八股文：

<https://github.com/Snailclimb/JavaGuide>

<https://github.com/Snailclimb/JavaGuide-Interview>

<https://github.com/CyC2018/CS-Notes>

几本不错的算法书：

代码随想录+leecode hot100+leecode 企业hot+剑指offer第二版+剑指offer 专项突破

《代码随想录》

<https://github.com/youngyangyang04/leetcode-master>

<https://programmercarl.com/>

企业题库：https://codetop.cc/home

代码随想录：https://www.programmercarl.com/

labuladong算法小抄：https://labuladong.gitee.io/algo/

编程珠玑、编程之美、算法新解、算法图解、大话数据结构、大学教科书数据结构与算法、算法导论（网易公开课有MIT讲解公开课）

PS：《代码随想录》 作者基于leetcode算法讲解强调刷题顺序，开源（导流了其他项目 star数都很多）、卖书（某一年京东计算机学科售卖第一名）、公众号（网站限制验证码、pdf资源导流进来等拉新）、知识星球（1W+人 156元/年扩展目标用户生态到 最强八股文.pdf、秋招攻略、面经、职业规划）、B站视频，同时扩展了自己的交际圈。不过看起来主要对象是校招或低级别同学，毕竟后面上来了之后就重视经验与管理沟通能力。

广告语：

《代码随想录》LeetCode 刷题攻略：200道经典题目刷题顺序，共60w字的详细图解，视频难点剖析，50余张思维导图，支持C++，Java，Python，Go，JavaScript等多语言版本，从此算法学习不再迷茫！

知识星球 强调重要的不仅仅是知识，还有同好的氛围

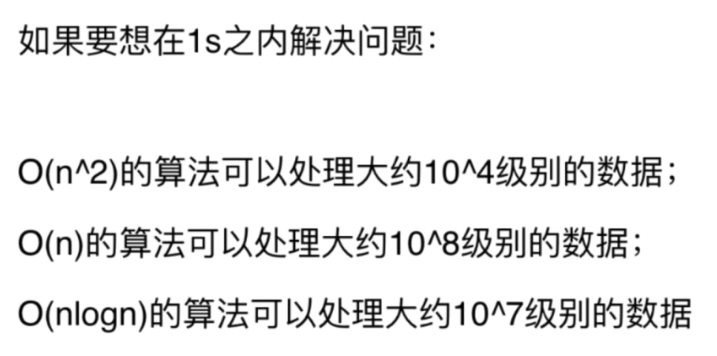
写程序，想到特殊的测试用例，一定要先记录下来，因为等会会错误地以为我已经满足了这个条件

题目没有说明数据是有序的、不重复的，但是范例十分特殊，经常会给人误导

-1. 一般ACM算法的平均时间复杂度是按照概率计算的。还有一种平摊分析（又称为均摊分析），比如java中的ArrayList在不断add的过程中会扩容，但是这个时间复杂度被平摊给整个add操作，所以不是很严重。与平均分析的区别，在于不是基于概率的。

1. 我的电脑循环一亿次空for循环还是很快的，也就是一亿次++。但是某些算法你说为什么这么慢？（cpu cache）

考虑CPU频率（1s同步脉冲次数）与**指令**的大概占用可以推算，级别是这个常数隐去了



10^9 是2的几次方呢？ 9\*log210 = 9\* lg10/lg2=9\*1/lg2=9\*3.3

据说有时候TLE，是因为算法复杂度前面的常数太大了。用STL的map容易导致超时（红黑树实现）。

分配10000字节的大数组时候与分配1000的数组这个时间差的明显。后者基本上没有怎么等待，而前者要等了。分配在程序运行中可以是一个异步操作，且一次分配足够大的，然后自己进行管控。

1. 气球的颜色代表着题目的易：

（1）亮蓝、金、黄色、花

（2）红

（3）绿

1. 来自题目的暗示

时间复杂度

如果要求结果去重，非常可能是排序后去解的

2.5. 如果编程语言中没有int的最大值常量设置，可以自己用十六进制写一个：0x7FFFFFFF; 无符号整型 ^uint(0) 有符号整型 int(^uint(0) >> 1)；对于golang 可以实现uint64 1<<64 -1 或者in64 1<<63-1因为规范要求表达式会在编译时无限精度计算，所以能得到准确值。-1即^uint(0)即FFFFFFFF，可以右移一位。

1. 简写

通过max、min、?:来缩短代码，提升可读性

python没有三元运算符:

max = a if a>b else b

print("a大于b") if a>b else ( print("a小于b") if a<b else print("a等于b") )

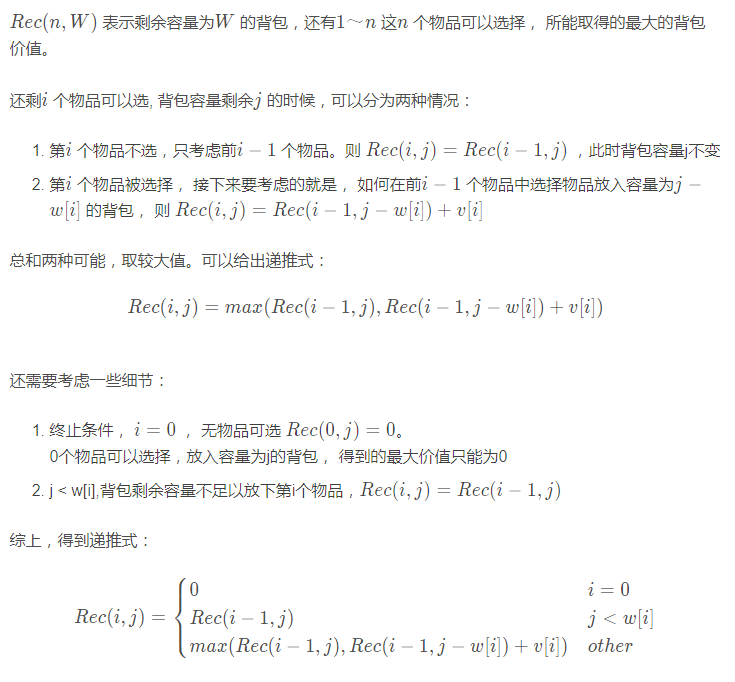
1000.0->1e3 1000->int(1e3)

1. 动态规划DP就是使用什么（**一般是多维数组，通常可以优化空间**）来记录中间的一个个子过程/子问题（总问题能被子问题以数学等式表达），那么到数组记录的最后就是我们要求的值。

用直接**递归（递归关键在于得到递推式）**的方式计算状态转移矩阵，会存在“重叠子问题”，例如背包问题。一些数值经过递归之后相当于重复计算，所以一般配合用矩阵记录过程中数据。单纯的矩阵写法是递推，即动态规划。递归加上记忆化搜索 约等于 动态规划，<https://blog.csdn.net/yoer77/article/details/70943462> 前面三块代码表示了三者之间的区别。

我总是容易想到 背包剩余重量MaxValue(5)=MaxValue(1)+M(4) =M(2)+M(3)，但这个推导式肯定不成立，不然继续加法拆解 5=1+1+1+1+1。

**01背包问题就直接考虑第i个物品拿还是不不拿，**我的例子中不拿物品重量为5才有可能后面M(2)+M(3)。







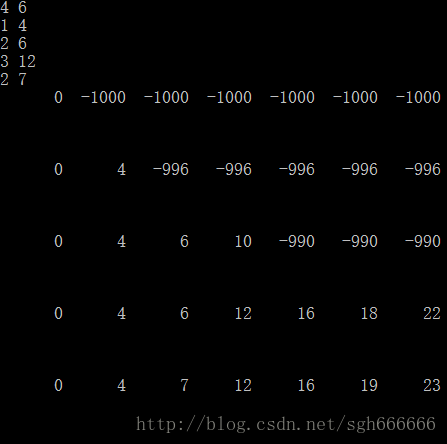
如果将记忆化搜索递归中的递归写法完全改为数组的写法，那么便是01背包的动态规划解法，递归递推式能直接改为数组关系式。







恰好装满在二维数组中设置第一列全为0，其余所有列都是-INF。如果题目不满足恰好装满，则d[W]是一个负数。可以在脑中模拟一下，W=5，给出两个W=2的物体，在W=3这列因为W=1而还是负数。



递归O(2^n)演进为动态规划 O(nW)，如果使用贪心算法以性价比求一个大致最优解那么只需要O(2n)。

完全背包的时候，利用了同一行之前求出来的值，减少求值。

3.5 递归、计算器、Linux目录解析 都使用栈，这是因为考虑栈先进后出特性表现出来的局部特性，我们“计算”后面一部分内容的时候不需要考虑过于前面的情况。如果你觉得栈与队列等价是因为在存储与写上一样，本来就是读的方式不一样。

3.6 使用队列进行BFS遍历的时候（如果已经遍历过的节点还存在队列中），如果已经知道深度，就最好使用pair<节点指针, int deepth>；否则，就要记录队列中的每一层对的范围区间。如果不需要考虑深度这个特性是最好的。坏处在于使用队列导致丢失了树（如果是一个多叉树）的结构信息，递归理解为preHandler（操作子节点）与postHandler（主要为了将什么信息传回去），队列是没有办法将信息传回个父节点的，只能单向走下去。

3.7. 有界队列是一种特殊的队列，当队列为空时，队列的获取操作将会阻塞获取线程，直到队列中有新增元素，当队列已满时，队列的插入操作将会阻塞插入线程，直到队列出现“空位”；或者丢弃内容。

3.8 优先队列通常采用堆结构实现，相关的几种堆： 二项堆、斐波那契堆、Pairing堆，固定优先级的多个队列

如果优先队列需要提供合并操作，这时候一般的大顶堆、小顶堆就不能够胜任，平白浪费了已经得到的有序性

二项堆（Binomial Tree）

可合并堆，如果不是弄成链表的形状，这就是一个排序堆

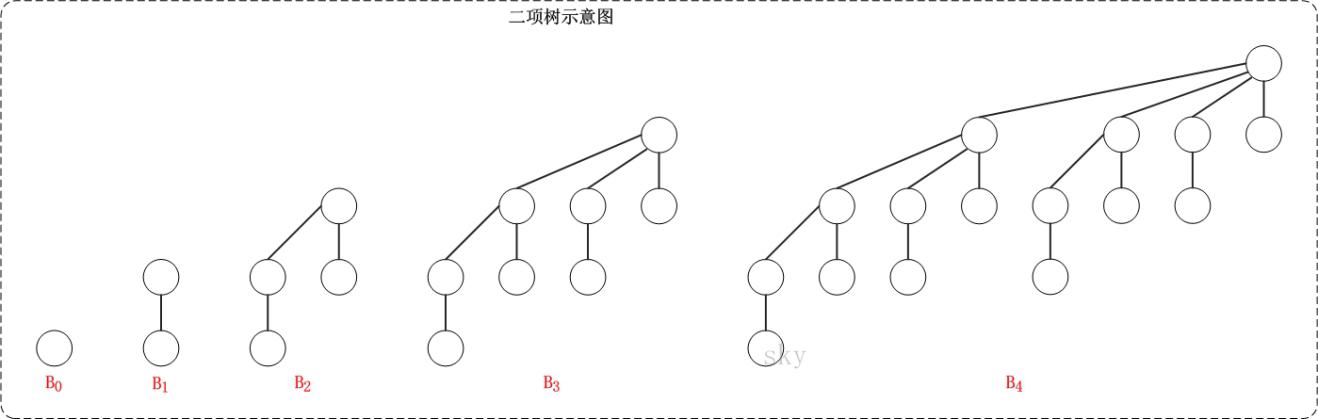
定义：

多个二项树构成二项堆

二项树的第0棵树只有一个结点；

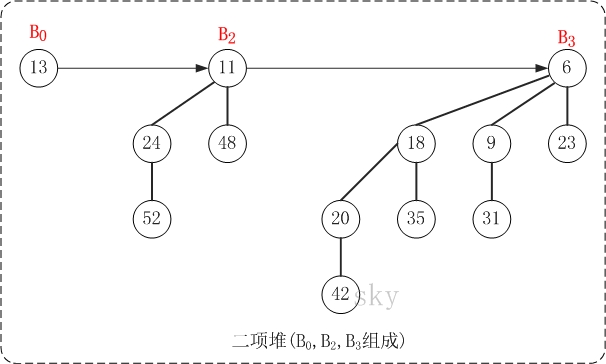
二项树的第K棵树的子结点由二项树的前面k-1棵组成的，其中一棵树是另一棵树根的最左孩子。

因为这种特殊的结构，函数编写的时候就可以使用递归

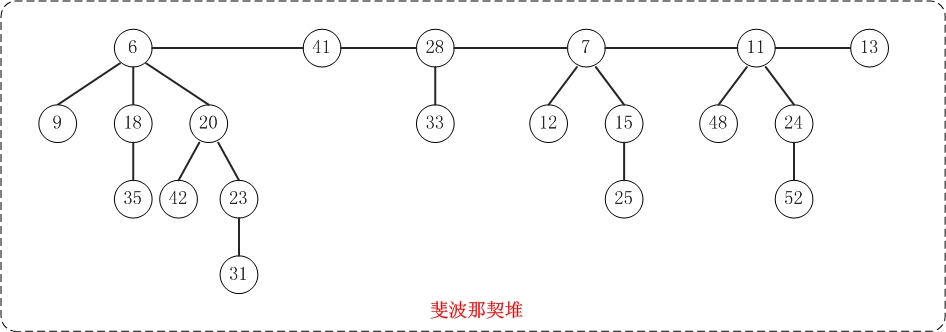


所以存在以下性质：2k个节点、高度为k、根的度数为k，它大于任何其它节点的度数、Bk在深度i处恰好有C(k,i)个节点，其中i=0,1,2,...,k

下图中的二项堆由二项树B0、B2、B3（都是最小堆）组成， 因为13(二进制是1101)的2个指数和为13=23 + 22 + 20, 因此具有13个节点的二项堆由度数为3, 2, 0的三棵二项树组成。



PS:这时候我就会想是否2^n能够构成所有的自然数，就是使用二进制能否表示所有的十进制问题。

斐波那契堆 是一种松散的二项堆（就是一般排序堆的集合）

因为结构过于随意，网上面基本没有相关的结构描述图片。

有一个指针直接指向最大或者最小的节点（只有这一个指针作为头指针，所以插入操作就是在这个指针的前后）。结构上同一层次节点采用环形双向链表连接（网上说是为了能够更好删除节点，但是我思考之后发现只有“比较的时候更快到达兄弟节点”这个作用）。插入的时候，直接插入，真正的合并操作在pop数据的时候，维护了一个度表，将一样degree的树合并 <https://blog.csdn.net/sykpour/article/details/26831783> 合并操作见https://blog.csdn.net/u012813201/article/details/62423763 最后面。

删除节点的时候，先将节点的数值转为负无穷，然后执行“节点值降低”算法（将节点往上面传），最后执行“pop最小节点”算法。

优点：除了删除元素，均摊时间基本是O(1) 本质上将很多相关的工作延期到pop数据的时候

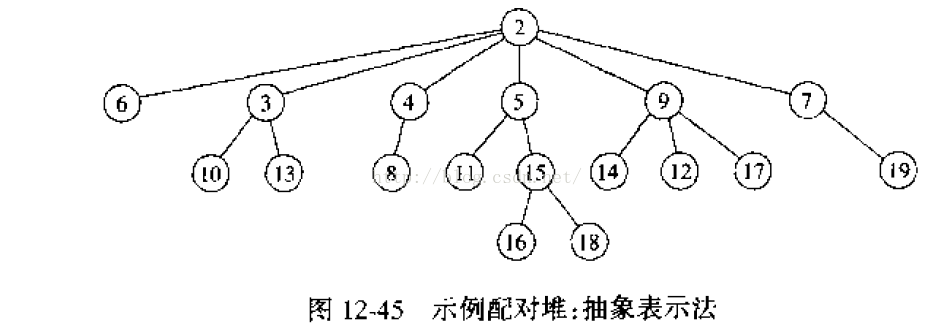
缺点：由于斐波那契堆得常数因子以及程序设计上的复杂度，使它不如通常的二叉堆合适。因此，它的价值仅存在于理论意义上。

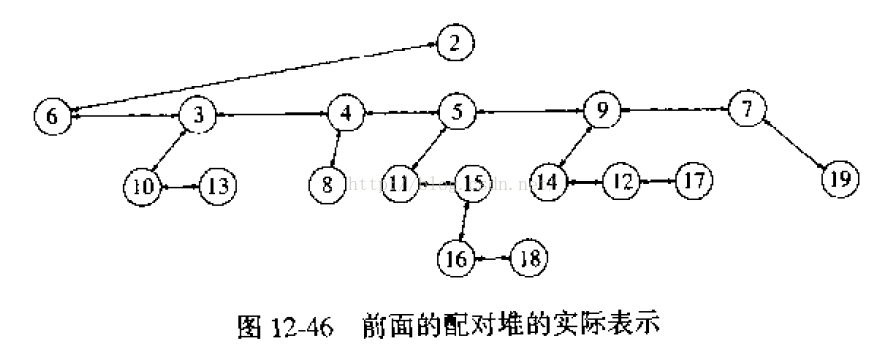


配对堆

一种简化的斐波那契堆，优势是在进行DecreaseKey的时候速度快于其他堆结构。能够用于优化Dijkstra算法。

配对堆不像其他娇气的堆，它对堆本身的形态没有什么要求，只要是棵树就行了，插入/合并也很任性，直接比较两个根的大小，优先级高的当爸爸，轻松O(1)O(1)完成插入/合并。

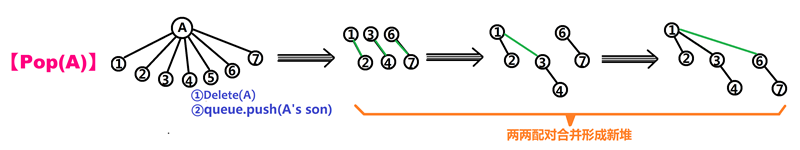




我觉得第一层的链表可以排一个序。

插入如果是单元素，单独视为一个堆，所以元素插入时间复杂度与合并操作的时间一样是O(1)

pop操作：



修改值：

对于小顶堆，如果是将值修改成为更小的，把子树取出来再重新插回去 O(1)；如果是改大比如对于Dijkstra算法，如果不想将树的广度变大，被改变节点为根元素的子堆将元素下沉，时间复杂度为O(logN)；如果变成3个堆，加到原来根元素所在位置，时间复杂度是O(1)。

左式堆:

也是为了能够方便合并排序堆

3.5 单源路径问题（一个顶点到其余顶点之间的最短路径）

松弛是指根据一条边的数值看看能不能对已经记录的最短路径再次缩短。

Dijkstra算法（要求图中没有负边）：<https://blog.csdn.net/heroacool/article/details/51014824> （加入新的点-》更新距离集合-》找出新的要加入的点，使用了贪心思想），默认的时间复杂度是O(n^2)，简单的优化是使用二分查找、优先队列这种堆排序来帮助查找最近距离的元素。使用上述几种堆能够优化时间复杂度（通过堆实现最小优先队列）：n 顶点个数 m 边的个数

用二叉堆优化Dijkstra算法的话是O( ( n + m ) log n )，这个复杂度与一般的堆一样。m边引起数据的沉降，n是从堆中pop数据。

斐波那契堆时间复杂度是O(m+nlogn)，实现的时候会采用配对堆

使用堆优化推荐的优化代码： <http://www.cnblogs.com/dilthey/p/9992463.html>

为什么要求没有负值边？因为每一次新加入的节点已经确定是找到了最短路径，如果存在负边，这个情况就不能成立，相当于不能使用贪心思想。

Bellman - ford算法是求含负权图的单源最短路径的一种算法，其原理为连续进行松弛，在每次松弛时把每条边都更新一下，若在n-1次（这是最大的可能长度，所以设置为最外层循环）松弛后还能更新，则说明图中有负环，因此无法得出结果，否则就完成。时间复杂度O(nm)，遍历每一个加入的节点的所有边:

第一，初始化所有点。每一个点保存一个值，表示从原点到达这个点的距离，将原点的值设为0，其它的点的值设为无穷大（表示不可达）。  
第二，进行循环，循环下标为从1到n－1（n等于图中点的个数）。在循环内部，遍历所有的边，进行松弛计算。  
第三，遍历途中所有的边（edge(u,v)），判断是否存在这样情况：

如何判断节点是否重复走过？ 直接看代码可以明白，并不需要担心是否经过重复的节点。https://www.cnblogs.com/godfray/p/4077146.html

上述几种算法用于路由，Bellman-Ford并不是因为出现了负值，而是基于距离矢量这种算法每一个路由不清楚图中的情况，只是知道自己到其余点之间的最小值，通过与周围路由之间共享路由表来迭代最短距离。而之后的链路状态使得每一个路由都知道了整个网络的布局情况，所以能够使用Dijkstra算法。

<https://www.cnblogs.com/jsjwk/p/9685194.html>

SPFA： 使用队列优化 Bellman - ford算法 ，采取的方法是动态逼近法：设立一个先进先出的队列用来保存待优化的结点，优化时每次取出队首结点u，并且用u点当前的最短路径估计值对离开u点所指向的结点v进行[松弛操作](https://baike.baidu.com/item/%E6%9D%BE%E5%BC%9B%E6%93%8D%E4%BD%9C)，如果v点的最短路径估计值有所调整，且v点不在当前的队列中，就将v点放入队尾。这样不断从队列中取出结点来进行松弛操作，直至队列空为止。如果一个点进入队列达到n次，则表明图中存在负环，没有最短路径。Bellman - ford代码中每一次将所有边松弛一次，从而多确定一个点的最短路径，但是很明显这是没有必要的，队列优化就是每一次将更新的点记录到队列中（可能是已知的点），将这些点相关联的边松弛就好。

SPFA还有SLF，LLL，滚动数组等优化。

3.6. 多源最短路径算法（任意两点之间的最短路径）

Floyd算法求的是所有点之间的最短路径。效率要高于执行|V|次[Dijkstra算法](https://baike.baidu.com/item/Dijkstra%E7%AE%97%E6%B3%95)，也要高于执行|V|次[SPFA算法](https://baike.baidu.com/item/SPFA%E7%AE%97%E6%B3%95)。时间复杂度为O(N^3)，空间复杂度为O(N^2)。代码写起来就是初始一个路径矩阵，然后三层循环：

　　for(k=0;k<n;k++)

　　for(i=0;i<n;i++)

　　for(j=0;j<n;j++)

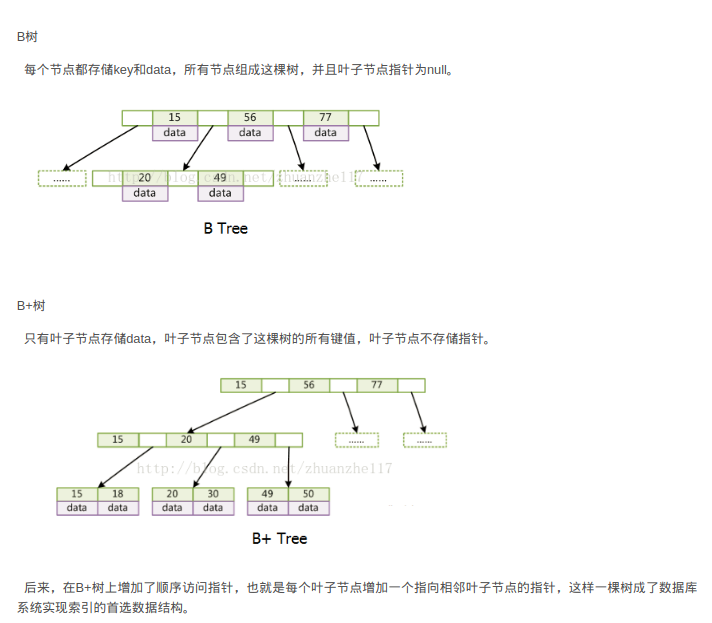
　　if(A[i][j]>(A[i][k]+A[k][j]))

　　A[i][j]=A[i][k]+A[k][j];

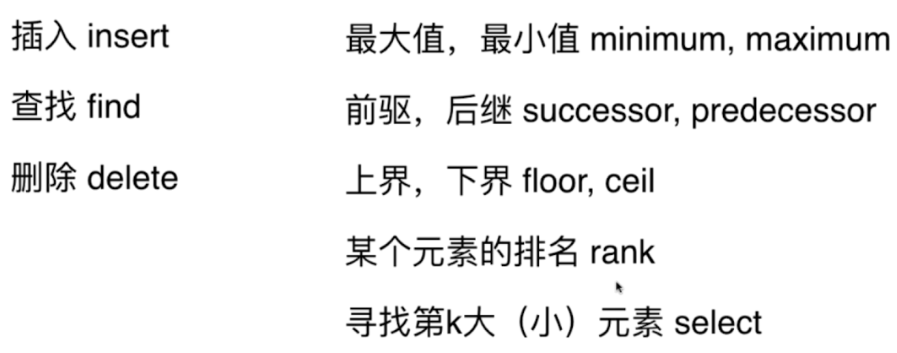
需要注意3层循环的先后顺序，对于新增节点k，更新i到j的距离。

循环的层数固定，可以在任何一个图中使用（即使图中存在负值边）

4.数据存储的时候推荐使用B-、B+ Tree或者Hash Table。B- 与Hash都不适合进行范围查找返回，所以数据库中使用的就是B+树。Hash Table单个查找倒是很快，但是如果数据量大了，一旦将自己分配的空间用完，那就要重新进行构建。所以对于数据量较大，会持续增长的情况，推荐使用B-、B+。



1. 二分搜索树需要支持的基本操作：



右边的几种名字需要以AVL树作为参考，知道代码的编写。排序的时候用的都是中序遍历。

7.排班表算法： 排课问题在70年代就证明是一个NP完全问题，即算法的计算时间是呈指数增长的

（1）基于回溯，按照需要的事时间段来安排人员（但是可以真正解决问题）

（2）基于优先级：第一优先级是某一个需要的时间段中可以选择的人员很少，应该先安排这样的时间段

第二优先级 早上最早的一班以及晚上最晚的一班，能够找到连贯的人的可能性比其他时间段小，应优先考虑

第三优先级是可以选择的人中人员的空闲时间总体程度也是不同的，应该先从总体空闲时间少的安排(考虑人员之间的公平，其实按照监考场数来发工资是最合理的，而不是作为一个业务需求)

良好的情况下一般这种解法都能够得到结果。但是我的数据库课程设计项目因为数据抽象的原因，不太能支持这种算法。

7.5

乘法hash： hash(key) = floor( m \* ( A \* key mod 1) )

A是一个小数，( A \* key mod 1) 表示使用运算后的小数部分

全域hash：

因为单独的hash在比较的时候容易被人针对（经常是举例两个公司设计的代码竞标），所以设计一个hash函数集，每一次使用的时候会随机从函数集中选取一个hash函数。

因为hash函数是随机选取的，所以两个元素发生碰撞的概率时候1/m 。

完美hash：

最坏情况也会是O(1)，通常使用双重hash表。

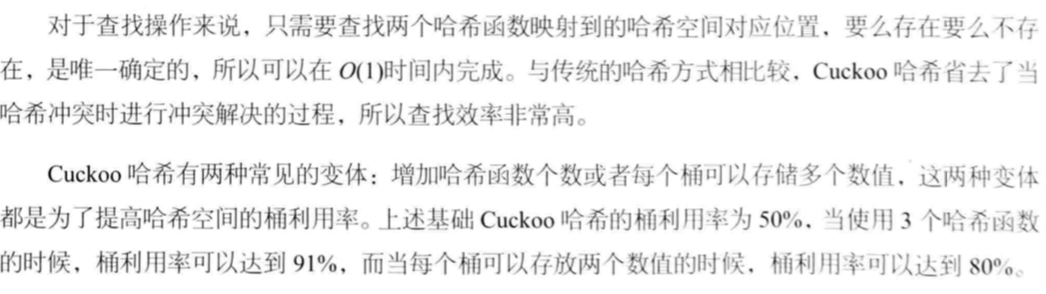
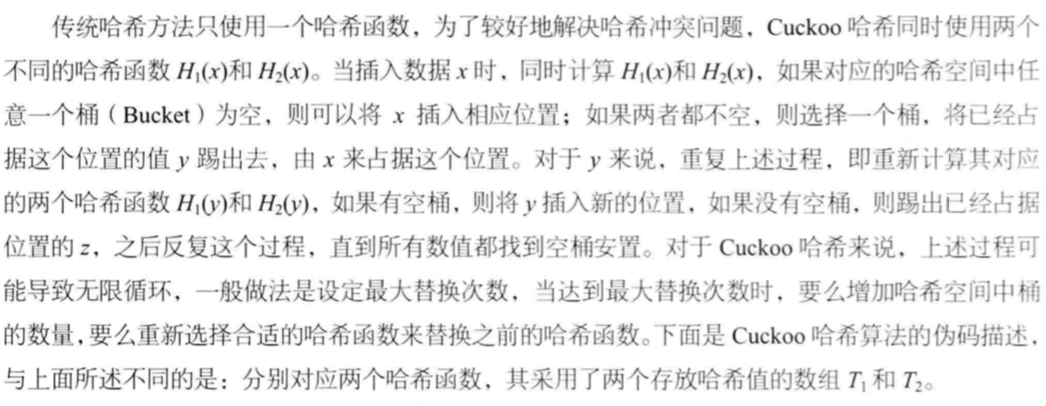
对hash冲突的解决：  
 从冲突的位置开始，线性占用之后的空间，这种办法会导致同类型的存储在相邻的位置（虽然我觉得更方便寻找同类的元素，但是这种容易导致hash表的空间不足，这种一般的hash办法记录表的负载因子来控制hash是否扩容）。所以有一种改进办法是将存储位置改成s - 12，s + 22，s - 22，s + 32

hash 可以通过再散列的办法解决冲突(实现上每一个index记录是否这个index有rehash的情况，并通过原值的equals判断是否找到对象)

双重数据结构：每一个index下面挂一个链表或者红黑树比较好（多级数据结构，甚至可以hash后面加一个hash，每一个index下面挂载的数组可以不是一样大的，自己记录大小）。从占用空间的角度看，这种一点都不具有可扩展性。

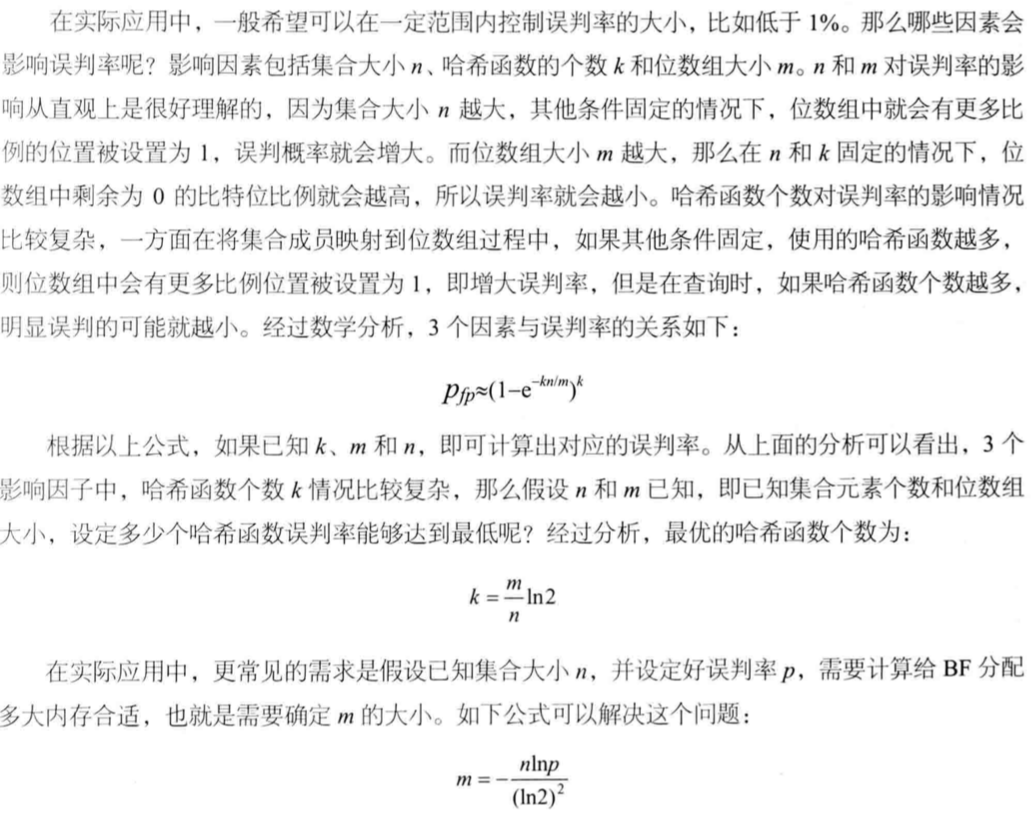
Cuckoo hash

有效解决hash冲突问题，O(1)删除与查找元素，在常数时间插入，大约50%的空间利用率

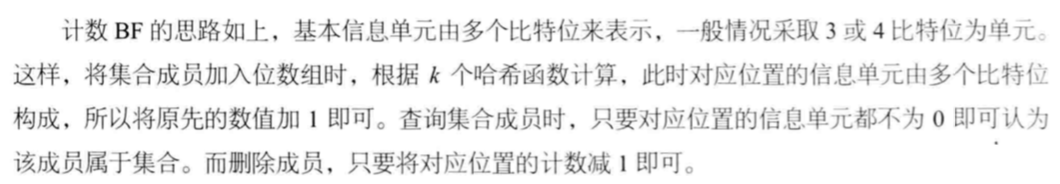


7.6. Boolean Filter 布隆过滤器

使用多个相互独立的hash函数，将元素映射到一个bit array中，存在误判的可能，不会漏判。



支持删除：



应用于：

浏览器的恶意URL判断（容纳误判）

网络爬虫对已经爬取的网站判断（速度要求）

缓存的是否存在判断（速度要求）

8.查找字符串：在一堆字符串里找一个字符串

（1）弄成字符串数组，通过遍历

（2）弄成数组其实不方便，因为字符串你是复制粘贴过来的，需要不停加上引号与逗号，没准可以直接括起来弄成一个大字符串然后找子串。这种方法就需要注意会不会你要寻找的字符串只是其他字符串的一部分。或者可以先试一下按照某一个间隔符划分成为数组。

（2）的思路是从本质上看到问题

9.使用缓冲区来存储内容的时候，回退会是一件很简单的事情，系统的输入缓冲区、文件的读取缓冲区（文件指针位置）等等，实在不行就实现自己的缓冲区。除了回退，还可以设立指定位置的还原点。

数据库的rollback需要记录的是关键时刻的还原点，还有文件

10. 对一个文本文件每一行采用\_.split(" ")，需要考虑空行（不要记空行应该记函数失败的可能）

11.



基数排序即桶排序，使用倒排索引的思想

上图中可以知道，堆排与归并排序在各种情况下都有不错的时间复杂度，所以被人喜爱，其中堆排序不稳定，归并排序稳定。

<https://www.cnblogs.com/mamamia/p/8602293.html> 还有动图演示，还介绍了一种计数排序

稳定与不稳定 是通过排序之后原数据（对于相同值的数据）先后顺序是否变化：

基数排序、冒泡排序、直接插入排序、折半插入排序、归并排序是稳定的排序算法。

快速排序、希尔排序、堆排序、直接选择排序不是稳定的排序算法。

冒泡排序是一种指令数目更多的选择排序，我都将最外层循环理解为“次数”，即变化过程中已排序部分的大小。插入排序也可以理解为已排序部分的大小。

插入排序：

//在后面插大数  
 public static void selectionSort(int[] array) {  
// if(array.length<0) throw new ArrayLengthException();  
 for( int i1=0; i1<array.length; ++i1 ) {  
 int maxNumberIndex = 0;  
 for(int i2=0; i2<array.length-i1; ++i2) {  
 if( array[i2]>array[maxNumberIndex] ) {maxNumberIndex=i2;}  
 }  
 int tmp = array[array.length-1-i1];  
 array[array.length-1-i1] = array[maxNumberIndex];  
 array[maxNumberIndex] = tmp;  
 }  
 }

//自前面插小数，位置刚好能直接和次数有关系，所以看起来代码会比插大数好看，没有这么多表达式来表示索引  
 public static void selectionSort2(int[] array) {  
 for( int i1=0; i1<array.length; ++i1 ) {  
 int minNumberIndex = 0;  
 for( int i2=i1; i2<array.length; ++i2 ) {  
 ...  
 }  
 }  
 ...  
 }

插入排序，Java里面有一个System.arraycopy（不能用，会破坏数据的），否则就要网址动图上面这么移动。

shell排序，长度差一开始就是length/2，如果length为奇数，那么也不用担心，按照排序思路进行每一个元素都会参与排序的。同间隔元素之间，一般使用插入排序实现。

如果每一次遍历从[0,当前间隔差)开始，还要再加一个循环来遍历，但是上面网址中，直接一个for (int i = gap; i < len; i++){...} 这样的循环解决了

网址上shell排序的例子举得不好，8个元素才会最容易体现那种交叉有序，但是不能时时刻刻保证交叉有序，不然有更好的排序方法而不是基于插入。其实基于冒泡也不错。总的来讲，时间复杂度还是很高的。

如果是对数组进行归并排序，不需要申请空间，按照他这种2、4、8...的思想合并就好。而合并两个有序数组，不该使用选择、冒泡、选择这几种办法

发现对于合并的思想，只能创建一个新的数组来容纳。对于已经两个有序的数组，O(1)的空间复杂度只能是通过插入排序实现了，因为数组内值超级反复大量的移动会导致，最终效果比直接使用插入排序还糟糕。所以明白了归并排序的重点了嘛？

对比，shell排序的那种有序（一个大部分元素有序的数组）倒是使用插入排序。

C++ 堆排函数:

make\_heap(v.begin,v.end,greater<int>());//用定义好的函数greater来排降序

sort\_heap(v.begin,v.end,greater<int>());

#include <iostream> // std::cout

#include <algorithm> // std::make\_heap, std::pop\_heap, std::push\_heap, std::sort\_heap

#include <vector> // std::vector

#include <time.h>

#include <stdlib.h>

using namespace std;

int main () {

srand((unsigned)time(NULL));

vector<int> v;//将myints复制到v

for(int i=0;i<10;i++)//随机生成10个数并排序

v.push\_back(rand()%20);

make\_heap (v.begin(),v.end());

sort\_heap (v.begin(),v.end());

cout << "final sorted range :";

for (unsigned i=0; i<v.size(); i++)

cout << ' ' << v[i]; //这样子输出并不是有序的

return 0;

}

自己实现数组堆排序的关键在于父节点与子节点索引之间的关系：

到时候画一颗三层的树就好  
父节点 = (子节点索引-1)/2 并结果取整

左子节点=...  
右子节点=...

对于近乎有序的数据，插入排序才是最好的选择

如果数据范围有限，比如某一省的高考成绩，存在一个不大的范围，排序使用基数排序是最好的选择。

快排：

单路、双路、三路<https://blog.csdn.net/k_koris/article/details/80585979>

如果数据中包含大量重复的元素，一般的快排方式会导致划分的两边不均匀，最终导致时间复杂度的退化。推荐使用三路快排。

快排依赖于数组的快速随机访问，所以对于链表推荐使用归并排序、堆排序

需要注意的事项

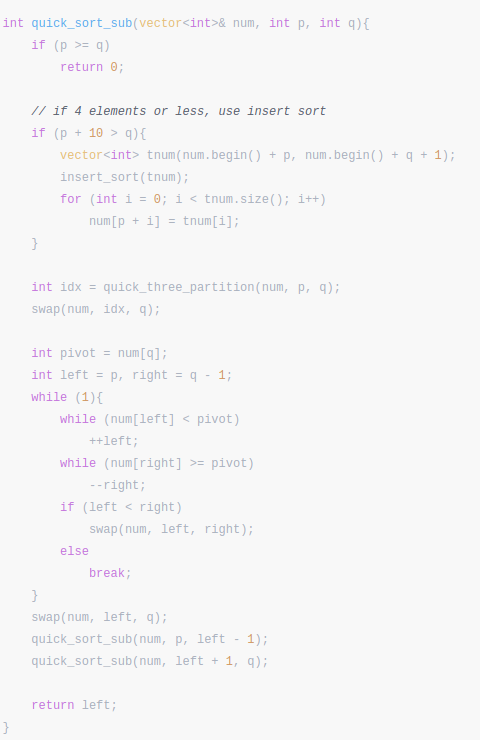
首先要写正确。通常使用递归实现。其递归相当于二叉树展开，因此如果要用迭代实现的话需要使用一个队列等数据结构来保存后续遍历信息，记录分段的点就好。

快速排序需要找到一个pivot值，如果顺序选择pivot则易造成N^2的复杂度，如果使用随机数则效果最好，但开销又太大，采取三数中值法比较合适。三数中值法指的是选取第一个值，最后一个值，数组中间的值的中值。有文献表明可以提升5%的运行时间。

当数组长度较小时，如10个元素以下，最好使用插入排序或者选择排序完成，以防止复杂度 常数因子过大或多次函数调用带来的开销。而递归到底层数组长度总是会变小的，因此这么做非常有必要。

在合并前后两部分数组时，采用两边夹方法，在前后两部分各找到一个大于和小于的值再交换。相比通常情况下找到比pivot小的值就进行交换，能提高运行效率。

双路快排代码：



堆排序如果已经排好一部分，这时候插入数据（如果需要大量插入数据，肯定不能使用连续存储类），将插入的数据作为最后一叶子节点加入，并重新建堆，判断之后如果之后一开始取出的数据连着都是刚刚插入的数据，还需考虑通过插入排序将这些数据插入到已经排好的部分中。

计数排序，在保证空间足够的前提下，这种排序的时间复杂度为O(n)。基于比较的排序，经过证明，在随机的情况下，时间复杂度最好也只能是O(nlogn)。桶排序也不是基于比较。

每一个数都可以使用二进制表示，比如32bits，我们通过多轮的计数排序，每一轮排序按照8bits进行，所以需要进行4轮 。如果将总时间复杂度的表达式写出来，求导可以推出当每一轮的长度选取logN的时候整个表达式处于最小值。

外部归并排序算法： 包含通过胜者树、败者树去优化K路归并

<http://www.360doc.com/content/15/0719/22/22633806_486084820.shtml>

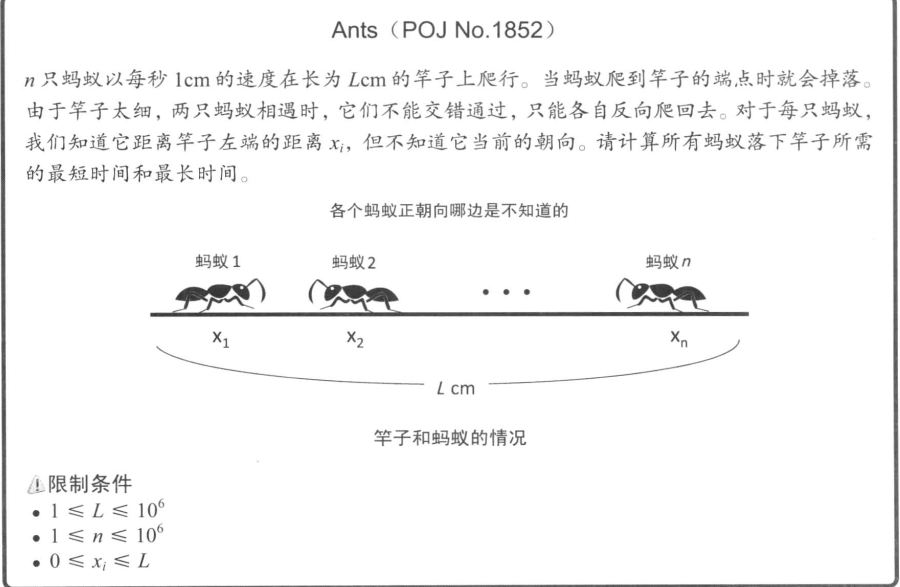
<http://blog.chinaunix.net/uid-26602509-id-3331691.html>

<https://www.cnblogs.com/charlesblc/p/6138908.html>

HashMap存了若干(name, age)这样的键值对，现在想按照年龄排序，打印出姓名？这时候适合使用桶排序

据说现在的最好的排序方法做到了将最坏情况下的时间复杂度控制在O(nlog(logn))，但是这种算法十分复杂。

1. 两个字符串之间的最长子串问题（本子上也有记录） <https://www.cnblogs.com/ider/p/longest-common-substring-problem-optimization.html>

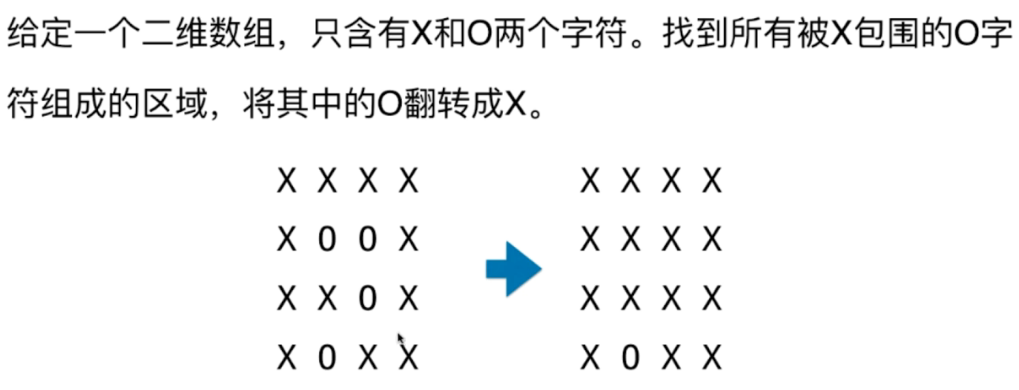


这道题目的关键思路是如何看待蚂蚁的运动，他们相撞然后各自调头返回？是不是会有一种感觉，有一只蚂蚁会不停地撞来撞去，然后这个问题就没有办法解决掉。为何不认真点，完整点做一次模拟，模拟不了大的n，那就模拟小一些的，然后就会发现蚂蚁的相撞可以认为是互相穿过。拓展一下，所以会造成速度交换的，在空间上面都可以认为是穿过。

1. 起点到终点的最短路径，广度优先遍历

求多维数组连通块的问题，又称为种子填充（floodfill），这种问题支持BFS或者DFS（回溯递归，DFS虽然可能走起来会有点绕，但是因为有一个记录所有走过的矩阵，所以就不用担心），每一次经过之后可以将数组中的值改变，如果不能改变原矩阵的值，就要自己记录一个是否经过的矩阵。

floodfill中最经典的是求被水环绕的岛屿个数，下面是一道变题：



其实实际上是O组成的岛屿问题，需要一个vector记录正在走的块indexes，还需要一个同大小矩阵记录所有已经走过的地方。



必要的判断还是要有的，优化的关键在于记录某一个点可以去太平洋还是去大西洋还是未知状态（一个int值足以表示这6种状态）。只需要在递归回来的时候，根据函数的返回值（能够成功的），记录在数组中即可。

其实还可以从边界开始按照大小关系，直接初始化可到达数组，当然实现上需要配合一个是否经过矩阵。

下面这道题目需要输出一个合理的路径或者路径长度，所以不能通过栈实现的深度有限遍历，因为深度遍历可能会走一条十分扭曲的路，根本不是我们想要见到的这样



14.5 操作数栈: 大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈

1. 去商店购物，选择合理的优惠方案

输入商店商品与价格二元组对

输入要去商店的购物产品以及个数

输入商品编号的组合方案以及优惠价格

根据以上信息去输出得到如何使用优惠能够让自己最受利

我想到用了贪心，计算每一个优惠方案中单个商品的优惠力度，然后排序，从高到低选取

但是贪心果然是不一定完美的，我举个例子：

商品1,2,3的分别要购买5,3,1个。

优惠方案

5个1和2个2能够优惠5元

1个1和1个3能够优惠2元

按照我的判断我会选择优惠2元，因为相当于每一个商品优惠1元，但是优惠5元才是最大的优惠。所以这种贪心只能用在商品购买量足够的前提下。

1. 背包不是一种解法，而是指一类问题，这类问题可以通过贪心、动态规划来解决。十分不推荐使用贪心，因为数据是离散的，在增加一个容量之后可能会因为一种优惠政策突然能再买一次而有了更好地优惠方案。
2. sqrt算法的实现：

介绍网站： <https://blog.csdn.net/qq_26499321/article/details/73724763>

下面速度只是反应了一个大概的级别

（1）二分法猜测结果（比系统方法慢了100倍左右）

（2）牛顿迭代法（比系统方法慢了10倍）：首先随便猜一个近似值x，然后不断令x等于x和a/x的平均数

（3）系统提供的库函数（编译器的函数都经过了严格的优化）

（4）曾经游戏商3D引擎提供的一个算法（比系统库函数还要快）

float InvSqrt(float x)

{

float xhalf = 0.5f\*x; //乘法代替除法

int i = \*(int\*)&x; // get bits for floating VALUE

i = 0x5f375a86-(i>>1); // gives initial guess y0

x = \*(float\*)&i; // convert bits BACK to float

x = x\*(1.5f-xhalf\*x\*x); // Newton step, repeating increases accuracy

return x;

}

这个方法本质上还是牛顿迭代法，关键在于 0x5f3759d这个魔法数字

1. 若把变量之间的关系看成一条有向边，那么多条关系表达式就能组成一个有向图，即拓扑排序。
2. 对工作人员成绩进行评定：<18为A 18<=且<27为B 27<=且<36为C 36<=为D。数据等差，差为9，所以设置原点为9（18-9），然后除以9使用switch。总的来说就是用一种等差划分，让数据范围边界除以划分点。
3. hash 取余对象最好是足够大的素数：<https://blog.csdn.net/maoliran/article/details/52082829> 避免因为key设置失误：

“ABC” = 128 \* 128 \* 65 + 128 \* 66 + 67，这种形式中，若M = 128，那么只要根据最后一个字符是否相同就可以判断是否会冲突，也就是说结果取决于最后一个字符，便会造成不均匀。

如果要尽可能避免碰撞的发生，需要设计一个足够良好的Hash散列函数，这个函数要与所有的元素有关（比如传入一个字符串就要与所有字符都有关），而且每个元素有关的权重要尽可能一样，同时产生的结果范围中各种结果的可能要差不多。

1. 给定几百万个IP段(如10.0.0.1 -- 10.0.0.100)，不重复，不交叉，现给你一个IP，求IP所在的IP 段。（不计IP段数据的预处理时间）

因为不记录预处理时间，所以只能使用方法一

方法一：其实一个IP对应一个整数，而整数的处理可以转换为bitmap，1x10^9这么多数的bit大概会占用120MB空间。ip转数字，就好比看成一个256进制，再转10进制；类似的纯字母字符串可以视为26进制的数据。

方法二：如果进行字符与访问比较，if会判断很多次。将ip转换为整数之后，判断一个范围也要判断两次？不，如果ip一定在段中，只需要判断范围中的较大值就好，一个个判断过来。这种方法可以和磁盘读写配合，于是内存占用不大。

配合二分法

方二找到的可能不是范围，所以可以根据index做一个对应。或者你可以补全ip空间上没有给出的那些段，那么上一次计算完的范围是可以推出这一次的开头。

1. 大量数据处理问题

因为是大量数据，基本都需要首先对数据进行划分，网上常见的划分是对数据%xxx，然后写到xxx个文件中。我觉得用文件大致偏移（使用除法计算）来划分文件更好，这样每一个文件的大小差不多，数据量上稍微有点偏差。

比如说频率的top k问题，这时候用hashmap，同时对于这种与重复相关的问题，将数据按照hash到不同文件就有意义了。如果需要寻找重复或者不重复，可以考虑用bitmap或者使用hashset（还有搜索二叉树/红黑树/Trie树）。虽然1G=2^32 bits，可以用于很大的bitmap。但是如果还是内存不够的话，这时候可以配合文件存储，进行划分。

bitmap可能只能包含数据中的大部分，所以还要配合其他数据结构。要么是存储不下，不过最担心的是因为数据的稀疏性，导致空间利用率的下降。

如果是只是top k部分排序问题，使用堆排序或者快排，结合外排，这种只需要部分有序就能够得出结论还是很赚的。对于top2问题，选择排序是最好的。

倒排索引的思想

能MapReduce是最好的

**给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url？**

处理相同hash但是字符串内容不同的话，所以要将hash结果与对应的字符串都存储到一个文件中。通过插入排序+二分法+记录一定百分比的位置减少二分次数（文件系统要能够通过链表很好支持插入）最终完成排序（这样其实没有hash的必要）。

跳跃表算是二分法上面的二分法上面的....二分法

**在2.5亿个整数中找出不重复的整数，注，内存不足以容纳这2.5亿个整数。**

原作者说采用2-Bitmap（每个数分配2bit，00表示不存在，01表示出现一次，10表示多次，11无意义）进行。因为单独的一个bit不能够判断多次重复的情况，只能判断“有无这个数字”。

**给40亿个不重复的unsigned int的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那40亿个数当中？**

这个问题在《编程珠玑》里有很好的描述，大家可以参考下面的思路，探讨一下：  
又因为2^32为40亿多，所以给定一个数可能在，也可能不在其中；

使用bitmap没有问题

这里我们把40亿个数中的每一个用32位的二进制来表示  
假设这40亿个数开始放在一个文件中。

    然后将这40亿个数分成两类:  
      1.最高位为0  
      2.最高位为1  
    并将这两类分别写入到两个文件中，其中一个文件中数的个数<=20亿，而另一个>=20亿（这相当于折半了）；  
与要查找的数的最高位比较并接着进入相应的文件再查找

    再然后把这个文件为又分成两类:（没必要再，直接第一次划分成多个小文件）  
      1.次最高位为0  
      2.次最高位为1

并将这两类分别写入到两个文件中，其中一个文件中数的个数<=10亿，而另一个>=10亿（这相当于折半了）；  
    与要查找的数次最高位比较并接着进入相应的文件再查找。  
    .......  
    以此类推，就可以找到了,而且时间复杂度为O(logn)

寻找一个无序数组中的中位数也是top k问题

一般的top k问题（顺序统计问题），如果使用优先队列，能够将时间复杂度优化到O(nlogK)（通过hashmap统计频率），如果K比较小的话这种方法反而比快排（期望时间是线性的）或者堆排这种部分排序反而好上不少。

对快排top K的改进：出现最坏情况的原因在于每一次选取的pivot可能不是很好，经过一下两步的选取能够得到一个不错的pivot，但是因为时间复杂度前面的常数过大导致基本不使用：

1.把n个元素每5个一组进行划分，并且找到每一组的中值；（舍弃最后元素不足的组）

2.递归（重复第一步操作）地选择所有的中值(一共是Floor(n/5)个元素)的中值x来当主元；

经过上面操作，小于pivot或者大于pivot的比例都至少是3/10。第一步的时间因为操作一个固定长度的数组寻找中位数是一个固定常数，所以是T(n/5)，可以通过任何方法来寻找第一步中的中位数，第二步算出来大致是T(7n/10)。 如果不是5个元素一组，而是3个元素一组会导致这个表达式一定成立，时间复杂度不会是线性的。

还需要单独考虑如何处理数字重复情况。

如果使用二叉搜索树等解决top k问题，需要记录每一颗子树的size大小

堆排序解决投top k问题只能将一个个内容取出排序

1. 哈希树

类似于Haffuman树，这里基于文件系统本身的结构，算是一种hash树

在计算机科学中，哈希树是一种持久性数据结构，可用于实现集合和映射，旨在替换纯函数式编程中的哈希表。 这种树在插入与删除的时候，不需要进行结构上的变换。

插入操作直接插入就好

删除操作不需要任何调整

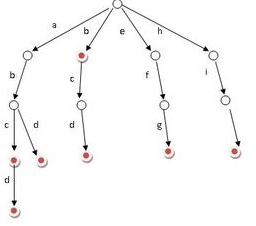
对质数数组（从大到小的顺序，树深度更小，因为数据更加可能分散）取余，也可以组成一个树：



上图哈希树(HashTree)在从4G个对象中找出所匹配的对象，比较次数不超过10次。在实际应用中，调整了质数的范围，使得比较次数一般不超过5次。也就是说：最多属于O(5)。

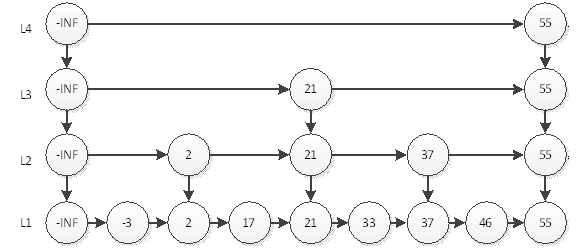
字典树Trie树：

利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希树高。



也是一种hash树，因为操作不需要树的结构变换。但是与质数分辨法比较，对字符串做了优化，其实可以将字符串看成26进制。

1. Bloom Filter很好的介绍 <https://blog.csdn.net/jiaomeng/article/details/1495500>
2. 跳跃表 能够将有序链表的元素查询类似于多重二分查找(用于已排序且元素不重复的数据中)



之前一直以为因为链表不能支持随机访问，就不能实现二分查找

但是还不能支持构造使用二叉搜索树来保存数据？其实跳跃表是一种二叉树

当插入大量元素之后，使用这种办法维护跳跃表：为了让每一个元素插入的代价不是很大，同时维护好信息，插入到应该的位置中后，按照“投硬币”，如果是正面就将节点上提并且再投一次；如果是反面就停止投币。甚至通过这种办法，将最左边的元素设置为负无穷节点，按照插入节点的元构造这一整个跳跃表。

与红黑树等平衡树相比，跳跃表具有以下优点：

* 插入速度非常快速，因为不需要进行旋转等操作来维护平衡性；
* 更容易实现；
* 支持无锁操作

30. 不过我认为部分连续，多个数组合并组成的结构也算是支持随机访问，每一个数组的大小不能“过小”。

31. 回溯 递归、循环

对于回溯问题不能仅仅用循环去搞定，因为循环能解决问题的是已知循环层数的，否则问题就会变得麻烦而不适合用循环，除非加队列等数据结构，只需要存储与去除任务就好。所以回溯的话还是用递归吧，递归只要知道终止条件，不需要知道递归的层数

arr = [1,7,3]

n = 1234用arr中的数 组成比n大的最小的数, 可以重复使用

/\*  
 \* 下面是本题的几种解法  
 \* 1. 从最高位开始回溯（递归或者循环写法）  
 \* 2. 直接找出可能需要特殊处理的点 我一开始的思路是直接寻找需要回溯的点  
 \*/  
#include <iostream>  
#include <vector>  
#include <string>  
#include <algorithm>  
  
**using** std::cin;  
**using** std::cout;**using** std::endl;  
**using** std::sort; **using** std::lower\_bound;  
**using** std::vector;  
**using** std::string;  
  
//void func(const vector<int>& vec, );  
  
**int** main(**void**) {  
 **int** arr[] = {1, 7, 3};  
 **const int** size = **sizeof**(arr)/**sizeof**(**int**);  
 **int** n = 1234;  
 **int** n0 = 1338; // 1371  
 **int** n1 = 3381; // 3711  
 **int** n2 = 7777; // 11111  
 **int** n3 = 7737; // 7771  
  
  
 **if**(n==0) **return** 0;  
  
 sort(arr,arr+size);  
 vector<**int**> value;  
 **for**(**int** i1=n3;i1!=0;i1/=10) value.emplace\_back(i1%10);  
 **for**(**int** i1=0;i1<value.size()/2;++i1) value[i1] ^= value[value.size()-1-i1] ^= value[i1] ^= value[value.size()-1-i1];  
  
 // arr 1 3 7  
 // value 1 2 3 4 下面的代码是真的骚  
 value.emplace\_back(10);  
 **int** index;  
 **for**(**int** i1=value.size()-1;i1!=-1;--i1) {  
 **if**( value[i1] > arr[size-1] ) {  
 **while**( i1!=0 && value[i1-1]==arr[size-1] ) --i1;  
 index = i1-1;  
 }  
 }  
 cout << index << endl;  
 value.pop\_back();  
  
 **if**(index==-1) {  
 cout << string(value.size()+1, **static\_cast**<**char**>('0'+arr[0])) << endl;  
 }**else** {  
 **int** i1=0;  
 **for**(;i1<index;++i1) {  
 **int** tmp = \*lower\_bound(arr,arr+size,value[i1]);  
 cout << tmp;  
 **if**( tmp > value[i1] ) **break**;  
 }  
 **if**(i1==index) {cout << \*(lower\_bound(arr,arr+size,value[i1])+1);}  
 **for**(i1+=1;i1<value.size();++i1) cout << arr[0];  
 cout << endl;  
 }  
}

32. (3+4\*5)/(2-1)+2 3 4 5\*+2 1-/2+中缀表达式 后缀表达式或者前缀表达式

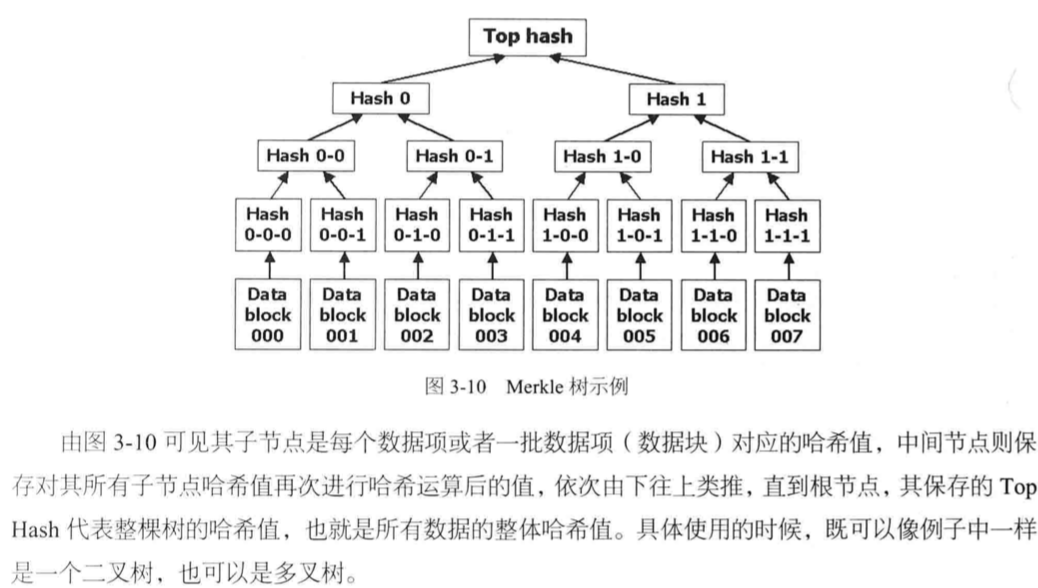
33. 用位运算区分奇偶数：

i&0x01 若为奇数则表达式为1，否则为0

还有种方法是%2,从位的角度上来看，效果是一样的。

1. 背包九讲
2. LSM 树 将大量随机写操作转换为批量的序列写，提升写效率但影响读
3. Merkle hash tree：

最早只是应用于签名验证算法，后来在分布式中能快速定位少量变化的数据。



1. LRU

LRU-k(Least recently used,中文翻译成“最近最少使用”但是一般是指LRU-1仅考虑时间而不考虑频次)算法

用数组或者链表存储元素，每一个元素有字段记录访问次数k、最近一次访问时间戳，当空间达到定值时，需要剔除访问次数最少的元素（如果访问次数相同，则最近一次访问时间最久远），可以O(n)遍历存储，也可以实时组织排序。对外提供put()、get()、clear()（put但是size已经满的时候使用）、size()接口即可。

存在一个线程/协程定时对存储进行清0，严格来讲，清0时不能对元素进行清除，需要加读写锁。不过实现上可以放宽。

如果是以LRU实现缓存，那么不妨在上层再配上一层map。

LRU-1：可以考虑用链表实现，每一访问都将元素挪到链表尾部

多层队列：设计多层优先队列，每一层都是LRU（最底层是FIFO）队列，数据被重复访问到一定次数就会上升，当一定时间没有被访问就会下降。多层队列的好处1.底层是FIFO等2.热点数据不会被一下子清空到0，而是逐步消热

Redis 实现非标准LRU、LFU（可能淘汰出来不是最佳置换） <https://redis.io/topics/lru-cache>

LRU （LRU-1，对象内时钟与全局时钟都只有24位，按秒为单位来表示才能存储194天，需要考虑溢出情况，比如对象内时钟反而大于全局时钟）Redis的LRU并不维护队列，只是根据配置的策略要么从所有的key中随机选择N个，要么从所有的设置了过期时间的key中选出N个键，然后再从这N个键中选出最久没有使用的一个key进行淘汰。

LFU(Least Frequently Used，LRU-k)把原来的key对象的内部时钟的24位分成两部分，前16位还代表时钟，后8位代表一个计数器。16位的情况下如果还按照秒为单位就会导致不够用，所以一般这里以时钟为单位。

Java中的LRU实现方式

可以直接以LinkedHashMap重写removeEldestEntry实现

在Java中LRU的实现方式是使用HashMap结合双向链表，HashMap的值是双向链表的节点，双向链表的节点也保存一份key value。

* 新增key value的时候首先在链表结尾添加Node节点，如果超过LRU设置的阈值就淘汰队头的节点并删除掉HashMap中对应的节点。
* 修改key对应的值的时候先修改对应的Node中的值，然后把Node节点移动队尾。
* 访问key对应的值的时候把访问的Node节点移动到队尾即可。

作者：后厂村老司机  
链接：https://www.jianshu.com/p/c8aeb3eee6bc  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

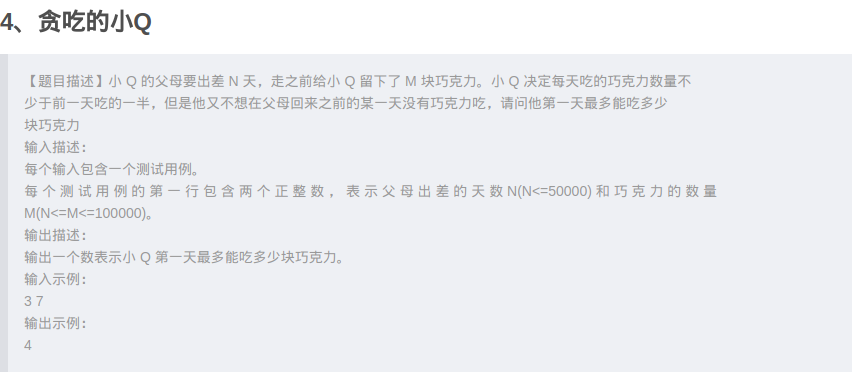


（1）本质上是一道01背包（完全填充背包）问题，可以使用动态规划

（best）但是考虑到物品只有两种，所以可以计算出两个物品“重量”的最小公倍数（两个物品的转换关系），使用一种物品尽可能填充并倒着while找下来，直到找到一种符合的解法。于是接下来的解法可以用最小公倍数的转换关系推出来。每一种解法比如2 3 3 3到底实际对应几种，还要看排列组合的运算。相加所有的解法数目便是最终答案。



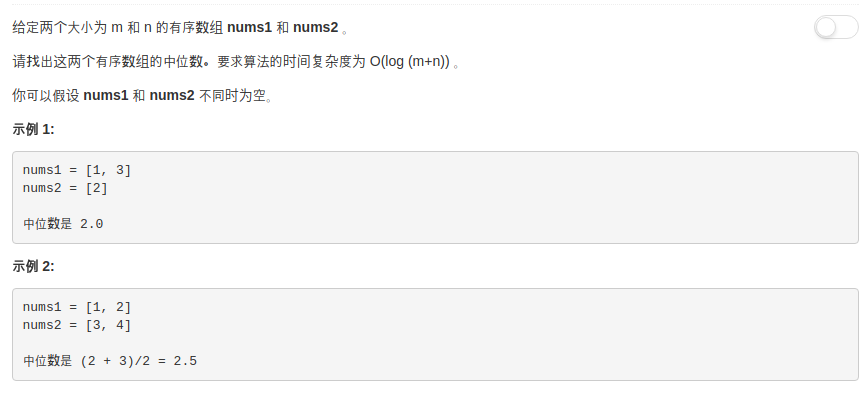
对机器与任务按照各自属性分别排序。从头走开始匹配，只要可以并且收益超过已经设置的任务，就重新设置机器对象的执行任务。



如果使用等比数列的公式，会因为浮点数的位数问题导致不能AC。

因为本质上是一个范围内寻找一个满足条件的数，所以可以使用二分查找。我想到对于数列一定长度元素求和的滑动窗口。

整数的反转（123->321）、取绝对值（int的正负范围略有不同）等都可能会溢出。



从题目的暗示看出来考点：需要一个用于多个有序数组的二分查找。

核心思想：如果我们去掉数组比中位数小的k个数，再去掉比中位数大的k个数，得到的子数组的中位数和原来的中位数相同。

a = [1,2,3,4,5] a的中位数为3

b = [6,7,8,9,10,11] b的中位数为8,9

a 的长度小于b（不选b是存在b的一半比整个a还长的情况），所以选择使用a的中位数长度，b[b.length-a.length/2]=9（对应b数组中倒数第三个数字），因为3<9，所以删除a中间比3小的，删除b中比9大的，最后得到

a = [4,5]

b = [6,7,8]

这种算法不断“删”下去，最终就可以找到中位数。时间复杂度符合题目要求。

变形：

求的不是中位数，求第K小或者第K大的数？

将K拆分为两个K/2 ，每次删除一段K/2（K为奇数，比如7，则后面应该删除3个元素）...更新拆分长度...不断往复

这种思想可以用于归并排序、外排

想起来大数据中top K，有一位网友会从每一个有序文件中都取出top K，最后对所有取出数据排序。在时间级别上，和我的方法一样，但是多乘了一个文件数量。



（0）O(n) Manacher 算法：

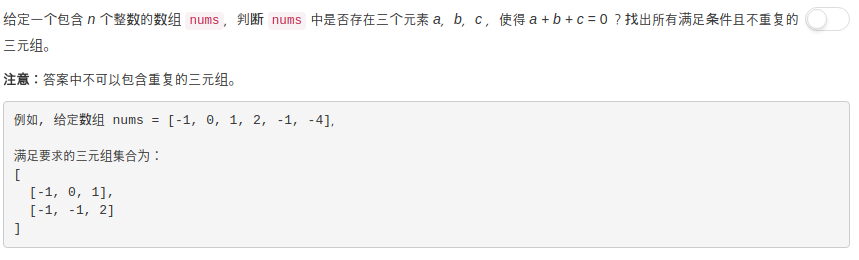
<https://www.cnblogs.com/z360/p/6375514.html>

<https://articles.leetcode.com/longest-palindromic-substring-part-ii/>

1. O(n) 寻找回文子串的对称轴，并从对称轴出发一直找下去（需要分奇偶长度的回文串情况考虑？baaaaac的识别？代码写起来有点复杂）（借鉴Manacher 算法的做法，转为奇数长度的回文），记录找到的最大回文子串，其实思想是动态规划。类似于KMP算法的处理，然后可以跳着移动对称轴。我认为是O(n)的原因是因为每一个后面字符都需要对称轴前面的字符判断一遍。

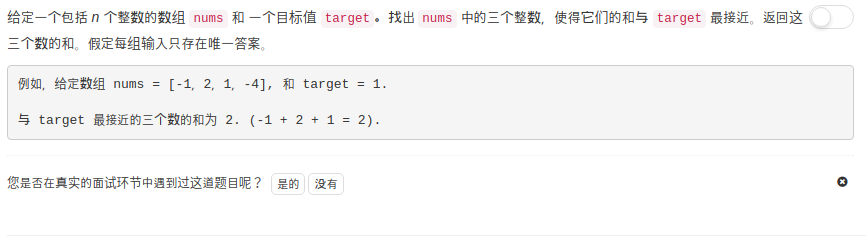
对于连续相同的字符情况不能直接跳跃，要作为特殊情况处理

（2）O(n^2) 像找最长重复子串一样构建一个二维数组，回文串在这个矩阵中会是一个左下右上的对角线形式



1. 本题先拍一个序，通过二重循环遍历计算两个数的和（或者用组合的思想来写循环），再利用二分查找寻找可能存在的第三个值，通过二分查找将时间复杂度下降到n^2。
2. 双重查找表也是这个时间复杂度。
3. a+b=-c 这时候只需要将a+b的值求出来，然后使用查找表就能达到O(n^2)
4. 排序之后，求两数之和的双向走的思想：<https://www.jianshu.com/p/0399dbefd67e>

变题：因为使用了最接近所以不能使用查找表思想



变题2：如果是寻找两数相加结果最接近0，应该排序以后从0或者正负分界的地方分别往两边走。两数相加等于任何一个数都可以类似。但是基于Hash的Map实现的查找表是最快的方法。

变题3：如果是四个数相加等于0，转换为a+b=-c-d，所以时间复杂度只需要O(n^2)即可。如果需要返回到底是哪几个数字相加，因为我们只需借助的是查找表的有无，使用unordered\_map<int, int>，比如将map[a+b]赋值为a，最终还能够求出b。

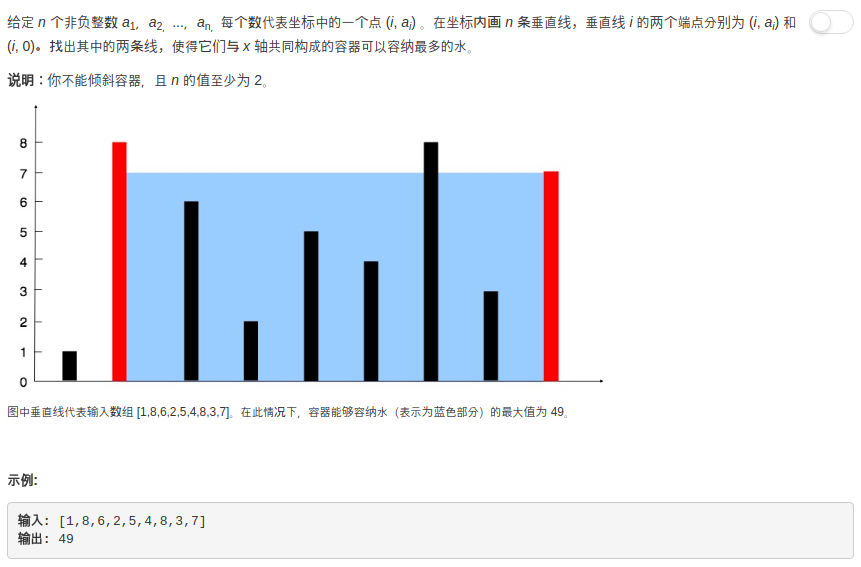
给出一个字符串数组，将其中所有可以通过颠倒字符顺序产生相同结果的单词进行分组。

For example, given: [“eat”, “tea”, “tan”, “ate”, “nat”, “bat”],   
Return:

[   
[“ate”, “eat”,”tea”],   
[“nat”,”tan”],   
[“bat”]   
]

相当于是按照单词中字母出现的次数分类。

不过最好的办法是将字符串排序处理，而不是按照上面所说的进行统计。



在由线段长度构成的数组中使用两个指针，一个放在开始，一个置于末尾。 此外，我们会使用变量 maxarea 来持续存储到目前为止所获得的最大面积。 在每一步中，我们会找出指针所指向的两条线段形成的区域，更新 maxarea，并将指向较短线段的指针向较长线段那端移动一步。

复杂度分析：

时间复杂度：O(n)，一次扫描。

空间复杂度：O(1)，使用恒定的空间。

字符串相乘： <https://www.cnblogs.com/grandyang/p/4395356.html>

先使用int数组存储，然后转为每一个位的char



思路还是很简单的，但是如何将思路转换为代码呢？

（1）需要先处理abnormal的情况，即所选范围还包含特殊情况（如6 7 0 1 2这种变化），一直处理直到选定出一个正常的渐变范围或者已经直接选出目标值。

（2）其实只需要一个if，分别找到往左走或者往右走的条件

排列相关问题：

相同元素不能相邻的排列数

求排列数应该是使用递归，又因为重叠子问题的存在优化为dp，相同元素相邻不能意味着状态中多一个变量去记录上一次选择的元素。

DeepinScreenshot_select-area_20181210201613

ai表示对应元素的个数

全排列最常见的写法是DFS



上图这种写法，空间复杂度是O（1），需要注意的是代码中的交换不能改为异或，因为可能需要自己与自己交换，一旦异或就会改变原有数据。使用异或等式交换两个数的前提是这两个数在同一种类型下不相等。异或会将相同数字相消，可用于去重。

如果是按照每次抽取一个数字去写，就会比较占用空间复杂度，因为要不断保存过程中的数据，而且这种可能性抽完要为了下种可能性还要将数据放回去，算法思想上算是回溯：

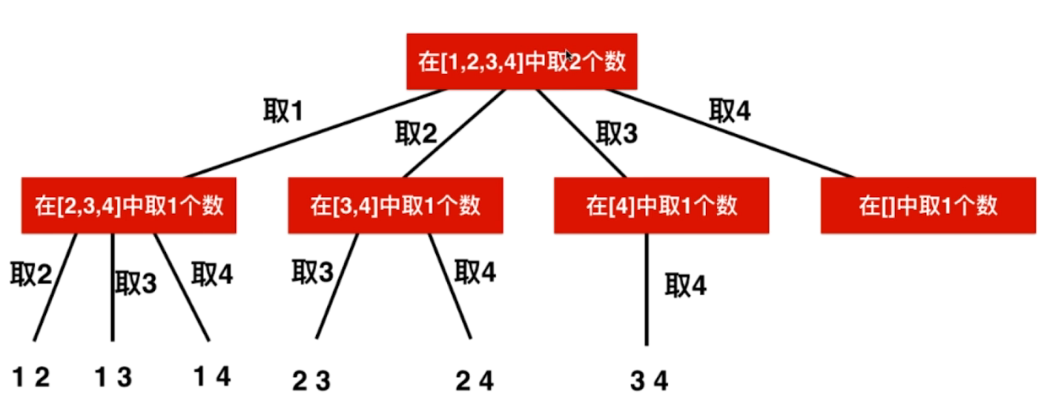


考虑如果是从n个数字中选择K个数字进行排列呢？相当于先进行选择，然后再执行全排列。

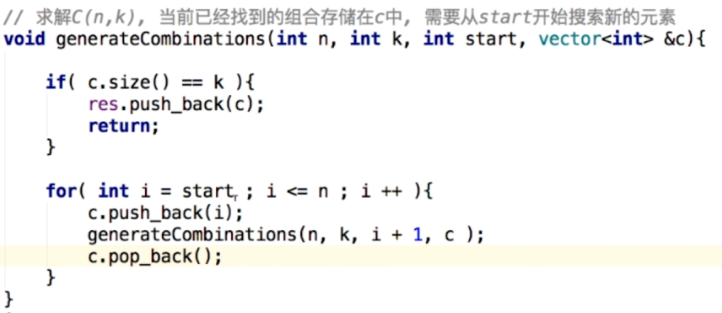
如果排列中存在相同的元素：

从n个数字中选择K个数字进行选择？

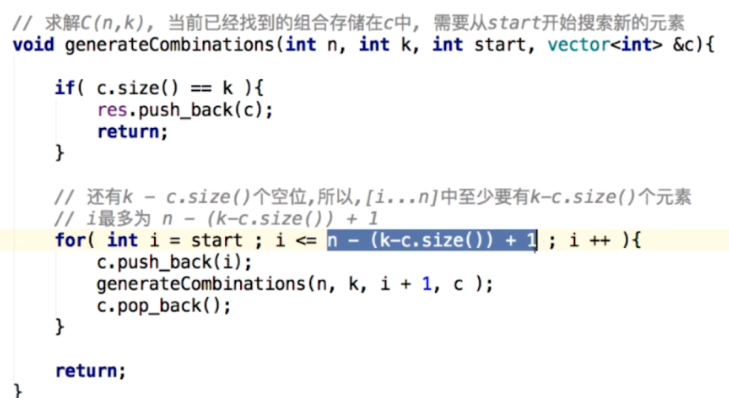
这个算法是经过剪枝的回溯，要将变量pop\_back出来

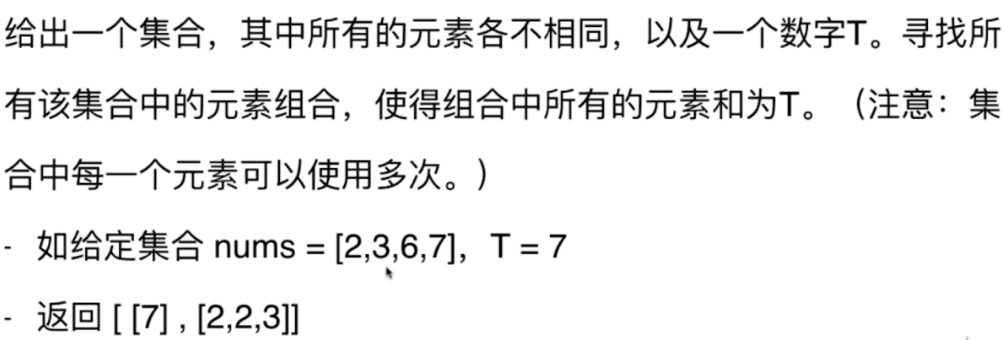


关键在于索引的走动，下图函数中第三个参数：



上面的函数可以对循环的上限进行剪枝操作：





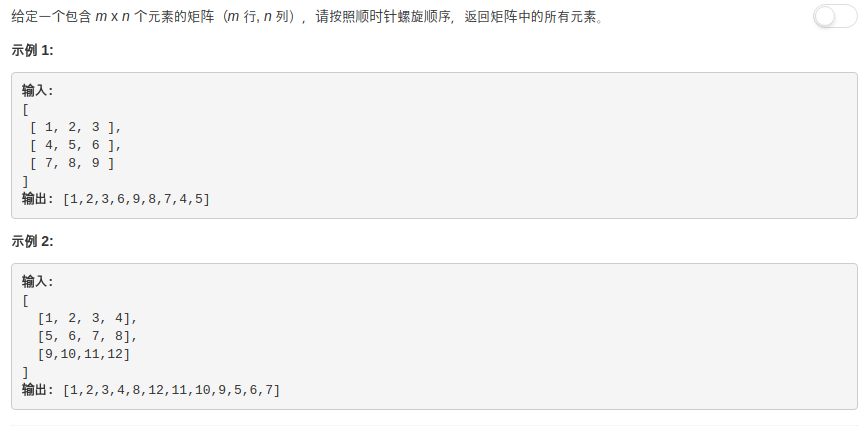
变题：

如果数组中元素只能使用一次，可使用回溯+排序解决这个问题

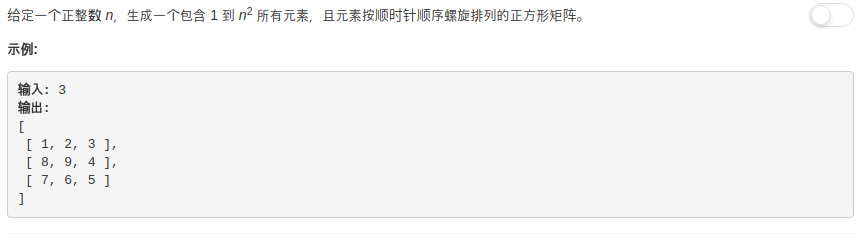
如果不仅仅每一个元素只能够使用一次，并且限制了元素使用个数，就变成了组合问题

一个存在重复元素的集合，问所有子集情况：

选择可以按照选和不选分叉为一棵树，想到Huffman的01树，想到我对选择用0到一个数字0xFFF之类的一层循环来遍历所有可能的选择。



一圈一圈走下去，但是截止条件是什么？如果监测圈的碰撞太麻烦了，最好是看你是不是走过了mn个元素。

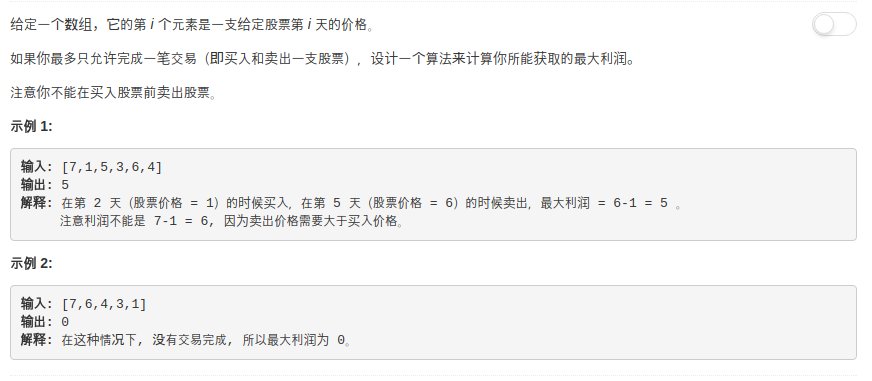


这种矩阵可以用坐标判断截止条件



将矩阵最右列与最下列初始化为1，每一个格子为下方各自与右边格子的和。

蠢的做法可以是当做一个有向图，进行遍历。



只需要走一遍哦～ 时间复杂度：O（n）

变题：





每一颗子树都可以视为子树内可以提供的最大值，相当于正负数组寻找最大连续子序列和的问题。需要走一遍深度遍历。

尾递归的原理：

     当编译器检测到一个函数调用是尾递归的时候，它就覆盖当前的活动记录而不是在栈中去创建一个新的。编译器可以做到这点，因为递归调用是当前活跃期内最后一条待执行的语句，于是当这个调用返回时栈帧中并没有其他事情可做，因此也就没有保存栈帧的必要了。通过覆盖当前的栈帧而不是在其之上重新添加一个，这样所使用的栈空间就大大缩减了，这使得实际的运行效率会变得更高。

以尾递归方式实现阶乘函数的实现：

int facttail(int n, int res)

{

if (n < 0)

return 0;

else if(n == 0)

return 1;

else if(n == 1)

return res;

else

return facttail(n - 1, n \*res);

}

//考虑到栈帧不会被保存：尾递归还要求函数中只能出现一次自己调用自己的过程，不能像递归写快排那样 。函数中的变量都转换为参数传递。

下面函数中对n的范围处理不到位

1)线性递归：

long recurve(long n)

{

      return  (n == 1) ? 1 : n \* recurve(n - 1);

}

2)尾递归：

long recurve\_tail(long n, long result)

{

          return (n == 1) ? result : recurve\_tail(n - 1, result \* n);

}

索引结构中相似性查询有两种基本的方式：一种是范围查询（range searches），另一种是K近邻查询（K-neighbor searches）。范围查询就是给定查询点和查询距离的阈值，从数据集中找出所有与查询点距离小于阈值的数据；K近邻查询是给定查询点及正整数K，从数据集中找到距离查询点最近的K个数据，当K=1时，就是最近邻查询（nearest neighbor searches）。

已知有很多很多个点（就当是二维的吧）坐标，现在新加入一个点，求离这个点最近的点坐标：

1. 线性扫描，是个人都能想到，而且看起来时间复杂度是O(n)，但是没有利用结构的已有情况
2. 点坐标一般会呈现聚集状态，建立索引来划分空间。根据划分的空间是否有混叠可以分为Clipping和Overlapping两种。前者划分空间没有重叠，其代表就是k-d树；后者划分空间相互有交叠，其代表为R树。

k-d 树： <https://www.cnblogs.com/eyeszjwang/articles/2429382.html> 任何一种将点视为直线来划分空间的办法都可以通过回溯解决问题。一开始按照方差与中位数来选点只是为了能够划分的更好，减少之后的回溯长度。范围查询与最邻近点查询都是不错的，但是top K 临近查询不如聚类与分类好。

BBF（Best Bin First是一种改进的k-d树最近邻查询算法）标准的k-d树查询过程可以看出其搜索过程中的“回溯”是由“查询路径”来决定的，并没有考虑查询路径上数据点本身的一些性质。还支持设置超时时间，如果超时会将找到的最好结果返回。https://www.cnblogs.com/eyeszjwang/articles/2437706.html

R树：B、B+树是将一个直线划分为多个线段，R树对应是B在高维空间上的扩展。

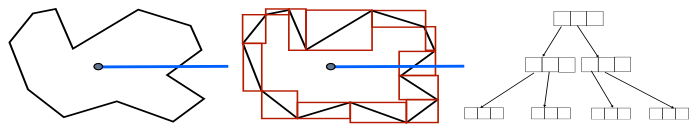


上图中先将相邻的餐厅通过一个小矩阵汇聚在一起，之后反复同一个大矩阵汇聚之前的小矩阵。矩阵的嵌套就是k-d树介绍中说的空间的重叠。

判断一个点是否在多边形内部：

1. 射线：从这个点发射射线，与多边形相交之后，如果交点个数是奇数（内部）还是偶数（外部）。要求这个射线不能刚好经过角坐标。http://alienryderflex.com/polygon/ poly数组中记录的角坐标，如果是相邻就表示角之间存在连线。
2. 图像识别，将内容转换为01矩阵，就成了判断目标点在二维矩阵中的值是0还是1

生活中因为地图中多边形（地域围栏）判断起来其实运算量很大，所以会将多边形转为矩阵进行判断。之后就可以使用R树，先判断用户是否在矩阵中，然后判断是否在多边形中。如何提高对复杂多边形执行射线法的计算效率呢？同样使用R树索引！笔者在实际应用中对边数较多（如超过1万）的多边形的边再单独进行R树索引，具体如图6所示，首先对多边形的每条边构建最小外包矩形，然后在这些最小外包矩形基础上构建R树索引（R树索引上的外包矩形未画出），这样射线法求交点的时候首先通过R树判断射线是否与外包矩形相交，最后对R树粗筛后的边进行精确求交判断，时间复杂度从O(N)降到O(Log(N))，大大提高了计算效率。



字符串常见问题：子串搜索KMP算法、最长公共子串、最长回文子串算法、编辑距离问题、字符串与栈

字符串的编辑距离（假设删除、替换、添加），任何一个操作的代价都是一样的。求两个字符串之间能够一样的最小编辑距离？

d[i][j]   =   min(d[i-1][j]+1,d[i][j-1]+1,d[i-1][j-1]+(s1[i]  ==  s2[j]?0:1));

时间复杂度*O*(*MN*)

空间复杂度可以降到*O*(2∗*max*{*M*,*N*})

#include <stdio.h>

#include <string.h>

#include <algorithm>

using std::min;

int lena, lenb;

char a[1010], b[1010];

void read() {

scanf("%s%s", a, b);

lena = strlen(a);

lenb = strlen(b);

}

int dp[1010][1010];

void work() {

for(int i=1; i<=lena; i++) dp[i][0] = i;

for(int j=1; j<=lenb; j++) dp[0][j] = j;

for(int i=1; i<=lena; i++)

for(int j=1; j<=lenb; j++)

if(a[i-1]==b[j-1])

dp[i][j] = dp[i-1][j-1];

else

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

//最右下角格子是矩形中其余三个格子最小值加1

//如果右下角的值是从左上角变来的，相当于是通过编辑变换来的

//如果是从上方或者左方变来的，相当于使用添加或者删除得到

printf("%d\n", dp[lena][lenb]);

}

int main() {

read();

work();

return 0;

}



本题的区别在于不同操作基于了不同的权重，其中删除与添加的权重应该是一样的，因为一个字符串通过添加变成另外一个字符串等价于另外一个字符串通过删除变回来。如果不一样的话，比如添加代价为3，删除代价为2，这是出题者的干扰，直接认为 添加代价=删除代价=2 。

将一个字符串按字符逐步压入栈中，在压入的过程中随时可以弹出，问最终能得到几种情况？

解：<https://www.cnblogs.com/jiayouwyhit/p/3222973.html>



数字不包含1，也不会出现左下角与右下角两个特殊情况。”23”对应9种字符串。

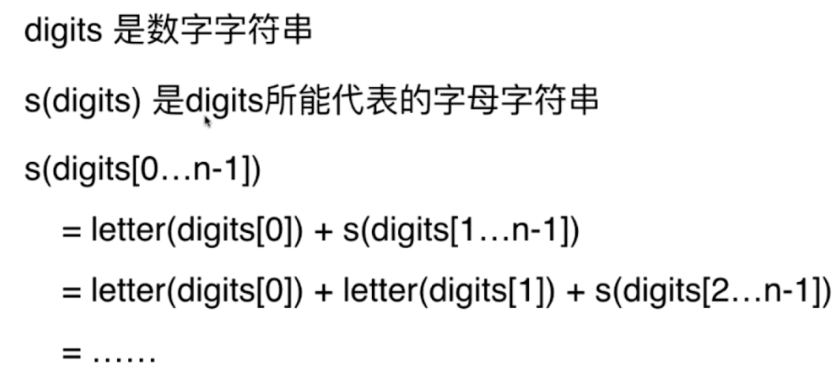
解：

C++中所有的字符元组，直接使用一个字符串数组存储就好

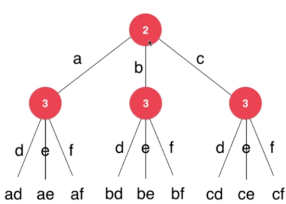
使用迭代（方法一）需要不断new存储对象（虽然本问题因为各个数字元组长度一样，所以直接统一存储在一个vector中就好），递归（方法二）直接中间传递参数就好

（1）我想的，其实就是将数字替换为数字对应的字符元组中的字符。这道题用迭代也是可以解决的，因为每一个元组的长度都是3 。

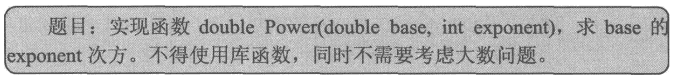
（2） 思想上与(1)一样，但是写成了递归的方式，这种递归与上面字符串压栈求种数很像



相当于DFS走了一颗这样的树：



分裂复制式思想（比如ad、ae都依赖于其那面的a），都可以借助递归的传参实现



同样，求幂时，我们没有必要使用时间复杂度为O(N)的连乘方式，这里N等于exponent的绝对值。而是可以使用递归的方式。

因为当n为偶数时，a^n = a^(n/2) \* a^(n/2)；

当n为奇数时，a^n = a^((n-1)/2) \* a^((n-1)/2) \* a。

时间复杂度为O(logN)，上面计算过的值可以存在数组里面，因为一定会用到，因为任何一个自然数都可以表示为二进制呀。

将a一次次乘上去是一种很愚蠢的办法

注意exponent的特殊取值

当x中包含偶数个1返回1，否则返回0 :

//首先明白是用异或，然后就是考虑到以异或的整体移位

bool OddOnes(int x)

{

x = x ^ (x >> 1);

x = x ^ (x >> 2);

x = x ^ (x >> 4);

x = x ^ (x >> 8);

x = x ^ (x >> 16);

return !(x & 1);

}

让我们生成一个掩码，用来标识出x中最左边的那个1

//除了最左边的1，两边都弄成全1或者全0，最后再弄成全0.

int leftmost\_one(unsigned int x)

{

unsigned int mask = x;

mask |= mask >> 1;

mask |= mask >> 2;

mask |= mask >> 4;

mask |= mask >> 8;

mask |= mask >> 16;

mask = (~mask) >> 1;

return x & mask;

}

static final int tableSizeFor(int cap) { //threshold更新函数，判断最高位的1到哪里了，还不如直接用与运算

int n = cap - 1;

n |= n >>> 1;

n |= n >>> 2;

n |= n >>> 4;

n |= n >>> 8;

n |= n >>> 16;

return (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;

}//使用无符号移位，因为n默认要求是>0的，虽然使用的是int类型

有一个数组，只有一个数字出现1次，其余数字出现3次，要求使用O(N)的时间复杂度且O(1)的空间复杂度找出这个出现一次的数字。

<https://blog.csdn.net/ojshilu/article/details/14762791>

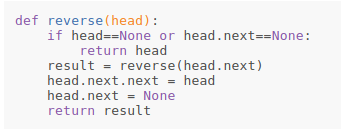
使用java计算两天之间的天数差，存在以下两个坑：

（1） 如果使用Date，月份的索引是从0开始

Date startDay = new Date(Y,M-1,D);  
Date endDay = new Date(Y2,A-1,B);  
//2017年7月1号 到2017年8月30号 或者到9月1号 都是61天  
long time = (endDay.getTime() - startDay.getTime())/(24\*3600\*1000);  
result[i1] = time;

1. 某一个时间点之前的天数会计算出现偏差

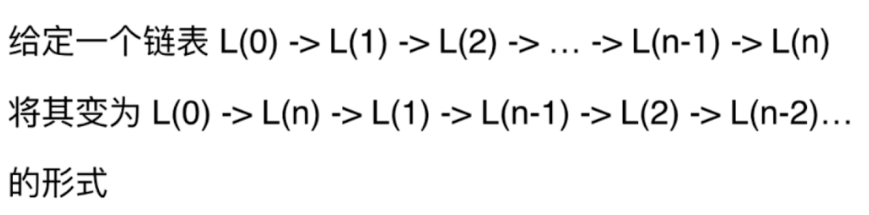
链表逆置的递归写法：



但是这段代码没有被头条面试官认出来了，估计因为受到了head.next.next = head的干扰。

如果是将链表中的指定范围（小心范围有坑）内的链表反转，就需要通过正常的写法。

单链表删除一个节点没有必要非要得到这个节点之前的节点，从这个节点开始直接复制就好。



关键是要找到中间元素，单链表中间元素只需要一次遍历（两个指针，一个走两步，一个走一步）（这种办法从指令数上看其实差不多）。

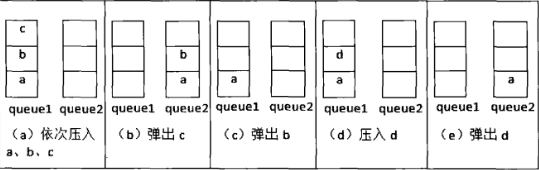
判断链表是不是回文链表：  
（1）先遍历一次链表得到链表长度。然后第二次遍历到链表中间，并在遍历的时候进行逆置

1. 将数据保存到数组中等
2. 如果链表中存储的是数值，可以通过一定的转换存储到一个变量中

判断一个数是不是happy number，不断将数字替换为该数字中各位数字的平方和，如果最终能够得到1就是。

这种题目如果最后得不到1，说明基本是因为死循环了。使用基于Hash的map查重。

两个队列实现一个栈：



两个栈实现一个队列倒是很简单合理

如果问到解析JSON的数据结构是什么，因为解析JSON通过递归解析，所以是栈。

有个村庄，每个人的年龄都是0-100岁，去做采访，第一行输入n,k。 n是村里有几个村民 k是采访了这村几个人  然后后面k行每行3个数 比如 1 2 5就是人2比人1大5岁 最后输出最大的人的年龄减最小的人的年龄，如果他们说的话有不符合逻辑就输出-1 ？

解：

这是一个图的问题，需要你首先确定所有没有前驱的节点，这些节点的年龄可直接设置为0 。按照图的关系一个个设置就好。要么设置一个数组int age[n];来存储所有，要么多个插入排序，每一个插入排序对应一个连通分量。

可能出现的特殊输入：

2 3 1

1 2 3 //这时候2有了前驱

4 5 99

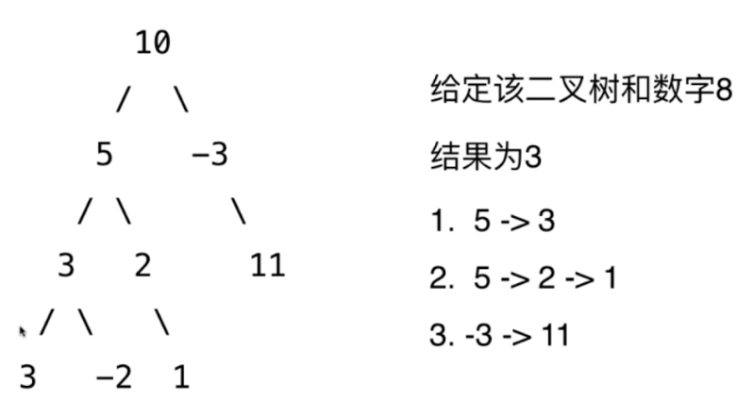
稳定婚姻匹配：<https://www.cnblogs.com/AndyJee/p/4986741.html>

给出一个二叉树以及一个数字sum，判断是否存在一条根到叶子的路径使得节点上数值和为sum：

DFS是肯定的了

1. 我考虑到递归，所以想着将过程中每一个数字存储到stack等中，或者直接使用stack
2. 递归的过程中，对sum进行加加减减（真减或者作为参数传递），变换需求。这种办法对这个问题用递归真正的分法。

变：  
如果没有限制是从根节点出发，也没有限制是叶子节点，只是限制这条路径一定是向下走的：



如果限制了一定会到叶子节点，则只需要在遍历的时候通过返回值来传递就好；如果限制是从根节点出发，则应该在遍历的时候传参。

两种思路：

1. 每次递归函数中会调用中可以考虑使用减去上一个节点数值的sum以及sum，比如在第二层数值为5，这个节点上，应该根据两种sum（-2、8）对子节点使用递归。如果要打印经过的路径，那么还需要传入一ArrayList。

上面这个思路看起来没问题，但是实际上写起啦，类似于

public static void findSum(Node node, int sum, ArrayList<Integer> record) {  
 record.add(node.value);  
 if( node.value == sum) {  
 record.forEach(System.*out*::println);  
 return;  
 }else {  
 if(node.left!=null) {  
 *findSum*(node.left,sum-node.value, record);  
 *findSum*(node.left, sum, new ArrayList<>());  
 }  
 ...  
 }  
}

可以看到每一个递归函数调用了两次自己，说明调用是按照2^n成长的。仔细想一下，发现同一个节点作为根节点 会被重复多次考虑。

1. 还有一种思路是将这个寻找过程分离出来，一个DFS遍历函数，每一个节点都会再调用一个尝试沿着路径寻找求和为sum的值。

求二叉树中所有左叶子节点上数值的和：

相当于遍历求和所有叶子节点的时候，加一个判断是不是左叶子节点或者忽略右叶子节点。

如果要返回二叉树从根到叶子节点的路径：

1. 平时想到的是使用vector在传参过程中记录
2. 但是直接递归函数返回一个vector也是可以的，存储的顺序支持倒序存入还是正序存入。不仅仅是vector，举一反三。

判断一颗二叉树是否左右对称：

其实很简单，只需要在第二层之后对遍历进行划分就好。

给定两个节点，寻找这两个节点的最近公共节点：

1. 老老实实遍历一遍的同时记录路径
2. 反转二叉树的连接

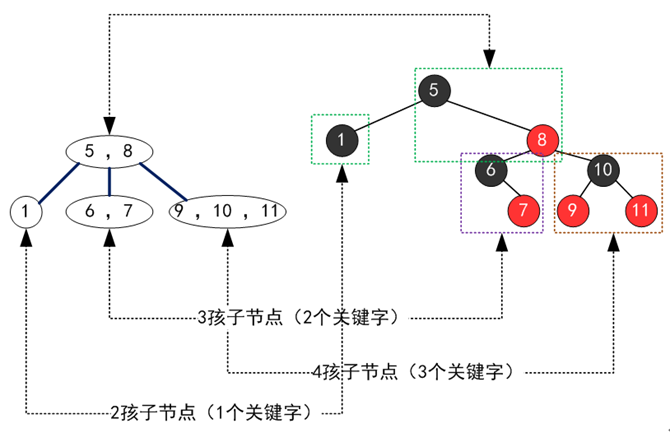
变题：

如果这两个节点是在二分搜索树中？

根据两个节点中的值，结合对二分搜索树的遍历能够直接找到开始分叉的节点（最近公共父节点）。

AVL的维护：<https://blog.csdn.net/collonn/article/details/20128205>

与红黑树等价的2-3-4树： 形态上存在多种对应关系

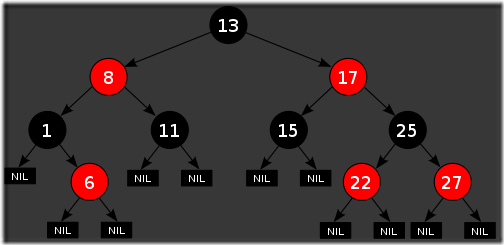


二叉搜索树 来排序 类似于堆排，在排序的过程中维护树（插入与删除都是logN级别），目前使用红黑树比较好，因为树不容易发生大变化。

红黑树（RB-Tree）是一种在插入与删除过程中不容易变化的二叉搜索树，红黑是用非严格的平衡来换取增删节点时候旋转次数的降低，任何不平衡都会在三次旋转之内解决，而AVL是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多。我自称除了NIL节点之后的最底层节点为底部节点。

红黑树的 染色规则：

1. 节点必须是红色或者黑色。
2. 根节点必须是黑色。
3. 叶节点(NIL)是黑色的。（NIL节点无数据，是空节点）
4. 红色节点必须有两个黑色儿子节点。所以红色节点如果不是底部节点一定是饱满的，同时红色节点不能相邻。
5. 从任一节点出发到其每个叶子节点的路径，黑色节点的数量是相等的。

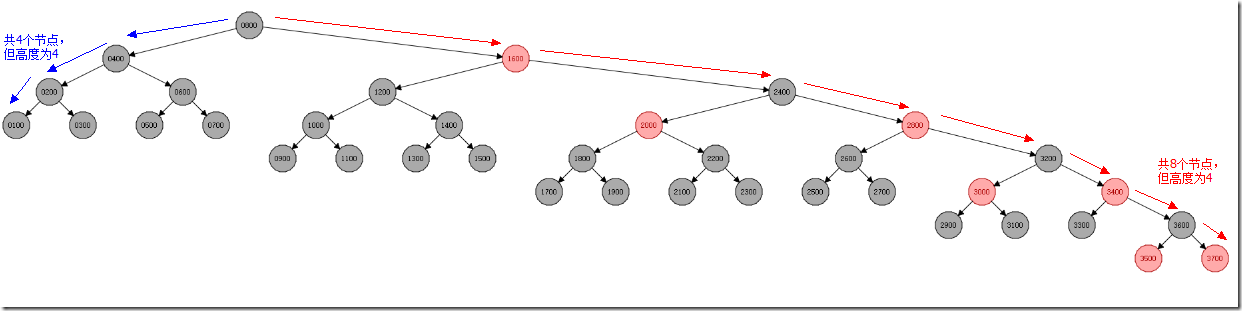


第5点表明： 从根节点到最远的叶子节点的路径长与到最近的叶子节点的路径长度相差不会超过2倍（差值来源于红色节点）。按照黑色节点来计算“高度”，若只是看黑色节点， 就是一个AVL 树。 同时表明了红黑树是一颗“平衡”树，不存在全节点的单链。

理论上最短路径：全黑色节点、

理论上最长路径：黑色节点与红色节点一样多

所以树中任意两条路径中的节点数相差不会超过一倍

****

**由此断定时间复杂度的级别；**

一棵有n个内结点的红黑树的高度至多为2lg(n+1) 逆否命题等价于

高度为h的红黑树，它的包含的内节点个数至少为 2h/2-1个（之所以说是内节点，是因为存在NIL的叶子节点） <=全都是黑色节点就是这样的树

2lg(n+1)理解为最短路径的两倍，就是上面这图片的情况

**添加节点与删除节点：**<https://blog.csdn.net/xiaojun111111/article/details/51898486>

基本原则是向红黑树中尽可能添加红色节点，因为这能够减少红黑树的变化；删除的时候，没有其余的原则，因为几个基本条件，所以删除的时候变化是唯一的，所以网址中的示例在删除的过程中甚至出现了全黑的树，之后删除了一个黑色节点就必须要在其他分支上转换红色节点。其实变往全黑树的那一步我不是很认同，因为我认为只有尽可能存在红色节点这个原则是对的，所以应该将第二层节点全都变成红色。

可以发现，添加节点要么是直接将红色节点添加或者需要将父节点中的颜色进行改变，这时候其实基本趋势是将红色上移，一直到根节点。如果是删除，就会直接删除红色节点或者有一种将红色节点下移的趋势。

打表：创建一个表记录之前过程中运算产生的数据，dp用于之后减少运算。

守卫棋盘（UVA11214）：给出m\*n棋盘上的目标点，求最少用几个皇后可以守卫所有目标点。

解：

1. 这是一道暴力搜索的题目
2. 如果将每一个棋盘上每一个目标按照皇后可以行走的路径，上下左右去扩张，然后每一次消除“最棒的交点”。

一个数组中非连续自增子序列最大长度？

{2,7,3,1,8,9,5,6,7} 应是{2,3,5,6,7}长度为5

走到3的时候发现比7小，但是比2大，所以有了{2,7}与{2,3}两种长度为2的；遇到8、9，变成{2,7,8,9}与{2,3,8,9}；再遇到5变成{2,7,8,9}与{2,3,8,9}与{2,3,5}；最后得到{2,3,5,6,7}

优化：

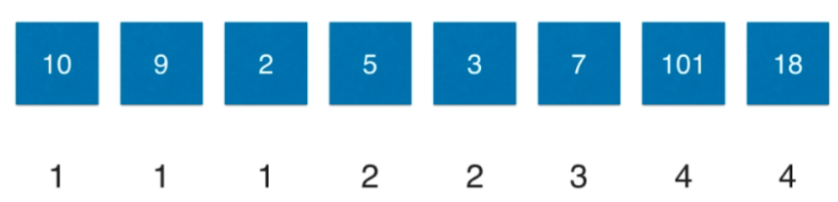
有了{2,3,5}就可以删除{2,3,9}

或许重复节点可以用有序多叉树代替

变题：

给一个长度为n的序列，要求删除一个连续子序列，使剩下的序列有一个长度最大的连续递增子序列。例如{5,3,4,9,2,8,6,7,1}删除{9,2,8}就有了最长连续递增子序列{3,4,6,7}。

1. 类似于上面使用分裂记录，此法最差的情况为O(n^2)
2. 在O(n)的时间内找出所有连续递增子序列，尝试合并。时间复杂度O(n^2)
3. dp：每一个元素对应一个数值记录到该元素为止的最长上升子序列，过程中记录产生的最大值：



给定一个长度为n的01串，选一个长度至少为L的连续子串，使得子串中数字的平均值最大。如果有多解，子串长度应尽量小，如果仍有多解，起点编号尽量小。

解：因为 (b+d)/(a+c)>b/a => b/a < (b+d)/(a+c) <d/c，所以每一个只需要判断L~2L的距离

给定多个区间，问至少删除多少个区间能够使区间之间不重合：

1. 相当于求区间不重合最多能够多少：

贪心计算，按照“截止时间”来选择

1. 动态规划：和求上面非连续递增子序列一样的想法，将区间按照区间的start、end排序，然后每一个元素记录一个数值去表示当前所有区间能够构成得到最长不重叠区间序列。

把一个包含m个正整数的序列划分成为K个非空的连续子序列，最终使得每一个子序列的和的最大值最小。

1. 想过将数组均分，之后调整边界。这种方式十分不合适，就算是人脑考虑也算是更加麻烦了。没有办法确定循环的终止条件。
2. 设所有数之和为M，则时间复杂度为O(nlogM)

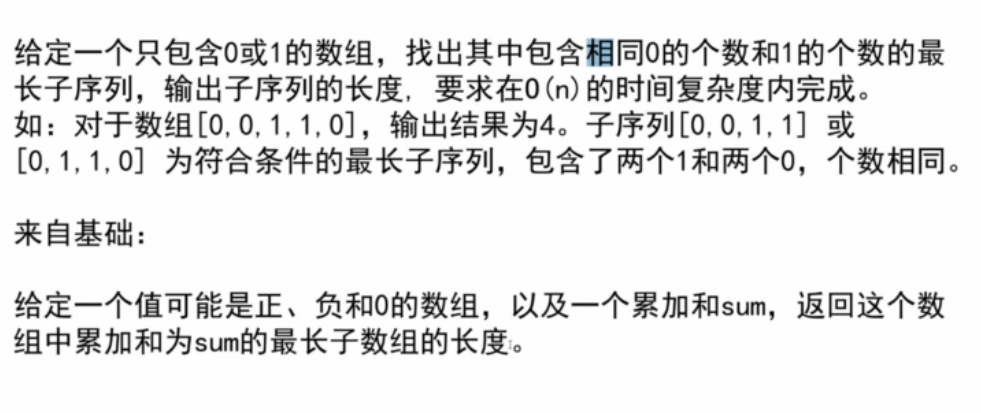
因为每一个子序列都要小于一个值x，所以通过二分法寻找这个x是最快的办法。

如果直接认为只需要除最后一段的每一段都尽可能刚好大于M/K，只能满足大部分用例，但是[3,2,2,4,1,4], K = 3会出错。

如果加上条件“存在多种情况，则越前面的子序列的和越小越好”。最好的办法先用方法（2）确定划分的最小值，比如先确定将[1,2,3,4,5,6,7,8,9,10] 切分成5段，最大值都是15，再去寻找第二种可能：：

1. [1,2,3,4,5] [6,7] [8] [9] [10]

2. [1,2,3,4] [5,6] [7,8] [9] [10]



使用动态规划是一定的，但是如何记录数据来避免重复的计算是一个难点，如果不考虑这是一道题目（因为这通常意味这道题目可以O(n)时间解决）

按照下面这个例子：

index : 0 1 2 3 4 5

value : 4 3 2 5 1 6

sum : 4 7 9 14 15 21

使用hash表存储sum与index的对应关系，如果要求的子序列和为6，当到index=4的时候，和为15，存入hash中，并且按照（15-6）在其中寻找是否之前已经出现过这样的sum。

寻找自序列中0和1的个数一样，可以将0变成-1，然后相当于求寻找和为0的子序列。本题目表明了这种最长连续子序列问题的本质，可以转换为两个索引0开始的数组之间的关系。

输入一个1~n（1<=n<=300）的序列，用不超过2n^2次操作将之变成升序。操作只有两种：1.交换前两个元素；2.把第一个元素移动到后面。对于输入序列4231，需要经过操作12122变成升序。

1. 图的BFS去解决。
2. 贪心思想：

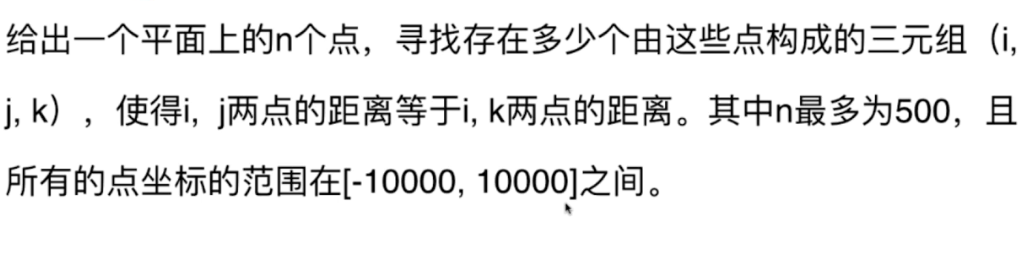
比较前面两个数，小的数应该被传到后面去？{4,1,2,3}就会一直得不到升序序列，需要考虑这种特殊情况。

给定 n 个工作，已知每个工作的工作需要时间 q 和 截止时间 d，问你最多完成多少个工作，每次最多能运行一个工作。

题目中给了两个时间，因为每一个工作的价值一样，所以关键在于截止时间，按照截止时间排序，如果不能完成当前最紧急的工作，就把这部分时间空出来给下一个。

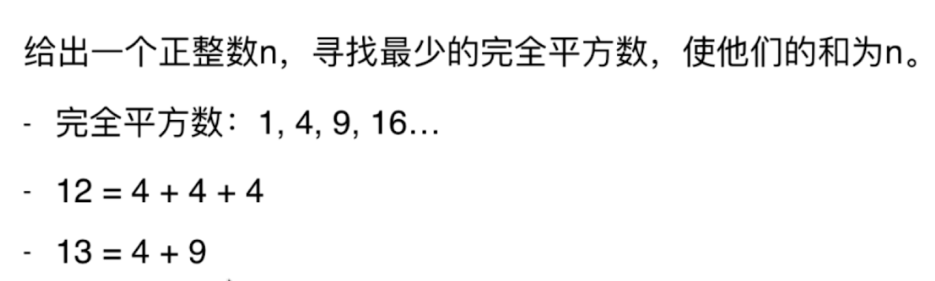
变题：

如果工作的价值与工作需求时间成正比，问最后能够得到的最大工作价值。即要求工人的工作时间尽可能饱和。这时候是一个背包问题。



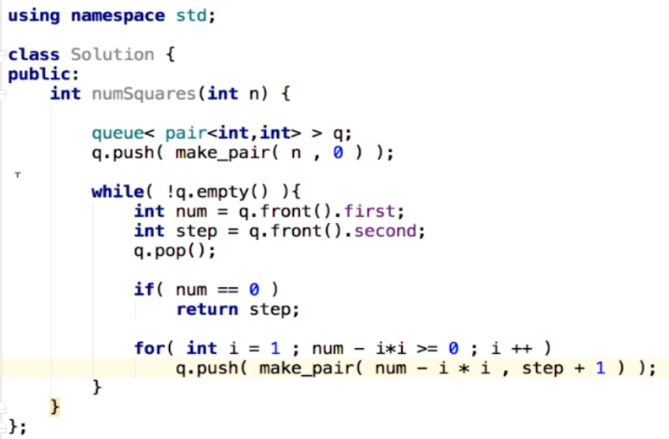
首先需要知道各个点之间的距离关系，为了避免浮点数麻烦，使用勾股定理但是最后并不开方。这一部分是必须经过的运算次数。

判断是否存在相同距离，接着使用查找表是最快的。

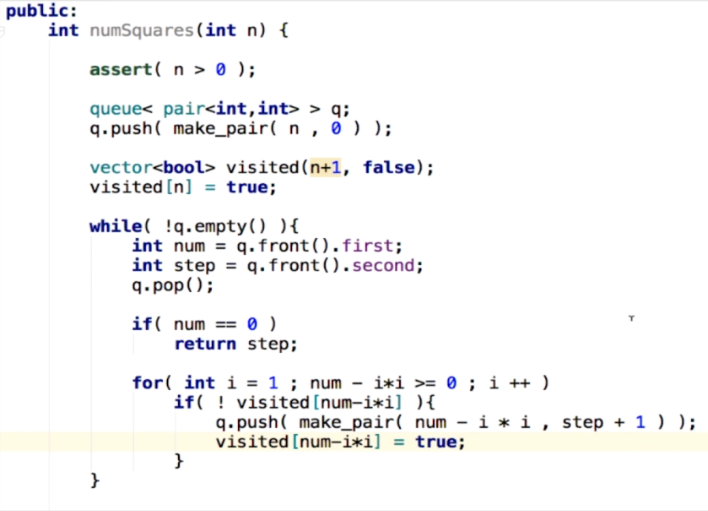


首先确定不存在无解的情况，几乎所有人都会使用贪心算法，但是这种算法因为1

的存在导致无效。如果数字表示一个节点，数字之间差一个平方数就形成一条边，本问题就相当于这个图中的最短路径。

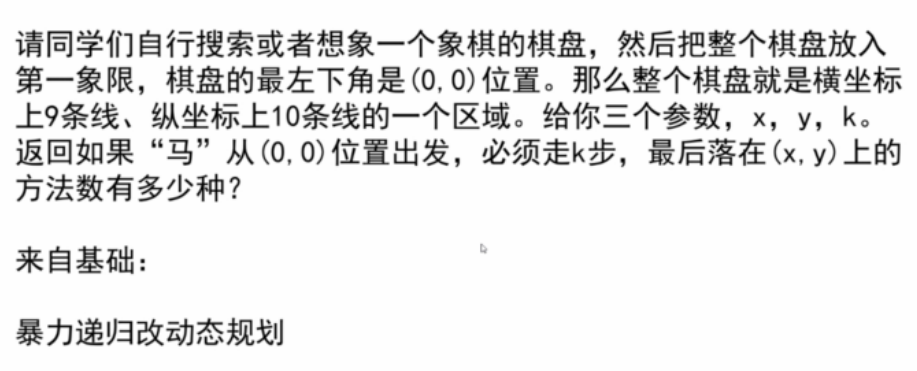


上面的代码很惊艳，倒着直接构造一个网，不过因为重复经过节点会存在性能问题。所以树的BFS不能直接推到图中。本题结合题目，判断是否经过的时候不需要一个set，直接用boolean就能够判断：



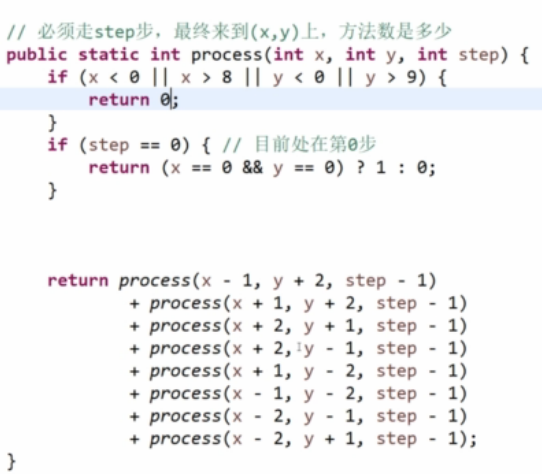
将一个字符串拆分，使得所有拆分结果都是一个回文字符串。

解：比如”aabab”，应该使用回溯进行暴力解法。

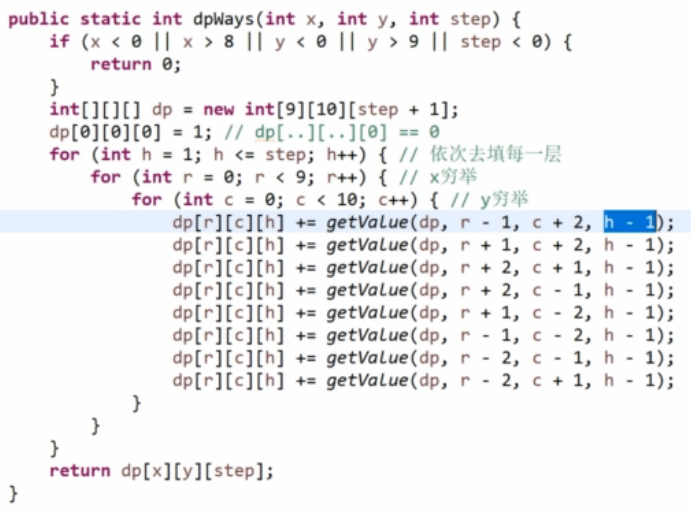


（0）我觉得x、y的增加以及k这个代价限制，是一道背包问题

（1）从马跳到一个位置有八种可能，如果不考虑边界，进行广度遍历或者深度遍历（递归）



可以创建一个三维数组，记录x、y、step，使用n-1 step产生n step：



给定n个点，找出其中最接近的一对点（不同于已知n个点的坐标，向其中加入一个点后找出与这个点最接近的点）：  
对于一维，可以直接先排序然后遍历最相邻的两个点的坐标。

对于二维，直接相邻已经不能保证包含最接近的点对了，所以关键在于需要知道要判断多远的距离？我们使用递归法（当然需要先排序），递归下来直到只有3个点，每一次都是左边的算最小值，右边的算右边的最小值，然后根据算好的左右最小值来看看是否中间存在更小的可能。

对于三维，可以使用和二维类似的算法，但是使用平面去切（而不是去考虑降维

将所有点包在最小凸多边形算法 （以点为端点的边数最少，而不是面积最小）

[Graham Scan凸包算法](https://segmentfault.com/a/1190000000488339)，是计算几何中一种基本算法。https://segmentfault.com/a/1190000000488339 有一个网站是这样寻找第一个点，找y最小的点，如果有多个再找其中x最小的

如果是面积最小，那么所有的点都一定是顶点，但是具体的算法我觉得没有办法，不规则多边型的面积没有办法很好地求。

**Leetcode：**

二分查找类问题：

技巧：

对于奇数长度有序数组，len//2与(len-1)//2 相同

对于偶数长度有序数组，len//2 找到中间那一对偏大值，(len-1)//2找到中间那一对偏小值。不过在二分搜索中，找到哪一个值并无所谓，计算中位数的时候比较关键。

二分查找中 m = l + (r-l)//2效果上即(len-1)//2，(l+r)//2存在整型溢出的问题，可以改为l + (r-l)//2<=> l + (r-l)>>2，如果>>2是无符号右移，还可以等于(l+r)>>2。二分查找比较时的写法上建议target<nums[m]而非nums[m]>=target。二分查找常见变种：“存在重复元素”、“旋转有序数组”、“两个有序数组”。

* [两个有序数组求中位数或Kth问题](https://leetcode.com/problems/median-of-two-sorted-arrays)

将分位点转为抛弃数组用不到的一部分，可以是头部，也可以头尾一起抛

* [在有序无重数组中，找到一个数的索引，或者他应该插入的位置](https://leetcode.com/problems/search-insert-position/)

我第一次写：

class Solution:  
 def searchInsert(self, nums: List[int], target: int) -> int:  
 l = 0 #固定模板  
 r = len(nums) – 1 #固定模板  
  
 if target <= nums[l]: # 这道题的专属边界条件  
 return l  
 elif target > nums[r]:  
 return r + 1  
  
 while l < r: #固定模板  
 m = l + (r - l) // 2 #固定模板 等价于 (l+(r-l))>>2好于 (l+r)//2   
 if m == l or m == r: #固定模板  
 break  
  
 if target < nums[m]:  
 r = m  
 else:  
 l = m  
  
 if nums[l] == target: #固定模板 还有判断nums[r]与target  
 return l  
 else:  
 return r

上面代码可读性的优化空间：

* 1. m = l+(r-l)//2，所以 m可能等于l而在数组长度不为1时一定不等于r，退出循环的条件与退出后代码 仅需要考虑 m==l
  2. target<nums[m]时,完全可以设置 r = m-1，否则l = m+1，但是这个改动会影响很多if-else，因为正是将m赋值给l、r所以需要m==l时break否则可能死循环。得到如下十分简短的代码

class Solution:  
 def searchInsert(self, nums: List[int], target: int) -> int:

if len(nums) == 0:  
 return -1

l = 0  
 r = len(nums) – 1 #l, r = 0, len(nums) - 1  
 while l <= r: # 这里是<=  
 m = l + (r - l) // 2  
  
 if nums[m] < target: #因为最后返回l，所以这里不能是<配r  
 l = m + 1  
 else: # target <= nums[m] 如果再else ==条件，则找不到最左元素  
 r = m - 1

return l # 等价于return r+1

**上面函数的while循环用的是<=，并且因为 l = m+1、r = m -1所以最终r = l-1这样的交错位置。这是必要的，因为我们其实将数组划分成为两部分，left左边的部分和right右边的部分，而且left左边的部分全部小于target，并以right结尾；right右边的部分全部大于等于target。**

**如果仅仅使用<，那么本题还需要补充一段尾部代码来解决边界case，则复杂了**

class Solution:  
 def searchInsert(self, nums: List[int], target: int) -> int:  
 l = 0  
 r = len(nums) - 1  
  
 while l < r:  
 m = l + (r - l) // 2  
  
 if target <= nums[m]:  
 r = m - 1  
 else:  
 l = m + 1  
  
 if nums[l] < target:  
 return l + 1  
 else:  
 return l

**如果使用<，并认为区间比如是左闭右开[left, right) 则复杂了**

class Solution:  
 def searchInsert(self, nums: List[int], target: int) -> int:  
 l = 0  
 r = len(nums) - 1  
  
 while l < r:  
 m = l + (r - l) // 2  
  
 if target <= nums[m]:  
 r = m  
 else:  
 l = m + 1  
  
 if nums[l] < target:  
 return l + 1  
 else:  
 return l

**总之不推荐使用 while l<r 而推荐用 while l<=r**

**从前二分查找会写l=m 与 r=m，这样不好，元素剔除不完全还需要在while中增加break分支**

* [在排序数组中查找元素的第一个和最后一个位置](https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/submissions/)

需要注意这种写法的BinarySearch是基于 “如果插入的话，插入的位置”返回的，所以需要在返回l的时候，增加一些判断直接返回[-1, -1]。

class Solution:  
 def searchRange(self, nums: List[int], target: int) -> List[int]:  
 l = self.BinarySearch(nums, target, True)  
 if l < 0 or l >= len(nums) or nums[l] != target:  
 return [-1, -1]  
 r = self.BinarySearch(nums, target, False)  
 return [l, r]  
  
 # if not found return -1  
 @staticmethod  
 def BinarySearch(nums: List[int], target: int, first: bool = True) -> int:  
 if len(nums) == 0:  
 return -1  
  
 l = 0  
 r = len(nums) - 1  
  
 while l <= r:  
 m = l + (r - l) // 2  
  
 if first:  
 if target <= nums[m]:  
 r = m - 1  
 else:  
 l = m + 1  
 else:  
 if target >= nums[m]:  
 l = m + 1  
 else:  
 r = m - 1  
 if first:  
 return l  
 else:  
 return r # 此时r = l - 1

继续缩短代码优化写法，但是直接简化脑子会转不过来，可读性也变差：找第一个元素位置与最后一个元素的可能位置关键在于等号使用，而且将searchRange函数中的边界判断挪了进来

@staticmethod  
def BinarySearch(nums: List[int], target: int, first: bool = True) -> int:  
 if len(nums) == 0:  
 return -1  
  
 l = 0  
 r = len(nums) - 1  
  
 while l <= r:  
 m = l + (r - l) // 2  
  
 if target < nums[m] or first and target <= nums[m]:  
 r = m - 1  
 else:  
 l = m + 1

# 此时r = l - 1  
 if not first:  
 l -= 1  
 if l < 0 or l >= len(nums) or nums[l] != target:   
 l = -1  
 return l

还有一种不错的姿势，【低理解难度】都理解为搜索左边界

class Solution:  
 def searchRange(self, nums: List[int], target: int) -> List[int]:  
 def binarySearch(nums: List[int], target: int) -> int:  
 left, right = 0, len(nums) - 1  
 while left <= right: # 不变量：左闭右闭区间  
 middle = left + (right - left) // 2  
 if nums[middle] >= target:  
 right = middle - 1  
 else:  
 left = middle + 1  
 return left # 若存在target，则返回第一个等于target的值  
  
 leftBorder = binarySearch(nums, target) # 搜索左边界  
 if leftBorder == len(nums) or nums[leftBorder] != target: # 情况一和情况二  
 return [-1, -1]  
 rightBorder = binarySearch(nums, target + 1) - 1 # 搜索右边界  
 return [leftBorder, rightBorder]

* [有序数组经过旋转（想想数组是环）后进行二分查找，如[4,5,6,7,0,1,2]](https://leetcode.com/problems/search-in-rotated-sorted-array/submissions/)

O(logN) 和二分查找一样利用两个边界点与中间点的比值，只是不能再仅仅依靠与中间值的比较了，需要再与边界值比较。

将数组一分为二，其中一定有一个是有序的，另一个可能是有序，也能是部分有序。此时有序部分用二分法查找。无序部分再一分为二，其中一个一定有序，另一个可能有序，可能无序。就这样循环。考验分类讨论的思维，我使用num[l]、num[m]、num[r]确定数组类型（标准答案仅使用nums[m]与nums[0]），而放入target比较进行确定范围时需要考虑等于号。

class Solution:  
 def search(self, nums: List[int], target: int) -> int:  
 if not nums:  
 return -1  
 l, r = 0, len(nums) - 1  
 while l <= r:  
 mid = (l + r) // 2  
 if nums[mid] == target:  
 return mid  
 if nums[l] <= nums[mid]:  
 if nums[l] <= target <= nums[mid]: #我将第二个改为<=  
 r = mid - 1  
 else:  
 l = mid + 1  
 else:  
 if nums[mid] < target <= nums[r]:  
 l = mid + 1  
 else:  
 r = mid - 1  
 return -1

旋转有序无重数组二分写题技巧：通过**比较nums[l]与nums[m]**判断这一部分是否有序，如果是无序的则另一段m到r一定有序。而非有序的部分一定与原题目一样，可能是一个小型旋转数组。关键在于**如果nums[a]<nums[b]，则不论旋转还是不旋转，这一段一定是有序的**。并注意模板代码中的红色比较符号。

不过官方的写法如下：

if nums[0] <= nums[mid]:  
 if nums[0] <= target < nums[mid]:  
 r = mid - 1  
 else:  
 l = mid + 1  
else:  
 if nums[mid] < target <= nums[len(nums) - 1]:  
 l = mid + 1  
 else:  
 r = mid - 1

需要注意的是，如果输入就像没有经过旋转的有序数组一样，并且target比最小的还小那么最终的l并不是插入位置0，而是成了l=len r=len-1，一直在l=mid+1。对应的，如果数组经过旋转，并且target比最大的还大，因为一直在r=mid-1导致l=0 r=-1。

* [变题：旋转有序有重数组内的值可能相同](https://leetcode.com/problems/search-in-rotated-sorted-array-ii)

可能出现这样的极端情况，在都是1的数组中，任意一个位置可以插入我们要找的0或者2。二分查找的l、m、r三个索引对应的数字都是1，并不能确定缩小至左侧范围还是右侧范围。所以对于这种几段情况，需要退化为线性查找，我是考虑到存在这种退化，说明题目判断的时候接受O(n)的算法，于是直接线性查找。标准答案里面，也到没有完全退化，而是将l=l+1, r=r-1来尝试继续在新区间上二分查找，因为一旦出现l、m、r上的值不完全相等 就可以缩小区间。

技巧：通过排除nums[l]==nums[m]==nums[r]来将问题降级为“旋转有序无重数组”

class Solution:  
 def search(self, nums: List[int], target: int) -> bool:  
 l = 0  
 r = len(nums) - 1  
 while l <= r:  
 m = l + (r - l) // 2  
 if nums[m] == target or nums[l] == target or nums[r] == target:  
 return True  
 if nums[l] == nums[m] == nums[r]:  
 l = l + 1  
 r = r - 1  
 continue  
  
 # target>m and target  
 if nums[l] <= nums[m]:  
 if nums[l] <= target <= nums[m]:  
 r = m - 1  
 else:  
 l = m + 1  
 else:  
 if nums[m] <= target <= nums[r]:  
 l = m + 1  
 else:  
 r = m - 1  
  
 if l >= len(nums):  
 return False  
 else:  
 return nums[l] == target

* 变题：寻找旋转数组中出现的最小值

我想 直接认为target=min\_int，不过python2中才有min\_int，python3中int类型是无界的。但是将这个条件直接套入模板，并处理掉小于最小与大于最大的特殊情况：

class Solution:  
 def findMin(self, nums: List[int]) -> int:  
 l = 0  
 r = len(nums) - 1  
  
 while l <= r:  
 m = l + (r - l) // 2  
  
 if nums[0] <= nums[m]:  
 l = m + 1  
 else:  
 r = m - 1  
 if l == len(nums):  
 l = 0  
 return nums[l]

官方解法：

区别在于r=m（因此需要l<r），if的顺序

class Solution:  
 def findMin(self, nums: List[int]) -> int:  
 l = 0  
 r = len(nums) - 1  
  
 while l < r:  
 m = l + (r - l) // 2  
  
 if nums[m] < nums[-1]:  
 r = m  
 else:  
 l = m + 1  
 # 下面这种写法可能导致l=len+1  
 # if nums[0] <= nums[m]:  
 # l = m + 1  
 # else:  
 # r = m  
 return nums[l]

* [在一个有序数组中找到K个最接近x的元素](https://leetcode.com/problems/find-k-closest-elements)，距离用以下条件判断：

|a - x| < |b - x|, or

|a - x| == |b - x| and a < b

1. 按照距离条件，编写自定义排序函数，重新排序。时间复杂度 O(nlogn) 空间复杂度O(k)
2. 二分搜索+双指针寻找。时间复杂度 O(logn+K) 空间复杂度O(k)

我使用方法2，但是因为二分查找的边界是0-len需要去除边界case，并且双指针走之前需要确定一个中心位置 数组内最接近x的索引，它可能是二分查找的有效索引结果 或 其减一。因为这些边界case，导致这道题的耗时大量上升。

* [找出数组中能组成三角形的三个数个数](https://leetcode.com/problems/valid-triangle-number)

1. O(n2logn) 经过排序通过二分法寻找第三个数
2. O(n2)可以优化掉二分搜索，实际上本次循环的搜索结果位置一定比上一次循环的搜索结果位置大。双指针。

* [寻找最近右区间](https://leetcode.com/problems/find-right-interval)

需要hash记录索引的映射关系

1. O(nlogn)排序+二分
2. O(nlogn)同“找出数组中能组成三角形的三个数个数”，如果分别按照起始点、结束点得到两个排序数组，用双指针，可以继续之前的位置走下去。

* [转变数组后最接近目标值的数组和](https://leetcode.com/problems/sum-of-mutated-array-closest-to-target/)

没有技巧，就是难bug free

二分法主要用于计算求和，因为本题对result可能的取值范围要求不高，否则这里也应该是二分。

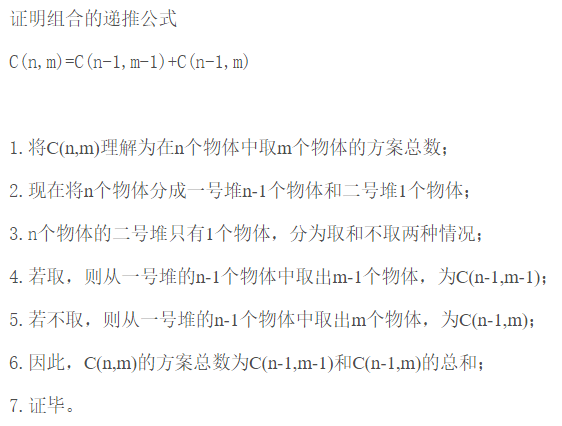
确认result查找范围 min(min(arr), target//len(arr))到max(arr)，如果认为查找边界是target//len(arr)开始则需要判断数组内所有元素小于target//len(arr)的特例情况。

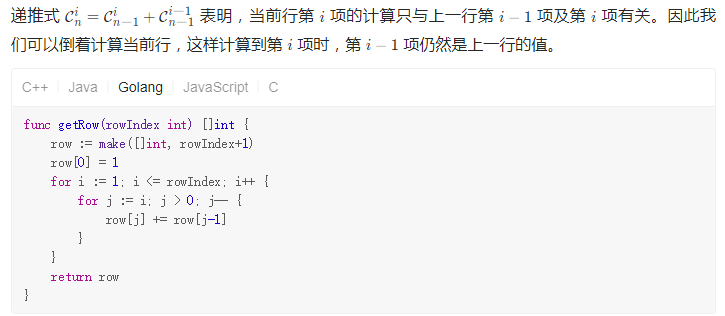
class Solution:  
 def findBestValue(self, arr: List[int], target: int) -> int:  
 arr.sort()  
 n = len(arr)  
 prefix = [0]  
 for num in arr:  
 prefix.append(prefix[-1] + num)  
  
 l, r, ans = 0, max(arr), -1  
 while l <= r:  
 mid = (l + r) // 2  
 it = bisect.bisect\_left(arr, mid)  
 cur = prefix[it] + (n - it) \* mid  
 if cur <= target:  
 ans = mid  
 l = mid + 1  
 else:  
 r = mid - 1  
  
 def check(x):  
 return sum(x if num >= x else num for num in arr)  
  
 choose\_small = check(ans)  
 choose\_big = check(ans + 1)  
 return ans if abs(choose\_small - target) <= abs(choose\_big - target) else ans + 1

杨辉三角：

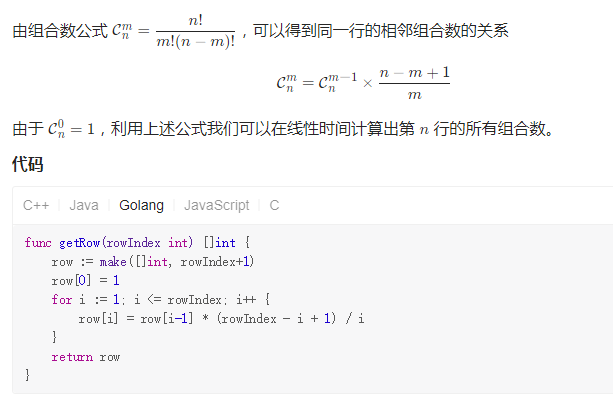
* [输入行数，返回该行的杨辉三角数组](https://leetcode.com/problems/pascals-triangle-ii/)

1. 如果老老实实，通过杨辉三角的二维数组计算，空间复杂度O(n2)
2. 滚动数组 空间复杂度O(2n)：仅使用两个数组，来互相充当下一个数组。
3. 单数组 空间复杂度O(1)（不考虑返回值的空间占用）：基于递推式





1. 如果能推算出等式，那么时间复杂度可以下降



Sum类问题：

* [2SUM数组中找出两个数相加和为给定的数。](https://leetcode.com/problems/two-sum)

这道题有两种思路：

1. 双指针O(nlogn)，一种思路从首尾搜索两个数的和，并逐步逼近。但是没有三指针走法。

2. Map优化一层for循环O(n)，固定一个数A，看SUM-A是否在这个数组之中。

* [3SUM 寻找数组中3个数字之和为0，并且不允许重复找出一样的3个](https://leetcode.com/problems/3sum)

这道题有两种思路：暴力解法是O(n3)，所以优化到O(n2)大部分题目其实非常好

1. 双指针：O(n2)最外层遍历选出一个数，转换为寻找target-nums[select]的2SUM问题
2. Map优化O(n2)：预处理一边两两相加结果，再遍历每一个数nums[i]，判断target-nums[i]是否在预处理的那个和中。

本题因为要去重，所以用双指针更加好，排序之后直接跳过一样的元素。LeetCode上我用方法二超时了，而且空间占用更大：

class Solution:  
 def threeSum(self, nums: List[int]) -> List[List[int]]:  
 results = []  
 # i1 as the base index to find the three numbers  
 for i1 in range(len(nums) - 2):  
 hash\_map = {}  
 for i2 in range(i1 + 1, len(nums)):  
 if nums[i2] not in hash\_map:  
 hash\_map[nums[i2]] = [i2]  
 else:  
 hash\_map[nums[i2]] += [i2]  
  
 for i2 in range(i1 + 1, len(nums)):  
 expect\_value = -nums[i1] - nums[i2]  
 if expect\_value not in hash\_map:  
 continue  
 length = len(hash\_map[expect\_value])  
 if expect\_value == nums[i2]:  
 length -= 1 # 这里需要记得减一  
 if length > 0:  
 one\_result = [expect\_value, nums[i1], nums[i2]]  
 one\_result.sort()  
 exists = False  
 for exist\_result in results:  
 if one\_result == exist\_result:  
 exists = True  
 if not exists:  
 results.append(one\_result)  
 return results

双指针方法，边界case（去重）导致我需要debug一段时间

class Solution:  
 def threeSum(self, nums: List[int]) -> List[List[int]]:  
 nums.sort()  
  
 results = []  
 # i1 as the base index to find the three numbers  
 i1 = 0  
 while i1 < len(nums) - 2:  
  
 expect\_sum = -nums[i1]  
 l = i1 + 1  
 r = len(nums) - 1  
  
 while l < r:  
 sum\_result = nums[l] + nums[r]  
 if expect\_sum < sum\_result:  
 r -= 1  
 elif expect\_sum == sum\_result:  
 results.append([nums[i1], nums[l], nums[r]])  
 # ignore the same  
 while l + 1 < len(nums) - 1 and nums[l] == nums[l + 1]:  
 l += 1  
 # ignore the same  
 while r - 1 > l and nums[r] == nums[r - 1]:  
 r -= 1  
 l += 1  
 r -= 1  
 else:  
 l += 1  
 # ignore the same  
 while i1 + 1 < len(nums) - 2 and nums[i1] == nums[i1 + 1]:  
 i1 += 1  
 i1 += 1  
 return results

* [3Sum Closest 寻找最近接进target的三数之和](https://leetcode.com/problems/3sum-closest)

因为是“接近”，所以没有办法使用map的方法。

因为最后不是3Sum一样输出去重后的嵌套数组，而是输出 最接近的那一个int值，所以可以少一些边界条件与“ignore the same”的代码。

* 4SUM

1. 先遍历第一个数，然后固定第一个数之后，转化为剩下元素的3SUM问题

2. 先遍历两个数，求和，然后转化为剩下元素的2SUM问题

数组走法之双指针

* [返回一个非递减有序数组的平方结果有序数组](https://leetcode.com/problems/squares-of-a-sorted-array/)

用双指针是没有问题的，不过因为返回的结果是从低到高排序，所以我想的是先找到中间最小的再往两边走。其实写起来，代码可读性不如“两边往中间走”容易。

class Solution:  
 def sortedSquares(self, nums: List[int]) -> List[int]:  
 l, r = 0, len(nums) - 1  
 result = [0] \* len(nums)  
 while l < r:  
 if abs(nums[l]) < abs(nums[r]):  
 result[r - l] = nums[r] \* nums[r]  
 r -= 1  
 else:  
 result[r - l] = nums[l] \* nums[l]  
 l += 1  
 if l == r:  
 result[0] = nums[r] \* nums[r]  
  
 return result

还是值得练练手与逻辑，我修了几次Bug才过，主要问题是想一点就写一点

数组走法之滑动窗口：

题目的条件暗示：连续的最短长度

* [给定一个含有 n 个正整数的数组和一个正整数 s ，找出该数组中满足其和 ≥ s 的长度最小的 连续 子数组，并返回其长度](https://leetcode.com/problems/minimum-size-subarray-sum/)

如果能确定思路是“右进左出”，那代码可以简短很多

class Solution:  
 def minSubArrayLen(self, s, nums):  
 total = left = right = 0  
 res = len(nums) + 1  
 while right < len(nums):  
 total += nums[right]  
 while total >= s:  
 res = min(res, right - left + 1)  
 total -= nums[left]  
 left += 1  
 right += 1  
 return res if res <= len(nums) else 0

class Solution:  
 def minSubArrayLen(self, s, nums):  
 l = sum = 0  
 res = len(nums) + 1  
 for i in range(len(nums)):  
 sum += nums[i]  
 while sum >= s:  
 res = min(res, i - l + 1)  
 sum -= nums[l]  
 l += 1  
 return res if res <= len(nums) else 0

移除数组

双指针 或者 Copy a new one by append

class Solution:  
 def removeElement(self, nums: List[int], val: int) -> int:  
 i = 0  
 n = len(nums)  
 while i<n:  
 if nums[i]==val:  
 nums[i] = nums[n-1]  
 n -= 1  
 else:  
 i += 1  
 return n

单链表问题：让人觉得单链表变成双链表就好写了，和数组相比，除了不好走回来，而且没有办法直接得知长度，以及通过索引快速找到目标（比如找到各类分位点）。

* 单链表求和比较: Give you a singly linked list. you can sum up the value of 1st and last node, 2nd and second from last node ... etc. Then return the max sum of this kind of pair.

My solution is easy, find the mid node, disconnect the right part/fraction, reverse the right part and then iterate on both lists.如果不介意增加额外的空间消耗，再加一个字段指针成双链表（doubly linked list），或者复制一份数组。

排列相关：

[寻找全排列的下一个字典序全排列，如[1,2,3]->[1,3,2]->[2,1,3]->[2,3,1]->..[1,2,3]](https://leetcode.com/problems/next-permutation)

因为是字典序，所以找到尽可能靠右的极小值位于i，并且它的右边还存在一个数比它大。我们交换他们的位置，并将i右侧的数组重新排序。让整体全排列，尽可能小地增大。

class Solution:  
 def nextPermutation(self, nums: List[int]) -> None:  
 i = len(nums) - 2  
 while i >= 0 and nums[i] >= nums[i + 1]:  
 i -= 1  
 if i >= 0:  
 j = len(nums) - 1  
 while j >= 0 and nums[i] >= nums[j]:  
 j -= 1  
 nums[i], nums[j] = nums[j], nums[i]  
  
 left, right = i + 1, len(nums) - 1  
 while left < right:  
 nums[left], nums[right] = nums[right], nums[left]  
 left += 1  
 right -= 1

上面程序中用了一些关键判断：

找极小值时仅使用nums[i] >= nums[i + 1]

找在极小值右侧比极小值大的最小数字时，依赖这时右侧已经是倒序排序，所以仅仅走了一边比较过来，找到j

然后将倒序转为正序，只需要头尾往中间走凉凉交换

我没有考虑到倒序的隐藏条件，所以导致我的程序量是上面的3倍而且容易留边界条件的bug

排序相关：

* 合并两个有序数组

1. 考虑到题目允许通过的时间复杂度（最好的方法），选用最简短的办法，直接将一个数组插入另一个数组尾部，调用系统sort函数
2. 如果考虑手写排序，尽可能利用题目提供的“有序”条件。**我犯了一个错：我模拟双指针从左往右走，并且直接在一个数组中逐个插入另外一个数组。**这种方法明显不够好，可能造成数组内数据的大量移动。更好的做法应该是，创造一个新数组，再双指针从左往右走。
3. 如果你希望省掉2的空间，这道题目应该使用逆向双指针。

相同元素：

1. 异或去重（无法判断单个的0）
2. hash、map、set。使用set可以判断是否在其中，也可以判断是否set与input长度一样。
3. 排序

* [存在同一元素2](https://leetcode.com/problems/contains-duplicate-ii/)：Given an integer array nums and an integer k, return true if there are two distinct indices i and j in the array such that nums[i] == nums[j] and abs(i - j) <= k.

1. 时间复杂度O(n),空间复杂度O(n) map记录 相同元素最近出现的节点
2. 时间复杂度O(n),空间复杂度O(k+1) 滑动窗口

我用了方法二，想的是双指针的滚动窗口。不仅没有一遍AC而且代码比标准答案多了一倍：

class Solution:  
 def containsNearbyDuplicate(self, nums: List[int], k: int) -> bool:  
 length = k + 1  
 if length == 1:  
 return False  
 l = 0  
 r = min(len(nums), length)  
 s = set(nums[:r])  
 if len(s) != r:  
 return True  
  
 while r < len(nums):  
 s.remove(nums[l])  
 s.add(nums[r])  
 if len(s) != length:  
 return True  
 l += 1  
 r += 1  
  
 return False

标准答案：

1. 双指针存在明显的转换关系，所以转为单指针

2. 用了 in判断set是否包含元素，所以可以与循环结合

class Solution:  
 def containsNearbyDuplicate(self, nums: List[int], k: int) -> bool:  
 s = set()  
 for i, num in enumerate(nums):  
 if i > k:  
 s.remove(nums[i - k - 1])  
 if num in s:  
 return True  
 s.add(num)  
 return False

* 寻找数组中仅出现一次的元素，其他元素出现3次

1. hash、set 时间复杂度O(n) 空间复杂度O(n)

方法a) 元素第一次出现加入setA，出现一次及以上转移到setB，最终setA.pop()

方法b) return int((3\*sum(set(nums))-sum(nums))/2)

1. 使用位的统计来优化空间复杂度： 时间复杂度O(n) 空间复杂度O(1)

由于数组中的元素都在int（即 32 位整数）范围内，所以用一个int[32]数组存下每一位出现过1的次数。最终%3就能找到那个不一样的值。

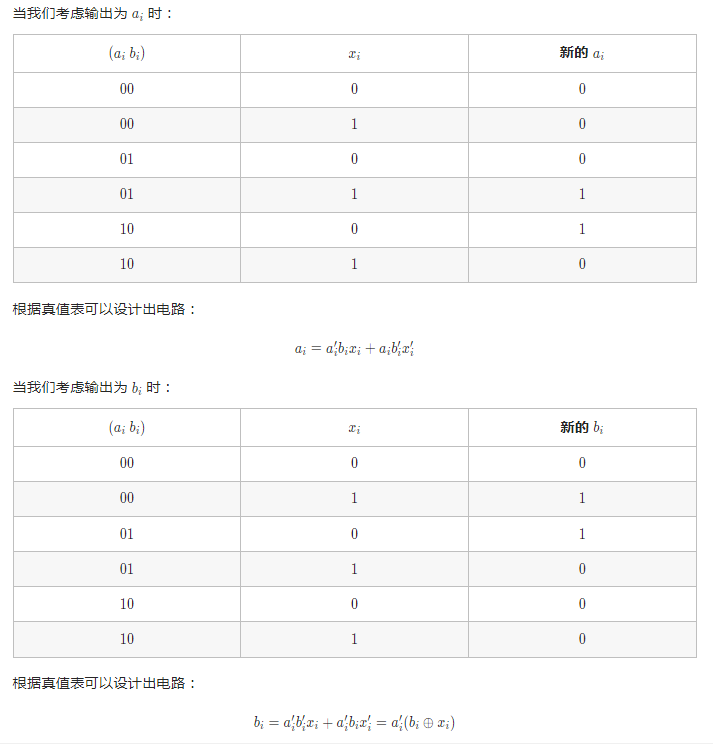
class Solution:  
 def singleNumber(self, nums: List[int]) -> int:  
 ans = 0  
 for i in range(32):  
 total = sum((num >> i) & 1 for num in nums)  
 if total % 3:  
 # Python 这里对于最高位需要特殊判断  
 if i == 31:  
 ans -= (1 << i)  
 else:  
 ans |= (1 << i)  
 return ans

1. 数字电路设计（真值表）时间复杂度O(n) 空间复杂度O(1)

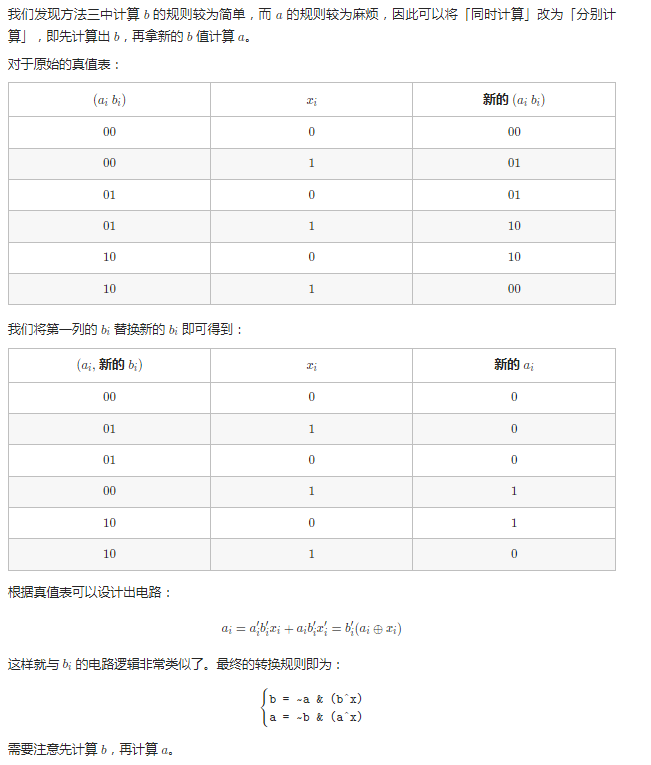
黑盒中存储了两个整数 a 和 b，进行二进制的%3统计，且会有三种情况：

* a 的第 i 位为 0 且 b 的第 i 位为 0，表示 0；
* a 的第 i 位为 0 且 b 的第 i 位为 1，表示 1；
* a 的第 i 位为 1 且 b 的第 i 位为 0，表示 2





class Solution:  
 def singleNumber(self, nums: List[int]) -> int:  
 a = b = 0  
 for num in nums:  
 a, b = (~a & b & num) | (a & ~b & ~num), ~a & (b ^ num)  
 return b



# 真值表优化

class Solution:  
 def singleNumber(self, nums: List[int]) -> int:  
 a = b = 0  
 for num in nums:  
 b = ~a & (b ^ num)  
 a = ~b & (a ^ num)  
 return b

* [missing number](https://leetcode.com/problems/missing-number/): Given an array nums containing n distinct numbers in the range [0, n], return the only number in the range that is missing from the array.

1. 补偿异或：对数组与0-n，进行遍历，并对所有数字进行异或，已存在数字出现2次，缺失数字仅出现1次
2. 将数字存入hash，然后遍历0-n来检查hash

class Solution:  
 def findDisappearedNumbers(self, nums: List[int]) -> List[int]:  
 s = set(nums)  
 result = []  
 for i in range(1,len(nums)+1):  
 if i not in s:  
 result.append(i)  
 return result

优化：注意到nums 的长度恰好也为 nn，能否让nums 充当哈希表

具体来说，遍历 nums，每遇到一个数 xx，就让nums[x−1] 增加 n（或者取反）。由于 nums 中所有数均在 [1,n] 中，增加以后，这些数必然大于 nn。最后我们遍历 nums，若 nums[i] 未大于n，就说明没有遇到过数 i+1，这样我们就找到了缺失的数字。

增加n：

class Solution:  
 def findDisappearedNumbers(self, nums: List[int]) -> List[int]:  
 n = len(nums)  
 for num in nums:  
 x = (num - 1) % n  
 nums[x] += n  
  
 ret = [i + 1 for i, num in enumerate(nums) if num <= n]  
 return ret

取反：

class Solution:  
 def findDisappearedNumbers(self, nums: List[int]) -> List[int]:  
  
 for n in nums:  
 nums[abs(n) - 1] = - abs(nums[abs(n) - 1])  
  
 res = []  
 for i, v in enumerate(nums):  
 if v > 0: # 如果大于0，说明没被占过，也就是没有出现过的数字  
 res.append(i + 1)  
  
 return res

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

1. 通过数学计算排除

class Solution:  
 def missingNumber(self, nums: List[int]) -> int:  
 n = len(nums)  
 return int(n\*(n+1)/2)-sum(nums)

摩尔投票：兵团对耗

* 给定一个数组，寻找出现次数大于len/2的所有元素。要求O(N)时间,O(1)空间

如果存在这样的元素，因为是大于len/2，最多存在一个。想象成兵团战争，如果存在>len/2的兵团，那么不论其他兵团之间是否为敌（最差他们不敌对），最终1v1消耗之后，活下来的一定是。所以算法就是利用对耗思想，找出这样的值，最后判断一下是否该值统计>len/2。

* 变题：次数>=len/2（len/3）

>=len2与>len/3，都需要同时记录两个值及其计数，我们认为这两个兵团联盟不会相消

这道题，**我上来就做> len/3**，结果虽然可以猜测是数学思想，反证法假设存在。但是没能理解如果在O(1)的空间条件中利用，**应该降级为先考虑>len/2，在推测>len/3**。

class Solution:  
 def majorityElement(self, nums: List[int]) -> List[int]:  
 ans = []  
 element1, element2 = 0, 0  
 vote1, vote2 = 0, 0  
  
 for num in nums:  
 # 如果该元素为第一个元素，则计数加1  
 if vote1 > 0 and num == element1:  
 vote1 += 1  
 # 如果该元素为第二个元素，则计数加1  
 elif vote2 > 0 and num == element2:  
 vote2 += 1  
 # 选择第一个元素  
 elif vote1 == 0:  
 element1 = num  
 vote1 += 1  
 # 选择第二个元素  
 elif vote2 == 0:  
 element2 = num  
 vote2 += 1  
 # 如果三个元素均不相同，则相互抵消1次  
 else:  
 vote1 -= 1  
 vote2 -= 1  
  
 cnt1, cnt2 = 0, 0  
 for num in nums:  
 if vote1 > 0 and num == element1:  
 cnt1 += 1  
 if vote2 > 0 and num == element2:  
 cnt2 += 1   
 # 检测元素出现的次数是否满足要求  
 if vote1 > 0 and cnt1 > len(nums) / 3:  
 ans.append(element1)  
 if vote2 > 0 and cnt2 > len(nums) / 3:  
 ans.append(element2)  
  
 return ans

二叉树：

二叉搜索树的中序遍历是升序序列，一个有序数组从中间“提”起来，就是一个高度平衡的二叉搜索树。通常题目会涉及到递归。

* [将有序数组转换为二叉搜索树](https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree)

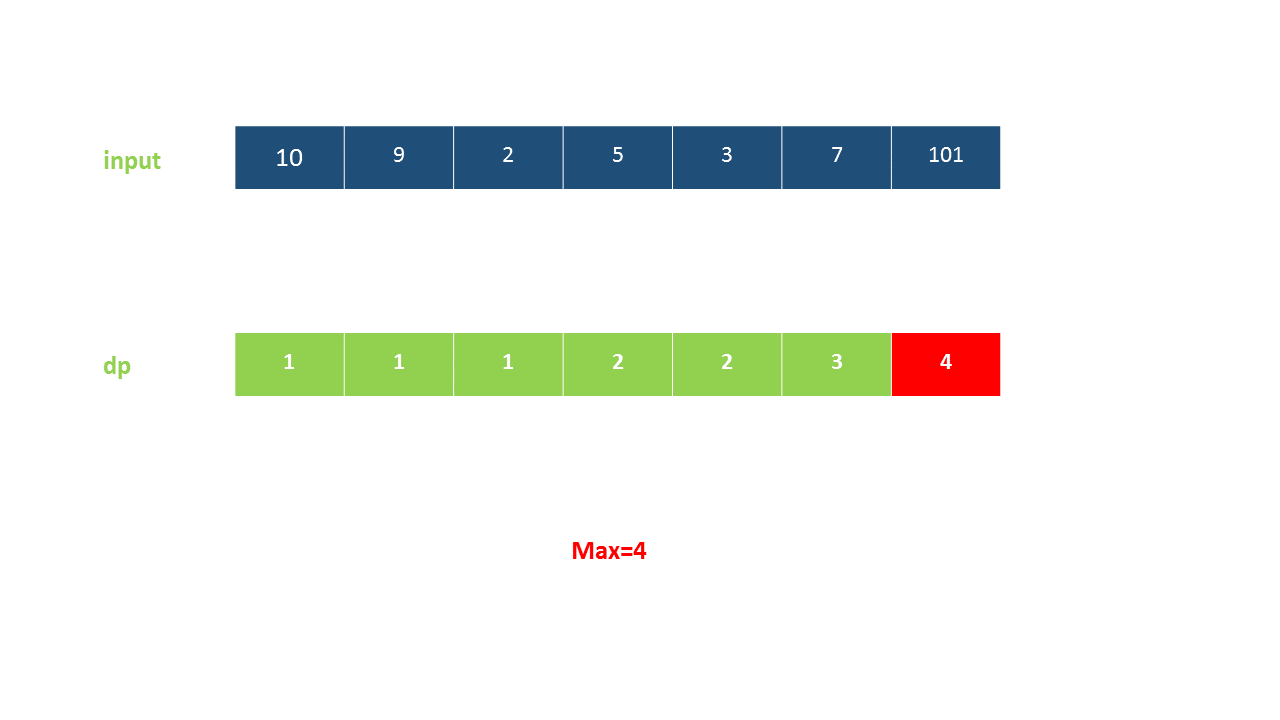
# Definition for a binary tree node.  
# class TreeNode:  
# def \_\_init\_\_(self, val=0, left=None, right=None):  
# self.val = val  
# self.left = left  
# self.right = right  
class Solution:  
 def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:  
 def helper(l, r):  
 if l > r:  
 return None  
  
 m = l + (r - l) // 2  
 return TreeNode(nums[m], helper(l, m - 1), helper(m + 1, r))  
  
 return helper(0, len(nums) - 1)

* 二叉树的前序、中序、后序遍历

动态规划：

# [最长上升子序列](https://leetcode.com/problems/longest-increasing-subsequence/)

1. O(n^2)动态规划 dp[i]=max(dp[j])+1,其中0≤j<i且num[j]<num[i]



class Solution:  
 def lengthOfLIS(self, nums: List[int]) -> int:  
 if not nums:  
 return 0  
 dp = []  
 for i in range(len(nums)):  
 dp.append(1)  
 for j in range(i):  
 if nums[i] > nums[j]:  
 dp[i] = max(dp[i], dp[j] + 1)  
 return max(dp)

dp[i] 表示以此结尾的所有数组中最长子序列长度

1. O(nlogn)贪心+二分查找

贪心实际是最接近人的理解的方法，我们的难点是怎么将思维整理并转化为代码实现。假如给出序列 mylist=[1， 3， 5， 6， 7， 2， 5， 7， 9， 10]，我们肉眼来找最长序列的时候，看到1, 3, 5, 6, 7递增，长度len1=5，我们怀疑这个就是目标序列，暂且记为答案一。 但是当遇到2(mylist[5])时，我们就考虑另一种答案，有没有可能从1, 2, ... (mylist[0], mylist[5]...)会找到一个新的子序列， 且比之前找的序列1, 3, 5, 6, 7更长呢？（记为答案二）我相信大部分人分析题目时，会有这个想法接下来，当我们继续验证答案二时，需要依赖2以后的数字，看是否能找到新的递增序列。然而，我们的目的是找最长的长度， 有两种结果，新的答案比第一次的答案短， 则我们保留原始答案一， 新的答案比第一次的答案长， 我们用新的方案。 但是过程我们要记录， 1）我新的答案，找到哪个数字了，以便继续往下找 2）如果新的答案不够长，保留答案一的结果。 回到题解，当我找到2时， 新的答案，第二个数字是2， 原来的[1, 3, 5, 6, 7], 3已经不在我新的答案序列了，我就把3替换为2， 依次进行， 如果我新的答案比第一次答案长， 整个序列被替换，如果没有第一个长，我替换的次数不够，原来方案一里最大的数字还在末尾， 原始的d的长度不会被替换。那么我们始终能用d的长度表示我们的结果。