1. Two Sum

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to* `target`.

There is a trade-off between both solutions: the first one is faster since you only use one loop but takes up more memory. The second one uses 2 loops but uses up no extra memory space.

`Time: O(n)` `Space: O(n)`

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
let twoSum = function (nums, target) {
  let map = new Map();
  for (let i = 0; i < nums.length; i++) {
    let num1 = target - nums[i];
    if (map.has(num1)) {
      return [i, map.get(num1)];
    }
    map.set(nums[i], i);
  }
};
```

`Time: O(n^2)` `Space: O(1)`

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

9. Palindrome Number

Given an integer `x`, return `true` *if* `x` is a *palindrome, and* `false` *otherwise.*

Brute solution

`Time: O(n)` `Space: O(n)`

```
/**
 * @param {number} x
 * @return {boolean}
 */
let isPalindrome = function (x) {
  let s = x.toString();
  let r = s.split("").reverse().join("");
  return s === r;
};
```

Optimised solution

`Time: O(log n)` `Space: O(1)`

```
/**
 * @param {number} x
 * @return {boolean}
 */
let isPalindrome = function (x) {
  // if negative or multitude of 10
  if (x < 0 || (x !== 0 && x % 10 === 0)) {
    return false;
  }
  // you build up half and remove the last number of x
  // until half > x
  let half = 0;
  while (x > half) {
    half = half * 10 + (x % 10);
    x = Math.floor(x / 10);
  }
  // if x has an uneven amount of numbers, half will be x exactly
  // if x has an even amount of numbers, hallf/10 and floored will be x
exactly
  return x === half || x === Math.floor(half / 10);
};
```

13. Roman to Integer

Given a roman numeral, convert it to an integer.

Difficulty: You always have to compare the current and the next value.

`Time: O(n)` `Space: O(n)`

```
const map = {
  I: 1,
  V: 5,
  X: 10,
  L: 50,
  C: 100,
  D: 500,
  M: 1000,
};

/**
 * @param {string} s
 * @return {number}
 */
let romanToInt = function (s) {
  let sum = 0;
  for (let i = 0; i < s.length; i++) {
    let current = map[`${s[i]}`];
    let next = map[`${s[i + 1]}`]; // if this does not exists, undefined
    if (next && next > current) {
      sum += next - current;
      i++;
    } else {
      sum += map[`${s[i]}`];
    }
  }
  return sum;
};
```

14. Longest Common Prefix

Write a function to find the longest common prefix string amongst an array of strings. `Time: O(n)` `Space: O(1)`

```
/**
 * @param {string[]} strs
 * @return {string}
 */
let longestCommonPrefix = (strs) => {
  if (!strs.length) return "";
  let prefix = strs[0];
  for (let i = 1; i < strs.length; i++) {
    while (strs[i].indexOf(prefix) !== 0) {
      prefix = prefix.substring(0, prefix.length - 1);
      if (prefix == "") {
        return "";
      }
    }
  }
  return prefix;
};
```

20. Valid Parentheses

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

`Time: O(n)` `Space: O(n)`

```
const openBrackets = ["(", "{", "["];
const closingBrackets = [")", "}", "]"];

/**
 * @param {string} s
 * @return {boolean}
 */
let isValid = function (s) {
  const stack = [];
  const sequence = s.split("");
  for (let i = 0; i < sequence.length; i++) {
    if (openBrackets.includes(sequence[i])) {
      stack.push(sequence[i]);
    }
    if (closingBrackets.includes(sequence[i])) {
      const peek = stack[stack.length - 1];
      if (
        (peek === "(" && sequence[i] === ")") ||
        (peek === "{" && sequence[i] === "}") ||
        (peek === "[" && sequence[i] === "]")
      ) {
        stack.pop();
      } else {
        return false;
      }
    }
  }
  return stack.length === 0;
};
```

234. Palindrome Linked List

Given the `head` of a singly linked list, return `true` *if it is a palindrome or* `false` *otherwise.*

We use a variation on the Floyd's Tortoise and Hare algorithm to go to the middle of the list and build up a reverse from the halfway point on. Then we move both halves and check if the values match.

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @return {boolean}
 */
let isPalindrome = function (head) {
  let slow, fast, prev, temp;
  slow = head;
  fast = head;
  // slow is in the middle of the list, fast is at the end
  while (fast && fast.next) {
    slow = slow.next;
    fast = fast.next.next;
  }
  // building up the reverse
  prev = slow;
  slow = slow.next;
  prev.next = null;
  while (slow) {
    temp = slow.next;
    slow.next = prev;
    prev = slow;
    slow = temp;
  }
  // putting fast at the beginning of the half,
  // slow at the end (beginning of reversed half)
  fast = head;
  slow = prev;
  // check if ever their values are not the same
  while (slow) {
    if (fast.val !== slow.val) return false;
    fast = fast.next;
    slow = slow.next;
  }
  return true;
};
```

383. Ransom Notes

Given two strings `ransomNote` and `magazine`, return `true` *if* `ransomNote` *can be constructed by using the letters from* `magazine` *and* `false` *otherwise.*

Each letter in `magazine` can only be used once in `ransomNote`.

Better for Time

`Time: O(n)` `Space: O(n)`

```
/**
 * @param {string} ransomNote
 * @param {string} magazine
```

```
  * @return {boolean}
  */
let canConstruct = function (ransomNote, magazine) {
  const ransomArray = ransomNote.split("");
  for (let i = 0; i < ransomArray.length; i++) {
    const indexAt = magazine.indexOf(ransomArray[i]);
    if (indexAt == -1) {
      return false;
    }
    magazine =
      magazine.slice(0, indexAt) + magazine.slice(indexAt + 1,
magazine.length);
  }
  return true;
};
```

Better for Space

`Time: O(n+m)` `Space: O(n)

```
/**
 * @param {string} ransomNote
 * @param {string} magazine
 * @return {boolean}
 */
let canConstruct = function (ransomNote, magazine) {
  let map = new Map();
  for (let n of magazine) {
    if (map.has(n)) {
      map.set(n, map.get(n) + 1);
    } else {
      map.set(n, 1);
    }
  }
  for (let m of ransomNote) {
    if (map.get(m)) {
      map.set(m, map.get(m) - 1);
    } else {
      return false;
    }
  }
  return true;
};
```

2235. Add Two Integers

Given two integers `num1` and `num2`, return the sum of the two integers

`Time: O(1)` `Space: O(1)`

```
let sum = (num1, num2) => num1 + num2;
```

2236. Root Equals Sum of Children

You are given the `root` of a **binary tree** that consists of exactly `3` nodes: the root, its left child, and its right child.

Return `true` *if the value of the root is equal to the sum of the values of its two children, or* `false` *otherwise*.

`Time: O(1)` `Space: O(1)`

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {boolean}
 */
let checkTree = function (root) {
  return root.val == root.left.val + root.right.val;
};
```

2619. Array Prootot ype Last

Write code that enhances all arrays such that you can call the `array.last()` method on any array and it will return the last element. If there are no elements in the array, it should return `-1`.

`Time: O(1)` `Space: O(1)`

```
Array.prototype.last = function () {
  return this.length ? this.at(this.length - 1) : -1;
};
```

2620. Counter

Given an integer `n`, return a `counter` function. This `counter` function initially returns `n` and then returns 1 more than the previous value every subsequent time it is called (`n`, `n + 1`, `n + 2`, etc).

`Time: O(1)` `Space: O(1)`

```
/**
 * @param {number} n
 * @return {Function} counter
 */
var createCounter = function (n) {
  return function () {
    return n++;
  };
};
```

2621. Sleep

Given a positive integer `millis`, write an asyncronous function that sleeps for `millis` milliseconds. It can resolve any value.

`Time: O(1)` `Space: O(1)`

```
/**
 * @param {number} millis
 */
async function sleep(millis) {
  return new Promise((resolve) => setTimeout(resolve, millis));
}
```

2623. Memoize

Given a function `fn`, return a **memoized** version of that function.

A **memoized** function is a function that will never be called twice with the same inputs. Instead it will return a cached value.

`Time: O() --> depends on the original function` `Space: O(n)`

```
/**
 * @param {Function} fn
 */
function memoize(fn) {
  const mem = {};
  return function (...args) {
    if (mem[args] !== undefined) return mem[args];
    mem[args] = fn(...args);
    return mem[args];
  };
}
```

2626. Array Reduce Transformation

Given an integer array `nums`, a reducer function `fn`, and an initial value `init`, return a **reduced** array.

`Time: O(n) ` `Space: O(1)

```
/**
 * @param {number[]} nums
 * @param {Function} fn
 * @param {number} init
 * @return {number}
 */
let reduce = function (nums, fn, init) {
  let val = init;
  for (let i = 0; i < nums.length; i++) {
    val = fn(val, nums[i]);
  }
  return val;
};
```

2629. Function Composition

Given an array of functions `[f1, f2, f3, ..., fn]`, return a new function `fn` that is the function composition of the array of functions.

The function composition of `[f(x), g(x), h(x)]` is `fn(x) = f(g(h(x)))`.

The function composition of an empty list of functions is the identity function `f(x) = x`.

`Time: O(n)` `Space: O(1)`

```
/**
 * @param {Function[]} functions
 * @return {Function}
 */
let compose = function (functions) {
  return function (x) {
    for (let i = functions.length - 1; i >= 0; i--) {
      const fn = functions[i];
      x = fn(x);
    }
    return x;
  };
};
```

Usage of reduceRight

array.reduceRight does the same as reduce, but starting from the right side (end) of the array.

`Time: O(n)` `Space: O(n)`

```
/**
 * @param {Function[]} functions
 * @return {Function}
 */
let compose = function (functions) {
  if (functions.length === 0) {
    return function (x) {
      return x;
    };
  }
  return functions.reduceRight(function (prevFn, nextFn) {
    return function (x) {
      return nextFn(prevFn(x));
    };
  });
};
```

2634. Filter Elements from Array

Given an integer array `arr` and a filtering function `fn`, return a new array with a fewer or equal number of elements.

`Time: O(n)` `Space: O(n)`

```
/**
 * @param {number[]} arr
 * @param {Function} fn
 * @return {number[]}
 */
var filter = function (arr, fn) {
  const filteredArr = [];
  for (let i = 0; i < arr.length; i++) {
    if (fn(arr[i], i)) {
      filteredArr.push(arr[i]);
    }
  }
  return filteredArr;
};
```

2635. Apply Transform Over Each Element in Array

Given an integer array `arr` and a mapping function `fn`, return a new array with a transformation applied to each element.

`Time: O(n)` `Space: O(n)`

```
/**
 * @param {number[]} arr
 * @param {Function} fn
 * @return {number[]}
 */
var map = function (arr, fn) {
  const mappedArr = [];
  for (let i = 0; i < arr.length; i++) {
    mappedArr.push(fn(arr[i], i));
  }
  return mappedArr;
};
```

2637. Promise Time Limit

Given an asyncronous function `fn` and a time t in milliseconds, return a new **time limited** version of the input function.

`Time: O(1)` `Space: O(1)`

```
/**
 * @param {Function} fn
 * @param {number} t
 * @return {Function}
 */
let timeLimit = function (fn, t) {
  return async function (...args) {
    const fns = fn(...args);
    const p = new Promise((res, rej) => {
      setTimeout(() => {
        rej("Time Limit Exceeded");
      }, t);
    });
    return Promise.race([fns, p]);
  };
};
```

2648. Generate Fibonacci Sequence

Write a generator function that returns a generator object which yields the **fibonacci sequence**.

`Time: is it O(1) because it always stops in a yield or is it O(n) because it is an infinite loop?` `Space: same here, is it O(1) because we only save 3 variables at a time, or O(n) because we use an infinite loop?`

```
/**
 * @return {Generator<number>}
 */
var fibGenerator = function* () {
  let a = 0;
  let b = 1;
  yield a;
  yield b;
  while (true) {
    let c = a + b;
    yield c;
    a = b;
    b = c;
  }
};
```

2665. Counter II

Write a function `createCounter`. It should accept an initial integer `init`. It should return an object with three functions.

The three functions are:

- `increment()` increases the current value by 1 and then returns it.
- `decrement()` reduces the current value by 1 and then returns it.
- `reset()` sets the current value to init and then returns it.

`Time: O(1)` `Space: O(1)`

Difficulty: in a closure, it takes the live value of the variables ==> you need to reassign count to init in the reset function and work on a new variable to keep init untouched.

```
/**
 * @param {integer} init
 * @return { increment: Function, decrement: Function, reset: Function }
 */
var createCounter = function (init) {
  let count = init || 0;
  return {
    increment: () => ++count,
    decrement: () => --count,
    reset: () => (count = init),
  };
};
```

2666. Allow One Function Call

Given a function `fn`, return a new function that is identical to the original function except that it ensures `fn` is called at most once.

- The first time the returned function is called, it should return the same result as `fn`.
- Every subsequent time it is called, it should return `undefined`

`Time: O(1)` `Space: O(1)`

```
/**
 * @param {Function} fn
 * @return {Function}
 */
var once = function (fn) {
  let canceled = false;
  let result;
  return function (...args) {
    if (canceled) {
      return undefined;
    } else {
      result = fn(...args);
      canceled = true;
      return result;
    }
  };
};
```

2667. Create Hello World Function

Write a function createHelloWorld. It should return a new function that always returns `"Hello World"`.

`Time: O(1)` `Space: O(1)`

```
/**
 * @return {Function}
 */
var createHelloWorld = function () {
  return function (...args) {
    return "Hello World";
  };
};
```

2677. Chunk Array

Given an array `arr` and a chunk size `size`, return a **chunked** array. A chunked array contains the original elements in `arr`, but consists of subarrays each of length `size`. The length of the last subarray may be less than `size` if `arr.length` is not evenly divisible by `size`.

`Time: O(n)` `Space: O(n+m)`

```
/**
 * @param {Array} arr
 * @param {number} size
 * @return {Array[]}
 */
var chunk = function (arr, size) {
  if (!arr.length) return [];
  const outer = [];
  let inner = [];
  for (let i = 0; i < arr.length; i++) {
    if ((i !== 0) & (i % size === 0)) {
      outer.push(inner);
      inner = [];
    }
    inner.push(arr[i]);
  }
  outer.push(inner);
  return outer;
};
```