Algorithms(JS):

Saturday, April 27, 2024 6:06 PM

Algorithms (Javascript):

Easy Category:

Problem 1 Two sum:
Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.
Learnings:
javascript has a new Map() class which has attributes .has() which returns a boolean of whether the map keys has that value o r not and .set() which takes 2 parameters as input, a key and a value and sets them. We can also use for in loops in javascript with the syntax: for (const index in nums) {}
Methodology:
I made a hash map, firstly, I iterate through each number and check if target - that index exists in map, if it does, then that pair (current and value of pairTarget) is returned, and I set the map after this with nums[index] and index.

Problem 2 Palindrome Number:
Given an integer x, return true if x is a palindrome and false otherwise.
Learnings: 0 is a palindrome, -ve numbers and non zero leading zero numbers are not palindromes. Math.floor() function exists, can return true or false with "temp===reverse" statements. x.toString() is a function that converts number to string. X2.length is a string function which returns length of string. Use let in a for loop instead of const since the variable is changing.
Methodology:
Declare 2 variables, temp and reverse. Modulus temp by 10 to get last digit, multiply reversed with 10 and add that digit to get the reversed of a number and keep doing it while temp > reversed (to only reverse half number) and return if temp and number are equal (for even digits) and temp == reversed/ 10 (for odd digits)

Problem 3 Roman to Integer:
Given a roman numeral, convert it to an integer.
Learnings:
a map can be defined as a constant for multiple constant numbers like: const map = {'I' : 1, 'V' : 5 } etc. also there are st r.indexof(substr) function which returns index of the substr or -1 if not found and str.replace(substr1, substr2) which replaces substr1 with substr2 in str (this is not in-place, this returns a new string which is to be re-assigned)
Methodology:
Firstly, I tried incorporating numbers like "IV" etc into the hash map as well and then checking their index, removing them f irst and adding their value and then running simple for loop on remaining singular indexes. A better way to do this with respect to this problem is to get 2 numbers, the current and next and if current is less than next, that means we are in a IV situation, we can add total += next - curr and increment I to skip the next character.

Problem 4 Longest Common Prefix:
Write a function to find the longest common prefix string amongst an array of strings.
Learnings: to sort an array, the function is array.sort(function(a,b) { return a.length - b.length; }); (note: this one was for string arrays). Also, need to develop edge cases thinking as well, in this case, edge cases were "a single string", "empty string", "multiple empty strings", "same string multiple times" for that, separate if conditions and sorting was needed before main algorithm
Methodology: it was simple, for all elements, if their first characters are same (kept check through a boolean variable) then proceed to n ext character, exit this while if smallest substring length reached or if boolean returns false on any single mismatch.

Problem 5 Valid Parentheses:
Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
Learnings:
we can return true or false with conditions like "return a==2" returns true if a is first prime number etc. the method to mak e const map = { ')' : '(' } is an object, and to access its values, we can use Object.values(map) and it returns an array of all the values (not keys). And then, we can use the arr.includes() f unction to check if the array has the thing specified. Similarly, map.hasOwnProperty() function is used to check if the map keys have the specified value. Also, we can use stacks (as an arrays) with declring it as const stack = []
and then use stack.push() or stack.pop() methods respectively. Also, its length is found using stack.length.
Methodology:
Basically just make an empty stack and a hash map of closing : opening bracket pairs. Then iterate through each character of s, a character is included in the values (opening bracket) then push it to stack. Or if it is a closing bracket, then if stack has nothing (checked by !stack.length) or that character is not equal to stack.pop then return false. And at the end, return stack.length === 0 to check if anything isnt left in the stack.

Problem 6 Merge Two Sorted Lists:
You are 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.
Learnings:
So, in Javascript, to make a ListNode, use the function:
function ListNode(val, next) { this.val = (val===undefined ? 0 : val) this.next = (next===undefined ? null : next) }
And the ListNode can be defined as: let dummy = new ListNode(); also, to append remaining list1 or list2 to the end of sorted list, instead of checking which is null, just do: remlist = list1 || list2. And obviously other things are the same old as: Curr.next = list1 or Curr.val = list1.val. Also, if you want to switch values of 2 variables (lists in this case), do: [list1, list2] = [list2, list1]
Methodology:
There were 2 methodologies. First one is the normie one where you make a dummy list, check which list1 or list2 has smaller v alue, assign dummy.next = list1 accordingly then increment the selected list (list1 = list1.next) in the if condition and dummy (dummy = dummy.next) outside if conditions in while loop to progress and use the list1 || list2 line to append the remaining list. Another fun method is the recursion method. Base case is if either of list is null (!list1 || !lis t2) then use ternary operator to return list1 or list2 accordingly. Now this method only returns list1, so we have to ensure list1 is the smallest one, for that, we place a check if list1 great er, then switch list1 and list2 and list1,next = recursive call with list1,next and list2. and finally, the head of the list is at list1 (since the recursive calls are what built the list, so list1 is returned).

Problem 7 Remove Duplicates from Sorted Array:
Given an integer array nums sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Then return the number of unique elements in nums.
Learnings:

If you revise your code multiple times, it will get efficient. Look for any variables that contribute nothing and can be remo ved. Previous concepts of stack, push, includes were used.
Methodology:
Make a unique array, run a for loop against nums, keep a k variable initialized to 0, if unique does not include nums[i] then push it to unique, assign nums[k] to unique[k] and increment k. then simply return k.

Problem 8 Remove Element:
Given an integer array nums and an integer val, remove all occurrences of val in nums in-place. The order of the elements may be changed. Then return the number of elements in nums which are not equal to val.
Learnings:
If you want to store something at an index and then increment index, you can use pre/post increment operators like: arr[count ++] = nums[i] etc.
Methodology:
Same as problem 7, run a for loop for each element and keep a counter, if element not value, assign it to the counter place i n nums and increment counter

Problem 9 Find the Index of the First Occurrence in a String:
Given two strings needle and haystack, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.
Learnings:
Str.substring(start,end) method takes arguments of starting and ending positions in the string.
Methodology:
Loop through the haystack, at every index, separate substring of size needle (I, i+needle.length) and compare it with needle if it matches, return i. outside the for loop, return -1

Problem 10 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 ind ex where it would be if it were inserted in order.
Learnings:
There is also an iterative approach to using binary search. Make left and right pointers and run a while loop until left < ri ght. Then proceed as usual.
Methodology:
Made 2 pointers for left and right, calculated mid in loop, if target > nums[mid] then left = mid + 1 else right = mid and th en finally return left. Outside of the while loop

Problem 11 Length of last word:
Given a string s consisting of words and spaces, return the length of the last word in the string.
Learnings:
To reverse a string, use a composition of the following builtin functions: str.split("").reverse().join(""); it is to be not ed that reverse() works only on arrays. Also, to remove whitespaces, use str.trim() function.
Methodology:
Reverse the string, trim it and run a while loop until " " is seen, keep concatenating last word and return its length. Optim izations: why keep record of word at all? Just keep incrementing counter and return it

Problem 12 Plus One:
You are given a large integer represented as an integer array digits, where each digits[i] is the ith digit of the integer. T he digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's. Increment the large integer by one and return the resulting array of digits.
Learnings:
For exponent, javascript takes it as ** operator. Although BigInt is a thing in javascript, it can only be used with initiali zation. Problems like this one can have integers not within the range of a normal variable, so write efficient code accordingly. Arr.Unshift(num) is a function that adds a number at the sta rting of an array in javascript.
Methodology:
I first decided to construct back the original number in the array by multiplying it with 10 power its position using a for l oop, adding 1 to it and then breaking it down by taking modulus and divison by 10. this approach failed as larger numbers (arrays) were introduced, so now the improved approach is t o use recursion to start calling from the right most (length-1) index of array and add 1 to it if it is less than 9, and return the array (base case). If it is greater than 9, then assi gn 0 to the current position, (place a check here if we are at left most index (index 0) and use the unshift function to add 1 at the start of the array, this means number was 99…9 and ret urn array) and recursively call function at the curr-1 index.

Problem 13 Add Binary:
Given two binary strings a and b, return their sum as a binary string.
Learnings:
'0b{a}' can be used to convert a binary string to a BigInt number, then you can add them using BigInt function and finally, t oString(2) converts the final answer to a binary string. This was a simple easy approach, but originally, this is to be done in the way said below. Also, if you want to add something to l eft or right of string, just use a + str or str + a accordingly.
Methodology:
We can start by aligning the two binary numbers by adding leading zeros to the shorter string so that both strings have equal lengths. Then we can add the digits from right to left (using a for loop that goes from length-1 to 0. at each iteration, manually cover all cases for eich bit being either 0 or 1. since we have 3 variables (a,b,carry) t his will result in 8 if conditions) and keep track of any carry generated. Finally, we add the carry to the leftmost position if any.

Problem 14 Sqrt(x):
Given a non-negative integer x, return the square root of x rounded down to the nearest integer. The returned integer should be non-negative as well.
Learnings:
Okay so…. Theres a way to find sqrt, from the number, keep subtracting consecutively increasing odd numbers and the integer p art of square root is the number of times the number was subtractable without going in -ve.
Methodology:
Exactly, that, I made 3 variables: subtractable (boolean), num (for consecutively increasing odd numbers) and count (to count the number of times it was subtractable) then I used a while loop with subtractable variable and within it, checked if x is less than 0 after updated and switched the boolean value . Else, incremented count by 1.

Problem 15 Climbing stairs:
You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many dis tinct ways can you climb to the top?
Learnings:
Okay so for this question, without even looking at the code solution, I began developing intuition. We can only make 2 moves. Climbing 1 stair or climbing 2 stairs. Think of it like this. If we have 2 stairs, we can make 2 moves. If we make step=1, then remaining stairs are 1 and that can be covered in 1 steps. If we make 2 steps, then the remaining stairs are 0 and that can be covered in 0 steps. So basically, for stairs =1, there is 1 way and for stairs =2, there are 2 ways. If stairs ar e 3, then if we climb 1 step, the remaining is 2 (=2) and if we climbg 2 steps, the remaining is 1 (=1) so in total, we have 2+1 =3 ways. To solidify this, if num stairs is 4, then 1 step r emaining stairs are 3 (=3) and if we take 2 steps, the remaining stairs are 2 (=2) so total = 3+2 =5. This is called dynamic programming, and this resembles the fibonacci sequence. One more thing, this question also involved memoization and for that, we can just use a simple Array defined by: const dp = new Array(n+1).fill(0);, dp[0]=1; dp[1]=1; and so on.
Methodology:
Used memoization, in a for loop from 2 to n, the ith index value of array is i-1 (1 step) + i-2 (2 steps), then simply return nth index

Problem 16 Remove Duplicates from Sorted List:
Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked lis t sorted as well.
Learnings:
In the while traversal loop, insteal of checking if curr.next != null, you can simply put condition while (curr && curr.next) also, in js, curr.next.next is a valid thing.
Methodology:
Pretty much the same as traversal tbh, make a new variable starting at head to traverse, while it and its next node is not nu ll, keep traversing (you traverse by curr = curr.next in the while loop) and just place a check for this question if curr.value is the same as curr.next.value then curr.next = curr.next. next (skip the middle repeated value node and move on to the next one)

Problem 17 Merge Sorted Array:
You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2 respectively. Merge nums1 and nums2 into a single array sorted in non-decreasing order. (in-place for nums1, size will be m+n)
Learnings:
If you want to shift numbers to the right, start from the very end (length-1) and put i=i-1 indexes and so on (for in place adjustments). For this question, there were some edge cases that I realized while dry running and thus, from now on, it is better to have a rough register to sketch my solutions first.
Methodology:
Used 2 pointers and a while loop, while both pointers were less than their boundries (m and n), if nums1 was less than just i ncrement itr1 as it has to stay in place, else, run a for loop to shift all numbers in nums1 1 digit to the right and then assign itr1 index to nums2[itr2] and increment all pointers (itr1 ,itr2 and m (to accommodate increased size)) lastly, if any left in nums2 (itr2<n) then add them to the ending of nums1. Using javascript, a more optimized solution would have been to append nums2 to end of nums1 and use the builtin sort function.

Problem 18 Binary tree InOrder Traversal:
Given the root of a binary tree, return the inorder traversal of its nodes' values.
Learnings:
We cant think such things, its better to remember them. InOrder traversal is left,val,right. Preorder traversal is val,left,r ight and postorder traversal is left,right,val.
Methodology:
These are done using recursive functions where argument is root, if root is null, then return empty list else for inorder, el se, recursive call on left node, append value, recursive call on right node and then return result.

Problem 19 Same Tree:
Given the roots of two binary trees p and q, write a function to check if they are the same or not.
Learnings:
Was different, first of all, if the if condition executes just one line, then simply don't use {} brackets and write the line infront of condition. And to return on multiple conditions, write && in return statement
Methodology:
First check if both null then true, if either of them null (OR) then return false, else, return val1==val2 && recursive call on left and recursive call on right.

Problem 20 Symmetric Tree:
Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).
Learnings:
Take a notebook and a pen and start sketching possible test cases and edge cases that come to your mind. Develop a rough algo rithm and then start debugging it according to those test cases. To declare an arrow function, do const FuncName = (arg1, arg2) => { //funct body}
Methodology:
It was easier to make another function isSame, which took input left and right nodes to compare, it checked if exactly one of the nodes is null or if their values don't match, then return false, If both are null, return true, but if both their values are same, then return isSame(left.left,right.right) && isSame(left.right,right.left). Here, we need to take care of which parent node's which child node we are passing as arguments, as they are supposed to be mirror nodes. Now in the main function , just check for edge case if root is null then return true, else, call isSame function on left and right nodes.

Problem 104 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 lon gest path from the root node down to the farthest leaf node.
Learnings:
The doing it all on paper first approach worked here, I was able to foresee possible execution steps and correct them before submitting. Now, this uses an approach to calculate height of tree (discussed in methodology) the key learning here is that approach can be shortened to: return Math.max(maxDepth(root. left), maxDepth(root.right)) + 1. and returning 0 if null root found. Another thing, null in javascript is represented by all lowercase null.
Methodology:
Useful algorithm to find the height of a tree recursively, uses 2 arguments, node and height if both left and right null, the n return height. If left exists only then return recursive call to node.left, h+1. similar if only right exists, call with right node. But if both exists, then call both of them using a Math.m ax function or a ternary operator (to return the larger of the 2 values)

Problem 108 Convert Sorted Array to Binary Search Tree:

Problem 111 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 no de down to the nearest leaf node. Note: A leaf is a node with no children
Learnings:
Not as such, same as Problem 104.
Methodology:
Same as problem 104, just reverse the condition at the end. (instead of return maximum depth, return minimum depth)