ANAGRAMS

```
// --- Directions
// Check to see if two provided strings are anagrams of eachother.
// One string is an anagram of another if it uses the same characters
// in the same quantity. Only consider characters, not spaces
// or punctuation.  Consider capital letters to be the same as lower case
// --- Examples
//   anagrams('rail safety', 'fairy tales') --> True
//   anagrams('RAIL! SAFETY!', 'fairy tales') --> True
//   anagrams('Hi there', 'Bye there') --> False

function anagrams(stringA, stringB) {
  const aCharMap = buildCharMap(stringA)
  const bCharMap = buildCharMap(stringB)

  if (Object.keys(aCharMap).length !== Object.keys(bCharMap).length) {
     return false;
  }
  for (let index in aCharMap){
      if(aCharMap[index] !== bCharMap[index]){
         return false;
      }
   }
   return true;

   // Better Solution
   // return cleanString(stringA) === cleanString(stringB);
}

// function cleanString(str) {
//     return str.replace(/[^\w]/g,"").toLowerCase().split('').sort().join('');
// }

function buildCharMap(str){
   var charMap = {};

   for (let char of str.replace(/[^\w]/g,'').toLowerCase()) {
      charMap[char] = charMap[char] + 1 || 1;
   }

   return charMap
}
```

ADD TWO NUMBERS

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */

var addTwoNumbers = function(l1, l2) {
   let carry = 0
   let head;
   let l3;
   while (l1 || l2) {
      let digit1 = 0
      let digit2 = 0

      if (l1) {
         digit1 = l1.val
         l1 = l1.next
      }

      if (l2) {
         digit2 = l2.val
         l2 = l2.next
      }
      let digit = (digit1 + digit2 + carry)%10
      carry = Math.floor((digit1 + digit2 + carry)/10)
      if (l3){
         l3.next = new ListNode(digit)
         l3 = l3.next
      } else {
         l3 = new ListNode(digit)
         head = l3
      }
   }

   if (carry){
      l3.next = new ListNode(carry)
   }
   return head;
};
```

Buy and Sell Stock

```
/**
 * @param {number[]} prices
 * @return {number}
 */
var maxProfit = function(prices) {
   let maxProfit = 0
   for (let i = 1;i < prices.length; i++){
      if (prices[i-1]<prices[i]){
         maxProfit += (prices[i]-prices[i-1])
      }
   }
    return maxProfit
 };
```

Buy and Sell Stock 2

```
/**
 * @param {number[]} prices
 * @return {number}
 */
var maxProfit = function(prices) {
   var maxProfit = 0
   var minPrice = Infinity
   for (let i = 0; i < prices.length; i++){
      if (prices[i] < minPrice){
         minPrice = prices[i]
      } else if (maxProfit < prices[i] - minPrice){
         maxProfit = prices[i] - minPrice;
      }
   }
    return maxProfit;
 };
```

Climbing Stairs

```
/**
 * @param {number} n
 * @return {number}
 */
var climbStairs = function(n) {
   let dp = [1,2]
   for (let i = 2;i < n; i++){
      dp[i] = dp[i-1]+dp[i-2]
   }
   return dp[n-1]
};
```

Convert Sorted Array to BST

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {number[]} nums
 * @return {TreeNode}
 */
function sortedArrayToBST(nums) {
   return addNode(nums, 0, nums.length - 1);
 }

 /**
  * @param {number[]} nums
  * @param {number} start
  * @param {number} end
  * @return {TreeNode}
  */
 function addNode(nums, start, end) {
   if (end < start) {
     return null;
   }

   // Use the middle element so the left and right halves will be about the same size
   const mid = Math.floor((start + end) / 2);
   const node = new TreeNode(nums[mid]);

   // Fill in the left and right subtrees
   node.left = addNode(nums, start, mid - 1);
   node.right = addNode(nums, mid + 1, end);

   return node;
 }
```

COUNT PRIMES

```
//O(n log log n)
// https://stackoverflow.com/questions/16472012/what-would-cause-an-algorithm-to-have-olog-log-n-complexity
// https://stackoverflow.com/questions/2582732/time-complexity-of-sieve-of-eratosthenes-algorithm
/**
 * @param {number} n
 * @return {number}
 */
var countPrimes = function(n) {
    if (n === 1 || n === 0){
        return 0
    }

    if (n === 2){
        return 0
    }
   let primes = []
   for (let i = 0; i < n; i++){
       primes[i] = true;
   }

    for (let i = 2; i < Math.sqrt(n); i++){
        if (primes[i]) {
            for (let j = i*i; j < n; j = j+i) {
                primes[j] = false;
            }
        }
    }
    let count = 0;
   for(let i = 2; i < n; i++) {
        if(primes[i] === true) {
            count++;
        }
    }
    return count
};
```

DELETE NODE FROM LINKED LIST

```
var deleteNode = function(node) {
    node.val = node.next.val
    node.next = node.next.next
};
```

EXCEL SHEET COLUMN NUMBER

```
/**
 * @param {string} s
 * @return {number}
 */
var titleToNumber = function(s) {
   let result = 0;
   for(let i = 0 ; i < s.length; i++) {
     result = result * 26 + (s.charCodeAt(i) - 64);
   }
   return result;
};
```

FACTORIAL TRAILING ZEROS

```
/**
 * @param {number} n
 * @return {number}
 */
var trailingZeroes = function(n) {
   let count = 0, x = 5;
   while (n >= x){
      count = count + Math.floor(n/x)
      x= x*5;
   }
   return count
};
```

FIRST UNIQUE CHARACTER

```
/**
 * @param {string} s
 * @return {number}
 */
var firstUniqChar = function(s) {
   let charMap = {}
   for (let char of s){
      if (charMap[char]){
         charMap[char]++
      } else {
         charMap[char] = 1
      }
   }
   for (let i = 0; i < s.length; i++){
      if (charMap[s[i]] === 1) {
         return i
      }
   }
   return -1
};
```

HAPPY NUMBER

```
/**
 * @param {number} n
 * @return {boolean}
 */
var isHappy = function(n) {
   var seen = {}
   while (n !== 1 && !seen[n]){
      seen[n] = true
      n = sumOfSquares(n)
   }

   return n===1 ? true : false
};

var sumOfSquares = (n) => {
   let sum = 0
   while (n>0){
      const digit = n%10
      sum = sum + digit*digit
      n = Math.floor(n/10)
   }
   return sum
}
// second solution using reduce for just sumOfSquares function
function sumOfSquares(numString) {
   return numString.toString().split('').reduce(function(sum, num) {
      return sum + Math.pow(num, 2);
   }, 0);
}
```

HOUSE ROBBER

```
var rob = function(nums) {
   if (nums.length === 0){
      return 0
   }

   if (nums.length === 1){
      return nums[0]
   }

   totals = [nums[0], Math.max(nums[0], nums[1])]
   for (i = 2; i < nums.length; i++){
      totals[i] = Math.max(totals[i-2] + nums[i],totals[i-1])
   }
   return totals[nums.length - 1]
};
```

INTEGER TO ROMAN

```
var intToRoman = function(num) {
   let romanMapArray = [
      [1000, 'M'], [900, 'CM'], [500, 'D'], [400, 'CD'], [100, 'C'], [90, 'XC'], [50, 'L'], [40, 'XL'], [10, 'X'], [9,
'IX'], [5, 'V'], [4,'IV'], [1, 'I']]

   let str = ''
   for (let entry of romanMapArray){
      if (num >= entry[0]){
         while (num >= entry[0]){
            str = str + entry[1]
            num = num - entry[0]
         }
      }
   }
   return str
};
```

INTERSECTION LINKED LIST

```
/**
 * @param {ListNode} headA
 * @param {ListNode} headB
 * @return {ListNode}
 */

 // Better Implementation
 var getIntersectionNode = function(headA, headB) {
   if (!headA || !headB){
      return null
   }

   let hANode = headA
   let hBNode = headB

   while (hANode !== hBNode){
      hANode = hANode.next
      hBNode = hBNode.next

      //
      // Any time they collide or reach end together without colliding
      // then return any one of the pointers.
      //
      if (hANode === hBNode){
         return hANode
      }
```

```
        //
        // If one of them reaches the end earlier then reuse it
        // by moving it to the beginning of other list.
        // Once both of them go through reassigning,
        // they will be equidistant from the collision point.
        //
        if (!hANode) {
            hANode = headB
        }

        if (!hBNode) {
            hBNode = headA
        }
    }

     // the answer has been found
    return hANode;
};



 // My Implementation
var getIntersectionNode = function(headA, headB) {
    let lengthA = 1
    let lengthB = 1

    if (headA === headB){
        return headA
    }

    if (!headA || !headB){
        return null
    }

    let current = headA
    while (current.next){
        current = current.next
        lengthA++
    }

    let tailA = current;

    current = headB
    while (current.next){
        current = current.next
        lengthB++
    }
```

```
    let tailB = current;

    if (tailA.val !== tailB.val) {
        return null
    }

    let diffLength = 0
    if (lengthA>lengthB){
        diffLength = lengthA - lengthB
        while(diffLength > 0){
            headA = headA.next
            diffLength--
        }
    } else {
        diffLength = lengthB - lengthA
        while(diffLength > 0){
            headB = headB.next
            diffLength--
        }
    }

    let currentA = headA
    let currentB = headB

    while (currentA){
        if (currentA === currentB){
            return currentA
        }
        currentA = currentA.next
        currentB = currentB.next
    }
};
```

INTERSECTION OF TWO ARRAYS

```
/**
 * @param {number[]} nums1
 * @param {number[]} nums2
 * @return {number[]}
 */
// My Submission
var intersect = function(nums1, nums2) {
    smallNumsMap = {}
    let small = []
    let large = []
    let result = []
    if (nums1.length > nums2.length) {
        small = nums2
        large = nums1
```

```
    } else {
        small = nums1
        large = nums2
    }

    for (let num of small){
        if (smallNumsMap[num] === undefined){
            smallNumsMap[num] = 1
        } else {
            smallNumsMap[num]++
        }
    }

    for (let num of large){
        if (smallNumsMap[num] > 0){
            result.push(num)
            smallNumsMap[num]--
        }
    }
    return result
};
```

LONGEST COMMON PREFIX

```
// Using Every Function
/**
 * @param {string[]} strs
 * @return {string}
 */
var longestCommonPrefix = function(strs) {
    if (!strs.length) {
        return ''
    }

     if (strs.length === 1){
         return strs[0]
     }

     result = ''

     for (let i = 0; i<strs[0].length ; i++){
         if (strs.every(str => str[i] === strs[0][i])){
             result = result + strs[0][i]
         } else {
             return result
         }
     }
     return result
 };
```

```
//Better Implementation
/**
 * @param {string[]} strs
 * @return {string}
 */
var longestCommonPrefix = function(strs) {
    if (!strs.length) {
        return ''
    }

    for (let i = 0; i < strs[0].length; i++){
        for (let str of strs){
            if (str[i] !== strs[0][i]){
                return str.slice(0, i);
            }
        }
    }

    return strs[0]
};

// My Implementation
/**
 * @param {string[]} strs
 * @return {string}
 */
var longestCommonPrefix = function(strs) {
    let commonPrefix = ""
    if (!strs[0]){
        return commonPrefix
    }
    for (let i = 0; i < strs[0].length; i++){
        for (let j = 1; j < strs.length; j++) {
            if (strs[0][i] !== strs[j][i]) {
                return commonPrefix;
            }
        }
        commonPrefix = commonPrefix + strs[0][i];
    }
    return commonPrefix
};
```

LONGEST PALINDROMIC SUBSTR

```
/**
 * @param {string} s
 * @return {string}
 */
var longestPalindrome = function(s) {
   if(s == null || s.length == 0) {
      return "";
   }

   let dp = []
   let res = ''
   for (let i = 0; i < s.length; i++){
      dp[i] = [];
   }
    for(let i = 0; i < s.length; i++) {
        for(let j = i; j >= 0; j--) {
           if (i -j < 2){
              if (s[j] === s[i]){
                 dp[j][i] = true
              } else {
                 dp[j][i] = false
              }
           } else {
              dp[j][i] = dp[j+1][i-1] && s.charAt(j) == s.charAt(i)
           }

           if (dp[j][i] && res.length < i - j + 1){
              res = s.slice(j, i+1);
           }
        }
    }
   console.log(res)
   return res
};
```

LONGEST SUBSTRING WITHOUT REPEAT

```
/**
 * @param {string} s
 * @return {number}
 */
var lengthOfLongestSubstring = function(s) {
    var seen = {}
    var maxLength = 0
    var start= 0
    for (let i = 0; i < s.length; i++) {
        if (seen[s[i]] !== undefined) {
            start = Math.max(start, seen[s[i]] + 1)
        }
        seen[s[i]] = i
        maxLength = Math.max(maxLength, i - start + 1)
    }
    return maxLength
};
```

LONGEST SUBSTR WITH 2 MOST DISTINCT CHARACTER

```
/**
 * @param {string} s
 * @return {number}
 */
var lengthOfLongestSubstringTwoDistinct = function(s) {
    let start = 0
    let seen = {}
    let maxlength = 0
    for (let i = 0; i < s.length; i++){
        seen[s[i]] = seen[s[i]] + 1 || 1
        while (Object.keys(seen).length > 2){
            if (seen[s[start]] === 1){
                delete seen[s[start]]
            } else {
                seen[s[start]]--
            }
            start++
        }
        if (maxlength < i - start + 1){
            maxlength = i - start + 1
        }
    }
    return maxlength
};
```

MAJORITY ELEMENT

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var majorityElement = function(nums) {
   let numCountMap = {}
   for (let num of nums) {
      if (numCountMap[num]){
         numCountMap[num]++
      } else {
         numCountMap[num] = 1
      }
   }
   for (let key in numCountMap){
      if (numCountMap[key] >= nums.length/2) {
         return key
      }
   }
};
```

MAX SUBARRAY SUM

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// Kadane's Algorithm
var maxSubArray = function(nums) {
   let maxEndingHere = nums[0]
   let maxSoFar = nums[0]
   for (let i = 1; i < nums.length; i++){
      maxEndingHere = Math.max(maxEndingHere + nums[i], nums[i])
      maxSoFar = Math.max(maxSoFar, maxEndingHere)
   }

   return maxSoFar

};
```

MERGE INTERVALS

```
/**
 * @param {Interval[]} intervals
 * @return {Interval[]}
 */
var merge = function(intervals) {
   if (intervals.length)
      return intervals

   intervals.sort((a,b) => {
      a.start - b.start;
   });

   let result = []
   let overlap = intervals[0]

   for (let i = 1; i < intervals.length; i++){
      if (overlap.end >= intervals[i].start){
         overlap.end = Math.max(intervals[i].end, overlap.end)
      } else {
         result.push(overlap);
         overlap = intervals[i];
      }
   }
   result.push(overlap)
   return result
};
```

MERGE K SORTED LINKED LIST

```
/**
 * @param {ListNode[]} lists
 * @return {ListNode}
 */
var mergeKLists = function(lists, lo = 0, high = lists.length - 1) {
   if (lists.length === 0){
      return lists
   }

   if (lo >= high){
      return lists[lo]
   }

   const mid = Math.floor((lo + high)/2)
   const left = mergeKLists(lists, lo, mid)
   const right = mergeKLists(lists, mid + 1, high)
   return mergeTwoLists(left, right)
};
```

```
const mergeTwoLists = (l1, l2) => {
   if (!l1){
      return l2
   }

   if (!l2){
      return l1
   }

   if (l1.val < l2.val){
      l1.next = mergeTwoLists(l1.next, l2)
      return l1
   } else {
      l2.next = mergeTwoLists(l1, l2.next)
      return l2
   }
}
```

MERGE SORTED ARRAY

```
/*
Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.

Note:

The number of elements initialized in nums1 and nums2 are m and n respectively.
You may assume that nums1 has enough space (size that is greater or equal to m + n) to hold additional elements from nums2.
Example:

Input:
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],       n = 3

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

/*
Method 1

This was the first and simplest method that popped into my head. Copy the second array into the "empty" half of the first array, and then sort it. Around 56ms (~80%) run time.

Simply using .sort() isn't enough as it will sort lexigraphically (worth remembering), so we have to pass it a custom sort function.
*/

var merge = function(nums1, m, nums2, n) {
```

```
    if (!n) return;

    for (let i = 0; i < n; i++) {
        nums1[m+i] = nums2[i];
    }

    nums1.sort(function (a,b) {
        return a - b;
    });

};

/*
Method 3

This method uses no additional memory and retains the 52ms (100%) runtime.

The biggest problem with starting from the front, is that you run into trouble with overwriting values
etc. and it likely won't work.

This method starts from the back - where our "empty" spaces are.
*/

/**
 * @param {number[]} nums1
 * @param {number} m
 * @param {number[]} nums2
 * @param {number} n
 * @return {void} Do not return anything, modify nums1 in-place instead.
 */
var merge = function(nums1, m, nums2, n) {
    var length = m + n

    m--
    n--
    while (length--){
        if (nums1[m] > nums2[n] || n < 0){
            nums1[length] = nums1[m--]
        } else {
            nums1[length] = nums2[n--]
        }
    }

}
```

MERGE TWO SORTED LINKED LIST

```
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */


 //Better Solution for Merge List

 var mergeTwoLists = function(l1, l2) {
   let dummyHead = new ListNode(0)
   let curr = dummyHead
   while (l1 && l2){
      if (l1.val > l2.val) {
         curr.next = l2
         l2 = l2.next
      } else {
         curr.next = l1
         l1 = l1.next
      }
      curr = curr.next
   }
   if (l1 != null) curr.next = l1;
   if (l2 != null) curr.next = l2;
   return dummyHead.next
};
 //My Solution needs Space complexity
var mergeTwoLists = function(l1, l2) {

   if (!l1){
      return l2
   }

   if (!l2){
      return l1
   }

   let newHead = null
   let head = null
   while(l1 && l2){
      if (l2.val > l1.val){
         if (!newHead){
            newHead = new ListNode(l1.val);
            head = newHead
         } else {
            newHead.next = new ListNode(l1.val);
            newHead = newHead.next
```

```
            }
            l1 = l1.next
        } else {
            if (!newHead){
                newHead = new ListNode(l2.val);
                head = newHead
            } else {
                newHead.next = new ListNode(l2.val);
                newHead = newHead.next
            }
            l2 = l2.next
        }
    }

    if (l2){
        newHead.next = l2;
    }

    if (l1){
        newHead.next = l1;
    }
    return head
};
```

MIN STACK

```
var MinStack = function() {
    this.minStack = []
    this.container = []
};

/**
 * @param {number} x
 * @return {void}
 */
MinStack.prototype.push = function(x) {
    this.container.push(x)
    if (this.minStack.length === 0 || this.minStack[this.minStack.length - 1] >= x){
        this.minStack.push(x)
    }
};

/**
 * @return {void}
 */
MinStack.prototype.pop = function() {
    let element = this.container.pop();
    if (element === this.minStack[this.minStack.length - 1]) {
        this.minStack.pop()
```

```
    }
};

/**
 * @return {number}
 */
MinStack.prototype.top = function() {
    return this.container[this.container.length- 1]
};

/**
 * @return {number}
 */
MinStack.prototype.getMin = function() {
    return this.minStack[this.minStack.length- 1]
};
```

MISSING NUMBER

```
/**
 * @param {number[]} nums
 * @return {number}
 */
// My Implementation
var missingNumber = function(nums) {
    let numMap = {}
    for (let i = 0; i <= nums.length; i++){
        numMap[i] = true
    }

    for (let i of nums){
        if (numMap[i]) {
            delete numMap[i]
        }
    }

    for (let i in numMap){
        return i
    }
};

// Using (n*(n+1))/2 - sum
/**
 * @param {number[]} nums
 * @return {number}
 */
var missingNumber = function(nums) {
    return nums.reduce((currSum, num) => currSum - num, (nums.length * (nums.length+1)/2))
 };
```

MISSING RANGES

/*
Given a sorted integer array nums, where the range of elements are in the inclusive range [lower, upper], return its missing ranges.

Example:

Input: nums = [0, 1, 3, 50, 75], lower = 0 and upper = 99,
Output: ["2", "4->49", "51->74", "76->99"]

Solution description
Compare the gap between two neighbor elements and output its range, simple as that right? This seems deceptively easy, except there are multiple edge cases to consider, such as the first and last element, which does not have previous and next element respectively. Also, what happens when the given array is empty? We should output the range “0->99”. As it turns out, if we could add two “artificial” elements, –1 before the first element and 100 after the last element, we could avoid all the above pesky cases.
*/

```
var findMissingRanges = function(nums, lower, upper) {
   nums = [lower-1, ...nums, upper+1]
   let output=[]
   for (let i = 1; i<nums.length; i++){
      if (nums[i] - nums[i-1] >= 2){
         let curr;
         if ((nums[i-1] + 1) === (nums[i] - 1)){
            curr = nums[i-1] + 1 + ''
         } else {
            curr = (nums[i-1] + 1) + "->" + (nums[i] - 1)
         }
         output.push(curr)
      }
   }
   return output
};
```

MOVE ZEROS

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

Example:

Input: [0,1,0,3,12]
Output: [1,3,12,0,0]
Note:

You must do this in-place without making a copy of the array.
Minimize the total number of operations.

*/

```
/**
 * @param {number[]} nums
 * @return {void} Do not return anything, modify nums in-place instead.
 */
var moveZeroes = function(nums) {
   let zeroCount = 0
   for (i = 0; i < nums.length; i++){
      if (nums[i] === 0){
         zeroCount++
      } else {
         const temp = nums[i]
         nums[i] = nums[i-zeroCount]
         nums[i-zeroCount] = temp
      }
   }
};
```

NUMBER OF 1 BIT

```
/**
 * @param {number} n - a positive integer
 * @return {number}
 */
var hammingWeight = function(n) {
   let bits = 0;
   while (n) {
     if ((n & 1) !== 0){
        bits++
     }
     n = n >>> 1
     }
   return bits;
 };
```

NUMBER OF ISLANDS

```
/**
 * @param {character[][]} grid
 * @return {number}
 */
var numIslands = function(grid) {
   if (!grid){
      return 0
   }
   let numberOfIslands = 0
   for (i = 0; i < grid.length; i++){
      for (j = 0 ; j< grid[0].length; j++){
         if (grid[i][j] === '1'){
            numberOfIslands++;
            changeOneToZero(grid, i, j);
         }
      }
   }

   return numberOfIslands
};

function changeOneToZero(grid, i, j){
   if ( i >= grid.length || i < 0 || j < 0 || j >= grid[0].length || grid[i][j] === '0'){
      return
   }

   if (grid[i][j] === '1') {
      grid[i][j] = 0
      changeOneToZero(grid, i - 1, j)
      changeOneToZero(grid, i, j - 1)
      changeOneToZero(grid, i + 1, j)
      changeOneToZero(grid, i, j + 1)
   }
}
```

ONE AWAY

```
/**
CTCI - 1.5
One Away:
There are three types of edits that can be performed on strings:
- insert a character,
- remove a character,
- or replace a character.
Given two strings, write a function to check if they are
one edit (or zero edits) away.
EXAMPLE
pale, ple -> true
pales, pale -> true
pale, bale -> true
pale, bake -> false
I: 2 strings
O: boolean
C: optimize
E: empty string,
time complexity: linear
space complexity: constant
*/

//if insert, then s1's current char should match s2's next char
//if remove, then s1's next char should match s2's current char
//if replace, then s1's next char should match s2's next char

//max one edit
//if diff in lengths is greater than max edit, return false

//iterate through strings at the same time, checking for diff
//store max length for forloop condition
//if diff is found, dec edits, check if edits dropped below zero, if so return false
//when forloop is done, return true


function oneAway(s1, s2){
   let maxLen = Math.max(s1.length, s2.length)
   let edits = 1

   if (Math.abs(s1.length - s2.length) > edits){
      return false;
   }

   for (let i = 0, j = 0; i < maxLen || j < maxLen; i++,j++){
      if (s1[i] !== s2[j]){
```

```
            edits--;
            if (edits<0){
               return false;
            }
            if (s1[i] === s2[j+1]){ // insert
               j++
            } else if (s1[i+1] === s2[j]){ // remove
               i++
            }
         }
      }
      return true
}

// Second solution where long string is calculated and only if removed is checked
// let oneAway = (s1, s2) => {
//   let edits = 1;
//   let long = s1.length > s2.length ? s1 : s2;
//   let short = s1.length <= s2.length ? s1 : s2;

//   let maxLen = Math.max(s1.length, s2.length);
//   let diff = long.length - short.length;

//   if (diff > edits) {
//     return false;
//   }

//   for (let i = 0, j = 0; i < maxLen || j < maxLen; i++, j++) {
//     let c1 = long[i];
//     let c2 = short[j];
//     if (c1 !== c2) {
//       edits--;
//       if (edits < 0) {
//         return false;
//       }
//       if (long[i + 1] === c2) { //inserted or removed
//         i++;
//       }
//     }
//   }
//   return true;
// };


console.log(
   oneAway('pale', 'ple') === true, //removed
   oneAway('pales', 'pale') === true, //inserted
   oneAway('pale', 'bale') === true, //replaced
```

```
  oneAway('pale', 'bake') === false,
  oneAway('p', '') === true,
  oneAway('', 'bake') === false,
  oneAway('pr', 'r') === true,
  oneAway('pr', 'rp') === false,
  oneAway('brrr', 'brrss') === false,
  oneAway('abc', 'acs') === false,
  oneAway('aple', 'aple') === true
);
```

PALINDROMIC LINKED LIST

```
/*
Given a singly linked list, determine if it is a palindrome.
Example 1
Input: 1->2
Output: false
Example 2:

Input: 1->2->2->1
Output: true
Follow up:
Could you do it in O(n) time and O(1) space?
*/

/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
// Stack Approach
var isPalindrome = function(head) {
   if (!head) {
      return true;
   }
   var slow = head;
   var fast = head;
   var stack = [head.val];
   if (!head.next){
      return true
   }

   while (fast.next && fast.next.next) {
      fast = fast.next.next;
      slow = slow.next
      stack.push(slow.val)
   }

   // Has odd number of elements so pop the middle element from stack
   if (!fast.next){
      stack.pop()
   }

   while (slow.next){
      slow = slow.next
      if (stack.pop() !== slow.val){
```

```
            return false
        }
    }
    return stack.length === 0
};

//O(n) time and O(n) space. Not fancy but I think it's elegant and understandable.
var isPalindrome = function(head) {
    let first = '';
    let second = '';
    while (head) {
        first = first + head.val;
        second = head.val + second;
        head = head.next;
    }
    return first === second;
};

 // using the Approach of reversing the linked list
/**
 * @param {ListNode} head
 * @return {boolean}
 */
var isPalindrome = function(head) {
    if (head === null){
        return true;
    }

    let newHead = new ListNode(head.val)
    let current = head;
    while(current.next !== null) {
        current = current.next;
        let temp = new ListNode(current.val);
        temp.next = newHead;
        newHead = temp;
    }

    current = head;
    let anotherCurrent = newHead;
    while (current !== null) {
        if (anotherCurrent.val !== current.val){
            return false
        }
        current = current.next
        anotherCurrent = anotherCurrent.next
    }
    return true
};
```

PALINDROMIC PERMUTATIONS

```
// Input: Tact Coa
// Output: True(permutations: "Taco cat", "atco cat", etc.)

function palindromePermutation(s) {
   let hash={}
   let charCount = 0;

   for (let char of s){
      if (char === ' '){
         continue;
      }

      if (hash[char]) {
         delete hash[char]
      } else {
         hash[char] = true
      }
      charCount++
   }

   if (charCount%2 === 0) {
      return Object.keys(hash).length === 0;
   } else {
      return Object.keys(hash).length === 1;
   }
}

console.log(
   palindromePermutation('taco cat') === true,
   palindromePermutation('atco cat') === true,
   palindromePermutation(' rac  ecar rara ') === true,
   palindromePermutation('chirpingmermaid') === false,
   palindromePermutation('aabbc') === true,
   palindromePermutation('aaaabbbbcc') === true,
   palindromePermutation('aabc') === false,
   palindromePermutation('') === true
 );
```

PALINDROME NUMBER

```
var isPalindrome = function(x) {
   // 1st solution
   // return x.toString() === x.toString().split('').reverse().join('')

   // 3rd Solution
   var isPalindrome = function(x) {

      if (x < 0){
         return false
      }

      let div = 1
      while (Math.floor(x/div) >= 10){
         div = div*10
      }

      while (x>0){
         const l = Math.floor(x/div)
         const r = Math.floor(x%10)

         if (l!==r) {
            return false
         }

         x = x%div
         x = Math.floor(x/10)
         div = Math.floor(div/100)

      }

      return true
   };
```

PASCALS TRIANGLE

```
/**
 * @param {number} numRows
 * @return {number[][]}
 */
var generate = function(numRows) {
   let triangle = []

    // First base case; if user requests zero rows, they get zero rows.
   if (numRows === 0){
      return triangle
   }

    // Second base case; first row is always [1].
```

```
    triangle.push([1]);

    for (let i = 1;i < numRows; i++){
        // The first row element is always 1.
        let row = [1]
        // Each triangle element (other than the first and last of each row)
        // is equal to the sum of the elements above-and-to-the-left and
        // above-and-to-the-right.
        for (let j = 1; j < i; j++) {
            row.push(triangle[i-1][j-1] + triangle[i-1][j])
        }
        // The last row element is always 1.
        row.push(1)
        triangle.push(row)
    }
    return triangle;
};

/*
Time complexity : O(numRows^2)

Space complexity : O(numRows^2)
*/
```

PLUS ONE

```
/*
Given a non-empty array of digits representing a non-negative integer, plus one to the integer.

The digits are stored such that the most significant digit is at the head of the list, and each element in the array contain a single digit.

You may assume the integer does not contain any leading zero, except the number 0 itself.

Example 1:

Input: [1,2,3]
Output: [1,2,4]
Explanation: The array represents the integer 123.
Example 2:

Input: [4,3,2,1]
Output: [4,3,2,2]
Explanation: The array represents the integer 4321.
*/
/**
 * @param {number[]} digits
 * @return {number[]}
 */
```

```
var plusOne = function(digits) {
    for (let i = digits.length - 1; i >= 0; i--) {
      if (digits[i] < 9) {
         digits[i]++
         return digits
      }
      digits[i] = 0
  }
  return [1, ...digits]
};

// use spread operation
var plusOne = function(digits) {
  let carry = 1;
  for (let i = digits.length - 1; i >= 0; i--) {
      let sum = carry + digits[i];
      digits[i] = sum % 10;
      carry = sum >= 10 ? 1 : 0;
  }
  return carry == 1 ? [1,...digits] : digits;
};
```

POWER OF THREE

/*
Given an integer, write a function to determine if it is a power of three.
Example 1:
Input: 27
Output: true

Example 2:
Input: 0
Output: false
Example 3:

Input: 9
Output: true
Example 4:

Input: 45
Output: false
Follow up:
Could you do it without using any loop / recursion?
*/

```
/**
 * @param {number} n
 * @return {boolean}
 */
var isPowerOfThree = function(n) {
   let powerOfThree = 3
   if (n === powerOfThree || n === 1){
      return true
   }
   while(n > powerOfThree){
      powerOfThree = powerOfThree*3
      if (powerOfThree === n){
         return true
      }
   }
   return false
};

// second solution
var isPowerOfThree = function (n) {
   if (n == 0) return false; // avoid recurssive loop
   while (n % 3 === 0) {
     n /= 3;
   }
   return n === 1;
 };
```

REMOVE DUPLICATES FROM SORTED ARRAY

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var removeDuplicates = function(nums) {
   let i = 0
   for (j = i+1; j< nums.length; j++){
      if (nums[i] !== nums[j]) {
         i++;
         nums[i] = nums[j]
      }
   }
   return i + 1
};
```

REVERSE LINKED LIST

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
var reverseList = function(head) {
   let newHead = null
   let currentNode = head
   while (currentNode) {
      let tempNode = new ListNode(currentNode.val)
      tempNode.next = newHead
      newHead = tempNode
      currentNode = currentNode.next
   }
    return newHead
 };

 // Second Solution
 var reverseList = function(head) {
   let newHead = null
   let currentNode = head
   while (currentNode) {
      let tempNode = currentNode.next
      currentNode.next = newHead
      newHead = currentNode
      currentNode = tempNode
   }
    return newHead
 };
```

REVERSE POLISH NOTATION

```
/*
Evaluate the value of an arithmetic expression in Reverse Polish Notation.
Valid operators are +, -, *, /. Each operand may be an integer or another expression.
Note:
Division between two integers should truncate toward zero.
The given RPN expression is always valid. That means the expression would always evaluate to a result and there won't be any divide by zero operation.
Example 1:
Input: ["2", "1", "+", "3", "*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9
Example 2:
Input: ["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"]
Output: 22
Explanation:
  ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
*/

/**
 * @param {string[]} tokens
 * @return {number}
 */
var evalRPN = function(tokens) {
    let stack = []
    for (let token of tokens){
        if (token === "+") {
            stack.push(stack.pop() + stack.pop());
        } else if (token === "-") {
            stack.push(-stack.pop() + stack.pop());
        } else if (token === "*") {
            stack.push(stack.pop() * stack.pop());
        } else if (token === "/") {
            let number1 = stack.pop()
            let number2 = stack.pop()
            stack.push(Math.trunc(number2/number1));
        } else {
            stack.push(parseInt(token))
        }
    }
    return stack.pop()
};
```

ROTATE MATRIX

```
/**
CTCI - 1.7
Rotate Matrix:
Given an image represented by an NxN matrix, where each pixel in the image is 4 bytes,
write a method to rotate the image by 90 degrees. Can you do this in place?
I: nxn matrix
O: rotated matrix - 90 deg, clockwise
C: rotate matrix in place, optimize
E: empty matrix, even and odd values for n
time complexity: O(n^2) - quadratic
space complexity: O(2) - constant
*/

function rotateMatrix(m) {
   const n = m.length;
   for (let i = 0; i < Math.floor(n/2); i++){
      for (let j = 0; j < (n-2*i-1); j++){
         //A->B Swap
         let temp = m[i][i+j]
         m[i][i+j] = m[i+j][n-i-1]
         m[i+j][n-i-1] = temp

         //A->C Swap
         temp = m[n-i-1][n-1-i-j]
         m[n-i-1][n-1-i-j] = m[i][i+j]
         m[i][i+j] = temp

         // A->D Swap
         temp = m[n-1-i-j][i]
         m[n-1-i-j][i] = m[i][i+j]
         m[i][i+j] = temp
      }
   }
   return m;
}

let compareMatrix = (a, b) => {
   if (!Array.isArray(a) || !Array.isArray(b)) {
     return a === b;
   } else {
     let out = true;
     for (let i = 0; i < Math.max(a.length, b.length); i++) {
       if (out) {
         out = compareMatrix(a[i], b[i]);
       } else {
         return false;
```

```
      }
     }
     return out;
   }
  };



  console.log(
    compareMatrix(rotateMatrix([[1, 2], [3, 4]]), [[3, 1], [4, 2]]),
    compareMatrix(rotateMatrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]]), [[7, 4, 1], [8, 5, 2], [9, 6, 3]]),
    compareMatrix(rotateMatrix([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]]),
                  [[13, 9, 5, 1], [14, 10, 6, 2], [15, 11, 7, 3], [16, 12, 8, 4]]),
    compareMatrix(rotateMatrix([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15], [16, 17, 18, 19, 20],
[21, 22, 23, 24, 25]]),
                  [[21, 16, 11, 6, 1], [22, 17, 12, 7, 2], [23, 18, 13, 8, 3], [24, 19, 14, 9, 4], [25, 20, 15, 10,
5]]),
    compareMatrix(rotateMatrix([]), []),
    compareMatrix(rotateMatrix([[]]), [[]]),
    compareMatrix(rotateMatrix([[1]]), [[1]])
  );
```

ROMAN TO INT

```
/**
 * @param {string} s
 * @return {number}
 */
var romanToInt = function(s) {
   let symbolToValue = {'I':1, 'V':5, 'X':10, 'L': 50, 'C':100, 'D':500, 'M': 1000}
   let result = 0
   for (let i = 0; i < s.length; i++) {
    if (symbolToValue[s[i]] < symbolToValue[s[i+1]]){
       result = result - symbolToValue[s[i]]
    } else {
       result = result + symbolToValue[s[i]]
    }
   }
   return result
};
```

ROTATE ARRAY

```
/*
Given an array, rotate the array to the right by k steps, where k is non-negative.

Example 1:

Input: [1,2,3,4,5,6,7] and k = 3
Output: [5,6,7,1,2,3,4]
Explanation:
rotate 1 steps to the right: [7,1,2,3,4,5,6]
rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 3 steps to the right: [5,6,7,1,2,3,4]
Example 2:

Input: [-1,-100,3,99] and k = 2
Output: [3,99,-1,-100]
Explanation:
rotate 1 steps to the right: [99,-1,-100,3]
rotate 2 steps to the right: [3,99,-1,-100]
Note:

Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.
Could you do it in-place with O(1) extra space?
*/
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {void} Do not return anything, modify nums in-place instead.
 */
```

```
 //Using reverse
 var rotate = function(nums, k) {
   if (!nums || !nums.length || !(k %= nums.length))
      return;
   reverse(nums, 0, nums.length - 1)
   reverse(nums, 0, k - 1)
   reverse(nums, k, nums.length - 1)
};

function reverse(nums, start, end){
   while (start < end) {
      let temp = nums[start]
      nums[start] = nums[end]
      nums[end] = temp
      start++
      end--
   }
}


 // using unshift and splice
var rotate = function(nums, k) {
   nums.unshift(...nums.splice(nums.length - k))
};


// Using pop and unshift
var rotate = function(nums, k) {
   while (k>0){
      let num = nums.pop()
      nums.unshift(num)
      k--
   }
 };
```

SENTENCE REVERSAL

```
/**
 * @param {string} str
 * @returns {string}
 */
var reverseWords = function(str) {
   return str.trim().split(/\s+/).reverse().join(" ")
};

var reverseWords = function(str) {
   let arr = str.split(' ');
   let ans = '';

   for(let i = arr.length-1; i > -1; i--)
      {
         if(arr[i] !== '')
         {
            ans += arr[i] + ' ';
         }
      }
   return ans.trim();
};
```

SQRT OF X

```
var mySqrt = function(x) {
   if (x === null || x === undefined){
      return -1
   }
   var left = 0
   var right = Math.floor(x/2) + 1
   var mid

   while (left <= right){
      mid = Math.floor((left + right)/2)
      if (mid * mid > x){
         right = mid - 1
      } else if (mid * mid < x){
         left = mid + 1
      } else {
         return mid
      }
   }
   return mid * mid <=x? mid: mid-1;
   // return right
};
```

STRING TO INTEGER ATOI

```
/**
 * @param {string} str
 * @return {number}
 */
var myAtoi = function(str) {
    let i = 0;
    let res = 0;
    let isNegative = false;

    str = str.trim()

    // 2. Optional +/-
    const maybeSign = str[i];
    if (maybeSign === '+' || maybeSign === '-') {
        isNegative = maybeSign === '-';
        i += 1;
    }

// 3. Process numbers and stop once an invalid character is found
    for (; i < str.length; i += 1) {
        const code = str.charCodeAt(i) - 48; // '0' is 48
        if (code < 0 || code > 9) {
            break;
        }
        res *= 10;
        res += code;
    }

    if (isNegative) {
        res = -res;
    }
    return Math.max(-(2**31), Math.min(2**31 - 1, res));
};
```

STRSTR

/*
Implement strStr().

Return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Example 1:

Input: haystack = "hello", needle = "ll"
Output: 2
Example 2:

Input: haystack = "aaaaa", needle = "bba"
Output: -1
Clarification:

What should we return when needle is an empty string? This is a great question to ask during an interview.

For the purpose of this problem, we will return 0 when needle is an empty string. This is consistent to C's strstr() and Java's indexOf().
*/

// Better Implementation

```
var strStr = function(haystack, needle) {
   if (!needle){
      return 0
   }
   for (let i = 0; i < haystack.length; i++){
      if (haystack.slice(i, i + needle.length) === needle){
         return i
      }
   }
   return -1
};
```

SUM BIT WISE

```
/*
Calculate the sum of two integers a and b, but you are not allowed to use the operator + and -.

Example 1:

Input: a = 1, b = 2
Output: 3
Example 2:

Input: a = -2, b = 3
Output: 1
*/

/**
 * @param {number} a
 * @param {number} b
 * @return {number}
 * Explanation:
 * let's say a and b are both one-bit variable
 * There are only four bit sum condition: (1, 1), (1, 0), (0, 1), (0, 0)
 * Their bit sum is (10), (1), (1), (0) in that sequence
 * for the condition (1, 0), (0, 1), (0, 0), we can use a^b
 * and for the condition (1, 1), we will need to use a&b<<1 (a&b make sure it's both 1, and left shift
for carry)
 * the final results are the combination of these two equations
 * Now we have: a + b = a&b<<1 + a^b
 */
var getSum = function(a, b) {
   if (a === 0){
      return b
   }

   if (b === 0){
      return a
   }

   while( b != 0){
       // imagine newA = a^b
      // imagine newB = a&b<<1
      // we keep looping until b goes to zero
      // b will eventually go to zero since b is keeping shifting to the left, and b is result of (a&b)
      let carry = (a&b) << 1
      a = a ^ b
      b = carry
   }
   return a
};
```

SWAP NODES IN PAIR

```
/*
Given a linked list, swap every two adjacent nodes and return its head.

Example:

Given 1->2->3->4, you should return the list as 2->1->4->3.
Note:

Your algorithm should use only constant extra space.
You may not modify the values in the list's nodes, only nodes itself may be changed.
*/

/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var swapPairs = function(head) {
   let dummyHead = new ListNode(0)
   dummyHead.next = head

    let p = head;
    let prev = dummyHead;

    while (p && p.next){
       let q = p.next
       let r = p.next.next

       prev.next = q
       q.next = p
       p.next = r

       prev = p
       p = r
    }

   return dummyHead.next
 };
```

STRING COMPRESSION

```
/**
CTCI - 1.6
String Compression:
Implement a method to perform basic string compression using the counts of repeated characters.
For example, the string aabcccccaaa would become a2b1c5a3.
If the "compressed" string would not become smaller than the original string,
your method should return the original string. You can assume the string
has only uppercase and lowercase letters (a - z).
I: string
O: compressed string
C: optimize
E: empty string, compressed string that's same length as the original string
*/

//time complexity: linear
//space complexity: constant


function stringCompression(s) {
   let count = 1
   let out = ''
   for (let i = 0; i < s.length; i++){
      let curChar = s[i];
      let nextChar = s[i + 1];
      if (curChar === nextChar){
         count++
      } else {
         out = out + curChar + String(count);
         count = 1;
      }
   }
   return out.length < s.length ? out : s;
}

console.log(
   stringCompression('aabcccccaaa') === 'a2b1c5a3',
   stringCompression('aa') === 'aa',
   stringCompression('aaAAaa') === 'aaAAaa',
   stringCompression('aaaAAaa') === 'a3A2a2',
   stringCompression('') === ''
 );
```

SYMMMETRIC TREE

```
/*
Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree [1,2,2,3,4,4,3] is symmetric:

   1
  / \
 2   2
/ \ / \
3  4 4  3
But the following [1,2,2,null,3,null,3] is not:
   1
  / \
 2   2
  \   \
  3    3
Note:
Bonus points if you could solve it both recursively and iteratively.
*/
// Recursive approach
var isSymmetric = function(root) {
   if(!root){
      return true;
   }
   return check(root.left, root.right);
};


var check = function(left, right){
   if(!left && !right){
      return true;
   }
   if(!left || !right){
      return false;
   }
   if (left.val !== right.val) {
      return false;
   }
   return check(left.left, right.right) && check(right.left, left.right);
};

// Iterative approach

var isSymmetric = function(root) {
   let queue = []

   queue.unshift(root)
```

```
    queue.unshift(root)

    while (queue.length){
        let left = queue.shift()
        let right = queue.shift()
        if (!left && !right) {
            continue;
        }
        if (!left || !right) {
            return false;
        }
        if (left.val !== right.val){
            return false;
        }
        queue.unshift(left.left)
        queue.unshift(right.right)
        queue.unshift(left.right)
        queue.unshift(right.left)
    }
    return true
}
```

TWO SUM

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
var twoSum = function(nums, target) {
    let map = {}
    for (let i=0;i < nums.length;i++){
        if (map[nums[i]] !== undefined){
            return [map[nums[i]], i]
        }
        map[target-nums[i]] = i
    }
};
```

TWO SUM SORTED

/*
Two Sum II - Input array is sorted
Given an array of integers that is already sorted in ascending order, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2.

Note:

Your returned answers (both index1 and index2) are not zero-based.
You may assume that each input would have exactly one solution and you may not use the same element twice.
Example:

Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore index1 = 1, index2 = 2.
*/

```
/**
 * @param {number[]} numbers
 * @param {number} target
 * @return {number[]}
 */
var twoSum = function(numbers, target) {
   let smallIndex = 0
   let largeIndex = numbers.length - 1

   for (let i = 0; i < numbers.length; i++){
      let sum = numbers[smallIndex] + numbers[largeIndex]
      if (sum > target){
         largeIndex--
      } else if (sum < target){
         smallIndex++
      } else {
         return [smallIndex+1, largeIndex+1]
      }
   }
};
```

URLIFY

```
// URLify == Convert spaces to %20 with true length of string given.

function URLify(str, trueLength = str.length){
   // first pass:: find the count of non space characters in the string
   //subtract the chars from the true length to see how many spaces are allowed to replace with %20

   // second pass:: if we see a space and there are still spaces left, append '%20' to output string
   // otherwise copy the current character.
   // when run out of spaces, append the empty string instead.

   let out = ''
   let trueCharacters = 0;

   for (let i = 0; i < str.length; i++){
      if (str[i] !== ' ') {
         trueCharacters++;
      }
   }

      let trueSpaces = trueLength - trueCharacters;

   for(let i = 0; i < str.length; i++){
      if (str[i] === ' ' && trueSpaces > 0){
         out = out + '%20'
         trueSpaces--;
      } else if (str[i] !== ' '){
         out = out + str[i]
      }
   }

   // if n is not yet reached there are still spaces left
   while (trueSpaces > 0){
      out = out + '%20'
      trueSpaces--;
   }
   return out;
}

console.log(
   URLify('Mr John Smith ', 13) === 'Mr%20John%20Smith',
   URLify('Mr John Smith ', 14) === 'Mr%20John%20Smith%20',
   URLify('   hi', 7) === '%20%20%20hi%20%20',
   URLify('   hi ', 3) === '%20hi',
   URLify('', 0) === '',
   URLify('', 2) === '%20%20',
   URLify('hel lo', 5) === 'hello'
 );
```

VALID PALINDROME

```
/*
Given a string, determine if it is a palindrome, considering only alphanumeric characters and
ignoring cases.

Note: For the purpose of this problem, we define empty string as valid palindrome.

Example 1:

Input: "A man, a plan, a canal: Panama"
Output: true
Example 2:

Input: "race a car"
Output: false
*/

/**
 * @param {string} s
 * @return {boolean}
 */
var isPalindrome = function(s) {
   var strippedString = s.replace(/[^\w]/g, '');
   var reversedString = strippedString.split('').reverse().join('');

   return strippedString.toLowerCase() == reversedString.toLowerCase();
};
```

VALID PARENTHESIS

```
/**
 * @param {string} s
 * @return {boolean}
 */

 // My Implementation
var isValid = function(s) {
   var stack = []
   if (s === ""){
      return true
   }
   if (s.length < 2){
      return false
   }
   for (let char of s) {
     if (char === '(' || char === '{' || char === '[') {
         stack.push(char)
      } else {
         let topElement = stack.pop();
```

```
            if (char === ")" && topElement === "("){
                continue;
            } else if (char === "}" && topElement === "{"){
                continue;
            } else if (char === "]" && topElement === "["){
                continue;
            } else {
                return false
            }
        }
    }
    if (stack.length !== 0){
        return false
    }
    return true;
};

// Proper Implementation

/**
 * @param {string} s
 * @return {boolean}
 */
var isValid = function(s) {
    var stack = []
    var parenMap = {
        '{' : '}',
        '[' : ']',
        '(' : ')',
    }

    for (let char of s){
        if (parenMap[char]){
            stack.push(parenMap[char])
        } else {
            if (char !== stack.pop()){
                return false;
            }
        }
    }

    return stack.length === 0

};
```

ZERO MATRIX

```
function zeroMatrix(m){
   let row = []
   let column = []
   for (let i = 0;i < m.length; i++){
      row.push(false);
   }
   for (let j = 0;j < m[0].length; j++){
      column.push(false);
   }
   for (let i = 0; i < m.length; i++){
      for (let j = 0; j < m[0].length; j++){
         if (m[i][j] === 0){
            row[i] = true
            column[j] = true
         }
      }
   }

   for (let i = 0;i < row.length;i++){
      if (row[i] === true){
         for (let j = 0; j < m[0].length; j++){
            m[i][j] = 0
         }
      }
   }

   for (let j = 0;j < column.length;j++){
      if (column[j] === true){
         for (let i = 0; i < m.length; i++){
            m[i][j] = 0
         }
      }
   }
   console.log(m)
   return m;
}

zeroMatrix([[0,2,3],[0,0,1],[9,7,4]])
```