目录

[第一题 字符串是否由重复子串构成 2](#_Toc54871308)

[第二题 有序数组，两数之和 3](#_Toc54871309)

[第三题 写一个类，把数组转化成链表，并实现增删改查的功能 3](#_Toc54871310)

[第五题 顺时针打印矩阵 6](#_Toc54871311)

[第六题 洗牌算法 7](#_Toc54871312)

[第八题 力扣746 使用最小花费爬楼梯 7](#_Toc54871313)

[第十题 实现sqrt函数 9](#_Toc54871314)

[第十三题 大数相加 10](#_Toc54871315)

[用两个栈实现队列 10](#_Toc54871316)

[十进制转二进制 11](#_Toc54871317)

[生成括号 12](#_Toc54871318)

[组合总和 13](#_Toc54871319)

[三个数最大乘积 15](#_Toc54871320)

[三数之和 15](#_Toc54871321)

[==============链表================ 16](#_Toc54871322)

[链表链表 反转链表 16](#_Toc54871323)

[未排序链表去重 17](#_Toc54871324)

[链表环 17](#_Toc54871325)

[======树============ 18](#_Toc54871326)

[树树树树树树树 数组转树 19](#_Toc54871327)

[二叉树右视图 19](#_Toc54871328)

[中序遍历非递归 20](#_Toc54871329)

[蛇形打印二叉树 21](#_Toc54871330)

[前序和中序重建二叉树 22](#_Toc54871331)

[二叉树翻转 23](#_Toc54871332)

[判断是否为平衡二叉树 23](#_Toc54871333)

[树中第k大的数 24](#_Toc54871334)

[二叉树深度 24](#_Toc54871335)

[打印所有路径 25](#_Toc54871336)

[寻找和为n的路径 26](#_Toc54871337)

[=======DP========== 26](#_Toc54871338)

[最长上升子序列 26](#_Toc54871339)

[最长公共子序列 27](#_Toc54871340)

[连续子数组最大和 27](#_Toc54871341)

[最大回文子串 28](#_Toc54871342)

[=======排序===== 29](#_Toc54871343)

[快速排序 29](#_Toc54871344)

[二分排序 30](#_Toc54871345)

[归并排序 31](#_Toc54871346)

[======数组======= 32](#_Toc54871347)

[合并两个有序数组 32](#_Toc54871348)

[旋转数组 32](#_Toc54871349)

[全排列 33](#_Toc54871350)

[数组扁平化 递归 33](#_Toc54871351)

[=======工程上的题 34](#_Toc54871352)

[hex转rgb 34](#_Toc54871353)

[原生实现call apply 35](#_Toc54871354)

[Promise.all race 36](#_Toc54871355)

[高阶组件 38](#_Toc54871356)

[函数柯理化 38](#_Toc54871357)

[函数防抖 39](#_Toc54871358)

[并发控制 40](#_Toc54871359)

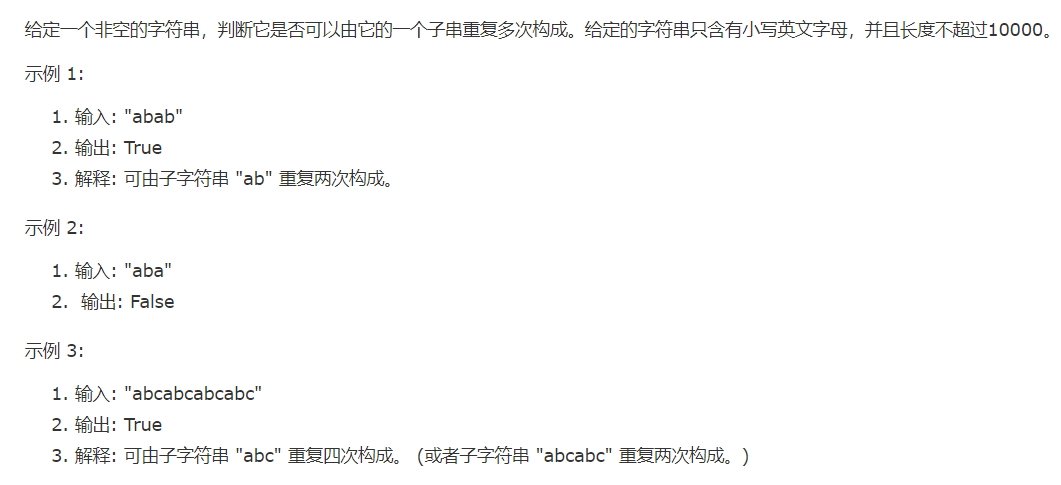
[获取给定k v 的dom节点 41](#_Toc54871360)

[模拟instanceof 42](#_Toc54871361)

[在window全局添加事件 43](#_Toc54871362)

[实现render函数 43](#_Toc54871363)

# 第一题 字符串是否由重复子串构成

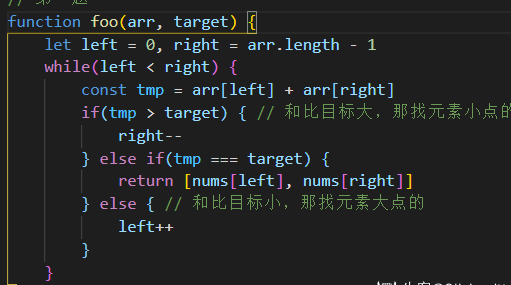


1. 这里 if length%i==0这句判断函数的作用是判断i是否的是length的约数，如果是就将s（字符串）分为length/i个窗口琢个比较，每符合一次向后滑动一次，只要有一次不符合就把res置为false。最后查看窗口的位置，如果已经滑到了s的最后，那么则符合题意，把res置为true

*function* fn(str) {  
 *let* length = str.length;  
 *let* res = *true*;  
 *for*(*let* i = 1; i<length;i++ ){  
 *let* subLen = i;  
 *if* (length %(subLen) === 0){  
 *let* temp = str.substr(0,subLen)  
 *for*(*let* j = 0; j<length;j++ ){  
 *if* (j%subLen===0&&str.substr(j,subLen)!==temp){  
 res = *false  
 break*;  
 }  
 *//如果滑动到最后一个 则表示全部匹配 置为true  
 if* (j===length-1) {  
 res = *true* }  
 }  
 }  
 }  
 *return* res  
}

# 第二题 有序数组，两数之和

给出一个有序数组和一个数字，输出数组中和为这个数字的下标，比如[1,2,3,4,5,6] 6=>[[1,5],[2,4]]



# 第三题 写一个类，把数组转化成链表，并实现增删改查的功能

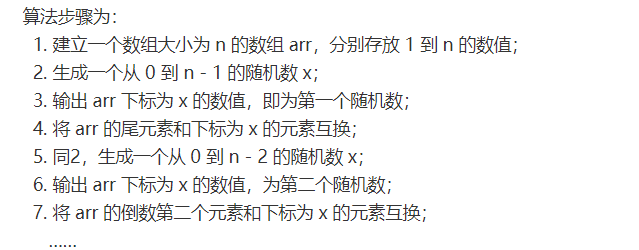
*function* Node(val){  
 *this*.val = val;  
 *this*.next = *null*;  
}  
*class* LinkedList {  
 constructor(arr) {  
 *if* (arr.length === 0) {  
 *this*.head = *null* } *else* {  
 *this*.head = *new* Node(arr[0])  
 *let* node, p = *this*.head;  
 *for* (*let* i = 1; i < arr.length; i++) {  
 node = *new* Node(arr[i]);  
 p.next = node;  
 p = node;  
 }  
 }  
 }  
  
 add(val) {  
 *let* p = *this*.head.next,previous = *this*.head;  
 *let* node = *new* Node(val)  
 *if* (val <= *this*.head.val){  
 *this*.head = node;  
 node.next = previous;  
 *return* }  
 *while* (p.next) {  
 *if* (val<=p.val){  
 previous.next = node;  
 node.next = p;  
 *return* }  
 p = p.next;  
 previous = previous.next  
 }  
 p.next = node  
 }  
 update(index,num){  
 *let* i=0,p=*this*.head;  
 *while* (p){  
 *if* (i===index) {  
 p.val = num  
 *return true* }*else* {  
 p=p.next;  
 i++;  
 }  
 }  
 *return false* }  
 *delete*(index){  
 *let* i=1,pre=*this*.head,p =*this*.head.next;  
 *if* (index===0){  
 *this*.head = *this*.head.next;  
 }  
 *while* (p){  
 *if* (i===index) {  
 pre.next = p.next0  
 *return true* }*else* {  
 p=p.next;  
 pre = pre.next  
 i++;  
 }  
 }  
 *return false* }  
 print(){  
 *let* p = *this*.head,arr = [];  
 *while* (p.next){  
 arr.push(p.val);  
 p = p.next  
 }  
 arr.push(p.val)  
 console.log(arr)  
 }  
}

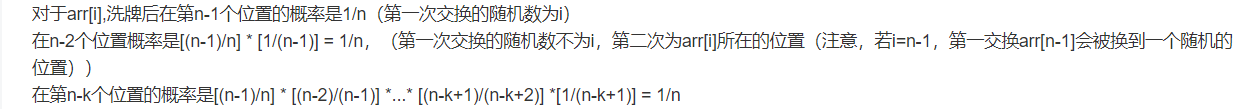
# 第五题 顺时针打印矩阵

* 每次打印都是从开头开始的即【0,0】【1,1】作以可以根据矩阵长度确定总共要遍历几次，即endFlag;
* 第一次从[0,0]开始，之后每次的行列值都在变化，所以行列之需要一个变量来存放；矩阵有四条边，所以row col有四种不同的变化；
* 每一轮的打印次数有公式可知 是 4（t-1）;t初始值是length，打印一次之后,t--;当t等于1时表示只有一个数，此时不用循环，直接打印这个数就可；
* 然后由于新一轮的顺时针打印row,col的边界条件会变化，所以用flag来表示变化

*function* fn(matrixs) {  
 *let* length = matrixs.length,width = matrixs[0].length  
 *let* row=0,col=0,t;*//t表示每一圈打印的个数  
 let* endFlag = Math.ceil(length/2);  
 *let* res=[],flag=1;*//flag表示这次是第几圈  
 if* (length===1){  
 t = width  
 } *else* {  
 t = width\*2+(length-2)\*2  
 }  
 *for*(*let* i = 0; i<endFlag;i++ ){  
  
 *for*(*let* j = 0; j<t;j++ ){  
 res.push(matrixs[row][col])  
 *if* (row===flag-1&&col!==(width-flag)){  
 col++  
 } *else if* (col===width-flag&&row!==(length-flag)){  
 row++  
 } *else if* (row===length-flag&&col!==flag-1){  
 col--  
 } *else if* (col===flag-1&&row!==flag){  
 row--  
 }  
 }  
 col++;  
 *//当前是最后一圈 并且length是奇数  
 if* (flag===endFlag-1&&length%2!==0){  
 t = width-flag\*2  
 } *else* {  
 t = (width-flag\*2)+(length-flag\*2)\*2  
 }  
 flag++  
 }  
 *return* res  
}

# 第六题 洗牌算法





*function* shuffle(arr) {  
 *let* t;  
 *for*(*let* i = arr.length-1; i>=0;i-- ){  
 *let* randomNum = parseInt(Math.random()\*(i+1))  
 t = arr[randomNum];  
 arr[randomNum] = arr[i];  
 arr[i] = t  
 }  
 console.log(arr)  
}

# 第八题 力扣746 使用最小花费爬楼梯

**第一步：定义子问题**

踏上第 i 级台阶的体力消耗为到达前两个阶梯的最小体力消耗加上本层体力消耗：

最后迈 1 步踏上第 i 级台阶：dp[i-1] + cost[i]

最后迈 2步踏上第 i 级台阶：dp[i-2] + cost[i]

**第二步：实现需要反复执行解决的子子问题部分**

所以踏上第 i 级台阶的最小花费为：

dp[i] = min(dp[i-2], dp[i-1]) + cost[i]

**第三步：识别并求解出边界条件**

// 第 0 级 cost[0] 种方案

dp[0] = cost[0]

// 第 1 级，有两种情况

// 1：分别踏上第0级与第1级台阶，花费cost[0] + cost[1]

// 2：直接从地面开始迈两步直接踏上第1级台阶，花费cost[1]

dp[1] = min(cost[0] + cost[1], cost[1]) = cost[1]

第一步push一个0是用来表示总共的值，即比较dp[len-1]，dp[len-2]的最小值

原始版本

*function* minCostClimbingStairs(cost) {  
 cost.push(0)  
 *let* dp = [],len = cost.length  
 dp[0] = cost[0];  
 dp[1] = cost[1];  
 *for*(*let* i = 2; i<len;i++ ){  
 dp[i] = Math.min(dp[i-1],dp[i-2])+cost[i]  
 }  
  
 *return* dp[len-1]  
}

优化版本

因为自始至终只用保存前两个的最优值，所以只需要两个变量即可，降低了空间复杂度

*function* minCostClimbingStairs\_Optimition(cost){  
 cost.push(0);  
 *let* len = cost.length,dp1=cost[0],dp2=cost[1],t;  
 *for*(*let* i = 2; i<len;i++ ){  
 t = dp2;  
 dp2 = Math.min(dp1,dp2)+cost[i];  
 dp1 = t;  
 }  
 *return* Math.min(dp1,dp2)

# 第十题 实现sqrt函数

二分法

*function* sqrt(num) {  
 *let* min=0,max=num,mid=num/2;  
 *while*(Math.abs(mid\*mid-num)>0.001){  
 *if* (mid\*mid>num){  
 max = mid;  
 }*else* {  
 min = mid;  
 }  
  
 mid = (min+max)/2  
  
 }  
 *return* mid  
}

用牛顿迭代法

*function* sqrt(num) {  
 *if* (num<0) {  
 *return* -1  
 }  
 *if* (num===0){  
 *return* 0  
 }  
 *let* lastVal,now= num/2;  
 *do* {  
 lastVal = now;  
 *// 核心公式* now = (lastVal+num/lastVal)/2;  
 }*while* (Math.abs(now-lastVal)>0.01)  
 *// floor 向下取整，即把小数部分全部舍弃  
 return* Math.floor(now\*1000)/1000   
}

# 第十三题 大数相加

先用padStart补0，再相加

*function* calcLongNum(s1,s2) {  
 *let* len1 = s1.length,len2=s2.length;  
 *let* maxlen = Math.max(len1,len2);  
 s1 = s1.padStart(maxlen+1,'0');  
 s2 = s2.padStart(maxlen+1,'0')  
 *let* arr1 = s1.split('').reverse(),arr2 = s2.split('').reverse();  
 *let* res = ''  
 *let* nowNum,carry = 0;  
 *for*(*let* i = 0; i<maxlen+1;i++ ){  
 nowNum = parseInt(arr1[i])+parseInt(arr2[i])+carry;  
 carry = parseInt(nowNum/10);  
 res = nowNum%10+res  
 }  
 *return* res.charAt(0)==='0'?res.substr(1):res;  
}

# 用两个栈实现队列

*function* push(node) {  
 *// write code here* stackPush.push(node);  
}  
  
*function* pop() {  
 *if* (stackPop.length==0 && stackPush.length==0) {  
 console.log("Queue is empty!");  
 } *else if* (stackPop.length==0) {  
 *while* (!stackPush.length==0) {  
 stackPop.push(stackPush.pop());  
 }  
 }  
 *return* stackPop.pop();  
}

# 十进制转二进制

*function* demicalToBinary(num) {  
 *if* (num>=1){  
 *if* (num%2){  
 *return* demicalToBinary((num-1)/2)+1  
 } *else* {  
 *return* demicalToBinary((num/2))+0  
 }  
 }   
 *return* ''  
}

# 生成括号

*var* generateParenthesis = *function*(n) {  
 *// 回溯思想  
 const* res = []  
 *const* go = (l, r, s) => { *// l : 已使用左括号数量 ; r : 已使用右括号数量  
 // 当左右括号剩余量都为n时 表示已经使用完 将结果插入插入到结果集中)  
 if* (l === n && r === n) {  
 res.push(s)  
 *return* }  
 *// 回溯剪枝条件：还有剩余左括号； 已使用的左括号大于右括号并且还有右括号可以使用  
 if* (l < n) {  
 go(l + 1, r, s + '(')  
 }  
 *if* (l > r && r < n){  
 go(l, r + 1,s + ')')  
 }  
 }  
 go(0, 0, '')  
 *return* res  
};

# 组合总和

**无重复元素**的数组 candidates 和一个目标数 target

每个数可以重复使用

*var* combinationSum = *function*(candidates, target) {  
 *const* res = [];  
 *const* dfs = (target, combine, idx) => {  
 *if* (target === 0) {  
 res.push(combine);  
 *return*;  
}  
 *if* (idx === candidates.length) {  
 *return*;  
}

*//每个数都可以用或者不用；  
 //不用 直接跳过* dfs(target, combine, idx + 1);  
 *// 用 选择当前数  
 if* (target - candidates[idx] >= 0) {  
 dfs(target - candidates[idx], [...combine, candidates[idx]], idx);  
 }  
 };  
 *//开始* dfs(target, [], 0);  
 *return* res;  
};

每个数只能使用一次

*var* combinationSum = *function*(candidates, target) {  
 *const* res = [];  
 *//相比于上面 这里加了一个排序* candidates.sort((a,b)=>{  
 *return* a-b  
 })  
 *const* dfs = (target, combine, idx) => {  
 *if* (target === 0) {  
 res.push(combine);  
 *return*;  
 }  
 *if* (idx === candidates.length) {  
 *return*;  
 }  
 *//每个数都可以用或者不用；  
 //不用 直接跳过* dfs(target, combine, idx + 1);  
 *// 用 ，选择当前数  
 if* (target - candidates[idx] >= 0) {  
 *if* (candidates[idx]!==candidates[idx-1]) {  
 dfs(target - candidates[idx], [...combine, candidates[idx]], idx+1);*//相比于上面 这里idx改成了idx+1* }  
 }  
 };  
 *//开始* dfs(target, [], 0);  
 console.log(res)  
};

# 三个数最大乘积

*var* maximumProduct = *function*(nums) {  
 nums.sort((a, b) => a - b)  
 *let* len = nums.length  
 *let* res1 = 1  
 *let* res2 = 1  
 *for*(*let* i = 1; i <= 3; i++) {  
 res1 \*= nums[len - i]  
 }  
 res2 = nums[0] \* nums[1] \* nums[len - 1]  
 *return* Math.max(  
 res1, res2  
 )  
};

# 三数之和

*var* threeSum = *function*(nums) {  
 *let* ans = [];  
 *const* len = nums.length;  
 *if*(nums == *null* || len < 3) *return* ans;  
 nums.sort((a, b) => a - b); *// 排序  
 for* (*let* i = 0; i < len ; i++) {  
 *if*(nums[i] > 0) *break*; *// 如果当前数字大于0，则三数之和一定大于0，所以结束循环  
 if*(i > 0 && nums[i] == nums[i-1]) *continue*; *// 去重  
 let* L = i+1;  
 *let* R = len-1;  
 *while*(L < R){  
 *const* sum = nums[i] + nums[L] + nums[R];  
 *if*(sum == 0){  
 ans.push([nums[i],nums[L],nums[R]]);  
 *while* (L<R && nums[L] == nums[L+1]) L++; *// 去重  
 while* (L<R && nums[R] == nums[R-1]) R--; *// 去重* L++;  
 R--;  
 }  
 *else if* (sum < 0) L++;  
 *else if* (sum > 0) R--;  
 }  
 }  
 *return* ans;  
};

# ==============链表================

# 链表链表 反转链表

*var* reverseList = *function*(head) {  
 *//let [prev, curr] = [null, head];  
 let* prev=*null*;  
 *let* curr=head;  
 *while* (curr) {  
 *let* tmp = curr.next; *// 1. 临时存储当前指针后续内容* curr.next = prev; *// 2. 反转链表* prev = curr; *// 3. 接收反转结果* curr = tmp; *// 4. 接回临时存储的后续内容* }  
 *return* prev;  
};

# 未排序链表去重

*var* removeDuplicateNodes = *function* (head) {  
 *var* p = head;  
 *while* (p) {  
 *var* q = p;  
 *while* (q.next) {  
 *if* (q.next.val == p.val) {  
 q.next = q.next.next;  
 } *else* {  
 q = q.next;  
 }  
 }  
 p = p.next;  
 }  
 *return* head;  
};

# 链表环

//给定一个链表，判断链表中是否有环。 *var* hasCycle = *function*(head) {  
 *if*(!head || !head.next) *return false  
 let* former= head  
 *let* latter= head.next  
 *while*(former!= latter){  
 *if*(!latter || ! latter.next) *return false* latter = latter.next.next  
 former = former.next  
 }  
 *return true*};

//环形链表第一个入环节点  
//给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。 *var* detectCycle = *function*(head) {  
 *if*(!head || !head.next) *return null  
 let* former = head;  
 *let* latter = head;  
 *while*(latter!= *null* && latter.next != *null*) {  
 former = former.next;  
 latter = latter.next.next;  
 *if*(former == latter) {  
 latter = head;  
 *while*(former!= latter) {  
 former = former.next;  
 latter = latter.next;  
 }  
 *return* former  
 }  
 }  
 *return null*};

# ======树============

# 树树树树树树树 数组转树

*function* array2Tree(arr){  
 *if*(!Array.isArray(arr) || !arr.length) *return*;  
 *let* map = {};  
 arr.forEach(item => map[item.id] = item);  
  
 *let* roots = [];  
 arr.forEach(item => {  
 *const* parent = map[item.parentId];  
 *if*(parent){  
 (parent.children ||

(parent.children=[])).push(item);  
 }  
 *else*{  
 roots.push(item);  
 }  
 })  
  
 *return* roots;  
}

# 二叉树右视图

*var* rightSideView = *function*(root) {  
 *if*(!root) *return* []  
 *let* arr = []  
 dfs(root, 0, arr)  
 *return* arr  
};  
*function* dfs (root, step, res) {  
 *if*(root){  
 *if*(res.length === step){  
 res.push(root.val)  
*// 当数组长度等于当前 深度 时, 把当前的值加入数组* }  
 dfs(root.right, step + 1, res)  
*// 先从右边开始, 当右边没了, 再轮到左边* dfs(root.left, step + 1, res)  
 }  
}

# 中序遍历非递归

*function* inorder(root) {*//中序非递归 BST第K小的数 第K大见下面  
 if* (!root) *return null*;  
 *var* stack = [];  
 *var* p = root;  
 *//var pre=-Infinity;  
 while* (stack.length > 0 || p) {  
 *if* (p) { *//当前非空，当前入栈，左移* stack.push(p);  
 p = p.left;  
 } *else* { *//栈弹出，并右移* p = stack.pop();  
 console.log(p.value);*//在此和前一个数比较 判断是否为二叉搜索树* p = p.right;  
 }  
 }  
}

*function* pre(root) {*//先序非递归 中左右  
 var* stack = [];  
 stack.push(root);  
 *while* (stack) {  
 *// 移除最后一个  
 var* temp = stack.pop();  
 console.log(temp.value);  
 *// 后进先出  
 if* (temp.right != *null*)  
 stack.push(temp.right);  
 *if* (temp.left != *null*)  
 stack.push(temp.left);  
 }  
}  
*function* pos(root) {*//后序非递归 中右左=>左右中   
 var* stack1 = [];  
 *var* stack2 = [];  
 stack1.push(root);  
 *while* (stack1) {  
 *var* temp = stack1.pop();  
 stack2.push(temp);  
 *if* (temp.left != *null*)  
 stack1.push(temp.left);  
 *if* (temp.right != *null*)  
 stack1.push(temp.right);  
 }  
 *while* (stack2) {  
 console.log(stack2.pop().value);  
 }  
}

# 蛇形打印二叉树

printSnake(){  
 *//1用来存奇数层 2用来存偶数层；偶数层从右子树开始存，同时存储的奇数层数字是从右边开始遍历，（同时奇数层得从左边开始遍历），符合先进后出，所以得用栈存放  
 let* stack1 = [],queue2 = [];  
 *let* level = 1,res = [];  
 stack1.push(*this*.root);  
 *while* (stack1.length!==0||queue2.length!==0){  
 *if* (level%2!==0){*//奇数层 从q1取值 并且把孩子节点存入queue2  
 const* node = stack1.pop();  
 res.push(node.key)  
 *if* (node.right){  
 queue2.push(node.right)  
 }  
 *if* (node.left){  
 queue2.push(node.left)  
 }  
 *if* (stack1.length ===0 ){  
 level++  
 }  
 }  
 *else* {  
 *const* node = queue2.shift();  
 res.push(node.key)  
 *if* (node.right){  
 stack1.push(node.right)  
 }  
 *if* (node.left){  
 stack1.push(node.left)  
 }  
 *if* (queue2.length ===0 ){  
 level++  
 }  
 }  
 }  
 console.log(res)  
}

# 前序和中序重建二叉树

*function* reConstructBinaryTree(pre, vin)  
{  
 *if* (pre.length == 0 || vin.length == 0) {  
 *return null*;  
 };  
 *var* index = vin.indexOf(pre[0]);  
 *var* left = vin.slice(0, index);  
 *var* right = vin.slice(index + 1);  
 *var* node = *new* TreeNode(vin[index]);  
 node.left = reConstructBinaryTree(pre.slice

(1, index + 1), left);  
 node.right = reConstructBinaryTree(pre.slice

(index + 1), right);  
 *return* node;  
}

# 二叉树翻转

*var* invertTree = *function*(root) {  
 *//判断当前树是否为 null  
 if*(root == *null*) *return* root;  
 *//左右子树结点交换  
 let* right = root.right;  
 *let* left = root.left;  
 root.right = left;  
 root.left = right;  
 *//分别对左右子树进行递归* invertTree(left);  
 invertTree(right);  
 *//返回树的根节点  
 return* root;  
};

# 判断是否为平衡二叉树

*function* IsBalanced(pRoot) { *//方法1：节点重复遍历了，影响效率了  
 if* (pRoot == *null*) *return true*;  
 *let* leftLen = TreeDepth(pRoot.left);  
 *let* rightLen = TreeDepth(pRoot.right);  
 *return* Math.abs(rightLen - leftLen) <= 1 && IsBalanced(pRoot.left) && IsBalanced(pRoot.right);  
 *//左右子树均平衡，且左右子树高度差不超过1*}

# 树中第k大的数

*function* kmax(root,k) {*//逆中序 右中左 第k大的数  
//也可以用递归中序放进一个数组  
 if* (!root) *return null*;  
 *var* stack = [];  
 *var* count=0;  
 *var* p = root;  
 *while* (stack.length > 0 || p) {  
 *if* (p) { *//当前非空，当前入栈，右移* stack.push(p);  
 p = p.right;  
 } *else* { *//栈弹出，并左移* p = stack.pop();  
 *if* (++count == k) {  
 *return* p.value;  
 }  
 p = p.left;  
 }  
 }  
}

# 二叉树深度

deep(){  
 *let* deepnum = 0  
 *return this*.\_deep(*this*.root)  
 }  
 \_deep(node){  
 *if* (node == *null*){  
 *return* 0  
 }  
 *let* left = *this*.\_deep(node.left),  
 right = *this*.\_deep(node.right);  
 *return* left>right? left+1:right+1  
 }

**最小深度**是从根节点到最近叶子节点的最短路径上的节点数量。

minDepth (root) {  
 *if* (root === *null*) {  
 *return* 0;  
 }  
 *if* (root.left === *null*) {  
 *return* minDepth(root.right)+1;  
 }  
 *if* (root.right === *null*) {  
 *return* minDepth(root.left)+1;  
  
 }  
 *return* Math.min(minDepth(root.left),

minDepth(root,right))+1;

# 打印所有路径

printAllPaths(node){  
 *this*.arr.push(node.key)  
 *if* (node.left===*null*&&node.right===*null*){  
 console.log(*this*.arr);  
 }*else* {  
 *if* (node.left){  
 *this*.printAllPaths(node.left)  
 }  
 *if* (node.right) {  
 *this*.printAllPaths(node.right)  
 }  
 }  
 *//这一步很关键  
 this*.arr.pop()  
}

# 寻找和为n的路径

*//寻找二叉树是否有一条路径 其和为指定值*pathSum(node,sum){  
 *if* (node === *null*){  
 *return false* }  
 *if* (node.left===*null*&&node.right===*null*){  
 *return* sum===node.key?*true*:*false* }  
 *return this*.pathSum(node.left,sum-node.key)||*this*.pathSum(node.right-node.key)  
}

# =======DP==========

# 最长上升子序列

*function* LISbyDP(arr) {  
 *if* (arr.length <= 1) {  
 *return* arr.length;  
 }  
 *let* maxLen = 0;  
 *let* fs = [];  
 fs[0] = 1;  
 *for* (*let* i = 1; i < arr.length; i++) {  
 fs[i] = 1; *//初始化为1 很关键 子序列最小也要包括自己  
 for* (*let* j = 0; j < i; j++) {  
 *if* (arr[j] < arr[i]) {  
 fs[i] = Math.max(fs[j] + 1, fs[i]); *//fs[i]表示到以arr[i]结尾的最长递增子序列的长度* }  
 }  
 }  
 maxLen = Math.max.apply(*null*, fs); *//计算出以每一个数结尾的最长递增子串数 求数组最大值  
 return* maxLen;  
}

# 最长公共子序列

*var* longestCommonSubsequence = *function* (text1, text2) {  
 *let* n = text1.length;  
 *let* m = text2.length;  
 *let* dp = Array(n + 1).fill(0).map(() => *new* Array(m + 1).fill(0)); *//多一个第0行 第0行和0列均为0 n\*m  
 for* (*let* i = 1; i <= n; i++) {  
 *for* (*let* j = 1; j <= m; j++) {  
 *if* (text1[i - 1] == text2[j - 1]) {  
 dp[i][j] = dp[i - 1][j - 1] + 1;  
 } *else* {  
 dp[i][j] = Math.max(dp[i][j - 1], dp[i - 1][j]);  
 }  
 }  
 }  
 *return* dp[n][m];  
};

# 连续子数组最大和

*var* maxSubArray = *function*(nums) {  
 *for*(*var* i = 1; i < nums.length; i++){  
 *if*(nums[i - 1] > 0){  
 nums[i] += nums[i - 1];  
 }  
 }  
 *return* Math.max(...nums)  
};  
*//DP法  
var* maxSubArray = *function* (nums) {  
 *if* (!nums.length) *return null  
 let* max = nums[0], record = nums[0];  
 *for* (*let* i = 1; i < nums.length; i++) {  
 record = Math.max(record + nums[i], nums[i]);  
 *if* (record > max) max = record;  
 }  
 *return* max  
};

# 最大回文子串

*var* longestPalindrome = *function*(s) {  
 *let* len = s.length;  
 *let* res = '';  
 *let* dp = *new* Array(len)  
 *for*(*let* i = 0; i<len;i++ ){  
 dp[i] = *new* Array(len).fill(0)  
 }  
  
 *for*(*let* start = len -1;start >= 0;start--){  
 *for*(*let* end = start;end < len;end++){

dp[start][end] = (s[start]===s[end]) &&

(end - start < 2 || dp[start+1][end-1]);

*if*(dp[start][end] && (end - start +1) >

res.length){  
 res = s.substring(start,end+1);  
 }  
 }  
 }  
 *return* res  
};

# =======排序=====

# 快速排序

*function* swift(arr) {  
 *if* (arr.length <= 1) {  
 *return* arr;  
 }  
 *var* middle = parseInt(arr.length / 2);  
 *var* arrleft = [],  
 arrright = [];  
 *for* (*let* i = 0; i < arr.length; i++) {  
 *if* (i == middle) {  
 *continue*;  
 }  
 arr[i] < arr[middle] ?

arrleft.push(arr[i]) : arrright.push(arr[i])  
 }  
 *return* swift(arrleft).concat

(arr[middle], swift(arrright))  
}

# 二分排序

*function* binarySort(array){  
 *for*(*let* i = 0;i<array.length;i++){  
 *let* temp = array[i];*//待插入到前面有序序列的值  
 let* left = 0;*//有序序列的左侧  
 let* right = i-1;*//有序序列的右侧  
 let* middle = 0;*//有序序列的中间  
 while*(left <= right){  
 middle = (left + right)/2;*//赋值  
 if*(temp<array[middle]){  
 right = middle-1;  
 }*else*{  
 left = middle + 1;  
 }  
 }  
 *for*(*let* j = i-1;j>=left;j--){  
 *//从i-1到left依次向后移动一位,等待temp值插入* array[j+1] = array[j];  
 }  
 *if*(left != i ){  
 array[left] = temp;  
 }  
 }  
 *return* array;  
}

# 归并排序

*function* Merger(a, b){  
 *var* n = a && a.length;  
 *var* m = b && b.length;  
 *var* c = [];  
 *var* i = 0, j = 0;  
  
 *while* (i < n && j < m)  
 {  
 *if* (a[i] < b[j])  
 c.push(a[i++]);  
 *else* c.push(b[j++]);  
 }  
 *while* (i < n)  
 c.push(a[i++]);  
 *while* (j < m)  
 c.push(b[j++]);  
 *return* c;  
}  
*function* merge\_sort(arr){  
 *if*(arr.length == 1)  
 *return* arr  
 *var* mid = Math.floor(arr.length/2)  
 *var* left = arr.slice(0,mid)  
 *var* right = arr.slice(mid)  
 *return* Merger(merge\_sort(left),merge\_sort(right)); *//合并左右部分*}

# ======数组=======

# 合并两个有序数组

*var* merge = *function*(nums1, m, nums2, n) {  
 *//len1 len2 表示当前位于数组哪个下标  
 let* len1 = m - 1;  
 *let* len2 = n - 1;  
 *let* len = m + n - 1;  
 *while* (len1 >= 0 && len2 >=0) {  
 *// 注意--符号在后面，表示先进行计算再减1，这种缩写缩短了代码* nums1[len] = nums1[len1] > nums2[len2] ? nums1[len1--] : nums2[len2--];  
 len --;  
  
 }  
 *// // 表示将nums2数组从下标0位置开始，拷贝到nums1数组中，从下标0位置开始，长度为len2+1* nums1.splice(0,len2+1,...nums2.slice(0,len2+1))  
 console.log(nums1)  
}

# 旋转数组

*function* rotate(arr,k) {  
 *if* (k<=0||k%arr.length==0){  
 *return* arr  
 }  
 *if* (k>arr.length){  
 k=k%arr.length  
 }  
 *var* newarr = arr.slice(1);  
 newarr.push(arr[0]);  
 k--;  
 *return* rotate(newarr,k)  
}

# 全排列

*function* fullpermutate(str) {  
 *var* result = [];  
 *if* (str.length > 1) {  
 *//遍历每一项  
 for* (*var* m = 0; m < str.length; m++) {  
 *//拿到当前的元素  
 var* left = str[m];  
 *//除当前元素的其他元素组合  
 var* rest = str.slice(0, m) +

str.slice(m + 1, str.length);  
 *//上一次递归返回的全排列  
 var* preResult = fullpermutate(rest);  
 *//组合在一起  
 for* (*var* i = 0; i < preResult.length; i++){  
 *var* tmp = left + preResult[i]  
 result.push(tmp);  
 }  
 }  
 } *else if* (str.length == 1) {  
 result.push(str);  
 }  
 *return* result;  
}

# 数组扁平化 递归

*function* flattenMd() {  
 *let* result = []  
 *return function* flatten(arr) {*//闭包* arr.forEach(item => {  
 *if* (Array.isArray(item)) {  
 flatten(item)  
 } *else* {  
 result.push(item)  
 }  
 })  
 *return* result  
 }  
}

非递归

*function* fn(arr){  
 *let* res = [];  
 *let* stack = [...arr];  
 *while*(stack.length!==0){  
 *let* item = stack.shift();  
 *if*(Array.isArray(item)){  
 stack.unshift(...item)  
 }*else*{  
 *if*(item!==*null*){  
 res.push(item)  
 }  
 }  
 }  
 *return* res;  
}

# =======工程上的题

# hex转rgb

*let* HEXtoRGB = *function* (str) {  
 *var* col = str.slice(1);  
 *let* copyStr = '';  
 *if* (/^[0-9A-Fa-f]{6}$/.test(col) || /^[0-9A-Fa-f]{3}$/.test(col)) {  
 *if* (col.length === 3) {  
 *for* (*let* i = 0; i < col.length; i++) {  
 copyStr += col.charAt(i)+col.charAt(i)  
 }  
 }  
 *return* `rgb(${parseInt(copyStr.substring(0, 2), 16)},${parseInt(copyStr.substring(2, 4), 16)},${parseInt(copyStr.substring(4, 6), 16)})`;  
 } *else* {   
 *return* "rgb(0,0,0)"   
 };  
};

*let* RGBtoHEX = *function* (str) {  
 *var* col = str.toUpperCase();  
 col = col.replace(/^RGB\((\d{1,3}?)\,(\d{1,3}?)\,(\d{1,3}?)\)$/, *function* (core, $1, $2, $3) {  
 console.log($1)  
 *var* a = parseInt($1), b = parseInt($2), c = parseInt($3), cache = "";  
 (a < 256 && b < 256 && c < 256) ? cache = '#' + a.toString(16) + b.toString(16) + c.toString(16) : cache = "#000";  
 *return* cache;  
 })  
 *return* col;  
}

# 原生实现call apply

*function* mycall(obj=window) {  
 *let* arr = []  
 *for*(*let* i=1; i<arguments.length; i++){  
 arr.push(arguments[i]);  
 }  
 *if* (obj === *null*){  
 *this* (...arr);  
 *return* }  
 obj.fn = *this*;  
  
 obj.fn(...arr);  
 *delete* obj.fn;  
}  
  
Function.prototype.mycall = mycall;  
*function* fn(a,b) {  
 console.log(a+b)  
};

# Promise.all race

Promise.all\_ = *function*(promises) {  
 *return new* Promise((resolve, reject) => {  
 *// Array.from()可将可迭代对象转化为数组* promises = Array.from(promises);  
 *if*(promises.length===0) {  
 resolve([]);  
 } *else* {  
 *let* result = [];  
 *let* index = 0;  
 *for*(*let* i=0; i<promises.length; i++) {  
 *// 考虑到promise[i]可能是thenable对象也可能是普通值* Promise.resolve(promises[i]).then(data => {  
 result[i] = data;  
 *if*(++index===promises.length) {  
 *// 所有的promise状态都是fulfilled，promise.all返回的实例才变成fulfilled状态* resolve(result);  
 }  
 }, err => {  
 reject(err);  
 *return*;  
 })  
 }  
 }  
 })  
}

**race**

Promise.ra\_ce = *function*(promises) {  
 promises = Array.from(promises);  
 *return new* Promise((resolve, reject) => {  
 *if*(promises.length===0) {  
 *return*;  
 } *else* {  
 *for*(*let* i=0; i<promises.length; i++) {  
 Promise.resolve(promises[i]).then(data => {  
 resolve(data);  
 *return*;  
 }, err => {  
 reject(err);  
 *return*;  
 })  
 }  
 }  
 })  
}

# 高阶组件

*const* simpleHoc = WrappedComponent => {  
 console.log('simpleHoc');  
 *return class extends* Component {  
 render() {  
 *return* <WrappedComponent {...*this*.props}/>  
 }  
 }  
}  
*export default* simpleHoc;

# 函数柯理化

*function* curry(fn,length) {  
 *return function* (...args) {  
 *if* (args.length>=length){  
 *return* fn(...args)  
 } *else* {  
 *return* curry(fn.bind(*null*,...args),length-args.length)  
 }  
 }  
}

# 函数防抖

*function* debounceAll(fn,delay,now) {  
 *let* time = *null*;  
 *return function* (...args) {  
 *if* (now){  
 *if* (!time){  
 fn.call(*this*,...args)  
 }  
  
 clearTimeout(time);  
 time = setTimeout(*function* () {  
 time = *null* },delay)  
  
  
 }  
 *else* {  
 clearTimeout(time);  
 time = setTimeout(*function* () {  
 fn.call(*this*,...args)  
 },delay)  
 }  
 }  
}

节流

*function* throttle(fn,delay) {  
*let* flag = *false*;*//一开始 没动  
 return function* (...args) {  
 *let* self = *this*;  
 *if* (flag){*//如果正在动就return  
 return* }*else*{  
 *//如果没动了 则可以开始动了* flag = *true*;  
 setTimeout(*function* () {  
 *// 绑定fn自己的this 和 参数，是为了让 throttle 函数最终返回的函数 this 指向不变以及依旧能接受到 e 参数。* fn.call(self,...args);  
 *//执行完 把标志置位false 表示已静止* flag = *false*;  
 },delay);  
 }  
 }  
 }  
  
 *function* throttleNow(fn,delay) {  
 *let* previous = 0;  
 *return function* (...args) {  
 *let* self = *this*;  
 *let* time = *new* Date();  
 *if* (time - previous >= delay){  
 console.log(time - previous)  
 fn.call(self,...args);  
 previous = time  
 } *else* {  
 *return* }  
 }  
 }

# 并发控制

*class* Scheduler {  
 constructor(limit){  
 *this*.limit = limit;  
 *this*.list = [];  
 *this*.workingNum = 0;  
 }  
 add(promiseCreator){  
 *this*.list.push(promiseCreator)  
 };  
 start(){  
 *for*(*let* i = 0; i<*this*.limit;i++ ){  
 *this*.doNext()  
 }  
 };  
 doNext(){  
 *if* (*this*.list.length&&*this*.workingNum <*this*.limit){  
 *this*.workingNum++;  
 *//从头取出 取出来的每一个都是一个函数  
 this*.list.shift()().then(()=>{  
 *//执行完成之后 把状态进行更改；然后执行下一个函数  
 this*.workingNum--;  
 *this*.doNext();  
 })  
 }  
 }  
  
}  
*let* scheduler = *new* Scheduler(2);  
*//延迟执行  
const* timeout = time=> *new* Promise(resolve => setTimeout(resolve,time));  
*const* addTask = (time,order)=>{  
 *//给调度器添加任务 每个任务都是一个函数 返回值是promise对象；* scheduler.add(()=>timeout(time).then(()=>console.log(order)))  
}  
addTask(1000,1)  
addTask(500,2)  
addTask(300,3)  
addTask(400,4)  
scheduler.start()

# 获取给定k v 的dom节点

function findAttr(property, value) {

//获得所有元素  
 let elements = document.getElementsByTagName('\*'),  
 arr = [];  
  
 // 类数组的迭代 不能直接使用forEach  
  
 //方法一  
 // [].forEach.call(elements,(item)=>console.log(item))  
  
 //方法二！！  
 elements = Array.from(elements); //把类数组转化为数组  
 elements.forEach((item, index) => {  
 //存储当前元素property对应的属性值  
 let itemValue = item.getAttribute(property);  
// className 可以有多个名字 所以要特殊判断   
 // 正则\b表示 字符边界 \bvalue\b 表示value要是单独的一个单词 不能是 itemvalue这样包含在其他单词里面  
 if (property === 'class') {  
 if (new RegExp('\\b' + value + '\\b').test(itemValue)  
 ) {  
 arr.push(item)  
 }  
 } else {  
 if (itemValue === value) {  
 //获取的值的传递的值相等  
 arr.push(item);  
 }  
 }  
 });  
  
 //方法三 => for循环遍历  
 return arr;  
 }

# 模拟instanceof

*function* instanceofObj(a, b) {  
 *// 模拟 a instanceof b  
 let* prototypeB = b.prototype;  
 *let* protoA = a.\_\_proto\_\_;  
 *let* state = *false*;  
 *while* (*true*) {  
 *if* (protoA == *null*) { *// 可能是 undefined* state = *false*;  
 *break*;  
 }  
 *if* (prototypeB === protoA) {  
 state = *true*;  
 *break*;  
 }  
 protoA = protoA.\_\_proto\_\_;  
 }  
 *return* state;  
}  
console.log(instanceofObj([], Array));  
instanceofObj([], Array); *//true*

# 在window全局添加事件

*var* \_wr = *function*(type) {  
 *var* orig = history[type];  
 *return function*() {  
 *var* rv = orig.apply(*this*, arguments);  
 *var* e = *new* Event(type);  
 e.arguments = arguments;  
 window.dispatchEvent(e);  
 *return* rv;  
 };  
};  
history.pushState = \_wr('pushState');  
history.replaceState = \_wr('replaceState');

# 实现render函数

*function* render(vDom) {  
 *//children是一个数组  
 const* {type,props,children} = vDom;  
 *const* el = document.createElement(type);  
 *for*(*let* key *in* props){  
 setAttr('el',key,props[key])  
 }  
 children.forEach(child=>{  
 *//判断child否是node节点类型对象 因为还可能是文本节点  
 if* (child *instanceof* Element){  
 child = render(child)  
 } *else*{  
 child = document.createTextNode(child)  
 }  
 el.appendChild(child)  
 })  
 *return* el;  
  
  
}  
*function* setAttr(node,prop,value){  
 *switch*(prop){  
 *case* 'value':  
 *if* (node.tagName==='INPUT' || node.tagName==='TEXTAREA'){  
 node.value = value  
 } *else* {  
 node.setAttribute(prop,value)  
 }  
 *break*;  
 *case* 'style':  
 *//行内样式* node.style.cssText = value;  
 *break*;  
 *default*:  
 node.setAttribute(prop,value)  
 *break* }  
}