1. Basic data structures in Java

Object (Interface)

├── Collection (Interface)

│ ├── List (Interface)

│ │ ├── **ArrayList:** get is quick, resizable

│ │ ├── **LinkedList:** add/delete first/last element is quick

│ │ ├── Vector (Out of date)

│ │ │ └── **Stack**

│ ├── Set (Interface)

│ │ ├── **HashSet**

│ │ ├── **LinkedHashSet**

│ │ ├── **TreeSet**

│ ├── Queue (Interface)

│ ├── Deque (Interface)

│ ├── **LinkedList**

│ ├── **ArrayDeque**

│ ├── **PriorityQueue**

├── Map (Interface)

├── **HashMap**

├── **TreeMap**

├── **LinkedHashMap**

├── **Hashtable**

* LIFO data structure (stack):

There is no built-in class in Java that strictly defines a stack class. Stack class in Java is a subclass of Vector, it has some methods that can access middle elements and do something that is not allowed in a stack. In interview, however, we use **Stack** class for simplicity and only use methods that is allowed in a stack structure.

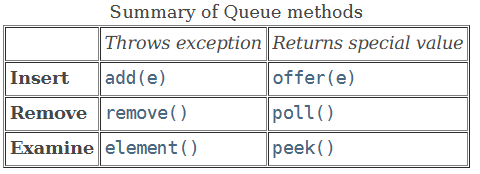
stack.push() stack.pop() stack.peek() stack.isEmpty()

* FIFO data structure (ArrayDeque):

We use this class to represent a FIFO queue when solving Leetcode problems. Similarly, there is no built-in class in Java that strictly defines a FIFO data structure.

Use **Queue** interface

queue.offer() queue.poll() queue.peek() queue.isEmpty()



DO NOT USE push or pop although ArrayDeque supports these methods. Use ArrayDeque as FIFO queue. – if we use Queue interface, we won’t be able to use push and pop.

* Double-ended queue (ArrayDeque):

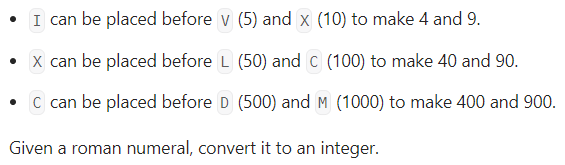
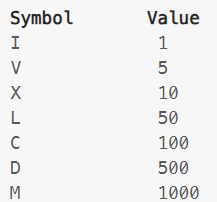
ArrayDeque is better than LinkedList because it generally provides better performance and lower memory overhead.

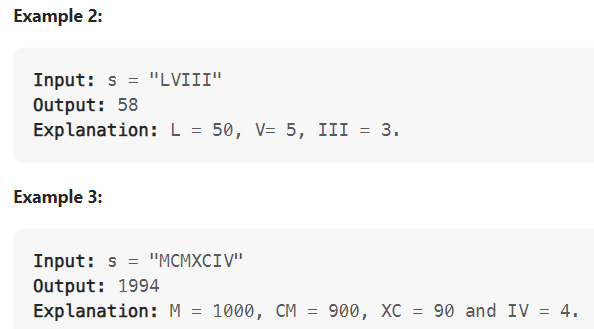
Use **Deque** interface

offerFirst() offerLast() peekFirst() peekLast() pollFirst() pollLast()

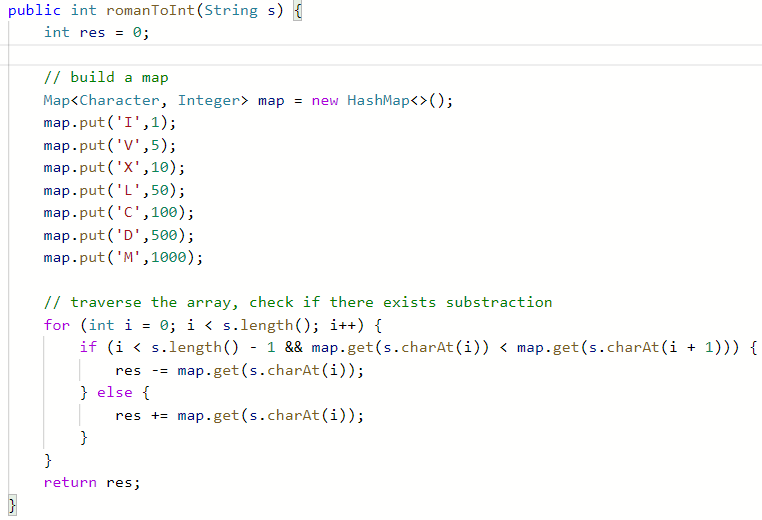
1. **String**

**Roman to Integer (Map)**

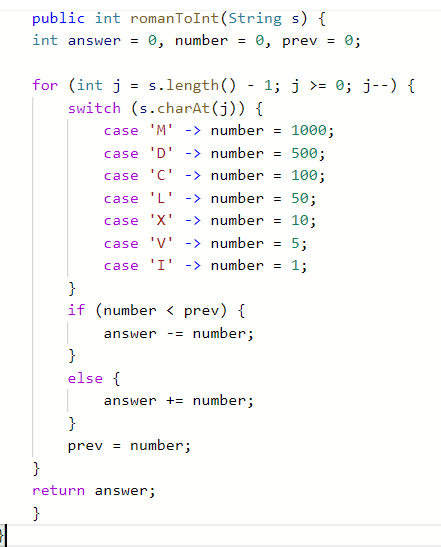




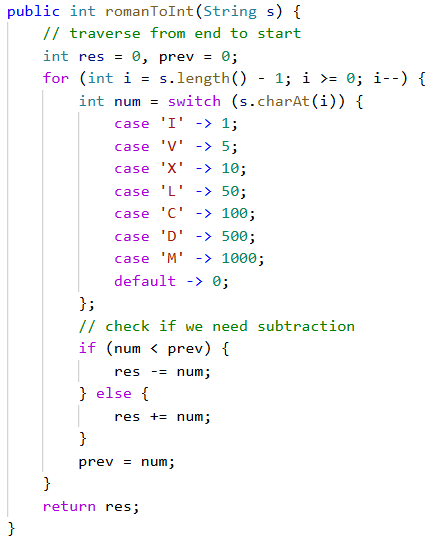
Solution 1: Using a map, traverse from beginning to end



Solution 2: Start from the end, compare current one and previous one, better runtime than Solution 1



because we assign number to 0, so we don’t need default here



in above code, we need default in switch because num doesn’t have an initial value

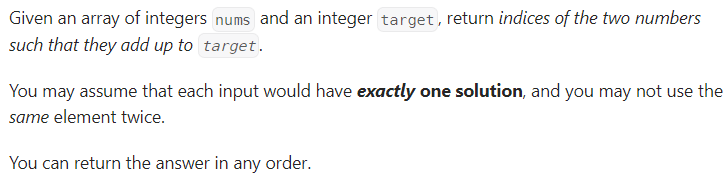
Tips:

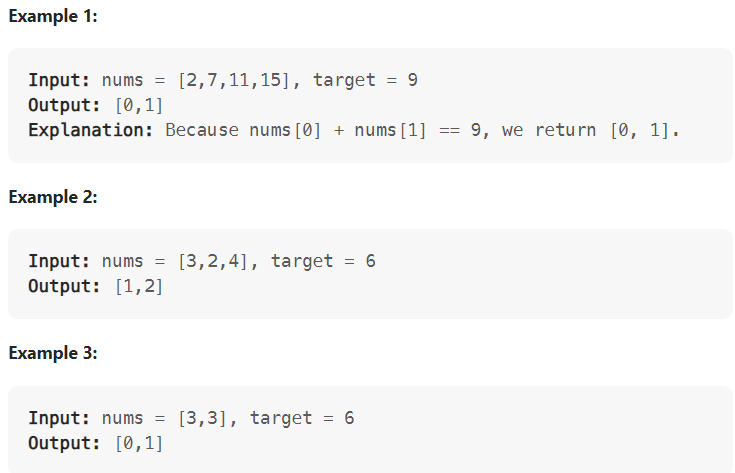
1. Character is denoted by ‘’, not “”
2. Choose character in a string, use s.charAt(i) function
3. After Java 12, switch can be used without break, as shown in Solution 2, we can also use more concise format
4. Length, add

|  |  |  |
| --- | --- | --- |
|  | size/length | add |
| Array | a.length | cannot add |
| ArrayList | al.size() | al.add() |
| String | s.length() | / |

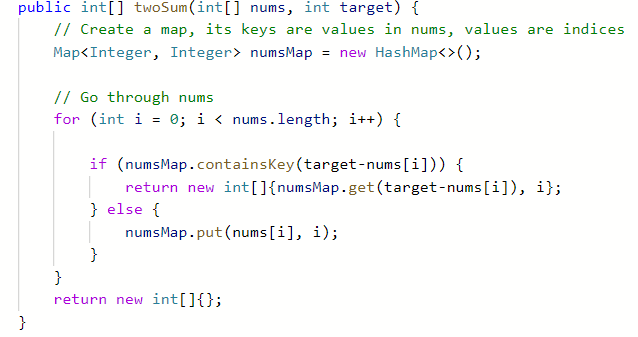
Java array has fixed length, once created, we can only assign values by index

**Two Sum (Map)**

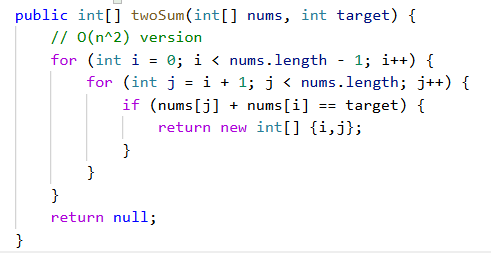




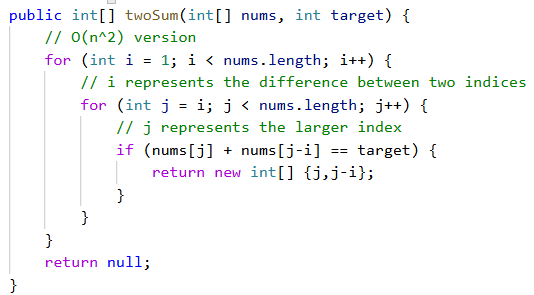
Solution 1: Traverse the array and build a map, time complexity O(n)



Solution 2: intuitive way, traverse the list, time complexity:



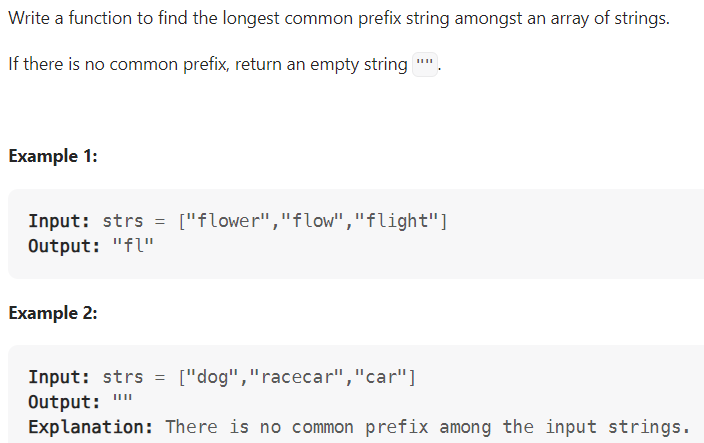
Solution 3: traverse different index interval, with faster runtime, time complexity:



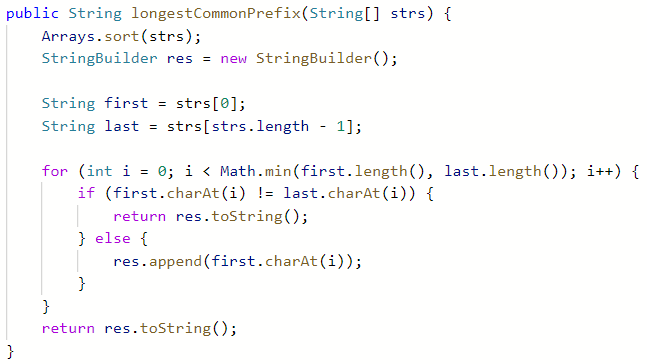
Tips:

1. map.get(key) can retrieve the corresponding value
2. map.containsKey(key) returns true if key exists in the map
3. map.containsValue(value) returns true if value exists in the map

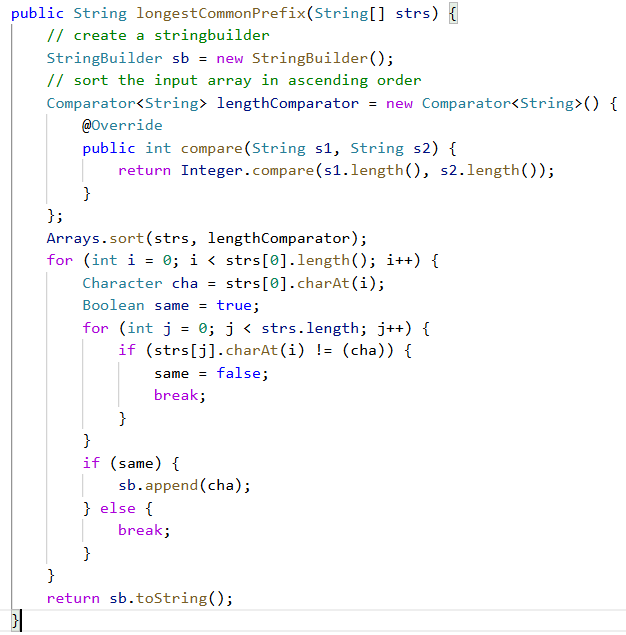
**Longest common prefix**



Solution 1: Sort the array, compare the first and last string, because the first and last are most different, we don’t have to compare the middle ones



Solution 2: intuitive way, sort the array in ascending order, the longest common prefix’s length is at most the length of first string; compare each character of first string with other strings’ corresponding character



Tips:

1. Sort an array, using Arrays.sort(); there is no built-in method to sort array in ascending length order, we need to create a new Comparator; Integer.compare(x,y) returns 0 if x equals y, a negative value if x < y, a positive value if x > y
2. Using StringBuilder to concatenate and modify strings is much faster than using +, because using a string class creates new strings in the heap memory, using StringBuilder does not create any extra objects in memory.
3. Creating variables to represent complex expression does not affect the running time and memory
4. Character cannot use equal method, use == or != instead
5. StringBuilder uses an array of characters internally, using toString method if we want String
6. StringBuilder has method delete(start, end) and append()

**Valid Parentheses (Stack)**

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

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

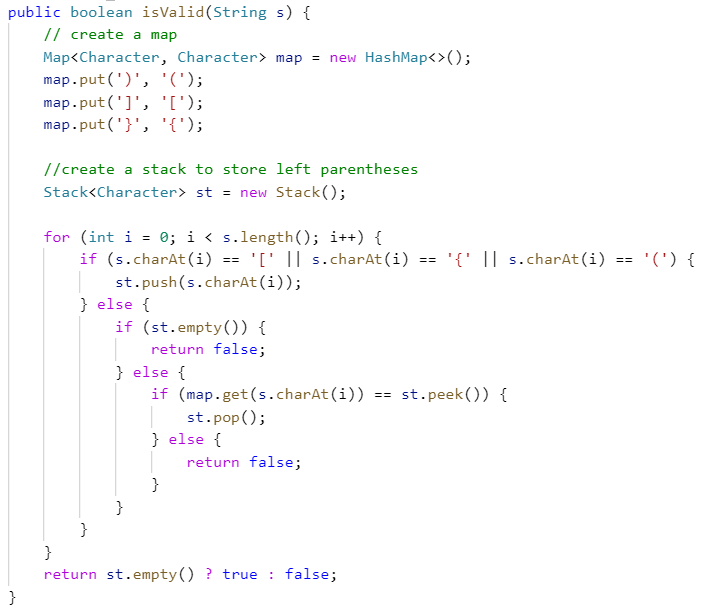
Solution 1: Use a stringbuilder to store left parentheses, when meeting right parentheses, get the last character and compare



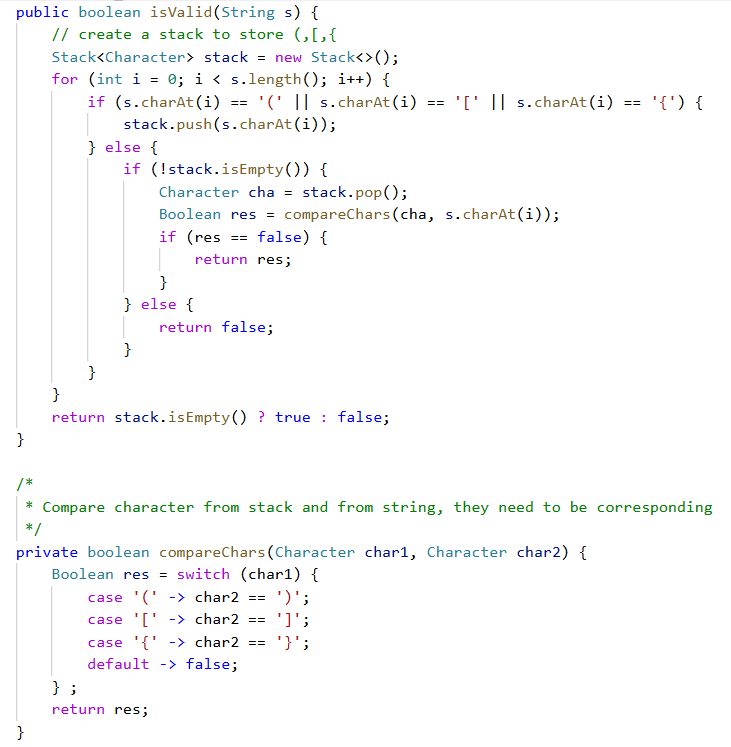
Bugs:

1. If we only have one character, such as “{” or “}“

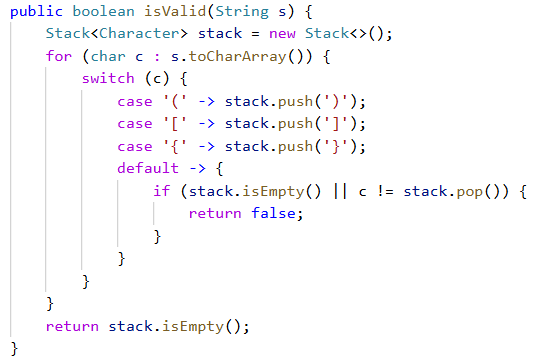
Solution 2: Use a stack to store left parentheses, use a map to find the right character



Solution 3: Use a stack to store left parentheses, if we meet right parenthesis, we pop the element from the stack and compare



Solution 4: Also use a stack, but push right parenthesis when encountering left parenthesis



Tips:

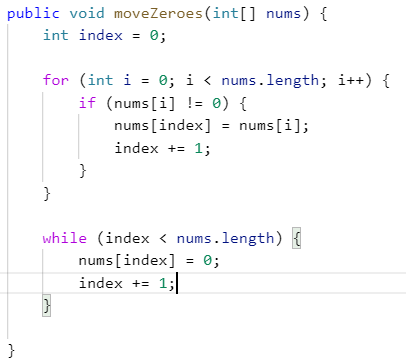
1. Stack, st.peek() does not change the stack, st.pop() delete the top element in the stack
2. st.empty() return true if the stack is empty; we prefer use **stack.isEmpty()** due to its consistency across different collection types and adherence to modern Java language
3. Difference between char and Character: char is a primitive type, Character is a wrapper class. char to Character: autoboxing; Character to char: unboxing. Both are built-in function in Java.
4. We can use str.toCharArray() method to convert a string to a char array
5. **Two pointers**

**Move zeros**

Given an integer array nums, move all 0's to the end of it while maintaining the relative order of the non-zero elements.

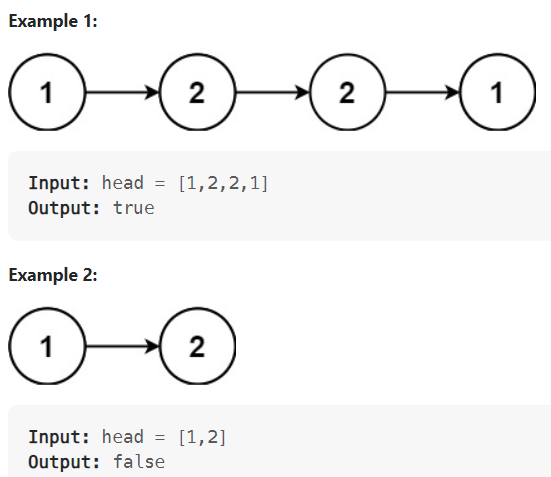
Note that you must do this in-place without making a copy of the array.

Solution 1: put all non-zero elements first, then assign others to 0



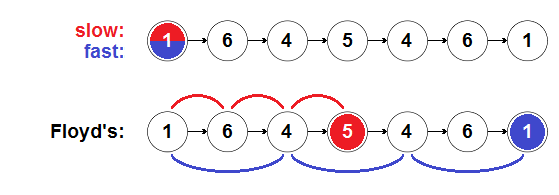
**Palindrome Linked list**

Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

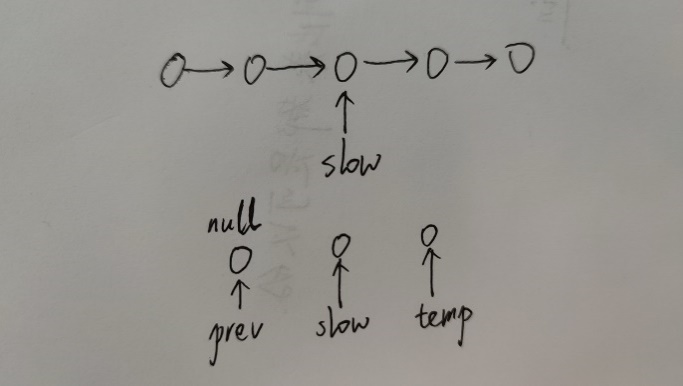


Solution 1:

1. traverse the list, find the middle one: use two pointers, a slow one and a fast one, a slow one moves one step in one iteration, a fast one moves two steps in one iteration, when the fast one reaches the end, the slow one is in the middle

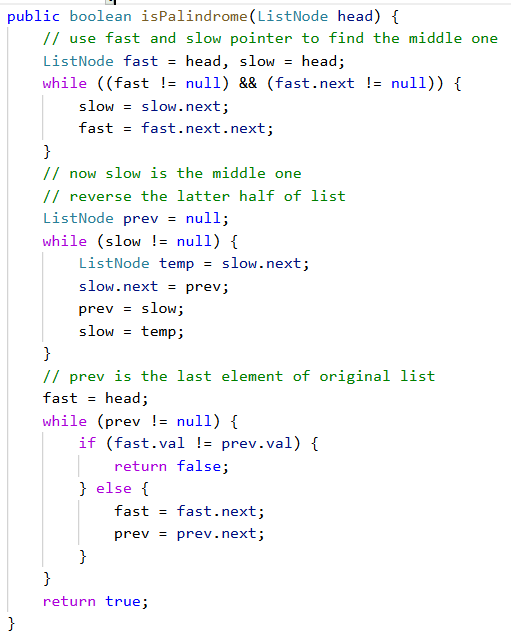


1. then change the latter half nodes’ next to its previous: 3 variables, prev, slow and temp; prev is assigned null, slow is middle node, temp is slow’s next node



when moving these 3 parameters, since we don’t care about temp, so we start from the leftmost one, set prev to slow, set slow to temp

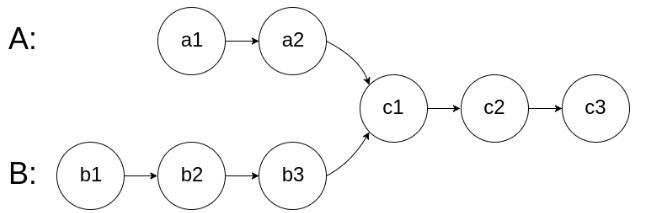
1. so we have two linked lists, then traverse from the start and the end of origin linked list and compare



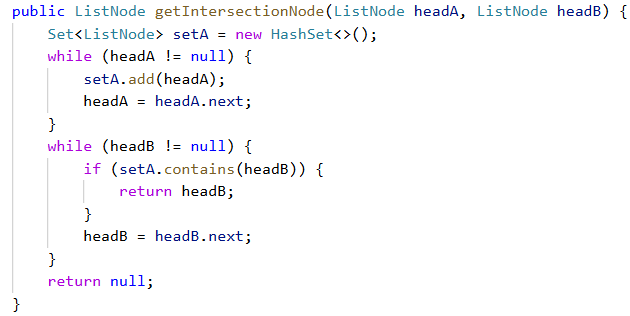
**Intersection of Two Linked Lists (Set)**

Given the heads of two singly linked-lists headA and headB, return the node at which the two lists intersect. If the two linked lists have no intersection at all, return null.

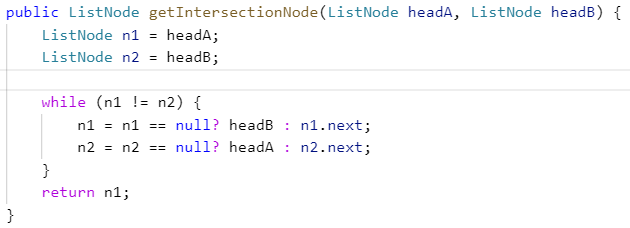
For example, the following two linked lists begin to intersect at node c1:



Solution 1: intuitive, use a set to store A’s element, and traverse B to determine whether there is an intersection, time complexity O(m+n)



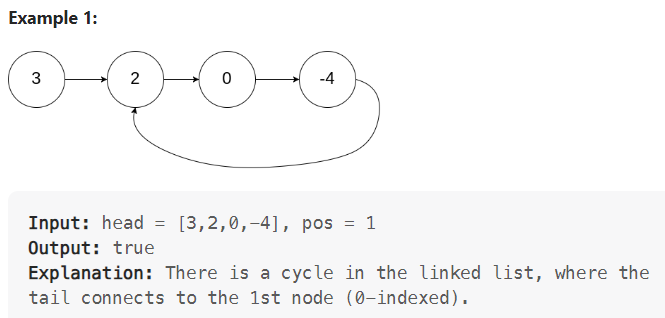
Solution 2: Use two pointers, starting at a1 and b1, traverse A and B and then B and A



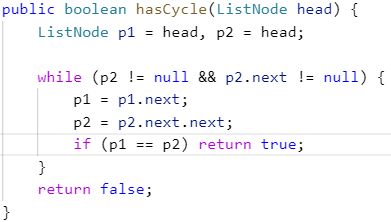
**Linked List Cycle**

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

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer.



Solution: Use two pointers, a slow one and a fast one. If the two pointers are the same, then there is a cycle



Tips:

1. **Short-circuit rule of &&, sequence matters**: if we use:

p2.next != null && p2 != null

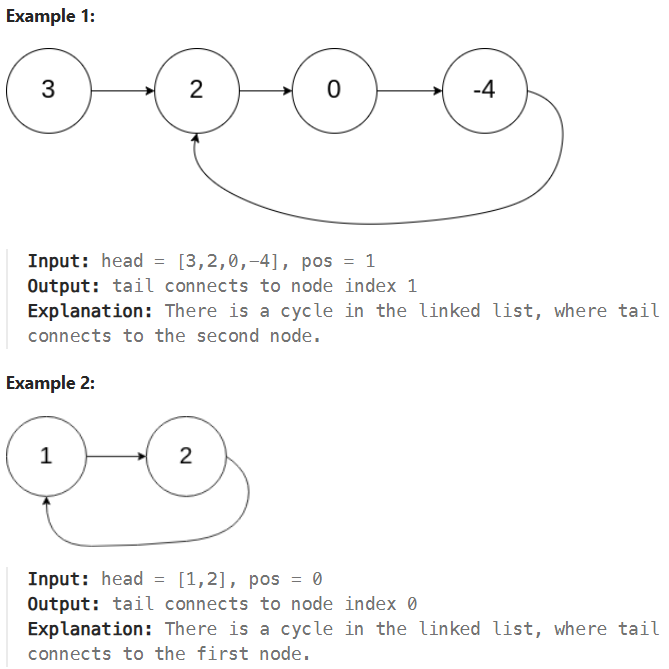
in while’s condition, we will have an error, because in a && clause, we calculate the first formula first, if it is false, then we don’t calculate the following ones

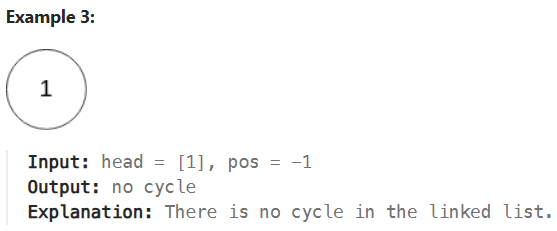
1. for reference types variables, == checks whether they are the same object

**Linked List Cycle II**

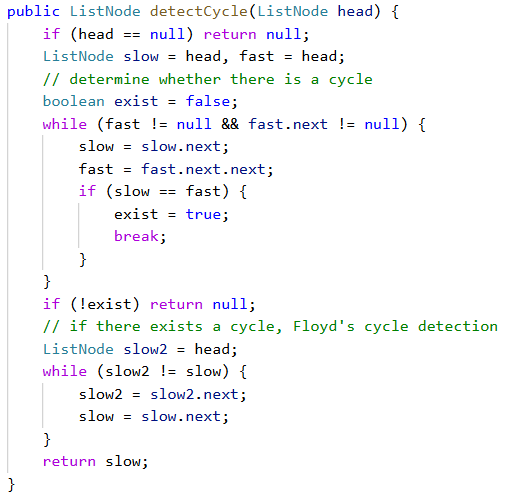
Given the head of a linked list, return the node where the cycle begins. If there is no cycle, return null.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter. Do not modify the linked list.





Solution: Floyd’s cycle detection algorithm. Note in while loop condition, we use fast != null first to avoid error.

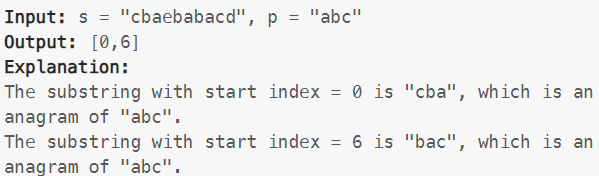


1. **Sliding Window**

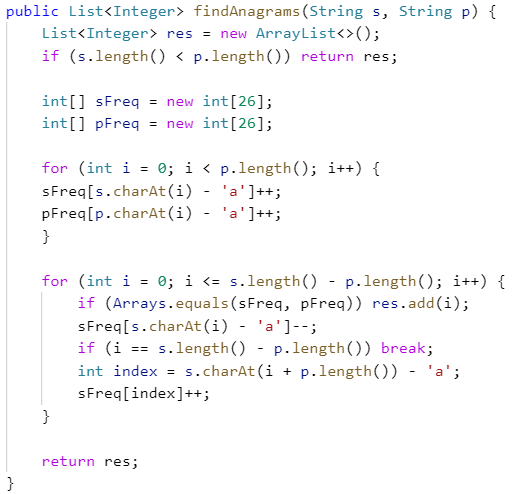
**M-Find all anagrams in a string**

Given two strings s and p, return an array of all the start indices of p's anagrams in s. You may return the answer in any order.

An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once. s and p consist of lowercase English letters.



Solution: Use a frequency array to represent p. Use another frequency array to represent sliding window in s.

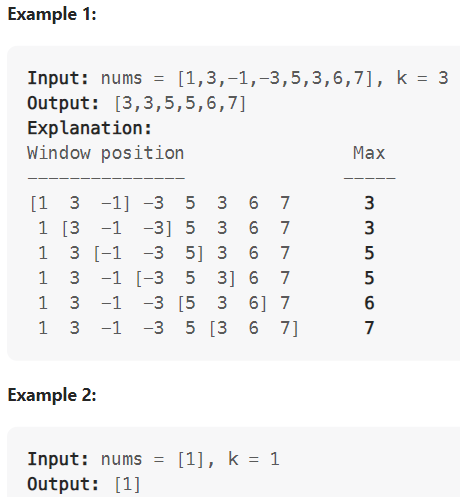


Tips:

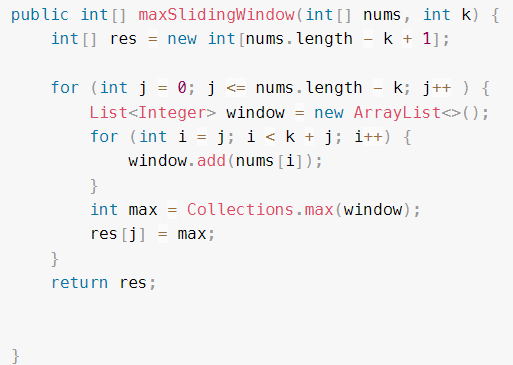
1. A char is actually stored as a number, so we can calculate chars as integer
2. I used a stringbuilder to store window, but it will cost too much time if the input is large
3. Check whether two arrays are equal, use Arrays.equals(array1, array2)

**H-Sliding window maximum**

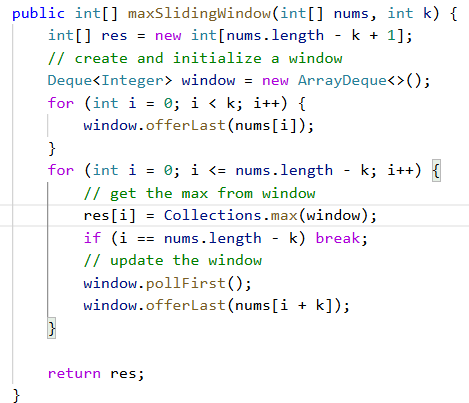
You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.



Solution 1: Build an arraylist every time, and find the max every time. This solution costs too much time when the input size is large.



Solution 2: Create a double-ended queue, use Collections.max() to find the maximum, update the window in each iteration



Collections.max() takes **O(k)** time, when k is very large (50000), I exceeded time limit. But still a suboptimal solution. It is a simple loop through all elements to find max.

Solution 3: Use a data structure called deque. Build an index deque to store the indices of current window.



Inside the for loop, the first code block is **a while loop rather than an if block**, so we check the index queue repeatedly to guarantee the first element is the largest.

if (i >= k-1) res[j++] ... this line means we use the origin j value first and plus 1 to j

Tips:

1. Deque is a new data structure. If q is a deque, then

|  |  |
| --- | --- |
| Action | Method |
| add element to the end | q.offerLast()  q.addLast() |
| add element to the start | q.offerFirst()  q.addFirst() |
| get element from the end | q.peekLast()  q.getLast() |
| get element from the start | q.peekFirst()  q.getFirst() |
| get element from the start and remove it | q.pollFirst()  q.removeFirst() |
| get element from the end and remove it | q.pollLast()  q.removeLast() |

The methods in the first line return false or null if fail, the methods in the second line throw exceptions if fail.

Generally speaking, when preparing for Leetcode and interview, we prefer use first kind of methods because they are more robust, simple and readable.

1. Get max and min

For two integers, we use Math.max(a,b) to get the max/min of them

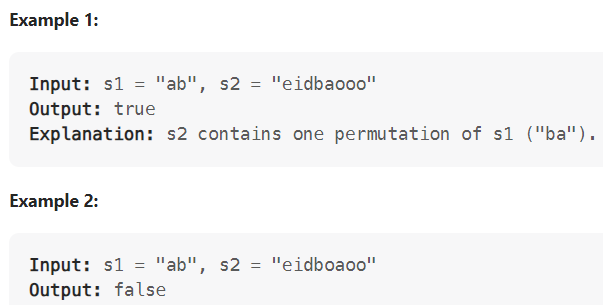
For an int[] array called nums, we use Arrays.stream(nums).max().getAsInt()

For a Collection data type called nums, such as ArrayList, ArrayDeque, we use Collections.max(nums)

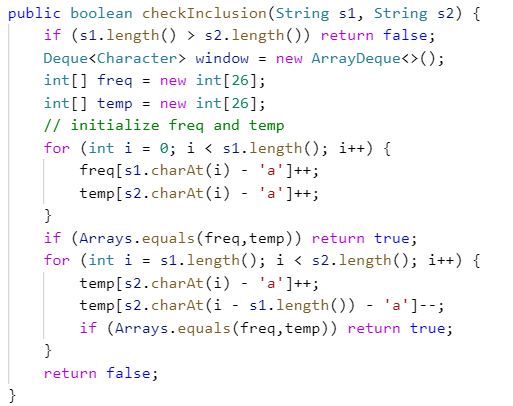
**M-Permutation in String**

Given two strings s1 and s2, return true if s2 contains a permutation of s1, or false otherwise.

In other words, return true if one of s1's permutations is the substring of s2.



Solution: Similar to 3.1, use two frequency array to determine whether the two strings are equal

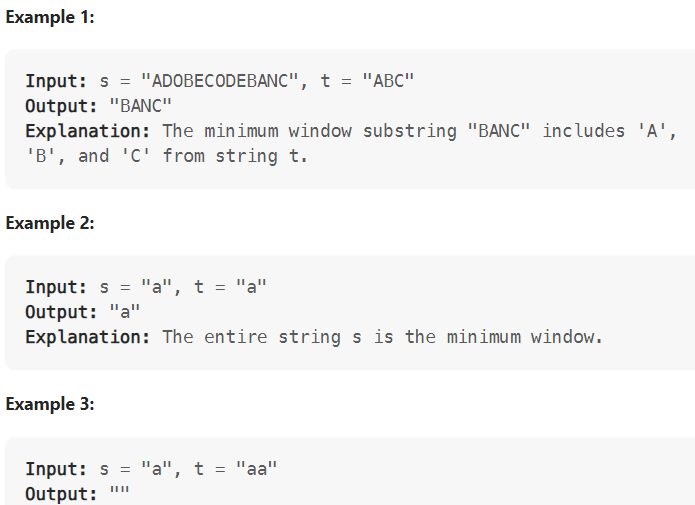


**H-Minium window substring**

Given two strings s and t of lengths m and n respectively, return the minimum window

substring of s such that every character in t (including duplicates) is included in the window. If there is no such substring, return the empty string "".

The testcases will be generated such that the answer is unique.



Solution 1: Use an array to represent window and String t, memory efficient compared to using a map. **Better solution** compared to Solution 2.



count is initialized as the length of String t

The most important data structure is **target array**. Although it is of length 128, but only a small part of it is used.

For chars in String t, the value could be positive, negative or zero.

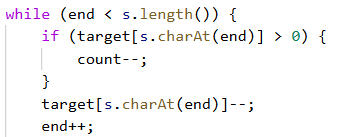
For chars not in String t, the value could be negative or zero.

**Positive** means the window still need this number of char to meet the requirement.

**Zero** means the window contains exact number of char.

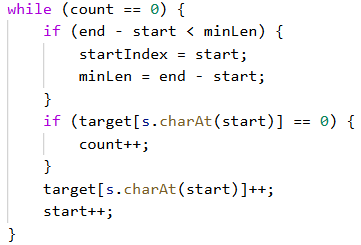
**Negative** means the window has more number of char than that in String t.

Analysis in while loop:



This part moves end pointer through the resource array. If we meet case like “aaaaabbcdd”, the target[‘a’] could be less than 0, if so, we don’t decrease count.

We start end from 0, then we will find the first window that meets our requirement.



When we find our solution, we move the start pointer to find a smaller window.

**Step 1**: First we update the minLen and startIndex. Then, we check the value of start char in target. There are two cases:

1. It is a char in String t. The value could be negative or zero.

Negative means the window contains more number of this char than needed, so we don’t increase count. But we need to update target array and move start pointer forward.

**Zero** means the window contains exact the number of this char. Then, since we have already recorded the information of this window, we can **increase count** and update target array and move start forward. Here we need to exit the while loop.

1. It is not a char in String t. The value must be negative.

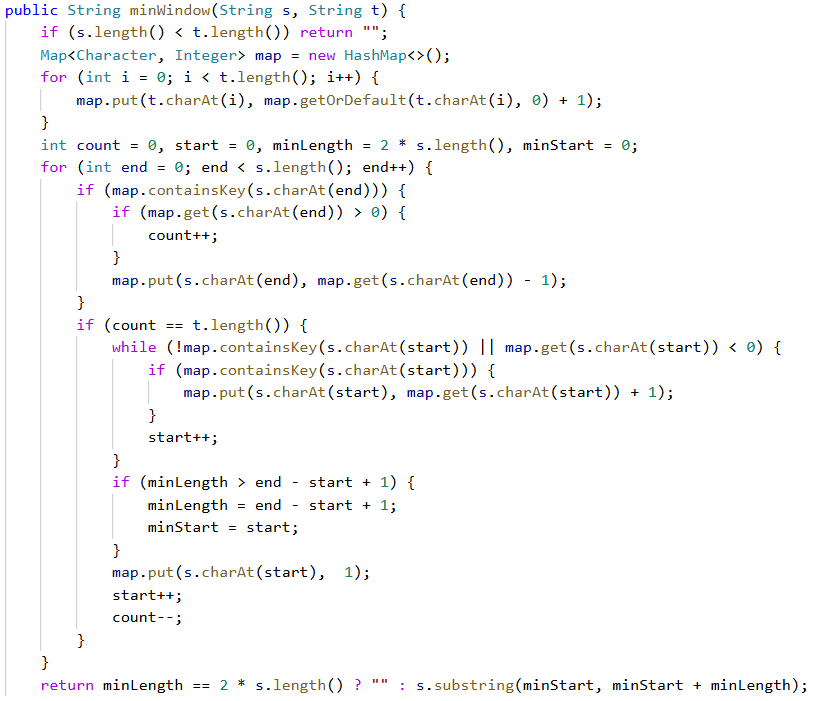
Because if it’s our first time to get a window, all chars not in String t have negative value. Then we update the target array and move start at once, so even one char changes from -1 to 0, our pointer has moved forward, **we will never trigger count++ due to a char not in String t**.

So, in this while loop, we always find a min window and move start pointer one step forward after we record the window information.

**Step 2**: check if the value is zero, if so, we change the count variable.

**Step 3**: update target array and move start forward regardless of whether the value is zero.

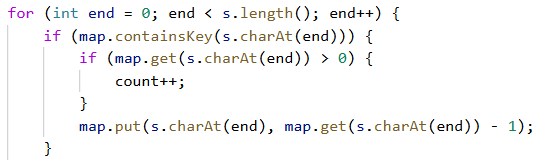
Solution 2: Use a map as key data structure



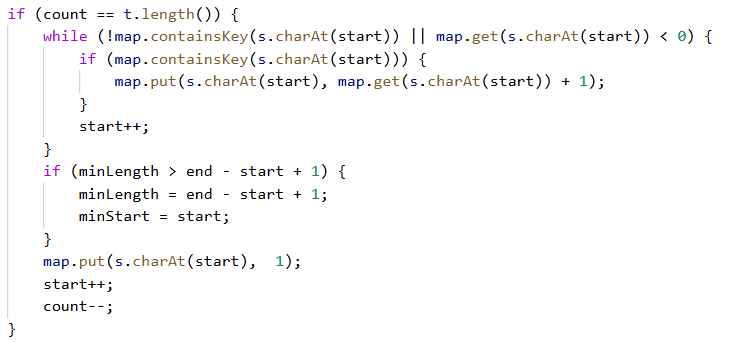
map only contains characters in String t

count is initialized as 0 to record whether we find a solution

value in map could be positive, negative or zero, initial state is all positive



this part moves end until count equals to t.length(), the value in map could be zero or negative. Zero means the window contains exact number of char, negative means the window contains more number of char than we need.



1. When we find a solution, we move the start pointer. The condition is the same as solution 1, if the char is not in map, or if it is in map but the value is negative, we move start forward. Besides, if it’s in map, we update the value.
2. Then we update the minStart and minLength.
3. When we finish the while loop, we know that we must meet a char in map, that value is 0, so we just need to set that to 1, and move start forward again. That’s the same with solution 1, where we update the target array and move start forward.
4. Finally we update count.

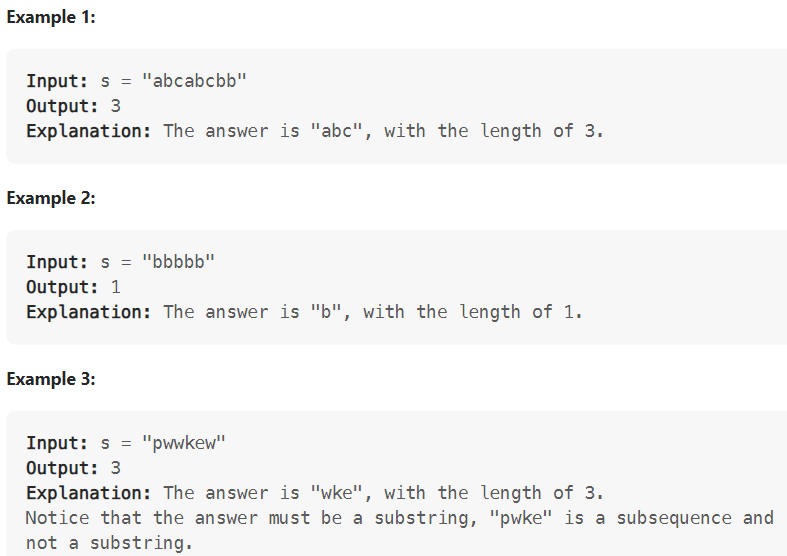
These four steps are slightly different with that in solution 1. In solution 1, we only use 3 steps.

Tips:

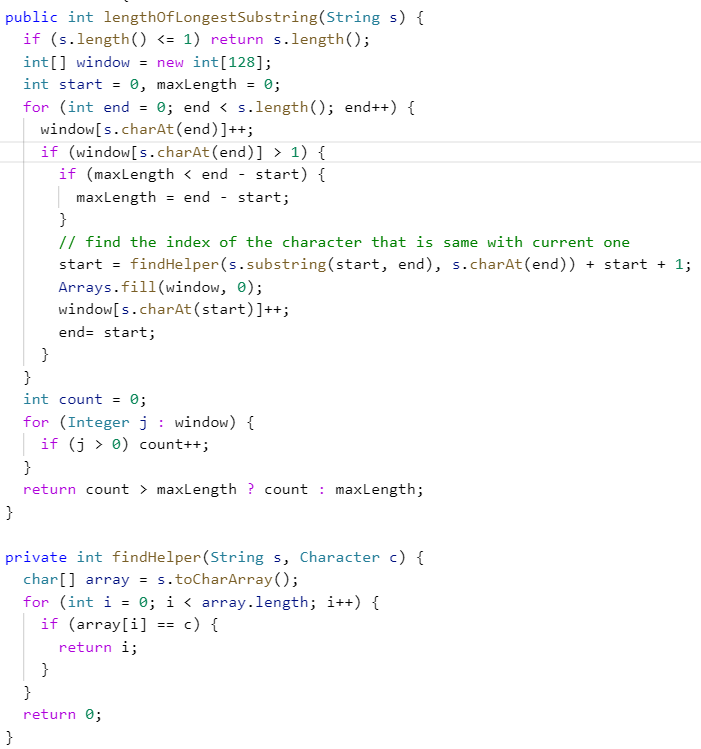
1. The characters A to Z range from 65 to 90, and lower case a to z 97 to 122. If the string contains both upper and lower case, we can use an array of 128 to represent them.
2. Map function getOrDefault(key, default) returns the key’s value or default value

**M-Longest substring without repeating characters**

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

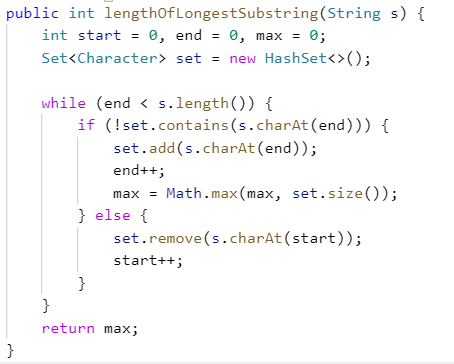


Solution 1: Use an array of 128 slots to represent the string. The value of the array should be 1 or 0. If the value is greater than 1, find the index of repeated character. Set start and end position accordingly.



This solution is not good because it changes end in the for loop, might be confusing

Solution 2: Use a set to store different strings



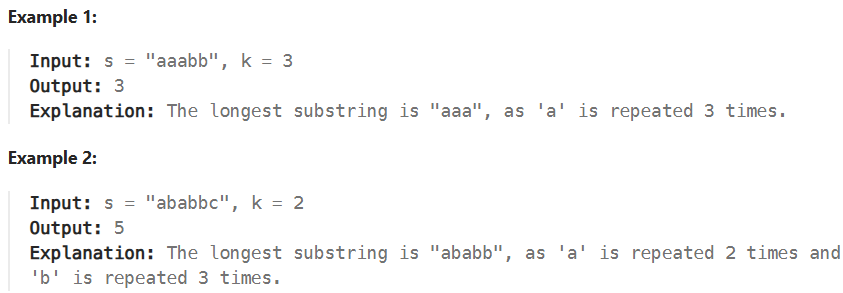
Tips:

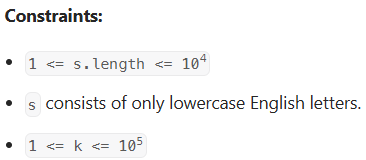
When dealing with unique characters, using a set can dramatically increase the process speed

**M-Longest substring with At Least K Repeating Characters**

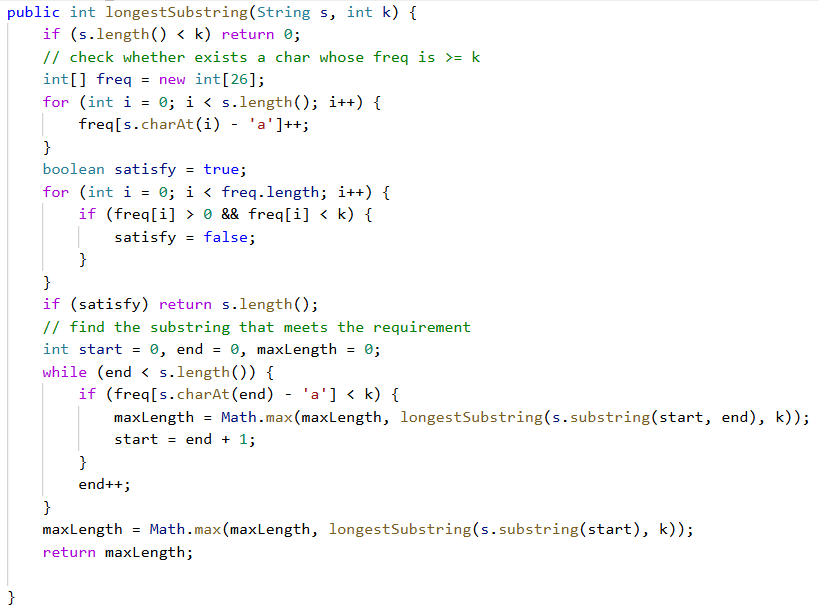
Given a string s and an integer k, return the length of the longest substring of s such that the frequency of each character in this substring is greater than or equal to k.

if no such substring exists, return 0.





Solution 1: Use recursive method. First traverse String s and find each character’s frequency. Then determine whether current string satisfies requirement. If not, split the origin string s by infrequent characters, and get the result from substring and current max value.

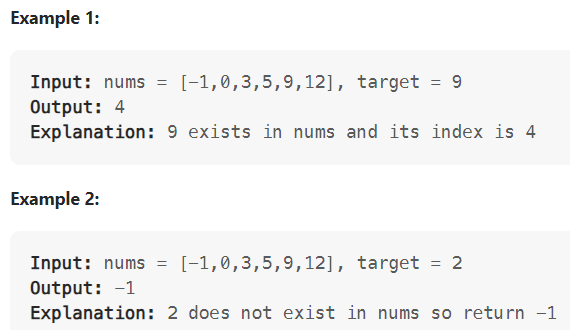


1. Binary search

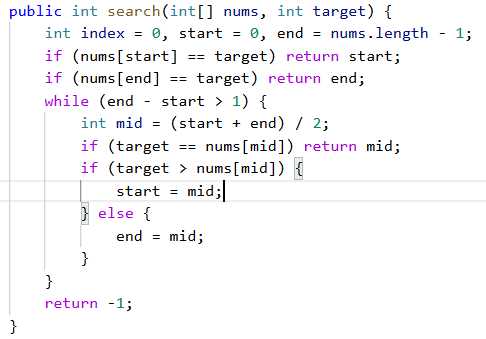
**Binary search**

Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If target exists, then return its index. Otherwise, return -1.

You must write an algorithm with O(log n) runtime complexity.

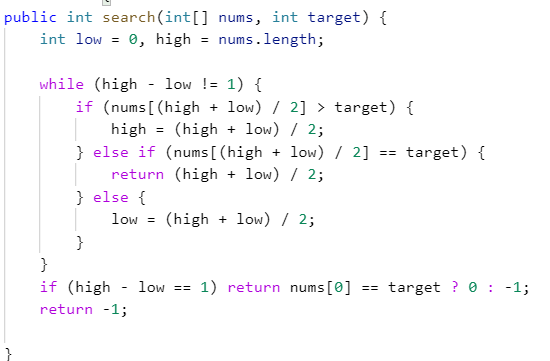


Solution 1: Compare middle one with target.



More intuitive than solution 2. Difference is using nums.length – 1 as end, and before while loop, we have to check start and end.

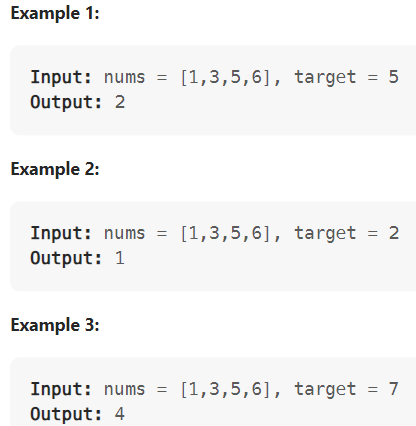
Solution 2: Use low and high to keep track of the start and end index, compare the middle one with the target



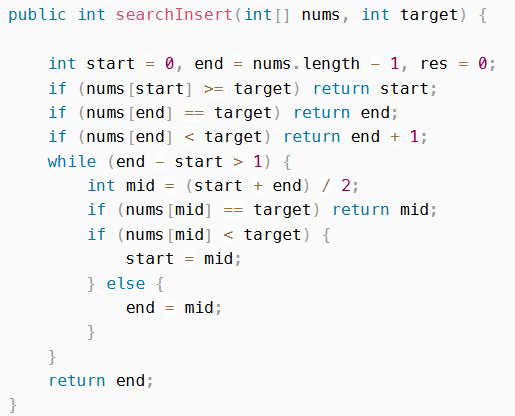
**Search insert position**

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.

You must write an algorithm with O(log n) runtime complexity.

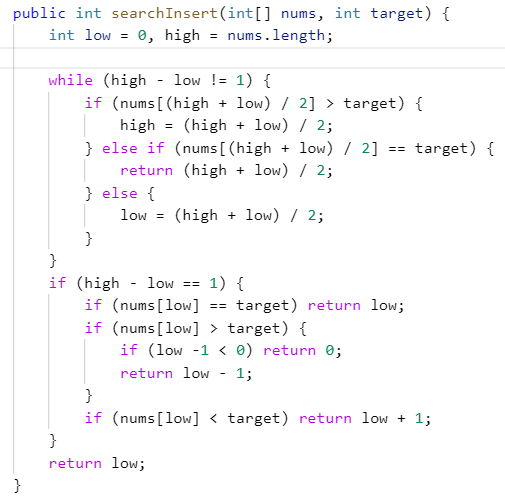


Solution 1: Same as Binary Search, we set start 0 and end nums.length – 1.

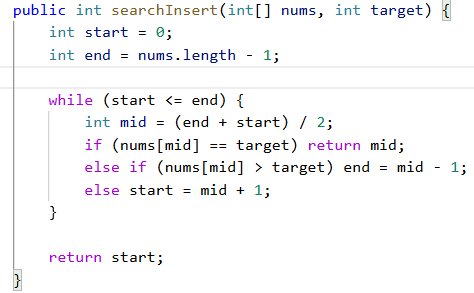


Similarly, we have to determine some special cases before the while loop.

Solution 2 : Same as 4.1, return the index when high – low is 1



Solution 3: we set the while loop’s condition to start <= end (end – start >= 0), and obtain succinct code. We need to change the update statement.

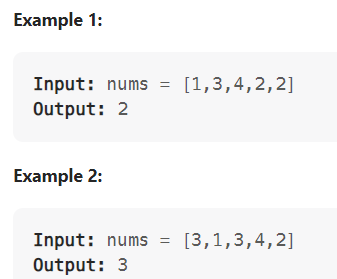


**M-Find the duplicate number**

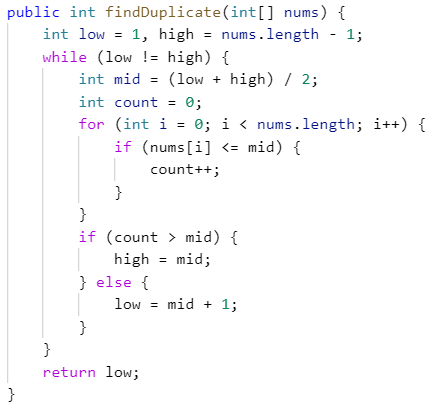
Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only one repeated number in nums, return this repeated number.

You must solve the problem without modifying the array nums and uses only constant extra space.



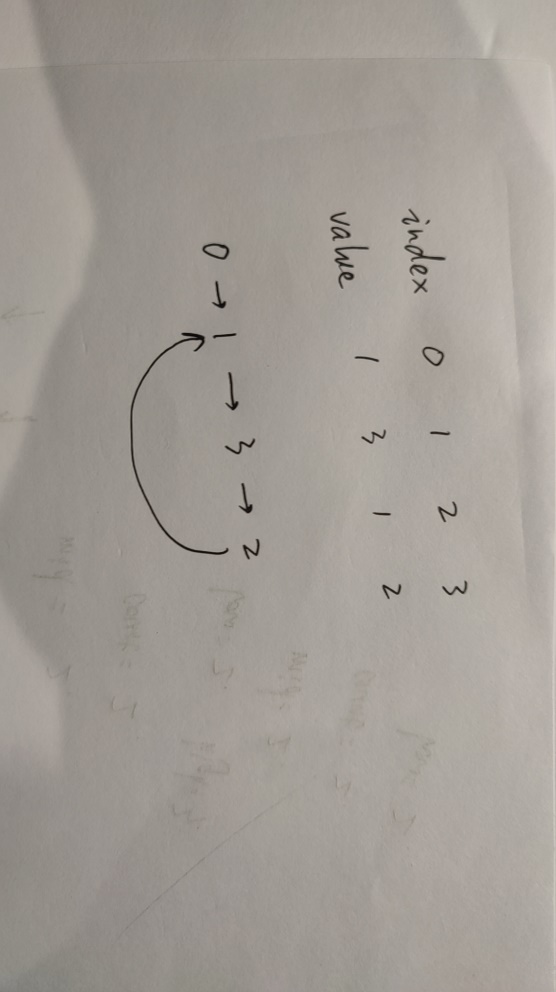
Solution 1: Use binary search. Instead search in the given array nums, we searh in integers 1 to n. Count the number of elements that is less than mid, then change the upper and lower bound

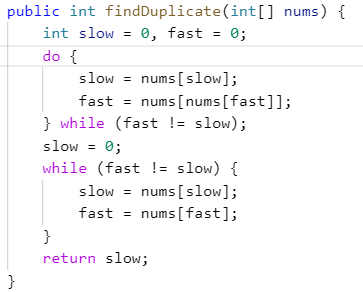


In the if-else block, if count <= mid, which means count = mid, that also means from low to mid(inclusive), there is no duplicate number, so we set low to mid + 1. If count > mid, which means there is a duplicate in low to mid(inclusive), so we set high to mid.

This solution takes O(nlogn) time and O(1) space.

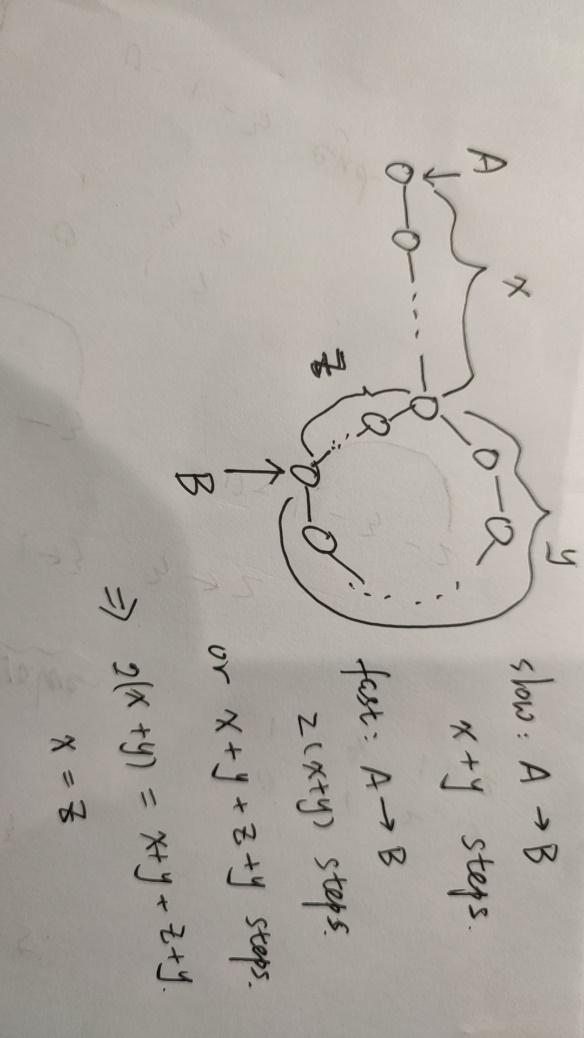
Solution 2: Treat the array as a linked list, use two pointer method to locate the start of the cycle.





In the first loop, we make slow and fast pointer meet. Then we set one pointer to the beginning, and let them move at the same speed, then we will find the start node of the cycle. Floyd’s Cycle Detection Algorithm, also known as Tortoise and Hare algorithm.

The proof:

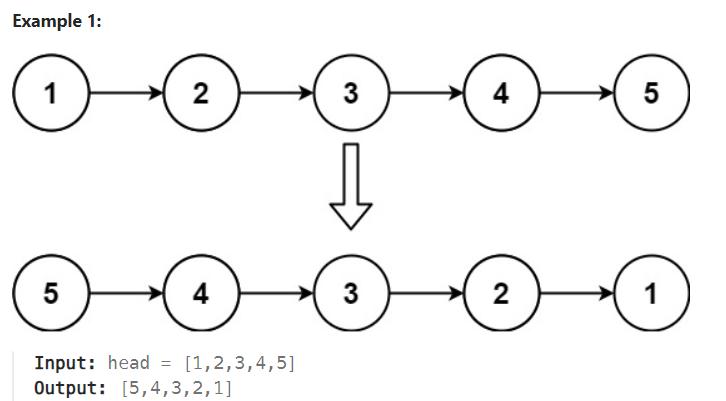


Tips:

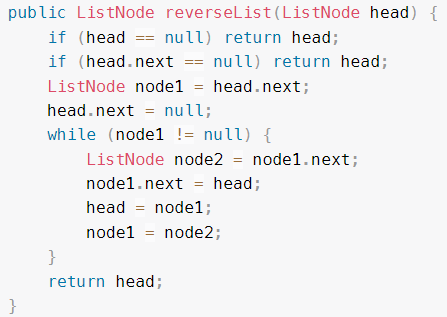
1. set.add() returns a boolean, so we can use it to determine whether an element is successfully added
2. **Recursion**

**Reverse linked list**

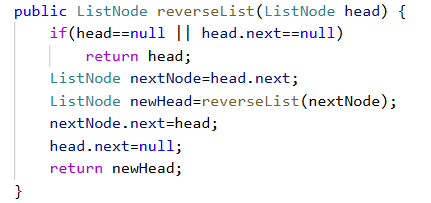
Given the head of a singly linked list, reverse the list, and return the reversed list.



Solution 1: Iterative way. Easy to understand.



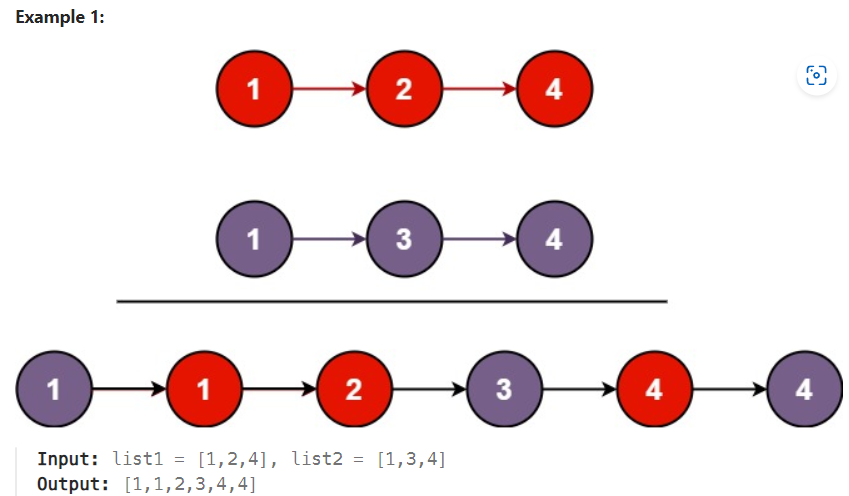
Solution 2: Recursive way.



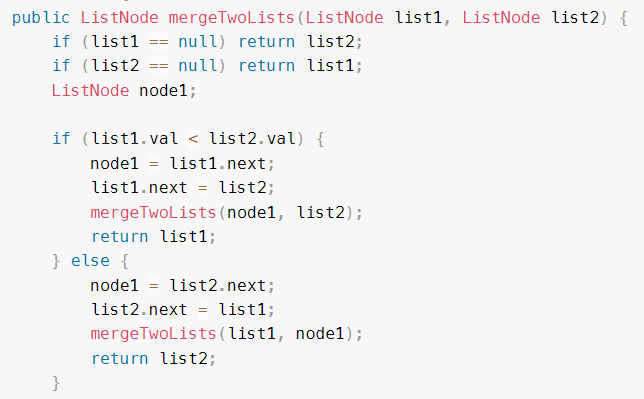
We have a base case, and some actions needed to be done in each call.

**Merge two sorted lists**

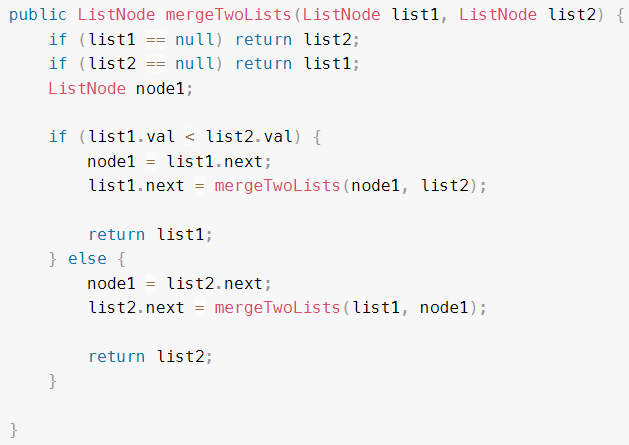
Given the heads of two sorted linked lists list1 and list2. Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.



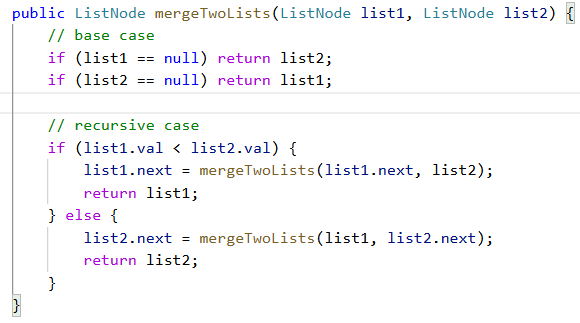
Solution 1: Recursive way. My original solution is:



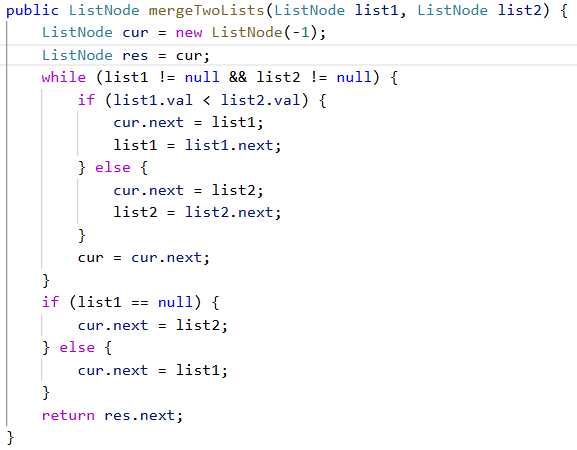
The above solution is wrong. For example, if list1 is {1,2,4} and list2 is {1,3,4}, after we link the first two element, then we will link 1 in list1 to 3 in list2. We shouldn’t assign list2 to list1.next. The right solution is:



A more succinct version



Solution 2: Iterative way. Create two ListNode, one is cur, we use it to build our links among elements. Another is res, it points to cur’s original position and we use it to get our result. Notice that the elements in both lists are in [0,50], so we set the value of cur to -1 to distinguish it from all other nodes.



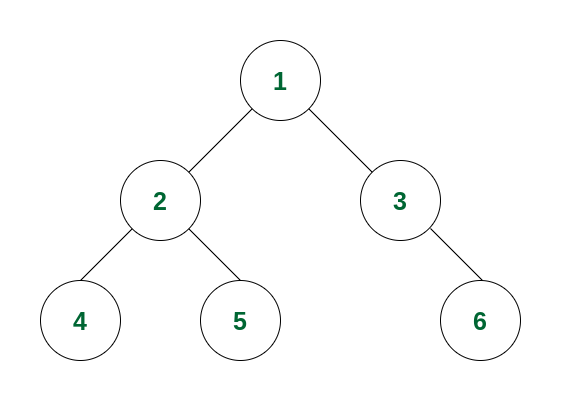
1. **Binary tree**

* **Depth and Height:**

**Depth:** upward, depth of a node is the number of edges from root to the node

**Height:** downward, height of a node is the number of edges from it to its deepest leaf

* **Traversal order**



In-order traversal:

4-2-5-1-3-6

Level-order traversal (Breadth-First Traversal): root is the first element

1-2-3-4-5-6

Pre-order traversal: root is the first element

1-2-4-5-3-6

Post-order traversal: root is the last element

4-5-2-6-3-1

In-order, pre-order and post-order means the root is traversed between, before and after left and right subtree.

If we want to build a binary tree from these traversal order, we can use:

in-order + pre-order traversal

in-order + post-order traversal

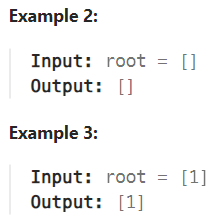
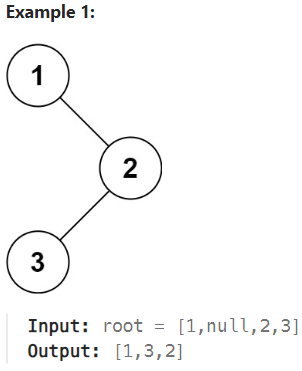
pre-order + post-order traversal

* **Binary Search Tree**

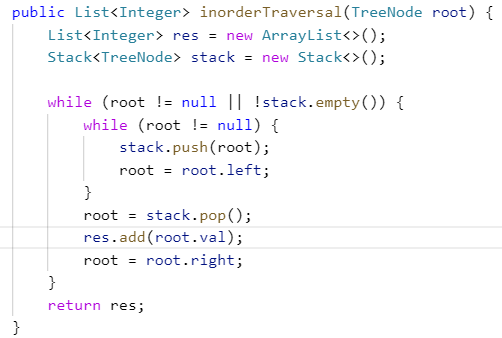
In a BST, all nodes in the left subtree are less than the root, and all nodes in the right subtree are greater than the root

**Binary tree inorder traversal**

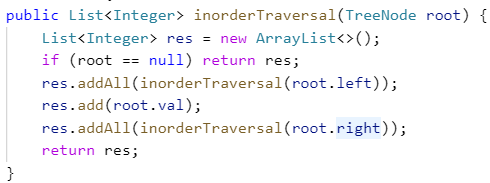
Given the root of a binary tree, return the inorder traversal of its nodes' values.



Solution 1: find the leftmost element from the root, use a stack to store the items, pop element out of stack, make its right node the new root



Solution 2: Very simple recursive solution

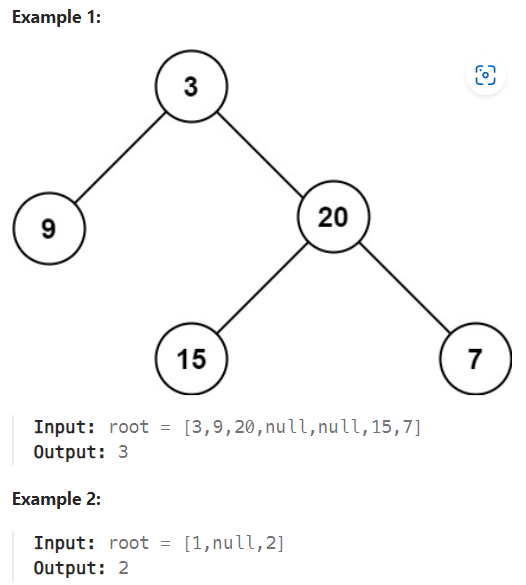


Tips:

1. If res is a List, then res.addAll(a) won’t change res if a is an empty list, and it will return false because res is unchanged.
2. Use a stack and pop its element to get in-order traversal of a tree.

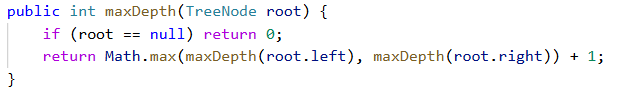
**Maximum Depth of Binary Tree**

Given the root of a binary tree, return its maximum depth. A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

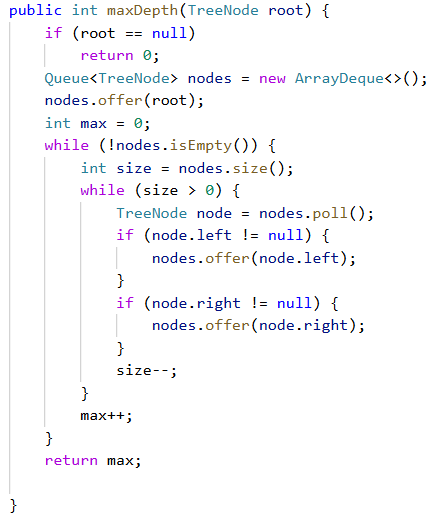


We cannot solve this problem without traversing all nodes, so we can simply use DFS or BFS without any modification.

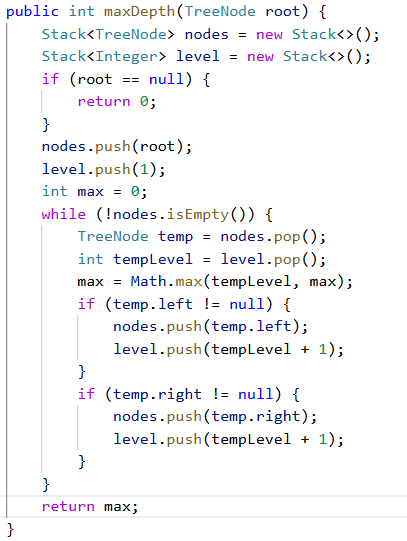
Solution 1: Recursive way.



Solution 2: BFS algorithm, use a FIFO queue to store nodes, each time we poll all elements out, we add 1 to depth.

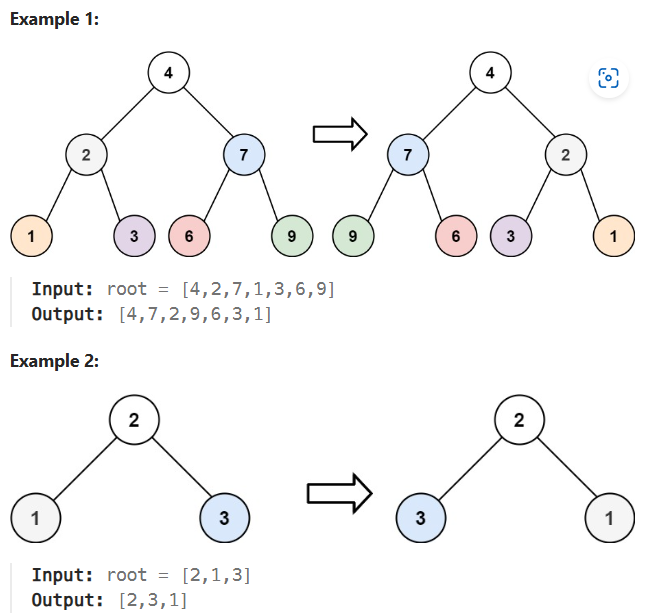


Solution 3: DFS solution, use a stack to store nodes and a stack to store level. We pop the top element of a stack, which is always an element with highest level.



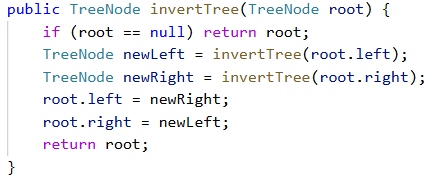
**Invert Binary Tree**

Given the root of a binary tree, invert the tree, and return its root.

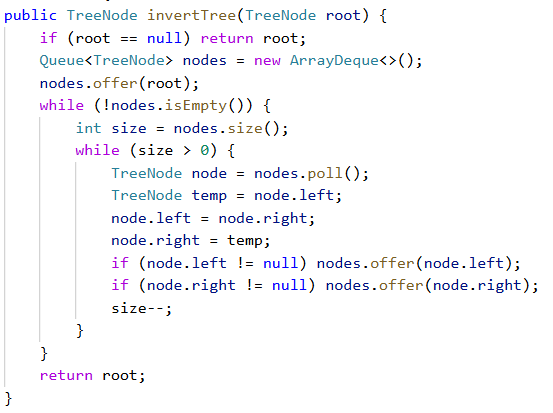


This question assumes that the root doesn’t change. In fact, we cannot reverse and create a new binary tree only with a level-traversal list.

Solution 1: Recursive way.



Solution 2: Iterative way. Use FIFO queue and level-traversal order.

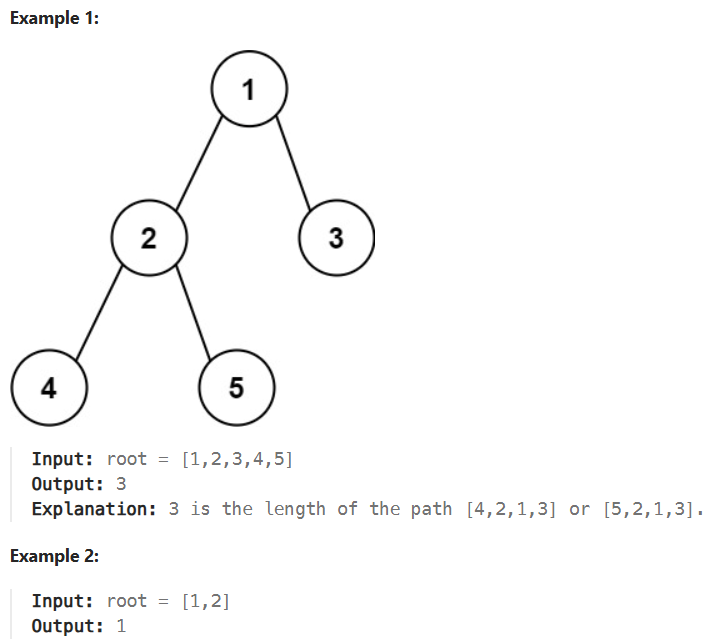


**Diameter of Binary Tree**

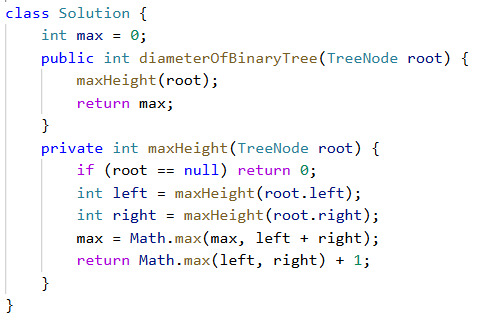
Given the root of a binary tree, return the length of the diameter of the tree.

The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.

The length of a path between two nodes is represented by the number of edges between them.

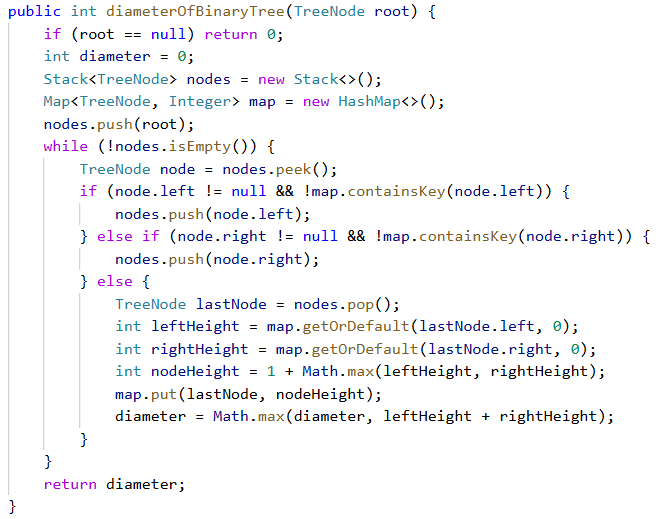


Solution 1: Recursive way. Diameter of each node is the max height of its left node plus the max height of its right node. Iterate through each node to find the diameter of entire tree.



maxHeight traverses through every node in the tree, it also updates our instance variable max to get the diameter of the entire tree.

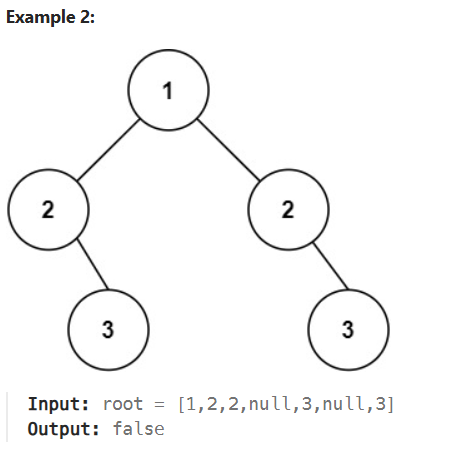
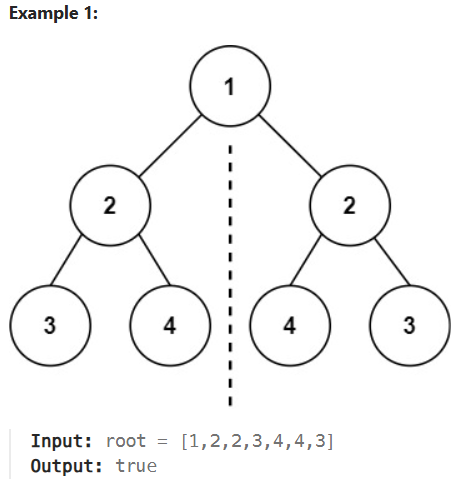
Solution 2: Iterative way. Use a stack to store tree nodes. The elements we pop up are in post-order, so root is the last element we pop up.



First we check whether it has left, and whether its left child is in the map. Then the right child. If it has left child and it’s not in the map, we pussh it to the stack. If it has left child but it’s in the map, we push its right child. If it has left and right child but they are both in the map, or its left and right child are both null, we deal with this node.

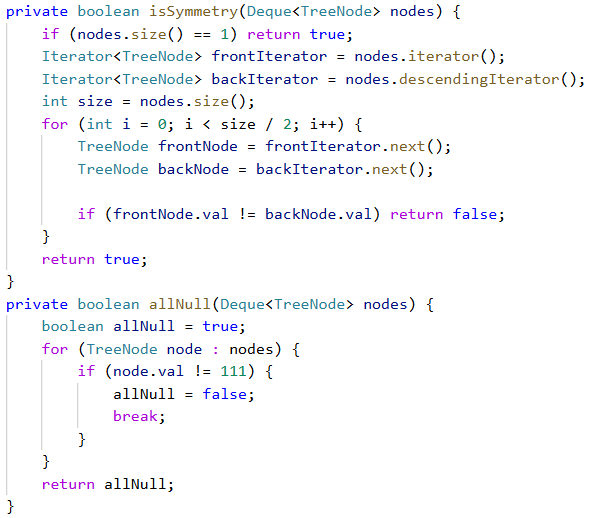
**Symmetric Tree**

Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).



Solution 1: Iterative way. Use level-order and a Deque to store nodes in each level. Since the value of our nodes is from -100 to 100, we use 111 to represent null nodes.



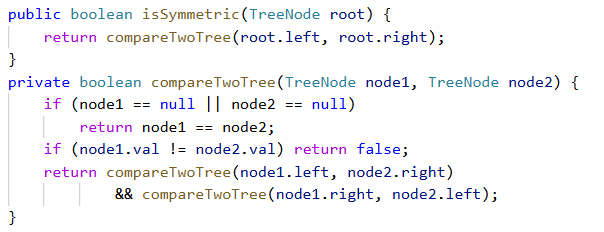


Our isSymmetry method won’t check the middle element if nodes size is odd, which never happens since we use 111 to represent null. But in future problems we might need to consider the middle node.

Solution 2: Recursive way. Since the problem tells us the tree has at least 1 node, we don’t need to check whether root is null. The tree is symmetry when:

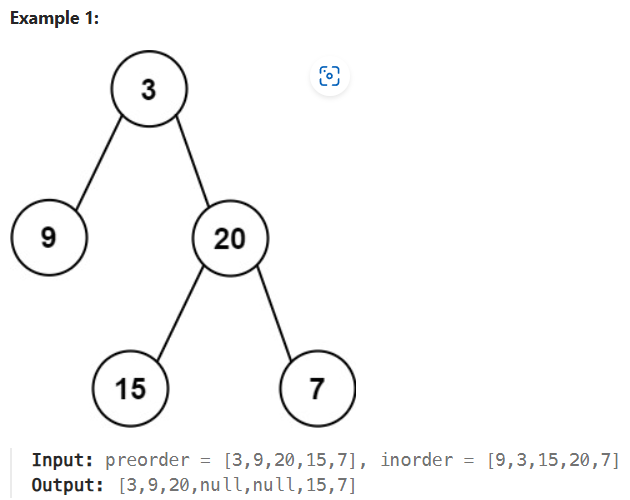
Base case, if left node and right node are both null, return true; if one of them is null but the other is not, return false. (This logic can be **simplified** like below)

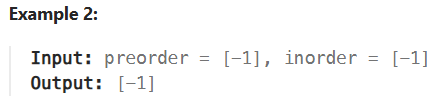
left node’s value equals right node’s value AND left node’s left subtree equals right node’s right subtree AND left node’s right subtree equals right node’s left subtree



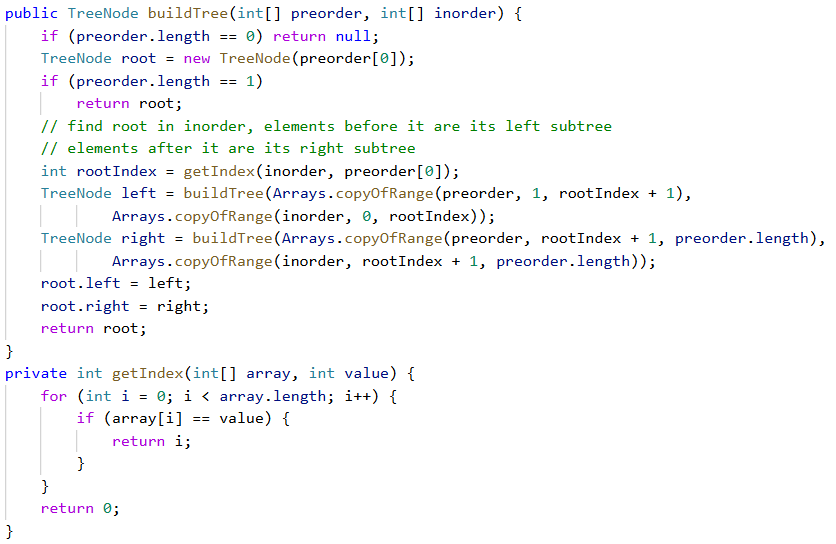
Construct Binary Tree from Preorder and Inorder Traversal

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





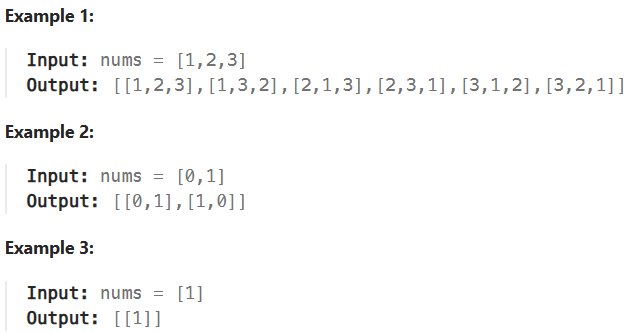
Solution 1: Recursive way. Find the root, split the inorder array into left-inorder and right-inorder array. Then find their root and do the recursive call.



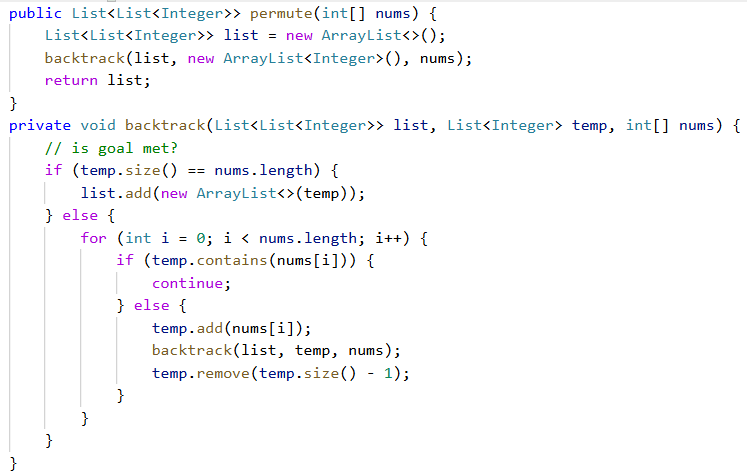
1. **Backtracking**

**M-Permutations**

Given an array nums of distinct integers, return all the possible permutations. You can return the answer in any order.



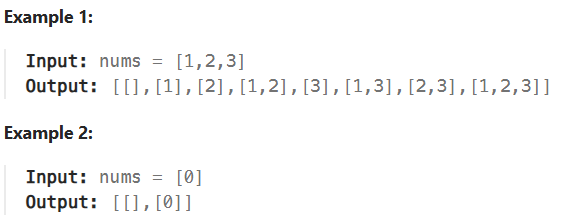
Solution 1: Backtracking + Recursion



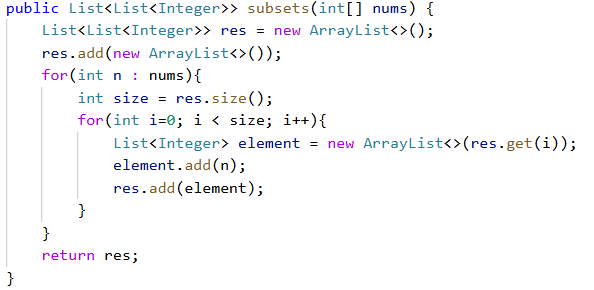
If the goal is met, we need to create a new object and add it to list, otherwise when we change temp, the content in list will change too. (we cannot use list.add(temp) directly)

**M-Subsets**

Given an integer array nums of unique elements, return all possible subsets (the power set). The solution set must not contain duplicate subsets. Return the solution in any order.



Solution 1: Intuitive for loop, no backtracking. Good solution.



The basic idea is: for current elements in res, let’s say [] and [1], we add one element “2” to them to create two new elements and keep original two elements, so we have 4 elements. Then we traverse all nums and do the same thing.

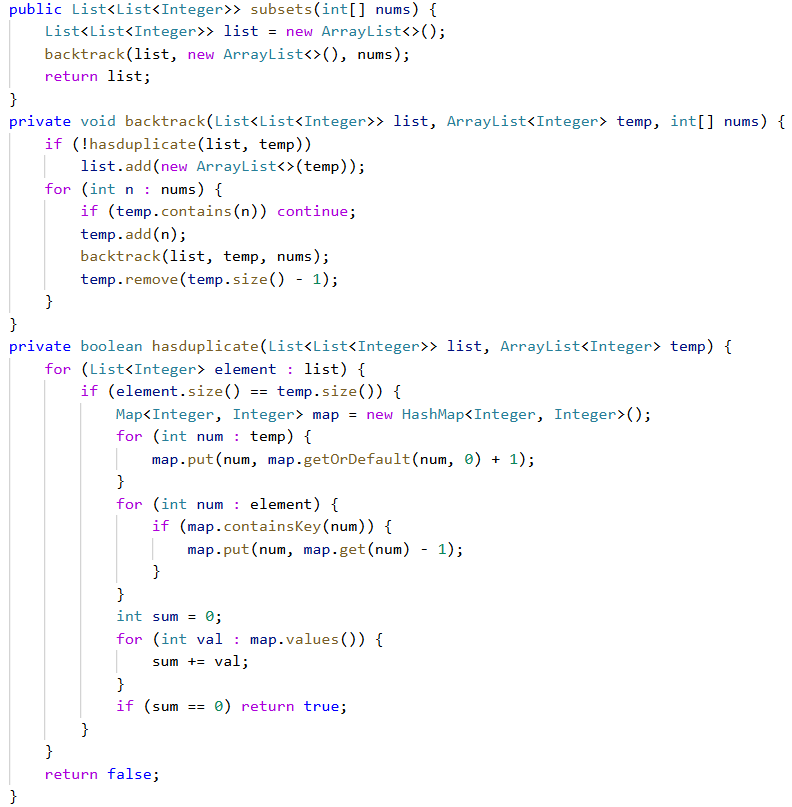
When get the i-th element, similar to Permutations, we also need to create a new object, otherwise we will change the original element too.

When traverse res, we need to store its orginal size in a variable, because its size is changed in every iteration and make the inner loop never end.

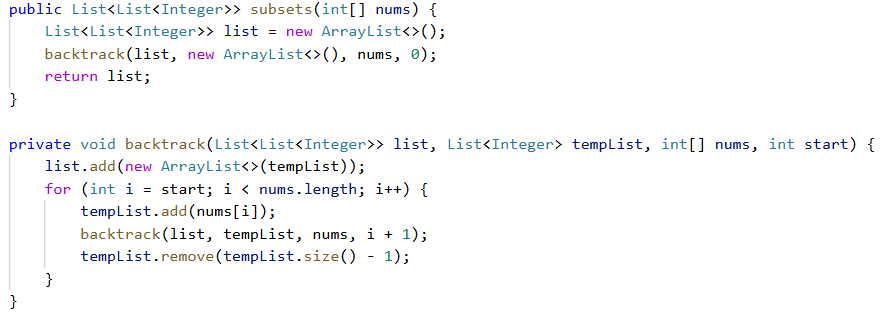
Solution 2: Backtracking. My intuitive way, use a hashset to add element. Exceed time limit when we have 10 elements.



Solution 3: Backtracking. My way, use a function to determine whether there is a duplicate in list. Also exceed time limit. Slower than solution 2.

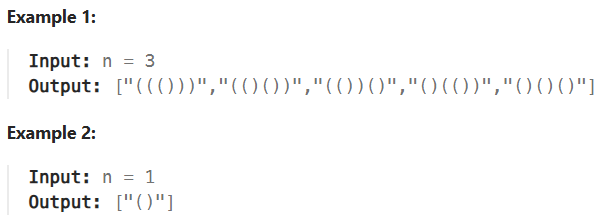


Solution 4: Backtracking. Someone’s solution. Not easy to understand, I think the first solution is the best.

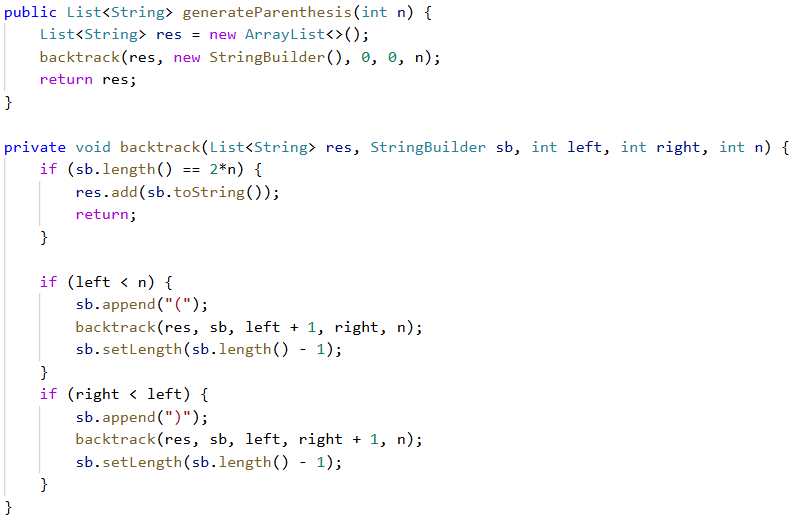


**M-Generate Parentheses**

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



Solution 1: Backtracking.



Firs if block is checking whether our goal is met. Then we add left parenthesis before adding right parenthesis. We use left and right to store number of “(” and “)” in our stringbuilder.

Tips:

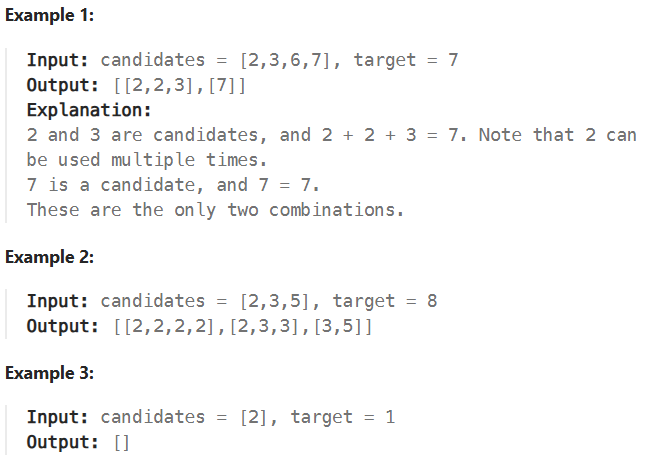
1. To delete last char in a stringbuilder, we can use sb.deleteCharAt(sb.length()-1) or sb.setLength(sb.length()-1). Using delete runs slower than setLength.

**M-Combination Sum**

Given an array of distinct integers candidates and a target integer target, return a list of all unique combinations of candidates where the chosen numbers sum to target. You may return the combinations in any order.

The same number may be chosen from candidates an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

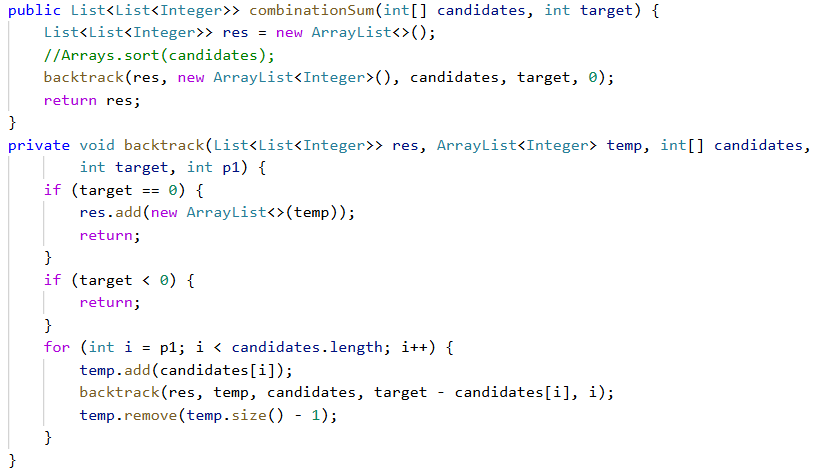
The test cases are generated such that the number of unique combinations that sum up to target is less than 150 combinations for the given input.



Solution 1: Use a stream to calculate the summation of temp, so the running time is slower.



Solution 2: Every time we add a candidates[i], we subtract target, that is, we use target – candidates[i] as our parameter in backtrack method. Better running time.



Tips:

1. We don’t have to sort the array.
2. DFS
3. BFS
4. Graph
5. Linked List
6. Sort
7. Trie
8. Stack/Queue
9. Priority Queue
10. Dynamic Programming