手写js

数组实现reduce、map

map

请补全JavaScript代码，要求实现Array.map函数的功能且该新函数命名为"_map"。输入：

```
[1,2]._map(i => i * 2)
```

复制

输出：

```
[2,4]
```

```
        Array.prototype._map = function(Fn) {
            if(!(Fn instanceof Function)){
                return false
            }
            const result = []
            for(let i = 0;i < this.length;i++){
                result.push(Fn(this[i]))
            }
            return result
}
```

reduce

```
// 填写JavaScript
        Array.prototype._reduce = function(fn){
            if(!(fn instanceof Function)){
                return
            }
            let sum = 0
            for(let i = 0;i < this.length - 1;i++){
                sum += fn(this[i],this[i+1])
            }
            return sum
        }
```

发布订阅模式

思路:

1. 创建一个对象
2. on，在里面查找是否有该函数，没有就添加，有的话就push进去
3. emit就是找到该事件来进行执行

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset=utf-8>
    </head>
    <body>

        <script type="text/javascript">
            class EventEmitter {
                // 补全代码
                constructor(){
                    this.events = {}
                }
```

```
                on(event,fn){
                    if(!this.events[event]){
                        this.events[event] = [fn]
                    }else{
                        this.events[event].push(fn)
                    }
                }
               once(event, fn) {
                const onceWrapper = (...args) => {
                    fn(...args);
                    this.off(event, onceWrapper);
                };
                this.on(event, onceWrapper);
            }

                emit(event){
                    if(this.events[event]){
                        this.events[event].forEach(item=>                        {
                            item()
                        })
                    }
                }
                off(event, fn) {
                    if (!this.events[event]) return;
        this.events[event]=this.events[event].filter(item => item !== fn);
                }
            }

        </script>
    </body>
</html>
```

观察者模式

```
const observe = new Set()
const queneJob = (fn)=>observe.add(fn)

const ProxyObj = (obj) => new Proxy(obj, { set });
function set(target,,key,val,reseiver){
    const result = Reflect.set(target,key,val,receiver)
    //观察对象改变时，执行这些函数
    observe.forEach(item=>{
        item()
    })
    return result

}
```

Object.freeze

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset=utf-8>
    </head>
    <body>

        <script type="text/javascript">
            const _objectFreeze = object => {
                // 补全代码
                //获取属性名字
                if(typeof object !== 'object' || object == null){
                    throw new TypeError(`the ${object} is not a object`)
                }
                const keys = Object.getOwnPropertyNames(object)
                const symbols = Object.getOwnPropertySymbols(object)
                ;[...keys,...symbols].forEach(item=>{
                    Object.defineProperty(object,item,{
                        configurable:false,
                        writable:false
                    })
                })
                //不能拓展
                Object.preventExtensions(object)
            }
        </script>
    </body>
</html>
```

jsonp

```
const newscript = document.createElement('script')
newscript.src = 'http://www.adb.com?callback=fn'
document.body.appendChild(newscript)
```

将传入的对象作为原型

Object.create

1. 创建一个构造函数
2. 将这个对象赋值给这个构造函数原型
3. 返回这个构造函数

```
function create(obj){
    function F(){}
    F.prototype = obj
    return new F()
}
```

typeOf

```
function typeOf(a){
    let res = Object.prototype.toString.call(a).split(" ")[1]
    const result = res.substring(0,res.length-1).toLowerCase()
    return result
}
```

instanceOf

找原型呗，找不到就往上找，如果真没有就是没有了

```
function instanceOf(left,right){
    let prop = Object.getPrototypeOf(left),
        prototype = right.prototype
    while(true){
        if(!prop)return false
        if(prop === prototype)return true
        prop = Object.getPrototypeOf(prop)
    }
}
```

手写new操作符

调用`new`的过程中会发生四个事情

1. 创建一个新的对象
2. 接着改变新的对象的原型
3. 新建对象要能访问构造函数属性
4. 返回值

```
function Factory(...args){
    const obj = new Object()
    let constructor = args[0]
    //将构造函数上面的共享的属性方法赋值给新创建的对象
    obj.__proto__ = constructor.prototype
    //将构造函数的实例上的方法和属性赋值给对象，指定this
    cosnt ret = constructor.apply(obj,...args)
    return typeof ret === 'object'?ret:obj
}
```

call、apply、bind

call

实现思路:

我们只要按照下面的步骤来实现 call() 即可：

1. 改造 foo 对象，给它添加 bar 方法
2. 执行 bar 方法
3. 复原 foo 对象，把添加的 bar 方法删掉

1. 改变this(通过添加一个属性来保存这个this，之后删除这个属性)
2. 调用函数

```
Function.prototype.call2 = function(ctx,...args){
    ctx = ctx || window
    const fnSymbol = new Symbol()
    //改造ctx方法
    ctx[fnSymbol] = this
    //执行ctx方法
    const result = ctx[fnSymbol](..args)
    //删除这个方法
    delete ctx[fnSymbol]
    return result
}
```

apply：传入的参数为数组

```
Function.prototype.apply2 = function(ctx,arr){
    ctx = ctx || window
    ctx.fn = this
    const reuslt = arr ? ctx.fn(..arr):ctx.fn()
    delete ctx.fn
    return result
}
```

bind：不立即执行函数，参数为数组

`bind()` 和 `call()` 的区别有两点：

- 它返回的是一个新的函数
- 如果使用 `new` 运算符调用，则忽略传入的 `this` 值

```
```

防抖和节流

防抖

触发**频率较高**的事件中，**消耗性能操作，连续操作之后只有最后一次生效。**

连续事件**停止**触发,**一段时间内不会再触发**，就执行业务代码。

核心步骤:

1. 开启定时器，保存定时器id
2. 清除上一次的定时器
3. 返回防抖的函数
4. 原函数的this可以正常使用
5. 原函数的参数可以正常使用

```
function debounce(fn,wait){
    let timeId
    return function(...args){
        let _this = this
        clearTimeout(timeId)
        timeId = setTimeout(()=>{
            fn.apply(_this,args)
        },wait)
    }
}
```

节流

使用场景:在触发频率高的事件中，执行消耗性能操作，**连续触发，单位时间只有一次生效。**当触发一个事件的时候就进行延迟，之后执行事件。

核心步骤:

1. 开启定时器，保存id
2. 判断是否打开了定时器，打开了后面的不执行
3. 定时器执行的时候，id设置为空
4. 返回一个节流的函数
5. 原函数的this可以正常使用
6. 原函数的参数可以正常使用

```
function throttle(fn,wait=0){
    let timeId
    return function(...args){
        let _this = this
        if(timeId){
            return
        }
        timeId = setTimeout(()=>{
            fn.apply(_this,args)
            timeId = undefined
        },wait)
    }
}
```

数组扁平化

ES5就是使用递归

```
function flatten(arr){
    const result = []
    for(let i = 0;i < arr.length;i++){
        if(Array.isArray(arr[i])){
            return flatten(arr[i])
        }else{
```

```
            return [arr[i]]
        }
    }
    return [].concat(...result)
}



function flatten(arr) {
    const result = [];
    for (let i = 0; i < arr.length; i++) {
        if (Array.isArray(arr[i])) {
            result.push(...flatten(arr[i]));
        } else {
            result.push(arr[i]);
        }
    }
    return result;
}
```

ES6方法

```
function flatten(arr){
    let result
    while(arr.some(item=>Array.isArray(item))){
        result = [].concat(...arr)
    }
    return result
}
```

函数传递参数求和

题目

```
//要求实现的函数
const foo = function(...args){}
const f1 = foo(1,2,3) f1.getValue() //6
const f2 = foo(1)(2,3) f2.getValue()//6
const f3 = foo(1)(2)(3)(4) f3.getValue()//10

//实现的思路
/**
1. 函数的参数不确定(args)
2. 支持链式调用
3. 函数所有参数进行累加
**/
//方法一:添加额外属性
const foo = (...args)=>{
    if(!Array.isArray(foo.arr)){
        foo.arr = []
    }
```

```
    foo.arr.push(...args)
    return foo
}
Function.prototype.getValue = function(){
    return foo.arr.reduce((pre,cur)=>{
        return pre + cur
    },0)
}
```

深浅拷贝

深拷贝

1.通过json格式来进行深拷贝

```
let person = {age:18,sex:'男'}
let  b = JSON.parse(JSON.stringify(person))
```

缺点:因为json格式很严格的，如果遇到了函数的话就不会拷贝进去。

2.递归实现深拷贝

其实深拷贝可以拆分成 2 步，浅拷贝 + 递归，浅拷贝时判断属性值是否是对象，如果是对象就进行递归操作，两个一结合就实现了深拷贝。

```
//判断对象
function isObject(res){
    return typeof res === 'object' && res !== null
}
function cloneDeep(source,hash = new WeakMap){
    if(!isObject(source))return source
    if(hash.get(source))return source

    const target = Array.isArray(source)?[]:{}
    hash.set(source,target)
    for(let key in source){
        //查看是否为对象
   if(Object.prototype.hasOwnPropertype.call(source,key))   {
            if(isObject(source[key])){
                target[key] = cloneDeep(source[key],hash)
            }else{
                target[key] = source[key]
            }
        }
    }
    return target
}

function cloneDeep(obj,hash = new WeakMap()){
    if(!(typeof obj === 'object' && obj !== null)){
        return obj
```

```
    }
    if(hash.get(obj))return hash.get(obj)
    const target = Array.isArray(obj)?[]:{}
    hash.set(obj,target)
    for(let key in obj){
        if(Object.prototype.hasOwnProperty.call(obj,key))           {
            //判断是不是数组或者对象是的话就及逆行递归，不是的话就进行设置。
            if(typeof obj[key] === 'object' && obj[key] !== null){
                target[key] = cloneDeep(obj[key],hash)
            }else{
                target[key] = obj[key]
            }

        }
    }
    return target
}
```

浅拷贝:使用[...],Object.assign()

promise

promise.all

Promise.all 的完成体应该符合以下特征：

1. 输入为 Iterator 类型的参数，可以是 Array，Map， Set，String，可能也得包括魔改的 Iterator（Symbol.iterator）之类
2. 若输入的可迭代数据里不是 Promise，则也需要原样输出
3. 返回一个 Promise 实例，可以调用 then 和 catch 方法
4. 输出在 then 里体现为保持原顺序的数组
5. 输出在 catch 体现为最早的 reject 返回值
6. 空 Iterator，resolve 返回空数组

```
function promiseAll(_promises){
    return new Promise((resolve,reject)=>{
        const promises = Array.from(_promises);
        const length = promises.length;
        const result = []
        let count = 0
        for(let i = 0;i < length;i++){
            //开始解决期约，成功的话就将结果加入进去，没有成功就reject
            Promise.resolve(promises[i]).then(val=>{
                result[i] = val
                if(++count === length){
                    //说明已经处理完了
                    resolve(result)
                }
            }).catch(e=>{
               reject(e)
```

```
            })

        }
    })
}
```

promise.race

```
function PromiseRace(promiseArr){
    return new Promise(resolve,reject){
        for(let promise of promiseArr){
            Promise.resolve(promise).then(resolve,reject)
        }
    }
}
```

promise封装xhr

```
function load(url) {
    return new Promise(function (resolve, reject) {
        const request = new XMLHttpRequest();
        request.onreadystatechange = function () {
            if (this.readyState === 4 && this.status == 200) {
                resolve(this.response);
            } else {
                reject(this.status);
            }
        };
        request.open("GET", url, true);
        request.send();
    });
}

// test
load("testUrl")
    .then((response) => {
        const result = JSON.parse(response);
        // ...
    })
    .catch((error) => {
        // ...
    });
```

实现Proxy

```
function MyProxy(obj,handler){
    let _target = deepClone(obj)
    Object.keys(_target).forEach(key=>{
        Object.defineProperty(_target,key,{
```

```
            get:()=>handler.get && handler.get(obj,key)
            set:(newVal)=>handler.set&&handler.set(obj,key,newVal)
        })
    })
    return _target
}
let person = {
  name: 'jack',
  city: 'Beijing',
};

let proxy = new MyProxy(person, {
  get: (target, propKey) => target[propKey],
  set: (target, propKey, value) => {
    target[propKey] = value;
  },
});
```

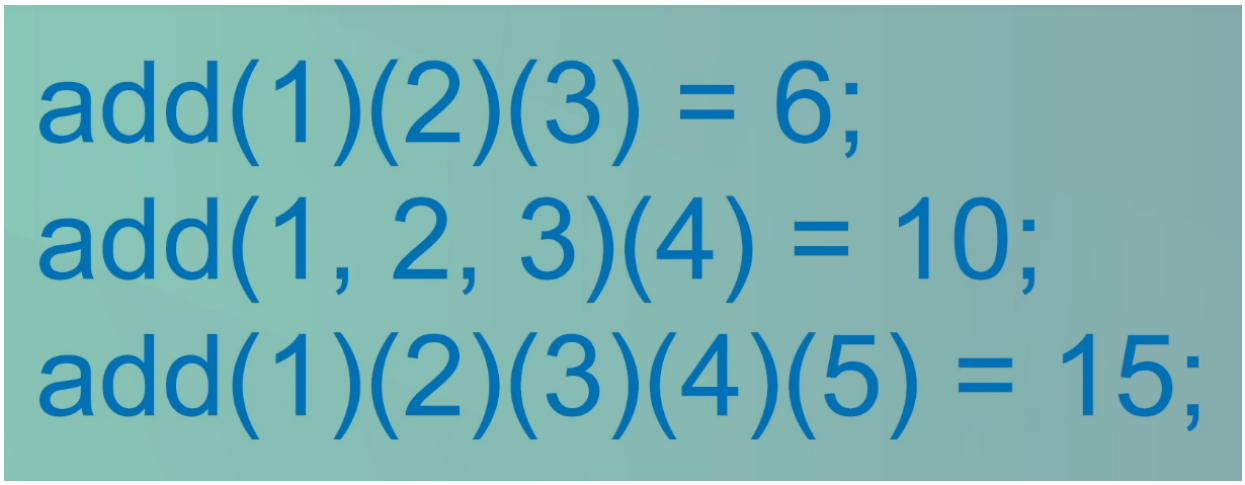
setTimeOut实现setInterval

```
function myInterval(fn,time){
    let context = this
    setTimeOut(()=>{
        fn.call(context)
        myInterval(fn,time)
    },time)
}
```

函数柯里化

为什么出现:实现柯里化，因为日常工作中会出现很多重复的参数，可以使用函数柯里化来进行复用

面试题




```
function curry(func){
    const args = []
    return function result(...args){
```

```
        if(args.length === 0){
            return func(...args)
        }else{
            args.push(...args)
            return result
        }
    }
}
function sum(...args){
    return args.reduce((prev,next)=>{
        return prev + next
    },0)
}


// 使用示例
const curriedSum = curry(sum);

console.log(curriedSum(1)(2)(3)); // 6
console.log(curriedSum(1, 2)(3)); // 6
console.log(curriedSum(1, 2, 3)); // 6
//实现的思路
/**
1. 函数的参数不确定(args)
2. 支持链式调用
3. 函数所有参数进行累加
**/
//方法一:添加额外属性
const foo = (...args)=>{
    if(!Array.isArray(foo.arr)){
        foo.arr = []
    }
    foo.arr.push(...args)
    return foo
}
Function.prototype.getValue = function(){
    return foo.arr.reduce((pre,cur)=>{
        return pre + cur
    },0)
}
```

compose实现

```
/**

 * 接收若干个函数作为参数，每个函数执行后的输出作为下一个函数的输入。
 * 执行方向是自右向左的，初始函数的参数在最右边。
 * @param  {...any} fns
 * @returns
   */
function compose(...fns){
```

```
    return function(x){
        return fns.reverse().reduce((args,fn)=>{
            return fn(args)
        },x)
    }
}

const add = x => x + 1;
const multiply = x => x * 2;
const minus = x => x - 1;

console.log(compose(minus, multiply, add)(1)) // 3
```

实现before

```
/**
 * 传入任意一个函数，只能调用指定的次数
 * @param {*} count 调用次数
 * @param {*} func 传入函数
 * @returns
 */
function before(count,func){
    let temp = count
    return function(...args){
        if(temp > 0){
            temp--
            const args = [...args]
            func.apply(this,args)
        }
    }
}

//例子
const log = a => console.log(a);

const log3 = before(3, log);

log3(2);
log3(1);
log3(3);
```

数组扁平化处理

```
const flatten = (arr)=>{
    let result = []
    for(let i = 0;i < arr.length;i++){
        if(Array.isArray(arr[i])){
            result = result.concat(flatten(arr[i]))
        }else{
            result.push(arr[i])
        }
    }
    return result
}
//例子
console.log(flatten([1,[1,2,[2,4]],3,5]));  // [1, 1, 2, 2, 4, 3, 5]
```

数组去重

1. 使用set来去重

```
const array = [1,1,2,3,4,3,4]
function unique(arr){
    const set = new Set(array)
    const result = [...set]
    return result
}
console.log(unique(array))
```

2. 普通的循环

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

手写监听数组

手写迭代器

```
class MyIterator{
    constructor(params){
        this.index = 0
        this.value = params
    }
    [Symbol.iterator](){
        return this
    }
    next(){
        return {
            value:this.value[this.index++]
            done:this.index > this.value.length?true:false
        }
    }
}
```

Promise

使用Promise实现隔一秒输出1，2，3

```
const arr = [1,2,3]
arr.reduce((pre,current)=>{
    return p.then(resolve=>{
        return new Promise((r)=>{
            setTimeout(()=>{
                r(current)
            }, 1000)
        })
    })
},Promise.resolve())
```

使用Promise实现红绿灯交替重复亮

红灯3秒亮一次，黄灯2秒亮一次，绿灯1秒亮一次；如何让三个灯不断交替重复亮灯？（用Promise实现）
三个亮灯函数已经存在：

```
function red() {
    console.log('red');
}
function green() {
    console.log('green');
}
function yellow() {
    console.log('yellow');
```

```
}
function light(timer,cb){
    return new Promise((resolve)=>{
        setTimeout(()=>{
            cb()
            resolve()
        },timer)
    })
}
const step = function(){
    Promise.resolve().then(()=>{
        return light(3000,red)
    }).then(()=>{
        return light(2000,green)
    }).then(()=>{
        return light(1000,yellow)
    }).then(()=>{
        return step()
    })
}
step()



function light(timer,fn){
    return new Promise(resolve=>{
        setTimeout(()=>{
            fn()
            resolve()
        },timer)
    })
}
const step = function(){
    Promise.resolve().then(()=>{
        return light(3000,red)
    }).then(()=>{
        return light(2000,yellow)
    }).then(()=>{
        return light(1000,green)
    }).then(()=>{
        return step()
    })
}
step()
```

实现mergePromise函数

实现mergePromise函数，把传进去的数组按顺序先后执行，并且把返回的数据先后放到数组data中。

```
const time = (timer) => {
  return new Promise(resolve => {
    setTimeout(() => {
```

```
      resolve()
    }, timer)
  })
}
const ajax1 = () => time(2000).then(() => {
  console.log(1);
  return 1
})
const ajax2 = () => time(1000).then(() => {
  console.log(2);
  return 2
})
const ajax3 = () => time(1000).then(() => {
  console.log(3);
  return 3
})

function mergePromise (ajaxArray) {
  // 在这里写代码
    const result = []
    const promise = Promise.resolve()
    ajaxArray.forEach(item=>{
        //这个promise.then(item)可以执行函数 + 获取返回值
        promise =  promise.then(item).then(res=>{
            result.push(res)
            return result
        })
    })
    return promise
}

mergePromise([ajax1, ajax2, ajax3]).then(data => {
  console.log("done");
  console.log(data); // data 为 [1, 2, 3]
});

function mergePromise (ajaxArray) {
  // 在这里写代码
    const result = []
    const promise = Promise.resolve()
    ajaxArray.forEach(item=>{
        promise = promise.then(item).then(res=>{
            result.push(res)
            return result
        })
    })
    return promise
}

// 要求分别输出
// 1
// 2
```

```
// 3
// done
// [1, 2, 3]
```

封装一个异步加载图片的方法

```
function loadImage(url){
    return new Promise((resolve,reject)=>{
        const img = new Image()
        img.onload = function(){
            console.log("一张图片加载完成")
            resolve(img)
        }
        img.onerror = function(){
            reject(new Error("出错了"))
        }
        img.src = url
    })
}
```

限制异步请求的并发个数

既然题目的要求是保证每次并发请求的数量为3，那么我们可以先请求`urls`中的前面三个(下标为`0,1,2`)，并且请求的时候使用`Promise.race()`来同时请求，三个中有一个先完成了(例如下标为`1`的图片)，我们就把这个当前数组中已经完成的那一项(第`1`项)换成还没有请求的那一项(`urls`中下标为`3`)

直到`urls`已经遍历完了，然后将最后三个没有完成的请求(也就是状态没有改变的`Promise`)用`Promise.all()`来加载它们。

```
function limitLoad(urls, handler, limit) {
  let sequence = [].concat(urls); // 复制urls
  // 这一步是为了初始化 promises 这个"容器"
  let promises = sequence.splice(0, limit).map((url, index) => {
    return handler(url).then(() => {
      // 返回下标是为了知道数组中是哪一项最先完成
      return index;
    });
  });
  // 注意这里要将整个变量过程返回，这样得到的就是一个Promise，可以在外面链式调用
  return sequence
    .reduce((pCollect, url) => {
      return pCollect
        .then(() => {
          return Promise.race(promises); // 返回已经完成的下标
        })
        .then(fastestIndex => { // 获取到已经完成的下标
            // 将"容器"内已经完成的那一项替换
          promises[fastestIndex] = handler(url).then(
            () => {
              return fastestIndex; // 要继续将这个下标返回，以便下一次变量
```

```
            }
          );
        })
        .catch(err => {
          console.error(err);
        });
    }, Promise.resolve()) // 初始化传入
    .then(() => { // 最后三个用.all来调用
      return Promise.all(promises);
    });
}
limitLoad(urls, loadImg, 3)
  .then(res => {
    console.log("图片全部加载完毕");
    console.log(res);
  })
  .catch(err => {
    console.error(err);
  });
```

继承

算法

哈希

两数之和

相当于在map这个里面存储了一个记录

```
function twoSum(nums: number[], target: number): number[] {
    const map = new Map()
    for(let i = 0;i < nums.length;i++){
        //寻找是不是有这个值
        const res = map.get(target - nums[i])
        if(res !== undefined){
            return [res,i]
        }
        map.set(nums[i],i)
    }
};
```

字母异位分组

思路:

key是经过排序后的str值，value就是和key一样的一组字母数组。

给你一个字符串数组，请你将 **字母异位词** 组合在一起。可以按任意顺序返回结果列表。

字母异位词 是由重新排列源单词的所有字母得到的一个新单词。

示例:

思路:就是进行map存储，将strs遍历得到的每一个进行排序，接着在map里面查是否有排序后的字符串的list，有的话就拿出来进行push一下当前的字符串，接着在map里面设置key，list。

```
输入: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
输出: [["bat"],["nat","tan"],["ate","eat","tea"]]
```

```
function groupAnagrams(strs: string[]): string[][] {
    const map = new Map()
    strs.forEach(str=>{
        const arr = Array.from(str)
        //进行排序
        arr.sort()
        const key = arr.toString()
        const list = map.get(key)?map.get(key):new Array()
        list.push(str)
        map.set(key,list)
    })
    return Array.from(map.values())
};
```

最长连续序列

1. 将数组元素存入 set 中
2. 遍历nums，如果 当前项 - 1 存在于 set ，说明当前项不是连续序列的起点，忽略，继续遍历
3. 如果当前项没有“左邻居”，它就是连续序列的起点，循环查看当前项连续的右邻居有多少个
4. 返回最长的连续次数

```
function longestConsecutive(nums: number[]): number {
    const set = new Set(nums)
    let max = 0
    for(let [key,value] of set.entries()){
        if(!set.has(value - 1)){
            //起点
            let count = 1
            let current = value
            //查看他的右邻居有哪些
```

```
            while(set.has(current + 1)){
                current++
                count++
            }
            max = Math.max(max,count)
        }
    }
    return max
};
```

双指针

移动零

给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

请注意，必须在不复制数组的情况下原地对数组进行操作。

```
/**
 * @param {number[]} nums
 * @return {void} Do not return anything, modify nums in-place instead.
 */
var moveZeroes = function(nums) {
    let l = 0,r = nums.length - 1
    while(l < r){
        if(nums[l] === 0){
            nums.splice(l,1)
            nums.push(0)
            r--
        }else{
            l++
        }
    }
};
```

盛最多的水的容器

给定一个长度为 `n` 的整数数组 `height` 。有 `n` 条垂线，第 `i` 条线的两个端点是 `(i, 0)` 和 `(i, height[i])` 。

找出其中的两条线，使得它们与 `x` 轴共同构成的容器可以容纳最多的水。

返回容器可以储存的最大水量。

```
/**
 * @param {number[]} height
 * @return {number}
 */
var maxArea = function(height) {
    let left = 0,right = height.length - 1
    let max = 0
```

```
    //左右指针寻找面积大的呗，大的就进行存储，让他进行循环一遍，找最大的。
    while(left <= right){
        const temp = Math.min(height[left],height[right])*(right - left)
        max = Math.max(temp,max)
        if(height[left] < height[right]){
            left++
        }else{
            right--
        }
    }
    return max
};
```

三数之和

给你一个整数数组 `nums` ，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足 `i != j`、`i != k` 且 `j != k` ，同时还满足 `nums[i] + nums[j] + nums[k] == 0` 。请你返回所有和为 `0` 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

```
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var threeSum = function(nums) {
    let result = []
    //排序
    nums.sort((a, b) => a - b)
    for(let i = 0;i < nums.length;i++){
        let one = nums[i], l = i + 1,r = nums.length - 1
        if(one > 0)return result
        if(i > 0&& nums[i] === nums[i - 1]){
            continue
        }
        while(l < r){
            let two = nums[l],three = nums[r]
            let sum = one + two + three
            if(sum > 0){
                r--
            }else if(sum < 0){
                l++
            }else{
                result.push([one,two,three])
                //去一下重
                while(l < r&& nums[l] === nums[l + 1]){
                    l++
                }
                while(l < r&&nums[r] === nums[r - 1]){
```

```
                    r--
                }
                r--
                l++
            }

        }
    }
    return result
};
```

滑动窗口

通用的算法套路

```
function huadong(s){
    let window
    let left = 0,right = 0
    while(right < s.length){
        const c = s[right]
        right++
        //进行窗口的一系列更新
        while(需要更新的条件){
            const d = s[left]
            left++
            //窗口内的一系列更新
        }
    }
}
```

无重复字符的最长子串

```
/**
 * @param {string} s
 * @return {number}
 */
var lengthOfLongestSubstring = function(s) {
    let i = 0
    let map = {}
    let res = 0
    for(let j = 0;j < s.length;j++){
        map[s[j]] = map[s[j]]+1|| 1
        while(map[s[j]] > 1){
            map[s[i]]--
            i++
        }
        res = Math.max(res,j - i + 1)
    }
    return res
};
```

找到字符串中所有的字母异位词

给定两个字符串 `s` 和 `p`，找到 `s` 中所有 `p` 的 **异位词** 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

异位词 指由相同字母重排列形成的字符串（包括相同的字符串）。

```
输入: s = "cbaebabacd", p = "abc"
输出: [0,6]
解释:
起始索引等于 0 的子串是 "cba", 它是 "abc" 的异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的异位词。
```

```
function findAnagrams(s: string, p: string): number[] {
    let need = new Map(),window = new Map()
    let left = 0,right = 0,valid = 0
    const res = []
    Array.from(p).forEach(item=>{
        need.get(item)?need.set(item,need.get(item) + 1):need.set(item,1)
    })
    while(right < s.length){
        const c = s[right]
        right++
        //进行窗口更新
        if(need.get(c)){
            window.get(c)?window.set(c,window.get(c) + 1):window.set(c,1)
            if(window.get(c) === need.get(c)){
                valid++
            }

        }
        while(right - left >= p.length){
            if(need.size === valid){
                res.push(left)
            }
            const d = s[left]
            left++
            if(need.get(d)){
                if(window.get(d) === need.get(d)){
                    valid--
                }
                window.set(d,window.get(d) - 1)
            }
        }
    }
    return res
};
```

子串

和为K的子数组

给你一个整数数组 `nums` 和一个整数 `k` ，请你统计并返回 *该数组中和为 `k` 的子数组的个数* 。

子数组是数组中元素的连续非空序列。

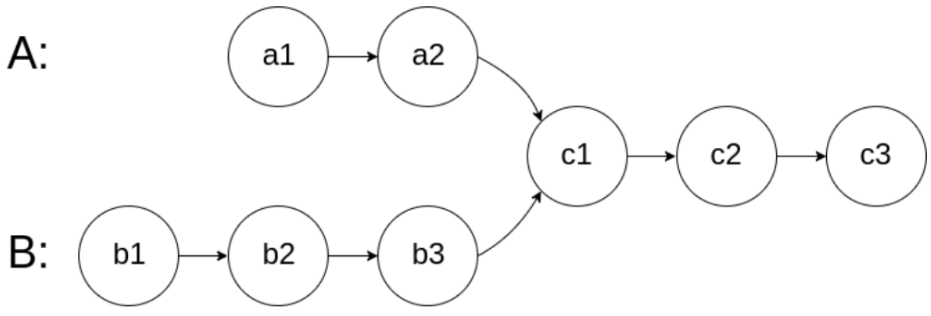
```
输入：nums = [1,1,1], k = 2
输出：2
```

思路：前缀和 (主要用于处理数组区间问题)+ hash表来做

```
function subarraySum(nums: number[], k: number): number {
    let pre = 0,count = 0,map = new Map()
    map.set(0,1)
    for(let i = 0;i < nums.length;i++){
        pre = pre + nums[i]
        const target = pre - k
        if(map.get(target)){
            count += map.get(target)
        }
        map.set(pre,map.get(pre)?map.get(pre)+1:1)
    }
    return count
};
```

链表

相交链表

> 图示两个链表在节点 `c1` 开始相交：
>
> 
>
>
> 题目数据 **保证** 整个链式结构中不存在环。
>
> **注意**，函数返回结果后，链表必须 **保持其原始结构** 。

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     val: number
 *     next: ListNode | null
 *     constructor(val?: number, next?: ListNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.next = (next===undefined ? null : next)
 *     }
 * }
 */

function getIntersectionNode(headA: ListNode | null, headB: ListNode | null):
ListNode | null {
    let A = headA,B = headB
    while(A!==B){
        A = A !== null?A.next :headB
        B = B !== null ? B.next :headA
    }
    return A
};
```

删除链表节点

给定单向链表的头指针和一个要删除的节点的值，定义一个函数删除该节点。返回删除后的链表的头节点。

1.此题对比原题有改动

2.题目保证链表中节点的值互不相同

3.该题只会输出返回的链表和结果做对比，所以若使用 C 或 C++ 语言，你不需要 free 或 delete 被删除的节点

示例1:

```
输入：
{2,5,1,9},5
复制
返回值：
{2,1,9}
复制
说明：
给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 2 -> 1 -> 9
```

```
/*class ListNode {
 *     val: number
 *     next: ListNode | null
 *     constructor(val?: number, next?: ListNode | null) {
```

```
 *         this.val = (val===undefined ? 0 : val)
 *         this.next = (next===undefined ? null : next)
 *     }
 * }
 */
/**
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 *
 * @param head ListNode类
 * @param val int整型
 * @return ListNode类
 */
export function deleteNode(head: ListNode, val: number): ListNode {
    if(!head)return null
    if(head.val === val)return head.next
    let node = head.next
    let prev = head
    while(node){
        if(node.val === val){
            //进行指针移动
            prev.next = node.next
           return head
        }
        prev = prev.next
        node = node.next
    }
    return null
}
```

链表中倒数最后k个节点

```
/*class ListNode {
 *     val: number
 *     next: ListNode | null
 *     constructor(val?: number, next?: ListNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.next = (next===undefined ? null : next)
 *     }
 * }
 */
/**
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 *
 * @param pHead ListNode类
 * @param k int整型
 * @return ListNode类
 */
export function FindKthToTail(pHead: ListNode, k: number): ListNode {
```

```
    if(!pHead)return null
    const length = getListLength(pHead);
    let node = pHead;
    let result;
    if (!length || k > length) {
        return null
    } else {
        for (let i = 0; i < length - k; i++) {
            node = node.next;
        }
        result = node;
        return result;
    }
}
//获取长度
function getListLength(pHead: ListNode): number {
    let node = pHead;
    let length = 0;
    while (node) {
        node = node.next;
        length++;
    }
    return length;
}
```

链表中换的入口节点

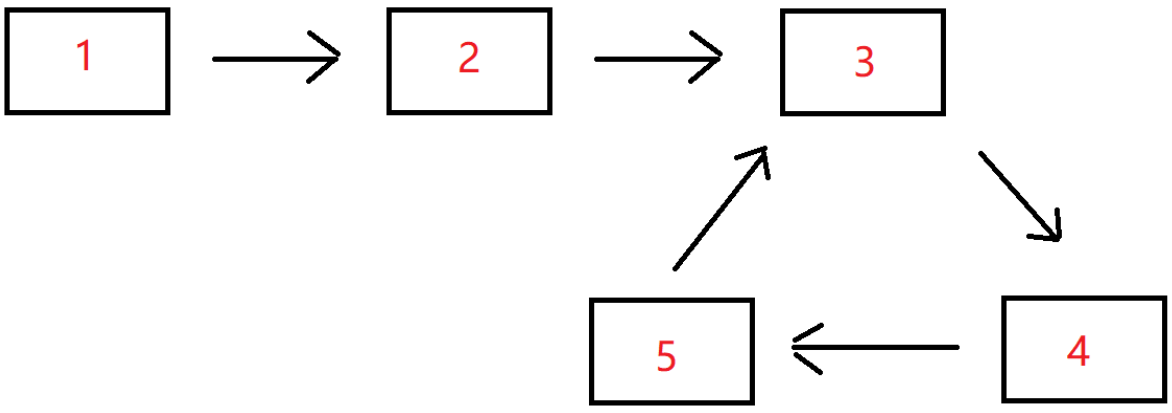
描述

给一个长度为n链表，若其中包含环，请找出该链表的环的入口结点，否则，返回null。

数据范围： n≤10000n≤10000，1<=结点值<=100001<=结点值<=10000

要求：空间复杂度 O(1)O(1)，时间复杂度 O(n)O(n)

例如，输入{1,2},{3,4,5}时，对应的环形链表如下图所示：



可以看到环的入口结点的结点值为3，所以返回结点值为3的结点。

输入描述：

输入分为2段，第一段是入环前的链表部分，第二段是链表环的部分，后台会根据第二段是否为空将这两段组装成一个无环或者有环单链表

返回值描述：

返回链表的环的入口结点即可，我们后台程序会打印这个结点对应的结点值；若没有，则返回对应编程语言的空结点即可。

```
/*class ListNode {
 *     val: number
 *     next: ListNode | null
 *     constructor(val?: number, next?: ListNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.next = (next===undefined ? null : next)
 *     }
 * }
 */
/**
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 *
 * @param pHead ListNode类
 * @return ListNode类
 */
export function EntryNodeOfLoop(pHead: ListNode): ListNode {
    // write code here
    const set = new Set()
    while(pHead){
        //第一次出现重复的节点就是入口点
        if(set.has(pHead)){
            return pHead
        }else{
            set.add(pHead)
            pHead = pHead.next
        }
    }
    return null
}
```

反转链表

思路：让指针发生了改变，而不是想像中的位置发生了改变

```
/*class ListNode {
 *     val: number
 *     next: ListNode | null
 *     constructor(val?: number, next?: ListNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.next = (next===undefined ? null : next)
 *     }
```

```
 * }
 */
/**
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 *
 * @param head ListNode类
 * @return ListNode类
 */
export function ReverseList(head: ListNode): ListNode {
    // write code her
    let prev = null
    let current = head
    if(!head)return null
    while(current){
        //用一个变量来保存这个节点
        let temp = current.next
        current.next = prev
        prev = current
        current = temp
    }
    return prev
}
```

合并两个排序的链表

```
/*class ListNode {
 *     val: number
 *     next: ListNode | null
 *     constructor(val?: number, next?: ListNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.next = (next===undefined ? null : next)
 *     }
 * }
 */
/**
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 *
 * @param pHead1 ListNode类
 * @param pHead2 ListNode类
 * @return ListNode类
 */
export function Merge(pHead1: ListNode, pHead2: ListNode): ListNode {
    // write code here
    if(!pHead1)return pHead2
    if(!pHead2)return pHead1
    if(pHead1.val <= pHead2.val){
        pHead1.next = Merge(pHead1.next,pHead2)
        return pHead1
```

```
    }else{
        pHead2.next = Merge(pHead1,pHead2.next)
        return pHead2
    }
}
```

js写链表*

```
//节点
class Node{
    constructor(element){
        this.element = element
        this.next = null
    }
}

//链表
class LinkList{
    //初始化链表
    constructor(){
        this.length = 0
        this.head  = null
    }
    isEmpthy(){
        return this.length  == 0
    }
    size(){
        return this.length
    }
    append(element){
        //加到后面
        let node = new Node(element)
        let current
        if(this.head === null){
            //头部为空
            this.head = node
        }else{
            //头部不为空
            current = this.head
            while(current.next){
                current = current.next
            }
            current.next = node
        }
        this.length++
    }
    find(element){
        let current = this.head
        let index = 0
        if(element === current.element){
```

```
            return 0
        }
        while(current.next){
            if(current.element === element){
                return index
            }
            index++
            current  = current.next
        }
        if(element === current.element){
            return index
        }
        return -1
    }

}
```

判断左右字符串括号是否匹配，利用堆栈(括号生成)*

数字 `n` 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 **有效的** 括号组合。

示例 1：

```
输入：n = 3
输出：["((()))","(()())","(())()","()(())","()()()"]
```

示例 2：

```
输入：n = 1
输出：["()"]
```

数字 `n` 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 **有效的** 括号组合。

示例 1：

```
输入：n = 3
输出：["((()))","(()())","(())()","()(())","()()()"]
```

示例 2：

```
输入：n = 1
输出：["()"]
```

```
function isValid(s: string): boolean {
    let stack:string[] = []
    for(let i = 0;i < s.length;i++){
        switch(s[i]){
            case '(':
                stack.push(')')
```

```
                break
            case '[':
                stack.push(']')
                break
            case '{':
                stack.push('}')
                break
            default:
                if(stack.pop() !== s[i]){
                    return false
                }
                break
        }
    }
     return stack.length === 0
};
```

回文链表

环形链表

双指针技巧

快慢指针，设置两个指针一个指针快，一个慢，如果没有环，会有出现null的情况，如果有环，他们两个指针会相遇。

还可以解决:无环单链表的中点。寻找单链表的倒数第k个位置

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

/**
 * @param {ListNode} head
 * @return {boolean}
 */
var hasCycle = function(head) {
    let fast ,slow
    fast = slow = head
    while(fast !== null && fast.next !== null){
        fast = fast.next.next
        slow = slow.next
        if(fast === slow)return true
    }
    return false
};
```

环形链表，已知有环，返回这个环的起始位置

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

/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var detectCycle = function(head) {
    let fast ,slow
    fast = slow = head
    while(fast !== null && fast.next !== null){
        fast = fast.next.next
        slow = slow.next
        if(fast === slow)break
    }
    if(fast === null || fast.next === null){
        return null
    }
    slow = head
    while(slow !== fast){
        fast  = fast.next
        slow = slow.next
    }
    return slow
};
```

回文链表

思路:

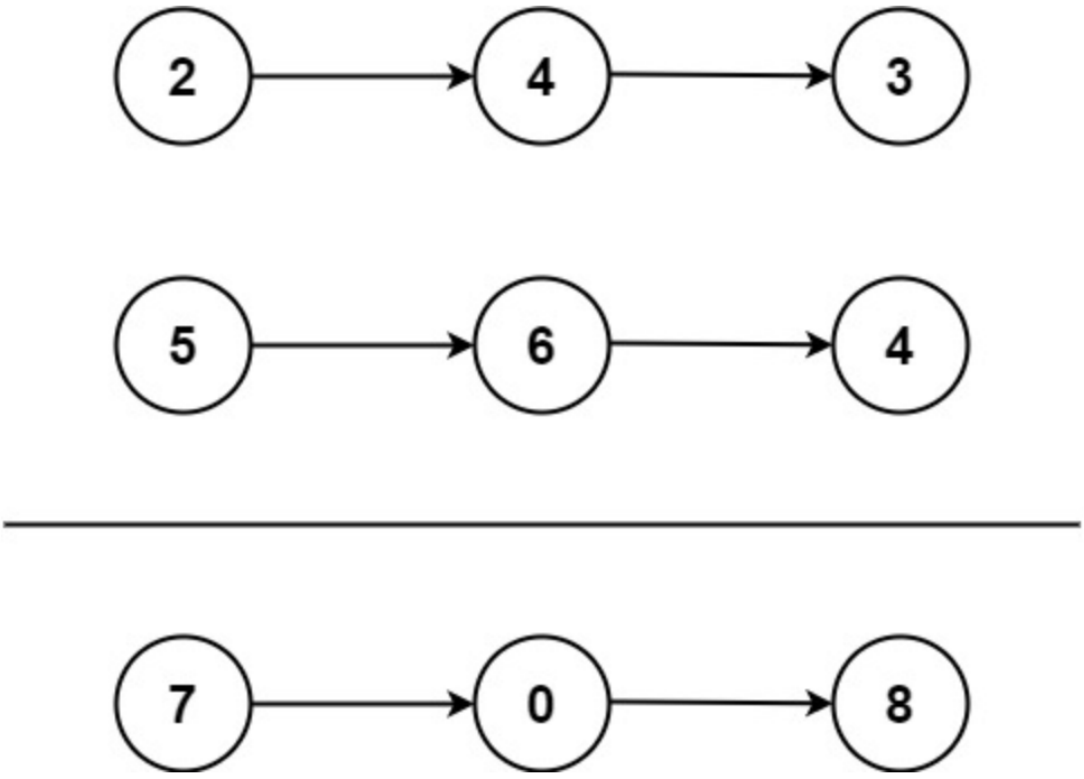
1. 遍历链表，将值放到数组里面
2. 接着就是使用双指针来进行判断是否为回文

```
/**
 * 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}
 */
var isPalindrome = function(head) {
    let arr = []
    while(head !== null){
        arr.push(head.val)
        head = head.next
    }
    for(let i = 0,j = arr.length - 1;i < j;i++,j--){
        if(arr[i] !== arr[j])return false
    }
    return true
};
```

两数之和

示例 1：



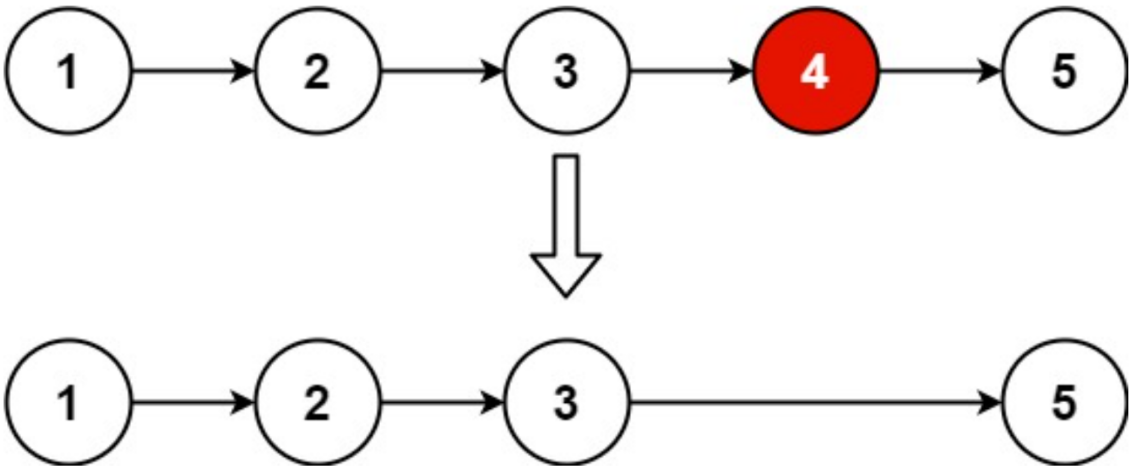
> **输入：** l1 = [2,4,3], l2 = [5,6,4]
> **输出：** [7,0,8]
> **解释：** 342 + 465 = 807.

```
function addTwoNumbers(l1: ListNode | null, l2: ListNode | null): ListNode | null {
    let dummy = new ListNode(0)
    let curr = dummy
    let carray = 0 //进位
    while(l1 !== null || l2 !== null || carray !== 0){
```

```
        let x = l1 ?l1.val :0
        let y = l2 ? l2.val : 0
        let sum = x + y + carray
        //计算进位
        carray = Math.floor(sum / 10)
        curr.next = new ListNode(sum % 10)
        curr = curr.next
        l1 = l1 ? l1.next:null
        l2 = l2 ? l2.next:null
    }
    return dummy.next
};
```

删除链表的倒数第N个节点

示例 1：



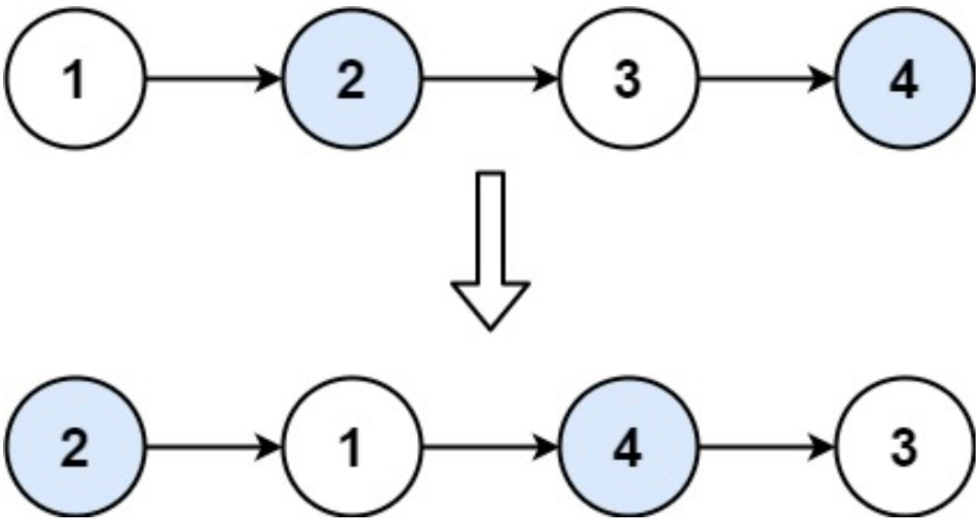
> **输入：** head = [1,2,3,4,5], n = 2
> **输出：** [1,2,3,5]

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     val: number
 *     next: ListNode | null
 *     constructor(val?: number, next?: ListNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.next = (next===undefined ? null : next)
 *     }
 * }
 */
//循环获得链表长度
    function getLength(head:ListNode){
        let count = 0
        while(head){
            count++
```

```
            head = head.next
        }
        return count
    }
function removeNthFromEnd(head: ListNode | null, n: number): ListNode | null {
    const length = getLength(head)
    let dummy = new ListNode(0,head)
    let curr = dummy
    for(let i = 0;i < length - n ;i++){
        curr = curr.next
    }
    curr.next = curr.next.next
    let ans = dummy.next
    return ans
};
```

两两交换链表中的节点

示例 1：



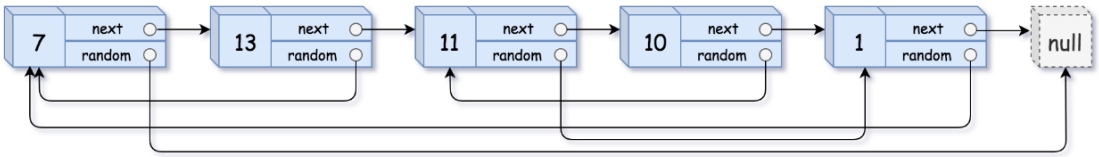
> **输入：** head = [1,2,3,4]
> **输出：** [2,1,4,3]

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     val: number
 *     next: ListNode | null
 *     constructor(val?: number, next?: ListNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.next = (next===undefined ? null : next)
 *     }
 * }
 */
```

```
function swapPairs(head: ListNode | null): ListNode | null {
    let dummy = new ListNode(0,head)
    let temp = dummy
    while(temp.next !== null && temp.next.next!== null){
        let temp1 = temp.next
        let temp2 = temp.next.next
        temp1.next = temp2.next
        temp2.next = temp1
        temp.next = temp2
        temp = temp1
    }
    return dummy.next
};
```

随机链表的复制

示例 1：



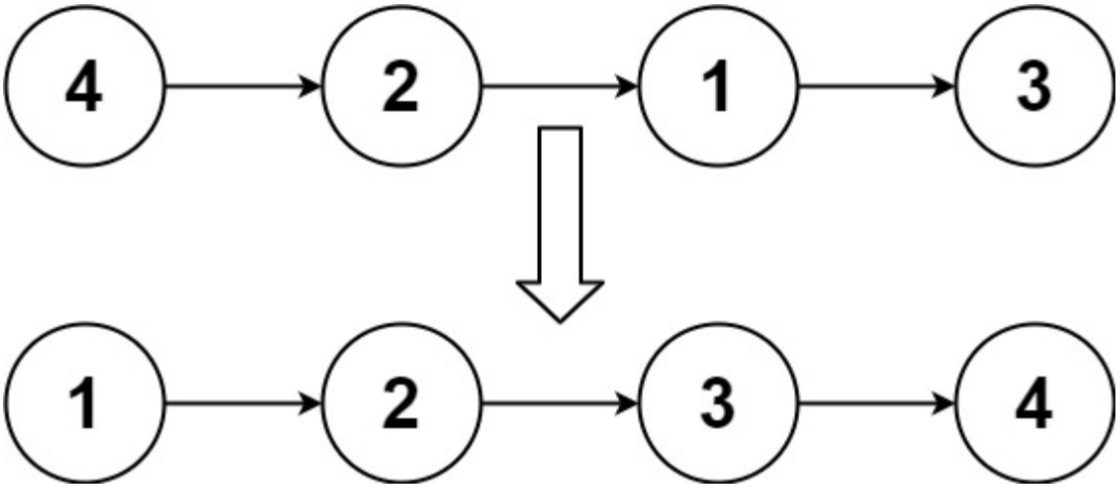
> **输入：** head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
> **输出：** [[7,null],[13,0],[11,4],[10,2],[1,0]]

```
/**
 * Definition for _Node.
 * class _Node {
 *     val: number
 *     next: _Node | null
 *     random: _Node | null
 *
 *     constructor(val?: number, next?: _Node, random?: _Node) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.next = (next===undefined ? null : next)
 *         this.random = (random===undefined ? null : random)
 *     }
 * }
 */

/**
就是使用map来存储一条链
接着复制其中的next和random
**/
```

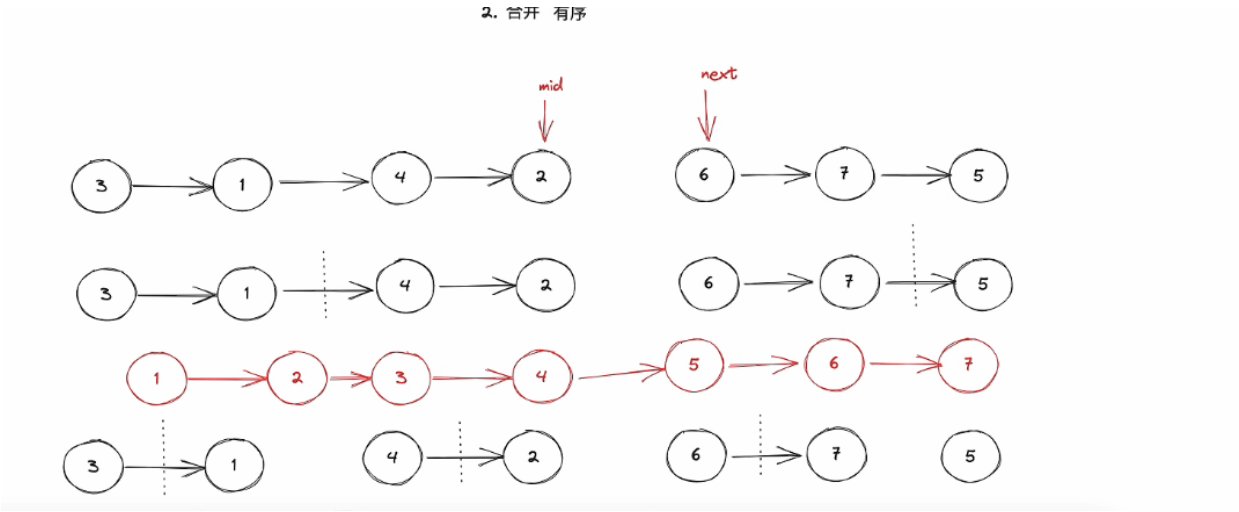
```
function copyRandomList(head: _Node | null): _Node | null {
    if(!head)return head
    let cur = head
    //进行生成链表
    const map = new Map()
    while(cur){
        map.set(cur,new _Node(cur.val))
        cur = cur.next
    }
    cur = head
    //第二次遍历，进行复制next和random或者null
    while(cur){
        map.get(cur).next = map.get(cur.next) || null
        map.get(cur).random = map.get(cur.random) || null
        cur = cur.next
    }
    return map.get(head)
};
```

排序链表

示例 1：



> **输入**: head = [4,2,1,3]
> **输出**: [1,2,3,4]




```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     val: number
 *     next: ListNode | null
 *     constructor(val?: number, next?: ListNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.next = (next===undefined ? null : next)
 *     }
 * }
 */

function sortList(head: ListNode | null): ListNode | null {
    //递归和合并
    /**
    1. 拆分:中间节点拆分
    2. 合并: 有序合并
     */
     return mergeSortRec(head)
};
//归并排序
function mergeSortRec(head){
    if(!head || !head.next){
        return head
    }

    //获取中间节点
    let middle = middleNode(head)
    //进行分割
    let temp = middle.next
    middle.next = null
    let left = head,right = temp
    //进行递归
    left = mergeSortRec(left)
    right = mergeSortRec(right)
    //进行合并
    return mergeTwoList(left,right)
```

```
}
//快慢指针寻找中间节点
function middleNode(head){
    let fast = head,slow = head
    while(fast && fast.next && fast.next.next){
        slow = slow.next
        fast = fast.next.next
    }
    return slow
}
//合并的思路就是新建一个链表在里面进行复制链表
function mergeTwoList(left,right){
    let preHead = new ListNode(-1)
    let current = preHead
    while(left && right){
        if(left.val < right.val){
            current.next = left
            left = left.next
        }else{
            current.next = right
            right = right.next
        }
        current = current.next
    }
    current.next = left || right
    return preHead.next
}
```

重排链表

给定一个单链表 `L` 的头节点 `head` ，单链表 `L` 表示为：

```
L0 → L1 → … → Ln - 1 → Ln
```

请将其重新排列后变为：

```
L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …
```

不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

数组

设计两端的可以想双指针,左右指针，常考

合并区间*

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [starti, endi]` 。请你合并所有重叠的区间，并返回 *一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间* 。

```
输入：intervals = [[1,3],[2,6],[8,10],[15,18]]
输出：[[1,6],[8,10],[15,18]]
解释：区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6].
```

```
function merge(intervals: number[][]): number[][] {
    //升序
    let resultSort = intervals((a,b)=>a[0] - b[0])
    const result = []
    for(let i = 1;i < resultSort.length;i++){
        if(resultSort[i][0] < resultSort[i-1][1]){
            //查找来进行合并
            resultSort[i] =[ resultSort[i - 1][0],Math.max(resultSort[i-1]
[1],resultSort[i][1])]
        }else{
            //没有就push进去
            result.push(resultSort[i - 1])
        }
    }
    //push最后一个
    result.push(resultSort[intervals.length - 1])
    return result
};
```

顺时针打印矩阵

描述

输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，例如，如果输入如下4 X 4矩阵：

```
[[1,2,3,4],
[5,6,7,8],
[9,10,11,12],
[13,14,15,16]]
```

则依次打印出数字

```
[1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10]
```

数据范围:

0 <= matrix.length <= 100

0 <= matrix[i].length <= 100

```
/**
```

```
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 * @param matrix int整型二维数组
 * @return int整型一维数组
 */
export function printMatrix(matrix: number[][]): number[] {
    // write code here
    const res = []
    if(matrix.length === 0)return res
    //四个边界
    let left = 0
    let right = matrix[0].length - 1
    let up = 0
    let down = matrix.length - 1
    while(left <= right&& up <= down){
        //从左到右
        for(let i = left;i <= right;i++){
            res.push(matrix[up][i])
        }
        up++
        if(up > down){
            break
        }
        //从上到下
        for(let i = up;i <= down;i++){
            res.push(matrix[i][right])
        }
        right--
        if(left > right){
            break
        }
        //从右到左
        for(let i = right;i >= left;i--){
            res.push(matrix[down][i])
        }
        down--
        if(up > down){
            break
        }
        //从下到上
        for(let i = down;i >= up;i--){
            res.push(matrix[i][left])
        }
        left++
        if(left > right){
            break
        }

    }
  return res
}
```

最长回文子序列*

思路:

双指针从中心向两边扩散

```
/**
 * @param {string} s
 * @return {string}
 */
var longestPalindrome = function(s) {
    let result = ''
    for(let i = 0;i < s.length;i++){
        //根据字符串的长度来进行判断
        let s1 = findZi(s,i,i)
        let s2 = findZi(s,i,i+1)
        result = result.length > s1.length?result:s1
        result = result.length > s2.length?result:s2
    }
    return result
};
//找到回文子串
function findZi(s,l,r){
    while(l >= 0 && r < s.length&&s.charAt(l) === s.charAt(r)){
        l--
        r++
    }
    return s.slice(l + 1,r)
}
```

扁平化嵌套数组*

题目链接:

https://leetcode.cn/problems/flatten-deeply-nested-array/description/

请你编写一个函数，它接收一个 **多维数组** `arr` 和它的深度 `n` ，并返回该数组的 **扁平化** 后的结果。

多维数组 是一种包含整数或其他 **多维数组** 的递归数据结构。

数组 **扁平化** 是对数组的一种操作，定义是将原数组部分或全部子数组删除，并替换为该子数组中的实际元素。只有当嵌套的数组深度大于 `n` 时，才应该执行扁平化操作。第一层数组中元素的深度被认为是 0。

请在没有使用内置方法 `Array.flat` 的前提下解决这个问题。

思路：就是有一个递归终止条件：n === 0的时候返回原数组，接着就是遍历数组里面的项了，判断是不是数组，是数组就接着进行递归否则就用数组包裹起来拍平放入最后结果中

```
type MultiDimensionalArray = (number | MultiDimensionalArray)[];

var flat = function (arr:  MultiDimensionalArray, n: number):  MultiDimensionalArray
{
    //这一层就是已经到达了相应的深度的数组,拨掉一层就可以了，将他...
```

```
    if(n === 0){
     return arr
    }
    let result = []
    for(const item of arr){
     result.push(...(Array.isArray(item)?flat(item  as MultiDimensionalArray,n-1):
[item]))
    }
    return result
};
```

合并两个有序的数组(*)

不利用额外空间，两个数组从后往前遍历，分别从m，n项开始从后往前遍历
哪个大将哪个添加至nums1后
当m已经小于0时，将剩余的nums添加至nums1前面

思路一：合并后排序

```
/**
 Do not return anything, modify nums1 in-place instead.
 */
function merge(nums1: number[], m: number, nums2: number[], n: number): void {
    const arr = nums1.slice(0,m).concat(nums2)
    arr.sort((a,b)=>a-b)
    for(let i = 0;i < m + n;i++){
        nums1[i] = arr[i]
    }
};
```

思路二：

```
var merge = function(nums1,m,nums2,n){
    var len = m+n-1;
    m = m-1;
    n = n -1;
    while(m>=0&&n>=0) {
        nums1[len] = nums1[m]>nums2[n]?nums1[m--]:nums2[n--];
        len--;
    }
    //当一个数组的元素都被比较并放入 nums1 后，可能 nums2 中仍有剩余元素未被放入（当 m 先用完
时）。
    return nums1.splice(0,n+1,...nums2.slice(0,n+1));
}
```

和为k的子数组

```
function subarraySum(nums: number[], k: number): number {
    let count = 0
    let sum = 0
    for(let i =0;i < nums.length;++i){
        sum = 0
        for(let j = i;j >= 0;--j){
            sum += nums[j]
            if(sum === k){
                count++
            }
        }
    }
    return count
};
```

最大子数组和(*)

给你一个整数数组 `nums` ，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

子数组

是数组中的一个连续部分

```
输入：nums = [-2,1,-3,4,-1,2,1,-5,4]
输出：6
解释：连续子数组 [4,-1,2,1] 的和最大，为 6 。
```

```
function maxSubArray(nums: number[]): number {
    let pre = 0,maxNum = nums[0]
    for(let i = 0;i < nums.length;i++){
        pre = Math.max(nums[i] + pre,nums[i])
        maxNum = Math.max(maxNum,pre)
    }
    return maxNum
};
```

//动态规划的方法做

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var maxSubArray = function(nums) {
    //找到转移方程dp[i] = Math.max(nums[i],nums[i] + dp[i - 1])
    const dp = []
    dp[0] = nums[0]
    let result = nums[0]
```

```
    for(let i = 1;i < nums.length;i++){
        dp[i] = Math.max(nums[i],nums[i] + dp[i - 1])
        result  = Math.max(result,dp[i])
    }
    return result
};
```

最长公共子序列（*）（二维动态规划）

求两个数组或者字符串的最长公共子序列问题，肯定是要用动态规划的。

给定两个字符串 `text1` 和 `text2`，返回这两个字符串的最长 **公共子序列** 的长度。如果不存在 **公共子序列** ，返回 `0` 。

一个字符串的 **子序列** 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

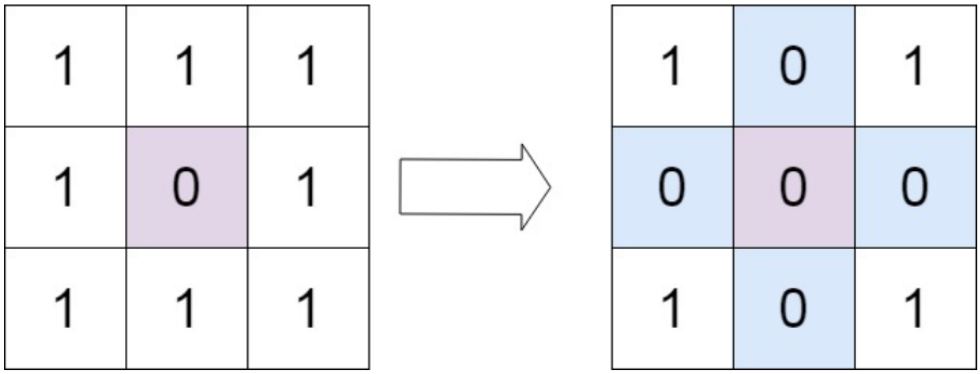
- 例如，`"ace"` 是 `"abcde"` 的子序列，但 `"aec"` 不是 `"abcde"` 的子序列。

两个字符串的 **公共子序列** 是这两个字符串所共同拥有的子序列

```
function longestCommonSubsequence(text1: string, text2: string): number {
    const m = text1.length,n = text2.length
    const dp = new Array(m + 1).fill(0).map(()=>new Array(n + 1).fill(0))
    for(let i = 1;i <= m;i++){
        const c1 = text1[i - 1]
        for(let j = 1;j <= n;j++){
            const c2 = text2[j - 1]
            if(c1 === c2){
                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[m][n]
};
```

矩阵置为0

示例 1：



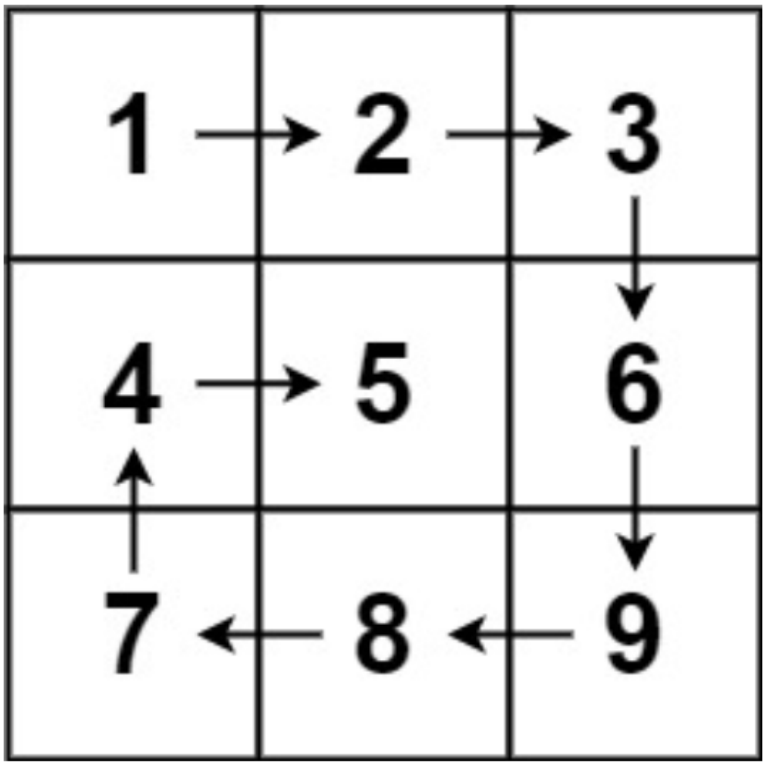
> **输入：** matrix = [[1,1,1],[1,0,1],[1,1,1]]
> **输出：** [[1,0,1],[0,0,0],[1,0,1]]

```
/**
 Do not return anything, modify matrix in-place instead.
 */
/**
思路：
1.就是设置了两个映射，记录一下是0的行，以及是0的列
2.接着就是在映射中找到循环相应的行将列置为0，其次列循环置为0也是这个操作
**/
function setZeroes(matrix: number[][]): void {
    let row = matrix.length
    let col = matrix[0].length

    let map1 = {},map2 = {}
    for(let i = 0;i < row;i++){
        for(let j = 0;j < col;j++){
            if(matrix[i][j] === 0){
                map1[i] = i
                map2[j] = j
            }
        }
    }
    //接着行循环置为零
    for(let i in map1){
        for(let j = 0;j < col;j++){
            matrix[i][j] = 0
        }
    }
    //接着列循环置为0
    for(let i in map2){
        for(let j = 0;j < row;j++){
            matrix[j][i] = 0
        }
    }
```

```
};
```

螺旋矩阵

示例 1：



> **输入：** matrix = [[1,2,3],[4,5,6],[7,8,9]]
> **输出：** [1,2,3,6,9,8,7,4,5]

示例 2：

```
function spiralOrder(matrix: number[][]): number[] {
    let left = 0
    let right = matrix[0].length - 1
    let top = 0
    let bottom = matrix.length - 1
    const result = []
    while(true){
        for(let i = left;i <= right;i++){
            result.push(matrix[top][i])
        }
        top++
        if(top > bottom)break
        for(let i = top;i <= bottom;i++){
            result.push(matrix[i][right])
        }
        right--
```

```
        if(right < left)break
        for(let i = right;i >= left;i--){
            result.push(matrix[bottom][i])
        }
        bottom--
        if(bottom < top)break
        for(let i = bottom;i >= top;i--){
            result.push(matrix[i][left])
        }
        left++
        if(left > right)break
    }
    return result
};
```

数组转化为树状数据结构

数组转化为树状数据结构

题目描述
实现一个函数，可以将数组转化为树状数据结构

1. 数组只有一个没有 parentId 的元素，为根节点
2. 输出一个根节点，打印出树状结构

// 入参格式参考：
const arr = [
{ id: 1, name: 'i1' },
{ id: 2, name: 'i2', parentId: 1 },
{ id: 4, name: 'i4', parentId: 3 },
{ id: 3, name: 'i3', parentId: 2 },
{ id: 7, name: 'i7', parentId: 3 },
{ id: 8, name: 'i8', parentId: 3 }
]

```
const flatToTree = (list)=>{
    const result = []
    const itemMap = {}
    list.map(item=>{
        const {id,pid} = item
        if(!itemMap[id]?.children){
            itemMap[id].children = []
        }
        itemMap[id] = {
            ...item,
            children:itemMap[id].children
        }
        //pid
        const treeItem = itemMap[id]
```

```
        if(pid === 0){
            result.push(treeItem)
        }else{
            if(treeItem[pid]?.children){
                itemMap[pid] = {
                    children:[]
                }
            }
            itemMap[pid]["children"].push(treeItem)
        }
    })
    return result
}
```

树状结构转数组

```
const treeeToFlat = (data)=>{
    const result = []
    const queue = [...data]
    while(queue.length){
        const node = queue.shift()
        const children = node.children
        if(children){
            queue.push(...children)
        }
        delete node.children
        //将children删除了，就可以加入结果了
        result.push(node)
    }
    return result
}
```

和为K的子数组

给你一个整数数组 `nums` 和一个整数 `k` ，请你统计并返回 *该数组中和为 `k` 的子数组的个数* 。

子数组是数组中元素的连续非空序列。

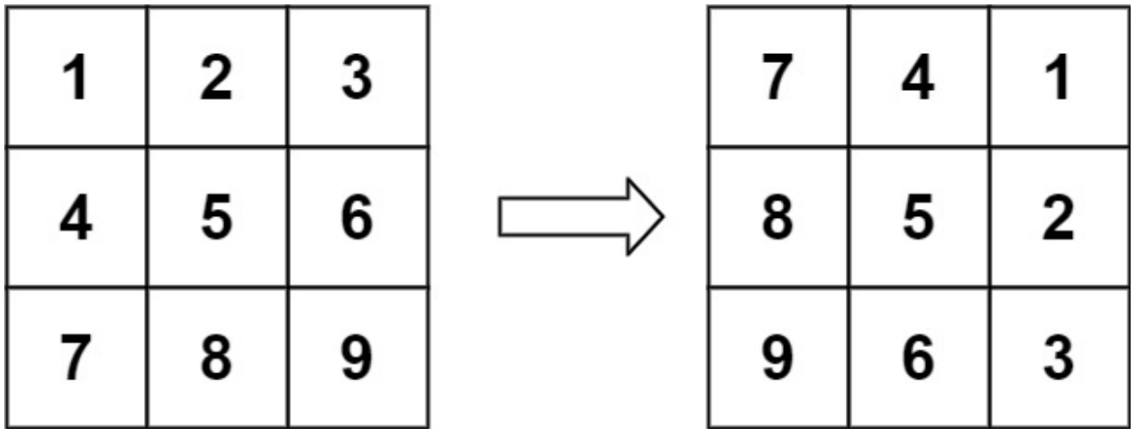
```
输入：nums = [1,1,1], k = 2
输出：2
```

```
/**
解题:前缀和 + hash表
**/
function subarraySum(nums: number[], k: number): number {
    let pre = 0,map = new Map(),count = 0
    for(let i = 0;i < nums.length;i++){
        pre += nums[i]
```

```
        if(pre === k)count++
        if(map.has(pre - k)){
            count += map.get(pre-k)
        }

        if(map.has(pre)){
            map.set(pre,map.get(pre) + 1)
        }else{
            map.set(pre,1)
        }
    }
    return count
};
```

旋转图像

示例 1：



> **输入：** matrix = [[1,2,3],[4,5,6],[7,8,9]]
> **输出：** [[7,4,1],[8,5,2],[9,6,3]]

```
/**
 Do not return anything, modify matrix in-place instead.
 */
function rotate(matrix: number[][]): void {
    //思路：先沿水平中轴线翻转，然后在沿主对角线翻转.
    const n = matrix.length
    for(let i = 0;i < Math.floor(n / 2);i++){
        for(let j = 0;j < n;j++){
            [matrix[i][j],matrix[n - i - 1][j]] = [matrix[n - i - 1][j],matrix[i]
[j]]
        }
    }

    for(let i = 0;i < n;i++){
        for(let j = 0;j < i;j++){
            [matrix[i][j],matrix[j][i]] = [matrix[j][i],matrix[i][j]]
```

```
        }
    }

};
```

搜索二维矩阵 II

编写一个高效的算法来搜索 `*m* x *n*` 矩阵 `matrix` 中的一个目标值 `target` 。该矩阵具有以下特性：

- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

```
function searchMatrix(matrix: number[][], target: number): boolean {
    //从左下角开始寻找的
    let col = 0
    let row = matrix[0].length - 1
    //一个限定条件:col和row的边界条件
    while(col  <= matrix.length - 1&&row >= 0){
        if(matrix[col][row] === target){
            return true
        }else if(matrix[col][row] < target){
            col++
        }else{
            row--
        }
    }
    return false
};
```

轮转数组

给定一个整数数组 `nums`，将数组中的元素向右轮转 `k` 个位置，其中 `k` 是非负数。

```
function rotate(nums,k){
    for(let i = 0;i < k;i++){
        const last = nums.pop()
        nums.unshift(last)
    }
}
```

二叉树

DFS

```
void traverse(TreeNode root) {
    // 判断 base case
    if (root == null) {
        return;
    }
    // 访问两个相邻结点：左子结点、右子结点
    traverse(root.left);
    traverse(root.right);
}
```

遍历

前序遍历:中、左、右

中序遍历:左、中、右

后序遍历:左、右、中

层序遍历:一层一层的遍历

二叉树最大深度

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */
function getDepth(node){
    if(node === null)return 0
    let leftDepth = getDepth(node.left)
    let rightDepth = getDepth(node.right)
    let depth = 1 + Math.max(leftDepth,rightDepth)
    return depth
}

function maxDepth(root: TreeNode | null): number {
    return getDepth(root)
};
```

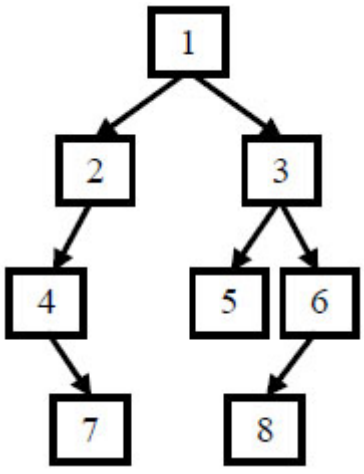
中序遍历

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */

function inorderTraversal(root: TreeNode | null): number[] {
    let res = []
    function dfs(root){
        if(!root)return
        dfs(root.left)
        res.push(root.val)
        dfs(root.right)
    }
    dfs(root)
    return res
};
```

重构二叉树(剑指)

给定节点数为 n 的二叉树的前序遍历和中序遍历结果，请重建出该二叉树并返回它的头结点。

例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建出如下图所示。



提示:

1.vin.length == pre.length

2.pre 和 vin 均无重复元素

3.vin出现的元素均出现在 pre里

4.只需要返回根结点，系统会自动输出整颗树做答案对比

数据范围：n≤2000n≤2000，节点的值 −10000≤val≤10000−10000≤val≤10000

要求：空间复杂度 $O(n)$$O(n)$，时间复杂度 $O(n)$$O(n)$

思路:前序遍历和中序遍历构建二叉树

```
export function reConstructBinaryTree(preOrder: number[], vinOrder: number[]):
TreeNode {
    // write code here
    if(!preOrder.length || !vinOrder.length)return null
    let root = preOrder[0]
    let index = vinOrder.indexOf(root)
    //前序遍历:中左右
    //中序遍历:左中后
    let vinLeft = vinOrder.slice(0,index )
    let vinRight = vinOrder.slice(index + 1)
    let preLeft = preOrder.slice(1,index + 1)
    let preRight = preOrder.slice(index+1)
    //基于根节点新建二叉树
    let node = new TreeNode(root)
    node.left = reConstructBinaryTree(preLeft,vinLeft)
    node.right = reConstructBinaryTree(preRight,vinRight)
    return node
}
```

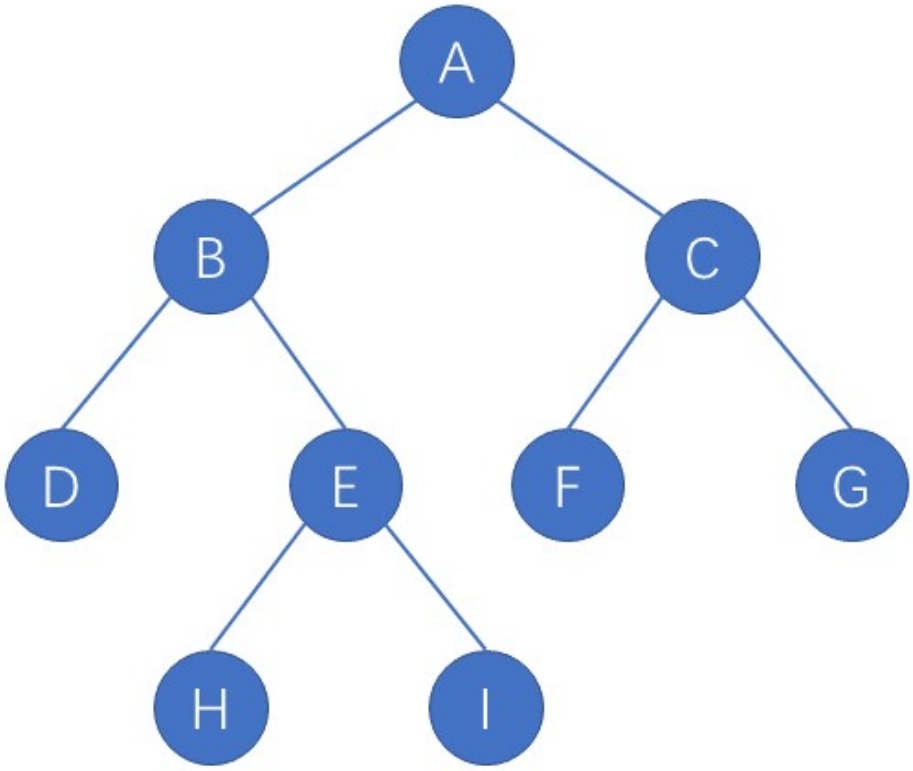
题目：二叉树的下一个节点

给定一棵二叉树的其中一个节点，请找出中序遍历序列的下一个节点。（树的后继）

注意：

- **如果给定的节点是中序遍历序列的最后一个，则返回空节点;**
- **二叉树一定不为空，且给定的节点一定不是空节点;**

解题思路



二叉树的中序遍历：{ [左子树], 根节点, [右子树] }

如图所示二叉树的中序遍历：D, B, H, E, I, A, F, C, G

分三种情况：

1. 如果该节点**有右子树**，那么下一个节点就是其**右子树中最左边的节点；**
2. 如果该节点**没有右子树**，且是其**父节点的左子节点**，那么下一个节点就是其父节点；
3. 如果该节点**没有右子树**，且是其**父节点的右子节点**，沿着父指针一直向上，直到找到一个是它**父节点的左子节点**的节点，如果这样的节点存在，那么这个节点的父节点即是所求。

例如：

- 情况 1：图中节点 B 的下一个节点是节点 H；
- 情况 2：图中节点 H 的下一个节点是节点 E；
- 情况 3：图中节点 I 的下一个节点是节点 A。

时空复杂度：O(height)，其中 height 是二叉树的高度， 空间复杂度：O(1)

代码

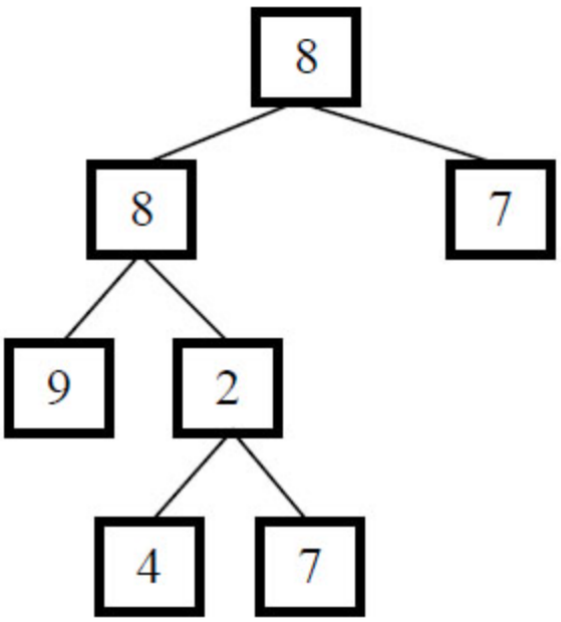
```
export function GetNext(pNode: TreeLinkNode): TreeLinkNode {
    if(pNode === null)return null
    // 1. 有右子树
    if(pNode.right !== null){
        //那么下一个就是
```

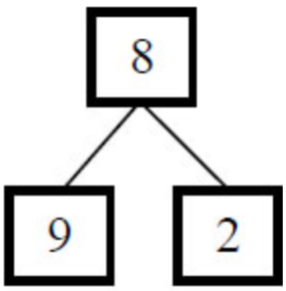
```
            let p = pNode.right
            while(p.left !== null){
                p = p.left
            }
            return p
        }
        // 2. 没有右子树
        //不是根节点
        while(pNode.next !== null){
            //这个节点是父节点的左节点就返回这个父节点
            if(pNode === pNode.next.left){
                //返回这个父节点
                return pNode.next
            }
            pNode = pNode.next
        }

}
```

树的子结构

输入两棵二叉树A，B，判断B是不是A的子结构。（我们约定空树不是任意一个树的子结构）

假如给定A为{8,8,7,9,2,#,#,#,#,4,7}，B为{8,9,2}，2个树的结构如下，可以看出B是A的子结构





数据范围:

0 <= A的节点个数 <= 10000

0 <= B的节点个数 <= 10000

示例:

输入:

```
{8,8,7,9,2,#,#,#,#,4,7},{8,9,2}
```

复制

返回值：

```
true
```

```
 /*class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */

/**
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 * @param pRoot1 TreeNode类
 * @param pRoot2 TreeNode类
 * @return bool布尔型
 */
export function HasSubtree(pRoot1: TreeNode, pRoot2: TreeNode): boolean {
    // write code here
    if(!pRoot1 || !pRoot2)return false
    return isSame(pRoot1,pRoot2) || HasSubtree(pRoot1.left,pRoot2) ||
HasSubtree(pRoot1.right,pRoot2)
}
function isSame(pRoot1:TreeNode,pRoot2:TreeNode):boolean{
    if(!pRoot2)return true
    else if(!pRoot1)return false
    if(pRoot1.val !== pRoot2.val)return false
    return isSame(pRoot1.left,pRoot2.left) && isSame(pRoot1.right,pRoot1.right)
}
```

二叉树的镜像(交换的算法)

描述

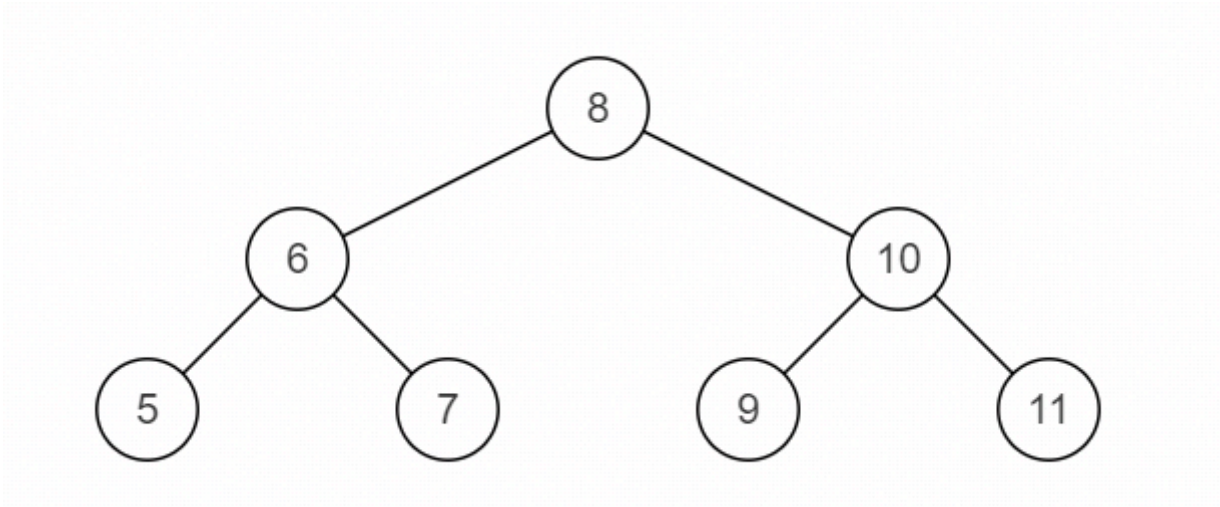
操作给定的二叉树，将其变换为源二叉树的镜像。

数据范围：二叉树的节点数 $0 \le n \le 10000 \le n \le 1000$ ， 二叉树每个节点的值 $0 \le val \le 10000 \le val \le 1000$

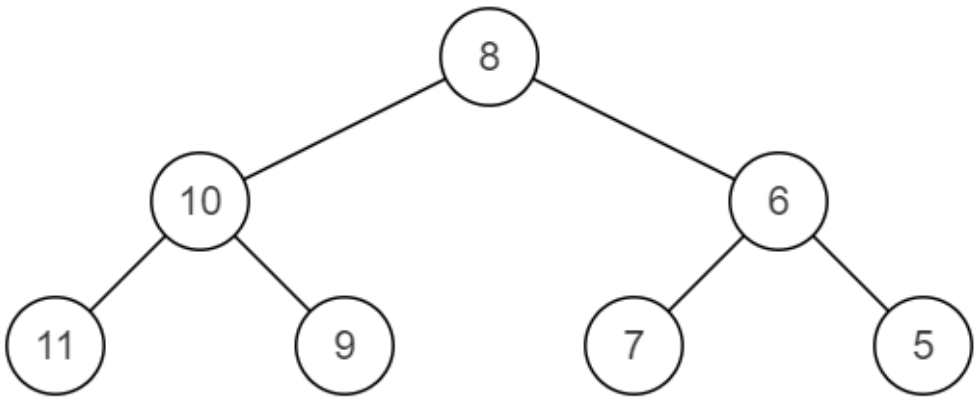
要求： 空间复杂度 $O(n)O(n)$ 。本题也有原地操作，即空间复杂度 $O(1)O(1)$ 的解法，时间复杂度 $O(n)O(n)$

比如：

源二叉树



镜像二叉树




```
/*class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */
/**
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 *
 * @param pRoot TreeNode类
 * @return TreeNode类
 */
export function Mirror(pRoot: TreeNode): TreeNode {
```

```
    // write code here
    return swap(pRoot)
}
function swap(a){
    if(a === null)return
    let temp = a.right
    a.right = a.left
    a.left = temp
    swap(a.left)
    swap(a.right)
    return a
}
```

翻转二叉树

```
```

对称二叉树

```
/**
 * 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}
 */
var isSymmetric = function(root) {
    function isEqual(m,n){
        if(!m&&!n){
            return true
        }
        if(m?.val !== n?.val ){
            return false
        }
        return isEqual(m.left,n.right)&&isEqual(m.right,n.left)
    }
    return isEqual(root,root)
};
```

二叉树的直径

```
/**
 * 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 {number}
 */
var diameterOfBinaryTree = function(root) {

    let maxLength = 0
    function getDepth(root){
        if(!root){
        return 0
        }else{
            let left = getDepth(root.left)
            let right = getDepth(root.right)
            maxLength = Math.max(maxLength,left + right)
            return Math.max(left,right)+1
        }
    }
    getDepth(root)
    return maxLength
};
```

层序遍历

```
/**
 * 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 {number[][]}
 */
var levelOrder = function(root) {
    const queue = [],result = []
    queue.push(root)
    if(!root){
```

```
        return result
    }
    while(queue.length){
        let ceng = []
        const length = queue.length
        for(let i = 0;i < length;i++){
            const node = queue.shift()
            ceng.push(node.val)
            node.left && queue.push(node.left)
            node.right && queue.push(node.right)
        }
        result.push(ceng)
    }
    return result
};
```

将有序数组转化为二叉搜索树

```
/**
 * 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 {number[]} nums
 * @return {TreeNode}
 */
var sortedArrayToBST = function(nums) {
   return dfs(nums,0,nums.length-1)
};
function dfs(nums,low,right){
    if(low > right)return null
    const mid = (low +Math.floor((right - low) / 2))
    const Node = new TreeNode(nums[mid])
    Node.left = dfs(nums,low,mid - 1)
    Node.right = dfs(nums,mid + 1,right)
    return Node
}
```

验证二叉搜索树

思路：中序遍历，如果不是升序那么就是错误的

```
/**
 * 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}
 */
var isValidBST = function(root) {
    const arr = []
    function dfs(root){
        if(root){
            dfs(root.left)
            arr.push(root.val)
            dfs(root.right)
        }
    }
    dfs(root)
    for(let i = 1;i < arr.length;i++){
        if(arr[i - 1] >= arr[i]){
            return false
        }
    }
    return true
};
```

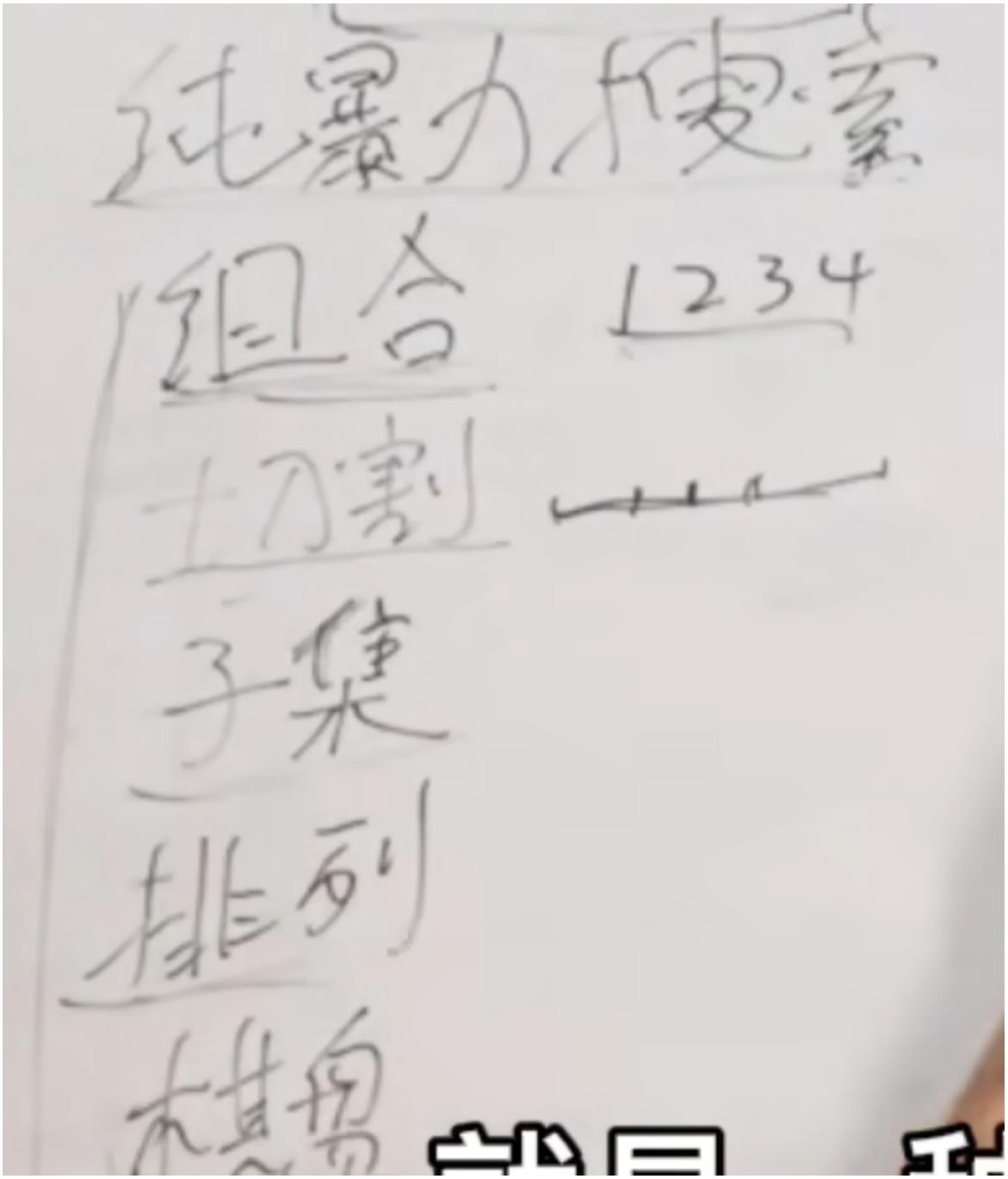
二叉树中第k个小的元素

思路：中序 + 返回第k个小的元素

```
/**
 * 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
 * @param {number} k
 * @return {number}
 */
var kthSmallest = function(root, k) {
    const arr = []
    function dfs(root){
        if(root){
            dfs(root.left)
            arr.push(root.val)
            dfs(root.right)
        }
```

```
    }
    dfs(root)
    return arr[k-1]
};
```

回溯



回溯模板

```
void backtracking(参数) {
    if (终止条件) {
        存放结果;
        return;
    }

    for (选择：本层集合中元素（树中节点孩子的数量就是集合的大小）) {
        处理节点;
        backtracking(路径，选择列表); // 递归
        回溯，撤销处理结果
    }
}
```

全排列

```
// 输入：nums = [1,2,3]
// 输出：[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
/**
 * @param {number[]} nums
 * @return {number[][]}
 */
var permute = function(nums) {
    const res = [], path = [];
    backtracking(nums, nums.length, []);
    return res;

    function backtracking(n, k, used) {
        if(path.length === k) {
            res.push(Array.from(path));
            return;
        }
        for (let i = 0; i < k; i++ ) {
            //中间变量记录哪些已经取了，就不会重新取了。
            if(used[i]) continue;
            path.push(n[i]);
            used[i] = true; // 同支
            backtracking(n, k, used);
            path.pop();
            used[i] = false;
        }
    }
};
```

电话号码的字母组合

给定一个仅包含数字 `2-9` 的字符串，返回所有它能表示的字母组合。答案可以按 **任意顺序** 返回。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。


```
function letterCombinations(digits: string): string[] {
    if(digits === '')return []
    //数字和字母的映射
    const strMap = {
        1:[],
        2: ['a', 'b', 'c'],
        3: ['d', 'e', 'f'],
        4: ['g', 'h', 'i'],
        5: ['j', 'k', 'l'],
        6: ['m', 'n', 'o'],
        7: ['p', 'q', 'r', 's'],
        8: ['t', 'u', 'v'],
        9: ['w', 'x', 'y', 'z'],
    }
    const result = []
    function backTracking(digits:string,curIndex:number,route:string[]):void{
        //收集结果，终止
        if(curIndex === digits.length){
            result.push(route.join(''))
            return
        }
        //进行获取映射的数组
        let tempArr  = strMap[digits[curIndex]]
        for(let i = 0;i < tempArr.length;i++){
            route.push(tempArr[i])
            backTracking(digits,curIndex+1,route)
            route.pop()
        }
    }
    backTracking(digits,0,[])
    return result
};
```

组合总和

```
function combinationSum(candidates: number[], target: number): number[][] {
    const result = []
    function
backTracking(candidates:number[],target:number,startIndex:number,route:number[],curS
um:number){
        if(curSum > target){
            return
        }
        if(curSum === target){
            result.push(route.slice())
            return
        }
        for(let i  = startIndex;i < candidates.length;i++){
            const tempVal = candidates[i]
            route.push(tempVal)
            backTracking(candidates,target,i,route,curSum + tempVal)
            route.pop()
        }
    }
    backTracking(candidates,target,0,[],0)
    return result
};
```

括号的生成

数字 `n` 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 **有效的** 括号组合。

```
输入：n = 3
输出：["((()))","(()())","(())()","()(())","()()()"]
```

```
function generateParenthesis(n: number): string[] {
    const res = []
    backTracking(0,0,n,res,'')
    return res
};
function backTracking(left,right,n,result,str){
    if(left == right && left === n){
        result.push(str)
        return
    }
    if(left < right){
        return
    }
    if(right < left){
        backTracking(left,right+1,n,result,str+')')
    }
    if(left < n){
```

```
        backTracking(left+1,right,n,result,str +'(')
    }

}
```

分割回文串

```
function partition(s: string): string[][] {
    const result = [],path = [],len = s.length
    backtracking(0)
    return result
    function backtracking (i){
        if(i >= len){
            result.push(Array.from(path))
            return
        }
        for(let j = i;j <len;j++){
            //判断是否为回文字符串
            if(!isPalindrom(s,i,j))continue
            path.push(s.slice(i,j+1))
            backtracking(j + 1)
            path.pop()
        }
    }
};

const isPalindrom = (s,l,r)=>{
    for(let i = l,j = r;i < j;i++,j--){
        if(s[i]!== s[j])return false
    }
    return true
}
```

复原IP地址**

栈和队列

用两个栈实现队列

描述

用两个栈来实现一个队列，使用n个元素来完成 n 次在队列尾部插入整数(push)和n次在队列头部删除整数(pop)的功能。 队列中的元素为int类型。保证操作合法，即保证pop操作时队列内已有元素。

数据范围： $n\leq 1000$

要求：存储n个元素的空间复杂度为 $O(n)$ ，插入与删除的时间复杂度都是 $O(1)$

示例1

输入:

```
["PSH1","PSH2","POP","POP"]
```

复制

返回值:

```
1,2
```

复制

说明:

```
"PSH1":代表将1插入队列尾部
"PSH2":代表将2插入队列尾部
"POP“:代表删除一个元素，先进先出=>返回1
"POP“:代表删除一个元素，先进先出=>返回2
```

示例2

输入:

```
["PSH2","POP","PSH1","POP"]
```

复制

返回值:

```
2,1
```

代码:

```
/**
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 *
 * @param node int整型
 * @return 无
 */
let stack1 = []
//将stack1里面的元素放到stack2尾部，接着pop出来就相当于stack1的头部了
let stack2 = []
export function push(node: number) {
    // write code here
    stack1.push(node)
}
```

```
/**
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 *
 * @param 无
 * @return int整型
 */
export function pop(): number {
    // 将这个stack1的元素放到stack2里面
    if(stack2.length === 0){
        while(stack1.length !== 0){
            stack2.push(stack1.pop())
        }
    }
    return stack2.pop()
}
```

包含min函数的栈

字符串解码

```

```

动态规划

基本类型:

背包问题、打家劫舍、股票问题、子序列问题

解题思路:

1. dp数组以及下标的的含义
2. 递推公式
3. dp数组初始化
4. 遍历顺序
5. 打印dp数组(来进行调试，寻找错误)

斐波那契数列

```
/**
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 *
 * @param n int整型
```

```
 * @return int整型
 */
export function Fibonacci(n: number): number {
    const dp = []
    dp[0] = 1
    dp[1] = 1
    for(let i = 2;i < n;i++){
      dp[i] =   dp[i - 1] + dp[i - 2]
    }
    return dp[n-1]
}
```

减绳子

描述

给你一根长度为 n 的绳子，请把绳子剪成整数长的 m 段（ m 、 n 都是整数， n > 1 并且 m > 1 ， m <= n ），每段绳子的长度记为 k[1],...,k[m] 。请问 k[1]*k[2]*...*k[m] 可能的最大乘积是多少？例如，当绳子的长度是 8 时，我们把它剪成长度分别为 2、3、3 的三段，此时得到的最大乘积是 18 。

数据范围： $2 \le n \le 60$2≤n≤60
进阶：空间复杂度 $O(1)$O(1) ，时间复杂度 $O(n)$O(n)

输入描述：

输入一个数n，意义见题面。

返回值描述：

输出答案。

```
function cutRope(number)
{
    // write code here
    if(number <= 3)return number - 1
    const dp = new Array(number + 1).fill(0)
    dp[1] = 1
    dp[2] = 2
    dp[3] = 3
    dp[4] = 4
    for(let i = 5;i <= number;i++){
        for(let j = 1;j < i;j++){
            dp[i] = Math.max(dp[i],j*dp[i - j])
        }
    }
    return dp[number]
}
module.exports = {
    cutRope : cutRope
};
```

爬楼梯

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

杨辉三角

```
function generate(numRows: number): number[][] {
    let dp = []
    for(let i = 0;i < numRows;i++){
        dp[i] = new Array(i + 1).fill(1)
        for(let j = 1;j < i;j++){
            dp[i][j] = dp[i - 1][j - 1] + dp[i-1][j]
        }
    }
    return dp
};
```

乘积最大子数组

给你一个整数数组 `nums` ，请你找出数组中乘积最大的非空连续

子数组

（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

测试用例的答案是一个 **32-位** 整数。

```
function maxProduct(nums: number[]): number {
    let result = nums[0]
    let prevMax = nums[0],prevMin = nums[0]
    for(let i = 1;i < nums.length;i++){
        const temp1 = prevMax * nums[i]
        const temp2 = prevMin * nums[i]
        prevMax = Math.max(temp1,temp2,nums[i])
        prevMin = Math.min(temp1,temp2,nums[i])
        result = Math.max(result,prevMax)
    }
    return result
};
```

不同路径

一个机器人位于一个 `m x n` 网格的左上角 （起始点在下图中标记为 “Start” ）。

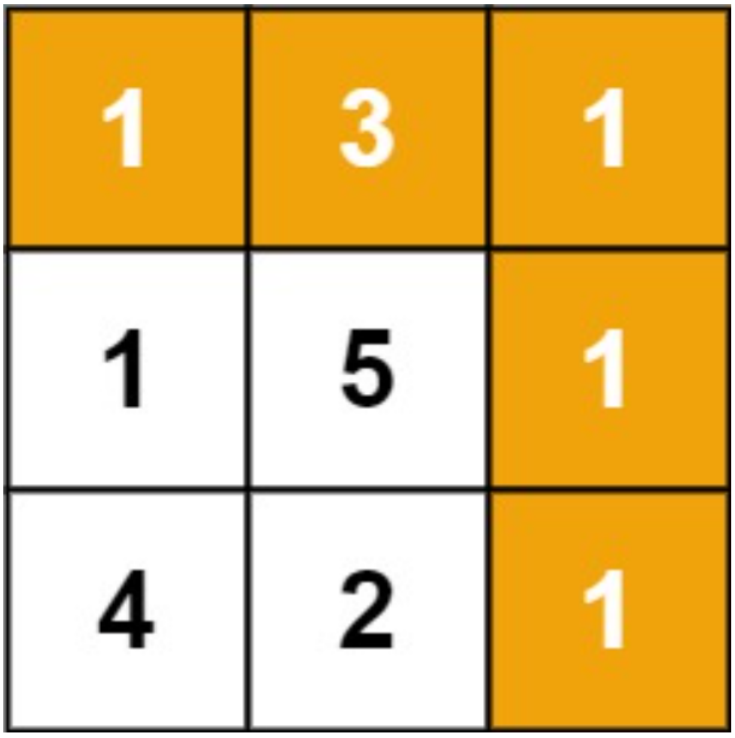
机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为 “Finish” ）。

问总共有多少条不同的路径?

```
/**
 * @param {number} m
 * @param {number} n
 * @return {number}
 */
var uniquePaths = function(m, n) {

    const dp = new Array(m).fill(0).map(()=>new Array(n).fill(0))
    for(let i = 0;i < m;i++){
        dp[i][0] = 1
    }
    for(let j = 0;j < n;j++){
        dp[0][j] = 1
    }
    for(let i = 1;i < m;i++){
        for(let j = 1;j < n;j++){
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1]
        }
    }
    return dp[m - 1][n - 1]
};
```

最小路径

示例 1：



> **输入：** grid = [[1,3,1],[1,5,1],[4,2,1]]
> **输出：** 7
> **解释：** 因为路径 1→3→1→1→1 的总和最小。

```
/**
 * @param {number[][]} grid
 * @return {number}
 */
var minPathSum = function(grid) {
    const m = grid.length,n = grid[0].length
    for(let i = 0;i < m;i++){
        for(let j = 0;j < n;j++){
            if(i === 0 && j === 0)continue;
            else if(i === 0)grid[i][j] = grid[i][j - 1] + grid[i][j]
            else if(j === 0)grid[i][j] = grid[i - 1][j] + grid[i][j]
            else{
                grid[i][j] = Math.min(grid[i][j - 1],grid[i - 1][j]) + grid[i][j]
            }
        }
    }
    return grid[m - 1][n - 1]
};
```

编辑路径

```
/**
 * @param {string} word1
 * @param {string} word2
 * @return {number}
 */
var minDistance = function(word1, word2) {
    const len1  = word1.length
    const len2 = word2.length
    const dp = Array.from(Array(len1 + 1),()=>Array(len2 + 1))
    dp[0][0] = 0
    for(let i = 1;i <= len1;i++){
        dp[i][0] = dp[i - 1][0] + 1
    }
    for(let i = 1;i <= len2;i++){
        dp[0][i] = dp[0][i - 1] + 1
    }
    //接着进行循环
    for(let i = 1;i <= len1;i++){
        for(j = 1;j <= len2;j++){
            if(word1[i - 1] === word2[j - 1]){
                dp[i][j] = dp[i - 1][j - 1]
            }else{
                dp[i][j] = Math.min(dp[i-1][j-1],dp[i-1][j],dp[i][j-1]) + 1
            }
        }
    }
    return dp[len1][len2]
};
```

二分法

二分法

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number}
 */
var search = function(nums, target) {
    let left = 0,right = nums.length - 1
    while(left <= right){
        let mid = Math.floor((right - left) / 2) + left
        if(target === nums[mid]){
            return mid
        }else if(nums[mid] > target){
            right = mid - 1
```

```
        }else{
            left = mid + 1
        }

    }
    return -1
};
```

搜索插入位置

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

请必须使用时间复杂度为 `O(log n)` 的算法。

```
输入: nums = [1,3,5,6], target = 5
输出: 2
```

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number}
 */
var searchInsert = function(nums, target) {
    let left = 0
    let right = nums.length - 1
    while(left <= right){
        let mid = Math.floor(left +(right - left)/2)
        if(nums[mid] === target){
            return mid
        }else if(nums[mid] < target){
            left = mid + 1
        }else{
            right = mid - 1
        }
    }
    return left
};
```

旋转数组的最小数

```
/**
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 *
 * @param nums int整型一维数组
```

```
 * @return int整型
 */
export function minNumberInRotateArray(nums: number[]): number {
    // write code here
    let left = 0,right = nums.length - 1
    let mid = (left + right) / 2
    while(left < right){
         mid = (left + right) / 2
        //进行循环
        if(nums[mid] > nums[right]){
            //异常在右侧部分
            left = mid + 1
        }else if(nums[mid] < nums[right]){
            //没有出现异常，那么肯可能出现的就在左边了
            right = mid - 1
        }else{
            //一个一个的寻找
           return nums.sort((a,b)=>a-b)[0]
        }
    }
    return mid
}
```

搜索二维矩阵

```
/**
 * @param {number[][]} matrix
 * @param {number} target
 * @return {boolean}
 */
var searchMatrix= function(matrix, target) {
    if(!matrix.length) return false;
    let x = matrix.length - 1, y = 0;
    //设置边界
    while(x >= 0 && y < matrix[0].length){
        //从后开始找
        if(matrix[x][y] === target){
            return true;
        }else if(matrix[x][y] > target){
            //换一行
            x--;
        }else{
            //一个一个查找
            y++;
        }
    }
    return false;
};
```

在排序数组中查找元素的第一个和最后一个位置

给你一个按照非递减顺序排列的整数数组 `nums`，和一个目标值 `target`。请你找出给定目标值在数组中的开始位置和结束位置。

如果数组中不存在目标值 `target`，返回 `[-1, -1]`。

你必须设计并实现时间复杂度为 `O(log n)` 的算法解决此问题。

> **示例 1：**
>
> > **输入：** nums = [5,7,7,8,8,10], target = 8
> > **输出：** [3,4]
>
> **示例 2：**
>
> > **输入：** nums = [5,7,7,8,8,10], target = 6
> > **输出：** [-1,-1]
>
> **示例 3：**
>
> > **输入：** nums = [], target = 0
> > **输出：** [-1,-1]

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
var searchRange = function(nums, target) {
    if(!nums.length)return [-1,-1]
  let l = 0,r = nums.length - 1
  while(l <= r){
    const mid = Math.floor(l + (r-l) / 2)
    if(target === nums[mid]){
        let left = right = mid
        //左右查找
        while(target === nums[left]){
            left--
        }
        while(target === nums[right]){
            right++
        }
        return [left+1,right-1]
    }else if(target > nums[mid]){
```

```
            l = mid + 1
        }else{
            r = mid - 1
        }
    }
    return [-1,-1]
};
```

搜索旋转排序数组

整数数组 `nums` 按升序排列，数组中的值 **互不相同** 。

在传递给函数之前，`nums` 在预先未知的某个下标 `k`（`0 <= k < nums.length`）上进行了 **旋转**，使数组变为 `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]`（下标 **从 0 开始** 计数）。例如，`[0,1,2,4,5,6,7]` 在下标 `3` 处经旋转后可能变为 `[4,5,6,7,0,1,2]` 。

给你 **旋转后** 的数组 `nums` 和一个整数 `target` ，如果 `nums` 中存在这个目标值 `target` ，则返回它的下标，否则返回 `-1` 。

你必须设计一个时间复杂度为 `O(log n)` 的算法解决此问题。

```
function search(nums: number[], target: number): number {
    let left = 0,right = nums.length - 1,mid = 0
    while(left <= right){
        mid = Math.floor((left + right) / 2)
        if(nums[mid] ===  target)return mid
        if(nums[mid] < nums[right]){
            //右边是有序的
            if(nums[mid] < target && target <= nums[right]){
                left = mid + 1
            }else{
                right = mid -1
            }
        }else{
            if(nums[left] <= target && target < nums[mid]){
                right = mid - 1
            }else{
                left = mid + 1
            }
        }
    }
    return -1
};
```

寻找旋转排序数组中最小值

已知一个长度为 `n` 的数组，预先按照升序排列，经由 `1` 到 `n` 次 **旋转** 后，得到输入数组。例如，原数组 `nums = [0,1,2,4,5,6,7]` 在变化后可能得到：

- 若旋转 `4` 次，则可以得到 `[4,5,6,7,0,1,2]`
- 若旋转 `7` 次，则可以得到 `[0,1,2,4,5,6,7]`

注意，数组 `[a[0], a[1], a[2], ..., a[n-1]]` **旋转一次** 的结果为数组 `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]` 。

给你一个元素值 **互不相同** 的数组 `nums` ，它原来是一个升序排列的数组，并按上述情形进行了多次旋转。请你找出并返回数组中的 **最小元素** 。

你必须设计一个时间复杂度为 `O(log n)` 的算法解决此问题。

```
function findMin(nums: number[]): number {
    let left = 0,right = nums.length - 1,mid = 0
    while(left < right){
        mid =Math.floor( (left + (right - left) / 2))
        if(nums[mid]  > nums[right]){
            left = mid +1
        }else{
            //这个值可能是最小值也可能在左边
            right = mid
        }
    }
    return nums[left]

};
```

图论

```
void dfs(int[][] grid, int r, int c) {
    // 判断 base case
    if (!inArea(grid, r, c)) {
        return;
    }
    // 如果这个格子不是岛屿，直接返回
    if (grid[r][c] != 1) {
        return;
    }
    grid[r][c] = 2; // 将格子标记为「已遍历过」

    // 访问上、下、左、右四个相邻结点
    dfs(grid, r - 1, c);
    dfs(grid, r + 1, c);
    dfs(grid, r, c - 1);
    dfs(grid, r, c + 1);
```

```
}

// 判断坐标 (r, c) 是否在网格中
boolean inArea(int[][] grid, int r, int c) {
    return 0 <= r && r < grid.length
            && 0 <= c && c < grid[0].length;
}
```

如何避免这样的重复遍历呢？答案是标记已经遍历过的格子。以岛屿问题为例，我们需要在所有值为 1 的陆地格子上做 DFS 遍历。每走过一个陆地格子，就把格子的值改为 2，这样当我们遇到 2 的时候，就知道这是遍历过的格子了。也就是说，每个格子可能取三个值：

0 —— 海洋格子
1 —— 陆地格子（未遍历过）
2 —— 陆地格子（已遍历过）

岛屿数量

```
function numIslands(grid: string[][]): number {
    let res = 0
    for(let i = 0;i < grid.length;i++){
        for(let j = 0;j < grid[0].length;j++){
            //找到陆地
            if(grid[i][j] == "1"){
                //进行渲染
               area(grid,i,j)
               res++
            }
        }
    }
    return res
};
//来进行渲染区域
const area = (grid: string[][],r:number,c:number)=>{
    //如果超过边界返回0
    if(!isArea(grid,r,c)){
        return
    }
    if(grid[r][c]!= "1"){
        return
    }
    grid[r][c] = "2"
    //接着进行渲染
   area(grid,r-1,c)
    area(grid,r+1,c)
    area(grid,r,c-1)
     area(grid,r,c+1)
}
```

```
//处理边界
const isArea = (grid:string[][],r:number,c:number):boolean=>{
    return r >= 0&& r < grid.length && c >= 0 && c < grid[0].length
}
```

岛屿面积

```
function numIslands(grid: string[][]): number {
    let res = 0
    for(let i = 0;i < grid.length;i++){
        for(let j = 0;j < grid[0].length;j++){
            //找到陆地
            if(grid[i][j] == "1"){
                //进行渲染
                let a = area(grid,i,j)
                res = Math.max(res,a)
            }
        }
    }
    return res
};
//来进行渲染区域
const area = (grid: string[][],r:number,c:number)=>{
    //如果超过边界返回0
    if(!isArea(grid,r,c)){
        return 0
    }
    if(grid[r][c]!= "1"){
        return 0
    }
    grid[r][c] = "2"
    //接着进行渲染
    return 1 + area(grid,r-1,c) + area(grid,r+1,c) + area(grid,r,c-1) +
area(grid,r,c+1)
}

//处理边界
const isArea = (grid:string[][],r:number,c:number):boolean=>{
    return r >= 0&& r < grid.length && c >= 0 && c <= grid[0].length
}
```

矩阵中的路径

https://www.nowcoder.com/practice/2a49359695a544b8939c77358d29b7e6?tpId=265&tags=&title=&difficulty=0&judgeStatus=0&rp=1&sourceUrl=%2Fexam%2Foj%2Fta%3FtpId%3D13

```
/**
```

```
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 *
 * @param matrix char字符型二维数组
 * @param word string字符串
 * @return bool布尔型
 */
function hasPath( matrix ,  word ) {
    for(let i = 0;i < matrix.length;i++){
        for(let j = 0;j < matrix[0].length;j++){
            if(dfs(matrix,word,i,j,0)){
                return true
            }
        }
    }
    return false

}
//其中这个i为row索引，j为column索引，k为记录的长度
function dfs(matrix,word,i,j,k){
    //判断边界
    if(i  >= matrix.length || i < 0 || j >= matrix[0].length || j < 0 || matrix[i]
[j] !== word[k])return false
    //如果长度一样匹配成功
    if(k === word.length -1)return true
    //标记已经访问过的
    matrix[i][j] = '#'
    //查询
    const res  = dfs(matrix,word,i-1,j,k+1) || dfs(matrix,word,i+1,j,k+1) ||
dfs(matrix,word,i,j+1,k+1) || dfs(matrix,word,i,j-1,k+1)
    //回退
    matrix[i][j] = word[k]
    return res

}
module.exports = {
    hasPath : hasPath
};
```

贪心

跳跃游戏

给你一个非负整数数组 `nums` ，你最初位于数组的 **第一个下标** 。数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标，如果可以，返回 `true` ；否则，返回 `false` 。

```
function canJump(nums: number[]): boolean {
    let end = nums.length - 1
    for(let i = nums.length - 2;i >= 0;i--){
        if(end - i <= nums[i]){
            end = i
        }
    }
    return end === 0
};
```

跳跃游戏二

给定一个长度为 `n` 的 **0 索引**整数数组 `nums`。初始位置为 `nums[0]`。

每个元素 `nums[i]` 表示从索引 `i` 向前跳转的最大长度。换句话说，如果你在 `nums[i]` 处，你可以跳转到任意 `nums[i + j]` 处:

- `0 <= j <= nums[i]`
- `i + j < n`

返回到达 `nums[n - 1]` 的最小跳跃次数。生成的测试用例可以到达 `nums[n - 1]`

> 输入: nums = [2,3,1,1,4]
> 输出: 2
> 解释: 跳到最后一个位置的最小跳跃数是 2。
> 从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

```
/
    **
    就是凑那个末尾，从后往前进行来凑
/

function jump(nums: number[]): number {
    let cur = nums.length - 1,step = 0
    while(cur > 0){
        for(let left = 0; left < cur;left++){
            if(nums[left] + left >= cur){
                cur = left;
                step++
            }
        }
    }
    return step;
};
```

划分字母区间

给你一个字符串 `s` 。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。

注意，划分结果需要满足：将所有划分结果按顺序连接，得到的字符串仍然是 `s` 。

返回一个表示每个字符串片段的长度的列表。

```
function partitionLabels(s: string): number[] {
    //统计每一个字符最后出现的位置
    //从头遍历字符，并更新字符的最远出现下标，如果找到字符最远出现位置下标和当前下标相等了，则找到了分割点
    let hash = {}
    for(let i = 0;i < s.length;i++){
        hash[s[i]] = i
    }
    const result = []
    let left  = 0,right = 0
    for(let i = 0;i < s.length;i++){
        right = Math.max(right,hash[s[i]])
        if(right === i){
            result.push(right - left + 1)
            left = i + 1
        }
    }
    return result
};
```

买卖股票的最佳时机

给定一个数组 `prices` ，它的第 `i` 个元素 `prices[i]` 表示一支给定股票第 `i` 天的价格。

你只能选择 **某一天** 买入这只股票，并选择在 **未来的某一个不同的日子** 卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 `0` 。

```
function maxProfit(prices: number[]): number {
    let result = 0
    let preMin = prices[0]
    for(let p of prices){
        //更新最大的结果
        result = Math.max(result,p - preMin)
        //更新最小的值
        preMin = Math.min(preMin,p)
    }
    return result
};
```

字符串

比较版本**

1.将两个version以.切割为数组
2.判断两者之间最长度值
3.循环判断
3.1 如果值相同就continue开始下一个新循环,因为当前是同版本号,只能往下走小版本号对比
3.2 利用parseInt()转换数值进行判断,可忽略前导零的情况
3.0 默认返回

```
/**
 * @param {string} version1
 * @param {string} version2
 * @return {number}
 */
var compareVersion = function(version1, version2) {
    const first = version1.split('.')
    const second = version2.split('.')
    const maxLength = Math.max(first.length,second.length)
    for(let i = 0;i < maxLength;i++){
        let current = first[i] || 0
        let next = second[i] || 0
        if(first[i] === second[i]){
            continue
        }
        if(parseInt(current) > parseInt(next)){
            return 1
        }else if(parseInt(current) < parseInt(next)){
            return -1
        }
    }
    return 0
};
```

整数反转

```
/**
 * @param {number} x
 * @return {number}
 */
var reverse = function(x) {
    let result = 0
    while(x !== 0){
        result = result * 10 + x % 10
        x = (x / 10) | 0
    }
    return (result | 0) === result?result:0
};
```

排序算法

一般在面试中最常考的是**快速排序和归并排序**，并且经常有面试官要求现场写出这两种排序的代码。对这两种排序的代码一定要信手拈来才行。还有插入排序、冒泡排序、堆排序、基数排序、桶排序等。面试官对于这些排序可能会要求比较各自的优劣、各种算法的思想及其使用场景。还有要会分析算法的时间和空间复杂度。

冒泡排序

时间复杂度O(n^2)

```
function bubbleSort(arr){
    for(let i = 0;i < arr.length - 1;i++){
        for(let j = 0;j < arr.length - 1 - i;j++){
            if(arr[j] > arr[j + 1]){
                swap(arr,j,j+1)
            }
        }
    }
}
function swap(arr,i,j){
    let temp = arr[i]
    arr[i] = arr[j]
    arr[j] = temp
}
```

选择排序

选择排序的思想是：双重循环遍历数组，每经过一轮比较，找到最小元素的下标，将其交换至首位。

```
function selectionSort(arr){
    let minIndex
    for(let i = 0;i < arr.length - 1;i++){
        minIndex = i
        for(let j = i + 1;j < arr.length;j++){
            if(arr[minIndex] > arr[j]){
                minIndex = j
            }
        }
        //进行交换到首位
        let temp = arr[i]
        arr[i] = arr[minIndex]
        arr[minIndex] = temp
    }
}
```

- 都是两层循环，时间复杂度都为 O(n2)$O(n2)$;
- 都只使用有限个变量，空间复杂度 O(1)$O(1)$。

其中冒泡排序是稳定的，选择排序不稳定。

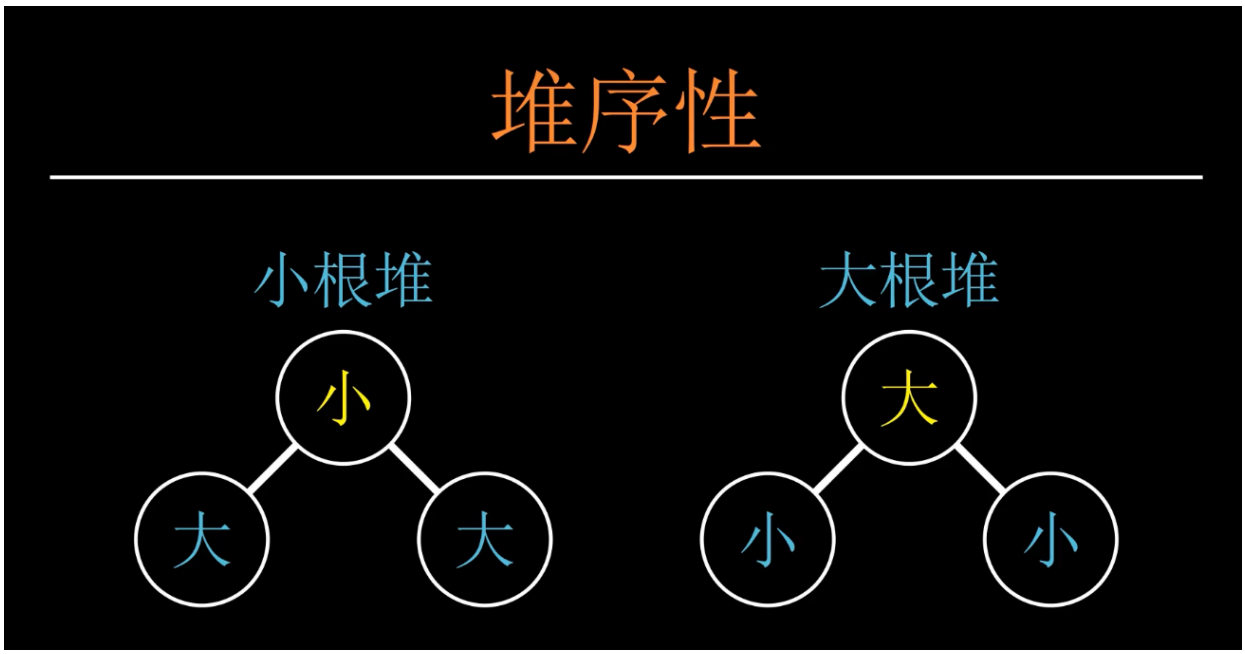
稳定的定义：

> 假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，r[i] = r[j]，且 r[i] 在 r[j] 之前，而在排序后的序列中，r[i] 仍在 r[j] 之前，则称这种排序算法是稳定的；否则称为不稳定的。

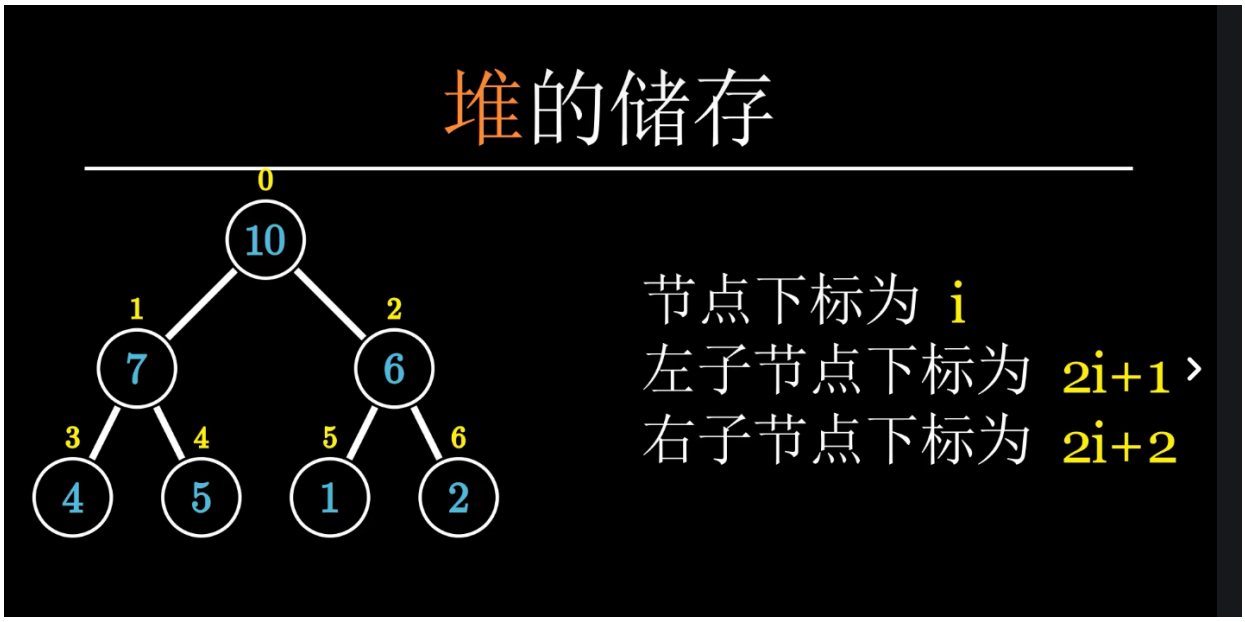
选择排序中，最小值和首位交换的过程可能会破坏稳定性。比如数列：[2, 2, 1]，在选择排序中第一次进行交换时，原数列中的两个 2 的相对顺序就被改变了，因此，我们说选择排序是不稳定的。

堆排序

堆必须是一个完全二叉树



堆的存储



建堆：自顶向下

1. 插入堆
2. 上滤

自底向上

对每一个父节点进行下滤

堆排序：使用大堆根的时候是正序的，小根堆的时候是倒序的

大顶堆和小顶堆特别适合存储一个大的数据集中的前K个频率高的

时间复杂度

初始化建堆（`buildMaxHeap`）和重建堆（`maxHeapify`，直译为大顶堆化）。所以时间复杂度要从这两个方面分析。

堆排序是一个优秀的排序算法，但是在实际应用中，快速排序的性能一般会优于堆排序，我们将在下一节介绍快速排序的思想。

```
function heapSort(arr){
    //初建堆
    buildMaxHeap(arr)
    for(let i = arr.length - 1;i > 0;i--){
        //将这个最大值放在后面
        swap(arr,0,i)
        //调整剩余的数组满足大顶堆
        maxHeapify(arr,0,i)
    }
}
function buildMaxHeap(arr){
    //从后面的最后一个非叶子节点往上滤
    for(let i = Math.floor(arr.length / 2) - 1;i >= 0;i--){
        //进行堆调整
        maxHeapify(arr,i,arr.length)
    }
```

```
}
function maxHeapify(arr,i,HeapSize){
    let l = i * 2 + 1
    let r = l + 1
    let largest = i
    if( l < HeapSize && arr[l] > arr[largest]){
        largest = l
    }
    if(r < HeapSize && arr[r] > arr[largest]){
        largest = r
    }
    //接着判断这个最大的索引是不是刚刚的
    if(largest !== i){
        swap(arr,i,largest)
        //再次调整大顶堆
        maxHeapify(arr,largest,HeapSize)
    }

}
function swap(arr,i,j){
    let temp  = arr[i]
    arr[i] = arr[j]
    arr[j] = temp
}
```

数组中的第K个最大元素

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */
var findKthLargest = function(nums, k) {
    buildHeap(nums)
    //调整k-1次
    for(let i = nums.length - 1;i > nums.length - k;i--){
        swap(nums,0,i)
        //重新构建大顶堆
        maxHeap(nums,0,i)
    }
    return nums[0]
};
function buildHeap(nums){
    for(let i = Math.ceil(nums.length / 2) - 1;i >= 0 ;i--){
        maxHeap(nums,i,nums.length)
    }
}
function maxHeap(arr,i,heapSize){
    let l = 2 * i + 1
    let r = l + 1
    let largest = i
```

```
    if(l < heapSize&&arr[l] > arr[largest]){
        largest = l
    }
    if(r < heapSize&&arr[r] > arr[largest]){
        largest = r
    }
    if(largest !== i){
        swap(arr,i,largest)
        maxHeap(arr,largest,heapSize)
    }

}

function swap(arr,i,j){
    let temp = arr[i]
    arr[i] = arr[j]
    arr[j] = temp
}
```

快速排序

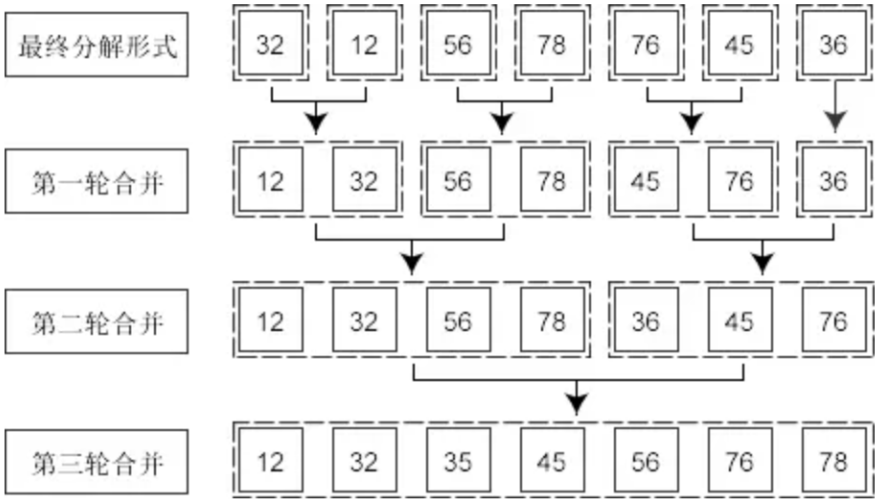
快速排序算法的基本思想是：

从数组中取出一个数，称之为基数（pivot）
遍历数组，将比基数大的数字放到它的右边，比基数小的数字放到它的左边。遍历完成后，数组被分成了左右两个区域
将左右两个区域视为两个数组，重复前两个步骤，直到排序完成

```
function quickSort(arr){
    if(arr.length <= 1)return arr
    let pivotIndex = Math.floor(arr.length / 2)
    let pivot = Math.splice(pivotIndex,1)[0]
    const left = []
    const right = []
    for(let 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))
}
```

尽管快速排序在最差情况下的时间复杂度是 O(n2)，但在实际应用中，通过合理选择基准元素（如随机选择或“三数取中”法），快速排序通常表现得非常高效，接近其平均时间复杂度 O(nlogn)。

归并排序




```
function merge(leftArr,rightArr){
    let result = []
    while(leftArr.length > 0&&rightArr.length >0){
        if(leftArr[0] < rightArr[0]){
            result.push(leftArr.shift())
        }else{
            result.push(rightArr.shift())
        }
    }
    return result.concat(leftArr).concat(rightArr)
}
function mergeSort(array){
    if(arr.length === 1)return array
    let middle = Math.floor(array.length / 2)
    let left = array.slice(0,middle)
    let right = array.slice(middle)
    return merge(mergeSort(left),mergeSort(right))
}
```

归并排序的时间复杂度始终为 O(nlogn)，无论输入数据的初始顺序如何。这使得归并排序在处理大规模数据时表现得非常稳定。然而，由于其空间复杂度为 O(n)，在内存使用方面可能不如一些原地排序算法（如快速排序）高效。

二进制中1的个数

输入一个整数 n ，输出该数32位二进制表示中1的个数。其中负数用补码表示。

```
/**
 * 代码中的类名、方法名、参数名已经指定，请勿修改，直接返回方法规定的值即可
 *
 *
```

```
 * @param n int整型
 * @return int整型
 */
export function NumberOf1(n: number): number {
    let num = 0
    while(n){
        num++
        //将一个数的二进制的后面的1变为0
        n = n & (n - 1)
    }
    return num

}
```

大数相加(字符串相加)

给定两个字符串形式的非负整数 `num1` 和 `num2` ，计算它们的和并同样以字符串形式返回。

你不能使用任何内建的用于处理大整数的库（比如 `BigInteger`），也不能直接将输入的字符串转换为整数形式。

```
function addStrings(num1: string, num2: string): string {
    const maxLength = Math.max(num1.length,num2.length)
    //进行填充
    let str1 = num1.padStart(maxLength,'0')
    let str2 = num2.padStart(maxLength,'0')
    let temp = 0//记录每次相加的结果
    let flag = 0//记录是否进位
    let result = ''
    //接着循环相加
    for(let i = maxLength - 1;i >= 0;i--){
        temp = Number(str1[i]) + Number(str2[i])+ flag
        flag = Math.floor(temp / 10)
        result = temp % 10 + result
    }
    //判断这个flag是否为1
    if(flag === 1){
        result = '1' + result
    }
    return result
};
```

字符串解码*

```
输入：s = "3[a]2[bc]"
输出："aaabcbc"
```

岛屿的数量，求出每个岛屿的面积*

LRU缓存

```
/**
 * @param {number} capacity
 */
var LRUCache = function(capacity) {
    this.limit = capacity
    this.cache = new Map()
};

/**
 * @param {number} key
 * @return {number}
 */
LRUCache.prototype.get = function(key) {
    let temp
    if(this.cache.has(key)){
        temp = this.cache.get(key)
        this.cache.delete(key)
        this.cache.set(key,temp)
    }
    return temp??-1
};

/**
 * @param {number} key
 * @param {number} value
 * @return {void}
 */
LRUCache.prototype.put = function(key, value) {
    if(this.cache.has(key)){
        this.cache.delete(key)
    }
    this.cache.set(key,value)

    if(this.cache.size > this.limit){
        this.cache.delete(this.cache.keys().next().value)
    }
};

/**
 * Your LRUCache object will be instantiated and called as such:
 * var obj = new LRUCache(capacity)
 * var param_1 = obj.get(key)
 * obj.put(key,value)
 */
```

下一个排序

整数数组的一个 **排列** 就是将其所有成员以序列或线性顺序排列。

- 例如，`arr = [1,2,3]` ，以下这些都可以视作 `arr` 的排列：`[1,2,3]`、`[1,3,2]`、`[3,1,2]`、`[2,3,1]` 。

整数数组的 **下一个排列** 是指其整数的下一个字典序更大的排列。更正式地，如果数组的所有排列根据其字典顺序从小到大排列在一个容器中，那么数组的 **下一个排列** 就是在这个有序容器中排在它后面的那个排列。如果不存在下一个更大的排列，那么这个数组必须重排为字典序最小的排列（即，其元素按升序排列）。

- 例如，`arr = [1,2,3]` 的下一个排列是 `[1,3,2]` 。
- 类似地，`arr = [2,3,1]` 的下一个排列是 `[3,1,2]` 。
- 而 `arr = [3,2,1]` 的下一个排列是 `[1,2,3]` ，因为 `[3,2,1]` 不存在一个字典序更大的排列。

给你一个整数数组 `nums` ，找出 `nums` 的下一个排列。

必须 **原地** 修改，只允许使用额外常数空间。

```
/**
 Do not return anything, modify nums in-place instead.
 */
function nextPermutation(nums: number[]): void {
    let i = nums.length - 2
    //寻找第一个小于右邻居的数
    while(i >= 0 && nums[i] > nums[i + 1]){
        i--
    }
    if(i >= 0){
        let j = nums.length - 1
        //寻找第一个右边大于i的值
        while(j >= 0&& nums[j] <= nums[i]){
            j--
        }
        //进行交换
        [nums[i],nums[j]] = [nums[j],nums[i]]
    }
    //如果没走上面的就说明是递减的
    let l = i+1
    let r = nums.length - 1
    while(l < r){
        [nums[l],nums[r]] = [nums[r],nums[l]]
        l++
        r--
    }
};
```

寻找重复数

给定一个包含 `n + 1` 个整数的数组 `nums` ，其数字都在 `[1, n]` 范围内（包括 `1` 和 `n`），可知至少存在一个重复的整数。

假设 `nums` 只有 **一个重复的整数** ，返回 **这个重复的数** 。

你设计的解决方案必须 **不修改** 数组 `nums` 且只用常量级 `O(1)` 的额外空间

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var findDuplicate = function(nums) {
    const map = new Map()
    for(let i = 0;i < nums.length;i++){
        if(map.has(nums[i])){
            return nums[i]
        }else{
            map.set(nums[i],1)
        }
    }
};
```