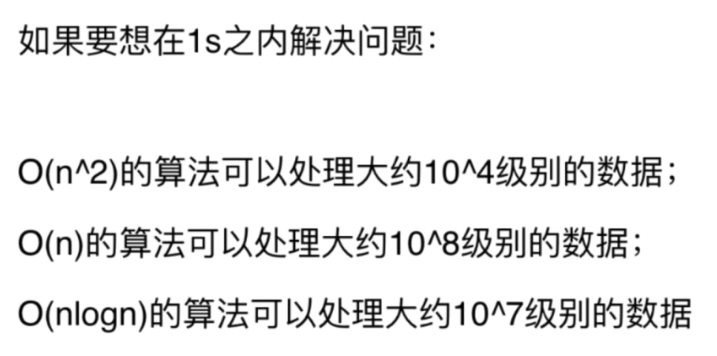
题目没有说明数据是有序的、不重复的，但是范例十分特殊，经常会给人误导

1. 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？



1.气球的颜色代表着题目的易：

（1）亮蓝、金、黄色、花

（2）红

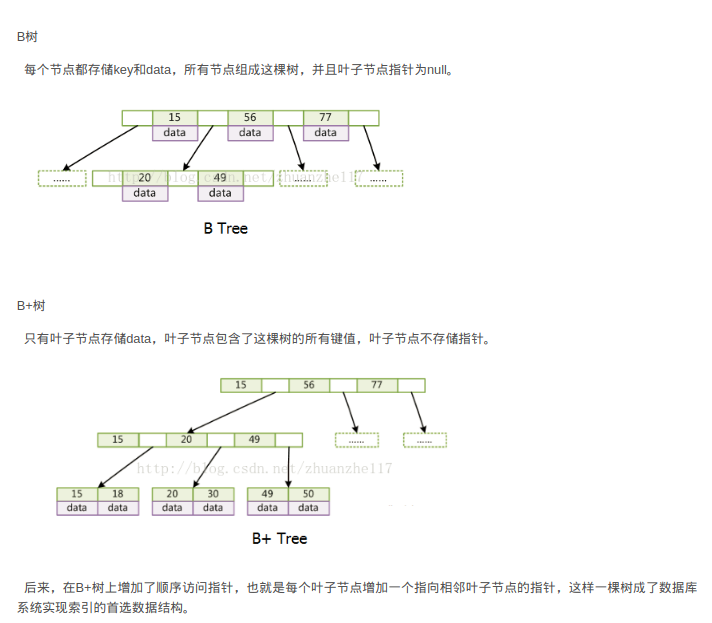
（3）绿

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

2.6 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。

3.动态规划就是使用什么（一般是多维数组）来记录中间的一个个子过程（子问题），那么到数组记录的最后就是我们要求的值。

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+。



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：第二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

第一优先级 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 hash 可以通过再散列的办法解决冲突，但是这种没有下面挂一个链表或者红黑树比较好。从占用空间的角度看，这种一点都不具有可扩展性。

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

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

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。

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];

return 0;

}

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

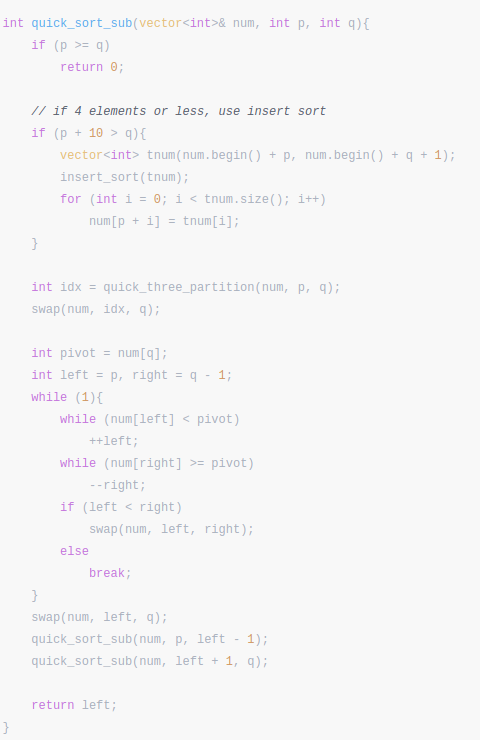
首先要写正确。通常使用递归实现。其递归相当于二叉树展开，因此如果要用迭代实现的话需要使用一个队列等数据结构来保存后续遍历信息，记录分段的点就好。

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。

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

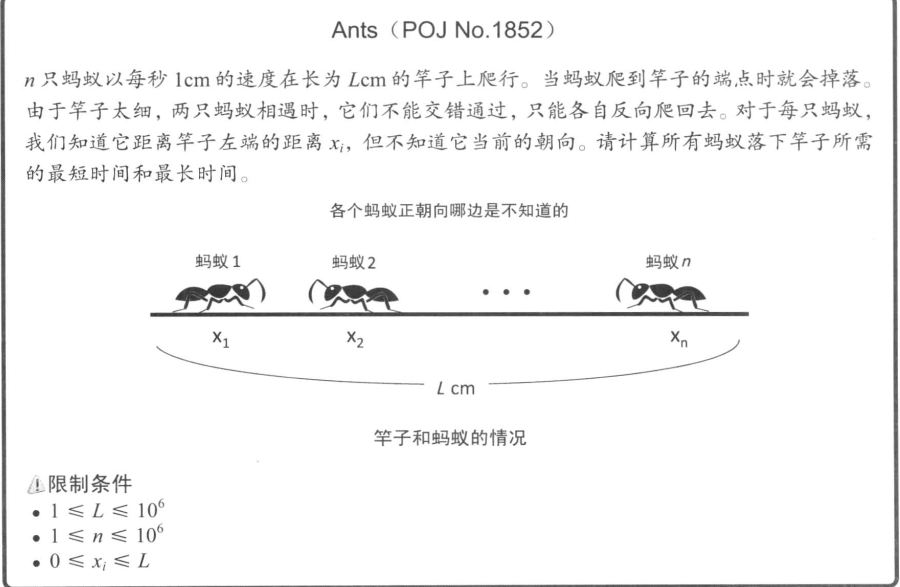
<http://www.360doc.com/content/15/0719/22/22633806_486084820.shtml>

<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

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>

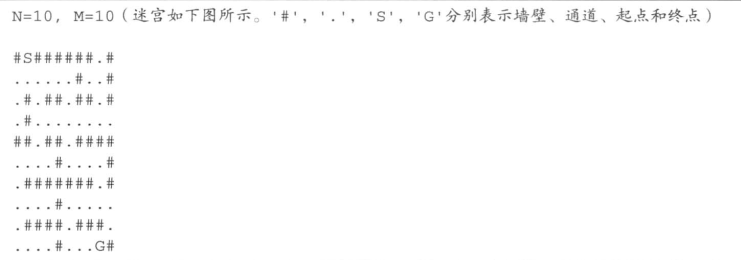


这道题目的关键思路是如何看待蚂蚁的运动，他们相撞然后各自调头返回？是不是会有一种感觉，有一只蚂蚁会不停地撞来撞去，然后这个问题就没有办法解决掉。为何不认真点，完整点做一次模拟，模拟不了大的n，那就模拟小一些的，然后就会发现蚂蚁的相撞可以认为是互相穿过。拓展一下，所以会造成速度交换的，在空间上面都可以认为是穿过。

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）

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

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。
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;

}

（4）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

18.DAG 有向无环图

最短、最长、路径计数都采用动态规划

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）

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。

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或者很好的Bloom Filter肯定是最好的，因为数据的量太大了。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？**

要使用hashset了

但是要处理相同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。

但是我担心会hash冲突，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)

类似于Haffuman树，这里基于文件系统本身的结构，算是一种hash树

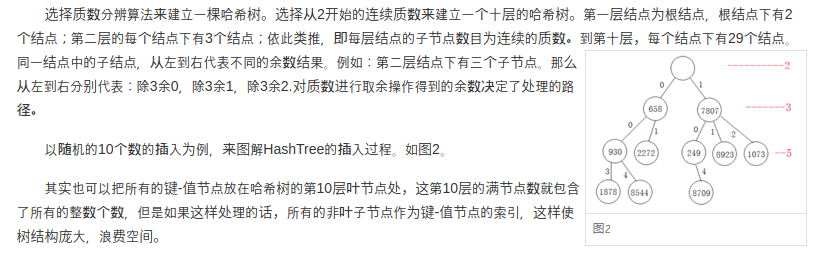
26.哈希树

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

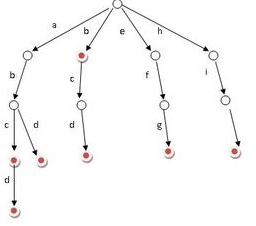
对质数数组（从大到小的顺序，树深度更小，因为数据更加可能分散）取余，也可以组成一个树：



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)。

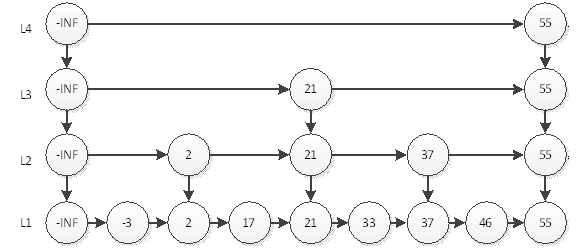
字典树Trie树：

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

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

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）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。



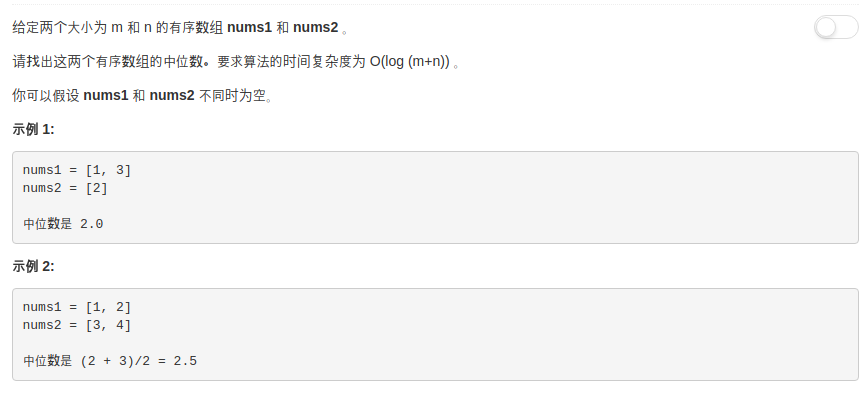
对机器与任务按照各自属性分别排序。从头走开始匹配，只要可以并且收益超过已经设置的任务，就重新设置机器对象的执行任务。



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。

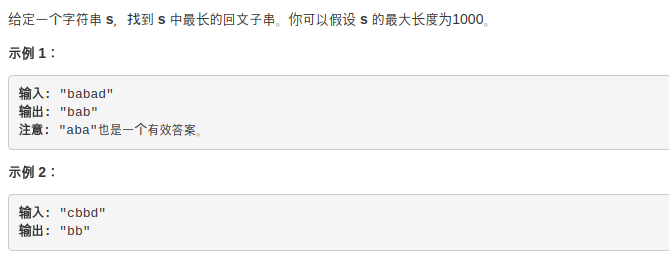
变形：

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。



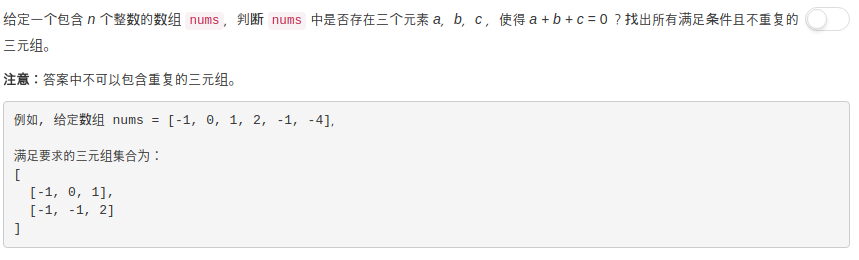
（0）O(n) Manacher 算法：

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

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

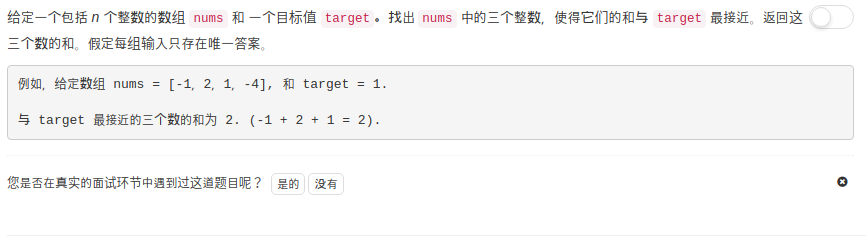
（1）O(n) 寻找回文子串的对称轴，并从对称轴出发一直找下去（需要分奇偶长度的回文串情况考虑？baaac的识别？代码写起来有点复杂）（借鉴Manacher 算法的做法，转为奇数长度的回文），记录找到的最大回文子串，其实思想是动态规划。类似于KMP算法的处理，然后可以跳着移动对称轴。我认为是O(n)的原因是因为每一个后面字符都需要对称轴前面的字符判断一遍。

（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。

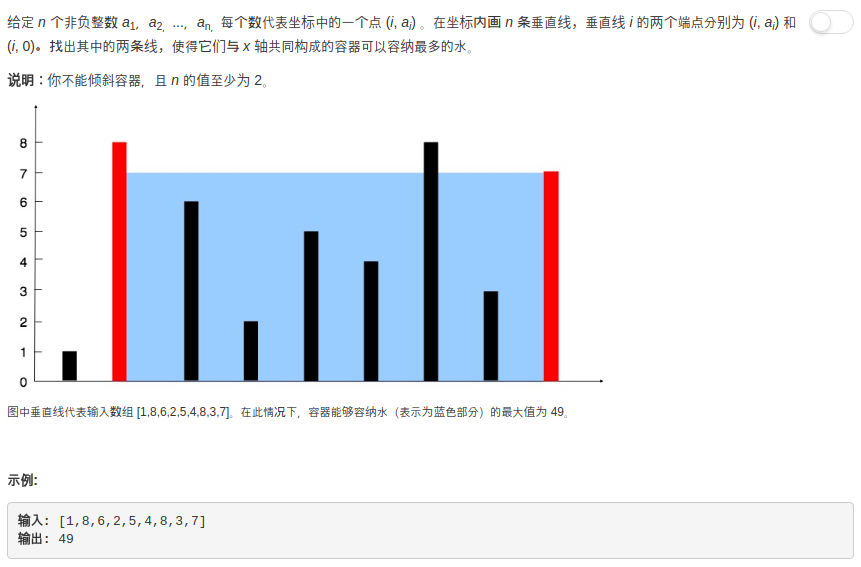
给出一个字符串数组，将其中所有可以通过颠倒字符顺序产生相同结果的单词进行分组。

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

全排列最常见的写法是DFS



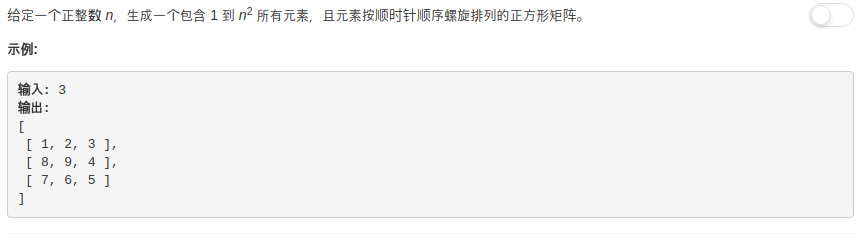
上图这种写法，空间复杂度是O（1），需要注意的是代码中的交换不能改为异或，因为可能需要自己与自己交换，一旦异或就会改变原有数据。使用异或等式交换两个数的前提是这两个数在同一种类型下不相等。

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

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。

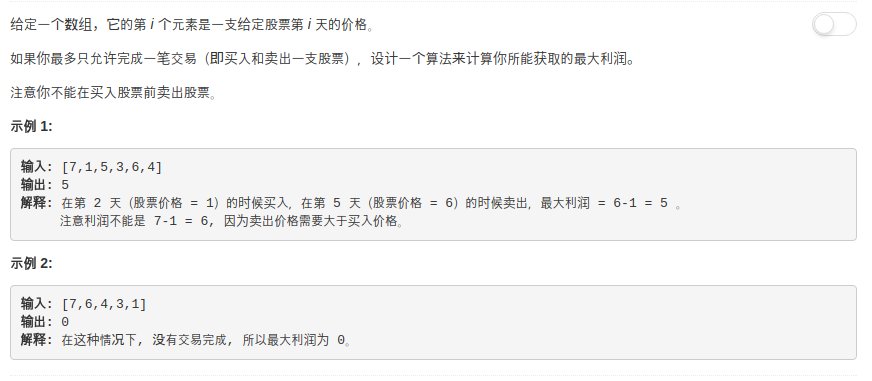


这种矩阵可以用坐标判断截止条件



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);

}

//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

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。

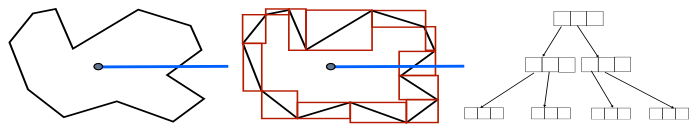


上图中先将相邻的餐厅通过一个小矩阵汇聚在一起，之后反复同一个大矩阵汇聚之前的小矩阵。矩阵的嵌套就是k-d树介绍中说的空间的重叠。

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;

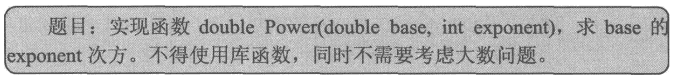
}



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>



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>

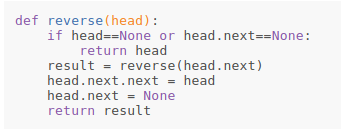
使用java计算两天之间的天数差，存在以下两个坑：

（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：

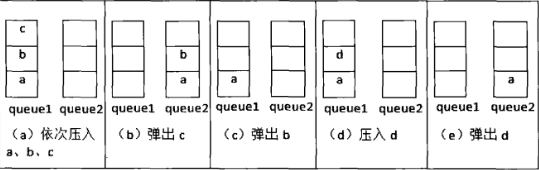


但是这段代码没有被头条面试官认出来了，估计因为受到了head.mext.next = head的干扰。

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：

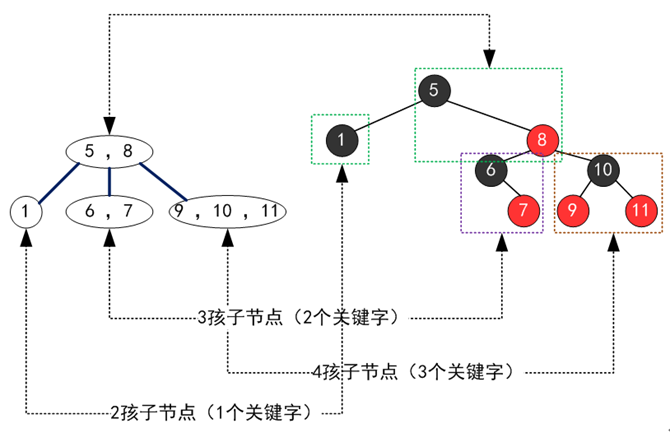
2 3 1

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>

与红黑树等价的2-3-4树： 形态上存在多种对应关系

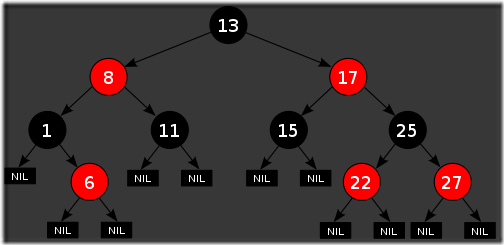


二叉搜索树 来排序 类似于堆排，在排序的过程中维护树（插入与删除都是logN级别），目前使用红黑树比较好，因为树不容易发生大变化。

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。

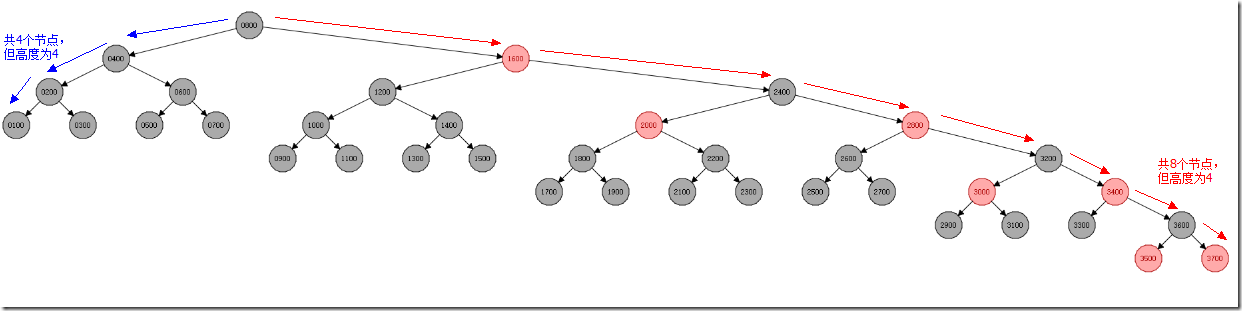


第5点表明： 从根节点到最远的叶子节点的路径长与到最近的叶子节点的路径长度相差不会超过2倍（差值来源于红色节点）。按照黑色节点来计算“高度”，若只是看黑色节点， 就是一个AVL 树。 同时表明了红黑树是一颗“平衡”树，不存在全节点的单链。

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，尽可能减少对红黑树的剩余节点的颜色变化，所以网址中的示例在删除的过程中甚至出现了全黑的树。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。