二分查找

```
var search = function(nums, target) {
    let low = 0,high =nums.length - 1;
    while(low<=high){
        let mid = Math.floor((high - low)/2) + low;
        let num = nums[mid];
        if(num === target){
            return mid;
        }else if(num > target){
            high = mid - 1;
        }else{
            low = low + 1;
        }
    }
    return - 1;
};
```

快速排序

```
function quickSort(arr) {
  if (arr.length <= 1) {
    return arr;
  }
  var pivotIndex = Math.floor(arr.length / 2);
  var pivot = arr.splice(pivotIndex, 1)[0];
  var left = [];
  var right = [];
  for (var i = 0; i < arr.length; i++) {
    if (arr[i] < pivot) {
      left.push(arr[i]);
    } else {
      right.push(arr[i]);
    }
  }
  return quickSort(left).concat([pivot], quickSort(right));
}
```

回文数

```
var isPalindrome = function(x) {
    if(x < 0){
        return false;
    }else if(x == 0){
        return true;
    }else {
        return x.toString().split("").reverse().join("") == x;
    }
};
```

爬楼梯

```
var climbStairs = function(n) {
    const f = []
    f[1] = 1
    f[2] = 2
    for(let i = 3;i <= n;i++){
        f[i]  = f[i-2] + f[i-1]
    }
    return f[n]
};
```

千位分隔符

```
var thousandSeparator = function(n) {
    const str = n.toString();
    const arr = [];
    for(let length = str.length,i=length;i>0;i-=3){
        arr.unshift(str.substring(i-3,i));
    }
    return arr.join('.');
};
```

反转字符串

```
var reverseString = function(s) {
    reverse(s)
};
var reverse = function(s){
    let l=-1,r=s.length;
    while(++l < --r){
        [s[l],s[r]] = [s[r],s[l]];
    }
}
```

有效括号对

```
const leftToRight = {
    "(":")",
    "[":"]",
    "{":"}"
}
var isValid = function(s) {
    if(!s){
        return false
    }
    const stack = []
    const len = s.length
    if(len % 2 == 1){
        return false
    }
    for(let i =0;i<len;i++){
        const ch = s[i]
        if(ch === "("||ch==="{"||ch==="[")
            stack.push(leftToRight[ch]);
        else{
            if(!stack.length || stack.pop()!==ch){
                return false
            }
        }
    }
    return !stack.length
};
```

两数之和

```
var twoSum = function(nums, target) {
    const map = new Map()
    let len = nums.length
    for(let i = 0;i<len;i++){
        const value = target - nums[i];
        if(map.has(value)){
            return [map.get(value),i]
        }else{
            map.set(nums[i],i)
        }
    }
};
```

合并有序数组

```
const merge = function (nums1, m, nums2, n) {
            let i = m-1,j = n-1,k=m+n-1
            while(i>=0&&j>=0){
                if(nums1[i]>=nums2[j]){
                    nums1[k] = nums1[i]
                    i--
                    k--
                } else {
                    nums1[k] = nums2[j]
                    j--
                    k--
                }
            }

        while(j>=0){
            nums1[k] = nums2[j]
            k--
            j--
        }
}
--------------------------------------
const merge = function (nums1, m, nums2, n) {
    nums1.splice(m,nums1.length-m,...nums2);
    nums1.sort((a,b) => a - b)
}
```

数组中数字出现次数

```
var singleNumbers = function(nums) {
    let numsSort = nums.sort((a,b)=>{ return a-b }),result = []
    for(let i=0;i<numsSort.length;i++){
        if(numsSort[i]===numsSort[i+1]){
            i++
        }else{
            result.push(numsSort[i])
            if(result.length===2) break
        }
    }
    return result
};
```

字符串去重

```
function duplicateRemoval(str) {
    return [...new Set(str.split(""))].join("")
}
```

最长不含重复字符的子字符串

```
var lengthOfLongestSubstring = function(s) {
    let arr = [];
    let max = 0;
    for(let item of s){
        if(arr.includes(item)){
            let index = arr.indexOf(item);
            arr.splice(0, index + 1);
        }
        arr.push(item);
        max = max > arr.length ? max : arr.length;
    }
    return max;
};
```

最长递增子序列

```
const lengthOfLIS = function(nums) {
    // 因为每个元素都能自己组成一个长度为1的子序列，所以dp数组所有元素的初始值设置为1
    const dp = new Array(nums.length).fill(1)
    for (let i = 1; i < nums.length; i++) {
        for (let j = 0; j < i; j++) {
            // 找出第i个元素之前的序列中，有多少个元素比其更小
            if (nums[j] < nums[i]) {
                // 相当于找到了一个j，则dp[i]就加1，一直从0找到i-1
                dp[i] = Math.max(dp[i], dp[j] + 1)
            }
        }
    }
    return Math.max(...dp)
}
```

删除链表倒数第n个元素

```
var removeNthFromEnd = function(head, n) {
    let p = head
    for(let i = 1;i < n;i++)
        p = p.next
    let dump = cur = {val:undefined,next:head}
    while(p.next){
        cur = cur.next
        p = p.next
    }
    cur.next = cur.next.next
    return dump.next
};
```

反转链表

```
var reverseList = function(head) {
    let pre = null;
    let cur = head;
    while(cur !== null){
        let next = cur.next;
        cur.next = pre;
        pre = cur;
        cur = next;
    }
    return pre;
};
```

删除链表重复元素

```
var deleteDuplicates = function(head) {
    let cur = head
    while(cur != null && cur.next != null){
        if(cur.val === cur.next.val){
                cur.next = cur.next.next
        }else{
            cur = cur.next
        }
    }
    return head
};
```

对称二叉树

```
var isSymmetric = function(root) {
    return isSymmetricCore(root,root)
};
var isSymmetricCore = function(n1,n2) {
    if(!n1 && !n2)
        return true;
    if(!n1 || !n2)
        return false;
    if(n1.val!==n2.val)
        return false;
    return isSymmetricCore(n1.left,n2.right) && isSymmetricCore
(n1.right,n2.left)
};
```

翻转二叉树

```
const invertTree = function(root) {
    // 定义递归边界
    if(!root) {
        return root;
    }
    // 递归交换右孩子的子结点
    let right = invertTree(root.right);
    // 递归交换左孩子的子结点
    let left = invertTree(root.left);
    // 交换当前遍历到的两个左右孩子结点
    root.left = right;
    root.right = left;
    return root;
};
```

中序遍历二叉树

```
var inorderTraversal = function(root) {
    const res = []
    const stack = []
    let cur = root
    while(cur||stack.length){
        while(cur){
            stack.push(cur)
            cur=cur.left
        }
        cur = stack.pop()
        res.push(cur.val)
        cur = cur.right
    }
    return res
};
```

层序遍历二叉树

```
var levelOrder = function(root) {
    const res = []
    if(!root){
        return res
    }
    const queue = []
    queue.push(root)
    while(queue.length){
        const lever = []
        const len = queue.length
        for(let i = 0;i<len;i++){
            const top  =queue.shift()
            lever.push(top.val)
            if(top.left){
                queue.push(top.left)
            }
            if(top.right){
                queue.push(top.right)
            }
        }
        res.push(lever)
    }
    return res
};
```

验证二叉搜索树

```
var isValidBST = function(root) {
    function dfs(root,minValue,maxValue){
        if(!root){
            return true
        }
        if(root.val <= minValue || root.val >= maxValue)
            return false
        return
dfs(root.left,minValue,root.val)&&dfs(root.right,root.val,maxValue)
    }
    return dfs(root,-Infinity,Infinity)
};
```

二叉树最近公共祖先

```
var lowestCommonAncestor = function(root, p, q) {
    if(!root || root === p || root === q) return root;
    let left = lowestCommonAncestor(root.left, p ,q);
    let right = lowestCommonAncestor(root.right, p, q);
    if(!left) return right;
    if(!right) return left;
    return root;
};
```

手写防抖

```
function debounce(callback, time) {
    // 定时器变量
    let timer = null;
    // 返回一个函数
    return function (e) {
        if (timer !== null) {
            // 清空定时器
            clearTimeout(timer);
        }
        // 启动定时器
        timer = setTimeout(() => {
            // 执行回调
            callback(this, e);
            // 重置定时器变量
            timer = null;
        }, time);
    }
}
```

手写节流

```
function throttle(callback, wait) {
    // 获取开始时间戳
    let start = Date.now();
    // 返回结果是一个函数
    return function (e) {
        // 获取当前时间戳
        let now = Date.now();
        if (now - start >= wait) {
            // 满足条件执行回调函数
            callback.call(this, e);
            // 修改开始时间
            start = now;
        }
    }
}
```

函数柯里化

```
function curry(fn, ...args) {
    return fn.length <= args.length ? fn(...args) : curry.bind(null, fn,
...args);
 }
```

手写Promise

```
//声明构造函数 -- executor 会在 Promise 内部立即同步调用,异步操作在执行器中执行
function Promise(executor) {
    //添加属性
    this.PromiseState = 'pending';//值只能为: resolved rejected
    this.PromiseResult = null;
    //声明属性 -- 数组
    this.callbacks = [];
    //保存实例对象的 this 的值
    const self = this;//self _this that -- this指向问题
    //resolve 函数
    function resolve(data) {
        //判断状态 -- 无论失败还是成功一个promise对象只能改变一次
        if (self.PromiseState !== 'pending') return;
        //1. 修改对象的状态 (promiseState)
        self.PromiseState = 'fulfilled';// === resolved
        //2. 设置对象结果值 (promiseResult)
        self.PromiseResult = data;
        //改变状态之后 -- 调用成功的回调函数 -- 遍历让多个回调都执行
        setTimeout(() => {
            self.callbacks.forEach(item => {
                item.onResolved(data);
            });
        });
    }
    //reject函数 -- 失败
    function reject(data) {
        //判断状态 -- 无论失败还是成功一个promise对象只能改变一次
        if (self.PromiseState !== 'pending') return;
        //1. 修改对象的状态 (promiseState)
        self.PromiseState = 'rejected';//
        //2. 设置对象结果值 (promiseResult)
        self.PromiseResult = data;
        //改变状态之后 -- 调用失败的回调函数 -- 遍历让多个回调都执行
        setTimeout(() => {
            self.callbacks.forEach(item => {
                item.onRejected(data);
            });
        })
    }
    try {
        //同步调用『执行器函数』
        executor(resolve, reject);
    } catch (e) {
        //修改 promise 对象状态为『失败』
        reject(e);
    }
}

//添加 then 方法 -- 用于得到成功value 的成功回调和用于得到失败reason的失败回调返回一个新的
promise对象
Promise.prototype.then = function (onResolved, onRejected) {
    const self = this;
    //判断回调函数参数
    if (typeof onRejected !== 'function') {
        onRejected = reason => {
            throw reason;
        }
    }
    if (typeof onResolved !== 'function') {
        onResolved = value => value;
        //value => { return value};
    }
    return new Promise((resolve, reject) => {
        //封装函数
        function callback(type) {
            try {
                //获取回调函数的执行结果
                let result = type(self.PromiseResult);
                //判断
                if (result instanceof Promise) {
                    //如果是 Promise 类型的对象
                    result.then(v => {
                        resolve(v);
                    }, r => {
                        reject(r);
                    })
                } else {
                    //结果的对象状态为『成功』
                    resolve(result);
                }
            } catch (e) {
                reject(e);
            }
        }
        //调用回调函数  PromiseState
        if (this.PromiseState === 'fulfilled') {
            setTimeout(() => {
                callback(onResolved);
            })
        }
        if (this.PromiseState === 'rejected') {
            setTimeout(() => {
                callback(onRejected);
            })
        }
        //判断 pending 状态
        if (this.PromiseState === 'pending') {
                //保存回调函数
                this.callbacks.push({
                    onResolved: function () {
                        callback(onResolved);
                    },
                    onRejected: function () {
                        callback(onRejected);
                    }
                });
            }
        })
}

//添加 catch 方法
Promise.prototype.catch = function (onRejected) {
    return this.then(undefined, onRejected);
}

//添加 resolve 方法 -- 返回一个成功/失败的promise对象
Promise.resolve = function (value) {
    //返回promise对象
    return new Promise((resolve, reject) => {
        if (value instanceof Promise) {
            value.then(v => {
                resolve(v);
            }, r => {
                reject(r);
            })
        } else {
            //状态设置为成功
            resolve(value);
        }
    });
}

//添加 reject 方法 -- 返回一个失败的promise对象
Promise.reject = function (reason) {
    return new Promise((resolve, reject) => {
        reject(reason);
    });
}

//添加 all 方法 -- 返回一个新的promise，只有所有的promise都成功才成功，只要有一个失败了就直
接失败
Promise.all = function (promises) {
    //返回结果为promise对象
    return new Promise((resolve, reject) => {
        //声明变量
        let count = 0;
        let arr = [];
        //遍历
        for (let i = 0; i < promises.length; i++) {
            //
            promises[i].then(v => {
                //得知对象的状态是成功
                //每个promise对象 都成功
                count++;
                //将当前promise对象成功的结果 存入到数组中
                arr[i] = v;
                //判断
                if (count === promises.length) {
                    //修改状态
                    resolve(arr);
                }
            }, r => {
                reject(r);
            });
        }
    });
}

//添加 race 方法 -- 返回一个新的promise，第一个完成的promise 的结果状态就是最终的结果状态
Promise.race = function (promises) {
    return new Promise((resolve, reject) => {
        for (let i = 0; i < promises.length; i++) {
            promises[i].then(v => {
                //修改返回对象的状态为 『成功』
                resolve(v);
            }, r => {
                //修改返回对象的状态为 『失败』
                reject(r);
            })
        }
    });
}
```

手写浅拷贝

```
function shallowCopy(object) {
    if (!object || typeof object !== "object") return;
    let newObject = Array.isArray(object) ? [] : {};
    for (let key in object) {
        if (object.hasOwnProperty(key)) {
            newObject[key] = object[key];
        }
    }
    return newObject;
}
```

手写深拷贝

```
function deepCopy(object) {
    if (!object || typeof object !== "object") return;
    let newObject = Array.isArray(object) ? [] : {};
    for (let key in object) {
        if (object.hasOwnProperty(key)) {
            newObject[key] = typeof object[key] === "object" ?
deepCopy(object[key]) : object[key];
        }
    }
    return newObject;
}
```

实现数组去重

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

实现数组的flat方法

```
function _flat(arr, depth) {
    if (!Array.isArray(arr) || depth <= 0) {
        return arr;
    }
    return arr.reducr((prev, cur) => {
        if (Array.isArray(cur)) {
            return pre.concat(_flat(cur, depth - 1));
        } elseP
        return pre.concat(cur);
    }, [])
}
```

js对象转换为树形结构

```
function jsonToTree(data) {
    let res = [];
    if (!Array.isArray(data)) {
        return res;
    }
    let map = {};
    data.forEach(item => {
        map[item.id] = item;
    });
    data.forEach(item => {
        let parent = map[item.pid];
        if (parent) {
            (parent.children || (parent.children = [])).push(item);
        } else {
            res.push(item);
        }
    });
    return res
}
```

解析URL Params为对象

```
function parseParam(url) {
    // 将? 后面的字符串取出来
    const paramsStr = /.+\?(.+)$/.exec(url)[1];
    // 将字符串以&分割后存到数组中
    const paramsArr = paramsStr.split('&');
    let paramsObj = {};
    paramsArr.forEach(param => {
        if (/=/.test(param)) {
            let [key, val] = param.split('=')
            val = decodeURIComponent(val);  //解码
            val = /^\d+$/.test(val) ? parseFloat(val) : val;
            if (paramsObj.hasOwnProperty(key)) {
                paramsObj[key] = [].concat(paramsObj[key], val);
            } else {
                paramsObj[key] = val;
            }
        } else {
            paramsObj[param] = true;
        }
    })
    return paramsObj;
}
```

手写红黄绿灯

```
//红黄绿:使用异步编程方案;循环打印:一轮打印完了以后递归重复这一过程使用异步编程方案
const taskRunner = (light, timeout) => {
    return new Promise((resolve) => {
        setTimeout(() => resolve(console.log(light)), timeout);
    })
}
const task = async () => {
    await taskRunner('红', 1000)
    await taskRunner('绿', 3000)
    await taskRunner('黄', 2000)
    task()
}
```

```
task()
```

手写实现发布-订阅模式

```
class EventCenter {
    // 1.定义事件容器
    constructor() {
        this.handlers = {}
    }
    // 2.添加事件方法（事件名，事件方法）
    addEventListener(type, handler) {
        // 创建新数组容器
        if (!this.handlers[type]) {
            this.handlers[type] = []
        }
        // 存入事件
        this.handlers[type].push(handler)
    }
    // 3.触发事件（事件名，事件参数）
    dispatchEvent(type, params) {
        // 没有注册该事件则抛出错误
        if (!this.handlers[type]) {
            return new Error('该事件未注册')
        }
        // 触发事件
        this.handlers[type].forEach(handler => {
            handler(...params)
        });
    }
    // 4.事件移除        （事件名，事件方法）
    removeEventListener(type, handler) {
        if (!this.handlers[type]) {
            return new Error('事件无效')
        }
        if (!handler) {
            // 移除事件
            delete this.handlers[type]
        } else {
            const index = this.handlers[type].findIndex(el => el === handler)
            if (index === -1) {
                return new Error('无法绑定该事件')
            }
        }
        // 移除事件
        this.handlers[type].splice(index, 1)
        if (this.handlers[type].length === 0) {
            delete this.handlers[type]
        }
    }
}
```