js-leetcode

Just answers and flow charts for various leetcode questions.

Example with gitbook plugin

This is an exercise:

Define a variable x equal to 10.

```
var x =
```

```
var x = 10;
```

```
assert(x == 10);
```

```
// This is context code available everywhere
// The user will be able to call magicFunc in his code
function magicFunc() {
    return 3;
}
```

---

0 - ./examples/add-binary.js

```
    var addBinary = function(a, b) {
    var [carry, i, j, res] = [0, a.length - 1, b.length - 1, ""];
    while(i >= 0 || j >= 0 || carry > 0) {
        const bita = i >= 0 ? Number(a[i]) : 0;
        const bitb = j >= 0 ? Number(b[j]) : 0;
        const sum = bita + bitb + carry;
        res = `${sum%2}${res}`;
        carry = Math.floor(sum / 2);
        i--; j--;
    }
    return res;
  };
```

Flowchart

./examples/add-binary.js-svg image

1 - ./examples/array-3-sum-equal-zero.js

```
    function sumToZero(arr) {
    if (!arr) return false

    arr.sort();
    let n = arr.length - 1;
    let sum = 0;
    let count = 0;

    //loop each item
    for (let i = 0; i < arr.length; i++) {
        let j = i + 1; //right of i
        let k = n; //end of arr
        while (k > j) {
            count++;
            sum = arr[i] + arr[j] + arr[k];
            console.log(i, j, k, sum);

            //found match...done
            if (sum === 0) {
                console.log('Found sum at', i, j, k, `(${arr[i]} +
${arr[j]} + ${arr[k]})`);
                return true;
            }
            // We didn't match. Let's try to get a little closer:
            //   If the sum was too big, decrement k.
            //   If the sum was too small, increment j.
            sum > 0 ? k-- : j++;
        }
    }

    return false;
}


//console.log(sumToZero([1, 1, 2, -1]));
//console.log(sumToZero([1, 2, 3, -3]));
//console.log(sumToZero([2, 1, -3, 3]));
console.log(sumToZero([3, 2, 1, 7, 9, 0, -4, 6]));






function threeSum(arr) {
    arr.sort();
    let k = arr.length - 1;
    for (let i = 0; i < arr.length; i++) {
        const a = arr[i];
        for (let j = i + 1; j < arr.length; j++) {
            const b = arr[j];
            console.log(a, b);
        }
    }


}
console.log(threeSum([2, 1, -3, 3]));
```

Flowchart

./examples/array-3-sum-equal-zero.js-svg image

---

2 - ./examples/array-pair-sum.js

```
    var arrayPairSum = function (nums) {
    let hash = [];
    for (let i = 0; i < 20001; ++i) {
        hash[i] = 0;
    }
    let sum = 0;
    let min = Number.MAX_VALUE;
    let max = Number.MIN_VALUE;
    for (let i = 0; i < nums.length; ++i) {
        let cur = nums[i] + 10000;
        ++hash[cur];
        min = Math.min(min, cur);
        max = Math.max(max, cur);
    }
    let evenOdd = 0;
    for (let i = min; i <= max; ++i) {
        let curAmount = hash[i];
        for (let j = 0; j < curAmount; ++j) {
            if (evenOdd == 0) {
                sum += i - 10000;
            }
            evenOdd ^= 1;
        }
    }
    return sum;
};
```

Flowchart

./examples/array-pair-sum.js-svg image

---

3 - ./examples/array-shuffle.js

```
    /**
 * @param {number[]} nums
 */
var Solution = function(nums) {
    this.nums = nums;
};

/**
 * Resets the array to its original configuration and return it.
 * @return {number[]}
 */
Solution.prototype.reset = function() {
    return this.nums;
};

/**
 * Returns a random shuffling of the array.
 * @return {number[]}
 */
Solution.prototype.shuffle = function() {
   /* let copy = this.nums.slice();
    for (let i = 0; i < copy.length; i++) {
        let rand = Math.floor(Math.random() * copy.length);
        let temp = copy[i];
        copy[i] = copy[rand];
        copy[rand] = temp;
    }
    return copy;*/

    let nums = [...this.nums];
    let i = nums.length;

    while (i--) {
      const n = Math.floor(Math.random() * nums.length);
      [nums[i], nums[n]] = [nums[n], nums[i]];
    }

    return nums;

   /* let temp = [];
    this.nums.forEach((val, key) => temp[key] = val);
    for (let i = 0; i < temp.length; i++) {
        let rand = Math.floor(Math.random() * (temp.length - i))
        let n = temp[i];
        temp[i] = temp[rand];
        temp[rand] = n;
    }

    return temp;*/
};
```

```
/**
 * Your Solution object will be instantiated and called as such:
 * var obj = Object.create(Solution).createNew(nums)
 * var param_1 = obj.reset()
 * var param_2 = obj.shuffle()
 */
```

Flowchart

./examples/array-shuffle.js-svg image

---

4 - ./examples/best-time-to-buy-sell-stock-2.js

```
    // Say you have an array for which the ith element is the price of a
given stock on day i.
//
// Design an algorithm to find the maximum profit. You may complete as
many transactions as you like (i.e., buy one and sell one share of the
stock multiple times).
//
// Note: You may not engage in multiple transactions at the same time
(i.e., you must sell the stock before you buy again).
//
// Example 1:
//
// Input: [7,1,5,3,6,4]
// Output: 7
// Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5),
profit = 5-1 = 4.
// Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit =
6-3 = 3.
//
// Example 2:
//
// Input: [1,2,3,4,5]
// Output: 4
// Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5),
profit = 5-1 = 4.
// Note that you cannot buy on day 1, buy on day 2 and sell them later, as
you are
// engaging multiple transactions at the same time. You must sell before
buying again.
//
// Example 3:
//
// Input: [7,6,4,3,1]
// Output: 0
// Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

```
/**
 * @param {number[]} prices
 * @return {number}
 */

// time O(n)
// space O(1)
function maxProfit(prices) {
    let profit = 0;

    for (let i = 0; i < prices.length - 1; i++) {
        const diff = prices[i + 1] - prices[i];
        if (diff > 0) profit += diff;
    }

    return profit;
}
```

Flowchart

./examples/best-time-to-buy-sell-stock-2.js-svg image

---

5 - ./examples/best-time-to-buy-sell-stock.js

```
    // Say you have an array for which the ith element is the price of a
given stock on day i.
//
// If you were only permitted to complete at most one transaction (i.e.,
buy one and sell one share of the stock), design an algorithm to find the
maximum profit.
//
// Note that you cannot sell a stock before you buy one.
//
// Example 1:
//
// Input: [7,1,5,3,6,4]
// Output: 5
// Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6),
profit = 6-1 = 5.
// Not 7-1 = 6, as selling price needs to be larger than buying price.
//
// Example 2:
//
// Input: [7,6,4,3,1]
// Output: 0
// Explanation: In this case, no transaction is done, i.e. max profit = 0.

/**
 * @param {number[]} prices
 * @return {number}
 */

// time O(n)
// space O(1)
function maxProfit(prices) {
    let min = Infinity;
    let max = 0;

    for (let i = 0; i < prices.length - 1; i++) {
        min = Math.min(min, prices[i]);
        max = Math.max(max, prices[i + 1] - min)
    }

    return max;
}
```

Flowchart

./examples/best-time-to-buy-sell-stock.js-svg image

---

6 - ./examples/big-o.js

```
    /*
## Big-O Analysis

Big-O analysis is a form of runtime analysis that measures the efficiency
of an algorithm in terms of the time it takes for the algorithm to run as
a function of the input size. It’s not a formal benchmark, just a simple
way to classify algorithms by relative efficiency when dealing with very
large input sizes.
How Big-O Analysis Works

In Big-O analysis, input size is assumed to be an unknown value n. In this
example, n simply represents the number of elements in an array. In other
problems, n may represent the number of nodes in a linked list, the number
of bits in a data type, or the number of entries in a hash table. After
determining what n means in terms of the input, you must determine how
many operations are performed for each of the n input items. “Operation”
is a fuzzy word because algorithms differ greatly.
Commonly, an operation is something that a real computer can do in a
constant amount of time, like adding an input value to a constant,
creating a new input item, or deleting an input value. In Big-O analysis,
the times for these operations are all considered equivalent.

- In both `CompareToMax` and `CompareToAll`, the operation of greatest
interest is comparing an array value to another value.

- In `CompareToMax`, each array element was compared once to a maximum
value. Thus, the n input items are each examined once, resulting in n
examinations.
```

```
This is considered O(n), usually referred to as linear time: The time
required to run the algorithm increases linearly with the number of input
items.
How to Do Big-O Analysis

The general procedure for Big-O runtime analysis is as follows:

1. Figure out what the input is and what n represents.
2. Express the number of operations the algorithm performs in terms of n.
3. Eliminate all but the highest-order terms.
4. Remove all constant factors.

For the algorithms you'll encounter in interviews, Big-O analysis should
be straightforward as long as you correctly identify the operations that
are dependent on the input size.
Which is Better?

The performance of most algorithms depends on n, the size of the input.
The algorithms can be classified as follows from best-to-worse
performance:

* O(log n) — An algorithm is said to be logarithmic if its running time
increases logarithmically in proportion to the input size.
* O(n) — A linear algorithm's running time increases in direct proportion
to the input size.
* O(n log n) — A superlinear algorithm is midway between a linear
algorithm and a polynomial algorithm.
* O(nc) — A polynomial algorithm grows quickly based on the size of the
input.
* O(cn) — An exponential algorithm grows even faster than a polynomial
algorithm.
* O(n!) — A factorial algorithm grows the fastest and becomes quickly
unusable for even small values of n.

The run times of different orders of algorithms separate rapidly as n gets
larger.

Consider the run time for each of these algorithm classes with n = 10:

* log 10 = 1
* 10 = 10
* 10 log 10 = 10
* 102 = 100
* 210= 1,024
* 10! = 3,628,800

Now double it to n = 20:

* log 20 = 1.30
* 20 = 20
* 20 log 20= 26.02
* 202 = 400
* 220 = 1,048,576
* 20! = 2.43×1018
```

```
Notation
Name
O(1)
Constant
O(log(n))
Logarithmic
O((log(n))c)
Poly-logarithmic
O(n)
Linear
O(n2)
Quadratic
O(nc)
Polynomial
O(cn)
Exponential


### [] O(1)

Consider the following function:

function increment(num){
  return ++num;
}

[]O(N)

Now, let's use the sequential search algorithm:

function sequentialSearch(array, item){
  for (var i=0; i<array.length; i++){
    if (item === array[i]){ //{1}
      return i;
    }
  }
  return -1;
}

[]O(N2)
For the O(n2) example, let's use the bubble sort algorithm:

function swap(array, index1, index2){
  var aux = array[index1];
  array[index1] = array[index2];
  array[index2] = aux;
}

function bubbleSort(array){
  var length = array.length;
  for (var i=0; i<length; i++){    //{1}
    for (var j=0; j<length-1; j++ ){ //{2}
      if (array[j] > array[j+1]){
        swap(array, j, j+1);
      }
    }
  }
}

*/

// O(1)
function increment(num) {
    return ++num;
}
console.log(increment(2));


// O(n)
function sequentialSearch(array, item) {
    for (var i = 0; i < array.length; i++) {
        if (item === array[i]) { //{1}
            return i;
        }
    }
    return -1;
}

let input1 = [9, 5, 2, 4, 3, 7, 6];

console.log(sequentialSearch(input1, 4));



// O(n^2)
function swap(array, index1, index2) {
    let aux = array[index1];
    array[index1] = array[index2];
    array[index2] = aux;
}

function bubbleSort(array) {
    let length = array.length;
    for (let i = 0; i < length; i++) { //{1}
        for (let j = 0; j < length - 1; j++) { //{2}
            if (array[j] > array[j + 1]) {
                swap(array, j, j + 1);
            }
        }
    }
    return array;
}
console.log(bubbleSort(input1));
```

Flowchart

./examples/big-o.js-svg image

---

7 - ./examples/binary-search.js

```
    function binarySearch(arr, val) {
    let lower_bound = 0;
    let upper_bound = arr.length - 1;

    while (lower_bound <= upper_bound) {
        let midpoint = Math.floor(upper_bound + lower_bound / 2);
        let value_at_midpoint = arr[midpoint];

        console.log(midpoint, value_at_midpoint);

        if (val < value_at_midpoint) {
            upper_bound = midpoint - 1;
        } else if (val > value_at_midpoint) {
            lower_bound = midpoint + 1;
        } else if (val == value_at_midpoint) {
            return midpoint;
        }
    }
    return null;
}

//console.log(binarySearch([1, 3, 4, 5, 6, 21, 43, 54, 86], 21));
//console.log(binarySearch([1, 3, 4, 5, 6, 21, 43, 54, 86], 1));

/**
 *
 * Given two unsorted arrays of distinct elements,
 * the task is to find all pairs from both arrays whose sum is equal to x.
 */

function findPairs(arr1, arr2, x) {
    let map = {};
    for (let i = 0; i < arr1.length; i++) {
        map[arr1[i]] = 0;
    }
    for (let j = 0; j < arr2.length; j++) {
        console.log(x - arr2[j]);
        if (map[x - arr2[j]]) {
            console.log('Found pair');
        }
    }

}
console.log(findPairs([1, 0, -4, 7, 6, 4], [0, 2, 4, -3, 2, 1], 8))
```

Flowchart

./examples/binary-search.js-svg image

---

8 - ./examples/bubble-sort.js

```
function swap(array, index1, index2) {
    var aux = array[index1];
    array[index1] = array[index2];
    array[index2] = aux;
}

function bubbleSort(array) {
    let length = array.length;
    for (let i = 0; i < length; i++) {
        console.log(i, array)
        for (let j = 0; j < length - 1; j++) {
            if (array[j] > array[j + 1]) {
                swap(array, j, j + 1);
            }
        }
    }
    return array;
}
console.log(bubbleSort([5,4,3,2,1]));
```

Flowchart

./examples/bubble-sort.js-svg image

---

9 - ./examples/climb-stairs.js

```
    /*
You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can
you climb to the top?

Note: Given n will be a positive integer.

Example 1:
```

Input: 2 Output: 2 Explanation: There are two ways to climb to the top.

1. 1 step + 1 step
2. 2 steps

```
Example 2:
```

Input: 3 Output: 3 Explanation: There are three ways to climb to the top.

1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

```
*/
var fib = function (n) {
    let fibo = [0, 1];
    if (n <= 2) return 1;
    for (var i = 2; i <= n; i++) {
        fibo[i] = fibo[i - 1] + fibo[i - 2];
    }
    return fibo[n];
}
/**
 * @param {number} n
 * @return {number}
 */
var climbStairs = function (n) {
    return fib(n + 1)
};
```

Flowchart

./examples/climb-stairs.js-svg image

---

10 - ./examples/coin-change.js

```
    const print = console.log.apply;
function makeChange(origAmt, coins) {
    var remainAmt = 0;
    if (origAmt % .25 < origAmt) {
       coins[3] = parseInt(origAmt / .25);
       remainAmt = origAmt % .25;
       origAmt = remainAmt;
    }
    if (origAmt % .1 < origAmt) {
       coins[2] = parseInt(origAmt / .1);
       remainAmt = origAmt % .1;
       origAmt = remainAmt;
    }
    if (origAmt % .05 < origAmt) {
       coins[1] = parseInt(origAmt / .05);
       remainAmt = origAmt % .05;
       origAmt = remainAmt;
    }
    coins[0] = parseInt(origAmt / .01);
 }

 function showChange(coins) {
    if (coins[3] > 0) {
       print("Number of quarters - " + coins[3] + " - " + coins[3] * .25);
    }
    if (coins[2] > 0) {
       print("Number of dimes - " + coins[2] + " - " + coins[2] * .10);
    }
    if (coins[1] > 0) {
       print("Number of nickels - " + coins[1] + " - " + coins[1] * .05);
    }
    if (coins[0] > 0) {
       print("Number of pennies - " + coins[0] + " - " + coins[0] * .01);
    }
 }

 var origAmt = .63;
 var coins = [];
 makeChange(origAmt, coins);
 showChange(coins);
```

Flowchart

./examples/coin-change.js-svg image

---

11 - ./examples/contains-duplicate.js

```
/**
 * Given an array of integers, find if the array contains any duplicates.

Your function should return true if any value appears at least twice in
the array, and it should return false if every element is distinct.

Example 1:

Input: [1,2,3,1]
Output: true
Example 2:

Input: [1,2,3,4]
Output: false
Example 3:
```

```
Input: [1,1,1,3,3,4,3,2,4,2]
Output: true
 * @param {number[]} nums
 * @return {boolean}
 */
var containsDuplicate = function (nums) {
    let result = false;
    let p1 = 0;
    let p2 = 0;
    let len = nums.length;
    let matchCount = 0;
    while (p1 < len) {
        let n1 = nums[p1];
        p2 = p1 + 1;
        while (p2 < len) {
            let n2 = nums[p2];
            p2++;
            if (n1 == n2) {
                result = true;
            }
        }
        p1++;
    }

    return result;
};

var containsDuplicate2 = function (nums) {
    var counter = {};
    for (var i = 0; i < nums.length; i++) {
        if (counter.hasOwnProperty(nums[i])) {
            return true;
        } else {
            counter[nums[i]] = 1;
        }
    }

    return false;
};

console.time('containsDuplicate');
console.log(containsDuplicate([9, 9, 1]));
console.log(containsDuplicate([1, 2, 3]));
console.timeEnd('containsDuplicate');


console.time('containsDuplicate2');
console.log(containsDuplicate2([9, 9, 1]));
console.log(containsDuplicate2([1, 2, 3]));
console.timeEnd('containsDuplicate2');
```

Flowchart

./examples/contains-duplicate.js-svg image

---

12 - ./examples/count-and-say.js

```
    // The count-and-say sequence is the sequence of integers with the first five terms as following:
//
// 1.     1
// 2.     11
// 3.     21
// 4.     1211
// 5.     111221
//
// 1 is read off as "one 1" or 11.
// 11 is read off as "two 1s" or 21.
// 21 is read off as "one 2, then one 1" or 1211.
// Given an integer n, generate the nth term of the count-and-say sequence.
//
// Note: Each term of the sequence of integers will be represented as a string.
//
// Example 1:
//
// Input: 1
// Output: "1"
//
// Example 2:
//
// Input: 4
// Output: "1211"

/**
 * @param {number} n
 * @return {string}
 */
function countAndSay(n) {
    let res = '1';

    for (let i = 1; i < n; i++) {
        res = say(res);
    }

    return res;
}

function say(str) {
    let res = '';
    let count = 0;
    let num = str[0];

    for (let i = 0; i < str.length; i++) {
        if (str[i] === num) {
            count++;
        } else {
            res += count + str[i - 1];
            count = 1;
            num = str[i];
        }
    }

    return res + count + num;
}
```

Flowchart

./examples/count-and-say.js-svg image

---

13 - ./examples/delete-nth-linked-list.js

```
    /*
## Delete Node in a Linked List

Write a function to delete a node (except the tail) in a singly linked
list, given only access to that node.

Given linked list -- head = [4,5,1,9], which looks like following:
```

```
4 -> 5 -> 1 -> 9
```

```
*/
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * // Since we do not have access to the node before the one we want to
delete, we cannot modify the next pointer of that node in any way.
// Instead, we have to replace the value of the node we want to delete
with the value in the node after it, and then delete the node after it.
// Because we know that the node we want to delete is not the tail of the
list, we can guarantee that this approach is possible.
```

```
 * @param {ListNode} node
 * @return {void} Do not return anything, modify node in-place instead.
 */
var deleteNode = function (node) {
    node.val = node.next.val;
    node.next = node.next.next;
};
```

Flowchart

./examples/delete-nth-linked-list.js-svg image

---

14 - ./examples/diagonal-traverse.js

```
    /**
 * Diagonal Traverse
 * Given a matrix of M x N elements (M rows, N columns),
 * return all elements of the matrix in diagonal order as shown in the
below image.
 *
 * @param {number[][]} matrix
 * @return {number[]}
 */
var findDiagonalOrder = function (matrix) {
    if (matrix.length === 0) {
        return [];
    }
    let len = matrix.length;
    let len2 = matrix[0].length;
    let len3 = len * len2;
    let res = [];
    let i = 0,
        j = 0,
        k = 0;

    while (k < len3) {
        res.push(matrix[i][j]);
        if ((i + j) % 2 === 1) {
            if (i === len - 1) {
                j++;
            } else if (j === 0) {
                i++;
            } else {
                i++;
                j--;
            }
        } else {
            if (j === len2 - 1) {
                i++;
            } else if (i === 0) {
                j++;
            } else {
                i--;
                j++;
            }
        }
        k++;
    }
    return res;
};
```

Flowchart

./examples/diagonal-traverse.js-svg image

15 - ./examples/example.js

```

        function indexSearch(list, element) {
        let currentIndex,
            currentElement,
            minIndex = 0,
            maxIndex = list.length - 1;

        while (minIndex <= maxIndex) {
            currentIndex = Math.floor(maxIndex + maxIndex) / 2;
            currentElement = list[currentIndex];

            if (currentElement === element) {
                return currentIndex;
            }

            if (currentElement < element) {
                minIndex = currentIndex + 1;
            }

            if (currentElement > element) {
                maxIndex = currentIndex - 1;
            }
        }

        return -1;
    }
```

Flowchart

./examples/example.js-svg image

16 - ./examples/fib.js

```
    function fib(n){
    if(n === 1 || n === 2){
        return 1;
    }
    return fib(n - 1) + fib(n - 2);
}

console.log(fib(5));
```

Flowchart

./examples/fib.js-svg image

---

17 - ./examples/first-bad-version.js

```
    // You are a product manager and currently leading a team to develop a
new product. Unfortunately, the latest version of your product fails the
quality check. Since each version is developed based on the previous
version, all the versions after a bad version are also bad.
//
// Suppose you have n versions [1, 2, ..., n] and you want to find out the
first bad one, which causes all the following ones to be bad.
//
// You are given an API bool isBadVersion(version) which will return
whether version is bad. Implement a function to find the first bad
version. You should minimize the number of calls to the API.
//
// Example:
//
// Given n = 5
//
// call isBadVersion(3) -> false
// call isBadVersion(5) -> true
// call isBadVersion(4) -> true
//
// Then 4 is the first bad version.

/**
 * Definition for isBadVersion()
 *
 * @param {integer} version number
 * @return {boolean} whether the version is bad
 * isBadVersion = function(version) {
 *     ...
 * };
 */
```

```
/**
 * @param {function} isBadVersion()
 * @return {function}
 */

/** Binary search */
function solution(isBadVersion) {
  /**
   * @param {integer} n Total versions
   * @return {integer} The first bad version
   */
  return function (n) {
    return find(1, n);
  };

  function find(l, r) {
    if (l === r) return l;

    const mid = Math.floor((l + r) / 2);
    return isBadVersion(mid) ? find(l, mid) : find(mid + 1, r);
  }
}
```

Flowchart

./examples/first-bad-version.js-svg image

---

18 - ./examples/first-uniq-char.js

```
    /*
## First Unique Character in a String
Given a string, find the first non-repeating character in it and return
it's index.
If it doesn't exist, return -1.

Examples:

s = "leetcode"
return 0.

s = "loveleetcode",
return 2.

> Note: You may assume the string contain only lowercase letters.
*/
/**
 * @param {string} s
 * @return {number}
 */
```

```
var firstUniqChar = function (s) {
    var map = {};
    for (var i = 0; i < s.length; i++) {
        let key = s[i];
        if (map[key] === undefined) { // if letter hasn't been encountered
            // set char as key and value ([index, count]) as tuple of the
index and letter count
            map[key] = [i, 1];
        } else {
            // increment letter count
            map[key][1] += 1;
            console.log(key, i)
        }
    }

    for (var k in map) {
        console.log(k, map[k]);
        // if a character count is equal to 1 it is the first unique, so
return
        if (map[k][1] === 1) return map[k][0];
    }
    return -1; // return -1 if we didnt find a unique in our map
};
console.log(firstUniqChar('abc'));
console.log(firstUniqChar('aba'));
console.log(firstUniqChar('loveleetcode'));
```

Flowchart

./examples/first-uniq-char.js-svg image

---

19 - ./examples/fizz-buzz.js

```
    /*
## Fizz Buzz
Write a program that outputs the string representation of numbers from 1
to n.

But for multiples of three it should output “Fizz” instead of the number
and for the multiples of five output “Buzz”. For numbers which are
multiples of both three and five output “FizzBuzz”.

Example:
```

n = 15,

Return: ["1", "2", "Fizz", 3 "4", "Buzz", "Fizz", 6 "7", "8", "Fizz", 9 "Buzz", "11", "Fizz", 12 "13", "14", "FizzBuzz"]

```
*/
/**
 * @param {number} n
 * @return {string[]}
 */
function fizzBuzz(n) {
    let arr = [];

    for (let i = 1; i <= n; i++) {
        if (i % 15 === 0) arr.push('FizzBuzz');
        else if (i % 3 === 0) arr.push('Fizz');
        else if (i % 5 === 0) arr.push('Buzz');
        else arr.push(String(i));
    }

    return arr;
}
```

Flowchart

./examples/fizz-buzz.js-svg image

---

20 - ./examples/group-anagrams.js

```
    /**
 * @param {string[]} strs
 * @return {string[][]}
 */


var groupAnagrams = function (strs) {

    // 1 sort each word's letters to enable matching
    // alph will be, e.g.: ['aet', 'aet', 'ant' ...]
    var alph = strs.map(word => word.split('').sort().join(''));

    // 2 create groups of indices of identical sorted words
    // locations will be, e.g.: { 'aet': [0, 1, 3], 'ant': [...] }
    var locations = {};
    for (var i = 0; i < alph.length; i++) {
        if (!locations[alph[i]]) {
            locations[alph[i]] = [i];
        } else {
            locations[alph[i]].push(i);
        }
    }
    console.log(locations);

    // 3 transform groups of indices into groups of original words
    // e.g. for 'aet', map [0, 1, 3] to words at those indices in strs
    var output = [];
    for (var word in locations) {
        output.push(locations[word].map(idx => strs[idx]));
    }
    return output;
};
console.log(
    groupAnagrams(["eat", "tea", "tan", "ate", "nat", "bat"])
)
```

Flowchart

./examples/group-anagrams.js-svg image

---

21 - ./examples/has-path-sum.js

```
    var hasPathSum = function(root, sum) {
    let node = root;
    if(!node){
        return false;
    }
    if(node === null){
        return (sum === 0);
    }
    let answer = false;
    let subsum = sum - node.val;
    if(subsum === 0 && node.left == null && node.right == null){
        return true;
    }
    if(node.left !== null){
       answer = answer || hasPathSum(node.left, subsum);
    }
    if(node.right !== null){
       answer = answer || hasPathSum(node.right, subsum);
    }
    return answer;
};
```

Flowchart

./examples/has-path-sum.js-svg image

---

22 - ./examples/hour-glass.js

```
    let input = [
    [1, 1, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0],
    [1, 1, 1, 0, 0, 0],
    [0, 0, 2, 4, 4, 0],
    [0, 0, 0, 2, 0, 0],
    [0, 0, 1, 2, 4, 0]
];

let input2 = [
    [1, 1, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0],
    [1, 1, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0]
];

function hourglassSum(arr) {
    let sum = 0;
    let maxSum = Number.MIN_SAFE_INTEGER;
    for (let i = 0; i < 4; i++) {
        const row = arr[i];
        for (let j = 0; j < 4; j++) {
            const col = row[j];
            sum = (
                arr[i][j] +
                arr[i][j + 1] +
                arr[i][j + 2] +
                arr[i + 1][j + 1] +
                arr[i + 2][j] +
                arr[i + 2][j + 1] +
                arr[i + 2][j + 2]
            );
            if (sum > maxSum) {
                maxSum = sum;
            }
        }

    }
    return maxSum
}

console.log(hourglassSum(input));
console.log(hourglassSum(input2));
```

Flowchart

./examples/hour-glass.js-svg image

---

23 - ./examples/house-robber.js

```
    // 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.
//
// Example 1:
//
// Input: [1,2,3,1]
// Output: 4
// Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
//   Total amount you can rob = 1 + 3 = 4.
//
// Example 2:
//
// Input: [2,7,9,3,1]
// Output: 12
// Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob
house 5 (money = 1).
//   Total amount you can rob = 2 + 9 + 1 = 12.

/**
 * @param {number[]} nums
 * @return {number}
 */

/** 1) Recursion */
// time O(n)
// space O(n)
function rob1(nums) {
    return count(nums.length - 1, [], nums)
}

function count(n, cache, nums) {
    if (n < 0) return 0;
    if (cache[n] !== undefined) return cache[n];

    cache[n] = Math.max(
        count(n - 2, cache, nums) + nums[n],
        count(n - 1, cache, nums)
    );

    return cache[n];
}

/** 2) Iteration */
// time O(n)
// space O(n)
function rob2(nums) {
    if (nums.length === 0) return 0;
    if (nums.length === 1) return nums[0];

    const totals = [nums[0], Math.max(nums[0], nums[1])];

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

    return totals[totals.length - 1];
}

/** 3) Iteration */
// time O(n)
// space O(1)
function rob3(nums) {
    if (nums.length === 0) return 0;
    if (nums.length === 1) return nums[0];

    let a = nums[0];
    let b = Math.max(nums[0], nums[1]);

    for (let i = 2; i < nums.length; i++) {
        const max = Math.max(a + nums[i], b);
        a = b;
        b = max;
    }

    return b;
}

/** 4) Iteration */
// time O(n)
// space O(1)
function rob(nums) {
    let a = 0;
    let b = 0;

    for (let i = 0; i < nums.length; i++) {
        if (i % 2 === 0) a = Math.max(a + nums[i], b);
        else b = Math.max(a, b + nums[i]);
    }

    return Math.max(a, b);
}
```

Flowchart

./examples/house-robber.js-svg image

---

24 - ./examples/is-palindrome.js

```
    function isPalindrome(word) {
    var s = new Stack();
    for (var i = 0; i < word.length; ++i) {
       s.push(word[i]);
    }
    var rword = "";
    while (s.length() > 0) {
       rword += s.pop();
    }
    if (word == rword) {
       return true;
     }
    else {
       return false;
    }
 }
```

Flowchart

./examples/is-palindrome.js-svg image

---

25 - ./examples/is-valid-params.js

```
    /**
 * @param {string} s
 * @return {boolean}
 */
function isValid(s) {
    let valid = false;
    let c;
    let dict = {
      '{': '}',
      '(': ')',
      '[': ']'
    };
    let stack = [];
    for (let i = 0; i < s.length; i++) {
      c = s[i];
      if (dict[c]) {
        stack.push(c);
      } else {
        if (dict[stack.pop()] !== c) {
          return false;
        }
      }
    }
    return (stack.length === 0);
  };
```

Flowchart

./examples/is-valid-params.js-svg image

26 - ./examples/knapsack.js

```
   function max(a, b) {
   return (a > b) ? a : b;
}

function knapsack(capacity, size, value, n) {
   if (n == 0 || capacity == 0) {
      return 0;
   }
   if (size[n-1] > capacity) {
      return knapsack(capacity, size, value, n-1);
   }
   else {
      return max(value[n-1] +
                 knapsack(capacity-size[n-1], size, value, n-1),
                 knapsack(capacity, size, value, n-1));
   }
}

var value = [4,5,10,11,13];
var size = [3,4,7,8,9];
var capacity = 16;
var n = 5;
```

Flowchart

./examples/knapsack.js-svg image

27 - ./examples/kth-largest-element.js

```
   /*
## Kth Largest Element in an Array

Find the kth largest element in an unsorted array. Note that it is the kth
largest element in the sorted order, not the kth distinct element.

Example 1:
```

Input: [3,2,1,5,6,4] and k = 2 Output: 5

```
Example 2:
```

Input: [3,2,3,1,2,4,5,5,6] and k = 4 Output: 4

```
Note:
- You may assume k is always valid, 1 ≤ k ≤ array's length.

*/
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */
var findKthLargest = function (nums, k) {
    var sorted = nums.sort(function (a, b) {
        return a - b;
    });
    var array = [];

    for (var i = sorted.length - 1; i >= sorted.length - k; i--) {
        array.push(sorted[i]);
    }

    return array.pop();
};
```

Flowchart

./examples/kth-largest-element.js-svg image

---

28 - ./examples/length-of-longest-substring.js

```
    /**
 *
 * Using a map to keep track of the count of each appeared characters.
Once the count of character at right is greater than 1,
we will try to move left pointer to approach to right until the count of
that character is less or equal to 1.
We update the max length of non-repeating substring during looping.
 *
 *
 */
const lengthOfLongestSubstring = function (str) {
    if (str.length === 0 || str === null) return 0
    const map = {};
    let len = str.length
    let left = 0;
    let right = 0;
    let max = 0;
    for (; right < len; right++) {
        let ch = str.charAt(right)
        map[ch] ? map[ch]++ : map[ch] = 1
        while (map[ch] > 1 && left < right) {
            map[str.charAt(left)]--
                left++
        }
        console.log(map)
        max = Math.max(max, right - left + 1)
    }
    return max
}

console.log(lengthOfLongestSubstring("pwwkew"));
```

Flowchart

./examples/length-of-longest-substring.js-svg image

---

29 - ./examples/letter-combinations.js

```

    const mappings = {
    1: "",
    2: "abc",
    3: "def",
    4: "ghi",
    5: "jkl",
    6: "mno",
    7: "pqrs",
    8: "tuv",
    9: "wxyz"
};

var letterCombinations = function (digits) {
    if (digits == null || digits === "") {
        return [];
    }
    let res = []; // initialize the result array
    let currIdx = 0; // keep track of the current index of digit we are
looking at
    let currStr = ""; // keep track of the current substring we are
exploring
    backtracking(res, digits, currIdx, currStr); // start recursion
    return res;
};
```

```
var backtracking = function (res, digits, currIdx, currStr) {
    if (currIdx === digits.length) {
        return res.push(currStr); // one of the solution now is complete,
push it to the array
    }

    const c = digits[currIdx]; // get the current character
    const mapping = mappings[c]; // get its mapping

    for (const s of mapping) { // iterate through every character in the
mapping
        currStr += s;
        backtracking(res, digits, currIdx + 1, currStr); // recursion
        currStr = currStr.slice(0, -1); // revert currStr back
    }
};

//["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
console.log(letterCombinations('23'));
```

Flowchart

./examples/letter-combinations.js-svg image

---

30 - ./examples/level-order.js

```
    var levelOrder = function (root) {
    if (root === null) {
        return [];
    }
    let result = [];
    let queue = [];

    queue.push(root);

    while (queue.length > 0) {
        let size = queue.length;
        let current = [];
        for (let i = 0; i < size; i++) {
            let head = queue.shift();
            current.push(head.val);
            if (head.left !== null) {
                queue.push(head.left)
            }
            if (head.right !== null) {
                queue.push(head.right)
            }
        }
        result.push(current);
    }
    return result;
};
```

Flowchart

./examples/level-order.js-svg image

---

31 - ./examples/longest-common-prefix.js

```
    /**
 * @param {string[]} strs
 * @return {string}
 */
var longestCommonPrefix = function (strs) {
    if (strs && strs.length === 0) {
        return '';
    }
    strs.sort((prev, next) => prev.length - next.length)
    shortestStr = strs[0]
    length = shortestStr && shortestStr.length
    if (!length) {
        return ''
    }
    for (let i = length; i > 0; i--) {
        const searchStr = shortestStr.substr(0, i);
        flag = strs.every((item) => item && item.startsWith &&
item.startsWith(searchStr))
        if (flag) {
            return searchStr
            break;
        }
    }
    return ''
};
```

Flowchart

./examples/longest-common-prefix.js-svg image

---

32 - ./examples/longest-palindromic-substring.js

```
    /*
## Longest Palindromic Substring
Given a string s, find the longest palindromic substring in s. You may
assume that the maximum length of s is 1000.
```

```
Example 1:
```

Input: "babad" Output: "bab"

```
Note: "aba" is also a valid answer.


Example 2:
```

Input: "cbbd" Output: "bb"

```
 */
/**
 * @param {string} s
 * @return {string}
 */
var longestPalindrome = function(s) {
    console.log(s);
};

function isPalindrome(word) {
    return word === word.split('').reverse().join('');
}




let isEven = (n) => n % 2 !== 1;
let isOdd = (n) => n % 2 === 1;



//[2,4,6,8,10].map( val => console.log(isEven(val)));
[1,3,5,7,9, 11, 13].map( val => console.log(isOdd(val)))






console.log(longestPalindrome('babad'));
```

Flowchart

./examples/longest-palindromic-substring.js-svg image

33 - ./examples/max-depth-of-binary-tree.js

```
    /*
## Maximum Depth of Binary Tree
Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the
root node down to the farthest leaf node.

> Note: A leaf is a node with no children.

Example:

Given binary tree [3,9,20,null,null,15,7],
```

```
3
```

/
9 20 /
15 7

```
return its depth = 3.
*/
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var maxDepth = function(root) {
    if (root === null) {
        return 0;
    }
    let left_depth = maxDepth(root.left);
    let right_depth = maxDepth(root.right);
    return Math.max(left_depth, right_depth) + 1;
};
```

Flowchart

./examples/max-depth-of-binary-tree.js-svg image

---

34 - ./examples/max-sub-array-len.js

```
    var maxSubArrayLen = function (nums, k) {
    let subarrayLength = 0;
    let sum = 0;
    let map = {
        0: -1
    };
    nums.forEach((number, i) => {
        sum += number;
        if (!map.hasOwnProperty(sum)) {
            map[sum] = i;
        }
        if (map.hasOwnProperty(sum - k)) {
            subarrayLength = Math.max(subarrayLength, i - map[sum - k]);
        }
    });
    return subarrayLength;
};

console.log(maxSubArrayLen([1, -1, 5, -2, 3], 3));
```

Flowchart

./examples/max-sub-array-len.js-svg image

---

35 - ./examples/max-sub-array.js

```


/*
Basic idea is to keep track of the largest sum at current index and to
achieve that we can compare the number on current index and the sum of the
number on current index plus previous sum (previous sum set to -Infinity).
If the result is greater than the number on current index that means it
has the
potential to be included in the contiguous subarray so we add to previous
sum.

But if the result is less than the number on current index that means we
can simply
ignore the result and set the previous sum to be the current number.

And as we compute previous sum, we can also compute max by find out what
the maximum previous sum is. (we set max to nums[0] in case nums has only
1 element)
*/
```

```
var maxSubArray = function(nums) {
    let currentMax = nums[0];
    let previousSum = -Infinity;

    for (let num of nums) {
        //console.log(previousSum);
        previousSum = Math.max(num, num + previousSum);
        currentMax = Math.max(previousSum, currentMax);
    }

    return currentMax;
};
console.log(maxSubArray([-2,1,-3,4,-1,2,1,-5,4]));
```

Flowchart

./examples/max-sub-array.js-svg image

---

36 - ./examples/maximum-sub-array.js

```
    /*
# Maximum Subarray

Given an integer array nums, find the contiguous subarray (containing at
least one number) which has the largest sum and return its sum.

Example:
```

Input: [-2, 1, -3, 4, -1, 2, 1, -5, 4], Output: 6 Explanation: [4,-1,2,1] has the largest sum = 6.

```
Follow up:
If you have figured out the O(n) solution, try coding another solution
using the divide and conquer approach, which is more subtle.

### Kadane’s Algorithm:
```

Initialize: max_so_far = 0 max_ending_here = 0

Loop for each element of the array (a) max_ending_here = max_ending_here + a[i] (b) if(max_ending_here < 0) max_ending_here = 0 (c) if(max_so_far < max_ending_here) max_so_far = max_ending_here return max_so_far

```
*/
```

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var maxSubArray = function (nums) {
    let max_so_far = Number.MIN_SAFE_INTEGER;
    let max_ending_here = 0;
    let size = nums.length;
    if(size === 1){
        return nums[0]
    }
    for (let index = 0; index < size; index++) {
        const num = nums[index];
        max_ending_here = max_ending_here + num;
        if(max_ending_here < num){
            max_ending_here = num;
        }
        if(max_so_far < max_ending_here){
            max_so_far = max_ending_here;
        }
        console.log(max_so_far);
    }
    return max_so_far;
};

var maxSubArray2 = function(nums) {
    for (let i = 1; i < nums.length; i++){
        nums[i] = Math.max(nums[i], nums[i] + nums[i - 1]);
    }
    return Math.max(...nums);
};
let input = [-2, 1, -3, 4, -1, 2, 1, -5, 4];


console.log(maxSubArray(input));
console.log(maxSubArray([-1]));
console.log(maxSubArray([-2, -1]));
console.assert(maxSubArray([-1]) === -1);

console.log(maxSubArray2(input));
console.log(maxSubArray2([-2, -1]));
```

Flowchart

./examples/maximum-sub-array.js-svg image

---

37 - ./examples/merge-arrays.js

```
    //imperative:
const mergeArrays_ = function (arrays) {
    let count = arrays.length;
    let newArray = [];
    let k = 0;
    for (let i = 0; i < count; i++) {
        for (let j = 0; j < arrays[i].length; j++) {
            newArray[k++] = arrays[i][j];
        }
    }
    return newArray;
};
console.log(mergeArrays_([
    [1, 2, 3],
    [4, 5],
    [6]
]));

// function-oriented
const mergeArrays = (...arrays) => [].concat(...arrays);
console.log(mergeArrays([1, 2, 3], [4, 5], [6]));
```

Flowchart

./examples/merge-arrays.js-svg image

---

38 - ./examples/merge-two-linked-lists.js

```
    /*
## Merge Two Sorted Lists

Merge two sorted linked lists and return it as a new list. The new list
should be made by splicing together the nodes of the first two lists.

Example:
```

Input: 1->2->4, 1->3->4 Output: 1->1->2->3->4->4

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

```
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */
var mergeTwoLists = function (l1, l2) {

};

function mergeTwoLists(l1, l2) {
    // base case
    if (!l1 || !l2) return l1 || l2;


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

Flowchart

./examples/merge-two-linked-lists.js-svg image

---

39 - ./examples/min-sub-array-len.js

```
    var minSubArrayLen = function(s, nums) {
    let ret = Infinity;
    for (let i = 0, j = 0, sum = 0; j <= nums.length; ) {
        if (sum < s || i >= j) {
            sum += nums[j++]
        } else {
            ret = Math.min(ret, j - i)
            sum -= nums[i++]
        }
    }
    return ret === Infinity ? 0 : ret
};
//minSubArrayLen(7, [2,3,1,2,4,3]);
```

Flowchart

./examples/min-sub-array-len.js-svg image

---

40 - ./examples/min-swaps.js

```
function minSwaps2(arr) {
    let answer = 0;
    arr.sort((a, b) => {
        answer++;
        return a > b;
    })
    console.log(arr);
    return answer % arr.length + 1;
}
console.log(minSwaps2([4, 3, 1, 2]));



function swap(arr, i, j) {
    var temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}


/**
 * Create a pivot point and sort that way
 * @param {*} arr
 */
function minSwaps3(arr) {
    let map = new Map();
    let answer = 0;
    let i = 0;
    arr.map((val, index) => map.set(val, index));

    /*
    map[Symbol.iterator] = function* () {
        yield* [...this.entries()].sort((a, b) => a[1] - b[1]);
    }
    */

    for (let [key, value] of map) {
        console.log(key + ' ' + value);

        for (let index = 0; index < arr.length; index++) {
            if (key !== arr[index]) {
                swap(arr, value, index);
                console.log('swap', key, index);
            }
            console.log(index);
        }

    }


    console.log(arr, map);
    return answer;
}

console.log(minSwaps3([4, 3, 1, 2]));
//console.log(minSwaps3([7, 1, 3, 2, 4, 5, 6]));


function swap2(arr, i) {
    let temp = arr[arr[i] - 1];
    arr[arr[i] - 1] = arr[i];
    arr[i] = temp;
}

function minimumSwaps(arr) {
    let count = 0;

    for (let i = 0; i < arr.length; i++) {
        //CHeck if next number is 1
        console.log(i, arr[i])
        if (arr[i] !== i + 1) {
            console.log('Swap',i, i + 1);
            swap2(arr, i);
            count++;
        }
    }
    return count;
}

console.log(minimumSwaps([4, 3, 1, 2]));
```

Flowchart

./examples/min-swaps.js-svg image

---

41 - ./examples/move-zeros.js

```
    /**
 * 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 left = 0;
    let len = nums.length;
    let numNonZero = 0;
    while (left < len) {
        if (nums[left] != 0) {
            let tmp = nums[numNonZero]
            nums[numNonZero] = nums[left]
            nums[left] = tmp;
            numNonZero += 1;
            left += 1
        } else {
            left += 1
        }
    }
    return nums;
};



var moveZeroes2 = function (nums) {
    let firstPos = 0;
    for (let i = 0; i < nums.length; i++) {
        if (nums[i] !== 0) {
            let temp = nums[firstPos];
            nums[firstPos] = nums[i];
            nums[i] = temp;
            firstPos++;
        }
    }
    return nums;
};

//console.time('moveZeroes2');
//console.log(moveZeroes2([0, 1, 0, 3, 12]));
//console.log(moveZeroes2([0, 0, 3, 1]));
//console.log(moveZeroes2([]));
//console.timeEnd('moveZeroes2');






/*
## 3Sum

Given an array nums of n integers, are there elements a, b, c in nums such
that a + b + c = 0?
Find all unique triplets in the array which gives the sum of zero.

> Note: The solution set must not contain duplicate triplets.
```

```
Example:

Given array nums = `[-1, 0, 1, 2, -1, -4]`

A solution set is:
```

[[-1, 0, 1], [-1, -1, 2]]

```

*/

/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var threeSum = function (nums) {
    nums.sort((a, b) => a - b);

    let results = [];
    let totalSum = 0;
    let i = 0
    while (i < nums.length - 2) {
        let j = i + 1;
        let k = nums.length - 1;
        while (j < k) {
            const sum = nums[i] + nums[j] + nums[k]
            if (sum < totalSum) {

                while (nums[++j] === nums[j - 1]);
            } else if (sum > totalSum) {
                while (nums[--k] === nums[k + 1]);
            } else {
                results.push([nums[i], nums[j], nums[k]])
                while (nums[++j] === nums[j - 1]);
                while (nums[--k] === nums[k + 1]);
            }
        }
        while (nums[++i] === nums[i - 1]);
    }
    return results;
};
//console.log(threeSum([-1, 0, 1, 2, -1, -4]));

/**
 * Validate if a given string is numeric.

Some examples:
"0" => true
" 0.1 " => true
"abc" => false
"1 a" => false
"2e10" => true
 * @param {string} s
 * @return {boolean}
 */
var isNumber = function (s) {

    return s.trim().length === 0 ? false : !isNaN(+s);
};

console.log(isNumber('1'));
console.log(isNumber('0'));
console.log(isNumber(' b '));
console.assert(isNumber(' 0.1 '));
console.assert(!isNumber(' b '));
console.assert(!isNumber('abc'));
console.assert(!isNumber('1 a'));
console.assert(isNumber('2e10'));





/**
 *
 * Maximum Size Subarray Sum Equals k
Given an array nums and a target value k, find the maximum length of a
subarray that sums to k. If there isn't one, return 0 instead.

Note:
The sum of the entire nums array is guaranteed to fit within the 32-bit
signed integer range.

Example 1:

Input: nums = [1, -1, 5, -2, 3], k = 3
Output: 4
Explanation: The subarray [1, -1, 5, -2] sums to 3 and is the longest.
Example 2:

Input: nums = [-2, -1, 2, 1], k = 1
Output: 2
Explanation: The subarray [-1, 2] sums to 1 and is the longest.
Follow Up:
Can you do it in O(n) time?

The intuition is to create a map where u keep track of previous sums and
their indecies.
You keep adding to the sum until you get to a point

- where the sum - k equals to one of the items already in the array.
- this would mean that the sum of all the numbers after that sum - k
number is k.

This would mean we have added up to the desired result, and we figure out
how long that array is based on the indecies.
 */
```

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */
var maxSubArrayLen = function (nums, k) {
    let subarrayLength = 0;
    let sum = 0;
    let map = {
        0: -1
    };
    nums.forEach((number, i) => {
        sum += number;
        if (!map.hasOwnProperty(sum)) {
            map[sum] = i;
        }
        if (map.hasOwnProperty(sum - k)) {
            subarrayLength = Math.max(subarrayLength, i - map[sum - k]);
        }
    });
    return subarrayLength;
};

console.log(maxSubArrayLen([1, -1, 5, -2, 3], 3));




function removeInvalidParentheses(s) {
    let queue = new Set([s]);
    while (queue.size) {
        const next = new Set();
        for (let v of queue) {
            if (isValid(v)) {
                return [...queue].filter(isValid);
            }

            for (let i = 0; i < v.length; i++) {
                next.add(v.slice(0, i) + v.slice(i + 1));
            }
        }
        queue = next;
    }
    return [''];
}

function isValid(str) {
    let bal = 0;
    for (let ch of str) {
        if (ch === '(') {
            bal++;
        } else if (ch === ')') {
            bal--;
        }
        if (bal < 0) {
            return false;
        }
    }
    return bal === 0;
}

/**

Remove the minimum number of invalid parentheses in order to make the
input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses (
and ).

Example 1:

Input: "()())()"
Output: ["()()()", "(())()"]
 */

 console.log(removeInvalidParentheses("(a)())()"))
 console.log(removeInvalidParentheses("()()(())"))
```

Flowchart

./examples/move-zeros.js-svg image

---

42 - ./examples/nested-list-weight-sum.js

```
    function depthSumHelper(list, depth) {
    let sum = 0;
    for (let n in list) {
        console.log(n);
        if (n.isInteger()) {
            sum += n.getInteger() * depth;
        } else {
            sum += depthSumHelper(n.getList(), depth + 1);
        }
    }
    return sum;
}

function depthSum(nestedList) {
    return depthSumHelper(nestedList, 1);
}

function swap(array, index1, index2) {
    var aux = array[index1];
    array[index1] = array[index2];
    array[index2] = aux;
}

function bubbleSort(array) {
    let length = array.length;
    for (let i = 0; i < length; i++) {
        console.log(i, array)
        for (let j = 0; j < length - 1; j++) {
            if (array[j] > array[j + 1]) {
                swap(array, j, j + 1);
            }
        }
    }
    return array;
}

function bubbleSort2(array) {
    var length = array.length;
    var cost = 0;
    for (var i = 0; i < length; i++) { //{1}
        cost++;
        for (var j = 0; j < length - 1; j++) { //{2}
            cost++;
            if (array[j] > array[j + 1]) {
                swap(array, j, j + 1);
            }
        }
    }
    console.log(`cost for bubbleSort with input size ${length} is
${cost}`);
}

//console.log(bubbleSort([7, 2, 4, 5, 9, 8, 1, 3, 6]));
```

Flowchart

./examples/nested-list-weight-sum.js-svg image

---

43 - ./examples/number-of-islands.js

```
    /*
Given a 2d grid map of '1's (land) and '0's (water), count the number of
islands.
```

```
An island is surrounded by water and is formed by connecting adjacent
lands horizontally or vertically.

> You may assume all four edges of the grid are all surrounded by water.

Example 1:
```

Input: 11110 11010 11000 00000

Output: 1

```


Example 2:
```

Input: 11000 11000 00100 00011 ['1', '1', '1', '0', '0'], ['0', '0', '1', '0', '0'], ['0', '0', '0', '1', '1'] Output: 3

```


*/
/**
 * @param {character[][]} grid
 * @return {number}
 */
var numIslands = function (grid) {
    let count = 0;

    const height = grid.length;
    const width = grid[0].length;
    const visited = Array(height * width).fill(false);

    for (let i = 0; i < height; i++) {
        for (let j = 0; j < width; j++) {
            const col = grid[i][j];
            //console.log(col);
            if (!visited[i * width + j] && grid[i][j] === 1) {
                count++;
                dfs(i, j);
            }
        }

    }

    function dfs(r, c) {
        visited[r * width + c] = true;
        let dr = [r - 1, r + 1, r, r];
        let dc = [c, c, c - 1, c + 1];
        for (let i = 0; i < 4; i++) {
            if (dr[i] >= 0 && dr[i] < height && dc[i] >= 0 && dc[i] <
width && grid[dr[i]][dc[i]] === 1 && !visited[dr[i] * width + dc[i]]) {
                dfs(dr[i], dc[i]);
            }
        }
        return;
    }

    return count;
};


console.log(numIslands([
    []
]));
console.log(numIslands([
    [1, 1, 1, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 0, 0, 1, 1]
]));
```

Flowchart

./examples/number-of-islands.js-svg image

---

44 - ./examples/one-edit-away.js

```

```

Flowchart

./examples/one-edit-away.js-svg image

---

45 - ./examples/pascal-triangle-2.js

```
    function pascalTriangle(lineNumber) {
    const currentLine = [1];
    const currentLineSize = lineNumber + 1;
    for (let numIndex = 1; numIndex < currentLineSize; numIndex += 1) {
      currentLine[numIndex] = currentLine[numIndex - 1] * (lineNumber -
numIndex + 1) / numIndex;
    }
    return currentLine;
  }
  /**
   * @param {number} rowIndex
   * @return {number[]}
   */
```

```
    var getRow = function(rowIndex) {
        return pascalTriangle(rowIndex);
    };
```

Flowchart

./examples/pascal-triangle-2.js-svg image

---

46 - ./examples/path-sum.js

```
    /*
## 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.

> Note: A leaf is a node with no children.

Example:

Given the below binary tree and sum = 22,
```

```
  5
 / \
4   8
```

/ /
11 13 4 / \
7 2 1

```
return `true`, as there exist a root-to-leaf path `5->4->11->2` which sum
is `22`.

*/
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * Time Complexity: O(n)
 * @param {TreeNode} root
 * @param {number} sum
 * @return {boolean}
 */
var hasPathSum = function(root, sum) {
    let node = root;
    if(!node){
        return false;
    }
    if(node === null){
        return (sum === 0);
    }
    let answer = false;
    let subsum = sum - node.val;
    if(subsum === 0 && node.left == null && node.right == null){
        return true;
    }
    if(node.left !== null){
       answer = answer || hasPathSum(node.left, subsum);
    }
    if(node.right !== null){
       answer = answer || hasPathSum(node.right, subsum);
    }
    return answer;
};
```

Flowchart

./examples/path-sum.js-svg image

---

47 - ./examples/permutations.js

```
    /*
# Permutations

Given a collection of distinct integers, return all possible permutations.

Example:
```

Input: [1,2,3] Output: [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]

```


*/
const permute = function (nums) {
    if (nums.length <= 1) {
        return [nums]
    }
    let res = []
    let last = nums.pop()
    let prevPerms = permute(nums)
    for (let prevPerm of prevPerms) {
        for (let i = 0; i <= prevPerm.length; i++) {
            let clone = prevPerm.slice(0)
            clone.splice(i, 0, last)
            res.push(clone)
        }
    }
    return res;
}

console.log(permute([1, 2, 3]))
```

Flowchart

./examples/permutations.js-svg image

---

48 - ./examples/pivot-index.js

```
    var pivotIndex = function(nums) {
    let i = 0;
     // Run through all the numbers in the array
    for (; i < nums.length; i++) {

        // Get the sum on either half of the potential pivot
        let leftSum = nums.slice(0, i).reduce((a, b) => a + b, 0)
        let rightSum = nums.slice(i+1).reduce((a, b) => a + b, 0)

        // Check if pivot
        if (leftSum === rightSum) { return i; }
    }

    // If we have run through the whole array and not found a pivot then
return -1
    return -1;
};

console.log(pivotIndex([1,7,3,6,5,6]));
```

Flowchart

./examples/pivot-index.js-svg image

---

49 - ./examples/plus-one.js

```
    /**
 * ## Plus One
 * @param {number[]} digits
 * @return {number[]}
 */
var plusOne = function(digits) {
    let i = digits.length;
    while (i--) {
        digits[i] = (digits[i] + 1) % 10;
        if (digits[i] > 0) {
            return digits;
        }
    }
    return [1, ...digits];
  };
```

Flowchart

./examples/plus-one.js-svg image

---

50 - ./examples/read-n-characters-given-read4.js

```
    /**
 * Definition for read4()
 *
 * @param {character[]} buf Destination buffer
 * @return {number} The number of characters read
 * read4 = function(buf) {
 *     ...
 * };
 */

/**
 * @param {function} read4()
 * @return {function}
 */
var solution = function (read4) {
    var internalBuf = [];

    /**
     * @param {character[]} buf Destination buffer
     * @param {number} n Maximum number of characters to read
     * @return {number} The number of characters read
     */
    return function (buf, n) {
        let readChars = 0;
        while (n > 0) {
            if (internalBuf.length === 0) {
                if (read4(internalBuf) === 0) {
                    return readChars;
                }
            }

            buf.push(internalBuf.shift());
            readChars++;
            n--;
        }
        return readChars;
    };
};
```

Flowchart

./examples/read-n-characters-given-read4.js-svg image

---

51 - ./examples/remove-duplicates.js

```
    /**
 * @param {number[]} nums
 * @return {number}
 */
var removeDuplicates = function (nums) {
    if (!nums || !nums.length) {
        return 0
    }
    let newIndex = 0;
    let p1 = 0;
    let p2 = 0;
    let len = nums.length;

    //while p1 < length
    while (p1 < len) {
        // console.log(p1, p2);
        while (p2 < len && nums[p1] == nums[p2]) {
            p2 += 1;

        }
        nums[newIndex] = nums[p1];
        newIndex += 1
        p1 = p2;
    }

    return newIndex;
};
console.time('removeDuplicates');
console.log(removeDuplicates([1, 2, 1, 1, 3]))
console.log(removeDuplicates([1, 2]))
console.timeEnd('removeDuplicates');
```

Flowchart

./examples/remove-duplicates.js-svg image

---

52 - ./examples/remove-element.js

```
    var removeElement = function (nums, val) {
    for (var i = 0; i < nums.length; i++) {
        if (nums[i] === val) {
            nums.splice(i, 1);
            i--;
        }
    }

    return nums.length;
};
```

Flowchart

./examples/remove-element.js-svg image

---

53 - ./examples/remove-invalid-parentheses.js

```
    function openBrace(s) {
    return ['('].includes(s)
}

function closeBrace(s) {
    return [')'].includes(s)
}

function removeInvalidParentheses(s) {
    let queue = new Set([s]);
    while (queue.size) {
        const next = new Set();
        for (let v of queue) {
            if (isValid(v)) {
                return [...queue].filter(isValid);
            }

            for (let i = 0; i < v.length; i++) {
                next.add(v.slice(0, i) + v.slice(i + 1));
            }
        }
        queue = next;
    }
    return [''];
}
```

```
function isValid(str) {
    let bal = 0;
    for (let ch of str) {
        if (ch === '(') {
            bal++;
        } else if (ch === ')') {
            bal--;
        }
        if (bal < 0) {
            return false;
        }
    }
    return bal === 0;
}

/**
 * @param {string} s
 * @return {string[]}
 */
var removeInvalidParentheses2 = function (s) {
    let dict = {
        '{': '}',
        '(': ')',
        '[': ']'
    };
    let stack = [];
    let input = s.split('');

    function closesMostRecentBrace(char) {
        console.log(s[s.length - 1])
        return (s[s.length - 1] === char);
    }

    for (let i = 0; i < input.length; i++) {
        const char = input[i];

        //# If the character is an opening brace, we push it onto the
stack:
        if (openBrace(char)) {
            stack.push(char);
            console.log('open', s.length);
        }
        //# If the character closes the most recent opening brace,
        else if (closeBrace(char)) {
            //Pop from stack
            if (closesMostRecentBrace(char)) {
                stack.pop();
                console.log('closed');
            } else {
                //# if the character does NOT close the most recent
opening brace
                console.log('the character does NOT close the most recent
opening brace');
```

```
                }
            }
        }
        console.log(stack);
        console.log(input);
    };

    console.log(removeInvalidParentheses("(b)(c))"));
    console.log(removeInvalidParentheses2("(b)(c))"));
    //console.log(removeInvalidParentheses("()())()"));
```

Flowchart

./examples/remove-invalid-parentheses.js-svg image

---

54 - ./examples/remove-nth-node-from-end-of-list.js

```
        /**
     * Definition for singly-linked list.
     * function ListNode(val) {
     *     this.val = val;
     *     this.next = null;
     * }
     */
    /**
     * @param {ListNode} head
     * @param {number} n
     * @return {ListNode}
     */
    var removeNthFromEnd = function (head, n) {
        if (!head) return head;

        let p1 = head;
        let p2 = head
        while (n--) {
            if (p2 === null) return null
            p2 = p2.next
        }
        while (p1 && p2.next) {
            p1 = p1.next
            p2 = p2.next
        }
        let removed = p1.next
        p1.next = removed.next
        removed.next = null
        return head
    };

    const removeNthFromEnd2 = function (head, n) {
```

```
    if (!head) return head;

    let prev = head;
    let curr = head.next;

    while (curr) {
        if (n > 0) {
            n -= 1;
        } else {
            prev = prev.next;
        }
        curr = curr.next;
    }

    if (n === 1) return head.next;

    prev.next = prev.next.next || null;
    return head;
};
```

Flowchart

./examples/remove-nth-node-from-end-of-list.js-svg image

---

55 - ./examples/reverse-words-2.js

```
    /**

Given a string, you need to reverse the order of characters in each word
within a sentence while still preserving whitespace and initial word
order.

Example 1:
Input: "Let's take LeetCode contest"
Output: "s'teL ekat edoCteeL tsetnoc"

Note: In the string, each word is separated by single space and there will
not be any extra space in the string.
 * @param {*} str
 */
var reverseWords2 = function (str) {
    let output = [];
    let words = str.split(' ');
    for (let index = 0; index < words.length; index++) {
        let word = words[index];
        let temp = [];
        for (let index = word.length - 1; index > -1; index--) {
            let w = word[index];
            temp.push(w);
        }
        output.push(temp.join(''));
    }
    return output.join(' ');
};
console.log(reverseWords2("Let's take LeetCode contest") === "s'teL ekat
edoCteeL tsetnoc");
```

Flowchart

./examples/reverse-words-2.js-svg image

---

56 - ./examples/reverse-words.js

```
    /**
 * Given an input string, reverse the string word by word.

Example:

Input: "the sky is blue",
Output: "blue is sky the".
 */

/**
 * @param {string} str
 * @returns {string}
 */
var reverseWords = function (str) {
    let output = [];
    let words = str.split(' ');
    for (let index = words.length - 1; index > -1; index--) {
        let word = words[index];
        if (word) {
            //console.log(word);
            output.push(word);
        }
    }
    return output.join(' ');
};

console.log(reverseWords('the sky is blue'));
console.log(reverseWords(' the sky is blue '));
```

Flowchart

./examples/reverse-words.js-svg image

---

57 - ./examples/roman-to-integer.js

```
    // Roman numerals are represented by seven different symbols: I, V, X,
L, C, D and M.
//
// Symbol       Value
// I             1
// V             5
// X             10
// L             50
// C             100
// D             500
// M             1000
//
// For example, two is written as II in Roman numeral, just two one's
added together. Twelve is written as, XII, which is simply X + II. The
number twenty seven is written as XXVII, which is XX + V + II.
//
// Roman numerals are usually written largest to smallest from left to
right. However, the numeral for four is not IIII. Instead, the number four
is written as IV. Because the one is before the five we subtract it making
four. The same principle applies to the number nine, which is written as
IX. There are six instances where subtraction is used:
//
// I can be placed before V (5) and X (10) to make 4 and 9.
// X can be placed before L (50) and C (100) to make 40 and 90.
// C can be placed before D (500) and M (1000) to make 400 and 900.
//
// Given a roman numeral, convert it to an integer. Input is guaranteed to
be within the range from 1 to 3999.
//
// Example 1:
//
// Input: "III"
// Output: 3
//
// Example 2:
//
// Input: "IV"
// Output: 4
//
// Example 3:
//
// Input: "IX"
// Output: 9
//
// Example 4:
//
// Input: "LVIII"
// Output: 58
// Explanation: C = 100, L = 50, XXX = 30 and III = 3.
//
// Example 5:
//
// Input: "MCMXCIV"
// Output: 1994
// Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.

/**
 * @param {string} s
 * @return {number}
 */
function romanToInt(s) {
    const map = {
        I: 1,
        V: 5,
        X: 10,
        L: 50,
        C: 100,
        D: 500,
        M: 1000
    };

    let prevNum = 0;
    let sum = 0;

    for (let c of s) {
        const num = map[c];

        sum = prevNum >= num ?
            sum + num :
            sum + num - prevNum * 2;

        prevNum = num;
    }

    return sum;
}
```

Flowchart

./examples/roman-to-integer.js-svg image

---

58 - ./examples/rotate-array.js

```
    var rotate = function (nums, k) {
    nums.unshift.apply(nums, nums.splice(nums.length - k, k))

};
//console.log(rotate([-1], 2))
//console.log(rotate([1,2,3], 4))
```

```
/**
 * Input: "Let's take LeetCode contest"
Output: "s'teL ekat edoCteeL tsetnoc"
*/


let n = 0;
let d = 0;
let data = [];
//console.log(reverseWords2("Let's take LeetCode contest"));

//Solution using array.shift()
function getResultsUsingArrayShift(data, d) {
    let temp = data.slice(0);
    for (let i = 0; i < d - 1; i++) {
        let first = temp.shift();
        temp.push(first);
    }
    return temp;
}

function rotateArray(a, d) {
    let i = 0;
    let size = a.length;
    let rotations = d - 1;
    return a.reduce((arr, number) => {
        arr[(i + (size - rotations)) % size] = number;
        i++;
        return arr;
    }, [])
}
console.log(rotateArray([1, 2, 3, 4, 5], 5));
console.log(rotateArray([1, 2], 2));
```

Flowchart

./examples/rotate-array.js-svg image

---

59 - ./examples/serialize-and-deserialize-binary-tree.js

```
    /*
## Serialize and Deserialize Binary Tree

Serialization is the process of converting a data structure or object into
a sequence of bits so that it can be stored in a file or memory buffer, or
transmitted across a network connection link to be reconstructed later in
the same or another computer environment.
```

```
Design an algorithm to serialize and deserialize a binary tree. There is
no restriction on how your serialization/deserialization algorithm should
work. You just need to ensure that a binary tree can be serialized to a
string and this string can be deserialized to the original tree structure.

Example:
```

You may serialize the following tree:

```
1
```

/
2 3 /
4 5

as "[1,2,3,null,null,4,5]"

```


Clarification: The above format is the same as how LeetCode serializes a
binary tree. You do not necessarily need to follow this format, so please
be creative and come up with different approaches yourself.

> Note: Do not use class member/global/static variables to store states.
Your serialize and deserialize algorithms should be stateless.

*/
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */

/**
 * Encodes a tree to a single string.
 *
 * @param {TreeNode} root
 * @return {string}
 */
var serialize = function(root) {
    let str = '';
    let buildString = function(node) {
        if (!node) {
            str += '# ';
            return;
        }
        str += node.val + ' ';
        buildString(node.left);
        buildString(node.right);

    }
    buildString(root);
    return str;
};

/**
 * Decodes your encoded data to tree.
 *
 * @param {string} data
 * @return {TreeNode}
 */
var deserialize = function(data) {
    let values = data.split(' '), len = values.length, idx = 0;
    let buildTree = function() {
        if (idx >= len || values[idx] === '#') {
            return null;
        }
        let node = new TreeNode(parseInt(values[idx]));
        ++idx;
        node.left = buildTree();
        ++idx;
        node.right = buildTree();
        return node;
    }
    return buildTree();
};

/**
 * Your functions will be called as such:
 * deserialize(serialize(root));
 */
```

Flowchart

./examples/serialize-and-deserialize-binary-tree.js-svg image

---

60 - ./examples/set-matrix-zeros.js

```
    /*
## Set Matrix Zeroes

Given a m x n matrix, if an element is 0, set its entire row and column to
0. Do it in-place.

Example 1:
```

Input: [[1,1,1], [1,0,1], [1,1,1]] Output: [[1,0,1], [0,0,0], [1,0,1]]

```


*/
/**
 * @param {number[][]} matrix
 * @return {void} Do not return anything, modify matrix in-place instead.
 */
var setZeroes = function (matrix) {
    let rowHasZero = false;
    let colHasZero = false;

    let rows = [];
    let cols = [];

    function nullifyRow(matrix, row) {
        for (let index = 0; index < matrix.length; index++) {
            matrix[row][index] = 0;
        }
    }

    function nullifyCol(matrix, col) {
        for (let index = 0; index < matrix.length; index++) {
            matrix[index][col] = 0;
        }
    }

    //check if first row has 0
    for (let j = 0; j < matrix[0].length; j++) {
        if (matrix[0][j] === 0) {
            rowHasZero = true;
            break;
        }
    }
    //check if first col has 0
    for (let i = 0; i < matrix.length; i++) {
        if (matrix[i][0] === 0) {
            colHasZero = true;
            break;
        }
    }

    // check for zeros in rest of array
    for (let i = 1; i < matrix.length; i++) {
        for (let j = 1; j < matrix[0].length; j++) {
            if (matrix[i][j] === 0) {
                rows[i] = true;
                cols[j] = true;
            }
        }
    }

      // Nullify rows
      for (let i = 1; i < matrix.length; i++) {
        if (matrix[i][0] === 0) {
            nullifyRow(matrix, i);
        }
    }
    // Nullify cols
    for (let j = 1; j < matrix[0].length; j++) {
        if (matrix[0][j] === 0) {
            nullifyCol(matrix, j);
        }
    }

    if(rowHasZero){
        nullifyRow(matrix, 0)
    }
    if(colHasZero){
        nullifyCol(matrix, 0)
    }
    return matrix;
};

console.log(
    setZeroes([
        [1, 1, 1],
        [1, 0, 1],
        [1, 1, 1]
    ])
);
console.log(
    setZeroes([
        [0,1,2,0],
        [3,4,5,2],
        [1,3,1,5]
      ])
);
var solution = function(isBadVersion) {
    /**
      * @param {integer} n Total versions
      * @return {integer} The first bad version
      */
     return function(n) {
         var min = 1;
         var max = n;
         var bad = -1;
         while (min <= max) {
             var mid = Math.floor((min+max)/2);
             if (isBadVersion(mid)) {
                 bad = mid;
                 max = mid-1;
             }
             else {
                 min = mid+1;
             }
         }
```

```
            return bad;
        };
    };

    console.log(solution(5)())
```

Flowchart

./examples/set-matrix-zeros.js-svg image

---

61 - ./examples/single-number.js

```
/**
 *

 ## Single Number
Given a non-empty array of integers, every element appears twice except
for one. Find that single one.

> Note: Your algorithm should have a linear runtime complexity O(n).

#### Example 1:
```

Input: [2,2,1] Output: 1

```
#### Example 2:
```

Input: [4,1,2,1,2] Output: 4

```
*/
/**
 * @param {number[]} nums
 * @return {number}
 */
var singleNumber = function(nums) {
    for (let i = 0; i < nums.length; i++) {
        let n1 = nums[i];
        if (nums.indexOf(n1) === nums.lastIndexOf(n1)) {
            return n1;
        }
    }
};
```

```
console.log(singleNumber([2, 2, 1]));
//console.log(singleNumber([4,1,2,1,2]));
console.log(singleNumber([1,1,1,1,1,5]));
```

Flowchart

./examples/single-number.js-svg image

---

62 - ./examples/sorted-array-to-bst.js

```
    // Given an array where elements are sorted in ascending order,
convert it to a height balanced BST.
//
// For this problem, a height-balanced binary tree is defined as a binary
tree in which the depth of the two subtrees of every node never differ by
more than 1.
//
// Example:
//
// Given the sorted array: [-10,-3,0,5,9],
//
// One possible answer is: [0,-3,9,-10,null,5], which represents the
following height balanced BST:
//
//       0
//      / \
//    -3   9
//    /   /
//  -10  5

/**
 * 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) {
    if (!nums.length) return null;

    const mid = Math.floor(nums.length / 2);

    let node = new TreeNode(nums[mid]);
    node.left = sortedArrayToBST(nums.slice(0, mid));
    node.right = sortedArrayToBST(nums.slice(mid + 1)); // make sure + 1,
because mid number is root node, so need skip it

    return node;
}
```

Flowchart

./examples/sorted-array-to-bst.js-svg image

---

63 - ./examples/spiral-order.js

```
    /*
k - starting row index
m - ending row index
l - starting column index
n - ending column index
i - iterator
*/
/**
 * @param {number[][]} matrix
 * @return {number[]}
 */
var spiralOrder = function (matrix) {
    var result = [];
     if (!matrix.length) {
         return result;
     }
     var i = 0,
         a = matrix,
         m = matrix.length,
         n = matrix[0].length,
         k = 0,
         l = 0;

     while (k < m && l < n) {
         //first row
         for (i = l; i < n; ++i) {
             result.push(a[k][i]);
         }
         k++;
         //last column
         for (i = k; i < m; ++i) {
             result.push(a[i][n - 1]);
         }
         n--;
         // Print the last row from the remaining rows
         if (k < m) {
             for (i = n - 1; i >= l; --i) {
                 result.push(a[m - 1][i]);
             }
```

```
            m--;
        }
        // Print the first column from the remaining columns
        if (l < n) {
            for (i = m - 1; i >= k; --i) {
                result.push(a[i][l]);
            }
            l++;
        }
    }
    return result;
};
```

Flowchart

./examples/spiral-order.js-svg image

---

64 - ./examples/str-str.js

```
    // 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().


/**
 * @param {string} haystack
 * @param {string} needle
 * @return {number}
 */
var strStr = function (haystack, needle) {
  let result = 0;
  if (needle === "") {
    return result;
  }
  let index = haystack.indexOf(needle);
  return index >= -1 ? index : result;
};

/** 1) Cheating */
function strStr1(haystack, needle) {
  return haystack.indexOf(needle);
}

/** 2) Brute force */
function strStr2(haystack, needle) {
  for (let i = 0; i < haystack.length - needle.length + 1; i++) {
    if (haystack.substr(i, needle.length) === needle) return i;
  }

  return -1;
}
```

Flowchart

./examples/str-str.js-svg image

---

65 - ./examples/sub-sets.js

```
    /**
Given a set of distinct integers, nums, return all possible subsets (the
power set).

Note: The solution set must not contain duplicate subsets.

Example:
```

Input: nums = [1,2,3] Output: [[3], [1], [2], [1,2,3], [1,3], [2,3], [1,2], []]

```
 */
const subsets = function (nums) {
    const res = [
        []
    ];
    if (nums.length === 0 || nums === null) {
        return res;
    }
    const len = nums.length;
    function helper(i, arr = []) {
        for (; i < len; i++) {
            let clone = arr.slice(0);
            clone.push(nums[i]);
            res.push(clone);
            if (i + 1 < len) {
                helper(i + 1, clone);
            }
        }
    }
    helper(0);
    return res;
}

console.log(subsets([1, 2, 3]));
console.log(subsets([1, 2]));




/**
 * @param {string} digits
 * @return {string[]}
 */
var letterCombinations = function (digits) {
    let hashTable = {
        2: 'abc',
        3: 'def',
        4: 'ghi',
        5: 'jkl',
        6: 'mno',
        7: 'pqrs',
        8: 'tuv',
        9: 'wxyz'
    };
    let output = [];

    function helper(digit, n) {
        if (digit === n) {
            return;
        }
        for (let i = 0; i < hashTable[digit].length; i++) {
            let curr = hashTable[digit][i];
            console.log(curr);
            output.push(hashTable[digit][i])
        }
    }
    digits.split('').forEach(d => {
        console.log(helper(d));
    })
    return output;
};
```

Flowchart

./examples/sub-sets.js-svg image

---

66 - ./examples/three-sum.js

```
    /**
 * @param {number[]} nums
 * @return {number[][]}
 */
var threeSum = function (nums) {
    nums.sort((a, b) => a - b);

    let results = [];
    let totalSum = 0;
    let i = 0
    while (i < nums.length - 2) {
        let j = i + 1;
        let k = nums.length - 1;
        while (j < k) {
            const sum = nums[i] + nums[j] + nums[k]
            if (sum < totalSum) {
                while (nums[++j] === nums[j - 1]);
            } else if (sum > totalSum) {
                while (nums[--k] === nums[k + 1]);
            } else {
                results.push([nums[i], nums[j], nums[k]])
                while (nums[++j] === nums[j - 1]);
                while (nums[--k] === nums[k + 1]);
            }
        }
        while (nums[++i] === nums[i - 1]);
    }
    return results;
};
console.log(threeSum([-1, 0, 1, 2, -1, -4]));
```

Flowchart

./examples/three-sum.js-svg image

---

67 - ./examples/tree-populate-next-pointer.js

```
    /*
## Populating Next Right Pointers in Each Node
Given a binary tree
```

```
```

struct TreeLinkNode { TreeLinkNode *left; TreeLinkNode *right; TreeLinkNode *next; }

```


Populate each next pointer to point to its next right node. If there is no
next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

Note:

- You may only use constant extra space.
- Recursive approach is fine, implicit stack space does not count as extra
space for this problem.
- You may assume that it is a perfect binary tree (ie, all leaves are at
the same level, and every parent has two children).

Example:

Given the following perfect binary tree,


```

```
 1
```

/
2 3 / \ /
4 5 6 7

```


After calling your function, the tree should look like:


```

```
 1 -> NULL
```

/
2 -> 3 -> NULL / \ /
4->5->6->7 -> NULL

```

*/
/**
 * Definition for binary tree with next pointer.
 * function TreeLinkNode(val) {
```

```
 *     this.val = val;
 *     this.left = this.right = this.next = null;
 * }
 */

/**
 * @param {TreeLinkNode} root
 * @return {void} Do not return anything, modify tree in-place instead.
 */

var connect = function(root) {
    if(!root) return;
    const queue = [root];

    while(queue.length) {
        const size  = queue.length;
        const level = queue.slice();

        for(let i = 0; i < size; i++) {
            const currentNode = queue.shift();
            currentNode.next  = level[i + 1];
            if(currentNode.left)  queue.push(currentNode.left);
            if(currentNode.right) queue.push(currentNode.right);
        }
    }
};
```

Flowchart

./examples/tree-populate-next-pointer.js-svg image

---

68 - ./examples/tree-preorder-traversal.js

```
    function TreeNode(val) {
    this.val = val;
    this.left = this.right = null;
}
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var preorderTraversal = function (root) {
    var result = [];
    traversal(root);

    function traversal(root) {
        if (root === null) {
            return result;
        } else {
            result.push(root.val);
            traversal(root.left);
            traversal(root.right);
        }
    };
    return result;
};

var postorderTraversal = function (root) {
    var result = [];
    traversal(root);

    function traversal(root) {
        if (root === null) {
            return result;
        } else {
            traversal(root.left);
            traversal(root.right);
            result.push(root.val);
        }
    };
    return result;
};

let root = new TreeNode(100);
root.left = new TreeNode(5);
root.right = new TreeNode(7);
root.right.left = new TreeNode(3);
root.right.right = new TreeNode(6);
root.left.left = new TreeNode(2);
root.left.right = new TreeNode(4);
root.left.left.left = new TreeNode(8);

//const {BinarySearchTree} = require('../src/index');


var maxDepth = function (root) {
    if (root === null) {
        return 0;
    }
    let left_depth = maxDepth(root.left);
    let right_depth = maxDepth(root.right);
    return Math.max(left_depth, right_depth) + 1;
};

//console.log(preorderTraversal(root));
//console.log(postorderTraversal(root));
//console.log(maxDepth(root));
```

```
/**
 * @param {number} x
 * @return {number}
 */
var reverse = function (x) {
    let out = [];
    let parts = String(x).split('');
    let j = parts.length - 1;
    let i = 0;
    let isNegative = true;
    for (; i < parts.length; i++) {
        if (parts[i] !== '-') {
            out[j] = parts[i];
            isNegative = false;
        }
        j--;
    }


    console.log(parts, out.join(''));

    return Number(out.join(''));
};
/**
## Reverse Integer
Given a 32-bit signed integer, reverse digits of an integer.

#### Example 1:

Input: 123
Output: 321

#### Example 2:

Input: -123
Output: -321

#### Example 3:

Input: 120
Output: 21
 */
var reverse = function (x) {
    const reversedInt = +String(Math.abs(x)).split('').reverse().join('');
    return reversedInt < 0x7FFFFFFF ? Math.sign(x) * reversedInt : 0;
};
console.log(reverse(123));
console.log(reverse(-123));
//console.assert(reverse(123) === 321);


/*
## First Unique Character in a String
```

```
Given a string, find the first non-repeating character in it and return
it's index.
If it doesn't exist, return -1.

Examples:

s = "leetcode"
return 0.

s = "loveleetcode",
return 2.

> Note: You may assume the string contain only lowercase letters.
*/
/**
 * @param {string} s
 * @return {number}
 */
var firstUniqChar = function (s) {
    var map = {};
    for (var i = 0; i < s.length; i++) {
        let key = s[i];
        if (map[key] === undefined) { // if letter hasn't been encountered
            // set char as key and value ([index, count]) as tuple of the
index and letter count
            map[key] = [i, 1];
        } else {
            // increment letter count
            map[key][1] += 1;
            console.log(key, i)
        }
    }

    for (var k in map) {
        console.log(k, map[k]);
        // if a character count is equal to 1 it is the first unique, so
return
        if (map[k][1] === 1) return map[k][0];
    }
    return -1; // return -1 if we didnt find a unique in our map
};
console.log(firstUniqChar('abc'));
console.log(firstUniqChar('aba'));
console.log(firstUniqChar('loveleetcode'));
```

Flowchart

./examples/tree-preorder-traversal.js-svg image

---

69 - ./examples/two-sum.js

```
    /**

## Two Sum

Given an array of integers, return indices of the two numbers such that
they add up to a specific target.

You may assume that each input would have exactly one solution, and you
may not use the same element twice.

Example:

Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].

 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */

//
/**
 * @param {number[]} numbers
 * @param {number} target
 * @return {number[]}
 */
var twoSum = function (numbers, target) {
    let dict = {};
    for (i in numbers) {
        if (target - numbers[i] in dict) {
            return [dict[target - numbers[i]] + 1, Number(i) + 1];
        }
        dict[numbers[i]] = Number(i);
    }
};

function twoSum1(nums, target) {
    for (let i = 0; i < nums.length; i++) {
        for (let j = 0; j < nums.length; j++) {
            if (i === j) continue;

            if (nums[i] + nums[j] === target) return [i, j];
        }
    }
}

function twoSum(nums, target) {
    let map = {};

    for (let i = 0; i < nums.length; i++) {
        const diff = target - nums[i];
        if (map[diff] !== undefined) return [map[diff], i];
        map[nums[i]] = i;
    }
}
console.log(twoSum([2, 7, 11, 15], 9));
```

Flowchart

./examples/two-sum.js-svg image

---

70 - ./examples/valid-params.js

```
    /**
 * @param {string} s
 * @return {boolean}
 */
function isValid(s) {
    let valid = false;
    let c;
    let dict = {
      '{': '}',
      '(': ')',
      '[': ']'
    };
    let stack = [];
    for (let i = 0; i < s.length; i++) {
      c = s[i];
      if (dict[c]) {
        stack.push(c);
      } else {
        if (dict[stack.pop()] !== c) {
          return false;
        }
      }
    }
    return (stack.length === 0);
  };
```

Flowchart

./examples/valid-params.js-svg image
