Data Structures, Algorithms, and Functions

In JavaScript

Last updated 2021-06-11
Version 2021

Table of Content

PART 1 - Language

Functions

Functions in JS can be written in multiple ways. Following are the ways you can write the same function that adds two numbers. There are more, but these cover the most common use cases.

```
function add(a, b) {
   return a + b
}
```

```
const add = (a, b) => a + b
```

```
const add = a => b => a + b
```

```
const add = function(a, b) {
   return a + b
}
```

```
const add = function add(a, b) {
   return a + b
}
```

Style #1

```
function add(a, b) {
   return a + b
}
```

- Keyword “function” marks this block of code as a function
- “add” is the name of the function. You will use this name to call it.
- (a, b) defines the input parameter list, or input variables that this function will receive
- A return statement tells you what is being returned.
- “+” is a language defined operator
- “return a + b” means the function add when called with two values will return sum of those two values
- We will call via “add(10, 20)” syntax and it will return 30 as output.

Important: This is the most common way to define a function. This function name is hoisted up. This will be explained later in the hoisting topic.

Style #2

```
const add = (a, b) => a + b
```

- add is now a variable.
- It is called arrow function expression.
    - Sometimes it is also called fat due to “=>” w.r.t to thin arrow “->” which javascript doesn’t have.
- The result is the same as above.
- One thing to note is that now you have to be careful of the position where this function is defined, since it is a const variable, it will not be hoisted up.
- We will call via “add(10, 20)” syntax and it will return 30 as output.

Style #3

```
const add = a => b => a + b
```

- This is the curried version.
    - We will cover currying and partial application later.
- We will call via “add(10)(20)” syntax and it will return 30 as output.
- “add(10)” will return a function.

Style #4

```
const add = function(a, b) {
   return a + b
}
```

- This is known as Anonymous Function Expression
- Has the same effect as function definitions and is called via function syntax.

Style #5

```
const add = function add(a, b) {
   return a + b
}
```

- This is known as Named Function Expression.
- This version is usable in recursion in comparison to AFE.

Loops

Loops go through a range for a variable. Loops can be infinite or finite. We use finite loops more than infinite ones.

For

```
for (let count = 0; count < 10; count++) {
   console.log(count)
}
```

A simple loop that prints or logs numbers from 0 to 9 (10 numbers).

While

```
let count = 0
while(count < 10) {
   console.log(count)
   count++
}
```

This while loop achieves the same result as for loop above.

Do While

```
let count = 0
do {
   console.log(count)
   count++
} while(count < 10);
```

This do-while loop also logs numbers from 0 to 9.

Recursion

```
function log(count = 0) {
   if(count < 0 || count >= 10) {
       return;
   }
   console.log(count);
   log(count+1)
}

log()
```

We define a function named log, which accepts a variable count with a default value of 0. The default value kicks in when we don't pass a value.

"log()" calls the function log, without the value, thus the default value of 0 kicks in and we get numbers from 0 to 9 printed via recursion.

The way recursion works in this example: log() -> log(1) -> log(2) -> … -> log(9) -> log(10)

Execution stops at log(10) as the count is equal to 10, function simply returns.

Numbers and Strings

Numbers

- 0
- -0
- +Infinity
- -Infinity
- NaN
- Integers
- Floating point numbers
- Some examples are: 42, 3.14, 1e3 (1000).
- Can be converted to strings via .toString()
- Real or floating point numbers / decimal numbers can be converted to integers via Math.floor, Math.ceil, Math.round functions.
- 0 and -0 differ in Object.is function.
- Integers can safely be used within range: Number.MAX_SAFE_INTEGER and Number.MIN_SAFE_INTEGER

BigInt

- Has “n” at the end of the normal integer.
- 0n is different from 0.
- Can be used above Number.MAX_SAFE_INTEGER

Strings

- Can be defined via following syntax:
    - Double quotes: “ string “
    - Single quotes: ‘ string ‘
    - Backtick: ` string `
    - String.raw: String.raw` string `
- Can be converted to numbers, if it container number via
    - parseInt
    - parseFloat
- Can be converted to objects or other values if it is a stringified json string via
    - JSON.parse(string)
- Strings are immutable in javascript

Part 2 - Problem Solving

Sum of Pair

#1

Problem statement
Find two numbers in an array such that their sum equals the given sum.

```
function pairof2(array, given) {
   for (let i = 0; i < array.length; i++) {
       for (let j = 0; j < array.length; j++) {
           if (i === j) continue
           if (array[i] + array[j] === given) return {first: i, second:
j}
       }
   }
   return {first: -1, second: -1}
}
```

How does this code work?

#2

Problem statement
Find three numbers in an array such that their sum equals the given sum

```
function pairof3(array, given) {
   for (let i = 0; i < array.length; i++) {
       if (array[i] >= given) continue
       for (let j = 0; j < array.length; j++) {
           if (i === j || array[i] + array[j] >= given) continue
           for (let k = 0; k < array.length; k++) {
               if (i === k || j === k) continue
               if (array[i] + array[j] + array[k] === given) return
{first: i, second: j, third: k}
           }
       }
   }
   return {first: -1, second: -1, third: -1}
}
```

These can be improved upon by observing array access and reducing scope.

Local Peak Finding

#1

Problem statement
Find a peak in an array of numbers. A peak is a number that is greater than or equal to its neighbours. Elements at the array boundaries cannot be peaks.

```
function peakfinder(array) {
   for(let i = 0; i < array.length; i++) {
       if((array[i] >= array[i-1]) && (array[i] >= array[i+1])) {
           return [i, array[i]]
       }
   }
   return [-1, null]
}
```

```
const r1 = randomArray()
console.log(r1, peakfinder(r1))
```

```
[13, 1, 18, 11, 6, 17, 23, 11, 22, 1]
[2, 18]
```

#2

Problem Statement

Find a peak in an array of numbers. A peak is a number that is greater than or equal to its neighbours. Elements at the array boundaries **can** be peaks.

```
function peakfinder(array) {
   for(let i = 0; i < array.length; i++) {
       const left = array[i-1] || -Infinity
       const right = array[i+1] || -Infinity
       const current = array[i]
       if((current >= left) && (current >= right)) {
           return [i, current]
       }
   }
   return [-1, null]
}
```

```
const r1 = randomArray()
console.log(r1, peakfinder(r1))
```

```
[5, 1, 6, 21, 16, 8, 13, 18, 15, 12]
[0, 5]
```

#3

Problem Statement

Find all peaks in an array of numbers. A peak is a number that is greater than or equal to its neighbours. Elements at the array boundaries cannot be peaks.

```
function peakfinder(array) {
   const peaks = []
   for(let i = 0; i < array.length; i++) {
       const left = array[i-1]
       const right = array[i+1]
       const current = array[i]
       if((current >= left) && (current >= right)) {
           peaks.push([i, current])
       }
   }
   return peaks
}
```

```
const r1 = randomArray()
console.log(r1, peakfinder(r1))
```

```
[2, 10, 17, 0, 16, 17, 3, 11, 21, 4]
[[2, 17], [5, 17], [8, 21]]

[14, 7, 24, 13, 0, 18, 22, 19, 9, 24]
[[2,24], [6,22]]
```

#4

Problem Statement

Find all peaks in an array of numbers. A peak is a number that is greater than or equal to its neighbours. Elements at the array boundaries **can** be peaks.

```
function peakfinder(array) {
   const peaks = []
   for(let i = 0; i < array.length; i++) {
       const left = array[i-1] || -Infinity
       const right = array[i+1] || -Infinity
       const current = array[i]
       if((current >= left) && (current >= right)) {
           peaks.push([i, current])
       }
   }
   return peaks
}
```

```
const r1 = randomArray()
console.log(r1, peakfinder(r1))
```

```
[2, 10, 17, 0, 16, 17, 3, 11, 21, 4]
[[2, 17], [5, 17], [8, 21]]

[19, 19, 17, 5, 17, 5, 15, 15, 3, 19]
[[0,19],[1,19],[4,17],[6,15],[7,15],[9,19]]
```

#5

Problem Statement

Find **a** peak in an array of numbers. A peak is a number that is greater than or equal to its neighbours. Elements at the array boundaries cannot be peaks. Do it recursively.

```
function recursivepeakfinder(array, i = 0) {
   if(i < 0 || i > array.length) {
       return
   }
   const current = array[i]
   const left = array[i-1]
   const right = array[i+1]
   if(current >= left && current >= right) {
       return [i, array[i]]
   }
   return recursivepeakfinder(array, i + 1)
}
```

```
const r1 = randomArray()
console.log(r1, JSON.stringify(recursivepeakfinder(r1)))
```

```
[17, 11, 21, 4, 16, 24, 2, 5, 19, 19] "[2,21]"
[3, 5, 24, 21, 4, 9, 14, 2, 23, 8] "[2,24]"
[10, 5, 4, 9, 13, 5, 10, 12, 7, 15] "[4,13]"
[10, 10, 9, 5, 20, 0, 22, 12, 1, 24] "[1,10]"
[15, 23, 1, 22, 11, 13, 8, 23, 0, 10] "[1,23]
```

#6

Problem Statement

Find **a** peak in an array of numbers. A peak is a number that is greater than or equal to its neighbours. Elements at the array boundaries **can** be peaks. Do it recursively.

```
function recursivepeakfinder(array, i = 0) {
   if(i < 0 || i > array.length) {
       return
   }
   const current = array[i]
   const left = array[i-1] || -Infinity
   const right = array[i+1] || -Infinity
   if(current >= left && current >= right) {
       return [i, array[i]]
   }
   return recursivepeakfinder(array, i + 1)
}
```

```
const r1 = randomArray()
console.log(r1, peakfinder(r1))
```

```
[11, 15, 6, 11, 22, 15, 11, 0, 23, 19] "[1,15]"
[9, 6, 17, 4, 9, 12, 19, 23, 20, 3] "[0,9]"
```

#7

Problem Statement

Find **all** peaks in an array of numbers. A peak is a number that is greater than or equal to its neighbours. Elements at the array boundaries cannot be peaks. Do it recursively.

```
function recursivepeakfinder(array, peaks = [], i = 0) {
   if(i < 0 || i > array.length) {
       return
   }
   const current = array[i]
   const left = array[i-1]
   const right = array[i+1]
   if(current >= left && current >= right) {
       peaks.push([i, array[i]])
   }
   recursivepeakfinder(array, peaks, i + 1)
   return peaks
}
```

```
const r1 = randomArray()
console.log(r1, JSON.stringify(recursivepeakfinder(r1)))
```

```
[ 22, 10, 10, 0, 22, 14, 2, 13, 6, 2 ]
[[2,10],[4,22],[7,13]]
```

#8

Problem Statement

Find **all** peaks in an array of numbers. A peak is a number that is greater than or equal to its neighbours. Elements at the array boundaries **can** be peaks. Do it recursively.

```
function recursivepeakfinder(array, peaks = [], i = 0) {
   if(i < 0 || i > array.length) {
       return
   }
   const current = array[i]
   const left = array[i-1] || -Infinity
   const right = array[i+1] || -Infinity
   if(current >= left && current >= right) {
       peaks.push([i, array[i]])
   }
   recursivepeakfinder(array, peaks, i + 1)
   return peaks
}
```

```
const r1 = randomArray()
console.log(r1, peakfinder(r1))
```

```
[ 9, 4, 11, 2, 14, 20, 5, 10, 16, 6 ]
[[0,9],[2,11],[5,20],[8,16]]
```

Binary Search

#1

Problem Statement
Given a sorted array of non-decreasing numbers. Find the given number. Use loops. Return [-1, null] if not found.

```
function binarysearch(array, given) {
   let low = 0
   let high = array.length
   let mid = Math.floor((low + high)/2)
   while(low<high) {
       mid = Math.floor((low+high)/2)
       if(array[mid] === given) {
           return [mid, given]
       } else if(array[mid] > given) {
           high = mid
       } else {
           low = mid + 1
       }
   }
   return [-1, null]
}
```

```
const s1 = sortedArray(10)
console.log(binarysearch(s1, 31))
console.log(binarysearch(s1, 2))
```

```
[-1, null]
[1, 2]
```

#2

Problem Statement
Given a sorted array of **non-increasing** numbers. Find the given number. Use loops. Return [-1, null] if not found.

```
function binarysearch(array, given) {
   let low = 0
   let high = array.length
   let mid = null
   while(low<high) {
       mid = Math.floor((low+high)/2)
       if(array[mid] === given) {
           return [mid, given]
       } else if(array[mid] > given) {
           low = mid + 1
       } else {
           high = mid
       }
   }
   return [-1, null]
}
```

```
const rs1 = reverseSortedArray(10)
console.log(binarysearch(rs1, 31))
console.log(binarysearch(rs1, 2))
```

```
[-1, null]
[8, 2]
```

#3

Problem Statement

Given a sorted array of non-decreasing numbers. Find the given number. Use loops. Return [-1, null] if not found. Do it recursively.

```
function recursivebinarysearch(array, given, low = 0, high = Infinity) {
   if(low >= high) {
       return [-1, null]
   }
   high = Math.min(array.length, high)
   const mid = Math.floor((low+high)/2)
   const current = array[mid]
   if(current === given) {
       return [mid, given]
   }
   if(current > given) {
       return recursivebinarysearch(array, given, low, mid)
   } else {
       return recursivebinarysearch(array, given, mid + 1, high)
   }
}
```

```
const s1 = sortedArray(10)
console.log(binarysearch(s1, 31))
console.log(binarysearch(s1, 2))
```

```
[-1, null]
[1, 2]
```

#4

Problem Statement
Given a sorted array of **non-increasing** numbers. Find the given number. Use loops. Return [-1, null] if not found. Do it recursively.

```
function recursivebinarysearch(array, given, low = 0, high = Infinity) {
   if(low >= high) {
       return [-1, null]
   }
   high = Math.min(array.length, high)
   const mid = Math.floor((low+high)/2)
   const current = array[mid]
   if(current === given) {
       return [mid, given]
   }
   if(current > given) {
       return recursivebinarysearch(array, given, mid+1, high)
   } else {
       return recursivebinarysearch(array, given, low, mid)
   }
}
```

```
const rs1 = reverseSortedArray(10)
console.log(binarysearch(rs1, 31))
console.log(binarysearch(rs1, 2))
```

```
[-1, null]
[8, 2]
```

Local Valley Finding

This is similar to Local Peak Finding, but for lower value than it’s neighbours.

Sorting

Bubble Sort

Theory

Compare elements to their neighbours and bubble up the bigger one of two, towards the end. You can choose far/right end for ascending and near/left end for descending sort.

#1

Problem statement
Ascending sort an array of numbers using the bubble sort algorithm.

```
function bubblesort1(array) {
   let total = 0, swaps = 0
   for(let i = 0; i < array.length; i++) {
       for(let j = 0; j < array.length - 1; j++) {
           total++
           if(array[j] > array[j+1]) {
               swaps++
               const tmp = array[j]
               array[j] = array[j+1]
               array[j+1] = tmp
           }
       }
   }
   console.log(total, swaps)
   return array
}
```

```
const ra = randomArray()
console.log(ra)
console.log(bubblesort(ra))
```

```
[10, 12, 4, 3, 22, 8, 14, 14, 4, 24]
90 16
[3, 4, 4, 8, 10, 12, 14, 14, 22, 24]
```

Notice that total is 90; and swaps is 16. This means that if the condition check ran 90 times, and swaps happened 16 times.

#2

Problem statement
Sort as above, but reduce the number checks/operations performed.

```
function bubblesort(array) {
   let total = 0, swaps = 0
   let swapped = true
   for(let i = 0; i < array.length && swapped; i++) {
       swapped = false
       for(let j = 0; j < array.length - 1; j++) {
           total++
           if(array[j] > array[j+1]) {
               swaps++
               swapped = true
               const tmp = array[j]
               array[j] = array[j+1]
               array[j+1] = tmp
           }
       }
   }
   console.log(total, swaps)
   return array
}
```

```
const ra = randomArray()
console.log(ra)
console.log(bubblesort(ra))
```

```
[10, 12, 4, 3, 22, 8, 14, 14, 4, 24]
63 16
[3, 4, 4, 8, 10, 12, 14, 14, 22, 24]
```

Notice that total has dropped to 63 from 90.
For [24, 6, 0, 2, 5, 15, 24, 10, 15, 19] total drops from 90 to 36.

Selection sort

Theory

Select the minimum element, and put it in its place. Repeat for the rest of the array.

Alternatively, you can select the maximum element and put it in its place.

#1

Problem statement
Selection sort a random array.

```
function selectionsort(array) {
   for(let i = 0; i < array.length; i++) {
       let minindex = i
       for(let j = i + 1; j < array.length; j++) {
           if(array[minindex] > array[j]) {
               minindex = j
           }
       }
       const temp = array[minindex]
       array[minindex] = array[i]
       array[i] = temp
   }
   return array
}
```

```
const rsa = reverseSortedArray()
console.log(rsa)
console.log(selectionsort([...rsa]))
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
const ra = randomArray()
console.log(ra)
console.log(selectionsort([...ra]))
```

```
[17, 14, 4, 14, 12, 11, 22, 12, 14, 15]
[4, 11, 12, 12, 14, 14, 14, 15, 17, 22]
```

Add total and swaps to see the comparisons.

#2

Problem statement
Descending order

```
function selectionsort(array) {
   let total = 0
   let swaps = 0
   const lastIndex = array.length - 1
   for(let i  = 0; i <= lastIndex; i++ ) {
       let minIndex = 0
       for(let j = 0; j <= lastIndex - i; j++) {
           total++
           if(array[minIndex] > array[j]) {
               swaps++
               minIndex = j
           }
       }
       const temp = array[minIndex]
       array[minIndex] = array[lastIndex - i]
       array[lastIndex - i] = temp
   }
   console.log(`total: ${total}; swaps: ${swaps}`)
   return array
}
```

```
const rsa = reverseSortedArray()
console.log(rsa)
console.log(selectionsort([...rsa]))
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
total: 55; swaps: 45
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
const ra = randomArray()
console.log(ra)
console.log(selectionsort([...ra]))
```

```
[18, 15, 22, 21, 3, 11, 17, 21, 18, 23]
total: 55; swaps: 9
[23, 22, 21, 21, 18, 18, 17, 15, 11, 3]
```

```
const sa = sortedArray()
console.log(sa)
console.log(selectionsort([...sa]))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
total: 55; swaps: 20
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Insertion sort

Theory

Pick one element; put it in its place relative to the already sorted array.

#1

Problem statement
Ascending sort an array using insertion sort algorithm.

```
function insertionsort(array) {
   let total = 0
   let swaps = 0
   for(let i = 1; i < array.length; i++) {
       for(let j = i - 1; j >= 0; j--) {
           total++
           if(array[j] > array[j+1]) {
               swaps++
               const temp = array[j]
               array[j] = array[j+1]
               array[j+1] = temp
           }
       }
   }
   console.log(`total: ${total}; swaps: ${swaps}`)
   return array
}
```

```
const rsa = reverseSortedArray()
console.log(rsa)
console.log(insertionsort([...rsa]))
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
total: 45; swaps: 45
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
const ra = randomArray()
console.log(ra)
console.log(insertionsort([...ra]))
```

```
[17, 11, 7, 0, 10, 15, 22, 16, 18, 24]
total: 45; swaps: 12
[0, 7, 10, 11, 15, 16, 17, 18, 22, 24]
```

```
const sa = sortedArray()
```

```
console.log(sa)
console.log(insertionsort([...sa]))

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
total: 45; swaps: 0
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Notice how for the sorted array, it swapped 0 times!

Merge sort

Theory

Break up arrays; compare two elements, and start putting back the sorted elements.

#1

```
function mergesort(array) {
   if(array.length <= 1) {
       return array
   }
   if(array.length === 2) {
       return array[0] < array[1] ? array : array.reverse()
   }
   let mid = Math.floor((0 + array.length)/2)
   const left = mergesort([...array.slice(0, mid)])
   const right = mergesort([...array.slice(mid)])
   return merge(left, right)
}

function merge(left, right) {
   if(left.length < 1 || right.length < 1) {
       return [...left, ...right]
   }
   if(left[left.length - 1] < right[0]) {
       return [...left, ...right]
   }
   let out = []
   let i = 0, j = 0
   while(i < left.length && j < right.length && out.length !==
left.length + right.length) {
       if(left[i] < right[j]) {
           out.push(left[i])
           i++
       }
       if(left[i] === right[j]) {
           out.push(left[i])
           out.push(right[j])
           i++
           j++
       }
       if(right[j] < left[i]) {
```

```
            out.push(right[j])
            j++
        }
    }

    if(i < left.length) {
        out = out.concat(left.slice(i))
    }
    if(j < right.length) {
        out = out.concat(right.slice(j))
    }

    return out
}
```

Array Partition 1

Theory

Pick the last element. Go through the array, move smaller or same elements (by value) towards front.

#1

Problem Statement
Partition an array such that the smaller or equal numbers to the last element come to front.
Return the partitioning index.

```
function sortForPartition(array, low = 0, high = Infinity) {
   high = Math.min(array.length -1, high)
   const ipi = high
   const pivot = array[ipi]
   let pi = low
   for(let ci = low; ci < high; ci++) {
       if(array[ci] <= pivot) {
           swap(array, ci, pi)
           pi++
       }
   }
   swap(array, pi, ipi)
   return {pi, array}
}
```

```
const rsa = reverseSortedArray()
console.log(rsa)
console.log(sortForPartition([...rsa]))

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[0, [1, 9, 8, 7, 6, 5, 4, 3, 2, 10]]


const ra = randomArray()
console.log(ra)
console.log(sortForPartition([...ra]))

[2, 10, 13, 4, 15, 22, 2, 21, 19, 10]
[4, [2, 10, 4, 2, 10, 22, 13, 21, 19, 15]

const sa = sortedArray()
console.log(sa)
```

```
console.log(sortForPartition([...sa]))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[9, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
```

Quicksort

Theory

Break the array in parts such that first has smaller numbers and second one has bigger numbers than a pivot. Recursively implement partition.

#1

```
function quicksort(array, low = 0, high = Infinity) {
   high = Math.min(array.length, high)
   if(low < high) {
       const {pi} = sortForPartition(array, low, high)
       quicksort(array, low, pi-1)
       quicksort(array, pi+1, high)
   }
   return array
}
```

```
const rsa = reverseSortedArray()
console.log(rsa)
console.log(quicksort([...rsa]))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

const ra = randomArray()
console.log(ra)
console.log(quicksort([...ra]))
[10, 16, 12, 16, 4, 13, 23, 18, 9, 1]
[1, 4, 9, 10, 12, 13, 16, 16, 18, 23]

const sa = sortedArray()
console.log(sa)
console.log(quicksort([...sa]))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Use sortForPartition from #1RC1 above.

The Partitioner

Does this make sense to you?

```
function thePartitioner(array) {
   if (array.length <= 1) {
       return array
   }
   if (array.length === 2) {
       return array[0] <= array[1] ? array : array.reverse()
   }
   const x = array[Math.floor(Math.random() * array.length)]
   const smaller = thePartitioner(array.filter(i => i < x))

   const same = array.filter(i => i === x)
   const rest = thePartitioner(array.filter(i => i > x))

   array = [...smaller, ...same, ...rest]
   return array
}
```

```
const rsa = reverseSortedArray(100000)
console.log(rsa)
console.log(thePartitioner([...rsa]))

const ra = randomArray(100000, Number.MAX_SAFE_INTEGER)
console.log(ra)
console.log(thePartitioner([...ra]))

const sa = sortedArray(100000)
console.log(sa)
console.log(thePartitioner([...sa]))
```

Numbers

```
function reverseANumber(n) {
   return parseInt(n.toString().split('').reverse().join(''), 10)
}
```

```
function reverseANumberExtended(n) {
       const nstr = n.toString()
       const narr = nstr.split('').reverse()
       let leadingZeros = 0
       for (let i = 0; i < narr.length; i++) {
           if (narr[i] === '0') {
               leadingZeros++
           } else {
               break
           }
       }
       return {number: parseInt(narr.join(''), 10), leadingZeros}
}
```

Arrays

```
function isArray(candidate) {
   return Object.prototype.toString.call(candidate) === '[object Array]'
}
```

```
const max = _ => Math.max.apply(null, _)
```

```
const min = _ => Math.min.apply(null, _)
```

```
const range = (min, max, step) =>
   Array(max - min + 1)
       .fill(0)
       .map((_, i) => i + min)
       .filter((_, i) => i % (Math.abs(step)) === 0)
```

```
const randomArray = (n = 10, below = 25) =>
    Array(n).fill(0).map(_ => Math.floor(Math.random() * below))
```

randomArray is a function that takes two params n and below, and returns an array of n numbers that are lower in value than below.

```
const sortedArray = (n = 10) =>
    Array(n).fill(0).map((_, i) => i + 1)
```

sortedArray is a function that takes one parameter n, and returns an array of numbers from 1 to n.

```
const reverseSortedArray = (n = 10) =>
    sortedArray(n).reverse()
```

reverseSortedArray is a function that takes one parameter n, and returns an array of numbers from n to 1.

```
function swap(array, i, j) {
   const temp = array[i]
   array[i] = array[j]
   array[j] = temp
}
```

swap is a function that takes three parameters, array, i, and j, and doesn’t return anything. This function however, switches the values at places i and j.

```
const arrayToObject = array => array.reduce((a, c) => Object.assign(a, {[c]: (a[c] || 0) + 1}), {})
```

```
function conditionalSum(fn, array) {
   return array.reduce((a, c) => a += fn(c) ? c : 0, 0)
}
console.log(conditionalSum(i => i % 2 === 0, sortedArray())) // 30
```

conditionalSum function takes two parameters, fn which is a function that accepts one parameter, and array which contains the numbers.

```
function scan(array, action, callback, initialValue) {
   array.reduce((a, c) => {
       a = action(a, c)
       callback(a)
       return a
    }, initialValue)
   return array
}

Generating factorial of n:
const productAction = (a, b) => a * b
const callback = x => console.log(x)
const initialValue = 1
scan(sortedArray(), productAction, callback, initialValue)

Generating sum of first n numbers:
const sumAction = (a, b) => a + b
const initialSumValue = 0
scan(sortedArray(), sumAction, console.log, initialSumValue)
```

scan function does what reduce does, but it also returns current accumulator value.

```
function repeat(ntimes, fn) {
   if(!ntimes || ntimes < 1) {
       return
   }
   for(let i = 0; i < ntimes; i++) {
       fn(i)
   }
}

repeat(10, x => console.log(`x is ${x}`))
```

repeat function takes two parameters ntimes and fn, it calls fn ntimes times and with the values from 0 to ntimes-1.

```
function repeatUntil(n, check, fn) {
   if(!check || !fn) {
       return
   }
   for(let i = 0; i < n && check(i); i++){
       fn(i)
   }
}
repeatUntil(100, x => x < 10, console.log)
```

repeatUntil will call fn with values from 0 to n; until the check(i) is true. As soon as check(i) returns false, it won't call fn further.

```
function whenTrue(value, truthFn, actionFn) {
   if(truthFn(value)) actionFn(value)
}
```

Printing numbers from 0 to 96, in step of 4:

```
repeat(100, n => {
   whenTrue(n, i => i % 4 === 0, console.log)
})
```

```
function rotateLeft(array, n) {
   return array.map(i => {
       let rotateBy = n % i.length
       return [...i.slice(rotateBy), ...i.slice(0, rotateBy)]
   })
}
```

```
function rotateRight(array, n) {
   return array.map(i => {
       let rotateBy = i.length - (n % i.length)
       return [...i.slice(rotateBy), ...i.slice(0, rotateBy)]
   })
}
```

```
function rotateUp(array, n) {
   const rotateBy = n % array.length
   return [...array.slice(rotateBy), ...array.slice(0, rotateBy)]
}
```

```
function rotateDown(array, n) {
   const rotateBy = array.length - n % array.length
   return [...array.slice(rotateBy), ...array.slice(0, rotateBy)]
}
```

```
function sumScan(array) {
   const out =[]
   array.reduce((a, c) => {
       const sum = a + c
       out.push(sum)
       return sum
   }, 0)
   return out
}
```

```
function actionScan(array, action, initialValue) {
   const out =[]
   array.reduce((a, c) => {
       const x = action(a, c)
       out.push(x)
       return x
   }, initialValue)
   return out
}
const productScan = (array) => actionScan(array, (a, b) => a * b , 1)
const sumScan = array => actionScan(array, (a, b) => a + b, 0)
```

```
function maxSumSubArray(input) {
   for (let i = 1; i < input.length; i++) {
       if (input[i - 1] > 0) {
           input[i] += input[i - 1]
       }
   }
   return Math.max.apply(null, input)
}
```

Sorts

```
const sortByAThenByB_Desc = (mappable, keyA, keyB) => mappable.sort((a, b) => a[keyA] === b[keyA] ? a[keyB] === b[keyB] ? 0 : a[keyB] > b[keyB] ? -1 : 1 : a[keyA] > b[keyA] ? -1 : 1)
```

```
const sortByAThenByB_Asc = (mappable, keyA, keyB) => sortAThenBDesc(mappable, keyA, keyB).reverse()
```

Inplace movements

Moving zeroes towards end

Problem statement
Given an array of numbers containing zeroes and non-zeroes, move all the zeroes to the end (right) while keeping relative positions of other values sorted.

```
function moveZerosToEnd(nums) {
   if(nums.length <= 1) {
       return nums
   }
   if(nums.every(i => i === 0)) {
       return nums
   }
   let zeroAtEndIndex = nums.length

   for (let i = 0; i < zeroAtEndIndex; i++) {
       if (nums[i] === 0) {
           for (let j = i + 1; j < zeroAtEndIndex; j++) {
               nums[j-1] = nums[j]
           }
           zeroAtEndIndex--
           i--
           nums[zeroAtEndIndex] = 0
       }
   }
   return nums
}
```

Remove/Discard given number inplace

```
function removeGiven(nums, given) {
   let ci = 0
   for (let i = 0; i < nums.length; i++) {
       if (nums[i] !== given) {
           nums[ci++] = nums[i]
       }
   }
   return nums.slice(0, ci)
}
```

```
function moveZeroesToEnd(input) {
   if(input.length <= 1) return input

   for(let i = 0, zi = 0; i < input.length; i++) {
       let current = input[i]
       if(current !== 0) {
           input[i] = input[zi]
           input[zi] = current
           zi++
       }
   }
   return input
}
```

Maximum Sub Array

```
function maxsubarray(array, low = 0, high = Infinity) {
   high = Math.min(high, array.length-1)
   const mid = Math.floor((low + high) / 2)
   if (high === low) {
       return {sum: array[low], array: array.slice(low, high + 1)}
   }
   const left = maxsubarray(array, low, mid)
   const right = maxsubarray(array, mid + 1, high)
   const cross = crossmaxsubarray(array, low, high)
   if (left.sum >= right.sum && left.sum >= cross.sum) {
       return left
   } else if (right.sum >= left.sum && right.sum >= cross.sum) {
       return right
   } else {
       return cross
   }
}

function crossmaxsubarray(array, low, high) {
   const mid = Math.floor((low + high) / 2)
   let lsum = 0, rsum = 0
   let li, ri
   let sum = 0
   for (let i = mid; i > low; i--) {
       sum += array[i]
       if (sum >= lsum) {
           lsum = sum
           li = i
       }
   }
   sum = 0
   for (let i = mid + 1; i < high; i++) {
       sum += array[i]
       if (sum >= rsum) {
           rsum = sum
           ri = i
```

```
        }
    }
    return {sum: lsum + rsum, array: array.slice(li, ri + 1)}
}
```

Matrices

```
const emptySquareMatrix = (n = 3) => {
   const matrix = new Array(n)
   for(let i = 0; i < n; i++) {
       matrix[i] = new Array(n)
   }
   return matrix
}
```

```
const squareMatrix = (n = 3, fill = 0) => {
   const matrix = emptySquareMatrix(n)
   for(let i = 0; i < n; i++) {
       for(let j = 0; j < n; j++) {
           matrix[i][j] = fill
       }
   }
   return matrix
}
```

```
const emptyMatrix = (rows = 3, cols = 3) => {
   const matrix = new Array(rows)
   for(let r = 0; r < rows; r++) {
       matrix[r] = new Array(cols)
   }
   return matrix
}
```

```
const emptyMatrix = (rows = 3, cols = 3) => {
   const matrix = new Array(rows)
   for(let r = 0; r < rows; r++) {
       matrix[r] = new Array(cols)
   }
   return matrix
}
```

```
const matrix = (rows = 3, cols = 3, fill = 0) => {
   const m = emptyMatrix(rows, cols)
   for(let r = 0; r < rows; r++) {
       for(let c = 0; c < cols; c++) {
           m[r][c] = fill
       }
   }
   return m
}
```

```
const sortedMatrix = (rows = 3, cols = 3, start = 1) => {
   const m = matrix(rows, cols)
   for(let r = 0; r < rows; r++) {
       for(let c = 0; c < cols; c++) {
           m[r][c] = start + r*cols + c
       }
   }
   return m
}
```

```
const reverseSortedMatrix = (rows = 3, cols = 3, start = 1) =>
   sortedMatrix(rows, cols, start).map(i => i.reverse()).reverse()
```

```
const matrixSum = (matrix = sortedMatrix(), rows = 3, cols = 3) => {
   let sum = 0
   for(let r = 0; r < rows; r++) {
       for(let c = 0; c < cols; c++) {
           sum += matrix[r][c]
       }
   }
   return sum
}
console.log(matrixSum()) // 45
console.log(matrixSum(sortedMatrix(5, 3), 5, 3)) //120
```

```
const maximumSumOfArray = matrix => Math.max.apply(null, matrix.map(i =>
i.reduce((a, c) => a + c, 0)))
```

```
const sum = (a, b) => a + b
const sumAMatrix = _ => _.map(i => i.reduce(sum, 0)).reduce(sum, 0)
```

```
const countNegatives = matrix => matrix.map(i => i.filter(j => j <
0).length).reduce((a, c) => a+c, 0)
```

Linked List

```
const NODE_DATA_NAME = 'value'
const NODE_NEXT_NAME = 'next'

function Node(value, next) {
   this[NODE_DATA_NAME] = (value === undefined ? 0 : value)
   this[NODE_NEXT_NAME] = (next === undefined ? null : next)
}
const NODE = Node
const start = new NODE(10)
```

```
function numberToLinkedList(n){
   const chars = n.toString().split('')
   const START = new NODE(chars[0])
   chars.slice(1).reduce((a, c) => a[NODE_NEXT_NAME] = new NODE(c),
START)
   return START
}
```

```
function stringToLinkedList(s) {
   const chars = s.split('')
   const START = new NODE(chars[0])
   chars.slice(1).reduce((a, c) => a[NODE_NEXT_NAME] = new NODE(c),
START)
   return START
}
```

```
function linkedListToString(START) {
   let str = ''
   let current = START
   while(current) {
       str += current[NODE_DATA_NAME]
       current = current[NODE_NEXT_NAME]
   }
   return str
}
```

```
function linkedListToNumber(START) {
   const str = linkedListToString(START)
   return parseInt(str, 10)
}
```

```
function linkedListToNumber(START) {
   const str = linkedListToString(START)
   return BigInt(str)
}
```

```
function linkedListToArray(START) {
   let out = []
   let current = START
   while(current) {
       out.push(current[NODE_DATA_NAME])
       current = current[NODE_NEXT_NAME]
   }
   return out
}
```

```
function arrayToLinkedList(array) {
   if(!Array.isArray(array) || array.length === 0) {
       return null
   }
   const START = new NODE(array[0])
   array.reduce((a, c) => a[NODE_NEXT_NAME] = new NODE(c), START)
   return START
}
```

Binary Tree

```
function swap(bst) {
   const tempLeft = bst.left
   const tempRight = bst.right
   bst.left = tempRight
   bst.right = tempLeft
}
function invert(bst) {
   if(bst) {
       swap(bst)
       invert(bst.left)
       invert(bst.right)
   }
}
```

Strings

```
function forEveryChar(str, fn) {
   str.split('').forEach(ch => fn(ch))
}

forEveryChar('DataStructuresAlgorithmsAndFunctions', x =>
console.log(x.toLowerCase()))
```

```
const stringToObject = str => arrayToObject(str.split(''))
```

```
const sortStringsByLength = _ => _.sort((a, b) => a.length >= b.length)]
```

```
const sortStringsByLength = _ => _.sort((a, b) => a.length === b.length
? 0 : a.length > b.length ? 1 : -1)
```

```
function tokenizeAtDuplicate(str) {
   const out = []
   let current = ''
   for(let i = 0; i < str.length; i++) {
       if(current.includes(str[i])) {
           out.push(current)
           current = ''
       }
       current += str[i]
   }
   out.push(current)
   return out
}
// dvdf => dv, df
```

```
function tokenizeOnDuplicate(str) {
   const out = []
   let current = ''
   for(let i = 0; i < str.length; i++) {
       if(current.includes(str[i])) {
           out.push(current)
           current = current.slice(current.indexOf(str[i]) + 1)
       }
       current += str[i]
   }
   out.push(current)
   return out
}
// dvdf => dv, vdf
```

```
function atoi(s /* string */) {

   s = s.trim() // no space

   let i = 0
   let multiplier = 1
   let n = 0

   if(s[i] === '-') {
       multiplier = -1
       i++
   } else if(s[i] === '+') {
       i++
   }
   if(s[i] < '0' || s[i] > '9') {
       return 0
   }
   while(i < s.length && s[i] >= '0' && s[i] <= '9') {
       n = n*10 + parseInt(s[i])
       i++
   }
   let result = n * multiplier
   if(result < -1 * Math.pow(2, 31)) {
       result = -1 * Math.pow(2, 31)
   } else if(result >= Math.pow(2, 31)) {
       result = Math.pow(2, 31) - 1
   }
   return result
}
```

```
function strstr(haystack, needle) {
   if (needle === '') {
       return 0
   }
   if (haystack === '') {
       return -1
   }
   if (needle.length > haystack.length) {
       return -1
   }
   let hl = haystack.length
   let nl = needle.length
   for (let i = 0; i < hl; i++) {
       while (haystack[i] !== needle[0] && haystack[i + nl - 1] ===
needle[nl - 1]) {
           i++
       }
       let matches = true
       for (let j = 0; j < nl && matches; j++) {
           matches = matches && haystack[i + j] === needle[j]
       }
       if (matches) { return i }
   }
   return -1
}
```

```
function areBracketsBalanced(s) {
   if (s.length === 0 || s === '') {
       return true
   }
   if (s.length === 1) {
       return false
   }
   const stack = []
   for (let i = 0; i < s.length; i++) {
       if (s[i] === '(') {
           stack.push('(')
       } else if (s[i] === ')') {
           if (stack[stack.length - 1] !== '(') {
               return false
           } else {
               stack.pop()
           }
       } else if (s[i] === '[') {
           stack.push('[')
       } else if (s[i] === ']') {
           if (stack[stack.length - 1] !== '[') {
               return false
           } else {
               stack.pop()
           }
       } else if (s[i] === '{') {
           stack.push('{')
       } else if (s[i] === '}') {
           if (stack[stack.length - 1] !== '{') {
               return false
           } else {
               stack.pop()
           }
       }
   }
   return stack.length === 0
}
```

Substrings

```
function findAllSubstrings(str, all=[]) {
   if(str.length === 0) {
       return
   }
   all.push(str)
   findAllSubstrings(str.slice(0, str.length -1), all)
   findAllSubstrings(str.slice(1, str.length), all)
   return all
}
```

What's wrong with it?

```
function findAllSubstrings(str) {
   if(str.length === 0) {
       return
   }
   const len = str.length
   const out = []
   for(let i = 0; i < len; i++) {
       for(let j = i; j <= len; j++) {
           let sliced = str.slice(i, j)
           if(sliced) out.push(sliced)
       }
   }
   return out
}
```

Numbers

```
function oldfashioneddivide(dividend, divisor) {
   if (dividend < 0 && divisor < 0) {
       return divide(dividend * -1, divisor * -1)
   }
   let multiplier = (dividend < 0 && divisor >= 0) || (dividend >= 0 &&
divisor < 0) ? -1 : 1
   const numr = Math.abs(dividend)
   const denr = Math.abs(divisor)
   if (denr === 1) {
       return Math.min(numr * multiplier, Math.pow(2, 31) - 1)
   }
   let q = 0
   let tmp = numr
   while (tmp >= denr) {
       tmp -= denr
       q++
   }
   return Math.min(q * multiplier, Math.pow(2, 31) - 1)
}
```

```
function findNthDigit(n) {
   let digits = []
   let curr = 1
   while(digits.length < n ) {
       digits = [...digits, ...curr.toString().split('')]
       curr++
   }
   return digits[n-1]
}
```

Part 3 - Applications

Files

```
const {readFileSync} = require('fs')
const data = readFileSync('./file.txt').toString()
const lines = data.split('\n')
```

WC

Problem Statement
Write a script (nodejs) that will print information as wc does.

#1

```
const {readFileSync} = require('fs')
const data = readFileSync('./data.txt').toString()
const lines = data.split('\n')

const numberOfLines = lines.length - 1
const words = data.split(/\s/).length - 1
const characters = data.split('').length

console.log(characters, words, numberOfLines)
```

#2 - executable script

```
#!/usr/bin/env node
const fs = require('fs')
const [node, source, input] = process.argv

const data = fs.readFileSync(input).toString()
const lines = data.split('\n')

const numberOfLines = lines.length - 1
const words = data.split(/\s/).length - 1
const characters = data.split('').length

console.log(numberOfLines, words, characters, input)
```

- Save as wc.js
- chmod +x wc.js
- ./wc.js <filename>

Part 4 - Challenges

Close but Incorrect

These are the implementations that look correct, feel correct, but aren’t. Do you take a challenge to figure out why?

Move zeroes to the end

```
function moveZeroesToEnd(input) {
   if(input.length <= 1) return input
   const joined = input.join('')
   const splitted = joined.split('0')
   const count = splitted.length -1
   const zeroes = Array(count).fill(0)
   const rest = splitted.filter(i => i).join('').split('')

   return [...rest, ...zeroes]
}
```

Find all substrings

```
function findAllSubstrings(str, all=[]) {
   if(str.length === 0) {
       return
   }
   all.push(str)
   findAllSubstrings(str.slice(0, str.length -1), all)
   findAllSubstrings(str.slice(1, str.length), all)
   return all
}
```

Maximum subarray

#1

```
function subArray(array) {
   if (array.length === 1) {
       return {sum: array[0], array}
   }
   const mid = Math.floor(array.length / 2)
   const left = subArray(array.slice(0, mid))
   const right = subArray(array.slice(mid))
   const cross = crossSubArray(array)
   if (left.sum >= right.sum && left.sum >= cross.sum) {
       return left
   } else if (right.sum >= left.sum && right.sum >= cross.sum) {
       return right
   } else {
       return cross
   }
}

function crossSubArray(array) {
   const mid = Math.floor(array.length / 2)
   let left, right
   let leftSum = 0
   let rightSum = 0
   for (let i = mid; i >= 0; i--) {
       let tempSum = leftSum + array[i]
       if (tempSum >= leftSum) {
           left = i
           leftSum += array[i]
       }
   }
   for (let i = mid + 1; i <= array.length; i++) {
       let tempSum = rightSum + array[i]
       if (tempSum >= rightSum) {
           rightSum += array[i]
           right = i
```

```
        }
    }
    const out = array.slice(left, right+1 )
    const sum = out.reduce((a, c) => a + c, 0)
    return {sum, array: out}
}

subArray([-1, -2, 3, -4, 5, 6, 7, -8])
subArray([10, -1, -2, 3, -4, 5, 6, 7, -8])
subArray([5, -1, -2, 3, -4, 5, 6, 7, -8])
subArray([4, -1, -2, 3, -4, 5, 6, 7, -8])
subArray([1, -1, -2, 3, -4, 5, 6, 7, -8])
```

#2

```
function maxsubarray(array, low = 0, high = Infinity) {
   high = Math.min(high, array.length)
   const mid = Math.floor((low + high) / 2)
   if (high === low) {
       return {sum: array[0], array: array.slice(low, high + 1)}
   }
   const left = maxsubarray(array, low, mid)
   const right = maxsubarray(array, mid + 1, high)
   const cross = crossmaxsubarray(array, low, high)
   if (left.sum >= right.sum && left.sum >= cross.sum) {
       return left
   } else if (right.sum >= left.sum && right.sum >= cross.sum) {
       return right
   } else {
       return cross
   }
}

function crossmaxsubarray(array, low, high) {
   const mid = Math.floor((low + high) / 2)
   let lsum = 0, rsum = 0
   let li = mid, ri = mid + 1

   for (let i = mid; i >= low; i--) {
       if (array[i] + lsum >= lsum) {
           lsum += array[i]
           li = i
       }
   }

   for (let i = mid + 1; i < high; i++) {
       if (array[i] + rsum >= rsum) {
           rsum += array[i]
           ri = i
       }
```

```
    }

    return {sum: lsum + rsum, array: array.slice(li, ri + 1)}
}
```