**Adobe Interview**

Introduction

For an Adobe interview, make sure to review your CS fundamentals. You should also prepare for behavioral questions, which will be asked in the Director or HR round.

We organized this list so you can get well-prepared for an Adobe interview.

Arrays and Strings

 Two Sum

 Longest Substring Without Repeating Characters

 Container With Most Water

 Integer to Roman

 Roman to Integer

 Longest Common Prefix

 3Sum

 3Sum Closest

 4Sum

**Substring with Concatenation of All Words**

 Spiral Matrix

 Product of Array Except Self

 Missing Number

**Find All Numbers Disappeared in an Array**

**Positions of Large Groups**

 Unique Email Addresses

Linked Lists

 Add Two Numbers

 Remove Nth Node From End of List

 Merge Two Sorted Lists

 Copy List with Random Pointer

 Linked List Cycle

 Reverse Linked List

Trees and Graphs

 Validate Binary Search Tree

**Sum of Left Leaves**

 Binary Tree Inorder Traversal

 Binary Tree Zigzag Level Order Traversal

Heaps, Queues, Stacks

 Merge k Sorted Lists

**Simplify Path**

**Basic Calculator - Hard**

**Remove K Digits**

Sorting and Searching

 Median of Two Sorted Arrays

**Search Insert Position**

 Merge Intervals

Dynamic Programming

 Longest Palindromic Substring

 Maximum Subarray

 Best Time to Buy and Sell Stock

 Climbing Stairs

**Maximal Rectangle**

 Maximum Product Subarray

 Regular Expression Matching

 Longest Increasing Subsequence

 Perfect Squares – Q&S

Design

 Min Stack

 LRU Cache

Others

 Reverse Integer

**Tenth Line**

**Add Digits**

**Nth Digit**

**Encode and Decode TinyURL**

 Jewels and Stones

**Rectangle Overlap**

SQL

 Combine Two Tables

 Nth Highest Salary

**Department Top Three Salaries**

**Big Countries**

**Substring with Concatenation of All Words**

You are given a string s and an array of strings words of **the same length**. Return all starting indices of substring(s) in s that is a concatenation of each word in words **exactly once**, **in any order**, and **without any intervening characters**.

You can return the answer in **any order**.

**Example 1:**

**Input:** s = "barfoothefoobarman", words = ["foo","bar"]

**Output:** [0,9]

**Explanation:** Substrings starting at index 0 and 9 are "barfoo" and "foobar" respectively.

The output order does not matter, returning [9,0] is fine too.

**Example 2:**

**Input:** s = "wordgoodgoodgoodbestword", words = ["word","good","best","word"]

**Output:** []

**Example 3:**

**Input:** s = "barfoofoobarthefoobarman", words = ["bar","foo","the"]

**Output:** [6,9,12]

**Constraints:**

* 1 <= s.length <= 104
* s consists of lower-case English letters.
* 1 <= words.length <= 5000
* 1 <= words[i].length <= 30
* words[i] consists of lower-case English letters.

**Find All Numbers Disappeared in an Array**

Given an array of integers where 1 ≤ a[i] ≤ *n* (*n* = size of array), some elements appear twice and others appear once.

Find all the elements of [1, *n*] inclusive that do not appear in this array.

Could you do it without extra space and in O(*n*) runtime? You may assume the returned list does not count as extra space.

**Example:**

**Input:**

[4,3,2,7,8,2,3,1]

**Output:**

[5,6]

   Hide Hint #1

This is a really easy problem if you decide to use additional memory. For those trying to write an initial solution using additional memory, think **counters!**

   Hide Hint #2

However, the trick really is to not use any additional space than what is already available to use. Sometimes, multiple passes over the input array help find the solution. However, there's an interesting piece of information in this problem that makes it easy to re-use the input array itself for the solution.

   Hide Hint #3

The problem specifies that the numbers in the array will be in the range [1, n] where n is the number of elements in the array. Can we use this information and modify the array in-place somehow to find what we need?

## Solution

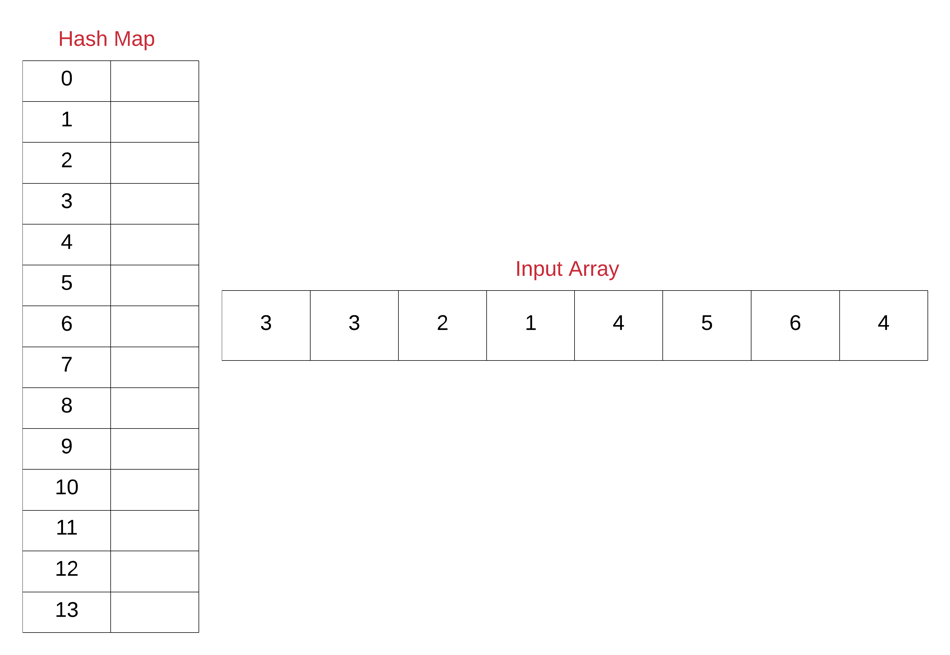
#### **Approach 1: Using Hash Map**

**Intuition**

The intuition behind using a hash map is pretty clear in this case. We are given that the array would be of size N and it should contain numbers from 1 to N. However, some of the numbers are missing. All we have to do is keep track of which numbers we encounter in the array and then iterate from 1 \cdots N1⋯*N* and check which numbers did not appear in the hash table. Those will be our missing numbers. Let's look at a formal algorithm based on this idea and then an animation explaining the same with the help of a simple example.

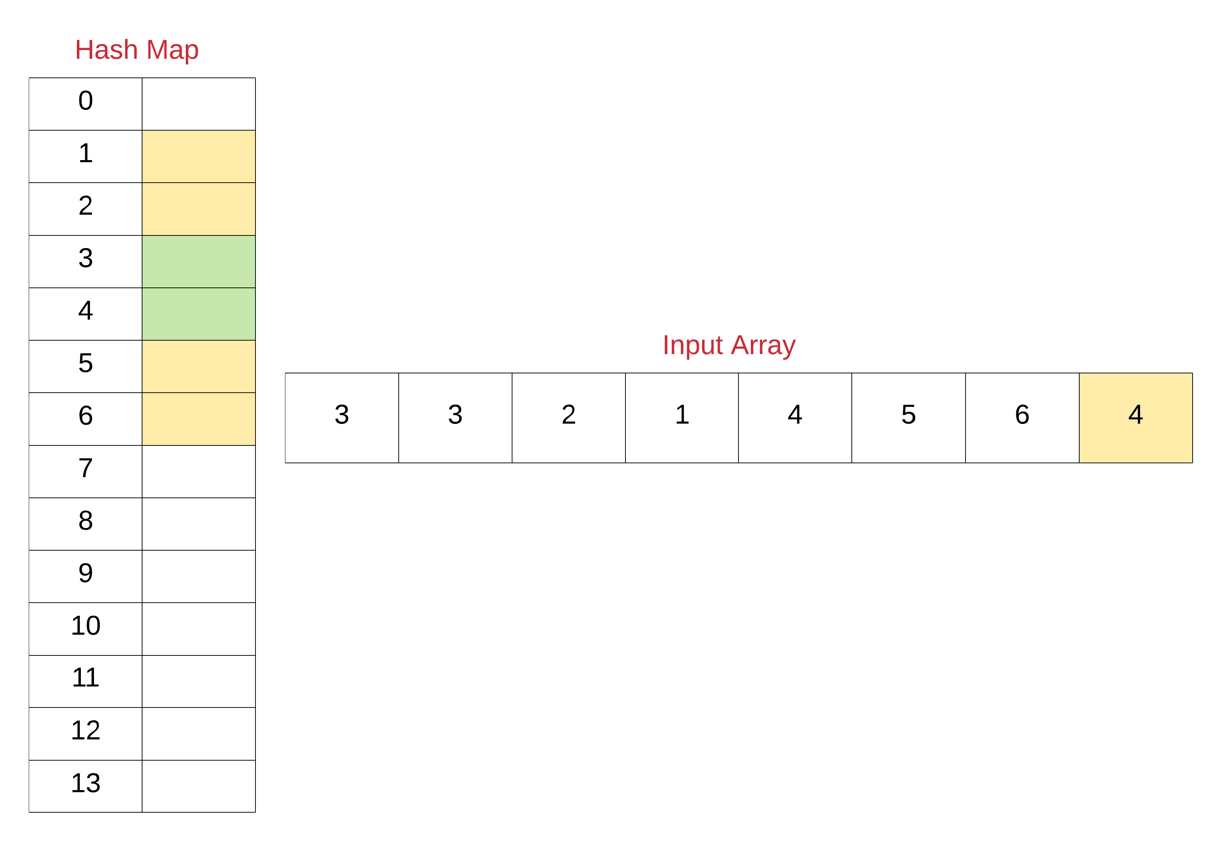
**Algorithm**

1. Initialize a hash map, hash to keep track of the numbers that we encounter in the array. Note that we can use a set data structure as well in this case since we are not concerned about the frequency counts of elements.

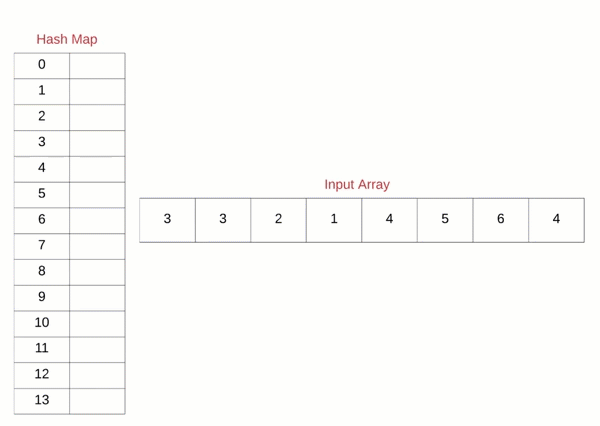


Note that for the purposes of illustration, we have use a hash map of size 14 and have ordered the keys of the hash map from 0 to 14. Also, we will be using a simple hash function that directly maps the array entries to their corresponding keys in the hash map. Usually, the mapping is not this simple and is dependent upon the hash function being used in the implementation of the hash map.

1. Next, iterate over the given array one element at a time and for each element, insert an entry in the hash map. Even if an entry were to exist before in the hash map, it will simply be over-written. For the above example, let's look at the final state of the hash map once we process the last element of the array.



1. Now that we know the unique set of elements from the array, we can simply find out the missing elements from the range 1 \cdots N1⋯*N*.
2. Iterate over all the numbers from 1 \cdots N1⋯*N* and for each number, check if there's an entry in the hash map. If there is no entry, add that missing number to a result array that we will return from the function eventually.



|  |
| --- |
| class Solution {  public List<Integer> findDisappearedNumbers(int[] nums) {    // Hash table for keeping track of the numbers in the array  // Note that we can also use a set here since we are not  // really concerned with the frequency of numbers.  HashMap<Integer, Boolean> hashTable = new HashMap<Integer, Boolean>();    // Add each of the numbers to the hash table  for (int i = 0; i < nums.length; i++) {  hashTable.put(nums[i], true);  }    // Response array that would contain the missing numbers  List<Integer> result = new LinkedList<Integer>();    // Iterate over the numbers from 1 to N and add all those  // that don't appear in the hash table.  for (int i = 1; i <= nums.length; i++) {  if (!hashTable.containsKey(i)) {  result.add(i);  }  }    return result;  }  } |

**Complexity Analysis**

* Time Complexity : O(N)*O*(*N*)
* Space Complexity : O(N)*O*(*N*)

#### **Approach 2: O(1) Space InPlace Modification Solution**

**Intuition**

We definitely need to keep track of all the unique numbers that appear in the array. However, we don't want to use any extra space for it. This solution that we will look at in just a moment springs from the fact that

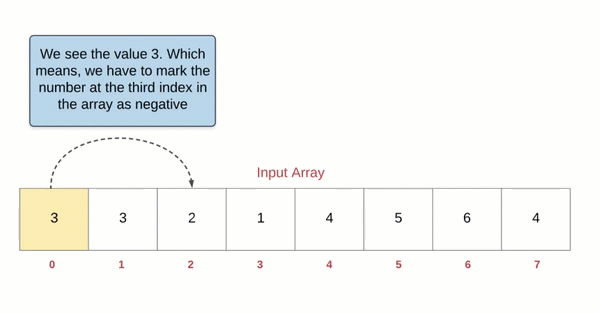
All the elements are in the range [1, N]

Since we are given this information, we can make use of the input array itself to somehow mark visited numbers and then find our missing numbers. Now, we don't want to change the actual data in the array but who's stopping us from changing the magnitude of numbers in the array? That is the basic idea behind this algorithm.

We will be negating the numbers seen in the array and use the sign of each of the numbers for finding our missing numbers. We will be treating numbers in the array as indices and mark corresponding locations in the array as negative.

**Algorithm**

1. Iterate over the input array one element at a time.
2. For each element nums[i], mark the element at the corresponding location negative if it's not already marked so i.e. nums[\; nums[i] \;- 1\;] \times -1*nums*[*nums*[*i*]−1]×−1 .
3. Now, loop over numbers from 1 \cdots N1⋯*N* and for each number check if nums[j] is negative. If it is negative, that means we've seen this number somewhere in the array.
4. Add all the numbers to the resultant array which don't have their corresponding locations marked as negative in the original array.



|  |
| --- |
| class Solution {  public List<Integer> findDisappearedNumbers(int[] nums) {    // Iterate over each of the elements in the original array  for (int i = 0; i < nums.length; i++) {    // Treat the value as the new index  int newIndex = Math.abs(nums[i]) - 1;    // Check the magnitude of value at this new index  // If the magnitude is positive, make it negative  // thus indicating that the number nums[i] has  // appeared or has been visited.  if (nums[newIndex] > 0) {  nums[newIndex] \*= -1;  }  }    // Response array that would contain the missing numbers  List<Integer> result = new LinkedList<Integer>();    // Iterate over the numbers from 1 to N and add all those  // that have positive magnitude in the array  for (int i = 1; i <= nums.length; i++) {    if (nums[i - 1] > 0) {  result.add(i);  }  }    return result;  }  } |

**Complexity Analysis**

* Time Complexity : O(N)*O*(*N*)
* Space Complexity : O(1)*O*(1) since we are reusing the input array itself as a hash table and the space occupied by the output array doesn't count toward the space complexity of the algorithm.

**Positions of Large Groups**

In a string s of lowercase letters, these letters form consecutive groups of the same character.

For example, a string like s = "abbxxxxzyy" has the groups "a", "bb", "xxxx", "z", and "yy".

A group is identified by an interval [start, end], where start and end denote the start and end indices (inclusive) of the group. In the above example, "xxxx" has the interval [3,6].

A group is considered **large** if it has 3 or more characters.

Return *the intervals of every****large****group sorted in****increasing order by start index***.

**Example 1:**

**Input:** s = "abbxxxxzzy"

**Output:** [[3,6]]

**Explanation**: "xxxx" is the only large group with start index 3 and end index 6.

**Example 2:**

**Input:** s = "abc"

**Output:** []

**Explanation**: We have groups "a", "b", and "c", none of which are large groups.

**Example 3:**

**Input:** s = "abcdddeeeeaabbbcd"

**Output:** [[3,5],[6,9],[12,14]]

**Explanation**: The large groups are "ddd", "eeee", and "bbb".

**Example 4:**

**Input:** s = "aba"

**Output:** []

**Constraints:**

* 1 <= s.length <= 1000
* s contains lower-case English letters only.

#### **Approach #1: Two Pointer [Accepted]**

**Intuition**

We scan through the string to identify the start and end of each group. If the size of the group is at least 3, we add it to the answer.

**Algorithm**

Maintain pointers i, j with i <= j. The i pointer will represent the start of the current group, and we will increment j forward until it reaches the end of the group.

We know that we have reached the end of the group when j is at the end of the string, or S[j] != S[j+1]. At this point, we have some group [i, j]; and after, we will update i = j+1, the start of the next group.

|  |
| --- |
| class Solution {  public List<List<Integer>> largeGroupPositions(String S) {  List<List<Integer>> ans = new ArrayList();  int i = 0, N = S.length(); // i is the start of each group  for (int j = 0; j < N; ++j) {  if (j == N-1 || S.charAt(j) != S.charAt(j+1)) {  // Here, [i, j] represents a group.  if (j-i+1 >= 3)  ans.add(Arrays.asList(new Integer[]{i, j}));  i = j + 1;  }  }  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of S.
* Space Complexity: O(N)*O*(*N*), the space used by the answer.

**Sum of Left Leaves**

Find the sum of all left leaves in a given binary tree.

**Example:**

3

/ \

9 20

/ \

15 7

There are two left leaves in the binary tree, with values **9** and **15** respectively. Return **24**.

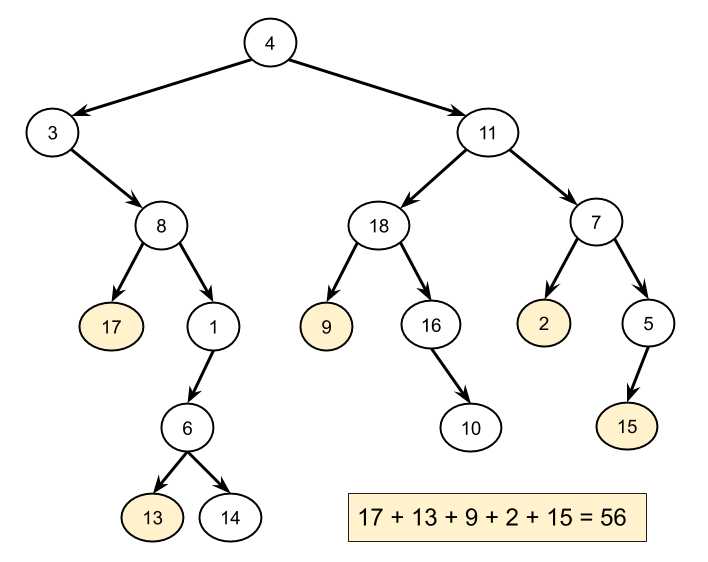
## Solution

Recall that a binary tree is made up of linked tree nodes, where each node has a reference to its left child and to its right child. We access the child nodes by using root.left and root.right. Tree traversal algorithms are used to explore all nodes in the tree. If you're completely confused now, we recommend checking out our [Binary Tree Explore Card](https://leetcode.com/explore/learn/card/data-structure-tree/) and then coming back to this question.

#### **Approach 1: Iterative Tree Traversal (Pre-order)**

**Intuition**

Here is an example of a binary tree. The left-leaf nodes are highlighted. Our task is to find all of these left-leaf nodes, add together their values, and return the final sum we get.



To get each left-leaf node's value, we need to "visit" each one. Note that because it is a sum we need, the order we visit the left-leaf nodes in doesn't matter. As such, we can pick any algorithm that visits all nodes of the tree. The simplest such algorithm is an **iterative pre-order traversal** (if you're not sure what this is, check out the [Binary Tree Explore Card](https://leetcode.com/explore/learn/card/data-structure-tree/134/traverse-a-tree/992/#pre-order-traversal)).

When we visit each node, we'll need to know whether or not it is a left-leaf node: this is the main challenge in this problem. Remember that once we're on a node, there is no link back up to its parent. This means that given a node, it is impossible to check whether or not it is a left node unless we have an existing reference to its parent. There are a couple of strategies for handling this problem:

1. While we're at a node, we can check if its left-child is a leaf node (instead of trying to check if the node itself is a left child).
2. When we're ready to visit the children of a node, we can pass some extra information down telling the left child that it is a left child.

The second strategy works well for the recursive implementation (Approach 2), but the first strategy is the best for the iterative, so is what we'll go with here.

Anyway, to do an iterative pre-order traversal, we start by putting the root onto a stack. Then, while the stack is non-empty, we take a node off the top, check if the node's left child is a leaf node and then add that child's value to a total if it is. Finally, we put the node's left child and right child onto the stack so that they can be visited too. Here is the algorithm in pseudocode.

define function sum\_of\_left\_leaves(root):

stack = a new stack

push root to stack

total = 0

while the stack is non-empty:

node = pop a node of stack

if node.left exists and node.left is a leaf:

total = total + node.left.value

if node.right exists:

push node.right to stack

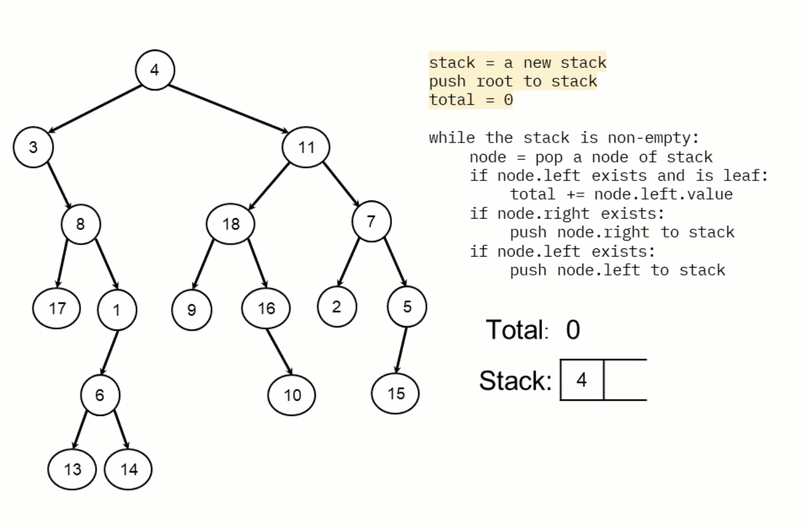
if node.left exists:

push node.left to stack

return total

Note that it doesn't matter whether we put the left or right child on the stack first. We just chose to put right and then left so that left is the next off, thus matching the standard pre-order traversal code template.

Here is an animation of this approach in action!



A pre-order traversal is a type of depth-first tree traversal. This is because it uses a stack to keep track of unvisited nodes. Alternatively, we could have used a breadth-first tree traversal; using a queue to keep track of unvisited nodes instead of a stack. This works because while the nodes are visited in a different order, this doesn't matter, as discussed above. In fact, we could have used any data structure that allowed us to put an unvisited node in and take a node out to visit.

**Algorithm**

This code uses a stack to keep track of the unvisited nodes. You could, however, replace it with a queue and the algorithm would still work (but would process the nodes in a different order). In Java, we use a Deque like a stack, instead of using Stack (Stack is deprecated).

|  |
| --- |
| class Solution {    private boolean isLeaf(TreeNode node) {  return node != null && node.left == null && node.right == null;  }  public int sumOfLeftLeaves(TreeNode root) {  if (root == null) {  return 0;  }  int total = 0;  Deque<TreeNode> stack = new ArrayDeque<>();  stack.push(root);  while (!stack.isEmpty()) {  TreeNode subRoot = stack.pop();  // Check if the left node is a leaf node.  if (isLeaf(subRoot.left)) {  total += subRoot.left.val;  }  // If the right node exists, put it on the stack.  if (subRoot.right != null) {  stack.push(subRoot.right);  }  // If the left node exists, put it on the stack.  if (subRoot.left != null) {  stack.push(subRoot.left);  }  }  return total;  }  } |

**Complexity Analysis**

Let N*N* be the number of nodes in the tree.

* Time complexity : O(N)*O*(*N*).

Each node is put onto the stack once, by its parent node. We know each node only has one parent because this is a tree. Therefore, each node is only taken off, and processed, once. Processing a node is an O(1)*O*(1) operation. Therefore, we get a total time complexity of N \cdot O(1) = O(N)*N*⋅*O*(1)=*O*(*N*).

* Space complexity : O(N)*O*(*N*).

Remember that in complexity analysis, we're always looking at the worst case. The worst-case tree is one where we have a long "strand" of left nodes, with each having a single right node. On one of these trees, the algorithm will work its way down the left nodes first, having at most one of them on the stack at a time. However, every right node that it encounters will be placed on the stack. With half of the nodes being these right nodes, the space used is proportional to the number of nodes in the tree, giving us a space complexity of O(N)*O*(*N*).

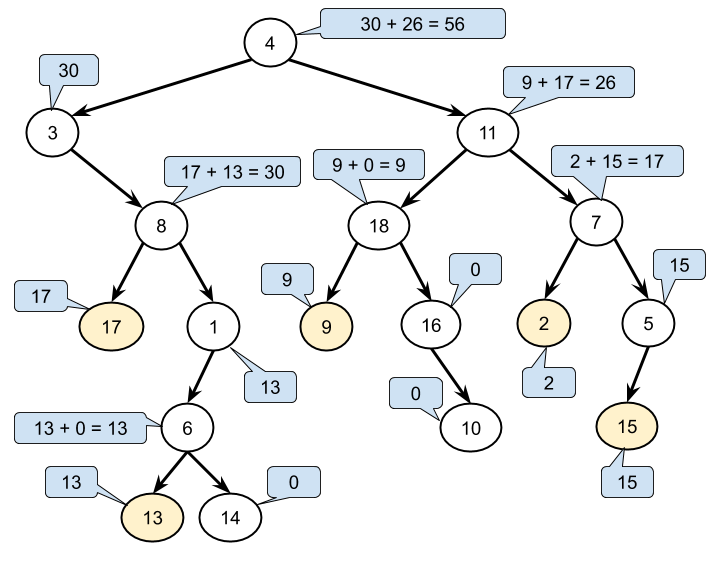
For the problem we've been given here on LeetCode, we haven't been told whether or not the input tree is balanced (most non-leaf nodes having 2 children, thus minimizing the maximum depth). Therefore, we have to assume it is not. **However**, in an interview, you might be asked what the time and space complexity are if the input was guaranteed to be a balanced tree. If it is, balanced, then the time complexity remains the same (we still have to visit all N*N* nodes), but the space complexity becomes O(D)*O*(*D*), where D*D* is the maximum depth. This is equivalent to O(\log \, N)*O*(log*N*).

#### **Approach 2: Recursive Tree Traversal (Pre-order)**

**Intuition**

Another way of traversing a tree is to use recursion to visit each node. If you're not familiar with recursion on trees, check out the [Binary Trees Explore Card](https://leetcode.com/explore/learn/card/data-structure-tree/17/solve-problems-recursively/534/).

Recall that recursive tree algorithms work by treating each node of the tree as the root of a subtree. The answer (i.e. the sum of left leaf node values) is then found for each subtree by finding the answers for the left and right subtrees and combining (adding) them together.



To implement a recursive function, we need to identify the base cases and recursive cases.

The **base case** is that this subtree is a leaf node (i.e. the subtree only contains a single node; its root node). The value we return for it depends on whether this subtree was to the left or the right of its parent. If it was to the left, we return its value. If it was to the right, we return zero.

The **recursive** case is that this subtree contains a left and/or right subtree (i.e. the subtree has more than just the root node in it). We call the recursive function on the left and right subtrees, add their results together and return the added result.

Like before though, we still have the problem of knowing whether or not the current subtree was to the left of its parent. With recursion though, there is a far more elegant solution than before: we can simply have an additional boolean parameter on our recursive function, specifying whether or not the subtree is a left one! Note that the top subtree is neither a left node, nor a right node, but we pass in false for it, as the purpose of the parameter is to specify whether or not it is a left subtree.

define function sum\_of\_left\_leaves(root):

return process\_subtree(root, false)

define function process\_subtree(subtree, is\_left):

if subtree is a leaf node:

if is\_left is true:

return subtree.value

else:

return 0

else:

total = 0

if subtree.left exists:

total = total + process\_subtree(subtree.left, true)

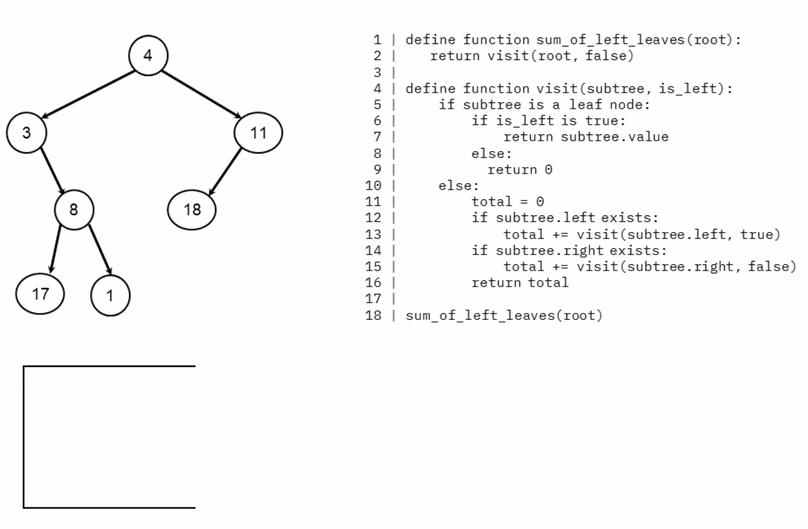
if subtree.right exists:

total = total + process\_subtree(subtree.right, false)

return total

Notice too that we don't need a global variable to keep track of the total; by returning the total for each subtree, the final total returned will be the total for the root node, which is the answer.

Here is an animation showing how it works.



Here is the algorithm, as we described it above.

|  |
| --- |
| class Solution {    public int sumOfLeftLeaves(TreeNode root) {  if (root == null) {  return 0;  }  return processSubtree(root, false);  }    private int processSubtree(TreeNode subtree, boolean isLeft) {    // Base case: This is a leaf node.  if (subtree.left == null && subtree.right == null) {  return isLeft ? subtree.val : 0;  }    // Recursive case: We need to add and return the results of the  // left and right subtrees.  int total = 0;  if (subtree.left != null) {  total += processSubtree(subtree.left, true);  }  if (subtree.right != null) {  total += processSubtree(subtree.right, false);  }  return total;  }  } |

We can simplify the code a bit by defining an additional base case: if the subtree is empty (null), then 0 should be returned. This means we no longer need to do null-checks in three separate places. This is a pattern you will see a lot for recursive tree algorithms.

|  |
| --- |
| class Solution {    public int sumOfLeftLeaves(TreeNode root) {  return processSubtree(root, false);  }    private int processSubtree(TreeNode subtree, boolean isLeft) {    // Base case: This is an empty subtree.  if (subtree == null) {  return 0;  }    // Base case: This is a leaf node.  if (subtree.left == null && subtree.right == null) {  return isLeft ? subtree.val : 0;  }    // Recursive case: We need to add and return the results of the  // left and right subtrees.  return processSubtree(subtree.left, true) + processSubtree(subtree.right, false);  }  } |

**Complexity Analysis**

Let N*N* be the number of nodes.

* Time complexity : O(N)*O*(*N*).

The code within the recursive function is all O(1)*O*(1). The function is called exactly once for each of the N*N* nodes. Therefore, the total time complexity of the algorithm is O(N)*O*(*N*).

* Space complexity : O(N)*O*(*N*).

In the worst case, the tree consists of nodes that form a single deep strand. In this case, the runtime-stack will have N*N* calls to processSubtree(...) on it at the same time, giving a worst-case space complexity of O(N)*O*(*N*).

#### **Approach 3: Morris Tree Traversal (Pre-order)**

**Intuition**

Note that this approach is quite advanced. Feel free to skip it if you're just starting out!

All of the algorithms we've looked at so far had a time complexity of O(N)*O*(*N*), and a space complexity of O(N)*O*(*N*).

We know it is impossible to reduce the time complexity any further, as we need to visit each node to access all the nodes.

The space complexity might initially seem impossible to reduce, as when a node has two children, we need to explore one immediately, and keep track of the other for exploration afterward (often, we explore the left subtree first, and keep track of the right subtree for later). Going down the tree, we can end up with many of these child nodes awaiting exploration. However, there is a tree traversal algorithm that requires O(N)*O*(*N*) time and only O(1)*O*(1) space: Morris Tree Traversal.

This traversal algorithm works by temporarily modifying the input tree so that before we explore a left subtree, we find the subtree root's in-order predecessor (which will never have a right child), and make its right link point back up to the root. Then we explore the left subtree. When we're done exploring the left subtree, the link back up to the root will then allow us to explore the right subtree. When we follow the link back up, we also remove it so that the input tree is restored. In this way, we can no longer need an auxiliary data structure to keep track of the right subtrees.

Given that this algorithm modifies the input tree, will we still be able to identify which nodes are left-leaves? As it turns out we still can. Whenever we reach a node for the first time, we know we haven't yet looked at its left subtree, and so have not modified it. Therefore, we can simply check if the left child is a leaf node, in the same way we did before.

For more information on this algorithm, check out [Approach 2 in the Preorder Traversal Solution Article](https://leetcode.com/problems/binary-tree-preorder-traversal/solution/).

**Algorithm**

These solutions are based on [C code](https://leetcode.com/problems/sum-of-left-leaves/discuss/124147/C-solution-with-Morris-Traversal) written by [@kamanelf](https://leetcode.com/kamanelf/) and the template from the [Preorder Traversal Solution Article](https://leetcode.com/problems/binary-tree-preorder-traversal/solution/).

|  |
| --- |
| class Solution {  public int sumOfLeftLeaves(TreeNode root) {  int totalSum = 0;  TreeNode currentNode = root;  while (currentNode != null) {  // If there is no left child, we can simply explore the right subtree  // without needing to worry about keeping track of currentNode's other  // child.  if (currentNode.left == null) {  currentNode = currentNode.right;  } else {  TreeNode previous = currentNode.left;  // Check if this left node is a leaf node.  if (previous.left == null && previous.right == null) {  totalSum += previous.val;  }  // Find the inorder predecessor for currentNode.  while (previous.right != null && !previous.right.equals(currentNode)) {  previous = previous.right;  }  // We've not yet visited the inorder predecessor. This means that we  // still need to explore currentNode's left subtree. Before doing this,  // we will put a link back so that we can get back to the right subtree  // when we need to.  if (previous.right == null) {  previous.right = currentNode;  currentNode = currentNode.left;  }  // We have already visited the inorder predecessor. This means that we  // need to remove the link we added, and then move onto the right  // subtree and explore it.  else {  previous.right = null;  currentNode = currentNode.right;  }  }  }  return totalSum;  }  } |

**Complexity Analysis**

* Time complexity : O(N)*O*(*N*).

Each node is visited at least once; with some nodes visited twice to remove the added links and move back up to the subtree root. However, no node is visited more than twice, so our time complexity is O(N)*O*(*N*).

* Space complexity : O(1)*O*(1).

We are only using constant extra space.

Note that while the input is modified, it is restored after the algorithm has finished running. The downside of this is that it is not thread-safe. Any other thread that needs to access the tree will have to wait until this algorithm has finished running. For applications that must support concurrent access, this is almost certainly not worth it given the availability of thread-safe alternatives.

**Simplify Path**

Given a string path, which is an **absolute path** (starting with a slash '/') to a file or directory in a Unix-style file system, convert it to the simplified **canonical path**.

In a Unix-style file system, a period '.' refers to the current directory, a double period '..' refers to the directory up a level, and any multiple consecutive slashes (i.e. '//') are treated as a single slash '/'. For this problem, any other format of periods such as '...' are treated as file/directory names.

The **canonical path** should have the following format:

* The path starts with a single slash '/'.
* Any two directories are separated by a single slash '/'.
* The path does not end with a trailing '/'.
* The path only contains the directories on the path from the root directory to the target file or directory (i.e., no period '.' or double period '..')

Return *the simplified****canonical path***.

**Example 1:**

**Input:** path = "/home/"

**Output:** "/home"

**Explanation:** Note that there is no trailing slash after the last directory name.

**Example 2:**

**Input:** path = "/../"

**Output:** "/"

**Explanation:** Going one level up from the root directory is a no-op, as the root level is the highest level you can go.

**Example 3:**

**Input:** path = "/home//foo/"

**Output:** "/home/foo"

**Explanation:** In the canonical path, multiple consecutive slashes are replaced by a single one.

**Example 4:**

**Input:** path = "/a/./b/../../c/"

**Output:** "/c"

**Constraints:**

* 1 <= path.length <= 3000
* path consists of English letters, digits, period '.', slash '/' or '\_'.
* path is a valid absolute Unix path.

## Solution

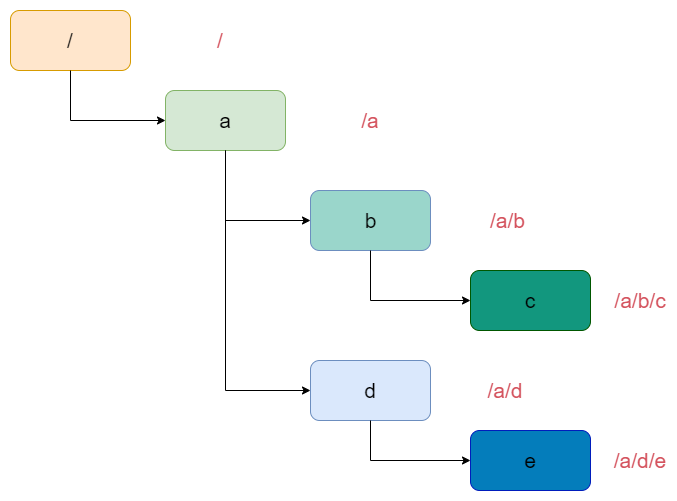
#### **Approach: Using Stacks**

**Intuition**

This is a direct implementation of one of the most common commands used (in one form of the other) on all the famous operating systems. For e.g. this directly translates to a part of the implementation of the cd command functionality in the Unix OS. It basically helps us to change directory. Obviously, there's much more to it than simply figuring out the smallest path of the final directory where the user wants to go. There is kernel level implementation involved that actually uses the underlying file system to change the directory you are in to the one where you want to go. You might argue that nobody would go crazy on the command line and want to run something like:

cd /a/b/c/.././././//d

However, our code needs to be able to handle the different scenarios and all the special characters like ., .., and /. All these weird scenarios will be taken care of by a couple of checks here and there. The main idea for solving this problem would remain the same. The heading of this approach mentions the data structure that we are going to use. However, we are going to work our way through the problem so as to understand as to why a stack fits in here. Let's look at a tree-ish representation of a simple directory path.



Now suppose that to a path like /a/b/c, we add another component like /a/b/c/... Now, this is interesting because the .. is no longer a sub-directory name. It has a special meaning and an indication to the operating system to move up one level in the directory structure thus transforming the overall path to just /a/b. It's as if we popped out the subdirectory c from the overall path. That's the core idea of this problem. If you think about it, the only actionable special character is ... The single dot is kind of a no-op because it simply means the current directory. So nothing changes in the overall path as such. Now that we have a general idea about how this problem can be potentially solved using the stack, let's get right into the algorithm and discuss the solution.

**Algorithm**

1. Initialize a stack, S that we will be using for our implementation.
2. Split the input string using / as the delimiter. This step is really important because no matter what, the given input is a valid path and we simply have to shorten it. So, that means that whatever we have between two / characters is either a directory name or a special character and we have to process them accordingly.
3. Once we are done splitting the input path, we will process one component at a time.
4. If the current component is a . or an empty string, we will do nothing and simply continue. Well if you think about it, the split string array for the string /a//b would be [a,,b]. yes, that's an empty string in between a and b. Again, from the perspective of the overall path, it doesn't mean anything.
5. If we encounter a double-dot .., we have to do some processing. This simply means go one level up in the current directory path. So, we will pop an entry from our stack if it's not empty.
6. Finally, if the component we are processing right now is not one of the special characters, then we will simply add it to our stack because it's a legitimate directory name.
7. Once we are done processing all the components, we simply have to connect all the directory names in our stack together using / as the delimiter and we will have our shortest path that leads us to the same directory as the one provided as an input.

|  |
| --- |
| class Solution {  public String simplifyPath(String path) {  // Initialize a stack  Stack<String> stack = new Stack<String>();  String[] components = path.split("/");  // Split the input string on "/" as the delimiter  // and process each portion one by one  for (String directory : components) {  // A no-op for a "." or an empty string  if (directory.equals(".") || directory.isEmpty()) {  continue;  } else if (directory.equals("..")) {  // If the current component is a "..", then  // we pop an entry from the stack if it's non-empty  if (!stack.isEmpty()) {  stack.pop();  }  } else {  // Finally, a legitimate directory name, so we add it  // to our stack  stack.add(directory);  }  }  // Stich together all the directory names together  StringBuilder result = new StringBuilder();  for (String dir : stack) {  result.append("/");  result.append(dir);  }  return result.length() > 0 ? result.toString() : "/" ;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*) if there are N*N* characters in the original path. First, we spend O(N)*O*(*N*) trying to split the input path into components and then we process each component one by one which is again an O(N)*O*(*N*) operation. We can get rid of the splitting part and just string together the characters and form directory names etc. However, that would be too complicated and not worth depicting in the implementation. The main idea of this algorithm is to use a stack. How you decide to process the input string is a personal choice.
* Space Complexity: O(N)*O*(*N*). Actually, it's 2N2*N* because we have the array that contains the split components and then we have the stack.

**Basic Calculator**

Given a string s representing an expression, implement a basic calculator to evaluate it.

**Example 1:**

**Input:** s = "1 + 1"

**Output:** 2

**Example 2:**

**Input:** s = " 2-1 + 2 "

**Output:** 3

**Example 3:**

**Input:** s = "(1+(4+5+2)-3)+(6+8)"

**Output:** 23

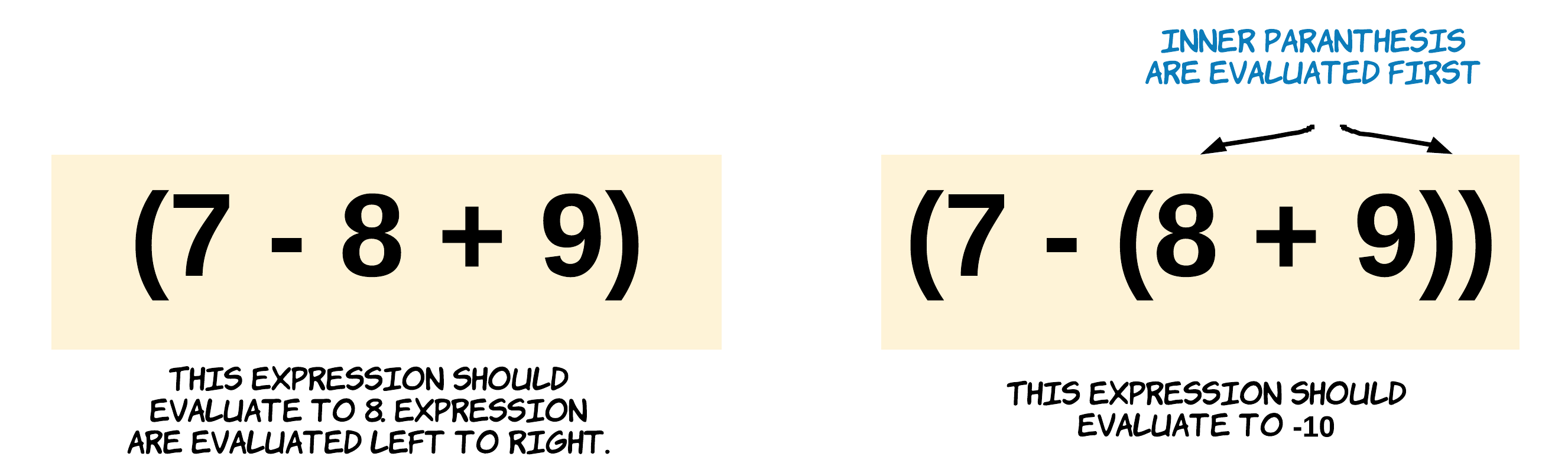
**Constraints:**

* 1 <= s.length <= 3 \* 105
* s consists of digits, '+', '-', '(', ')', and ' '.
* s represents a valid expression.

## Solution

This problem is all about understanding the following:

* Input always contains valid strings
* The rules of addition and subtraction
* Implication of precedence by parenthesis
* Spaces do not affect the evaluation of the input expression

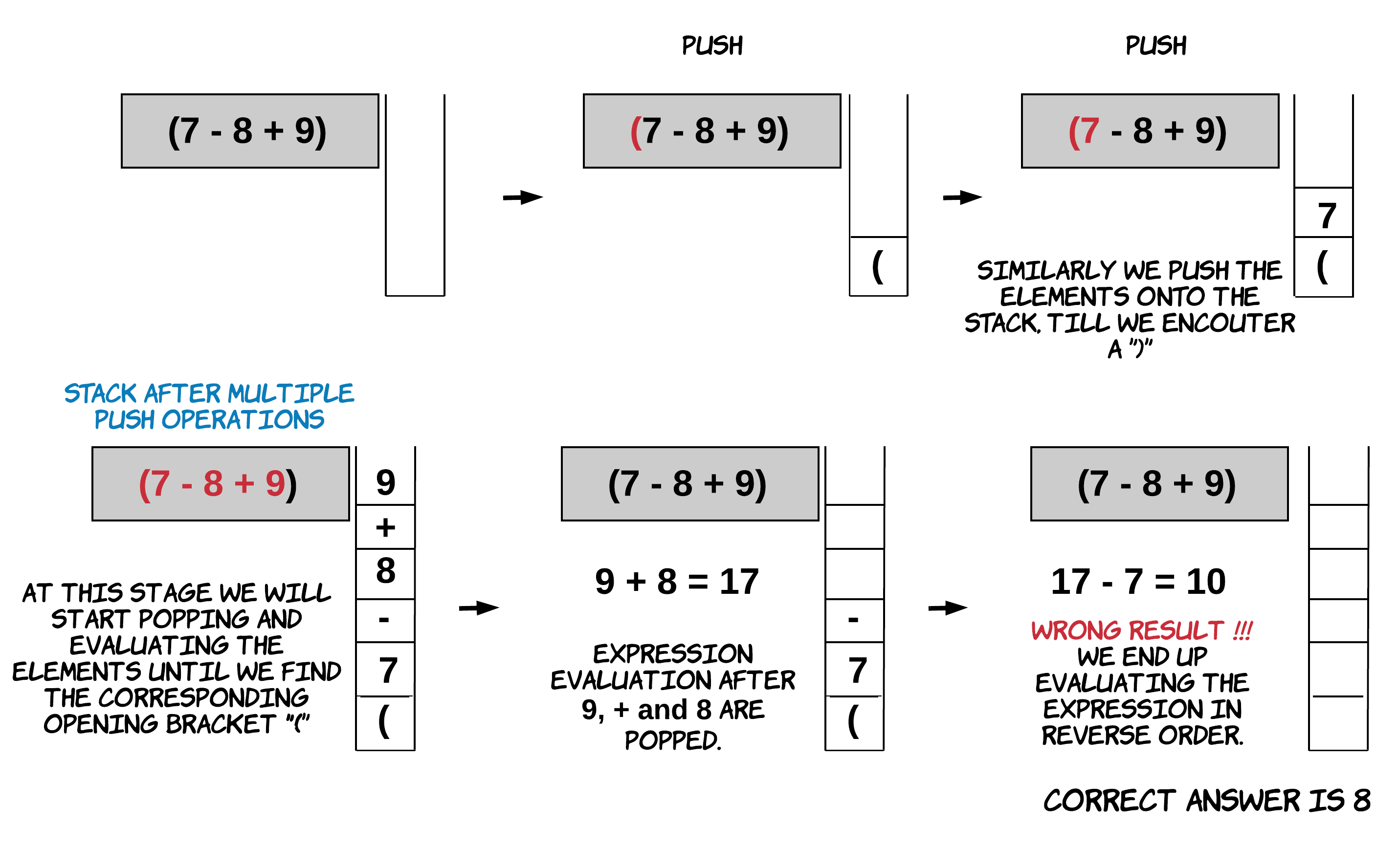


#### **Approach 1: Stack and String Reversal**

**Intuition**

This question qualifies really well for a stack question. Since the expression might have parenthesis, we can use a stack to find the value for each sub-expression within a parenthesis. Essentially, we need to delay processing the main expression until we are done evaluating the interim sub-expressions within parenthesis and to introduce this delay, we use a stack.

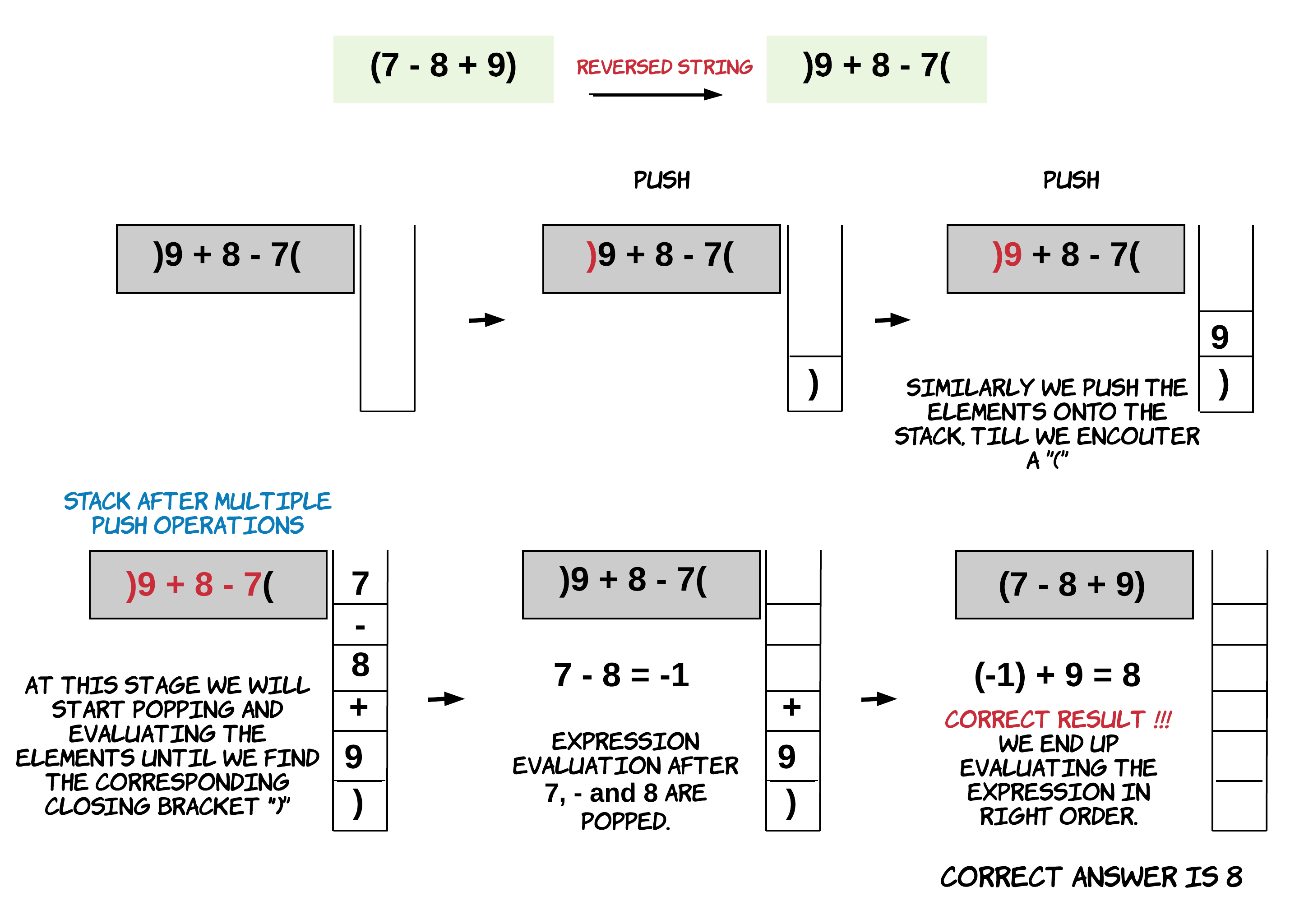
We push the elements of the expression one by one onto the stack until we get a closing bracket ). Then we pop the elements from the stack one by one and evaluate the expression on-the-go. This is done till we find the corresponding ( opening bracket. This kind of evaluation is very common when using the stack data structure. However, if you notice the way we calculate the final answer, you will realize that we actually process the values from right to left whereas it should be the other way around.



From the above example we realize that following the simple stack push and pop methodology will not help us here. We need to understand how + and - work. + follows the associative property. For the expression A+B+C*A*+*B*+*C*, we have (A+B)+C = A+(B+C)(*A*+*B*)+*C*=*A*+(*B*+*C*). However, - does not follow this rule which is the root cause of all the problems in this approach.

If we use a stack and read the elements of the expression from left to right, we end up evaluating the expression from right-to-left. This means we are expecting (A-B)-C(*A*−*B*)−*C* to be equal to (C-B)-A(*C*−*B*)−*A* which is not correct. Subtraction is neither associative nor commutative.

This problem could be solved very easily by reversing the string and then using basic drill using a stack. Reversing a string helps since we now put the elements of the expression into the stack from right to left and evaluation for the expression is done correctly from left to right.



**Algorithm**

1. Iterate the expression string in reverse order one character at a time. Since we are reading the expression character by character, we need to be careful when we are reading digits and non-digits.
2. The operands could be formed by multiple characters. A string "123" would mean a numeric 123, which could be formed as: 123 >> 120 + 3 >> 100 + 20 + 3. Thus, if the character read is a digit we need to form the operand by multiplying a power of 10 to the current digit and adding it to the overall operand. We do this since we are processing the string in the reverse order.
3. The operands could be formed by multiple characters. We need to keep track of an on-going operand. This part is a bit tricky since in this case the string is reversed. Once we encounter a character which is not a digit, we push the operand onto the stack.
4. When we encounter an opening parenthesis (, this means an expression just ended. Recall we have reversed the expression. So an opening bracket would signify the end of the an expression. This calls for evaluation of the expression by popping operands and operators off the stack till we pop corresponding closing parenthesis. The final result of the expression is pushed back onto the stack.

Note: We are evaluating all the sub-expressions within the main expression. The sub-expressions on the right get evaluated first but the main expression itself is evaluated from left to right when all its components are resolved, which is very important for correct results.

For eg. For expression A - (B+C) + (D+E-F)*A*−(*B*+*C*)+(*D*+*E*−*F*), D+E-F*D*+*E*−*F* is evaluated before B+C*B*+*C*. While evaluating D+E-F*D*+*E*−*F* the order is from left to right. Similarly for the parent expression, all the child components are evaluated and stored on the stack so that final evaluation is left to right.

1. Push the other non-digits onto to the stack.
2. Do this until we get the final result. It's possible that we don't have any more characters left to process but the stack is still non-empty. This would happen when the main expression is not enclosed by parenthesis. So, once we are done evaluating the entire expression, we check if the stack is non-empty. If it is, we treat the elements in it as one final expression and evaluate it the same way we would if we had encountered an opening bracket.

We can also cover the original expression with a set of parenthesis to avoid this extra call.

|  |
| --- |
| class Solution {  public int evaluateExpr(Stack<Object> stack) {  int res = 0;  if (!stack.empty()) {  res = (int) stack.pop();  }  // Evaluate the expression till we get corresponding ')'  while (!stack.empty() && !((char) stack.peek() == ')')) {  char sign = (char) stack.pop();  if (sign == '+') {  res += (int) stack.pop();  } else {  res -= (int) stack.pop();  }  }  return res;  }  public int calculate(String s) {  int operand = 0;  int n = 0;  Stack<Object> stack = new Stack<Object>();  for (int i = s.length() - 1; i >= 0; i--) {  char ch = s.charAt(i);  if (Character.isDigit(ch)) {  // Forming the operand - in reverse order.  operand = (int) Math.pow(10, n) \* (int) (ch - '0') + operand;  n += 1;  } else if (ch != ' ') {  if (n != 0) {  // Save the operand on the stack  // As we encounter some non-digit.  stack.push(operand);  n = 0;  operand = 0;  }  if (ch == '(') {  int res = evaluateExpr(stack);  stack.pop();  // Append the evaluated result to the stack.  // This result could be of a sub-expression within the parenthesis.  stack.push(res);  } else {  // For other non-digits just push onto the stack.  stack.push(ch);  }  }  }  //Push the last operand to stack, if any.  if (n != 0) {  stack.push(operand);  }  // Evaluate any left overs in the stack.  return evaluateExpr(stack);  }  } |

**Complexity Analysis**

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

#### **Approach 2: Stack and No String Reversal**

**Intuition**

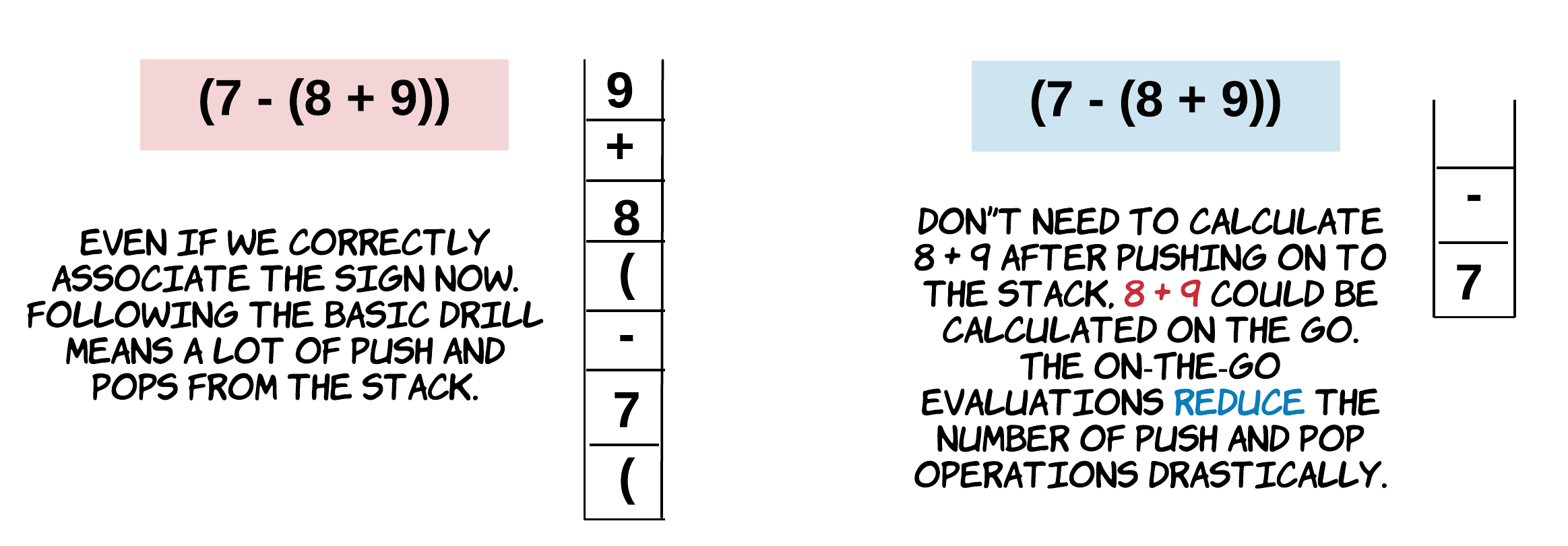
A very easy way to solve the problem of associativity for - we tackled in the previous approach, is to use - operator as the magnitude for the operand to the right of the operator. Once we start using - as a magnitude for the operands, we just have one operator left which is addition and + is associative.

for e.g. A - B - C*A*−*B*−*C* could be re-written as A + (-B) + (-C)*A*+(−*B*)+(−*C*).

The re-written expression would follow associativity rule. Thus evaluating the expression from right or left, won't change the result.

What we need to keep in mind is that the expressions given would be complicated, i.e. there would be expressions nested within other expressions. Even if we have something like (A - (B - C)) we need to associate the negative sign outside of B-C with the result of B-C instead of just with B.

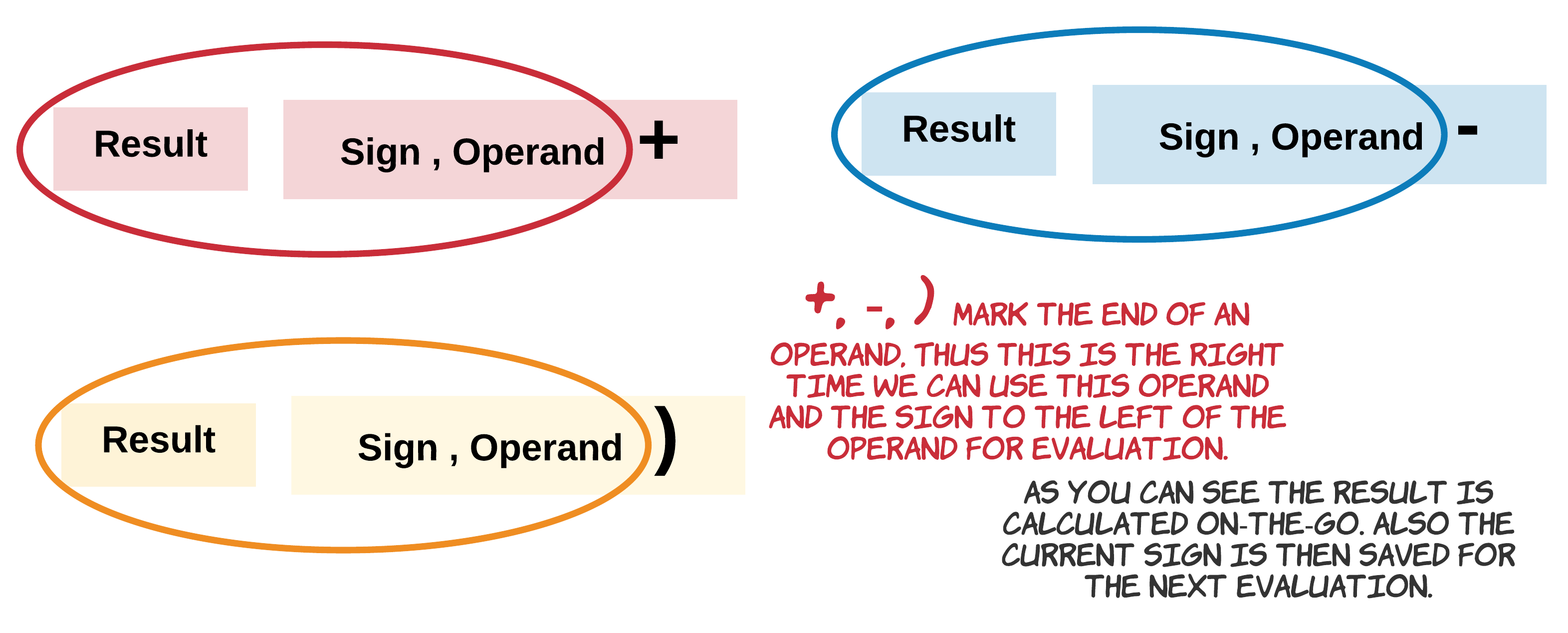
We can solve this problem by following the basic drill before and associating the sign with the expression to the right of it. However, the approach that we will instead take has a small twist to it in that we will be evaluating most of the expression on-the-go. This reduces the number of push and pop operations.



Follow the below steps closely. This algorithm is inspired from [discussion post](https://leetcode.com/problems/basic-calculator/discuss/62361/southpenguin) by [southpenguin](https://leetcode.com/southpenguin/).

**Algorithm**

1. Iterate the expression string one character at a time. Since we are reading the expression character by character, we need to be careful when we are reading digits and non-digits.
2. The operands could be formed by multiple characters. A string "123" would mean a numeric 123, which could be formed as: 123 >> 120 + 3 >> 100 + 20 + 3. Thus, if the character read is a digit we need to form the operand by multiplying 10 to the previously formed continuing operand and adding the digit to it.
3. Whenever we encounter an operator such as + or - we first evaluate the expression to the left and then save this sign for the next evaluation.



1. If the character is an opening parenthesis (, we just push the result calculated so far and the sign on to the stack (the sign and the magnitude) and start a fresh as if we are calculating a new expression.
2. If the character is a closing parenthesis ), we first calculate the expression to the left. The result from this would be the result of the expression within the set of parenthesis that just concluded. This result is then multiplied with the sign, if there is any on top of the stack. Remember we saved the sign on top of the stack when we had encountered an open parenthesis? This sign is associated with the parenthesis that started then, thus when the expression ends or concludes, we pop the sign and multiply it with result of the expression. It is then just added to the next element on top of the stack.

|  |
| --- |
| class Solution {  public int calculate(String s) {  Stack<Integer> stack = new Stack<Integer>();  int operand = 0;  int result = 0; // For the on-going result  int sign = 1; // 1 means positive, -1 means negative  for (int i = 0; i < s.length(); i++) {  char ch = s.charAt(i);  if (Character.isDigit(ch)) {  // Forming operand, since it could be more than one digit  operand = 10 \* operand + (int) (ch - '0');  } else if (ch == '+') {  // Evaluate the expression to the left,  // with result, sign, operand  result += sign \* operand;  // Save the recently encountered '+' sign  sign = 1;  // Reset operand  operand = 0;  } else if (ch == '-') {  result += sign \* operand;  sign = -1;  operand = 0;  } else if (ch == '(') {  // Push the result and sign on to the stack, for later  // We push the result first, then sign  stack.push(result);  stack.push(sign);  // Reset operand and result, as if new evaluation begins for the new sub-expression  sign = 1;  result = 0;  } else if (ch == ')') {  // Evaluate the expression to the left  // with result, sign and operand  result += sign \* operand;  // ')' marks end of expression within a set of parenthesis  // Its result is multiplied with sign on top of stack  // as stack.pop() is the sign before the parenthesis  result \*= stack.pop();  // Then add to the next operand on the top.  // as stack.pop() is the result calculated before this parenthesis  // (operand on stack) + (sign on stack \* (result from parenthesis))  result += stack.pop();  // Reset the operand  operand = 0;  }  }  return result + (sign \* operand);  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N is the length of the string. The difference in time complexity between this approach and the previous one is that every character in this approach will get processed exactly once. However, in the previous approach, each character can potentially get processed twice, once when it's pushed onto the stack and once when it's popped for processing of the final result (or a subexpression). That's why this approach is faster.
* Space Complexity: O(N)*O*(*N*), where N is the length of the string.

**Remove K Digits**

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

**Note:**

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

**Example 1:**

Input: num = "1432219", k = 3

Output: "1219"

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

**Example 2:**

Input: num = "10200", k = 1

Output: "200"

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

**Example 3:**

Input: num = "10", k = 2

Output: "0"

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

## Solution

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

**Intuition**

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

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

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

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

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

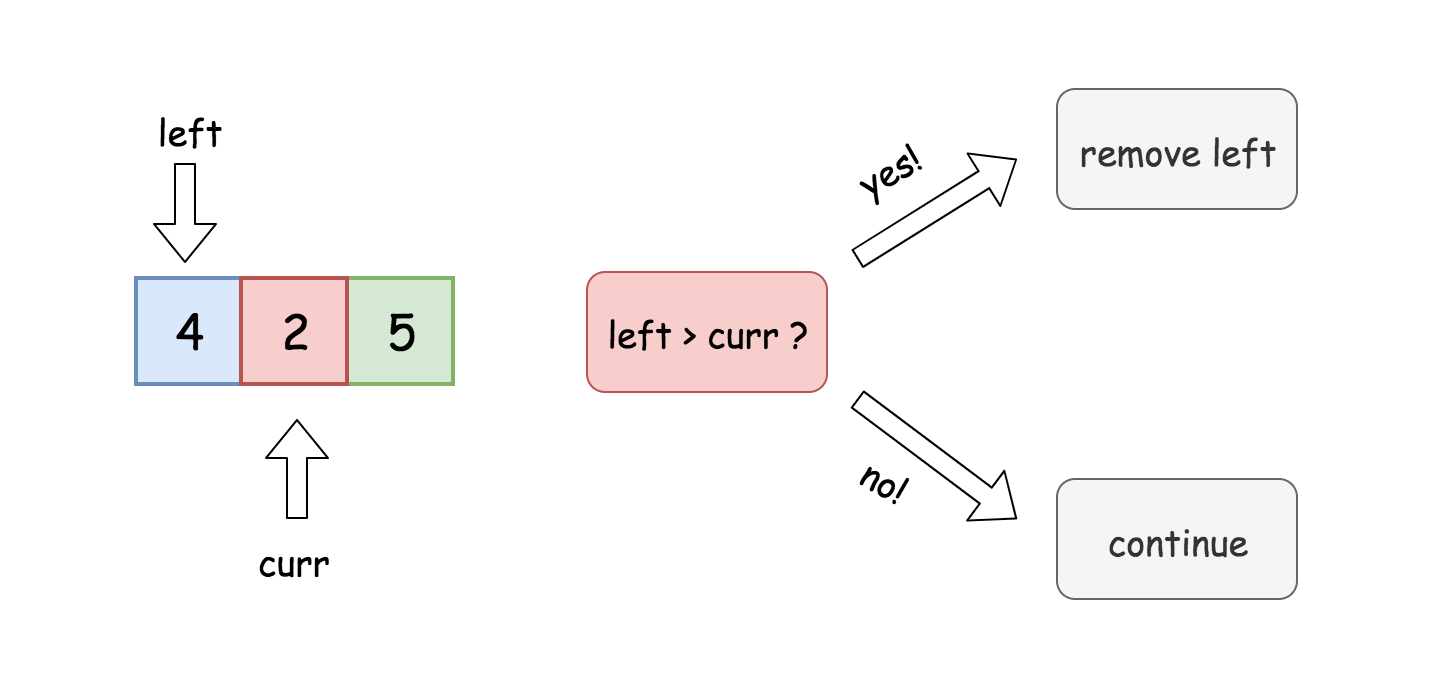
#### **Approach 2: Greedy with Stack**

**Intuition**

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

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

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



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

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

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

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

**Algorithm**

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

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

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

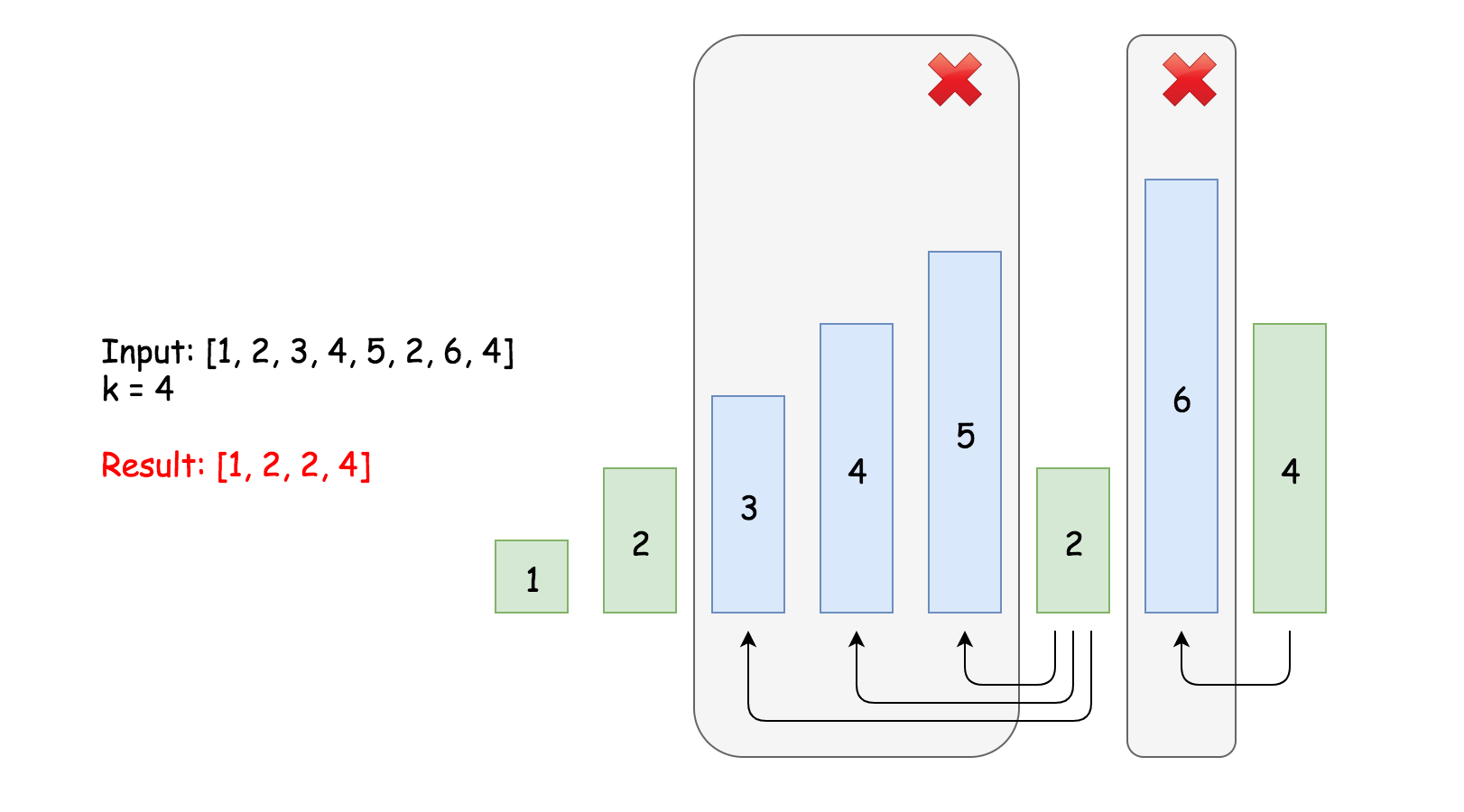
On the other hand, we did provide a **proof by contradiction**, with the simple example of 425 in the Intuition section, that by repeatedly applying the rule we would obtain the optimal solution.

**Implementation**

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

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

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



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

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

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

Here are some sample implementations.

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

**Complexity Analysis**

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

**Search Insert Position**

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

**Example 1:**

**Input:** nums = [1,3,5,6], target = 5

**Output:** 2

**Example 2:**

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

**Output:** 1

**Example 3:**

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

**Output:** 4

**Example 4:**

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

**Output:** 0

**Example 5:**

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

**Output:** 0

**Constraints:**

* 1 <= nums.length <= 104
* -104 <= nums[i] <= 104
* nums contains **distinct** values sorted in **ascending** order.
* -104 <= target <= 104

## Solution

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

**Intuition**

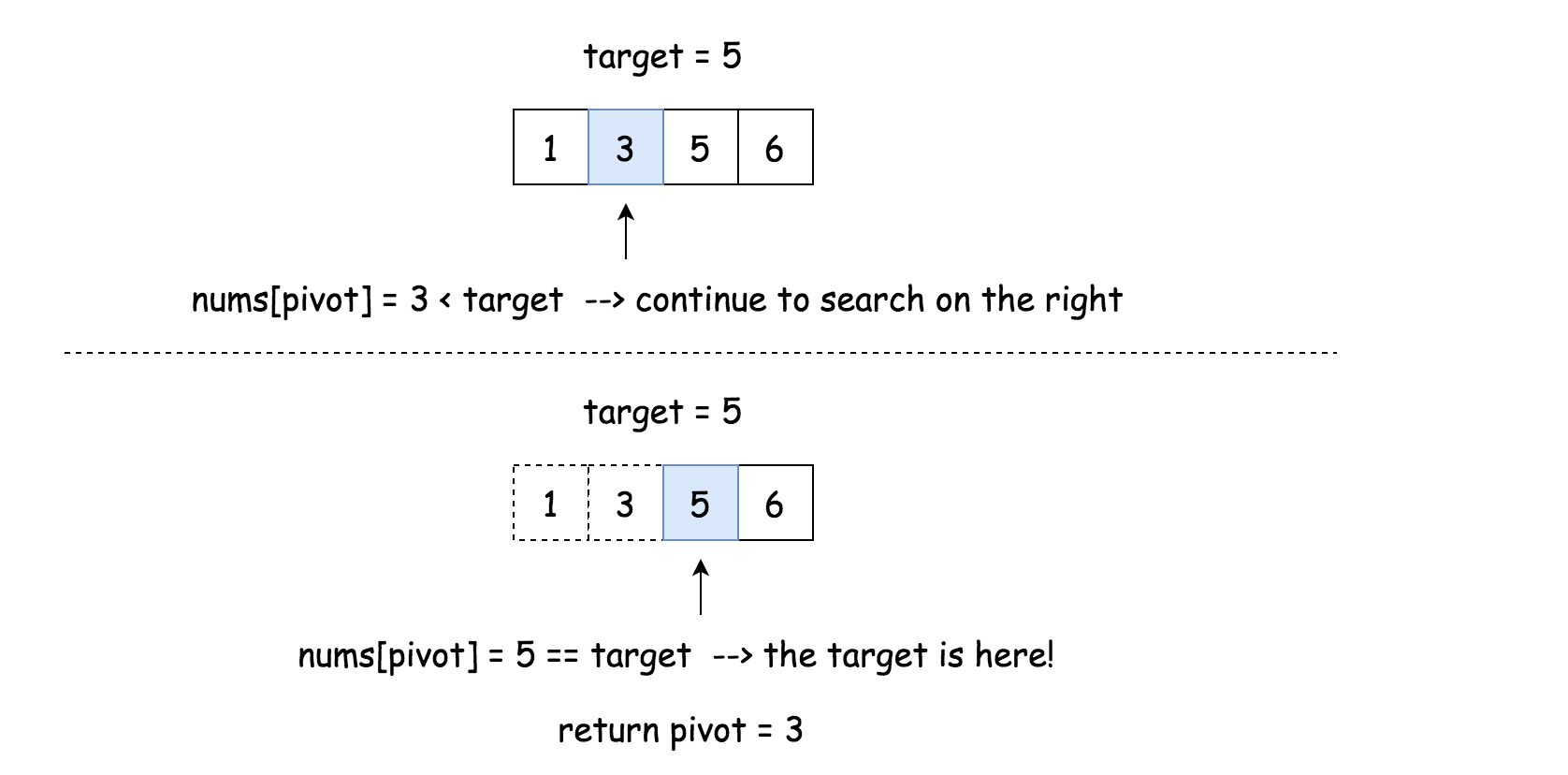
Based on the description of the problem, we can see that it could be a good match with the [binary search](https://en.wikipedia.org/wiki/Binary_search_algorithm) algorithm.

Binary search is a search algorithm that find the position of a target value within a sorted array.

Usually, within binary search, we compare the target value to the middle element of the array at each iteration.

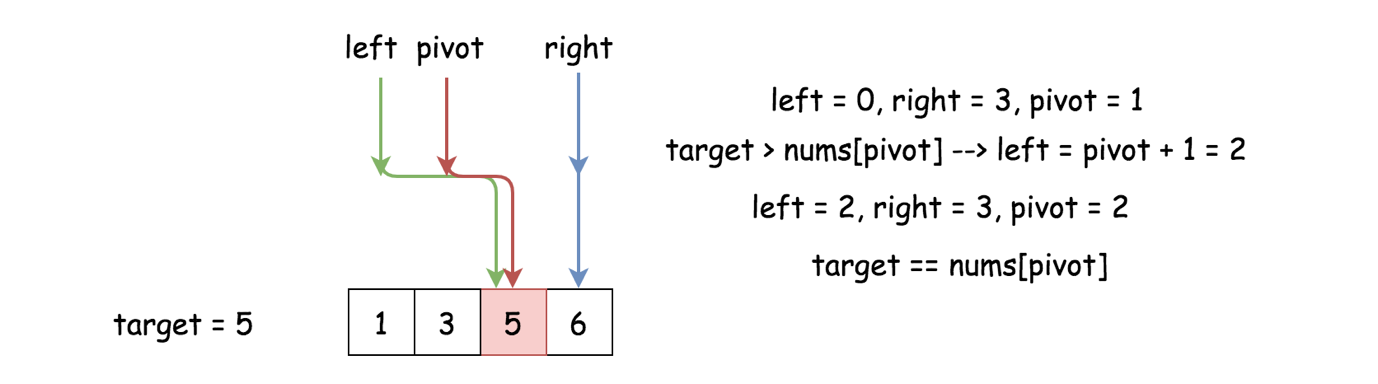
* If the target value is equal to the middle element, the job is done.
* If the target value is less than the middle element, continue to search on the left.
* If the target value is greater than the middle element, continue to search on the right.

Here we showcase a simple example on how it works.



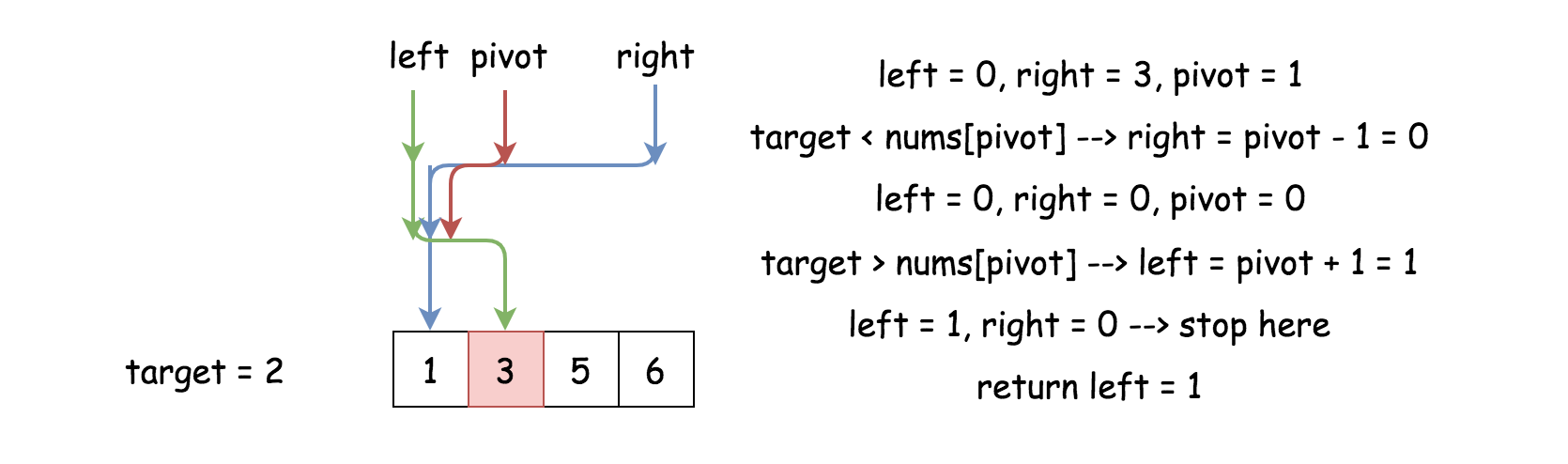
To mark the search boundaries, one could use two pointers: left and right. Starting from left = 0 and right = n - 1, we then move either of the pointers according to various situations:

* While left <= right:
  + Pivot index is the one in the middle: pivot = (left + right) / 2. The pivot also divides the original array into two subarray.
  + If the target value is equal to the pivot element: target == nums[pivot], we're done.
  + If the target value is less than the pivot element target < nums[pivot], continue to search on the left subarray by moving the right pointer right = pivot - 1.
  + If the target value is greater than the pivot element target > nums[pivot], continue to search on the right subarray by moving the left pointer left = pivot + 1.



What if the target value is not found?

In this case, the loop will be stopped at the moment when right < left and nums[right] < target < nums[left]. Hence, the proper position to insert the target is at the index left.



**Integer Overflow**

Let us now stress the fact that pivot = (left + right) // 2 works fine for Python3, which has arbitrary precision integers, but it could cause some issues in Java and C++.

If left + right is greater than the maximum int value 2^{31} - 1231−1, it overflows to a negative value. In Java, it would trigger an exception of ArrayIndexOutOfBoundsException, and in C++ it causes an illegal write, which leads to memory corruption and unpredictable results.

Here is a simple way to fix it:

|  |
| --- |
| pivot = left + (right - left) / 2; |

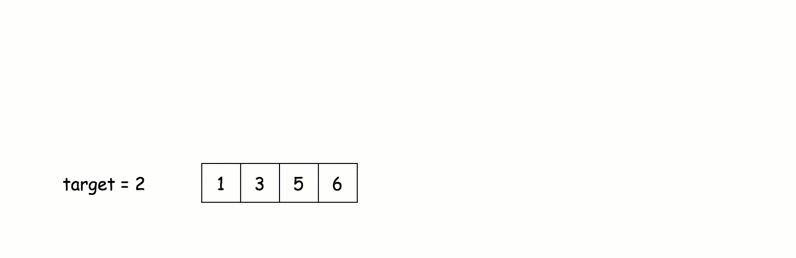
and here is a bit more complicated but probably faster way using the bit shift operator.

|  |
| --- |
| pivot = (right + left) >>> 1; |

**Algorithm**

* Initialize the left and right pointers : left = 0, right = n - 1.
* While left <= right:
  + Compare middle element of the array nums[pivot] to the target value target.
    - If the middle element *is* the target, *i.e.* target == nums[pivot]: return pivot.
    - If the target is not here:
      * If target < nums[pivot], continue to search on the left subarray. right = pivot - 1.
      * Else continue to search on the right subarray. left = pivot + 1.
* Return left.

**Implementation**



|  |
| --- |
| class Solution {  public int searchInsert(int[] nums, int target) {  int pivot, left = 0, right = nums.length - 1;  while (left <= right) {  pivot = left + (right - left) / 2;  if (nums[pivot] == target) return pivot;  if (target < nums[pivot]) right = pivot - 1;  else left = pivot + 1;  }  return left;  }  } |

**Complexity Analysis**

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

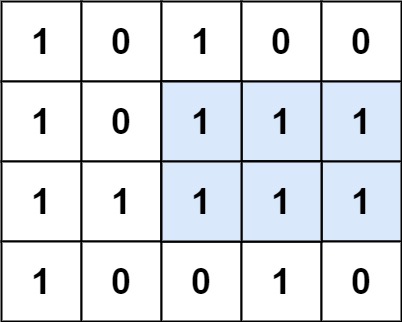
Let us compute the time complexity with the help of [master theorem](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)) T(N) = aT\left(\frac{N}{b}\right) + \Theta(N^d)*T*(*N*)=*aT*(*bN*​)+Θ(*Nd*). The equation represents dividing the problem up into a*a* subproblems of size \frac{N}{b}*bN*​ in \Theta(N^d)Θ(*Nd*) time. Here at each step there is only one subproblem *i.e.* a = 1, its size is a half of the initial problem *i.e.* b = 2, and all this happens in a constant time *i.e.* d = 0. As a result, \log\_b{a} = dlog*b*​*a*=*d* and hence we're dealing with [case 2](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)#Case_2_example) that results in \mathcal{O}(n^{\log\_b{a}} \log^{d + 1} N)O(*n*log*b*​*a*log*d*+1*N*) = \mathcal{O}(\log N)O(log*N*) time complexity.

* Space complexity : \mathcal{O}(1)O(1) since it's a constant space solution.

**Maximal Rectangle**

Given a rows x cols binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return *its area*.

**Example 1:**



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

**Output:** 6

**Explanation:** The maximal rectangle is shown in the above picture.

**Example 2:**

**Input:** matrix = []

**Output:** 0

**Example 3:**

**Input:** matrix = [["0"]]

**Output:** 0

**Example 4:**

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

**Output:** 1

**Example 5:**

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

**Output:** 0

**Constraints:**

* rows == matrix.length
* cols == matrix.length
* 0 <= row, cols <= 200
* matrix[i][j] is '0' or '1'.

## Solution

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

**Algorithm**

Trivially we can enumerate every possible rectangle. This is done by iterating over all possible combinations of coordinates (x1, y1) and (x2, y2) and letting them define a rectangle with the coordinates being opposite corners. This is too slow to pass all test cases.

**Complexity Analysis**

* Time complexity : O(N^3M^3)*O*(*N*3*M*3), with N being the number of rows and M the number of columns.

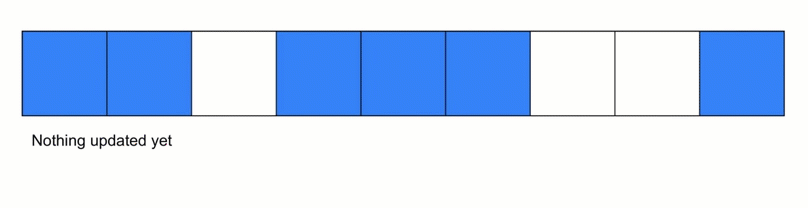
Iterating over all possible coordinates is O(N^2M^2)*O*(*N*2*M*2), and iterating over the rectangle defined by two coordinates is an additional O(NM)*O*(*NM*). O(NM) \* O(N^2M^2) = O(N^3M^3)*O*(*NM*)∗*O*(*N*2*M*2)=*O*(*N*3*M*3).

* Space complexity : O(1)*O*(1).

#### **Approach 2: Dynamic Programming - Better Brute Force on Histograms**

**Algorithm**

We can compute the maximum width of a rectangle that ends at a given coordinate in constant time. We do this by keeping track of the number of consecutive ones each square in each row. As we iterate over each row we update the maximum possible width at that point. This is done using row[i] = row[i - 1] + 1 if row[i] == '1'.



Once we know the maximum widths for each point above a given point, we can compute the maximum rectangle with the lower right corner at that point in linear time. As we iterate up the column, we know that the maximal width of a rectangle spanning from the original point to the current point is the running minimum of each maximal width we have encountered.

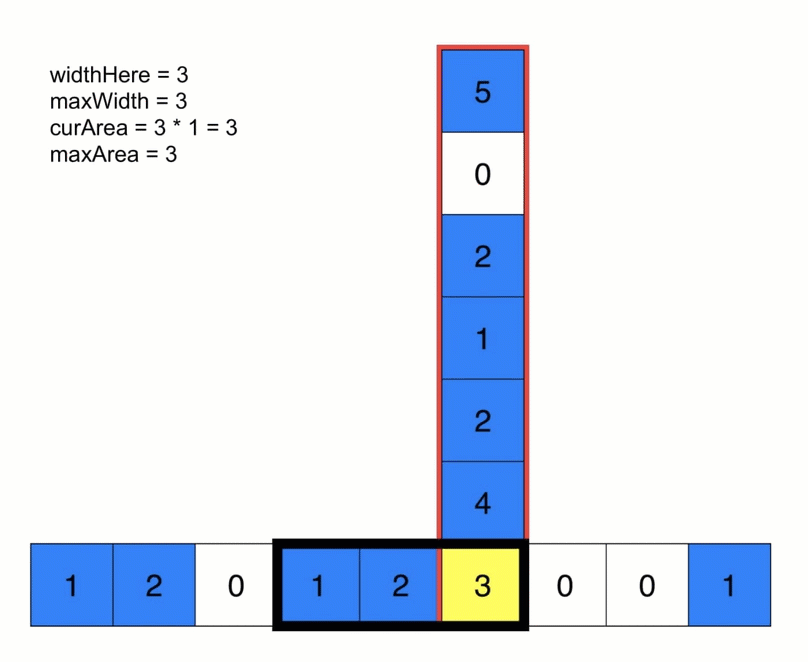
We define:

maxWidth = min(maxWidth, widthHere)*maxWidth*=*min*(*maxWidth*,*widthHere*)

curArea = maxWidth \* (currentRow - originalRow + 1)*curArea*=*maxWidth*∗(*currentRow*−*originalRow*+1)

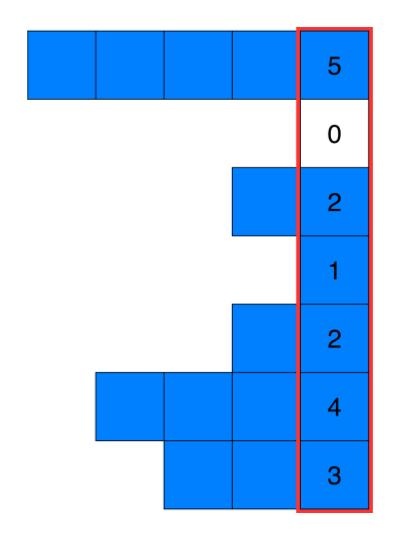
maxArea = max(maxArea, curArea)*maxArea*=*max*(*maxArea*,*curArea*)

The following animation makes this more clear. Given the maximal width of all points above it, let's calculate the maximum area of any rectangle at the bottom yellow square:



Repeating this process for every point in our input gives us the global maximum.

Note that our method of precomputing our maximum width essentially breaks down our input into a set of histograms, with each column being a new histogram. We are computing the maximal area for each histogram.



As a result, the above approach is essentially a repeated use of the better brute force approach detailed in [84 - Largest Rectangle in Histogram](https://leetcode.com/problems/largest-rectangle-in-histogram/solution/).

|  |
| --- |
| class Solution {  public int maximalRectangle(char[][] matrix) {  if (matrix.length == 0) return 0;  int maxarea = 0;  int[][] dp = new int[matrix.length][matrix[0].length];  for(int i = 0; i < matrix.length; i++){  for(int j = 0; j < matrix[0].length; j++){  if (matrix[i][j] == '1'){  // compute the maximum width and update dp with it  dp[i][j] = j == 0? 1 : dp[i][j-1] + 1;  int width = dp[i][j];  // compute the maximum area rectangle with a lower right corner at [i, j]  for(int k = i; k >= 0; k--){  width = Math.min(width, dp[k][j]);  maxarea = Math.max(maxarea, width \* (i - k + 1));  }  }  }  } return maxarea;  }  } |

**Complexity Analysis**

* Time complexity : O(N^2M)*O*(*N*2*M*). Computing the maximum area for one point takes O(N)*O*(*N*) time, since it iterates over the values in the same column. This is done for all N \* M*N*∗*M* points, giving O(N) \* O(NM) = O(N^2M)*O*(*N*)∗*O*(*NM*)=*O*(*N*2*M*).
* Space complexity : O(NM)*O*(*NM*). We allocate an equal sized array to store the maximum width at each point.

#### **Approach 3: Using Histograms - Stack**

**Algorithm**

In the previous approach we discussed breaking the input into a set of histograms - one histogram representing the substructure at each column. To compute the maximum area in our rectangle, we merely have to compute the maximum area of each histogram and find the global maximum (note that the below approach builds a histogram for each row instead of each column, but the idea is still the same).

Since [Largest Rectangle in Histogram](https://leetcode.com/problems/largest-rectangle-in-histogram/) is already a problem on leetcode, we can just borrow the fastest stack-based solution [here](https://leetcode.com/problems/largest-rectangle-in-histogram/solution/) and apply it onto each histogram we generate. For an in-depth explanation on how the Largest Rectangle in Histogram algorithm works, please use the links above.

|  |
| --- |
| class Solution {  // Get the maximum area in a histogram given its heights  public int leetcode84(int[] heights) {  Stack < Integer > stack = new Stack < > ();  stack.push(-1);  int maxarea = 0;  for (int i = 0; i < heights.length; ++i) {  while (stack.peek() != -1 && heights[stack.peek()] >= heights[i])  maxarea = Math.max(maxarea, heights[stack.pop()] \* (i - stack.peek() - 1));  stack.push(i);  }  while (stack.peek() != -1)  maxarea = Math.max(maxarea, heights[stack.pop()] \* (heights.length - stack.peek() -1));  return maxarea;  }  public int maximalRectangle(char[][] matrix) {  if (matrix.length == 0) return 0;  int maxarea = 0;  int[] dp = new int[matrix[0].length];  for(int i = 0; i < matrix.length; i++) {  for(int j = 0; j < matrix[0].length; j++) {  // update the state of this row's histogram using the last row's histogram  // by keeping track of the number of consecutive ones  dp[j] = matrix[i][j] == '1' ? dp[j] + 1 : 0;  }  // update maxarea with the maximum area from this row's histogram  maxarea = Math.max(maxarea, leetcode84(dp));  } return maxarea;  }  } |

Note that the code under the function leetcode84 is a direct copy paste from the final solution in [84 - Largest Rectangle in Histogram](https://leetcode.com/problems/largest-rectangle-in-histogram/solution/).

**Complexity Analysis**

* Time complexity : O(NM)*O*(*NM*). Running leetcode84 on each row takes M (length of each row) time. This is done N times for O(NM)*O*(*NM*).
* Space complexity : O(M)*O*(*M*). We allocate an array the size of the the number of columns to store our widths at each row.

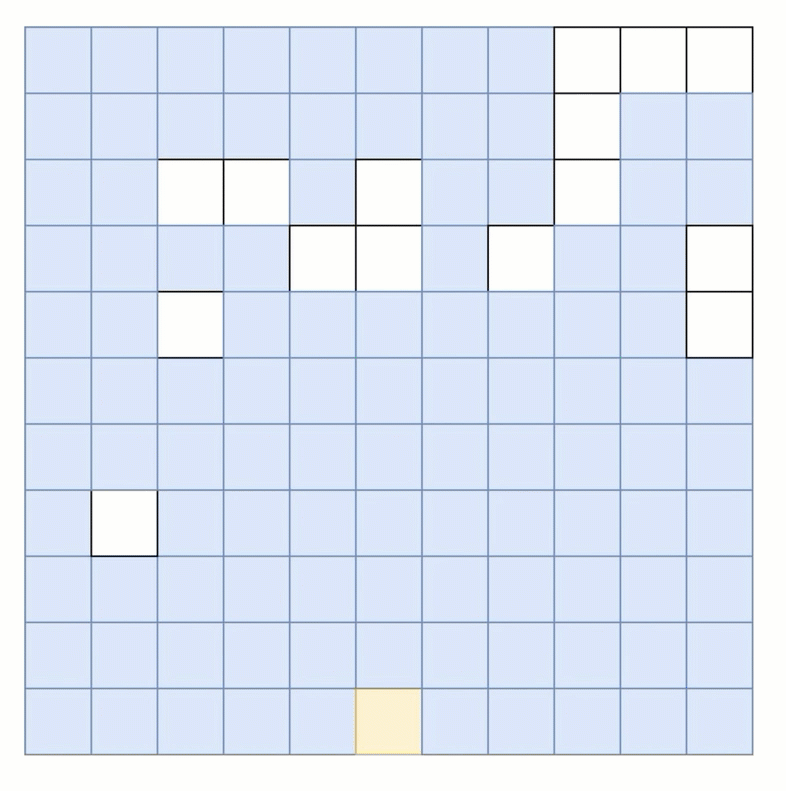
#### **Approach 4: Dynamic Programming - Maximum Height at Each Point**

**Intuition**

Imagine an algorithm where for each point we computed a rectangle by doing the following:

* Finding the maximum height of the rectangle by iterating upwards until a 0 is reached
* Finding the maximum width of the rectangle by iterating outwards left and right until a height that doesn't accommodate the maximum height of the rectangle

For example finding the rectangle defined by the yellow point:



We know that the maximal rectangle must be one of the rectangles constructed in this manner.

Given a maximal rectangle with height h, left bound l, and right bound r, there must be a point on the interval [l, r] on the rectangle's base where the number of consecutive ones (height) above the point is <=h. If this point exists, then the rectangle defined by the point in the above manner will be the maximal rectangle, as it will reach height h iterating upward and then expand to the bounds of [l, r] as all heights within those bounds must accommodate h for the rectangle to exist.

If this point does not exist, then the rectangle cannot be maximum, as you would be able to create a bigger rectangle by simply increasing the height of original rectangle, since all heights on the interval [l, r] would be greater than h.

As a result for each point you only need to compute h, l, and r - the height, left bound, and right bound of the rectangle it defines.

Using dynamic programming, we can use the h, l, and r of each point in the previous row to compute the h, l, and r for every point in the next row in linear time.

**Algorithm**

Given row matrix[i], we keep track of the h, l, and r of each point in the row by defining three arrays - height, left, and right.

height[j] will correspond to the height of matrix[i][j], and so on and so forth with the other arrays.

The question now becomes how to update each array.

Height:

This one is easy. h is defined as the number of continuous ones in a line from our point. We explored how to compute this in Approach 2 in one row with:

row[j] = row[j - 1] + 1 if row[j] == '1'

We can just make a minor modification for it to work for us here:

new\_height[j] = old\_height[j] + 1 if row[j] == '1' else 0

Left:

Consider what causes changes to the left bound of our rectangle. Since all instances of zeros occurring in the row above the current one have already been factored into the current version of left, the only thing that affects our left is if we encounter a zero in our current row.

As a result we can define:

new\_left[j] = max(old\_left[j], cur\_left)

cur\_left is one greater than rightmost occurrence of zero we have encountered. When we "expand" the rectangle to the left, we know it can't expand past that point, otherwise it'll run into the zero.

Right:

Here we can reuse our reasoning in left and define:

new\_right[j] = min(old\_right[j], cur\_right)

cur\_right is the leftmost occurrence of zero we have encountered. For the sake of simplicity, we don't decrement cur\_right by one (like how we increment cur\_left) so we can compute the area of the rectangle with height[j] \* (right[j] - left[j]) instead of height[j] \* (right[j] + 1 - left[j]).

This means that technically the base of the rectangle is defined by the half-open interval [l, r) instead of the closed interval [l, r], and right is really one greater than right boundary. Although the algorithm will still work if we don't do this with right, doing it this way makes the area calculation a little cleaner.

Note that to keep track of our cur\_right correctly, we must iterate from right to left, so this is what is done when updating right.

With our left, right, and height arrays appropriately updated, all that there is left to do is compute the area of each rectangle.

Since we know the bounds and height of rectangle j, we can trivially compute it's area with height[j] \* (right[j] - left[j]), and change our max\_area if we find that rectangle j's area is greater.

|  |
| --- |
| class Solution {  public int maximalRectangle(char[][] matrix) {  if (matrix.length == 0)  return 0;  int m = matrix.length;  int n = matrix[0].length;  int[] left = new int[n]; // initialize left as the leftmost boundary possible  int[] right = new int[n];  int[] height = new int[n];  Arrays.fill(right, n); // initialize right as the rightmost boundary possible  int maxarea = 0;  for (int i = 0; i < m; i++) {  int cur\_left = 0, cur\_right = n;  // update height  for (int j = 0; j < n; j++) {  if (matrix[i][j] == '1')  height[j]++;  else  height[j] = 0;  }  // update left  for (int j = 0; j < n; j++) {  if (matrix[i][j] == '1')  left[j] = Math.max(left[j], cur\_left);  else {  left[j] = 0;  cur\_left = j + 1;  }  }  // update right  for (int j = n - 1; j >= 0; j--) {  if (matrix[i][j] == '1')  right[j] = Math.min(right[j], cur\_right);  else {  right[j] = n;  cur\_right = j;  }  }  // update area  for (int j = 0; j < n; j++) {  maxarea = Math.max(maxarea, (right[j] - left[j]) \* height[j]);  }  }  return maxarea;  }  } |

The code and idea for the above solution originates from user [morrischen2008](https://leetcode.com/morrischen2008/).

**Complexity Analysis**

* Time complexity : O(NM)*O*(*NM*). In each iteration over N we iterate over M a constant number of times.
* Space complexity : O(M)*O*(*M*). M is the length of the additional arrays we keep.

**Tenth Line**

Given a text file file.txt, print just the 10th line of the file.

**Example:**

Assume that file.txt has the following content:

Line 1

Line 2

Line 3

Line 4

Line 5

Line 6

Line 7

Line 8

Line 9

Line 10

Your script should output the tenth line, which is:

Line 10

**Note:**  
1. If the file contains less than 10 lines, what should you output?  
2. There's at least three different solutions. Try to explore all possibilities.

**Add Digits**

Given a non-negative integer num, repeatedly add all its digits until the result has only one digit.

**Example:**

**Input:** 38

**Output:** 2

**Explanation:** The process is like: 3 + 8 = 11, 1 + 1 = 2.

  Since 2 has only one digit, return it.

**Follow up:**  
Could you do it without any loop/recursion in O(1) runtime?

   Hide Hint #1

A naive implementation of the above process is trivial. Could you come up with other methods?

   Hide Hint #2

What are all the possible results?

   Hide Hint #3

How do they occur, periodically or randomly?

   Hide Hint #4

You may find this [Wikipedia article](https://en.wikipedia.org/wiki/Digital_root) useful.

#### **Overview**

The value we're asked to compute is the so-called [Digital Root](https://en.wikipedia.org/wiki/Digital_root). Logarithmic time solution is easy to write, although the main question here is how to fit into a constant time.

|  |
| --- |
| class Solution {  public int addDigits(int num) {  int digitalRoot = 0;  while (num > 0) {  digitalRoot += num % 10;  num = num / 10;    if (num == 0 && digitalRoot > 9) {  num = digitalRoot;  digitalRoot = 0;  }  }  return digitalRoot;  }  } |

#### **Approach 1: Mathematical: Digital Root**

**Formula for the Digital Root**

There is a known formula to compute a digital root in a decimal numeral system

dr\_{10}(n) = 0, \qquad \text{if } n = 0*dr*10​(*n*)=0,if *n*=0

dr\_{10}(n) = 9, \qquad \text{if } n = 9 k*dr*10​(*n*)=9,if *n*=9*k*

dr\_{10}(n) = n \mod 9, \qquad \text{if } n \ne 9 k*dr*10​(*n*)=*n*mod9,if *n*​=9*k*

How to derive it? Probably, you already know the following proof from school, where it was used for a divisibility by 9: "The original number is divisible by 9 if and only if the sum of its digits is divisible by 9". Let's revise it briefly.

The input number could be presented in a standard way, where d\_0, d\_1, .. d\_k*d*0​,*d*1​,..*dk*​ are digits of n:

n = d\_0 + d\_1 \cdot 10^1 + d\_2 \cdot 10^2 + ... + d\_k \cdot 10^k*n*=*d*0​+*d*1​⋅101+*d*2​⋅102+...+*dk*​⋅10*k*

One could expand each power of ten, using the following:

10 = 9 \cdot 1 + 1 \\ 100 = 99 + 1 = 9 \cdot 11 + 1 \\ 1000 = 999 + 1 = 9 \cdot 111 + 1 \\ ... \\ 10^k = 1\underbrace{00..0}\_\text{k times} = \underbrace{99..9}\_\text{k times} + 1 = 9 \cdot \underbrace{11..1}\_\text{k times} + 110=9⋅1+1100=99+1=9⋅11+11000=999+1=9⋅111+1...10*k*=1k times00..0​​=k times99..9​​+1=9⋅k times11..1​​+1

That results in

n = d\_0 + d\_1 \cdot (9 \cdot 1 + 1) + d\_2 \cdot(9 \cdot 11 + 1) + ... + d\_k \cdot (9 \cdot \underbrace{11..1}\_\text{k times} + 1)*n*=*d*0​+*d*1​⋅(9⋅1+1)+*d*2​⋅(9⋅11+1)+...+*dk*​⋅(9⋅k times11..1​​+1)

and could be simplified as

n = (d\_0 + d\_1 + d\_2 + ... + d\_k) + 9 \cdot (d\_1 \cdot 1 + d\_2 \cdot 11 + ... + d\_k \cdot \underbrace{11..1}\_\text{k times})*n*=(*d*0​+*d*1​+*d*2​+...+*dk*​)+9⋅(*d*1​⋅1+*d*2​⋅11+...+*dk*​⋅k times11..1​​)

The last step is to take the modulo from both sides:

n \mod 9 = (d\_0 + d\_1 + d\_2 + ... + d\_k) \mod 9*n*mod9=(*d*0​+*d*1​+*d*2​+...+*dk*​)mod9

and to consider separately three cases: the sum of digits is 0, the sum of digits is divisible by 9, and the sum of digits is not divisible by nine:

dr\_{10}(n) = 0, \qquad \text{if } n = 0*dr*10​(*n*)=0,if *n*=0

dr\_{10}(n) = 9, \qquad \text{if } n = 9 k*dr*10​(*n*)=9,if *n*=9*k*

dr\_{10}(n) = n \mod 9, \qquad \text{if } n \ne 9 k*dr*10​(*n*)=*n*mod9,if *n*​=9*k*

**Implementation**

The straightforward implementation is

|  |
| --- |
| class Solution {  public int addDigits(int num) {  if (num == 0) return 0;  if (num % 9 == 0) return 9;  return num % 9;  }  } |

though two last cases could be merged into one

dr\_{10}(n) = 0, \qquad \text{if } n = 0*dr*10​(*n*)=0,if *n*=0

dr\_{10}(n) = 1 + (n - 1) \mod 9, \qquad \text{if } n \ne 0*dr*10​(*n*)=1+(*n*−1)mod9,if *n*​=0

|  |
| --- |
| class Solution {  public int addDigits(int num) {  return num == 0 ? 0 : 1 + (num - 1) % 9;  }  } |

**Complexity Analysis**

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

**Nth Digit**

Find the *n*th digit of the infinite integer sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

**Note:**  
*n* is positive and will fit within the range of a 32-bit signed integer (*n* < 231).

**Example 1:**

**Input:**

3

**Output:**

3

**Example 2:**

**Input:**

11

**Output:**

0

**Explanation:**

The 11th digit of the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ... is a 0, which is part of the numbe

**Encode and Decode TinyURL**

Note: This is a companion problem to the [System Design](https://leetcode.com/discuss/interview-question/system-design/) problem: [Design TinyURL](https://leetcode.com/discuss/interview-question/124658/Design-a-URL-Shortener-(-TinyURL-)-System/" \t "_blank).

TinyURL is a URL shortening service where you enter a URL such as https://leetcode.com/problems/design-tinyurl and it returns a short URL such as http://tinyurl.com/4e9iAk.

Design the encode and decode methods for the TinyURL service. There is no restriction on how your encode/decode algorithm should work. You just need to ensure that a URL can be encoded to a tiny URL and the tiny URL can be decoded to the original URL.

## Solution

#### **Approach #1 Using Simple Counter[Accepted]**

In order to encode the URL, we make use of a counter(i*i*), which is incremented for every new URL encountered. We put the URL along with its encoded count(i*i*) in a HashMap. This way we can retrieve it later at the time of decoding easily.

|  |
| --- |
| public class Codec {  Map<Integer, String> map = new HashMap<>();  int i = 0;  public String encode(String longUrl) {  map.put(i, longUrl);  return "http://tinyurl.com/" + i++;  }  public String decode(String shortUrl) {  return map.get(Integer.parseInt(shortUrl.replace("http://tinyurl.com/", "")));  }  } |

**Performance Analysis**

* The range of URLs that can be decoded is limited by the range of \text{int}int.
* If excessively large number of URLs have to be encoded, after the range of \text{int}int is exceeded, integer overflow could lead to overwriting the previous URLs' encodings, leading to the performance degradation.
* The length of the URL isn't necessarily shorter than the incoming \text{longURL}longURL. It is only dependent on the relative order in which the URLs are encoded.
* One problem with this method is that it is very easy to predict the next code generated, since the pattern can be detected by generating a few encoded URLs.

#### **Approach #2 Variable-length Encoding[Accepted]**

**Algorithm**

In this case, we make use of variable length encoding to encode the given URLs. For every \text{longURL}longURL, we choose a variable codelength for the input URL, which can be any length between 0 and 61. Further, instead of using only numbers as the Base System for encoding the URLSs, we make use of a set of integers and alphabets to be used for encoding.

|  |
| --- |
| public class Codec {  String chars = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";  HashMap<String, String> map = new HashMap<>();  int count = 1;  public String getString() {  int c = count;  StringBuilder sb = new StringBuilder();  while (c > 0) {  c--;  sb.append(chars.charAt(c % 62));  c /= 62;  }  return sb.toString();  }  public String encode(String longUrl) {  String key = getString();  map.put(key, longUrl);  return "http://tinyurl.com/" + key;  count++;  }  public String decode(String shortUrl) {  return map.get(shortUrl.replace("http://tinyurl.com/", ""));  }  } |

**Performance Analysis**

* The number of URLs that can be encoded is, again, dependent on the range of \text{int}int, since, the same count*count* will be generated after overflow of integers.
* The length of the encoded URLs isn't necessarily short, but is to some extent dependent on the order in which the incoming \text{longURL}longURL's are encountered. For example, the codes generated will have the lengths in the following order: 1(62 times), 2(62 times) and so on.
* The performance is quite good, since the same code will be repeated only after the integer overflow limit, which is quite large.
* In this case also, the next code generated could be predicted by the use of some calculations.

#### **Approach #3 Using hashcode[Accepted]**

**Algorithm**

In this method, we make use of an inbuilt function \text{hashCode()}hashCode() to determine a code for mapping every URL. Again, the mapping is stored in a HashMap for decoding.

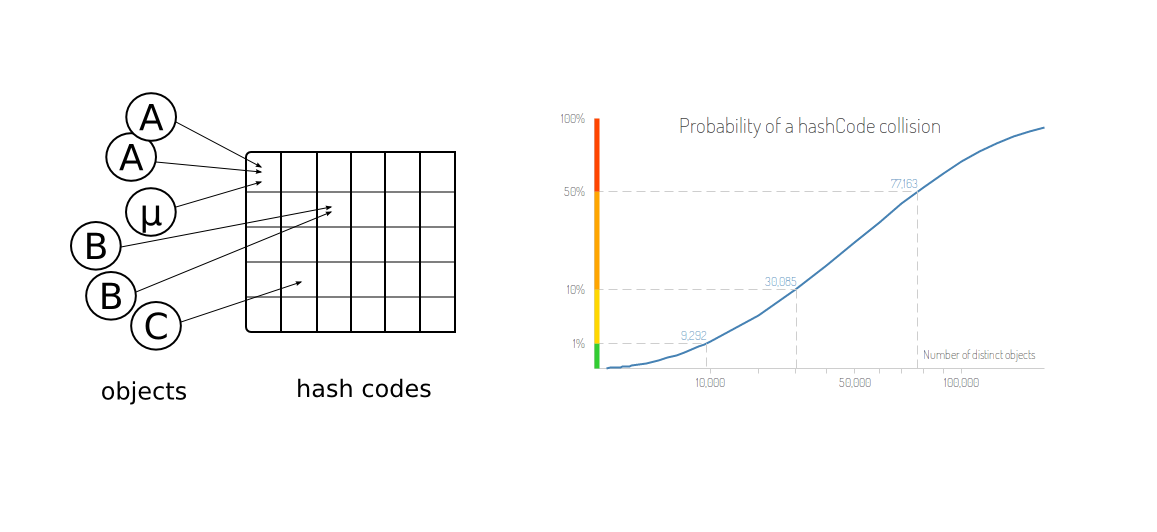
The hash code for a String object is computed(using int arithmetic) as −

s[0]\*31^{(n - 1)} + s[1]\*31^{(n - 2)} + ... + s[n - 1]*s*[0]∗31(*n*−1)+*s*[1]∗31(*n*−2)+...+*s*[*n*−1] , where s[i] is the ith character of the string, n is the length of the string.

|  |
| --- |
| public class Codec {  Map<Integer, String> map = new HashMap<>();  public String encode(String longUrl) {  map.put(longUrl.hashCode(), longUrl);  return "http://tinyurl.com/" + longUrl.hashCode();  }  public String decode(String shortUrl) {  return map.get(Integer.parseInt(shortUrl.replace("http://tinyurl.com/", "")));  }  } |

**Performance Analysis**

* The number of URLs that can be encoded is limited by the range of \text{int}int, since \text{hashCode}hashCode uses integer calculations.
* The average length of the encoded URL isn't directly related to the incoming \text{longURL}longURL length.
* The \text{hashCode()}hashCode() doesn't generate unique codes for different string. This property of getting the same code for two different inputs is called collision. Thus, as the number of encoded URLs increases, the probability of collisions increases, which leads to failure.
* The following figure demonstrates the mapping of different objects to the same hashcode and the increasing probability of collisions with increasing number of objects.



* Thus, it isn't necessary that the collisions start occuring only after a certain number of strings have been encoded, but they could occur way before the limit is even near to the \text{int}int. This is similar to birthday paradox i.e. the probability of two people having the same birthday is nearly 50% if we consider only 23 people and 99.9% with just 70 people.
* Predicting the encoded URL isn't easy in this scheme.

#### **Approach #4 Using random number[Accepted]**

**Algorithm**

In this case, we generate a random integer to be used as the code. In case the generated code happens to be already mapped to some previous \text{longURL}longURL, we generate a new random integer to be used as the code. The data is again stored in a HashMap to help in the decoding process.

|  |
| --- |
| public class Codec {  Map<Integer, String> map = new HashMap<>();  Random r = new Random();  int key = r.nextInt(Integer.MAX\_VALUE);  public String encode(String longUrl) {  while (map.containsKey(key)) {  key = r.nextInt(Integer.MAX\_VALUE);  }  map.put(key, longUrl);  return "http://tinyurl.com/" + key;  }  public String decode(String shortUrl) {  return map.get(Integer.parseInt(shortUrl.replace("http://tinyurl.com/", "")));  }  } |

\*\*Performance Analysis\*\*

* The number of URLs that can be encoded is limited by the range of \text{int}int.
* The average length of the codes generated is independent of the \text{longURL}longURL's length, since a random integer is used.
* The length of the URL isn't necessarily shorter than the incoming \text{longURL}longURL. It is only dependent on the relative order in which the URLs are encoded.
* Since a random number is used for coding, again, as in the previous case, the number of collisions could increase with the increasing number of input strings, leading to performance degradation.
* Determining the encoded URL isn't possible in this scheme, since we make use of random numbers.

#### **Approach #5 Random fixed-length encoding[Accepted]**

**Algorithm**

In this case, again, we make use of the set of numbers and alphabets to generate the coding for the given URLs, similar to Approach 2. But in this case, the length of the code is fixed to 6 only. Further, random characters from the string to form the characters of the code. In case, the code generated collides with some previously generated code, we form a new random code.

|  |
| --- |
| public class Codec {  String alphabet = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";  HashMap<String, String> map = new HashMap<>();  Random rand = new Random();  String key = getRand();  public String getRand() {  StringBuilder sb = new StringBuilder();  for (int i = 0; i < 6; i++) {  sb.append(alphabet.charAt(rand.nextInt(62)));  }  return sb.toString();  }  public String encode(String longUrl) {  while (map.containsKey(key)) {  key = getRand();  }  map.put(key, longUrl);  return "http://tinyurl.com/" + key;  }  public String decode(String shortUrl) {  return map.get(shortUrl.replace("http://tinyurl.com/", ""));  }  } |

**Performance Analysis**

* The number of URLs that can be encoded is quite large in this case, nearly of the order (10+26\*2)^6(10+26∗2)6.
* The length of the encoded URLs is fixed to 6 units, which is a significant reduction for very large URLs.
* The performance of this scheme is quite good, due to a very less probability of repeated same codes generated.
* We can increase the number of encodings possible as well, by increasing the length of the encoded strings. Thus, there exists a tradeoff between the length of the code and the number of encodings possible.
* Predicting the encoding isn't possible in this scheme since random numbers are used.

**Rectangle Overlap**

An axis-aligned rectangle is represented as a list [x1, y1, x2, y2], where (x1, y1) is the coordinate of its bottom-left corner, and (x2, y2) is the coordinate of its top-right corner. Its top and bottom edges are parallel to the X-axis, and its left and right edges are parallel to the Y-axis.

Two rectangles overlap if the area of their intersection is **positive**. To be clear, two rectangles that only touch at the corner or edges do not overlap.

Given two axis-aligned rectangles rec1 and rec2, return true*if they overlap, otherwise return*false.

**Example 1:**

**Input:** rec1 = [0,0,2,2], rec2 = [1,1,3,3]

**Output:** true

**Example 2:**

**Input:** rec1 = [0,0,1,1], rec2 = [1,0,2,1]

**Output:** false

**Example 3:**

**Input:** rec1 = [0,0,1,1], rec2 = [2,2,3,3]

**Output:** false

**Constraints:**

* rect1.length == 4
* rect2.length == 4
* -109 <= rec1[i], rec2[i] <= 109
* rec1[0] <= rec1[2] and rec1[1] <= rec1[3]
* rec2[0] <= rec2[2] and rec2[1] <= rec2[3]

#### **Approach #1: Check Position [Accepted]**

**Intuition**

If the rectangles do not overlap, then rec1 must either be higher, lower, to the left, or to the right of rec2.

**Algorithm**

The answer for whether they don't overlap is LEFT OR RIGHT OR UP OR DOWN, where OR is the logical OR, and LEFT is a boolean that represents whether rec1 is to the left of rec2. The answer for whether they do overlap is the negation of this.

The condition "rec1 is to the left of rec2" is rec1[2] <= rec2[0], that is the right-most x-coordinate of rec1 is left of the left-most x-coordinate of rec2. The other cases are similar.

Note: we should also check if either of the rectangle is actually a line. If this is the case, then we cannot have any positive overlapping according to the definition.

|  |
| --- |
| class Solution {  public boolean isRectangleOverlap(int[] rec1, int[] rec2) {  // check if either rectangle is actually a line  if (rec1[0] == rec1[2] || rec1[1] == rec1[3] ||  rec2[0] == rec2[2] || rec2[1] == rec2[3]) {  // the line cannot have positive overlap  return false;  }  return !(rec1[2] <= rec2[0] || // left  rec1[3] <= rec2[1] || // bottom  rec1[0] >= rec2[2] || // right  rec1[1] >= rec2[3]); // top  }  } |

**Complexity Analysis**

* Time and Space Complexity: O(1)*O*(1).

#### **Approach #2: Check Area [Accepted]**

**Intuition**

If the rectangles overlap, they have positive area. This area must be a rectangle where both dimensions are positive, since the boundaries of the intersection are axis aligned.

Thus, we can reduce the problem to the one-dimensional problem of determining whether two line segments overlap.

**Algorithm**

Say the area of the intersection is width \* height, where width is the intersection of the rectangles projected onto the x-axis, and height is the same for the y-axis. We want both quantities to be positive.

The width is positive when min(rec1[2], rec2[2]) > max(rec1[0], rec2[0]), that is when the smaller of (the largest x-coordinates) is larger than the larger of (the smallest x-coordinates). The height is similar.

|  |
| --- |
| class Solution {  public boolean isRectangleOverlap(int[] rec1, int[] rec2) {  return (Math.min(rec1[2], rec2[2]) > Math.max(rec1[0], rec2[0]) && // width > 0  Math.min(rec1[3], rec2[3]) > Math.max(rec1[1], rec2[1])); // height > 0  }  } |

**Complexity Analysis**

* Time and Space Complexity: O(1)*O*(1).

**Department Top Three Salaries**

The Employee table holds all employees. Every employee has an Id, and there is also a column for the department Id.

+----+-------+--------+--------------+

| Id | Name | Salary | DepartmentId |

+----+-------+--------+--------------+

| 1 | Joe | 85000 | 1 |

| 2 | Henry | 80000 | 2 |

| 3 | Sam | 60000 | 2 |

| 4 | Max | 90000 | 1 |

| 5 | Janet | 69000 | 1 |

| 6 | Randy | 85000 | 1 |

| 7 | Will | 70000 | 1 |

+----+-------+--------+--------------+

The Department table holds all departments of the company.

+----+----------+

| Id | Name |

+----+----------+

| 1 | IT |

| 2 | Sales |

+----+----------+

Write a SQL query to find employees who earn the top three salaries in each of the department. For the above tables, your SQL query should return the following rows (order of rows does not matter).

+------------+----------+--------+

| Department | Employee | Salary |

+------------+----------+--------+

| IT | Max | 90000 |

| IT | Randy | 85000 |

| IT | Joe | 85000 |

| IT | Will | 70000 |

| Sales | Henry | 80000 |

| Sales | Sam | 60000 |

+------------+----------+--------+

**Explanation:**

In IT department, Max earns the highest salary, both Randy and Joe earn the second highest salary, and Will earns the third highest salary. There are only two employees in the Sales department, Henry earns the highest salary while Sam earns the second highest salary.

#### **Approach: Using JOIN and sub-query [Accepted]**

**Algorithm**

A top 3 salary in this company means there is no more than 3 salary bigger than itself in the company.

select e1.Name as 'Employee', e1.Salary

from Employee e1

where 3 >

(

select count(distinct e2.Salary)

from Employee e2

where e2.Salary > e1.Salary

)

;

In this code, we count the salary number of which is bigger than e1.Salary. So the output is as below for the sample data.

| Employee | Salary |

|----------|--------|

| Henry | 80000 |

| Max | 90000 |

| Randy | 85000 |

Then, we need to join the **Employee** table with **Department** in order to retrieve the department information.

**MySQL**

SELECT

d.Name AS 'Department', e1.Name AS 'Employee', e1.Salary

FROM

Employee e1

JOIN

Department d ON e1.DepartmentId = d.Id

WHERE

3 > (SELECT

COUNT(DISTINCT e2.Salary)

FROM

Employee e2

WHERE

e2.Salary > e1.Salary

AND e1.DepartmentId = e2.DepartmentId

)

;

| Department | Employee | Salary |

|------------|----------|--------|

| IT | Joe | 70000 |

| Sales | Henry | 80000 |

| Sales | Sam | 60000 |

| IT | Max | 90000 |

| IT | Randy | 85000 |

**Big Countries**

There is a table World

+-----------------+------------+------------+--------------+---------------+

| name | continent | area | population | gdp |

+-----------------+------------+------------+--------------+---------------+

| Afghanistan | Asia | 652230 | 25500100 | 20343000 |

| Albania | Europe | 28748 | 2831741 | 12960000 |

| Algeria | Africa | 2381741 | 37100000 | 188681000 |

| Andorra | Europe | 468 | 78115 | 3712000 |

| Angola | Africa | 1246700 | 20609294 | 100990000 |

+-----------------+------------+------------+--------------+---------------+

A country is big if it has an area of bigger than 3 million square km or a population of more than 25 million.

Write a SQL solution to output big countries' name, population and area.

For example, according to the above table, we should output:

+--------------+-------------+--------------+

| name | population | area |

+--------------+-------------+--------------+

| Afghanistan | 25500100 | 652230 |

| Algeria | 37100000 | 2381741 |

+--------------+-------------+--------------+

## Solution

#### **Approach I: Using WHERE clause and OR [Accepted]**

**Intuition**

Use WHERE clause in SQL to filter these records and get the target countries.

**Algorithm**

According to the definition, a big country meets at least one of the following two conditions:

1. It has an area of bigger than 3 million square km.
2. It has a population of more than 25 million.

So for the first condition, we can use the following code to get the big countries of this type.

SELECT name, population, area FROM world WHERE area > 3000000

In addition, we can use below code to get big countries of more than 25 million people.

SELECT name, population, area FROM world WHERE population > 25000000

As most people may already come into mind, we can use OR to combine these two solutions for the two sub-problems together.

**MySQL**

SELECT

name, population, area

FROM

world

WHERE

area > 3000000 OR population > 25000000

;

#### **Approach II: Using WHERE clause and UNION [Accepted]**

**Algorithm**

The idea of this approach is the same as the first one. However, we use UNION instead of OR.

**MySQL**

SELECT

name, population, area

FROM

world

WHERE

area > 3000000

UNION

SELECT

name, population, area

FROM

world

WHERE

population > 25000000

;

Note: This solution runs a little bit faster than the first one. However, they do not have big difference.