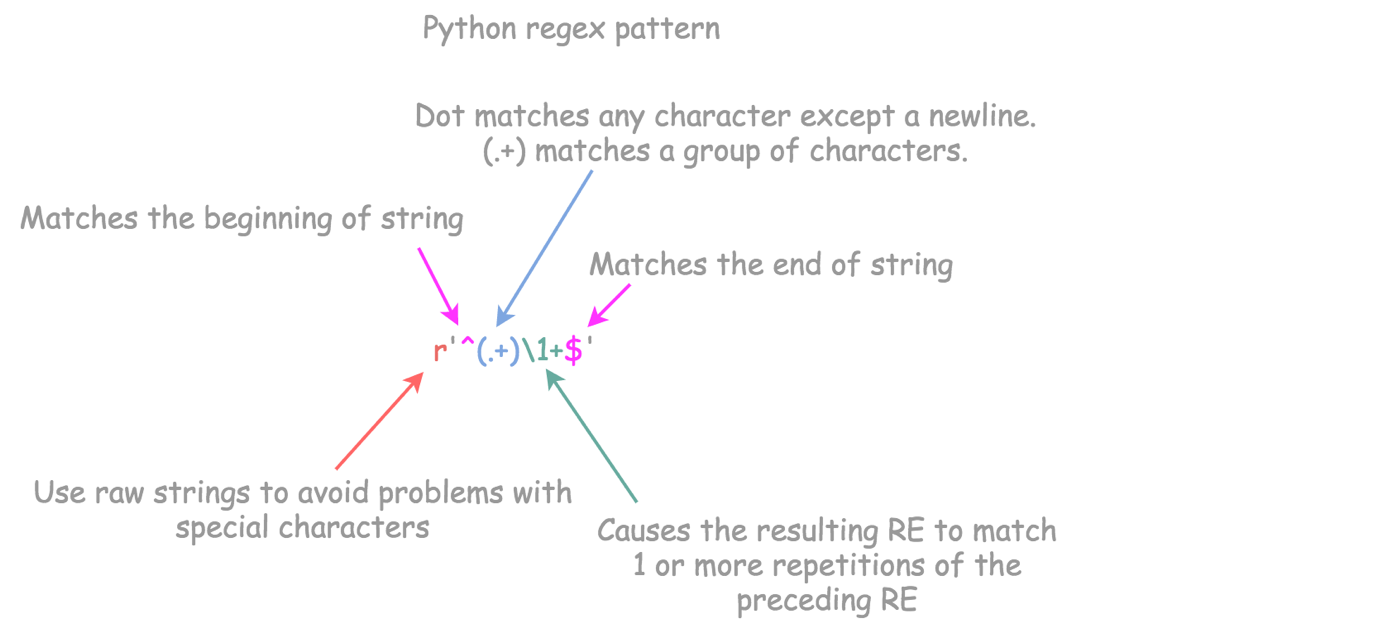
#### Overview

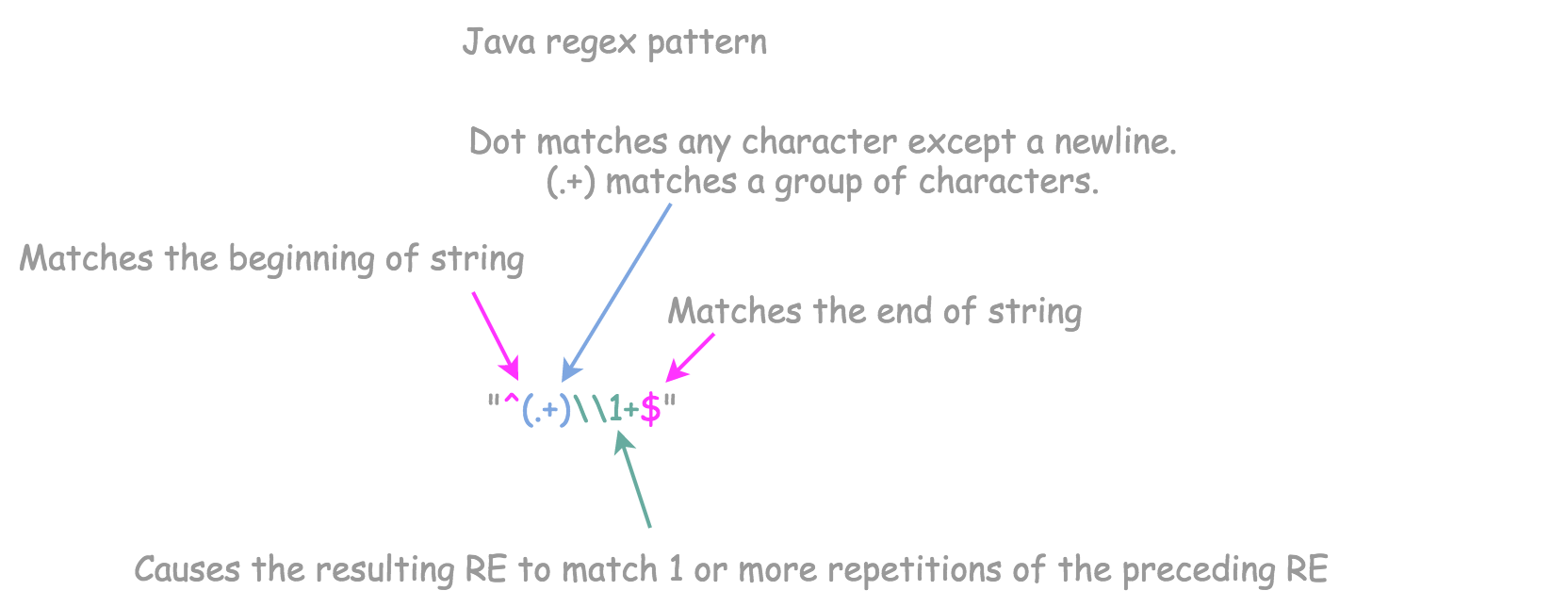
The problem could be solved in many ways. Easy approaches have \mathcal{O}(N^2)O(*N*2) time complexity, though one could improve it by using one of [string searching algorithms](https://en.wikipedia.org/wiki/String-searching_algorithm#Single-pattern_algorithms).



#### Approach 1: Regex

To use regex during the interviews is like to use built-in functions, the community has no single opinion about it yet, and it's a sort of risk.





**Implementation**

|  |
| --- |
| import java.util.regex.Pattern;  class Solution {  public boolean repeatedSubstringPattern(String s) {  Pattern pat = Pattern.compile("^(.+)\\1+$");  return pat.matcher(s).matches();  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N^2)O(*N*2) because we use greedy regex pattern. Once we have a +, the pattern is greedy.

The difference between the greedy and the non-greedy match is the following:

* + the non-greedy match will try to match as few repetitions of the quantified pattern as possible.
  + the greedy match will try to match as many repetitions as possible.

The worst-case situation here is to check all possible pattern lengths from N to 1 that would result in \mathcal{O}(N^2)O(*N*2) time complexity.

* Space complexity: \mathcal{O}(1)O(1). We don't use any additional data structures, and everything depends on internal regex implementation, which is evolving quite fast nowadays. [If you're interested to dig depeer, here is a famous article by Russ Cox](https://swtch.com/~rsc/regexp/regexp1.html) which inspired a lot of [discussions and code changes in Python community](https://mail.python.org/pipermail/python-ideas/2007-April/000407.html).

#### Approach 2: Concatenation

Repeated pattern string looks like PatternPattern, and the others like Pattern1Pattern2.

Let's double the input string:

PatternPattern --> PatternPatternPatternPattern

Pattern1Pattern2 --> Pattern1Pattern2Pattern1Pattern2

Now let's cut the first and the last characters in the doubled string:

PatternPattern --> \*atternPatternPatternPatter\*

Pattern1Pattern2 --> \*attern1Pattern2Pattern1Pattern\*

It's quite evident that if the new string contains the input string, the input string is a repeated pattern string.

**Implementation**

|  |
| --- |
| class Solution {  public boolean repeatedSubstringPattern(String s) {  return (s + s).substring(1, 2 \* s.length() - 1).contains(s);  }  } |

**Complexity Analysis**

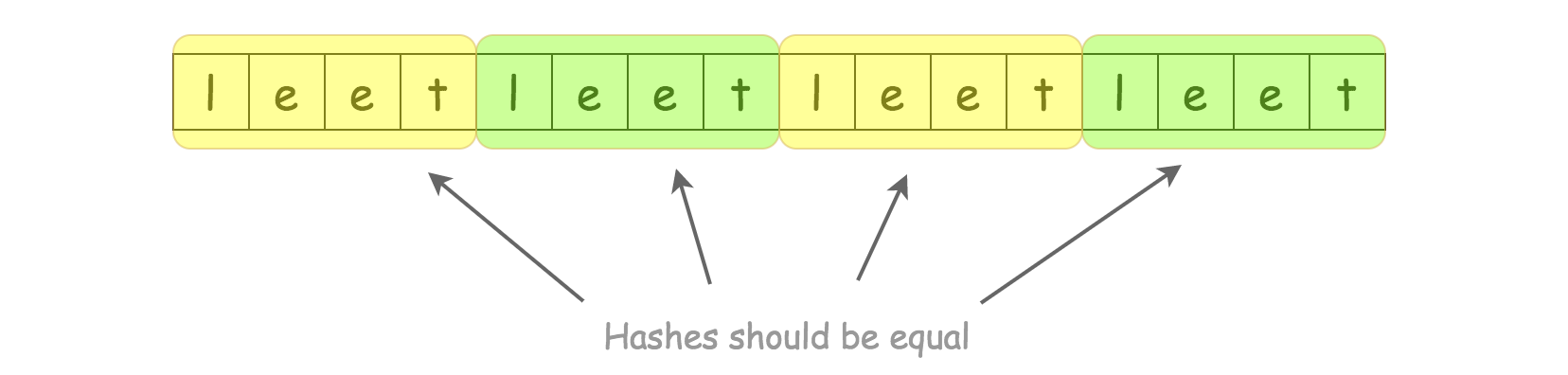
* Time complexity: \mathcal{O}(N^2)O(*N*2) because of the way in and contains are implemented.
* Space complexity: \mathcal{O}(N)O(*N*), the space is implicitly used to keep s + s string.

#### Approach 3: Find Divisors + Rabin-Karp

**Rabin-Karp**

Rabin-Karp is a linear-time \mathcal{O}(N)O(*N*) string searching algorithm:

* Move a sliding window of length L*L* along the string of length N*N*.
* Check hash of the string in the sliding window.



In some situations, one has to implement a particular hash algorithm to fit in a linear time, for example, we used rolling hash algorithm for the problem [Longest Duplicate Substring](https://leetcode.com/problems/longest-duplicate-substring/).

For the current problem the standard hash / hashCode is enough because the idea is to check only lengths L, which are divisors of N. This way we're not sliding, we're jumping:

* the first string is 0..L
* the second string is L..2L
* ...
* the last string is N - L..N

To copy characters in sliding window takes time L, to compute hash - time L as well. In total, there are N / L substrings, that makes it all work in a linear time \mathcal{O}(N)O(*N*).

**Find divisors**

Now the only problem is to find divisors of N. Let's iterate to the square root of N, and for each identified divisor i calculate the paired divisor N / i.

**Algorithm**

* Deal with base cases: n <= 2.
* Iterate from \sqrt{n}*n*​ to 1.
  + For each divisor n % i == 0:
    - Compute paired divisor n / i.
    - Use Rabin-Karp to check substrings of the lengths l = i and l = n / i:
      * Take as a reference hash first\_hash the hash of the first substring of length l.
      * Jump along the string with a step of length l while the hash of the current substring is equal to first\_hash.
      * If the hashes of all substrings along the way are equal, the input string consists of repeated patterns of length l. Return True.

Side note. The good practice is to verify the equality of two substrings after the hash match. This logic is not hard to add, and it could bring you kudos during the interview.

**Implementation**

|  |
| --- |
| class Solution {  public boolean repeatedSubstringPattern(String s) {  int n = s.length();  if (n < 2) return false;  if (n == 2) return s.charAt(0) == s.charAt(1);    for (int i = (int)Math.sqrt(n); i > 0; i--) {  if (n % i == 0) {  List<Integer> divisors = new ArrayList<>();  divisors.add(i);  if (i != 1) {  divisors.add(n / i);  }  for (int l : divisors) {  String tmp = s.substring(0, l);  int firstHash = tmp.hashCode();  int currHash = firstHash;  int start = l;  while (start != n && currHash == firstHash) {  tmp = s.substring(start, start + l);  currHash = tmp.hashCode();  start += l;  }  if (start == n && currHash == firstHash) {  return true;  }  }  }  }  return false;  }  } |

**Complexity Analysis**

* Time complexity: up to \mathcal{O}(N \sqrt{N})O(*NN*​). \mathcal{O}(\sqrt{N})O(*N*​) to compute all divisors and \mathcal{O}(N)O(*N*) for each divisor "verification". That's an upper-bound estimation because [divisor function grows slower than \sqrt{N}*N*​](https://en.wikipedia.org/wiki/Divisor_function#Growth_rate).
* Space complexity: up to \mathcal{O}(\sqrt{N})O(*N*​) to keep a copy of each substring during the hash computation.

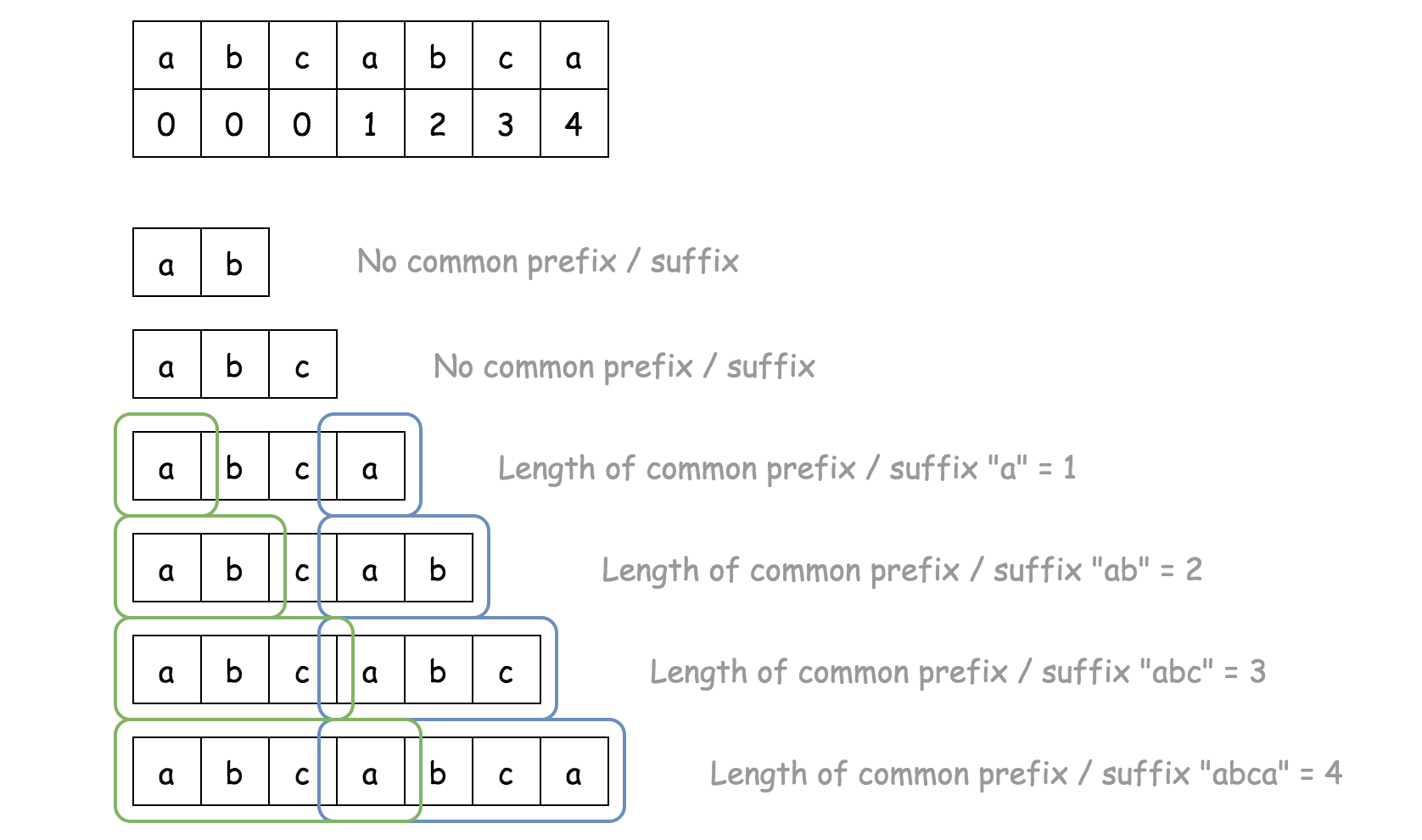
#### Approach 4: Knuth-Morris-Pratt Algorithm (KMP)

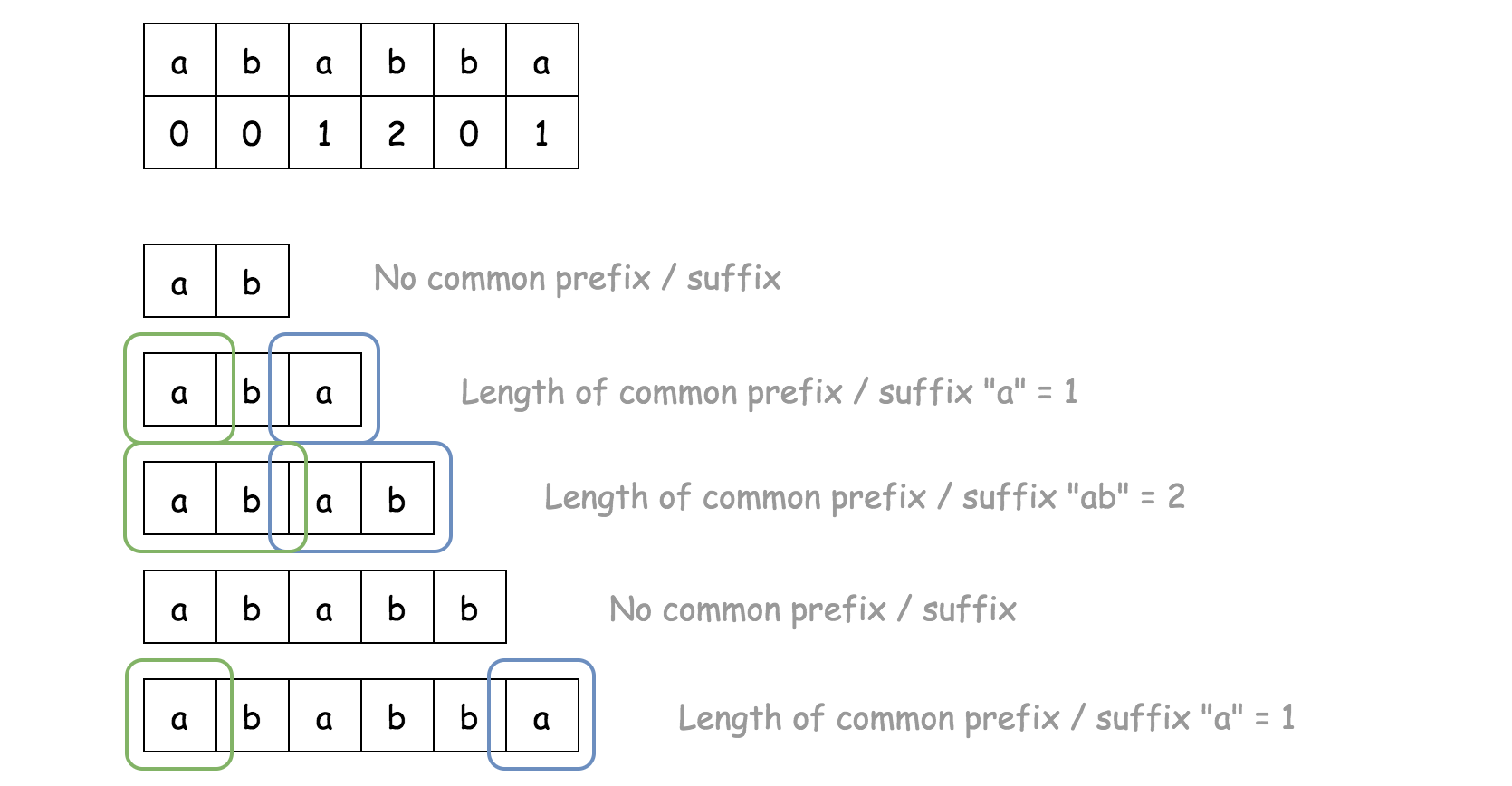
**Lookup Table**

Rabin-Karp is the best fit for the multiple pattern search, whereas KMP is typically used for the single pattern search.

The key to KMP is the partial match table, often called lookup table, or failure function table. **It stores the length of the longest prefix that is also a suffix.**

Here are two examples

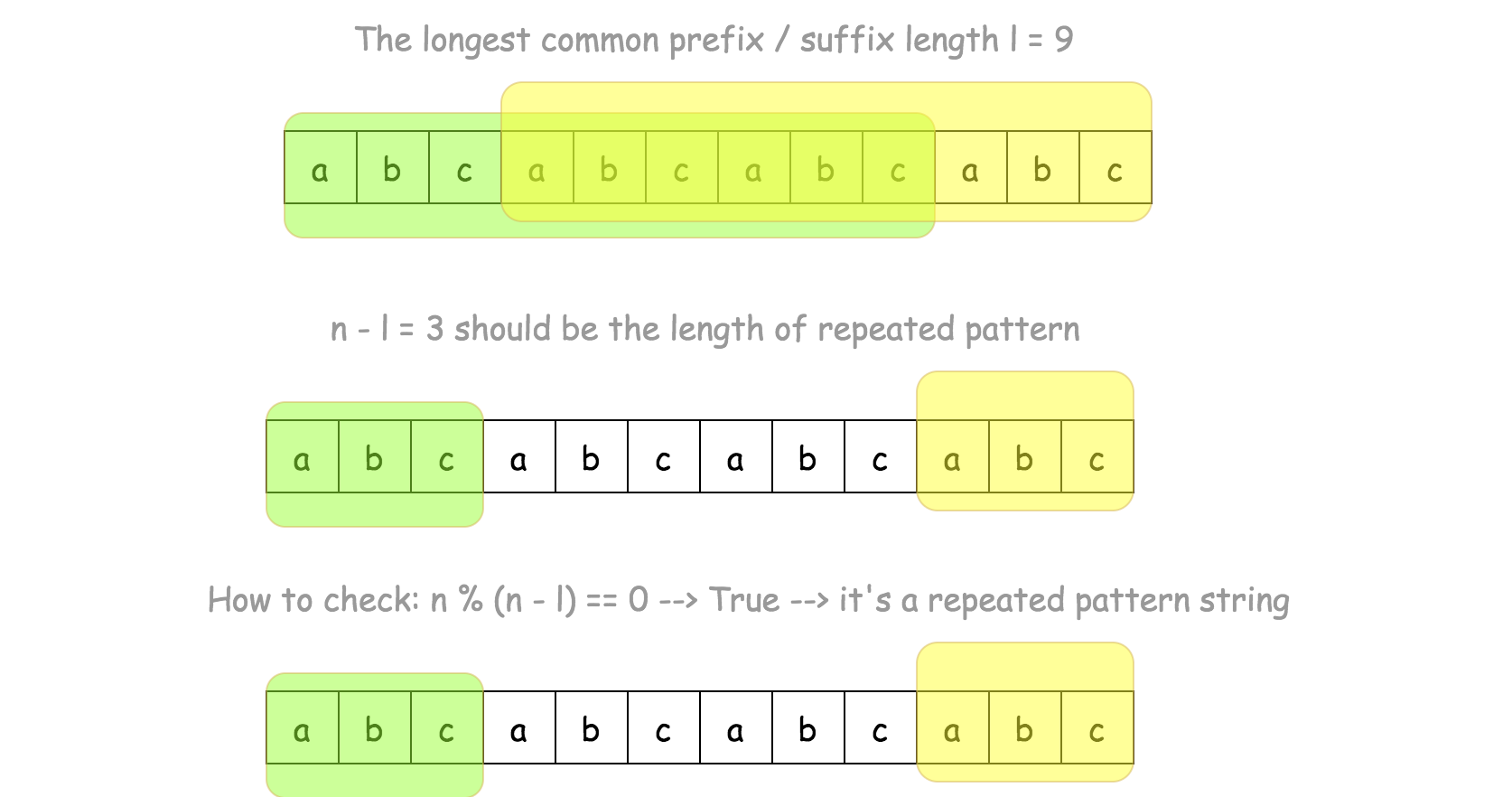


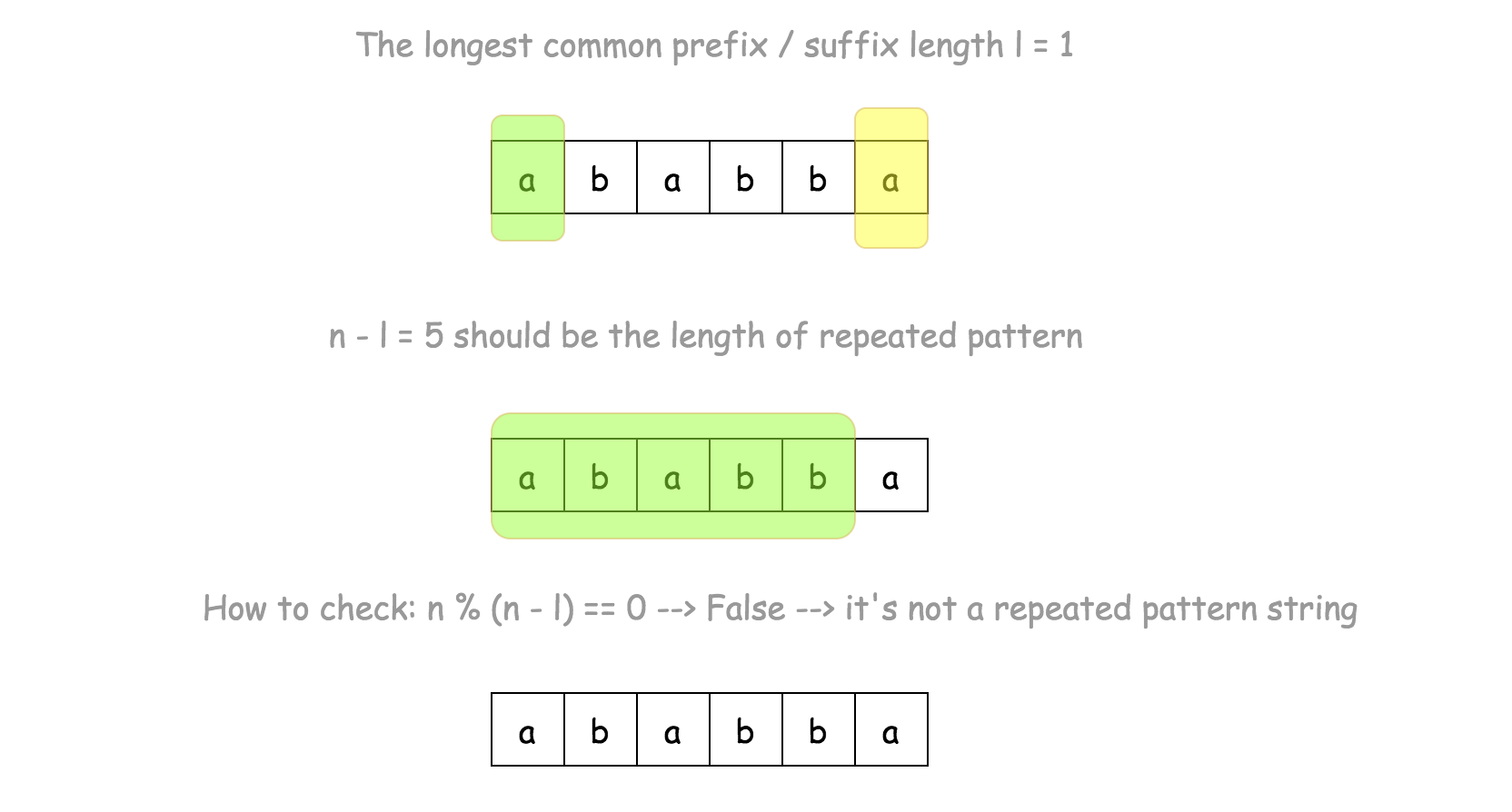


**How to Get an Answer**

Once we have a lookup table, we know the length l of common prefix/suffix for the input string: l = dp[n - 1].

That means that n - l should the length of the repeated sequence. To confirm that, one should verify if n - l is a divisor of n.





**Algorithm**

* Construct lookup table:
  + dp[0] = 0 since one character is not enough to speak about proper prefix / suffix.
  + Iterate over i from 1 to n:
    - Introduce the second pointer j = dp[i - 1].
    - While j > 0 and there is no match s[i] != s[j], do one step back to consider a shorter prefix: j = dp[j - 1].
    - If we found a match s[i] == s[j], move forward: j += 1
    - Write down the length of common prefix / suffix: dp[i] = j.
* Now we have a length of common prefix / suffix for the entire string: l = dp[n - 1].
* The string is a repeated pattern string if this length is nonzero and n - l is a divisor of n. Return l != 0 and n % (n - l) == 0.

**Implementation**

|  |
| --- |
| class Solution {  public boolean repeatedSubstringPattern(String s) {  int n = s.length();  int[] dp = new int[n];  // Construct partial match table (lookup table).  // It stores the length of the proper prefix that is also a proper suffix.  // ex. ababa --> [0, 0, 1, 2, 1]  // ab --> the length of common prefix / suffix = 0  // aba --> the length of common prefix / suffix = 1  // abab --> the length of common prefix / suffix = 2  // ababa --> the length of common prefix / suffix = 1  for (int i = 1; i < n; ++i) {  int j = dp[i - 1];  while (j > 0 && s.charAt(i) != s.charAt(j)) {  j = dp[j - 1];  }  if (s.charAt(i) == s.charAt(j)) {  ++j;  }  dp[i] = j;  }  int l = dp[n - 1];  // check if it's repeated pattern string  return l != 0 && n % (n - l) == 0;  }  } |

**Complexity Analysis**

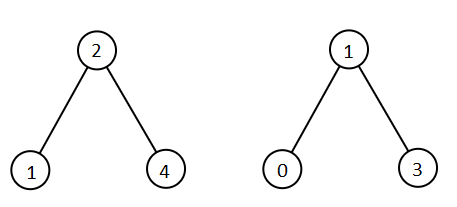
* Time complexity: \mathcal{O}(N)O(*N*). During the execution, j could be decreased at most N*N* times and then increased at most N*N* times, that makes overall execution time to be linear \mathcal{O}(N)O(*N*).
* Space complexity: \mathcal{O}(N)O(*N*) to keep the lookup table.

**All Elements in Two Binary Search Trees**

Given two binary search trees root1 and root2.

Return a list containing all the integers from both trees sorted in **ascending** order.

**Example 1:**



**Input:** root1 = [2,1,4], root2 = [1,0,3]

**Output:** [0,1,1,2,3,4]

**Example 2:**

**Input:** root1 = [0,-10,10], root2 = [5,1,7,0,2]

**Output:** [-10,0,0,1,2,5,7,10]

**Example 3:**

**Input:** root1 = [], root2 = [5,1,7,0,2]

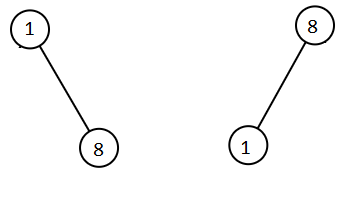
**Output:** [0,1,2,5,7]

**Example 4:**

**Input:** root1 = [0,-10,10], root2 = []

**Output:** [-10,0,10]

**Example 5:**



**Input:** root1 = [1,null,8], root2 = [8,1]

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

**Constraints:**

* Each tree has at most 5000 nodes.
* Each node's value is between [-10^5, 10^5].

   Hide Hint #1

Traverse the first tree in list1 and the second tree in list2.

   Hide Hint #2

Merge the two trees in one list and sort it.

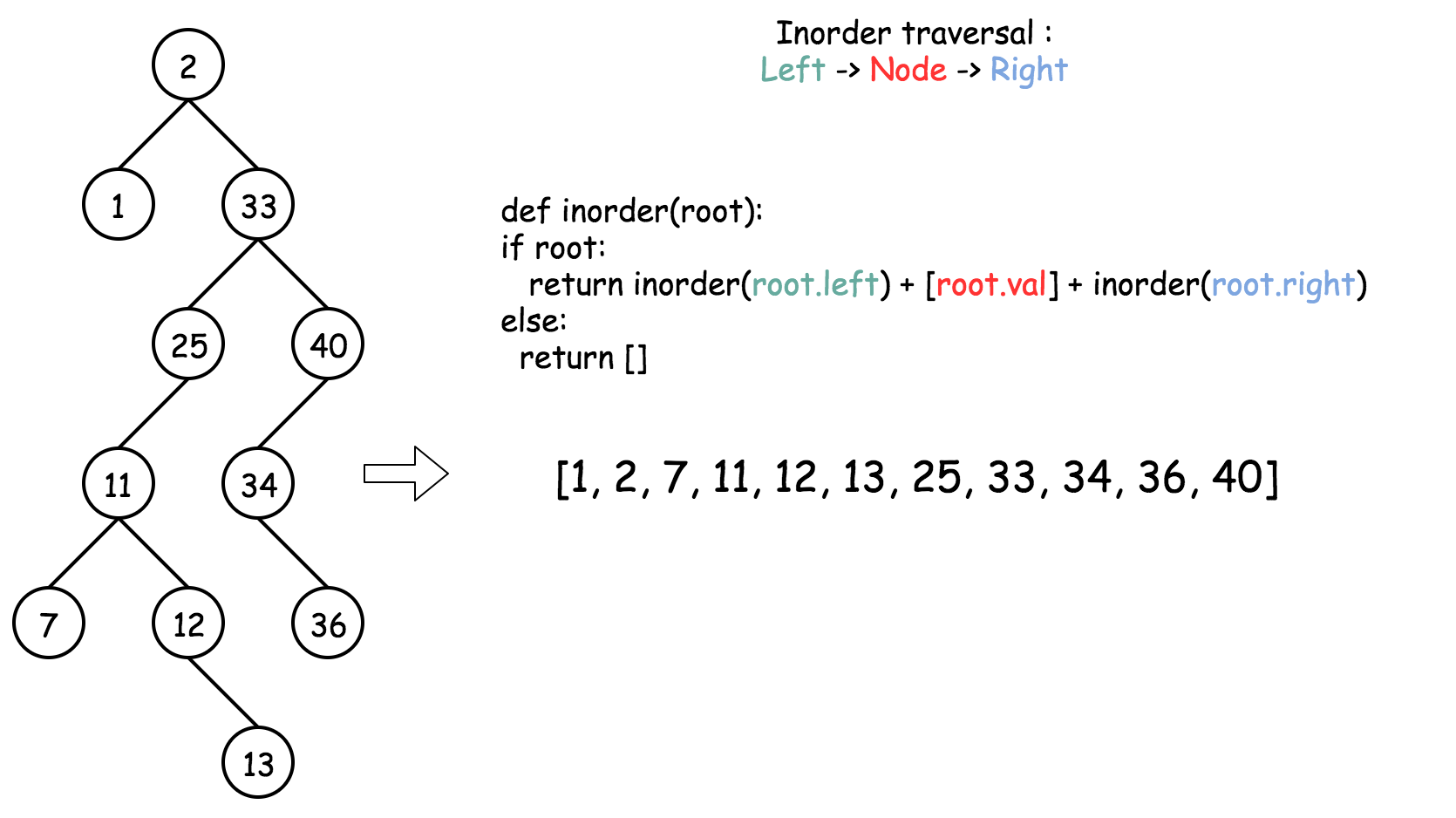
## Solution

#### Prerequisites

To solve this problem we will use recursive and iterative inorder traversals. Here are prerequisites you might want to check:

1. There are three DFS ways to traverse the tree: preorder, postorder and inorder. Please check two minutes picture explanation, if you don't remember them quite well: [here is Python version](https://leetcode.com/problems/binary-tree-inorder-traversal/discuss/283746/all-dfs-traversals-preorder-inorder-postorder-in-python-in-1-line) and [here is Java version](https://leetcode.com/problems/binary-tree-inorder-traversal/discuss/328601/all-dfs-traversals-preorder-postorder-inorder-in-java-in-5-lines).
2. Inorder traversal of BST is an array sorted in the ascending order.
3. To compute inorder traversal follow the direction Left -> Node -> Right.

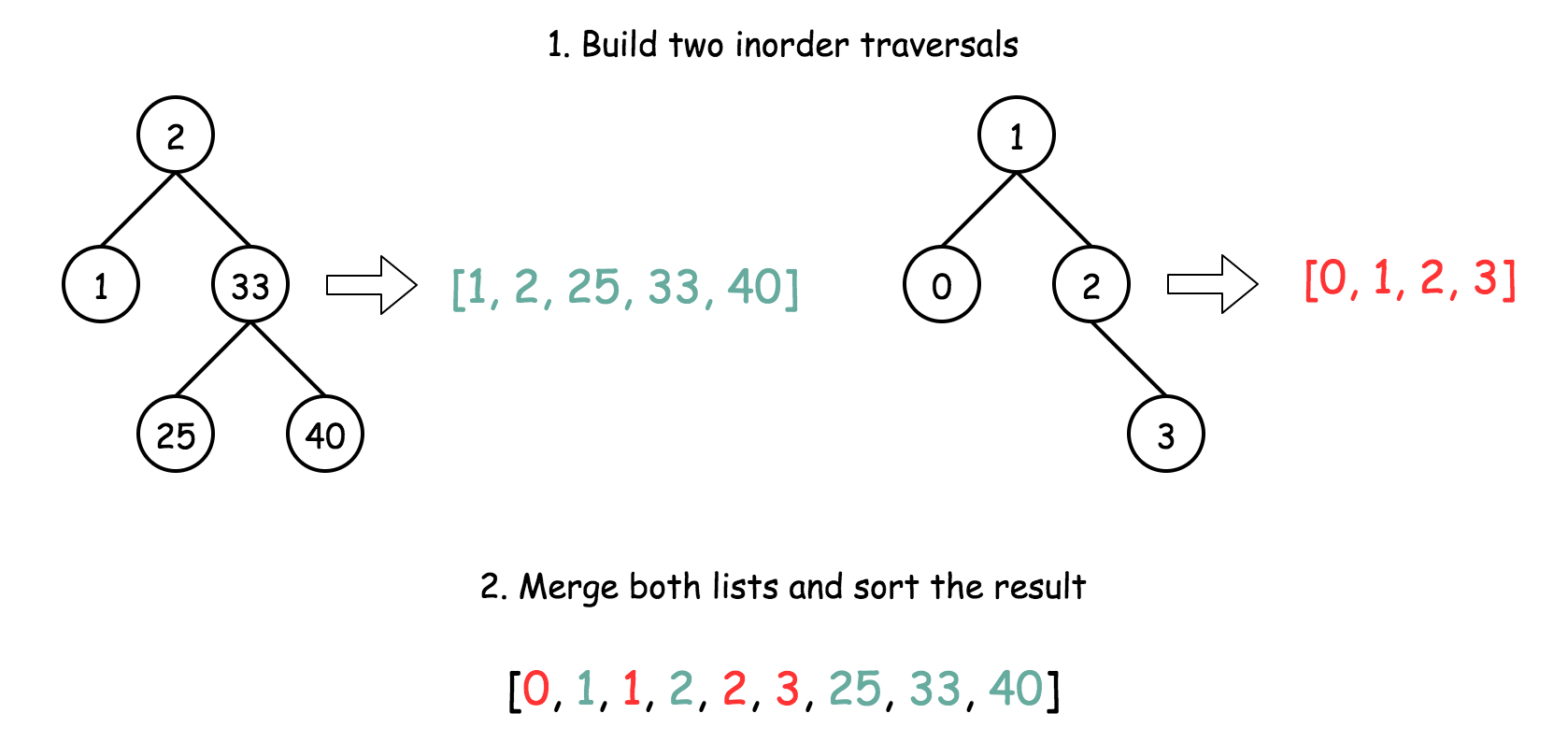
|  |
| --- |
| public List<Integer> inorder(TreeNode root, List<Integer> arr) {  if (root == null) return arr;  inorder(root.left, arr);  arr.add(root.val);  inorder(root.right, arr);  return arr;  } |



#### Approach 1: Recursive Inorder Traversal + Sort, Linearithmic Time.

Let's start from the shortest possible solution:

* Implement recursive inorder traversal, 1 line in Python, 5 lines in Java.
* Compute inorder traversal of each tree.
* Merge both lists and then sort the result.



This solution takes one minute to write, but the time complexity is linearithmic.

**Implementation**

|  |
| --- |
| /\*\*  \* Definition for a binary tree node.  \* public class TreeNode {  \* int val;  \* TreeNode left;  \* TreeNode right;  \* TreeNode(int x) { val = x; }  \* }  \*/  class Solution {  public List<Integer> inorder(TreeNode root, List<Integer> arr) {  if (root == null) return arr;  inorder(root.left, arr);  arr.add(root.val);  inorder(root.right, arr);  return arr;  }  public List<Integer> getAllElements(TreeNode root1, TreeNode root2) {  List<Integer> output = new ArrayList<>();  Stream.of(inorder(root1, new ArrayList()), inorder(root2, new ArrayList())).forEach(output::addAll);  Collections.sort(output);  return output;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}((N + M)\log(M + N))O((*N*+*M*)log(*M*+*N*)), where M*M* and N*N* are node numbers. To build inorder traversals takes \mathcal{O}(N + M)O(*N*+*M*), to merge and sort the resulting lists - \mathcal{O}((N + M)\log(M + N))O((*N*+*M*)log(*M*+*N*)).
* Space complexity: \mathcal{O}(N + M)O(*N*+*M*) to keep the output.

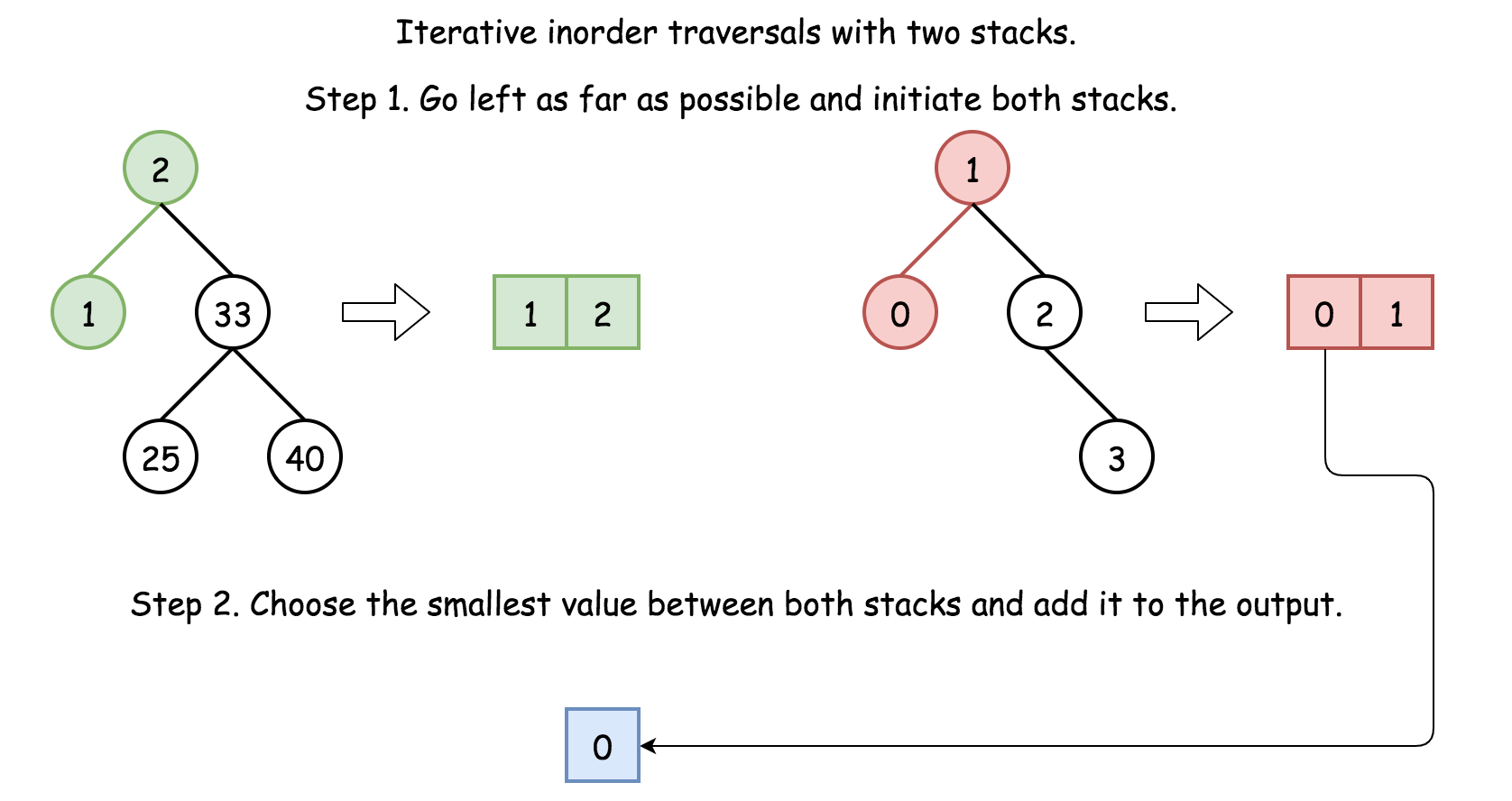
#### Approach 2: Iterative Inorder Traversal, One Pass, Linear Time.

**Intuition**

Now let's optimise the first approach.

First, since both inorder traversals are already sorted, [one could merge them into one sorted list in linear time](https://leetcode.com/articles/merged-two-sorted-lists/). Though it's still two passes solution: first to build two inorder traversals and then to merge them.

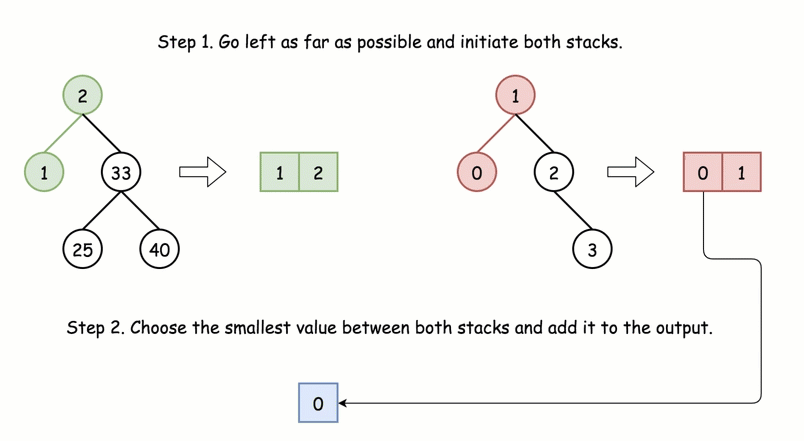
More elegant way here is to build iteratively inorder traversals for both trees in parallel, and at each step update the output list by the smallest value between both trees. That will be one pass solution. Here is how it works:



**Algorithm**

* Do iterative inorder traversal of both trees in parallel.
  + At each step add the smallest available value in the output.
* Return output list.

**Implementation**



|  |
| --- |
| /\*\*  \* Definition for a binary tree node.  \* public class TreeNode {  \* int val;  \* TreeNode left;  \* TreeNode right;  \* TreeNode(int x) { val = x; }  \* }  \*/  class Solution {  public List<Integer> getAllElements(TreeNode root1, TreeNode root2) {  ArrayDeque<TreeNode> stack1 = new ArrayDeque(), stack2 = new ArrayDeque();  List<Integer> output = new ArrayList();  while (root1 != null || root2 != null || !stack1.isEmpty() || !stack2.isEmpty()) {  // update both stacks  // by going left till possible  while (root1 != null) {  stack1.push(root1);  root1 = root1.left;  }  while (root2 != null) {  stack2.push(root2);  root2 = root2.left;  }  // Add the smallest value into output,  // pop it from the stack,  // and then do one step right  if (stack2.isEmpty() || !stack1.isEmpty() && stack1.getFirst().val <= stack2.getFirst().val) {  root1 = stack1.pop();  output.add(root1.val);  root1 = root1.right;  }  else {  root2 = stack2.pop();  output.add(root2.val);  root2 = root2.right;  }  }  return output;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N + M)O(*N*+*M*), where M*M* and N*N* are node numbers. It's one-pass approach along each tree.
* Space complexity: \mathcal{O}(N + M)O(*N*+*M*) to keep the output and both stacks.

#### Further Reading

Here we implemented recursive and iterative inorder traversals. There is also Morris inorder traversal, which is used for the problems which require constant space solution. Here is detailed comparison and implementation of all three inorder traversals: [Recover BST](https://leetcode.com/articles/recover-binary-search-tree/).

**Image Overlap**

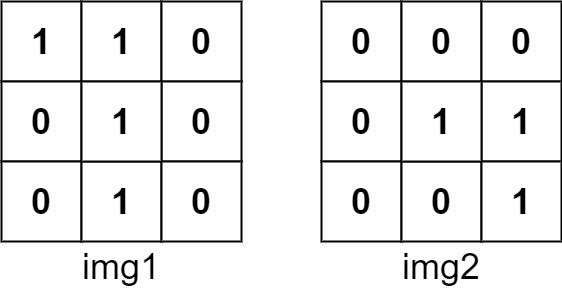
You are given two images img1 and img2 both of size n x n, represented as binary, square matrices of the same size. (A binary matrix has only 0s and 1s as values.)

We translate one image however we choose (sliding it left, right, up, or down any number of units), and place it on top of the other image.  After, the overlap of this translation is the number of positions that have a 1 in both images.

(Note also that a translation does **not** include any kind of rotation.)

What is the largest possible overlap?

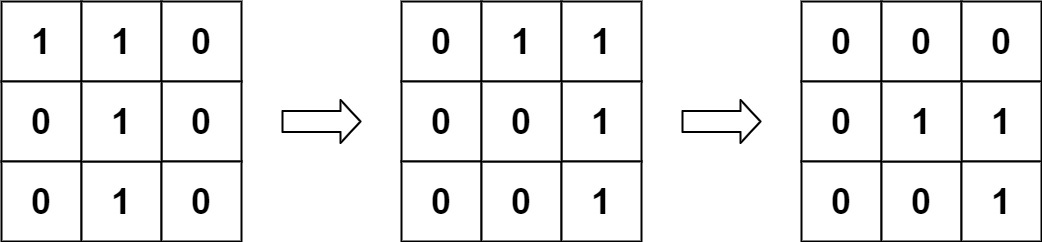
**Example 1:**



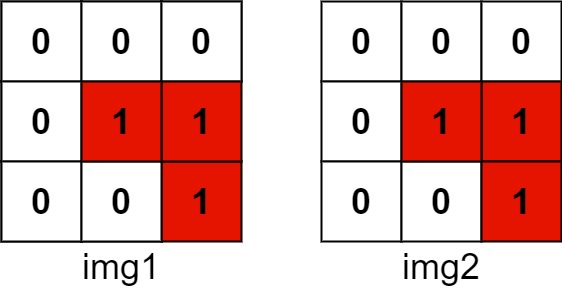
**Input:** img1 = [[1,1,0],[0,1,0],[0,1,0]], img2 = [[0,0,0],[0,1,1],[0,0,1]]

**Output:** 3

**Explanation:** We slide img1 to right by 1 unit and down by 1 unit.



The number of positions that have a 1 in both images is 3. (Shown in red)



**Example 2:**

**Input:** img1 = [[1]], img2 = [[1]]

**Output:** 1

**Example 3:**

**Input:** img1 = [[0]], img2 = [[0]]

**Output:** 0

**Constraints:**

* n == img1.length
* n == img1[i].length
* n == img2.length
* n == img2[i].length
* 1 <= n <= 30
* img1[i][j] is 0 or 1.
* img2[i][j] is 0 or 1.

## Solution

#### Overview

First of all, this is a really fun problem to solve, as one would discover later. In addition, it could be a practical problem in real world. For instance, if one can find the maximal overlapping zone between two images, one could clip the images to make them smaller and more focused.

In this article, we will cover three approaches as follows:

* We could solve the problem intuitively by enumerating all the possible overlapping zones.
* Or more efficiently, we can apply some knowledge of ***linear algebra*** (or geometry), as we will present another solution in the second approach.
* Finally, we could even solve it with the conception of **convolution**, as in Convolution Neural Network (a.k.a. CNN), which is the backbone operation for the image recognition models nowadays.

#### Approach 1: Shift and Count

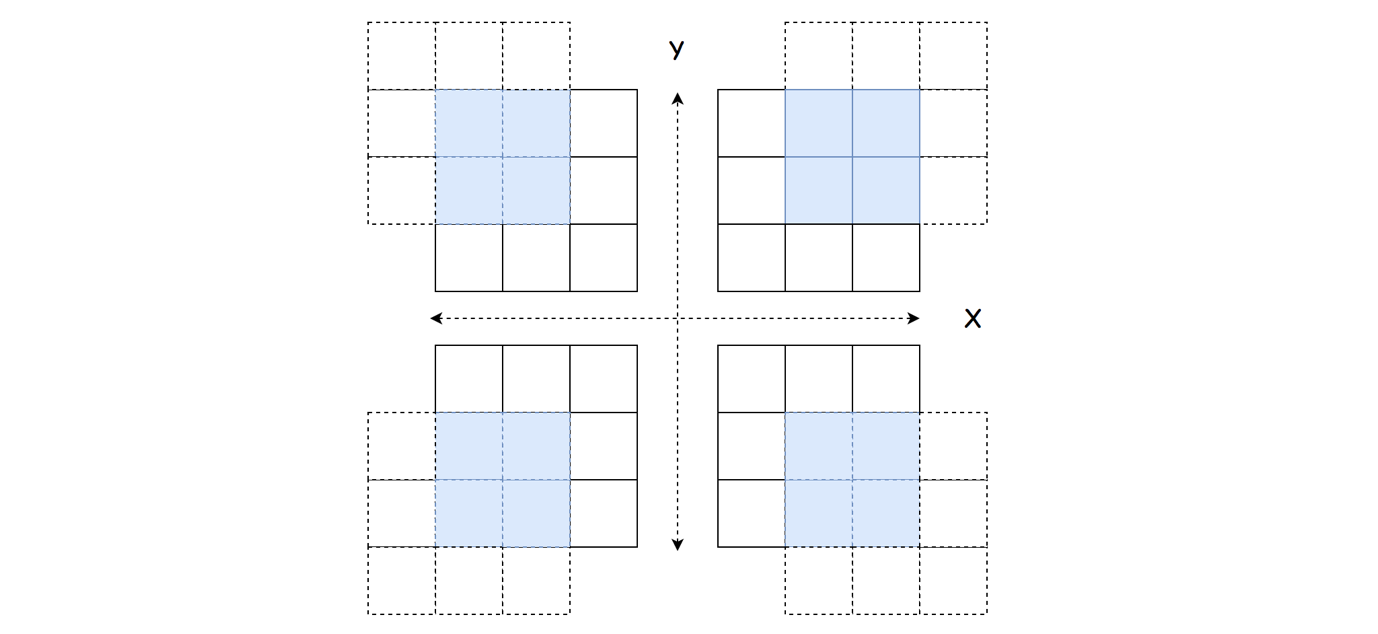
**Intuition**

As stated in the problem description, in order to calculate the number of ones in the overlapping zone, we should first **shift** one of the images. Once the image is shifted, it is intuitive to **count** the numbers.

Therefore, a simple idea is that one could come up all possible overlapping zones, by shifting the image matrix, and then simply count within each overlapping zone.

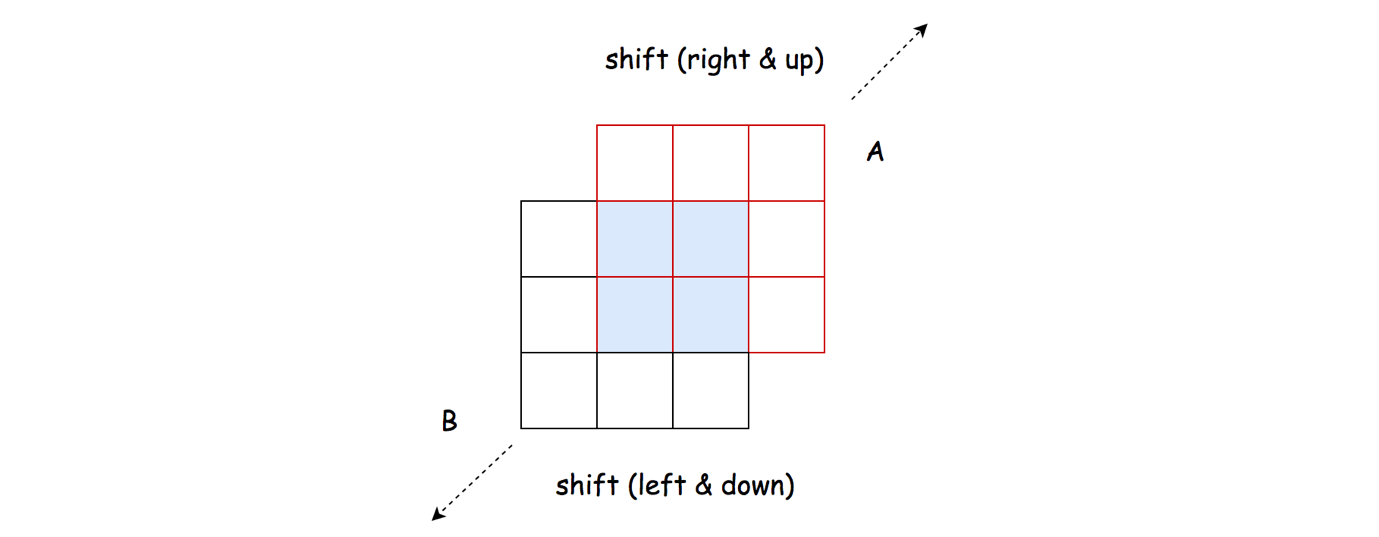
The image matrix could be shifted in four directions, i.e. left, right, up and down.

We could represent the shifting with a 2-axis coordinate as follows, where the X-axis indicates the shifting on the directions of left and right and the Y-axis indicates the shifting of up and down.



For instance, the coordinate of (1, 1) represents that we shift the matrix to the right by one and to the up side by one as well.

One important insight is that shifting one matrix to a direction is **equivalent** to shifting the other matrix to the opposite direction, in the sense that we would have the same overlapping zone at the end.



For example, by shifting the matrix A to one step on the right, is same as shifting the matrix B to the left by one step.

**Algorithm**

Based on the above intuition, we could implement the solution step by step. First we define the function shift\_and\_count(x\_shift, y\_shift, M, R) where we shift the matrix M with reference to the matrix R with the shifting coordinate (x\_shift, y\_shift) and then we count the overlapping ones in the overlapping zone.

* The algorithm is organized as a loop over all possible combinations of shifting coordinates (x\_shift, y\_shift).
* More specifically, the ranges of x\_shift and y\_shift are both [0, N-1] where N*N* is the width of the matrix.
* At each iteration, we invoke the function shift\_and\_count() twice to shift and count the overlapping zone, first with the matrix B as the reference and vice versa.

|  |
| --- |
| class Solution {  /\*\*  \* Shift the matrix M in up-left and up-right directions  \* and count the ones in the overlapping zone.  \*/  protected int shiftAndCount(int xShift, int yShift, int[][] M, int[][] R) {  int leftShiftCount = 0, rightShiftCount = 0;  int rRow = 0;  // count the cells of ones in the overlapping zone.  for (int mRow = yShift; mRow < M.length; ++mRow) {  int rCol = 0;  for (int mCol = xShift; mCol < M.length; ++mCol) {  if (M[mRow][mCol] == 1 && M[mRow][mCol] == R[rRow][rCol])  leftShiftCount += 1;  if (M[mRow][rCol] == 1 && M[mRow][rCol] == R[rRow][mCol])  rightShiftCount += 1;  rCol += 1;  }  rRow += 1;  }  return Math.max(leftShiftCount, rightShiftCount);  }  public int largestOverlap(int[][] A, int[][] B) {  int maxOverlaps = 0;  for (int yShift = 0; yShift < A.length; ++yShift)  for (int xShift = 0; xShift < A.length; ++xShift) {  // move the matrix A to the up-right and up-left directions.  maxOverlaps = Math.max(maxOverlaps, shiftAndCount(xShift, yShift, A, B));  // move the matrix B to the up-right and up-left directions, which is equivalent to moving A to the down-right and down-left directions  maxOverlaps = Math.max(maxOverlaps, shiftAndCount(xShift, yShift, B, A));  }  return maxOverlaps;  }  } |

**Complexity Analysis**

Let N*N* be the width of the matrix.

First of all, let us calculate the number of all possible shiftings, (i.e. the number of overlapping zones).

For a matrix of length N*N*, we have 2(N-1)2(*N*−1) possible offsets along each axis to shift the matrix. Therefore, there are in total 2(N-1) \cdot 2(N-1) = 4(N-1)^22(*N*−1)⋅2(*N*−1)=4(*N*−1)2 possible overlapping zones to calculate.

* Time Complexity: \mathcal{O}(N^4)O(*N*4)
  + As discussed before, we have in total 4(N-1)^24(*N*−1)2 possible overlapping zones.
  + The size of the overlapping zone is bounded by \mathcal{O}(N^2)O(*N*2).
  + Since we iterate through each overlapping zone to find out the overlapping ones, the overall time complexity of the algorithm would be 4(N-1)^2 \cdot \mathcal{O}(N^2) = \mathcal{O}(N^4)4(*N*−1)2⋅O(*N*2)=O(*N*4).
* Space Complexity: \mathcal{O}(1)O(1)
  + As one can see, a constant space is used in the above algorithm.

#### Approach 2: Linear Transformation

**Intuition**

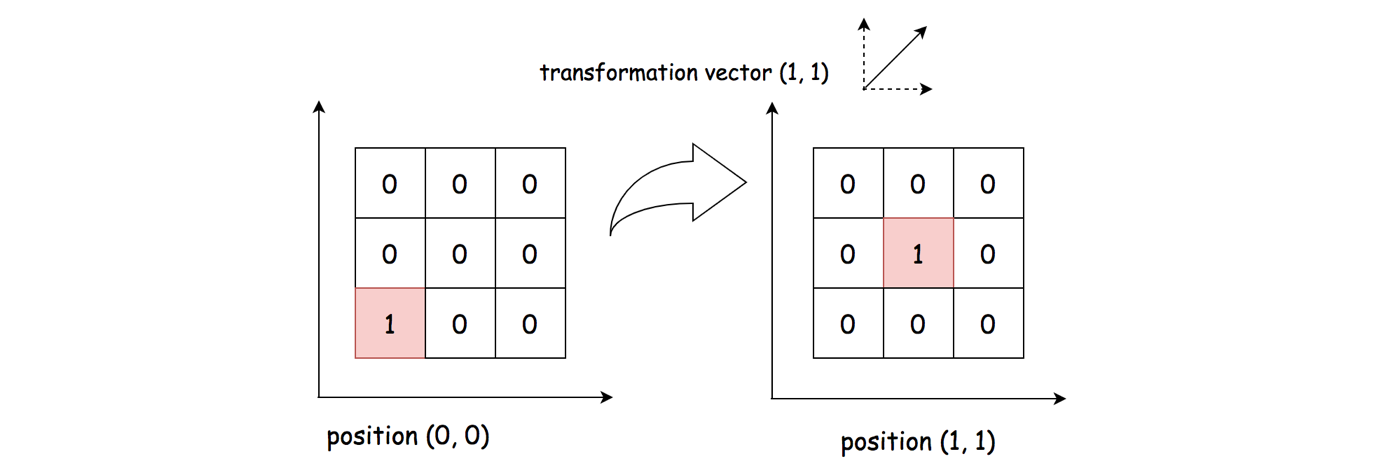
One drawback of the above algorithm is that we would scan through those zones that are filled with zeros over and over, even though these zones are not of our interests.

Because for those cells filled with zero, no matter how we shift, they would not add up to the final solutions. As a follow-up question, we could ask ourselves that, can we ***focus*** on those cells with ones?

The answer is yes. The idea is that we filter out those cells with ones in both matrices, and then we apply the **linear transformation** to align the cells.

First of all, we define a 2-dimension coordinate, via which we could assign a unique coordinate to each cell in the matrix, e.g. a cell can be indexed as I = (X\_i, Y\_i)*I*=(*Xi*​,*Yi*​).

Then to shift a cell, we can obtain the new position of the cell by applying a linear transformation. For example, to shift the cell to the right by one and to the up side by one is to apply the linear transformation vector of V = (1, 1)*V*=(1,1). The new index of the cell can be obtained by I + V = (X\_i + 1, Y\_i + 1)*I*+*V*=(*Xi*​+1,*Yi*​+1).



Furthermore, given two matrices, we have two non-zero cells respectively in the matrices as P\_a =(X\_a, Y\_a)*Pa*​=(*Xa*​,*Ya*​) and P\_b = (X\_b, Y\_b)*Pb*​=(*Xb*​,*Yb*​). To **align** these cells together, we would need a transformation vector as V\_{ab} = (X\_b - X\_a, Y\_b - Y\_a)*Vab*​=(*Xb*​−*Xa*​,*Yb*​−*Ya*​), so that P\_a + V\_{ab} = P\_b*Pa*​+*Vab*​=*Pb*​.

Now, the key insight is that all the cells in the **same** overlapping zone would share the **same** linear transformation vector.

Based on the above insight, we can then use the transformation vector V\_{ab}*Vab*​ as a key to **group** all the non-zero cells alignments between two matrices. Each group represents an overlapping zone. Naturally, the size of the overlapping zone would be the size of the group as well.

**Algorithm**

The algorithm can be implemented in two overall steps.

* First, we filter out those non-zero cells in each matrix respectively.
* Second, we do a cartesian product on the non-zero cells. For each pair of the products, we calculate the corresponding linear transformation vector as V\_{ab} = (X\_b - X\_a, Y\_b - Y\_a)*Vab*​=(*Xb*​−*Xa*​,*Yb*​−*Ya*​). Then, we count the number of the pairs that have the same transformation vector, which is also the number of ones in the overlapping zone.

Here are some sample implementation which are inspired from the user [TejPatel18](https://leetcode.com/problems/image-overlap/discuss/150504/Python-Easy-Logic) in the discussion forum.

|  |
| --- |
| class Solution {  protected List<Pair<Integer, Integer>> non\_zero\_cells(int[][] M) {  List<Pair<Integer, Integer>> ret = new ArrayList<>();  for (int row = 0; row < M.length; ++row)  for (int col = 0; col < M.length; ++col)  if (M[row][col] == 1)  ret.add(new Pair<>(row, col));  return ret;  }  public int largestOverlap(int[][] A, int[][] B) {  List<Pair<Integer, Integer>> A\_ones = non\_zero\_cells(A);  List<Pair<Integer, Integer>> B\_ones = non\_zero\_cells(B);  int maxOverlaps = 0;  HashMap<Pair<Integer, Integer>, Integer> groupCount = new HashMap<>();  for (Pair<Integer, Integer> a : A\_ones)  for (Pair<Integer, Integer> b : B\_ones) {  Pair<Integer, Integer> vec =  new Pair<>(b.getKey() - a.getKey(), b.getValue() - a.getValue());  if (groupCount.containsKey(vec)) {  groupCount.put(vec, groupCount.get(vec) + 1);  } else {  groupCount.put(vec, 1);  }  maxOverlaps = Math.max(maxOverlaps, groupCount.get(vec));  }  return maxOverlaps;  }  } |

**Complexity Analysis**

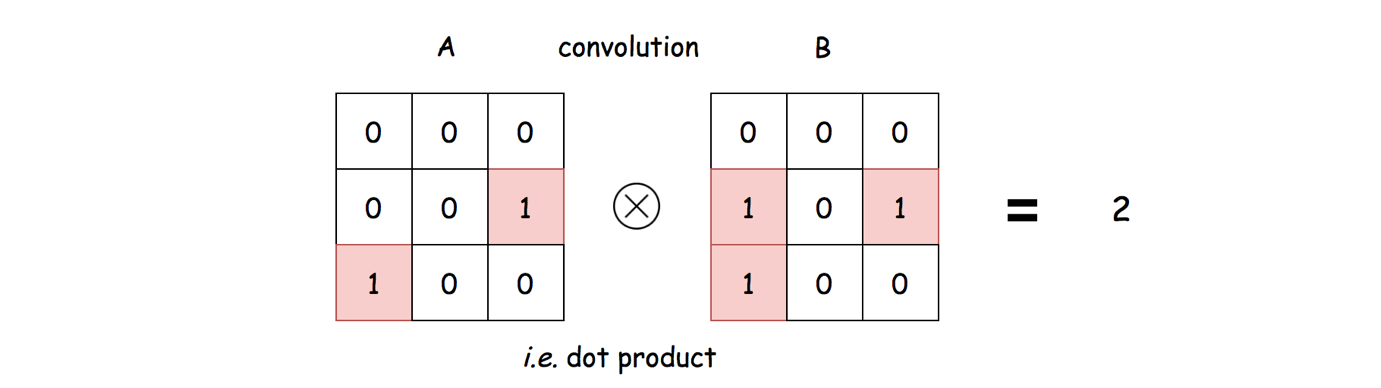
Let M\_a, M\_b*Ma*​,*Mb*​ be the number of non-zero cells in the matrix A and B respectively. Let N*N* be the width of the matrix.

* Time Complexity: \mathcal{O}(N^4)O(*N*4).
  + In the first step, we filter out the non-zero cells in each matrix, which would take \mathcal{O}(N^2)O(*N*2) time.
  + In the second step, we enumerate the cartesian product of non-zero cells between the two matrices, which would take \mathcal{O}(M\_a \cdot M\_b)O(*Ma*​⋅*Mb*​) time. In the worst case, both M\_a*Ma*​ and M\_b*Mb*​ would be up to N^2*N*2, i.e. matrix filled with ones.
  + To sum up, the overall time complexity of the algorithm would be \mathcal{O}(N^2) + \mathcal{O}(N^2 \cdot N^2) = \mathcal{O}(N^4)O(*N*2)+O(*N*2⋅*N*2)=O(*N*4).
  + Although this approach has the same time complexity as the previous approach, it should run faster in practice, since we ignore those zero cells.
* Space Complexity: \mathcal{O}(N^2)O(*N*2)
  + We kept the indices of non-zero cells in both matrices. In the worst case, we would need the \mathcal{O}(N^2)O(*N*2) space for the matrices filled with ones.

#### Approach 3: Imagine Convolution

**Intuition**

As it turns out, the number of overlapped ones in an overlapping zone is equal to the result of performing a [convolution operation](https://en.wikipedia.org/wiki/Kernel_(image_processing)) between two matrices.

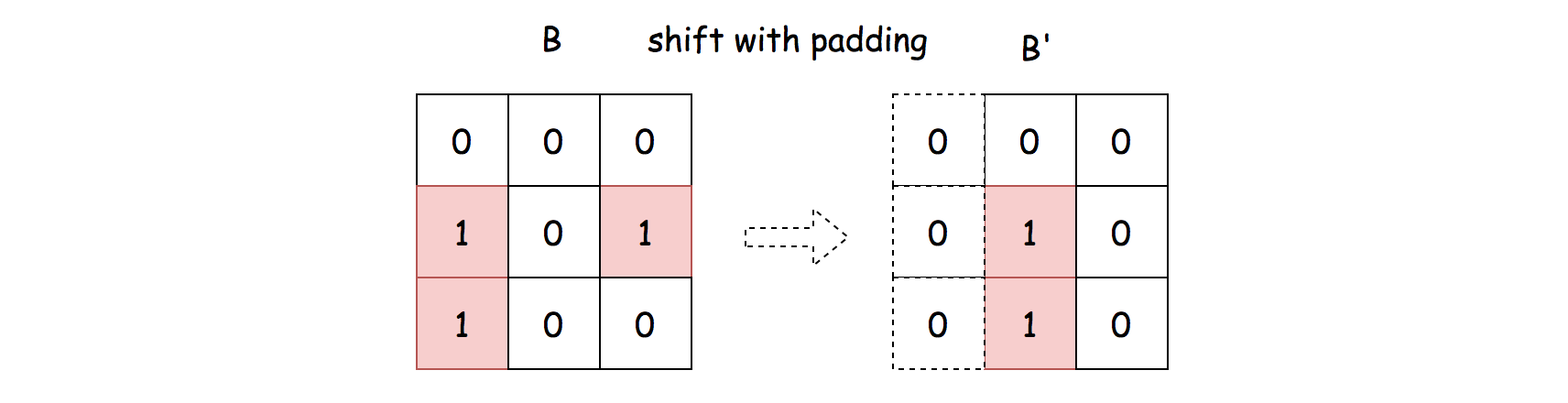


To put it simple, the convolution between two matrices is to perform a **dot product** between them, i.e. \sum\_{x}^{N}\sum\_{y}^{N}\big(V\_{xy}(A) \cdot V\_{xy}(B)\big)∑*xN*​∑*yN*​(*Vxy*​(*A*)⋅*Vxy*​(*B*)) where V\_{xy}(A)*Vxy*​(*A*) and is the value of the cell in the matrix A at the position of (x, y)(*x*,*y*).

From the above formulas, one can clearly see why the result of convolution is the number of overlapping ones.

Usually, the image convolution is performed between an image and a specific **kernel** matrix, in order to obtain certain effects such as blurring or sharpening. In our case, we would perform the convolution between the matrix A and the **shifted** matrix B which serves as a kernel.

More importantly, we should shift the matrix with truncation and **zero padding**, in order to obtain a proper kernel for convolution.

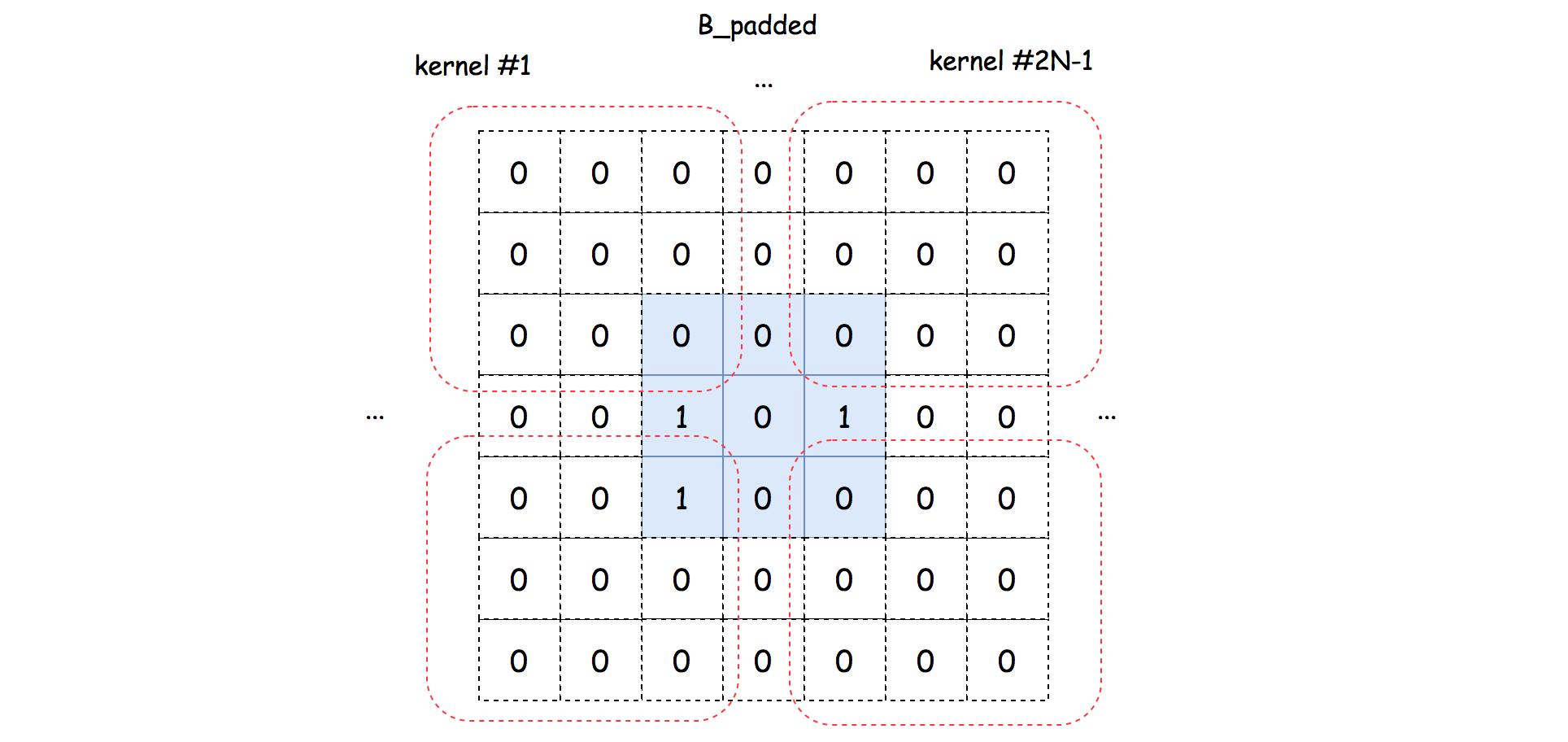


As a result, rather than manually counting the number of overlapping ones, we could perform the convolution operation instead.

**Algorithm**

Overall, we enumerate all possible kernels (by shifting the matrix B), and then perform the convolution operation to count the overlapping ones. The algorithm can be broke down into the following steps:

* First of all, we extend both the width and height of the matrix B to N + 2(N-1) = 3N-2*N*+2(*N*−1)=3*N*−2, and pad the extended cells with zeros, as follows:



* From the extended and padded matrix B, we then can extract the kernel one by one.
* For each kernel, we perform the convolution operation with the matrix A to count the number of overlapping ones.
* At the end, we return the maximal value of overlapping ones.

Here are some sample implementations that are inspired from the solution of user [HeroKillerEver](https://leetcode.com/problems/image-overlap/discuss/131344/An-interesting-SOLUTION%3A-Let-us-think-about-it-as-%22Convolution%22) in the discussion forum.

|  |
| --- |
| class Solution {  protected int convolute(int[][] A, int[][] kernel, int xShift, int yShift) {  int result = 0;  for (int row = 0; row < A.length; ++row)  for (int col = 0; col < A.length; ++col)  result += A[row][col] \* kernel[row + yShift][col + xShift];  return result;  }  public int largestOverlap(int[][] A, int[][] B) {  int N = A.length;  int[][] B\_padded = new int[3 \* N - 2][3 \* N - 2];  for (int row = 0; row < N; ++row)  for (int col = 0; col < N; ++col)  B\_padded[row + N - 1][col + N - 1] = B[row][col];  int maxOverlaps = 0;  for (int xShift = 0; xShift < 2 \* N - 1; ++xShift)  for (int yShift = 0; yShift < 2 \* N - 1; ++yShift) {  maxOverlaps = Math.max(maxOverlaps,  convolute(A, B\_padded, xShift, yShift));  }  return maxOverlaps;  }  } |

Note that, in the Python solution, we utilise the numpy package, which is a well-known library in the tasks of data processing and machine learning.

The numpy library is highly optimized for the matrix operations, which is why it runs faster than the Approach #1, although they have the same time complexity.

**Complexity Analysis**

Let N*N* be the width of the matrix.

* Time Complexity: \mathcal{O}(N^4)O(*N*4)
  + We iterate through (2N-1) \cdot (2N-1)(2*N*−1)⋅(2*N*−1) number of kernels.
  + For each kernel, we perform a convolution operation, which takes \mathcal{O}(N^2)O(*N*2) time.
  + To sum up, the overall time complexity of the algorithm would be (2N-1) \cdot (2N-1) \cdot \mathcal{O}(N^2) = \mathcal{O}(N^4)(2*N*−1)⋅(2*N*−1)⋅O(*N*2)=O(*N*4).
* Space Complexity: \mathcal{O}(N^2)O(*N*2)
  + We extend the matrix B to the size of (3N-2) \cdot (3N-2)(3*N*−2)⋅(3*N*−2), which would require the space of \mathcal{O}(N^2)O(*N*2).

**Word Pattern**

Given a pattern and a string s, find if s follows the same pattern.

Here **follow** means a full match, such that there is a bijection between a letter in pattern and a **non-empty** word in s.

**Example 1:**

**Input:** pattern = "abba", s = "dog cat cat dog"

**Output:** true

**Example 2:**

**Input:** pattern = "abba", s = "dog cat cat fish"

**Output:** false

**Example 3:**

**Input:** pattern = "aaaa", s = "dog cat cat dog"

**Output:** false

**Example 4:**

**Input:** pattern = "abba", s = "dog dog dog dog"

**Output:** false

**Constraints:**

* 1 <= pattern.length <= 300
* pattern contains only lower-case English letters.
* 1 <= s.length <= 3000
* s contains only lower-case English letters and spaces ' '.
* s **does not contain** any leading or trailing spaces.
* All the words in s are separated by a **single space**.

## Solution

This problem is similar to [Isomorphic Strings](https://leetcode.com/problems/isomorphic-strings/).

#### Approach 1: Two Hash Maps

**Intuition**

The most naive way to start thinking about this problem is to have a single hash map, tracking which character (in pattern) maps to what word (in s). As you scan each character-word pair, update this hash map for characters which are not in the mapping. If you see a character which already is one of the keys in mapping, check whether the current word matches with the word the character maps to. If they do not match, you can immediately return False, otherwise, just keep on scanning until the end.

This type of check will work well for cases such as:

* "abba" and "dog cat cat dog" -> Returns True.
* "abba" and "dog cat cat fish" -> Returns False.

But it will fail for:

* "abba" and "dog dog dog dog" -> Returns True (Expected False).

A fix for this is to have two hash maps, one for mapping characters to words and the other for mapping words to characters. While scanning each character-word pair,

* If the character is **NOT** in the character to word mapping, you additionally check whether that word is also in the word to character mapping.
  + If that word is already in the word to character mapping, then you can return False immediately since it has been mapped with some other character before.
  + Else, update both mappings.
* If the character **IS IN** in the character to word mapping, you just need to check whether the current word matches with the word which the character maps to in the character to word mapping. If not, you can return False immediately.

**Implementation**

|  |
| --- |
| class Solution {  public boolean wordPattern(String pattern, String s) {  HashMap <Character, String> map\_char = new HashMap();  HashMap <String, Character> map\_word = new HashMap();  String[] words = s.split(" ");  if (words.length != pattern.length())  return false;  for (int i = 0; i < words.length; i++) {  char c = pattern.charAt(i);  String w = words[i];  if (!map\_char.containsKey(c)) {  if (map\_word.containsKey(w)) {  return false;  } else {  map\_char.put(c, w);  map\_word.put(w, c);  }  } else {  String mapped\_word = map\_char.get(c);  if (!mapped\_word.equals(w))  return false;  }  }  return true;  }  } |

**Complexity Analysis**

* Time complexity : O(N)*O*(*N*) where N*N* represents the number of words in s or the number of characters in pattern.
* Space complexity : O(M)*O*(*M*) where M*M* represents the number of unique words in s. Even though we have two hash maps, the character to word hash map has space complexity of O(1)*O*(1) since there can at most be 26 keys.

Addendum: Rather than keeping two hash maps, we can only keep character to word mapping and whenever we find a character that is not in the mapping, you can check whether the word in current character-word pair is already **one of the values** in the character to word mapping. However, this is trading time off for better space since checking for values in a hash map is a O(M)*O*(*M*) operation where M*M* is the number of key value pairs in the hash map. Thus, if we decide to go this way, our time complexity will be O(NM)*O*(*NM*) where N*N* is the number of unique characters in pattern.

Another similar approach to Approach 1 would be using hash set to keep track of words which have been encountered. Instead of checking whether the word is already in the word to character mapping, you just need to check whether the word is in the encountered word hash set. And, rather than updating the word to character mapping, you just need to add the word to the encountered word hash set. Hash set would have a better practical space complexity even though the big-O space complexity for hash set and hash map is the same.

#### Approach 2: Single Index Hash Map

**Intuition**

Rather than having two hash maps, we can have a single index hash map which keeps track of the first occurrences of each character in pattern and each word in s. As we go through each character-word pair, we insert unseen characters from pattern and unseen words from s.

The goal is to make sure that the indices of each character and word match up. As soon as we find a mismatch, we can return False.

Let's go through some examples.

* pattern: 'abba'
* s: 'dog cat cat dog'

1. 'a' and 'dog' -> map\_index = {'a': 0, 'dog': 0}
   * Index of 'a' and index of 'dog' are the same.
2. 'b' and 'cat' -> map\_index = {'a': 0, 'dog': 0, 'b': 1, 'cat': 1}
   * Index of 'b' and index of 'cat' are the same.
3. 'b' and 'cat' -> map\_index = {'a': 0, 'dog': 0, 'b': 1, 'cat': 1}
   * 'b' is already in the mapping, no need to update.
   * 'cat' is already in the mapping, no need to update.
   * Index of 'b' and index of 'cat' are the same.
4. 'a' and 'dog' -> map\_index = {'a': 0, 'dog': 0, 'b': 1, 'cat': 1}
   * 'a' is already in the mapping, no need to update.
   * 'dog' is already in the mapping, no need to update.
   * Index of 'a' and index of 'dog' are the same.

* pattern: 'abba'
* s: 'dog cat fish dog'

1. 'a' and 'dog' -> map\_index = {'a': 0, 'dog': 0}
   * Index of 'a' and index of 'dog' are the same.
2. 'b' and 'cat' -> map\_index = {'a': 0, 'dog': 0, 'b': 1, 'cat': 1}
   * Index of 'b' and index of 'cat' are the same.
3. 'b' and 'fish' -> map\_index = {'a': 0, 'dog': 0, 'b': 1, 'cat': 1, 'fish': 2}
   * 'b' is already in the mapping, no need to update.
   * Index of 'b' and index of 'fish' are NOT the same. Returns False.

**Implementation**

Differentiating between character and string: In Python there is no separate char type. And for cases such as:

* pattern: 'abba'
* s: 'b a a b'

Using the same hash map will not work properly. A workaround is to prefix each character in pattern with "char\_" and each word in s with "word\_".

|  |
| --- |
| class Solution {  public boolean wordPattern(String pattern, String s) {  HashMap map\_index = new HashMap();  String[] words = s.split(" ");  if (words.length != pattern.length())  return false;  for (Integer i = 0; i < words.length; i++) {  char c = pattern.charAt(i);  String w = words[i];  if (!map\_index.containsKey(c))  map\_index.put(c, i);  if (!map\_index.containsKey(w))  map\_index.put(w, i);  if (map\_index.get(c) != map\_index.get(w))  return false;  }  return true;  }  } |

**Complexity Analysis**

* Time complexity : O(N)*O*(*N*) where N*N* represents the number of words in the s or the number of characters in the pattern.
* Space complexity : O(M)*O*(*M*) where M*M* is the number of unique characters in pattern and words in s.

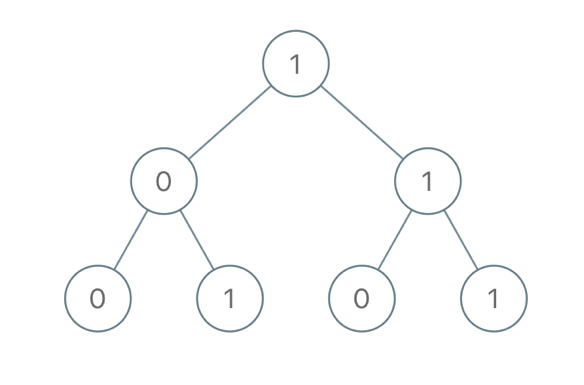
**Sum of Root To Leaf Binary Numbers**

You are given the root of a binary tree where each node has a value 0 or 1.  Each root-to-leaf path represents a binary number starting with the most significant bit.  For example, if the path is 0 -> 1 -> 1 -> 0 -> 1, then this could represent 01101 in binary, which is 13.

For all leaves in the tree, consider the numbers represented by the path from the root to that leaf.

Return the sum of these numbers. The answer is **guaranteed** to fit in a **32-bits** integer.

**Example 1:**



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

**Output:** 22

**Explanation:** (100) + (101) + (110) + (111) = 4 + 5 + 6 + 7 = 22

**Example 2:**

**Input:** root = [0]

**Output:** 0

**Example 3:**

**Input:** root = [1]

**Output:** 1

**Example 4:**

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

**Output:** 3

**Constraints:**

* The number of nodes in the tree is in the range [1, 1000].
* Node.val is 0 or 1.

   Hide Hint #1

Find each path, then transform that path to an integer in base 10.

## Solution

#### Overview

**Prerequisites: Bitwise Trick**

If you work with decimal representation, the conversion of 1->2 into 12 is easy. You start from curr\_number = 1, then shift one register to the left and add the next digit: curr\_number = 1 \* 10 + 2 = 12.

If you work with binaries 1 -> 1 -> 3, it's the same. You start from curr\_number = 1, then shift one register to the left and add the next digit: curr\_number = (1 << 1) | 1 = 3.

**Prerequisites: Tree Traversals**

There are three DFS ways to traverse the tree: preorder, postorder and inorder. Please check two minutes picture explanation, if you don't remember them quite well: [here is Python version](https://leetcode.com/problems/binary-tree-inorder-traversal/discuss/283746/all-dfs-traversals-preorder-inorder-postorder-in-python-in-1-line) and [here is Java version](https://leetcode.com/problems/binary-tree-inorder-traversal/discuss/328601/all-dfs-traversals-preorder-postorder-inorder-in-java-in-5-lines).

**Optimal Strategy to Solve the Problem**

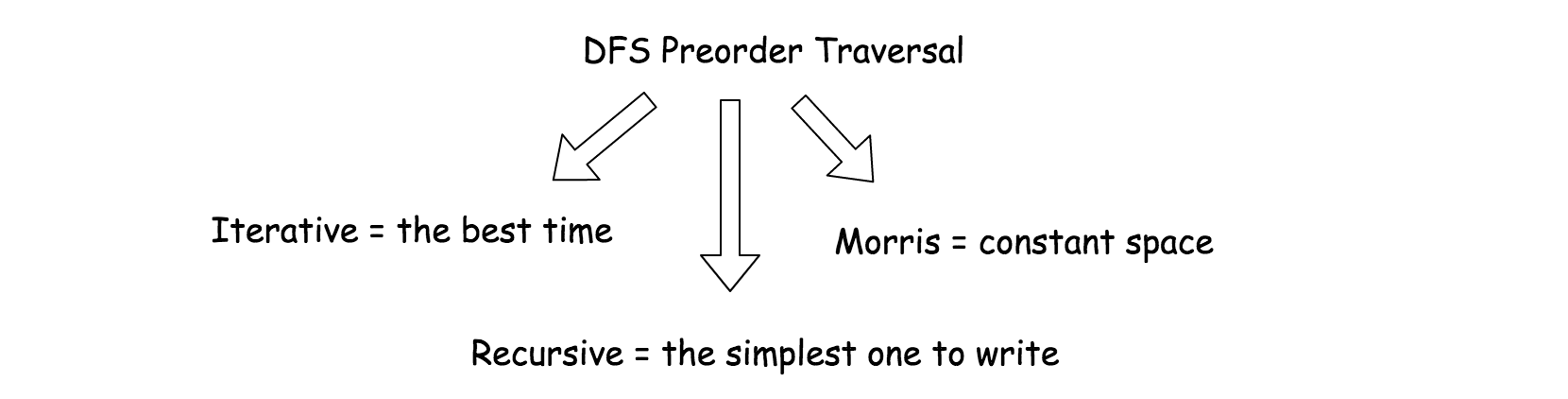
Root-to-left traversal is so-called DFS preorder traversal. To implement it, one has to follow straightforward strategy Root->Left->Right.

Since one has to visit all nodes, the best possible time complexity here is linear. Hence all interest here is to improve the space complexity.

There are 3 ways to implement preorder traversal: iterative, recursive and Morris.

Iterative and recursive approaches here do the job in one pass, but they both need up to \mathcal{O}(H)O(*H*) space to keep the stack, where H*H* is a tree height.

Morris approach is two-pass approach, but it's a constant-space one.

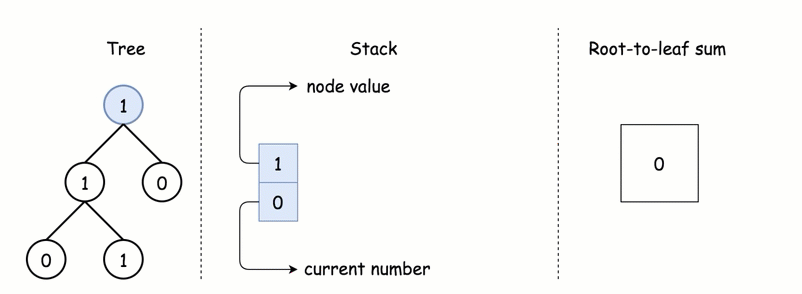


#### Approach 1: Iterative Preorder Traversal.

**Intuition**

Here we implement standard iterative preorder traversal with the stack:

* Push root into stack.
* While stack is not empty:
  + Pop out a node from stack and update the current number.
  + If the node is a leaf, update root-to-leaf sum.
  + Push right and left child nodes into stack.
* Return root-to-leaf sum.



**Implementation**

Note, that [Javadocs recommends to use ArrayDeque, and not Stack as a stack implementation](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html).

|  |
| --- |
| class Solution {  public int sumRootToLeaf(TreeNode root) {  int rootToLeaf = 0, currNumber = 0;  Deque<Pair<TreeNode, Integer>> stack = new ArrayDeque();  stack.push(new Pair(root, 0));  while (!stack.isEmpty()) {  Pair<TreeNode, Integer> p = stack.pop();  root = p.getKey();  currNumber = p.getValue();  if (root != null) {  currNumber = (currNumber << 1) | root.val;  // if it's a leaf, update root-to-leaf sum  if (root.left == null && root.right == null) {  rootToLeaf += currNumber;  } else {  stack.push(new Pair(root.right, currNumber));  stack.push(new Pair(root.left, currNumber));  }  }  }  return rootToLeaf;  }  } |

**Complexity Analysis**

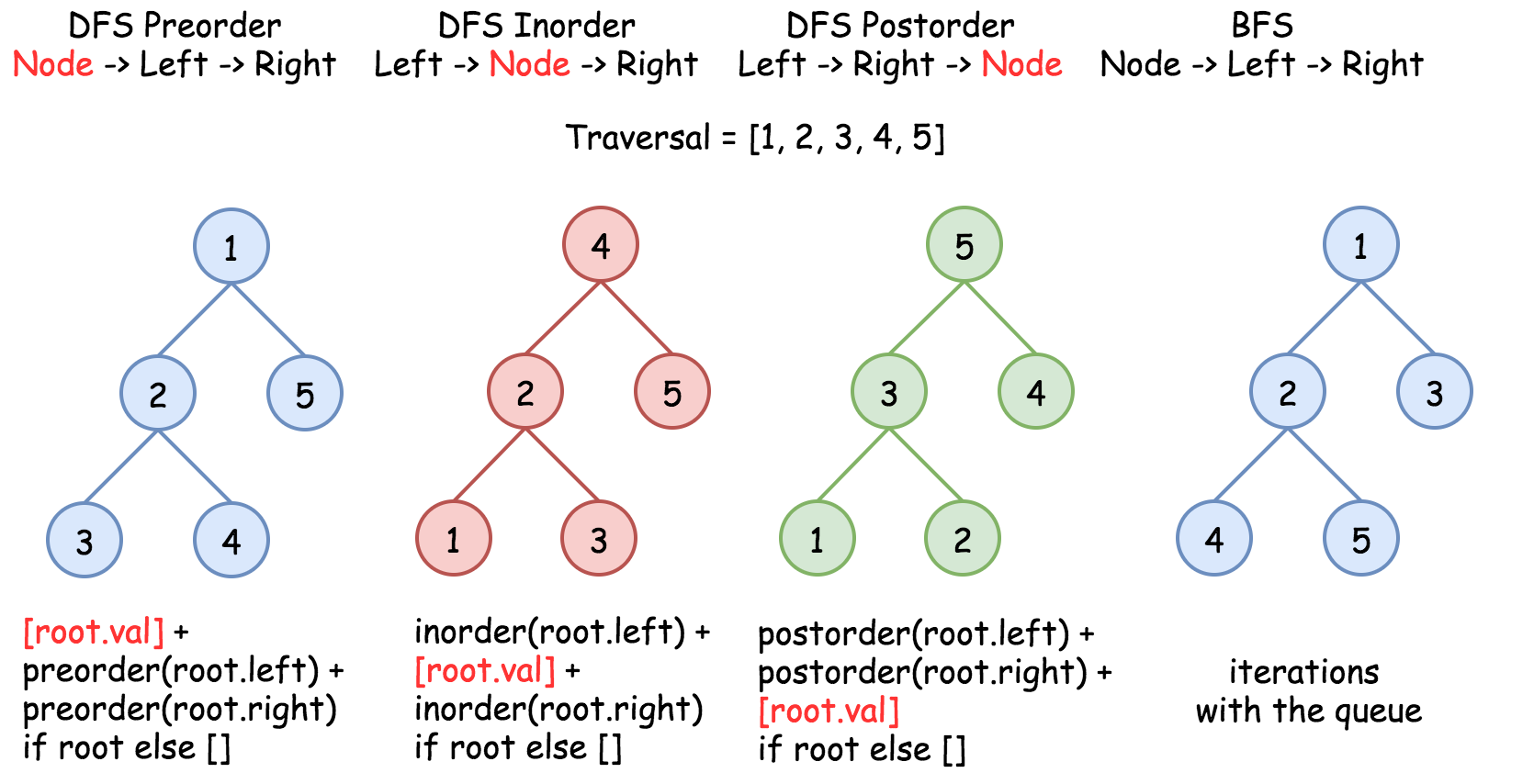
* Time complexity: \mathcal{O}(N)O(*N*), where N*N* is a number of nodes, since one has to visit each node.
* Space complexity: up to \mathcal{O}(H)O(*H*) to keep the stack, where H*H* is a tree height.

#### Approach 2: Recursive Preorder Traversal.

Iterative approach 1 could be converted into recursive one.

Recursive preorder traversal is extremely simple: follow Root->Left->Right direction, i.e. do all the business with the node (= update the current number and root-to-leaf sum), and then do the recursive calls for the left and right child nodes.

P.S. Here is the difference between preorder and the other DFS recursive traversals. On the following figure the nodes are enumerated in the order you visit them, please follow 1-2-3-4-5 to compare different DFS strategies implemented as recursion.



**Implementation**

|  |
| --- |
| class Solution {  int rootToLeaf = 0;    public void preorder(TreeNode r, int currNumber) {  if (r != null) {  currNumber = (currNumber << 1) | r.val;  // if it's a leaf, update root-to-leaf sum  if (r.left == null && r.right == null) {  rootToLeaf += currNumber;  }  preorder(r.left, currNumber);  preorder(r.right, currNumber);  }  }  public int sumRootToLeaf(TreeNode root) {  preorder(root, 0);  return rootToLeaf;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*), where N*N* is a number of nodes, since one has to visit each node.
* Space complexity: up to \mathcal{O}(H)O(*H*) to keep the recursion stack, where H*H* is a tree height.

#### Approach 3: Morris Preorder Traversal.

We discussed already iterative and recursive preorder traversals, which both have great time complexity though use up to \mathcal{O}(H)O(*H*) to keep the stack. We could trade in performance to save space.

The idea of Morris preorder traversal is simple: to use no space but to traverse the tree.

How that could be even possible? At each node one has to decide where to go: to the left or to the right, traverse the left subtree or traverse the right subtree. How one could know that the left subtree is already done if no additional memory is allowed?

The idea of [Morris](https://www.sciencedirect.com/science/article/pii/0020019079900681) algorithm is to set the temporary link between the node and its [predecessor](https://leetcode.com/articles/delete-node-in-a-bst/): predecessor.right = root. So one starts from the node, computes its predecessor and verifies if the link is present.

* There is no link? Set it and go to the left subtree.
* There is a link? Break it and go to the right subtree.

There is one small issue to deal with : what if there is no left child, i.e. there is no left subtree? Then go straightforward to the right subtree.

**Implementation**

|  |
| --- |
| class Solution {  public int sumRootToLeaf(TreeNode root) {  int rootToLeaf = 0, currNumber = 0;  int steps;  TreeNode predecessor;  while (root != null) {  // If there is a left child,  // then compute the predecessor.  // If there is no link predecessor.right = root --> set it.  // If there is a link predecessor.right = root --> break it.  if (root.left != null) {  // Predecessor node is one step to the left  // and then to the right till you can.  predecessor = root.left;  steps = 1;  while (predecessor.right != null && predecessor.right != root) {  predecessor = predecessor.right;  ++steps;  }  // Set link predecessor.right = root  // and go to explore the left subtree  if (predecessor.right == null) {  currNumber = (currNumber << 1) | root.val;  predecessor.right = root;  root = root.left;  }  // Break the link predecessor.right = root  // Once the link is broken,  // it's time to change subtree and go to the right  else {  // If you're on the leaf, update the sum  if (predecessor.left == null) {  rootToLeaf += currNumber;  }  // This part of tree is explored, backtrack  for(int i = 0; i < steps; ++i) {  currNumber >>= 1;  }  predecessor.right = null;  root = root.right;  }  }  // If there is no left child  // then just go right.  else {  currNumber = (currNumber << 1) | root.val;  // if you're on the leaf, update the sum  if (root.right == null) {  rootToLeaf += currNumber;  }  root = root.right;  }  }  return rootToLeaf;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*), where N*N* is a number of nodes.
* Space complexity: \mathcal{O}(1)O(1).

**Combination Sum III**

Find all valid combinations of k numbers that sum up to n such that the following conditions are true:

* Only numbers 1 through 9 are used.
* Each number is used **at most once**.

Return a list of all possible valid combinations. The list must not contain the same combination twice, and the combinations may be returned in any order.

**Example 1:**

**Input:** k = 3, n = 7

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

**Explanation:**

1 + 2 + 4 = 7

There are no other valid combinations.

**Example 2:**

**Input:** k = 3, n = 9

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

**Explanation:**

1 + 2 + 6 = 9

1 + 3 + 5 = 9

2 + 3 + 4 = 9

There are no other valid combinations.

**Example 3:**

**Input:** k = 4, n = 1

**Output:** []

**Explanation:** There are no valid combinations. [1,2,1] is not valid because 1 is used twice.

**Example 4:**

**Input:** k = 3, n = 2

**Output:** []

**Explanation:** There are no valid combinations.

**Example 5:**

**Input:** k = 9, n = 45

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

**Explanation:**

1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45

​​​​​​​There are no other valid combinations.

**Constraints:**

* 2 <= k <= 9
* 1 <= n <= 60

## Solution

#### Approach 1: Backtracking

**Intuition**

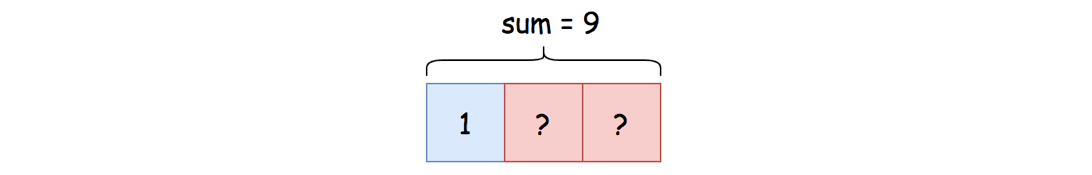
The problem asks us to come up with some fixed-length combinations that meet certain conditions.

To solve the problem, it would be beneficial to build a combination by hand.

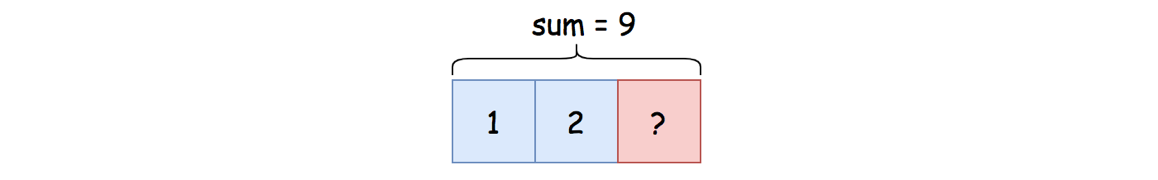
If we represent the combination as an array, we then could fill the array **one element at a time**.

For example, given the input k=3*k*=3 and n=9*n*=9, i.e. the size of the combination is 3, and the sum of the digits in the combination should be 9. Here are a few steps that we could do:

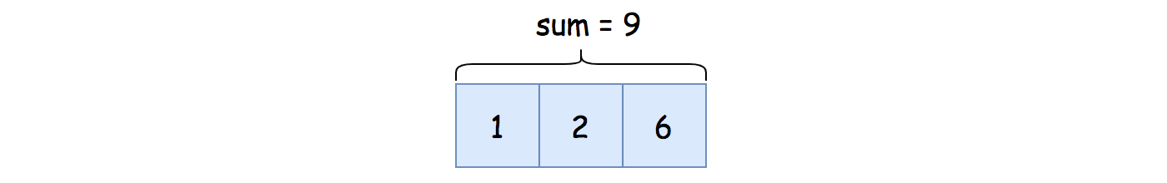
* 1). We could pick a digit for the **first** element in the combination. Initially, the list of candidates is [1, 2, 3, 4, 5, 6, 7, 8. 9] for any element in the combination, as stated in the problem. Let us pick 1 as the first element. The current combination is [1].



* 2). Now that we picked the first element, we have two more elements to fill in the final combination. Before we proceed, let us review the conditions that we should fullfil for the next steps.
  + Since we've already picked the digit 1, we should exclude the digit from the original candidate list for the remaining elements, in order to ensure that the combination does not contain any **duplicate** digits, as required in the problem.
  + In addition, the sum of the remaining two elements should be 9 - 1 = 89−1=8.
* 3). Based on the above conditions, for the second element, we could have several choices. Let us pick the digit 2, which is not a duplicate of the first element, plus it does not exceed the desired sum (i.e. 88) neither. The combination now becomes [1, 2].

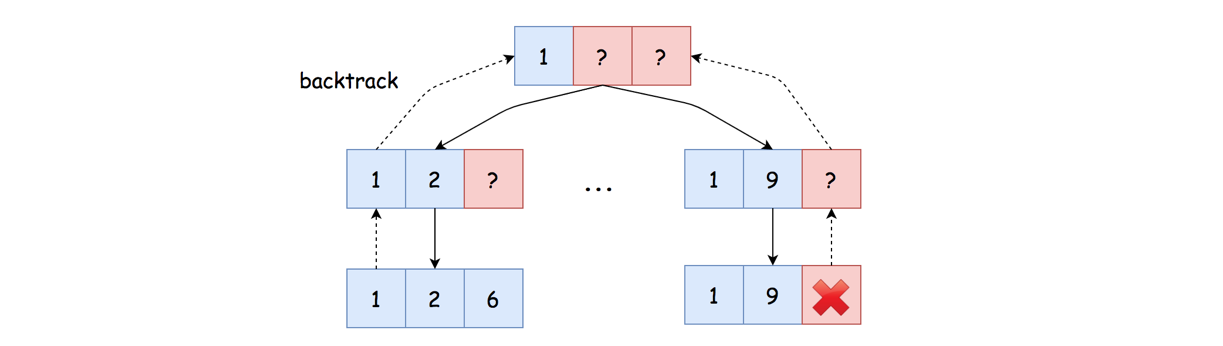


* 4). Now for the third element, with all the constraints, it leaves us no choice but to pick the digit 6 as the final element in the combination of [1, 2, 6].



* 5). As we mentioned before, for the second element, we could have several choices. For instance, we could have picked the digit 3, instead of the digit 2. Eventually, it could lead us to another solution as [1, 3, 5].
* 6). As one can see, for each element in the combination, we could **revisit** our choices, and **try out** other possibilities to see if it leads to a valid solution.

If you have followed the above steps, it should become evident that **backtracking** would be the technique that we could use to come up an algorithm for this problem.



Indeed, we could resort to backtracking, where we try to fill the combination **one element at a step**. Each choice we make at certain step might lead us to a final solution. If not, we simply revisit the choice and try out other choices, i.e. backtrack.

**Algorithm**

There are many ways to implement a backtracking algorithm. One could also refer to our [Explore card](https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/) where we give some examples of backtracking algorithms.

To implement the algorithm, one could literally follow the steps in the Intuition section. However, we would like to highlight a key **trick** that we employed, in order to ensure the **non-redundancy** among the digits within a single combination, as well as the **non-redundancy** among the combinations.

The trick is that we pick the candidates **in order**. We treat the candidate digits as a list with order, i.e. [1, 2, 3, 4, 5, 6, 7, 8. 9]. At any given step, once we pick a digit, e.g. 6, we will not consider any digits before the chosen digit for the following steps, e.g. the candidates are reduced down to [7, 8, 9].

With the above strategy, we could ensure that a digit will never be picked twice for the same combination. Also, all the combinations that we come up with would be unique.

Here are some sample implementations based on the above ideas.

|  |
| --- |
| class Solution {  protected void backtrack(int remain, int k,  LinkedList<Integer> comb, int next\_start,  List<List<Integer>> results) {  if (remain == 0 && comb.size() == k) {  // Note: it's important to make a deep copy here,  // Otherwise the combination would be reverted in other branch of backtracking.  results.add(new ArrayList<Integer>(comb));  return;  } else if (remain < 0 || comb.size() == k) {  // Exceed the scope, no need to explore further.  return;  }  // Iterate through the reduced list of candidates.  for (int i = next\_start; i < 9; ++i) {  comb.add(i + 1);  this.backtrack(remain - i - 1, k, comb, i + 1, results);  comb.removeLast();  }  }  public List<List<Integer>> combinationSum3(int k, int n) {  List<List<Integer>> results = new ArrayList<List<Integer>>();  LinkedList<Integer> comb = new LinkedList<Integer>();  this.backtrack(n, k, comb, 0, results);  return results;  }  } |

**Complexity Analysis**

Let K*K* be the number of digits in a combination.

* Time Complexity: \mathcal{O}(\frac{9! \cdot K}{(9-K)!})O((9−*K*)!9!⋅*K*​)
  + In a worst scenario, we have to explore all potential combinations to the very end, i.e. the sum n*n* is a large number (n > 9 \* 9*n*>9∗9). At the first step, we have 99 choices, while at the second step, we have 88 choices, so on and so forth.
  + The number of exploration we need to make in the worst case would be P(9, K) = \frac{9!}{(9-K)!}*P*(9,*K*)=(9−*K*)!9!​, assuming that K <= 9*K*<=9. By the way, K*K* cannot be greater than 9, otherwise we cannot have a combination whose digits are all unique.
  + Each exploration takes a constant time to process, except the last step where it takes \mathcal{O}(K)O(*K*) time to make a copy of combination.
  + To sum up, the overall time complexity of the algorithm would be \frac{9!}{(9-K)!} \cdot \mathcal{O}(K) = \mathcal{O}(\frac{9! \cdot K}{(9-K)!})(9−*K*)!9!​⋅O(*K*)=O((9−*K*)!9!⋅*K*​).
* Space Complexity: \mathcal{O}(K)O(*K*)
  + During the backtracking, we used a list to keep the current combination, which holds up to K*K* elements, i.e. \mathcal{O}(K)O(*K*).
  + Since we employed recursion in the backtracking, we would need some additional space for the function call stack, which could pile up to K*K* consecutive invocations, i.e. \mathcal{O}(K)O(*K*).
  + Hence, to sum up, the overall space complexity would be \mathcal{O}(K)O(*K*).
  + **Note that**, we did not take into account the space for the final results in the space complexity.

**Inorder Successor in BST II**

Given a node in a binary search tree, find the in-order successor of that node in the BST.

If that node has no in-order successor, return null.

The successor of a node is the node with the smallest key greater than node.val.

You will have direct access to the node but not to the root of the tree. Each node will have a reference to its parent node. Below is the definition for Node:

class Node {

public int val;

public Node left;

public Node right;

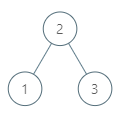
public Node parent;

}

**Follow up:**

Could you solve it without looking up any of the node's values?

**Example 1:**

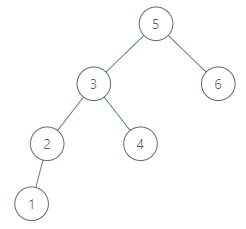


**Input:** tree = [2,1,3], node = 1

**Output:** 2

**Explanation:** 1's in-order successor node is 2. Note that both the node and the return value is of Node type.

**Example 2:**

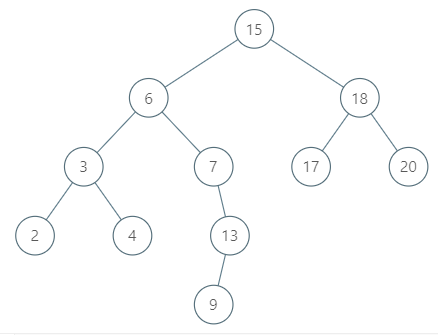


**Input:** tree = [5,3,6,2,4,null,null,1], node = 6

**Output:** null

**Explanation:** There is no in-order successor of the current node, so the answer is null.

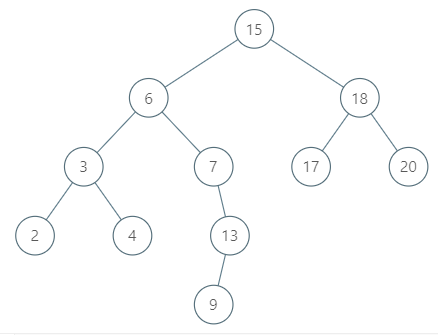
**Example 3:**



**Input:** tree = [15,6,18,3,7,17,20,2,4,null,13,null,null,null,null,null,null,null,null,9], node = 15

**Output:** 17

**Example 4:**



**Input:** tree = [15,6,18,3,7,17,20,2,4,null,13,null,null,null,null,null,null,null,null,9], node = 13

**Output:** 15

**Example 5:**

**Input:** tree = [0], node = 0

**Output:** null

**Constraints:**

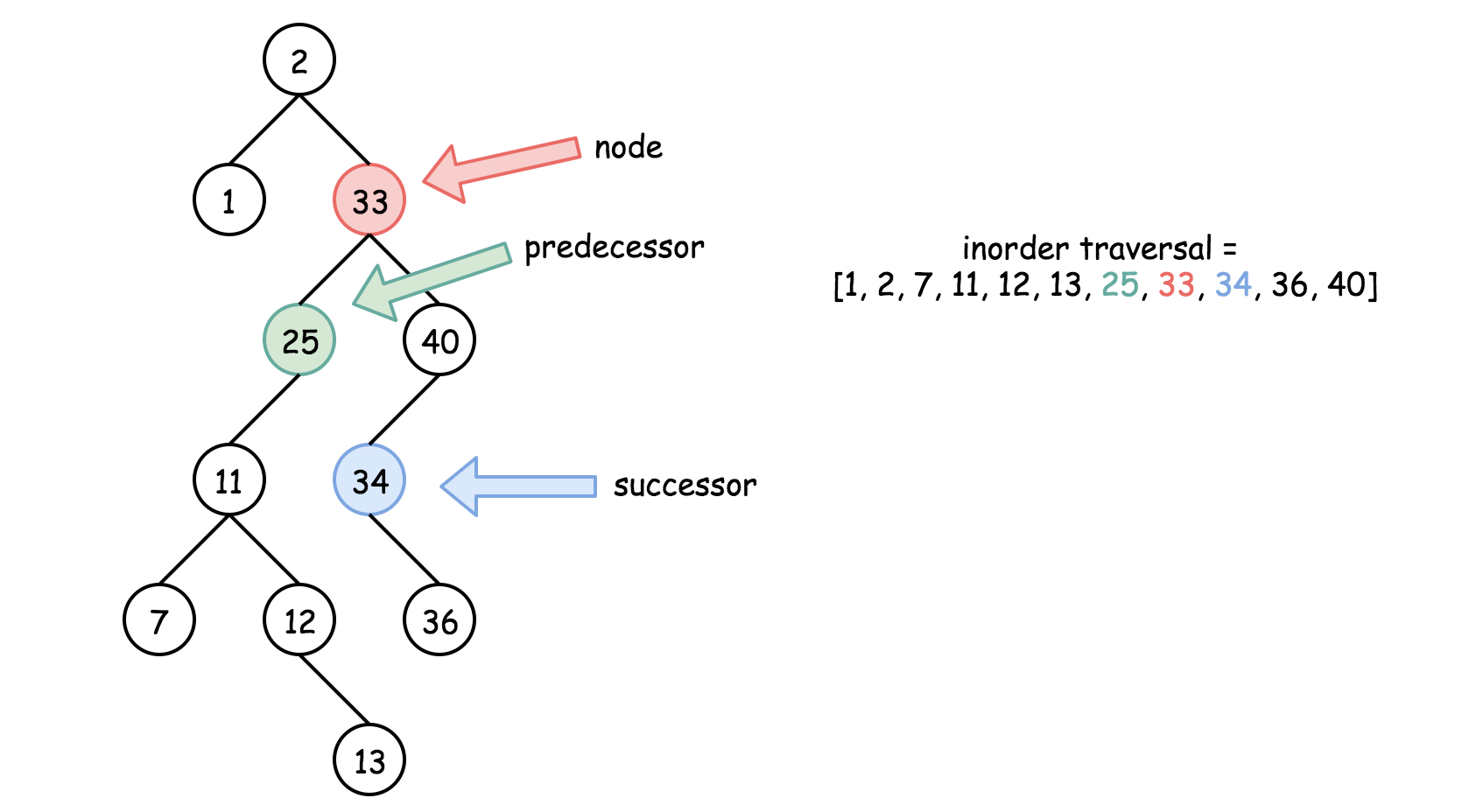
* -10^5 <= Node.val <= 10^5
* 1 <= Number of Nodes <= 10^4
* All Nodes will have unique values.

## Solution

#### Successor and Predecessor

Successor = "after node", i.e. the next node in the inorder traversal, or the smallest node after the current one.

Predecessor = "before node", i.e. the previous node in the inorder traversal, or the largest node before the current one.

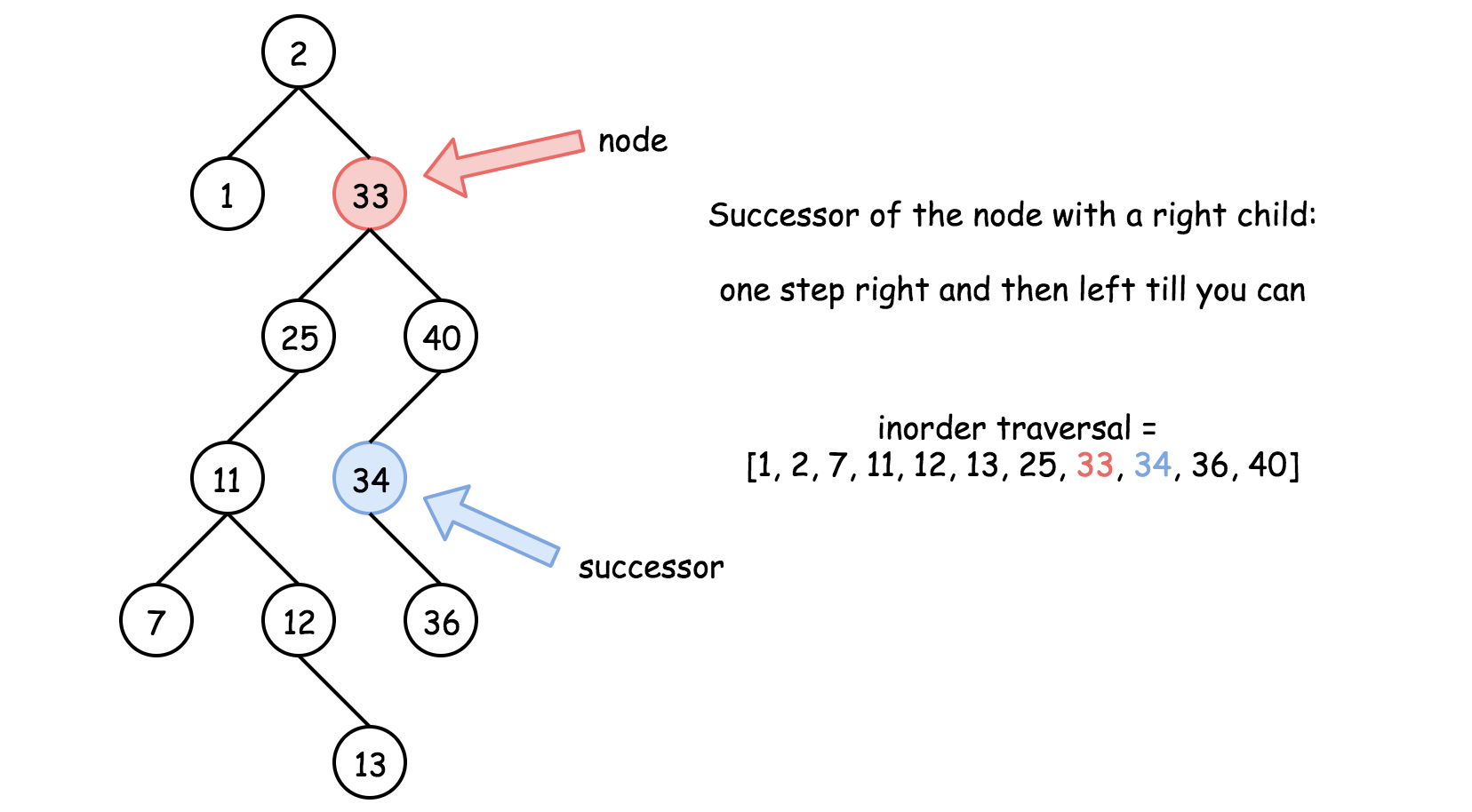


#### Approach 1: Iteration

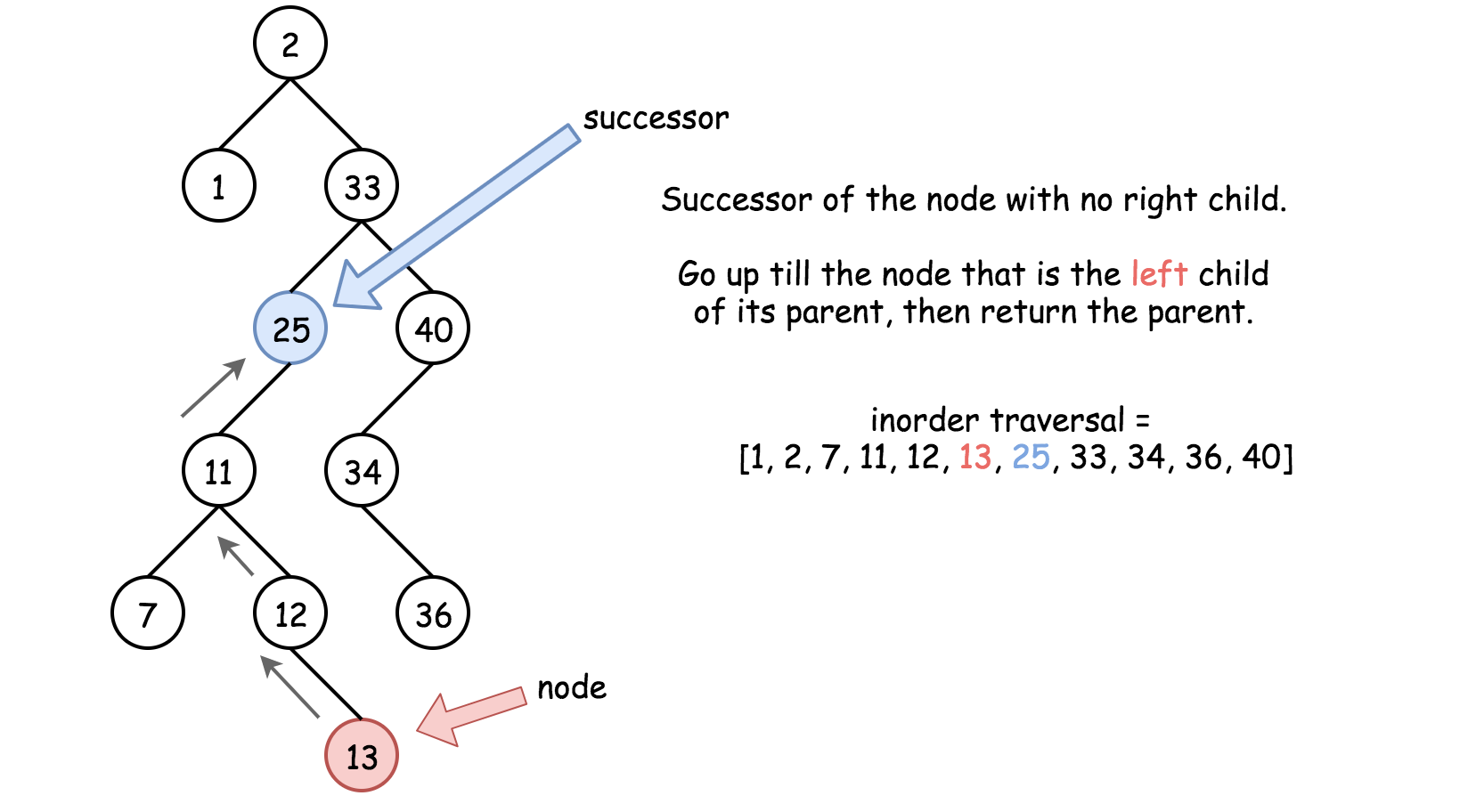
**Intuition**

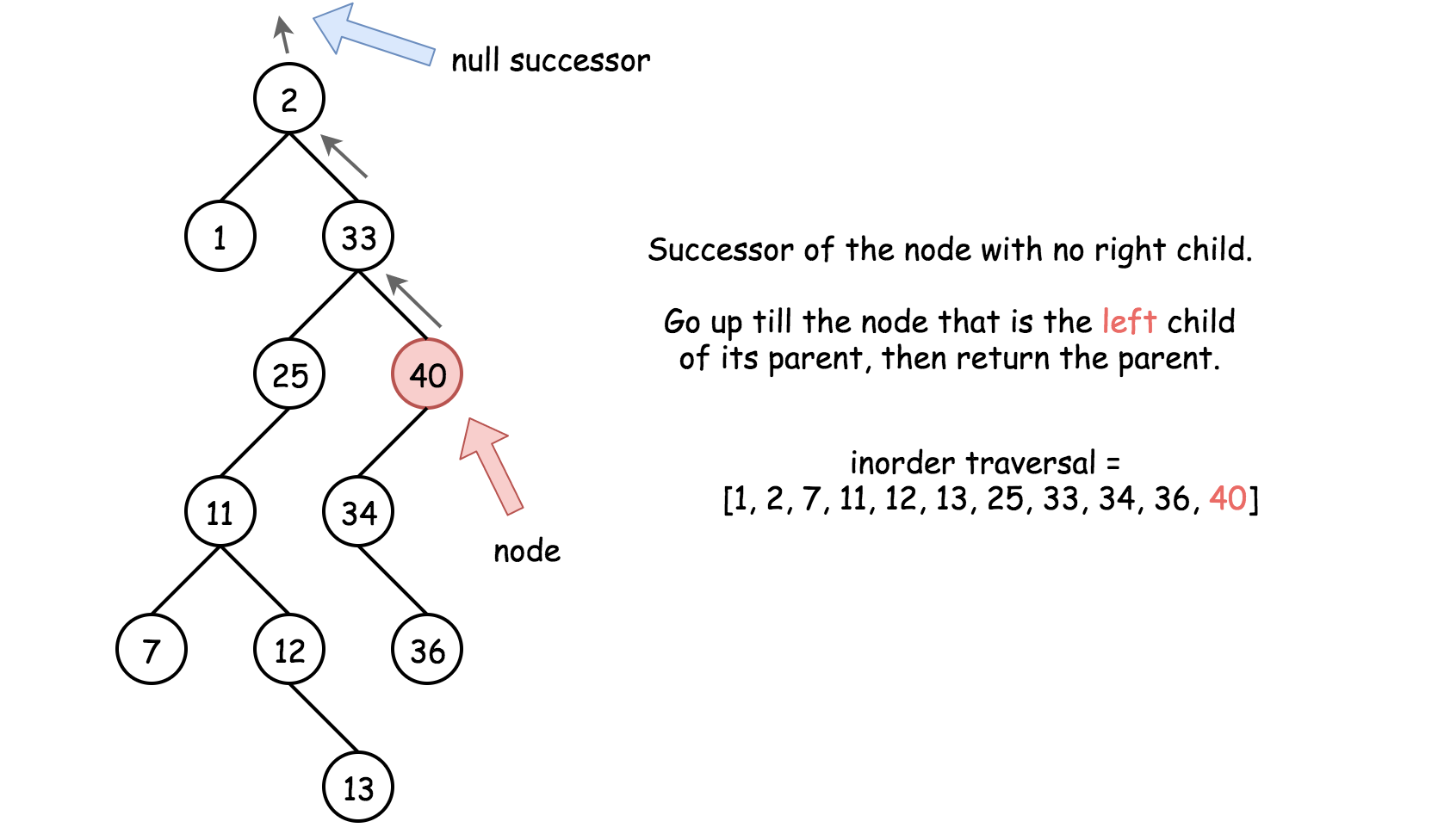
There are two possible situations here :

* Node has a right child, and hence its successor is somewhere lower in the tree. To find the successor, go to the right once and then as many times to the left as you could.



* Node has no right child, then its successor is somewhere upper in the tree. To find the successor, go up till the node that is left child of its parent. The answer is the parent. Beware that there could be no successor (= null successor) in such a situation.





**Algorithm**

1. If the node has a right child, and hence its successor is somewhere lower in the tree. Go to the right once and then as many times to the left as you could. Return the node you end up with.
2. Node has no right child, and hence its successor is somewhere upper in the tree. Go up till the node that is left child of its parent. The answer is the parent.

**Implementation**

|  |
| --- |
| class Solution {  public Node inorderSuccessor(Node x) {  // the successor is somewhere lower in the right subtree  if (x.right != null) {  x = x.right;  while (x.left != null) x = x.left;  return x;  }  // the successor is somewhere upper in the tree  while (x.parent != null && x == x.parent.right) x = x.parent;  return x.parent;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(H)O(*H*), where H*H* is the height of the tree. That means \mathcal{O}(\log N)O(log*N*) in the average case, and \mathcal{O}(N)O(*N*) in the worst case, where N*N* is the number of nodes in the tree.
* Space complexity : \mathcal{O}(1)O(1), since no additional space is allocated during the calculation.

JS recursive solution. Pretty damn proud of my code for once lmao

var inorderSuccessor = function(node) {

if (node.right) return goDown(node.right);

return goUp(node.parent, node.val);

};

function goDown(node) {

if (!node || !node.left) return node;

return goDown(node.left);

}

function goUp(node, val) {

if (!node) return node;

if (node.val > val) return node;

return goUp(node.parent, val);

}

**Length of Last Word**

Given a string s consists of some words separated by spaces, return the length of the last word in the string. If the last word does not exist, return 0.

A **word** is a maximal substring consisting of non-space characters only.

**Example 1:**

**Input:** s = "Hello World"

**Output:** 5

**Example 2:**

**Input:** s = " "

**Output:** 0

**Constraints:**

* 1 <= s.length <= 104
* s consists of only English letters and spaces ' '.

## Solution

#### Approach 1: String Index Manipulation

**Intuition**

There is no doubt that this is an easy problem. Yet, it could be a good exercise for one to practice string manipulation, which is definitely common during interviews.

In this article, we start with some approaches that manipulate string indexes, then we look at how to use the built-in string functions to solve the problem.

The intuition is simple, as it pretty much given away from the name of the problem, i.e. first we **locate** the last word, then we **count** the length of the last word.

One should pay attention to some edge cases though:

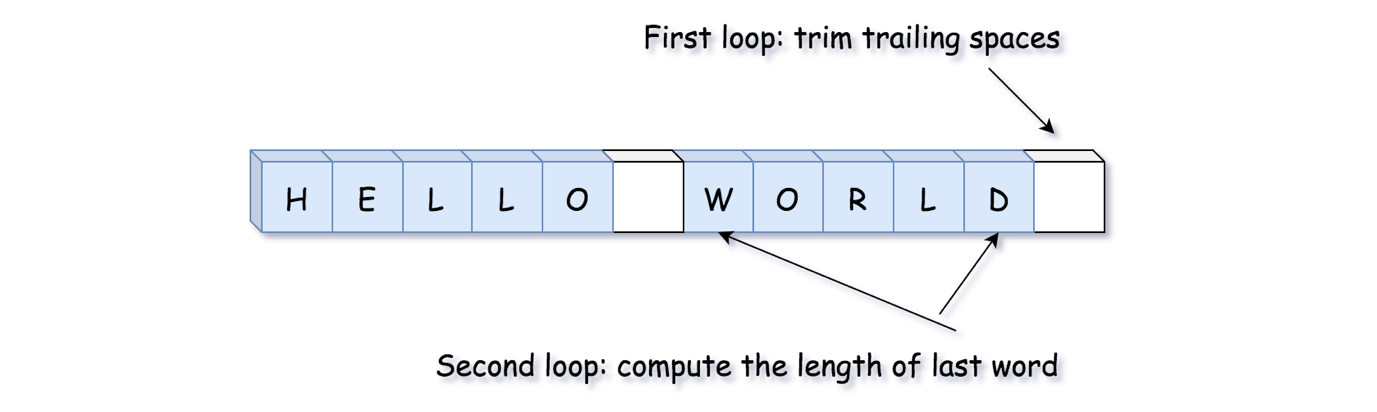
* The input string could be empty.
* There could be some trailing spaces in the input string, e.g. hello <space>.
* There might only be one word in the given string.

The challenge is to build a concise yet comprehensive solution that could handle all above cases.

**Algorithm**

One can break down the solution into two steps:

* First, we would try to locate the last word, starting from the end of the string. We iterate the string in reverse order, consuming the empty spaces. When we first come across a non-space character, we know that we are at the last character of the last word.
* Second, once we locate the last word. We count its length, starting from its last character. Again, we could use a loop here.



Here is what it looks like, a solution with **two loops**:

|  |
| --- |
| class Solution {  public int lengthOfLastWord(String s) {  // trim the trailing spaces  int p = s.length() - 1;  while (p >= 0 && s.charAt(p) == ' ') {  p--;  }  // compute the length of last word  int length = 0;  while (p >= 0 && s.charAt(p) != ' ') {  p--;  length++;  }  return length;  }  } |

**Complexity**

* Time Complexity: \mathcal{O}(N)O(*N*), where N*N* is the length of the input string.

In the worst case, the input string might contain only a single word, which implies that we would need to iterate through the entire string to obtain the result.

* Space Complexity: \mathcal{O}(1)O(1), only constant memory is consumed, regardless the input.

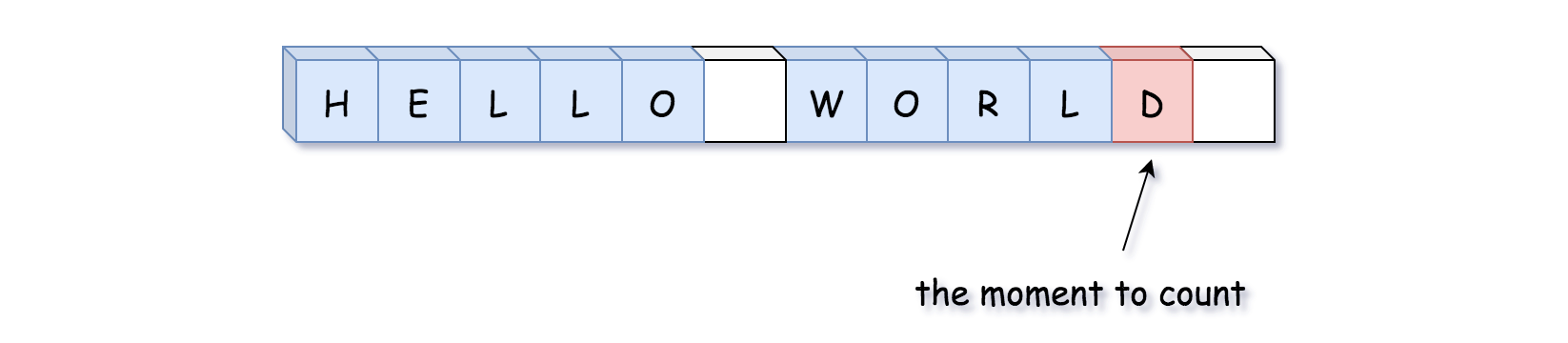
#### Approach 2: One-loop Iteration

**Intuition**

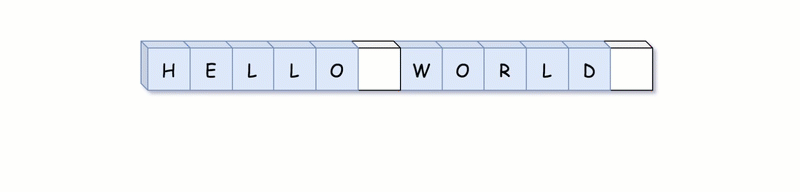
In the above approach, we applied two loops. One is used to locate the last word, and the other one to calculate its length.

We could actually complete the same tasks within a single loop.

The trick is that we could define a condition, i.e. the precise moment that we should start to count the length of the word.



In the following animation, we demonstrate how the above algorithm works.



**Algorithm**

Here are some sample implementations with comments.

|  |
| --- |
| class Solution {  public int lengthOfLastWord(String s) {  int p = s.length(), length = 0;  while (p > 0) {  p--;  // we're in the middle of the last word  if (s.charAt(p) != ' ') {  length++;  }  // here is the end of last word  else if (length > 0) {  return length;  }  }  return length;  }  } |

**Complexity**

* Time Complexity: \mathcal{O}(N)O(*N*), where N*N* is the length of the input string.

This approach has the same time complexity as the previous approach. The only difference is that we combined two loops into one.

* Space Complexity: \mathcal{O}(1)O(1), again a constant memory is consumed, regardless the input.

#### Approach 3: Built-in String Functions

**Intuition**

As we mentioned at the beginning of the article, we could also resort to the **built-in** functions of the String data structure, in order to solve the problem.

In fact, String is such an important data type in many programming languages, that often it comes with a rich set of built-in functions that can help one to accomplish many common tasks, such as trimming the empty spaces in the string etc.

It would be really helpful to get proficient with those built-in functions.

**Algorithm**

In different programming languages, the sets of built-in functions associated with String are different though. In this section, we showcase some examples.

In Python, we would use the following built-in functions to accomplish the tasks:

* str.isspace(): this function determines if the str contains only spaces.
* str.split(delimiter): this function could split the input string into several substrings, based on the given delimiter (by default, the delimiter is space).

One can find the list of built-in functions in Python from the [official documentation](https://docs.python.org/3/library/stdtypes.html#str).

In Java, here is a list of built-in functions that we would use:

* String.trim(): this function returns a copy of the string, with the leading and trailing whitespaces trimmed.
* String.length(): this function returns the length of the string.
* String.lastIndexOf(char): this function returns the index of the last occurrence of the given character.

Again, one can find more details about the APIs for the String data structure in Java, from the [official documentation](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html).

|  |
| --- |
| class Solution {  public int lengthOfLastWord(String s) {  s = s.trim(); // trim the trailing spaces in the string  return s.length() - s.lastIndexOf(" ") - 1;  }  } |

**Complexity Analysis**

* Time Complexity: \mathcal{O}(N)O(*N*), where N*N* is the length of the input string.

Since we use some built-in function from the String data type, we should look into the complexity of each built-in function that we used, in order to obtain the overall time complexity of our algorithm.

It would be safe to assume the time complexity of the methods such as str.split() and String.lastIndexOf() to be \mathcal{O}(N)O(*N*), since in the worst case we would need to scan the entire string for both methods.

* Space Complexity: \mathcal{O}(N)O(*N*). Again, we should look into the built-in functions that we used in the algorithm.

In the Java implementation, we used the function String.trim() which returns a ***copy*** of the input string without leading and trailing whitespace. Therefore, we would need \mathcal{O}(N)O(*N*) space for our algorithm to hold this copy.

In the Python implementation, we used str.split(), which returns a list of **substrings** that are separated by the space delimiter. As a result, we would need \mathcal{O}(N)O(*N*) space for our algorithm to store this list.

**Robot Bounded In Circle**

On an infinite plane, a robot initially stands at (0, 0) and faces north. The robot can receive one of three instructions:

* "G": go straight 1 unit;
* "L": turn 90 degrees to the left;
* "R": turn 90 degrees to the right.

The robot performs the instructions given in order, and repeats them forever.

Return true if and only if there exists a circle in the plane such that the robot never leaves the circle.

**Example 1:**

**Input:** instructions = "GGLLGG"

**Output:** true

**Explanation:** The robot moves from (0,0) to (0,2), turns 180 degrees, and then returns to (0,0).

When repeating these instructions, the robot remains in the circle of radius 2 centered at the origin.

**Example 2:**

**Input:** instructions = "GG"

**Output:** false

**Explanation:** The robot moves north indefinitely.

**Example 3:**

**Input:** instructions = "GL"

**Output:** true

**Explanation:** The robot moves from (0, 0) -> (0, 1) -> (-1, 1) -> (-1, 0) -> (0, 0) -> ...

**Constraints:**

* 1 <= instructions.length <= 100
* instructions[i] is 'G', 'L' or, 'R'.

   Hide Hint #1

Calculate the final vector of how the robot travels after executing all instructions once - it consists of a change in position plus a change in direction.

   Hide Hint #2

The robot stays in the circle iff (looking at the final vector) it changes direction (ie. doesn't stay pointing north), or it moves 0.

## Solution

#### Overview

The robot's [trajectory attractor](https://en.wikipedia.org/wiki/Attractor) is a set of trajectories toward which a system tends to evolve. The question may sound a bit theoretical - is this attractor is limited or not. In other words, if there exists a circle in the plane such that the robot never leaves the circle.

| **Diverging Trajectory** | **Limit Cycle Trajectory** |
| --- | --- |
| bla | bla |

Figure 1. Diverging Trajectory vs Limit Cycle Trajectory.

Why is it interesting to know? There is a bunch of practical problems related to topology, networks planning, and password brute-forcing. For all these problems, the first thing to understand is do we work within a limited space or the behavior of our system could drastically diverge at some point?

| **Diverging Trajectory** | **Limit Cycle Trajectory** |
| --- | --- |
| bla | bla |

Figure 2. Diverging Trajectory vs Limit Cycle Trajectory.

**Draw some trajectories**

[Here is a Jupiter notebook used to draw all figures in this article](https://github.com/leetcode/solution_assets/blob/master/solution_assets/1041_robot_bounded_in_circle/robot_trajectory.ipynb). Do not hesitate to play with it in local or on the online platforms. Drawing different trajectories might help to notice some patterns.

#### Approach 1: One Pass

**Intuition**

This solution is based on two facts about the limit cycle trajectory.

* After at most 4 cycles, the limit cycle trajectory returns to the initial point x = 0, y = 0. That is related to the fact that 4 directions (north, east, south, west) define the repeated cycles' plane symmetry [[proof]](https://leetcode.com/problems/robot-bounded-in-circle/solution/#appendix-mathematical-proof).

| **Ex. 1: Trajectory 1** | **Ex. 2: Trajectory 2** |
| --- | --- |
| bla | bla |

Figure 3. After 4 cycles the limit cycle trajectory returns to the initial point *x = 0, y = 0*.

* We do not need to run 4 cycles to identify the limit cycle trajectory. One cycle is enough. There could be two situations here.
  + First, if the robot returns to the initial point after one cycle, that's the limit cycle trajectory.
  + Second, if the robot doesn't face north at the end of the first cycle, that's the limit cycle trajectory. Once again, that's the consequence of the plane symmetry for the repeated cycles [[proof]](https://leetcode.com/problems/robot-bounded-in-circle/solution/#appendix-mathematical-proof).

| **Ex. 1: Trajectory 1** | **Ex. 2: Trajectory 2** |
| --- | --- |
| bla | bla |

Figure 4. If at the end of one cycle the robot doesn't face north, that's the limit cycle trajectory.

**Algorithm**

* Let's use numbers from 0 to 3 to mark the directions: north = 0, east = 1, south = 2, west = 3. In the array directions we could store corresponding coordinates changes, i.e. directions[0] is to go north, directions[1] is to go east, directions[2] is to go south, and directions[3] is to go west.
* The initial robot position is in the center x = y = 0, facing north idx = 0.
* Now everything is ready to iterate over the instructions.
  + If the current instruction is R, i.e. to turn on the right, the next direction is idx = (idx + 1) % 4. Modulo here is needed to deal with the situation - facing west, idx = 3, turn to the right to face north, idx = 0.
  + If the current instruction is L, i.e. to turn on the left, the next direction could written in a symmetric way idx = (idx - 1) % 4. That means we have to deal with negative indices. A more simple way is to notice that 1 turn to the left = 3 turns to the right: idx = (idx + 3) % 4.
  + If the current instruction is to move, we simply update the coordinates: x += directions[idx][0], y += directions[idx][1].
* After one cycle we have everything to decide. It's a limit cycle trajectory if the robot is back to the center: x = y = 0 or if the robot doesn't face north: idx != 0.

**Implementation**

|  |
| --- |
| class Solution {  public boolean isRobotBounded(String instructions) {  // north = 0, east = 1, south = 2, west = 3  int[][] directions = new int[][]{{0, 1}, {1, 0}, {0, -1}, {-1, 0}};  // Initial position is in the center  int x = 0, y = 0;  // facing north  int idx = 0;    for (char i : instructions.toCharArray()) {  if (i == 'L')  idx = (idx + 3) % 4;  else if (i == 'R')  idx = (idx + 1) % 4;  else {  x += directions[idx][0];  y += directions[idx][1];  }  }    // after one cycle:  // robot returns into initial position  // or robot doesn't face north  return (x == 0 && y == 0) || (idx != 0);  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*), where N*N* is a number of instructions to parse.
* Space complexity: \mathcal{O}(1)O(1) because the array directions contains only 4 elements.

#### Appendix: Mathematical Proof

Let's provide a strict mathematical proof.

**If the robot doesn't face north at the end of the first cycle, then that's the limit cycle trajectory.**

First, let's check which direction the robot faces after 4 cycles.

Let's use numbers from 0 to 3 to mark the directions: north = 0, east = 1, south = 2, west = 3. After one cycle the robot is facing direction k != 0.

After 4 cycles, the robot faces direction (k \* 4) % 4 = 0, i.e. after 4 cycles, the robot is always facing north.

Second, let's find the robot coordinates after 4 cycles.

The robot initial coordinates are x = y = 0. After one cycle, the new coordinates are x\_1 = x + \Delta x*x*1​=*x*+Δ*x*, y\_1 = y + \Delta y*y*1​=*y*+Δ*y*, where both \Delta xΔ*x* and \Delta yΔ*y* could be positive or negative.

Let's consider four situations.

* After one cycle, the robot faces north.

Then here is what we have after 4 cycles:

x\_4 = x + \Delta x + \Delta x - \Delta x + \Delta x = x + 4 \Delta x*x*4​=*x*+Δ*x*+Δ*x*−Δ*x*+Δ*x*=*x*+4Δ*x*

y\_4 = y + \Delta y + \Delta y + \Delta y + \Delta y = y + 4 \Delta y*y*4​=*y*+Δ*y*+Δ*y*+Δ*y*+Δ*y*=*y*+4Δ*y*

* After one cycle, the robot faces east.

Then here is what we have after 4 cycles:

x\_4 = x + \Delta x + \Delta y - \Delta x - \Delta y = x*x*4​=*x*+Δ*x*+Δ*y*−Δ*x*−Δ*y*=*x*

y\_4 = y + \Delta y - \Delta x - \Delta y + \Delta x = y*y*4​=*y*+Δ*y*−Δ*x*−Δ*y*+Δ*x*=*y*

* After one cycle, the robot faces south.

Then here is what we have after 4 cycles:

x\_4 = x + \Delta x - \Delta x + \Delta x - \Delta x = x*x*4​=*x*+Δ*x*−Δ*x*+Δ*x*−Δ*x*=*x*

y\_4 = y + \Delta y - \Delta y + \Delta y - \Delta y = y*y*4​=*y*+Δ*y*−Δ*y*+Δ*y*−Δ*y*=*y*

* After one cycle, the robot faces west.

Then here is what we have after 4 cycles:

x\_4 = x + \Delta x - \Delta y - \Delta x + \Delta y = x*x*4​=*x*+Δ*x*−Δ*y*−Δ*x*+Δ*y*=*x*

y\_4 = y + \Delta y + \Delta x - \Delta y - \Delta x = y*y*4​=*y*+Δ*y*+Δ*x*−Δ*y*−Δ*x*=*y*

Hence, if the robot doesn't face north at the end of the first cycle, then after 4 cycles, the robot is back to the initial coordinates and faces north.

The following statement is a straight consequence.

**After at most 4 cycles, the limit cycle trajectory returns to the initial point.**

**Sequential Digits**

An integer has sequential digits if and only if each digit in the number is one more than the previous digit.

Return a **sorted** list of all the integers in the range [low, high] inclusive that have sequential digits.

**Example 1:**

**Input:** low = 100, high = 300

**Output:** [123,234]

**Example 2:**

**Input:** low = 1000, high = 13000

**Output:** [1234,2345,3456,4567,5678,6789,12345]

**Constraints:**

* 10 <= low <= high <= 10^9

   Hide Hint #1

Generate all numbers with sequential digits and check if they are in the given range.

   Hide Hint #2

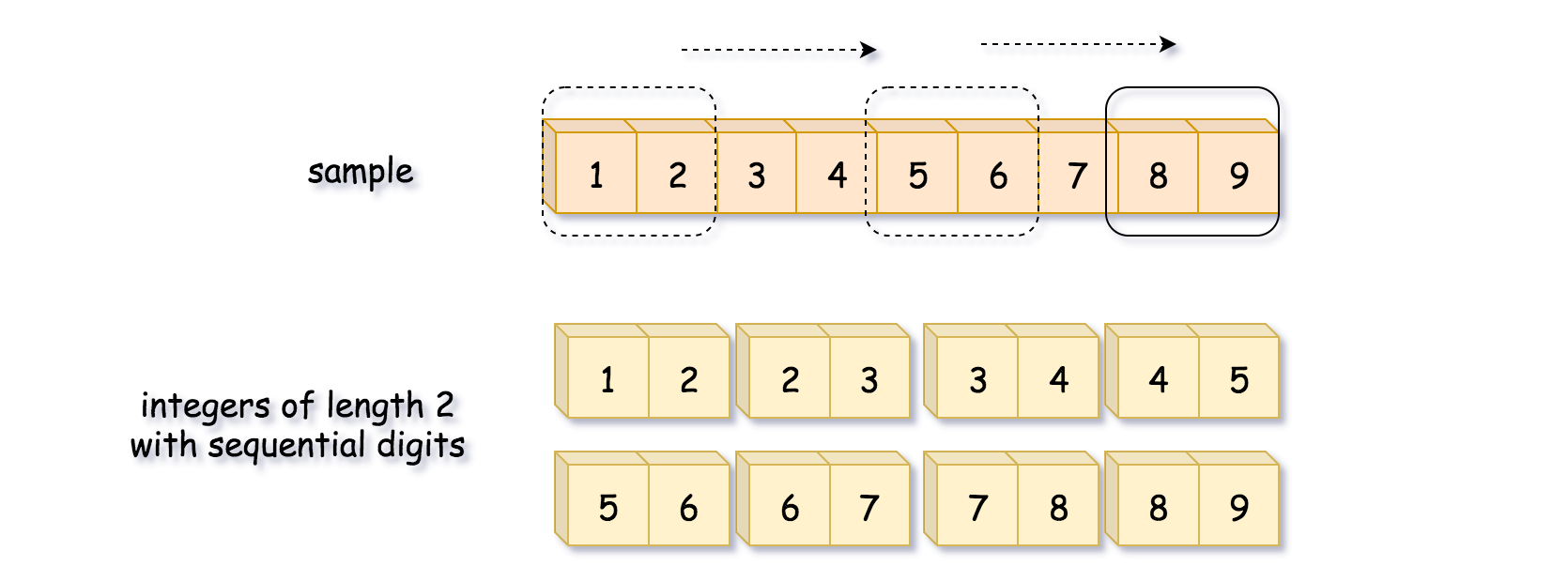
Fix the starting digit then do a recursion that tries to append all valid digits.

## Solution

#### Approach 1: Sliding Window

One might notice that all integers that have sequential digits are substrings of string "123456789". Hence to generate all such integers of a given length, just move the window of that length along "123456789" string.

The advantage of this method is that it will generate the integers that are already in the sorted order.



**Algorithm**

* Initialize sample string "123456789". This string contains all integers that have sequential digits as substrings. Let's implement sliding window algorithm to generate them.
* Iterate over all possible string lengths: from the length of low to the length of high.
  + For each length iterate over all possible start indexes: from 0 to 10 - length.
    - Construct the number from digits inside the sliding window of current length.
    - Add this number in the output list nums, if it's greater than low and less than high.
* Return nums.

**Implementation**

|  |
| --- |
| class Solution {  public List<Integer> sequentialDigits(int low, int high) {  String sample = "123456789";  int n = 10;  List<Integer> nums = new ArrayList();  int lowLen = String.valueOf(low).length();  int highLen = String.valueOf(high).length();  for (int length = lowLen; length < highLen + 1; ++length) {  for (int start = 0; start < n - length; ++start) {  int num = Integer.parseInt(sample.substring(start, start + length));  if (num >= low && num <= high) nums.add(num);  }  }  return nums;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(1)O(1). Length of sample string is 9, and lengths of low and high are between 2 and 9. Hence the nested loops are executed no more than 8 \times 8 = 648×8=64 times.
* Space complexity: \mathcal{O}(1)O(1) to keep not more than 36 integers with sequential digits.

#### Approach 2: Precomputation

Actually, there are 36 integers with the sequential digits. Here is how we calculate it.

Starting from 9 digits in the sample string, one could construct 9 - 2 + 1 = 8 integers of length 2, 9 - 3 + 1 = 7 integers of length 3, and so on and so forth. In total, it would make 8 + 7 + ... + 1 = 36 integers.

As one can see, we could precompute the results all at once and then select the integers that are less than high and greater than low.

**Implementation**

|  |
| --- |
| class Seq {  public List<Integer> nums = new ArrayList();  Seq() {  String sample = "123456789";  int n = 10;  for (int length = 2; length < n; ++length) {  for (int start = 0; start < n - length; ++start) {  int num = Integer.parseInt(sample.substring(start, start + length));  nums.add(num);  }  }  }  }  class Solution {  public static Seq s = new Seq();  public List<Integer> sequentialDigits(int low, int high) {  List<Integer> output = new ArrayList();  for (int num : s.nums) {  if (num >= low && num <= high) output.add(num);  }  return output;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(1)O(1) both for precomputation and during runtime. Precomputation: Length of sample string is 9, and the nested loops are executed 8 \times 8 = 648×8=64 times. Runtime: One iterates over an array of 36 integers.
* Space complexity: \mathcal{O}(1)O(1) to keep 36 integers that have sequential digits.

**Unique Paths III**

On a 2-dimensional grid, there are 4 types of squares:

* 1 represents the starting square.  There is exactly one starting square.
* 2 represents the ending square.  There is exactly one ending square.
* 0 represents empty squares we can walk over.
* -1 represents obstacles that we cannot walk over.

Return the number of 4-directional walks from the starting square to the ending square, that **walk over every non-obstacle square exactly once**.

**Example 1:**

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

**Output:** 2

**Explanation:** We have the following two paths:

1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2)

2. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2)

**Example 2:**

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

**Output:** 4

**Explanation:** We have the following four paths:

1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2),(2,3)

2. (0,0),(0,1),(1,1),(1,0),(2,0),(2,1),(2,2),(1,2),(0,2),(0,3),(1,3),(2,3)

3. (0,0),(1,0),(2,0),(2,1),(2,2),(1,2),(1,1),(0,1),(0,2),(0,3),(1,3),(2,3)

4. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2),(2,3)

**Example 3:**

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

**Output:** 0

**Explanation:**

There is no path that walks over every empty square exactly once.

Note that the starting and ending square can be anywhere in the grid.

**Note:**

1. 1 <= grid.length \* grid[0].length <= 20

## Solution

#### Overview

Whenever we see the context of grid traversal, the technique of backtracking or DFS (Depth-First Search) should ring a bell.

In terms of this problem, it fits the bill perfectly, with a canonical setting, unlike another similar problem called [robot room cleaner](https://leetcode.com/problems/robot-room-cleaner/) which has certain twists.

As a reminder, [backtracking](https://en.wikipedia.org/wiki/Backtracking) is a general algorithm for finding all (or some) solutions to some problems with constraints. It incrementally builds candidates to the solutions, and abandons a candidate as soon as it determines that the candidate cannot possibly lead to a solution.

In this article, we will showcase how to apply the backtracking algorithm to solve this problem.

#### Approach 1: Backtracking

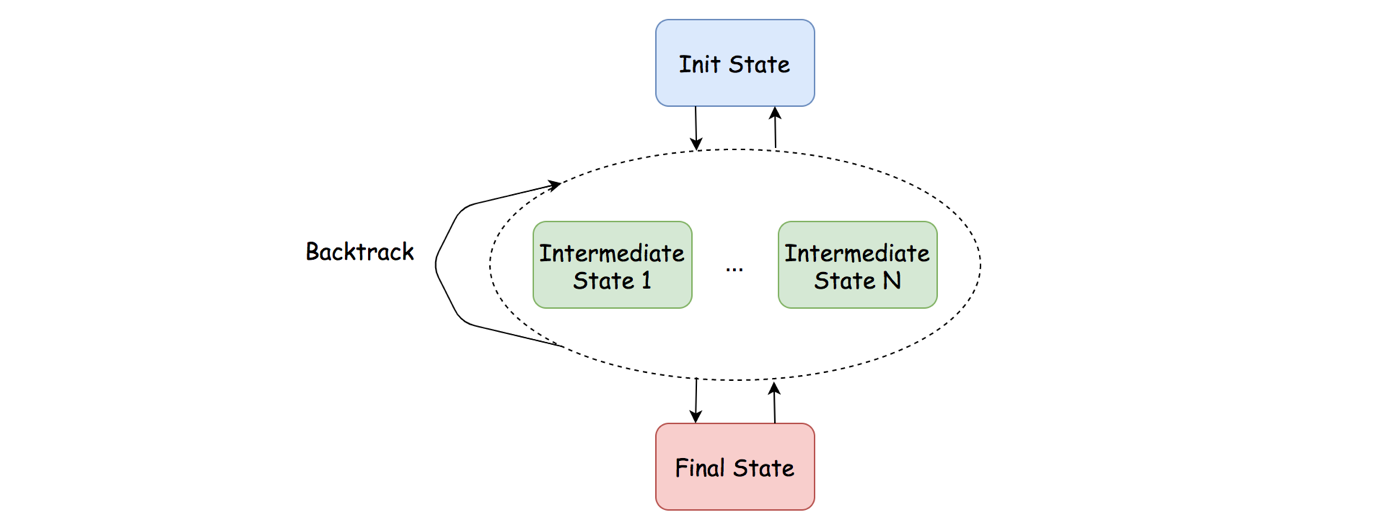
**Intuition**

We can consider backtracking as a state machine, where we start off from an initial state, each action we take will move the state from one to another, and there should be some final state where we reach our goal.

As a result, let us first clarify the initial and the final states of the problem.

* **Initial State**
  + There are different types of squares/cells in a grid.
  + There are an origin and a destination cell, which are not given explicitly.
  + Initially, all the cells are not **visited**.
* **Final State**
  + We reach the destination cell, i.e. cell filled with the value 2.
  + We have visited all the non-obstacle cells, including the empty cells (i.e. filled with 0) and the initial cell (i.e. 1).

With the above definition, we can then translate the problem as finding all paths that can lead us from the initial state to the final state.



More specifically, we could summarise the steps to implement the backtracking algorithm for this problem in the following pseudo code.

def backtrack(cell):

1. if we arrive at the final state:

path\_count ++

return

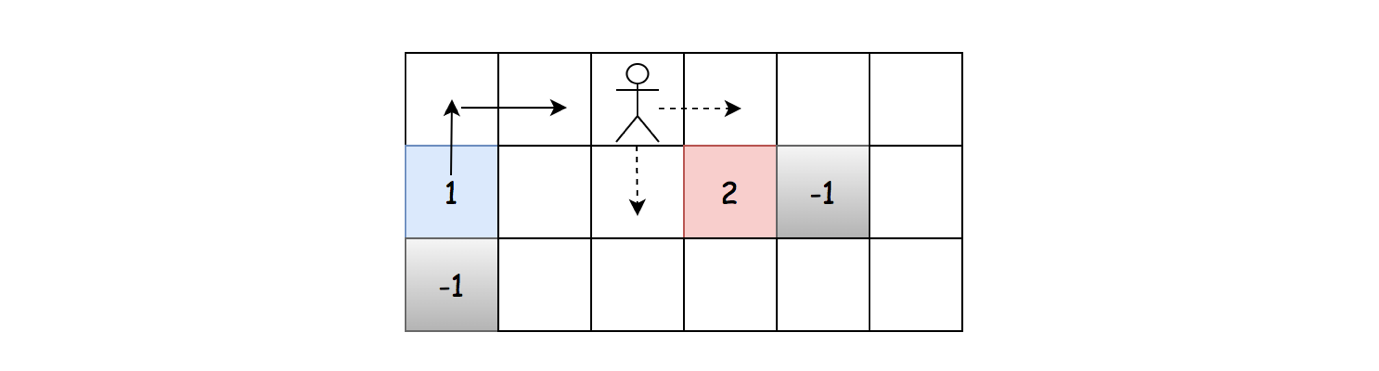
2. mark the current cell as visited

3. for next\_cell in 4 directions:

if next\_cell is not visited and non-obstacle:

backtrack(next\_cell)

4. unmark the current cell



**Algorithm**

As one can see, backtracking is more of a methodology to solve a specific type of problems. For a backtracking problem, it would not be exaggerating to say that there are a thousand backtracking implementations in a thousand people's eyes, as one would find out in the implementation later.

Here we would simply show one example of implementation, following the pseudo code shown in the intuition section.

|  |
| --- |
| class Solution {  int rows, cols;  int[][] grid;  int path\_count;  protected void backtrack(int row, int col, int remain) {  // base case for the termination of backtracking  if (this.grid[row][col] == 2 && remain == 1) {  // reach the destination  this.path\_count += 1;  return;  }  // mark the square as visited. case: 0, 1, 2  int temp = grid[row][col];  grid[row][col] = -4;  remain -= 1; // we now have one less square to visit  // explore the 4 potential directions around  int[] row\_offsets = {0, 0, 1, -1};  int[] col\_offsets = {1, -1, 0, 0};  for (int i = 0; i < 4; ++i) {  int next\_row = row + row\_offsets[i];  int next\_col = col + col\_offsets[i];  if (0 > next\_row || next\_row >= this.rows ||  0 > next\_col || next\_col >= this.cols)  // invalid coordinate  continue;  if (grid[next\_row][next\_col] < 0)  // either obstacle or visited square  continue;  backtrack(next\_row, next\_col, remain);  }  // unmark the square after the visit  grid[row][col] = temp;  }  public int uniquePathsIII(int[][] grid) {  int non\_obstacles = 0, start\_row = 0, start\_col = 0;  this.rows = grid.length;  this.cols = grid[0].length;  // step 1). initialize the conditions for backtracking  // i.e. initial state and final state  for (int row = 0; row < rows; ++row)  for (int col = 0; col < cols; ++col) {  int cell = grid[row][col];  if (cell >= 0)  non\_obstacles += 1;  if (cell == 1) {  start\_row = row;  start\_col = col;  }  }  this.path\_count = 0;  this.grid = grid;  // kick-off the backtracking  backtrack(start\_row, start\_col, non\_obstacles);  return this.path\_count;  }  } |

Here we would like to highlight some important design decisions we took in the above implementation. As one can imagine, with different decisions, one would have variations of backtracking implementations.

* **In-place Modification**
  + This is an important technique that allows us to save some space in the algorithm.
  + In order to mark the cell as **visited**, often the case we use some matrix or hashtable with boolean values to keep track of the state of each cell, i.e. whether it is visited or not.
  + With the in-place technique, we simply assign a specific value to the cell in the grid, rather than creating an additional matrix or hashtable.
* **Boundary Check**
  + There are several boundary conditions that we could check during the backtracking, namely whether the coordinate of a cell is valid or not and whether the current cell is visited or not.
  + We could do the checking right before we make the recursive call, or at the beginning of the backtrack function.
  + We decided to go with the former one, which could save us some recursive calls when the boundary check does not pass.

**Complexity Analysis**

Let N*N* be the total number of cells in the input grid.

* Time Complexity: \mathcal{O}(3^N)O(3*N*)
  + Although technically we have 4 directions to explore at each step, we have at most 3 directions to try at any moment except the first step. The last direction is the direction where we came from, therefore we don't need to explore it, since we have been there before.
  + In the worst case where none of the cells is an obstacle, we have to explore each cell. Hence, the time complexity of the algorithm is \mathcal{O}(4 \* 3 ^{(N-1)}) = \mathcal{O}(3^N)O(4∗3(*N*−1))=O(3*N*).
* Space Complexity: \mathcal{O}(N)O(*N*)
  + Thanks to the in-place technique, we did not use any additional memory to keep track of the state.
  + On the other hand, we apply recursion in the algorithm, which could incur \mathcal{O}(N)O(*N*) space in the function call stack.
  + Hence, the overall space complexity of the algorithm is \mathcal{O}(N)O(*N*).

**Car Pooling**

You are driving a vehicle that has capacity empty seats initially available for passengers.  The vehicle **only** drives east (ie. it **cannot** turn around and drive west.)

Given a list of trips, trip[i] = [num\_passengers, start\_location, end\_location] contains information about the i-th trip: the number of passengers that must be picked up, and the locations to pick them up and drop them off.  The locations are given as the number of kilometers due east from your vehicle's initial location.

Return true if and only if it is possible to pick up and drop off all passengers for all the given trips.

**Example 1:**

**Input:** trips = [[2,1,5],[3,3,7]], capacity = 4

**Output:** false

**Example 2:**

**Input:** trips = [[2,1,5],[3,3,7]], capacity = 5

**Output:** true

**Example 3:**

**Input:** trips = [[2,1,5],[3,5,7]], capacity = 3

**Output:** true

**Example 4:**

**Input:** trips = [[3,2,7],[3,7,9],[8,3,9]], capacity = 11

**Output:** true

**Constraints:**

1. trips.length <= 1000
2. trips[i].length == 3
3. 1 <= trips[i][0] <= 100
4. 0 <= trips[i][1] < trips[i][2] <= 1000
5. 1 <= capacity <= 100000

   Hide Hint #1

Sort the pickup and dropoff events by location, then process them in order.

## Solution

### **Overview**

It is one of the classical problems related to intervals, and we have some similar problems such as [Meeting Rooms II](https://leetcode.com/problems/meeting-rooms-ii/) at LeetCode. Below, two approaches are introduced: the simple Time Stamp approach, and the Bucket Sort approach.

### **Approach 1: Time Stamp**

**Intuition**

A simple idea is to go through from the start to end, and check if the actual capacity exceeds capacity.

To know the actual capacity, we just need the number of passengers changed at each timestamp.

We can save the number of passengers changed at each time, sort it by timestamp, and finally iterate it to check the actual capacity.

**Algorithm**

We will initialize a list to store the number of passengers changed and the corresponding timestamp and then sort it.

Note that in Java, we do not have a nice API to do this. However, we can use a TreeMap, which can help us to sort during insertion. You can use a PriorityQueue instead.

Finally, we just need to iterate from the start timestamp to the end timestamp and check if the actual capacity meets the condition.

|  |
| --- |
| class Solution {  public boolean carPooling(int[][] trips, int capacity) {  Map<Integer, Integer> timestamp = new TreeMap<>();  for (int[] trip : trips) {  int startPassenger = timestamp.getOrDefault(trip[1], 0) + trip[0];  timestamp.put(trip[1], startPassenger);  int endPassenger = timestamp.getOrDefault(trip[2], 0) - trip[0];  timestamp.put(trip[2], endPassenger);  }  int usedCapacity = 0;  for (int passengerChange : timestamp.values()) {  usedCapacity += passengerChange;  if (usedCapacity > capacity) {  return false;  }  }  return true;  }  } |

**Complexity Analysis**

Assume N*N* is the length of trips.

* Time complexity: \mathcal{O}(N\log(N))O(*N*log(*N*)) since we need to iterate over trips and sort our timestamp. Iterating costs \mathcal{O}(N)O(*N*), and sorting costs \mathcal{O}(N\log(N))O(*N*log(*N*)), and adding together we have \mathcal{O}(N) + \mathcal{O}(N\log(N)) = \mathcal{O}(N\log(N))O(*N*)+O(*N*log(*N*))=O(*N*log(*N*)).
* Space complexity: \mathcal{O}(N)O(*N*) since in the worst case we need \mathcal{O}(N)O(*N*) to store timestamp.

### **Approach 2: Bucket Sort**

**Intuition**

Note that in the problem there is a interesting constraint:

1. 0 <= trips[i][1] < trips[i][2] <= 1000

What pops into the mind is [Bucket Sort](https://en.wikipedia.org/wiki/Bucket_sort), which is a sorting algorithm in \mathcal{O}(N)O(*N*) time but requires some prior knowledge for the range of the data.

We can use it instead of the normal sorting in this method.

What we do is initial 1001 buckets, and put the number of passengers changed in corresponding buckets, and collect the buckets one by one.

**Algorithm**

We will initial 1001 buckets, iterate trip, and save the number of passengers changed at i mile in the i-th bucket.

|  |
| --- |
| class Solution {  public boolean carPooling(int[][] trips, int capacity) {  int[] timestamp = new int[1001];  for (int[] trip : trips) {  timestamp[trip[1]] += trip[0];  timestamp[trip[2]] -= trip[0];  }  int usedCapacity = 0;  for (int number : timestamp) {  usedCapacity += number;  if (usedCapacity > capacity) {  return false;  }  }  return true;  }  } |

**Complexity Analysis**

Assume N*N* is the length of trip.

* Time complexity: \mathcal{O}(max(N, 1001))O(*max*(*N*,1001)) since we need to iterate over trips and then iterate over our 1001 buckets.
* Space complexity : \mathcal{O}(1001)=\mathcal{O}(1)O(1001)=O(1) since we have 1001 buckets.

|  |
| --- |
| class Solution {  private class Leg implements Comparable<Leg> {  public int location;  public int numPassengers;    public Leg(int location, int numPassengers) {  this.location = location;  this.numPassengers = numPassengers;  }    public int compareTo(Leg that) {  if (this.location == that.location) {  return this.numPassengers - that.numPassengers;  }  return this.location - that.location;  }  }  public boolean carPooling(int[][] trips, int capacity) {  if (trips.length == 0) {  return false;  }  PriorityQueue<Leg> pq = new PriorityQueue();    for (int[] trip : trips) {  pq.add(new Leg(trip[1], trip[0]));  pq.add(new Leg(trip[2], -trip[0]));  }    int rem = capacity;  while (pq.size() > 0) {  rem = rem - pq.poll().numPassengers;  if (rem < 0) {  return false;  }  }    return true;  }  } |

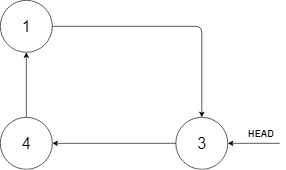
**Insert into a Sorted Circular Linked List**

Given a node from a **Circular Linked List** which is sorted in ascending order, write a function to insert a value insertVal into the list such that it remains a sorted circular list. The given node can be a reference to any single node in the list, and may not be necessarily the smallest value in the circular list.

If there are multiple suitable places for insertion, you may choose any place to insert the new value. After the insertion, the circular list should remain sorted.

If the list is empty (i.e., given node is null), you should create a new single circular list and return the reference to that single node. Otherwise, you should return the original given node.

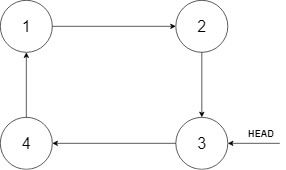
**Example 1:**



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

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

**Explanation:** In the figure above, there is a sorted circular list of three elements. You are given a reference to the node with value 3, and we need to insert 2 into the list. The new node should be inserted between node 1 and node 3. After the insertion, the list should look like this, and we should still return node 3.



**Example 2:**

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

**Output:** [1]

**Explanation:** The list is empty (given head is null). We create a new single circular list and return the reference to that single node.

**Example 3:**

**Input:** head = [1], insertVal = 0

**Output:** [1,0]

**Constraints:**

* 0 <= Number of Nodes <= 5 \* 10^4
* -10^6 <= Node.val <= 10^6
* -10^6 <= insertVal <= 10^6

## Solution

#### Approach 1: Two-Pointers Iteration

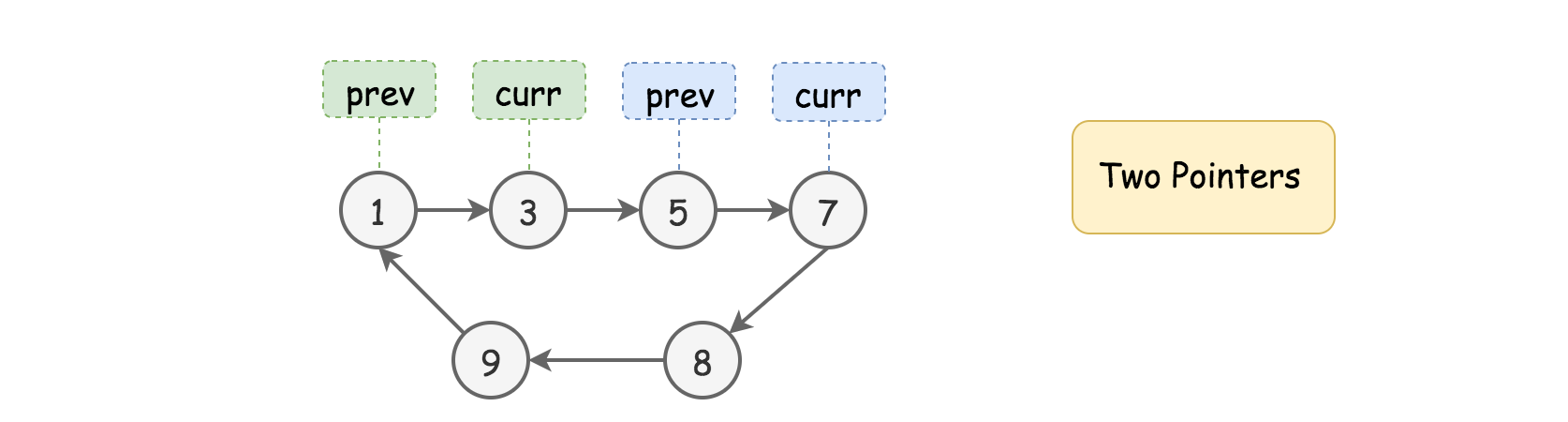
**Intuition**

As simple as the problem might seem to be, it is actually not trivial to write a solution that covers all cases.

Often the case for the problems with linked list, one could apply the approach of **Two-Pointers Iteration**, where one uses two pointers as surrogate to traverse the linked list.

One of reasons of having two pointers rather than one is that in singly-linked list one does not have a reference to the precedent node, therefore we keep an additional pointer which points to the precedent node.

For this problem, we iterate through the cyclic list using two pointers, namely prev and curr. When we find a suitable place to insert the new value, we insert it between the prev and curr nodes.



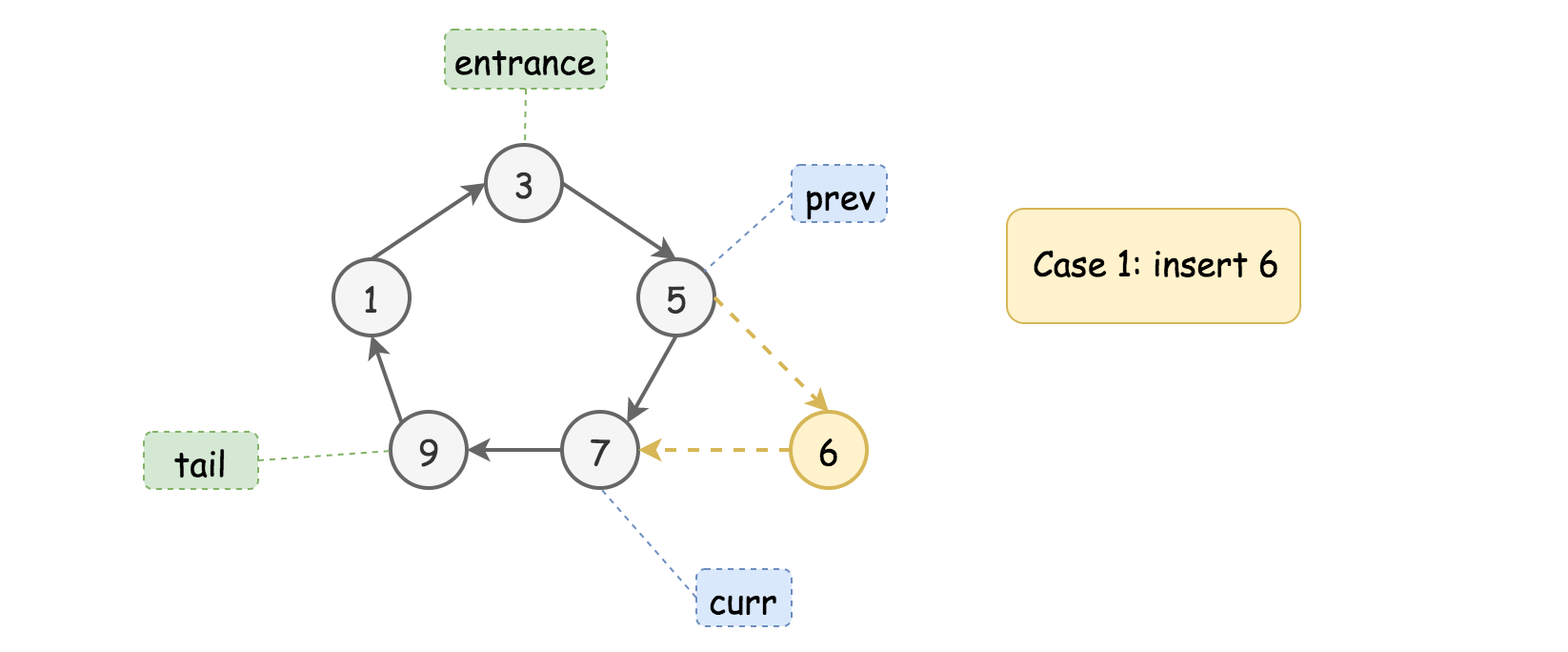
**Algorithm**

First of all, let us define the skeleton of two-pointers iteration algorithm as follows:

* As we mentioned in the intuition, we loop over the linked list with two pointers (i.e. prev and curr) step by step. The termination condition of the loop is that we get back to the starting point of the two pointers (i.e. prev == head)
* During the loop, at each step, we check if the current place bounded by the two pointers is the right place to insert the new value.
* If not, we move both pointers one step forwards.

Now, the tricky part of this problem is to sort out different cases that our algorithm should deal with within the loop, and then design a concise logic to handle them sound and properly. Here we break it down into three general cases.

**Case 1).** The value of new node sits between the minimal and maximal values of the current list. As a result, it should be inserted within the list.



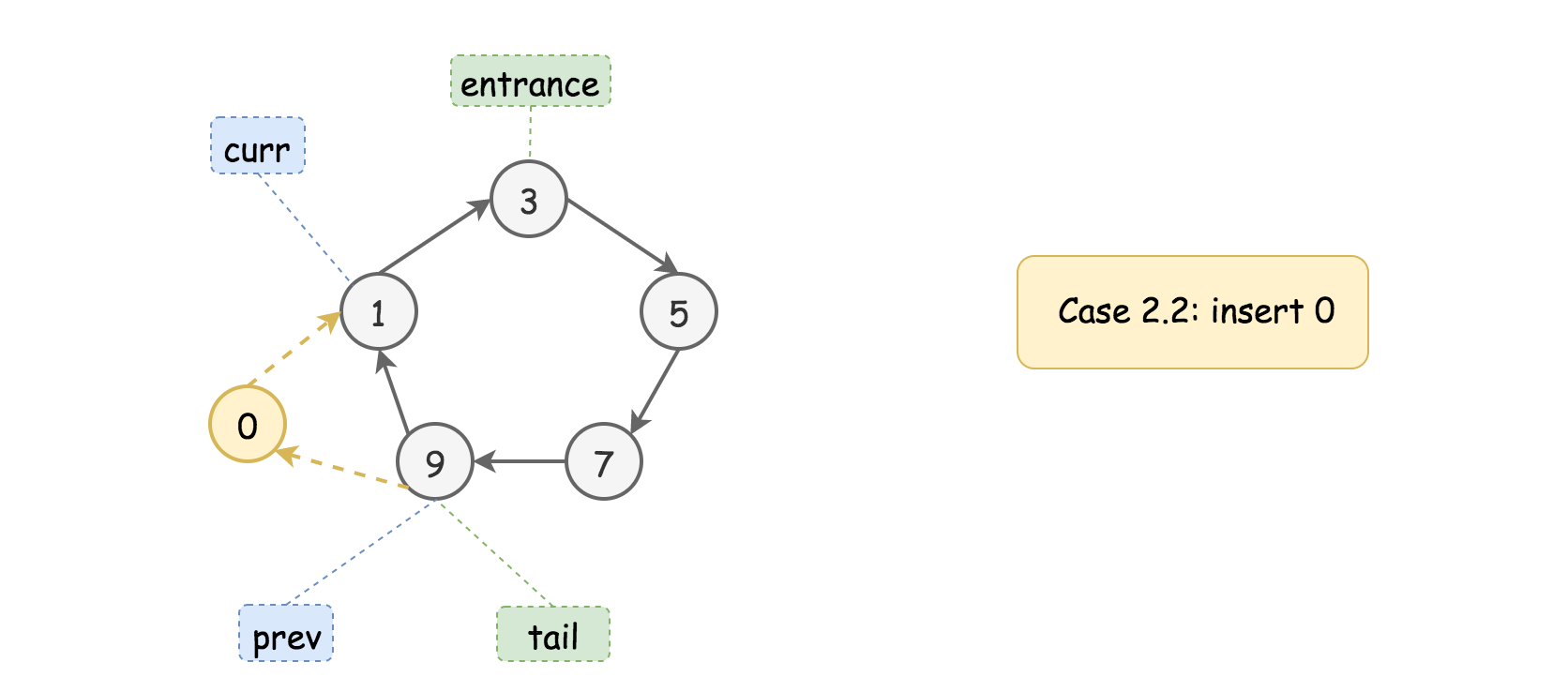
As we can see from the above example, the new value (6) sits between the minimal and maximal values of the list (i.e. 1 and 9). No matter where we start from (in this example we start from the node {3}), the new node would end up being inserted between the nodes {5} and {7}.

The condition is to find the place that meets the constraint of *{prev.val <= insertVal <= curr.val}*.

**Case 2).** The value of new node goes beyond the minimal and maximal values of the current list, either less than the minimal value or greater than the maximal value. In either case, the new node should be added right after the tail node (i.e. the node with the maximal value of the list).

Here are the examples with the same input list as in the previous example.





Firstly, we should locate the position of the **tail** node, by finding a descending order between the adjacent, i.e. the condition of {prev.val > curr.val}, since the nodes are sorted in ascending order, the tail node would have the greatest value of all nodes.

Furthermore, we check if the new value goes beyond the values of tail and head nodes, which are pointed by the prev and curr pointers respectively.

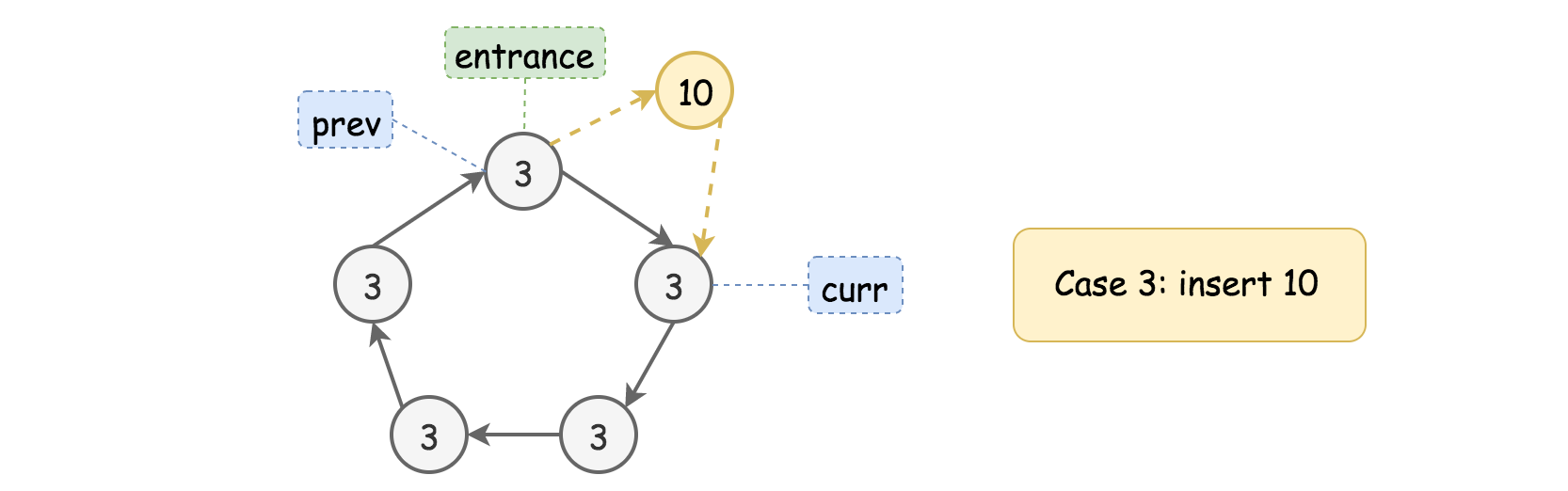
The Case 2.1 corresponds to the condition where the value to be inserted is greater than or equal to the one of tail node, i.e. {insertVal >= prev.val}.

The Case 2.2 corresponds to the condition where the value to be inserted is less than or equal to the head node, i.e. {insertVal <= curr.val}.

Once we locate the tail and head nodes, we basically **extend** the original list by inserting the value in between the tail and head nodes, i.e. in between the prev and curr pointers, the same operation as in the Case 1.

Case 3). Finally, there is one case that does not fall into any of the above two cases. This is the case where the list contains uniform values.

Though not explicitly stated in the problem description, our sorted list can contain some duplicate values. And in the extreme case, the entire list has only one single unique value.



In this case, we would end up looping through the list and getting back to the starting point.

The followup action is just to add the new node after any node in the list, regardless the value to be inserted. Since we are back to the starting point, we might as well add the new node right after the starting point (our entrance node).

Note that, we cannot skip the iteration though, since we have to iterate through the list to determine if our list contains a single unique value.

The above three cases cover the scenarios within and after our iteration loop. There is however one minor **corner** case we still need to deal with, where we have an **empty** list. This, we could easily handle before the loop.

|  |
| --- |
| class Solution {  public Node insert(Node head, int insertVal) {  if (head == null) {  Node newNode = new Node(insertVal, null);  newNode.next = newNode;  return newNode;  }  Node prev = head;  Node curr = head.next;  boolean toInsert = false;  do {  if (prev.val <= insertVal && insertVal <= curr.val) {  // Case 1).  toInsert = true;  } else if (prev.val > curr.val) {  // Case 2).  if (insertVal >= prev.val || insertVal <= curr.val)  toInsert = true;  }  if (toInsert) {  prev.next = new Node(insertVal, curr);  return head;  }  prev = curr;  curr = curr.next;  } while (prev != head);  // Case 3).  prev.next = new Node(insertVal, curr);  return head;  }  } |

**Complexity Analysis**

* Time Complexity: \mathcal{O}(N)O(*N*) where N*N* is the size of list. In the worst case, we would iterate through the entire list.
* Space Complexity: \mathcal{O}(1)O(1). It is a constant space solution.

**Gas Station**

There are n gas stations along a circular route, where the amount of gas at the ith station is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the ith station to its next (i + 1)th station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays gas and cost, return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1. If there exists a solution, it is **guaranteed** to be **unique**

**Example 1:**

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

**Output:** 3

**Explanation:**

Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 4. Your tank = 4 - 1 + 5 = 8

Travel to station 0. Your tank = 8 - 2 + 1 = 7

Travel to station 1. Your tank = 7 - 3 + 2 = 6

Travel to station 2. Your tank = 6 - 4 + 3 = 5

Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.

Therefore, return 3 as the starting index.

**Example 2:**

**Input:** gas = [2,3,4], cost = [3,4,3]

**Output:** -1

**Explanation:**

You can't start at station 0 or 1, as there is not enough gas to travel to the next station.

Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 0. Your tank = 4 - 3 + 2 = 3

Travel to station 1. Your tank = 3 - 3 + 3 = 3

You cannot travel back to station 2, as it requires 4 unit of gas but you only have 3.

Therefore, you can't travel around the circuit once no matter where you start.

**Constraints:**

* gas.length == n
* cost.length == n
* 1 <= n <= 104
* 0 <= gas[i], cost[i] <= 104

## Solution

#### Approach 1: One pass.

**Intuition**

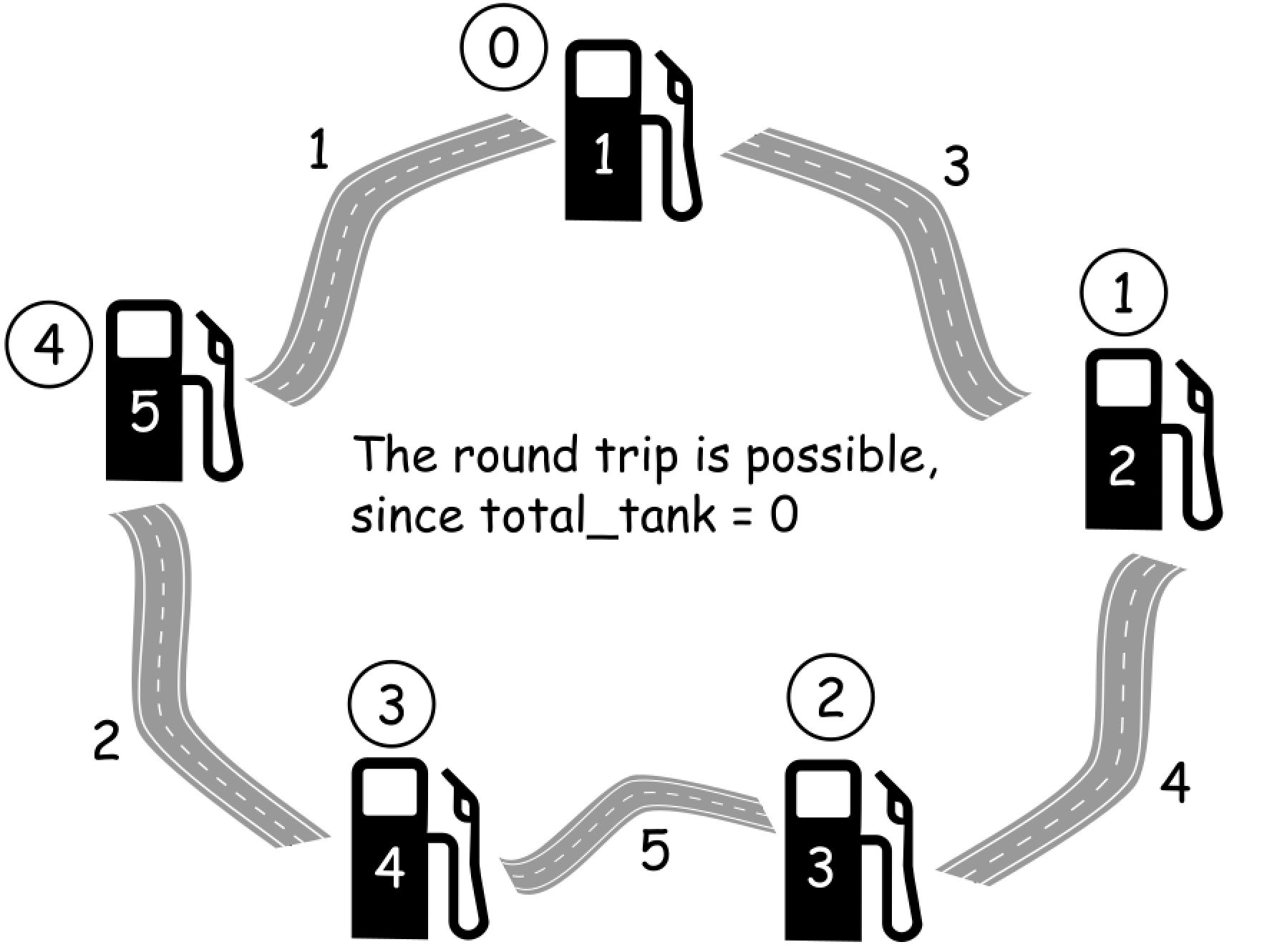
The first idea is to check every single station :

* Choose the station as starting point.
* Perform the road trip and check how much gas we have in tank at each station.

That means \mathcal{O}(N^2)O(*N*2) time complexity, and for sure one could do better.

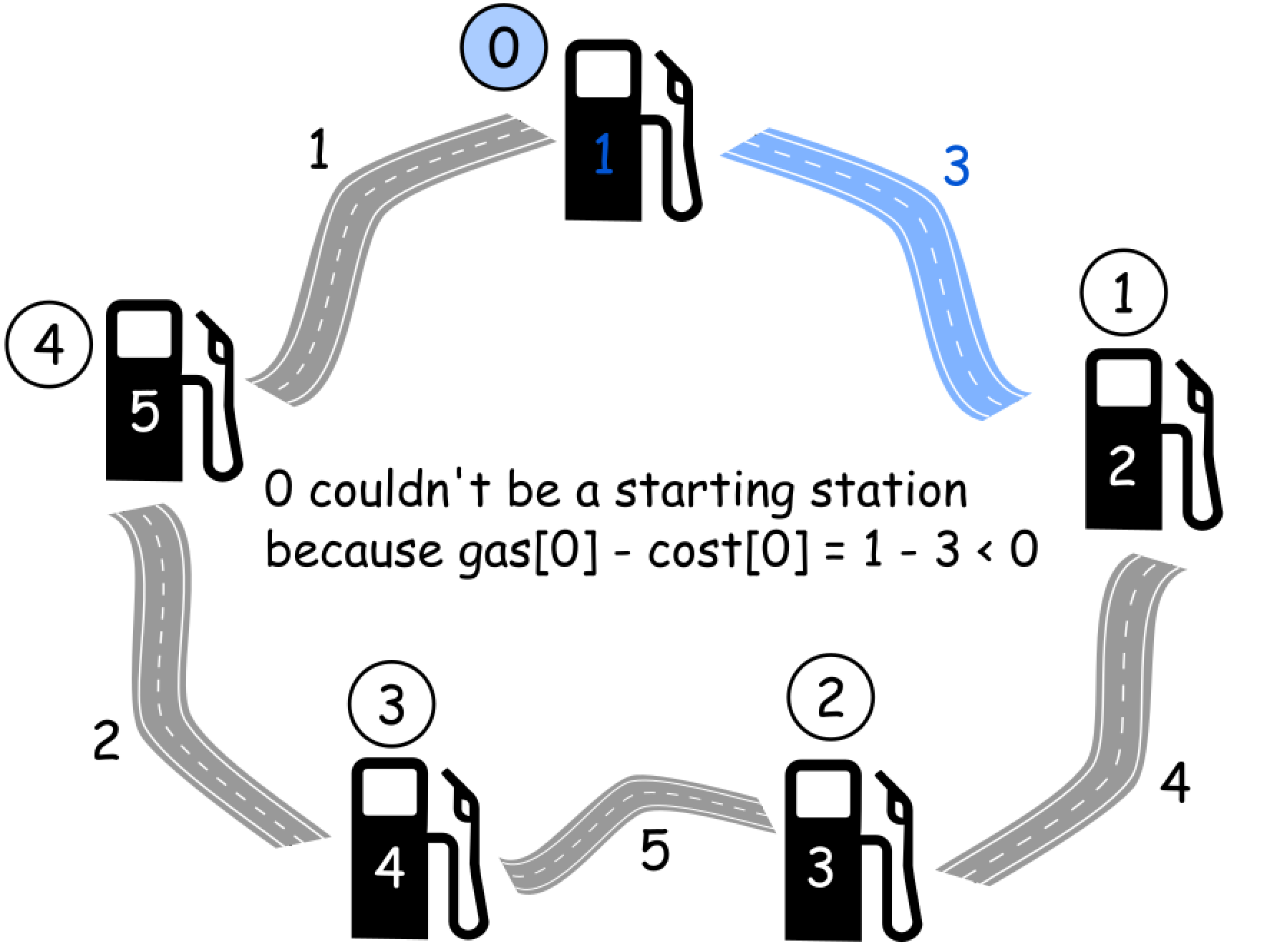
Let's notice two things.

It's impossible to perform the road trip if sum(gas) < sum(cost). In this situation the answer is -1.



One could compute total amount of gas in the tank total\_tank = sum(gas) - sum(cost) during the round trip, and then return -1 if total\_tank < 0.

It's impossible to start at a station i if gas[i] - cost[i] < 0, because then there is not enough gas in the tank to travel to i + 1 station.



The second fact could be generalized. Let's introduce curr\_tank variable to track the current amount of gas in the tank. If at some station curr\_tank is less than 0, that means that one couldn't reach this station.

Next step is to mark this station as a new starting point, and reset curr\_tank to zero since one starts with no gas in the tank.

**Algorithm**

Now the algorithm is straightforward :

1. Initiate total\_tank and curr\_tank as zero, and choose station 0 as a starting station.
2. Iterate over all stations :
   * Update total\_tank and curr\_tank at each step, by adding gas[i] and subtracting cost[i].
   * If curr\_tank < 0 at i + 1 station, make i + 1 station a new starting point and reset curr\_tank = 0 to start with an empty tank.
3. Return -1 if total\_tank < 0 and starting station otherwise.

**Why this works**

Let's imagine the situation when total\_tank >= 0 and the above algorithm returns N\_s*Ns*​ as a starting station.

Algorithm directly ensures that it's possible to go from N\_s*Ns*​ to the station 00. But what about the last part of the round trip from the station 00 to the station N\_s*Ns*​ ?

How one could ensure that it's possible to loop around to N\_s*Ns*​ ?

Let's use here the [proof by contradiction](https://en.wikipedia.org/wiki/Proof_by_contradiction) and assume that there is a station 0 < k < N\_s0<*k*<*Ns*​ such that one couldn't reach this station starting from N\_s*Ns*​.

The condition total\_tank >= 0 could be written as

\sum\_{i = 0}^{i = N}{\alpha\_i} \ge 0 \qquad (1)∑*i*=0*i*=*N*​*αi*​≥0(1) where \alpha\_i = \textrm{gas[i]} - \textrm{cost[i]}*αi*​=gas[i]−cost[i].

Let's split the sum on the right side by the starting station N\_s*Ns*​ and unreachable station k :

\sum\_{i = 0}^{i = k}{\alpha\_i} + \sum\_{i = k + 1}^{i = N\_s - 1}{\alpha\_i} + \sum\_{i = N\_s}^{i = N}{\alpha\_i} \ge 0 \qquad (2)∑*i*=0*i*=*k*​*αi*​+∑*i*=*k*+1*i*=*Ns*​−1​*αi*​+∑*i*=*Ns*​*i*=*N*​*αi*​≥0(2)

The second term is negative by the algorithm definition - otherwise the starting station would be before N\_s*Ns*​. It could be equal to zero only in the case of k = N\_s - 1*k*=*Ns*​−1.

\sum\_{i = k + 1}^{i = N\_s - 1}{\alpha\_i} \le 0 \qquad (3)∑*i*=*k*+1*i*=*Ns*​−1​*αi*​≤0(3)

Equations (2) and (3) together results in

\sum\_{i = 0}^{i = k}{\alpha\_i} + \sum\_{i = N\_s}^{i = N}{\alpha\_i} \ge 0 \qquad (4)∑*i*=0*i*=*k*​*αi*​+∑*i*=*Ns*​*i*=*N*​*αi*​≥0(4)

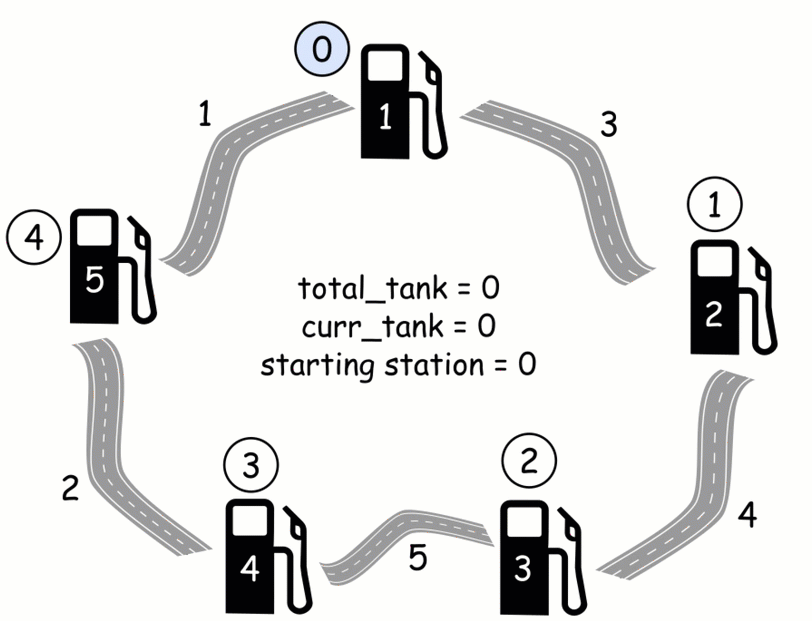
At the same time the station k*k* is supposed to be unreachable from N\_s*Ns*​ that means

\sum\_{i = N\_s}^{i = N}{\alpha\_i} + \sum\_{i = 0}^{i = k}{\alpha\_i} < 0 \qquad (5)∑*i*=*Ns*​*i*=*N*​*αi*​+∑*i*=0*i*=*k*​*αi*​<0(5)

Eqs. (4) and (5) together result in a contradiction. Therefore, the initial assumption — that there is a station 0 < k < N\_s0<*k*<*Ns*​ such that one couldn't reach this station starting from N\_s*Ns*​ — must be false.

Hence, one could do a round trip starting from N\_s*Ns*​, that makes N\_s*Ns*​ to be an answer. The answer is unique according to the problem definition.

**Implementation**



|  |
| --- |
| class Solution {  public int canCompleteCircuit(int[] gas, int[] cost) {  int n = gas.length;  int total\_tank = 0;  int curr\_tank = 0;  int starting\_station = 0;  for (int i = 0; i < n; ++i) {  total\_tank += gas[i] - cost[i];  curr\_tank += gas[i] - cost[i];  // If one couldn't get here,  if (curr\_tank < 0) {  // Pick up the next station as the starting one.  starting\_station = i + 1;  // Start with an empty tank.  curr\_tank = 0;  }  }  return total\_tank >= 0 ? starting\_station : -1;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) since there is only one loop over all stations here.
* Space complexity : \mathcal{O}(1)O(1) since it's a constant space solution.

**Further reading**

There are numerous variations of gas problem, here are some examples :

[Find the cheapest path between two stations if at most Δ stops are allowed.](https://www.sciencedirect.com/science/article/pii/S002001901730203X)

[Find the cheapest path between two stations if the vehicle has a given tank capacity.](https://link.springer.com/chapter/10.1007/978-3-540-75520-3_48)

Found a great explanation for this problem here <https://www.youtube.com/watch?v=nTKdYm_5-ZY&list=PLupD_xFct8mETlGFlLVrwbLwcxczbgWRM&index=8&t=0s> If any one didn't totally get the above solution

**Find the Difference**

You are given two strings s and t.

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

Return the letter that was added to t.

**Example 1:**

**Input:** s = "abcd", t = "abcde"

**Output:** "e"

**Explanation:** 'e' is the letter that was added.

**Example 2:**

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

**Output:** "y"

**Example 3:**

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

**Output:** "a"

**Example 4:**

**Input:** s = "ae", t = "aea"

**Output:** "a"

**Constraints:**

* 0 <= s.length <= 1000
* t.length == s.length + 1
* s and t consist of lower-case English letters.

## Solution

Let's reiterate the problem in our head. String t is nothing but shuffled string s with one extra character. This means if length of string s is N length of string t would be N + 1.

i.e. String t = **shuffled**(String s + Any character).

The shuffling is what stops us from doing a character by character comparison across the two strings.

This problem, even though pretty simple can have multiple ways of attacking it. That is what makes this problem an interesting one too. Let's look at some of the approaches and also try to understand how the complexity of different solution varies with just simple tricks applied.

#### Approach 1: Sorting

**Intuition**

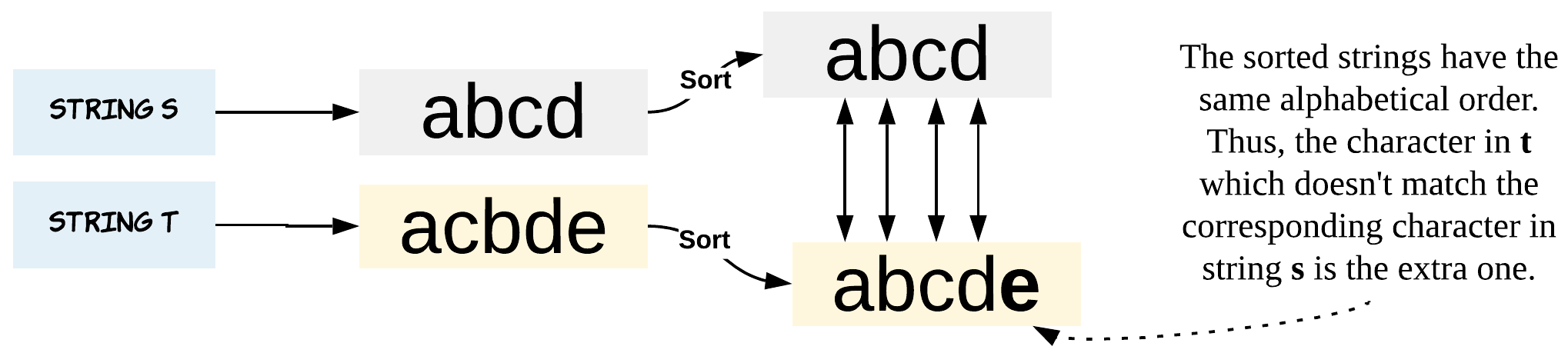
The obvious choice is sorting. Why obvious?

It's obvious because the first thing we might think of is, what if string t was not shuffled. If string t was not shuffled this problem would have been so easy.

And then next we might end up bringing the order between the two strings. What better than sorting both the strings.

i.e. **sort**(String t) = **sort**(shuffled(String s + Any character)).

That said, this could be one of the most brute ways of solving this problem. (There are other brute ways too. The intent is not to challenge your brute instincts :P)



Have you played Spot the Difference games, where you match an orange to orange and rule out the possibility? That's exactly what we are doing after sorting the strings.

**Algorithm**

1. Sort the string s and string t.
2. Iterate through the length of strings and do a character by character comparison. This just checks if the current character in string t is present in string s.
3. Once we encounter a character which is in string t but not in string s, we have found the extra character string t was hiding all this while.

|  |
| --- |
| class Solution {  public char findTheDifference(String s, String t) {  // Sort both the strings  char[] sortedS = s.toCharArray();  char[] sortedT = t.toCharArray();  Arrays.sort(sortedS);  Arrays.sort(sortedT);  // Character by character comparison  int i = 0;  while (i < s.length()) {  if (sortedS[i] != sortedT[i]) {  return sortedT[i];  }  i += 1;  }  return sortedT[i];  }  } |

**Complexity Analysis**

* Time Complexity: O(Nlog(N))*O*(*Nlog*(*N*)), where N*N* is length of the strings. Sorting is the most expensive operation of this algorithm. Sorting would take O(Nlog(N))*O*(*Nlog*(*N*)) time. Iterating both the strings for character by character comparison would take another O(N)*O*(*N*) time.
* Space Complexity: O(N)*O*(*N*). The sorted character arrays would take O(N)*O*(*N*) each. An important thing to note here is that we are converting the String in java to an array first and then sorting it. That's what takes the additional space. In Python, we can just sort the given input inplace by using the sort method. If you can get around the conversion to a temporary array in Java as well, then we will have an O(1)*O*(1) solution here.

#### Approach 2: Using HashMap

This approach is also not very tricky. What is important is to analyze its complexity.

We might just think in worst case the string is of length N and each character has a frequency of 1. This would result in a hash map of O(N)*O*(*N*) space. This is when your attention to detail comes to test.

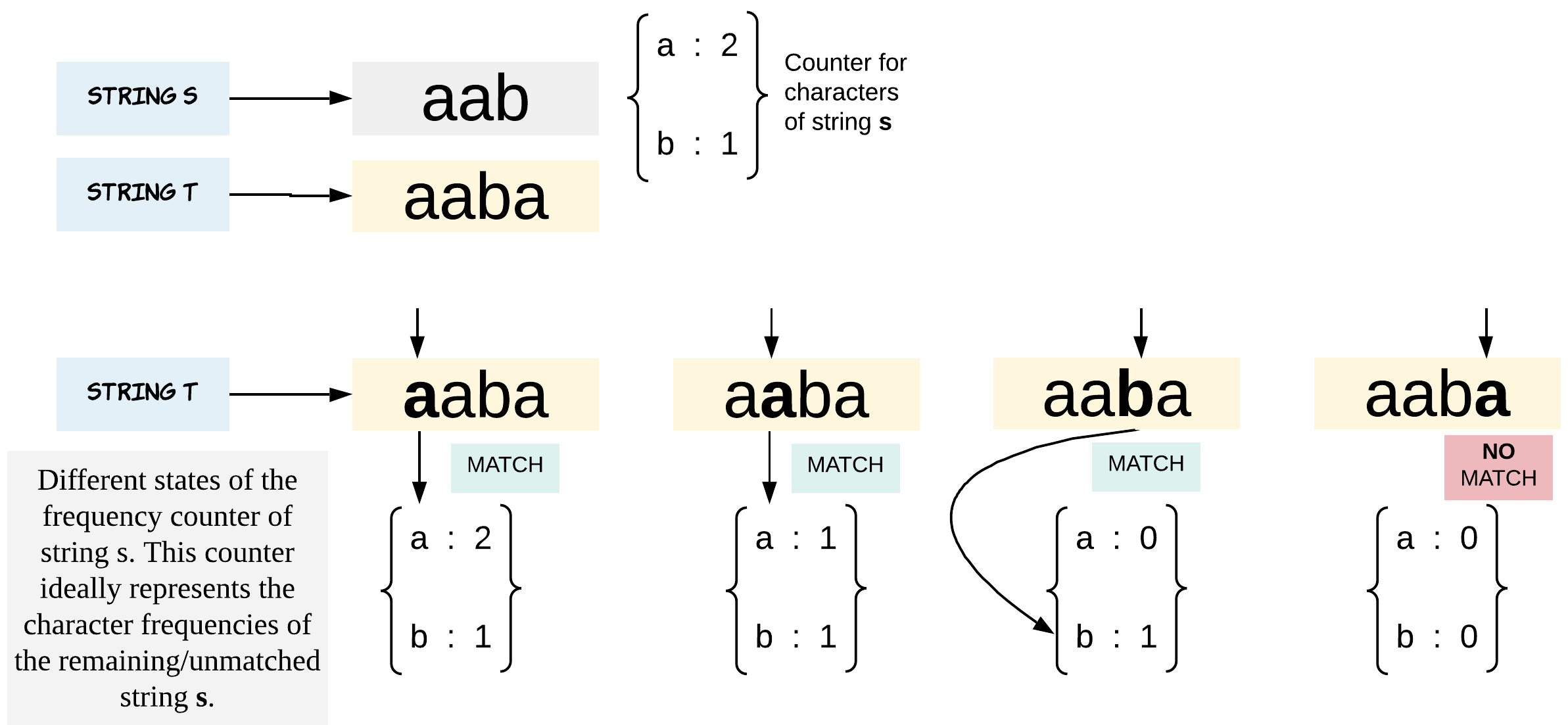
The problem states, string s and t consists of only lowercase letters.

The above statement implies we only have 26 characters i.e. [a, z]. Thus, we have a space complexity for just 26 characters.

It's always good to clarify this with the interviewer as now the space complexity would just be constant. Thus, this approach can also be implemented using array of length 26 as a hash table, where each index corresponds to a letter from [a, z].

**Algorithm**

1. Store all the characters of string s in a hash map called counterS. The key would be the character and value would be number of times the character appeared in the string.
2. Now, iterate through string t and for each character, check if it is present in the hash map counterS.
3. If the character is present in counterS then we just decrement the corresponding value by 1.
4. If the character is not present in counterS or has a frequency of zero in counterS it means we have found the extra character of string t.



Note - We are dropping the frequency of a character by 1 every time there is a match. This helps us find out the extra character which is present in both s and t but the number of occurrences vary. Thus keeping frequency is equally important.

|  |
| --- |
| class Solution {  public char findTheDifference(String s, String t) {  char extraChar = '\0';  // Prepare a counter for string s.  // This hash map holds the characters as keys and respective frequency as value.  HashMap <Character,Integer> counterS = new HashMap <>();  for (int i = 0; i < s.length(); i++) {  char ch = s.charAt(i);  counterS.put(ch, counterS.getOrDefault(ch, 0) + 1);  }  // Iterate through string t and find the character which is not in s.  for (int i = 0; i < t.length(); i += 1) {  char ch = t.charAt(i);  int countOfChar = counterS.getOrDefault(ch, 0);  if (countOfChar == 0) {  extraChar = ch;  break;  } else {  // Once a match is found we reduce frequency left.  // This eliminates the possibility of a false match later.  counterS.put(ch, countOfChar - 1);  }  }  return extraChar;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is length of the strings. Since, we iterate through both the strings once.
* Space Complexity: O(1)*O*(1). The problem states string s and string t have lowercase letters. Thus, the total number of unique characters and eventually buckets in the hash map possible are just 26.

#### Approach 3: Bit Manipulation

Don't be scared. This approach is as simple as scary it might sound.

The trick is simple. To use bitwise XOR operation on all the elements. XOR would help to eliminate the alike and only leave the odd duckling.

To understand how this works, let's brush up our XOR concepts first.

0 ^ 0 = 0

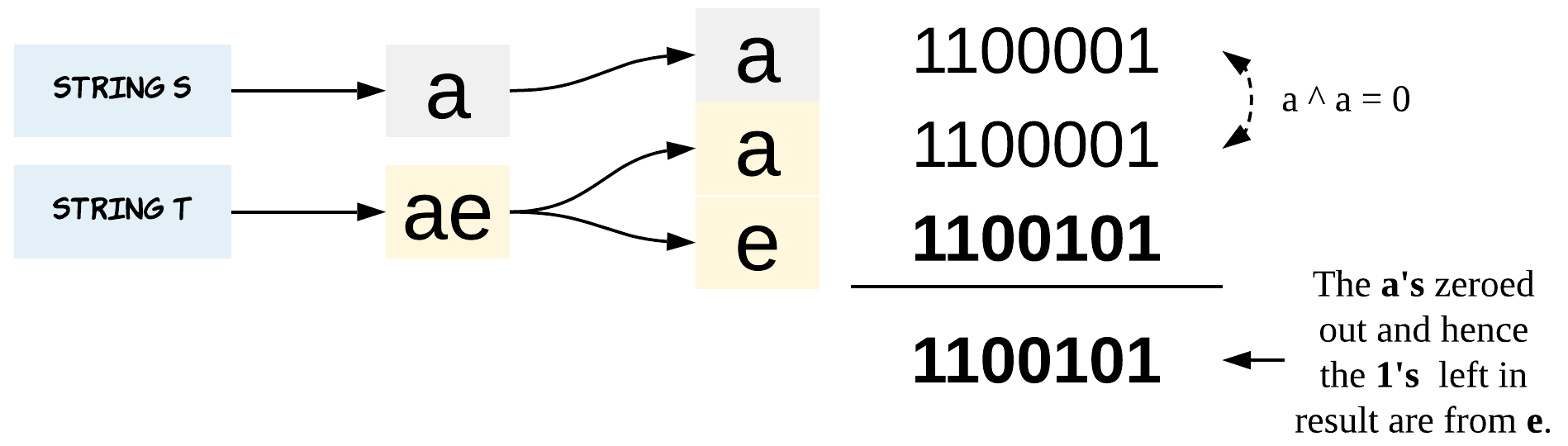
0 ^ 1 = 1

1 ^ 0 = 1

1 ^ 1 = 0

Look at how the similar ones just even out. This is what we would use to our advantage. When all the other similar pairs just even out or reduce to a zero, the different one would remain.

Thus, the left over bits after XORing all the characters from string s and string t would be from the extra character of string t.



XOR matches apples to apples and oranges to oranges and returns 0 when match happens. What is left, is the difference.

**Algorithm**

1. Initialize a variable ch which would hold the XORed results.
2. XOR all the characters with ch while iterating through string s.
3. XOR all the characters with ch while iterating through string t. (Alternatively, we could have also combined steps 2 and 3).
4. Return ch as the answer.

|  |
| --- |
| class Solution {  public char findTheDifference(String s, String t) {  // Initialize ch with 0, because 0 ^ X = X  // 0 when XORed with any bit would not change the bits value.  char ch = 0;  // XOR all the characters of both s and t.  for (int i = 0; i < s.length(); i += 1) {  ch ^= s.charAt(i);  }  for (int i = 0; i < t.length(); i += 1) {  ch ^= t.charAt(i);  }  // What is left after XORing everything is the difference.  return ch;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is length of the strings. Since, we iterate through both the strings once.
* Space Complexity: O(1)*O*(1).

|  |
| --- |
| public char findTheDifference(String s, String t) {  int sSum = 0;  int tSum = t.charAt(t.length() - 1);  for(int i = 0; i < s.length() ; i++){  sSum += s.charAt(i);  tSum += t.charAt(i);  }    return (char)(tSum - sSum);  } |

**Teemo Attacking**

In LOL world, there is a hero called Teemo and his attacking can make his enemy Ashe be in poisoned condition. Now, given the Teemo's attacking **ascending** time series towards Ashe and the poisoning time duration per Teemo's attacking, you need to output the total time that Ashe is in poisoned condition.

You may assume that Teemo attacks at the very beginning of a specific time point, and makes Ashe be in poisoned condition immediately.

**Example 1:**

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

**Output:** 4

**Explanation:** At time point 1, Teemo starts attacking Ashe and makes Ashe be poisoned immediately.

This poisoned status will last 2 seconds until the end of time point 2.

And at time point 4, Teemo attacks Ashe again, and causes Ashe to be in poisoned status for another 2 seconds.

So you finally need to output 4.

**Example 2:**

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

**Output:** 3

**Explanation:** At time point 1, Teemo starts attacking Ashe and makes Ashe be poisoned.

This poisoned status will last 2 seconds until the end of time point 2.

However, at the beginning of time point 2, Teemo attacks Ashe again who is already in poisoned status.

Since the poisoned status won't add up together, though the second poisoning attack will still work at time point 2, it will stop at the end of time point 3.

So you finally need to output 3.

**Note:**

1. You may assume the length of given time series array won't exceed 10000.
2. You may assume the numbers in the Teemo's attacking time series and his poisoning time duration per attacking are non-negative integers, which won't exceed 10,000,000.

## Solution

#### Approach 1: One pass

**Intuition**

The problem is an example of merge interval questions which are now [quite popular in Google](https://leetcode.com/discuss/interview-question/280433/Google-or-Phone-screen-or-Program-scheduling).

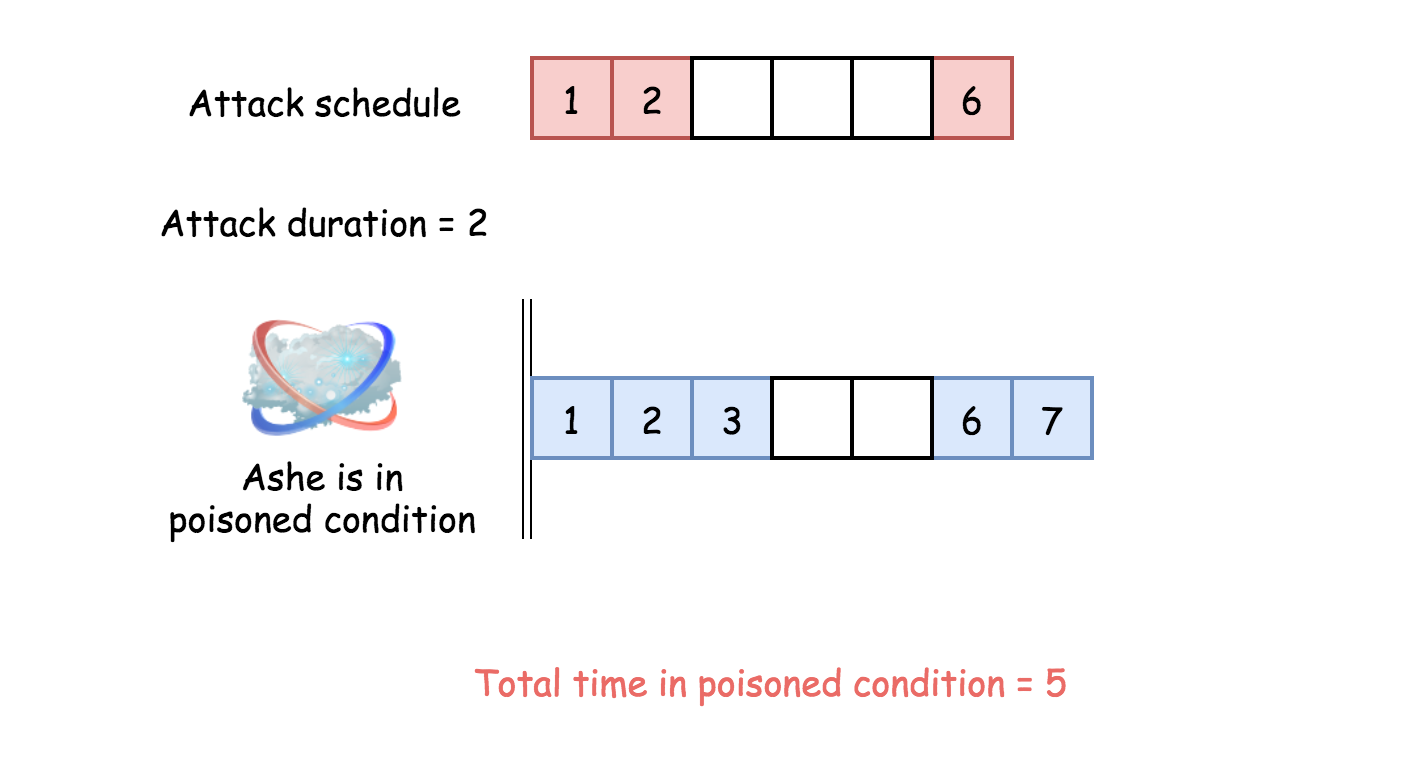
Typically such problems could be solved in a linear time in the case of sorted input, like [here](https://leetcode.com/articles/insert-interval/), and in \mathcal{O}(N \log N)O(*N*log*N*) time otherwise, [here is an example](https://leetcode.com/articles/merge-intervals/).

Here one deals with a sorted input, and the problem could be solved in one pass with a constant space. The idea is straightforward: consider only the interval between two attacks. Ashe spends in a poisoned condition the whole time interval if this interval is shorter than the poisoning time duration duration, and duration otherwise.

**Algorithm**

* Initiate total time in poisoned condition total = 0.
* Iterate over timeSeries list. At each step add to the total time the minimum between interval length and the poisoning time duration duration.
* Return total + duration to take the last attack into account.

**Implementation**



|  |
| --- |
| class Solution {  public int findPoisonedDuration(int[] timeSeries, int duration) {  int n = timeSeries.length;  if (n == 0) return 0;  int total = 0;  for(int i = 0; i < n - 1; ++i)  total += Math.min(timeSeries[i + 1] - timeSeries[i], duration);  return total + duration;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*), where N is the length of the input list, since we iterate the entire list.
* Space complexity : \mathcal{O}(1)O(1), it's a constant space solution.