-Number of Islands 🡪 find number of islands

DFS or BFS does not matter, count the number of times the search gets triggered. Mark all visited ones. Number of trigger times = number of islands

-Add Two Numbers 🡪 Add two numbers that are represented by linkedlist in reverse

While loop condition is an or of either head not null. Account for a null head by adding zero if iterator of a list is null.

-Add Two Numbers II – Given two non-empty linked lists representing two non-negative integers. The most significant digit comes first and each of their nodes contain a single digit. Add the two numbers and return it as a linked list. Reversing the lists is not allowed.

Find the length of both lists and make a deep copy of the larger list. Increment an iterator ListNode to the spot in the longer list where the two lists have equal length as well as an additional marker for the node prior to the start of the sublist. This will be for in case of a carry bit after adding the shorter list and the longer list's sublist. Recursively add the longer list's sublist and the shorter list whilst tracking the carry.

If there is an extra carry after the addition of the lists, recursively add that carry from the spot of the longer list's sublist.

-Add Binary – Given two binary strings, return their sum (also a binary string).

While loop with two pointers at the ends of both strings. Decrement both pointers and track a “carry” flag for addition carry. If one pointer is past the beginning of the string, substitute 0 for the value at that string’s index.

-Compare Version Numbers – Compare two version numbers version1 and version2. If version1 > version2 return 1; if version1 < version2 return -1; otherwise return 0. You may assume that the version strings are non-empty and contain only digits and the . character. The . character does not represent a decimal point and is used to separate number sequences.

Split both strings on a period. Go through and compare the number at each corresponding index. If it is past the length of a specific version, default the version to 0.

-Implement Trie Prefix Tree(Design) – Implement a trie with insert, search, and startsWith methods.

Create a class for a TrieNode. Each TrieNode has potentially 26 children, so an array of length 26

that holds a TrieNode if that particular letter is a child of the current node. Defaults to all nulls.

A boolean field to determine if the current TrieNode is the termination of a word that has been

Inserted into the Trie, or if it is just a prefix.

To insert a string, iterate down the Trie tree through the TrieNodes, creating new TrieNode leaves where necessary. At the last letter, set terminate to true. Search iterates down the tree and checks if the letters exist in the Trie path. The last letter check should see if the terminate flag is set.

Checking a prefix iterates down the tree to check if the prefix path exists.

-LRU Cache (System Design) 🡪 Design LRU Cache with initial capacity and O(1) get and put

HashMap of key to value. LinkdList that ordered LRU status. HashMap of key to ListNode for O(1) access to a node. ListNode pointer to head and tail.

-Insert Delete GetRandom O(1) (System Design) – Design a data structure that supports all following operations in average O(1) time. insert(val): Inserts an item val to the set if not already present. remove(val): Removes an item val from the set if present. getRandom: Returns a random element from current set of elements. Each element must have the same probability of being returned.

ArrayList to store all the values. HashMap of values to the index in the arraylist that

the value is stored at.

Insert goes onto the end of the arraylist.

Remove swaps the element with the last element of the arraylist and then removes the last

element in the arraylist.

Use Random class of Java to return a random integer between 0 inclusive

and the length of the arraylist exclusive.

-Insert Delete GetRandom O(1) Duplicates allowed (System Design) – Design a data structure that supports all following operations in average O(1) time. Duplicate elements are allowed. insert(val): Inserts an item val to the collection. remove(val): Removes an item val from the collection if present. getRandom: Returns a random element from current collection of elements. The probability of each element being returned is linearly related to the number of same value the collection contains.

Arraylist to store all the values. HashMap of values to indices that values are at, HashSet.

Remove swaps the first index found for that element with the last element and deletes the last

element. Both hashsets for both values get updated.

Insert goes onto the end of the arraylist with the index updated in the hashmap.

Random class to return a random value amongst the arraylist.

-Random Pick with Weight (System Design) – Given an array w of positive integers, where w[i] describes the weight of index i(0-indexed), write a function pickIndex which randomly picks an index in proportion to its weight.

Construct a prefixes sum array where each index of the array is the cumulative sum of the weights array through that particular index.

To pick an index, generate a random value between the total sum and 1 inclusive. Binary Search through the prefixes array until the index is found where the random value is less than the element in the value in the current index of the prefixes sum array but greater than the value in the previous index.

-Shuffle an Array – Shuffle a set of numbers without duplicates.

Have a reference to the original array and create a shuffled array that is the size of the input array. Create an ArrayList that is a copy at first of the original array.

Build the shuffled array:

Have a pointer to the end of the ArrayList and continually randomly picking an index between the start and the end inclusive. Input the randomly picked index's value into the current index of

the shuffled array. Swap the current value at the random index with the end index's value of the

arraylist and decrement the end index and increment the insert index. Everything past the end

index in the arraylist are values that have already been picked.

-Time Based Key Value Store (System Design) – Create a timebased key-value store class TimeMap, that supports two operations.

1. set(string key, string value, int timestamp)

Stores the key and value, along with the given timestamp.

2. get(string key, int timestamp)

Returns a value such that set(key, value, timestamp\_prev) was

called previously, with timestamp\_prev <= timestamp.

If there are multiple such values, it returns the one with the largest

timestamp\_prev.

If there are no values, it returns the empty string ("").

HashMap of keys to ArrayList of Tuples. Tuples consist of value and timestamp pairs. Timestamps are always increasing so always append to end of ArrayList. Binary Search when a get is called to find the most recent earlier timestamp value.

-Design Tic Tac Toe (System Design) – Design a Tic-tac-toe game that is played between two players on a n x n grid. A move is guaranteed to be valid and is placed on an empty block. Once a winning condition is reached, no more moves is allowed. A player who succeeds in placing n of their marks in a horizontal, vertical, or diagonal row wins the game.

Two HashMaps, one for each player. The key is the row, and the value is a HashSet for each column in that row that the player has a piece on.

To check for a winning state - Check is the HashSet of the row that the respective player made a move on equals the length of the board. Check if there are the same number of keys(rows) as n and then check if each HashSet has the column number. Check diagonals by iterating through the rows as well and checking for if they contain the col # that is the same as the row # and if they have the col # that is the board length minus the row number.

-Min Stack – Design a stack that supports push, pop, top, and retrieving the minimum

element in constant time.

ArrayList that pushes onto end of list and removes from end of list. ArrayList holds a Pair object that consists of the value and the minimum of the value of the stack up to that point. The minimum tracking in the Pair class allows for O(1) access to the min value at all times.

-Design HashSet – Design a HashSet without using any built-in hash table libraries. To be specific, your design should include these functions:

add(value): Insert a value into the HashSet.

contains(value) : Return whether the value exists in the HashSet or not.

remove(value): Remove a value in the HashSet. If the value does not exist in the HashSet, do nothing.

Array of size large prime number. Hash each element by taking the mod of the key and the size of the array. If collisions occur, use an arraylist to chain elements that hash to the same index. Iterate through the arraylist to find the appropriate element.

-Design HashMap – Design a HashMap without using any built-in hash table libraries. To be specific, your design should include these functions:

put(key, value) : Insert a (key, value) pair into the HashMap. If the value already exists in the HashMap, update the value.

get(key): Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key.

remove(key) : Remove the mapping for the value key if this map contains the mapping for the key.

Create an array of a size that is large and a prime number. Hash each key value by taking the mod of it and the size of the array to find the index. If multiple values Hash to the same index for collisions, create an ArrayList in that bucket to hold each value(chaining). To find a value in a chained bucket, iterate through the list at that index.

**Two Pointer Approach/Sliding Window:**

-Trapping Rain Water 🡪 Array of elevations, compute how much water gets trapped in.

Two pointer approach: Iterate left pointer until first nonzero height.

(1)Move right pointer until hit an elevation that is higher than left pointer. Keep track of tallest right pointer has seen while iterating right. If right hits end without finding taller than left, move right to the tallest that was seen.

(2)Find the shortest pillar between the left pointer and right pointer. Increment left up one. Subtract current left pointer from shortest and add to total. Increment left until hit right. Repeat process until left index is about to hit right.

Repeat the above until left hits end of array.

-Longest Substring Without Repeating Characters

Two pointer approach: move right pointer right until a duplicate is hit. Move left pointer right until duplicate is gone. Track size when there are no duplicates. Finish when right is at the end.

-Longest Substring with At Most Two Distinct Characters – Given a string s, find the length of the longest substring t that contains at most 2 distinct characters.

Sliding window with two pointers approach. Increment right pointer until the end of the string. Keep a Hashmap of characters and their respective counts. If the number of unique characters between the left and right pointer is greater than two, increment the left pointer until there are <= 2 unique characters in the window. Track the largest window.

-Find All Anagrams in a String – Given a string s and a non-empty string p, find all the start indices of p's anagrams in s.

Get the character counts of the substring and store it in a char array of length 26.

Sliding window approach. Move the right pointer until the window length is equal to the substring length and track the character counts.

Check for the anagram by comparing character counts in both 26 length count arrays.

Move both the left and the right pointers right by 1 and recheck after each slide.

-3Sum – return all unique triplets which sum to zero O(n^2)

**Two pointer approach for sorted array:** for sorted array, left pointer at index 0, right index at the very right. If sum is less than target, increment left index, otherwise if it is greater, decrement right. If two pointers go past each other, then target not there.

For 3 sum: sort array (n log n) and for each index use two pointer approach to find target sum that is negative of current sum. Do not run two pointer on duplicate values so increment pointer until it is not equal to previous and then run two pointer.

-4Sum – Return all unique quadruplets in an array that sum to a given target. O(n^3)

Utilize two pointer approach for sum in sorted array. Sort the array. Create a HashSet of quadruplets already seen before. Outer loop iterates from 0 to n - 4, inner loop from outer\_loop + 1 to n - 3, and two pointer approach from inner\_loop + 1 and end both towards the middle (two pointer approach for sorted array).

This approach can be extended to k\_sum for a O(n^(k - 1)) by adding more looping for each additional k.

-Container With Most Water – n nonnegative integers that are points at (index, height[index]) find two lines which together with x axis forms a container that contains the most water. Line is parallel to x axis.

Two pointer approach: Start with pointer at left end and pointer at right end. Track the left and right index of the maximum container seen so far. Whichever pointer has a shorter height, shift that pointer inwards by one. Stop when the pointers meet/cross.

Proof: We starts with the widest container, l = 0 and r = n - 1. Let's say the left one is shorter: h[l] < h[r]. Then, this is already the largest container the left one can possibly form. There's no need to consider it again. Therefore, we just throw it away and start again with l = 1 and r = n -1.

-Minimum Window Substring – Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n). **Input: S** = "ADOBECODEBANC", **T** = "ABC"

**Output:** "BANC"

Sliding Window – Left and right pointer starting at index 0. Keep incrementing right pointer until all the letters in T are in the substring. Move left pointer right until substring does not contain all the letters anymore. Repeat until the right index is at the end. Then keep moving left pointer right until it is no longer a valid substring. Track the indices that contain all the letters in T and have the smallest difference.

The check for a valid substring can be done through arrays representing the values of a-z, A-Z in an ascii table. An array of 52. The check takes O(52).

-Minimum Size Subarray Sum – minimum size of subarray with sum greater than or equal to *k.*

Similar to minimum window substring or other two pointer approach through arrays. Move right pointer right until sum is greater than or equal to *k.* Move left pointer right while sum is *>= k.* Keep track of largest subarray which satisfies problem conditions.

-Read N Characters Given Read4 – Given a file and assume that you can only read the file using a given method read4, implement a method to read n characters. The read function will only be called once for each test case. You may assume the destination buffer array, buf, is guaranteed to have enough space for storing n characters.

Pass in to read4 a buf array of size 4. Keep count of the total total number of characters read after each call to read4 and stop if either the total number after one call is < 4 or the total count reaches n.

-Valid parentheses 🡪 String of just ‘(‘, ‘)’, ‘{‘, ‘}’, ‘[‘, or ‘]’. Valid is all opens are closed by the same type and opens must be closed in correct order.

Use a stack. Push onto stack if left bracket. Pop off stack if right bracket. If popped is incorrect matching bracket or if stack is empty when trying to pop it is false. If at end, more left 🡪 stack is not empty then it is false. Otherwise if stack is empty at the very end, it is true.

-Rotting Oranges – Each cell of grid has empty, fresh, or rotting orange. Adjacent to a rotting orange rots per time step. Min time for all rotting.

BFS: Add all rotting oranges into BFS queue. BFS expands the rotting in rounds. Track each round by changing the cell getting added into BFS queue if it is fresh to a timer of previous adjacent cell that causing the current cell to be added into queue plus some value. At end, return the largest valued cell and calculate the original number of rounds.

-Product of Array Except Self – return an array where each spot is the product of all elements except for element at particular index

Make array that is product of everything to the left of that index. Keep track of product accumulating from the right and multiple current product with current position in array from the right.

-Longest Consecutive Sequence – Given an unsorted array of integers, find the length of the longest consecutive elements sequence. Your algorithm should run in O(n) complexity.

Add all numbers to a HashSet. Iterate through the hashset but do not visit numbers already seen. Expand left and right from the current number checking if they exist in the nums HashSet and tracking the longest sequence.

Add all seen numbers into a seen Hashset.

Note: Cannot modify a Collections while iterating through it unless using Iterator.remove and can only call remove once pe iterator.next call.

-Reverse Linked List

Iterate through list and attach like this: Sentinel -> attach here -> previous node

-Minimum Remove to Make Valid Parentheses - *parentheses string* is valid if and only if:

* It is the empty string, contains only lowercase characters, or
* It can be written as AB (A concatenated with B), where A and B are valid strings, or
* It can be written as (A), where A is a valid string.

Valid parentheses boils down to at any point cannot have more ( to the right of ) and more ) to the left of (.

Start from left and increment counter every time the character is (. Decrement the counter if it is ). If the counter is below zero, at that index to a HashSet. Do the same process from the right and increment a counter for ) and decrement it for a (. Add all negative counter indices to a HashSet. Remove all characters at the indices in the HashSet.

-String Transforms Into Another String – Given two strings str1 and str2 of the same length, determine whether you can transform str1 into str2 by doing zero or more conversions. In one conversion you can convert all occurrences of one character in str1 to any other lowercase English character. Return true if and only if you can transform str1 into str2.

Iterate through the strings and have a mapping from the the character in the first string to the corresponding character in the second string at the same index. If at any point, a character from the first string is mapped to two different characters in the second string, return false.

If both strings have every single character, then return false. This means that there is no character to serve as a "buffer" in the first string, but it only matters if the second string also has all the characters otherwise the first string can use the character not present in the second string as the buffer.

Iterate through the mappings and if any character gets mapped to itself and the second string has all the characters, return false. The strings are already established to not be the same, therefore if a character is mapped to itself, that means that those characters cannot be changed.

Since the second string has a full character set, there is no buffer character that can be used to transform characters.

i.e.

a cannot be changed, and any time a character gets changed, now two positions have the same character when the other string only has 1 of each character.

acdefg....yzb

abcdef....xyz

**Dynamic Programming:**

-Best Time to Buy and Sell Stock – Array of stock values on day i

Dynamic Programming – Keep track of currently seen minimum value. Keep track of largest seen prices[i] – min\_value seen as iterate one pass over array.

-Subarray Sum Equals K – Given an array of integers, find total number of continuous subarrays that sum to K.

Dynamic Programming – Start a cumulative sum from the start of the array. For each sum, track the number of times that sum has been seen in a HashMap. Find the difference between K and current sum at the current index and increment the number of times that sum has been seen from the HashMap. Add the number of times the difference has been seen in the HashMap to the total number of subarray sums equals k.

-Maximum Subarray – maximum subarray in an array

Dynamic Programming: sum array from the right and track the max. If current index value is larger than current sum + current index value(means sum currently is negative and current value is positive), restart sum at current index and sum right. Repeat process until hit the end and return the max sum.

-Decode Ways – Given a non-empty string containing only digits, count the number of ways it can be decoded. ‘A’ – 1, ‘B’ – 2 … ‘Z’ – 26.

**Bottom up = tabulation**

**Dynamic Programming:** Using a sliding window. Create an array of length 2. Set *arr[0] = 1* if first character if not ‘0’. Set arr[1] to one if first two characters >= 10 and <=26. Add *arr[0]* to *arr[1]*.

Iterate from 3 to the length of the string *s*. If *i* is odd, modify arr[0]. If *i* is even, modify *arr[1].* If the current character != ‘0’, set the current arr index to the other arr index’s value. If the current character and the previous one make a number in between ten and 26, add the original arr index’s value to the current arr index’s value. At the end, return arr[0] if the length of s is odd, arr[1] if the length of s is even.



-Climbing Stairs – climbing stair case, n steps to reach the top. You can climb in either 1 or 2 steps. How many ways can you climb to the top.

Bottom up = tabulation

Dynamic Programming: Iterate from 3 to equals *n*. Have a variable called sum which sums the total number of ways to current *i. first = 1*, *second = 2*. Within each iteration, *sum = 0;* *sum += first + second.* If *i % 2 != 0, first = sum, else second = sum*. *sum* at the end is the total number of ways up the stairs.

-Fibonacci Number – The Fibonacci numbers, commonly denoted F(n) form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. Given N, calculate F(N).

Bottom up = tabulation

Sliding window of size 2 from 0 to n calculating the Fibonnaci sequence. Store 0th and 1st Fibonnacci numbers first. The next value is always the sum of the previous two. So alternate, if i is even put it into window[0], otherwise into window[1].

-Minimum Path Sum – *m x n* grid with non-negative integers, find a path from top left to bottom right which minimizes the sum along its path. You can only move down or right.

Bottom up = tabulation

**Dynamic Programming**: Create another grid of *m x n* size that tracks costs and set them all to -1. Set the top left of the cost grid to the original grid’s value in the top left spot. Iterate through the cost grid in ascending row and column order, and set the new square on minimum of the top square and left square plus the cost of the current square. Return the bottom right square of the cost matrix.

- Coin Change – given coins of different value and a total amount of money, find the fewest number of coins to make that amount if possible. Each coin value can be used infinitely many times.

Bottom up = tabulation

**Dynamic Programming:** DP array of size amount + 1. Each value in array with index i represents the least amount of coins that can make up amount i. For each index, iterate through each coin value and if the index equals a coin value, set it to the index to value 1, otherwise, check the index – (each value in coin) and if the index – coin\_value + 1 is less than what is in the current index or the current index is 0, set the current index value to previous index + 1. Return the value in the largest index of the dp array.

**-** Word Break – Given a **non-empty** string s and a dictionary wordDict containing a list of **non-empty** words, determine if s can be segmented into a space-separated sequence of one or more dictionary words. i.e. applepenapple [“apple”, “pen”] 🡪 true “apple” “pen” “apple”

Bottom up = tabulation

**Dynamic Programming:** Iterate through the characters of the String s. Keep a HashSet of indexes that can be segmented from the words in word dict. If the substring from 0 to current index is in the dictionary, add the current index into the index set. Otherwise, iterate through the word dict and if the current index – length of current iterated string in word dict >= 0 and the current index – length is in the HashSet of indexes, add the current index to the HashSet. Return if the HashSet contains the index of the s.length() – 1 at the end.

-House Robber -- You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night. Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

**Dynamic Programming -- bottom up = tabulation**

Create a n sized tabulate array. Set the first index value equal to nums[0] and the second array value equal to max(nums[0], nums[1]). --> At the second house base case, only rob the house that would give more money.

Iterate from index 3 to the end of the nums array. At each step, put into the tabulate table at that index the max(tabulate[index - 2] + nums[index], tabulate[index - 1]), represents

the two options of either robbing the current index house or not.

Return the last index value of tabulate table.

-Next Permutation – Rearrange numbers into the next greater permutation of numbers. If it cannot be larger, rearrange them into the lowest possible order (ascending order).

Starting from the right and moving left, find the first value that is smaller than the value previous to it (index k). Find the smallest value to the right of index k that is larger than the value at index k. If there are multiple values that are equal to the smallest value to the right of k, take the rightmost one. Swap the value at index k with the rightmost smallest value to the right of k that is larger than k. Reverse the order of all the elements to the right of k.

**Binary Search/Divide and Conquer:**

-Search in a Rotated Sorted Array – Find a target value in a sorted array that is rotated at a pivot point i.e. [1,2,3,4,5] 🡪 rotated [4,5,6,7,0,1,2]

Special cases: if length = 0, length = 1, length = 2

Otherwise, if nums[0] < nums[nums.length – 1] 🡪 unrotated, call binary search

Else:

Binary Search – Find pivot point with binary search

1. If mid > nums[0] 🡪 pivot to the right, find\_pivot(nums, mid+1, upper)
2. If nums[mid] <= nums[0] 🡪 pivot is mid or to the left
   1. Check for mid == 0, if nums[mid] == nums[0] return 0, pivot is first index
   2. Else
      1. Check if current is pivot, nums[mid] < nums[mid – 1]
      2. Otherwise find\_pivot(nums, lower, mid)

If nums[0] > target, binarySearch(pivot, end)

Otherwise, binarySearch(start, pivot)

-Search a 2D Matrix – search for a value in an *m x n* matrix. Each row in the matrix is sorted in ascending order. The first number in a row is greater than the last number in the previous row.

Binary Search – During binary search, implicitly convert row and column indices into 1D array indices. *row = mid / arr[0].length, col = mid % arr[0].length*.

-Find First and Last Position of Element in Sorted Array – Given an array of integers nums sorted in ascending order, find the starting and ending position of a given target value. If the target is not found in the array, return [-1, -1].

Binary Search – Binary search for the start of the position. Modify to normal binary search to check if found if index is 0 or if index – 1 does not equal target 🡪 found start.

Similarly, if index is length -1 or index + 1 does not equal target 🡪 found end.

-Median of Two Sorted Arrays – Two sorted arrays of size *m* and *n*. Find the median of the two sorted arrays in O(log(m + n)).

Find the smallest of the two arrays. Partition that array in a binary search type manner.

int start = 0;

int end = nums1.length;

while(start <= end)

{

int mid\_1 = (start + end) / 2;

The second array gets partitioned such that the number of elements on the left in array 1 and array 2 are equal to the number of elements on the right.

int mid\_2 = (nums1.length + nums2.length + 1) / 2 - mid\_1; // + 1 handles both even and

// always takes the ceiling on odd numbers

Note: The mid numbers do not represent indices, rather they represent the partitions in between indices.

n1\_1 n1\_2 n1\_3 n1\_4

p0 p1 p2 p3 p4 the partitions lie between indices

mid\_1 is not 0, there are elements to the left of the partition in nums1. Take the largest element which should be *num1[mid\_1 – 1]*, and the same with *nums2.* If *mid\_1* or *mid\_2* are 0, then set the respective left values for each array to *Integer.MIN\_VALUE.* Similarly, take the smallest values right of the partition for *nums1* and *num2*. If there are no values to the right of the partition for an array because *mid\_1* or *mid\_2* is not *< nums1.length/nums2.length* then set the value to *Integer.MAX\_VALUE.*

Conditions that the partition found the correct spot: *n1\_left <= n2\_right && n2\_left <= n1\_right*. This means that the splits have occurred directly at the center of what would have been the combined array.

If the total number of elements is even, 🡪 return avg of max(two elements on left) and min(two elements on right).

Odd 🡪 max(two elements on left). The left partitions sum has +1 more element that the right partitions sum, because the formula used to find the partition of the sum on the left side was *(nums1.length + nums2.length* ***+ 1) / 2*.** The **+1** ensures for odd numbers the ceiling gets taken so that there is 1 more element on the left. For even numbers, the floor gets taken so there are still even partitions between left and right.

If the condition for finding the correct partition was *not* met:

If max of the left side of *nums1* is greater than the right of the partition of nums2, need to move the partition of nums1 to the left. Set right to mid – 1 and recalculate both mids.

Otherwise, if max of left of *nums2* is greater than min of right side of *nums1,* then the partition for *nums2* needs to be moved right. Set left to mid + 1 and recalculate both mids.

If the loop exits after the loop condition no longer is satisfied, then the arrays were not sorted to begin with.

-Search a 2D Matrix II – Write an efficient algorithm that searches for a value in an m x n matrix.

This matrix has the following properties: Integers in each row are sorted in ascending from left to right. Integers in each column are sorted in ascending from top to bottom.

Start at bottom left corner. Move up if current number is greater than target. Move right if current number is less than target.

-Leftmost Column with at Least a One -- A binary matrix means that all elements are 0 or 1. For

each individual row of the matrix, this row is sorted in non-decreasing order. Given a row-sorted binary matrix binaryMatrix, return leftmost column index(0-indexed) with at least a 1 in it. If such index doesn't exist, return -1. You can't access the Binary Matrix directly. You may only access the matrix using a BinaryMatrix interface: Submissions making more than 1000 calls to BinaryMatrix.get

will be judged Wrong Answer.

Binary search each row for the first 1 in that row. Return the smallest column found.

-Spiral Matrix – Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order

Go around in layers of the matrix. Hardest part is getting indexes correct and stop conditions correct.

-Copy List with Random Pointer – LinkedList where each node has a copy to a random node in the list. Return a deep copy of the list.

Interleave the newly created list with the original list. O1 -> n1 -> o2 -> n2…. The new list must come after the original because to get the new random node it needs to access o1.random.next. After interleaving, separate the lists and return the new list.

**Heap:**

-Kth largest element in an array 🡪 use a minheap

-Find the Median of a Data Stream (System Design) – Supports addNum and getMedian

Two balanced Heaps approach – Minheap and maxheap. After each insert, make sure each heap’s size is within one of each other. If not, pop off the root of the larger heap and add it to the smaller heap. If the total size is odd, return whichever root falls onto the correct count. If total size is odd, return the avg of both roots.

-Top K Frequent Elements – Non-empty array of integers, return k most frequent elements.

HashMap of number to counts. MinHeap of size k of pairs of values to counts.

-High Five – array of students and test scores, avg of top five test scores per student.

HashMap of student id to arraylist of tests. MinHeap for top five.

-Merge k sorted lists

Min heap of all the listnodes with their values. After removal of root which is the min, move node to its next and reinsert into heap. If it is null, do not reinsert and do not heapify. No extra array space needed because each node is inserted directly into heap so it marks its lists head and the value.

-Merge Sorted Arrays – Two sorted arrays, the first array has enough buffer space at end to hold the second array.

Start adding elements from the end of the first array, and add the elements in descending order.

-Decode String – Given an encoded string, return its decoded string. The encoding rule is: k[encoded\_string], where the encoded string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

Function to iterate starting from input index. If a string is a number, get the number of copies and recurse on the index of the start of the inner string(past the first '['). The call returns the string

that has been copied the appropriate amount of times. Append that string to the current call stack’s StringBuffer and increment the iterating index past the entire string that was repeated (use the counts of the number of left and right brackets, increment left for each '[' and decrement for each ']'. Break out when the left count is zero).

Repeat the string the appropriate number of times and return that string.

**Backtracking:**

-Word Search -- > find if word exists in word search (2d grid) 🡪 backtracking algorithm

<https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/>

**Backtracking**: mark status of currently visited path so do not hit same spot again. Before returning back from current call, remove that mark so that a different path can use that spot again. Remove the mark so that a parent recursive call can go down a different path that uses that spot. 🡨 Removal requires a temporary local variable rather than directly returning the recursive call.

private boolean backtrack(char[][] board, String word, int count,

boolean[][] visited, int x, int y)

{

if (x < 0 || x >= board[0].length || y < 0 || y >= board.length)

{

return false;

}

else if (word.charAt(count) != board[y][x])

{

return false;

}

else if (visited[y][x])

{

return false;

}

else if (count == word.length() - 1)

{

return word.charAt(count) == board[y][x];

}

else

{

**visited[y][x] = true;**

boolean return\_val = backtrack(board, word, count + 1, visited, x - 1, y) ||

backtrack(board, word, count + 1, visited, x + 1, y) ||

backtrack(board, word, count + 1, visited, x, y - 1) ||

backtrack(board, word, count + 1, visited, x, y + 1);

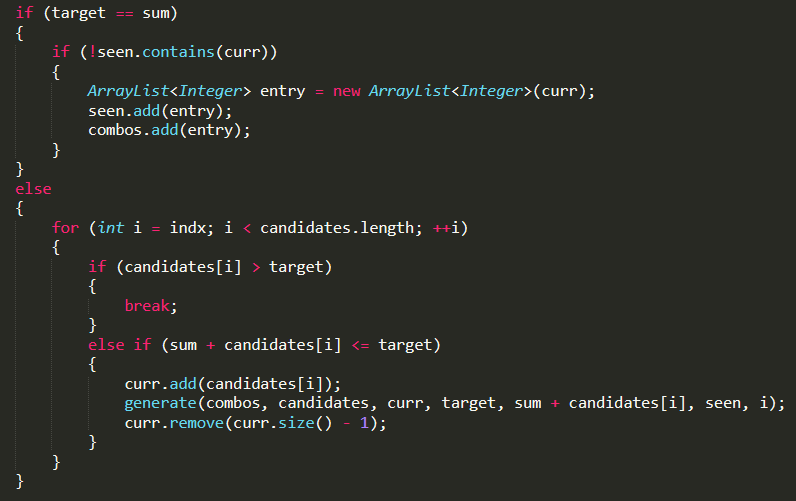
**visited[y][x] = false;**

return return\_val;

}

-Combination Sum -- Given a set of candidate numbers (candidates) (without duplicates) and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target.

 🡨 first call



For loop through the *candidates* array starting from argument *indx*(first call to function is 0) to the end.

Backtracking -

Add the current value to the array and recurse with the current index and then

after the recursive call returns, remove the value from the array before the next iteration.

[1, 2, 3, 4, 5] : 10 🡪 array gets pre-sorted

Recursive calls:

1

1 1

1 1 1

1 1 1 1

...

...

1 1 1 1 1 1 1 1 1 1 --> add to list

1 1 1 1 1 1 1 1 1 --> back up recursive call stack, last 1 removed (nothing added, sum + 2 > target)

1 1 1 1 1 1 1 1 --> up recursive call stack, removes 1 (nothing added, sum + 2 > target)

1 1 1 1 1 1 1 1 2 --> add to list

..

..

..

-Permutations – given list of distinct integers, return all possible permutations

Add a new arraylist to answer arraylist with each starting number in list as a starting number. Recurse and if the size of the list is still shorter than the num list, iterate through the num list and check to and for each one that is not currently in the list, create a new list with that number and recurse again. 



-Subsets – given a set of distinct integers, nums, return all possible subsets(the power set)

Backtracking – recurse through the indices of the numbers list and add the current index value to the list and recurse and increment the index. Then remove the current index value from the list and recurse and increment the index.

Follow the recursive calls:

[1, 2, 3]

1, 2, 3 --> add

1, 2 --> add

1, 3 --> add

1 --> add

2, 3 --> add

2 --> add

3 --> add

null --> add



-Restore IP Addresses – Given a string containing only digits, restore it by returning all possible valid IP address combinations. Valid 🡪 Four integers(each between 0 and 255) separated by a single dot.

Recurse through the string and track the current index. Append the current

substring of index, index + 1 character after, and index + 2 characters after,

checking the string for validity -- does not extend past end of the string and

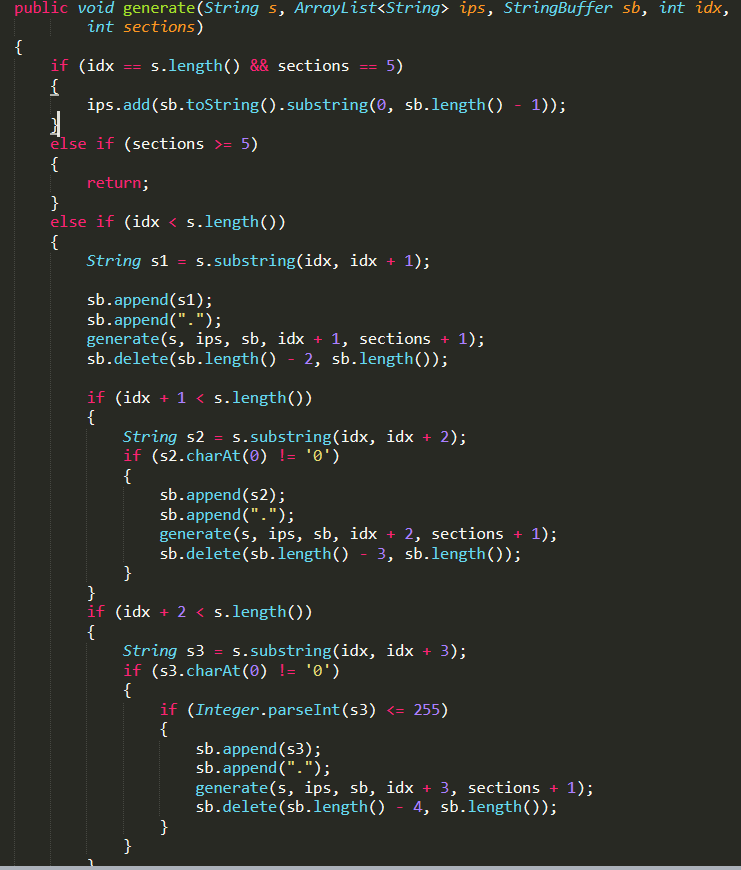
does not start with '0' and <= 255.

Also track the number of "sections", the sections between the ".". Once the section count

hits 5, starting at 1, it is the base case.

Backtrack after each recursive call by removing what was added i.e. 1 character,

and adding 2 characters and recursing and so on.



-Word Break II – Given a non-empty string s and a dictionary wordDict containing a list of non-empty words, add spaces in s to construct a sentence where each word is a valid dictionary word. Return all such possible sentences.

First check that each character in the string s occurs in the wordDict. <-- Key performance

Backtracking -- Starting at index 0, iterate through the wordDict and check if the substring of the current index to index equal to the wordDict word size. If so, recurse forward to that index and repeat the process while adding the string to the StringBuffer.

If the index equals the length of the string, add the string to the answer list. After each recursive call returns the first time, remove the current word from the end of the StringBuffer.



-Reverse Nodes in k-Group – Reverse the nodes of a linkedlist k at a time. If there are left out nodes at the end, return them in the original order.

Constant space and linear time. For each group, reverse it in constant space using the technique sentinel -> attach next here -> nodes previous. After function call, return 3 markers as an array, the beginning of the reversed chunk, the end of the reversed chunk, and the next node past the reversed chunk. These allow the proper placement of pointers in the list after reversal.



-Text Justification – Given an array of words and a width maxWidth, format the text such that each line has exactly maxWidth characters and is fully (left and right) justified. You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly maxWidth characters. Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right. For the last line of text, it should be left justified and no extra space is inserted between words.

Iterate through the words array. Count the number of strings that can be put into a single line, factoring in the at minimum 1 space in between each character. (Also factoring in the last word does not need a space after it.)

Special case: If only 1 word can go on the line, pad up until maxWidth after the word

with spaces and continue to the next iteration.

Otherwise - # of separations = # words - 1. The number of spaces between each word is

maxWitdh - #characters of words in line / # of seperations. Leftover =

#characters of words in line % # of separations. For each separation at the

start, add 1 space on top of number of spaces if there is still leftover. After adding

the extra space, decrement leftover count.

Remove the last line and separate each word with 1 space and pad the end with extra spaces until the line equals maxWidth.

-Word Ladder – Given *beginWord* and *endWord* and a dictionary word list, find shortest transformation from start to end. Only one letter can be changed at a time and each intermediate word must be in the word list.

BFS – put the beginning word into the queue. Add a new arraylist to the map with the word as the key and the arraylist with the word in it. For each current word, iterate through each character and change each character to each letter in the alphabet. If the new word exists in the set, get the arraylist of the previous word, the arraylists are the chains of word sequences. Create a new arraylist from the previous word arraylist and add the new word to it. Put the new chain arraylist into the map with the new word as the key Put the new word into the queue. If the current word equals the end word, get the arraylist from the map with the endword as the key and return the length as the chain length.

-Longest String Chain – List of words – *word1* is predecessor of *word2* if one letter can be added anywhere in *word1* to make it *word2.* All intermediary words must be in the list. Return the longest chain.

Sort the words by increasing length. Create a HashMap of lengths to Sets of Strings with that length. Create a HashMap of Strings to ArrayLists that are the sequences of strings up to the key String.

For each string *s* in the words list, iterating in ascending length order, remove one character from each spot in the word. If the shorter word exists in the map of lengths to sets of strings, get the sequence(arraylist) of the shorter word. Copy it, and add the existing longer word to the new arraylist. Make a set if it doesn’t exist for words of the length of *s* and add *s*  to the set and put it in the HashMap of length to sets of words. Add the newly created sequence to the sequences HashMap with the *s*  as the key and the arraylist as the value.

If none of the words with a single character removed exists the sequences or lengths, add the current string to a set with the length as a key and the set as the value to the lengths HashMap. Add a new arraylist with the current word as the key and the arraylist with only the current word as the value. Add that to the sequences HashMap.

At the end, iterate through all the sequences HashMap and return the length of the longest ArrayList.

**Trees:**

-Binary Search Tree Iterator(System Design) – Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST. Calling next() will return the next smallest number in the BST. next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

Create a stack that initially contains all values of calling root.left until the leaf is reached. After each pop from a next() call, check if the current popped off node has any right children. If not, continue, otherwise add the current node's right child and calls to root.left until it hits a leaf. Continue this process.

The stack contains a one slice height of nodes in the tree.

Worst case average for next() is O(n) for completely unbalanced tree to the left. The populate\_stack() function only gets called for nodes with right children and only runs for O(n) in the case of completely skewed trees.

**-** Serialize and Deserialize Binary Tree(System Design) – Convert binary tree to string and back to binary tree.

Serialize – DFS with stringbuffer. Counter integer as ID of current node(need to use an array so that the it is a pointer and the value increments through all recursive calls. String is “root.val:parent\_id:counter[0]:left “. Parent ID is passed through the recursive call since it is DFS. Node ID is counter[0] which gets incremented in each recursive call. Left is a Boolean and gets set from the caller function as true when it calls the left child and false on the right child.

Deserialize – Split the string data on the “ “. Make a new TreeNode for each node data and put a it in a HashMap with key as the ID and the node as the value. Iterate through all the nodes, and grab the parent ID from the node data in the string and the parent node using the parent ID from the HashMap and grab the child node with the child id from the node HashMap. Set the parent child node to the appropriate left or right child based on the left flag true or false.

-Lowest Common Ancestor of a Binary Tree – Find the Lowest Common Ancestor of two given nodes in a binary tree. LCA(lowest node that has both p and q as descendants, where a node can be a descendant of itself)

Recurse in a DFS search down the tree. If the current node is null, return null. If one of the target nodes is hit, return that target node. If both the left and right child return not null then the current node is the LCA, return the current node, otherwise, return the non-null node.

The LCA will get propogated to the top because, once both left and right children return non-null nodes, the current node is the LCA and gets returned up the recursive stack. Once the stack returns to each respective node, if it is a parent of both target nodes but not the LCA, one of the children will return back a null node. That call will always return the non-null node which will be the LCA.

Out of scope of the problem, but if the target nodes were *not* guaranteed to be in the tree, a one pass over the tree would need to be conducted at the start to check for both nodes.



-Binary Tree Right Side View - Given a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

**Example:**

**Input:** [1,2,3,null,5,null,4]

**Output:** [1, 3, 4]

**Explanation:**

1 <---

/ \

2 3 <---

\ \

5 4 <---

DFS down the tree with an ArrayList. If the arraylist size >= depth + 1(0 depth at root), set the index to the value, otherwise append onto the end of the list(first time current depth has been reached). DFS naturally traverses left to right if left child is called recursively first.

-Binary Tree Vertical Order Traversal – Return vertical traversal(top to bottom, column by column). If two nodes are in the same row and column, the order should be left to right.

Create a new class Pair that is the node and integer offset from the center. BFS of Pairs and for each left and right child create a new pair that tracks both the node and the offset. If offset is negative, put it in the appropriate indexed arraylist for left nodes and if it is non-negative put it into arraylist for right nodes properly indexed. Reverse the left arraylist and append the right arraylist TreeNode values.

**Input:** [3,9,8,4,0,1,7,null,null,null,2,5] (0's right child is 2 and 1's left child is 5)

3 **Output:**

/\ [

/ \ [4],

9 8 [9,5],

/\ /\ [3,0,1],

/ \/ \ [8,2],

4 01 7 [7]

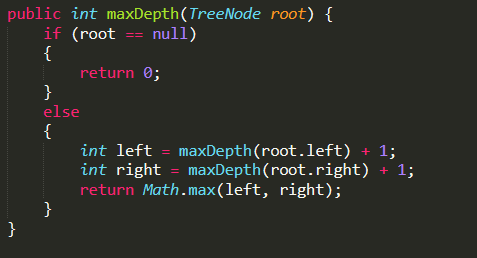
/\ ]

/ \

5 2

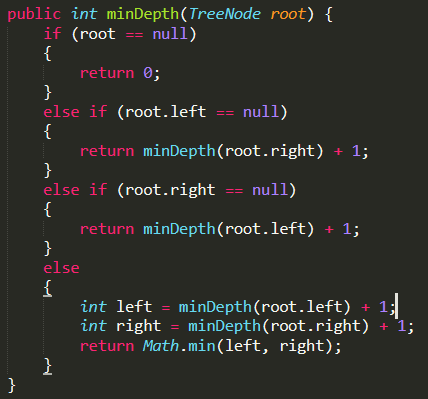
-Maximum Depth of Binary Tree – Given a binary tree, find its maximum depth. The longest path from the root node down to the farthest leaf node. A leaf node is a node with no children.

If a current node is null, return 0. Otherwise, recurse on the left and right children and increment both return values by +1. Return the maximum between the two.



-Minimum Depth of Binary Tree – Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from the root down to the nearest leaf node. A leaf node has no children.

If the current node is null, return 0. Otherwise, if the left child is null, recurse on the right child and return the return value + 1. If the right child is null, recurse on the left child and return the return value + 1. Otherwise, recurse on both left and right, increment both by +1 and return the minimum between the two.



-Binary Tree Level Order Traversal -- Given a binary tree, return the level order traversal of its nodes'

values. (ie, from left to right, level by level).

DFS to traverse the tree. Track the depth through a function argument and at each depth

check if the current depth is in the arraylist by making sure the size of the arraylist

is greater than the current depth. Append the node onto the current depth's arraylist.

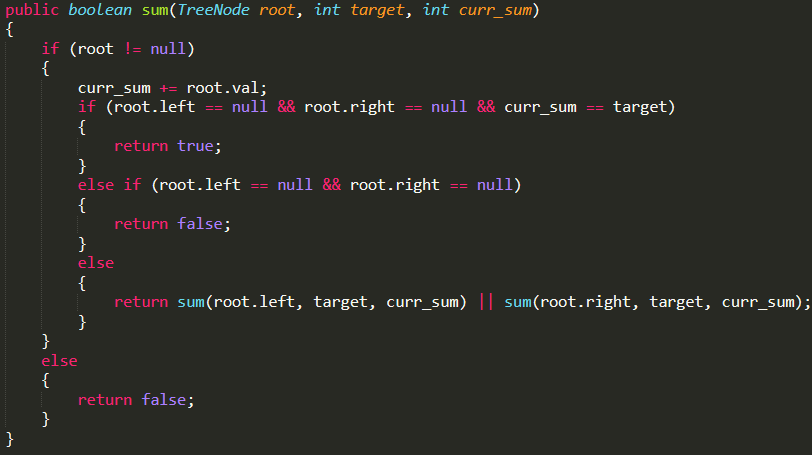
Left to right occurs by calling left child before right in DFS.

-Path Sum – Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

DFS down the tree and for each call to the child node, pass the curr\_sum + root.val.

If both child nodes are null --> leaf node, check if sum = target. Return true, otherwise

return false from the leaf node. At non-leaf node, return left recursive call || right recursive call because only one needs to match to target. That return gets passed up to tree as one of the parents’ children return calls.



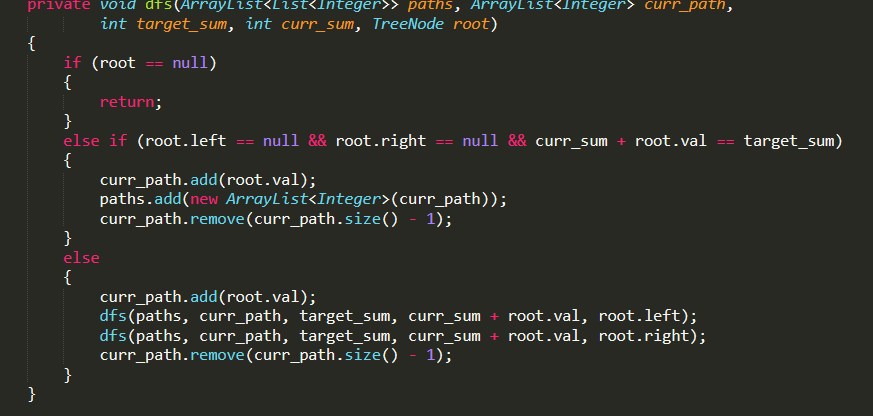
-Path Sum II -- Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

DFS down the tree with a tracked path ArrayList and current sum. Increment sum with

current node value at each recursive stack frame. Once a leaf is reached, if it equals the target, copy the list into another list and put into the answer list.

Prior to each recursive call stack returning, remove the current node from the path

arraylist.



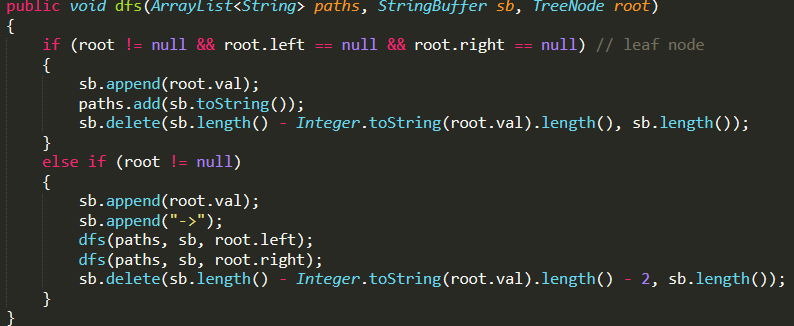
-Binary Tree Paths – Given a binary tree, return all root-to-leaf paths.

DFS down the tree and append the current state with "->" afterwards. Once a leaf is hit,

append the leaf node value and add the string to the list. After the return from a DFS call,

and prior to returning from the current stack call, revert the state back(cleanup) by deleting

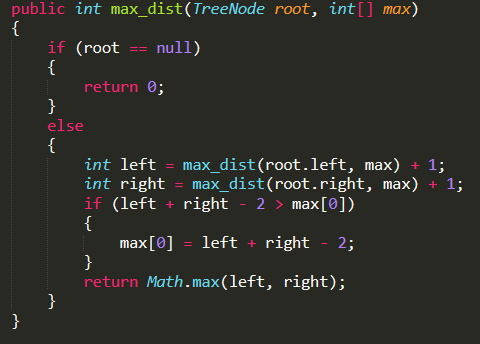
the appropriate end indices of the current stack frame state in StringBuffer.



-Diameter of Binary Tree -- Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the longest path between any two nodes in a tree.

At each node, calculate the max distance between the left and right subtree. --> Calculate max left height and max right height and sum them together.

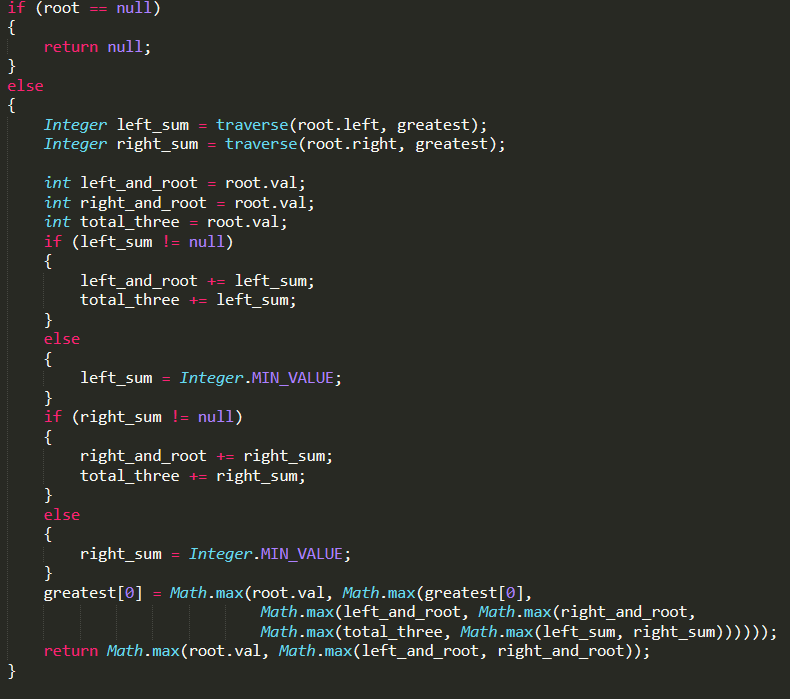
Set a tracker for max difference seen so far. i.e. Array of size 1 as input value since no pass by reference in Java for primitive types.



-Binary Tree Maximum Path Sum – Given a non-empty binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path must contain at least one node and does not need to go through the root. The path value is the sum of the node values along the path.

At each node, take the maximum of the total on the left + root, total on right + root, root, sum of left, right, and root, and the current seen maximum and set the current maximum to the max over those values. Return after each recursive call the maximum of left + root, right + root, or root.



-Subtree of Another Tree – Given two non-empty binary trees s and t, check whether tree t

has exactly the same structure and node values with a subtree of s.

If the root of the subtree and the current node in the larger tree have the same value

check if the smaller tree and the current subtree are equal. Recurse through both trees

checking that that the smaller tree and the current subtree are equal.

-Average of Levels in Binary Tree – Given a non-empty binary tree, return the average value of the nodes on each level in the form of an array.

DFS through the tree and add all nodes to an arraylist of arraylists where each arraylist is all nodes in that level.

-Two Sum II – Input Array is Sorted – sorted array in ascending order, find two numbers that add up to a specific target.

Two pointer approach for finding target in sorted array – pointer at beginning, pointer at end. If sum is greater than target, decrement right pointer, if it is smaller, increment left pointer. If the pointers cross then the sum does not exist.

-Palindrome Permutation – Given a string determine if a permutation of it could form a palindrome.

If the number of characters is odd, ensure that all character counts are even except 1 character which is one. If the count is even, all character counts must be even.

-Valid Palindrome – Given a string, determine if it is a palindrome ignoring cases and considering only alphanumeric characters.

Left and right pointer from end and start. If a character is invalid, move the respective pointer towards the middle. Only check if both pointers are valid when comparing characters.

-Valid Palindrome II – Given a non-empty string, at most one character can be deleted. Judge whether it can be made a palindrome.

Two sets of left and right indexes from each end. Keep moving both sets of pointers inwards while the characters are equal. Otherwise if only one set is equal, move that set of pointers inwards. If the first set is not equal and the increment flag has not been set yet, move the first set left pointer in and second set right pointer inwards. Otherwise, if neither set is equal and increment flag has already been set, return false. At the end return true.

-Delete Node in a Linked List – delete a node in a linkedlist given only access to that node

Copy next node’s data to current node. Move nodes next pointer to next.next. Cannot accomplish given access to tail node.

- Meeting Rooms – Given array of meeting time intervals as start and end times [s1,e1]…, determine if a person can attend all meetings.

Sort the intervals by ascending start time. Any overlapping intervals will now be adjacent. Iterate through the list from the beginning and check neighboring intervals at each index if they overlap.

-Meeting Rooms II – Array of meeting time intervals with start and end times *[[s1, e1],[s2,e2]…]* s.t. *(si < ei)*, find the minimum number of conference rooms required.

Sort the intervals by ascending start time. Add the first interval into a minheap with the ending times as the key. Iterate through the sorted intervals starting at the second interval. If the head of the heap has an end time less than or equal to the current interval, pop off the head. Otherwise do nothing. Add the current interval’s end time into the heap. At the end, the size of the heap is the number of conference rooms required.

Each entry in the priority queue represents an occupied room. Popping off the head represents a room becoming free. Adding an entry into the queue represents a new room being allocated. Since the rooms are sorted by ascending start time, and the queue is by first room free, it is a simulation of the actual progression of meetings.

-Merge Intervals – Given a collection of intervals, merge all overlapping intervals

Sort array by ascending start time. All overlapping intervals will now be adjacent. Merge by starting to iterate from the beginning of the array.

-Partition Labels – a string *S* of lowercase English letters is given. Partition the string into as many parts as possible so that each letter appears in at most one part, and return a list of integers representing the size of those parts.

Create intervals where each interval start is the index of a letter’s first occurrence and the end is the index of the last occurrence of that letter. Sort the intervals by start time and merge intervals that overlap(overlaps will be in contiguous sections). Return the length of intervals after the merge.

-Interval List Intersections – Given two lists of pairwise disjoint closed intervals, return the intersection of the two interval lists.

Intervals overlap if I1\_start <= I2\_start && i1\_end >= i2\_start || i2\_start <= i1\_start && i2\_end >= i1\_start.

Interval intersection is max of the two starts and min of the two ends. Increment the pointer of the list with the interval that has earlier end time. If they are equal, increment both.

-Print Immutable LinkedList in Reverse

Recurse until the end of the list, print after the recursive call so it prints out while moving back up the call stack.

if(head != null)

{

printLinkedListInReverse(head.getNext());

head.printValue();

}

-Longest Palindromic Substring – Given a string **s**, find the longest palindromic substring in s.

Palindromes mirror around its center. There are 2n – 1 centers in the string: n centers that are centered on each character in **s**, and n – 1 centers on the whitespaces between the characters(they must be even length palindromes). Iterate through each center and expand in the left and right directions. Check for the longest palindrome.

int left;

int right;

if (i % 2 == 0) // on an actual character

{

left = i / 2;

right = i / 2;

}

else // in between characters

{

left = i / 2;

right = i / 2 + 1;

}

while (left >= 0 && right < length)

{

if (s.charAt(left) == s.charAt(right))

{

if (right - left + 1 > max)

{

max = right - left + 1;

l\_i = left;

r\_i = right;

}

--left;

++right;

}

else

{

break;

}

}

-Verifying an Alien Dictionary – Given a sequence of *words*, and the *order* of the alphabet, verify if the given sequence of *words* is sorted lexicographically in accordance with the *order*.

Store a HashMap of each letter with their corresponding order by iterating through the *order* string sequence. Find the longest string and store that as max\_length. Iterate through each word in the list and compare it to the previous word in the list. For each compare, iterate from 0 to max\_length, and compare each letter to ensure the one that comes before is either less than or equal to the one that comes after.

-Move Zeroes – Given an array, move all 0’s to the end of the array while maintaining the relative order of the non-zero elements. Do it in-place and minimize the total number of operations.

Count the number of zeros. Iterate through the beginning of the array and move all nonzero elements to the front. Zero of the back by counting the number of and stopping when it hits the zero count. *N* operations.

-Sliding Puzzle – 2x3 board with each tile represented by integers 1-5 and a square represented by 0. The 0 can move 4-directionally adjacent(up, down, left, right). The board is solved if it is in state *[[1,2,3],[4,5,0]]*. Given a puzzle board, return the minimum number of moves required to solve it. If it is impossible, return -1.

Use BFS by expanding each state to the next possible states. Encode the 2D board state as a 1D arraylist and convert between the index and the coordinates. Use a HashMap to track the number of moves to each state, and use the HashMap keys(the arraylist states) to ensure no prior visited state gets enqueued.

-Find All Duplicates in an Array – array of integers *1 <= a[i] <= n* (*n* = size of array), some elements appear twice and others appear once. Find all elements that appear twice in constant space and linear time.

Use the indexes of the arrays as keys. Since the value of an array element will always lie in the range of 1 – size of the array, each time an element with a value *i* is encountered, multiply the subsequent value in index *i* by -1. i.e. *a[x] = i* 🡪 *a[i] = a[i] \* -1*. One pass through array. While iterating through, if *a[i] <* 0 where a*[x] = i,* then *i* has already appeared before so the current value *i* is the second occurrence of it so add it to the return list as a value that appears twice.

-Find All Numbers Disappeared in an Array – array of integers  *1<= a[i] <= n (n = size of array)*, some elements appear twice and others once. Find all elements [1,n] inclusive that do not appear in the array in constant space and linear time.

Iterating through the array, at index *i,* if the *a[absolute value(a[i])]* is positive, set the value in that index to negative. If it is already negative, do nothing. All indices *i* that contain positive value afterwards, the value *i* is a number that did not appear in the array.

-First Missing Positive – given an unsorted integer array, find the smallest missing positive integer.

Find the minimum positive integer in the array, if it is greater than 1, return 1. Set all negative numbers and 0 to a dummy value such as *Integer.MAX\_VALUE.* Iterate through the array and if *a[i] – min\_value* is within the bounds of the array and a positive value, set the index *a[a[i] – min\_value]* to a negative, \* -1. Find the first index *i* that holds a positive value and return *min\_value + i.* If the whole array has negative values, return the minimum value added to the length of the array.

-Missing Number – Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array.

Have a flag to check if the last number i.e. nums.length is present. Set the value of at the index where index = nums[i] to the negative of the value. If the value is 0, set it to min integer value. Check which index is still not negative, return that index.

**Math:**

-Arranging Coins – n coins to form a staircase shape, where k-th row must have exactly k coins. Return number of full staircase rows that can be formed.

¤

¤ ¤ sum i = 1 to i = x: x(x+1) / 2

¤ ¤ 2n >= x(x+1)

Completing the square: 2n >= x^2 + x 🡪 (x + ½) ^2 – ¼

x = floor(sqrt(2n + ¼) – ½)

-Reverse Integer – Given a 32-bit signed integer, reverse digits of an integer.

Take the absolute value of the number. Edge Case - If the value is the minimum integer value, -2^32 absolute value will still be negative, it will overflow if reversed, return 0. Otherwise, pick each digit off from the last significant digit one by one. Do this by % 10. Remove the LSD by dividing by 10 after each loop iteration. Multiply the reverse value by 10 prior at the start of the loop. Add the LSD to the current reversed value.

To check for overflow, divide by 10 after each multiple and subtract what was added on to see if they equal to previous value before the operation was performed.

**Greedy:**

-Jump Game – Given an array of non-negative integers, you are initially positioned at the first index of the array. Each element in the array represents your maximum jump length at that position. Determine if you are able to reach the last index.

Keep track of the furthest index that can be while iterating through the array. Start at index 0, the furthest that can be reached is 0 + nums[0]. If the current index is <= the furthest, update the furthest with the max of the furthest and the current index + nums[current index]. If i > furthest index, return false. Otherwise if the furthest >= last index, return true.

**Graph:**

-Critical Connections in a Network – There are n servers numbered from 0 to n-1 connected by undirected server-to-server connections forming a network where connections[i] = [a, b] represents a connection between servers a and b. Any server can reach any other server directly or indirectly through the network. A critical connection is a connection that, if removed, will make some server unable to reach some other server. Return all critical connections in the network in any order.

Tarjan's Algorithm

Create an adjacency list for the graph -- for each vertex have a list of all connected vertices.

Have an array the size of the number of vertices that will store the low value, or the lowest value that each vertex is connected to. Values will be assigned to vertices in the order that they are visited.

Track an array of visited vertices so that DFS does not recurse to vertices already visited. Have a timer that increments in each call that tracks visit order.

Choose any node to start the DFS with. For each node that it is connected to, mark it as visited, set its current low\_val equal to the timer(order visited). Iterate through all connected nodes in the adjacency list and if it is the parent, skip it.

If the next node has not been visited, recurse to it. Set the current low value to the minimum of

the current node low value and the neighbor nodes low val.

If the low value of the neighbor node is less than the current id, it is a critical edge.

Intuition: After an update, the lowest value node the neighbor has seen is greater than the

lowest value node the current node has seen, hence previous nodes that have been visited are connected to the neighbor node ONLY through the current edge.

Otherwise if the node has already been visited, update the current low value with the minimum of the current low value and the neighbor low value.