# **前言**

* 高效学习方法：五毒神掌法

五遍刷题法，强调练习算法题要过遍数，算法题不仅是做一遍通过而已，而是要通过**刻意练习**达到对算法熟练的效果

* + 第一遍：5-10分钟如果没有思路，就去题解里面按照高赞排序找最佳题解看，记住方法和代码
  + 第二遍：自己写代码
  + 第三遍：一天之后重做
  + 第四遍：一周之后重做
  + 第五遍：面试前一周重做
* 做题误区：希望一遍理解 + 做题只做一遍
* 看题之后没思路的时候：
  + 暴力解题法
  + 分析基本情况
  + 找最近重复子问题（climbstairs问题）
* 算法到最后的3种基本情况：
  + if…else…
  + for (while) loop
  + recursion
* 时间复杂度、空间复杂度：
  + 优化代码的时间复杂度方法：空间换时间/升维
* 循环遍历基本功

Python:

for i in range(0, data.length - 1):

for j in range(i+1, data.length):

pass

c++:

for(i=0; i<data.length-1;i++)

{ For(j=i+1; j<data.length; j++)

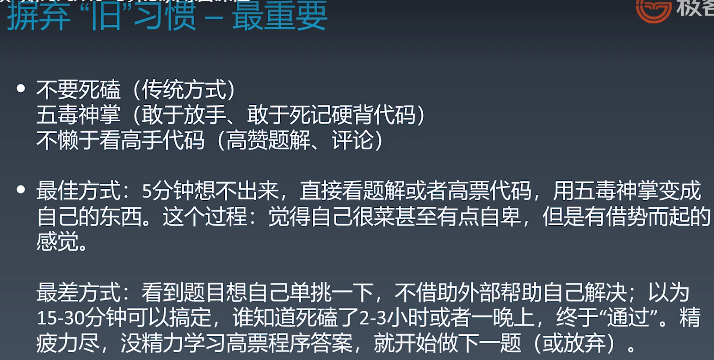
{…… } }

* 从两边向中间遍历基本功

C++: for(int i=0, int j=data.length-1; I < j; ){… }

Python: I = 0; J= data.length – 1

While i<j: i+=1; j-=1



# **Ww01 数组链表栈队列**

## **数组 (一对一)**

* 在内存中占用一块连续的地址；
* 优点：索引访问/查询O(1)
* 缺点：插入/删除需要移动元素，时间复杂度高O(n)
* 实战
  + 两数之和🡺hash map
  + **三数之和** 🡺关键是去重复
    - **排序 + 三指针** 🡺 对四数之和等是通用的方法
    - 转换为双指针 + hash\_map
  + 移动零 🡺 双指针，从前往后对比+互换
  + 盛最多水的容器🡺双指针法：从两端向中间
  + 爬楼梯
  + 删除排序数组中的重复项🡺双指针法：从前向后
  + 旋转数组🡺 对数组进行3次翻转
  + 合并2个有序数组🡺 3指针法 + 从后向前遍历
  + 加一🡺 从后往前，判断进位
* 方法总结
  + 双指针法/三指针法
  + 从前往后考虑/从后往前考虑/从两端向中间
  + 3sum问题-数组类型问题-双指针法

## **链表 (一对一)**

* 内存中不占用连续的地址
* 优点：插入/删除不用移 动元素，效率高， 时间复杂度O(1)
* 缺点：不能索引访问/查询，访问元素时间复杂度O(n)
* 分类：
* 单向/双向/循环
* 实战：
  + **反转链表** 🡺 3 pointers 迭代法，不复杂
  + 两两交互链表中的结点 🡺 创建1个结点 + 3 pointers 🡺 不要和反转链表搞混
  + **环形链表1/2** 🡺 快慢指针 / 第一次相遇后让一个指针回到head重新出发，两个指针每次都前进一步，**再次相遇**的位置就是交点位置
  + K个一组翻转链表🡺 翻转链表 + K个分组 🡺 新建头结点 + 双指针(start:K个的前一个， end：k个的后面一个)
  + 合并2个有序链表🡺创建一个头结点
  + 链表的倒数第k个值🡺快慢指针法
* 方法总结：
* **没有太多算法的东西，主要是next/pre指针操作，方法固定，熟练练习**
* 双指针/三指针/快慢指针/先后指针
* 环形链表问题的3种思路：暴力法/hash表/快慢指针/先后指针

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | prepend | append | lookup | insert | delete |
| array | O(1) | O(1) | O(1) | **O(n)** | **O(n)** |
| linklist | O(1) | O(1) | **O(n)** | O(1) | O(1) |

## **跳表**

* 基于链表，只能用于元素有序的情况
* 跳表对标的是平衡树和二分查找
* **升维 + 空间换时间**
* 插入/删除/搜索 都是O(logn)， 空间复杂度O(n)

**总结**：3种数据结构的原理和实现 + 基本操作的时间复杂度/空间复杂度 + 工程运用

## **栈 stack**

* 先进后出LIFO
* 插入和删除O(1)
* 查询O(n)，因为无序

## **队列 queue**

* 先进先出 FIFO
* 插入和删除O(1)
* 查询O(n)

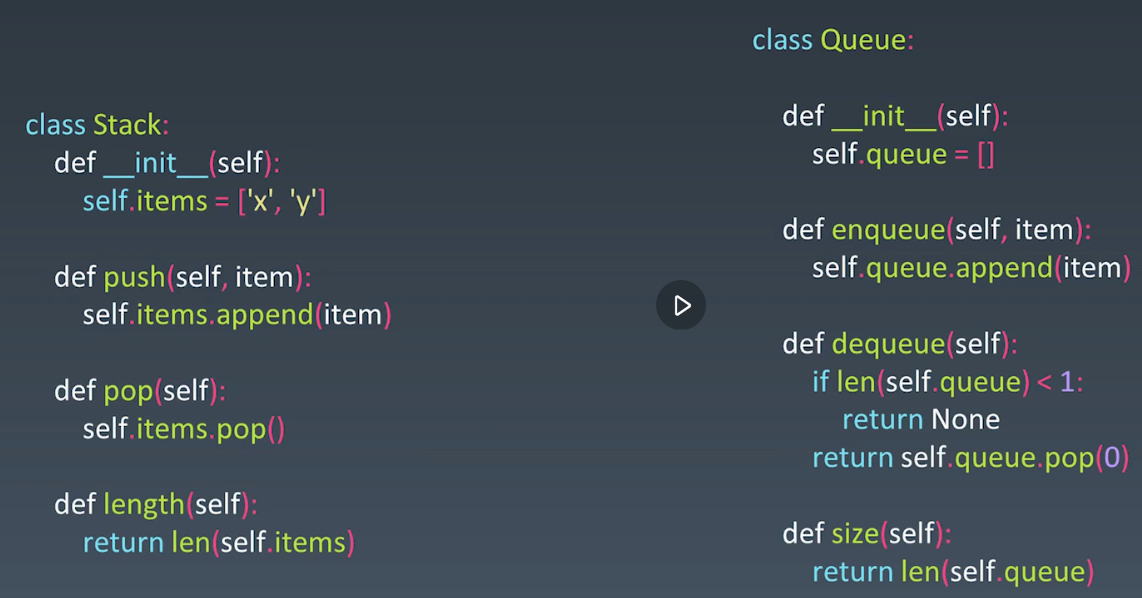
## **优先队列 priority queue**

* 插入O(1)
* 取出O(logn) ,按照元素的优先级取出

## **双端队列 deque**

* 头和尾都可以进行插入和删除
* 插入和删除O(1)
* 查询O(n)

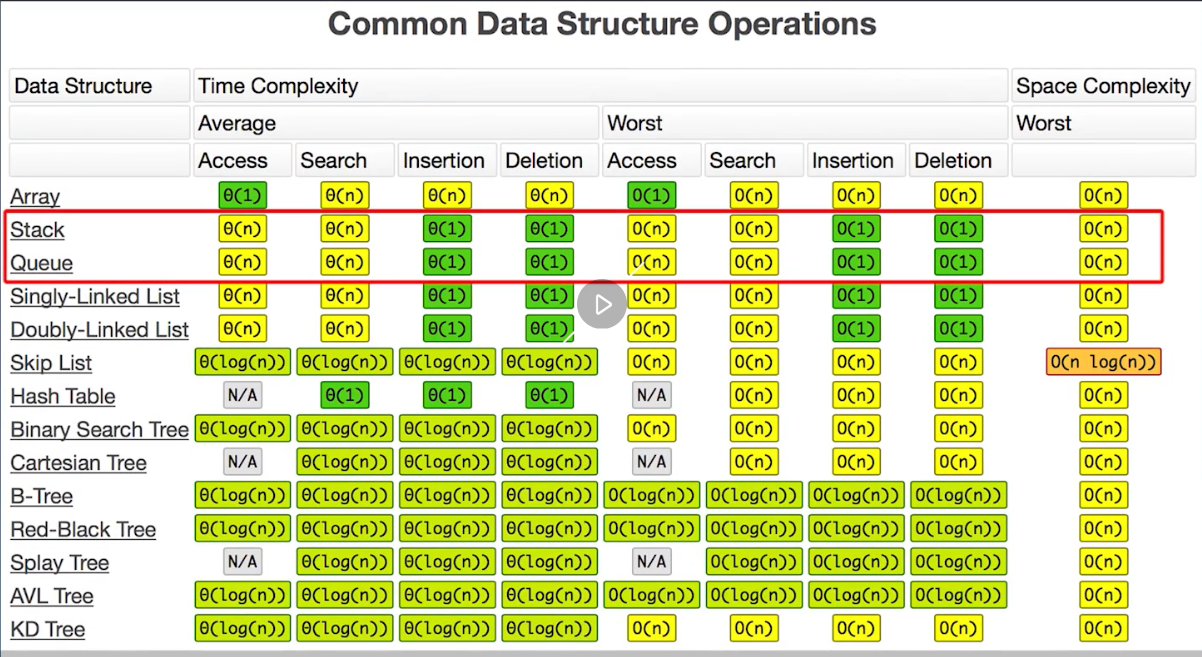
## **Stack和queue的python实现(代码模板)**



Python中的heapq + 高性能的container库: collections

**总结**：

* Stack, queue, deque的操作和复杂度
* PriorityQueue的特点和操作复杂度
* 查询stack, queue, deque, priorityQueue的系统接口的方法
* 各种数据结构的复杂度分析



## **实战**

* 什么样的问题可以用栈来解决：有最近相关性的问题=》剥洋葱型成对的问题
* 最小括号问题：
  + 暴力法：将有效括号replace为空
  + 栈
* 最小栈问题：
  + 辅助栈法（使用2个栈：一个维护出入关系， 一个维护最小栈）
* 用2个栈/队列可以解决的问题： 用栈实现队列， 用队列实现栈
* 柱状图中的最大矩形问题：
  + 暴力法
  + stack
* 滑动窗口问题： 用队列解决

# **Ww02 二叉树**

## **集合**

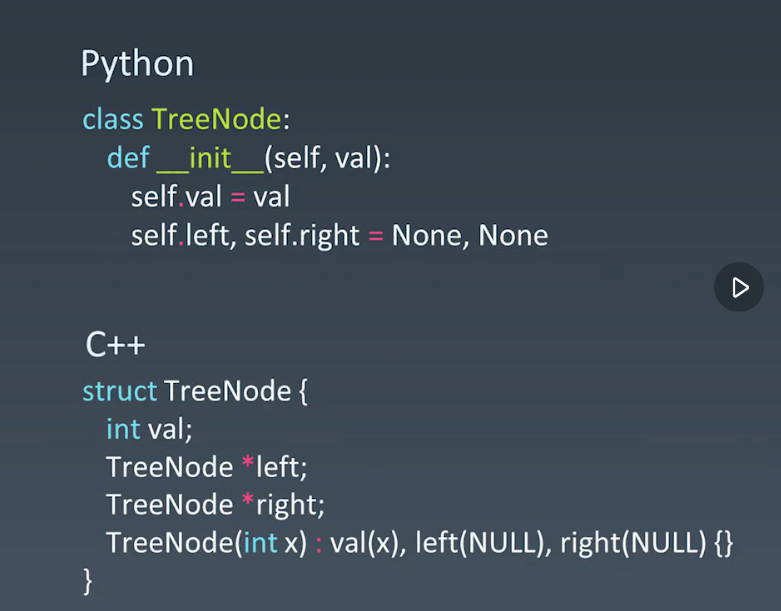
## **哈希表**

* 定义
* 通过键key直接进行访问的数据结构；
* 通过将key关键值映射到表中的一个位置（索引）来访问记录，以加快访问速度；
* 这个映射函数叫哈希函数，存放记录的数组叫哈希表；
* 哈希碰撞hash collision：不同的数据通过哈希函数得到相同的索引/位置
* 解决办法：依次放在下一个位置/在这个位置建立一个链表，存放映射到这个位置的所有数据=>访问时间复杂度从O(1)->O(n)
* 目标：设计的哈希映射函数出现最少的碰撞
* 时间复杂度：通常情况下O(1),worst下O(n):退化成链表
* 实现
* 哈希表在语言中被抽象为map和set：
* Map: key – value键值对, key不可重复，value可以重复
* Set： 不重复value的集合， value不可重复
* 在python中map就是dict， set就是set
* 实战
* 有效的字母异位词 🡺 hash\_map
* 字母异位词分组 🡺 sort() + hash\_map
* Towsum -> O(n) 🡺 hash\_map

## **映射**

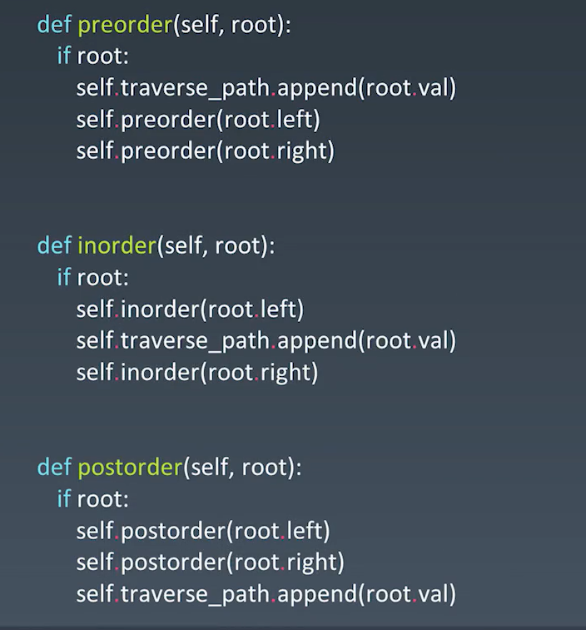
## **树(代码模板)**

* 链表是特殊化的树，树是特殊化的图；
* **定义**：树由根节点和它的子树构成 -> 递归的定义
* **特点**：一对多的二维数据结构
* **代码实现**：->形成机械化的记忆



## **二叉树 (一对多)(代码模板)**

* 二叉树遍历：
  + 前序遍历： 中左右
  + 中序遍历： 左中右
  + 后序遍历： 左右中
* 代码实现：->形成机械化的记忆



## **二叉搜索树 BST(binary search tree) (几种遍历的递归+迭代代码模板)**

* 左子树**所有结点**小于根结点，右子树**所有结点**大于根结点，并且所有的子树都是二叉搜索树 -> 递归的定义
* 按照中序遍历二叉搜索树得到的是升序数组
* 树的面试题的解法一般都是递归
* 实战：
  + **二叉树的中序遍历**
* 递归->不需要去规避或者带有色眼睛去看待递归
* 迭代：维护一个栈🡺**颜色标记法**

(1, root)入栈，开始循环

栈顶出栈，如果为空，继续下一个

如果flag为1🡺(1, left)/(0, cur)/(1, right)依次入栈

如果flag为0, 访问

* + 二叉树的前序遍历
    - 递归
    - **迭代：维护一个栈**

根结点入栈，开始循环

栈顶元素出栈，访问该元素

将其孩子结点按相反的顺序入栈

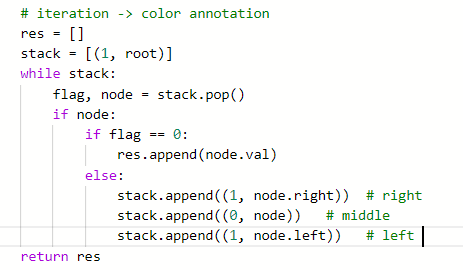
* + 二叉树的后序遍历
    - 递归
    - 迭代
      * 类似前序遍历的迭代方法，左子树先进栈，最后逆序
      * 颜色标记法
  + **总结二叉树的3种深度遍历方法**
    - 递归
    - **迭代：颜色标记法🡺 记住代码模板**

(1, root)入栈，开始循环

栈顶出栈，非空时：

如果flag为1：(1, left)/(0, cur)/(1, right)依次入栈 🡺 和遍历顺序相反

如果flag为0, 访问



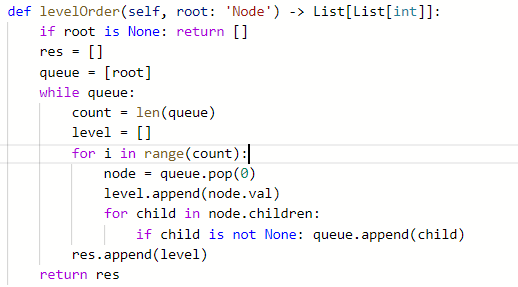
* + N叉树的层序遍历(每一层用一个list表示)
    - 维护一个队列

根节点入队列，开始循环

统计当前队列长度，开始循环🡺 level

访问结点，结点的孩子结点依次入队列

遍历一层结束Level表示这一层结果

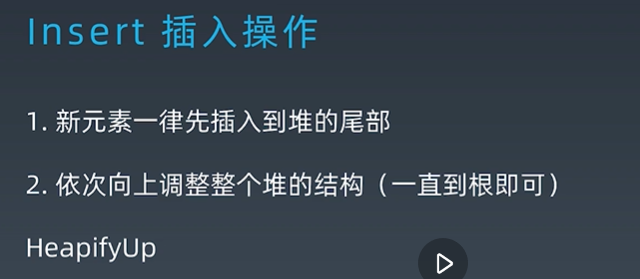


## **堆 heap**

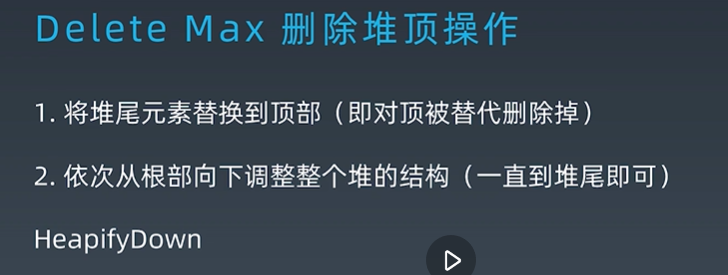
* 可以迅速找到一堆数种最大或者最小值的数据结构；大顶堆/小顶堆
* 常见的由二叉堆，斐波那契堆
* 常见API：
  + Find\_max: O(1)
  + Delete-max: O(logn)
  + Insert(create):O(logn)

## **二叉堆 (一种特殊的二叉树)**

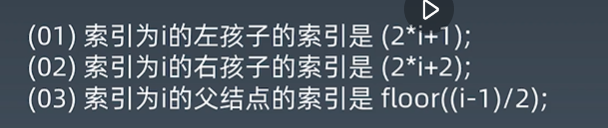
* 定义:通过完全二叉树实现
* 二叉堆（大顶）满足：
  + 是一棵完全树
  + **树中任意结点的值总是大于等于其子结点的值**
* 基本操作以及实现细节, 时间复杂度
* **用数组实现**
* 返回最大值
* 插入：插入到堆尾；从堆尾出发依次向上调整堆



* 删除最大值：用堆尾替换根节点；从根节点出发向下依次调整堆，直到堆尾即可



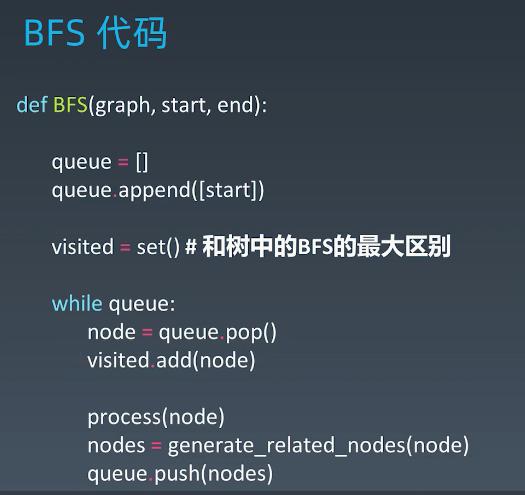
* 满二叉树：深度为k，且含有2^k – 1个结点的二叉树，即除了叶节点之外，所有的结点都有2个子节点；
* **完全二叉树**:
  + 满二叉树的特例；
  + 树的深度为h，除第h层外，其他各层的结点都有2个子结点，第h层所有结点都连续集中在最左边

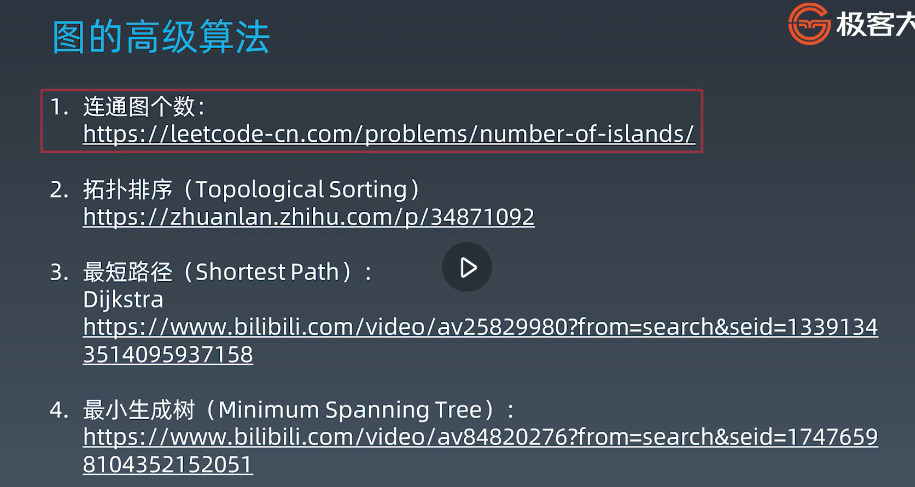


* 实战：
  + 最小的k个数
    - Sort O(nlogn)
    - 维护一个heap O(klogn) ?
    - Quick-sort
  + 滑动窗口最大值
    - 双端队列
    - Heap
  + **Top k**： 结合hash\_map 和heap **很重要**

## **图 （多对多）**

* 图的属性，分类和相关算法
  + 点：度，出度，入度，连通性
  + 边：权重
  + 有向图，无向图
* 相关算法：
  + DFS深度优先搜索 (代码模板一定要记下来)
  + BFS广度优先搜索(代码模板一定要记下来)
  + 高级算法





# **Ww03 递归分治回溯**

## **泛型递归(代码模板)**

* 递归的实现+特性
  + 递归本质是循环，是不断调用函数本身的循环，向下进入到下一层，向上回到原来一层；
  + 每一层的环境和数据都是一份拷贝（是局部的，不会互相影响），主角穿越到不同层中（函数的参数，会发生变化）
* 递归的3个思维要点
  + 不要人肉递归（最大误区）
  + 找到最近最简方法，将其拆解成可重复解决子问题（最小重复子问题）
  + 数学归纳法
* 递归最典型 的例子：Fibonacci数列和n!
* Python代码模板

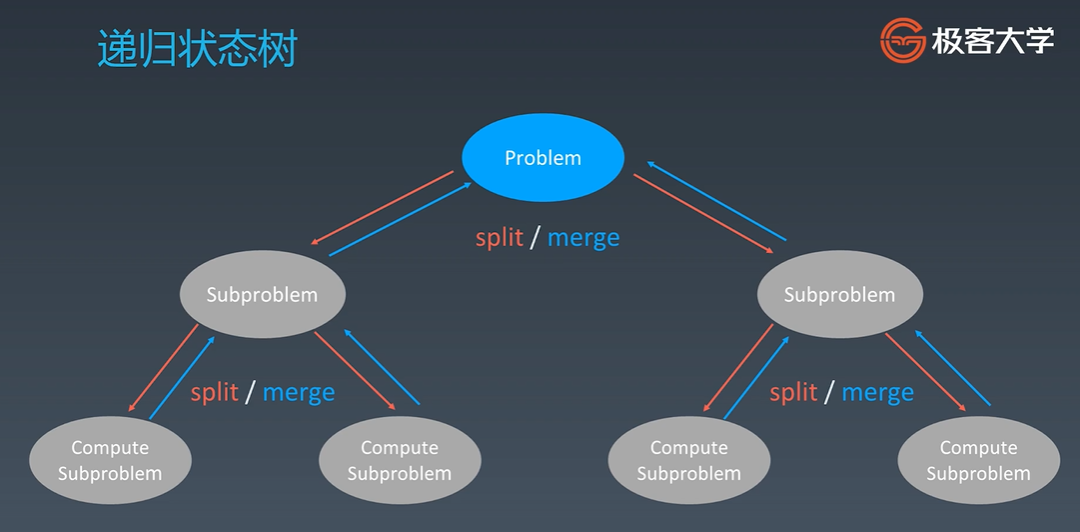


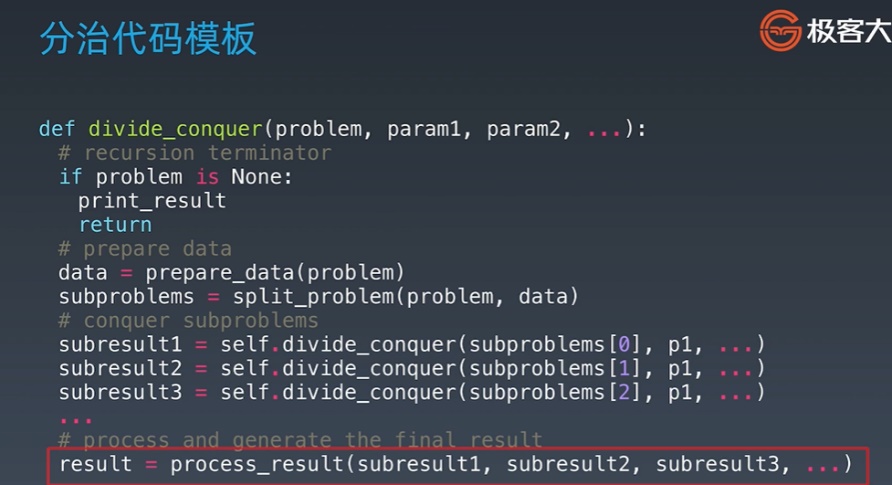
* 实战
* 爬楼梯问题：
  + 根据给出条件反向推导出递归公式：只能走1步或2步
  + 当n时，如果站在n-1上，走一步到n；如果站在n-2上，直接走2步到n；
  + F(n) = f(n-1) + f(n-2)
  + 考虑会不会有漏/重复
* 括号生成问题：
  + 可以先把所有情况生成出来，然后输出之前筛选有效的括号
  + 生成的时候就加判断条件：1)left未超标就可以放左括号 2)left >= right才可以放右括号
  + 括号生成问题给后面很多问题提供了一个思路，就是给出n个格子，这个格子里面可以放左括号，也可以放右括号
* 判断一个数组是否是二叉搜索树的后序遍历🡺 递归 + 分治
* 二叉树的最近公共祖先🡺分治
* 根据前序遍历和中序遍历生成一棵二叉树 🡺 分治

## **树的递归**

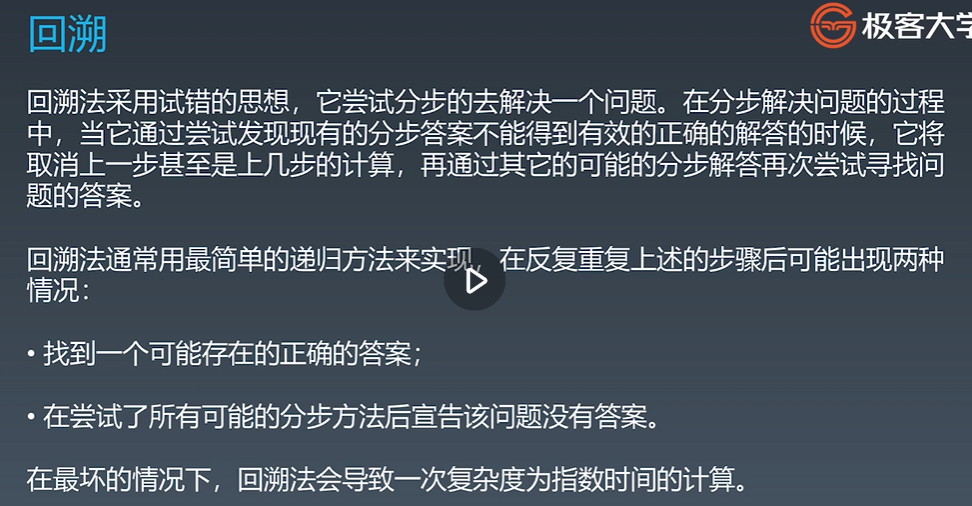
* 判断二叉搜索树问题
  + 递归（不能只递归比较左右结点，要递归左子树和右子树）
  + 中序遍历是递增数列，遍历过程中判断当前元素是否大于等于前一个元素即可

## **分治(divide and conquer)(代码模板)**

* 本质上是递归，特殊的递归
* 遇到一个问题找它的重复性:
  + 最优重复性->动态规划；
  + 最近重复性->一般的递归，分治，回溯
* 分治算法的思想
* 分治代码模板



## **回溯(backtracking)**



* 最典型的应用是处理八皇后问题和数组问题
* 实战
  + Pow(x, n)
    - 暴力法：for循环 O(n)
    - 分治法： O(logN)
  + 子集问题：
    - 分治法
    - 迭代法\*
  + 众数
  + 电话号码的字母组合-分治
  + N皇后问题-分治

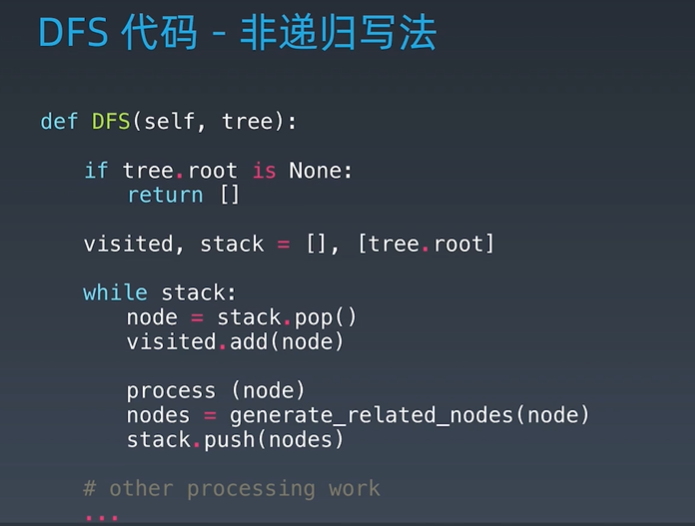
# **Ww04 深度/广度优先搜索/贪心算法/二分查找**

## **深度优先搜索和广度优先搜索(代码模板)**

* 搜索的本质就是把所有的结点都遍历一次且只遍历一次，按照遍历顺序不同分为深度优先搜索和广度优先搜索，实现方式有递归 和 非递归两种方式：
  + 递归： 代码比较简单
  + 非递归：栈/队列 + for loop
* 深度优先搜索：
  + 递归
  + 非递归-手动维护一个栈
* 广度优先搜索：非递归-手动维护一个队列

**程序模板要写的滚瓜烂熟**

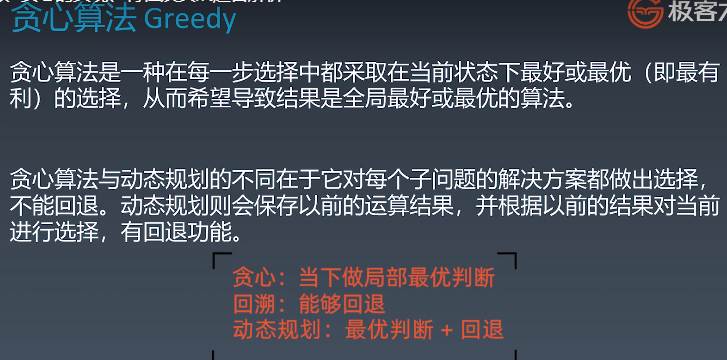
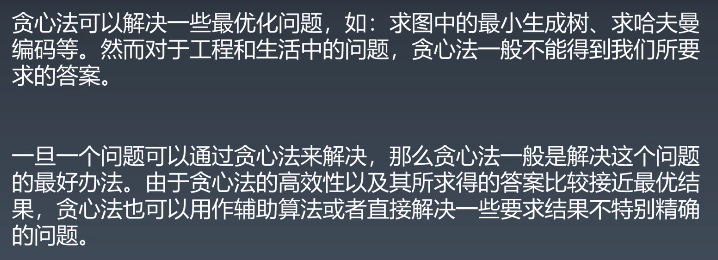
* 实战：
  + 找重复性 -最小重复子问题
  + 二叉树的层序遍历

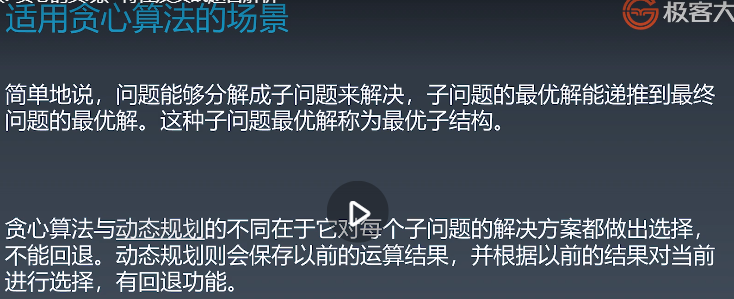
 



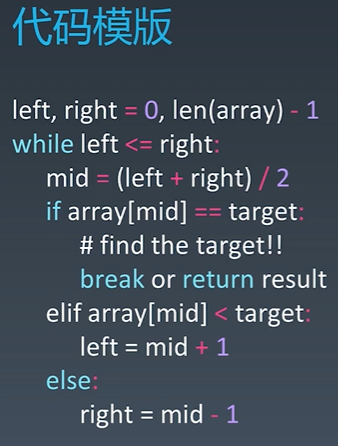
## **贪心算法**

* 每一步都采取局部最优，以期望达到全局最优；
* 适用于贪心法的情况：**有特殊性**/能够证明用最简单粗暴的贪心法可以得到最优解，局部最优可以导致全局最优
* 贪心算法的**难点**是怎样证明贪心算法是最优的
* 贪心算法可以从前面开始，也可以从局部开始贪心，也可以从后往前



## **二分查找 (代码模板)**

* 使用二分查找的前提是：
  + 数组具有单调性monotonicity
  + 数组有界 bounded
  + 能够通过索引进行访问 index accessible
* 每次找数组的中值，通过对比target和中值大小缩小搜索范围，从而减小时间复杂度 **思维逻辑 代码能力 debug能力**

# **Ww05-06动态规划(模板)**

## **递归感触**

* 拒绝人肉递归：低效+很累
* 找到最近最简方法，将其拆解成可重复解决的问题
* 数学归纳思维（抵制人肉递归的诱惑）
* 画递归状态树\*\*

## **动态规划得实现及关键点 dynamic programming**

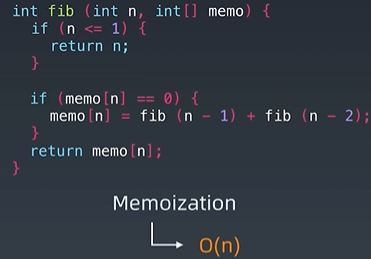
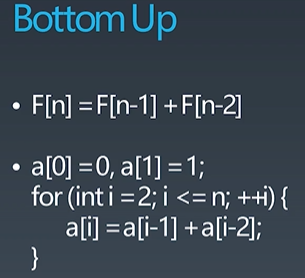
* **理解**：divide & conquer + optimal substructure分治 + 最优子结构
* **关键点**（3句话）：
  + 动态规划和递归或者分治没有根本上的区别（关键看有无最优子结构）
  + 共性：找到重复子问题
  + 差异性：**最优子结构**，中途可以淘汰次优解

\*所谓最优子结构：最小值是..最多是..最快…

* **对时间复杂度的影响**
  + 傻递归：一般指数级
  + 动态规划：一般是多项式级别甚至线性级别
* **编码**(2种思维方式)
  + 自顶向下：递归 + 记忆化搜索 recursive + memorize
  + 自下向上递推bottom up： 递推 + for loop

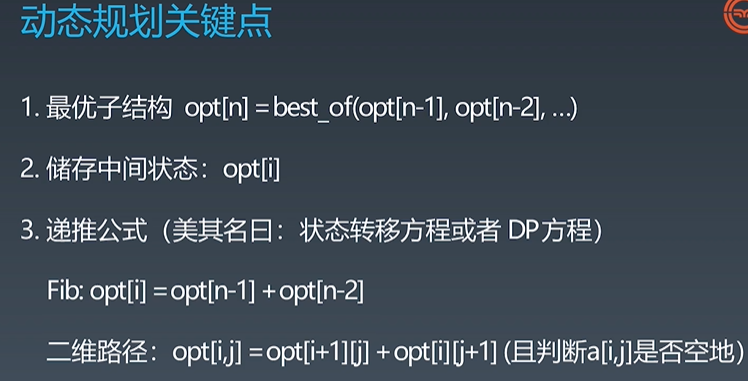
## **Fibonacci数列（一维）**

* 傻递归：时间复杂度是指数级的 2^n
* 动态规划：线性时间复杂度O(n )

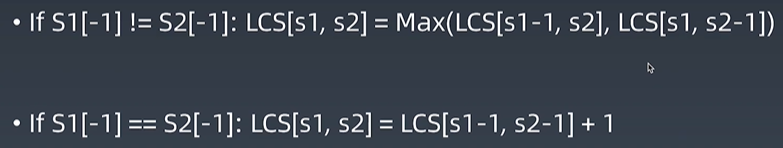
## **Count the paths（二维）**

* 最小重复子问题：右点到终点 / 下点到终点
* 最优子结构：
  + paths(start, end) = paths[A, end]+ paths[B, end]
  + 不同问题种的最优子结构不同，可能是取最大值，可能是取最小值，这里是二者之和
* **关键**是找对**递推公式** ->使用数学归纳法，从最简单的情况出发（递归终止条件，递推初始值）
* **建议使用递推的思维方式**



## **最长公共子序列（2个字符串）**

* 2个字符串构建一个二维数组 -> 思维转移
* 递推公式



## **动态规划思维小结**

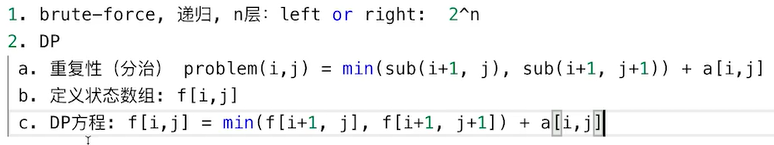
* 打破自己的思维惯性，形成机器思维->找重复性（机器只会if..else/for loop/recursive）
* 理解复杂逻辑的关键

## **爬楼梯问题的进阶**

* 初始问题1,2 🡺F(n) = F(n-1) + F(n-2)
* 1,2,3
* **相邻两步的步伐不能相同**

## **三角形最小路径和**

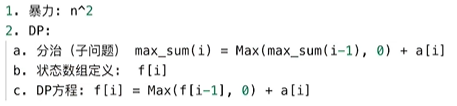
* Brute-force，递归， n层：left or right: 2^n
* DP
* 重复性（分治）
* 定义状态数组（OPT应该是什么样的）
* DP方程



Python中add LRU\_cache ?

## **子序列的最大和**

* 提高自己认知能力的时候：找到自相似性办法，即重复性办法，能够化繁为简，保证逻辑上简洁+严谨可证明的；
* **认知误区**：大概看一下，好像可以是这样的，数学上不严谨，逻辑上没有自相似性；
* 暴力： n^2
* DP
  + 分治(子问题)
  + 最大子序和 = 当前元素自身最大 , 或者包含之前后最大
  + 状态数组定义
  + DP方程

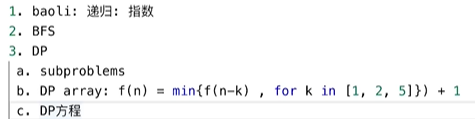


## **子序列的最大乘积**

把正的最大值保存下来，把负的最大值也保存下来

## **零钱兑换**

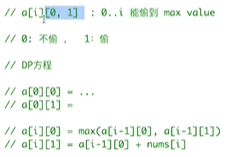
* 暴力：递归，指数级
* BFS
* DP

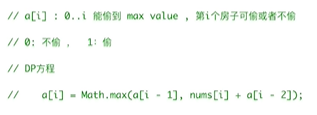


## **子序列的最大集**

## **打家结舍**

2种方法

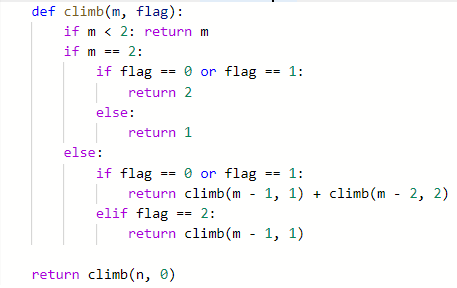




## **打家劫舍2**

## **总结动态规划**

* 对这类题有3种方法
  + 递归 O(2^n)
  + 动态规划
    - top-down: 递归 + memorize O(n^2) O(n^2)
      * 递归终止条件：
      * Memory[i][j] is not None
      * Index constraints
    - down-top: 递推 + loop
  + 三种方法依次从时间和空间上进行了改进
  + 一种trick：对于某些有约束问题可以增加一个状态标识符, 比如相邻2个不能同时偷
* 习题总结
  + **分类**
    - 一维数组的dp
    - 二维数组的dp
    - 字符串的dp -> 最长公共子序列
  + **Fobinacci数列**
    - Dp[i] = dp[I - 1] + dp[I - 2]
  + **零钱兑换(3种)**
    - 很多变换/理解的基石
    - 所有硬币组合🡺爬楼梯问题
      * dp[i] = dp[I - coin] for coin in coins
      * 注意dp数组初始化 + 遍历嵌套顺序
    - 最少硬币个数(可以重复多次)🡺背包问题
      * dp[i] = min(dp[I – coin] + 1 for coin in coins)
    - 最少硬币个数(每种硬币最多n次)
  + **爬楼梯问题**(2种)
    - 相邻两步可以重复🡺 Fibonacci数列
    - 不可以连续走两个两步
      * 增加一个标志位: 0, 1, 2, 根据标志位确定下一步分治情况



* + **路径计数**(3种)
    - 从网格左上角到右下角
    - 无障碍物:

dp[i][j] = dp[i][j+1] + dp[i+1][j]

* + - 有障碍物

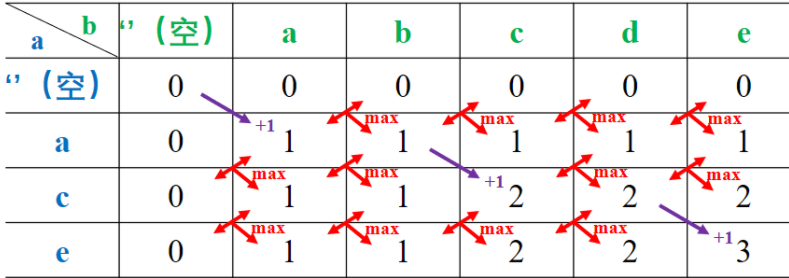
增加一个判断条件：当当前位置没有障碍物的时候执行dp方程

两个边界的初始化和判断

* + - 最小路径和: 每个位置有个权重

dp[i][j] = grid[i][j] + min(grid[i][j+1], grid[i+1][j])

* + **最长公共子序列**🡺转换为二维数组dp/和路径问题有点像
    - dp[i][j] = dp[i-1][j-1] + 1 if x[i] == y[j]
    - dp[i][j] = max(dp[i][j-1], dp[i-1][j]) if x[i] != y[j]



* + 背包问题
    - 01背包问题
    - 完全背包问题
  + **三角形最小路径和**
    - F[i][j]—状态数组：表示从i行j列出发的三角形最小路径和
    - F[i][j] = min(f[i+1][j], f[i+1][j+1]) + triangle[i][j] —状态方程,从底层向上
  + **连续子序列问题**
    - 最大和
      * Dp[i]表示以索引i为终点的子序列的连续子序列的和
      * Dp[i] = max(nums[i], dp[i-1] + nums[i]) 要么是当前元素，要么是之前序列+当前元素
      * Max(dp[i]) 选最大返回
    - 最大乘积
      * Pre\_max/ pre\_min/ res 之前序列的最大值/最小值/最终输出值
      * Cur\_max = max(pre\_max \* nums[i], pre\_min\*nums[i], nums[i])
      * Cur\_min = min(pre\_max \* nums[i], pre\_min\*nums[i], nums[i])
      * Res = cur\_max
      * Pre\_max, pre\_min = cur\_max, cur\_min

🡺保存之前乘积的最大值和最小值

* + **打家劫舍**(2种)
    - 不能偷相邻2个房子
      * 思路1：增加一个状态0/1表示当前房子偷还是不偷,每次更新2个状态

dp[i]表示从第0个房子到i能偷到的最大值

dp[i][0] = max(dp[i-1][1], dp[i-1][0])

dp[i][1] = dp[i-1][0] + nums[i]

最后返回max(dp[n-1][0], dp[n-1][1])

* + - * 思路2：不增加状态

dp[i] = max(dp[i-1] + 0, dp[i-2] + nums[i])

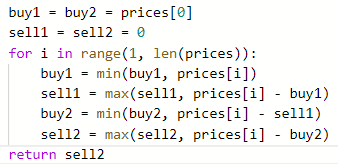
* + - 在1基础上，所有房子首尾相连
      * 转换为2个第一种情况：不偷最后一个 / 不偷第一个， 取二者最大值

Max(helper(nums[1:]), helper(nums[:-1]))

* + **买卖股票的最佳时机**
    - 最多买卖一次的最大利润
      * 遍历一遍，记录到目前位置的最小价格和最大利润
    - 可以多次买卖的最大利润
      * 思路: 只要当天比前一天高，就可以统计到最大利润中
      * 含有冷冻期1天
      * 含有手续费
    - 最多可以买卖两次的最大利润(如果定义状态数组是3维)
      * 4个状态对应的dp方程(结合情况1的理解)

sellx = max(sellx, prices[i] - buyx)

buyx = min(buyx, prices[i] – sellx\_pre) / 尤其注意buy2的更新



* + - 最多可以完成k笔交易的最大利润…
  + 括号生成问题

# **Ww07高级搜索**

* 剪枝
* 双向BFS
* 启发式搜索
* 初级搜索总结
  + 朴素搜索
  + 优化方式：不重复(fibonacci)、剪枝(生成括号问题)
  + 搜索方向：
    - BFS: 广度优先搜索
    - DFS: 深度优先搜索

🡺 双向搜索/启发式搜索

# **Ww08字典树/并查集/红黑树/AVL树/位运算**

* 字典树/Trie树/前缀树
  + 数据结构：树形结构
  + 核心思想：

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

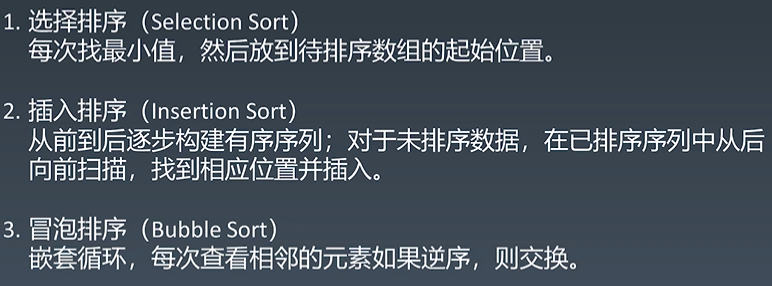
* + 基本特性
    - 根节点不包含字符，除根节点之外每个结点只包含一个字符
    - 从根节点到某一结点，路径上经过的字符连接起来是该结点对应的字符串
    - 每个结点的所有子结点包含的字符都不相同
  + 应用：统计和排序大量的字符串/搜索引擎系统用于文本词频统计
  + 优点：最大限度地减少无谓的字符串比较
* 并查集
* 红黑树
* AVL树
* 位运算

# **Ww09排序**

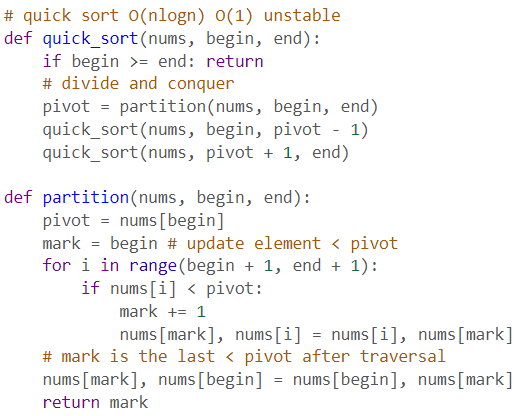
**分类**

<https://www.cnblogs.com/onepixel/p/7674659.html>

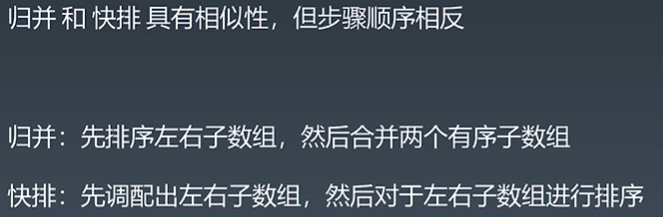
* 比较类排序
  + 面试重点：nlogn的3个排序：快速排序/归并排序/堆排序
  + 简单的初级排序O(n^2)
    - 选择排序：每遍历一次找剩余元素最小值，放到合适位置
    - 插入排序：遍历未排序序列，依次将未排序元素插入到排好序列中，方法是从后向前依次比较
    - 冒泡排序：两两比较，遍历一次之后最大元素沉底



* + 高级排序 O(nlogn)
    - **快速排序 quick sort**
      * 基于分治思想
      * 确定基准点pivot, 使得左侧的点小于pivot，右侧的点大于pivot，然后对左右侧分别进行快速排序

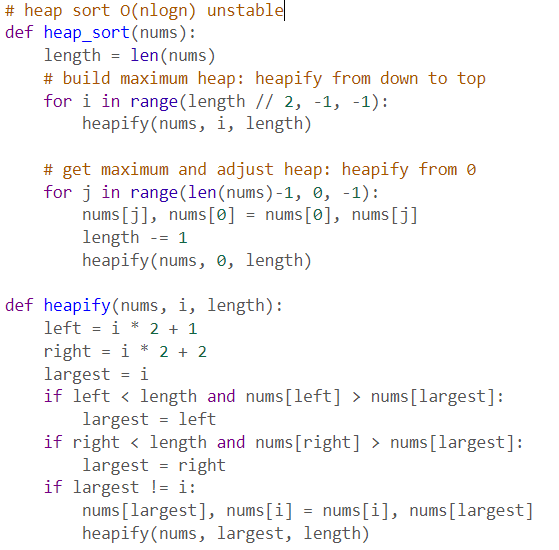


* + - 归并排序merge sort 🡺 基于分治
      * 把长度为n的序列分成两个长度为n/2的子序列
      * 对两个子序列分别采用归并排序
      * 将两个排好序的子序列合并成一个最终的排序序列

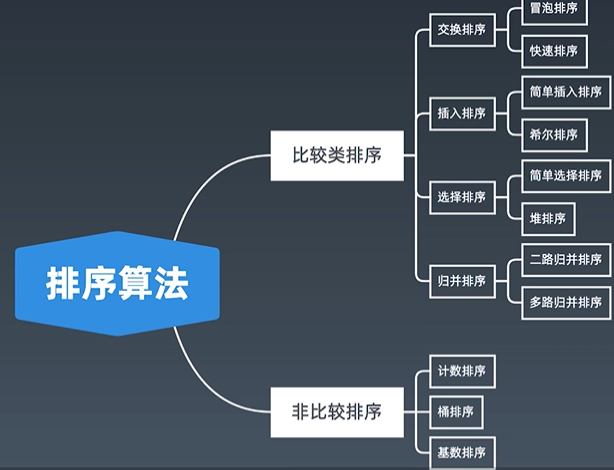
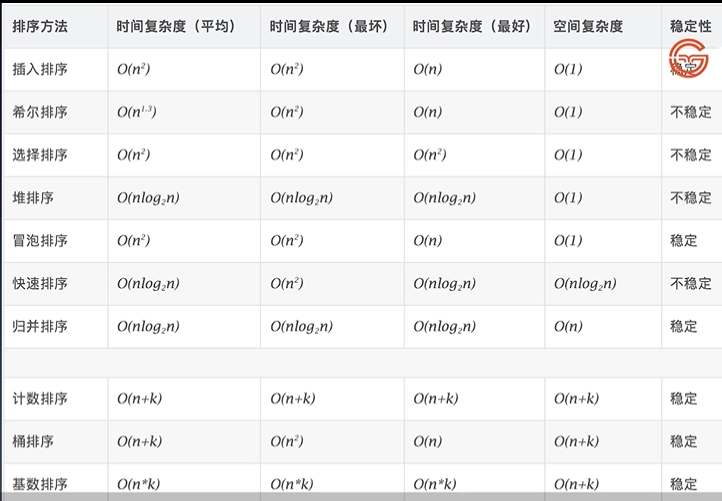




* + - 堆排序 heap sort O(nlogn) 🡺 递归
      * 构造大顶堆: 自第一个非叶子节点开始向上
        + 每次调用heapify，将从该结点出发的子堆调整为大顶堆
        + 从底向上依次进行
      * 将自下向上遍历: 堆顶元素和堆底元素互换
      * 重新调整为大顶堆： 从0出发调整
        + Heapify 的时间复杂度是O(logn)
        + 比较当前结点和两个孩子结点
        + 和两个孩子结点中较大的互换
        + 以较大的孩子结点出发重新调整子堆为大顶堆



* 非比较类排序

**实战**

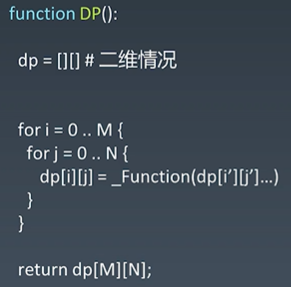
* 异位词
* 合并区间 🡺排序 + 一次扫描 🡺 高频
* 逆序对 / 翻转对 🡺 经典
  + 暴力法
  + merge sort

# **Ww09布隆过滤器+LRU cache**

* 布隆过滤器
  + 类似于hash map
  + 如果查询到的二进制位有一个不为1，查询结果确定为no
  + 如果查询到的二进制位都为1，查询结果不确定返回probably
* LRU cache (Least Recently Used)
  + 两个要素：大小+替换策略
  + Hash table + double linkedlist

# **Ww09高级动态规划**

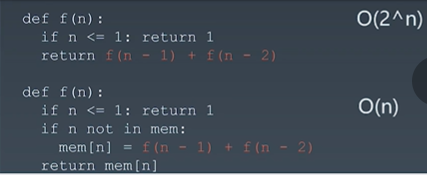
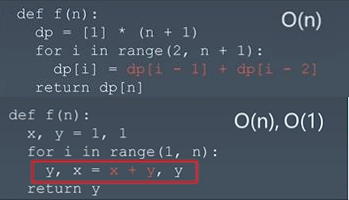
* 动态规划和分治
  + 动态规划 = 分治 + 最优子结构
  + 关键：有无最优子结构
  + 共性：找到重复子问题
  + 差异：最优子结构，中途可以淘汰次优解



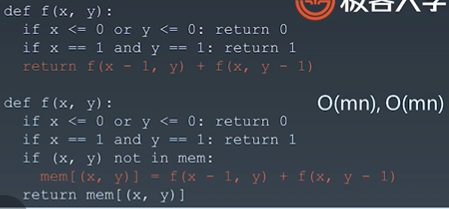
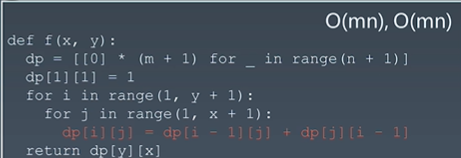
* **常见的DP问题和DP方程**
  + 爬楼梯问题/Fibonacci数列/硬币兑换

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

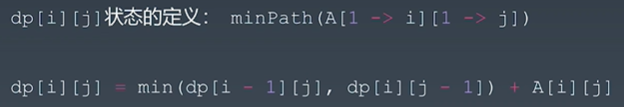
* + - 递归 O(n^2)
    - DP-top-down: recursion + memory O(n) O(n)
    - DP-down-top: iteration O(n) O(n)
    - 空间优化：两个变量 O(n) O(1)

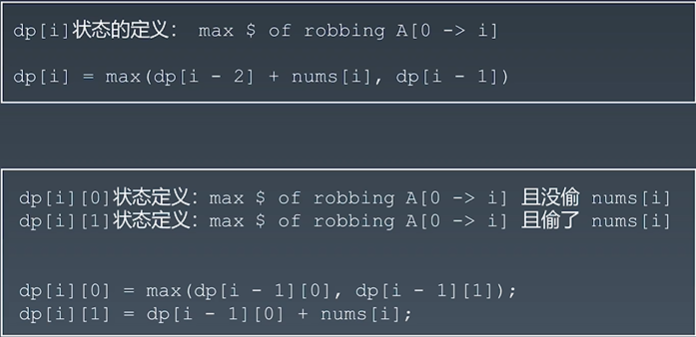
* + 不同路径问题：有无障碍物
    - f(x, y) = f(x-1, y) + f(x, y-1) O(mn) O(mn)
    - 3种实现代码

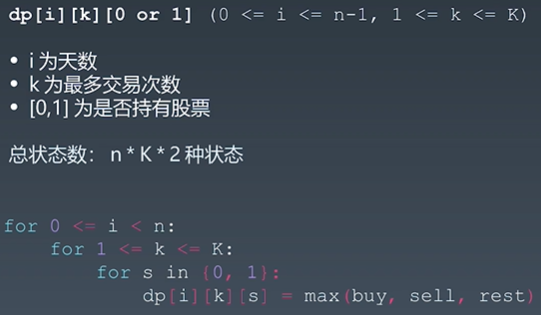
* + 最小路径和 (有无障碍物)

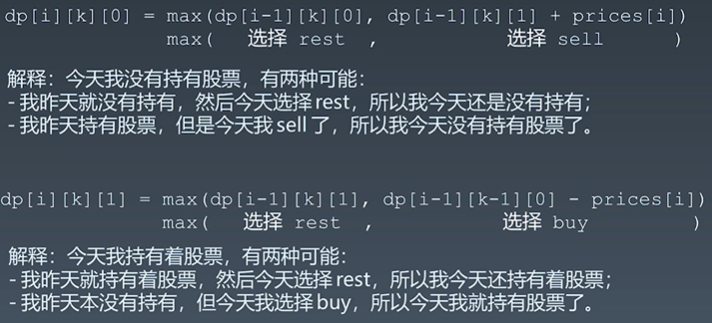


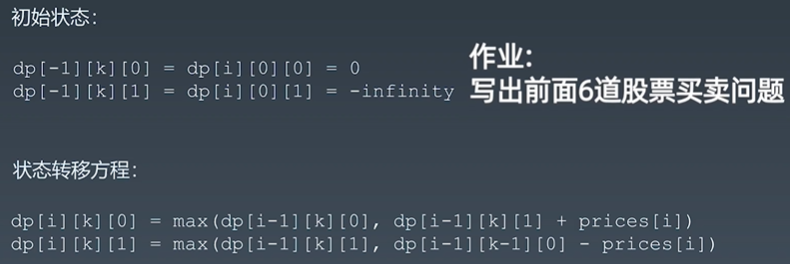
* + 打家劫舍问题：2种（房子是否首尾相连）



* + 股票买卖问题
    - 一个方法团灭6个问题







* 高级动态规划
  + 复杂度来源
    - 状态拥有更多维度(二维，三维甚至更多)
    - 状态方程更加复杂
    - 多练+多思考🡺内功+ 逻辑思维+数学
  + **爬楼梯问题进阶**
    - 走1、2、3步，或者给出一个数组
      * 双重循环

Dp[i] += dp[I – arr[k]] array is given steps array

* + - * Dp数组初始化？
    - 相邻两步不能一样
      * 增加一个维度
      * 三重循环
    - 最小花费爬楼梯
  + **编辑距离**
    - 字符串问题🡺升维

edit\_dist(i, j) = edit\_dist(i-1, j-1) if w1[i] == w2[j]

edit\_dist(i, j) = min(edit\_dist(i-1, j-1) + 1,

edit\_dist(i-1, j) + 1,

edit\_dist(I, j-1) + 1) if w1[i] != w2[j]

