

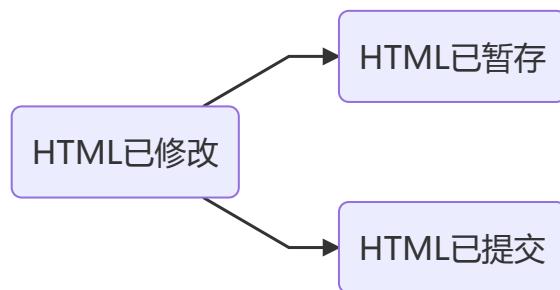
什么是Git?

git是一个分布式的版本控制软件——作者Linus

什么是版本控制?

多人协作，共同修改，修改版本

Git提交流程：



提交本低修改文件到暂存区

```
git add +文件名
```

将文件放到提交区

```
git commit -m ""
```

查看仓库的提交状态

```
git status
```

```
Changnie@DESKTOP-IVKQ1RP MINGW64 ~/learn_git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

```
git log
```

查看提交记录：

```
Changnie@DESKTOP-IVKQ1RP MINGW64 ~/learn_git (master)
$ git log
commit b2b272aabdb926c32e8f5e437f404b83f967e78f (HEAD -> master)
Author: Changnie <yingshuachuixuexiaoyi@126.com>
Date:   Wed Jan 27 18:59:38 2021 +0800

    web2.0

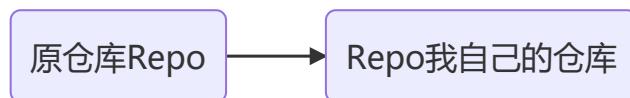
commit ae3fca46c475af87c1860fe06380fa6bb9b06885
Author: Changnie <yingshuachuixuexiaoyi@126.com>
Date:   Wed Jan 27 18:57:18 2021 +0800

    web1.0

commit 620e592c4a85cdd308b744cf5f96d9f3380405fd
Author: Changnie <yingshuachuixuexiaoyi@126.com>
Date:   Wed Jan 27 18:44:07 2021 +0800

    creat learn_git.html
```

训练营作业提交流程



原仓库与现仓库之间彼此独立，原仓库不会影响到fork的仓库

如何高效学习算法和数据结构

三分看视频理解，七分联系

视频

最佳办法：1.5~2.0倍速播放、难点（暂定+反复）

最差办法：类似看剧，原速看完，仅一遍

类比

Keep健身

Leetcode高效刷题

刷哪些题？

1. 高频题目

leetcode热题100

剑指offer 67

GitHub: <https://github.com/CyC2018/CS-Notes/blob/master/notes/Leetcode%20%E9%A2%98%E8%A7%A3%20-%20%E7%9B%AE%E5%BD%95.md>

2. 针对弱项刷题 ---> 刷20道

二分查找/dp/bfs/dfs/字符串匹配算法

3. 面对面试的公司刷题

面试公司的题库

刷题三部曲

1. 选择一门合适的语言
2. 深入理解基础的数据结构

知道具体每一个步骤是干什么的，例如二分，使用场景是哪里，时间复杂度

推荐网站：<https://visualgo.net/zh/bst>

3. 按照特定模块进行系统性刷题

按照不同的算法（大话数据结构）

例如二分

一，二分查找是什么，是干什么用的，二分的算法原理

二，然后做相应的类型题，在做题过程中熟悉二分

对于同类型题目，一定要举一反三，触类旁通。

例如：

1. 递增数组中，查找指定数字
2. 在递增数组中，查找相同的数出现的次数

由二分查找，延伸出二叉搜索树、AVL树、红黑树等等

4. 每天刷多长时间

- (1) 选择一个固定的时间段
- (2) 可以给自己一个做题时间，做不出就去看答案

5. 刷题

- (1) 理解题目意思
- (2) 看数据量选择算法
- (3) 判断临界条件
- (4) 有思路就写，没有思路就将自己知道的算法想一遍，看看哪个更匹配这道题
- (5) 看题解过程

最后

再次刷题

对之前做过的题目进行巩固

按照科学的方法刷题，事半功倍

承认其复杂性

1. 数据结构和算法有其客观存在的复杂度
2. 一遍或两遍不理解，很正常（**坚持五毒神掌**）
3. **脑图+反复（五毒神掌）是最为有效的办法**

摒弃“旧”习惯-最重要

不要死磕（传统方法）

五毒神掌（敢于放手，敢于死记硬背代码）

不懒于看高手代码（高赞题解、评论）

最佳方法

5分钟想不出来，直接看题解或者高票代码，用五毒神掌变成自己的东西。

这个过程：觉得自己很菜甚至有点自卑，但是有借势而起的感觉。

最差方式

看到题目自己单挑一下，不借助外部帮助自己解决；以为15-30分钟可以搞定，谁知道死磕了2-3小时或者一晚上，终于“通过”--> 精疲力尽，没精力学习高票程序答案，就开始做下一题（或放弃）

误区

1. 忘记践行五毒神掌（有规律地过遍数）
2. 谨记“通过”只是开始，关键要看高票代码和高质量题解！（官方题解较为复杂）先广度优先再深度优先
3. LeetCode题目不能只做一遍

线上课程

预习——基础知识自己预习和查看

课堂互动——跟着一起思考、回答问题

课后作业——按照切题办法

期待效果

职业顶尖级别-对于算法数据结构地理解

一线互联网公司面试

LeetCode 300+ 积累

学习步骤

1. chunk it up 切碎知识点
2. 刻意练习
3. feedback 总结反馈 (看高手代码-->学习)

数据结构知识脉络

1. 一维数据结构

基础：数组（Array），链表(linked list)

高级：栈stack，队列queue，双端队列deque，集合set，映射map（hash or map）

2. 二维数据结构

基础：树tree，图graph

高级：二叉搜索树binary search tree (red-black tree, AVL)，堆heap，并查集disjoint set，字典树Trie

3. 特殊数据结构

位运算Bitwise，布隆过滤器BloomFilter

LRU Cache

算法

If-else,switch --> branch

for, while loop --> Iteration

递归Recursion (Divide/Conquer/Backtree)

搜索search：深度优先搜索Depth first search，广度优先搜索 Breadth first search

动态规划 Dynamic Programming

二分查找 Binary Search

贪心 Greedy

数学 Math，几何 Geometry

刻意练习

过遍数 --> 五毒神掌

练习缺陷、弱点地方

不舒服，枯燥 --> 在成长

反馈

及时反馈

主动型反馈（自己去找）

- 高手代码（GitHub, leetcode）

- 第一视角

被动式反馈（高手给你指点）

- code review

- 评语

刷题技巧（五遍刷题法）

切题四件套

Clarification - 确保对题目的理解是正确的

Possible solutions - 把所有可能的方法过一遍（时间&空间复杂度）

Coding - 多写

Test cases - 测试样例

五遍刷题法（任何一个题目做5遍）

1. 刷题第一遍：

5分钟：读题+思考

直接看解法：注意！多解法，比较解法优劣

背诵、默写好的解法

2. 刷题第二遍：

马上自己写 -> LeetCode提交

多种解法比较、体会 -> 优化！

3. 刷题第三遍：

过了一天，再重复做题

不同解法的熟练程度 -> 专项练习

4. 刷题第四遍：

过了一周：反复回来练习相同题目

5. 刷题第五遍

面试前一周恢复性训练

小结

职业训练：拆分知识点、刻意练习、反馈

五步刷题法（五毒神掌）

做算法题的最大误区：只做一遍

训练环境设置

默认浏览器：chrome

Windows：Microsoft new terminal

编译器：Vscode

插件：Leetcode插件

LeetCode：去掉网址中的CN，可以去到国际站的讨论区的最优解（Most Votes）

Code Style - google research

if ()

快捷键

跳转到行头、行尾

全选粘贴

自顶向下的编程方式

现代代码写作方式：类似于新闻的方式；最关键的函数写在上面，子函数写在下面

以主干逻辑为主，先解决主干逻辑

算法的时间复杂度

Big O notation

O(1): Constant Complexity 常数复杂度

```
int n = 1000;
System.out.println("Hey - your input is: " + n);
```

O(log n): 对数复杂度

当n=4时，循环只会执行两次

```
for (int i = 1; i < n; i = i * 2) {
    System.out.println("Hey - I'm busy looking at: " + i);}
```

O(n): 线性时间复杂度

```

for (int i = 1; i <= n; i++) {
    System.out.println("Hey - I'm busy looking at: " + i);
}

```

$O(n^2)$: 平方

```

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        System.out.println("Hey - I'm busy looking at: " + i + " and " +
j); }
}

```

$O(n^3)$: 立方

$O(2^n)$: 指数

$O(n!)$: 阶乘

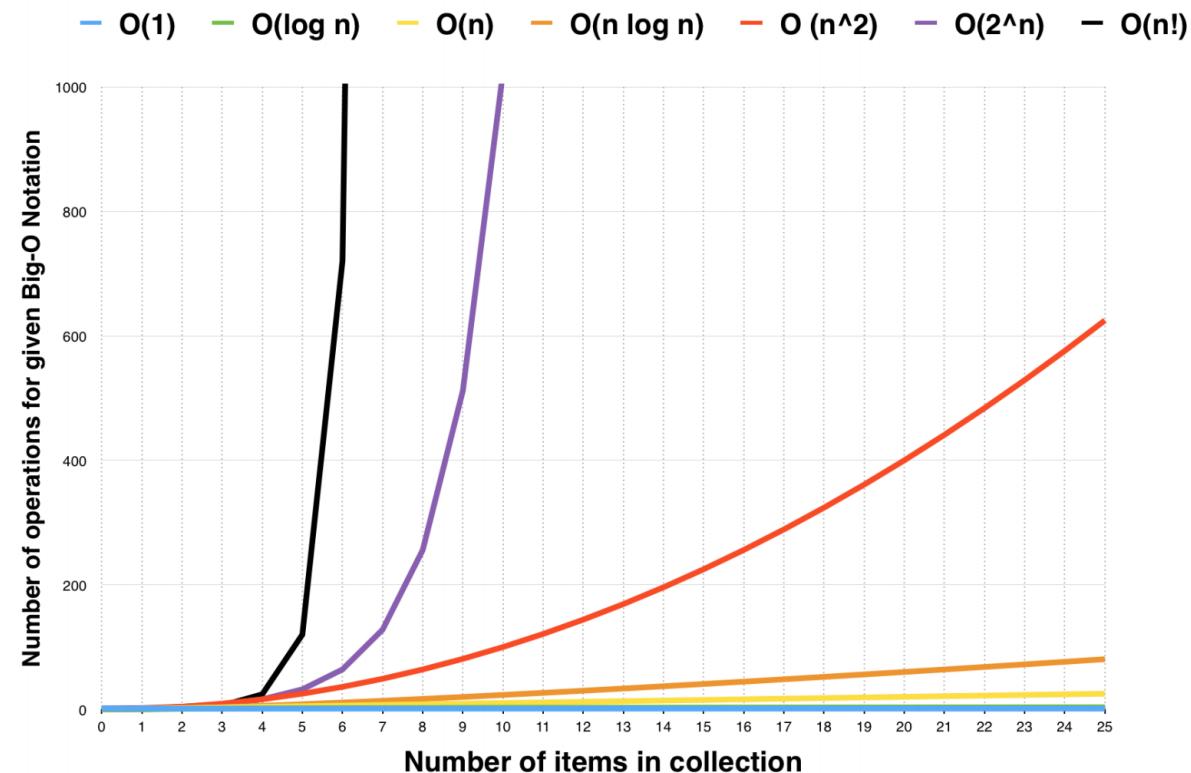
注意：只看最高复杂度的运算

$O(k^n)$: 指数

```

int fib(int n){
    if(n<2)return n;
    return fib(n-1) + fib(n-2);
}

```



写程序要对时间复杂度和空间复杂度有理解；用最简单的时间和空间复杂度写代码就是最优解

递归

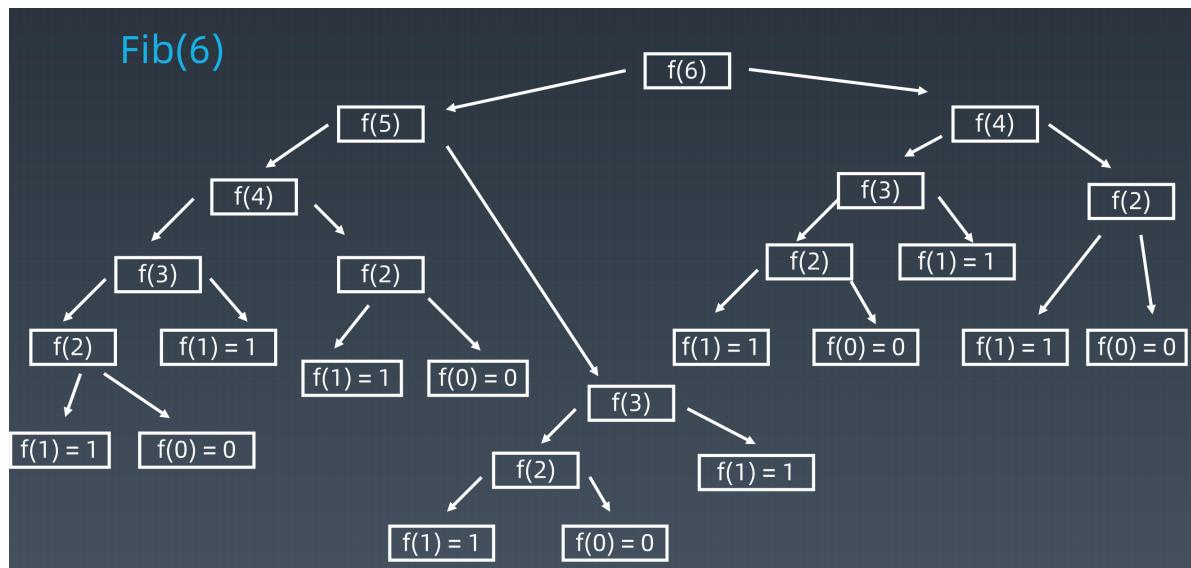
Fib: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

- $F(n) = F(n - 1) + F(n - 2)$

- 面试 (直接用递归)

```
int fib(int n) {  
    if (n < 2) return n;  
    return fib(n - 1) + fib(n - 2);  
}
```

时间复杂度计算：画出递归树



计算时间复杂度：查看return后面一共加了多少项

每层有两个节点，每增加一个节点，所需要计算的数是按 2^n 上升的

主定理（重要）

Application to common algorithms [edit]

Algorithm	Recurrence relationship	Run time	Comment
Binary search	$T(n) = T\left(\frac{n}{2}\right) + O(1)$	$O(\log n)$	Apply Master theorem case $c = \log_b a$, where $a = 1, b = 2, c = 0, k = 0$ [5]
Binary tree traversal	$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$	$O(n)$	Apply Master theorem case $c < \log_b a$ where $a = 2, b = 2, c = 0$ [5]
Optimal sorted matrix search	$T(n) = 2T\left(\frac{n}{2}\right) + O(\log n)$	$O(n)$	Apply the Akra-Bazzi theorem for $p = 1$ and $g(u) = \log(u)$ to get $\Theta(2n - \log n)$
Merge sort	$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$	$O(n \log n)$	Apply Master theorem case $c = \log_b a$, where $a = 2, b = 2, c = 1, k = 0$

思考

二叉树遍历 - 前序、中序、后序: $O(N)$

图的遍历: $O(N)$

搜索算法: DFS、BFS - $O(N)$

二分查找: $O(\log N)$

二叉树的每个节点访问且仅访问一次

图的每个节点访问且只访问一次

搜索算法 (DFS/BFS) 访问的节点只访问一次

二分查找: 参考递归

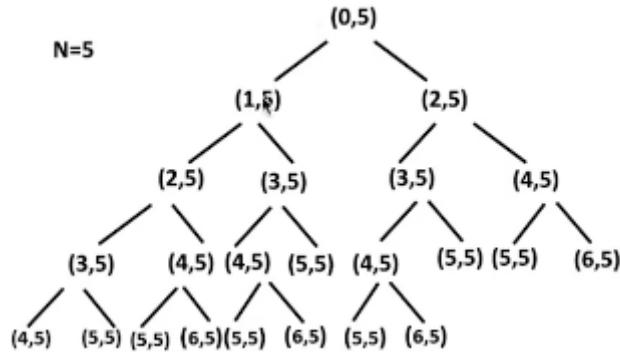
算法的空间复杂度

1. 数组的长度 (数组的长度就是空间的复杂度)
2. 递归的深度 (递归的深度就是空间的复杂度)

Example

爬楼梯问题的空间复杂度分析

```
class Solution {
    public int climbStairs(int n) {
        if(n==1)
            return 1;
        if(n==2)
            return 2;
        int a=1,b=2,temp;
        //从第三阶开始 都是前两阶的和
        // 因为要么爬一步上去(此时在n-1阶) 要么爬两步上去(在n-2阶)
        for(int i=3;i<=n;i++){
            temp=a;
            a=b;
            b=temp+a;
        }
        return b;
        //实际上就是斐波那契数列 但是递归会超出时间限制 climbStairs(n-1)+climbStairs(n-2);
    }
}
```



空间复杂度---->查看树的深度，树形递归有n层因此空间复杂度为O(n)

小结

常用工具配置

基本功和编程指法

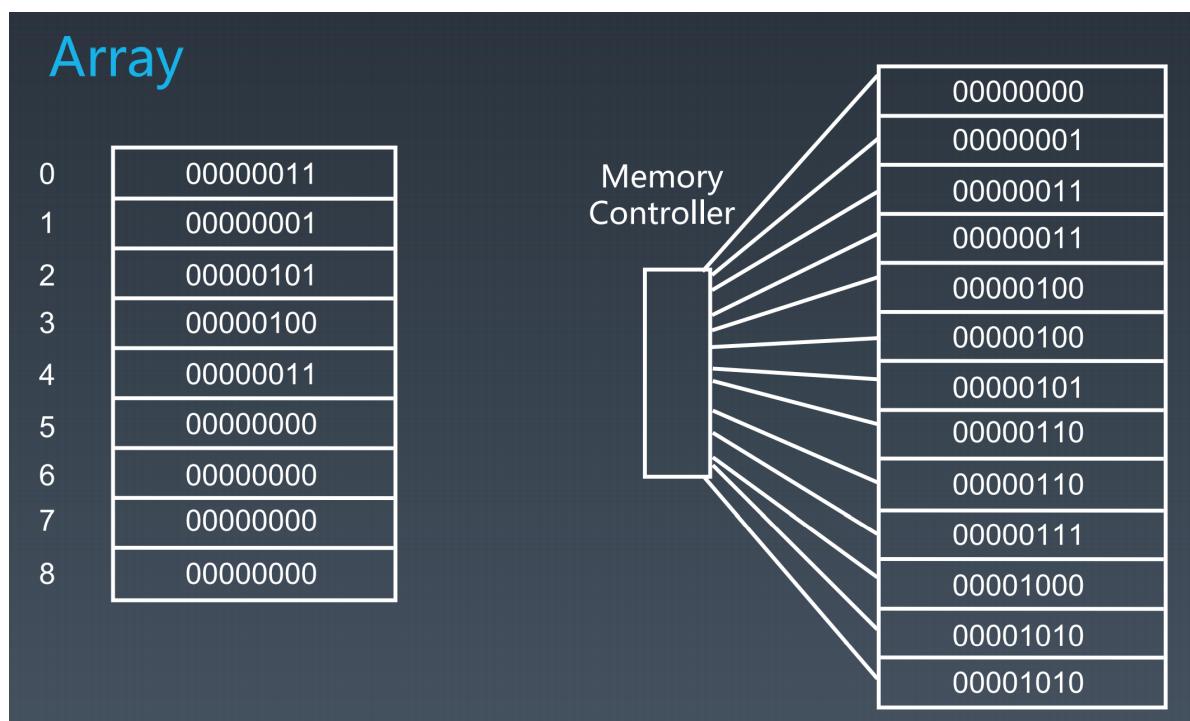
Week01

Array

javascript

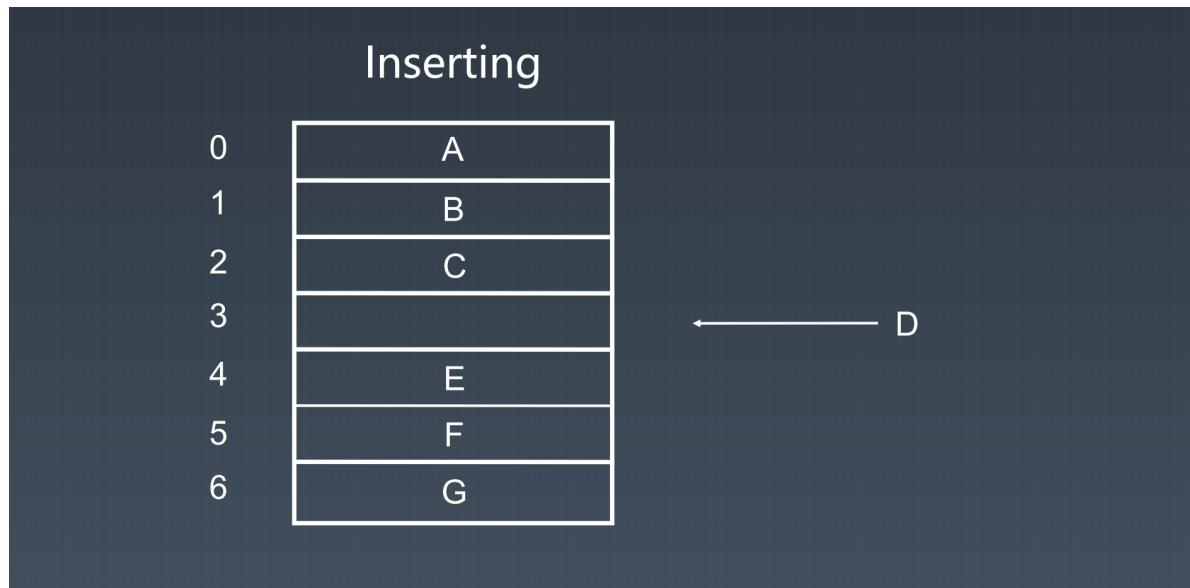
```
let x = [1,2,3];
```

内存地址



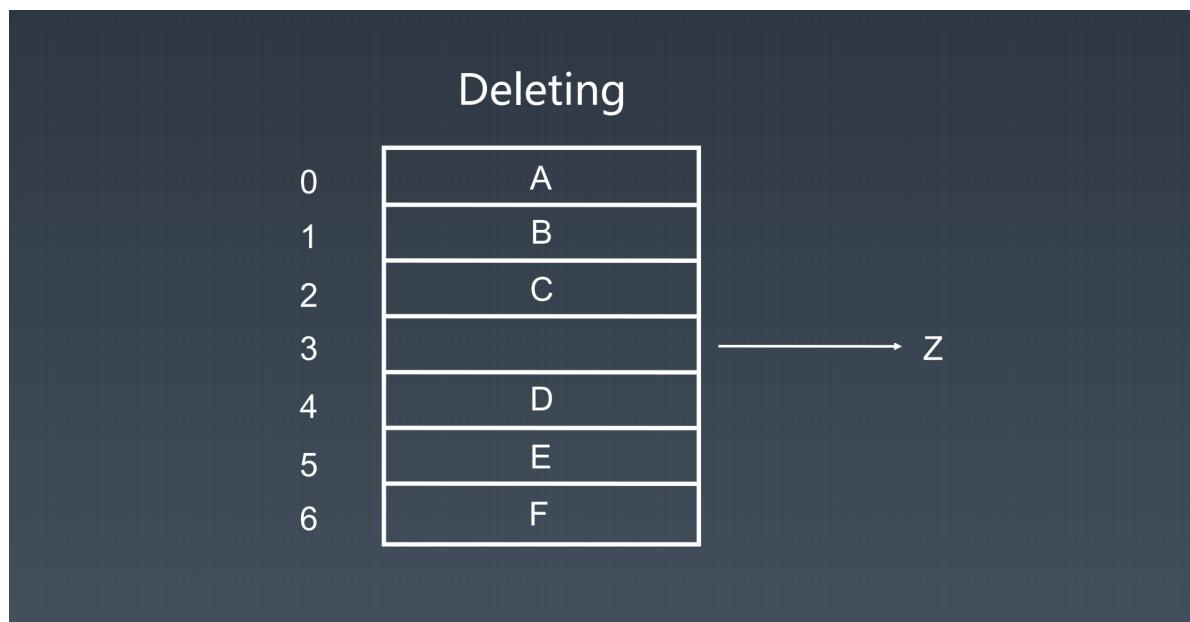
在内存中开辟了一段内存地址，访问任何一个元素的时间复杂度都是一样O(1)

Array增加元素

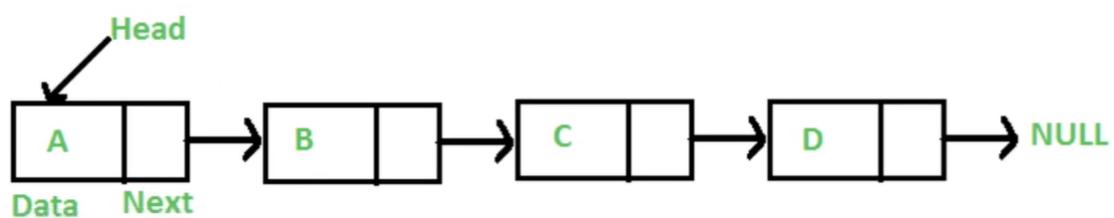


时间复杂度 $O(n)$ --> 好的情况下只需要插入末尾 $O(1)$, 坏的情况下要挪动整个数组

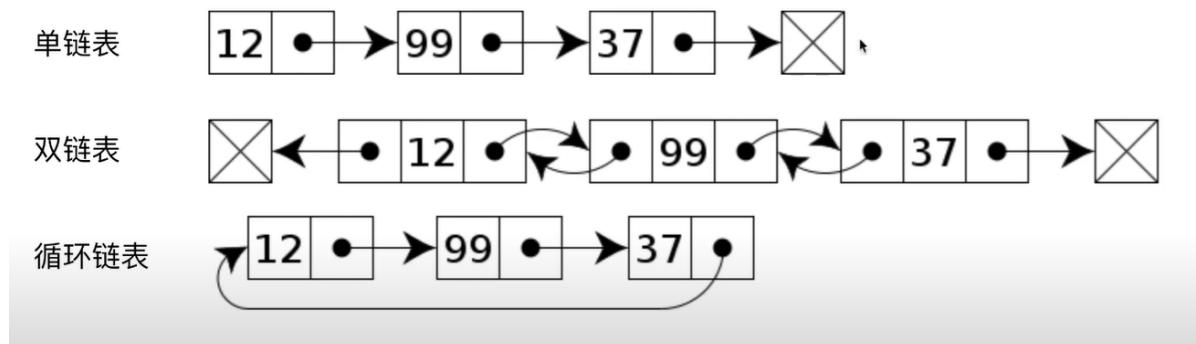
Array删除元素



Linked List



链表的类型



创建链表 (用Java实现)

```
public class LinkedList{  
    static class ListNode{ //每个节点包括value, next指针  
        int val;  
        ListNode next;  
        public ListNode(int val){  
            this.val = val;  
        }  
        ListNode head;  
        ListNode tail;  
        int size;  
        public LinkedList(){ //初始化链表  
            head = null;  
            tail = null;  
            size = 0;  
        }  
    }  
}
```

用JavaScript创建单链表

每个node中包含element和next指针

```
function LinkedList(){  
    var length = 0;  
    var head = null;  
  
    var Node = function (element) { //define Node --> Node includes element,next  
        this.element = element;  
        this.next = null;  
    };  
  
    this.size = function () {  
        return length;  
    };  
  
    this.head = function () {  
        return head;  
    };  
}
```

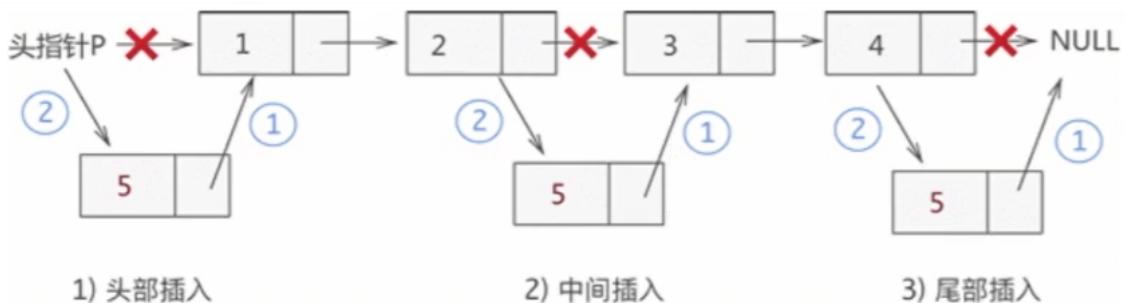
最后一个元素的next指针指向空

双向链表：前后都有指针

循环链表：尾指针指向头指针

Linked List 增加结点

插入元素的3种方式：



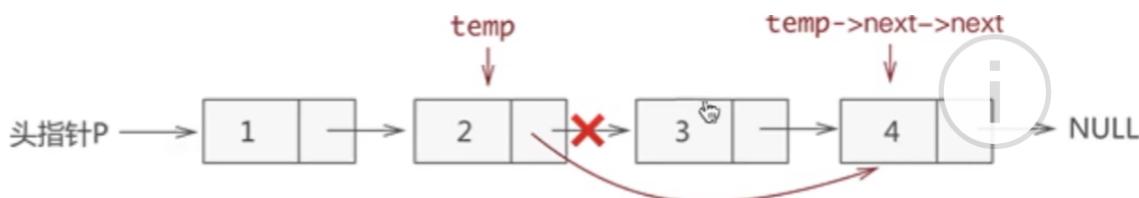
Java实现

```
public void insert(int position,int number){  
    if(position > size){  
        return;  
    }  
    ListNode newNode = new ListNode(number);  
    if(position == 0){ //采用头插法  
        newNode.next = head;  
        head = newNode;  
        if(tail == null){ //整个链表只插入了一个元素  
            tail = newNode;  
        }  
        size++;  
    }else if(position == size){//采用尾插法  
        this.append(number);  
    }else{  
        ListNode prev = head; //ListNode用来寻找插入位置  
        for(int i=0;i<position - 1;i++){  
            prev = prev.next;  
        }  
        ListNode next= prev.next;  
        newNode.next = next;  
        prev.next = newNode;  
    }  
}  
  
public void append(int number){  
    ListNode newNode = new ListNode(number);  
    if(tail == null){  
        tail = newNode;  
    }else{  
        tail.next = newNode;  
        tail = newNode;  
    }  
    size++;  
}
```

JavaScript创建

```
// Add function
this.add = function (element) {
    var node = new Node(element);
    if(head === null){ //如果head指向null说明原链表中没有结点，因此add结点后head指向新结点node
        head = node;
    }else{
        var currentNode = head;
        while(currentNode.next){
            currentNode = currentNode.next; //依次向后移动
        }
        currentNode.next = Node;
    }
    length++;
};
```

Linked List 删除结点



缺点：访问链表中的任何一个数的操作变得不简单

```
public void delete(int number){
    if(head==null && head.val == number){ //头节点就是想要删除的结点
        head = head.next;
        size--;
        if(size==0){ //删除结点后不存在其他节点
            tail=head;
        }
    }else{
        ListNode prev = head; //ListNode->去寻找要删除的节点
        ListNode cur = head;
        while(prev!=null && cur !=null){
            if(cur.val==number){
                if(cur==tail){ //要删除的元素在tail
                    tail = prev;
                }
                prev.next = cur.next;
                size--;
                return;
            }
            prev = prev.next;
            cur = cur.next; //没找到就一直继续
        }
    }
}
```

Linked List - Insert

```
public int search(int number){  
    ListNode cur = head;  
    for(int index=0;cur!=null;index++){  
        if(cur.val == number){  
            return index;  
        }  
        cur = cur.next;  
    }  
    return -1;  
}
```

Linked List - Update

```
public int search(int oldValue, int newValue){  
    ListNode cur = head;  
    for(int index=0;cur!=null;index++){  
        if(cur.val == oldValue){  
            cur.val = newValue;  
            return index;  
        }  
        cur = cur.next;  
    }  
    return -1;  
}
```

Linked List时间复杂度

时间复杂度

prepend	O(1)
append	O(1)
lookup	O(n)
insert	O(1)
delete	O(1)

prepend --> 在头节点插入

append --> 在末尾插入

insert & delete --> 只需要做两个操作 (常数次)

ArrayList 时间复杂度

时间复杂度

prepend	$O(1)$
append	$O(1)$
lookup	$O(n)$
insert	$O(n)$
delete	$O(n)$

LinkedList 时间复杂度

插入: $O(n)$
删除: $O(n)$
查找: $O(n)$
更新: $O(n)$

跳表

链表元素有序的时候



跳表的特点

注意：只能用于元素有序的情况。

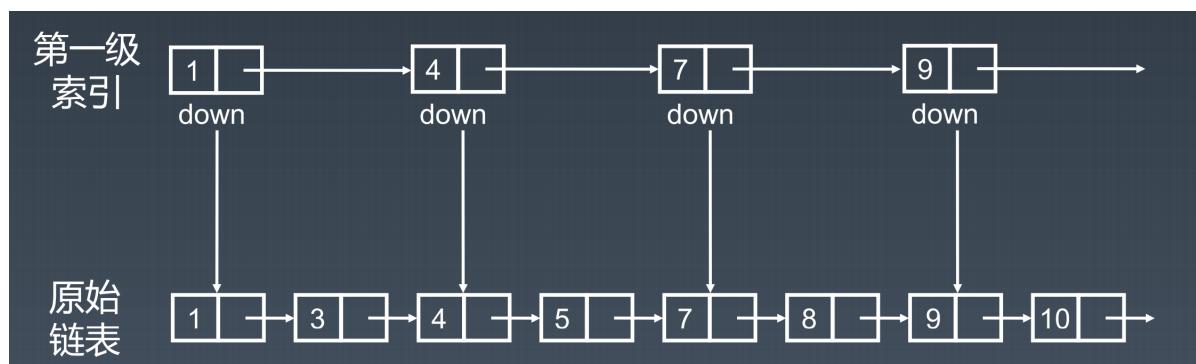
所以，跳表（skip list）对标的是平衡树（AVL Tree）和二分查找，是一种插入/删除/搜索都是 $O(\log n)$ 的数据结构。1989 年出现。

它最大的优势是原理简单、容易实现、方便扩展、效率更高。因此在一些热门的项目里用来替代平衡树，如 Redis、LevelDB 等。



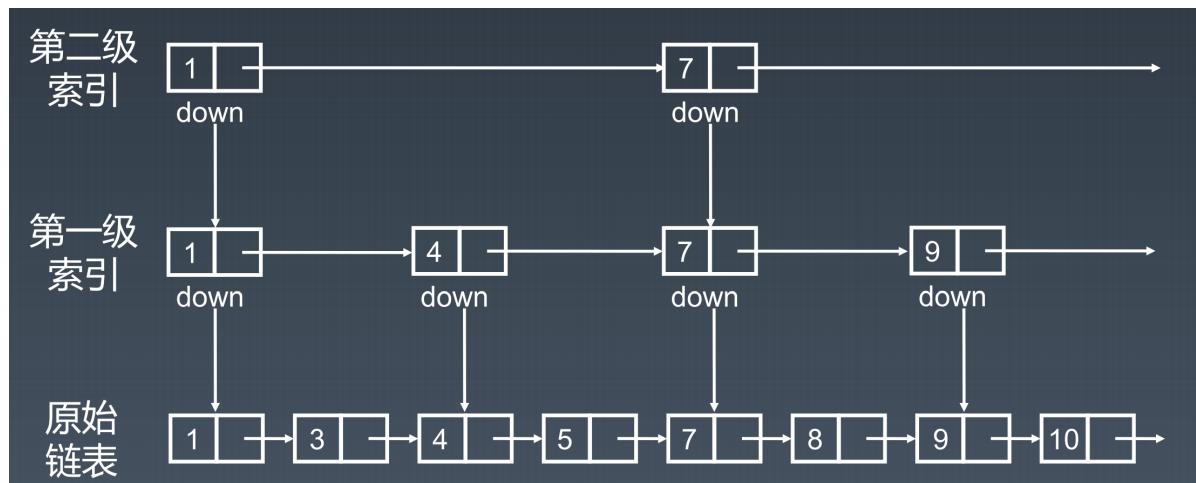
如何提高链表线性查找的效率？

添加第一级索引

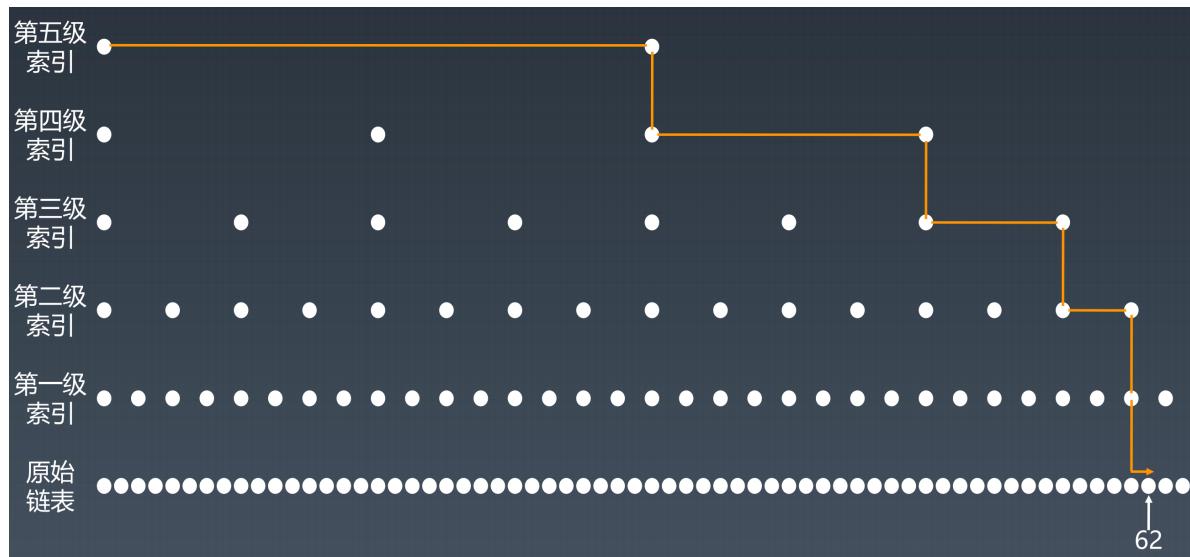


通过第一级索引对原始链表进行索引，元素就 locate 在大于且小于之间

添加第二级索引



添加多级索引



跳表查询的时间复杂度

$n/2$ 、 $n/4$ 、 $n/8$ 、第 k 级索引结点的个数就是 $n/(2^k)$

假设索引有 h 级，最高级的索引有 2 个结点。 $n/(2^h) = 2$ ，从而求得 $h = \log_2(n)-1$

跳表的空间复杂度的分析

原始链表大小为 n ，每 2 个结点抽 1 个，每层索引的结点数：

$$\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \dots, 8, 4, 2$$

原始链表大小为 n ，每 3 个结点抽 1 个，每层索引的结点数：

$$\frac{n}{3}, \frac{n}{9}, \frac{n}{27}, \dots, 9, 3, 1$$

空间复杂度是 $O(n)$

小结

- 数组、链表、跳表的原理和实现
- 三者的时间复杂度、空间复杂度
- 工程运用
- 跳表：升维思想 + 空间换时间

题目练习

移动零

练习步骤：

1. 5-10分钟：读题和思考
2. 有思路：自己开始做和写代码；不然，马上看题解
3. 默写背诵，熟练
4. 第二遍开始自己写

283. 移动零

难度 简单 931 ★ ⌂ 文章 ⌂

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。

示例：

输入： `[0,1,0,3,12]`

输出： `[1,3,12,0,0]`

说明：

1. 必须在原数组上操作，不能拷贝额外的数组。
2. 尽量减少操作次数。

思路：

1. 每次在走的时候就统计0的个数，对于非0元素挪到没有0的元素去
2. 开一个新数组，碰到0往后放 //不符合题意
3. 索引方式

方法I：暴力解法

首先对数组进行遍历，判断每一个元素是否为0，0的排到后面去，非0的排在前面

(时间慢)

解题思路：

- (1) 寻找数组中不为0的元素--j用来统计数组中不为0的数
- (2) 找到后存放在tmp中

(3) 将这个数原本的位置存放成

(4) 将这个不为0的数放到前面

```
var moveZeroes = function(nums) {  
    let j=0;  
    for(i=0;i<nums.length;i++){  
        if(nums[i] != 0){  
            var tmp = nums[i];  
            nums[i] = 0;  
            nums[j] = tmp;  
            j++;  
        }  
    }  
}
```

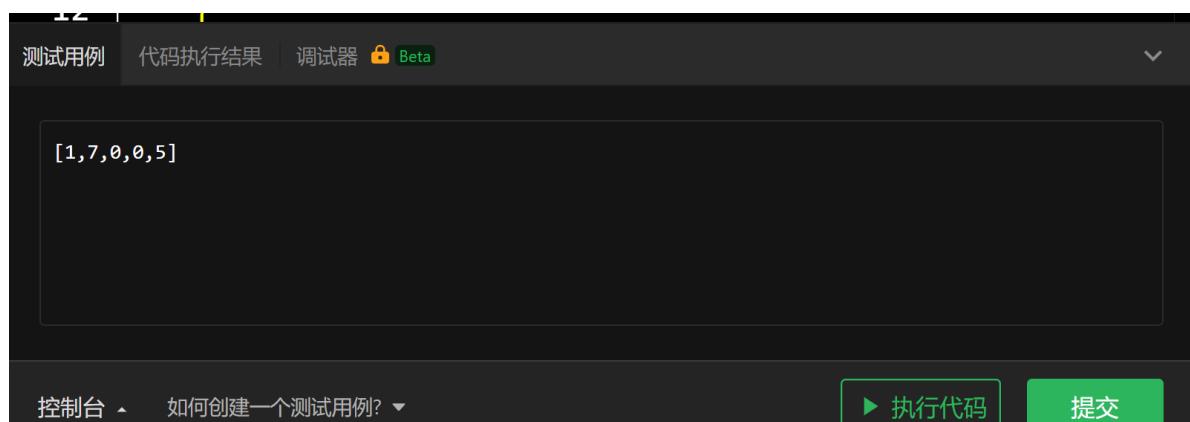
方法II --> 优解

首先将非0的元素移动到数组的前面，计算0的个数

从count计数后面开始向nums数组补零

```
var moveZeroes = function(nums) {  
    let count=0;  
    for(let i=0;i<nums.length;i++){  
        if(nums[i]!<0){  
            nums[count] = nums[i];  
            count++;  
        }  
    }  
    for(let j=count;j<nums.length;j++){  
        nums[j] = 0;  
    }  
}
```

更改测试样例-leetcode



去国际站找最优解

复制网址-去掉网址中的cn-去找最优解

盛水最多的容器

<https://leetcode-cn.com/problems/container-with-most-water/>

解题思路：

1. 枚举：left bar, right bar, $(x-y) * \text{height_diff}$ ---> $O(n^2)$

```
var maxArea = function(height) {  
    let max = 0;  
    for(var i=0;i<height.length-1;i++){  
        for(var j=i+1;j<height.length;j++){  
            var area = (j-i)*Math.min(height[i],height[j]);  
            max = Math.max(max,area);  
        }  
    }  
    return max;  
};
```

2. 双指针夹逼：

根据面积计算规则，面积是由两个柱子的距离和柱子最低高度决定的。

所以，一开始前后指针指向第一根柱子和最后一根柱子，计算这两根柱子的面积，此时他们距离是最大的。

由于高度收到最低的限制，所以前后指针中高度最低的往中间移动，知道找到比它高的柱子（因为距离在减少，所以只有高度增大才有机会比之前的大），再重新计算面积，并和前面的比较，取最大值。

知道前后指针重合。

```
var maxArea = function(height) {  
    let max = 0;  
    for(let i=0,j=height.length-1;i<j;){  
        //先判断再减  
        let minHeight = height[i] < height[j] ? height[i++]:height[j--];  
        //此时已经进行横坐标缩小操作了，因此需要把1再加上  
        let area = (j-i+1) * minHeight;  
        max = Math.max(max,area);  
    }  
    return max;  
};
```

最大误区：做题只做一遍

优化的思想：空间换时间

懵逼的时候：

- (1) 是否能暴力
- (2) 基本情况 --> 化繁为简

找最近重复子问题

if else/ for while/ recursion

爬楼梯(本质：斐波那契数列)

70. 爬楼梯

难度 **简单** 1499 收藏 分享 切换为英文 接收动态 反馈

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

示例 1：

```
输入: 2
输出: 2
解释: 有两种方法可以爬到楼顶。
1. 1 阶 + 1 阶
2. 2 阶
```

示例 2：

```
输入: 3
输出: 3
解释: 有三种方法可以爬到楼顶。
1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶
```

方法I：递归

$n = 1; 1$

$n = 2; 2$

$n = 3; f(1) + f(2)$ //第三级台阶是上到第二级台阶走两步；或者是上到第一级台阶走一步

$n = 4; f(2) + f(3)$

$f(n) = f(n-1) + f(n-2)$

```
var climbStairs = function(n) {
  let result = [0,1,2];
  for(var i=3;i<=n;i++){
    result[i] = result[i-1] + result[i-2];
  }
  return result[n];
};
```

方法II：滚动数组

```
var climbStairs = function(n) {
    let p=0,q=1
    for(let i=1;i<=n;i++){
        var r=p+q;
        p=q;
        q=r;
    }
    return r;
};
```

加一

66. 加一

难度 **简单** 629 收藏 分享 切换为英文 接收动态 反馈

给定一个由 **整数** 组成的 **非空** 数组所表示的非负整数，在该数的基础上加一。

最高位数字存放在数组的首位，数组中每个元素只存储**单个**数字。

你可以假设除了整数 0 之外，这个整数不会以零开头。

示例 1：

```
输入: digits = [1,2,3]
输出: [1,2,4]
解释: 输入数组表示数字 123。
```

示例 2：

```
输入: digits = [4,3,2,1]
输出: [4,3,2,2]
解释: 输入数组表示数字 4321。
```

示例 3：

```
输入: digits = [0]
输出: [1]
```

解题思路：

- (1) 先判断最后一位是否等于9；如果不等于9的话，直接最后位+1并返回数组值
- (2) 当最后一位为9时，令最后一位为0，并返回循环继续向上+1，直到return
- (3) 当遍历完整个数组都无法找到不为9的数时，说明数组为[9,9,9,...,9]；此时需要在第一位push0并返回数组

```
var plusOne = function(digits) {
    for(let i=digits.length-1;i>=0;i--){
        //0-8 -----> +1
        if(digits[i]==9){
            digits[i]++;
        }
```

```
    return digits;
}
//9 ----> 0
digits[i] = 0;
}
//[9,9,9,9] ---->第一位进位1
digits[0] = 1; //把digits[0]=1
digits.push(0); //在后面补一个0
return digits;
};
```

两数之和 (高频考题)

1. 两数之和

难度 简单 10184 ☆ ⓘ ⓘ ⓘ

给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出 **和为目标值** 的那 **两个** 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。

你可以按任意顺序返回答案。

示例 1：

```
输入: nums = [2,7,11,15], target = 9
输出: [0,1]
解释: 因为 nums[0] + nums[1] == 9 , 返回 [0, 1] 。
```

示例 2：

```
输入: nums = [3,2,4], target = 6
输出: [1,2]
```

方法I：两层循环，枚举a和b如果 $a+b = target$ 那么返回

```
var twoSum = function(nums, target) {
    for(let i=0;i<nums.length-1;i++){
        for(let j=i+1;j<nums.length;j++){
            if(nums[i]+nums[j]== target){
                return [i,j];
            }
        }
    }
};
```

方法II：哈希表O(n)

- (1) 将nums[i] 放入到hash table中
- (2) 查看target - nums[i] 是否在hash table里

```
var twoSum = function(nums, target) {
    let map={};
    for(let i=0;i<nums.length;i++){
        let n=target - nums[i];
        if(n in map) return [i,map[n]];
        map[nums[i]] = i;
    }
};
```

三数之和 (高频考题)

15. 三数之和

难度 中等

2916



给你一个包含 n 个整数的数组 nums ，判断 nums 中是否存在三个元素 a, b, c ，使得 $a + b + c = 0$ ？请你找出所有和为 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。

示例 1:

输入: $\text{nums} = [-1, 0, 1, 2, -1, -4]$

输出: $[[-1, -1, 2], [-1, 0, 1]]$

示例 2:

输入: $\text{nums} = []$

输出: $[]$

审题:

1. 返回**不重复**的三元组
2. 会有复数，无序
3. 可能不存在（实际要求返回空数组）
4. $a + b = -c$
5. 数组内有重复数字，结果有可能有重复

解题思路:

方法I：暴力求解

找出两个数 $a+b = -c$ (target 不是一个定值)

方法II：hash + 两重暴力（将 $\text{target} \rightarrow -c$ 保存到hash表里）

方法III：双指针左右夹逼

-4	-1	-1	-1	0	1	2
^K	^i			^j		

$$\text{nums}[k] + \text{nums}[i] + \text{nums}[j] = -3 < 0$$

res = []

-4	-1	-1	-1	0	1	2
^K		^i		^j		

$$\text{nums}[k] + \text{nums}[i] + \text{nums}[j] = -2 < 0$$

res = []

-4	-1	-1	-1	0	1	2
^K	^i			^j		

$$\text{nums}[k] + \text{nums}[i] + \text{nums}[j] = 0 == 0$$

res = [[-1, -1, 2]]

<https://leetcode-cn.com/problems/3sum/solution/3sumpai-xu-shuang-zhi-zhen-yi-dong-by-jyd/>

Code:

```

var threeSum = function (nums) {
  const res = []
  const len = nums.length
  nums.sort((a, b) => a - b) // 升序排列

  for (let i = 0; i < len - 2; i++) {
    if (nums[i] === nums[i - 1]) continue // 如果第一位位i有一样的，就直接进位

    let left = i + 1, right = len - 1

    while (left < right) {
      let sum = nums[i] + nums[left] + nums[right]
      if (sum === 0) {
        // 这一行注意
        // 在push的时候，用后++法传入原始值并做双指针--
        res.push([nums[i], nums[left++], nums[right--]])
      }

      // 判断左右指针是否和上一次一样，一样就跳到下一个去重
      while (nums[left] === nums[left - 1]) left++

      // 看其他人没写right++，其实要写的，因为left变了，right一定要变
      // 如果right还和上次一样，肯定不会sum = 0
      while (nums[right] === nums[right + 1]) right--

      } else {
        // 如果三个数没凑成0，就看是大了还是小了，大了就-right，小了+left
        if (sum > 0) right--
        if (sum < 0) left++
      }
    }
  }

  return res;
}

```

合并两个有序数组

给你两个有序整数数组 `nums1` 和 `nums2`，请你将 `nums2` 合并到 `nums1` 中，使 `nums1` 成为一个有序数组。

初始化 `nums1` 和 `nums2` 的元素数量分别为 `m` 和 `n`。你可以假设 `nums1` 的空间大小等于 `m + n`，这样它就有足够的空间保存来自 `nums2` 的元素。

示例 1：

```
输入: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
输出: [1,2,2,3,5,6]
```

示例 2：

```
输入: nums1 = [1], m = 1, nums2 = [], n = 0
输出: [1]
```

方法I：暴力解法-->O(n)

```
var merge = function(nums1, m, nums2, n) {
  if(n==0) return nums1;
  for(let i=0;i<n;i++){
    nums1[m+i] = nums2[i];
  }
  return nums1.sort((a,b)=>(a - b));
}
```

方法II：快速解法

从num1中取m个数，从num2中取n个数，最终数组的长度m+n-1

分别从1和2中选取较大的数依次填入num1中

```
var merge = function(nums1, m, nums2, n) {
  if(n==0) return nums1;
  let idx1 = m-1, idx2 = n-1, idx3 = m+n-1
  while(idx2 >= 0){
    nums1[idx3--] = nums1[idx1] > nums2[idx2] ? nums1[idx1--] : nums2[idx2--];
  }
  return nums1;
};
```

旋转数组

189. 旋转数组

难度 中等 896 收藏 叉 文档

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。

进阶：

- 尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。
- 你可以使用空间复杂度为 $O(1)$ 的 **原地** 算法解决这个问题吗？

示例 1：

输入: `nums = [1,2,3,4,5,6,7], k = 3`

输出: `[5,6,7,1,2,3,4]`

解释:

向右旋转 1 步: `[7,1,2,3,4,5,6]`

向右旋转 2 步: `[6,7,1,2,3,4,5]`

向右旋转 3 步: `[5,6,7,1,2,3,4]`

示例 2：

输入: `nums = [-1,-100,3,99], k = 2`

输出: `[3,99,-1,-100]`

需要思考的点:

- (1) 当 $k=0$ 时，返回的值
- (2) 当 $k>\text{nums.length}$ 时，数组如何移动

方法I：暴力解法 (LeetCode不让通过)

`array.pop()` ----> 删除数组中最后一个元素

`array.unshift()` ---> 向数组开头加入一个元素

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

方法II：快速解法

nums.splice(-k) ---> 从尾部移除倒数第k个元素

... => 将二维数组降维成一维数组

```
var rotate = function(nums, k) {
  k %= nums.length; //找出k移动的距离
  if(k === 0) return;
  nums.unshift(...nums.splice(-k));
  return nums;
};
```

方法III：空间复杂度O(1)

reverse-3:

(1) 将数组从结尾向头进行翻转

nums=[1,2,3,4,5,6,7] ----> nums=[7,6,5,4,3,2,1]

(2) 对0->k-1 的数组进行反转(k=3)

nums=[5,6,7,4,3,2,1]

(3) 再对 k->length -1 的数组进行反转

nums=[5,6,7,1,2,3,4]

```
const reverse = (nums, from, to) => {
  for(let i = from, j = to; i < j; i++, j--) {
    let t = nums[i];
    nums[i] = nums[j];
    nums[j] = t;
  }
}

var rotate = (nums, k) => {
  k = k % nums.length; //当k值大于数组时，取余为0，返回的是数组原来的值
  reverse(nums, 0, nums.length - 1);
  reverse(nums, 0, k - 1);
  reverse(nums, k, nums.length - 1);
  return nums;
};
```

删除有序数组中的重复项

26. 删除有序数组中的重复项

难度 简单 2076 收藏 分享 切换为英文 接收动态 反馈

给你一个有序数组 `nums`，请你 **原地** 删除重复出现的元素，使每个元素 **只出现一次**，返回删除后数组的新长度。

不要使用额外的数组空间，你必须在 **原地修改输入数组** 并在使用 $O(1)$ 额外空间的条件下完成。

说明:

为什么返回数值是整数，但输出的答案是数组呢？

请注意，输入数组是以「引用」方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下：

```
// nums 是以“引用”方式传递的。也就是说，不对实参做任何拷贝
int len = removeDuplicates(nums);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中 该长度范围内 的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
```

示例 1:

输入: `nums = [1,1,2]`

输出: `2, nums = [1,2]`

解释: 函数应该返回新的长度 `2`，并且原数组 `nums` 的前两个元素被修改为 `1, 2`。不需要考虑数组中超出新长度后面的元素。

示例 2:

输入: `nums = [0,0,1,1,1,2,2,3,3,4]`

输出: `5, nums = [0,1,2,3,4]`

解释: 函数应该返回新的长度 `5`，并且原数组 `nums` 的前五个元素被修改为 `0, 1, 2, 3, 4`。不需要考虑数组中超出新长度后面的元素。

解题思路:

方法I：“移动零解法”---->时间复杂度 $O(n)$ /空间复杂度 $O(1)$

首先将非重复的元素放到数组的前面，并计录非元素的个数count

将count数组后面的元素从数组中移除

```

var removeDuplicates = function(nums) {
  let count = 0;
  for(let i=0;i<nums.length;i++){
    if(nums[i]!=nums[i-1]){
      nums[count] = nums[i];
      count++;
    }
  }
  for(let j=nums.length-1;j>=count;j--){
    nums.pop(nums[j]);
  }
};

```

Linked List定义

链表中的每一个元素指向下一个元素，每一个node里包含element 和 next元素

用JavaScript手动实现链表的功能

```

function LinkedList(){
  var length =0;
  var head = null;

  var Node = function (element) { //define Node --> Node includes element,next
    this.element = element;
    this.next = null; //next指针指向空
  };

  this.size = function () {
    return length;
  };

  this.head = function () {
    return head;
  };
}

```

用JavaScript实现add / remove / isEmpty / Find 功能

Add Function

```

this.add = function (element) {
  var node = new Node(element);
  if(head ===null){ //如果head指向null说明原链表中没有结点，因此add结点后head指向新结点node
    head = node;
  }else{
    var currentNode = head;
    while(currentNode.next){
      currentNode = currentNode.next; //依次向后移动
    }
    currentNode.next = Node;
  }
  length++;
};

```

Remove Function

```
this.remove = function (element) {  
    var currentNode = head;  
    var previousNode;  
    if(currentNode.element === element)//如果要删除的元素就在头节点  
        head = currentNode.next; //那么直接让头节点指向下一个节点处  
    else{  
        while(currentNode.element !== element){ //一直找到要删除的element的所在地  
            previousNode = currentNode;  
            currentNode = currentNode.next;  
        }  
        previousNode.next = currentNode.next;  
    }  
    length--;  
};
```

isEmpty Function

```
this.isEmpty = function () {  
    return length === 0;  
};
```

Addat Function

```
this.addAt = function (index,element) {  
    var node = new Node(element); //想要插入的node  
  
    var currentNode = head; //在一开始的时候, currentNode在head  
    var previousNode;  
    var currentIndex = 0;  
  
    if(index>length) { //如果要插入的index大于linkedlist长度  
        return false;  
    }  
    if(index == 0){ //插入头结点  
        node.next = currentNode; //var currentNode = head;  
        head = node; //将node放在head的位置  
    }else{  
        while(currentNode < index){ //go through each index until we find the correct index  
            currentIndex++;  
            previousNode = currentNode;  
            currentNode = currentNode.next;  
        }  
        node.next = currentNode; //插入node  
        previousNode.next = node;  
    }  
    length++;  
}
```

Linked List 题目：

面试前一定要再写一遍高频题！！

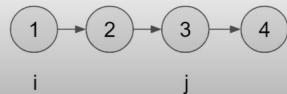
链表的解题法：

Two Pointers

两个指针指向Linked List节点，不再是index

两个指针必定同向而行

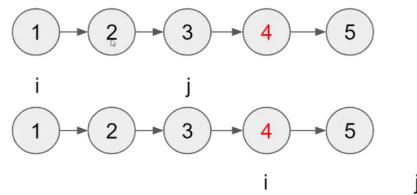
1. 一个快一个慢，距离隔开多少
2. 两个指针移动速度



Linked List找到数第k个节点

1. 一个快一个慢，距离隔开多少
2. 两个指针移动速度

Example: k = 2



两个指针先隔开k个位置，然后每次相同速度向前进直到快指针出界，慢指针就停留在倒数第k个节点

1. 反转链表

206. 反转链表

难度 简单 1549 收藏 分享 切换为英文 接收动态 反馈

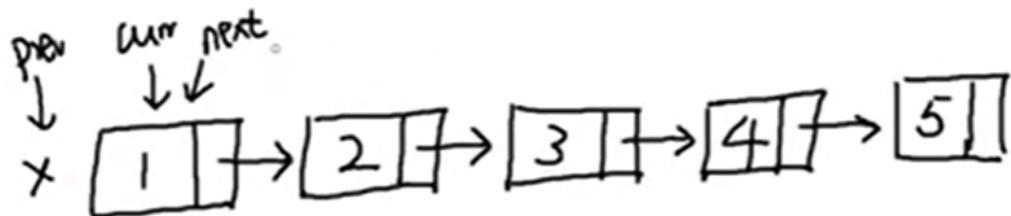
反转一个单链表。

示例：

输入： 1->2->3->4->5->NULL

输出： 5->4->3->2->1->NULL

方法I：双指针



```

while {
    ① next = curr.next
    ② curr.next = prev
    ③ prev = curr
    ④ curr = next
}
return prev

```

解题思路：

curr是一个帮助prev & next 定位的指针

curr & next 永远指向同一个节点； prev指向curr的前一个节点

移动需要4个步骤：

- (1) next走到curr.next
- (2) curr.next 指向prev
- (3) prev走到curr的位置
- (4) curr走到next现在的位置

需要prev,curr,next三种元素的原因：

next需要去占用位置， prev必须要在curr移动前去到next所占用的位置

Attention: 最后的返回值应该是prev， 指针的位置---> 因为改变的是指针的指向， 因此应该返回 prev

```

var reverseList = function(head) {
  let prev=null;
  let curr= next = head;
  while(curr){
    next = curr.next;
    curr.next = prev;
    prev = curr;
    curr = next;
  }
  return prev;
};

```

2. 两两交换链表中的结点

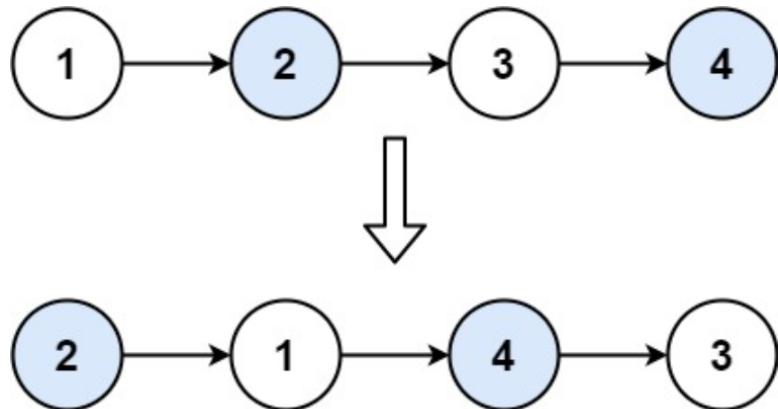
24. 两两交换链表中的节点

难度 中等 838 收藏 分享 切换为英文 接收动态 反馈

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你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。

示例 1：



输入: head = [1,2,3,4]

输出: [2,1,4,3]

```
var swapPairs = function(head) {
    if(head === null) return null;
    if(head.next === null) return head;
    let n1 = head.next;
    let n2 = head.next.next;
    n1.next = head;
    head.next = swapPairs(n2);
    //为什么返回值是n1?
    //因为在进行节点交换之后, n1变成了头节点的指针
    return n1;
};
```

3. 环形链表

给定一个链表，判断链表中是否有环。

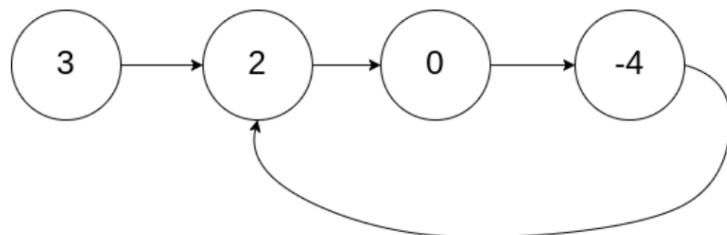
如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 `pos` 是 `-1`，则在该链表中没有环。**注意：**`pos` 不作为参数进行传递，仅仅是为了标识链表的实际情况。

如果链表中存在环，则返回 `true`。否则，返回 `false`。

进阶：

你能用 $O(1)$ (即，常量) 内存解决此问题吗？

示例 1：

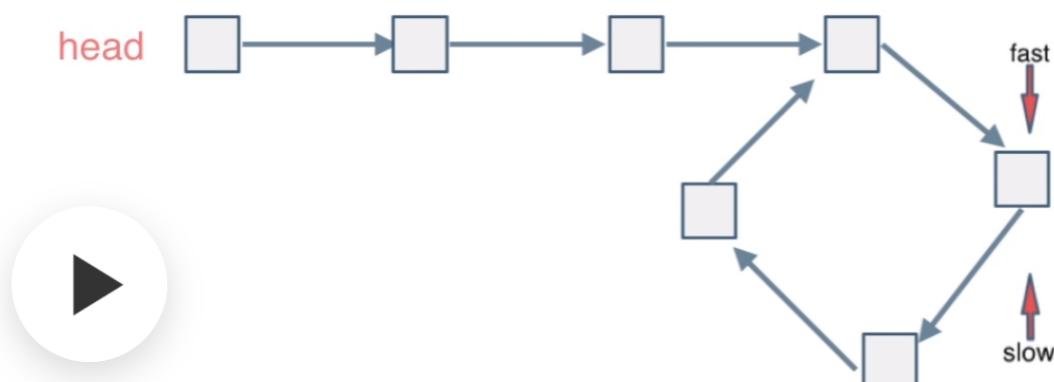


输入： `head = [3, 2, 0, -4], pos = 1`

输出： `true`

方法I：遍历列表，hash/ set 记录下来访问过的所有节点，查看原来的指针是否有重复元素，有则

方法II：快慢指针，快指针一次走一个格，慢指针一次走两个格，如果有环的话，快慢指针会相遇



环内相遇

<https://leetcode-cn.com/problems/linked-list-cycle/solution/141-huan-xing-lian-biao-shuang-zhi-zhen-zhao-huan-/>

Attention: `&&` 和 `&` and `|` 和 `||` 的区别

在JavaScript中“`&&`”和“`||`”是逻辑运算符；“`&`”和“`|`”是位运算符

```

var hasCycle = function(head) {
    if(head === null) {
        return false;
    }
    let slow = head;
    let fast = head;
    // 因为快指针一次走两步，因此需要对fast.next.next也进行判断
    while(fast.next !== null && fast.next.next !== null){
        slow = slow.next;
        fast = fast.next.next;
        if(slow === fast){
            return true;
        }
    }
    // 如果fast指针走到头，快慢指针还没有相遇，那么链表中没环
    return false;
};

```

4. 环形链表II

142. 环形链表 II

难度 中等 888 收藏 分享 切换为英文 接收动态 反馈

给定一个链表，返回链表开始入环的第一个节点。如果链表无环，则返回 `null`。

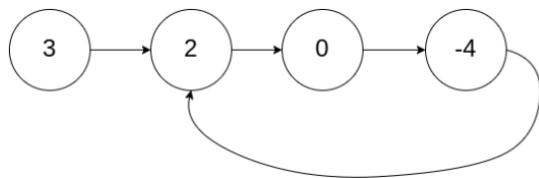
为了表示给定链表中的环，我们使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 `pos` 是 `-1`，则在该链表中没有环。**注意，`pos` 仅仅是用于标识环的情况，并不会作为参数传递到函数中。**

说明：不允许修改给定的链表。

进阶：

- 你是否可以使用 `O(1)` 空间解决此题？

示例 1：



输入: `head = [3,2,0,-4], pos = 1`

输出: 返回索引为 1 的链表节点

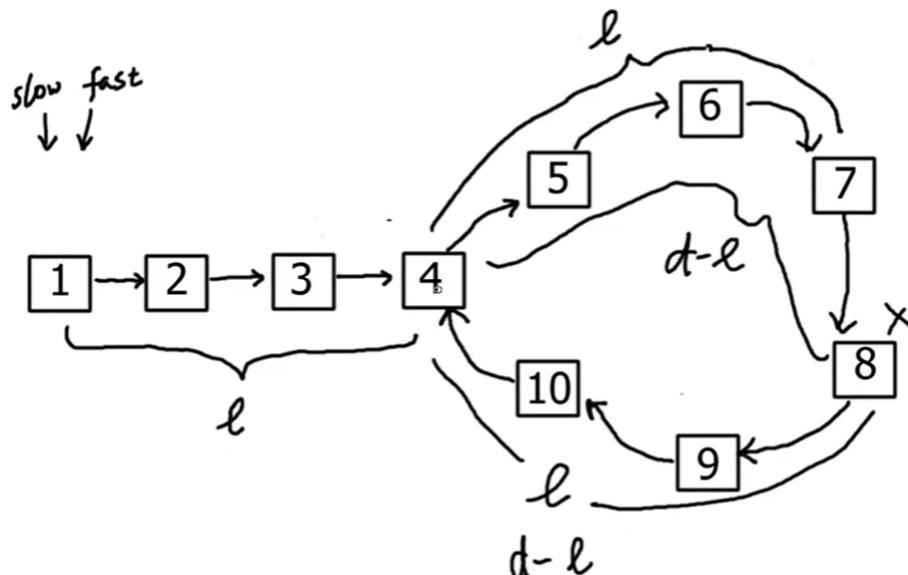
解释: 链表中有一个环，其尾部连接到第二个节点。

解题思路：

弗洛伊迪算法：

分别设置 `slow`, `fast` 两个指针，当两个指针相遇时，另 `fast` 回到链表的头部；当 `slow`, `fast` 再次相遇时的点就是环形链表的入口位置

分析有效性:



fast ---> 8 & slow --->4 ---> fast比slow每次都多走一步

环状链表---> fast想要追赶上slow需要走 $d-l$ 步，当slow走了 $d-l$ 步时， $slow==fast$ ，此时 $l=d-l \Rightarrow d=l$

因此想知道环形链表的入口位置，只需要另fast回到起始位置，当 $fast==slow$ 时，位置就是环形链表的入口位置

Code:

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

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

```
    slow = slow.next;
    fast = fast.next;
}
return fast;
};
```

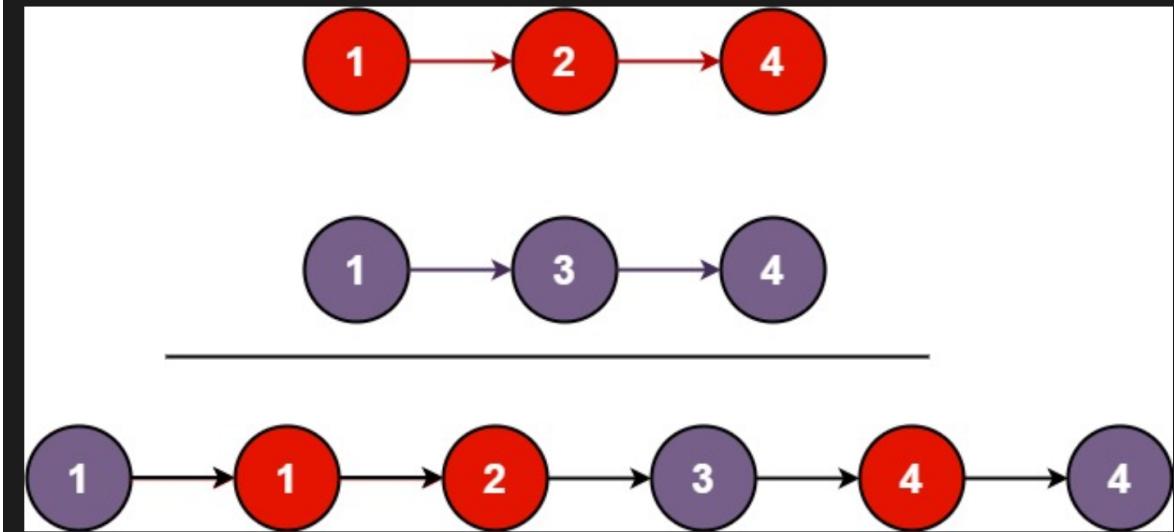
5. 合并两个有序链表

21. 合并两个有序链表

难度 简单 1744 收藏 分享 切换为英文 接收动态 反馈

将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例 1:



输入: $l1 = [1, 2, 4]$, $l2 = [1, 3, 4]$

输出: $[1, 1, 2, 3, 4, 4]$

示例 2:

输入: $l1 = []$, $l2 = []$

输出: $[]$

示例 3:

输入: $l1 = []$, $l2 = [\theta]$

输出: $[\theta]$

解题思路:

方法I：暴力解法 ---> 时间复杂度 $O(M+N)$ & 空间复杂度 $O(1)$

设置一个prev指针，分别遍历L1 & L2链表，prev指向L1 / L2中较大的值

方法II：开辟新链表法 ----> 时间复杂度 $O(\max(M,N))$ & 空间复杂度 $O(N)$

```

var mergeTwoLists = function(l1, l2) {
  let curr = new ListNode();
  //最后返回的是整个链表，但是curr此时已经走到链表的最后；因此需要在开头设置一个变量最后return链表的值
  let dummy = curr;
  while(l1 !== null && l2 !== null){
    if(l1.val < l2.val){
      curr.next = l1;
      l1 = l1.next;
    }else{ // l2中的值小于l1
      curr.next = l2;
      l2 = l2.next;
    }
    curr = curr.next; //新开辟的链表往后挪动一位
  }
  //当完成遍历之后，此时会出现l1 or l2中仍然有值得情况
  if(l1 !== null){
    curr.next = l1;
  }
  if(l2 !== null){
    curr.next = l2;
  }
  return dummy.next;
};

```

6. 删除有序链表中的重复项

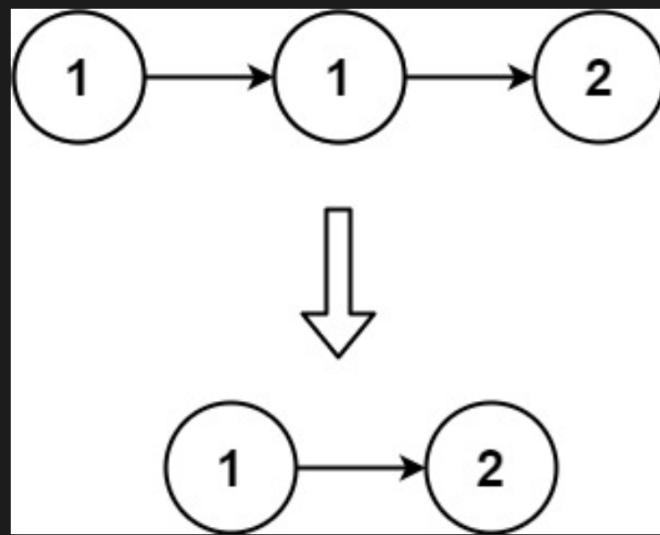
83. 删除排序链表中的重复元素

难度 简单 583 ☆ ⌂ ⌂ ⌂ ⌂

存在一个按升序排列的链表，给你这个链表的头节点 `head`，请你删除所有重复的元素，使每个元素 **只出现一次**。

返回同样按升序排列的结果链表。

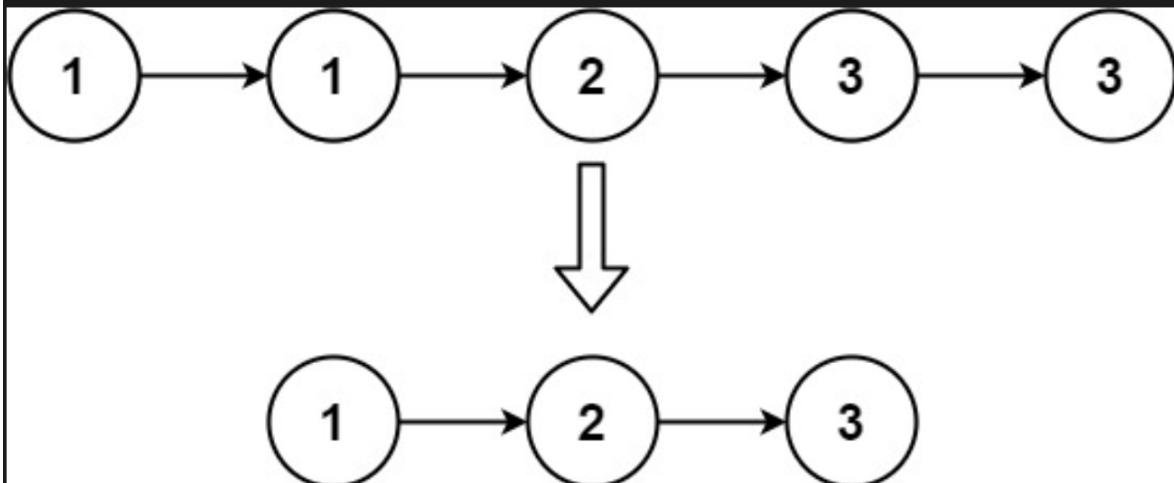
示例 1：



输入: head = [1,1,2]

输出: [1,2]

示例 2:



输入: head = [1,1,2,3,3]

输出: [1,2,3]

解题思路:

链表中的删除操作 => 当发现重复元素时，指针跳过当前元素直接指向下一个不重复的元素

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

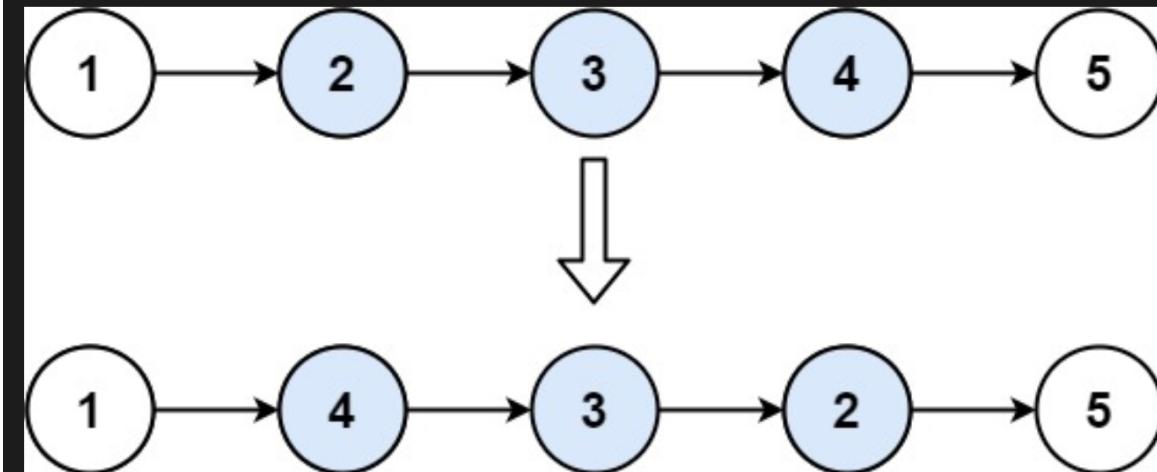
7. 反转链表II

92. 反转链表 II

难度 中等 917 ☆ ⓘ ⓘ ⓘ

给你单链表的头指针 `head` 和两个整数 `left` 和 `right`，其中 `left <= right`。请你反转从位置 `left` 到位置 `right` 的链表节点，返回 **反转后的链表**。

示例 1：



输入: `head = [1,2,3,4,5], left = 2, right = 4`

输出: `[1,4,3,2,5]`

解题思路：

两个步骤：

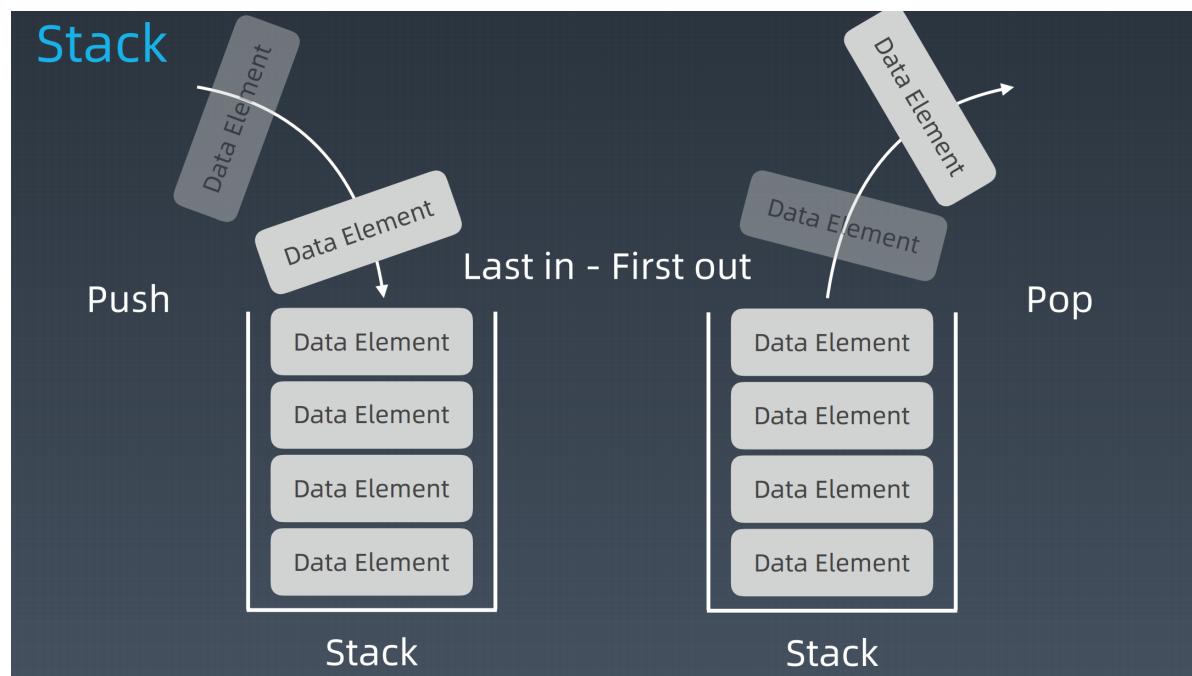
- (1) 反转`left`至`right`之间的链表
- (2) 将反转后的链表与原链表拼接

```
var reverseBetween = function(head, left, right) {  
    let prev = null;  
    let curr = head;  
    let next = head;  
  
    //第一步：找到left的位置  
    for(let i=1;i<left;i++){  
        prev = curr;  
        curr = curr.next;  
    }  
  
    //在这里需要记住prev和curr的起始位置  
    let prev2 = prev;  
    let curr2 = curr;  
  
    //反转中间部分的链表  
    for(let j=left;j<=right;j++){  
        next = curr.next;  
        curr.next = prev;  
        prev = curr;  
        curr = next;  
    }  
}
```

```
// 将反转后的链表与之前的部分进行拼接
if(prev2 != null){
    prev2.next = prev; //段的拼接
} else{
    head = prev; //如果是从head部分开始反转，则需要将头部进行拼接
}
curr2.next = curr;
return head;
};
```

Week 02

栈、队列、双端队列、优先队列



让我们来看一个使用Java脚本中的数组的堆栈类示例： -
示例：

```
// Stack class
class Stack {

    // Array is used to implement stack
    constructor()
    {
        this.items = [];
    }

    // Functions to be implemented
    // push(item)
    // pop()
    // peek()
    // isEmpty()
    // printStack()
}
```



用JavaScript实现Queue

```
//创建一个队列的类
class Queue{
    constructor(){
        this.count = 0;//记录队列的数量
        this.lowestCount = 0;//记录当前队列头部的位置
        this.items = [];//用来存储元素。
    }

    //Add element
    enqueue(element){
        this.items[this.count] = element;
        this.count++;
    }
```

```

}

//delete element
dequeue(){
    if(this.isEmpty()){
        return null;
    }
    let result = this.items[this.lowestCount];
    delete this.items[this.lowestCount];
    this.lowestCount++;
    return result;
}

//查看队列头部元素
peek(){
    return this.items[this.lowestCount];
}

// 判断队列是否为空
isEmpty(){
    return this.count - this.lowestCount === 0;
}

//清除队列中的所有元素
clear(){
    this.count = 0;
    this.lowestCount = 0;
    this.items = [];
}

// 查看队列的长度
size(){
    return this.count - this.lowestCount;
}

//查看队列中所有的元素
toString(){
    if(this.isEmpty())return "queue is null";
    let objString = this.items[this.lowestCount];
    for(let i = this.lowestCount+1; i < this.count;i++){
        objString = `${objString},${this.items[i]}`;
    }
    return objString;
}
}

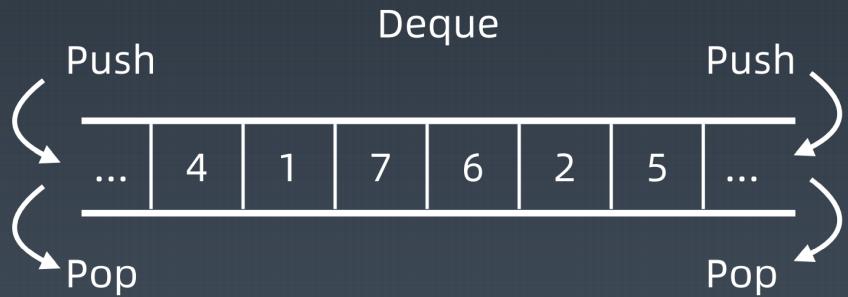
```

Stack & Queue 关键点

- Stack: 先入后出；添加、删除皆为 O(1)
- Queue: 先入先出；添加、删除皆为 O(1)

查询O(n) --> 因为队列无序

Deque: Double-End Queue



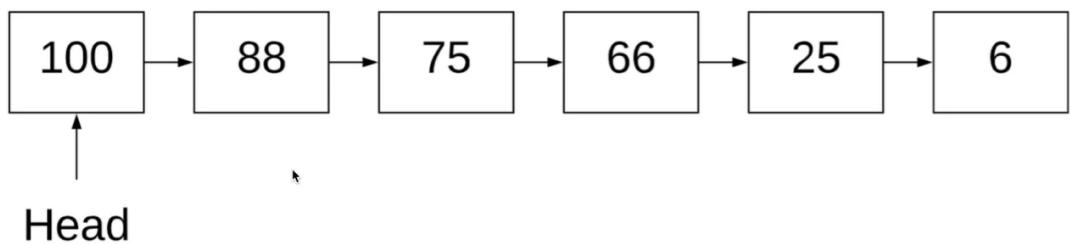
Deque

1. 简单理解：两端可以进出的 Queue
Deque - double ended queue

2. 插入和删除都是 O(1) 操作

Priority Queue

PriorityQueue



- push: 插入一个新的元素
- pop: 将优先级最高的元素弹出（删除）
- peek: 查看优先级最高的元素数值



优先队列：优先从大到小排列，将头结点指向数字最大的位置

Define Priority Queue

```

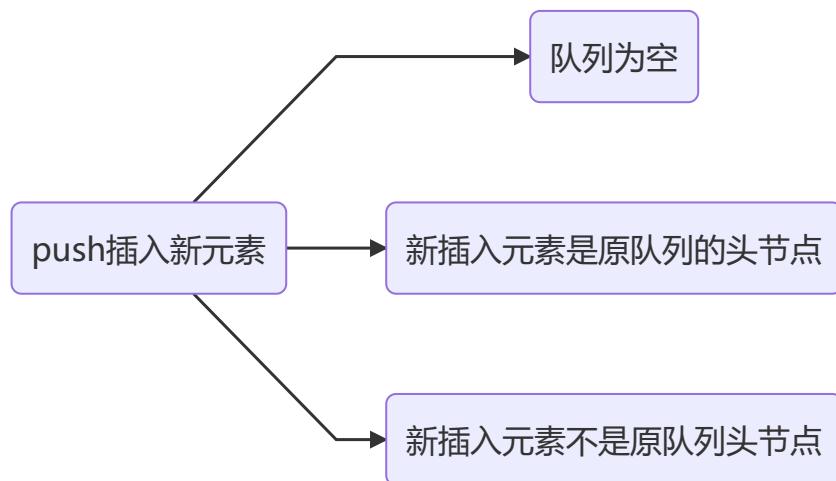
public class PriorityQueue{
    static class Node{
        int value;
        int priority;
        Node next;
    }

    public Node(int value,int priority){
        this.value = value;
        this.priority = priority;
    }
}

Node head = null;
}

```

push: 插入一个新的元素 (时间复杂度O(n))



```

public void push(int value,int priority){
    if(head == null){
        head = new Node(value,priority);
        return;
    }
    Node cur = head;
    Node newNode = new Node(value,priority);
    if(head.priority<priority){
        newNode.next = head;
        this.head = newNode;
    }else{
        while(cur.next!= null && cur.next.priority>priority){
            cur = cur.next;
        }
        newNode.next = cur.next;
        cur.next = newNode;
    }
}

```

插入push的时间复杂度: O(1)

找出peek的时间复杂度: O(log n) --> 按照优先级取出

```

public Node peek(){
    return head;
}

```

```

}

public Node pop(){
    if(head == null){
        return null;
    }

    Node temp = head;
    head = head.next;
    return temp;
}

public boolean isEmpty(){
    return head == null;
}

```

复杂度分析

Data Structure	Time Complexity								Space Complexity	
	Average				Worst					
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion		
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Stack	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Queue	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Singly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Doubly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Skip List	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$	
Hash Table	N/A	$O(1)$	$O(1)$	$O(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Binary Search Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Cartesian Tree	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
B-Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
Red-Black Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
Splay Tree	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
AVL Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
KD Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	

实战题目：

1.有效的括号

20. 有效的括号

难度 简单 2128 ⭐ ⚡ 🔍

给定一个只包括 '(', ')', '{', '}', '[', ']' 的字符串 `s`，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。

示例 1：

输入： `s = "()"`

输出： `true`

示例 2：

输入： `s = "()[]{}"`

输出： `true`

解题思路：

1. 暴力：不断`replace`匹配的括号，一直到最后不能替换的时候字符串为空

```
var isValid = function(s) {
    while(s.length){
        var temp = s;
        s = s.replace('()', '');
        s = s.replace('[]', '');
        s = s.replace('{}', '');
        if(s == temp) return false
    }
    return true;
}
```

2. Stack：判断如果元素是左括号则进行入栈操作，如果是右括号则与之前进栈的元素进行比较，匹配则进行删除操作，如果都匹配则栈为空

```
var isValid = function(s) {
    let map = { // 定义一个匹配库，并且利用键值和，进行比较判断是否匹配
        '(-1,
        ')':1,
        '{:-2,
        '}':2,
        '[':-3,
```

```
'T':3
}

let stack = []
for(let i=0;i < s.length;i++){
  if(map[s[i]] < 0){
    stack.push(s[i])
  }else {
    let last = stack.pop()
    if(map[last] + map[s[i]] != 0){
      return false;
    }
  }
}
if(stack.length > 0) return false;
return true
};
```

2.最小栈

155. 最小栈

难度 简单      

设计一个支持 `push` , `pop` , `top` 操作，并能在常数时间内检索到最小元素的栈。

- `push(x)` —— 将元素 x 推入栈中。
- `pop()` —— 删除栈顶的元素。
- `top()` —— 获取栈顶元素。
- `getMin()` —— 检索栈中的最小元素。

示例:

输入:

```
["MinStack","push","push","push","getMin","pop","top","getM
[],[-2],[0],[-3],[],[],[],[]]
```

输出:

```
[null,null,null,null,-3,null,0,-2]
```

思路分析:

题目中的`push(x)`, `pop()`, `top()`操作都可以直接调用`Stack`的API实现。`getMin()`检索栈中最小元素这个方法的实现，可以借助一个辅助栈。

初始化两个栈，一个是stack用于对元素x进行入栈、出栈操作；一个是辅助栈min，用于存储stack中已存元素中的最小元素。这里辅助栈的入栈操作是和元素x放入栈stack同步的，也就是说stack中每入栈一个元素，min中相应的要存入当前最小元素。

如何判断当前最小元素

两种情况：

- (1) min栈中没有值，任何值都是起始栈的最小值因此需要存入Infinity
- (2) min栈中有值，需要讲min中值提出来与新加入的x进行对比，将较小者放进栈中

动画演示：<https://leetcode-cn.com/problems/min-stack/solution/dong-hua-yan-shi-155-zui-xiao-zhan-by-ha-dnp9/>

```
var MinStack = function() {
    this.stack = [];
    this.min = [];//min中只保存比stack中小的元素
};

MinStack.prototype.push = function(x) {
    //初始化最小栈值的操作，如果最小栈内没元素则min=Infinity;若有元素，则另min=min[min.length-1]
    let min = this.min.length > 0 ? this.min[this.min.length-1] : Infinity;
    this.stack.push(x);
    this.min.push(Math.min(x, min));
};

MinStack.prototype.pop = function() {
    this.stack.pop();
    this.min.pop();
};

MinStack.prototype.top = function() {
    return this.stack[this.stack.length-1];
};

MinStack.prototype.getMin = function() {
    return this.min[this.min.length-1];
};
```

3. 用队列实现栈

225. 用队列实现栈

难度 简单

334

收藏

分享

切换为英文

接收动态

反馈

请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通队列的全部四种操作（push、top、pop 和 empty）。

实现 MyStack 类：

- void push(int x) 将元素 x 压入栈顶。
- int pop() 移除并返回栈顶元素。
- int top() 返回栈顶元素。
- boolean empty() 如果栈是空的，返回 true；否则，返回 false。

注意：

- 你只能使用队列的基本操作——也就是 push to back、peek/pop from front、size 和 empty 这些操作。
- 你所使用的语言也许不支持队列。你可以使用 list（列表）或者 deque（双端队列）来模拟一个队列，只要是标准的队列操作即可。

示例：

输入：

```
["MyStack", "push", "push", "top", "pop", "empty"]
[], [1], [2], [], [], []]
```

输出：

```
[null, null, null, 2, 2, false]
```

解释：

```
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);
myStack.top(); // 返回 2
myStack.pop(); // 返回 2
myStack.empty(); // 返回 False
```

提示：

- $1 \leq x \leq 9$
- 最多调用 100 次 push、pop、top 和 empty
- 每次调用 pop 和 top 都保证栈不为空

解题思路：

push() ----> 从数组尾部添加元素

pop() ----> 从数组尾部移除元素

```
/*
 * Initialize your data structure here.
 */
var MyStack = function() {
    this.queue = [];
};

/**
```

```
* Push element x onto stack.  
* @param {number} x  
* @return {void}  
*/  
MyStack.prototype.push = function(x) {  
    this.queue.push(x);  
}  
  
/**  
 * Removes the element on top of the stack and returns that element.  
 * @return {number}  
*/  
MyStack.prototype.pop = function() {  
    // pop()从尾部弹出一个元素  
    return this.queue.pop();  
}  
  
/**  
 * Get the top element.  
 * @return {number}  
*/  
MyStack.prototype.top = function() {  
    return this.queue[this.queue.length-1];  
}  
  
/**  
 * Returns whether the stack is empty.  
 * @return {boolean}  
*/  
MyStack.prototype.empty = function() {  
    return this.queue.length ==0;  
};
```

4. 用栈实现队列

请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（push、pop、peek、empty）：

实现 MyQueue 类：

- void push(int x) 将元素 x 推到队列的末尾
- int pop() 从队列的开头移除并返回元素
- int peek() 返回队列开头的元素
- boolean empty() 如果队列为空，返回 true；否则，返回 false

说明：

- 你只能使用标准的栈操作——也就是只有 push to top, peek/pop from top, size, 和 is empty 操作是合法的。
- 你所使用的语言也许不支持栈。你可以使用 list 或者 deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。

进阶：

- 你能否实现每个操作均摊时间复杂度为 $O(1)$ 的队列？换句话说，执行 n 个操作的总时间复杂度为 $O(n)$ ，即使其中一个操作可能花费较长时间。

```
var MyQueue = function() {
    this.is = [];
    this.os = [];
};

MyQueue.prototype.push = function(x) {
    this.is.push(x);
};

MyQueue.prototype.pop = function() {
    if(!this.os.length){
        while(this.is.length){
            this.os.push(this.is.pop());
        }
    }
    return this.os.pop();
};

MyQueue.prototype.peek = function() {
    if(!this.os.length){
        while(this.is.length){
            this.os.push(this.is.pop());
        }
    }
    return this.os[this.os.length - 1];
};

MyQueue.prototype.empty = function() {
    return !this.is.length && !this.os.length;
};
```

5. 设计循环队列

622. 设计循环队列

难度 中等 192 收藏 分享 切换为英文 接收动态 反馈

设计你的循环队列实现。循环队列是一种线性数据结构，其操作表现基于 FIFO（先进先出）原则并且队尾被连接在队首之后以形成一个循环。它也被称为“环形缓冲器”。

循环队列的一个好处是我们可以利用这个队列之前用过的空间。在一个普通队列里，一旦一个队列满了，我们就不能插入下一个元素，即使在队列前面仍有空间。但是使用循环队列，我们能使用这些空间去存储新的值。

你的实现应该支持如下操作：

- `MyCircularQueue(k)` : 构造器，设置队列长度为 k 。
- `Front` : 从队首获取元素。如果队列为空，返回 -1。
- `Rear` : 获取队尾元素。如果队列为空，返回 -1。
- `enQueue(value)` : 向循环队列插入一个元素。如果成功插入则返回真。
- `deQueue()` : 从循环队列中删除一个元素。如果成功删除则返回真。
- `isEmpty()` : 检查循环队列是否为空。
- `isFull()` : 检查循环队列是否已满。

示例：

```
MyCircularQueue circularQueue = new MyCircularQueue(3); // 设置长度为 3
circularQueue.enQueue(1); // 返回 true
circularQueue.enQueue(2); // 返回 true
circularQueue.enQueue(3); // 返回 true
circularQueue.enQueue(4); // 返回 false, 队列已满
circularQueue.Rear(); // 返回 3
circularQueue.isFull(); // 返回 true
circularQueue.deQueue(); // 返回 true
circularQueue.enQueue(4); // 返回 true
circularQueue.Rear(); // 返回 4
```

解题思路：

用数组模拟队列

定义：queue数组，size长度，count记录数组中的数

enQueue: 当queue不为full时，queue.push(value)，count++，返回true；否则返回false

deQueue: 当queue不为空时，queue.shift(),count--,返回true；否则返回false

Front: 当queue不为空时，return queue[0]

Rear: 当queue不为空时，return queue[this.queue.length-1]

isEmpty: this.count == 0

isFull: this.count == this.size

```
/*
 * @param {number} k
 */
var MyCircularQueue = function(k) {
  this.queue = [];
  this.size = k;
```

```

this.count = 0;
};

/**
 * @param {number} value
 * @return {boolean}
 */
MyCircularQueue.prototype.enQueue = function(value) {
  if(!this.isEmpty()){
    this.queue.push(value);
    this.count++;
    return true;
  }
  return false;
};

/**
 * @return {boolean}
 */
MyCircularQueue.prototype.deQueue = function() {
  if(!this.isEmpty()){
    this.queue.shift();
    this.count--;
    return true;
  }
  return false;
};

/**
 * @return {number}
 */
MyCircularQueue.prototype.Front = function() {
  if(!this.isEmpty()){
    return this.queue[0];
  }
  return -1;
};

/**
 * @return {number}
 */
MyCircularQueue.prototype.Rear = function() {
  if(!this.isEmpty()){
    return this.queue[this.queue.length-1];
  }
  return -1;
};

/**
 * @return {boolean}
 */
MyCircularQueue.prototype.isEmpty = function() {
  if(this.count == 0){
    return true;
  }
  return false;
};

```

```

/**
 * @return {boolean}
 */
MyCircularQueue.prototype.isFull = function() {
  if(this.count == this.size){
    return true;
  }
  return false;
};

```

6. 设计循环双端队列

641. 设计循环双端队列

难度 中等 81 收藏 分享 切换为英文 接收动态 反馈

设计实现双端队列。

你的实现需要支持以下操作：

- MyCircularDeque(k): 构造函数，双端队列的大小为k。
- insertFront(): 将一个元素添加到双端队列头部。如果操作成功返回 true。
- insertLast(): 将一个元素添加到双端队列尾部。如果操作成功返回 true。
- deleteFront(): 从双端队列头部删除一个元素。如果操作成功返回 true。
- deleteLast(): 从双端队列尾部删除一个元素。如果操作成功返回 true。
- getFront(): 从双端队列头部获得一个元素。如果双端队列为空，返回 -1。
- getRear(): 获得双端队列的最后一个元素。如果双端队列为空，返回 -1。
- isEmpty(): 检查双端队列是否为空。
- isFull(): 检查双端队列是否满了。

示例：

```

MyCircularDeque circularDeque = new MyCircularDeque(3); // 设置容量大小为3
circularDeque.insertLast(1); // 返回 true
circularDeque.insertLast(2); // 返回 true
circularDeque.insertFront(3); // 返回 true
circularDeque.insertFront(4); // 已经满了，返回 false
circularDeque.getRear(); // 返回 2
circularDeque.isFull(); // 返回 true
circularDeque.deleteLast(); // 返回 true
circularDeque.insertFront(4); // 返回 true
circularDeque.getFront(); // 返回 4

```

```

/**
 * Initialize your data structure here. Set the size of the deque to be k.
 * @param {number} k
 */
var MyCircularDeque = function(k) {
  this.queue = [];
  this.size = k;
  this.count = 0;
};

/**
 * Adds an item at the front of Deque. Return true if the operation is successful.
 * @param {number} value
 * @return {boolean}
 */
MyCircularDeque.prototype.insertFront = function(value) {
  if(!this.isFull()){
    // unshift() 从头部插入一个元素
    this.queue.unshift(value);
  }
};

```

```

        this.count++;
        return true;
    }
    return false;
};

/** 
 * Adds an item at the rear of Deque. Return true if the operation is successful.
 * @param {number} value
 * @return {boolean}
 */
MyCircularDeque.prototype.insertLast = function(value) {
    if(!this.isEmpty()){
        // push()从数组尾部添加元素
        this.queue.push(value);
        this.count++;
        return true;
    }
    return false;
};

/** 
 * Deletes an item from the front of Deque. Return true if the operation is successful.
 * @return {boolean}
 */
MyCircularDeque.prototype.deleteFront = function() {
    if(!this.isEmpty()){
        //shift() 从数组头部删除元素
        this.queue.shift();
        this.count--;
        return true;
    }
    return false;
};

/** 
 * Deletes an item from the rear of Deque. Return true if the operation is successful.
 * @return {boolean}
 */
MyCircularDeque.prototype.deleteLast = function() {
    if(!this.isEmpty()){
        //pop() 从数组尾部删除元素
        this.queue.pop();
        return true;
    }
    return false;
};

/** 
 * Get the front item from the deque.
 * @return {number}
 */
MyCircularDeque.prototype.getFront = function() {
    if(!this.isEmpty()){
        return this.queue[0]
    }
    return -1;
};

```

```
/**  
 * Get the last item from the deque.  
 * @return {number}  
 */  
MyCircularDeque.prototype.getRear = function() {  
    if(!this.isEmpty()){  
        return this.queue[this.queue.length-1];  
    }  
    return -1;  
};  
  
/**  
 * Checks whether the circular deque is empty or not.  
 * @return {boolean}  
 */  
MyCircularDeque.prototype.isEmpty = function() {  
    if(this.queue.length == 0){  
        return true;  
    }  
    return false;  
};  
  
/**  
 * Checks whether the circular deque is full or not.  
 * @return {boolean}  
 */  
MyCircularDeque.prototype.isFull = function() {  
    if(this.queue.length ==this.size){  
        return true;  
    }  
    return false;  
};
```

7.柱状图中最大的矩形

84. 柱状图中最大的矩形

难度 困难

1162

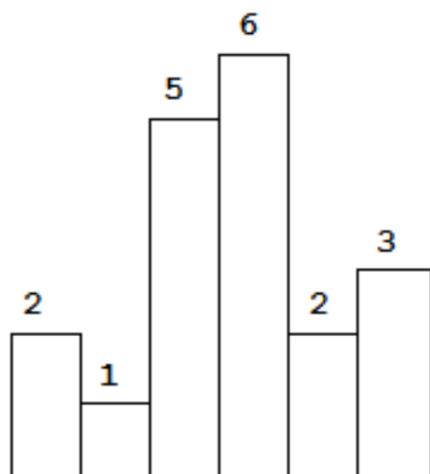


文

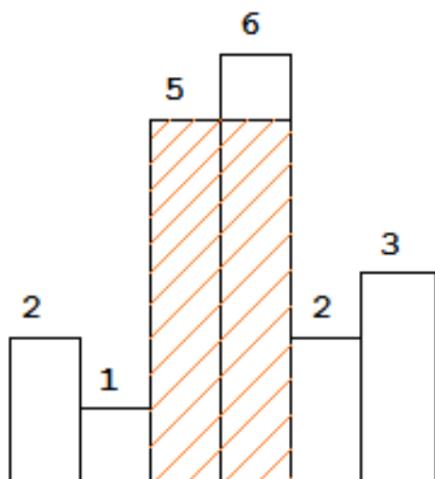


给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1。

求在该柱状图中，能够勾勒出来的矩形的最大面积。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 `[2, 1, 5, 6, 2, 3]`。

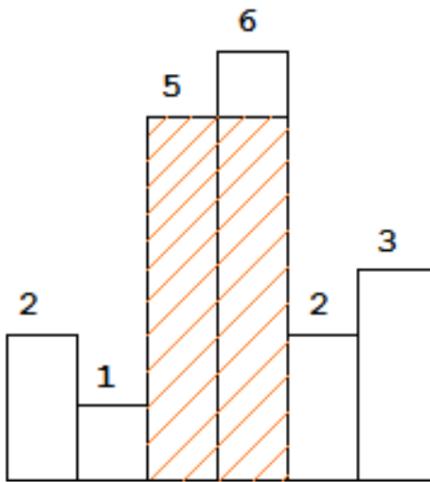


图中阴影部分为所能勾勒出的最大矩形面积，其面积为 `10` 个单位。

示例：

输入： `[2, 1, 5, 6, 2, 3]`

输出： `10`



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 10 个单位。

示例：

输入： [2,1,5,6,2,3]

输出： 10

解题思路：

方法I：暴力解法 --> 时间复杂度O(n^2)

```
for i -> 0, n-2
  for j -> i+1, n-1
    (i, j) --> 最小高度, area
    update max -> area
```

```
var largestRectangleArea = function(heights) {
  let maxArea = 0;
  for(let i=0;i<heights.length;i++){
    let minHeight = heights[i];
    for(let j=i;j<heights.length;j++){
      minHeight = minHeight>heights[j]? heights[j]:minHeight;
      let area = minHeight * (j-i+1);
      maxArea = Math.max(maxArea,area);
    }
  }
  return maxArea;
};
```

方法II：暴力加速法 --> O(n^2)

确定矩形的高，分别向左右延伸查看左右的边界

```

for i -> 0, n-1
    //找到 left bound, right bound,
    area = height[i] * (right - left)
    update maxArea;

```

```

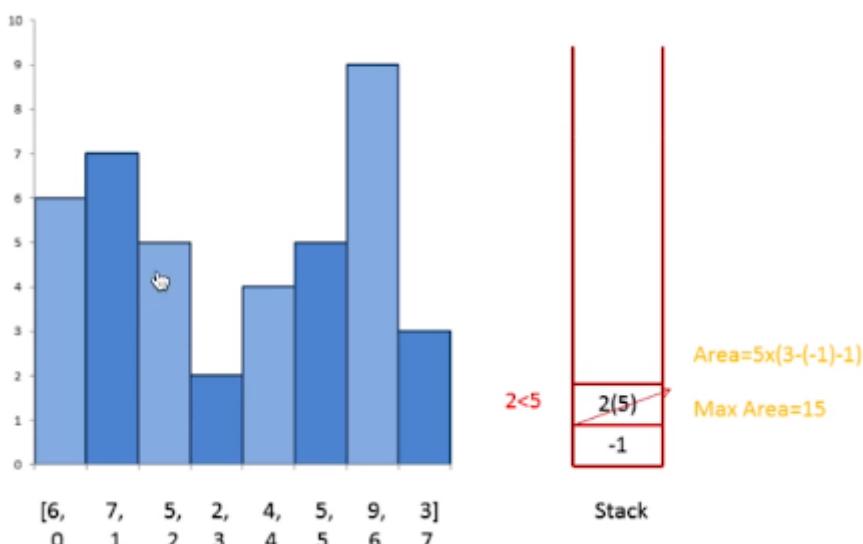
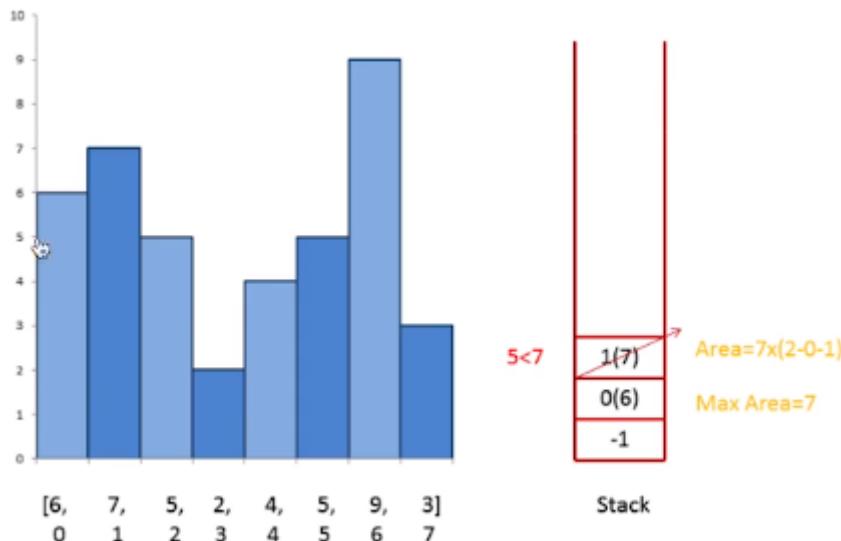
var largestRectangleArea = function(heights) {
    let maxArea = 0;
    for(let i=0;i<heights.length;i++){
        let left = i-1;
        let right = i+1;
        while(left>=0 && heights[left]>=heights[i]){left--};
        while(right<heights.length && heights[right]>=heights[i]){right++};
        let area = heights[i] * (right - left-1);
        maxArea = Math.max(area,maxArea);
    }
    return maxArea;
};

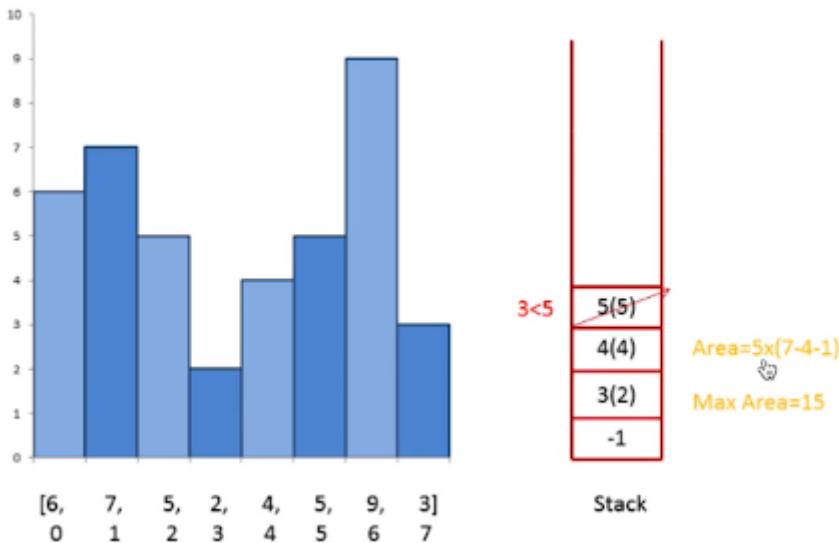
```

方法III：栈方法 ---->时间复杂度O(n)

优化思想---->找到左右边界的时候并不需要用循环，因为前面已经探过一遍了，因此可以使用栈去记录左右边界

维护栈---->从小到大--->大于则进栈，小于则处理栈





```

var largestRectangleArea = function(heights) {
    // Add a zero-height element
    // so that we don't need to deal with last element of heights outside the for loop
    heights.push(0);
    // so that stack always have an extra element, thus `stack[stack.length - 1]` is always valid
    let stack = [-1];
    let maxArea = 0;
    for (let i = 0; i < heights.length; i++) {
        let stackTopElement;
        let minHeight = 0;
        while ((stackTopElement = stack[stack.length - 1]) != -1 && heights[i] <
heights[stackTopElement]) {
            stack.pop();
            minHeight = heights[stackTopElement];
            let count = i - stack.length - 1;
            maxArea = Math.max(maxArea, minHeight * count);
        }
        stack.push(i);
    }
    heights.pop();
    return maxArea;
};

```

8.滑动窗口最大值

239. 滑动窗口最大值

难度 困难 857 收藏 分享 切换为英文 接收动态 反馈

给你一个整数数组 `nums`，有一个大小为 `k` 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 `k` 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

示例 1：

输入: `nums = [1,3,-1,-3,5,3,6,7], k = 3`

输出: `[3,3,5,5,6,7]`

解释:

滑动窗口的位置	最大值
<code>[1 3 -1] -3 5 3 6 7</code>	<code>3</code>
<code>1 [3 -1 -3] 5 3 6 7</code>	<code>3</code>
<code>1 3 [-1 -3 5] 3 6 7</code>	<code>5</code>
<code>1 3 -1 [-3 5 3] 6 7</code>	<code>5</code>
<code>1 3 -1 -3 [5 3 6] 7</code>	<code>6</code>
<code>1 3 -1 -3 5 [3 6 7]</code>	<code>7</code>

方法I：暴力解法O(n*k)

枚举窗口的起点位置 0 ---> `length - k`

写两个嵌套循环去

```
var maxSlidingWindow = function(nums, k) {
    let res=[];
    let max=-Infinity;
    for(let i=0;i<nums.length-k+1;i++){
        for(let j=i;j<i+k;j++){
            let windows = nums[j];
            if(nums[j]>max) max=nums[j]
        }
        res.push(max);
        max = -Infinity;
    }
    return res;
};
```

方法II：双端队列 O(n+k)

解题思路：

双端队列`q[0]`用来存储遍历过的最大值

```
var maxSlidingWindow = function(nums, k) {
    let res=[];
    let q=[];
    for(let i=0;i<nums.length;i++){
        while(q.length-1>=0 && nums[i]>q[q.length-1]) q.pop();
        q.push(nums[i]);
        const j=i-k+1;
```

```

if(j >= 0){
    res.push(q[0]);
    if(nums[j] === q[0]) q.shift() //解决测试用例：[1,-1] 1
}
return res;
};

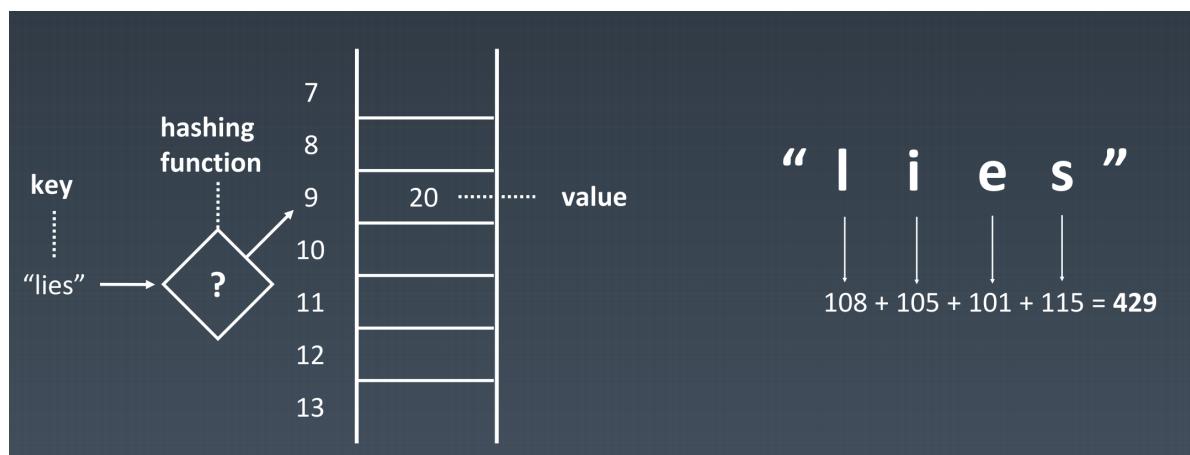
```

Week03

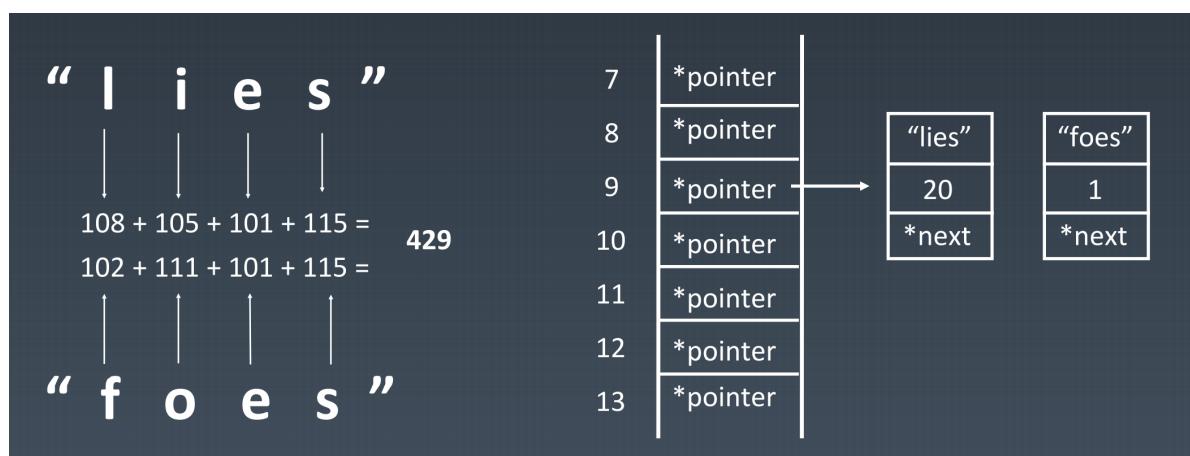
哈希表、映射、集合

哈希表 (Hash table) , 也叫散列表, 是根据关键码值 (Key value) 而直接进行访问的数据结构。它通过把关键码值映射到表中一个位置来访问记录, 以加快查找的速度。这个映射函数叫作散列函数 (Hash Function) , 存放记录的数组叫作哈希表 (或散列表)

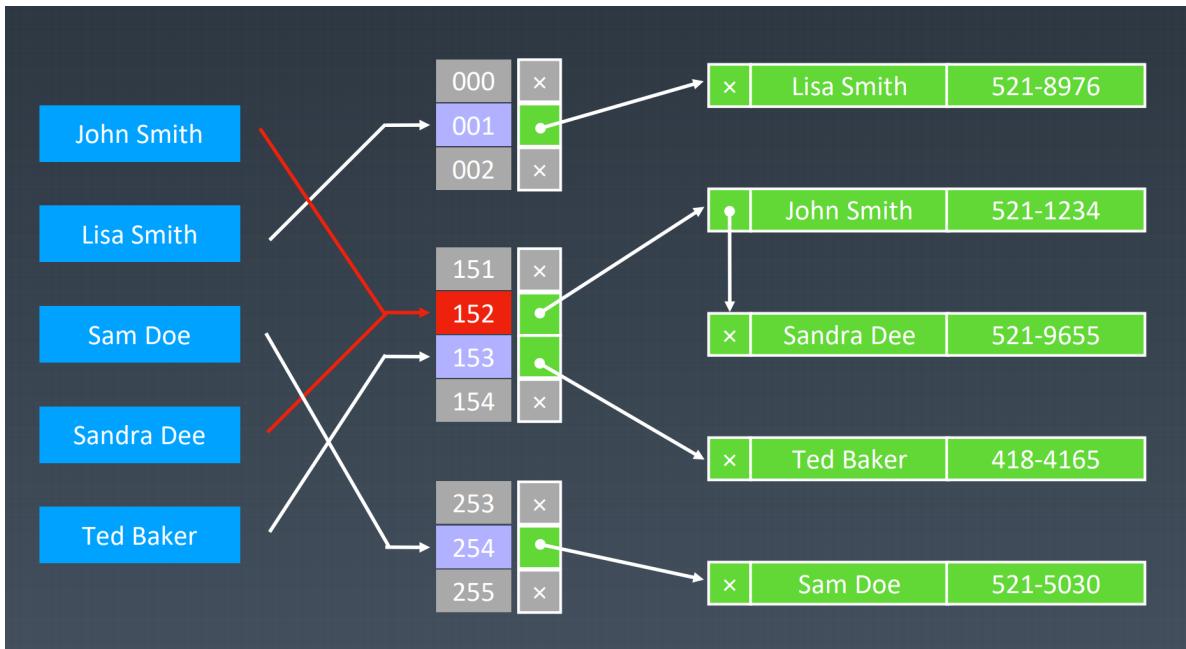
“lies”通过ASCII表中字母数据相加和, 放入hash function中进行计算, 最后返回int值是9;



哈希碰撞: 经过hash function得到相同的index值时



拉链式解决冲突: 在相同的位置拉出一个链表, 将会发生冲突的元素依次存放在链表中



时间复杂度：增删改查O(1) --在不冲突的情况下

Java Code

- Map: key-value对, key不重复value可以重复

```
- new HashMap() / new TreeMap()
- map.set(key, value)
- map.get(key)
- map.has(key)
- map.size()
- map.clear()
```

- Set: 不重复元素的集合

```
- new HashSet() / new TreeSet()
- set.add(value)
- set.delete(value)
- set.hash(value)
```

Python

```
list_x = [1, 2, 3, 4]

map_x = {
    'jack' : 100,
    '张三' : 80,
    'selina' : 90,
    ...
}

set_x = { 'jack' , 'selina' , 'Andy' }
set_y = set([ 'jack' , 'selina' , 'jack' ])
```

map和set的区别 (JavaScript)

set和map的主要应用场景在 数据重构和数据存储

set是一种叫做集合的数据结构， map是一种叫做字典的数据结构

集合和字典的区别

共同点：集合、字典 可以储存不重复的值

不同点：集合是以[value , value]的方式存储，字典是以[value , key]的方式进行存储

集合 (set)

ES6 新增的一种新的数据结构，类似于数组，但成员是唯一且无序的，**没有重复的值**

Set 本身是一种构造函数，用来生成 Set 数据结构

Set 对象允许你储存任何类型的唯一值，无论是原始值或者是对象引用

```
// 定义一个集合
const numberSet = new Set();
// add function
numberSet.add(1);
numberSet.add(2);
numberSet.add(4);
numberSet.add(7);
console.log(numberSet); // {1,2,4,7}

// delete function
numberSet.delete(4);
console.log(numberSet); // {1,2,7}

// has function
console.log(numberSet.has(7)) // true

// 返回set的长度
console.log(numberSet.size) // 3
```

注意：向 Set 加入值的时候，不会发生类型转换，所以 5 和 "5" 是两个不同的值。Set 内部判断两个值是否不同，使用的算法叫做“Same-value-zero equality”，它类似于**精确相等**运算符（`==`），主要的区别是 **Nan 等于自身，而精确相等运算符认为 Nan 不等于自身。**

操作方法

```
add(value); //新增，相当于 array里的push
delete(value); //存在即删除集合中value
has(value); //判断集合中是否存在 value
clear(); //清空集合
```

遍历方法

```
keys(); //返回一个包含集合中所有键的迭代器。
values(); //返回一个包含集合中所有值得迭代器。
entries(); //返回一个包含Set对象中所有元素得键值对迭代器。
forEach(callbackFn, thisArg);
//用于对集合成员执行callbackFn操作，如果提供了 thisArg 参数，回调中的this会是这个参数，没有返回值
```

实战题目

1.有效的字母词异位

242. 有效的字母异位词

难度 简单 336

给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。

示例 1：

输入: $s = "anagram"$, $t = "nagaram"$

输出: true

示例 2：

输入: $s = "rat"$, $t = "car"$

输出: false

异位词：字母出现次数一样，但是顺序不一样

切题4步：

1.看清题意

2.找出最优（时间&空间复杂度最优）

3.写代码

4.测试样例

解题思路

1. 暴力：将字符串按照从大到小sort，比较sorted array ==? $O(N \log N)$

```
var isAnagram = function(s, t) {
    if(Array.from(s).sort().join(" ") == Array.from(t).sort().join("")){
        return true
    }else{
        return false
    }
};
```

2. 哈希表：统计每个字符的频次，在 s 中进行加法，在 t 中进行减法，看最后结果是不是0

step:

构建 countS 和 countT 两个列表，0-25 分别代表 26 个字母

统计不同的字母出现的频次 --> 建立 26 个数组分别代表 a-z

再对两个数组进行比较 --> 数组相等 => 两个数组中出现字母的频次相等

```

var isAnagram = function(s, t) {
    if(s.length !== t.length) return false
    let count = [];
    let ca = 97;
    for(let i=0;i<26;i++){
        count.push(0)
    }
    for(let j=0;j<s.length;j++){
        count[s.charCodeAt(j) - ca]++;
        count[t.charCodeAt(j) - ca]--;
    }
    for(let k=0;k<count.length;k++){
        if(count[k] !== 0) return false
    }
    return true
}

```

2.字母异位词分组

49. 字母异位词分组

难度 中等  650    

给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

示例:

```

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

```

说明:

- 所有输入均为小写字母。
- 不考虑答案输出的顺序。

解题思路:

对于数组中的每一个单词，分别按照字母顺序排序

```

var groupAnagrams = function(strs) {
  let res = {}
  for(let str of strs){
    let tmp = str.split("").sort().join("")//将str按字母排序
    if(res[tmp] == null) //排序后tmp相等的插入到同一级里
      res[tmp] = [str]
    else
      res[tmp].push(str)
  }
  return Object.values(res) //只返回res中的values的值
};

//如果是返回 res 则结果为:
//{ aet: [ 'eat', 'tea', 'ate' ], ant: [ 'tan', 'nat' ], abt: [ 'bat' ] }

```

3. 两数之和

1. 两数之和

难度 简单 10234

给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出 **和为目标值** 的那 **两个** 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍。

你可以按任意顺序返回答案。

示例 1：

```

输入: nums = [2,7,11,15], target = 9
输出: [0,1]
解释: 因为 nums[0] + nums[1] == 9 , 返回 [0, 1] 。

```

示例 2：

```

输入: nums = [3,2,4], target = 6
输出: [1,2]

```

hash --> $a + b == target$ --> 存在

for each a 去检查 $target - a$ 是否也存在于数组中 --> hash table

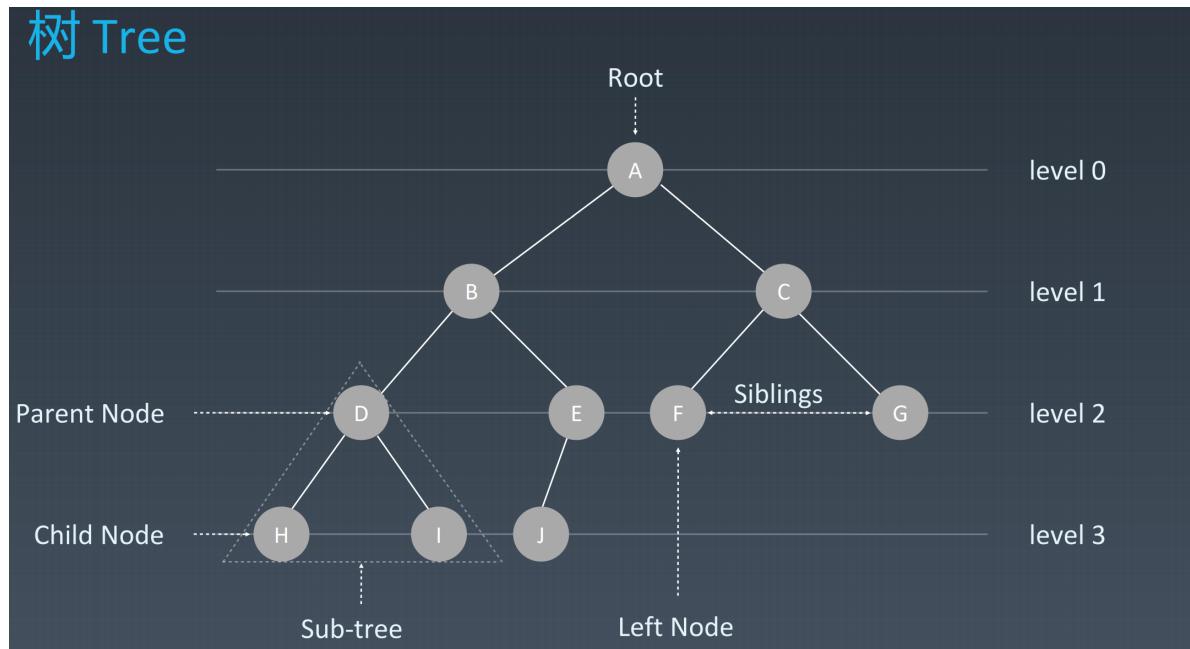
```

var twoSum = function(nums, target) {
  let map={};
  for(let i=0;i<nums.length;i++){
    let n= target - nums[i];
    if(n in map) return [map[n],i];
    map[nums[i]] = i;
  }
};

```

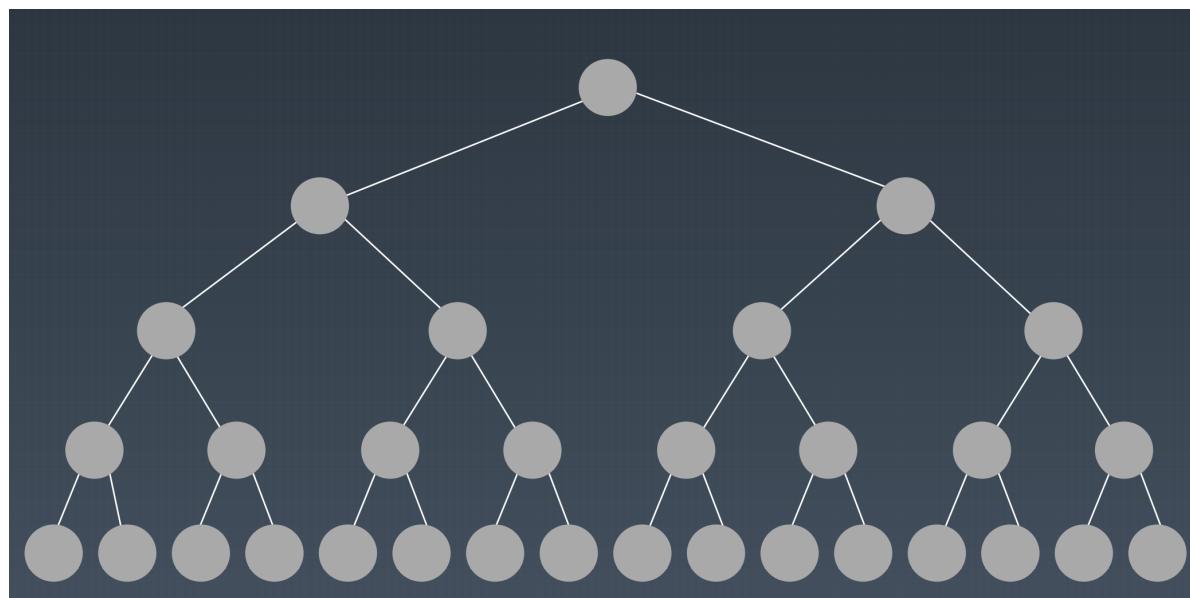
Week03

树、二叉树和二叉搜索树

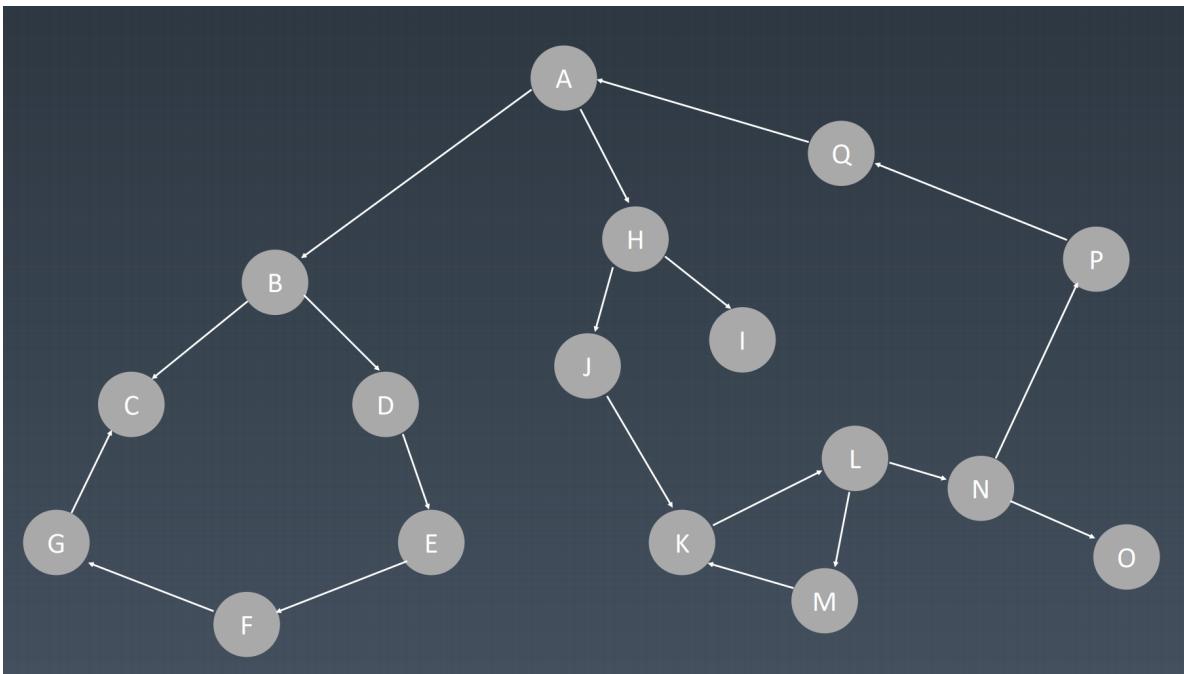


树和图的区别：树没有环，而图永远都有环

二叉树 Binary Tree



Graph



Java Code

```
public class TreeNode {
    public int val;
    public TreeNode left, right;
    public TreeNode(int val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
```

二叉树遍历 Pre-order/In-order/Post-order

- 1.前序 (Pre-order) : 根-左-右
- 2.中序 (In-order) : 左-根-右
- 3.后序 (Post-order) : 左-右-根

树的遍历基于递归

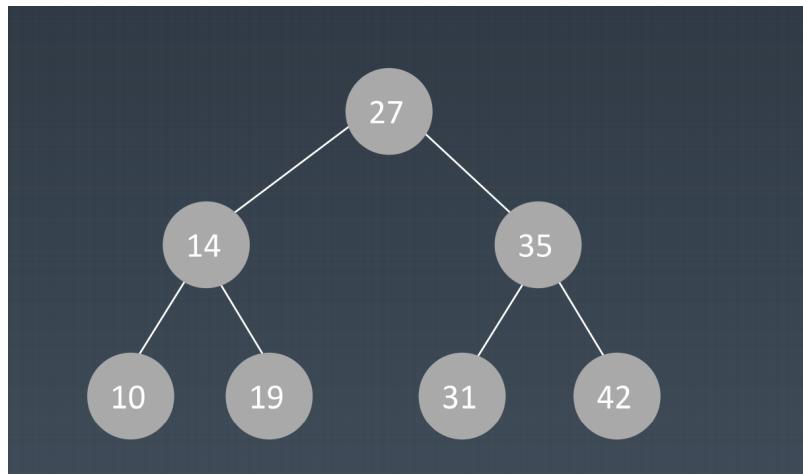
```
def preorder(self, root):
    if root:
        self.traverse_path.append(root.val)
        self.preorder(root.left)
        self.preorder(root.right)

def inorder(self, root):
    if root:
        self.inorder(root.left)
        self.traverse_path.append(root.val)
        self.inorder(root.right)

def postorder(self, root):
    if root:
        self.postorder(root.left)
        self.postorder(root.right)
```

```
self.postorder(root.right)  
self.traverse_path.append(root.val)
```

二叉搜索树 Binary Search Tree



二叉搜索树，也称二叉排序树、有序二叉树（Ordered Binary Tree）、排序二叉树（Sorted Binary Tree），是指一棵空树或者具有下列性质的二叉树：

1. 左子树上**所有结点**的值均小于它的根结点的值； (10&19&14<27)
2. 右子树上**所有结点**的值均大于它的根结点的值； (31&42&35>27)
3. 以此类推：左、右子树也分别为二叉查找树（这就是 重复性！）

中序遍历：升序排列

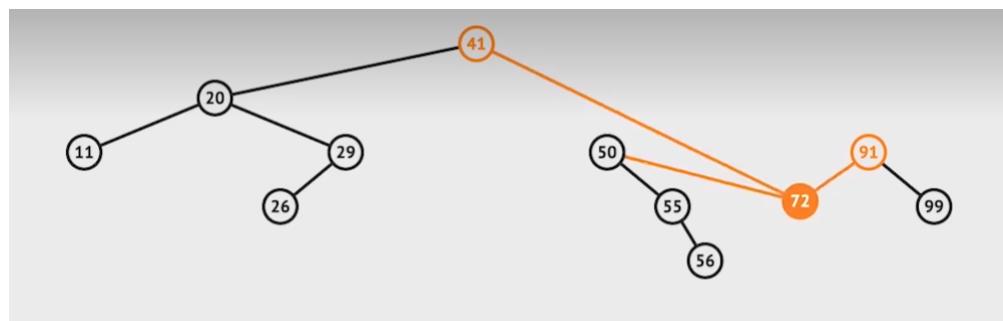
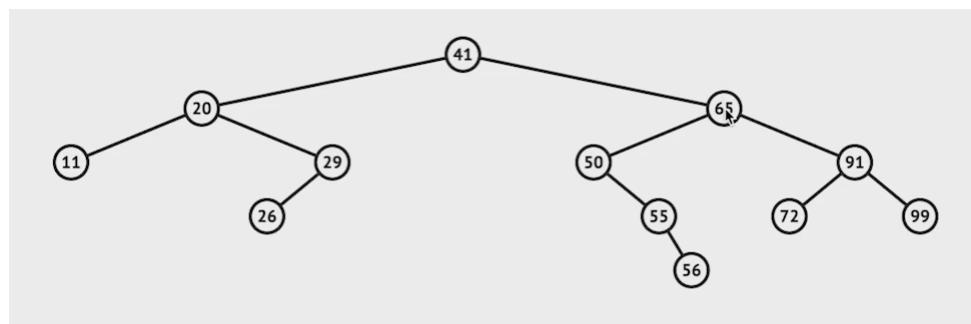
二叉搜索树常见操作

查询 $O(\log N)$ --> 每次只搜索一半的结点

插入新结点 $O(\log N)$

删除 $O(\log N)$

--当删除一些关键性结点时：找到第一个大于所删除结点将其作为新根节点



时间复杂度

查询、插入、删除: $O(\log N)$; 当退化成单列树时, 时间复杂度为 $O(N)$

实战题目

树的遍历

前序遍历: 根-左-右

中序遍历: 在每一个结点中都优先从左边开始遍历, 当左结点全部遍历完成后再对右结点进行遍历

后序遍历: 左-右-根

已知二叉树的前序遍历和中序遍历, 如何得到它的后序遍历

其实, 只要知道其中任意两种遍历的顺序, 我们就可以推断出剩下的一种遍历方式的顺序, 这里我们只是以: 知道前序遍历和中序遍历, 推断后序遍历作为例子, 其他组合方式原理是一样的。要完成这个任务, 我们首先要利用以下几个特性

- 特性A, 对于前序遍历, 第一个肯定是根节点;
- 特性B, 对于后序遍历, 最后一个肯定是根节点;
- 特性C, 利用前序或后序遍历, 确定根节点, 在中序遍历中, 根节点的两边就可以分出左子树和右子树;
- 特性D, 对左子树和右子树分别做前面3点的分析和拆分, 相当于做递归, 我们就可以重建出完整的二叉树;

我们以一个例子做一下这个过程, 假设:

- 前序遍历的顺序是: CABGHEDF
- 中序遍历的顺序是: GHBADEF

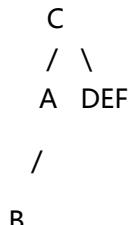
第一步, 我们根据特性A, 可以得知根节点是C, 然后, 根据特性C, 我们知道左子树是: GHBA, 右子树是: DEF。



第二步, 取出左子树, 左子树的前序遍历是: ABGH, 中序遍历是: GHBA, 根据特性A和C, 得出左子树的父节点是A, 并且A没有右子树。



第三步, 使用同样的方法, 前序是BGH, 中序是GHB, 得出父结点是B, H是G的右子树



/

G

/

H

第四步，回到右子树，它的前序是EDF，中序是DEF，依然根据特性A和C，得出父节点是E，左右节点是D和F

C

/ \

A E

/ / \

B D F

/

H

/

G

1. 二叉树的前序遍历

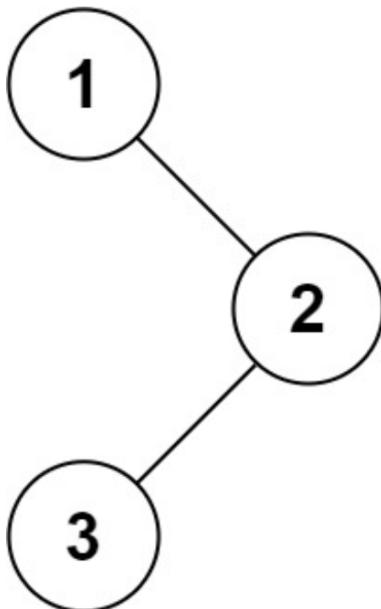
时间复杂度O(n) & 空间复杂度O(n)

144. 二叉树的前序遍历

难度 中等 511 收藏 分享 切换为英文 接收动态 反馈

给你二叉树的根节点 `root`，返回它节点值的 **前序** 遍历。

示例 1：



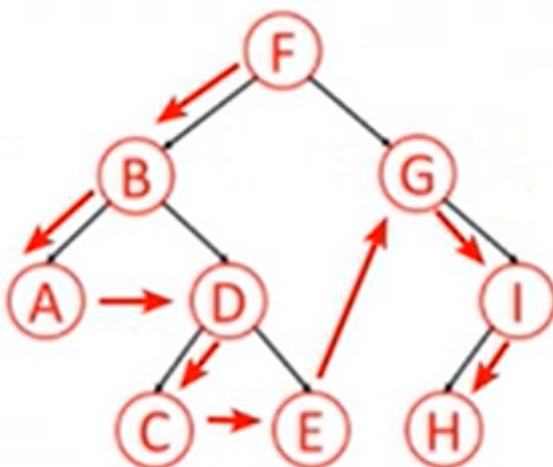
输入: `root = [1,null,2,3]`

输出: `[1,2,3]`

方法I：递归法

解题思路：

将根节点放入`res`中，依次向下递归直到将所有左节点走完，之后再进行右节点的递归



Preorder: `F B A D C E G I H`

```
var preorderTraversal = function(root) {
  if(!root) return[];
  let res = [root.val];
  if(root.left){
    res.push(...preorderTraversal(root.left)))
  }
  if(root.right){
    res.push(...preorderTraversal(root.right)))
  }
  return res;
};
```

方法II：迭代法

解题思路：

设置一个stack栈用来保存遍历过程中另一边的值，res用来保存遍历的结果

stack栈 ---> 先进后出，中序遍历左根右，因此需要先把root.right先push进栈中

2. 二叉树的中序遍历(root->left->right)

时间复杂度O(n) & 空间复杂度O(n)

94. 二叉树的中序遍历

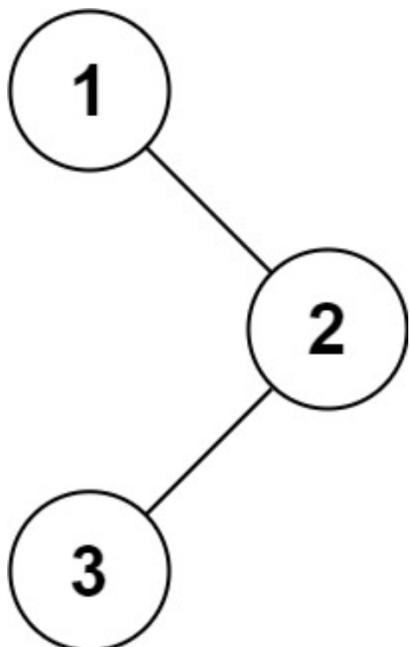
难度 中等

852



给定一个二叉树的根节点 root，返回它的 中序 遍历。

示例 1：



输入: root = [1,null,2,3]

输出: [1,3,2]

3. 二叉树的后序遍历

时间复杂度O(n) & 空间复杂度O(n)

145. 二叉树的后序遍历

难度 中等 517

给定一个二叉树，返回它的 *后序遍历*。

示例：

输入： [1,null,2,3]

```
1
 \
 2
 /
3
```

输出： [3,2,1]

进阶：递归算法很简单，你可以通过迭代算法完成吗？

```
var postorderTraversal = function(root) {
  const res = [];
  const postorder = (root) =>{
    if(!root) return false;
    postorder(root.left);
    postorder(root.right);
    res.push(root.val);
  }
  postorder(root);
  return res;
};
```

4. N叉树的前序遍历

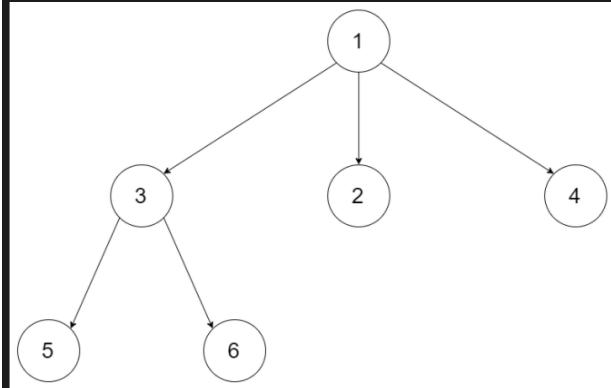
给定一个 N 叉树，返回其节点值的 **前序遍历**。

N 叉树 在输入中按层序遍历进行序列化表示，每组子节点由空值 `null` 分隔（请参见示例）。

进阶：

递归法很简单，你可以使用迭代法完成此题吗？

示例 1：



输入： `root = [1,null,3,2,4,null,5,6]`

输出： `[1,3,5,6,2,4]`

```
var preorder = function(root) {  
    const res = [];  
    const pre = (root) => {  
        if(!root) return;  
        res.push(root.val);  
        root.children.forEach( child => pre(child));  
    }  
    pre(root);  
    return res;  
};
```

5. N叉树的后序遍历

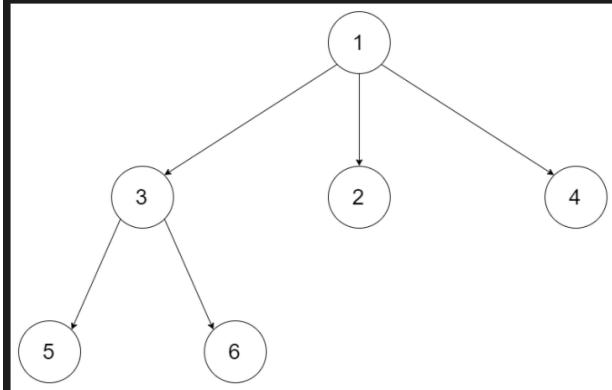
给定一个 N 叉树，返回其节点值的 **后序遍历**。

N 叉树 在输入中按层序遍历进行序列化表示，每组子节点由空值 `null` 分隔（请参见示例）。

进阶：

递归法很简单，你可以使用迭代法完成此题吗？

示例 1：



输入： `root = [1,null,3,2,4,null,5,6]`

输出： `[5,6,3,2,4,1]`

```
var postorder = function(root) {
  const res = [];
  const post = (root) =>{
    if(!root) return;
    root.children.forEach(child => post(child));
    res.push(root.val);
  }
  post(root);
  return res;
};
```

6.二叉树的层序遍历

7.堆和二叉堆的实现和特性

Heap: 可以迅速找到一堆数中的最大值或者最小值的数据结构(堆！=二叉堆)

将根节点最大的堆叫做大顶堆或大根堆，根节点最小的堆叫做小顶堆或者小根堆；常见的堆有二叉堆、斐波那契堆等

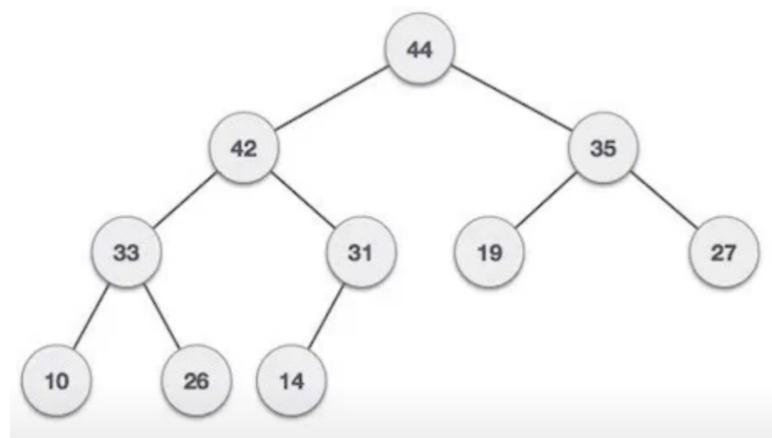
假设是大顶堆，则常见操作（API）：

find-max: O(1)

delete-max: O(logN)

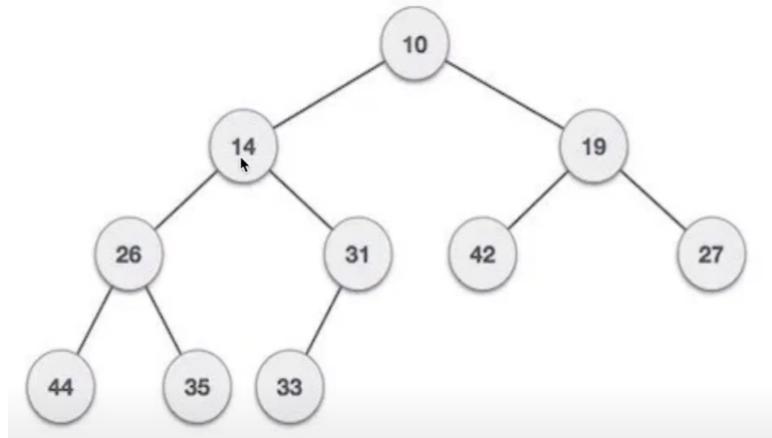
insert (create): O(logN) or O(1)

max-heap:



顶节点的值是最大值，它后面节点的值都不如第一个节点的值大

Min-heap:

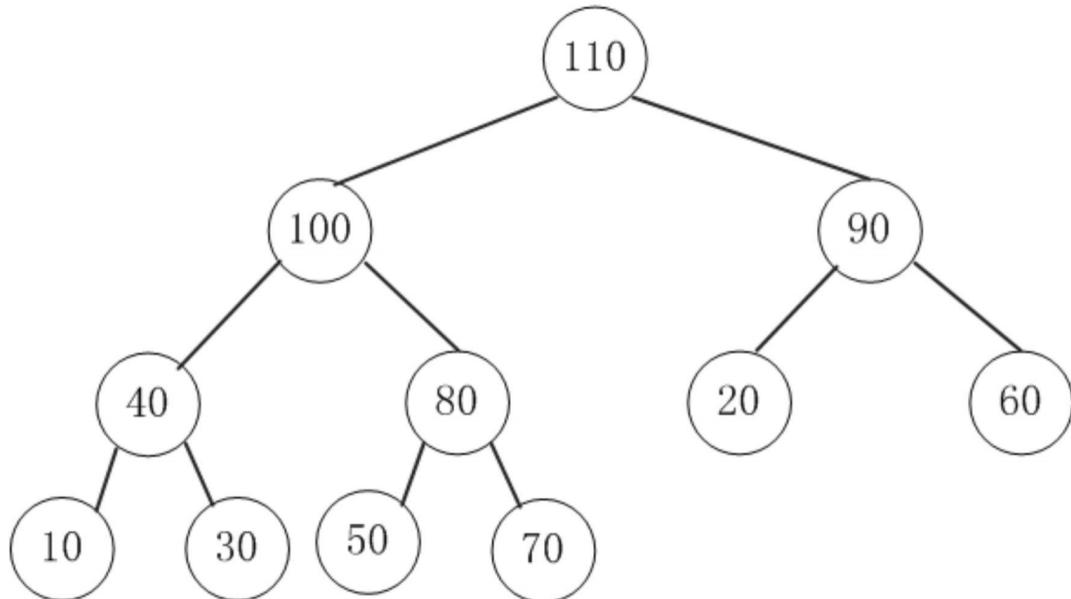


头节点的值比其他节点值都小

堆的实现:

Operation	find-max	delete-max	insert	increase-key	meld
Binary ^[8]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Binomial ^{[8][9]}	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$ ^[b]	$\Theta(\log n)$	$\Theta(\log n)$ ^[c]
Fibonacci ^{[8][10]}	$\Theta(1)$	$\Theta(\log n)$ ^[b]	$\Theta(1)$	$\Theta(1)$ ^[b]	$\Theta(1)$
Pairing ^[11]	$\Theta(1)$	$\Theta(\log n)$ ^[b]	$\Theta(1)$	$\Theta(\log n)$ ^{[b][d]}	$\Theta(1)$
Brodal ^{[14][e]}	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing ^[16]	$\Theta(1)$	$\Theta(\log n)$ ^[b]	$\Theta(1)$	$\Theta(1)$ ^[b]	$\Theta(1)$
Strict Fibonacci ^[17]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2-3 heap ^[18]	$\Theta(\log n)$	$\Theta(\log n)$ ^[b]	$\Theta(\log n)$ ^[b]	$\Theta(1)$?

二叉堆性质



通过完全二叉树来实现 --> (完全二叉树：除了最后一层结点，其他的层都是满的)

二叉堆（大顶）它满足下列性质：

性质一：是一棵完全树

性质二：树中任意结点的值总是 \geq 子结点的值

二叉堆实现细节

1.二叉堆一般都通过“数组”来实现

2.假设“第一个元素”在数组中的索引为0的话，则父节点和子节点的位置关系如下：

O(1) 索引为i的左儿子的索引是： $(2*i + 1)$;

O(2) 索引为i的右儿子的索引是： $(2*i + 2)$;

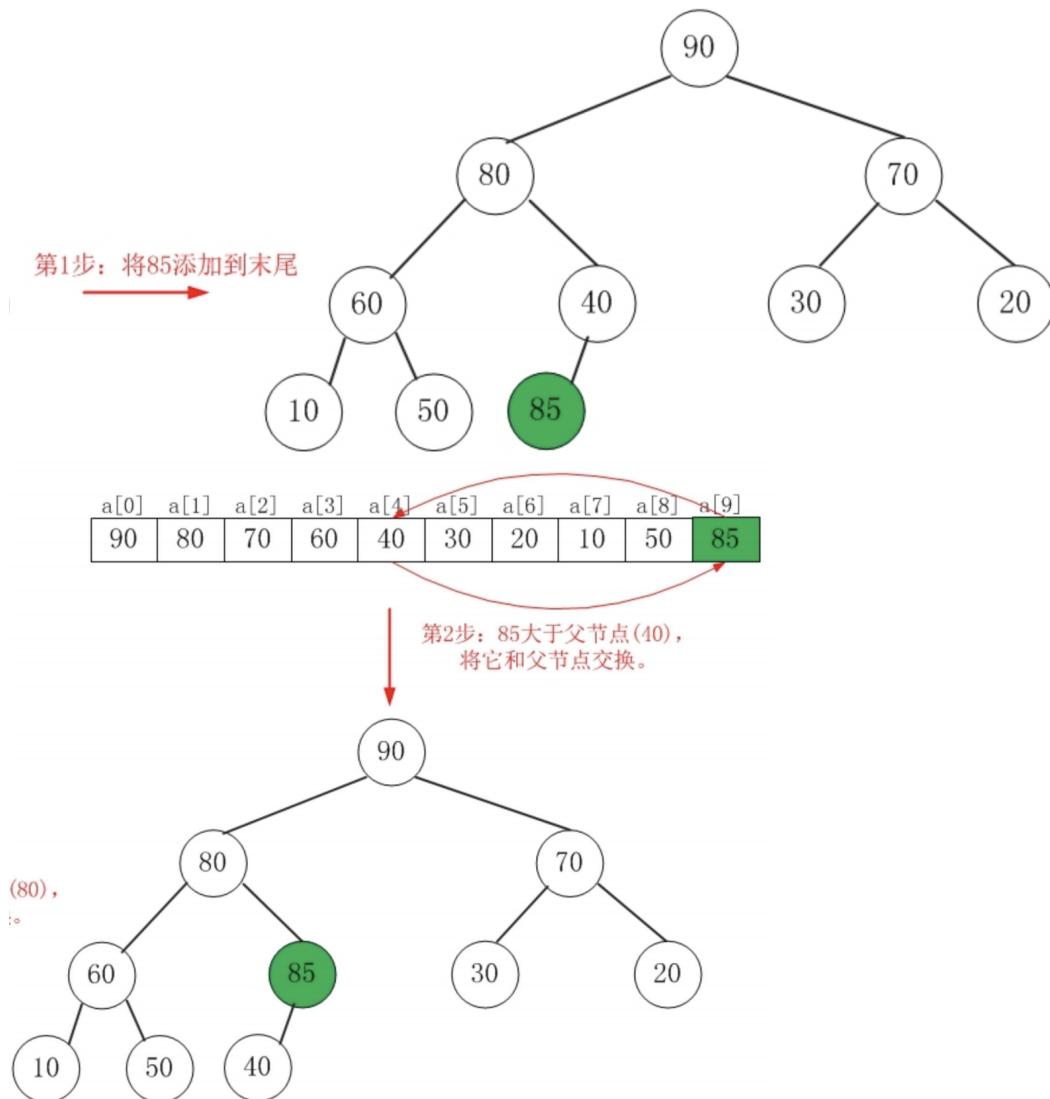
O(3) 索引为i的父结点的索引是： $\text{floor}((i-1)/2)$; 取整

一维数组：[110,100,90,40,80,20,60,10,30,50,70]

Insert 插入操作 O (log2N)

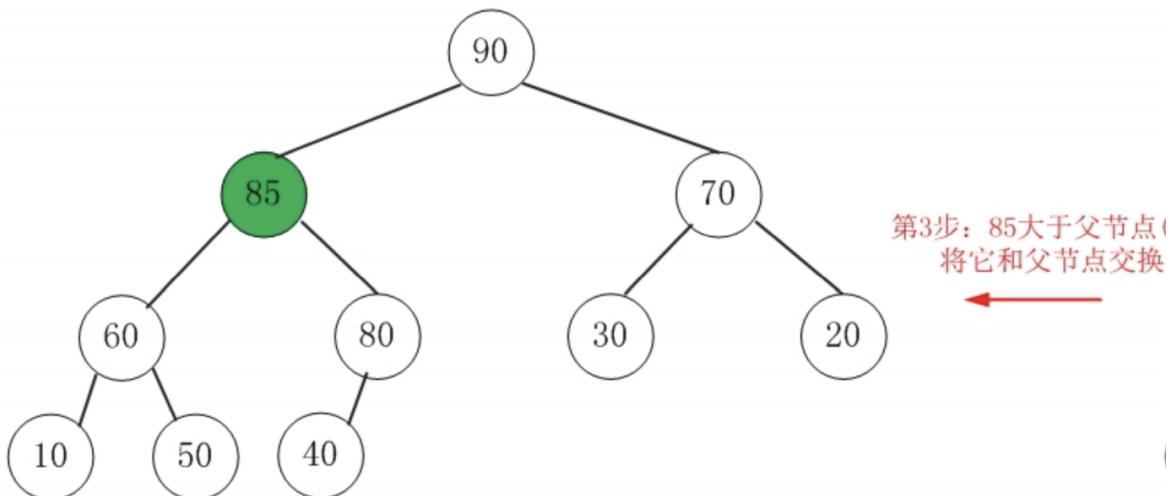
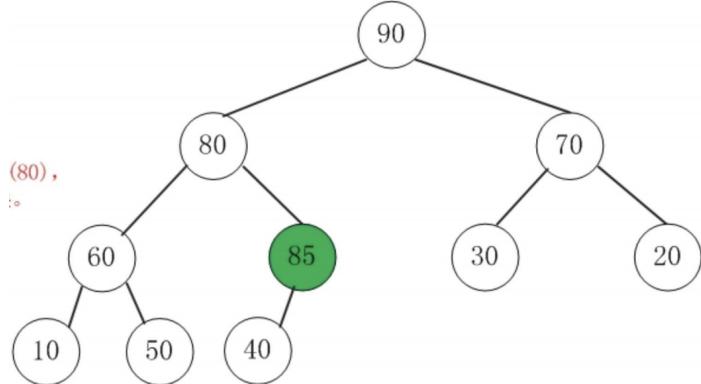
1.新元素一律插到堆的尾部

2.依次向上调整整个堆的结构 (一直到根即可)



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
90	80	70	60	40	30	20	10	50	85

第2步：85大于父节点(40)，将它和父节点交换。

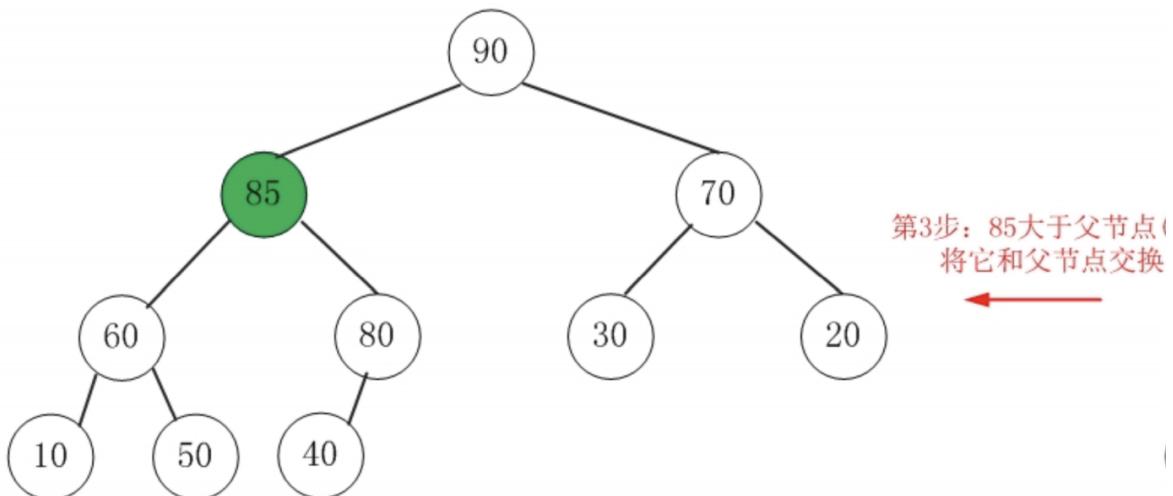
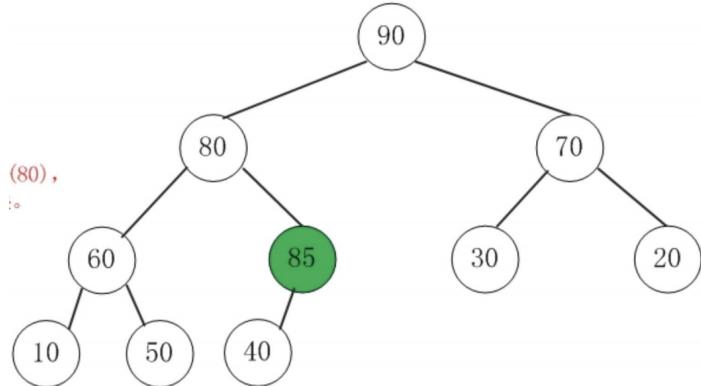


第3步：85大于父节点(80)，将它和父节点交换

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
90	85	70	60	80	30	20	10	50	40

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
90	80	70	60	40	30	20	10	50	85

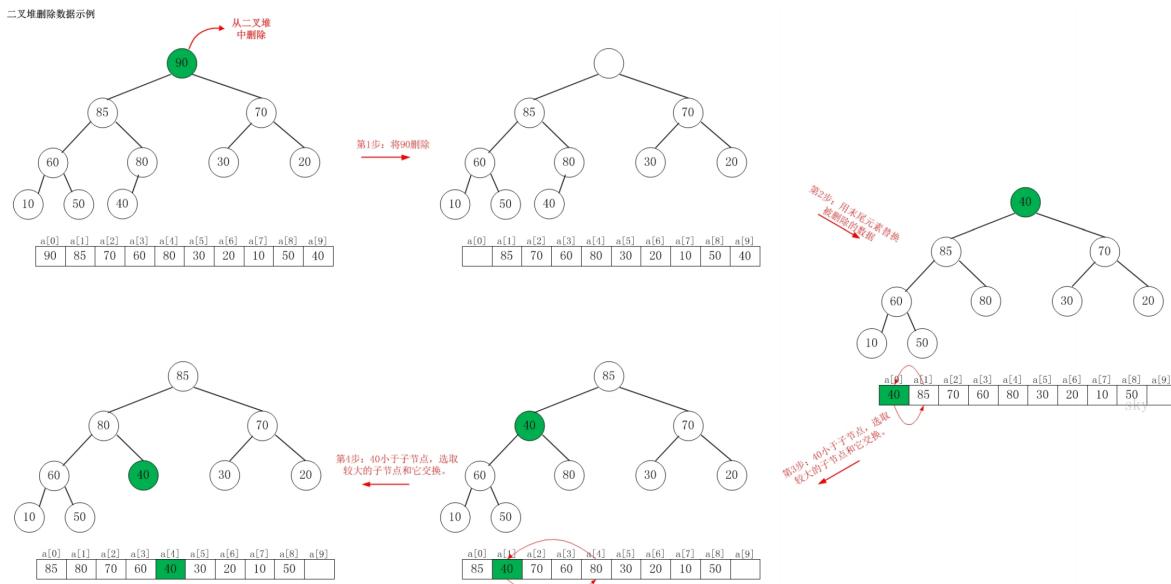
第2步：85大于父节点(40)，将它和父节点交换。



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
90	85	70	60	80	30	20	10	50	40

Delete Max 删除堆顶元素 O(log2N)

1. 将堆尾元素替换到顶部
2. 依次从根部向下调整整个堆的数据结构



Java Code

```
//定义一个二叉堆
public class BinaryHeap {
    private static final int d = 2; //表示二叉堆
    private int[] heap; //用一维数组实现
    private int heapSize;

    public boolean isEmpty(){
        return heapSize == 0;
    }

    public boolean isFull(){
        return heapSize == heap.length;
    }

    private int parent(int i){ //求i的parent结点的索引
        return (i-1)/d;
    }

    private int kthChild(int i,int k){ //求索引为i的子元素的结点
        return d * i+k;
    }

    //二叉堆的插入
    public void insert(int x){
        if(isFull()){
            throw new NoSuchElementException("Heap is full, No space to insert new element.");
        }
        heap[heapSize] = x;
        heapSize++;
        heapifyUp(heapSize - 1); //调用heapifyUp这个函数
    }

    //heapifyUp函数
    private void heapifyUp(int i){
        int insertValue = heap[i];
        while(i>0 && insertValue > heap[parent(i)]){ //一直找到insertValue应该存入的点
            heap[i] = heap[parent(i)];
            i = parent(i);
        }
        heap[i] = insertValue;
    }
}
```

```

}

//二叉堆的删除
public int delete(int x){
    if(isEmpty()){
        throw new NoSuchElementException("Heap is empty,No element to delete");
    }
    int key= heap[x];
    heap[x] = heap[heapSize - 1];
    heapSize--;
    heapifyDown(x);
    return key;
}
//heapifyDown函数
private void heapifyDown(int i){
    int child;
    int temp = heap[i];
    while(kthChild(i,k:1) < heapSize){
        child = maxChild(i); //调用maxChild函数
        if(temp >= heap[child]) { //如果父结点大于子结点, 满足堆的性质
            break;
        }
        heap[i] = heap[child]; //把儿子的值赋给他本身
        i = child;
    }
    heap[i] = temp;
}

//maxChild函数
private int maxChild(int i){
    int leftChild = kthChild(i,k:1);
    int rightChild = kthChild(i,k:2);
    return heap[leftChild] > heap[rightChild] ? leftChild : rightChild;
}
}

```

JavaScript Code ---- Minheap

heap的: 索引值

索引为i的左孩子的索引是(2 * i + 1);

索引为i的右孩子的索引是(2 * i + 2);

索引为i的父结点的索引是(i - 2) / 2;

```

class MinHeap {
    constructor() {
        // index为0时赋值null,便于计算子节点index和父节点index的关系
        this.heap = [null]
    }

    insert(node) {
        this.heap.push(node);
        if (this.heap.length > 1) {
            let current = this.heap.length - 1;
            // ->> heapifyUp
            while (current > 1 &&
                   this.heap[Math.floor(current / 2)] > this.heap[current]) {

```

```

// 交换当前节点和父节点
[this.heap[Math.floor(current / 2)], this.heap[current]] = [this.heap[current],
this.heap[Math.floor(current / 2)]];
current = Math.floor(current / 2);
}
// <<- heapifyUp
}
}

/** 删除堆顶元素 */
remove() {
let smallest = this.heap[1];
if (this.heap.length > 2) {
this.heap[1] = this.heap[this.heap.length - 1];
this.heap.splice(this.heap.length - 1);
if (this.heap.length === 3) {
if (this.heap[1] > this.heap[2]) {
[this.heap[1], this.heap[2]] = [this.heap[2], this.heap[1]];
}
return smallest;
}
// ->> heapifyDown
let current = 1;
let leftChildIndex = current * 2;
let rightChildIndex = current * 2 + 1;

while (
this.heap[leftChildIndex] &&
this.heap[rightChildIndex] &&
(this.heap[current] > this.heap[leftChildIndex] ||
this.heap[current] > this.heap[rightChildIndex])) {
if (this.heap[leftChildIndex] < this.heap[rightChildIndex]) {
[this.heap[current], this.heap[leftChildIndex]] = [this.heap[leftChildIndex],
this.heap[current]];
current = leftChildIndex;
} else {
[this.heap[current], this.heap[rightChildIndex]] = [this.heap[rightChildIndex],
this.heap[current]];
current = rightChildIndex;
}
leftChildIndex = current * 2;
rightChildIndex = current * 2 + 1;
// <<- heapifyDown
}
} else if (this.heap.length === 2) {
this.heap.splice(1, 1);
} else {
return null;
}

return smallest;
}
}

```

JavaScript Code --- Maxheap

```
class MaxHeap {
    constructor() {
        this.heap = [null]
    }

    insert(node) {
        this.heap.push(node);

        if (this.heap.length > 1) {
            let current = this.heap.length - 1;
            // ->> heapifyUp
            while (current > 1 &&
                this.heap[Math.floor(current / 2)] < this.heap[current]) {
                // 交换当前节点和父节点
                [this.heap[Math.floor(current / 2)], this.heap[current]] = [this.heap[current],
                this.heap[Math.floor(current / 2)]];
                current = Math.floor(current / 2);
            }
            // <<- heapifyUp
        }
    }

    remove() {
        let smallest = this.heap[1];
        if (this.heap.length > 2) {
            this.heap[1] = this.heap[this.heap.length - 1];
            this.heap.splice(this.heap.length - 1);
            if (this.heap.length === 3) {
                if (this.heap[1] < this.heap[2]) {
                    [this.heap[1], this.heap[2]] = [this.heap[2], this.heap[1]];
                }
                return smallest;
            }
            // ->> heapifyDown
            let current = 1;
            let leftChildIndex = current * 2;
            let rightChildIndex = current * 2 + 1;

            while (
                this.heap[leftChildIndex] &&
                this.heap[rightChildIndex] &&
                (this.heap[current] < this.heap[leftChildIndex] ||
                 this.heap[current] < this.heap[rightChildIndex])) {
                if (this.heap[leftChildIndex] > this.heap[rightChildIndex]) {
                    [this.heap[current], this.heap[leftChildIndex]] = [this.heap[leftChildIndex],
                    this.heap[current]];
                    current = leftChildIndex;
                } else {
                    [this.heap[current], this.heap[rightChildIndex]] = [this.heap[rightChildIndex],
                    this.heap[current]];
                    current = rightChildIndex;
                }
                leftChildIndex = current * 2;
                rightChildIndex = current * 2 + 1;
            }
        }
    }
}
```

```

    // <<- heapifyDown
}

} else if (this.heap.length === 2) {
    this.heap.splice(1, 1);
} else {
    return null;
}

return smallest;
}
}

```

实战例题

1.最小的k个数

剑指 Offer 40. 最小的k个数

难度 简单 189 ⭐ ⚡ 🔍

输入整数数组 arr，找出其中最小的 k 个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

示例 1：

输入：arr = [3,2,1], k = 2
输出：[1,2] 或者 [2,1]

示例 2：

输入：arr = [0,1,2,1], k = 1
输出：[0]

方法1：sort --> O(NlogN)

```

var getLeastNumbers = function(arr, k) {
    let res = [];
    arr.sort((a,b)=>b-a);
    for(let i=0;i<k;i++){
        res.push(arr.pop());
    }
    return res;
};

```

一行代码

```

var getLeastNumbers = function(arr, k) {
    return arr.sort((a,b)=>a-b).slice(0,k);
};

```

方法II：heap --> O(NlogK)

建立一个空heap，将arr中的数存放到heap中（小顶堆）；依次从heap中取k个元素

js手动写一个minHeap（学习为主）

```

/*
 * @param {number[]} arr
 * @param {number} k
 * @return {number[]}
 */
class MinHeap{
    constructor(arr){
        this.heap = (arr && arr.sort((a, b) => a - b)) || ["empty!"];
    }
    push(newVal){
        let newIndex = this.heap.length;
        this.heap[newIndex] = newVal;//将新元素放在最后一个元素，然后向上调整。
        this.siftUp(newIndex);
    }
    /**向上调整的过程需要注意
     * 调整到index == 1的时候，说明已经到头了，就不需要调整了
     */
    siftUp(index){
        if(index == 1){
            return;
        }
        let father = (index >> 1);
        if(this.heap[index] < this.heap[father]){
            let temp = this.heap[index];
            this.heap[index] = this.heap[father];
            this.heap[father] = temp;
            this.siftUp(father);
        }else{
            return;
        }
    }
    pop(){
        if(this.heap.length == 2){
            return this.heap.pop();
        }
        if(this.heap.length == 1){
            return this.heap[0];
        }
        let res = this.heap[1];
        this.heap[1] = this.heap.pop(); // 将最后一个元素放到第一个元素，然后进行向下调整
        this.siftDown(1);
        return res;
    }
    /**向下调整的过程要进行判断
     * 首先，应该选择最小的那个元素交换，如果较小的孩子还是比自己大，那就不需要交换了，说明找到了位
     置
     * 如果右孩子不为空，那么就说明，左右孩子都不为空，那么就要与两个孩子都比较，然后交换
    */
}
```

```

* 如果右孩子为空，但是左孩子不为空，那么久要与左孩子比较，然后交换
* 如果左孩子为空，就不需要再移动了，说明当前这个位置就是正确的
*/
siftDown(index){
    let left = index * 2;
    let right = index * 2 + 1;
    if(this.heap[right] != undefined){
        let next = this.heap[left] < this.heap[right] ? left : right;
        if(this.heap[next] < this.heap[index]){
            let temp = this.heap[index];
            this.heap[index] = this.heap[next];
            this.heap[next] = temp;
            this.siftDown(next);
        }else{
            return;
        }
    }else if(this.heap[left] != undefined){
        let next = left;
        if(this.heap[next] < this.heap[index]){
            let temp = this.heap[index];
            this.heap[index] = this.heap[next];
            this.heap[next] = temp;
        }else{
            return;
        }
    }else if(this.heap[left] == undefined){
        return;
    }
}
peak(){
    return this.heap[1] || this.heap[0];
}
}

var getLeastNumbers = function(arr, k) {
    let heap = new MinHeap();
    for(let i = 0; i < arr.length; i++){
        heap.push(arr[i]);
    }
    let res = [];
    for(let i = 0; i < k; i++){
        res.push(heap.pop());
    }
    return res;
};

```

方法III： quick-sort(快速排序) -->时间复杂度NlogN

```

var getLeastNumbers = function(arr, k) {
    return quickSort(arr).slice(0,k);
};

function quickSort(arr) {
    if(arr.length <=1){
        return arr;
    }
    const pivot = arr[arr.length - 1];
    const leftArr = [];
    const rightArr = [];

```

```

for(const el of arr.slice(0,arr.length - 1)){
    el < pivot ? leftArr.push(el) : rightArr.push(el);
}
return [...quickSort(leftArr),pivot,...quickSort(rightArr)];
}

```

2.滑动窗口中的最大值

给你一个整数数组 `nums`，有一个大小为 `k` 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 `k` 个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值。

示例 1:

输入: `nums = [1,3,-1,-3,5,3,6,7]`, `k = 3`

输出: `[3,3,5,5,6,7]`

解释:

滑动窗口的位置	最大值
-----	-----
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

方法I: 双端队列

什么是双端队列:

<https://leetcode-cn.com/problems/sliding-window-maximum/solution/yi-ge-shi-pin-kan-dong-shuang-duan-dui-l-tqgn/>

```

var maxSlidingWindow = function(nums, k) {
    let deque = []
    let res = []
    for(let i = 0; i < nums.length; i++){
        while(deque && nums[deque[deque.length - 1]] < nums[i]){ //cutting tails
            deque.pop()
        }
        deque.push(i)//new tail-->this means nodes in the deque are decreaseing
        if(deque[0] === i - k){ //cutting heads, now the head is the biggest and the newest
            deque.shift()
        }
        if(i >= k - 1){ //pushing the res node when long enough
            res.push(nums[deque[0]])
        }
    }
    return res
}

```

方法II：heap (js需要手写heap，再进行调用)

```
var medianSlidingWindow = function(nums, k) {
  const window = new Window();
  for (let i = 0; i < k - 1; i++) window.add(nums[i]);
  let res = [];
  for (let i = k - 1; i < nums.length; i++) {
    window.add(nums[i]);
    res.push(window.median());
    window.remove(nums[i - k + 1]);
  }
  return res;
};

class Window {
  constructor() {
    this.minHeap = new Heap((a,b) => a < b);
    this.maxHeap = new Heap((a,b) => a > b);
  }

  add(value) {
    this.heap(value).add(value);
    this.balance();
  }

  remove(value) {
    this.heap(value).remove(value);
    this.balance();
  }

  median() {
    if (this.minHeap.size() === this.maxHeap.size()) {
      return (this.minHeap.peak() + this.maxHeap.peak()) / 2;
    }
    return this.minHeap.peak();
  }

  heap(value) {
    return BigInt(value) < this.median() ? this.maxHeap : this.minHeap
  }

  balance() {
    const diff = this.maxHeap.size() - this.minHeap.size()
    if (diff > 0) this.minHeap.add(this.maxHeap.pop());
    else if (diff < -1) this.maxHeap.add(this.minHeap.pop());
  }
}

class Heap {
  constructor(fn) {
    this.store = [];
    this.fn = fn;
  }

  peak() {
    return this.store[0] || 0;
  }
```

```

size() {
  return this.store.length;
}

isEmpty() {
  return this.store.length === 0;
}

add(value) {
  this.store.push(value);
  this.heapifyUp(this.store.length - 1);
}

remove(value) {
  const idx = this.store.indexOf(value);
  if (idx === this.store.length - 1) return this.store.pop();
  this.store[idx] = this.store.pop();
  this.heapifyDown(this.heapifyUp(idx));
}

pop0 {
  if (this.store.length < 2) return this.store.pop();
  const result = this.store[0];
  this.store[0] = this.store.pop();
  this.heapifyDown(0);
  return result;
}

heapifyDown(parent) {
  const childs = [1,2].map((n) => parent * 2 + n).filter((n) => n < this.store.length);
  let child = childs[0];
  if (childs[1] && this.fn(this.store[childs[1]], this.store[child])) {
    child = childs[1];
  }
  if (child && this.fn(this.store[child], this.store[parent])) {
    const temp = this.store[child];
    this.store[child] = this.store[parent];
    this.store[parent] = temp;
    return this.heapifyDown(child);
  }
  return parent;
}

heapifyUp(child) {
  const parent = Math.floor((child - 1) / 2);
  if (child && this.fn(this.store[child], this.store[parent])) {
    const temp = this.store[child];
    this.store[child] = this.store[parent];
    this.store[parent] = temp;
    return this.heapifyUp(parent);
  }
  return child;
}

```

3.前k个高频元素 (高频题目)

```
// Time Complexity: O(nlog(n))
// Space Complexity: O(n)
```

347. 前 K 个高频元素

难度 中等 630

给定一个非空的整数数组，返回其中出现频率前 k 高的元素。

示例 1:

```
输入: nums = [1,1,1,2,2,3], k = 2
输出: [1,2]
```

示例 2:

```
输入: nums = [1], k = 1
输出: [1]
```

提示:

你可以假设给定的 k 总是合理的，且 $1 \leq k \leq$ 数组中不相同的元素的个数。

你的算法的时间复杂度必须优于 $O(n \log n)$ ， n 是数组的大小。

题目数据保证答案唯一，换句话说，数组中前 k 个高频元素的集合是唯一的。

你可以按任意顺序返回答案。

解题思路:

1.用哈希表对数字进行统计 $O(N)$

2.利用Object.keys(hashmap)获取哈希表中的键，对值进行排序，返回前 k 个高频元素

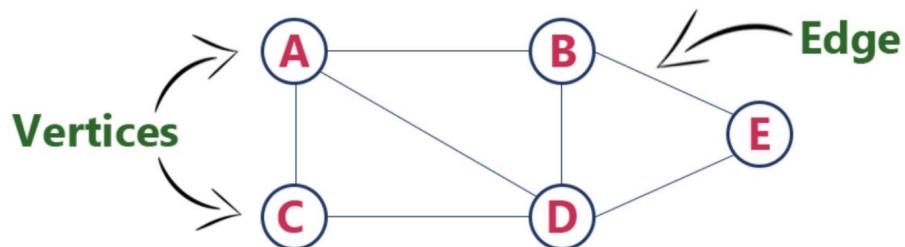
3.顺序输出前 k 个最大元素

```
var topKFrequent = function(nums, k) {
    if(nums.length<=1) return nums;
    let hashmap = {};
    for(let num of nums){
        hashmap[num] == undefined ? hashmap[num]=1: hashmap[num]++;
    }
    return Object.keys(hashmap).sort((a,b)=>hashmap[b] - hashmap[a]).slice(0,k);
};
```

图的实现和特性

图的属性和分类

图的定义：有点+有边



图的属性

Graph(V,E)

V - vertex:点

度-入度和出度 --> 度：一个点连了多少条边

点与点之间：连通与否

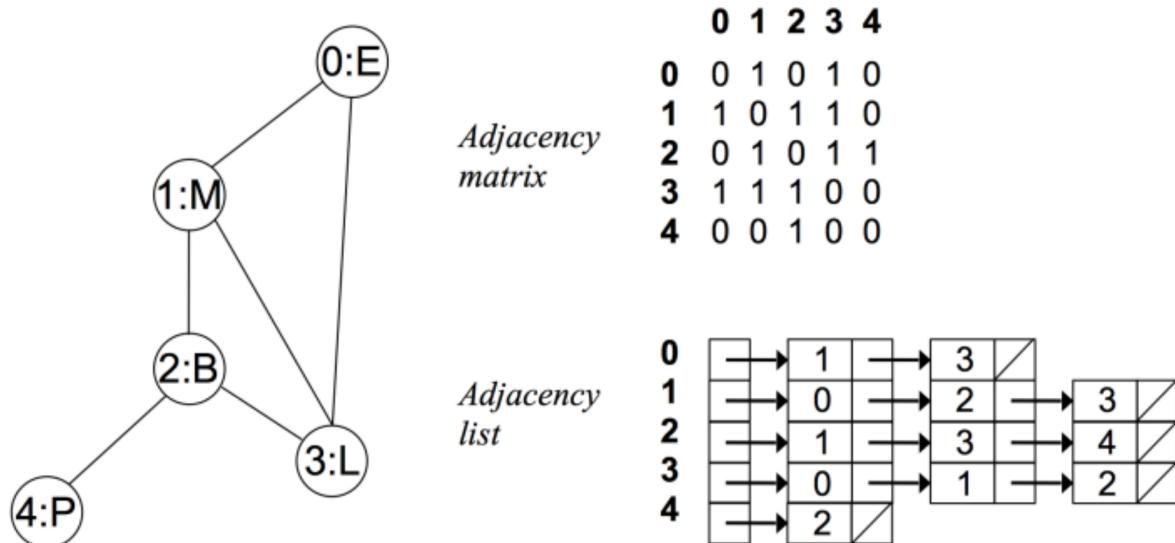
E - edge: 边

有向和无向 (单行线)

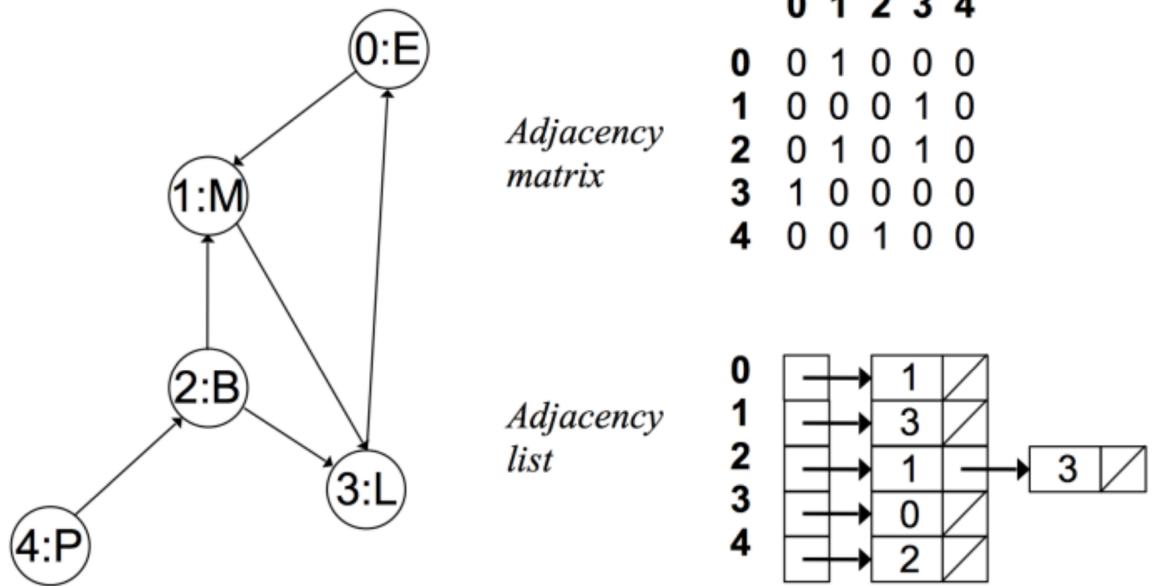
权重 (边长)

图的表示和分类

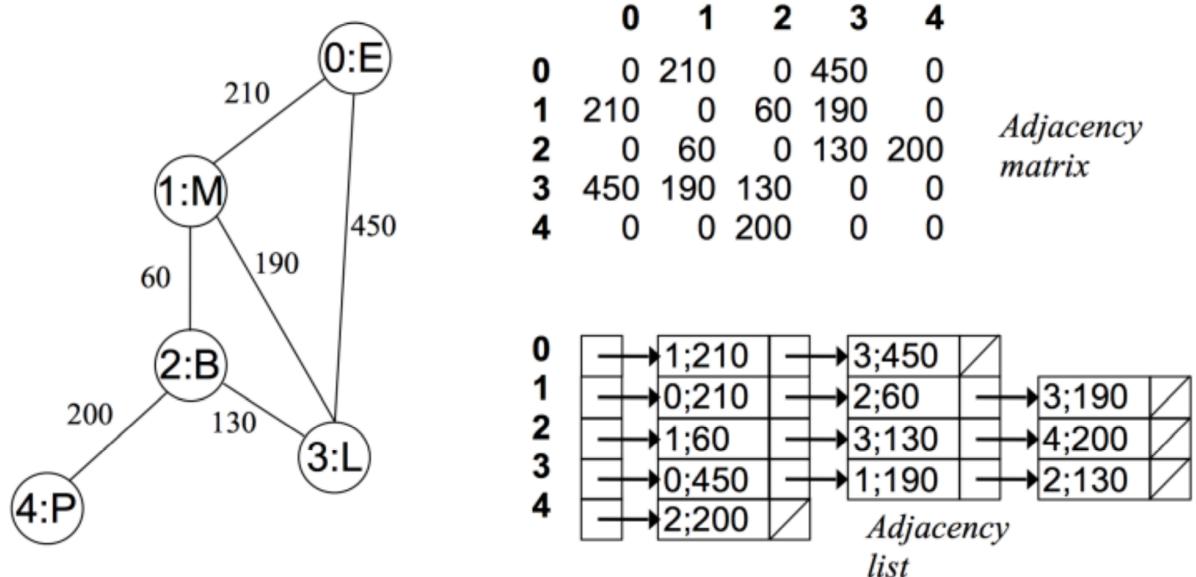
图：无向无权图



图：有向无权图



图：无向有权图



基于图的相关计算（面试考的少）

代码模板一定要死记硬背 --> 条件反射

DFS代码 - 递归写法

```

visited = set() # 和树中的DFS最大区别

def dfs(node, visited):
    if node in visited: # terminator
        # already visited
        return

    visited.add(node)
    # process current node here.
    ...
    for next_node in node.children():
        if not next_node in visited:
            dfs(next_node, visited)

```

BFS代码

```

def BFS(graph, start, end):
    queue = []
    queue.append([start])
    visited = set() # 和树中的BFS的最大区别

    while queue:
        node = queue.pop()
        visited.add(node)

        process(node)
        nodes = generate_related_nodes(node)
        queue.push(nodes)

```

图的高级算法

1.连通图个数

<https://leetcode-cn.com/problems/number-of-islands/>

2.拓扑排序

<https://zhuanlan.zhihu.com/p/34871092>

3.最短路径

<https://www.bilibili.com/video/av25829980?from=search&seid=13391343514095937158>

4.最小生成树

<https://www.bilibili.com/video/av84820276?from=search&seid=17476598104352152051>

week03-递归

泛型递归、树的递归

树的面试题解法一般都是递归

原因： 1.节点的定义（用递归来实现） 2.重复性（自相似性）

树的前中序遍历

```

def preorder(self,root):

```

```

if root:
    self.traverse_path.append(root.val)
    self.preorder(root.left)
    self.preorder(root.right)

def inorder(self, root):
    if root:
        self.inorder(root.left)
        self.traverse_path.append(root.val)
        self.inorder(root.right)

def postorder(self, root):
    if root:
        self.postorder(root.left)
        self.postorder(root.right)
        self.traverse_path.append(root.val)

```

递归 Recursion

递归 - 循环

通过函数体来进行的循环

递归的特点

不能跳跃，必须一层一层的下；再一层一层地回来

通过参数来进行函数不同层之间的传递变量

发生和携带变化

简单的递归

计算 $n!$

$n! = 1 * 2 * 3 * \dots * n$

```

def Factorial(n):
    if n<=1:
        return 1
    return n*Factorial(n-1)

```

```

factorial(6)
6 * factorial(5)
6 * (5 * factorial(4))
6 * (5 * (4 * factorial(3)))
6 * (5 * (4 * (3 * factorial(2)))))
6 * (5 * (4 * (3 * (2 * factorial(1))))))
6 * (5 * (4 * (3 * (2 * 1)))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
6 * (5 * 24)
6 * 120
720

```

结构模块总共有4块

- (1) 递归终结条件
- (2) 处理当前层逻辑
- (3) 下探到下一层
- (4) 清理当前层

Java代码模板

```
public void recur(int level, int param) {  
    // terminator  
    if (level > MAX_LEVEL) { // process result  
        return;  
    }  
  
    // process current logic  
    process(level, param);  
  
    // drill down  
    recur(level + 1, newParam);  
  
    // restore current status  
}
```

JavaScript递归模板

```
const recursion = (level, params) =>{  
    // recursion terminator  
    if(level > MAX_LEVEL){  
        process_result  
        return  
    }  
  
    // process current level  
    process(level, params)  
  
    //drill down  
    recursion(level+1, params)  
  
    //clean current level status if needed  
}
```

养成机械化的记忆

一开始写递归，能马上写下来模板

思维要点

不要人肉进行递归（最大误区）

找到最近最简方法，将其拆解成可重复解决的问题（重复子问题）

数学归纳法思维

实战题目

1.爬楼梯

70. 爬楼梯

难度 简单 1466 ★ ⚡ 文章 ⌂

假设你正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定 n 是一个正整数。

示例 1：

输入： 2

输出： 2

解释： 有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶

2. 2 阶

示例 2：

输入： 3

输出： 3

解释： 有三种方法可以爬到楼顶。

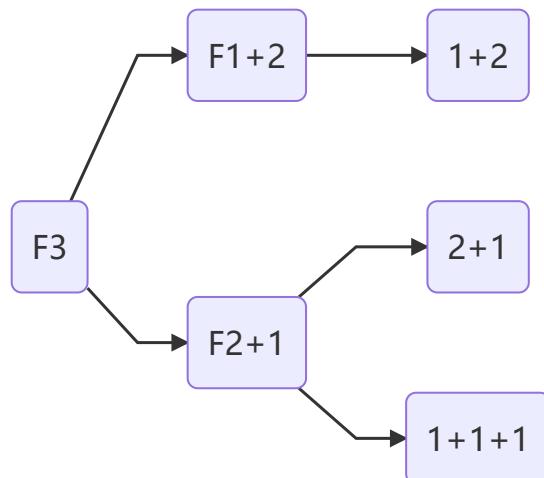
1. 1 阶 + 1 阶 + 1 阶

2. 1 阶 + 2 阶

3. 2 阶 + 1 阶

解题思路：

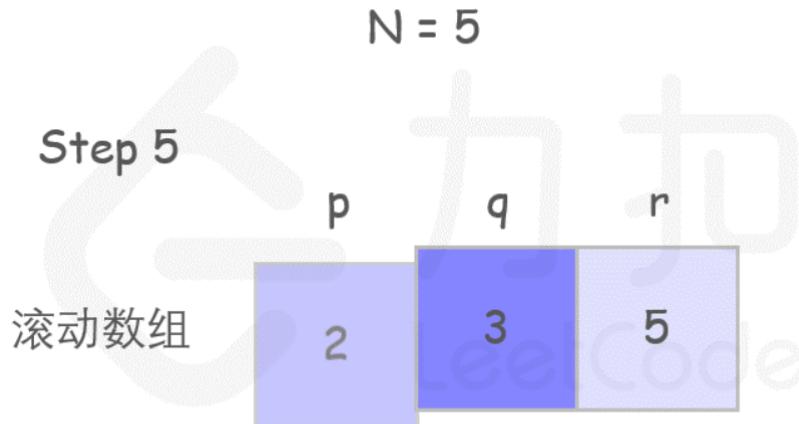
每一级台阶可以选择跨一步或者是跨两步



方法I：数组

```
var climbStairs = function(n){  
    let res=[0,1,2];  
    for(let i=3;i<=n;i++){  
        res[i] = res[i-1] + res[i-2];  
    }  
    return res[n];  
}
```

方法II：滚动数组



<https://leetcode-cn.com/problems/climbing-stairs/solution/pa-lou-ti-by-leetcode-solution/>

```
var climbStairs = function(n){  
    let p=0,q=1;  
    for(let i=1;i<=n;i++){  
        r=p+q;  
        p=q;  
        q=r;  
    }  
    return r;  
}
```

2.青蛙跳台

剑指 Offer 10- II. 青蛙跳台阶问题

难度 简单 172 收藏 分享 切换为英文 接收动态 反馈

一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

答案需要取模 $1e9+7$ (1000000007)，如计算初始结果为： 1000000008 ，请返回 1 。

示例 1：

输入: $n = 2$

输出: 2

示例 2：

输入: $n = 7$

输出: 21

示例 3：

输入: $n = 0$

输出: 1

解题思路：

利用数组来保存已经递归过的值

```
var numWays = function(n) {
    let res = [1,1,2];
    if(n === 0 || n === 1){
        return res[1];
    }
    for(let i=3;i<=n;i++){
        res[i] = (res[i-2] + res[i-1]) % 1000000007;
    }
    return res[n];
};
```

3.括号生成

22. 括号生成

难度 中等

1556



文



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：

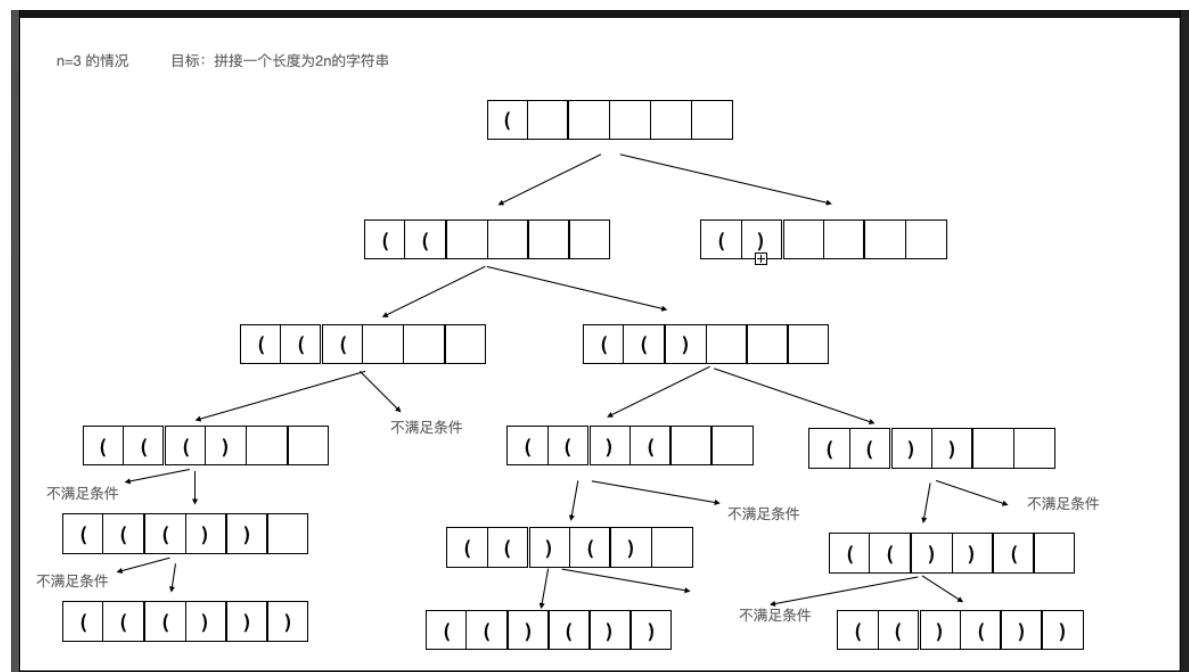
输入: $n = 1$

输出: `["()"]`

解题思路：

问题转换成往格子里放括号的问题；当 $n=1$ 时，共有两个格子可以放括号；当 $n=2$ 时，共有4个格子可以放括号；当 $n=3$ 时，共有6个格子可以放括号；

对于每一个格子来说， n 为括号的数量，左括号使用数量不能超过 n ，右括号使用数量不能超过左括号使用数量



为何使用递归方式--> 从6个格子-->5个格子-->4个格子-->3个格子

如何实现括号的合法性 ==> 剪枝(在生成括号之前，提前去掉一些不合法的括号)

```

var generateParenthesis = function(n) {
  const res=[];
  const dfs =(left,right,str)=>{
    if(str.length == 2*n){
      res.push(str);
      return;
    }
    if(left<n){
      dfs(left+1,right,str+"(");
    }
    if(right<left){
      dfs(left,right+1,str+")");
    }
  }
  dfs(0,0,"");
  return res;
};

```

如果只生成括号，不考虑合法性

```

var generateParenthesis = function(n) {
  let str = [];
  const dfs = (level,max,s) => {
    // terminator
    if(level >= max){
      str.push(s);
      return;
    }
    //process
    let s1 = s + "(";
    let s2 = s + ")";

    //drill down
    dfs(level+1,max,s1);
    dfs(level+1,max,s2);

    //reverse state
  }
  dfs(0,2*n,"");
  return str;
};

```

4.验证二叉搜索树

98. 验证二叉搜索树

难度 中等

920



给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

- 节点的左子树只包含**小于**当前节点的数。
- 节点的右子树只包含**大于**当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1：

输入：

```
2
 / \
1   3
```

输出： true

对于二叉搜索树的每一个节点来说，左子树必须小于右子树，且二叉搜索树的中序遍历是升序遍历

空树是二叉搜索树

解题思路：

根据二叉搜索树的中序遍历是升序遍历这个特点，首先对其进行中序遍历，将遍历结果保存到一个数组中

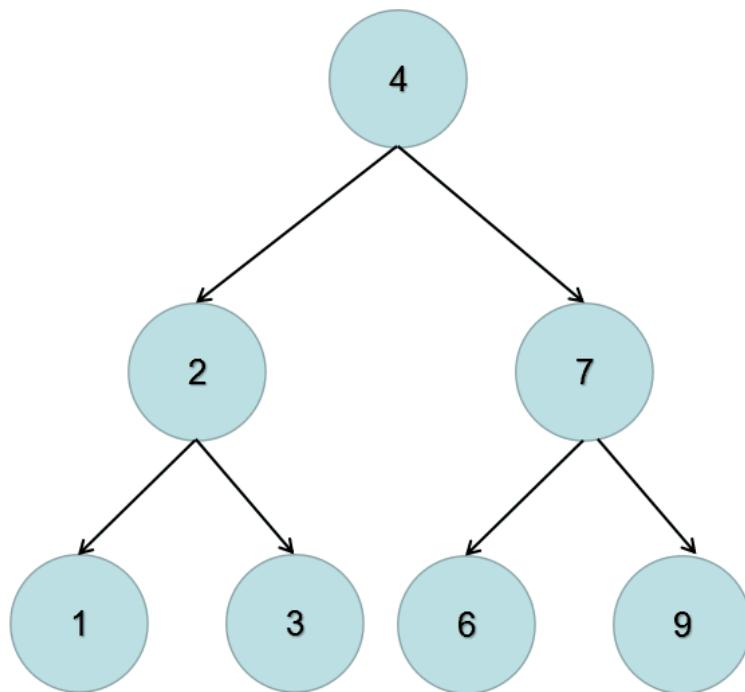
再对数组进行判断，是否是严格升序

```
var isValidBST = function(root) {
    // 空树是二叉搜索树
    if(!root) return true;

    let res = [];
    const inorder = (root) =>{
        if(!root) return;
        inorder(root.left);
        res.push(root.val);
        inorder(root.right);
    }
    inorder(root);

    for(let i=0;i<res.length;i++){
        if(res[i] <= res[i-1]) return false;
    }
    return true;
};
```

5. 翻转二叉树



```
var invertTree = function(root) {
  if(!root) return null;
  const left = invertTree(root.left);
  const right = invertTree(root.right);
  root.left = right;
  root.right = left;
  return root;
};
```

6. 二叉树的最大深度

104. 二叉树的最大深度

难度 简单 952 收藏 分享 切换为英文 接收动态 反馈

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明：叶子节点是指没有子节点的节点。

示例：

给定二叉树 [3, 9, 20, null, null, 15, 7]，

```
      3
     / \
    9  20
   /   \
  15   7
```

返回它的最大深度 3。

解题思路：

方法I：递归法

分别比较左右子树的深度，谁的值大就返回谁

```
var maxDepth = function(root) {  
    return root === null ? 0 : Math.max(maxDepth(root.left),maxDepth(root.right))+1;  
};
```

方法II：迭代法

遍历这个二叉树一共有多少层，返回的层数就是二叉树的最大层数

7. 二叉树的最小深度

111. 二叉树的最小深度

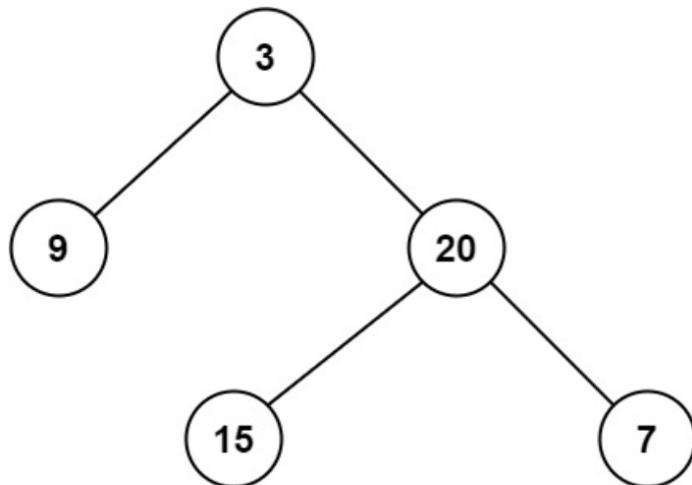
难度 简单 566 收藏 分享 切换为英文 接收动态 反馈

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明：叶子节点是指没有子节点的节点。

示例 1：



输入：root = [3,9,20,null,null,15,7]

输出：2

方法I：递归

与最大深度思想相同，但需要注意的是，需要增加一个判断，判断左右子树是否为空；当左右子树为空时，最小深度为另一边子树的深度+1

```

var minDepth = function(root) {
    if(!root) return 0;

    if(root.left === null) return minDepth(root.right)+1;
    if(root.right === null) return minDepth(root.left)+1;

    return Math.min(minDepth(root.left),minDepth(root.right))+1;
};

```

8. 二叉树的序列化与反序列化

解题思路：

<https://leetcode-cn.com/problems/serialize-and-deserialize-binary-tree/solution/shou-hui-tu-ji-e-gei-chu-dfshe-bfsliang-chong-jie-f/>

DFS：递归

```

var serialize = function(root) {
    if(!root) return "X";

    const left = serialize(root.left);
    const right = serialize(root.right);

    return root.val + "," + left + "," + right;
};

var deserialize = function(data) {
    const list = data.split(",");

    const buildTree = (list) =>{
        const rootVal = list.shift();
        if(rootVal === 'X'){
            return null;
        }
        const root = new TreeNode(rootVal);
        root.left = buildTree(list);
        root.right = buildTree(list);
        return root;
    }

    return buildTree(list);
};

```

9. 二叉树的最近公共祖先

解题思路：

对于根节点 root, p、q 的分布，有两种可能：

p、q 分居 root 的左右子树，则 LCA 为 root

p、q 存在于 root 的同一侧子树中，就变成规模小一点的相同问题

求解：

从左右子树分别进行递归，即查找左右子树上是否有p节点或者q节点

左右子树均无p节点或q节点

左子树找到，右子树没有找到，返回左子树的查找结果

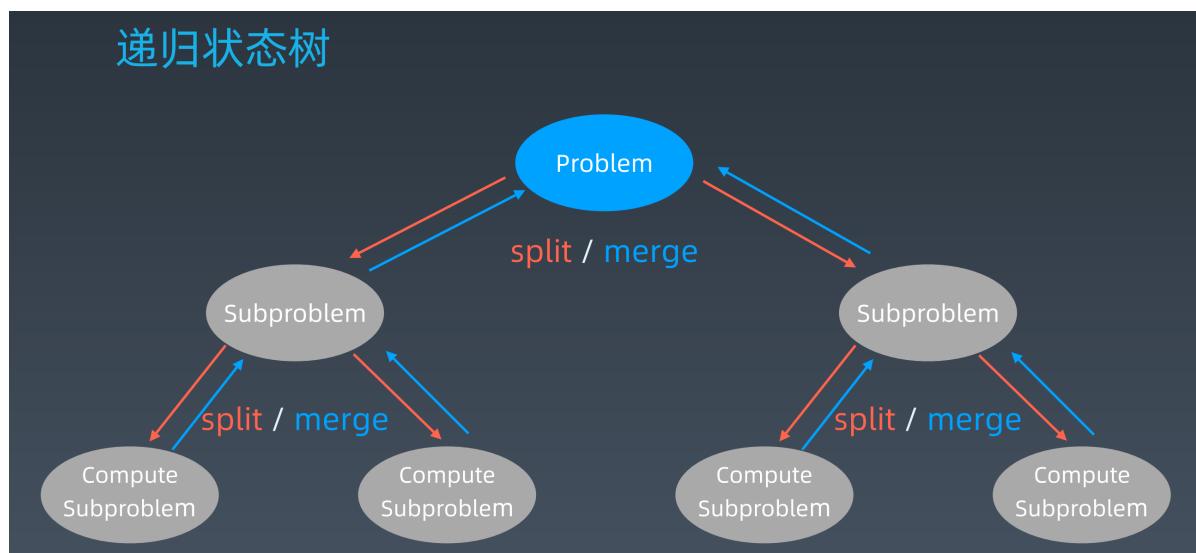
右子树找到，左子树没有找到，返回右子树的查找结果

左、右子树均能找到

说明此时的p节点和q节点在当前root节点两侧，返回root节点

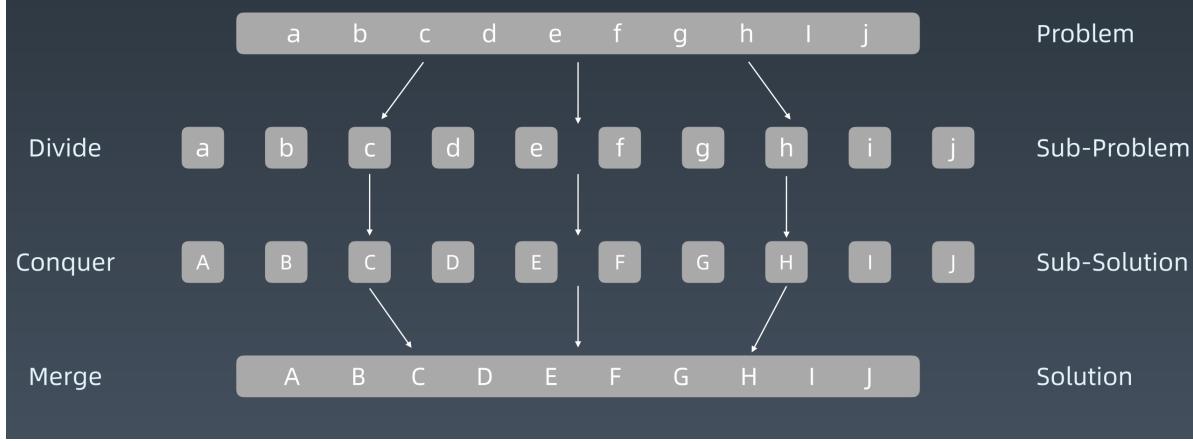
```
var lowestCommonAncestor = function(root, p, q) {  
    // 如果p,q是root，则他们的最近公共祖先一定是root，直接返回root  
    if(root === null || root === p || root === q){  
        return root;  
    }  
    const left = lowestCommonAncestor(root.left,p,q);  
    const right = lowestCommonAncestor(root.right,p,q);  
    if(left && right){  
        return root;  
    }  
    if(left === null){  
        return right;  
    }  
    return left;  
};
```

分治（一种特殊的递归方式）



本质-->找重复子问题

分治 Divide & Conquer



a-j 拆分成各自的字母，分别进行大小写转化再放到一起

递归代码模板

```
var recursion=function(level,param1,param2,...){  
    //terminator  
    if(level>max_level){  
        process_result;  
        return;  
    }  
  
    //process  
    process(level,data...)  
  
    //drill down  
    recursion(level+1,p1,...)  
  
    //set up  
    return function(0,p1,p2,...)  
}
```

分治代码模板

```
const divide_conquer = (problem, params) => {  
  
    // recursion terminator  
  
    if (problem == null) {  
  
        process_result  
  
        return  
  
    }  
  
    // process current problem  
  
    subproblems = split_problem(problem, data)  
  
    subresult1 = divide_conquer(subproblem[0], p1)  
  
    subresult2 = divide_conquer(subproblem[1], p1)
```

```
subresult3 = divide_conquer(subproblem[2], p1)

...
// merge

result = process_result(subresult1, subresult2, subresult3)

// revert the current level status

}
```

回溯backtracking

回溯法采用试错的思想，它尝试分步的去解决一个问题。在分步解决问题的过程 中，当它通过尝试发现现有的分步答案不能得到有效的正确的解答的时候，它将 取消上一步甚至是上几步的计算，再通过其它的可能的分步解答再次尝试寻找问 题的答案。

回溯法通常用最简单的递归方法来实现，在反复重复上述的步骤后可能出现两种情况：

找到一个可能存在的正确的答案；

在尝试了所有可能的分步方法后宣告该问题没有答案。

在最坏的情况下，回溯法会导致一次复杂度为指数时间的计算。

实战题目

1.pow(x,n)

50. Pow(x, n)

难度 中等

580



文



实现 $\text{pow}(x, n)$ ，即计算 x 的 n 次幂函数（即， x^n ）。

示例 1：

输入: $x = 2.00000, n = 10$

输出: 1024.00000

示例 2：

输入: $x = 2.10000, n = 3$

输出: 9.26100

示例 3：

输入: $x = 2.00000, n = -2$

输出: 0.25000

解释: $2^{-2} = 1/2^2 = 1/4 = 0.25$

方法I：暴力法-->时间复杂度O(n)

超出时间限制

```
var myPow = function(x, n) {
    if(n<0) {
        x= 1/x;
        n = -n;
    }
    let res=1;
    for(let i=1;i<=n;i++){
        res*=x;
    }
    return res;
};
```

方法II：分治法-->时间复杂度O(log N)

解题思路：

$2^{10} \rightarrow 2^5 * 2^5$

如果n是奇数的话，那么在基础上多乘一个x

每次将n的次数减半，因此时间复杂度为O(logN)

```
var myPow = function(x, n) {
    if(n<0){
        return myPow(1/x,-n);
    }
    if(n === 0) return 1;
    if(n === 1) return x;
    if(n >= 2){
        let half = myPow(x,Math.floor(n/2));
        return n % 2 ===0 ? half * half : half * half * x;
    }
};
```

2.子集

78. 子集

难度 中等 996

给你一个整数数组 `nums`，数组中的元素 **互不相同**。返回该数组所有可能的子集（幂集）。

解集 **不能** 包含重复的子集。你可以按 **任意顺序** 返回解集。

示例 1:

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

示例 2:

```
输入: nums = [0]
输出: [[], [0]]
```

方法I：递归

参考括号生成的问题，在每一层可以选择添加数字or不添加数字

注意：

引用类型指向的是地址，因为list改变时，已经添加到ans里的list会同时改变
因此需要使用`.slice()` ==>`slice`方法会返回一个新数组

```
var subsets = function(nums) {
    let ans = [];
    let list = [];
    const dfs = (ans,nums,list,index)=>{
        // terminator
        if(index === nums.length){
            // 引用类型指向的是地址，因为list改变时，已经添加到ans里的list会同时改变
            // 因此需要使用.slice() ==>slice方法会返回一个新数组
            ans.push(list.slice());
        }
    }
};
```

```
        return;
    }

    //drill down
    // not pick the number at this index
    dfs(ans,nums,list,index+1);

    //pick the number at this index
    list.push(nums[index]);
    dfs(ans,nums,list,index+1);

    // reverse state
    list.pop();
}
dfs(ans,nums,list,0);
return ans;
};
```

3.多数元素

169. 多数元素

难度 简单 873

给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现次数 **大于** $\lfloor n/2 \rfloor$ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1：

输入: [3,2,3]

输出: 3

示例 2：

输入: [2,2,1,1,1,2,2]

输出: 2

方法I：排序法

多数元素大于 $n/2$, 排序超过数组长度一半的元素就是多数元素

```
var majorityElement = function(nums) {
    nums.sort((a,b) => a-b);
    let index = Math.floor(nums.length/2);
    return nums[index];
};
```

方法II：利用哈希表进行统计

哈希表存储各项出现次数，循环哈希表返回超过 $n/2$ 数

```
var majorityElement = function(nums) {
    let hash = {};
    for(let i=0;i<nums.length;i++){
        if(!hash[nums[i]]){
            hash[nums[i]] = 1;
        }else {
            hash[nums[i]]++;
        }
    }
    let keys = Object.keys(hash);
    let values = Object.values(hash);
    let max = Math.max(...values);
    return keys[values.indexOf(max)];
};
```

4.电话号码的字母组合

17. 电话号码的字母组合

难度 中等

1134



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。



示例 1：

输入: digits = "23"

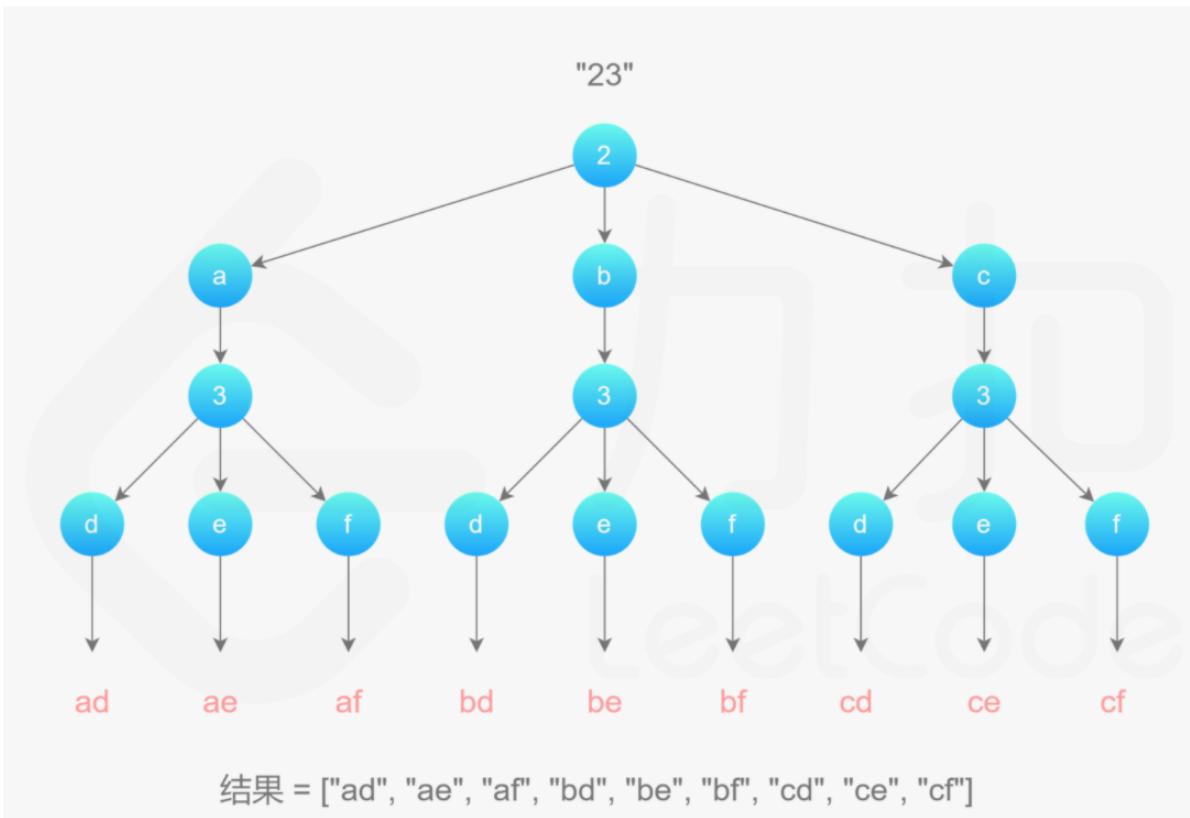
输出: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]

示例 2：

输入: digits = ""

输出: []

解题思路：



方法I：递归

```

var letterCombinations = function(digits) {
  if(digits === ""){
    return [];
  }

  let map = new Map();
  map.set("2",'abc')
  map.set("3",'def')
  map.set("4",'ghi')
  map.set("5",'jkl')
  map.set("6",'mno')
  map.set("7",'pqrs')
  map.set("8",'tuv')
  map.set("9",'wxyz')

  let res = [];
  const dfs = (curStr,index) => {
    // terminator
    if(index > digits.length -1){
      res.push(curStr);
      return;
    }

    // process
    let letters = map.get(digits[index]); // "abc", "def",...
    for(let letter of letters){
      // drill down
      dfs(curStr+letter,index+1);
    }
  }

  dfs("",0)
}

```

```
    return res;  
};
```

5. 全排列

46. 全排列

难度 中等 1517 收藏 分享 切换为英文 接收动态 反馈

给定一个不含重复数字的数组 `nums`，返回其 **所有可能的全排列**。你可以 **按任意顺序** 返回答案。

示例 1：

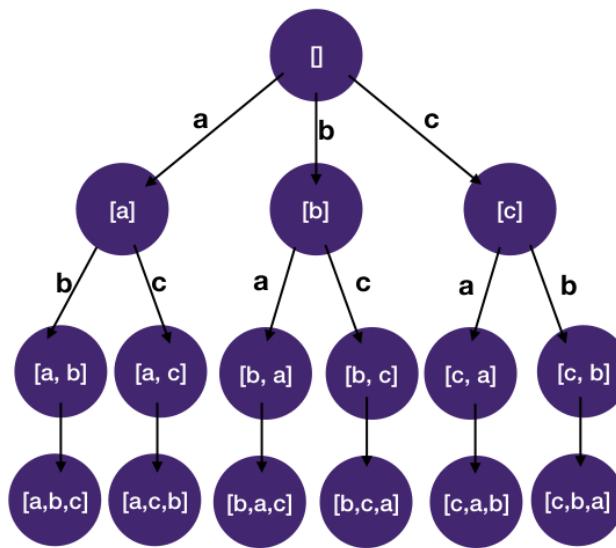
```
输入: nums = [1,2,3]  
输出: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

示例 2：

```
输入: nums = [0,1]  
输出: [[0,1],[1,0]]
```

示例 3：

```
输入: nums = [1]  
输出: [[1]]
```



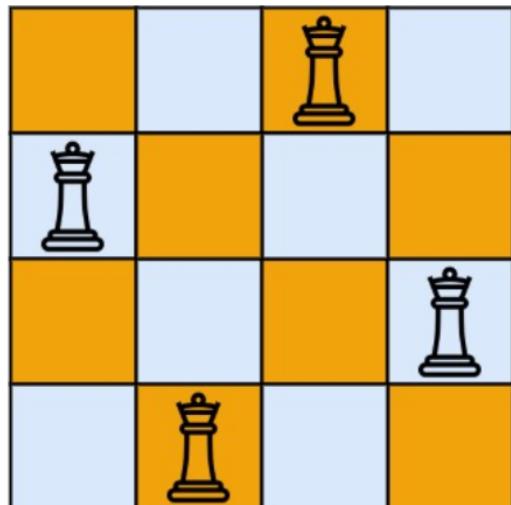
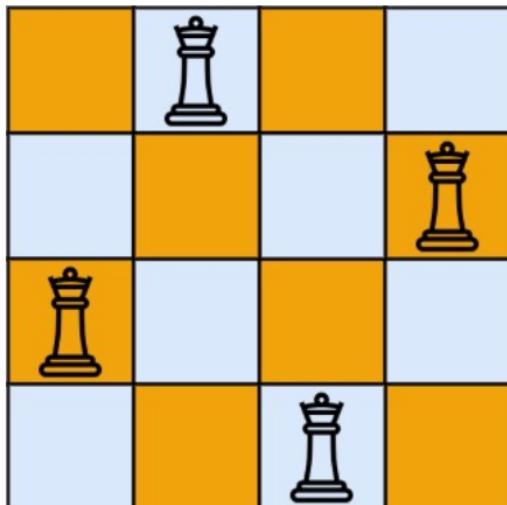
6.N皇后问题

n 皇后问题 研究的是如何将 n 个皇后放置在 $n \times n$ 的棋盘上，并且使皇后彼此之间不能相互攻击。

给你一个整数 n ，返回所有不同的 **n 皇后问题** 的解决方案。

每一种解法包含一个不同的 **n 皇后问题** 的棋子放置方案，该方案中 ‘Q’ 和 ‘.’ 分别代表了皇后和空位。

示例 1：



输入： $n = 4$

输出： `[[".Q..", "...Q", "Q...", "...Q"], ["..Q.", "Q...", "...Q", ".Q.."]]`

解释： 如上图所示，4 皇后问题存在两个不同的解法。

示例 2：

输入： $n = 1$

输出： `[["Q"]]`

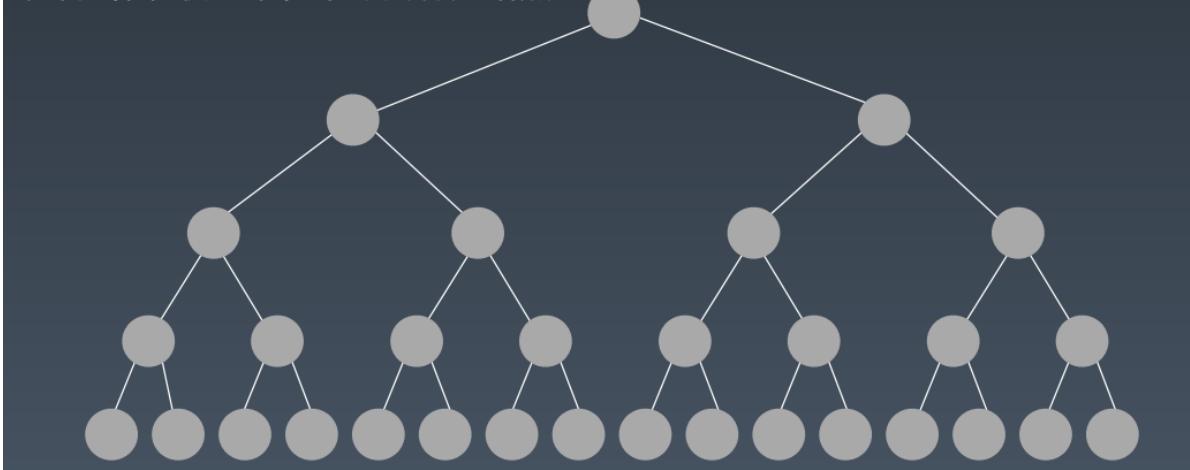
解题思路：

Week04

深度优先搜索、广度优先搜索的实现和特性

遍历搜索

在树（图/状态集）中寻找特定结点



实例代码

```
public class TreeNode{  
    public int val;  
    public TreeNode left,right;  
    public TreeNode(int val){  
        this.val = val;  
        this.left = null;  
        this.right = null;  
    }  
}
```

每个节点都要访问一次 ----> 避免浪费资源

每个节点仅仅要访问一次

对于节点的访问顺序不限

- 深度优先: depth first search
- 广度优先: breadth first search

深度优先搜索DFS(Deep First Search)-代码示例

Python

```
def dfs(node):  
    if node in visited:  
        #already visited  
        return  
    visited.add(node) #把node加到已访问的结点中去  
  
    #process current node  
    #...#logic here  
    dfs(node.left)  
    dfs(node.right)
```

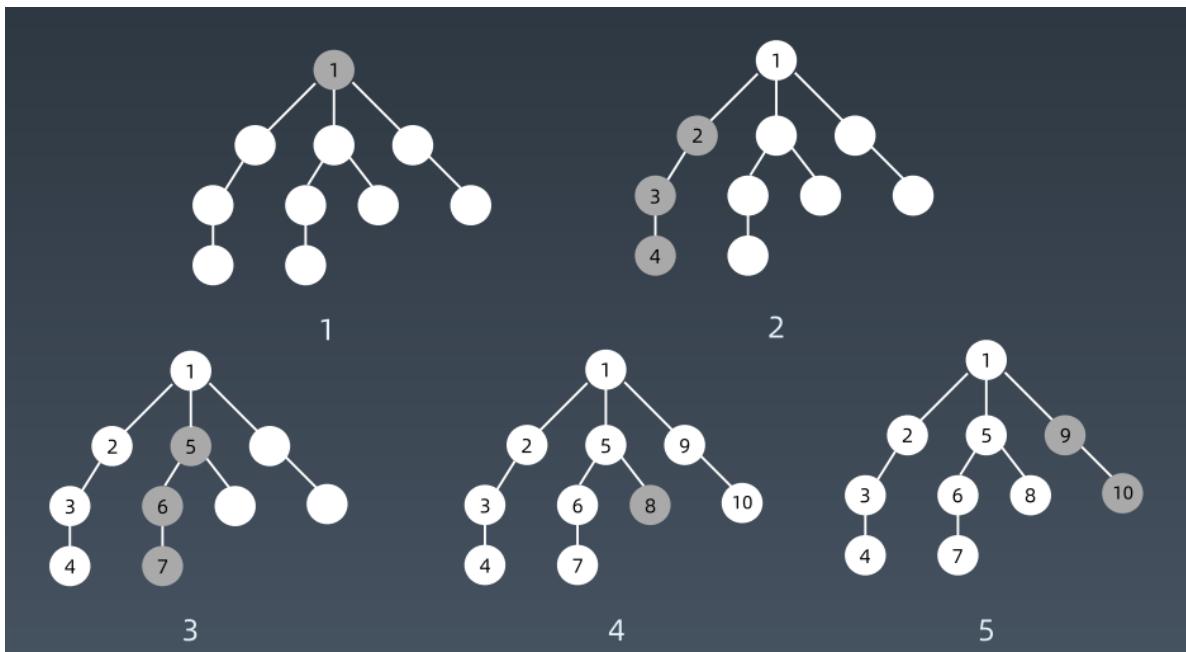
JavaScript

```

const visited = new Set()
const dfs = node => {
  if (visited.has(node))
    return
  visited.add(node)
  dfs(node.left)
  dfs(node.right)
}

```

多叉树的遍历顺序



DFS代码-递归写法---->多叉树

```

visited = set()
def dfs(node, visited):
  if node in visited: # terminator
    # already visited
    return visited.add(node)
  # process current node here.
  ...
  for next_node in node.children():
    if not next_node in visited:
      dfs(next_node, visited)

```

DFS代码-非递归写法

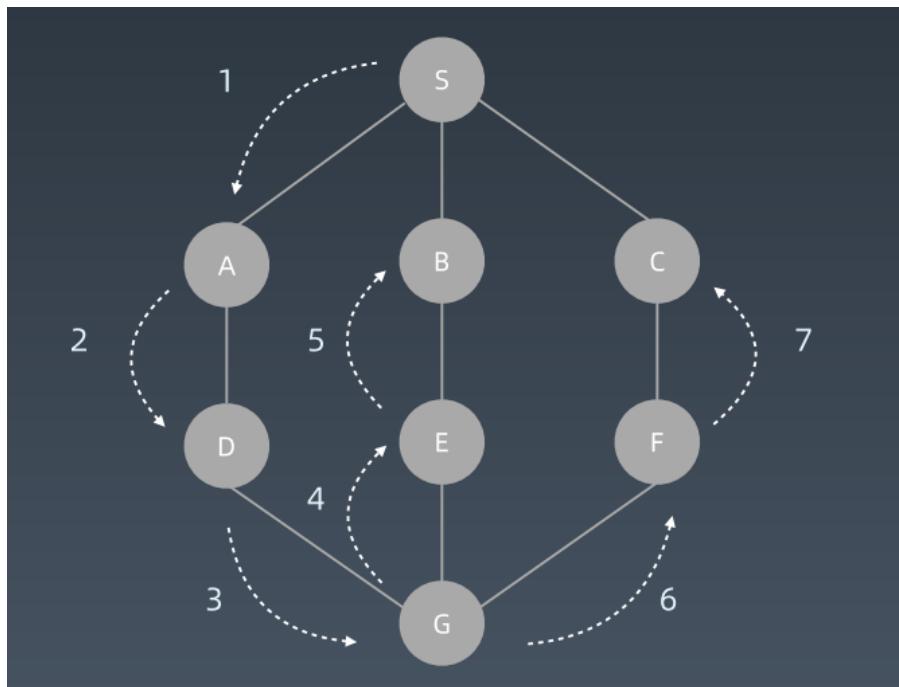
```

def DFS(self, tree):
    if tree.node is None:
        return []
    visited, stack = [], [tree.root]
    while stack:
        node = stack.pop()
        visited.add(node)

        process(node)
        nodes = generate_related_nodes(node)
        stack.push(nodes)
        # other processing work
    ...

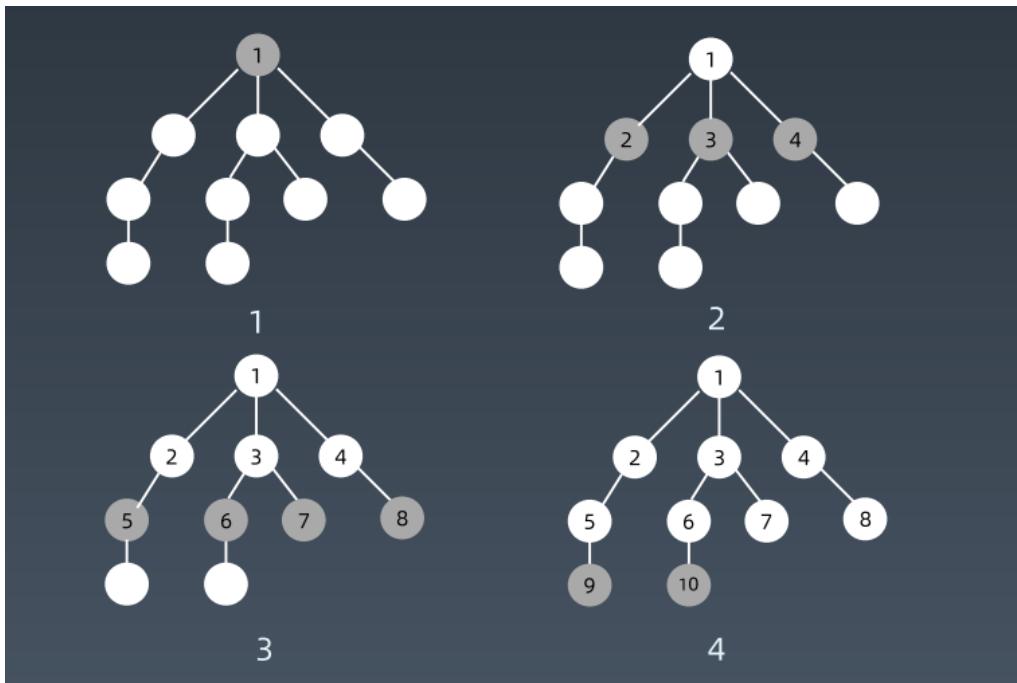
```

图的遍历顺序

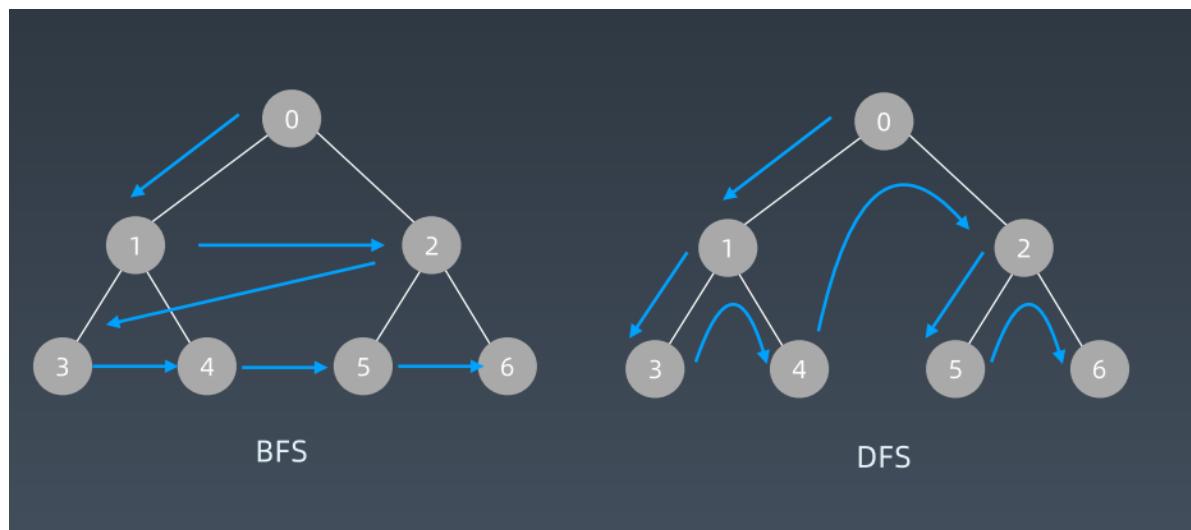


广度优先搜索-Breadth-First-Search

遍历顺序



DFS&BFS区别



代码模板-JavaScript

```
const bfs = (root) => {
  let result = [], queue = [root]
  while (queue.length > 0) {
    let level = [], n = queue.length
    for (let i = 0; i < n; i++) {
      let node = queue.pop()
      level.push(node.val)
      if (node.left) queue.unshift(node.left)
      if (node.right) queue.unshift(node.right)
    }
    result.push(level)
  }
  return result
};
```

实战题目

1.二叉树的层序遍历

102. 二叉树的层序遍历

难度 中等

784

收藏

分享

切换为英文

接收动态

反馈

给你一个二叉树，请你返回其按 **层序遍历** 得到的节点值。 (即逐层地，从左到右访问所有节点)。

示例：

二叉树： [3, 9, 20, null, null, 15, 7]，

```
3
 / \
9  20
 /  \
15   7
```

返回其层序遍历结果：

```
[
  [3],
  [9,20],
  [15,7]
]
```

解题思路：

方法I：BFS

```
var levelOrder = function(root) {
  if (!root) {
    return [];
  }
  const stack = [];
  stack.push(root);
  const result = [];
  while (stack.length > 0) {
    const size = stack.length;
    const temp = [];
    for (let i = 0; i < size; i++) {
      const node = stack.shift();
      temp.push(node.val);
      if (node.left) {
        stack.push(node.left);
      }
      if (node.right) {
        stack.push(node.right);
      }
    }
    result.push(temp);
  }
  return result;
}
```

```
};
```

方法II：DFS --->记录下当前层

2.最小基因变化

一条基因序列由一个带有8个字符的字符串表示，其中每个字符都属于 "A"， "C"， "G"， "T" 中的任意一个。

假设我们要调查一个基因序列的变化。一次基因变化意味着这个基因序列中的一个字符发生了变化。

例如，基因序列由 "AACCGGTT" 变化至 "AACCGGTA" 即发生了一次基因变化。

与此同时，每一次基因变化的结果，都需要是一个合法的基因串，即该结果属于一个基因库。

现在给定3个参数 — start, end, bank，分别代表起始基因序列，目标基因序列及基因库，请找出能够使起始基因序列变化为目标基因序列所需的最少变化次数。如果无法实现目标变化，请返回 -1。

注意：

1. 起始基因序列默认是合法的，但是它并不一定会出现在基因库中。
2. 如果一个起始基因序列需要多次变化，那么它每一次变化之后的基因序列都必须是合法的。
3. 假定起始基因序列与目标基因序列是不一样的。

示例 1：

```
start: "AACCGGTT"
end:   "AACCGGTA"
bank:  ["AACCGGTA"]
```

返回值： 1

3.在每个树行中找最大值

515. 在每个树行中找最大值

难度 中等

125

收藏

分享

切换为英文

接收动态

反馈

您需要在二叉树的每一行中找到最大的值。

示例：

输入：

```
1
 / \
3   2
 / \   \
5   3   9
```

输出：[1, 3, 9]

4. 岛屿数量

200. 岛屿数量

难度 中等

994

收藏

分享

切换为英文

接收动态

反馈

给你一个由 '1' (陆地) 和 '0' (水) 组成的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

示例 1：

```
输入: grid = [
["1","1","1","1","0"],
["1","1","0","1","0"],
["1","1","0","0","0"],
["0","0","0","0","0"]
]
```

输出：1

示例 2：

```
输入: grid = [
["1","1","0","0","0"],
["1","1","0","0","0"],
["0","0","1","0","0"],
["0","0","0","1","1"]
]
```

输出：3

解题思路：

按照二维数组，当找到1的时候，将1前后左右上下的1标记为0，用count来计数岛屿数量

贪心算法 Greedy

贪心算法是一种在每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是全局最好或最优的算法

贪心算法与动态规划的不同在于它对每个子问题的解决方案都做出选择，不能回退。动态规划则会保存以前的运算结果，并根据以前的结果对当前进行选择，有回退功能。

贪心法可以解决一些最优化问题，如：求图中的最小生成树、求哈夫曼编码等。然而对于工程和生活中问题，贪心法一般不能得到我们所要求的答案

贪心算法 VS 动态规划

贪心：当下做局部最优判断

回溯：能够回退

动态规划：最优判断 + 回退

示例：

当硬币可选集合固定：Coins = [20, 10, 5, 1] 求最少可以几个硬币拼出总数。比如 total = 36



贪心算法反例

非整除关系的硬币，可选集合：Coins = [10, 9, 1]

求拼出总数为 18 最少需要几个硬币？

最优解法 ----> $9*2 = 18$

贪心算法：

$$18 - 10 = 8$$



$$8 - 1 = 7$$



$$7 - 1 = 6$$



.....

$$1 - 1 = 0$$



适用贪心算法的场景

贪心算法与动态规划的不同在于它对每个子问题的解决方案都做出选择，不能回退。动态规划则会保存以前的运算结果，并根据以前的结果对当前进行选择，有回退功能。

实战题目

1. 分发饼干

455. 分发饼干

难度 简单 306

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。

对每个孩子 i ，都有一个胃口值 $g[i]$ ，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j ，都有一个尺寸 $s[j]$ 。如果 $s[j] \geq g[i]$ ，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

示例 1：

输入： $g = [1, 2, 3]$, $s = [1, 1]$

输出： 1

解释：

你有三个孩子和两块小饼干，3个孩子的胃口值分别是：1, 2, 3。

虽然你有两块小饼干，由于他们的尺寸都是1，你只能让胃口值是1的孩子满足。

所以你应该输出1。

示例 2：

输入： $g = [1, 2]$, $s = [1, 2, 3]$

输出： 2

解释：

你有两个孩子和三块小饼干，2个孩子的胃口值分别是1, 2。

2. 买卖股票的最佳时机II

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1：

输入： [7,1,5,3,6,4]

输出： 7

解释： 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = $5-1 = 4$ 。

随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = $6-3 = 3$ 。

示例 2：

输入： [1,2,3,4,5]

输出： 4

解释： 在第 1 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = $5-1 = 4$ 。

注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。

因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

解题思路：

只要前一天的利润大于当前天的利润，进行锁定价格差的操作

```
var maxProfit = function(prices){
    let profit=0;
    for(let i=1;i<prices.length;i++){
        let prev = prices[i-1];
        let curr = prices[i];
        if(prev<curr){
            profit+=curr - prev;
        }
    }
    return profit;
}
```

3.跳跃游戏

给定一个非负整数数组 `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。

示例 1:

输入: `nums = [2,3,1,1,4]`

输出: `true`

解释: 可以先跳 1 步，从下标 0 到达下标 1，然后再从下标 1 跳 3 步到达最后一个下标。

示例 2:

输入: `nums = [3,2,1,0,4]`

输出: `false`

解释: 无论怎样，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0，所以永远不可能到达最后一个下标。

解题思路:

方法I: 递归

递归终止条件: 跳到最后一个位置

按层数递归看最大能跳到哪一层

方法II: 暴力解法

两重循环 ---> 每一个数最大的跳动位置写成true

方法III: 贪心算法

```
var jump = function(nums) {
    if (nums.length === 1) return 0; // 第90个测试用例[0]，不用跳，返回0即可。
    // start到end这个区间，用以表示我们第一跳都能在哪些点落脚。
    let start = 0;
    let end = nums[0];
    let max = Math.max(start, end);
    let step = 1;
    while (1) {
        // nums.length - 1 是我们的目的地，假如我们最远的落点已经能到达目的地，退出返回跳跃次数即可。
        if (end >= nums.length - 1) return step;
        // 看看我们每一跳最远的地方分别是哪里；
        for (let i = start; i <= end; i++) {
            max = Math.max(i+nums[i], max);
        }
        step++;
        // 假如跳了半天，最远的距离不过是先前一跳的最远距离的话，那还是别蹦跶了，洗洗睡吧。
        if (max === end) break;
        [start, end] = [end+1, max];
    }
}
```

```
    return -1;  
};
```

二分查找

二分查找的前提

目标函数单调性 (单调递增或者递减)

存在上下界 (bounded)

能够通过索引访问 (index accessible)

代码模板

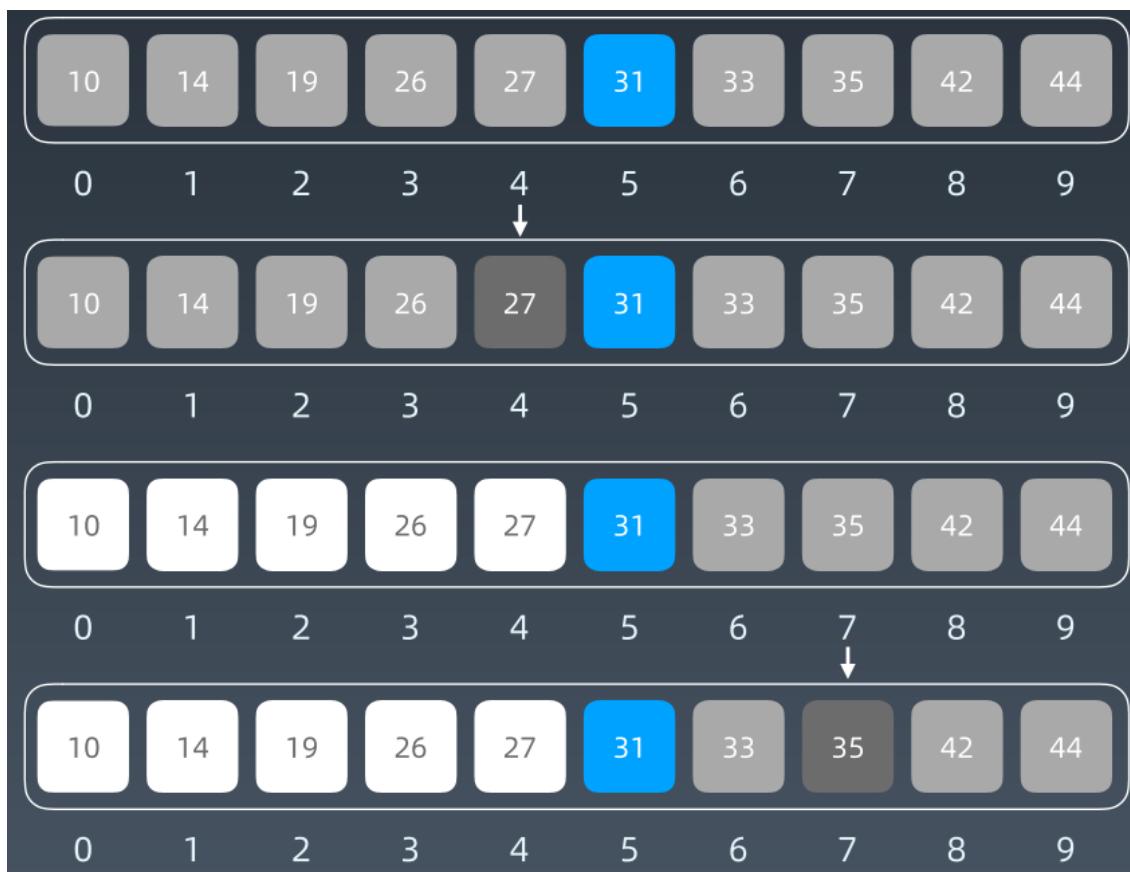
```
let left=0,right=len(array)-1;  
  
while(left<=right){  
  
    let mid = (left+right) >> 1; //位运算 比=(left + right)/2快  
    if(array[mid]==target){return array[mid]}  
    else if(array[mid] < target) left = mid+1;  
    else right = mid - 1;  
}
```

示例

在递增数组里

[10, 14, 19, 26, 27, 31, 33, 35, 42, 44]

查找: 31



解题步骤：

先找到中间值--->27

$31 > 27$ ---->从后半部分开始查找

找到后半部分中间值35

$31 < 35$ ---> 在31-35中间查找

找到中间值33

$31 < 33$ ---->找到31

实战题目

1.x的平方根

69. x 的平方根

难度 简单 收藏 分享 切换为英文 接收动态 反馈

实现 `int sqrt(int x)` 函数。

计算并返回 x 的平方根，其中 x 是非负整数。

由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

示例 1:

输入： 4

输出： 2

示例 2:

输入： 8

输出： 2

说明： 8 的平方根是 2.82842...，

由于返回类型是整数，小数部分将被舍去。

解题思路

方法I：二分查找

$y=x^2$ ($x>0$) ---->单调递增 ---->可以用二分查找

`return right` ----> 舍去小数部分，尝试left, right后选择right

```

var mySqrt = function(x) {
    if(x==0 || x==1){
        return x;
    }
    let left=1,right = x;
    while(left<=right){
        let mid = parseInt((left+right)/2);
        if(mid * mid==x) return mid;
        else if(mid * mid <x) left=mid+1;//mid小于说明要相加
        else right = mid -1;
    }
    return right;//用debug查找是left还是right
};

```

方法II：牛顿迭代法

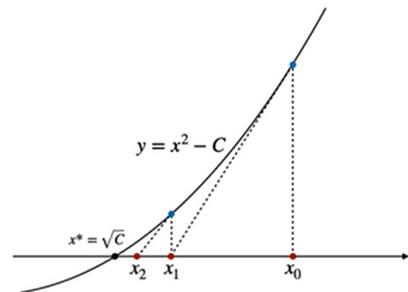
用直线代替曲线，泰勒一阶展开式

· 方法三：牛顿法

我们选择 $x_0 = C$ 作为初始值。

在每一步迭代中，我们通过当前的交点 x_i ，找到函数图像上的点 $(x_i, x_i^2 - C)$ ，作一条斜率为 $f'(x_i) = 2x_i$ 的直线，直线的方程为：

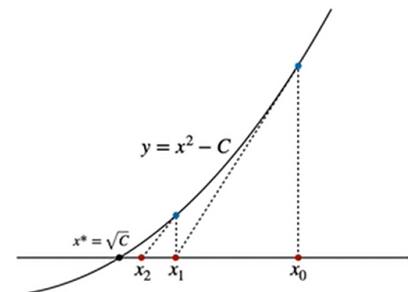
$$\begin{aligned}y_l &= 2x_i(x - x_i) + x_i^2 - C \\&= 2x_i x - (x_i^2 + C)\end{aligned}$$



· 方法三：牛顿法

与横轴的交点为方程 $2x_i x - (x_i^2 + C) = 0$ 的解，即为新的迭代结果 x_{i+1} ：

$$x_{i+1} = \frac{1}{2} \left(x_i + \frac{C}{x_i} \right)$$



在进行 k 次迭代后， x_k 的值与真实的零点 \sqrt{C} 足够接近，即可作为答案。

```

var mySqrt = function (x) {
    let r = x;
    while(r*r>x){
        r = ((r+x/r)/2) |0;
    }
    return r;
}

```

2. 山脉数组的峰顶索引

852. 山脉数组的峰顶索引

难度 简单

174



A



符合下列属性的数组 `arr` 称为 **山脉数组**：

- `arr.length >= 3`
- 存在 `i` ($0 < i < arr.length - 1$) 使得：
 - $arr[0] < arr[1] < \dots < arr[i-1] < arr[i]$
 - $arr[i] > arr[i+1] > \dots > arr[arr.length - 1]$

给你由整数组成的山脉数组 `arr`，返回任何满足 $arr[0] < arr[1] < \dots < arr[i - 1] < arr[i] > arr[i + 1] > \dots > arr[arr.length - 1]$ 的下标 `i`。

示例 1：

输入: `arr = [0,1,0]`

输出: 1

示例 2：

输入: `arr = [0,2,1,0]`

输出: 1

示例 3：

输入: `arr = [0,10,5,2]`

输出: 1

示例 4：

输入: `arr = [3,4,5,1]`

输出: 2

示例 5：

输入: `arr = [24,69,100,99,79,78,67,36,26,19]`

输出: 2

方法I：暴力解法遍历数组---->时间复杂度O(n)

```

var peakIndexInMountainArray = function(arr) {
    let max = -Infinity;
    let index = 0;
    for(let i=0;i<arr.length;i++){
        if(arr[i]>max){
            max = arr[i];
            index = i;
        }
    }
    return index;
};

```

方法II：二分查找---->时间复杂度O(logN)

```

var peakIndexInMountainArray = function(arr) {
    let left = 0,right = arr.length-1,ans;
    while(left <= right){
        //let mid = Math.floor((left+right)/2);
        let mid = (left+right) >> 1;
        if(arr[mid]>arr[mid+1]){
            ans = mid;
            right = mid -1;
        }else{
            left = mid +1;
        }
    }
    return ans;
};

```

3.有效的完全平方数

367. 有效的完全平方数

难度 简单 194 收藏 分享 切换为英文 接收动态 反馈

给定一个正整数 num ，编写一个函数，如果 num 是一个完全平方数，则返回 True，否则返回 False。

说明：不要使用任何内置的库函数，如 `sqrt`。

示例 1：

输入： 16
输出： True

示例 2：

输入： 14
输出： False

解题思路：

判断是否可以使用二分查找: $f(x) = x^2$, 函数单调递增

从0->x处寻找 $mid * mid == x$ 的数

```
var isPerfectSquare = function(num) {  
    if(num == 0 || num == 1) return true;  
    let left = 0, right = num;  
    while(left <= right){  
        let mid = (left+right)>>1;  
        if(mid * mid == num) return true;  
        else if(mid * mid > num){  
            right = mid - 1;  
        } else{  
            left = mid + 1;  
        }  
    }  
    return false;  
};
```

4. 搜索旋转排序数组

33. 搜索旋转排序数组

难度 中等 1199 收藏 分享 切换为英文 接收动态 反馈

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` ($0 \leq k < \text{nums.length}$) 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`，则返回它的索引，否则返回 `-1`。

示例 1：

```
输入: nums = [4,5,6,7,0,1,2], target = 0  
输出: 4
```

示例 2：

```
输入: nums = [4,5,6,7,0,1,2], target = 3  
输出: -1
```

四步做题法：

1. 审题-->和面试官确认条件 (输入, 输出, 边界条件)

2. 想所有的解法 (时间复杂度)

3. 写代码

4. 测试样例

方法I：暴力解法

找到第一个无序的位置(二分法) $O(\log N)$ -->升序-->二分

```
var binarySearch = function (nums){  
    let left=0,right=nums.length-1;  
    while (left<=right){  
        let mid=Math.floor((left+right)/2);  
        if(nums[mid]>nums[mid - 1]&&nums[mid]>nums[mid+1]) return mid;  
        if(nums[left]<=nums[mid]){  
            left =mid+1;  
        }else{  
            right = mid-1;  
        }  
    }  
}
```

方法II：二分查找（条件：单调&边界&index）

```
var search = function(nums, target) {  
    let left=0,right=nums.length -1;  
    while(left<=right){  
        let mid =Math.floor((left+right)/2);  
        //一次性找到target  
        if(nums[mid]==target) return mid;  
        //判断左边单调还是右边单调  
        if(nums[left]<=nums[mid]){ //左边单调&&右边不单调  
            //继续用二分法查找target  
            if(nums[left]<=target && target<=nums[mid]){  
                right = mid-1;  
            }else{  
                left= mid+1;  
            }  
        }  
        else { //左边不单调&右边单调  
            if(nums[mid]<=target && target<=nums[right]){  
                left = mid+1;  
            }else{  
                right =mid-1;  
            }  
        }  
    }  
    return -1;  
};
```

5. 搜索旋转排序数组II

81. 搜索旋转排序数组 II

难度 中等

443

收藏

分享

切换为英文

接收动态

反馈

已知存在一个按非降序排列的整数数组 `nums`，数组中的值不必互不相同。

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

给你 **旋转后** 的数组 `nums` 和一个整数 `target`，请你编写一个函数来判断给定的目标值是否存在于数组中。如果 `nums` 中存在这个目标值 `target`，则返回 `true`，否则返回 `false`。

示例 1：

```
输入: nums = [2,5,6,0,0,1,2], target = 0
输出: true
```

解题思路：

与搜索旋转排序数组的不同：

(1) 测试样例中加入了重复项：

```
nums = [1,0,1,1,1],target = 0
nums=[3,1,2,3,3,3],target = 2;
```

此时应该可能无法判断哪边是单调区间，因此需要多一步判断

```
if(nums[left]==nums[mid] && nums[mid]==nums[right]){
    left++;
    right--;
}
```

对于这种情况，我们只能将当前二分区间的左边界加一，右边界减一，然后在新区间上继续二分查找，让 `left` 和 `right` 逐渐向数组不重复的部分靠拢

(2) 注意左右的边界：

```
else if(nums[left]<=nums[mid])
    if(nums[left]<=target && target<nums[mid])
else{
    if(nums[mid]<target && target<=nums[right])
}
```

题解：

```
var search = function(nums, target) {
    let left = 0, right = nums.length - 1;
    while(left <= right){
        let mid = left + right >> 1;
        if(nums[mid] == target) return true;
        //数组中的重复值使我们无法判断哪边是单调区间时
        //使left++,right--
        if(nums[left] == nums[mid] && nums[mid] == nums[right]){
            left++;
            right--;
        } else if(nums[left] <= target && target < nums[mid]) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    return false;
}
```

```

    left++;
    right--;
}
else if(nums[left] <= nums[mid]){
    //target在左边的单调区间中
    if(nums[left] <= target && target < nums[mid]){
        right = mid - 1;
    }else{
        left = mid + 1;
    }
}
else{
    //mid落在右边的单调区间中
    if(nums[mid] < target && target <= nums[right]){
        left = mid + 1;
    }else{
        right = mid - 1;
    }
}
return false;
};

```

6. 搜索二维矩阵

74. 搜索二维矩阵

难度 中等 444 收藏 分享 切换为英文 接收动态 反馈

编写一个高效的算法来判断 $m \times n$ 矩阵中，是否存在一个目标值。该矩阵具有如下特性：

- 每行中的整数从左到右按升序排列。
- 每行的第一个整数大于前一行的最后一个整数。

示例 1：

1	3	5	7
10	11	16	20
23	30	34	60

输入： matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
输出： true

解题思路：

矩阵 = 二维数组，矩阵中的每一个数组都是升序排列，因此想到二分查找

方法I：暴力解法 两重循环遍历matrix--->时间复杂度O(n^2)

```
var searchMatrix = function(matrix, target) {
    for(let i=0;i<matrix.length;i++){
        for(let j=0;j<matrix[i].length;j++){
            if(matrix[i][j] == target) return true;
        }
    }
    return false;
};
```

方法II：一个二分查找---->时间复杂度O(nlogn)

```
var searchMatrix = function(matrix, target) {
    for(let i=0;i<matrix.length;i++){
        for(let j=0;j<matrix[i].length;j++){
            let left = 0,right = matrix[i].length - 1;
            while (left <= right){
                let mid = left+right >> 1;
                if(matrix[i][mid] == target) return true;
                else if(matrix[i][mid] > target){
                    right = mid - 1;
                }else{
                    left = mid + 1;
                }
            }
        }
    }
    return false;
};
```

方法III：矩阵降维成一维数组

通过ES6的新增方法flat()将矩阵降维成一维数组，因为矩阵从小到大递增，因此可以使用二分查找的方法。此方法的时间复杂度为O(logN)

```
var searchMatrix = function(matrix, target) {
    newMatrix = matrix.flat();
    let left = 0,right = newMatrix.length-1;
    while(left <= right){
        let mid = left+right >> 1;
        if(newMatrix[mid] == target) return true;
        else if(newMatrix[mid] > target){
            right = mid - 1;
        }else{
            left = mid + 1;
        }
    }
    return false;
};
```

153. 寻找旋转排序数组中的最小值

难度 中等 496 收藏 分享 切换为英文 接收动态 反馈

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 $\text{nums} = [0, 1, 2, 4, 5, 6, 7]$ 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 $[a[0], a[1], a[2], \dots, a[n-1]]$ 旋转一次 的结果为数组 $[a[n-1], a[0], a[1], a[2], \dots, a[n-2]]$ 。

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 。

示例 1：

```
输入: nums = [3,4,5,1,2]
输出: 1
解释: 原数组为 [1,2,3,4,5] , 旋转 3 次得到输入数组。
```

示例 2：

```
输入: nums = [4,5,6,7,0,1,2]
输出: 0
解释: 原数组为 [0,1,2,4,5,6,7] , 旋转 4 次得到输入数组。
```

示例 3：

```
输入: nums = [11,13,15,17]
输出: 11
解释: 原数组为 [11,13,15,17] , 旋转 4 次得到输入数组。
```

解题思路：

nums 是旋转数组，旋转前是升序数组，因此最小值一定存在于中间

```
var findMin = function(nums) {
    if(nums.length == 1) return nums[0];
    let left = 0,right = nums.length-1;
    //数组并没有被旋转，最右边的数大于最左边的数，返回数组中的第一个数
    if(nums[right]>nums[left]) return nums[0];
    //二分查找
    while(left < right){
        let mid = left+right >>1;
        //找到最小值的第一个种情况
        if(nums[mid-1]>nums[mid]) return nums[mid];
        //找到最小值的第二种情况
        if(nums[mid]>nums[mid+1]) return nums[mid+1];
        //继续二分查找
        if(nums[mid]>nums[left]){
            left = mid+1;
        }else{
    }
```

```
    right = mid -1;  
}  
}  
};
```

期中直播答疑

1. 关于链表和数组区别：

✓ 数组必须事先定义固定的长度（元素个数），不能适应数据动态增减的情况	✗ 链表动态地进行存储分配，能动态增减，且能方便地插入、删除数据项
✓ 数组利用下标定位，时间复杂度为O(1)；链表定位元素时间复杂度O(n)	✗ 两者数组插入或删除元素都是O(n)

2. 二叉树前中后序遍历的时间复杂度 ----> O(N)

题解：所有结点访问且只访问一次

3. 较难分析的代码：

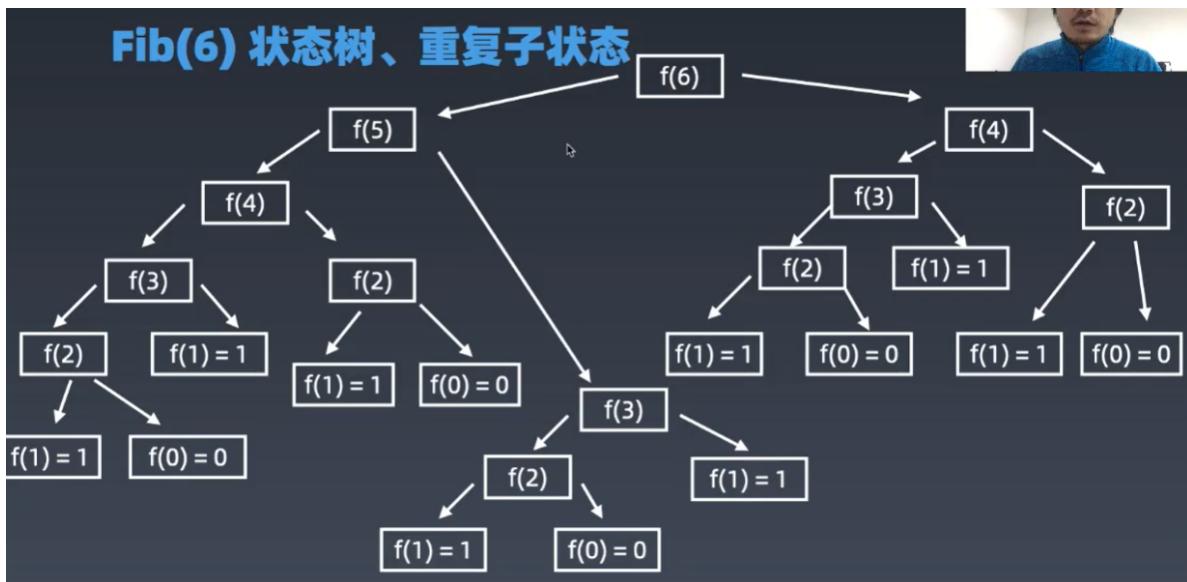
```
class Solution {  
public:  
    bool isMatch(string s, string p) {  
        if (p.empty()) return s.empty();  
  
        bool isFirstMatch = !s.empty() && (s[0] == p[0] || '.' == p[0]);  
  
        if (p.size() >= 2 && p[1] == '*') {  
            return (isFirstMatch && isMatch(s.substr(1), p)) ||  
                   isMatch(s, p.substr(2));  
        }  
  
        return isFirstMatch && isMatch(s.substr(1), p.substr(1));  
    }  
};
```

时间复杂度：

递归只调用一次就是n；调用2次就是 2^n ；

斐波那契求和 $O(2^n)$

Fib(6) 状态树、重复子状态



4.N是单词数，M是单词长度

```

def ladderLength(self, beginWord, endWord, wordList):
    wordList = set(wordList)
    queue = collections.deque([(beginWord, 1)])
    while queue:
        word, length = queue.popleft()
        if word == endWord:
            return length
        for i in range(len(word)):
            for c in 'abcdefghijklmnopqrstuvwxyz':
                next_word = word[:i] + c + word[i+1:]
                if next_word in wordList:
                    queue.append((next_word, length+1))
                    wordList.remove(next_word)
    return 0
  
```

wordlist = set() O(n)

O(N*M)

5.二叉排序树的特点

A . 左子树小于根；右子树大于根；且对左右子树同理可得。

B . 不能是空树

C . 左右子树为二叉排序树

D . 中序遍历是递增数列

空树也是二叉搜索树

6.

单选题 二叉树的后序遍历是 {1,3,2,6,5,7,4}，中序遍历是{1,2,3,4,5,6,7}，则错误的是？

中序遍历--->左子树1,2,3 右子树 5,6,7

106-从中序与后序遍历序列构造二叉树

合并两个有序链表

```
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if (l1 == null) return l2;
        if (l2 == null) return l1;
        if (l1.val < l2.val) {
            l1.next = mergeTwoLists(l1.next, l2);
            return l1;
        }
        else {
            l2.next = mergeTwoLists(l2.next, l1);
            return l2;
        }
    }
}
```

合并K个链表

颜色分类

四数之和

滑动窗口最大值 (队列)

7.n阶对称矩阵

将10阶对称矩阵压缩存储到一维数组A中，则数组A的长度最少为 ()
(4分)

A 100

B 40

C 55

D 80

3阶对称矩阵：

[1,2,4]

[2,3,5]

[4,5,6]

对角线左右两侧对称，因此只需要存储倒三角形状的数组

对于10阶数组来说：

第一行存1个/第二行存2个/第三行存3个/第四行存4个....第10行存10个

一共需要存储的数组为 $(1+2+3+\dots+n)=n*(1+n)/2$

动态规划

成就表

Array 数组	Search 查询
Linked List 链表 (动画)	Recursion 递归 (动画)
Stack 栈	DFS 深度优先搜索
Queue 队列	BFS 广度优先搜索
HashTable 哈希表 (动画)	Divide & Conquer 分治
Set、Map	Backtracking 回溯
Tree 二叉树	Greedy 贪心
BST 二叉搜索树 (动画)	Binary Search 二叉查找

推荐网站：visualgo.net ----> 常见的数据结构可视化网站

空树是二叉搜索树

递归

求n的阶乘 ---->每次只调用自己一次

$f(n - 1) + f(n - 2)$ ---->二叉树结构

坚持

1.五毒神掌

2.脑图+笔记总结

3.微信群里多交流

Week05 动态规划

递归的3种状态

1.递归终止条件

2.处理当前层

3.下探到下一层

4.恢复当前层状态 (if needed)

代码模板

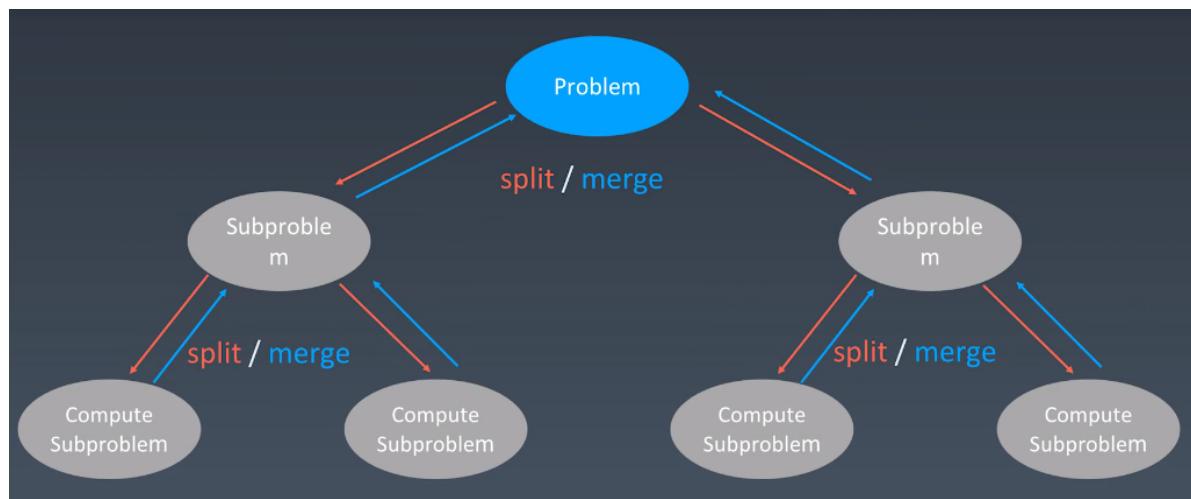
递归

```

const recursion = (level, params) =>{
    // recursion terminator
    if(level > MAX_LEVEL){
        process_result
        return
    }
    // process current level
    process(level, params)
    //drill down
    recursion(level+1, params)
    //clean current level status if needed
}

```

分治 ---> 递归的一种分支



分治

```

const divide_conquer =(problem, params) =>{
    // recursion terminator
    if(problem == null){

        process_result

        return
    }

    //process current problem

    subproblems = split_problem( problem, data)

    subresult1 = divide_conquer(subproblem[0], p1)

    subresult2 = divide_conquer(subproblem[1], p1)

    subresult3 = divide_conquer(subproblem[2], p1)

    ...

    //merge

    result = process_result(subresult1, subresult2, subresult3)

    //revert the current level status
}

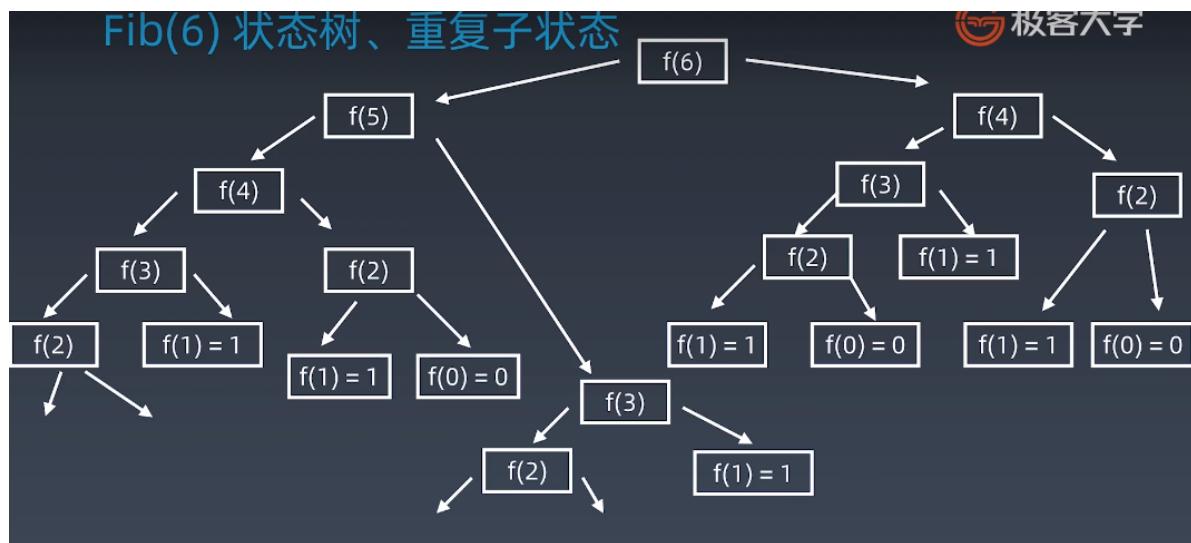
```

}

重要思维

- 1.人肉递归低效、很累
 - 2.找到最近最简单方法，将其拆解成可重复解决的问题
 - 3.数学归纳法（抵制人肉递归的诱惑）
- 本质：寻找重复性 ----> 计算机指令集
if-else/recursion/loop

掌握递归的状态 ----> 画递归的状态树



动态规划

1.wiki定义：

https://en.wikipedia.org/wiki/Dynamic_programming

2.“Simplifying a complicated problem by breaking it down into simpler sub-problems” (in a recursive manner)

3.Divide & Conquer + Optimal substructure

分治+最优子结构

关键点

动态规划和递归或者分治没有根本上的区别（关键看有无最优的子结构）

共性：找到重复子问题

差异性：最优子结构、中途可以淘汰次优解

实战题目

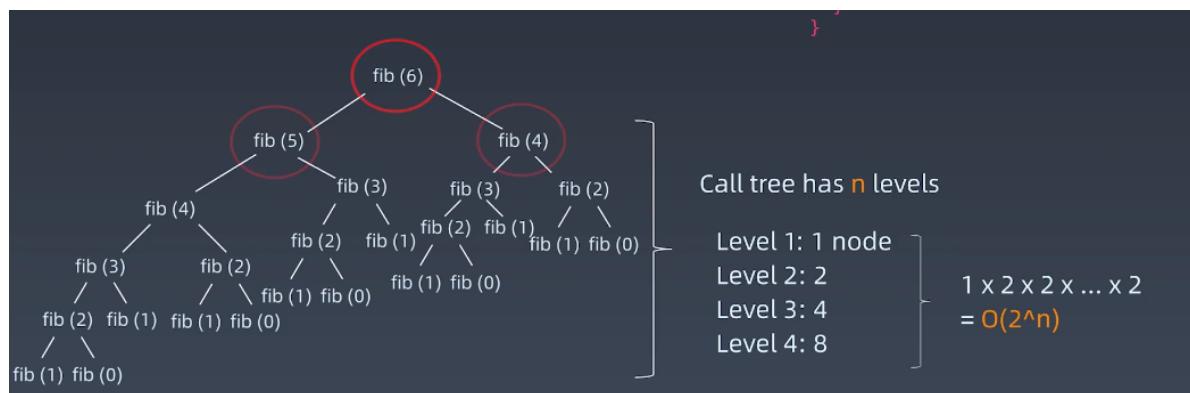
1.斐波那契数组

```
int fib(int n){  
    if(n<=0){  
        return 0;  
    }else if(n == 1){  
        return 1  
    }else{  
        return fib(n-1) + fib(n-2);  
    }  
}
```

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$

$$\text{fib}(0) = 0$$

$$\text{fib}(1) = 1$$



Java普通解法

```
int fib(int n){  
    if(n<=0){  
        return 0;  
    }else if(n == 1){  
        return 1;  
    }else{  
        return fib(n-1)+fib(n-2);  
    }  
}
```

Java简化写法

```
int fib(int n){  
    return n<=1?n:fib(n-1)+fib(n-2);  
}
```

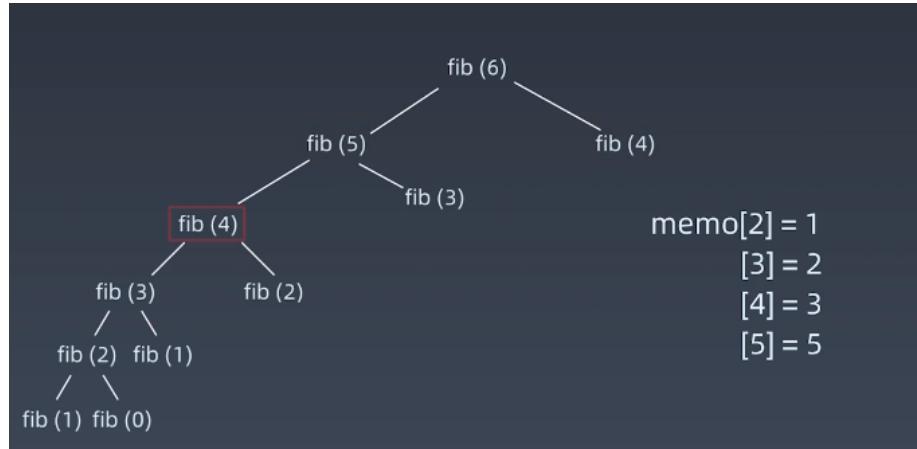
改变时间复杂度 --->增加缓存

将之前已经计算过的存到数组中，下一次再进行调用

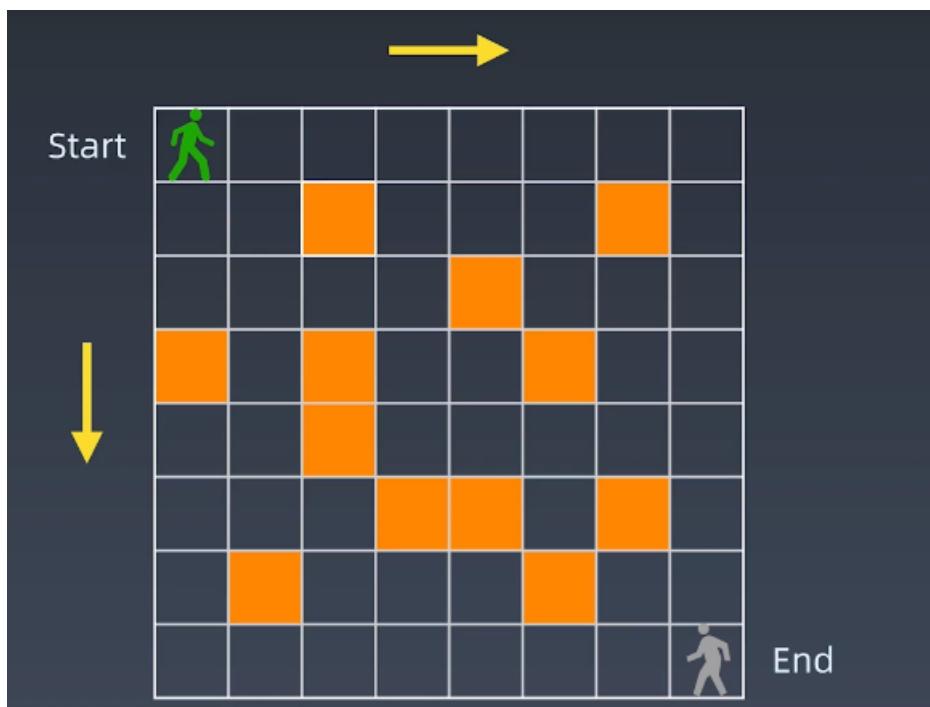
```

int fib(int n, int[] memo){
    if(n<=0){
        return 0; //return n
    }else if(n == 1){
        return 1; //return n
    }else if(memo[n] == 0){
        memo[n] = fib(n-1) + fib(n-2);
    }
    return memo[n];
}

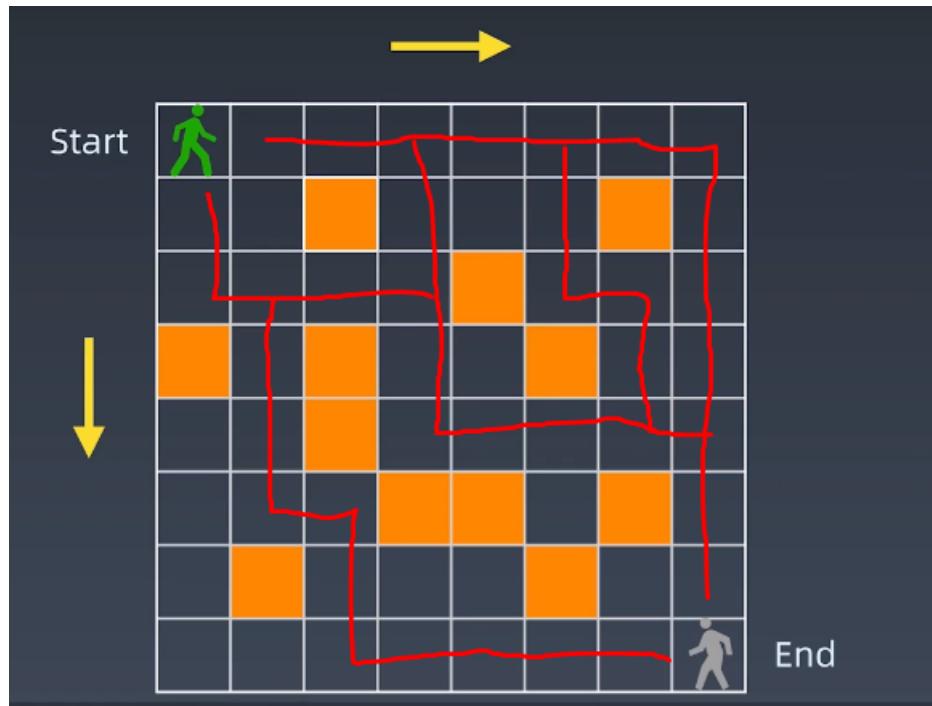
```



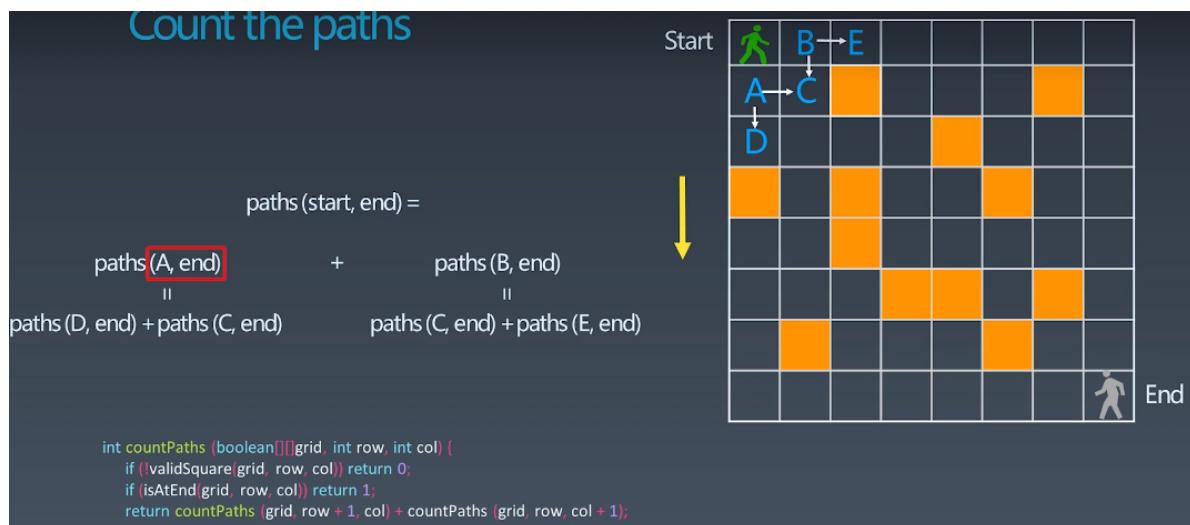
2.路径计数



一个人从Start走到End，只能选择向右走或者向下走，一共有几种走法



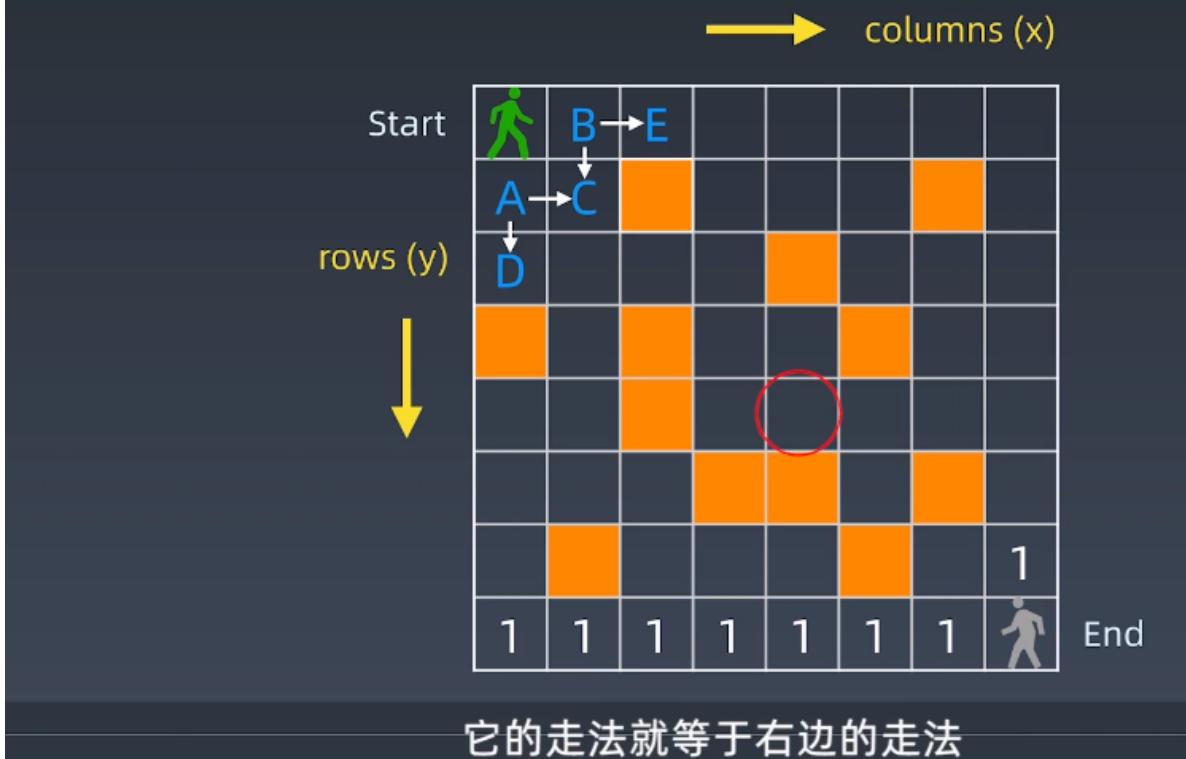
分治思想



自底向上的递归方法：

将靠近end的格子先走一遍

Count the paths

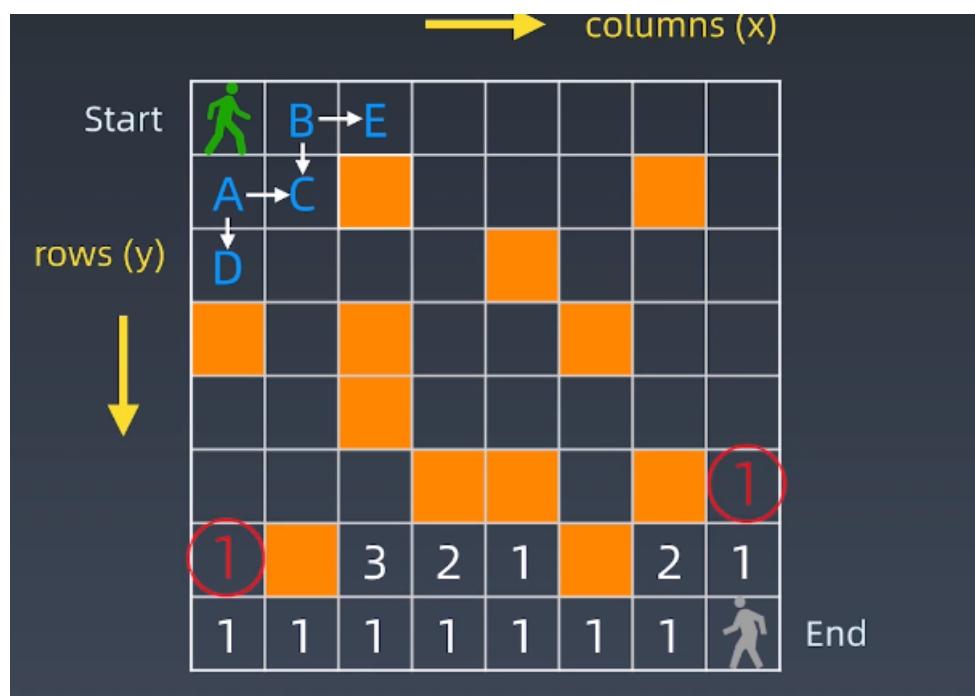


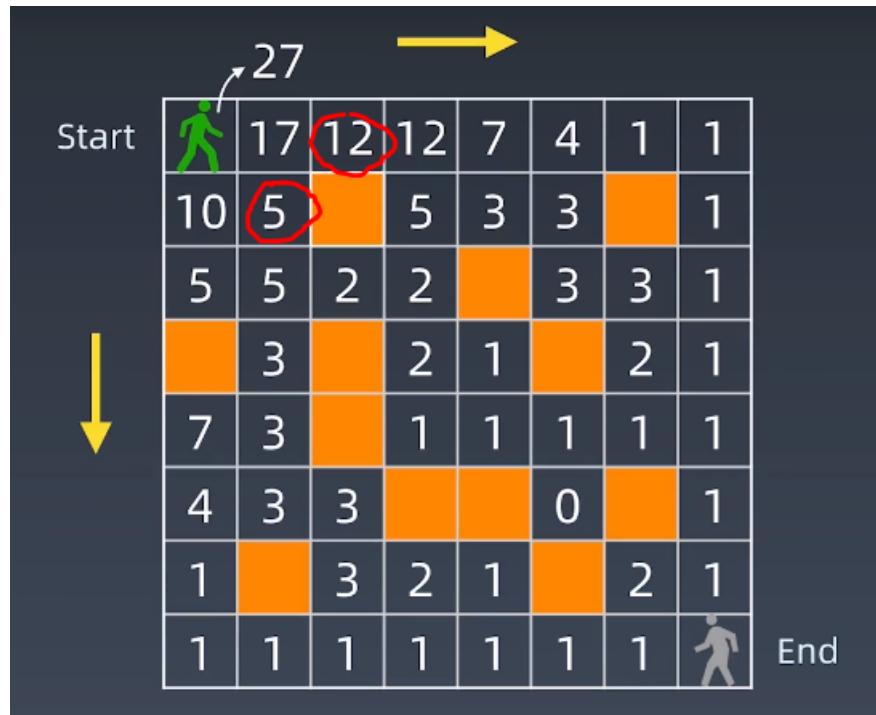
状态转移方程 (DP方程)

```
// 递推公式: opt[i,j] = opt[i+1,j] + opt[i,j+1]
// 完整逻辑:
if a[i,j] = "empty":
    opt[i,j] = opt[i+1,j] + opt[i,j+1]
else:
    opt[i,j] = 0
```

判断每格棋盘的点是不是空地，如果是空地则进行递推

如果不是则等于0 (不可走的点)





一共可能的走法 ---> 27种走法

动态规划的关键点

1. 最优子结构 $\text{opt}[n] = \text{best_of}(\text{opt}[n-1], \text{opt}[n-2], \dots)$

2. 储存中间状态: $\text{opt}[i]$

3. 递推公式 (美其名曰: 状态转移方程或者 DP 方程)

Fib: $\text{opt}[i] = \text{opt}[n-1] + \text{opt}[n-2]$

二维路径: $\text{opt}[i,j] = \text{opt}[i+1][j] + \text{opt}[i][j+1]$ (且判断 $a[i,j]$ 是否空地)

3. 不同路径

一个机器人位于一个 $m \times n$ 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？

示例 1：



输入： $m = 3, n = 7$

输出： 28

示例 2:

输入: m = 3, n = 2

输出: 3

解释:

从左上角开始, 总共有 3 条路径可以到达右下角。

1. 向右 -> 向下 -> 向下
2. 向下 -> 向下 -> 向右
3. 向下 -> 向右 -> 向下

示例 3:

输入: m = 7, n = 3

输出: 28

示例 4:

输入: m = 3, n = 3

输出: 6

解题思路

从end位置开始, 从下往上循环, 从右往左循环

4.不同路径II

解题思路:

参考障碍物的那道题目

并对障碍物进行判断

5.字符串问题

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

解题思路:

(1) 暴力求解 ----> O(2^n)

枚举text1/text2 中所有的子序列 ---> 递归 (取/不取生成子序列)

查看其是否在另一个text中存在

可能的情况：

1. S1 = "" S2 = 任意字符串
2. S1 ="A" S2 = 任意
3. S1 =".....A" S2 = ".....A"

(2) 转化成二维数组 ----> 最常见的求解字符串问题

动态规划小结

1. 打破自己的思维惯性，形成机器思维
2. 理解复杂逻辑的关键
3. 也是职业进阶的要点要领

补充内容：MIT algorithm course

B 站搜索： mit 动态规划

动态规划习题

6.爬楼梯

1.上一级两级三级楼梯应该怎么走 (easy)

```
n = 0, {0}  
n = 1, {1}  
n = 2, {11 , 2}  
n = 3, {111, 12, 21, 3}
```

```
var climbStairs3 = function (n){  
    let res=[0,1,2,4];  
    for(let i=4;i<=n;i++){  
        res[i] = res[i-3] + res[i-2] + res[i-1];  
    }  
    return res[n]  
}
```

2.相邻两步的步伐不能相同 (medium)

dp => $f(n) = f(n-1) + f(n-2) + f(n-3)$

7.三角形最小路径和

1. brute-force, 递归, n层: left or right ---> 2^n (每次可以选择在下一层往左或者往右走)
2. DP
 - a. 重复性 (分治) $\text{sub}(i,j) = \min(\text{sub}(i+1, j), \text{sub}(i+1, j+1)) + a[i, j]$
 - b. 定义状态数组: $f[i,j]$
 - c. DP方程: $f[i,j] = \min(f[i+1, j], f[i+1, j+1]) + a[i, j]$

高票回答：

[https://leetcode.com/problems/triangle/discuss/38735/Python-easy-to-understand-solutions-\(top-down-bottom-up\)](https://leetcode.com/problems/triangle/discuss/38735/Python-easy-to-understand-solutions-(top-down-bottom-up))

7.最大子序和

给定一个整数数组 `nums`，找到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。

示例 1：

```
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 。
```

示例 2：

```
输入: nums = [1]
输出: 1
```

示例 3：

```
输入: nums = [0]
输出: 0
```

示例 4：

```
输入: nums = [-1]
输出: -1
```

解题思路：

1.暴力求解

所有的枚举起点和终点（起点必须从正数开始正数结尾）---->O(N^2)

2.dp:

- 分治（子问题） $\text{max_sum}(i) = \text{MAX}(\text{max_sum}(i-1), 0) + a[i]$
- 状态数组定义： $f[i]$
- dp方程： $f[i] = \text{MAX}(f[i-1], 0) + a[i]$

8.零钱兑换

给定不同面额的硬币 `coins` 和一个总金额 `amount`。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 `-1`。

你可以认为每种硬币的数量是无限的。

示例 1:

```
输入: coins = [1, 2, 5], amount = 11
```

```
输出: 3
```

```
解释: 11 = 5 + 5 + 1
```

示例 2:

```
输入: coins = [2], amount = 3
```

```
输出: -1
```

示例 3:

```
输入: coins = [1], amount = 0
```

```
输出: 0
```

示例 4:

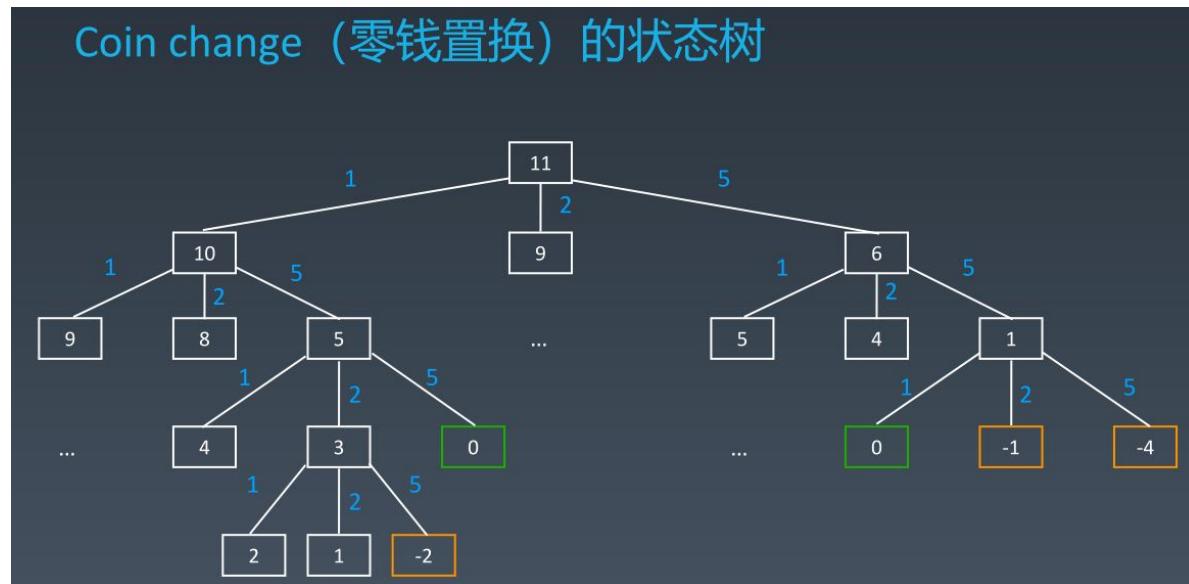
```
输入: coins = [1], amount = 1
```

```
输出: 1
```

解题思路:

转换思路=>参考爬楼梯的题目

1.暴力法-->递归（从结果溯源）



第一次变负的结点停掉

树的层次表示已经用了几个硬币

想要凑成至少需要3个硬币

2.BFS ---> 广度优先遍历

3.dp: $f(n) = \min\{f(n-k), \text{for } k \in [1, 2, 5]\} + 1$

a. subproblem

b. dp array:

c. dp 方程

Week07 高级搜索

目录

剪枝

双向 BFS

启发式搜索 (A*)

重要：过遍数！！不要死磕

初级搜索

1. 朴素搜索

2. 优化方式: 不重复 (fibonacci) [将中间值存放到数组中/顺推避免生成中间值]、剪枝 (生成括号问题) [状态数不满足条件的时候就不进行搜索]

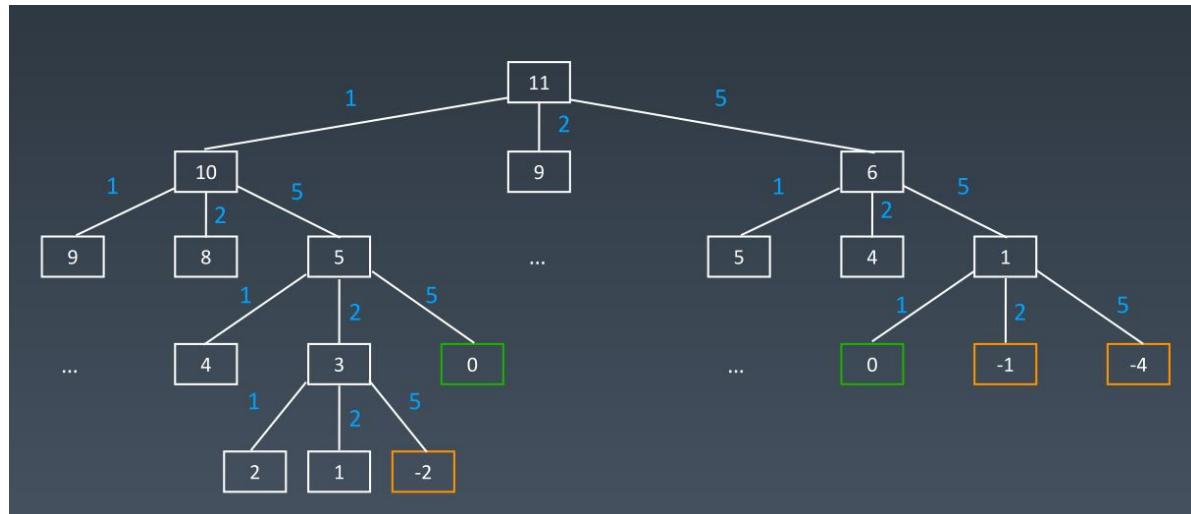
3. 搜索方向:

DFS: depth first search 深度优先搜索 ---> 栈/队列(先入后出, 先入先出)

BFS: breadth first search 广度优先搜索

双向搜索、启发式搜索 ---> 优先队列(优先级搜索)

Coin change 零钱置换的状态树



搜索问题 -----> 树形结合思想

代码模板

1. terminator
2. process current node here
3. drill down
4. 恢复当前的状态

Java

```
public List<List<Integer>> levelOrder(TreeNode root) {  
    List<List<Integer>> allResults = new ArrayList<>();  
    if(root == null){  
        return allResults;  
    }  
    travel(root,0,allResults);  
    return allResults  
}  
  
  
private void travel(TreeNode root,int level,List<List<Integer>> results){  
    if(results.size() == level){  
        results.add(new ArrayList<>());  
    }  
    results.get(level).add(root.val);  
    if(root.left != null){  
        travel(root.left,level+1,results);  
    }  
    if(root.right != null){  
        travel(root.right,level+1,results);  
    }  
}
```

JavaScript

```
const visited = new Set();  
const dfs = node =>{  
    if(visited.has(node)) return  
    visited.add(node)  
    dfs(node.left)  
    dfs(node.right)  
}
```

非递归形式--->用stack模拟

```
def DFS(self,tree):  
  
    if tree.root is None:  
        return []  
  
    visited,stack = [],[tree.root]  
  
    while stack:  
        node = stack.pop()  
        visited.add(node)  
  
        process(node)
```

```
nodes = generate_related_nodes(node)
stack.push(nodes)
```

```
#other processing work
```

```
...
```

BFS模板

```
const bfs = (root)=> {
  let result=[],queue = [root]
  while(queue.length > 0){
    let level =[], n=queue.length
    for(let i=0;i<n;i++){
      let node = queue.pop()
      level.push(node.val)
      if(node.left) queue.unshift(node.left)
      if(node.right) queue.unshift(node.right)
    }
    result.push(level)
  }
  return result
};
```

剪枝

分支不够好/次优分支，直接剪掉

回溯法

回溯法采用试错的思想，它尝试分步的去解决一个问题。在分步解决问题的过程中，当它通过尝试发现现有的分步答案不能得到有效的正确的解答的时候，它将取消上一步甚至是上几步的计算，再通过其它的可能的分步解答再次尝试寻找问题的答案。

回溯法通常用最简单的递归方法来实现，在反复重复上述的步骤后可能出现两种情况：

- 找到一个可能存在的正确的答案
- 在尝试了所有可能的分步方法后宣告该问题没有答案

在最坏的情况下，回溯法会导致一次复杂度为指数时间的计算

实战题目

1.爬楼梯

2.括号生成

剪枝的条件：left < n & right < left

使用动态规划DP求解：

<https://leetcode-cn.com/problems/generate-parentheses/solution/zui-jian-dan-yi-dong-de-do/>

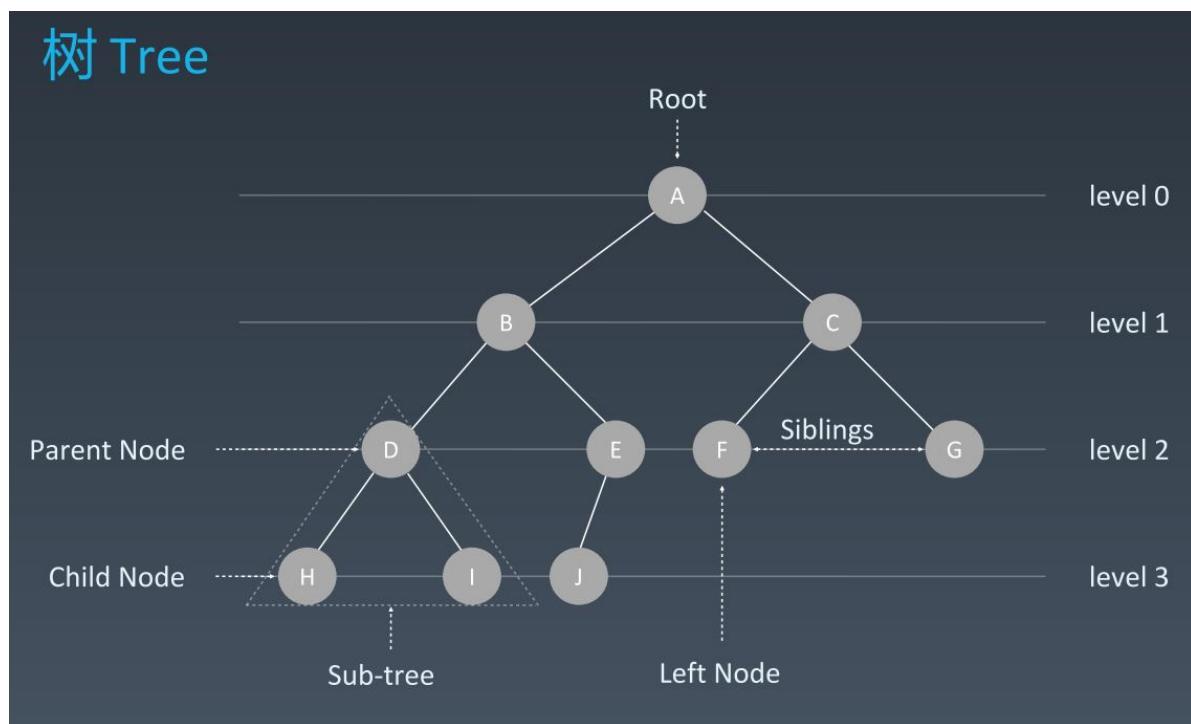
3.N皇后

4.有效的数独

Week08

字典树和并查集

树Tree



二叉树的层次遍历-Javascript

```
var levelOrder = function(root) {
    let res = [];
    if(root === null){
        return res;
    }
    //利用队列
    let queue = [root];
    while(queue.length > 0){
        let levelRes = [];
        let nums = queue.length;
        for(let i = 0; i < nums; i++){
            let curr = queue.shift();
            levelRes.push(curr.val);
            if(curr.left !== null){
                queue.push(curr.left);
            }
            if(curr.right !== null){
                queue.push(curr.right);
            }
        }
        res.push(levelRes);
    }
}
```

```
    return res;  
};
```

解题思路：利用队列

在while循环里（即队列不为空时）

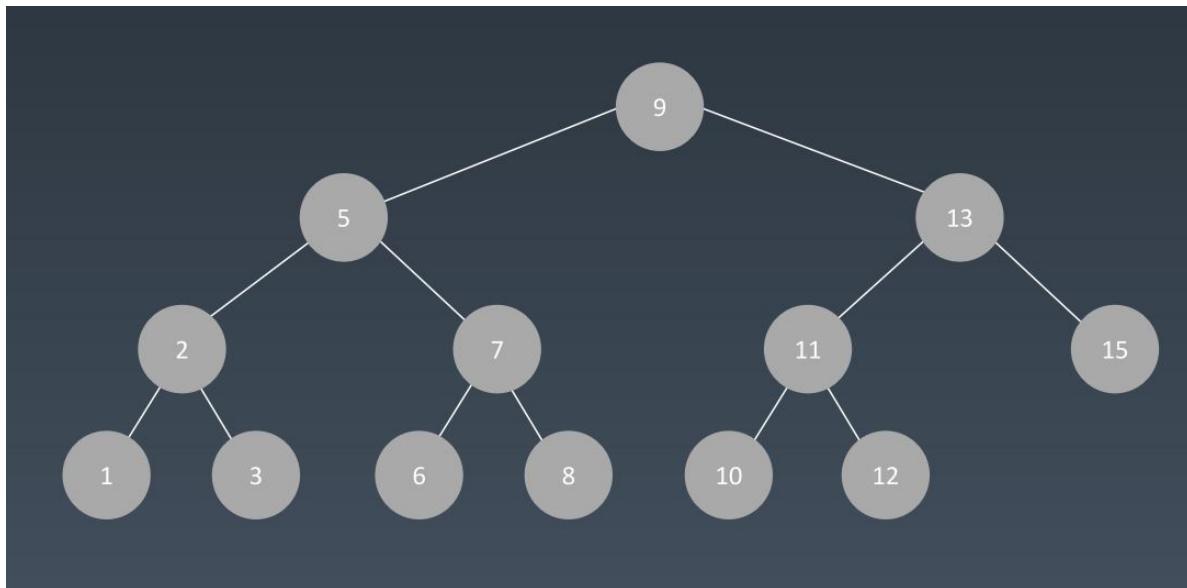
先记录一下当前队列中数量（即该层树的结点个数）

利用for循环，一个一个shift出结点，每个结点出去的时候，判断一下其有没有左右子树，有的话push进队列..

for循环结束后（即遍历完一层了），将该层结果放入最终结果中

return 结果

二叉搜索树



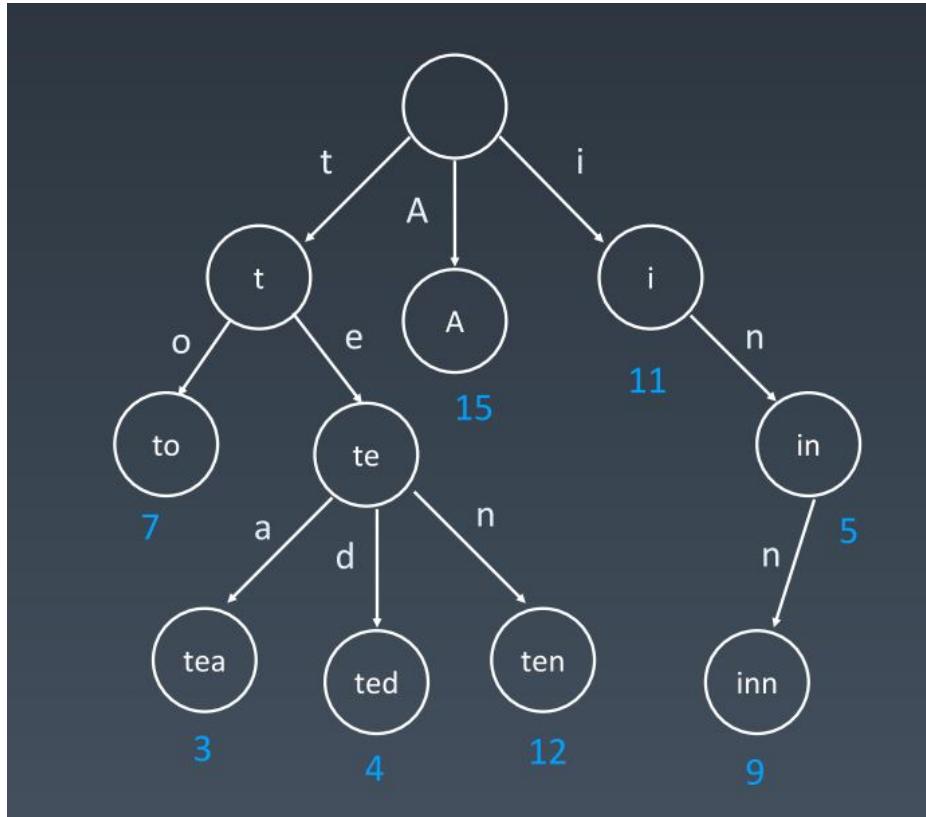
定义：任意一个结点，左子树的所有结点都要小于根结点，右子树的所有结点都要大于根结点，对于任何子树都满足这两个特性

二叉搜索树中序遍历是升序 ---> 查找效率会变高

字典树基本结构

字典树，即 Trie 树，又称单词查找树或键树，是一种树形结构。典型应用是用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计

它的优点是：最大限度地减少无谓的字符串比较，查询效率比哈希表高

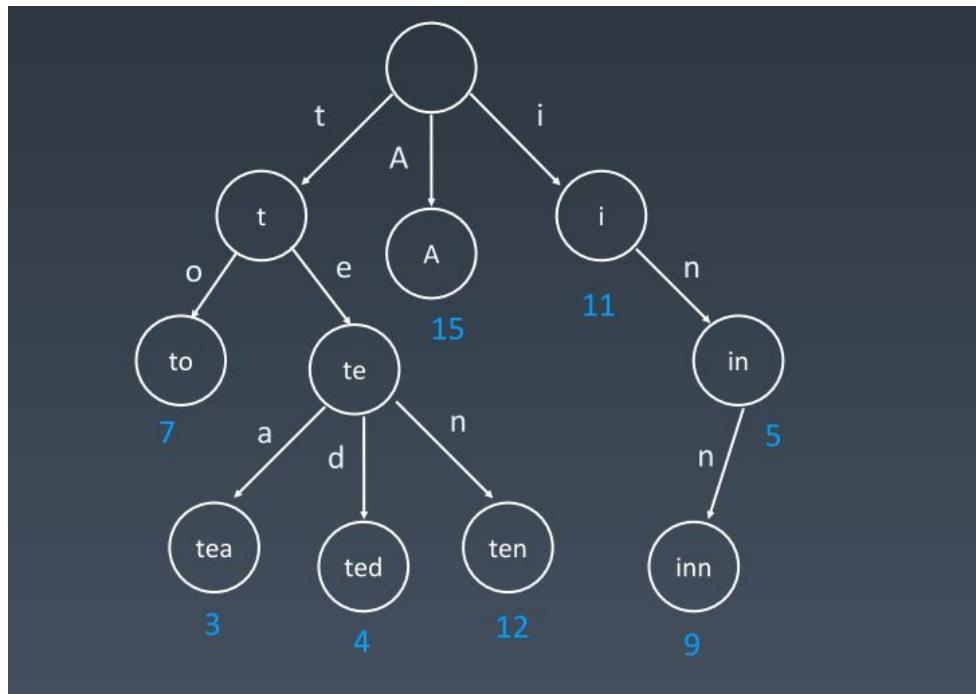


trie树是一种多叉树

基本性质

1. 结点本身不存完整单词；
2. 从根结点到某一结点，路径上经过的字符连接起来，为该结点对应的字符串；
3. 每个结点的所有子结点路径代表的字符都不相同

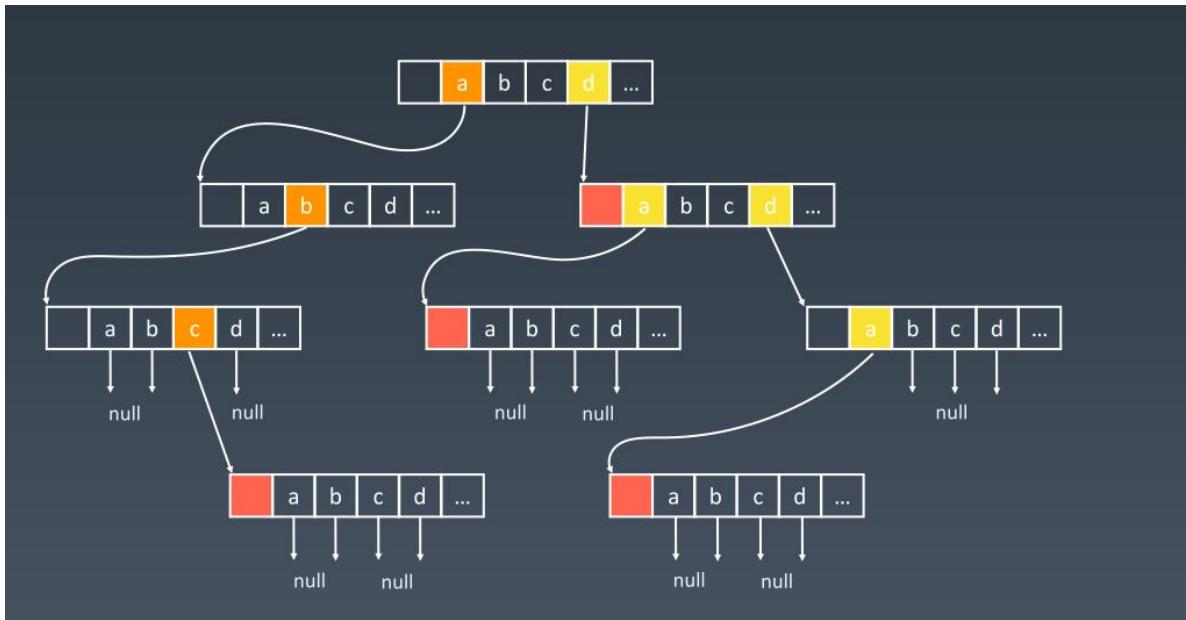
结点存储额外信息



15--->单词统计的计数

节点还可以存储其他额外信息

结点的内部实现



指向下一个结点的不同指针，使用相应的字符指向下一个结点

核心思想

Trie 树的核心思想是空间换时间

利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的

实现题目

1. 实现Trie

Java的实现

```
class TrieNode { //实现TrieNode
    public char val;
    public boolean isWord; //标记是否是word
    public TrieNode[] children = new TrieNode[26];
    public TrieNode() {}
    TrieNode(char c){
        TrieNode node = new TrieNode();
        node.val = c;
    }
}

public class Trie{
    private TrieNode root;
    public Trie(){
        root = new TrieNode();
        root.val = ' ';
    }
    public void insert(String word){//把每一个单词放入字典树中
        TrieNode ws = root;
        for(int i=0;i<word.length();i++){
            char c = word.charAt(i);
            if(ws.children[c - 'a'] == null){
                ws.children[c - 'a']= new TrieNode(c);
            }
            ws=ws.children[c - 'a'];
        }
    }
}
```

```

        }
        ws.isWord = true;
    }

    public boolean search(String word){//search有没有相应结点
        TrieNode ws = root;
        for(int i=0;i<word.length();i++){
            char c = word.charAt(i);
            if(ws.children[c - 'a']==null) return false;
            ws = ws.children[c - 'a'];
        }
        return ws.isWord;
    }

    public boolean startsWith(String prefix){
        TrieNode ws = root;
        for(int i=0;i<prefix.length();i++){
            char c = prefix.charAt(i);
            if(ws.children[c - 'a'] == null) return false;
            ws = ws.children[c - 'a'];
        }
        return true;
    }
}

```

Javascript实现

```

class Trie{
    constructor(){
        this.root = {};
        this.endOfWord = "$";
    }

    insert(word){
        let node = this.root;
        for(let ch of word){
            node[ch] = node[ch] || {};
            node = node[ch];
        }
    }

    search(word){
        let node = this.root;
        for(let ch of word){
            if(!node[ch]) return false;
            node = node[ch];
        }
        return node[this.endOfWord] === this.endOfWord;
    }

    startsWith(word){
        let node = this.root;
        for(let ch of word){
            if(!node[ch]) return false;
            node = node[ch];
        }
        return true;
    }
}

```

```
let trie = new Trie();
console.log(trie.insert("apple"));
console.log(trie.search("apple")); // 返回 true
console.log(trie.search("app")); // 返回 false
console.log(trie.startsWith("app")); // 返回 true
console.log(trie.insert("app"));
console.log(trie.search("app")); // 返回 true
```

2. 单词搜索 II

212. 单词搜索 II

难度 困难 355 ★★★

给定一个 $m \times n$ 二维字符网格 board 和一个单词（字符串）列表 words，找出所有同时在二维网格和字典中出现的单词。

单词必须按照字母顺序，通过 **相邻的单元格** 内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母在一个单词中不允许被重复使用。

示例 1：

o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v

输入： board = [["o","a","a","n"],["e","t","a","e"], ["i","h","k","r"],["i","f","l","v"]], words = ["oath","pea","eat","rain"]

输出： ["eat","oath"]

解题思路：

四联通：上下左右相邻

方法I： words遍历 ----> board search

时间复杂度 $O(N \times m \times m \times 4^k)$

方法II： trie

(1) all words ---> Trie 构建起prefix

(2) board, DFS

时间复杂度 $O(N \times 4^k)$

Python实现

```
dx = [-1, 1, 0, 0]
dy = [0, 0, -1, 1]
END_OF_WORD = "#"
class Solution(object):
    def findWords(self, board, words):
        if not board or not board[0]: return []
        if not words: return [] self.result = set()

        # 构建trie
        root = collections.defaultdict()
        for word in words:
            node = root
            for char in word:
                node = node.setdefault(char, collections.defaultdict())
                node[END_OF_WORD] = END_OF_WORD

        self.m, self.n = len(board), len(board[0])
        for i in xrange(self.m):
            for j in xrange(self.n):
                if board[i][j] in root:
                    self._dfs(board, i, j, "", root)
        return list(self.result)
```

JavaScript实现

```
var findWords = function(board, words) {
    var TrieNode=function(){
        this.isEnd=false;
        this.next={};
    }
    var Trie=function(){
        this.root=new TrieNode();
    }

    Trie.prototype.insert=function(word){
        if(!word) return false;
        let node=this.root;
        for(let i=0;i<word.length;i++){
            if(!node.next[word[i]]){
```

```

        node.next[word[i]]=new TrieNode();
    }
    node=node.next[word[i]];
}
node.isEnd=true;
}

Trie.prototype.search=function(word){
let node=this.searchByPath(word);
return node!=null?node.isEnd:false;
}

Trie.prototype.searchByPath=function(word){
if(!word.length) return true;
let node=this.root;
for(let i=0;i<word.length;i++){
    if(!node.next[word[i]]) return false;
    node=node.next[word[i]];
}
return node;
}

Trie.prototype.startsWith=function(prefix){
if(!prefix.length) return true;
let node=this.searchByPath(prefix);
return node!=null;
}

let m=board.length;
let n=board[0].length;

let wordTrie=new Trie();
for(let i=0;i<words.length;i++){
    wordTrie.insert(words[i]);
}

var DFS=function(i,j,curStr,currNode){
//找到某单词的最后一个了
if(currNode.isEnd){
    result.push(curStr);
    currNode.isEnd=false;
}
//到边界就不能继续了
if(i<0||j<0||i>m||j>n){return;}

const restore=board[i][j];
//已经访问过也就是 '#' 或者字典树中没有这个节点，就直接返回
if(restore==='#'||!currNode.next[restore]){return;}

board[i][j]='#';
curStr += restore;
//向上下左右四个方向进行前进
DFS(i-1,j,curStr,currNode.next[restore]);
DFS(i+1,j,curStr,currNode.next[restore]);
DFS(i,j-1,curStr,currNode.next[restore]);
DFS(i,j+1,curStr,currNode.next[restore]);

//回溯
}

```

```
        board[i][j]=restore;
    }
    let result=[];
    for(let i=0;i<m;i++){
        for(let j=0;j<n;j++){
            DFS(i,j,"",wordTrie.root);
        }
    }
    return result;
};
```

并查集Disjoint Set

适用场景

组团、配对问题

Group or not?

基本操作

- makeSet(s): 建立一个新的并查集，其中包含 s 个单元素集合
- unionSet(x, y): 把元素 x 和元素 y 所在的集合合并，要求 x 和 y 所在的集合不相交，如果相交则不合并
- find(x): 找到元素 x 所在的集合的代表，该操作也可以用于判断两个元素是否位于同一个集合，只要将它们各自的代表比较一下就可以了

实战题目

1. 省份数量

547. 省份数量

难度 中等

524

收藏

分享

切换为英文

接收动态

反馈

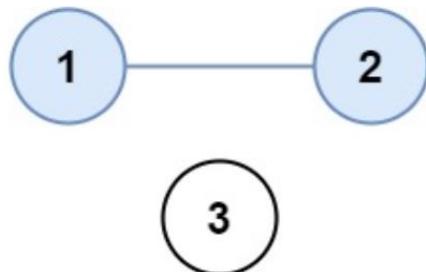
有 n 个城市，其中一些彼此相连，另一些没有相连。如果城市 a 与城市 b 直接相连，且城市 b 与城市 c 直接相连，那么城市 a 与城市 c 间接相连。

省份 是一组直接或间接相连的城市，组内不含其他没有相连的城市。

给你一个 $n \times n$ 的矩阵 isConnected ，其中 $\text{isConnected}[i][j] = 1$ 表示第 i 个城市和第 j 个城市直接相连，而 $\text{isConnected}[i][j] = 0$ 表示二者不直接相连。

返回矩阵中 **省份** 的数量。

示例 1：



输入： `isConnected = [[1,1,0],[1,1,0],[0,0,1]]`

输出： 2

解题思路：

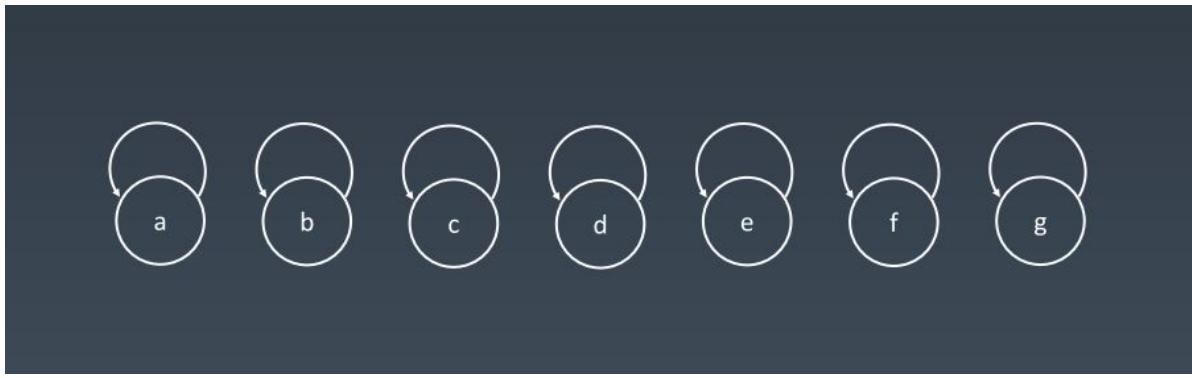
1.DFS/BFS (类似岛屿问题)

染色问题---> 遍历node: if node ==1 count ++ --->将node和附近结点变成0

```
var findCircleNum = function(M) {
    const isVisited = [];
    let res = 0;
    function mapDFS(i) {
        for (let j = 0; j < M.length; j++) {
            if(M[i][j] && !isVisited[j]) {
                isVisited[j] = true;
                mapDFS(j);
            }
        }
    }
    for(let i = 0; i < M.length; i++) {
        if(!isVisited[i]) {
            isVisited[i] = true;
            mapDFS(i);
            res++;
        }
    }
    return res;
}
```

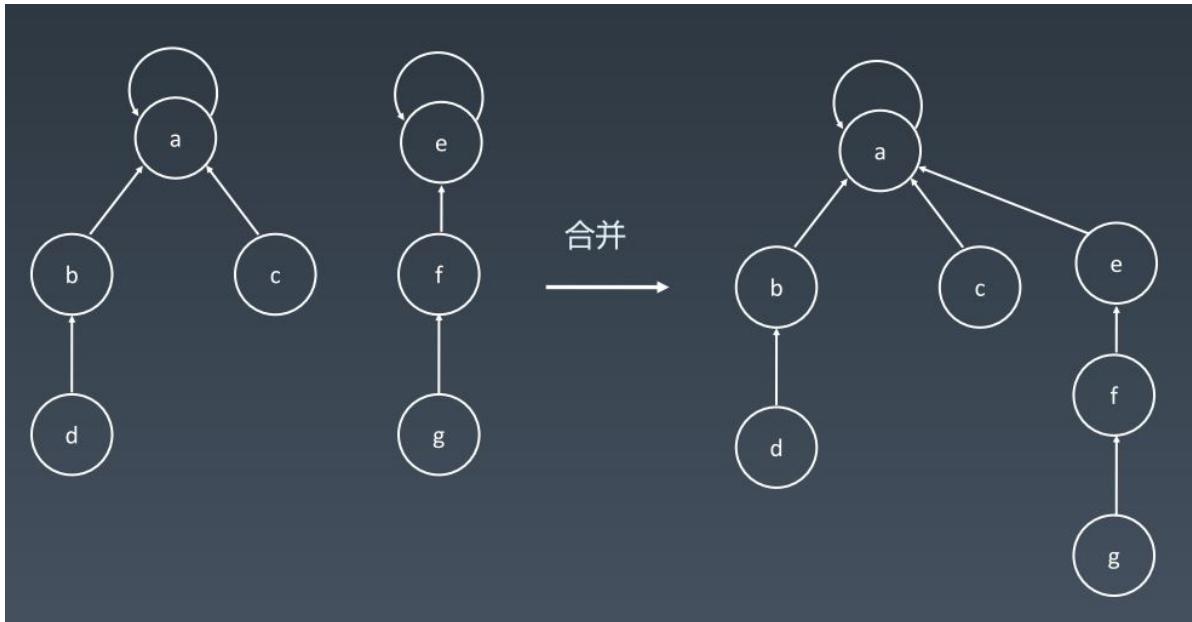
2.并查集

初始化

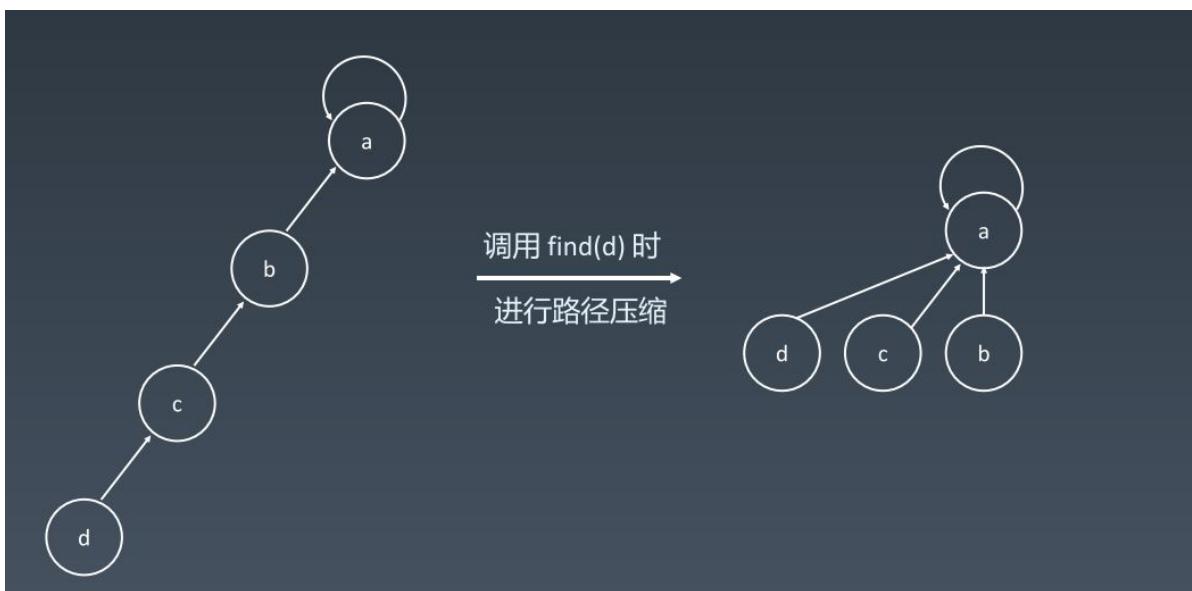


一开始每一个元素拥有parent数组指向自己 --> 自己是自己的集合

查询、合并



路径压缩



Java实现并查集

```
class UnionFind {
    private int count = 0;
    private int[] parent;
    public UnionFind(int n) {
        count = n;
        parent = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }
    public int find(int p) {
        while (p != parent[p]) {
            parent[p] = parent[parent[p]]; //找到领头元素
            p = parent[p];
        }
        return p;
    }
    public void union(int p, int q) { //合并操作
        int rootP = find(p); //p的领头元素
        int rootQ = find(q); //q的领头元素
        if (rootP == rootQ) return;
        parent[rootP] = rootQ; //合并领头元素
        count--; //独立集合减少1
    }
}
```

JavaScript并查集模板

```
class unionFind {
    constructor(n) {
        this.count = n;
        this.parent = new Array(n);
        for (let i = 0; i < n; i++) {
            this.parent[i] = i;
        }
    }
    find(p) {
        let root = p;
        while (parent[root] !== root) {
            root = parent[root];
        }
        // 压缩路径
        while (parent[p] !== p) {
            let x = p;
            p = this.parent[p];
            this.parent[x] = root;
        }
        return root;
    }
    union(p, q) {
        let rootP = find(p);
        let rootQ = find(q);
        if (rootP === rootQ) return;
        parent[rootP] = rootQ;
        count--;
    }
}
```

```

        this.parent[rootP] = rootQ;
        this.count--;
    }
}

```

JavaScript-省份数量

```

var findCircleNum = function(M) {
    let count = M.length, parent = [], rank = Array(M.length).fill(0);
    function find(i) {
        while(parent[i] !== undefined) i = parent[i];
        return i;
    }
    function union(i, j) {
        const rootI = find(i), rootJ = find(j);
        if (rootI === rootJ) return;
        if (rank[rootI] > rank[rootJ]) {
            parent[rootJ] = rootI;
            rank[rootI]++;
        } else {
            parent[rootI] = rootJ;
            rank[rootJ]++;
        }
        count--;
    }
    for(let i = 0; i < M.length; i++) {
        for(let j = 0; j <= i; j++) {
            if (M[i][j]) union(i, j);
        }
    }
    return count;
}

```

高级树、AVL树和红黑树

二叉树遍历Pre-order/In-order/Post-order

1. 前序(Pre-order): 根-左-右
2. 中序(In-order): 左-根-右
3. 后序(Post-order): 左-右-根

```

def preorder(self, root):
    if root:
        self.traverse_path.append(root.val)
        self.preorder(root.left)
        self.preorder(root.right)

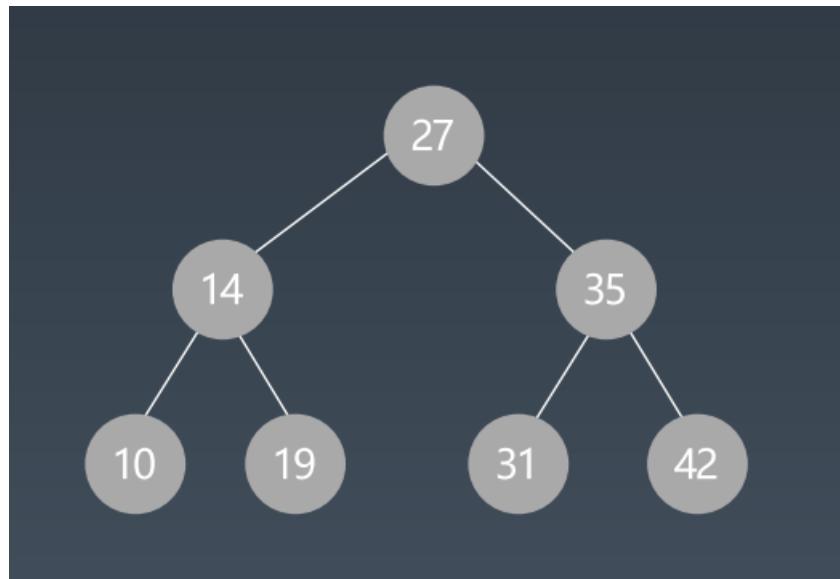
def inorder(self, root):
    if root:
        self.inorder(root.left)
        self.traverse_path.append(root.val)
        self.inorder(root.right)

def postorder(self, root):
    if root:
        self.postorder(root.left)

```

```
self.postorder(root.right)  
self.traverse_path.append(root.val)
```

二叉搜索树 Binary Search Tree



时间复杂度 $O(\log_2 N)$

二叉搜索树，也称二叉搜索树、有序二叉树（Ordered Binary Tree）、排序二叉树（Sorted Binary Tree），是指一棵空树或者具有下列性质的二叉树：1. 左子树上所有结点的值均小于它的根结点的值；2. 右子树上所有结点的值均大于它的根结点的值；3. 以此类推：左、右子树也分别为二叉查找树（这就是重复性！）

中序遍历：升序排列

如何查找结点 ----> 时间复杂度与层数有关 $O(\log_2 N)$ N 表示树中总的结点个数

保证性能的关键 ---> 不退化成链表

1. 保证二维维度！ —> 左右子树结点平衡 (recursively)
2. Balanced
3. https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree

Type

2-3 Tree

AA tree

AVL tree

B-tree

black-red tree

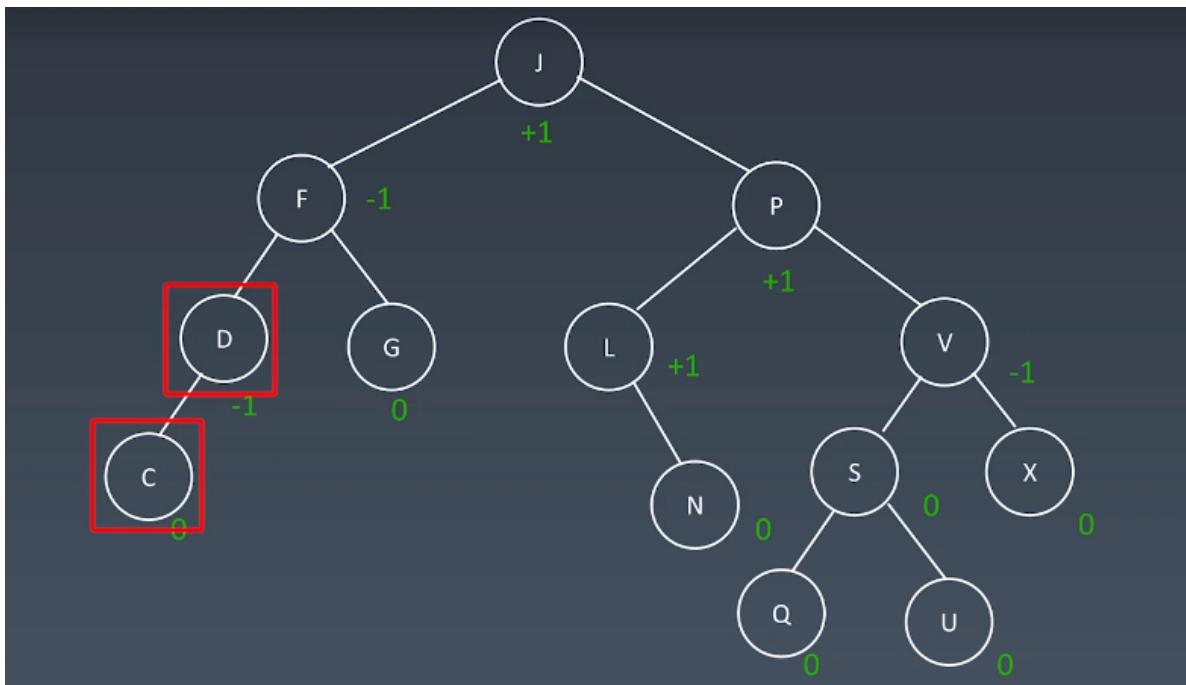
AVL Tree

1. 发明者G. M. Adelson-Velsky和Evgenii Landis
2. Balance Factor (平衡因子)：
是它的左子树的高度减去它的右子树的高度（有时相反）

balance factor = {-1, 0, 1}

3. 通过旋转操作来进行平衡（四种）
4. https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree

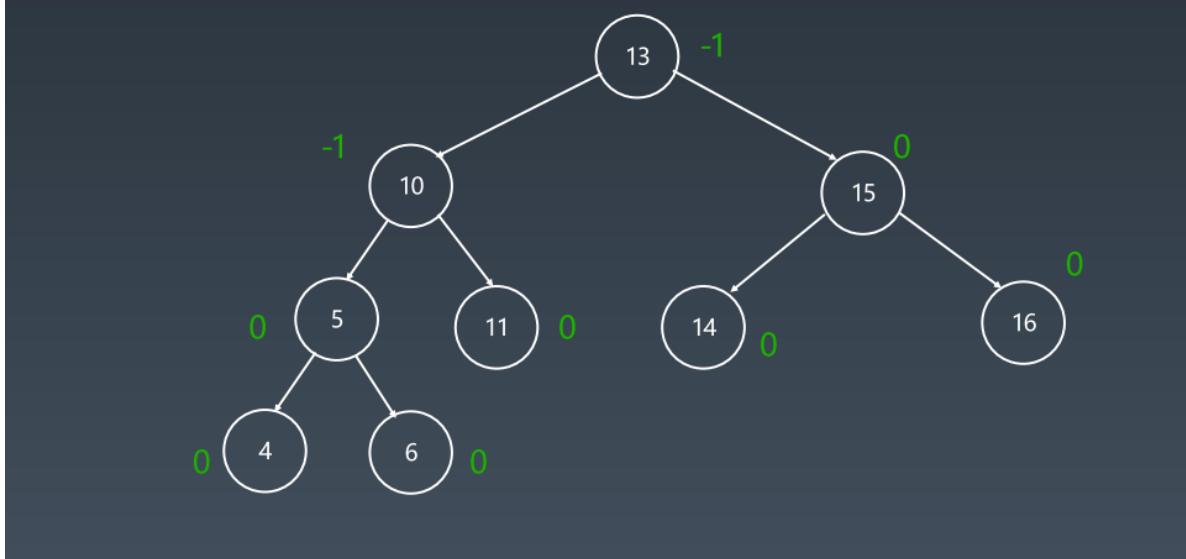
记录左右子树高度



右 - 左 = 平衡因子

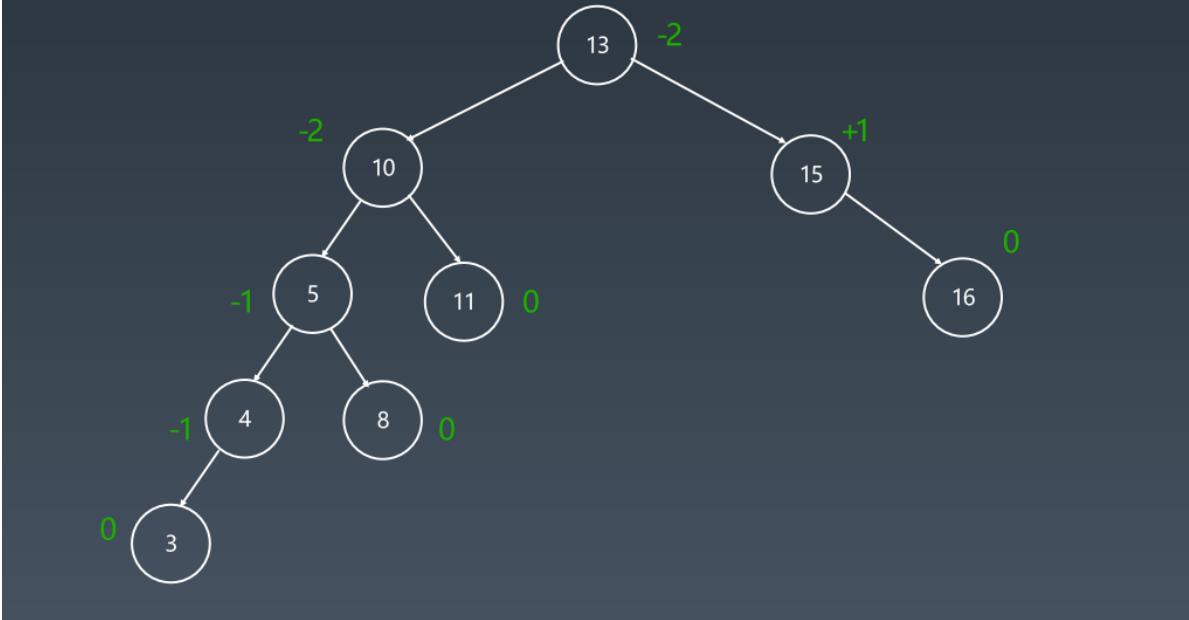
保持平衡因子绝对值不超过1 ----> 二叉搜索树

增加 14



$2 - 3 = -1$

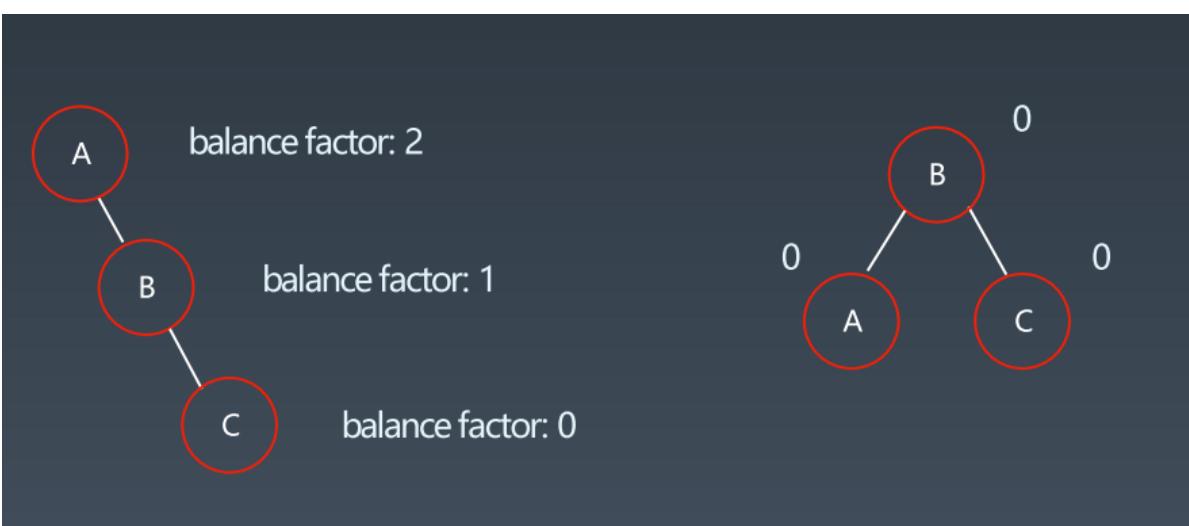
增加 3



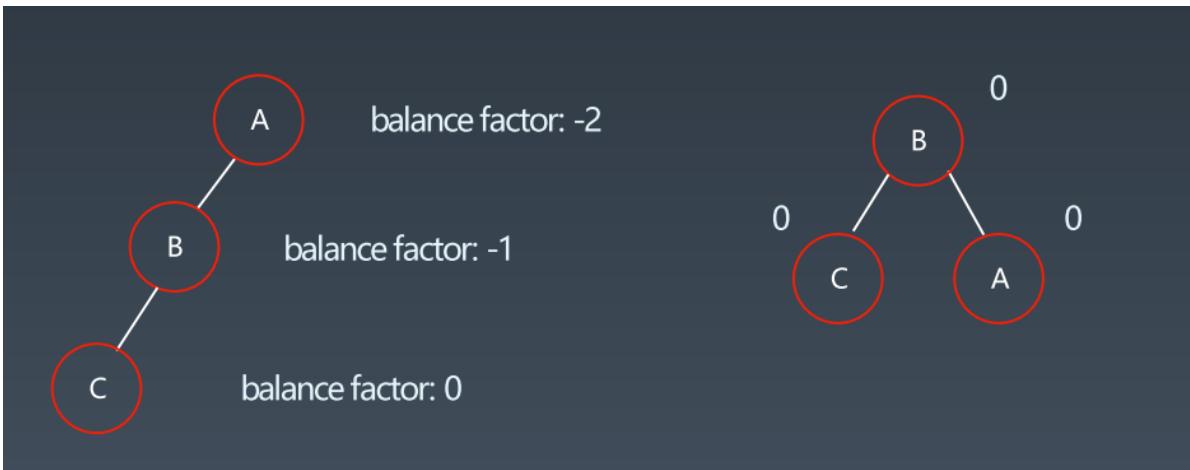
旋转操作

- 1.左旋
- 2.右旋
- 3.左右璇
- 4.右左璇

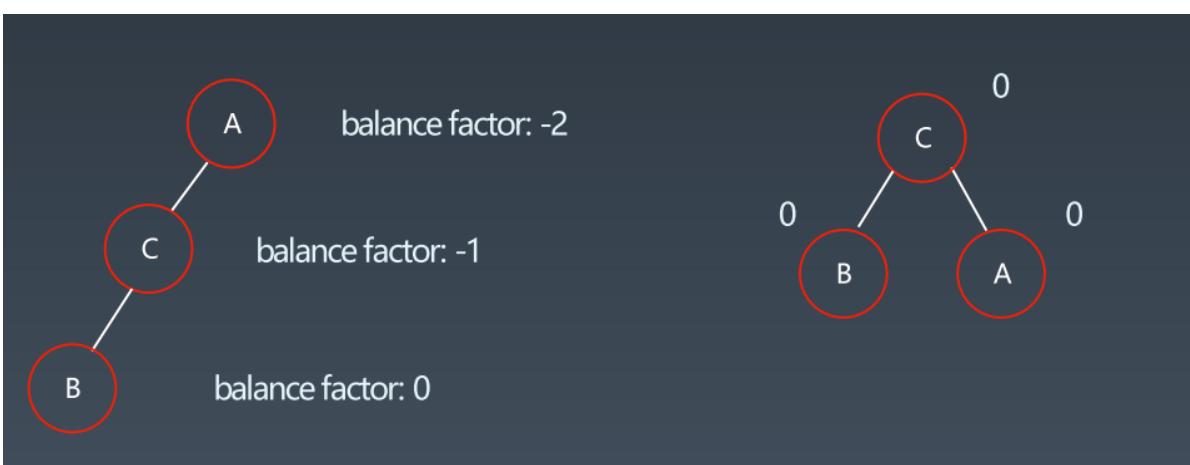
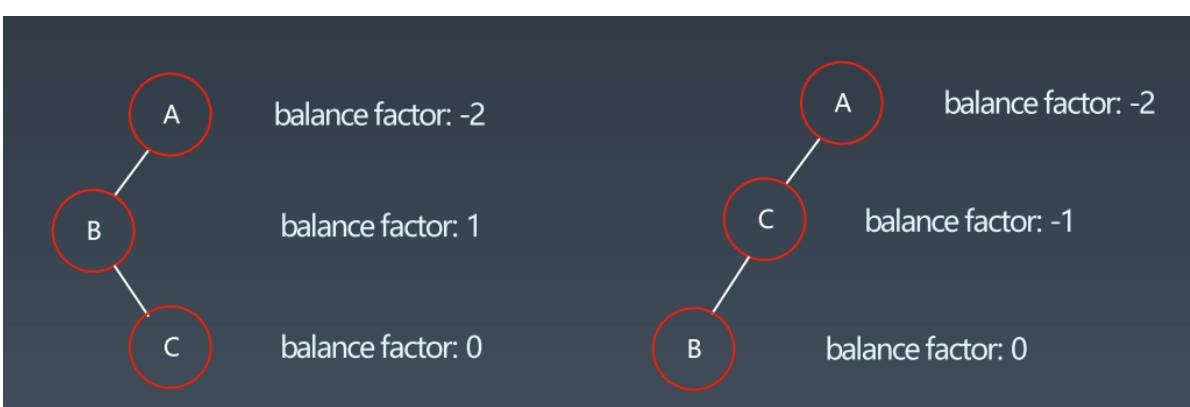
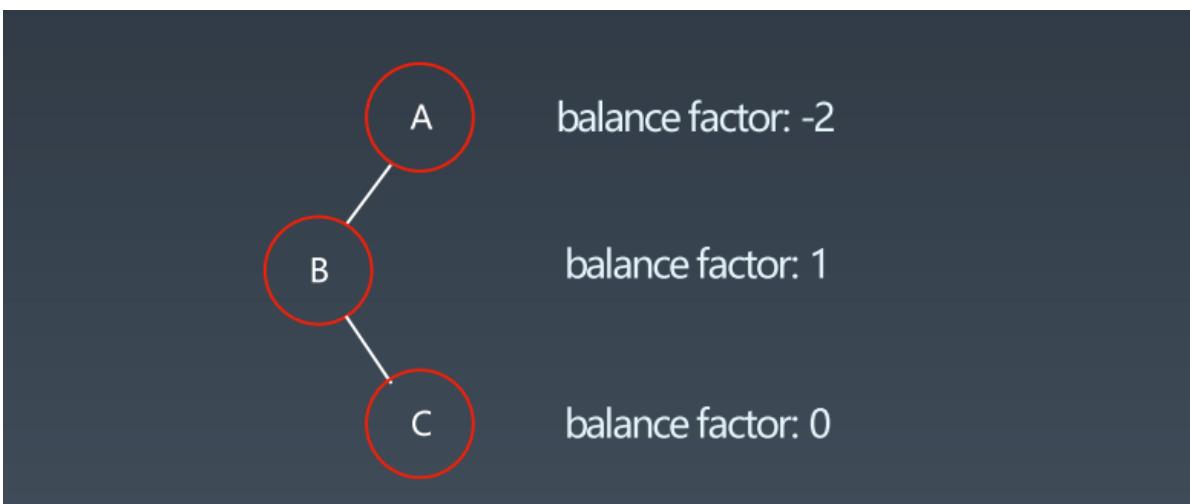
子树形态：右右子树-->左旋



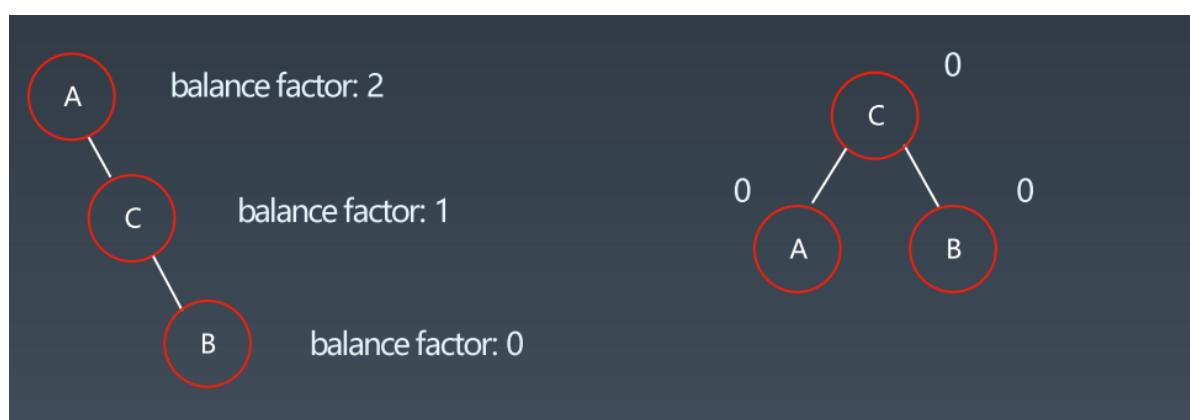
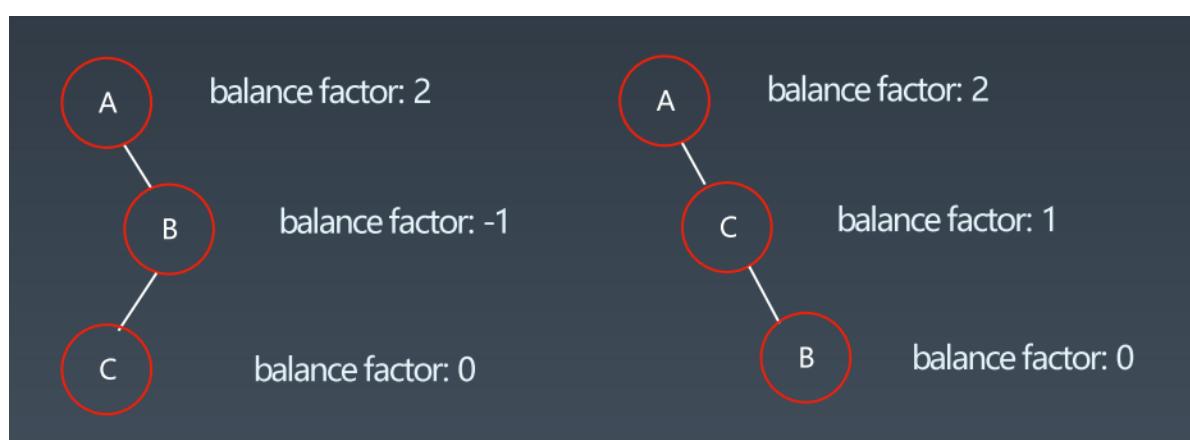
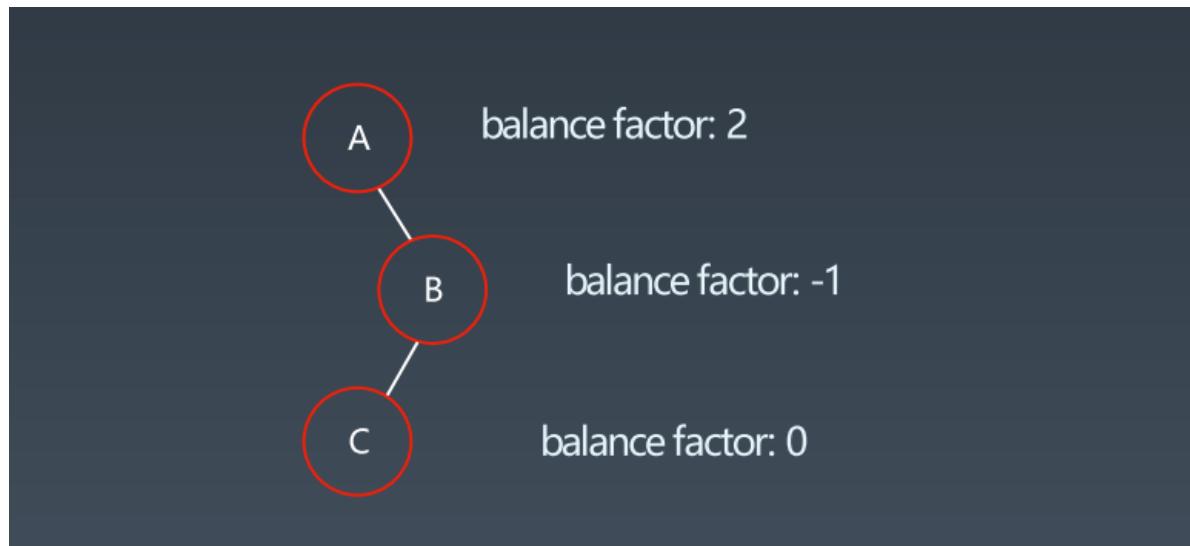
子树形态：左左子树 -> 右璇



子树形态：左右子树—> 左右旋



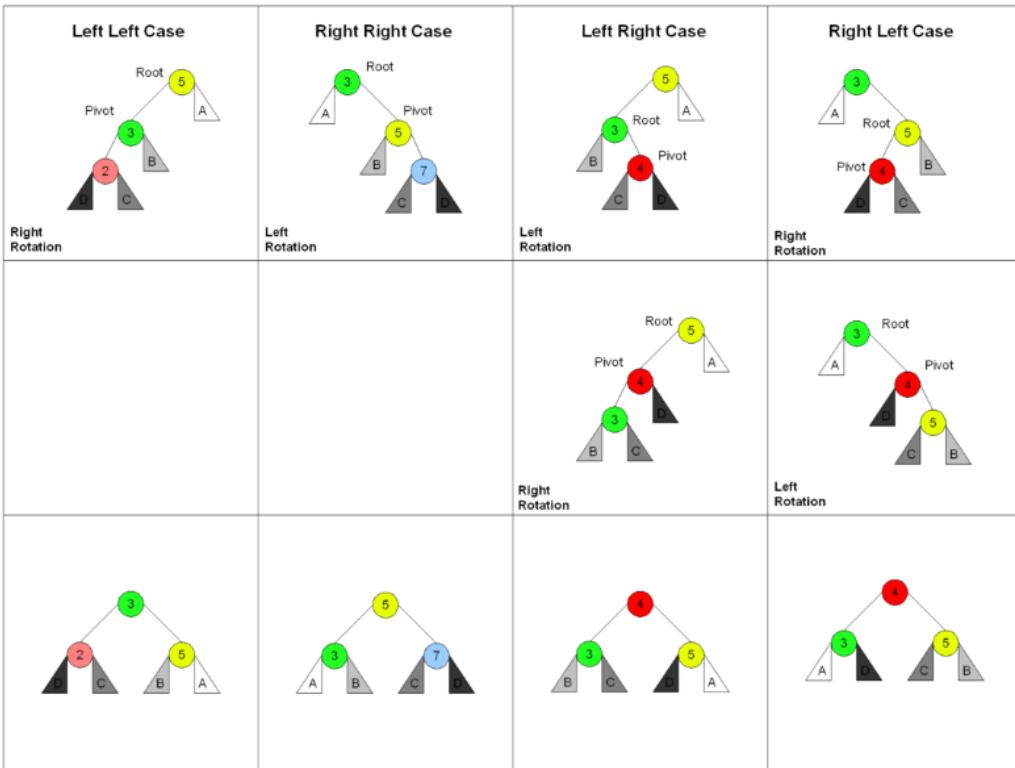
子树形态：右左子树—> 右左旋



带有子树的旋转

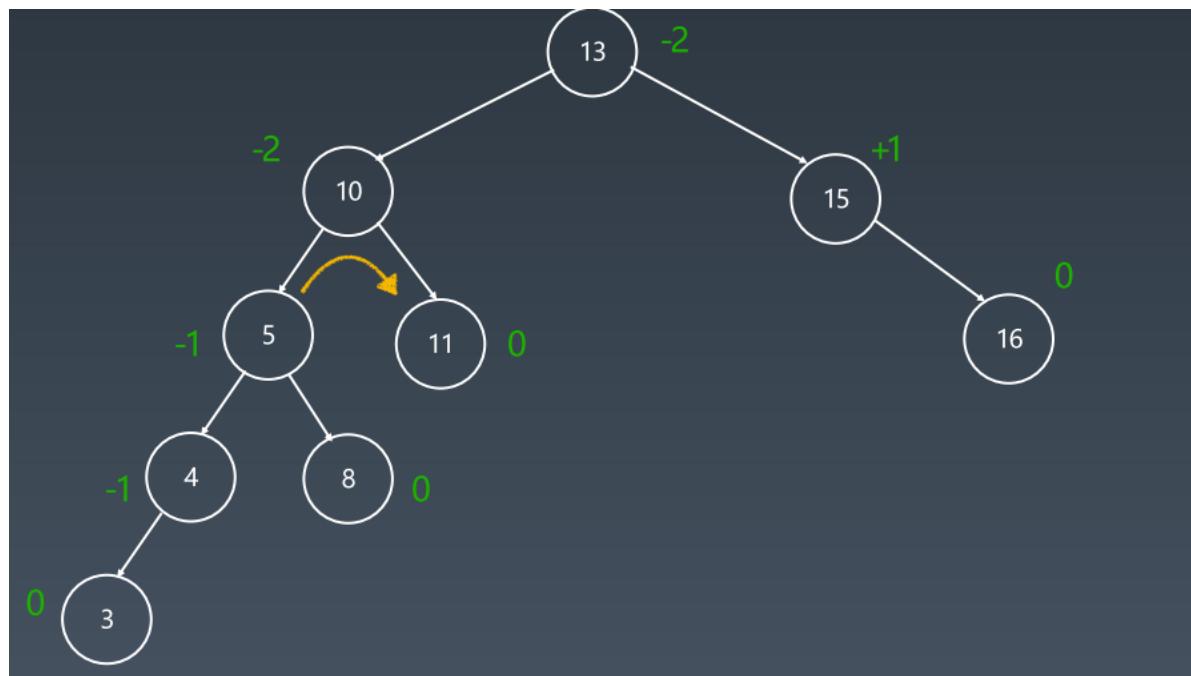
There are 4 cases in all, choosing which one is made by seeing the direction of the first 2 nodes from the unbalanced node to the newly inserted node and matching them to the top most row.

Root is the initial parent before a rotation and **Pivot** is the child to take the root's place.



AVL的由来：查询的时间复杂度 = 树的深度 => 平衡因子

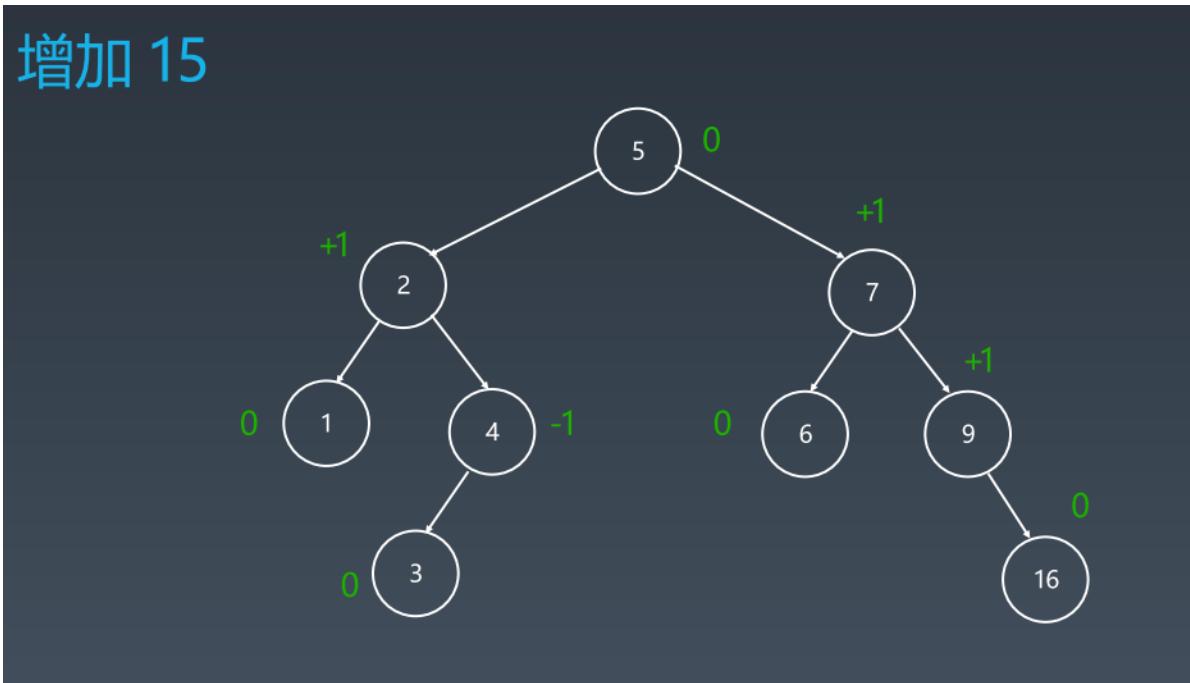
向右旋转



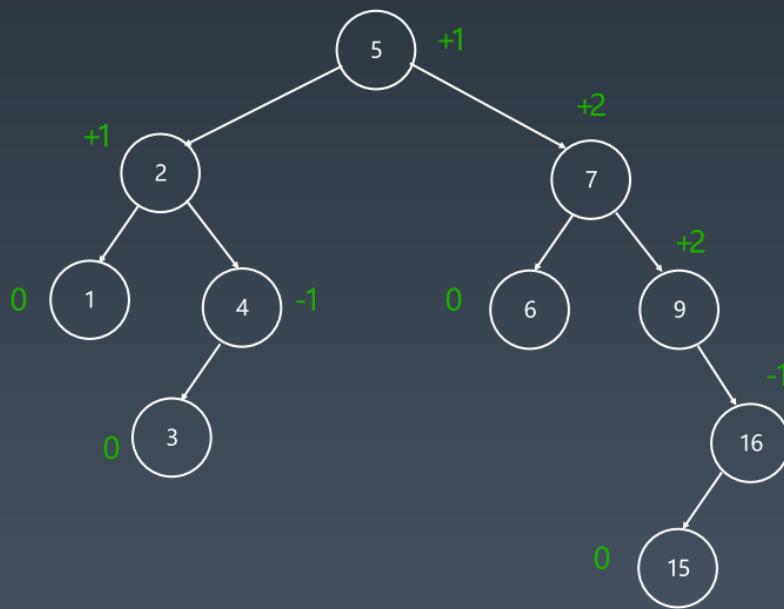
5向上提，8挂在10下面



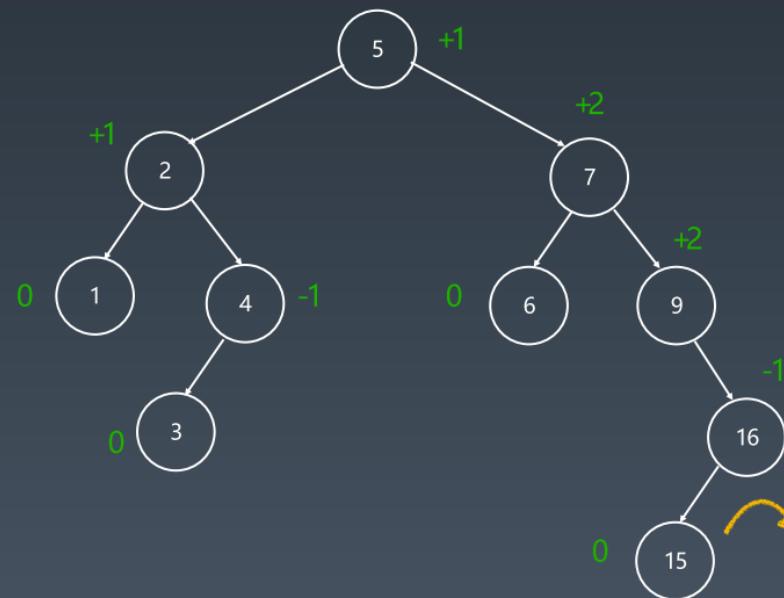
增加 15

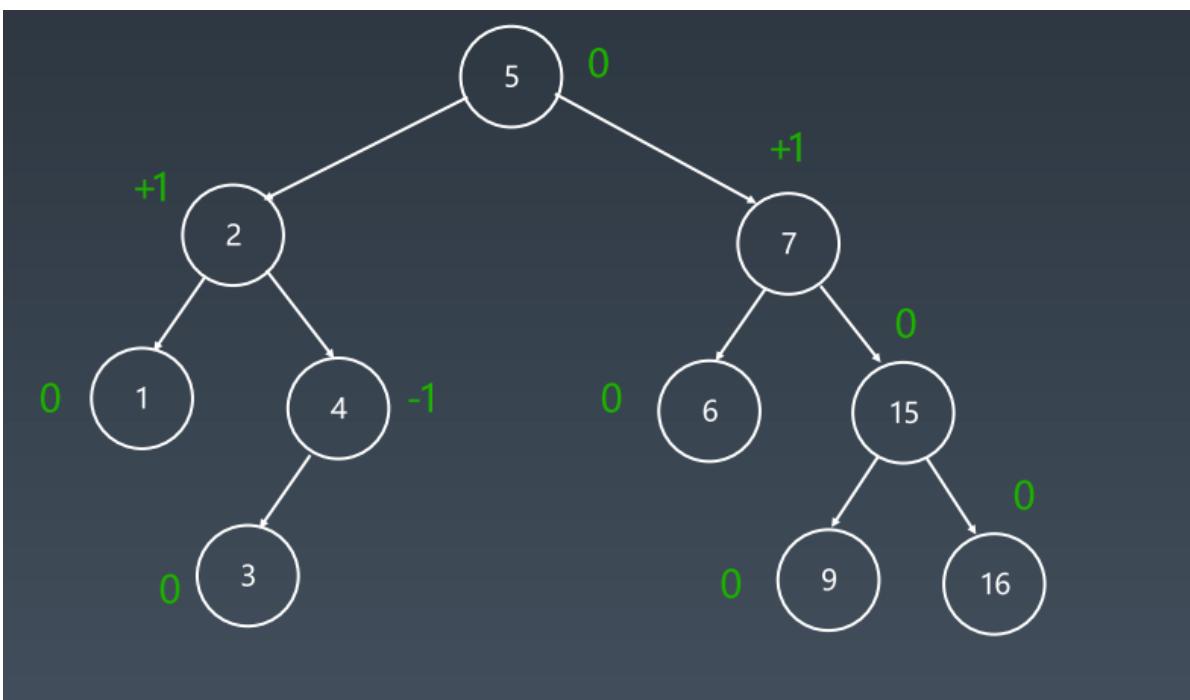
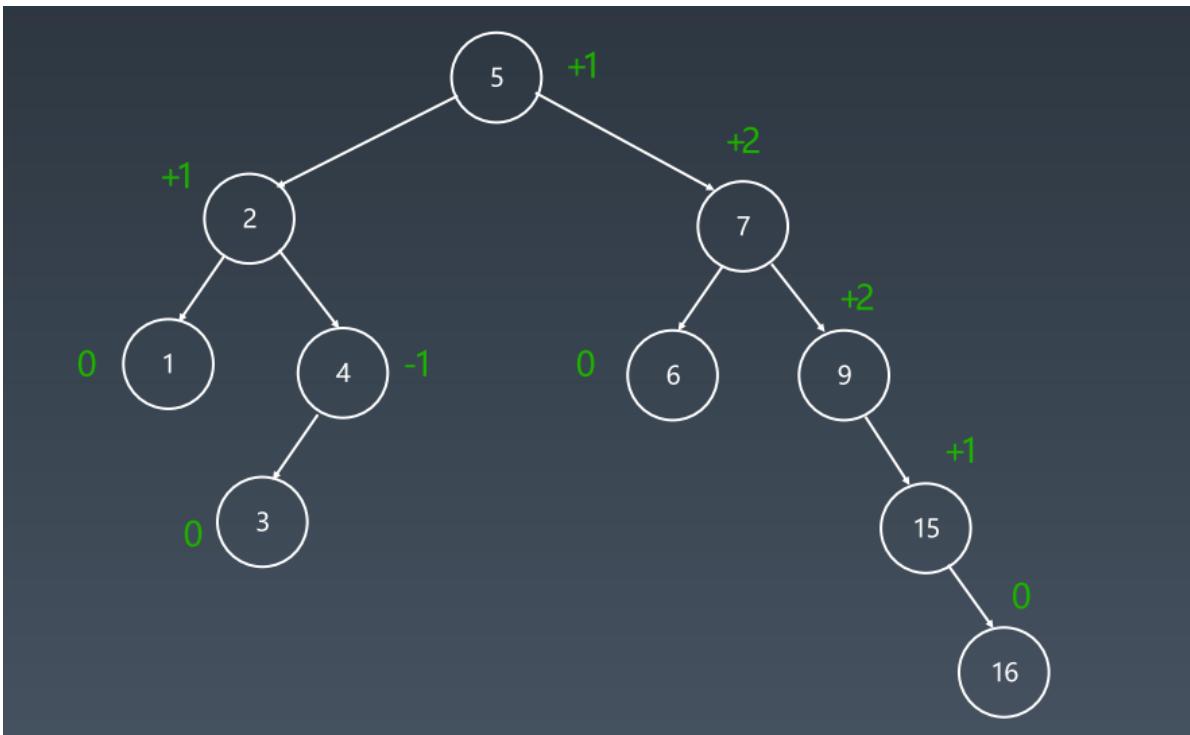


增加 15



向右旋转





AVL总结

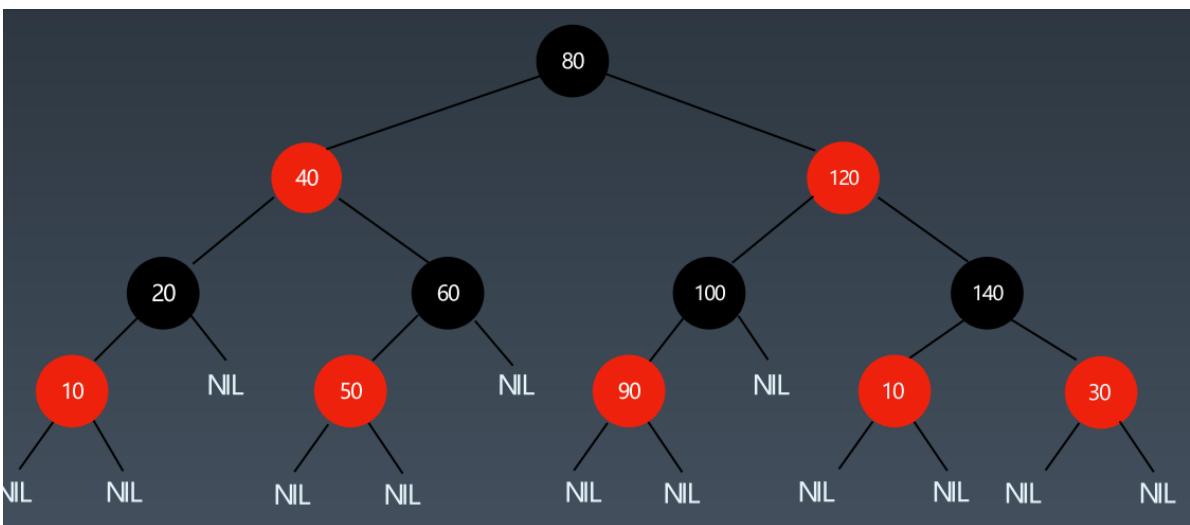
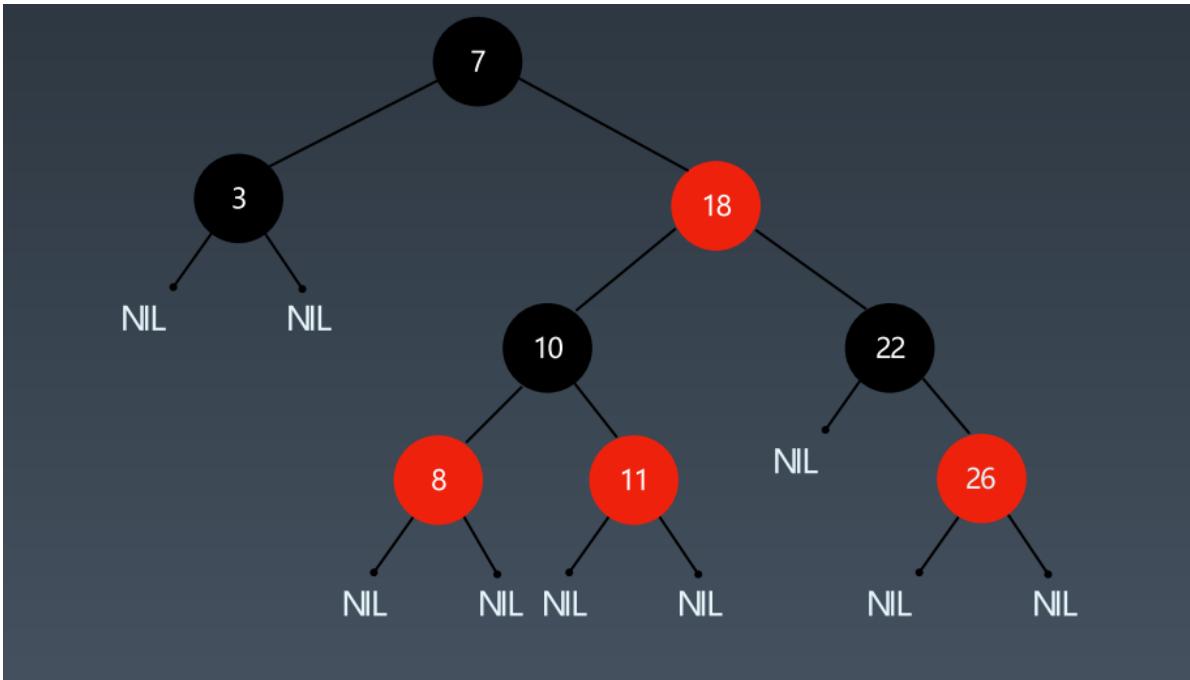
- 平衡二叉搜索树
- 每个结点存balance factor = {-1, 0, 1}
- 四种旋转操作
- 不足：结点需要存储额外信息、且调整次数频繁

红黑树

红黑树是一种近似平衡的二叉搜索树 (BinarySearch Tree)，它能够确保任何一个结点的左右子树的高度差小于两倍。具体来说，红黑树是满足如下条件的二叉 搜索树：

- 每个结点要么是红色，要么是黑色
- 根结点是黑色
- 每个叶结点 (NIL结点，空结点) 是黑色的

- 不能有相邻接的两个红色结点
- 从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点



关键性质

从根到叶子的最长的可能路径不多于最短的可能路径的两倍长

对比

- AVL trees provide faster lookups than Red Black Trees because they are more strictly balanced.
- Red Black Trees provide faster insertion and removal operations than AVL trees as fewer rotations are done due to relatively relaxed balancing.
- AVL trees store balance factors or heights with each node, thus requires storage for an integer per node whereas Red Black Tree requires only 1 bit of information per node.
- Red Black Trees are used in most of the language libraries like map, multimap, multiset in C++ whereas AVL trees are used in databases where faster retrievals are required.

读操作非常非常多，写操作非常非常少的情况下，用AVL树，比如数据库，微博等；

写操作非常多或者插入和删除操作一半一半，读操作非常少用红黑树比如map，set等。

位运算

为什么需要位运算

- 机器里的数字表示方式和存储格式就是二进制
- 十进制 \leftrightarrow 二进制：如何转换？

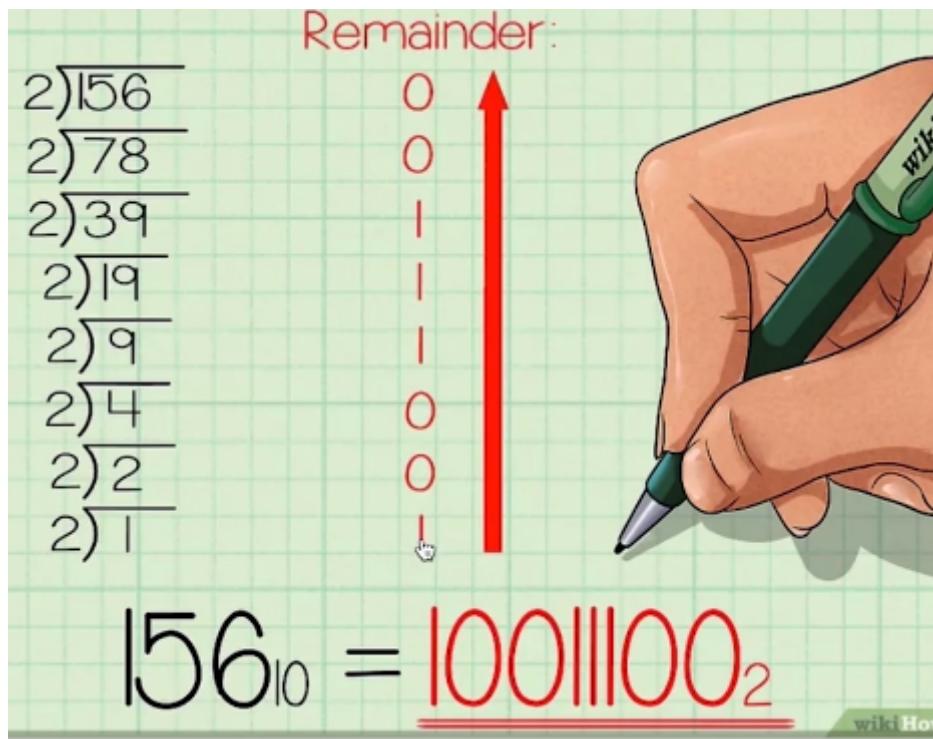
<https://zh.wikihow.com/%E4%BB%8E%E5%8D%81%E8%BF%9B%E5%88%B6%E8%BD%AC%E6%8D%A2%E4%B8%BA%E4%BA%8C%E8%BF%9B%E5%88%B6>

4(d): 0100

8(d): 01000

5(d): 0101

6(d): 0110



位运算符

含义	运算符	实例
左移	$<<$	$0011 => 0110$
右移	$>>$	$0110 => 0011$
按位或	$ $	$0011 1011 => 1011$
按位与	$\&$	$0011 \& 1011 => 0011$
按位取反	\sim	$0011 => 1100$
按位异或（相同为零不同为一）	\wedge	$0011 \wedge 1011 => 1000$

XOR - 异或

异或：相同为0，不同为1。也可用“不进位加法”来理解。

异或操作的一些特点：

$x \wedge 0 = x$ //与0相同就是0，不相同就是它本身

$x \wedge 1s = \sim x$ // 注意 $1s = \sim 0$

$x \wedge (\sim x) = 1s$

$x \wedge x = 0$

$c = a \wedge b \Rightarrow a \wedge c = b, b \wedge c = a$ // 交换两个数

$a \wedge b \wedge c = a \wedge (b \wedge c) = (a \wedge b) \wedge c$ // associative

指定位置的位运算

1. 将x 最右边的n 位清零: $x \& (\sim 0 << n)$
2. 获取x 的第n 位值 (0 或者1) : $(x >> n) \& 1$
3. 获取x 的第n 位的幂值: $x \& (1 << n)$
4. 仅将第n 位置为1: $x | (1 << n)$
5. 仅将第n位置为0: $x \& (\sim (1 << n))$
6. 将x 最高位至第n 位 (含) 清零: $x \& ((1 << n) - 1)$

实战位运算要点

- 判断奇偶: $x \% 2 == 1 \rightarrow (x \& 1) == 1$ $x \% 2 == 0 \rightarrow (x \& 1) == 0$
- $x >> 1 \rightarrow x / 2$.

即: $x = x / 2; \rightarrow x = x >> 1;$

$mid = (left + right) / 2; \rightarrow mid = (left + right) >> 1;$

- $X = X \& (X-1)$ 清零最低位的1

- $X \& -X \Rightarrow$ 得到最低位的1

- $X \& \sim X \Rightarrow 0$

实战题目

1.位1的个数

191. 位1的个数

难度 简单 333 收藏 分享 切换为英文 接收动态 反馈

编写一个函数，输入是一个无符号整数（以二进制串的形式），返回其二进制表达式中数字位数为 '1' 的个数（也被称为汉明重量）。

提示：

- 请注意，在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。
- 在 Java 中，编译器使用二进制补码记法来表示有符号整数。因此，在上面的 [示例 3](#) 中，输入表示有符号整数 -3。

示例 1：

输入：00000000000000000000000000001011

输出：3

解释：输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。

示例 2：

输入：000000000000000000000000000010000000

输出：1

解释：输入的二进制串 000000000000000000000000000010000000 中，共有一位为 '1'。

解题思路

1. for loop: 0 ----> 32

2.%2看最后1位是否是1, /2把最后一位移掉 (32)

3.&1 判断最后一位是0/1, x = x >> 1; (32)

4. while(x>0) { count++;x = x & (x - 1); } ---> 有多少个1就循环多少次

Java实现

```
public int hammingWeight(int n){  
    int sum = 0;  
    while( n!= 0){  
        sum++;  
        n &= (n-1)  
    }  
    return sum;  
}
```

Python的四种解法

方法I：调用函数

```
class Solution(object):
    def hammingWeight(self,n):
        """
        :type n: int
        :rtype: int
        """
        return bin(n).count('1')
```

方法II：手动循环计算1的个数

```
class Solution(object):
    def hammingWeight(self,n):
        """
        :type n: int
        :rtype: int
        """
        n = bin(n)
        count = 0
        for c in n:
            if c == "1":
                count += 1
        return count
```

方法III：十进制转二进制的方法。每次对2取余判断是否是1，是的话就count++

```
class Solution(object):
    def hammingWeight(self,n):
        """
        :type n: int
        :rtype: int
        """
        count = 0
        while n:
            res = n%2
            if res == 1:
                count +=1
            n // 2
        return count
```

方法IV：把n与1进行与运算，将得到n的最低位数字。因此可以取出最低位数，再将n右移一位。循环此步骤，直到n等于0

```
class Solution(object):
    def hammingWeight(self,n):
        """
        :type n: int
        :rtype: int
        """

        count = 0
        while n:
            count += n&1
            n >>= 1
        return count
```

2.2的幂

231. 2的幂

难度 简单 297 收藏 分享 切换为英文 接收动态 反馈

给定一个整数，编写一个函数来判断它是否是 2 的幂次方。

示例 1：

```
输入: 1
输出: true
解释: 20 = 1
```

示例 2：

```
输入: 16
输出: true
解释: 24 = 16
```

示例 3：

```
输入: 218
输出: false
```

解题思路：

2的幂有且仅有一个二进制位是1 ---> $n > 0 \ \& \ (n \ \& \ (n - 1)) == 0$

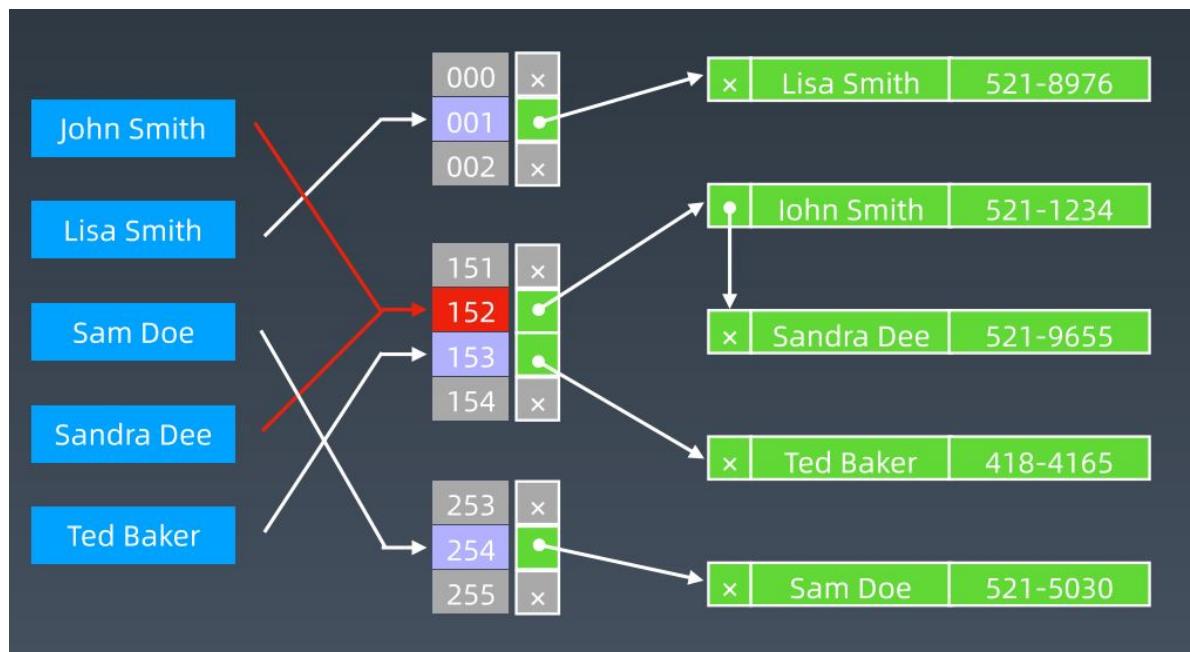
Java

```
class Solution {
    public boolean isPowerOfTwo(int n) {
        if (n == 0) return false;
        long x = (long) n;
        return (x & (x - 1)) == 0;
    }
}
```

Week09 布隆过滤器

布隆过滤器 Bloom Filter

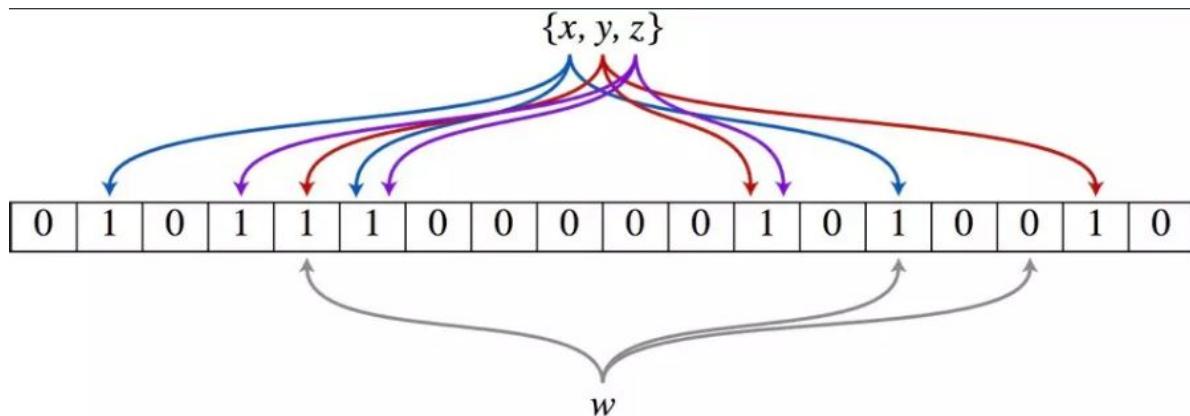
HashTable + 拉链存储重复元素



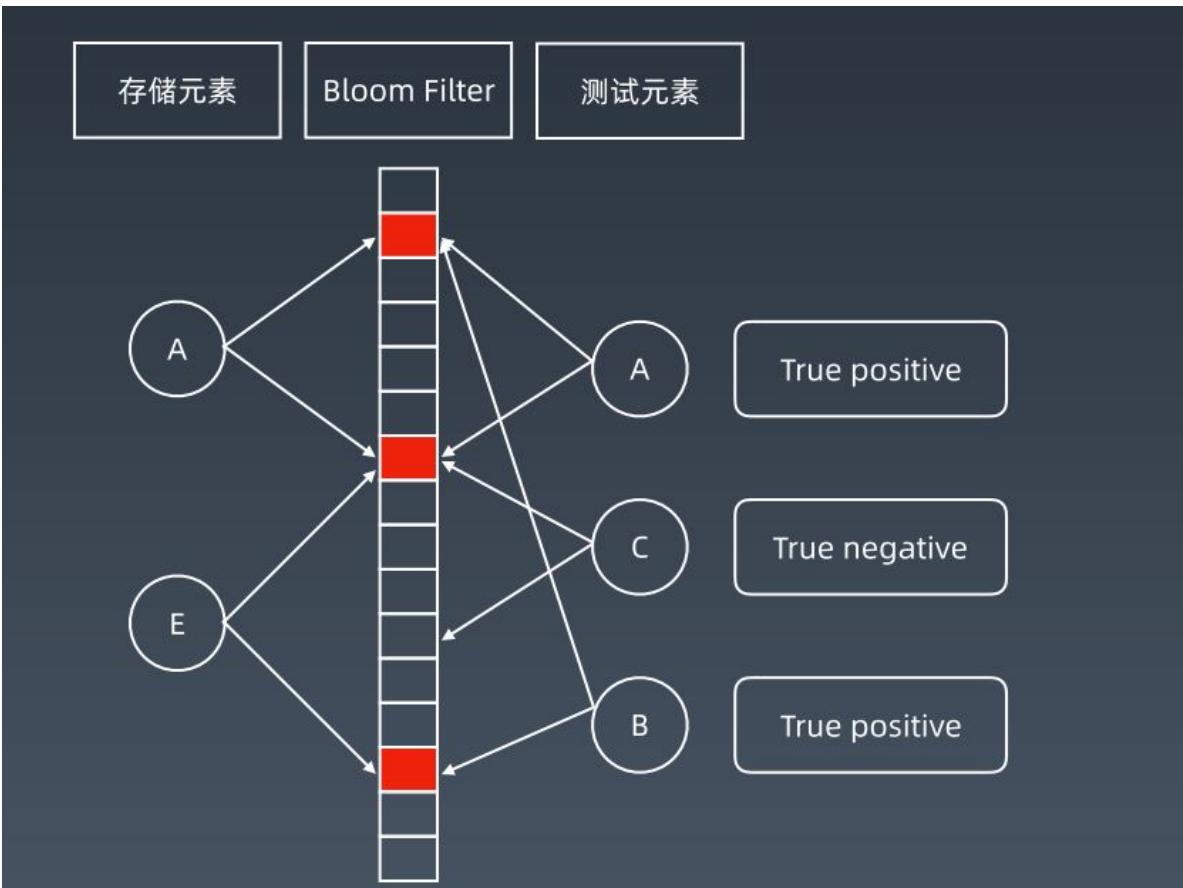
Bloom Filter vs Hash Table

一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中
优点是空间效率和查询时间都远远超过一般的算法，
缺点是有一定的误识别率和删除困难

布隆过滤器示意图



布隆过滤器的误差



A在布隆过滤器

C不在布隆过滤器中

B分配到的二进制位恰好都为1，但一开始并没有存储过B，而B所分配的二进制位恰好和之前的若干个元素所占的二进制位是有重合的，因此B在索引里面，因此对于B的判断是有误差的

元素在布隆过滤器中查，如果不在肯定是不在的；如果二进制存在，那此元素可能不在

案例

1. 比特币网络
2. 分布式系统 (Map-Reduce) — Hadoop、search engine
3. Redis 缓存
4. 垃圾邮件、评论等的过滤

Python实现

bitarray ----> 存储二进制位的数组

```
from bitarray import bitarray
import mmh3
class BloomFilter:
    def __init__(self, size, hash_num):
        self.size = size // 总共有多少元素放里面
        self.hash_num = hash_num // 一个元素放进来分成多少个二进制位
        self.bit_array = bitarray(size)
        self.bit_array.setall(0) //一开始二进制的数组为0

    def add(self, s):
        for seed in range(self.hash_num):
            result = mmh3.hash(s, seed) % self.size
            self.bit_array[result] = 1

    def lookup(self, s):
        for seed in range(self.hash_num):
            result = mmh3.hash(s, seed) % self.size
```

```

result = mmh3.hash(s, seed) % self.size
if self.bit_array[result] == 0: //确定不存在
    return "Nope"
return "Probably"

bf = BloomFilter(500000, 7)
bf.add("dantezhao")
print (bf.lookup("dantezhao"))
print (bf.lookup("yy"))

```

其他实现-Java

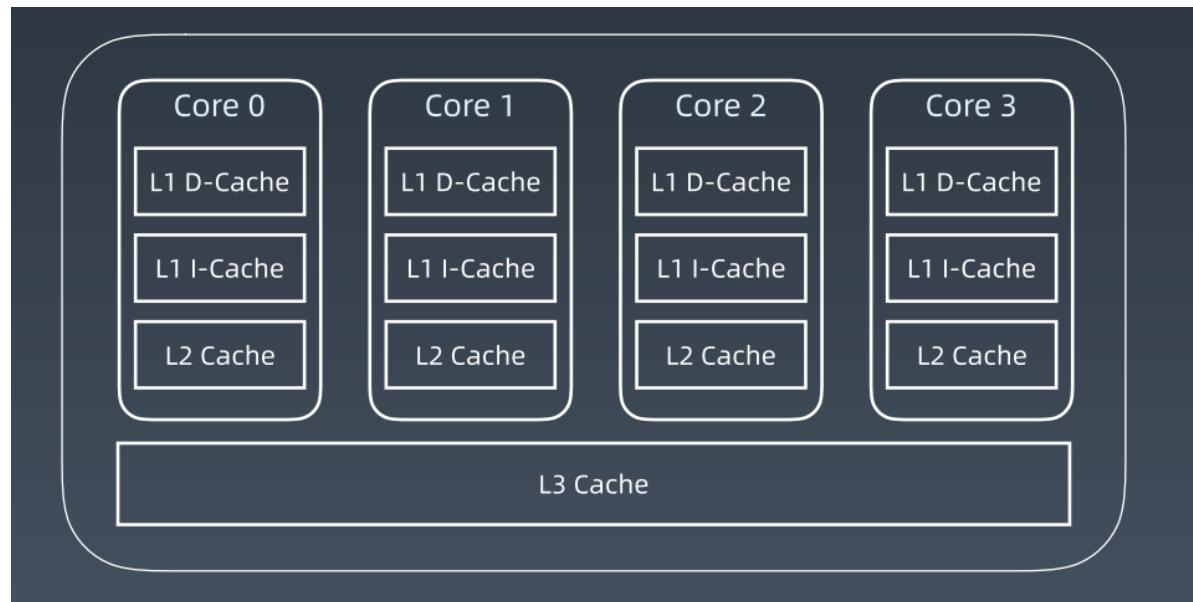
<https://github.com/lovasoa/bloomfilter/blob/master/src/main/java/BloomFilter.java>

<https://github.com/Baqend/Orestes-Bloomfilter>

Cache缓存

- 1.记忆
- 2.钱包-存储柜
- 3.代码模块

CPU Socket



LRU Cache

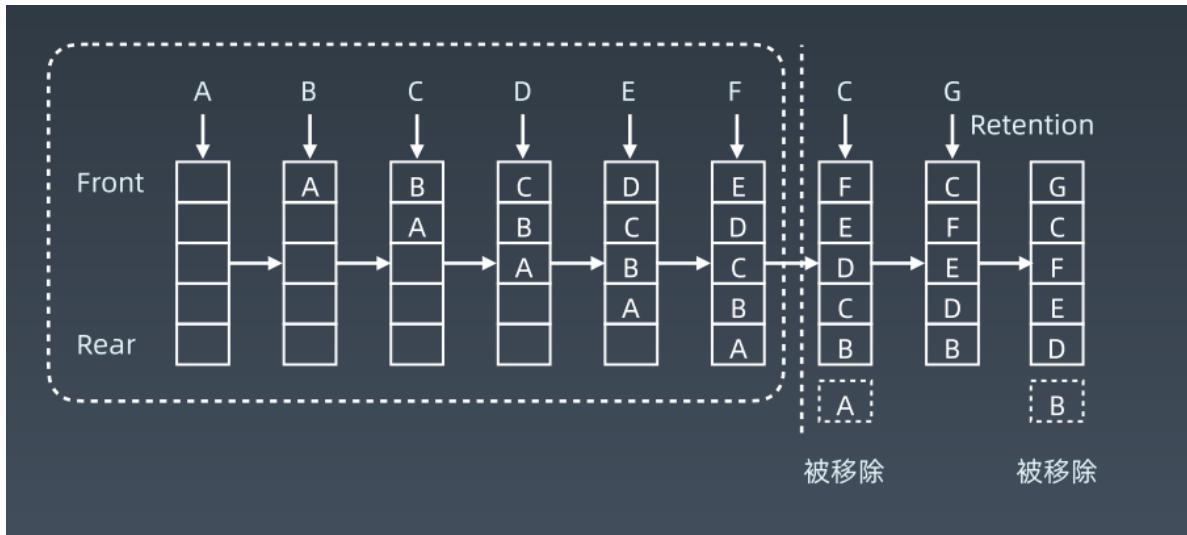
两个要素：大小、替换策略

Hash Table + Double LinkedList

O(1) 查询

O(1) 修改、更新

LRU cache工作示例



替换策略

LFU - least frequently used

LRU - least recently used

实战题目

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。

实现 LRUCache 类：

- `LRUCache(int capacity)` 以正整数作为容量 `capacity` 初始化 LRU 缓存
- `int get(int key)` 如果关键字 `key` 存在于缓存中，则返回关键字的值，否则返回 `-1`。
- `void put(int key, int value)` 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字-值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。

进阶：你是否可以在 $O(1)$ 时间复杂度内完成这两种操作？

示例：

输入

```
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
```

输出

```
[null, null, null, 1, null, -1, null, -1, 3, 4]
```

解释

```
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1);    // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
lRUCache.get(2);    // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
lRUCache.get(1);    // 返回 -1 (未找到)
```

Python实现

```
class DLinkedNode:
    def __init__(self, key=0, value=0):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUcache:
    def __init__(self, capacity: int):
        self.cache = dict()
        # 使用伪头部和伪尾部节点
        self.head = DLinkedNode()
        self.tail = DLinkedNode()
        self.head.next = self.tail
        self.tail.prev = self.head
        self.capacity = capacity
        self.size = 0

    def get(self, key: int) -> int:
        if key not in self.cache:
            return -1
        # 如果 key 存在，先通过哈希表定位，再移到头部
        node = self.cache[key]
        self.moveToHead(node)
        return node.value

    def put(self, key: int, value: int) -> None:
        if key not in self.cache:
            # 如果 key 不存在，创建一个新的节点
            node = DLinkedNode(key, value)
            # 添加进哈希表
            self.cache[key] = node
            # 添加至双向链表的头部
            self.addToHead(node)
            self.size += 1
            if self.size > self.capacity:
                # 如果超出容量，删除双向链表的尾部节点
                removed = self.removeTail()
                # 删除哈希表中对应的项
                self.cache.pop(removed.key)
                self.size -= 1
        else:
            # 如果 key 存在，先通过哈希表定位，再修改 value，并移到头部
            node = self.cache[key]
            node.value = value
            self.moveToHead(node)

    def addToHead(self, node):
        node.prev = self.head
        node.next = self.head.next
        self.head.next.prev = node
        self.head.next = node
```

```

def removeNode(self, node):
    node.prev.next = node.next
    node.next.prev = node.prev

def moveToHead(self, node):
    self.removeNode(node)
    self.addToHead(node)

def removeTail(self):
    node = self.tail.prev
    self.removeNode(node)
    return node

```

Java实现

```

public class LRUCache {
    class DLinkedNode {
        int key;
        int value;
        DLinkedNode prev;
        DLinkedNode next;
        public DLinkedNode() {}
        public DLinkedNode(int _key, int _value) {key = _key; value = _value;}
    }

    private Map<Integer, DLinkedNode> cache = new HashMap<Integer, DLinkedNode>();
    private int size;
    private int capacity;
    private DLinkedNode head, tail;

    public LRUCache(int capacity) {
        this.size = 0;
        this.capacity = capacity;
        // 使用伪头部和伪尾部节点
        head = new DLinkedNode();
        tail = new DLinkedNode();
        head.next = tail;
        tail.prev = head;
    }

    public int get(int key) {
        DLinkedNode node = cache.get(key);
        if (node == null) {
            return -1;
        }
        // 如果 key 存在，先通过哈希表定位，再移到头部
        moveToHead(node);
        return node.value;
    }

    public void put(int key, int value) {
        DLinkedNode node = cache.get(key);
        if (node == null) {
            // 如果 key 不存在，创建一个新的节点
            DLinkedNode newNode = new DLinkedNode(key, value);
            // 添加进哈希表
            cache.put(key, newNode);
        } else {
            // 如果 key 存在，先通过哈希表定位，再移到头部
            moveToHead(node);
        }
    }
}

```

```

    // 添加至双向链表的头部
    addToHead(newNode);
    +size;
    if (size > capacity) {
        // 如果超出容量，删除双向链表的尾部节点
        DLinkedNode tail = removeTail();
        // 删除哈希表中对应的项
        cache.remove(tail.key);
        --size;
    }
}
else {
    // 如果 key 存在，先通过哈希表定位，再修改 value，并移到头部
    node.value = value;
    moveToHead(node);
}
}

private void addToHead(DLinkedNode node) {
    node.prev = head;
    node.next = head.next;
    head.next.prev = node;
    head.next = node;
}

private void removeNode(DLinkedNode node) {
    node.prev.next = node.next;
    node.next.prev = node.prev;
}

private void moveToHead(DLinkedNode node) {
    removeNode(node);
    addToHead(node);
}

private DLinkedNode removeTail() {
    DLinkedNode res = tail.prev;
    removeNode(res);
    return res;
}
}

```

JAVASCRIPT实现

```

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

/**
 * @param {number} key
 * @return {number}
 */
LRUCache.prototype.get = function(key) {

```

```

let val = this.map.get(key);
if (val) {
    this.map.delete(key);
    this.map.set(key, val);
    return val;
}
return -1;
};

/**
 * @param {number} key
 * @param {number} value
 * @return {void}
 */
LRUCache.prototype.put = function(key, value) {
    if (this.map.has(key)) {
        this.map.delete(key);
    } else if (this.capacity == this.map.size) {
        let delKey = this.map.keys().next().value;
        this.map.delete(delKey);
    }
    this.map.set(key, 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)
 */

```

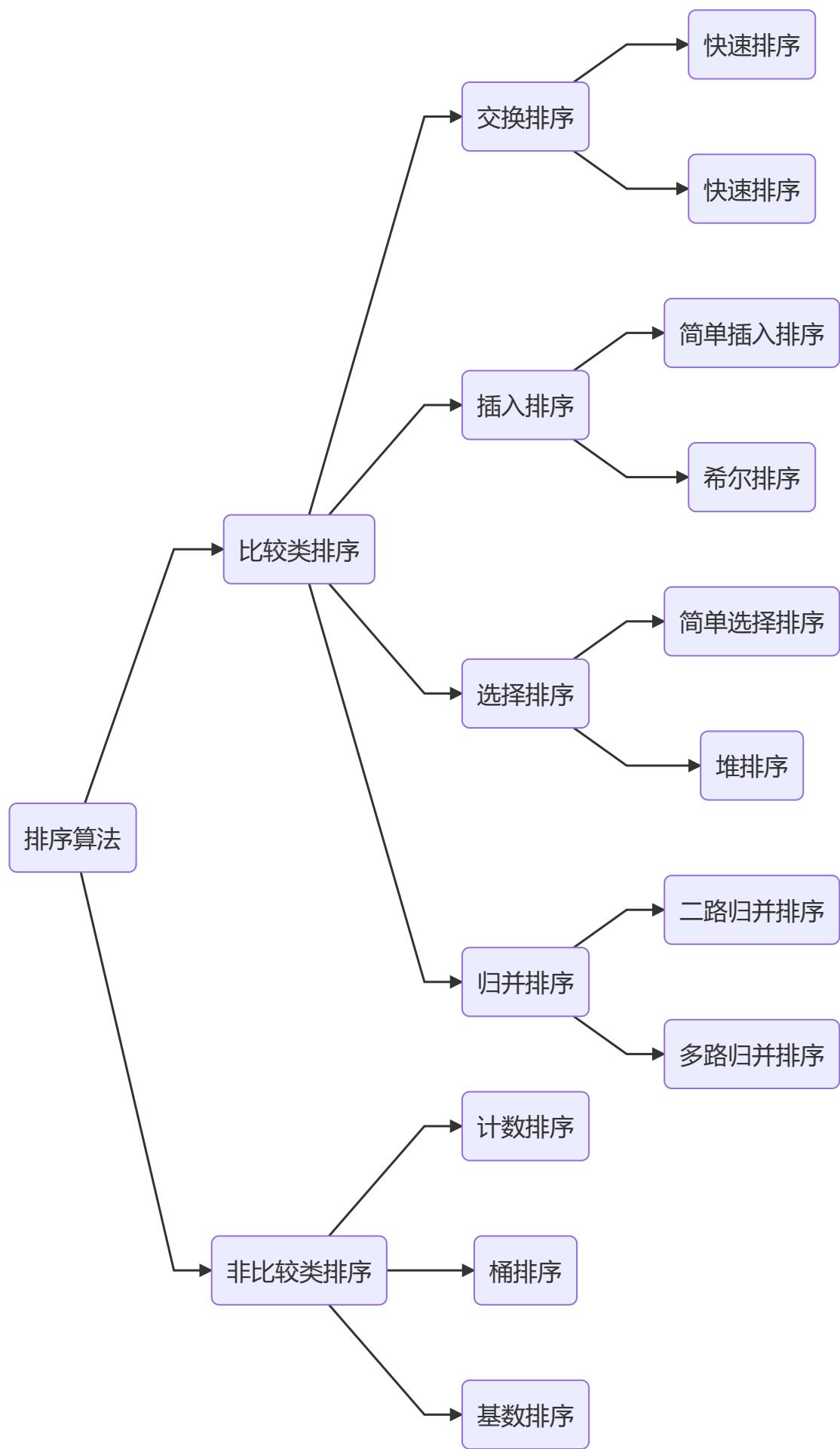
Week10-排序算法

1. 比较类排序

通过比较来决定元素间的相对次序，由于其时间复杂度不能突破 $O(n \log n)$ ，因此也称为非线性时间比较类排序

2. 非比较类排序

不通过比较来决定元素间的相对次序，它可以突破基于比较排序的时间下界，以线性时间运行，因此也称为线性时间非比较类排（一般用于整数排序）



算法时间复杂度

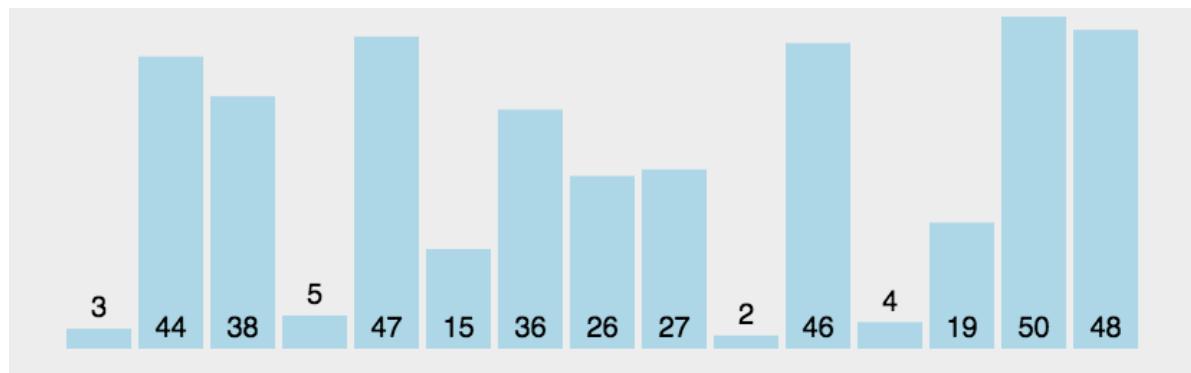
排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$	$O(n \log_2 n)$	不稳定
归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

重点：

堆排序，快速排序，归并排序

1. 冒泡排序 (Bubble Sort)

- 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
- 针对所有的元素重复以上的步骤，除了最后一个；
- 重复步骤1~3，直到排序完成。



两两交换，保证每次最大值存放到数组最后

```

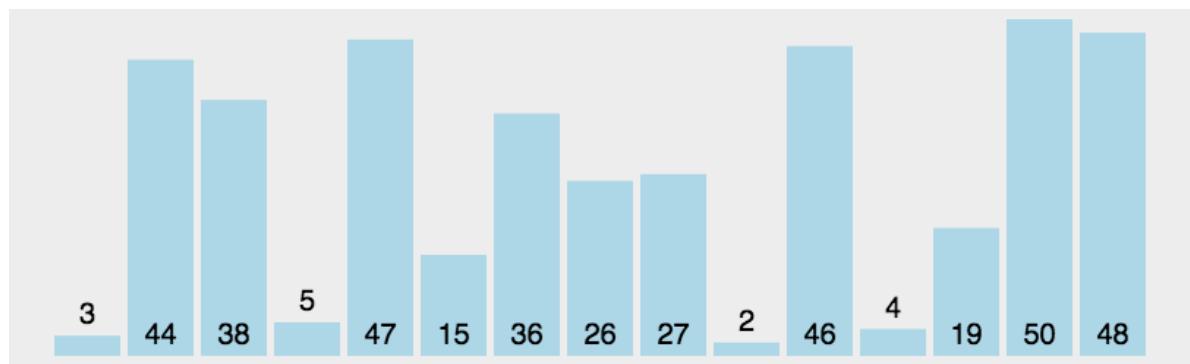
function bubbleSort(arr) {
  for(let i=0;i<arr.length-1;i++){
    for(let j=0;j<arr.length;j++){
      if(arr[j]>arr[j+1]){
        let temp = arr[j+1];
        arr[j+1] = arr[j];
        arr[j] = temp;
      }
    }
  }
  return arr;
}

```

2.选择排序 (Selection Sort)

n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果。具体算法描述如下：

- 初始状态：无序区为R[1..n]，有序区为空；
- 第i趟排序(i=1,2,3...n-1)开始时，当前有序区和无序区分别为R[1..i-1]和R(i..n)。该趟排序从当前无序区中-选出关键字最小的记录 R[k]，将它与无序区的第1个记录R交换，使R[1..i]和R[i+1..n)分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；
- n-1趟结束，数组有序化了。



解题思路：

以第i个元素为标杆，依次向后遍历找出剩下元素中最小的，放到第i的位置

```

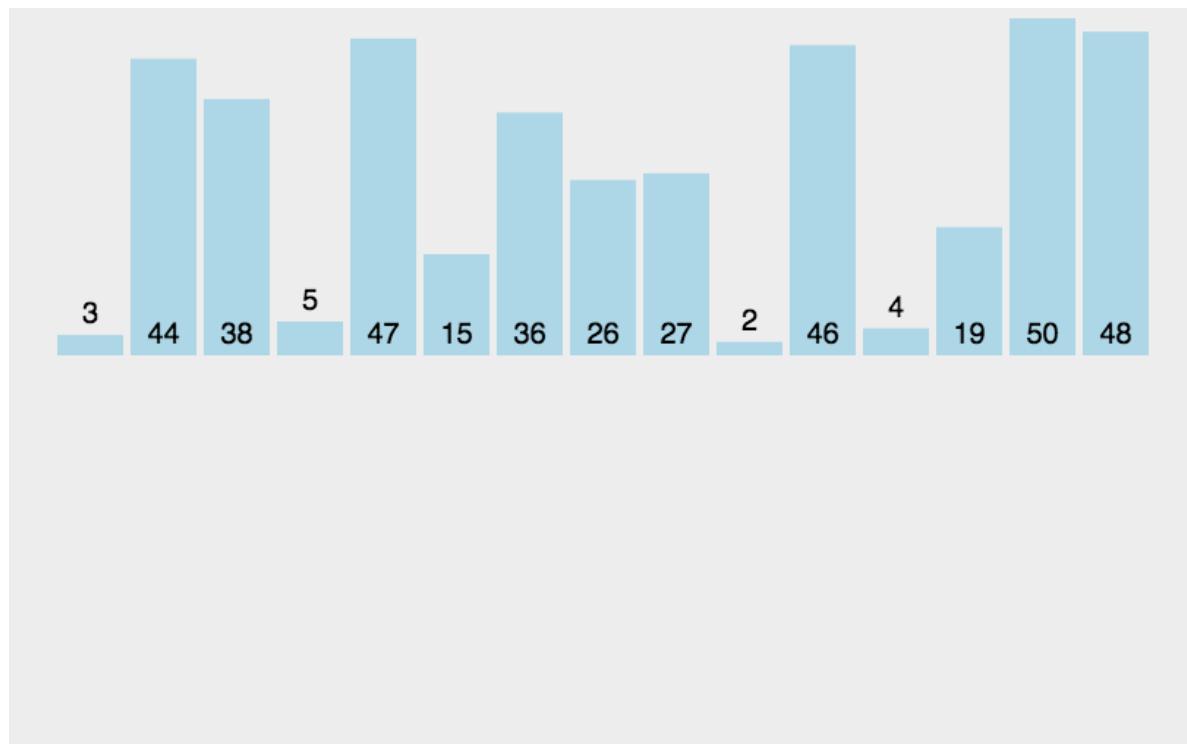
function selectSort(arr){
  //记录下当前最小值的索引值，以便进行元素替换
  for(let i=0;i<arr.length;i++){
    let minIndex = i;
    for(let j=i+1;j<arr.length;j++){
      if(arr[j]<arr[minIndex]){
        minIndex = j;
      }
    }
    let temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
  return arr;
}

```

3.插入排序(Insertion Sort)

一般来说，插入排序都采用in-place在数组上实现。具体算法描述如下：

- 从第一个元素开始，该元素可以认为已经被排序；
- 取出下一个元素，在已经排序的元素序列中从后向前扫描；
- 如果该元素（已排序）大于新元素，将该元素移到下一位置；
- 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；
- 将新元素插入到该位置后；
- 重复步骤2~5。



```
function insertSort(arr) {  
    for(let i=1;i<arr.length;i++){  
        let current = arr[i];  
        let j = i-1;  
        while(j>=0 && current < arr[j]){  
            //所有元素向后移动一位  
            arr[j+1] = arr[j];  
            j--;  
        }  
        //跳出循环，j已经-1了  
        arr[j+1] = current;  
    }  
    return arr;  
}
```

4.希尔排序 (Shell Sort)

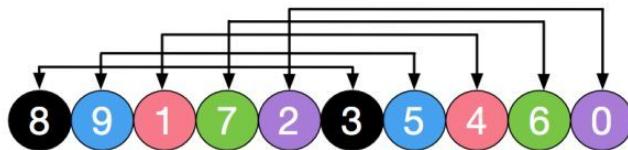
先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

- 选择一个增量序列 t_1, t_2, \dots, t_k ，其中 $t_i > t_j$, $t_k=1$ ；
- 按增量序列个数 k ，对序列进行 k 趟排序；
- 每趟排序，根据对应的增量 t_i ，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1时，整个序列作为一个表来处理，表长度即为整个序列的长度。

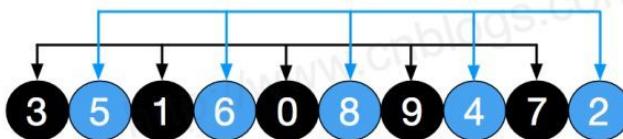
原始数组 以下数据元素颜色相同为一组



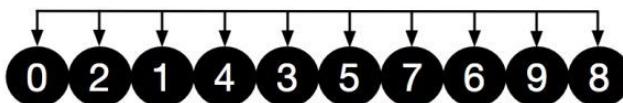
初始增量 $gap = \text{length}/2 = 5$ ，意味着整个数组被分为5组，[8,3] [9,5] [1,4] [7,6] [2,0]



对这5组分别进行直接插入排序，结果如下，可以看到，像3, 5, 6这些小元素都被调到前面了，然后缩小增量 $gap = 5/2 = 2$ ，数组被分为2组 [3,1,0,9,7] [5,6,8,4,2]



对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。再缩小增量 $gap = 2/2 = 1$ ，此时，整个数组为1组[0,2,1,4,3,5,7,6,9,8]，如下



经过上面的“宏观调控”，整个数组的有序化程度成果喜人。

此时，仅仅需要对以上数列简单微调，无需大量移动操作即可完成整个数组的排序。

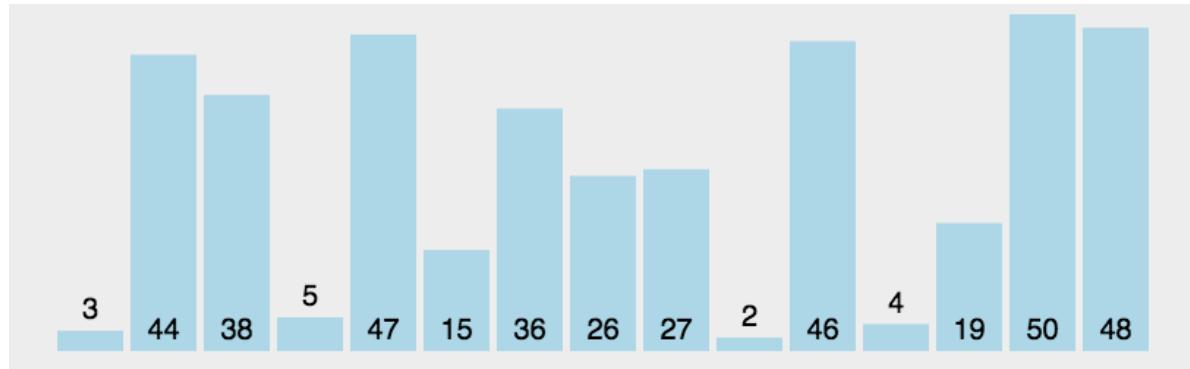


```
function shellSort(arr) {
    var len = arr.length;
    for(var gap = Math.floor(len / 2); gap > 0; gap = Math.floor(gap / 2)) {
        // 注意：这里和动图演示的不一样，动图是分组执行，实际操作是多个分组交替执行
        for(var i = gap; i < len; i++) {
            var j = i;
            var current = arr[i];
            while(j - gap >= 0 && current < arr[j - gap]) {
                arr[j] = arr[j - gap];
                j = j - gap;
            }
            arr[j] = current;
        }
    }
    return arr;
}
```

5. 快速排序 (Quick Sort)

快速排序使用分治法来把一个串 (list) 分为两个子串 (sub-lists)。具体算法描述如下：

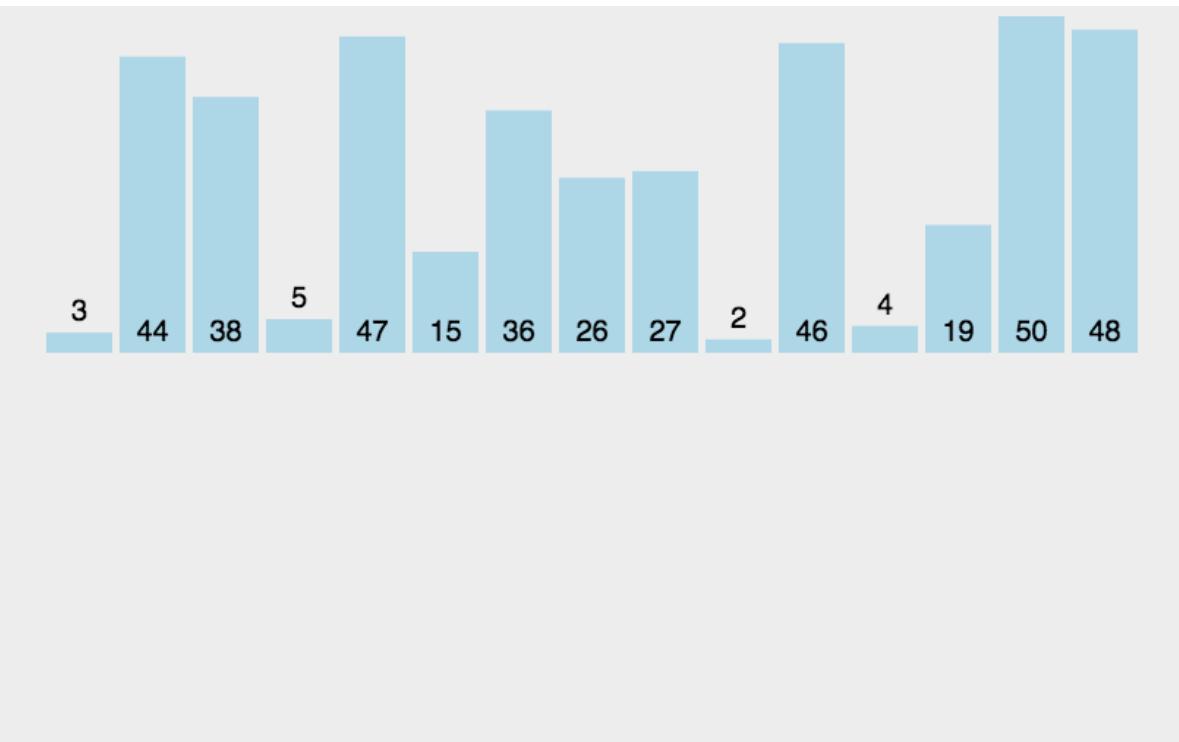
- 从数列中挑出一个元素，称为“基准” (pivot)；
- 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区 (partition) 操作；
- 递归地 (recursive) 把小于基准值元素的子数列和大于基准值元素的子数列排序。



```
function quickSort(arr) {
  if(arr.length <=1){
    return arr;
  }
  const pivot = arr[arr.length - 1];
  const leftArr = [];
  const rightArr = [];
  for(const el of arr.slice(0,arr.length - 1)){
    el < pivot ? leftArr.push(el) : rightArr.push(el);
  }
  return [...quickSort(leftArr),pivot,...quickSort(rightArr)];
}
```

6. 归并排序(Merge Sort)

- 把长度为n的输入序列分成两个长度为n/2的子序列；
- 对这两个子序列分别采用归并排序；
- 将两个排序好的子序列合并成一个最终的排序序列。



```

function mergeSort(arr) {
    if(arr.length<=1) return arr;
    let mid = Math.floor(arr.length)/2
    let leftArr = arr.slice(0,mid);
    let rightArr = arr.slice(mid,arr.length);
    return merge(mergeSort(leftArr),mergeSort(rightArr));
}
function merge(left,right){
    let result = [];
    while(left.length>0 && right.length>0){
        left[0] > right[0] ? result.push(right.shift()):result.push(left.shift());
    }
    while (left.length>0){
        result.push(left.shift());
    }
    while (right.length>0){
        result.push(right.shift());
    }
    return result;
}

```

7.堆排序 (Heap Sort)

- 将初始待排序关键字序列(R1,R2,...Rn)构建成大顶堆，此堆为初始的无序区；
- 将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,...,Rn-1)和新的有序区(Rn)，且满足R[1,2...n-1]<=R[n]；
- 由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,...,Rn-1)调整为新堆，然后再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2,...,Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为n-1，则整个排序过程完成。

```
varlen; // 因为声明的多个函数都需要数据长度，所以把len设置成为全局变量
```

```

function buildMaxHeap(arr) { // 建立大顶堆
    len = arr.length;
    for(var i = Math.floor(len/2); i >= 0; i--) {

```

```

        heapify(arr, i);
    }

}

function heapify(arr, i) { // 堆调整
    var left = 2 * i + 1,
        right = 2 * i + 2,
        largest = i;

    if(left < len && arr[left] > arr[largest]) {
        largest = left;
    }

    if(right < len && arr[right] > arr[largest]) {
        largest = right;
    }

    if(largest != i) {
        swap(arr, i, largest);
        heapify(arr, largest);
    }
}

function swap(arr, i, j) {
    var temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

function heapSort(arr) {
    buildMaxHeap(arr);

    for(var i = arr.length - 1; i > 0; i--) {
        swap(arr, 0, i);
        len--;
        heapify(arr, 0);
    }
    return arr;
}

```

实战题目

1. 数组的相对排序

1122. 数组的相对排序

难度 简单 180 收藏 分享

给你两个数组，`arr1` 和 `arr2`，

- `arr2` 中的元素各不相同
- `arr2` 中的每个元素都出现在 `arr1` 中

对 `arr1` 中的元素进行排序，使 `arr1` 中项的相对顺序和 `arr2` 中的相对顺序相同。未在 `arr2` 中出现过的元素需要按照升序放在 `arr1` 的末尾。

示例：

```
输入: arr1 = [2,3,1,3,2,4,6,7,9,2,19], arr2 = [2,1,4,3,9,6]
输出: [2,2,2,1,4,3,3,9,6,7,19]
```

解题思路：

分四种情况：

1. a, b都在arr2中
2. a在arr2中, b不在arr2中 ==> a在b的前面
3. a不在arr2中, 且b在arr2中 ==> b在a的前面
4. a, b都不在arr2中 ==> a, b升序排列

```
var relativeSortArray = function(arr1, arr2) {
    return arr1.sort((a,b)=>{
        let ia = arr2.indexOf(a);
        let ib = arr2.indexOf(b);
        if(ia === -1 && ib === -1){
            return a-b;
        }else if(ia ===-1 && ib !== -1){
            return 1;
        }else if(ia !==-1 && ib ===-1){
            return -1;
        }else{
            return ia - ib;
        }
    })
};
```

2. 有效的字母异位词

242. 有效的字母异位词

难度 简单 405 ★ ⚡ 🔍

给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。

注意：若 s 和 t 中每个字符出现的次数都相同，则称 s 和 t 互为字母异位词。

示例 1：

输入: $s = \text{"anagram"}, t = \text{"nagaram"}$

输出: true

示例 2：

输入: $s = \text{"rat"}, t = \text{"car"}$

输出: false

解题思路：

法I：暴力法(排序法)：

将字符串用split方法拆分，进行排序后再重组，判断s和t是否相等

```
var isAnagram = function(s, t) {
  s = s.split("").sort().join(" ")
  t = t.split("").sort().join(" ");
  if(s == t){
    return true;
  }else {
    return false;
  }
};
```

法II：利用哈希表分别对出现字母的频次做统计

```
var isAnagram = function(s, t) {
  if(s.length != t.length) return false;
  let count = [];
  let ca = 97;
  for(let i=0;i<=26;i++){
    count.push(0)
  }
  for(let j=0;j<s.length;j++){
    count[s.charCodeAt(j)-ca]++;
    count[t.charCodeAt(j)-ca]--;
  }
  for(let k=0;k<count.length;k++){
    if(count[k] != 0){
```

```
        return false;
    }
}
return true;
};
```

3. 合并区间

56. 合并区间

难度 中等 山 1016 ★ ⚡ 文 ⚡

以数组 intervals 表示若干个区间的集合，其中单个区间为 intervals[i] = [start_i, end_i]。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。

示例 1：

```
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].
```

示例 2：

```
输入: intervals = [[1,4],[4,5]]
输出: [[1,5]]
解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

解题思路：

1. 按照子数组中的第一个数，将子数组从小到大进行升序排列
2. 遍历每个子数组，依次判断子数组的右端点和下一个子数组的左端点的值，如果 > 则合并区间，否则就直接push到结果的数组中
3. 最后判断一下curr.length是否大于0，将剩余的curr依次push到结果中

```
var merge = function(intervals) {
  if(intervals.length<2) return intervals;

  //按照升序排列
  intervals.sort((a,b)=>a[0]-b[0]);

  let curr = intervals[0];
  let res = [];
  for(let interval of intervals){
    if(curr[1] >= interval[0]){
      curr[1] = Math.max(curr[1],interval[1]);
    } else {
      res.push(curr);
      curr = interval;
    }
  }
  if(curr.length>0) res.push(curr);
  return res;
};
```

```
    }else{
        res.push(curr);
        curr = interval;
    }
}
if(curr.length!=0){
    res.push(curr);
}
return res;
};
```

4. 翻转对

493. 翻转对

难度 困难 292 ☆ ↑ 文 ↑ ↓

给定一个数组 `nums`，如果 $i < j$ 且 $\text{nums}[i] > 2 * \text{nums}[j]$ 我们就将 (i, j) 称作一个**重要翻转对**。

你需要返回给定数组中的重要翻转对的数量。

示例 1:

输入: [1,3,2,3,1]
输出: 2

示例 2:

输入: [2,4,3,5,1]
输出: 3

解题思路:

方法I：暴力解法 $O(n^2)$

```
var reversePairs = function(nums) {
    let count = 0;
    for(let i=0;i<nums.length-1;i++){
        for(let j=i+1;j<nums.length;j++){
            if(nums[i]>2*nums[j]){
                count++;
            }
        }
    }
    return count;
};
```

方法II：归并排序

