# 目录

**第一部分 数据结构**

* **第一章 字符串**
  + [1.0 本章导读](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/01.00.md)
  + [1.1 旋转字符串](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/01.01.md)
  + [1.2 字符串包含](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/01.02.md)
  + [1.3 字符串转换成整数](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/01.03.md)
  + [1.4 回文判断](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/01.04.md)
  + [1.5 最长回文子串](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/01.05.md)
  + [1.6 字符串的全排列](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/01.06.md)
  + [1.10 本章习题](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/01.10.md)
* **第二章 数组**
  + [2.0 本章导读](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.00.md)
  + [2.1 寻找最小的 k 个数](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.01.md)
  + [2.2 寻找和为定值的两个数](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.02.md)
  + [2.3 寻找和为定值的多个数](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.03.md)
  + [2.4 最大连续子数组和](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.04.md)
  + [2.5 跳台阶](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.05.md)
  + [2.6 奇偶排序](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.06.md)
  + [2.7 荷兰国旗](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.07.md)
  + [2.8 矩阵相乘](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.08.md)
  + [2.9 完美洗牌](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.09.md)
  + [2.15 本章习题](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.15.md)
* **第三章 树**
  + [3.0 本章导读](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.00.md)
  + [3.1 红黑树](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.01.md)
  + [3.2 B树](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.02.md)
  + [3.3 最近公共祖先LCA](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.03.md)
  + [3.10 本章习题](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.10.md)

**第二部分 算法心得**

* **第四章 查找匹配**
  + [4.1 有序数组的查找](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/04.01.md)
  + [4.2 行列递增矩阵的查找](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/04.02.md)
  + [4.3 出现次数超过一半的数字](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/04.03.md)
* **第五章 动态规划**
  + [5.0 本章导读](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/05.00.md)
  + [5.1 最大连续乘积子串](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/05.01.md)
  + [5.2 字符串编辑距离](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/05.02.md)
  + [5.3 格子取数](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/05.03.md)
  + [5.4 交替字符串](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/05.04.md)
  + [5.10 本章习题](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/05.10.md)

**第三部分 综合演练**

* **第六章 海量数据处理**
  + [6.0 本章导读](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/06.00.md)
  + [6.1 关联式容器](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/06.01.md)
  + [6.2 分而治之](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/06.02.md)
  + [6.3 simhash算法](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/06.03.md)
  + [6.4 外排序](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/06.04.md)
  + [6.5 MapReduce](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/06.05.md)
  + [6.6 多层划分](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/06.06.md)
  + [6.7 Bitmap](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/06.07.md)
  + [6.8 Bloom filter](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/06.08.md)
  + [6.9 Trie树](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/06.09.md)
  + [6.10 数据库](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/06.10.md)
  + [6.11 倒排索引](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/06.11.md)
  + [6.15 本章习题](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/06.15.md)
* **第七章 机器学习**
  + [7.1 K 近邻算法](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/07.01.md)
  + [7.2 支持向量机](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/07.02.svm.md)
* **附录 更多题型**
  + [附录A 语言基础](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/08.00.md)
  + [附录B 概率统计](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/08.01.md)
  + [附录C 智力逻辑](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/08.02.md)
  + [附录D 系统设计](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/08.03.md)
  + [附录E 操作系统](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/08.04.md)
  + [附录F 网络协议](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/08.05.md)
* 注：上述所有的文章已于2014年6月30日暂停大的优化，若没发现bug，将暂不再改动；但如果有bug，请大家随时不吝指出，一经确认，立即修正。所有进一步的修改、改动、优化请见于预计2015年上半年左右上市的纸质版。感谢众位，thanks。

July、二零一四年八月十四日。

* **第一章 字符串** 本章导读

字符串相关的问题在各大互联网公司笔试面试中出现的频率极高，比如微软经典的单词翻转题：输入“I am a student.”，则输出“student. a am I”。

本章重点介绍6个经典的字符串问题，分别是旋转字符串、字符串包含、字符串转换成整数、回文判断、最长回文子串、字符串的全排列，这6个问题要么从暴力解法入手，然后逐步优化，要么多种思路多种解法。

读完本章后会发现，好的思路都是在充分考虑到问题本身的特征的前提下，或巧用合适的数据结构，或选择合适的算法降低时间复杂度（避免不必要的操作），或选用效率更高的算法。

# 1.1 旋转字符串

### 题目描述

给定一个字符串，要求把字符串前面的若干个字符移动到字符串的尾部，如把字符串“abcdef”前面的2个字符'a'和'b'移动到字符串的尾部，使得原字符串变成字符串“cdefab”。请写一个函数完成此功能，要求对长度为n的字符串操作的时间复杂度为 O(n)，空间复杂度为 O(1)。

### 分析与解法

#### 解法一：暴力移位法

初看此题，可能最先想到的方法是按照题目所要求的，把需要移动的字符一个一个地移动到字符串的尾部，如此我们可以实现一个函数LeftShiftOne(char\* s, int n) ，以完成移动一个字符到字符串尾部的功能，代码如下所示：

void LeftShiftOne(char\* s, int n)

{

char t = s[0]; //保存第一个字符

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

{

s[i - 1] = s[i];

}

s[n - 1] = t;

}

因此，若要把字符串开头的m个字符移动到字符串的尾部，则可以如下操作：

void LeftRotateString(char\* s, int n, int m)

{

while (m--)

{

LeftShiftOne(s, n);

}

}

下面，我们来分析一下这种方法的时间复杂度和空间复杂度。

针对长度为n的字符串来说，假设需要移动m个字符到字符串的尾部，那么总共需要 m\*n 次操作，同时设立一个变量保存第一个字符，如此，时间复杂度为O(m \* n)，空间复杂度为O(1)，空间复杂度符合题目要求，但时间复杂度不符合，所以，我们得需要寻找其他更好的办法来降低时间复杂度。

#### 解法二：三步反转法

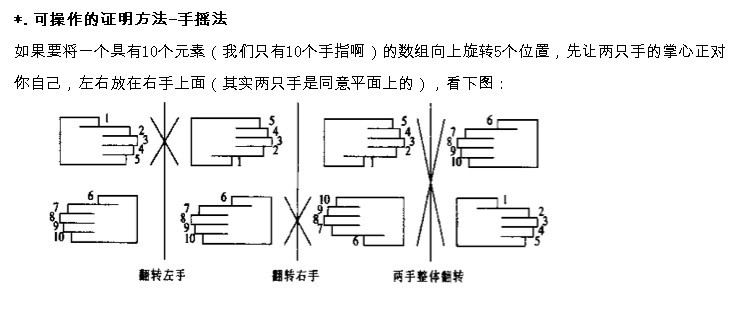
对于这个问题，换一个角度思考一下。

将一个字符串分成X和Y两个部分，在每部分字符串上定义反转操作，如X^T，即把X的所有字符反转（如，X="abc"，那么X^T="cba"），那么就得到下面的结论：(X^TY^T)^T=YX，显然就解决了字符串的反转问题。

例如，字符串 abcdef ，若要让def翻转到abc的前头，只要按照下述3个步骤操作即可：

1. 首先将原字符串分为两个部分，即X:abc，Y:def；
2. 将X反转，X->X^T，即得：abc->cba；将Y反转，Y->Y^T，即得：def->fed。
3. 反转上述步骤得到的结果字符串X^TY^T，即反转字符串cbafed的两部分（cba和fed）给予反转，cbafed得到defabc，形式化表示为(X^TY^T)^T=YX，这就实现了整个反转。

如下图所示：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/1/3.jpeg)

代码则可以这么写：

void ReverseString(char\* s,int from,int to)

{

while (from < to)

{

char t = s[from];

s[from++] = s[to];

s[to--] = t;

}

}

void LeftRotateString(char\* s,int n,int m)

{

m %= n; //若要左移动大于n位，那么和%n 是等价的

ReverseString(s, 0, m - 1); //反转[0..m - 1]，套用到上面举的例子中，就是X->X^T，即 abc->cba

ReverseString(s, m, n - 1); //反转[m..n - 1]，例如Y->Y^T，即 def->fed

ReverseString(s, 0, n - 1); //反转[0..n - 1]，即如整个反转，(X^TY^T)^T=YX，即 cbafed->defabc。

}

这就是把字符串分为两个部分，先各自反转再整体反转的方法，时间复杂度为O(n)，空间复杂度为O(1)，达到了题目的要求。

### 举一反三

1、链表翻转。给出一个链表和一个数k，比如，链表为1→2→3→4→5→6，k=2，则翻转后2→1→6→5→4→3，若k=3，翻转后3→2→1→6→5→4，若k=4，翻转后4→3→2→1→6→5，用程序实现。

2、编写程序，在原字符串中把字符串尾部的m个字符移动到字符串的头部，要求：长度为n的字符串操作时间复杂度为O(n)，空间复杂度为O(1)。 例如，原字符串为”Ilovebaofeng”，m=7，输出结果为：”baofengIlove”。

3、单词翻转。输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变，句子中单词以空格符隔开。为简单起见，标点符号和普通字母一样处理。例如，输入“I am a student.”，则输出“student. a am I”。

# 1.2字符串包含

## 题目描述

给定两个分别由字母组成的字符串A和字符串B，字符串B的长度比字符串A短。请问，如何最快地判断字符串B中所有字母是否都在字符串A里？

为了简单起见，我们规定输入的字符串只包含大写英文字母，请实现函数bool StringContains(string &A, string &B)

比如，如果是下面两个字符串：

String 1：ABCD

String 2：BAD

答案是true，即String2里的字母在String1里也都有，或者说String2是String1的真子集。

如果是下面两个字符串：

String 1：ABCD

String 2：BCE

答案是false，因为字符串String2里的E字母不在字符串String1里。

同时，如果string1：ABCD，string 2：AA，同样返回true。

## 分析与解法

题目描述虽长，但题意很明了，就是给定一长一短的两个字符串A，B，假设A长B短，要求判断B是否包含在字符串A中。

初看似乎简单，但实现起来并不轻松，且如果面试官步步紧逼，一个一个否决你能想到的方法，要你给出更好、最好的方案时，恐怕就要伤不少脑筋了。

### 解法一

判断string2中的字符是否在string1中?最直观也是最简单的思路是，针对string2中每一个字符，逐个与string1中每个字符比较，看它是否在String1中。

代码可如下编写：

bool StringContain(string &a,string &b)

{

for (int i = 0; i < b.length(); ++i) {

int j;

for (j = 0; (j < a.length()) && (a[j] != b[i]); ++j)

;

if (j >= a.length())

{

return false;

}

}

return true;

}

假设n是字符串String1的长度，m是字符串String2的长度，那么此算法，需要O(n\*m)次操作。显然，时间开销太大，应该找到一种更好的办法。

### 解法二

如果允许排序的话，我们可以考虑下排序。比如可先对这两个字符串的字母进行排序，然后再同时对两个字串依次轮询。两个字串的排序需要(常规情况)O(m log m) + O(n log n)次操作，之后的线性扫描需要O(m+n)次操作。

关于排序方法，可采用最常用的快速排序，参考代码如下：

//注意A B中可能包含重复字符，所以注意A下标不要轻易移动。这种方法改变了字符串。如不想改变请自己复制

bool StringContain(string &a,string &b)

{

sort(a.begin(),a.end());

sort(b.begin(),b.end());

for (int pa = 0, pb = 0; pb < b.length();)

{

while ((pa < a.length()) && (a[pa] < b[pb]))

{

++pa;

}

if ((pa >= a.length()) || (a[pa] > b[pb]))

{

return false;

}

//a[pa] == b[pb]

++pb;

}

return true;

}

### 解法三

有没有比快速排序更好的方法呢？

我们换一种角度思考本问题：

假设有一个仅由字母组成字串，让每个字母与一个素数对应，从2开始，往后类推，A对应2，B对应3，C对应5，......。遍历第一个字串，把每个字母对应素数相乘。最终会得到一个整数。

利用上面字母和素数的对应关系，对应第二个字符串中的字母，然后轮询，用每个字母对应的素数除前面得到的整数。如果结果有余数，说明结果为false。如果整个过程中没有余数，则说明第二个字符串是第一个的子集了（判断是不是真子集，可以比较两个字符串对应的素数乘积，若相等则不是真子集）。

思路总结如下：

1. 按照从小到大的顺序，用26个素数分别与字符'A'到'Z'一一对应。
2. 遍历长字符串，求得每个字符对应素数的乘积。
3. 遍历短字符串，判断乘积能否被短字符串中的字符对应的素数整除。
4. 输出结果。

如前所述，算法的时间复杂度为O(m+n)的最好的情况为O(n)（遍历短的字符串的第一个数，与长字符串素数的乘积相除，即出现余数，便可退出程序，返回false），n为长字串的长度，空间复杂度为O(1)。

//此方法只有理论意义，因为整数乘积很大，有溢出风险

bool StringContain(string &a,string &b)

{

const int p[26] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,61, 67, 71, 73, 79, 83, 89, 97, 101};

int f = 1;

for (int i = 0; i < a.length(); ++i)

{

int x = p[a[i] - 'A'];

if (f % x)

{

f \*= x;

}

}

for (int i = 0; i < b.length(); ++i)

{

int x = p[b[i] - 'A'];

if (f % x)

{

return false;

}

}

return true;

}

此种素数相乘的方法看似完美，但缺点是素数相乘的结果容易导致整数溢出。

### 解法四

如果面试官继续追问，还有没有更好的办法呢？计数排序？除了计数排序呢？

事实上，可以先把长字符串a中的所有字符都放入一个Hashtable里，然后轮询短字符串b，看短字符串b的每个字符是否都在Hashtable里，如果都存在，说明长字符串a包含短字符串b，否则，说明不包含。

再进一步，我们可以对字符串A，用位运算（26bit整数表示)计算出一个“签名”，再用B中的字符到A里面进行查找。

// “最好的方法”，时间复杂度O(n + m)，空间复杂度O(1)

bool StringContain(string &a,string &b)

{

int hash = 0;

for (int i = 0; i < a.length(); ++i)

{

hash |= (1 << (a[i] - 'A'));

}

for (int i = 0; i < b.length(); ++i)

{

if ((hash & (1 << (b[i] - 'A'))) == 0)

{

return false;

}

}

return true;

}

这个方法的实质是用一个整数代替了hashtable，空间复杂度为O(1)，时间复杂度还是O(n + m)。

## 举一反三

1、变位词

* 如果两个字符串的字符一样，但是顺序不一样，被认为是兄弟字符串，比如bad和adb即为兄弟字符串，现提供一个字符串，如何在字典中迅速找到它的兄弟字符串，请描述数据结构和查询过程。

# 1.3字符串转换成整数

## 题目描述

输入一个由数字组成的字符串，把它转换成整数并输出。例如：输入字符串"123"，输出整数123。

给定函数原型int StrToInt(const char \*str) ，实现字符串转换成整数的功能，不能使用库函数atoi。

## 分析与解法

本题考查的实际上就是字符串转换成整数的问题，或者说是要你自行实现atoi函数。那如何实现把表示整数的字符串正确地转换成整数呢？以"123"作为例子：

* 当我们扫描到字符串的第一个字符'1'时，由于我们知道这是第一位，所以得到数字1。
* 当扫描到第二个数字'2'时，而之前我们知道前面有一个1，所以便在后面加上一个数字2，那前面的1相当于10，因此得到数字：1\*10+2=12。
* 继续扫描到字符'3'，'3'的前面已经有了12，由于前面的12相当于120，加上后面扫描到的3，最终得到的数是：12\*10+3=123。

因此，此题的基本思路便是：从左至右扫描字符串，把之前得到的数字乘以10，再加上当前字符表示的数字。

思路有了，你可能不假思索，写下如下代码：

int StrToInt(const char \*str)

{

int n = 0;

while (\*str != 0)

{

int c = \*str - '0';

n = n \* 10 + c;

++str;

}

return n;

}

显然，上述代码忽略了以下细节：

1. 空指针输入：输入的是指针，在访问空指针时程序会崩溃，因此在使用指针之前需要先判断指针是否为空。
2. 正负符号：整数不仅包含数字，还有可能是以'+'或'-'开头表示正负整数，因此如果第一个字符是'-'号，则要把得到的整数转换成负整数。
3. 非法字符：输入的字符串中可能含有不是数字的字符。因此，每当碰到这些非法的字符，程序应停止转换。
4. 整型溢出：输入的数字是以字符串的形式输入，因此输入一个很长的字符串将可能导致溢出。

上述其它问题比较好处理，但溢出问题比较麻烦，所以咱们来重点看下溢出问题。

一般说来，当发生溢出时，取最大或最小的int值。即大于正整数能表示的范围时返回MAX\_INT：2147483647；小于负整数能表示的范围时返回MIN\_INT：-2147483648。

我们先设置一些变量：

* sign用来处理数字的正负，当为正时sign > 0，当为负时sign < 0
* n存放最终转换后的结果
* c表示当前数字

而后，你可能会编写如下代码段处理溢出问题：

//当发生正溢出时，返回INT\_MAX

if ((sign == '+') && (c > MAX\_INT - n \* 10))

{

n = MAX\_INT;

break;

}

//发生负溢出时，返回INT\_MIN

else if ((sign == '-') && (c - 1 > MAX\_INT - n \* 10))

{

n = MIN\_INT;

break;

}

但当上述代码转换" 10522545459"会出错，因为正常的话理应得到MAX\_INT：2147483647，但程序运行结果将会是：1932610867。

为什么呢？因为当给定字符串" 10522545459"时，而MAX\_INT是2147483647，即MAX\_INT(2147483647) < n10(1052254545\10)，所以当扫描到最后一个字符‘9’的时候，执行上面的这行代码：

c > MAX\_INT - n \* 10

已无意义，因为此时(MAX\_INT - n \* 10)已经小于0，程序已经出错。

针对这种由于输入了一个很大的数字转换之后会超过能够表示的最大的整数而导致的溢出情况，我们有两种处理方式可以选择：

* 一个取巧的方式是把转换后返回的值n定义成long long，即long long n；
* 另外一种则是只比较n和MAX\_INT / 10的大小，即：
  + 若n > MAX\_INT / 10，那么说明最后一步转换时，n\*10必定大于MAX\_INT，所以在得知n > MAX\_INT / 10时，当即返回MAX\_INT。
  + 若n == MAX\_INT / 10时，那么比较最后一个数字c跟MAX\_INT % 10的大小，即如果n == MAX\_INT / 10且c > MAX\_INT % 10，则照样返回MAX\_INT。

对于上面第一种方式，虽然我们把n定义了长整型，但最后返回时系统会自动转换成整型。咱们下面主要来看第二种处理方式。

对于上面第二种方式，先举两个例子说明下：

* 如果我们要转换的字符串是"2147483697"，那么当我扫描到字符'9'时，判断出214748369 > MAX\_INT / 10 = 2147483647 / 10 = 214748364（C语言里，整数相除自动取整，不留小数），则返回MAX\_INT；
* 如果我们要转换的字符串是"2147483648"，那么判断最后一个字符'8'所代表的数字8与MAX\_INT % 10 = 7的大小，前者大，依然返回MAX\_INT。

一直以来，我们努力的目的归根结底是为了更好的处理溢出，但上述第二种处理方式考虑到直接计算n \* 10 + c 可能会大于MAX\_INT导致溢出，那么便两边同时除以10，只比较n和MAX\_INT / 10的大小，从而巧妙的规避了计算n\*10这一乘法步骤，转换成计算除法MAX\_INT/10代替，不能不说此法颇妙。

如此我们可以写出正确的处理溢出的代码：

c = \*str - '0';

if (sign > 0 && (n > MAX\_INT / 10 || (n == MAX\_INT / 10 && c > MAX\_INT % 10)))

{

n = MAX\_INT;

break;

}

else if (sign < 0 && (n > (unsigned)MIN\_INT / 10 || (n == (unsigned)MIN\_INT / 10 && c > (unsigned)MIN\_INT % 10)))

{

n = MIN\_INT;

break;

}

从而，字符串转换成整数，完整的参考代码为：

int StrToInt(const char\* str)

{

static const int MAX\_INT = (int)((unsigned)~0 >> 1);

static const int MIN\_INT = -(int)((unsigned)~0 >> 1) - 1;

unsigned int n = 0;

//判断是否输入为空

if (str == 0)

{

return 0;

}

//处理空格

while (isspace(\*str))

++str;

//处理正负

int sign = 1;

if (\*str == '+' || \*str == '-')

{

if (\*str == '-')

sign = -1;

++str;

}

//确定是数字后才执行循环

while (isdigit(\*str))

{

//处理溢出

int c = \*str - '0';

if (sign > 0 && (n > MAX\_INT / 10 || (n == MAX\_INT / 10 && c > MAX\_INT % 10)))

{

n = MAX\_INT;

break;

}

else if (sign < 0 && (n >(unsigned)MIN\_INT / 10 || (n == (unsigned)MIN\_INT / 10 && c > (unsigned)MIN\_INT % 10)))

{

n = MIN\_INT;

break;

}

//把之前得到的数字乘以10，再加上当前字符表示的数字。

n = n \* 10 + c;

++str;

}

return sign > 0 ? n : -n;

}

## 举一反三

1. 实现string到double的转换

分析：此题虽然类似于atoi函数，但毕竟double为64位，而且支持小数，因而边界条件更加严格，写代码时需要更加注意。

# 1.4回文判断

## 题目描述

回文，英文palindrome，指一个顺着读和反过来读都一样的字符串，比如madam、我爱我，这样的短句在智力性、趣味性和艺术性上都颇有特色，中国历史上还有很多有趣的回文诗。

那么，我们的第一个问题就是：判断一个字串是否是回文？

## 分析与解法

回文判断是一类典型的问题，尤其是与字符串结合后呈现出多姿多彩，在实际中使用也比较广泛，而且也是面试题中的常客，所以本节就结合几个典型的例子来体味下回文之趣。

### 解法一

同时从字符串头尾开始向中间扫描字串，如果所有字符都一样，那么这个字串就是一个回文。采用这种方法的话，我们只需要维护头部和尾部两个扫描指针即可。

代码如下：：

bool IsPalindrome(const char \*s, int n)

{

//非法输入

if (s == NULL || n < 1)

return false;

char \*front, \*back;

//初始化头指针和尾指针

front = s;

back = s + n - 1;

while (front < back)

{

if (\*front != \*back)

return false; // 不是回文，立即返回

++front;

--back;

}

return true; // 是回文

}

这是一个直白且效率不错的实现，时间复杂度：O(n)，空间复杂度：O(1)。

### 解法二

上述解法一从两头向中间扫描，那么是否还有其它办法呢？我们可以先从中间开始、然后向两边扩展查看字符是否相等。参考代码如下：

bool IsPalindrome2(const char \*s, int n)

{

if (s == NULL || n < 1)

return false; // 非法输入

char \*first, \*second;

int m = ((n >> 1) - 1) >= 0 ? (n >> 1) - 1 : 0; // m is themiddle point of s

first = s + m;

second = s + n - 1 - m;

while (first >= s)

if (s[first--] != s[second++])

return false; // not equal, so it's not apalindrome

return true; // check over, it's a palindrome

}

时间复杂度：O(n)，空间复杂度：O(1)。

虽然本解法二的时空复杂度和解法一是一样的，但很快我们会看到，在某些回文问题里面，这个方法有着自己的独到之处，可以方便的解决一类问题。

## 举一反三

1、判断一条单向链表是不是“回文”

分析：对于单链表结构，可以用两个指针从两端或者中间遍历并判断对应字符是否相等。但这里的关键就是如何朝两个方向遍历。由于单链表是单向的，所以要向两个方向遍历的话，可以采取经典的快慢指针的方法，即先位到链表的中间位置，再将链表的后半逆置，最后用两个指针同时从链表头部和中间开始同时遍历并比较即可。

2、判断一个栈是不是“回文”

分析：对于栈的话，只需要将字符串全部压入栈，然后依次将各字符出栈，这样得到的就是原字符串的逆置串，分别和原字符串各个字符比较，就可以判断了。

# 1.5最长回文子串

### 题目描述

给定一个字符串，求它的最长回文子串的长度。

### 分析与解法

最容易想到的办法是枚举所有的子串，分别判断其是否为回文。这个思路初看起来是正确的，但却做了很多无用功，如果一个长的子串包含另一个短一些的子串，那么对子串的回文判断其实是不需要的。

#### 解法一

那么如何高效的进行判断呢？我们想想，如果一段字符串是回文，那么以某个字符为中心的前缀和后缀都是相同的，例如以一段回文串“aba”为例，以b为中心，它的前缀和后缀都是相同的，都是a。

那么，我们是否可以可以枚举中心位置，然后再在该位置上用扩展法，记录并更新得到的最长的回文长度呢？答案是肯定的，参考代码如下：

int LongestPalindrome(const char \*s, int n)

{

int i, j, max,c;

if (s == 0 || n < 1)

return 0;

max = 0;

for (i = 0; i < n; ++i) { // i is the middle point of the palindrome

for (j = 0; (i - j >= 0) && (i + j < n); ++j){ // if the length of the palindrome is odd

if (s[i - j] != s[i + j])

break;

c = j \* 2 + 1;

}

if (c > max)

max = c;

for (j = 0; (i - j >= 0) && (i + j + 1 < n); ++j){ // for the even case

if (s[i - j] != s[i + j + 1])

break;

c = j \* 2 + 2;

}

if (c > max)

max = c;

}

return max;

}

代码稍微难懂一点的地方就是内层的两个 for 循环，它们分别对于以 i 为中心的，长度为奇数和偶数的两种情况，整个代码遍历中心位置 i 并以之扩展，找出最长的回文。

#### 解法二、O(N)解法

在上文的解法一：枚举中心位置中，我们需要特别考虑字符串的长度是奇数还是偶数，所以导致我们在编写代码实现的时候要把奇数和偶数的情况分开编写，是否有一种方法，可以不用管长度是奇数还是偶数，而统一处理呢？比如是否能把所有的情况全部转换为奇数处理？

答案还是肯定的。这就是下面我们将要看到的Manacher算法，且这个算法求最长回文子串的时间复杂度是线性O(N)的。

首先通过在每个字符的两边都插入一个特殊的符号，将所有可能的奇数或偶数长度的回文子串都转换成了奇数长度。比如 abba 变成 #a#b#b#a#， aba变成 #a#b#a#。

此外，为了进一步减少编码的复杂度，可以在字符串的开始加入另一个特殊字符，这样就不用特殊处理越界问题，比如$#a#b#a#。

以字符串12212321为例，插入#和$这两个特殊符号，变成了 S[] = "$#1#2#2#1#2#3#2#1#"，然后用一个数组 P[i] 来记录以字符S[i]为中心的最长回文子串向左或向右扩张的长度（包括S[i]）。

比如S和P的对应关系：

* S # 1 # 2 # 2 # 1 # 2 # 3 # 2 # 1 #
* P 1 2 1 2 5 2 1 4 1 2 1 6 1 2 1 2 1

可以看出，P[i]-1正好是原字符串中最长回文串的总长度，为5。

接下来怎么计算P[i]呢？Manacher算法增加两个辅助变量id和mx，其中id表示最大回文子串中心的位置，mx则为id+P[id]，也就是最大回文子串的边界。得到一个很重要的结论：

* 如果mx > i，那么P[i] >= Min(P[2 \* id - i], mx - i)

C代码如下：

//mx > i，那么P[i] >= MIN(P[2 \* id - i], mx - i)

//故谁小取谁

if (mx - i > P[2\*id - i])

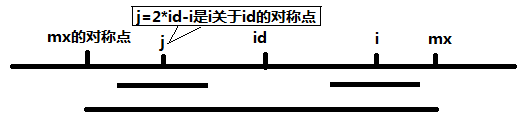
P[i] = P[2\*id - i];

else //mx-i <= P[2\*id - i]

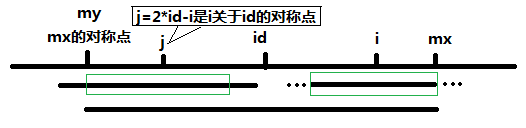
P[i] = mx - i;

下面，令j = 2\*id - i，也就是说j是i关于id的对称点。

当 mx - i > P[j] 的时候，以S[j]为中心的回文子串包含在以S[id]为中心的回文子串中，由于i和j对称，以S[i]为中心的回文子串必然包含在以S[id]为中心的回文子串中，所以必有P[i] = P[j]；

[](https://camo.githubusercontent.com/6e347c94c2660435bde50fa5305135229c8d76cf/687474703a2f2f7777772e66656c69783032312e636f6d2f626c6f672f6174746163686d656e742f313331383437363238345f37393335346134372e706e67)

当 P[j] >= mx - i 的时候，以S[j]为中心的回文子串不一定完全包含于以S[id]为中心的回文子串中，但是基于对称性可知，下图中两个绿框所包围的部分是相同的，也就是说以S[i]为中心的回文子串，其向右至少会扩张到mx的位置，也就是说 P[i] >= mx - i。至于mx之后的部分是否对称，再具体匹配。

[](https://camo.githubusercontent.com/ff7edaf62027622be69d38aa3412b542328f8386/687474703a2f2f7777772e66656c69783032312e636f6d2f626c6f672f6174746163686d656e742f313331383437383131345f34333739666235632e706e67)

此外，对于 mx <= i 的情况，因为无法对 P[i]做更多的假设，只能让P[i] = 1，然后再去匹配。

综上，关键代码如下：

//输入，并处理得到字符串s

int p[1000], mx = 0, id = 0;

memset(p, 0, sizeof(p));

for (i = 1; s[i] != '\0'; i++)

{

p[i] = mx > i ? min(p[2 \* id - i], mx - i) : 1;

while (s[i + p[i]] == s[i - p[i]])

p[i]++;

if (i + p[i] > mx)

{

mx = i + p[i];

id = i;

}

}

//找出p[i]中最大的

此Manacher算法使用id、mx做配合，可以在每次循环中，直接对P[i]的快速赋值，从而在计算以i为中心的回文子串的过程中，不必每次都从1开始比较，减少了比较次数，最终使得求解最长回文子串的长度达到线性O(N)的时间复杂度。

参考：<http://www.felix021.com/blog/read.php?2040> 。另外，这篇文章也不错：<http://leetcode.com/2011/11/longest-palindromic-substring-part-ii.html> 。

# 1.6字符串的全排列

### 题目描述

输入一个字符串，打印出该字符串中字符的所有排列。

例如输入字符串abc，则输出由字符a、b、c 所能排列出来的所有字符串

abc、acb、bac、bca、cab 和 cba。

### 分析与解法

#### 解法一、递归实现

从集合中依次选出每一个元素，作为排列的第一个元素，然后对剩余的元素进行全排列，如此递归处理，从而得到所有元素的全排列。以对字符串abc进行全排列为例，我们可以这么做：以abc为例

* 固定a，求后面bc的排列：abc，acb，求好后，a和b交换，得到bac
* 固定b，求后面ac的排列：bac，bca，求好后，c放到第一位置，得到cba
* 固定c，求后面ba的排列：cba，cab。

代码可如下编写所示：

void CalcAllPermutation(char\* perm, int from, int to)

{

if (to <= 1)

{

return;

}

if (from == to)

{

for (int i = 0; i <= to; i++)

cout << perm[i];

cout << endl;

}

else

{

for (int j = from; j <= to; j++)

{

swap(perm[j], perm[from]);

CalcAllPermutation(perm, from + 1, to);

swap(perm[j], perm[from]);

}

}

}

#### 解法二、字典序排列

首先，咱们得清楚什么是字典序。根据维基百科的定义：给定两个偏序集A和B,(a,b)和(a′,b′)属于笛卡尔集 A × B，则字典序定义为

(a,b) ≤ (a′,b′) 当且仅当 a < a′ 或 (a = a′ 且 b ≤ b′)。

所以给定两个字符串，逐个字符比较，那么先出现较小字符的那个串字典顺序小，如果字符一直相等，较短的串字典顺序小。例如：abc < abcd < abde < afab。

那有没有这样的算法，使得

* 起点： 字典序最小的排列, 1-n , 例如12345
* 终点： 字典序最大的排列，n-1, 例如54321
* 过程： 从当前排列生成字典序刚好比它大的下一个排列

答案是肯定的：有，即是STL中的next\_permutation算法。

在了解next\_permutation算法是怎么一个过程之前，咱们得先来分析下“下一个排列”的性质。

* 假定现有字符串(A)x(B)，它的下一个排列是：(A)y(B’)，其中A、B和B’是“字符串”(可能为空），x和y是“字符”，前缀相同，都是A，且一定有y > x。
* 那么，为使下一个排列字典顺序尽可能小，必有：
  + A尽可能长
  + y尽可能小
  + B’里的字符按由小到大递增排列

现在的问题是：找到x和y。怎么找到呢？咱们来看一个例子。

比如说，现在我们要找21543的下一个排列，我们可以从左至右逐个扫描每个数，看哪个能增大（至于如何判定能增大，是根据如果一个数右面有比它大的数存在，那么这个数就能增大），我们可以看到最后一个能增大的数是：x = 1。

而1应该增大到多少？1能增大到它右面比它大的那一系列数中最小的那个数，即：y = 3，故此时21543的下一个排列应该变为23xxx，显然 xxx(对应之前的B’）应由小到大排，于是我们最终找到比“21543”大，但字典顺序尽量小的23145，找到的23145刚好比21543大。

由这个例子可以得出next\_permutation算法流程为：

next\_permutation算法

* 定义
  + 升序：相邻两个位置ai < ai+1，ai 称作该升序的首位
* 步骤（二找、一交换、一翻转）
  + 找到排列中最后（最右）一个升序的首位位置i，x = ai
  + 找到排列中第i位右边最后一个比ai 大的位置j，y = aj
  + 交换x，y
  + 把第(i+ 1)位到最后的部分翻转

还是拿上面的21543举例，那么，应用next\_permutation算法的过程如下：

* x = 1；
* y = 3
* 1和3交换
  + 得23541
* 翻转541
  + 得23145

23145即为所求的21543的下一个排列。参考实现代码如下：

bool CalcAllPermutation(char\* perm, int num){

int i;

//①找到排列中最后（最右）一个升序的首位位置i，x = ai

for (i = num - 2; (i >= 0) && (perm[i] >= perm[i + 1]); --i){

;

}

// 已经找到所有排列

if (i < 0){

return false;

}

int k;

//②找到排列中第i位右边最后一个比ai 大的位置j，y = aj

for (k = num - 1; (k > i) && (perm[k] <= perm[i]); --k){

;

}

//③交换x，y

swap(perm[i], perm[k]);

//④把第(i+ 1)位到最后的部分翻转

reverse(perm + i + 1, perm + num);

return true;

}

然后在主函数里循环判断和调用calcAllPermutation函数输出全排列即可。

#### 解法总结

由于全排列总共有n!种排列情况，所以不论解法一中的递归方法，还是上述解法二的字典序排列方法，这两种方法的时间复杂度都为O(n!)。

### 类似问题

1、已知字符串里的字符是互不相同的，现在任意组合，比如ab，则输出aa，ab，ba，bb，编程按照字典序输出所有的组合。

分析：非简单的全排列问题（跟全排列的形式不同,abc全排列的话，只有6个不同的输出）。 本题可用递归的思想，设置一个变量表示已输出的个数，然后当个数达到字符串长度时，就输出。

//copyright@ 一直很安静 && World Gao

//假设str已经有序

void perm(char\* result, char \*str, int size, int resPos)

{

if(resPos == size)

printf("%s\n", result);

else

{

for(int i = 0; i < size; ++i)

{

result[resPos] = str[i];

perm(result, str, size, resPos + 1);

}

}

}

2、如果不是求字符的所有排列，而是求字符的所有组合，应该怎么办呢？当输入的字符串中含有相同的字符串时，相同的字符交换位置是不同的排列，但是同一个组合。举个例子，如果输入abc，它的组合有a、b、c、ab、ac、bc、abc。

3、写一个程序，打印出以下的序列。

(a),(b),(c),(d),(e)........(z)

(a,b),(a,c),(a,d),(a,e)......(a,z),(b,c),(b,d).....(b,z),(c,d).....(y,z)

(a,b,c),(a,b,d)....(a,b,z),(a,c,d)....(x,y,z)

....

(a,b,c,d,.....x,y,z)

# 1.10本章字符串和链表的习题

**1、第一个只出现一次的字符**

在一个字符串中找到第一个只出现一次的字符。如输入abaccdeff，则输出b。

**2、对称子字符串的最大长度**

输入一个字符串，输出该字符串中对称的子字符串的最大长度。比如输入字符串“google”，由于该字符串里最长的对称子字符串是“goog”，因此输出4。

提示：可能很多人都写过判断一个字符串是不是对称的函数，这个题目可以看成是该函数的加强版。

**3、编程判断俩个链表是否相交**

给出俩个单向链表的头指针，比如h1，h2，判断这俩个链表是否相交。为了简化问题，我们假设俩个链表均不带环。

问题扩展：

* 如果链表可能有环列?
* 如果需要求出俩个链表相交的第一个节点列?

**4、逆序输出链表**

输入一个链表的头结点，从尾到头反过来输出每个结点的值。

**5、在O(1)时间内删除单链表结点**

给定单链表的一个结点的指针，同时该结点不是尾结点，此外没有指向其它任何结点的指针，请在O(1)时间内删除该结点。

**6、找出链表的第一个公共结点**

两个单向链表，找出它们的第一个公共结点。

**7、在字符串中删除特定的字符**

输入两个字符串，从第一字符串中删除第二个字符串中所有的字符。

例如，输入”They are students.”和”aeiou”，则删除之后的第一个字符串变成”Thy r stdnts.”。

**8、字符串的匹配**

在一篇英文文章中查找指定的人名，人名使用二十六个英文字母（可以是大写或小写）、空格以及两个通配符组成（*、?），通配符“*”表示零个或多个任意字母，通配符“?”表示一个任意字母。如：“J\* Smi??” 可以匹配“John Smith” .

**9、字符个数的统计**

char \*str = "AbcABca";写出一个函数，查找出每个字符的个数，区分大小写，要求时间复杂度是n（提示用ASCII码）

**10、最小子串**

给一篇文章，里面是由一个个单词组成，单词中间空格隔开，再给一个字符串指针数组，比如 char \*str[]={"hello","world","good"};

求文章中包含这个字符串指针数组的最小子串。注意，只要包含即可，没有顺序要求。

提示：文章也可以理解为一个大的字符串数组，单词之前只有空格，没有标点符号。

**11、字符串的集合**

给定一个字符串的集合，格式如：{aaa bbb ccc}， {bbb ddd}，{eee fff}，{ggg}，{ddd hhh}要求将其中交集不为空的集合合并，要求合并完成后的集合之间无交集，例如上例应输出{aaa bbb ccc ddd hhh}，{eee fff}， {ggg}。

提示：并查集。

**12、五笔编码**

五笔的编码范围是a ~ y的25个字母，从1位到4位的编码，如果我们把五笔的编码按字典序排序，形成一个数组如下： a, aa, aaa, aaaa, aaab, aaac, … …, b, ba, baa, baaa, baab, baac … …, yyyw, yyyx, yyyy其中a的Index为0，aa的Index为1，aaa的Index为2，以此类推。

* 编写一个函数，输入是任意一个编码，比如baca，输出这个编码对应的Index；
* 编写一个函数，输入是任意一个Index，比如12345，输出这个Index对应的编码。

**13、最长重复子串**

一个长度为10000的字符串，写一个算法，找出最长的重复子串，如abczzacbca,结果是bc。

提示：此题是后缀树/数组的典型应用，即是求后缀数组的height[]的最大值。

**14、字符串的压缩**

一个字符串，压缩其中的连续空格为1个后，对其中的每个字串逆序打印出来。比如"abc efg hij"打印为"cba gfe jih"。

**15、最大重复出现子串**

输入一个字符串，如何求最大重复出现的字符串呢？比如输入ttabcftrgabcd,输出结果为abc, canffcancd,输出结果为can。

给定一个字符串，求出其最长的重复子串。

分析：使用后缀数组，对一个字符串生成相应的后缀数组后，然后再排序，排完序依次检测相邻的两个字符串的开头公共部分。 这样的时间复杂度为：

* 生成后缀数组 O(N)
* 排序 O(NlogN\*N) 最后面的 N 是因为字符串比较也是 O(N)
* 依次检测相邻的两个字符串 O(N \* N)

故最终总的时间复杂度是 O(N^2\*logN)

**16、字符串的删除**

删除模式串中出现的字符，如“welcome to asted”,模式串为“aeiou”那么得到的字符串为“wlcm t std"，要求性能最优。

**17、字符串的移动**

字符串为\*号和26个字母的任意组合，把 \*号都移动到最左侧，把字母移到最右侧并保持相对顺序不变，要求时间和空间复杂度最小。

**18、字符串的包含**

输入：

L:“hello”“july”

S:“hellomehellojuly”

输出：S中包含的L一个单词，要求这个单词只出现一次，如果有多个出现一次的，输出第一个这样的单词。

**19、倒数第n个元素**

链表倒数第n个元素。

提示：设置一前一后两个指针，一个指针步长为1，另一个指针步长为n，当一个指针走到链表尾端时，另一指针指向的元素即为链表倒数第n个元素。

**20、回文字符串**

将一个很长的字符串，分割成一段一段的子字符串，子字符串都是回文字符串。有回文字符串就输出最长的，没有回文就输出一个一个的字符。

例如：

habbafgh

输出h,abba,f,g,h。

提示：一般的人会想到用后缀数组来解决这个问题。

**21、最长连续字符**

用递归算法写一个函数，求字符串最长连续字符的长度，比如aaaabbcc的长度为4，aabb的长度为2，ab的长度为1。

**22、字符串反转**

实现字符串反转函数。

**22、字符串压缩**

通过键盘输入一串小写字母(a~z)组成的字符串。请编写一个字符串压缩程序，将字符串中连续出席的重复字母进行压缩，并输出压缩后的字符串。 压缩规则：

* 仅压缩连续重复出现的字符。比如字符串"abcbc"由于无连续重复字符，压缩后的字符串还是"abcbc"。
* 压缩字段的格式为"字符重复的次数+字符"。例如：字符串"xxxyyyyyyz"压缩后就成为"3x6yz"。

要求实现函数： void stringZip(const char \*pInputStr, long lInputLen, char \*pOutputStr);

* 输入pInputStr： 输入字符串lInputLen： 输入字符串长度
* 输出 pOutputStr： 输出字符串，空间已经开辟好，与输入字符串等长；

注意：只需要完成该函数功能算法，中间不需要有任何IO的输入输出

示例

* 输入：“cccddecc” 输出：“3c2de2c”
* 输入：“adef” 输出：“adef”
* 输入：“pppppppp” 输出：“8p”

**23、集合的差集**

已知集合A和B的元素分别用不含头结点的单链表存储，请求集合A与B的差集，并将结果保存在集合A的单链表中。例如，若集合A={5,10,20,15,25,30}，集合B={5,15,35,25}，完成计算后A={10,20,30}。

**24、最长公共子串**

给定字符串A和B，输出A和B中的第一个最长公共子串，比如A=“wepiabc B=“pabcni”，则输出“abc”。

**25、均分01**

给定一个字符串，长度不超过100，其中只包含字符0和1,并且字符0和1出现得次数都是偶数。你可以把字符串任意切分，把切分后得字符串任意分给两个人，让两个人得到的0的总个数相等，得到的1的总个数也相等。

例如，输入串是010111,我们可以把串切位01, 011,和1，把第1段和第3段放在一起分给一个人，第二段分给另外一个人，这样每个人都得到了1个0和两个1。我们要做的是让切分的次数尽可能少。

考虑到最差情况，则是把字符串切分(n - 1)次形成n个长度为1的串。

**26、合法字符串**

用n个不同的字符（编号1 - n），组成一个字符串，有如下2点要求：

* 1、对于编号为i 的字符，如果2 \* i > n，则该字符可以作为最后一个字符，但如果该字符不是作为最后一个字符的话，则该字符后面可以接任意字符；
* 2、对于编号为i的字符，如果2 \* i <= n，则该字符不可以作为最后一个字符，且该字符后面所紧接着的下一个字符的编号一定要 >= 2 \* i。

问有多少长度为M且符合条件的字符串。

例如：N = 2，M = 3。则abb, bab, bbb是符合条件的字符串，剩下的均为不符合条件的字符串。

假定n和m皆满足：2<=n,m<=1000000000)。

**27、最短摘要生成**

你我在百度或谷歌搜索框中敲入本博客名称的前4个字“结构之法”，便能在第一个选项看到本博客的链接，如下图2所示：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/21~22/22.1.gif)

在上面所示的图2中，搜索结果“结构之法算法之道-博客频道-CSDN.NET”下有一段说明性的文字：“程序员面试、算法研究、编程艺术、红黑树4大经典原创系列集锦与总结 作者：July--结构之法算法...”，我们把这段文字称为那个搜索结果的摘要，亦即最短摘要。我们的问题是，请问，这个最短摘要是怎么生成的呢?

**28、实现memcpy函数**

已知memcpy的函数为： void\* memcpy(void\* dest , const void\* src , size\_t count)其中dest是目的指针，src是源指针。不调用c++/c的memcpy库函数，请编写memcpy。

分析：参考代码如下：

void\* memcpy(void \*dst, const void \*src, size\_t count)

{

//安全检查

assert( (dst != NULL) && (src != NULL) );

unsigned char \*pdst = (unsigned char \*)dst;

const unsigned char \*psrc = (const unsigned char \*)src;

//防止内存重复

assert(!(psrc<=pdst && pdst<psrc+count));

assert(!(pdst<=psrc && psrc<pdst+count));

while(count--)

{

\*pdst = \*psrc;

pdst++;

psrc++;

}

return dst;

}

**29、实现memmove函数**

分析：memmove函数是的标准函数，其作用是把从source开始的num个字符拷贝到destination。最简单的方法是直接复制，但是由于它们可能存在内存的重叠区，因此可能覆盖了原有数据。

比如当source+count>=dest&&source<dest时，dest可能覆盖了原有source的数据。解决办法是从后往前拷贝，对于其它情况，则从前往后拷贝。

参考代码如下：

//void \* memmove ( void \* destination, const void \* source, size\_t num );)

void\* memmove(void\* dest, void\* source, size\_t count)

{

void\* ret = dest;

if (dest <= source || dest >= (source + count))

{

//正向拷贝

//copy from lower addresses to higher addresses

while (count --)

\*dest++ = \*source++;

}

else

{

//反向拷贝

//copy from higher addresses to lower addresses

dest += count - 1;

source += count - 1;

while (count--)

\*dest-- = \*source--;

}

return ret;

}

* **第二章 数组** 本章导读

笔试和面试中，除了字符串，另一类出现频率极高的问题便是与数组相关的问题。在阅读完第1章和本第二章后，读者会慢慢了解到解决面试编程题的有几种常用思路。首先一般考虑“万能的”暴力穷举（递归、回溯），如求n个数的全排列或八皇后（N皇后问题）。但因为穷举时间复杂度通常过高，所以需要考虑更好的方法，如分治法（通过分而治之，然后归并），以及空间换时间（如活用哈希表）。

此外，选择合适的数据结构可以显著提升效率，如寻找最小的k个数中，用堆代替数组。

再有，如果题目允许排序，则可以考虑排序。比如，寻找和为定值的两个数中，先排序，然后用前后两个指针往中间扫。而如果如果已经排好序了（如杨氏矩阵查找中），则想想有无必要二分。但是，如果题目不允许排序呢？这个时候，我们可以考虑不改变数列顺序的贪心算法（如最小生成树Prim、Kruskal及最短路dijkstra），或动态规划（如 01背包问题，每一步都在决策）。

最后，注意细节处理，不要忽略边界条件，如字符串转换成整数。

# 2.1寻找最小的k个数

## 题目描述

输入n个整数，输出其中最小的k个。

## 分析与解法

### 解法一

要求一个序列中最小的k个数，按照惯有的思维方式，则是先对这个序列从小到大排序，然后输出前面的最小的k个数。

至于选取什么的排序方法，我想你可能会第一时间想到快速排序（我们知道，快速排序平均所费时间为n\*logn），然后再遍历序列中前k个元素输出即可。因此，总的时间复杂度：O（n \* log n)+O(k)=O（n \* log n）。

### 解法二

咱们再进一步想想，题目没有要求最小的k个数有序，也没要求最后n-k个数有序。既然如此，就没有必要对所有元素进行排序。这时，咱们想到了用选择或交换排序，即：

1、遍历n个数，把最先遍历到的k个数存入到大小为k的数组中，假设它们即是最小的k个数；  
2、对这k个数，利用选择或交换排序找到这k个元素中的最大值kmax（找最大值需要遍历这k个数，时间复杂度为O（k））；  
3、继续遍历剩余n-k个数。假设每一次遍历到的新的元素的值为x，把x与kmax比较：如果x < kmax ，用x替换kmax，并回到第二步重新找出k个元素的数组中最大元素kmax‘；如果x >= kmax，则继续遍历不更新数组。

每次遍历，更新或不更新数组的所用的时间为O（k）或O（0）。故整趟下来，时间复杂度为n\*O（k）=O（n\*k）。

### 解法三

更好的办法是维护容量为k的最大堆，原理跟解法二的方法相似：

* 1、用容量为k的最大堆存储最先遍历到的k个数，同样假设它们即是最小的k个数；
* 2、堆中元素是有序的，令k1<k2<...<kmax（kmax设为最大堆中的最大元素）
* 3、遍历剩余n-k个数。假设每一次遍历到的新的元素的值为x，把x与堆顶元素kmax比较：如果x < kmax，用x替换kmax，然后更新堆（用时logk）；否则不更新堆。

这样下来，总的时间复杂度:O（k+（n-k）\*logk）=O（n\*logk）。此方法得益于堆中进行查找和更新的时间复杂度均为：O(logk)（若使用解法二：在数组中找出最大元素，时间复杂度：O（k））。

### 解法四

在《数据结构与算法分析--c语言描述》一书，第7章第7.7.6节中，阐述了一种在平均情况下，时间复杂度为O（N）的快速选择算法。如下述文字：

* 选取S中一个元素作为枢纽元v，将集合S-{v}分割成S1和S2，就像快速排序那样
  + 如果k <= |S1|，那么第k个最小元素必然在S1中。在这种情况下，返回QuickSelect(S1, k)。
  + 如果k = 1 + |S1|，那么枢纽元素就是第k个最小元素，即找到，直接返回它。
  + 否则，这第k个最小元素就在S2中，即S2中的第（k - |S1| - 1）个最小元素，我们递归调用并返回QuickSelect(S2, k - |S1| - 1)。

此算法的平均运行时间为O(n)。

示例代码如下：

//QuickSelect 将第k小的元素放在 a[k-1]

void QuickSelect( int a[], int k, int left, int right )

{

int i, j;

int pivot;

if( left + cutoff <= right )

{

pivot = median3( a, left, right );

//取三数中值作为枢纽元，可以很大程度上避免最坏情况

i = left; j = right - 1;

for( ; ; )

{

while( a[ ++i ] < pivot ){ }

while( a[ --j ] > pivot ){ }

if( i < j )

swap( &a[ i ], &a[ j ] );

else

break;

}

//重置枢纽元

swap( &a[ i ], &a[ right - 1 ] );

if( k <= i )

QuickSelect( a, k, left, i - 1 );

else if( k > i + 1 )

QuickSelect( a, k, i + 1, right );

}

else

InsertSort( a + left, right - left + 1 );

}

这个快速选择SELECT算法，类似快速排序的划分方法。N个数存储在数组S中，再从数组中选取“中位数的中位数”作为枢纽元X，把数组划分为Sa和Sb俩部分，Sa<=X<=Sb，如果要查找的k个元素小于Sa的元素个数，则返回Sa中较小的k个元素，否则返回Sa中所有元素+Sb中小的k-|Sa|个元素，这种解法在平均情况下能做到O(n)的复杂度。

更进一步，《算法导论》第9章第9.3节介绍了一个最坏情况下亦为O(n)时间的SELECT算法，有兴趣的读者可以参看。

## 举一反三

1、谷歌面试题：输入是两个整数数组，他们任意两个数的和又可以组成一个数组，求这个和中前k个数怎么做？

分析：

“假设两个整数数组为A和B，各有N个元素，任意两个数的和组成的数组C有N^2个元素。

那么可以把这些和看成N个有序数列：

A[1]+B[1] <= A[1]+B[2] <= A[1]+B[3] <=…

A[2]+B[1] <= A[2]+B[2] <= A[2]+B[3] <=…

…

A[N]+B[1] <= A[N]+B[2] <= A[N]+B[3] <=…

问题转变成，在这N^2个有序数列里，找到前k小的元素”

2、有两个序列A和B,A=(a1,a2,...,ak),B=(b1,b2,...,bk)，A和B都按升序排列。对于1<=i,j<=k，求k个最小的（ai+bj）。要求算法尽量高效。

3、给定一个数列a1,a2,a3,...,an和m个三元组表示的查询，对于每个查询(i，j，k)，输出ai，ai+1，...，aj的升序排列中第k个数。

# 2.2寻找和为定值的两个数

## 题目描述

输入一个数组和一个数字，在数组中查找两个数，使得它们的和正好是输入的那个数字。

要求时间复杂度是O(N)。如果有多对数字的和等于输入的数字，输出任意一对即可。

例如输入数组1、2、4、7、11、15和数字15。由于4+11=15，因此输出4和11。

### 分析与解法

咱们试着一步一步解决这个问题（注意阐述中数列有序无序的区别）：

直接穷举，从数组中任意选取两个数，判定它们的和是否为输入的那个数字。此举复杂度为O(N^2)。很显然，我们要寻找效率更高的解法

题目相当于，对每个a[i]，查找sum-a[i]是否也在原始序列中，每一次要查找的时间都要花费为O(N)，这样下来，最终找到两个数还是需要O（N^2）的复杂度。那如何提高查找判断的速度呢？

答案是二分查找，可以将O(N)的查找时间提高到O(log N)，这样对于N个a[i]，都要花logN的时间去查找相对应的sum-a[i]是否在原始序列中，总的时间复杂度已降为O(N log N)，且空间复杂度为O(1)。 （如果有序，直接二分O(N log N)，如果无序，先排序后二分，复杂度同样为O（N log N + N log N）= O(N log N)，空间复杂度总为O(1)）。

可以继续优化做到时间O(N)么？

#### 解法一

根据前面的分析，a[i]在序列中，如果a[i]+a[k]=sum的话，那么sum-a[i]（a[k])也必然在序列中。 举个例子，如下： 原始序列：

* 1、 2、 4、 7、11、15

用输入数字15减一下各个数，得到对应的序列为：

* 14、13、11、8、4、 0

第一个数组以一指针i 从数组最左端开始向右扫描，第二个数组以一指针j 从数组最右端开始向左扫描，如果第一个数组出现了和第二个数组一样的数，即a[i]=a[j]，就找出这俩个数来了。 如上，i，j最终在第一个，和第二个序列中找到了相同的数4和11，所以符合条件的两个数，即为4+11=15。 怎么样，两端同时查找，时间复杂度瞬间缩短到了O(N)，但却同时需要O(N)的空间存储第二个数组。

#### 解法二

当题目对时间复杂度要求比较严格时，我们可以考虑下用空间换时间，上述解法一即是此思想，此外，构造hash表也是典型的用空间换时间的处理办法。

即给定一个数字，根据hash映射查找另一个数字是否也在数组中，只需用O(1)的时间，前提是经过O(N)时间的预处理，和用O(N)的空间构造hash表。

但能否做到在时间复杂度为O(N)的情况下，空间复杂度能进一步降低达到O(1)呢？

#### 解法三

如果数组是无序的，先排序(N log N)，然后用两个指针i，j，各自指向数组的首尾两端，令i=0，j=n-1，然后i++，j--，逐次判断a[i]+a[j]?=sum，

* 如果某一刻a[i]+a[j] > sum，则要想办法让sum的值减小，所以此刻i不动，j--；
* 如果某一刻a[i]+a[j] < sum，则要想办法让sum的值增大，所以此刻i++，j不动。

所以，数组无序的时候，时间复杂度最终为O(N log N + N)=O(N log N)。

如果原数组是有序的，则不需要事先的排序，直接用两指针分别从头和尾向中间扫描，O(N)搞定，且空间复杂度还是O(1)。

下面，咱们先来实现此思路（这里假定数组已经是有序的），代码可以如下编写：

void TwoSum(int data[], unsigned int length, int sum)

{

//sort(s, s+n); 如果数组非有序的，那就事先排好序O(N log N)

int begin = 0;

int end = length - 1;

//俩头夹逼，或称两个指针两端扫描法，很经典的方法，O(N)

while (begin < end)

{

long currSum = data[begin] + data[end];

if (currSum == sum)

{

//题目只要求输出满足条件的任意一对即可

printf("%d %d\n", data[begin], data[end]);

//如果需要所有满足条件的数组对，则需要加上下面两条语句：

//begin++

//end--

break;

}

else{

if (currSum < sum)

begin++;

else

end--;

}

}

}

### 解法总结

不论原序列是有序还是无序，解决这类题有以下三种办法：

* 1、二分（若无序，先排序后二分），时间复杂度总为O(N log N)，空间复杂度为O（1）；
* 2、扫描一遍X-S[i] 映射到一个数组或构造hash表，时间复杂度为O(N)，空间复杂度为O(N)；
* 3、两个指针两端扫描（若无序，先排序后扫描），时间复杂度最后为：有序O(N)，无序O(N log N + N)=O(N log N)，空间复杂度都为O(1)。

所以，要想达到时间O(N)，空间O(1)的目标，除非原数组是有序的（指针扫描法），不然，当数组无序的话，就只能先排序，后指针扫描法或二分（时间 O(Nlog N)，空间O(1)），或映射或hash（时间O(N)，空间O(N)）。时间或空间，必须牺牲一个，达到平衡。

综上，若是数组有序的情况下，优先考虑两个指针两端扫描法，以达到最佳的时O(N)，空O(1)效应。否则，如果要排序的话，时间复杂度最快当然是只能达到O(N log N)，空间O(1)则不在话下。

## 问题扩展

1. 如果在返回找到的两个数的同时，还要求你返回这两个数的位置列？
2. 如果需要输出所有满足条件的整数对呢?
3. 如果把题目中的要你寻找的两个数改为“多个数”，或任意个数列?

## 举一反三

1、在二元树中找出和为某一值的所有路径 输入一个整数和一棵二元树，从树的根结点开始往下访问一直到叶结点所经过的所有结点形成一条路径，然后打印出和与输入整数相等的所有路径。 例如输入整数22和如下二元树

10

/ \  
5 12  
/ \  
4 7

则打印出两条路径：10, 12和10, 5, 7。 其中，二元树节点的数据结构定义为：

struct BinaryTreeNode // a node in the binary tree

{

int m\_nValue; // value of node

BinaryTreeNode \*m\_pLeft; // left child of node

BinaryTreeNode \*m\_pRight; // right child of node

};

2、有一个数组a，设有一个值n。在数组中找到两个元素a[i]和a[j]，使得a[i]+a[j]等于n，求出所有满足以上条件的i和j。

3、3-sum问题

给定一个整数数组，判断能否从中找出3个数a、b、c，使得他们的和为0，如果能，请找出所有满足和为0个3个数对。

4、4-sum问题

给定一个整数数组，判断能否从中找出4个数a、b、c、d，使得他们的和为0，如果能，请找出所有满足和为0个4个数对。

# 2.3寻找和为定值的多个数

## 题目描述

输入两个整数n和sum，从数列1，2，3.......n 中随意取几个数，使其和等于sum，要求将其中所有的可能组合列出来。

## 分析与解法

### 解法一

注意到取n，和不取n个区别即可，考虑是否取第n个数的策略，可以转化为一个只和前n-1个数相关的问题。

* 如果取第n个数，那么问题就转化为“取前n-1个数使得它们的和为sum-n”，对应的代码语句就是sumOfkNumber(sum - n, n - 1)；
* 如果不取第n个数，那么问题就转化为“取前n-1个数使得他们的和为sum”，对应的代码语句为sumOfkNumber(sum, n - 1)。

参考代码如下：

list<int>list1;

void SumOfkNumber(int sum, int n)

{

// 递归出口

if (n <= 0 || sum <= 0)

return;

// 输出找到的结果

if (sum == n)

{

// 反转list

list1.reverse();

for (list<int>::iterator iter = list1.begin(); iter != list1.end(); iter++)

cout << \*iter << " + ";

cout << n << endl;

list1.reverse()//此处还需反转回来

}

list1.push\_front(n); //典型的01背包问题

SumOfkNumber(sum - n, n - 1); //“放”n，前n-1个数“填满”sum-n

list1.pop\_front();

SumOfkNumber(sum, n - 1); //不“放”n，n-1个数“填满”sum

}

### 解法二

这个问题属于子集和问题（也是背包问题）。本程序采用回溯法+剪枝，其中X数组是解向量，t=∑(1,..,k-1)Wi\*Xi, r=∑(k,..,n)Wi，且

* 若t+Wk+W(k+1)<=M,则Xk=true，递归左儿子(X1,X2,..,X(k-1),1)；否则剪枝；
* 若t+r-Wk>=M && t+W(k+1)<=M,则置Xk=0，递归右儿子(X1,X2,..,X(k-1),0)；否则剪枝；

本题中W数组就是(1,2,..,n),所以直接用k代替WK值。

代码编写如下：

//输入t， r， 尝试Wk

void SumOfkNumber(int t, int k, int r, int& M, bool& flag, bool\* X)

{

X[k] = true; // 选第k个数

if (t + k == M) // 若找到一个和为M，则设置解向量的标志位，输出解

{

flag = true;

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

{

if (X[i] == 1)

{

printf("%d ", i);

}

}

printf("\n");

}

else

{ // 若第k+1个数满足条件，则递归左子树

if (t + k + (k + 1) <= M)

{

SumOfkNumber(t + k, k + 1, r - k, M, flag, X);

}

// 若不选第k个数，选第k+1个数满足条件，则递归右子树

if ((t + r - k >= M) && (t + (k + 1) <= M))

{

X[k] = false;

SumOfkNumber(t, k + 1, r - k, M, flag, X);

}

}

}

void search(int& N, int& M)

{

// 初始化解空间

bool\* X = (bool\*)malloc(sizeof(bool)\* (N + 1));

memset(X, false, sizeof(bool)\* (N + 1));

int sum = (N + 1) \* N \* 0.5f;

if (1 > M || sum < M) // 预先排除无解情况

{

printf("not found\n");

return;

}

bool f = false;

SumOfkNumber(0, 1, sum, M, f, X);

if (!f)

{

printf("not found\n");

}

free(X);

}

## 0-1背包问题

0-1背包问题是最基础的背包问题，其具体描述为：有N件物品和一个容量为V的背包。放入第i件物品耗费的费用是Ci，得到的价值是Wi。求解将哪些物品装入背包可使价值总和最大。

简单分析下：这是最基础的背包问题，特点是每种物品仅有一件，可以选择放或不放。用子问题定义状态：即F[i, v]表示前i件物品恰放入一个容量为v的背包可以获得的最大价值。则其状态转移方程便是：

* F[i, v] = max{F[i-1, v], F[i-1, v-Ci ] + Wi}

根据前面的分析，我们不难理解这个方程的意义：“将前i件物品放入容量为v的背包中”这个子问题，若只考虑第i件物品的策略（放或不放），那么就可以转化为一个只和前 i-1 件物品相关的问题。即：

* 如果不放第i件物品，那么问题就转化为“前i-1件物品放入容量为v的背包中”，价值为 F[i-1, v ]；
* 如果放第i件物品，那么问题就转化为“前i-1件物品放入剩下的容量为v-Ci的背包中”，此时能获得的最大价值就是F[i-1, v-Ci]再加上通过放入第i件物品获得的价值Wi。

伪代码如下：

F[0,0...V] ← 0

for i ← 1 to N

for v ← Ci to V

F[i, v] ← max{F[i-1, v], F[i-1, v-Ci] + Wi }

这段代码的时间和空间复杂度均为 O(VN)，其中时间复杂度应该已经不能再优化了，但空间复杂度却可以优化到O(V)。感兴趣的读者可以继续思考或者参考网上一个不错的文档《背包问题九讲》。

## 举一反三

1、《挑战程序设计竞赛》的开篇有个类似的抽签问题，挺有意思，题目如下：

将写有数字的n个纸片放入一个纸箱子中，然后你和你的朋友从纸箱子中抽取4张纸片，每次记下纸片上的数字后放回子箱子中，如果这4个数字的和是m，代表你赢，否则就是你的朋友赢。

请编写一个程序，当纸片上所写的数字是k1，k2，k3，k4，..，kn时，是否存在抽取4次和为m的方案，如果存在，输出YES；否则，输出NO。

限制条件：

* 1 <= n <= 50
* 1 <= m <= 10^8
* 1 <= ki <= 10^8

分析：最容易想到的方案是用4个for循环直接穷举所有方案，时间复杂度为O（N^4）,主体代码如下：

//通过4重for循环枚举所有方案

for (int a = 0; a < n, a++)

{

for (int b = 0; b < n; b++)

{

for (int c = 0; c < n; c++)

{

for (int d = 0; d < n; d++)

{

if (k[a] + k[b] + k[c] + k[d] == m)

{

f = true;

}

}

}

}

}

但如果当n远大于50时，则程序会显得非常吃力，如此，我们需要找到更好的办法。

提供两个思路：

①最内侧关于d的循环所做的事情：检查是否有d满足ka+ kb +kc + kd = m，移动下式子，等价于：检查是否有d使得kd = m - ka - kb - kc，也就是说，只要检查k中所有元素，判断是否有等于m-ka-kb-ka 的元素即可。设m-ka-kb-ka = x，接下来，就是要看x是否存在于数组k中，此时，可以先把数组k排序，然后利用二分查找在数组k中找x；

②进一步，内侧的两个循环所做的事情：检查是否有c和d满足kc + kd = m - ka -kb。同样，可以预先枚举出kc+kd所得的n^2数字并排好序，便可以利用二分搜索继续求解。

2、给定整数a1、a2、a3、...、an，判断是否可以从中选出若干个数，使得它们的和等于k（k任意给定，且满足-10^8 <= k <= 10^8）。

分析：此题相对于本节“寻找满足条件的多个数”如出一辙，不同的是此题只要求判断，不要求把所有可能的组合给输出来。因为此题需要考虑到加上a[i]和不加上a[i]的情况，故可以采用深度优先搜索的办法，递归解决。

3、有n个数，输出期中所有和为s的k个数的组合。

分析：此题有两个坑，一是这里的n个数是任意给定的，不一定是：1,2,3...n，所以可能有重复的数（如果有重复的数怎么处理？）；二是不要求你输出所有和为s的全部组合，而只要求输出和为s的k个数的组合。

举个例子，假定n=6，这6个数为：1 2 1 3 0 1，如果要求输出和为3的全部组合的话，

* 1 2
* 1 2 0
* 0 3
* 1 1 1
* 1 1 1 0

而题目加了个限制条件，若令k=2，则只要求输出：[{1,2}, {0,3}] 即可。

# 2.4最大连续子数组和

## 题目描述

输入一个整形数组，数组里有正数也有负数。数组中连续的一个或多个整数组成一个子数组，每个子数组都有一个和。 求所有子数组的和的最大值，要求时间复杂度为O(n)。

例如输入的数组为1, -2, 3, 10, -4, 7, 2, -5，和最大的子数组为3, 10, -4, 7, 2， 因此输出为该子数组的和18。

## 分析与解法

### 解法一

求一个数组的最大子数组和，我想最直观最野蛮的办法便是，三个for循环三层遍历，求出数组中每一个子数组的和，最终求出这些子数组的最大的一个值。 令currSum[i, …, j]为数组A中第i个元素到第j个元素的和（其中0 <= i <= j < n），maxSum为最终求到的最大连续子数组的和。

且当全是负数的情况时，我们可以让程序返回0，也可以让程序返回最大的那个负数，这里，我们让程序返回最大的那个负数。

参考代码如下：

int MaxSubArray(int\* A, int n)

{

int maxSum = a[0]; //全负情况，返回最大负数

int currSum = 0;

for (int i = 0; i < n; i++)

{

for (int j = i; j < n; j++)

{

for (int k = i; k <= j; k++)

{

currSum += A[k];

}

if (currSum > maxSum)

maxSum = currSum;

currSum = 0; //这里要记得清零，否则的话sum最终存放的是所有子数组的和。

}

}

return maxSum;

}

此方法的时间复杂度为O(n^3)。

### 解法二

事实上，当我们令currSum为当前最大子数组的和，maxSum为最后要返回的最大子数组的和，当我们往后扫描时，

* 对第j+1个元素有两种选择：要么放入前面找到的子数组，要么做为新子数组的第一个元素；
  + 如果currSum加上当前元素a[j]后不小于a[j]，则令currSum加上a[j]，否则currSum重新赋值，置为下一个元素，即currSum = a[j]。
* 同时，当currSum > maxSum，则更新maxSum = currSum，否则保持原值，不更新。

即

currSum = max(a[j], currSum + a[j])

maxSum = max(maxSum, currSum)

举个例子，当输入数组是1, -2, 3, 10, -4, 7, 2, -5，那么，currSum和maxSum相应的变化为：

* currSum： 0 1 - 1 3 13 9 16 18 13
* maxSum ： 0 1 1 3 13 13 16 18 18

参考代码如下：

int MaxSubArray(int\* a, int n)

{

int currSum = 0;

int maxSum = a[0]; //全负情况，返回最大数

for (int j = 0; j < n; j++)

{

currSum = (a[j] > currSum + a[j]) ? a[j] : currSum + a[j];

maxSum = (maxSum > currSum) ? maxSum : currSum;

}

return maxSum;

}

## 问题扩展

1. 如果数组是二维数组，同样要你求最大子数组的和列?
2. 如果是要你求子数组的最大乘积列?
3. 如果同时要求输出子段的开始和结束列?

## 举一反三

1 给定整型数组，其中每个元素表示木板的高度，木板的宽度都相同，求这些木板拼出的最大矩形的面积。并分析时间复杂度。

此题类似leetcode里面关于连通器的题，需要明确的是高度可能为0，长度最长的矩形并不一定是最大矩形，还需要考虑高度很高，但长度较短的矩形。如[5,4,3,2,4,5,0,7,8,4,6]中最大矩形的高度是[7,8,4,6]组成的矩形，面积为16。

2、环面上的最大子矩形

《算法竞赛入门经典》 P89 页。

3、最大子矩阵和

一个MN的矩阵，找到此矩阵的一个子矩阵，并且这个子矩阵的元素的和是最大的，输出这个最大的值。如果所有数都是负数，就输出0。 例如：35的矩阵：

1 2 0 3 4

2 3 4 5 1

1 1 5 3 0

和最大的子矩阵是：

4 5

5 3

最后输出和的最大值17。

4、允许交换两个数的位置 求最大子数组和。

来源：<https://codility.com/cert/view/certDUMWPM-8RF86G8P9QQ6JC8X/details> 。

# 2.5跳台阶问题

### 题目描述

一个台阶总共有n 级，如果一次可以跳1 级，也可以跳2 级。

求总共有多少总跳法，并分析算法的时间复杂度。

### 分析与解法

#### 解法一

首先考虑最简单的情况。如果只有1级台阶，那显然只有一种跳法。如果有2级台阶，那就有两种跳的方法了：一种是分两次跳，每次跳1级；另外一种就是一次跳2级。

现在我们再来讨论一般情况。我们把n级台阶时的跳法看成是n的函数，记为f(n)。

* 当n>2时，第一次跳的时候就有两种不同的选择：
  + 一是第一次只跳1级，此时跳法数目等于后面剩下的n-1级台阶的跳法数目，即为f(n-1)；
  + 另外一种选择是第一次跳2级，此时跳法数目等于后面剩下的n-2级台阶的跳法数目，即为f(n-2)。

因此n级台阶时的不同跳法的总数f(n)=f(n-1)+f(n-2)。

我们把上面的分析用一个公式总结如下：

/ 1 n = 1

f(n)= 2 n = 2

\ f(n-1) + f(n-2) n > 2

原来上述问题就是我们平常所熟知的Fibonacci数列问题。可编写代码，如下：

long long Fibonacci(unsigned int n)

{

int result[3] = {0, 1, 2};

if (n <= 2)

return result[n];

return Fibonacci(n - 1) + Fibonacci(n - 2);

}

那么，如果一个人上台阶可以一次上1个，2个，或者3个呢？这个时候，公式是这样写的：

/ 1 n = 1

f(n)= 2 n = 2

4 n = 3 //111, 12, 21, 3

\ f(n-1)+f(n-2)+f(n-3) n > 3

#### 解法二

解法一用的递归的方法有许多重复计算的工作，事实上，我们可以从后往前推，一步步利用之前计算的结果递推。

初始化时，dp[0]=dp[1]=1，然后递推计算即可：dp[n] = dp[n-1] + dp[n-2]。

参考代码如下：

//1, 1, 2, 3, 5, 8, 13, 21..

int ClimbStairs(int n)

{

int dp[3] = { 1, 1 };

if (n < 2)

{

return 1;

}

for (int i = 2; i <= n; i++)

{

dp[2] = dp[0] + dp[1];

dp[0] = dp[1];

dp[1] = dp[2];

}

return dp[2];

}

### 举一反三

1、兔子繁殖问题

13世纪意大利数学家斐波那契在他的《算盘书》中提出这样一个问题：有人想知道一年内一对兔子可繁殖成多少对，便筑了一道围墙把一对兔子关在里面。已知一对兔子每一个月可以生一对小兔子，而一对兔子出生后.第三个月开始生小兔子假如一年内没有发生死亡，则一对兔子一年内能繁殖成多少对？

分析：这就是斐波那契数列的由来，本节的跳台阶问题便是此问题的变形，只是换了种表述形式。

2、换硬币问题。

想兑换100元钱，有1,2,5,10四种钱，问总共有多少兑换方法。

const int N = 100;

int dimes[] = { 1, 2, 5, 10 };

int arr[N + 1] = { 1 };

for (int i = 0; i < sizeof(dimes) / sizeof(int); ++i)

{

for (int j = dimes[i]; j <= N; ++j)

{

arr[j] += arr[j - dimes[i]];

}

}

此问题还有一个变形，就是打印出路径目前只想到要使用递归来解决这个问题。对此，利用一个vector来保存路径，每进入一层，push\_back一个路径，每退出一层，pop\_back一个路径。

# 2.6奇偶调序

## 题目描述

输入一个整数数组，调整数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。要求时间复杂度为O(n)。

### 分析与解法

最容易想到的办法是从头扫描这个数组，每碰到一个偶数，拿出这个数字，并把位于这个数字后面的所有数字往前挪动一位。挪完之后在数组的末尾有一个空位，然后把该偶数放入这个空位。由于每碰到一个偶数，需要移动O(n)个数字，所以这种方法总的时间复杂度是O(n^2)，不符合题目要求。

事实上，若把奇数看做是小的数，偶数看做是大的数，那么按照题目所要求的奇数放在前面偶数放在后面，就相当于小数放在前面大数放在后面，联想到快速排序中的partition过程，不就是通过一个主元把整个数组分成大小两个部分么，小于主元的小数放在前面，大于主元的大数放在后面。

而partition过程有以下两种实现：

* 一头一尾两个指针往中间扫描，如果头指针遇到的数比主元大且尾指针遇到的数比主元小，则交换头尾指针所分别指向的数字；
* 一前一后两个指针同时从左往右扫，如果前指针遇到的数比主元小，则后指针右移一位，然后交换各自所指向的数字。

类似这个partition过程，奇偶排序问题也可以分别借鉴partition的两种实现解决。

为何？比如partition的实现一中，如果最终是为了让整个序列元素从小到大排序，那么头指针理应指向的就是小数，而尾指针理应指向的就是大数，故当头指针指的是大数且尾指针指的是小数的时候就不正常，此时就当交换。

#### 解法一

借鉴partition的实现一，我们可以考虑维护两个指针，一个指针指向数组的第一个数字，我们称之为头指针，向右移动；一个指针指向最后一个数字，称之为尾指针，向左移动。

这样，两个指针分别从数组的头部和尾部向数组的中间移动，如果第一个指针指向的数字是偶数而第二个指针指向的数字是奇数，我们就交换这两个数字。

因为按照题目要求，最终是为了让奇数排在数组的前面，偶数排在数组的后面，所以头指针理应指向的就是奇数，尾指针理应指向的就是偶数，故当头指针指向的是偶数且尾指针指向的是奇数时，我们就当立即交换它们所指向的数字。

思路有了，接下来，写代码实现：

//判断是否为奇数

bool IsOddNumber(int data)

{

return data & 1 == 1;

}

//交换两个元素

void swap(int\* x, int\* y)

{

int temp = \*x;

\*x = \*y;

\*y = temp;

}

//奇偶互换

void OddEvenSort(int \*pData, unsigned int length)

{

if (pData == NULL || length == 0)

return;

int \*pBegin = pData;

int \*pEnd = pData + length - 1;

while (pBegin < pEnd)

{

//如果pBegin指针指向的是奇数，正常，向右移

if (IsOddNumber(\*pBegin))

{

pBegin++;

}

//如果pEnd指针指向的是偶数，正常，向左移

else if (!IsOddNumber(\*pEnd))

{

pEnd--;

}

else

{

//否则都不正常，交换

swap(\*pBegin, \*pEnd);

}

}

}

本方法通过头尾两个指针往中间扫描，一次遍历完成所有奇数偶数的重新排列，时间复杂度为O(n)。

#### 解法二

我们先来看看快速排序partition过程的第二种实现是具体怎样的一个原理。

partition分治过程，每一趟排序的过程中，选取的主元都会把整个数组排列成一大一小的序列，继而递归排序完整个数组。如下伪代码所示：

PARTITION(A, p, r)

1 x ← A[r]

2 i ← p - 1

3 for j ← p to r - 1

4 do if A[j] ≤ x

5 then i ← i + 1

6 exchange A[i] <-> A[j]

7 exchange A[i + 1] <-> A[r]

8 return i + 1

举个例子如下：现要对数组data = {2, 8,7, 1, 3, 5, 6, 4}进行快速排序，为了表述方便，令i指向数组头部前一个位置，j指向数组头部元素，j在前，i在后，双双从左向右移动。

① j 指向元素2时，i 也指向元素2，2与2互换不变

i p/j

2 8 7 1 3 5 6 4(主元)

② 于是j 继续后移，直到指向了1，1 <= 4，于是i++，i 指向8，故j 所指元素1 与 i 所指元素8 位置互换：

i j

2 1 7 8 3 5 6 4

③ j 继续后移，指到了元素3,3 <= 4，于是同样i++，i 指向7，故j 所指元素3 与 i 所指元素7 位置互换：

i j

2 1 3 8 7 5 6 4

④ j 一路后移，没有再碰到比主元4小的元素：

i j

2 1 3 8 7 5 6 4

⑤ 最后，A[i + 1] <-> A[r]，即8与4交换，所以，数组最终变成了如下形式：

2 1 3 4 7 5 6 8

这样，快速排序第一趟完成。就这样，4把整个数组分成了俩部分，2 1 3,7 5 6 8，再递归对这两部分分别进行排序。

借鉴partition的上述实现，我们也可以维护两个指针i和j，一个指针指向数组的第一个数的前一个位置，我们称之为后指针i，向右移动；一个指针指向数组第一个数，称之为前指针j，也向右移动，且前指针j先向右移动。如果前指针j指向的数字是奇数，则令i指针向右移动一位，然后交换i和j指针所各自指向的数字。

因为按照题目要求，最终是为了让奇数排在数组的前面，偶数排在数组的后面，所以i指针理应指向的就是奇数，j指针理应指向的就是偶数，所以，当j指针指向的是奇数时，不正常，我们就当让i++，然后交换i和j指针所各自指向的数字。

参考代码如下：

//奇偶互换

void OddEvenSort2(int data[], int lo, int hi)

{

int i = lo - 1;

for (int j = lo; j < hi; j++)

{

//data[j]指向奇数，交换

if ( IsOddNumber(data[j]) )

{

i = i + 1;

swap(&data[i], &data[j]);

}

}

swap(&data[i + 1], &data[hi]);

}

此解法一前一后两个指针同时向右扫描的过程中，也是一次遍历完成奇数偶数的重新排列，故时间复杂度也为O(n)。

### 举一反三

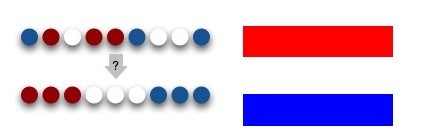
一个未排序整数数组，有正负数，重新排列使负数排在正数前面，并且要求不改变原来的正负数之间相对顺序，比如： input: 1,7,-5,9,-12,15 ans: -5,-12,1,7,9,15 要求时间复杂度O(n),空间O(1)。

分析：如果本题没有这个要求“并且要求不改变原来的正负数之间相对顺序”，那么同奇偶数排序是一道题，但加上这个不能改变正负数之间的相对顺序后，便使得问题变得比较艰难了，若有兴趣，读者可以参考这篇论文《STABLE MINIMUM SPACE PARTITIONING IN LINEAR TIME》。

# 2.7荷兰国旗

### 题目描述

拿破仑席卷欧洲大陆之后，代表自由，平等，博爱的竖色三色旗也风靡一时。荷兰国旗就是一面三色旗（只不过是横向的），自上而下为红白蓝三色。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/41~42/41.1.jpg)

该问题本身是关于三色球排序和分类的，由荷兰科学家Dijkstra提出。由于问题中的三色小球有序排列后正好分为三类，Dijkstra就想象成他母国的国旗，于是问题也就被命名为荷兰旗问题（Dutch National Flag Problem）。

下面是问题的正规描述： 现有n个红白蓝三种不同颜色的小球，乱序排列在一起，请通过两两交换任意两个球，使得从左至右，依次是一些红球、一些白球、一些蓝球。

### 分析与解法

初看此题，我们貌似除了暴力解决并无好的办法，但联想到我们所熟知的快速排序算法呢？

我们知道，快速排序依托于一个partition分治过程，在每一趟排序的过程中，选取的主元都会把整个数组排列成一大一小的部分，那我们是否可以借鉴partition过程设定三个指针完成重新排列，使得所有球排列成三个不同颜色的球呢？

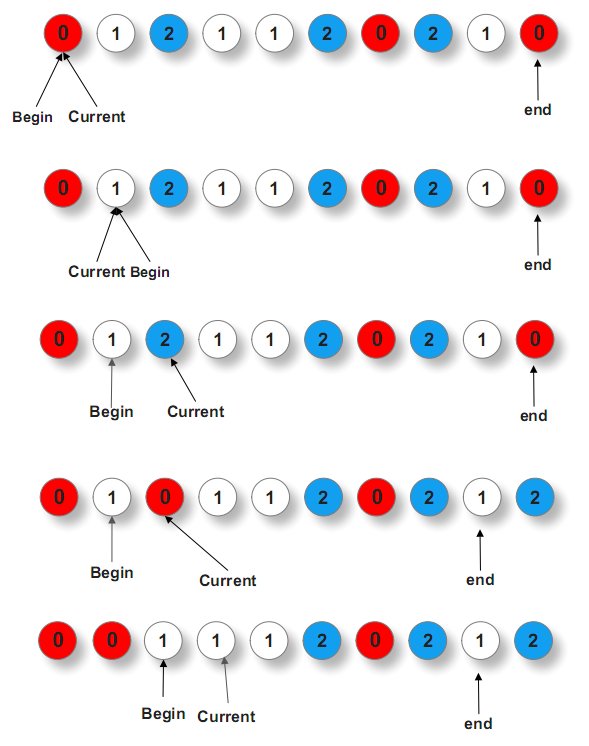
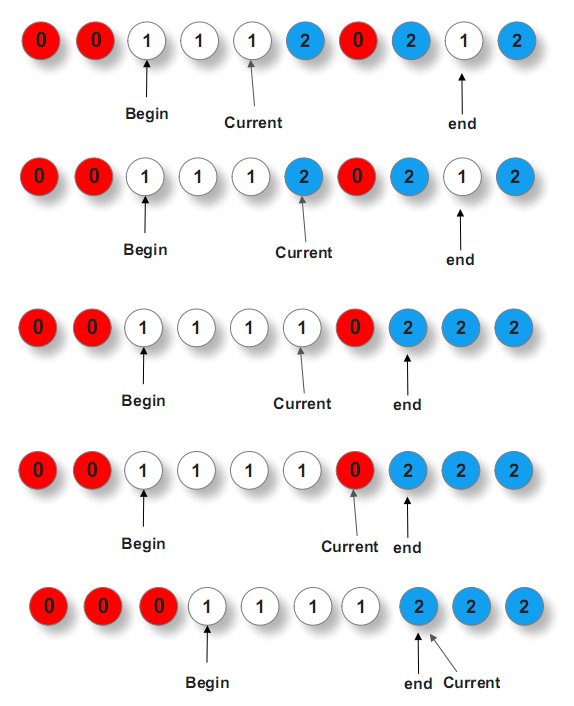
#### 解法一

通过前面的分析得知，这个问题类似快排中partition过程，只是需要用到三个指针：一个前指针begin，一个中指针current，一个后指针end，current指针遍历整个数组序列，当

1. current指针所指元素为0时，与begin指针所指的元素交换，而后current++，begin++ ；
2. current指针所指元素为1时，不做任何交换（即球不动），而后current++ ；
3. current指针所指元素为2时，与end指针所指的元素交换，而后，current指针不动，end-- 。

为什么上述第3点中，current指针所指元素为2时，与end指针所指元素交换之后，current指针不能动呢？因为第三步中current指针所指元素与end指针所指元素交换之前，如果end指针之前指的元素是0，那么与current指针所指元素交换之后，current指针此刻所指的元素是0，此时，current指针能动么？不能动，因为如上述第1点所述，如果current指针所指的元素是0，还得与begin指针所指的元素交换。

ok，说这么多，你可能不甚明了，直接引用下gnuhpc的图，就一目了然了：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/41~42/41.3.jpg)[](https://camo.githubusercontent.com/39648f0f0b2a257306d5560958a2d79a5d5367a8/687474703a2f2f68692e6373646e2e6e65742f6174746163686d656e742f3230313130322f32352f383339343332335f31323938363431323235654a34462e6a7067)

参考代码如下：

//引用自gnuhpc

while( current<=end )

{

if( array[current] ==0 )

{

swap(array[current],array[begin]);

current++;

begin++;

}

else if( array[current] == 1 )

{

current++;

}

else //When array[current] =2

{

swap(array[current],array[end]);

end--;

}

}

### 举一反三

给定一个字符串里面只有"R" "G" "B" 三个字符，请排序，最终结果的顺序是R在前 G中 B在后。

要求：空间复杂度是O(1)，且只能遍历一次字符串。

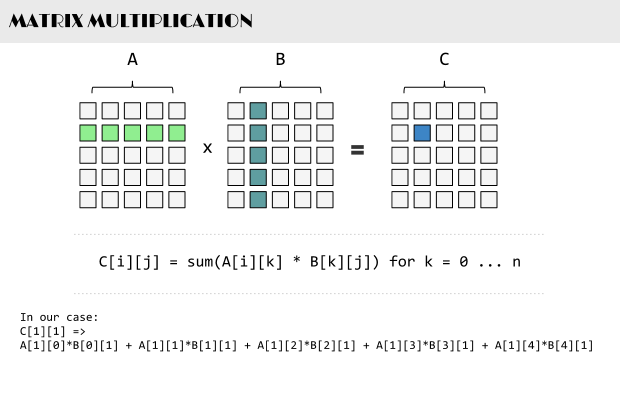
# 2.8矩阵相乘

## 题目描述

请编程实现矩阵乘法，并考虑当矩阵规模较大时的优化方法。

## 分析与解法

根据wikipedia上的介绍：两个矩阵的乘法仅当第一个矩阵A的行数和另一个矩阵B的列数相等时才能定义。如A是m×n矩阵，B是n×p矩阵，它们的乘积AB是一个m×p矩阵，它的一个元素其中 1 ≤ i ≤ m, 1 ≤ j ≤ p。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/41~42/42.1.png)

值得一提的是，矩阵乘法满足结合律和分配率，但并不满足交换律，如下图所示的这个例子，两个矩阵交换相乘后，结果变了：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/41%7E42/42.1-2.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/41~42/42.1-2.png)

下面咱们来具体解决这个矩阵相乘的问题。

### 解法一、暴力解法

其实，通过前面的分析，我们已经很明显的看出，两个具有相同维数的矩阵相乘，其复杂度为O(n^3)，参考代码如下：

//矩阵乘法，3个for循环搞定

void MulMatrix(int\*\* matrixA, int\*\* matrixB, int\*\* matrixC)

{

for(int i = 0; i < 2; ++i)

{

for(int j = 0; j < 2; ++j)

{

matrixC[i][j] = 0;

for(int k = 0; k < 2; ++k)

{

matrixC[i][j] += matrixA[i][k] \* matrixB[k][j];

}

}

}

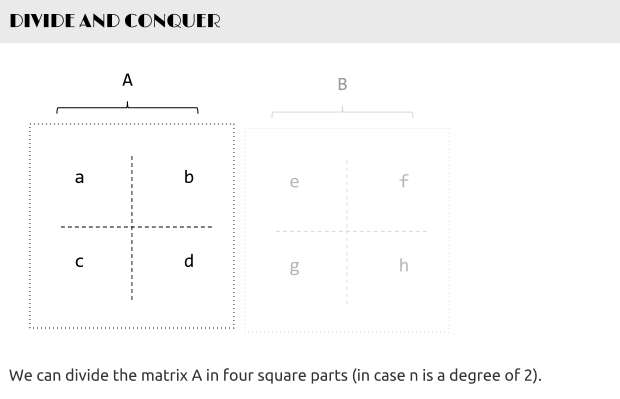
}

### 解法二、Strassen算法

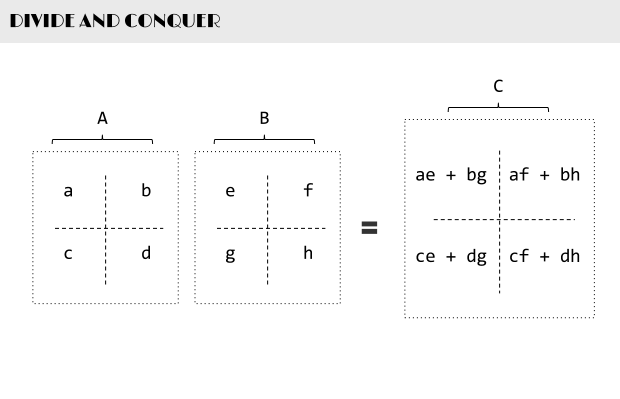
在解法一中，我们用了3个for循环搞定矩阵乘法，但当两个矩阵的维度变得很大时，O（n^3）的时间复杂度将会变得很大，于是，我们需要找到一种更优的解法。

一般说来，当数据量一大时，我们往往会把大的数据分割成小的数据，各个分别处理。遵此思路，如果丢给我们一个很大的两个矩阵呢，是否可以考虑分治的方法循序渐进处理各个小矩阵的相乘，因为我们知道一个矩阵是可以分成更多小的矩阵的。

如下图，当给定一个两个二维矩阵A B时：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/41~42/42.2.png)

这两个矩阵A B相乘时，我们发现在相乘的过程中，有8次乘法运算，4次加法运算：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/41~42/42.3.png)

矩阵乘法的复杂度主要就是体现在相乘上，而多一两次的加法并不会让复杂度上升太多。故此，我们思考，是否可以让矩阵乘法的运算过程中乘法的运算次数减少，从而达到降低矩阵乘法的复杂度呢？答案是肯定的。

1969年，德国的一位数学家Strassen证明O(N^3)的解法并不是矩阵乘法的最优算法，他做了一系列工作使得最终的时间复杂度降低到了O(n^2.80)。

他是怎么做到的呢？还是用上文A B两个矩阵相乘的例子，他定义了7个变量：

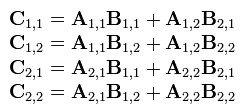
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/41~42/42.4.png)

如此，Strassen算法的流程如下：

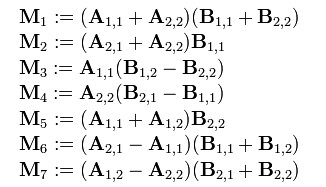
* 两个矩阵A B相乘时，将A, B, C分成相等大小的方块矩阵：

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/41%7E42/42.5.png](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/41~42/42.5.png)

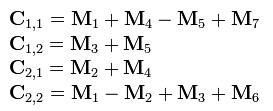
* 可以看出C是这么得来的：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/41~42/42.6.jpeg)

* 现在定义7个新矩阵（读者可以思考下，这7个新矩阵是如何想到的）：

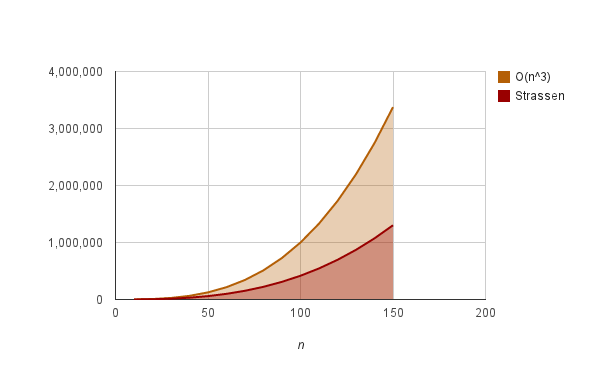
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/41~42/42.7.jpeg)

* 而最后的结果矩阵C 可以通过组合上述7个新矩阵得到：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/41~42/42.8.jpeg)

表面上看，Strassen算法仅仅比通用矩阵相乘算法好一点，因为通用矩阵相乘算法时间复杂度是[equation](https://camo.githubusercontent.com/aeef0df9107adbd6463c9e4fe6748f7815dec922/687474703a2f2f6c617465782e636f6465636f67732e636f6d2f6769662e6c617465783f2537426e253545333d6e2535452537426c6f675f3238253744253744)，而Strassen算法复杂度只是[equation](https://camo.githubusercontent.com/f7e265e7402a954d8853d9724b203e0f648a12bf/687474703a2f2f6c617465782e636f6465636f67732e636f6d2f6769662e6c617465783f2537424f286e2535452537426c6f675f3237253744293d4f286e253545253742322e38303725374429253744)。但随着n的变大，比如当n >> 100时，Strassen算法是比通用矩阵相乘算法变得更有效率。

如下图所示：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/41~42/42.9.png)

根据wikipedia上的介绍，后来，Coppersmith–Winograd 算法把 N\* N大小的矩阵乘法的时间复杂度降低到了：[equation](https://camo.githubusercontent.com/3828207037dbef5a343930907319c428337e9443/687474703a2f2f6c617465782e636f6465636f67732e636f6d2f6769662e6c617465783f2537424f286e253545253742322e33373534373725374429253744)，而2010年，Andrew Stothers再度把复杂度降低到了[equation](https://camo.githubusercontent.com/b4ca1c880f8008d0063ab91cd95d8271dd5846e1/687474703a2f2f6c617465782e636f6465636f67732e636f6d2f6769662e6c617465783f2537424f286e253545253742322e3337333625374429253744)，一年后的2011年，Virginia Williams把复杂度最终定格为：[equation](https://camo.githubusercontent.com/9e7e3d9b8139728ee592e2b6dc0f3ed69605b6d0/687474703a2f2f6c617465782e636f6465636f67732e636f6d2f6769662e6c617465783f2537424f286e253545253742322e3337323725374429253744)。

# 2.9完美洗牌算法

## 题目详情

有个长度为2n的数组{a1,a2,a3,...,an,b1,b2,b3,...,bn}，希望排序后{a1,b1,a2,b2,....,an,bn}，请考虑有无时间复杂度o(n)，空间复杂度0(1)的解法。

**题目来源**：此题是去年2013年UC的校招笔试题，看似简单，按照题目所要排序后的字符串蛮力变化即可，但若要完美的达到题目所要求的时空复杂度，则需要我们花费不小的精力。OK，请看下文详解，一步步优化。

## 分析与解法

### 解法一、蛮力变换

题目要我们怎么变换，咱们就怎么变换。此题@陈利人也分析过，在此，引用他的思路进行说明。为了便于分析，我们取n=4，那么题目要求我们把

a1，a2，a3，a4，**b1，b2，b3，b4**

变成

a1，b1，a2，b2，a3，b3，a4，b4

#### 1.1、步步前移

仔细观察变换前后两个序列的特点，我们可做如下一系列操作：

第①步、确定b1的位置，即让b1跟它前面的a2，a3，a4交换：

a1，b1，a2，a3，a4，**b2，b3，b4**

第②步、接着确定b2的位置，即让b2跟它前面的a3，a4交换：

a1，b1，a2，b2，a3，a4，**b3，b4**

第③步、b3跟它前面的a4交换位置：

a1，b1，a2，b2，a3，b3，a4，b4

b4已在最后的位置，不需要再交换。如此，经过上述3个步骤后，得到我们最后想要的序列。但此方法的时间复杂度为O（N^2），我们得继续寻找其它方法，看看有无办法能达到题目所预期的O（N）的时间复杂度。

#### 1.2、中间交换

当然，除了如上面所述的让b1，b2，b3，b4步步前移跟它们各自前面的元素进行交换外，我们还可以每次让序列中最中间的元素进行交换达到目的。还是用上面的例子，针对a1，a2，a3，a4，b1，b2，b3，b4

第①步：交换最中间的两个元素a4，b1，序列变成（待交换的元素用粗体表示）：

**a1，a2，a3**，b1，a4，**b2，b3，b4**

第②步，让最中间的两对元素各自交换：

**a1，a2**，b1，a3，b2，a4，**b3，b4**

第③步，交换最中间的三对元素，序列变成：

a1，b1，a2，b2，a3，b3，a4，b4

同样，此法同解法1.1、步步前移一样，时间复杂度依然为O（N^2），我们得下点力气了。

### 解法二、完美洗牌算法

玩过扑克牌的朋友都知道，在一局完了之后洗牌，洗牌人会习惯性的把整副牌大致分为两半，两手各拿一半对着对着交叉洗牌，如下图所示：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/35/35.1.jpg)

如果这副牌用a1 a2 a3 a4 b1 b2 b3 b4表示（为简化问题，假设这副牌只有8张牌），然后一分为二之后，左手上的牌可能是a1 a2 a3 a4，右手上的牌是b1 b2 b3 b4，那么在如上图那样的洗牌之后，得到的牌就可能是b1 a1 b2 a2 b3 a3 b4 a4。

技术来源于生活，2004年，microsoft的Peiyush Jain在他发表一篇名为：“A Simple In-Place Algorithm for In-Shuffle”的论文中提出了完美洗牌算法。

这个算法解决一个什么问题呢？跟本题有什么联系呢？

Yeah，顾名思义，完美洗牌算法解决的就是一个完美洗牌问题。什么是完美洗牌问题呢？即给定一个数组a1,a2,a3,...an,b1,b2,b3..bn,最终把它置换成b1,a1,b2,a2,...bn,an。读者可以看到，这个完美洗牌问题本质上与本题完全一致，只要在完美洗牌问题的基础上对它最后的序列swap两两相邻元素即可。

即：

a1,a2,a3,...an,b1,b2,b3..bn

通过完美洗牌问题，得到：

b1,a1,b2,a2,b3,a3... bn,an

再让上面相邻的元素两两swap，即可达到本题的要求：

a1,b1,a2,b2,a3,b3....,an,bn

也就是说，如果我们能通过完美洗牌算法（时间复杂度O(N)，空间复杂度O(1)）解决了完美洗牌问题，也就间接解决了本题。

虽然网上已有不少文章对上篇论文或翻译或做解释说明，但对于初学者来说，理解难度实在太大，再者，若直接翻译原文，根本无法看出这个算法怎么一步步得来的，故下文将从完美洗牌算法的最基本的原型开始说起，以让读者能对此算法一目了然。

#### 2.1、位置置换pefect\_shuffle1算法

为方便讨论，我们设定数组的下标从1开始，下标范围是[1..2n]。 还是通过之前n=4的例子，来看下每个元素最终去了什么地方。

起始序列：a1 a2 a3 a4 b1 b2 b3 b4 数组下标：1 2 3 4 5 6 7 8 最终序列：b1 a1 b2 a2 b3 a3 b4 a4

从上面的例子我们能看到，前n个元素中，

第1个元素a1到了原第2个元素a2的位置，即1->2；

第2个元素a2到了原第4个元素a4的位置，即2->4；

第3个元素a3到了原第6个元素b2的位置，即3->6；

第4个元素a4到了原第8个元素b4的位置，即4->8；

那么推广到一般情况即是：前n个元素中，第i个元素去了 第（2 \* i）的位置。

上面是针对前n个元素，那么针对后n个元素，可以看出：

第5个元素b1到了原第1个元素a1的位置，即5->1；

第6个元素b2到了原第3个元素a3的位置，即6->3；

第7个元素b3到了原第5个元素b1的位置，即7->5；

第8个元素b4到了原第7个元素b3的位置，即8->7；

推广到一般情况是，后n个元素，第i个元素去了第 (2 \* (i - n) ) - 1 = 2 \* i - (2 \* n + 1) = (2 \* i) % (2 \* n + 1) 个位置。

再综合到任意情况，任意的第i个元素，我们最终换到了 (2 \* i) % (2 \* n + 1)的位置。为何呢？因为：

当0 < i < n时， 原式= (2i) % (2 \* n + 1) = 2i；

当i > n时，原式(2 \* i) % (2 \* n + 1)保持不变。

因此，如果题目允许我们再用一个数组的话，我们直接把每个元素放到该放得位置就好了。也就产生了最简单的方法pefect\_shuffle1，参考代码如下：

// 时间O(n)，空间O(n) 数组下标从1开始

void PefectShuffle1(int \*a, int n)

{

int n2 = n \* 2, i, b[N];

for (i = 1; i <= n2; ++i)

{

b[(i \* 2) % (n2 + 1)] = a[i];

}

for (i = 1; i <= n2; ++i)

{

a[i] = b[i];

}

}

但很明显，它的时间复杂度虽然是O(n)，但其空间复杂度却是O(n)，仍不符合本题所期待的时间O(n)，空间O(1)。我们继续寻找更优的解法。

与此同时，我也提醒下读者，根据上面变换的节奏，我们可以看出有两个圈，

一个是1 -> 2 -> 4 -> 8 -> 7 -> 5 -> 1；

一个是3 -> 6 -> 3。

下文2.2.1、走圈算法cycle\_leader将再次提到这两个圈。

#### 2.2、完美洗牌算法perfect\_shuffle2

##### 2.2.1、走圈算法cycle\_leader

因为之前perfect\_shuffle1算法未达到时间复杂度O（N）并且空间复杂度O（1）的要求，所以我们必须得再找一种新的方法，以期能完美的解决本节开头提出的完美洗牌问题。

让我们先来回顾一下2.1节位置置换perfect\_shuffle1算法，还记得我之前提醒读者的关于当n=4时，通过位置置换让每一个元素到了最后的位置时，所形成的两个圈么？我引用下2.1节的相关内容：

当n=4的情况：

起始序列：a1 a2 a3 a4 b1 b2 b3 b4 数组下标：1 2 3 4 5 6 7 8 最终序列：b1 a1 b2 a2 b3 a3 b4 a4

即通过置换，我们得到如下结论：

“于此同时，我也提醒下读者，根据上面变换的节奏，我们可以看出有两个圈，

一个是1 -> 2 -> 4 -> 8 -> 7 -> 5 -> 1；

一个是3 -> 6 -> 3。”

这两个圈可以表示为（1,2,4,8,7,5）和（3,6），且perfect\_shuffle1算法也已经告诉了我们，不管你n是奇数还是偶数，每个位置的元素都将变为第（2\*i） % （2n+1）个元素：

因此我们只要知道圈里最小位置编号的元素即圈的头部，顺着圈走一遍就可以达到目的，且因为圈与圈是不相交的，所以这样下来，我们刚好走了O（N）步。

还是举n=4的例子，且假定我们已经知道第一个圈和第二个圈的前提下，要让1 2 3 4 5 6 7 8变换成5 1 6 2 7 3 8 4：

第一个圈：1 -> 2 -> 4 -> 8 -> 7 -> 5 -> 1 第二个圈：3 -> 6 -> 3：

原始数组：1 2 3 4 5 6 7 8 数组下标：1 2 3 4 5 6 7 8

走第一圈：5 1 3 2 7 6 8 4 走第二圈：5 1 6 2 7 3 8 4

上面沿着圈走的算法我们给它取名为cycle\_leader，这部分代码如下：

//数组下标从1开始，from是圈的头部，mod是要取模的数 mod 应该为 2 \* n + 1，时间复杂度O(圈长）

void CycleLeader(int \*a, int from, int mod)

{

int t,i;

for (i = from \* 2 % mod; i != from; i = i \* 2 % mod)

{

t = a[i];

a[i] = a[from];

a[from] = t;

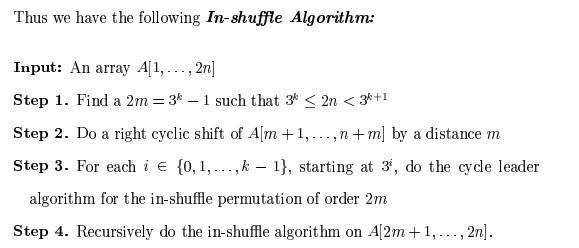
}

}

##### 2.2.2、神级结论：若2\*n=（3^k - 1），则可确定圈的个数及各自头部的起始位置

下面我要引用此论文“A Simple In-Place Algorithm for In-Shuffle”的一个结论了，即 对于2\*n = （3^k-1）这种长度的数组，恰好只有k个圈，且每个圈头部的起始位置分别是1,3,9，...3^(k-1)。

论文原文部分为：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/35/35.2.jpg)

也就是说，利用上述这个结论，我们可以解决这种特殊长度2\*n = （3^k-1）的数组问题，那么若给定的长度n是任意的咋办呢？此时，我们可以采取分而治之算法的思想，把整个数组一分为二，即拆分成两个部分：

让一部分的长度满足神级结论：若2\*m = （3^k-1），则恰好k个圈，且每个圈头部的起始位置分别是1,3,9，...3^(k-1)。其中m < n，m往神级结论所需的值上套；

剩下的n-m部分单独计算；

当把n分解成m和n-m两部分后，原始数组对应的下标如下（为了方便描述，我们依然只需要看数组下标就够了）：

原始数组下标：1..m m+1.. n， n+1 .. n+m, n+m+1,..2\*n

且为了能让前部分的序列满足神级结论2\*m = （3^k-1），我们可以把中间那两段长度为n-m和m的段交换位置，即相当于把m+1..n，n+1..n+m的段循环右移m次（为什么要这么做？因为如此操作后，数组的前部分的长度为2m，而根据神级结论：当2m=3^k-1时，可知这长度2m的部分恰好有k个圈）。

而如果读者看过本系列第一章、左旋转字符串的话，就应该意识到循环位移是有O（N）的算法的，其思想即是把前n-m个元素（m+1.. n）和后m个元素（n+1 .. n+m）先各自翻转一下，再将整个段（m+1.. n， n+1 .. n+m）翻转下。

这个翻转的代码如下：

//翻转字符串时间复杂度O(to - from)

void reverse(int \*a, int from, int to)

{

int t;

for (; from < to; ++from, --to)

{

t = a[from];

a[from] = a[to];

a[to] = t;

}

}

//循环右移num位 时间复杂度O(n)

void RightRotate(int \*a, int num, int n)

{

reverse(a, 1, n - num);

reverse(a, n - num + 1, n);

reverse(a, 1, n);

}

翻转后，得到的目标数组的下标为：

目标数组下标：1..m n+1..n+m m+1 .. n n+m+1,..2\*n

OK，理论讲清楚了，再举个例子便会更加一目了然。当给定n=7时，若要满足神级结论2\*n=3^k-1，k只能取2，继而推得n‘=m=4。

原始数组：a1 a2 a3 a4 a5 a6 a7 b1 b2 b3 b4 b5 b6 b7

既然m=4，即让上述数组中有下划线的两个部分交换，得到：

目标数组：a1 a2 a3 a4 b1 b2 b3 b4 a5 a6 a7 b5 b6 b7

继而目标数组中的前半部分a1 a2 a3 a4 b1 b2 b3 b4部分可以用2.2.1、走圈算法cycle\_leader搞定，于此我们最终求解的n长度变成了n’=3，即n的长度减小了4，单独再解决后半部分a5 a6 a7 b5 b6 b7即可。

##### 2.2.3、完美洗牌算法perfect\_shuffle3

从上文的分析过程中也就得出了我们的完美洗牌算法，其算法流程为：

输入数组　A[1..2 \* n]

step 1 找到 2 \* m = 3^k - 1 使得 3^k <= 2 \* n < 3^(k +1)

step 2 把a[m + 1..n + m]那部分循环移m位

step 3 对每个i = 0,1,2..k - 1，3^i是个圈的头部，做cycle\_leader算法，数组长度为m，所以对2 \* m + 1取模。

step 4 对数组的后面部分A[2 \* m + 1.. 2 \* n]继续使用本算法, 这相当于n减小了m。

上述算法流程对应的论文原文为：

以上各个步骤对应的时间复杂度分析如下：

因为循环不断乘3的，所以时间复杂度O(logn)

循环移位O(n)

每个圈，每个元素只走了一次，一共2\*m个元素，所以复杂度omega(m), 而m < n，所以 也在O(n)内。T(n - m)

因此总的时间复杂度为 T(n) = T(n - m) + O(n) ，m = omega(n) ，解得：T(n) = O(n)。

此完美洗牌算法实现的参考代码如下：

//copyright@caopengcs 8/24/2013

//时间O(n)，空间O(1)

void PerfectShuffle2(int \*a, int n)

{

int n2, m, i, k, t;

for (; n > 1;)

{

// step 1

n2 = n \* 2;

for (k = 0, m = 1; n2 / m >= 3; ++k, m \*= 3)

;

m /= 2;

// 2m = 3^k - 1 , 3^k <= 2n < 3^(k + 1)

// step 2

right\_rotate(a + m, m, n);

// step 3

for (i = 0, t = 1; i < k; ++i, t \*= 3)

{

cycle\_leader(a , t, m \* 2 + 1);

}

//step 4

a += m \* 2;

n -= m;

}

// n = 1

t = a[1];

a[1] = a[2];

a[2] = t;

}

##### 2.2.4、perfect\_shuffle2算法解决其变形问题

啊哈！以上代码即解决了完美洗牌问题，那么针对本章要解决的其变形问题呢？是的，如本章开头所说，在完美洗牌问题的基础上对它最后的序列swap两两相邻元素即可，代码如下：

//copyright@caopengcs 8/24/2013

//时间复杂度O(n)，空间复杂度O(1)，数组下标从1开始，调用perfect\_shuffle3

void shuffle(int \*a, int n)

{

int i, t, n2 = n \* 2;

PerfectShuffle2(a, n);

for (i = 2; i <= n2; i += 2)

{

t = a[i - 1];

a[i - 1] = a[i];

a[i] = t;

}

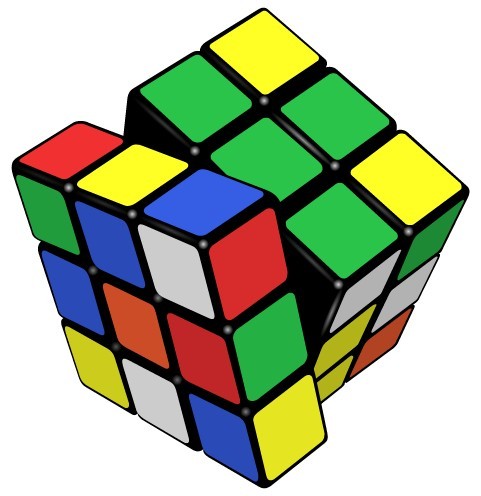
}

上述的这个“在完美洗牌问题的基础上对它最后的序列swap两两相邻元素”的操作（当然，你也可以让原数组第一个和最后一个不变，中间的2 \* (n - 1)项用原始的标准完美洗牌算法做），只是在完美洗牌问题时间复杂度O(N)空间复杂度O(1)的基础上再增加O(N)的时间复杂度，故总的时间复杂度O(N)不变，且理所当然的保持了空间复杂度O(1)。至此，咱们的问题得到了圆满解决！

## 问题扩展

### 神级结论是如何来的？

我们的问题得到了解决，但本章尚未完，即决定完美洗牌算法的神级结论：若2\*n=（3^k - 1），则恰好只有k个圈，且每个圈头部的起始位置分别是1,3,9，...3^(k-1)，是如何来的呢？

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/35/35.3.jpg)

要证明这个结论的关键就是：这所有的圈合并起来必须包含从1到M之间的所有正数，一个都不能少。这个证明有点麻烦，因为证明过程中会涉及到群论等数论知识，但再远的路一步步走也能到达。

首先，让咱们明确以下相关的概念，定理，及定义（搞清楚了这些东西，咱们便证明了一大半）：

概念1 mod表示对一个数取余数，比如3 mod 5 =3，5 mod 3 =2；

定义1 欧拉函数ϕ(m) 表示为不超过m（即小于等于m）的数中，与m互素的正整数个数

定义2 若ϕ(m)=Ordm(a) 则称a为m的原根，其中Ordm(a)定义为：a ^d （ mod m），其中d=0,1,2,3…，但取让等式成立的最小的那个d。

结合上述定义1、定义2可知，2是3的原根，因为2^0 mod 3 = 1, 2^1 mod 3 = 2, 2^2 mod 3 = 1, 2^3 mod 3 = 2，{a^0 mod m，a^1 mod m，a^2}得到集合S={1,2}，包含了所有和3互质的数，也即d=ϕ(2)=2，满足原根定义。

而2不是7的原根，这是因为2^0 mod 7 = 1, 2^1 mod 7 = 2, 2^2 mod 7 = 4, 2^3 mod 7 = 1，2^4 mod 7 = 2，2^5 mod 7 = 4，2^6 mod 7 = 1，从而集合S={1,2,4}中始终只有1、2、4三种结果，而没包含全部与7互质的数（3、6、5便不包括）,，即d=3，但ϕ(7)=6，从而d != ϕ(7)，不满足原根定义。

再者，如果说一个数a，是另外一个数m的原根，代表集合S = {a^0 mod m, a^1 mod m, a^2 mod m…… }，得到的集合包含了所有小于m并且与m互质的数，否则a便不是m的原根。而且集合S = {a^0 mod m, a^1 mod m, a^2 mod m…… }中可能会存在重复的余数，但当a与m互质的时候，得到的{a^0 mod m, a^1 mod m, a^2 mod m}集合中，保证了第一个数是a^0 mod m，故第一次发现重复的数时，这个重复的数一定是1，也就是说，出现余数循环一定是从开头开始循环的。

定义3 对模指数，a对模m的原根定义为 [https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/35/35.4.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/35/35.4.jpg),st:[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/35/35.5.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/35/35.5.jpg)中最小的正整数d

再比如，2是9的原根，因为[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/35/35.6.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/35/35.6.jpg)，为了让[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/35/35.7.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/35/35.7.jpg)除以9的余数恒等于1，可知最小的正整数d=6，而ϕ(m)=6，满足原根的定义。

定理1 同余定理：两个整数a，b，若它们除以正整数m所得的余数相等，则称a，b对于模m同余，记作[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/35/35.8.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/35/35.8.jpg)，读做a与b关于模m同余。

定理2 当p为奇素数且a是[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/35/35.9.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/35/35.9.jpg)的原根时⇒ a也是[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/35/35.10.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/35/35.10.jpg)的原根

定理3 费马小定理：如果a和m互质，那么a^ϕ(m) mod m = 1

定理4 若(a,m)=1 且a为m的原根，那么a是(Z/mZ)\*的生成元。

取a = 2, m = 3。

我们知道2是3的原根，2是9的原根，我们定义S(k)表示上述的集合S，并且取x = 3^k（x表示为集合S中的数）。

所以：

S(1) = {1, 2}

S(2) = {1, 2, 4, 8, 7, 5}

我们没改变圈元素的顺序，由前面的结论S(k)恰好是一个圈里的元素，且认为从1开始循环的，也就是说从1开始的圈包含了所有与3^k互质的数。

那与3^k不互质的数怎么办？如果0 < i < 3^k与 3^k不互质，那么i 与3^k的最大公约数一定是3^t的形式（只包含约数3)，并且 t < k。即gcd(i , 3^k) = 3^t，等式两边除以个3 ^ t，即得gcd( i/(3^t)，3^(k - t) ) = 1， i/(3^t) 都与3^(k - t) 互质了，并且i / (3^t) < 3^(k - t), 根据S(k)的定义，可见i/(3^t) 在集合S(k - t)中。

同理，任意S(k - t)中的数x，都满足gcd(x , 3^k) = 1,于是gcd(3^k , x\* 3^t) = 3 ^ t, 并且x3^t < 3^k。可见S(k - t)中的数x3^t 与 i形成了一一对应的关系。

也就是说S(k - t)里每个数x\* 3^t形成的新集合包含了所有与3^k的最大公约数为3^t的数，它也是一个圈,原先圈的头部是1，这个圈的头部是3^t。

于是，对所有的小于 3^k的数，根据它和3^k的最大公约数，我们都把它分配到了一个圈里去了，且k个圈包含了所有的小于3^k的数。

下面，举个例子，如caopengcs所说，当我们取“a = 2, m = 3时，

我们知道2是3的原根，2是9的原根，我们定义S(k)表示上述的集合S，并且x= 3^k。

所以S(1) = {1, 2}

S(2) = {1, 2, 4, 8, 7, 5}

比如k = 3。 我们有：

S(3) = {1, 2 ,4 , 8, 16, 5, 10, 20, 13, 26, 25, 23, 19, 11, 22, 17, 7, 14} 包含了小于27且与27互质的所有数，圈的首部为1，这是原根定义决定的。

那么与27最大公约数为3的数，我们用S(2)中的数乘以3得到。 S(2) \* 3 = {3, 6, 12, 24, 21, 15}, 圈中元素的顺序没变化，圈的首部是3。

与27最大公约数为9的数，我们用S(1)中的数乘以9得到。 S(1) \* 9 = {9, 18}, 圈中得元素的顺序没变化，圈的首部是9。

因为每个小于27的数和27的最大公约数只有1, 3, 9这3种情况，又由于前面所证的一一对应的关系，所以S(2) \* 3包含了所有小于27且与27的最大公约数为3的数，S(1) \* 9 包含了所有小于27且和27的最大公约数为9的数。”

换言之，若定义为整数，假设/N定义为整数Z除以N后全部余数的集合，包括{0...N-1}等N个数，而（/N)\*则定义为这Z/N中{0...N-1}这N个余数内与N互质的数集合。

则当n=13时，2n+1=27，即得/N =｛0,1,2,3,.....,26}，（/N)\*相当于就是｛0,1,2,3,.....,26}中全部与27互素的数的集合；

而2^k(mod 27)可以把（/27)\*取遍，故可得这些数分别在以下3个圈内：

取头为1，（/27)\*＝｛1,2,4,8,16,5,10,20,13,26,25,23,19,11,22,17,7,14｝，也就是说，与27互素且小于27的正整数集合为{1,2,4,8,16,5,10,20,13,26,25,23,19,11,22,17,7,14}，因此ϕ(m) = ϕ(27)=18, 从而满足[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/35/35.11.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/35/35.11.jpg)的最小d = 18，故得出2为27的原根；

取头为3，就可以得到｛3,6,12,24,21,15｝，这就是以3为头的环，这个圈的特点是所有的数都是3的倍数，且都不是9的倍数。为什么呢？因为2^k和27互素。

具体点则是：如果3×2^k除27的余数能够被9整除，则有一个n使得32^k=9n(mod 27)，即32^k－9n能够被27整除，从而3\*2^k－9n＝27m，其中n，m为整数，这样一来，式子约掉一个3，我们便能得到2^k＝9m＋3n，也就是说，2^k是3的倍数，这与2^k与27互素是矛盾的，所以，3×2^k除27的余数不可能被9整除。

此外，2^k除以27的余数可以是3的倍数以外的所有数，所以，2^k除以27的余数可以为1,2,4,5,7,8，当余数为1时，即存在一个k使得2^k-1=27m，m为整数。

式子两边同时乘以3得到：32^k-3=81m是27的倍数，从而32^k除以27的余数为3；

同理，当余数为2时，2^k - 2 = 27m，=> 32^k- 6 =81m，从而32^k除以27的余数为6；

当余数为4时，2^k - 4 = 37m，=> 32^k - 12 =81m，从而32^k除以27的余数为12；

同理，可以取到15，21，24。从而也就印证了上面的结论：取头为3，就可以得到｛3,6,12,24,21,15｝。 取9为头，这就很简单了，这个圈就是｛9,18}

你会发现，小于27的所有自然数，要么在第一个圈里面，也就是那些和27互素的数；要么在第二个圈里面，也就是那些是3的倍数，但不是9的倍数的数；要么在第三个圈里面，也就是是9倍数的数，而之所以能够这么做，就是因为2是27的本原根。证明完毕。

最后，咱们也再验证下上述过程：

因为[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/35/35.12.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/35/35.12.jpg)，故：

i = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27

由于n=13，2n+1 = 27，据此[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/35/35.13.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/35/35.13.jpg)公式可知，上面第 i 位置的数将分别变成下述位置的：

i = 2 4 6 8 10 12 14 16 18 20 22 24 26 1 3 5 7 9 11 13 15 17 19 21 23 25 0

根据i 和 i‘ 前后位置的变动，我们将得到3个圈：

1->2->4->8->16->5->10->20->13->26->25->23->19->11->22->17->7->14->1；

3->6->12->24->21->15->3

9->18->9

没错，这3个圈的数字与咱们之前得到的3个圈一致吻合，验证完毕。

## 举一反三

至此，本章开头提出的问题解决了，完美洗牌算法的证明也证完了，是否可以止步了呢？OH，NO！读者有无思考过下述问题：

1、既然完美洗牌问题是给定输入：a1,a2,a3,……aN,b1,b2,b3,……bN，要求输出：b1,a1,b2,a2,……bN,aN；那么有无考虑过它的逆问题：即给定b1,a1,b2,a2,……bN,aN,，要求输出a1,a2,a3,……aN,b1,b2,b3,……bN ？

2、完美洗牌问题是两手洗牌，假设有三只手同时洗牌呢？那么问题将变成：输入是a1,a2,……aN, b1,b2,……bN, c1,c2,……cN，要求输出是c1,b1,a1,c2,b2,a2,……cN,bN,aN，这个时候，怎么处理？

# 2.15本章数组和队列的习题

**1、不用除法运算**

两个数组a[N]，b[N]，其中A[N]的各个元素值已知，现给b[i]赋值，b[i] = a[0]*a[1]*a[2]...\*a[N-1]/a[i]； 要求：

* 1.不准用除法运算
* 2.除了循环计数值，a[N],b[N]外，不准再用其他任何变量（包括局部变量，全局变量等）
* 3.满足时间复杂度O(n)，空间复杂度O(1)。

提示：题目要求b[i] = a[0]*a[1]*a[2]...*a[N-1]/a[i] ，相当于求：a[0]*a[1]*a[2]*a[3]...a[i-1]*a[i+1]..*a[N-1]，等价于除掉当前元素a[i]，其他所有元素(a[i]左边部分，和a[i]右边部分)的积。

记left[i]=∏a[k], (k=1...i-1); right=∏a[k], (k=i+1...n)，根据题目描述b[i]=left[i] \* right[i]， 对于每一个b[i]初始化为1，left[i]和right[i]两部分可以分开两次相乘，即对于循环变量i=1...n, b[i]=left[i];b[n-i]=right[n-i]， 循环完成时即可完成计算。

参考代码如下所示：

void Multiplication(int a[], int output[], int length)

{

int left = 1;

int right = 1;

for (int i = 0; i < length; i++)

output[i] = 1;

for (int i = 0; i < length; i++)

{

output[i] \*= left;

output[length - i - 1] \*= right;

left \*= a[i];

right \*= a[length - i - 1];

}

}

**3、找出数组中唯一的重复元素**

1-1000放在含有1001个元素的数组中，只有唯一的一个元素值重复，其它均只出现一次。 每个数组元素只能访问一次，设计一个算法，将它找出来；不用辅助存储空间，能否设计一个算法实现？

**4、找出唯一出现的数**

一个数组里，数都是两两出现的，但是有三个数是唯一出现的，找出这三个数。

**5、找出反序的个数**

给定一整型数组，若数组中某个下标值大的元素值小于某个下标值比它小的元素值，称这是一个反序。 即：数组a[]; 对于i < j 且 a[i] > a[j],则称这是一个反序。 给定一个数组，要求写一个函数，计算出这个数组里所有反序的个数。

**6、**

有两个序列A和B,A=(a1,a2,...,ak),B=(b1,b2,...,bk),A和B都按升序排列，对于1<=i,j<=k，求k个最小的（ai+bj），要求算法尽量高效。

**8**

假设一个大小为100亿个数据的数组，该数组是从小到大排好序的，现在该数组分成若干段，每个段的数据长度小于20「也就是说：题目并没有说每段数据的size 相同，只是说每个段的 size < 20 而已」，然后将每段的数据进行乱序（即：段内数据乱序），形成一个新数组。请写一个算法，将所有数据从小到大进行排序，并说明时间复杂度。

**9**

20个排序好的数组，每个数组500个数，按照降序排序好的，让找出500个最大的数。

**10**

O(1)空间内实现矩阵转置。

**11**

有N个数，组成的字符串，如012345，求出字串和取MOD3==0的子串，如012 12 123 45。

**12**

从一列数中筛除尽可能少的数使得从左往右看，这些数是从小到大再从大到小的。

提示：双端 LIS 问题，用 DP 的思想可解。

**13**

有两个序列a,b，大小都为n,序列元素的值是任意整数，无序。要求：通过交换a,b中的元素，使[序列a元素的和]与[序列b元素的和]之间的差最小。

例如:

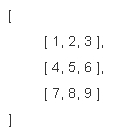
var a=[100,99,98,1,2, 3];

var b=[1, 2, 3, 4,5,40]。

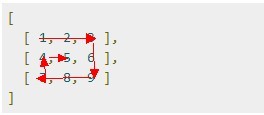
**14、螺旋矩阵**

Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order。一句话，即为螺旋矩阵问题。

举个例子，给定如下的一个矩阵:

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/40.1.jpg)

你应该返回：[1,2,3,6,9,8,7,4,5]。如下图所示，遍历顺序为螺旋状：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/40.2.jpg)

**15**

给你10分钟时间，根据上排给出十个数，在其下排填出对应的十个数 要求下排每个数都是先前上排那十个数在下排出现的次数。

上排的十个数如下：

0，1，2，3，4，5，6，7，8，9

举一个例子，

数值: 0,1,2,3,4,5,6,7,8,9

分配: 6,2,1,0,0,0,1,0,0,0

0在下排出现了6次，1在下排出现了2次，

2在下排出现了1次，3在下排出现了0次....

以此类推..

**16**

对于一个整数矩阵，存在一种运算，对矩阵中任意元素加一时，需要其相邻（上下左右），某一个元素也加一，现给出一正数矩阵，判断其是否能够由一个全零矩阵经过上述运算得到。

**17**

一个整数数组，长度为n，将其分为m份，使各份的和相等，求m的最大值。

比如{3，2，4，3，6} 可以分成

* {3，2，4，3，6} m=1;
* {3,6}{2,4,3} m=2
* {3,3}{2,4}{6} m=3

所以m的最大值为3。

**18**

求一个数组的最长递减子序列 比如{9，4，3，2，5，4，3，2}的最长递减子序列为{9，5，4，3，2}。

**19**

如何对n个大小都小于100的整数进行排序，要求时间复杂度O(n)，空间复杂度O(1)。

**20**

输入一个正数n，输出所有和为n连续正数序列。例如输入15，由于1+2+3+4+5=4+5+6=7+8=15，所以输出3个连续序列1-5、4-6和7-8。

**21、找出数组中两个只出现一次的数字**

一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。要求时间复杂度是O(n)，空间复杂度是O(1)。

**22、找出数组中两个只出现一次的数字**

题目：一个整型数组里除了两个数字之外，其他的数字都出现了两次。 请写程序找出这两个只出现一次的数字。要求时间复杂度是O(n)，空间复杂度是O(1)。

**23、把数组排成最小的数**

输入一个正整数数组，将它们连接起来排成一个数，输出能排出的所有数字中最小的一个。例如输入数组{32, 321}，则输出这两个能排成的最小数字32132。

**24、旋转数组中的最小元素**

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个排好序的数组的一个旋转，输出旋转数组的最小元素。例如数组{3, 4, 5, 1, 2}为{1, 2, 3, 4, 5}的一个旋转，该数组的最小值为1。

提示：从头到尾遍历数组一次，就能找出最小的元素，时间复杂度显然是O(N)。但这个思路没有利用输入数组的特性，请读者继续思考更好的解法。

**25**

N个鸡蛋放到M个篮子中，篮子不能为空，要满足：对任意不大于N的数量，能用若干个篮子中鸡蛋的和表示。

写出函数，对输入整数N和M，输出所有可能的鸡蛋的放法。

比如对于9个鸡蛋5个篮子 解至少有三组：1 2 4 1 1 1 2 2 2 2 1 2 3 2 1

**26**

请把一个整形数组中重复的数字去掉。例如：

1, 2, 0, 2, -1, 999, 3, 999, 88

答案应该是：

1, 2， 0， -1， 999， 3， 88

**27**

有一台机器，上面有m个储存空间。然后有n个请求，第i个请求计算时需要占 R[i]个空间，储存计算结果则需要占据O[i]个空间（据O[i]个空间（其中O[i]<R[i]）。问怎么安排这n个请求的顺序，使得所有请求都能完成。你的算法也应该能够判断出无论如何都不能处理完的情况。

比方说，m=14，n=2，R[1]=10，O[1]=5，R[2]=8，O[2]=6。在这个例子中，我们可以先运行第一个任务，剩余9个单位的空间足够执行第二个任务；但如果先走第二个任务，第一个任务执行时空间就不够了，因为10>14-6。

**28**

在一维坐标轴上有n个区间段，求重合区间最长的两个区间段。

**29**

如果用一个循环数组q[0..m-1]表示队列时,该队列只有一个队列头指针front,不设队列尾指针rear，求这个队列中从队列投到队列尾的元素个数（包含队列头、队列尾）。

**30**

给定一个实数数组，按序排列（从小到大）,从数组从找出若干个数，使得这若干个数的和与M最为接近，描述一个算法，并给出算法的复杂度。

有N个正实数(注意是实数，大小升序排列) x1 , x2 ... xN，另有一个实数M。 需要选出若干个x，使这几个x的和与 M 最接近。 请描述实现算法，并指出算法复杂度。

**31**

有无序的实数列V[N]，要求求里面大小相邻的实数的差的最大值，关键是要求线性空间和线性时间。

**32**

一个数组保存了N个结构，每个结构保存了一个坐标，结构间的坐标都不相同，请问如何找到指定坐标的结构（除了遍历整个数组，是否有更好的办法）？

提示：要么预先排序，二分查找。要么哈希。hash的话，坐标(x,y)你可以当做一个2位数，写一个哈希函数，把（x,y）直接转成“(x,y)”作为key，默认用string比较。或如Edward Lee所说，将坐标(x, y)作为 Hash 中的 key。例如(m, n)，通过 (m,n) 和 (n, m) 两次查找看是否在 HashMap 中。也可以在保存时就规定 (x, y) , x < y ，在插入之前做个判断。

**33**

现在有1千万个随机数，随机数的范围在1到1亿之间。现在要求写出一种算法，将1到1亿之间没有在随机数中的数求出来。

提示：编程珠玑上有此类似的一题，如果有足够的内存的话可以用位图法，即开一个1亿位的bitset，内存为100m/8== 12.5m, 然后如果一个数有出现，对应的bitset上标记为1，最后统计bitset上为0的即可。

**34**

有Ｎ＋２个数，N个数出现了偶数次，２个数出现了奇数次（这两个数不相等），问用O（1）的空间复杂度，找出这两个数，不需要知道具体位置，只需要知道这两个值。

提示：xor一次，得到2个奇数次的数之和x。第二步，以x（展开成二进制）中有1的某位（假设第i位为1）作为划分，第二次只xor第i位为1的那些数，得到y。然后x xor y以及y便是那两个数。

**35**

一个整数数组，有n个整数，如何找其中m个数的和等于另外n-m个数的和？

**36**

一个数组，里面的数据两两相同，只有两个数据不同，要求找出这两个数据。要求时间复杂度0（N）空间复杂度O（1）。

**37**

一个环形公路，上面有Ｎ个站点，A1, ..., AN，其中Ai和Ai+1之间的距离为Di,AN和A1之间的距离为D0。 高效的求第i和第j个站点之间的距离，空间复杂度不超过O(N)。

**38**

将一个较大的钱，不超过1000000(10^6)的人民币，兑换成数量不限的100、50、10、5、2、1的组合，请问共有多少种组合呢？

**39**

对于一个数组{1,2,3}它的子数组有{1,2}，{1,3}{2,3}，{1,2,3}，元素之间可以不是连续的，对于数组{5,9,1,7,2,6,3,8,10,4}，升序子序列有多少个？

或者换一种表达为：数组int a[]={5,9,1,7,2,6,3,8,10,4} 。求其所有递增子数组(元素相对位置不变)的个数， 例如：{5，9}，{5，7，8，10}，{1，2，6，8}。

**40**

M\*M的方格矩阵，其中有一部分为障碍，八个方向均可以走，现假设矩阵上有Q+1节点，从(X0，Y0)出发到其他Q个节点的最短路径。 其中，1<=M<=1000，1<=Q<=100。

**41**

设子数组A[0:k]和A[k+1:N-1]已排好序(0≤K≤N-1)。试设计一个合并这2个子数组为排好序的数组A[0:N-1]的算法。要求算法在最坏情况下所用的计算时间为O(N)，只用到O(1)的辅助空间。

提示：此题来源于在高德纳的计算机程序设计艺术第三卷第五章排序。

**42**

一个数组[1,2,3,4,6,8,9,4,8,11,18,19,100]前半部分是是一个递增数组，后面一个还是递增数组，但整个数组不是递增数组，那么怎么最快的找出其中一个数？

**43**

数组中的数分为两组，让给出一个算法，使得两个组的和的差的绝对值最小，数组中的数的取值范围是0<x<100，元素个数也是大于0， 小于100 。

比如a[]={2,4,5,6,7},得出的两组数｛2，4，6｝和｛5，7｝，abs(sum(a1)-sum(a2))=0；

比如｛2，5，6，10｝，abs（sum(2,10)-sum(5,6))=1,所以得出的两组数分别为｛2，10｝和｛5，6｝。

**44**

从1....n中随机输出m个不重复的数

**45**

数组al[0,mid-1] 和 al[mid,num-1]，都分别有序。将其merge成有序数组al[0,num-1]，要求空间复杂度O(1)。

**46、求旋转数组的最小元素**

把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个排好序的数组的一个旋转，输出旋转数组的最小元素。例如数组{3, 4, 5, 1, 2}为{1, 2, 3, 4, 5}的一个旋转，该数组的最小值为1。

**47**

在一个平面坐标系上，有两个矩形，它们的边分别平行于X和Y轴。 其中，矩形A已知， ax1(左边), ax2（右边）, ay1（top的纵坐标）, ay2（bottom纵坐标）. 矩形B，类似，就是 bx1, bx2, by1, by2。这些值都是整数就OK了。 要求是，如果矩形没有交集，返回-1， 有交集，返回交集的面积。int area(rect const& a, rect const& b) { ... }

**48**

一个数组里，数都是两两出现的，但是有三个数是唯一出现的，找出这三个数。

提示：3个数唯一出现，各不相同。由于x与a、b、c都各不相同，因此x^a、x^b、x^c都不等于0。所以无法简单的用异或解决此问题。

**49、计算逆序数组对**

给定一整型数组，若数组中某个下标值大的元素值小于某个下标值比它小的元素值，称这是一个反序。 即：数组a[]; 对于i < j 且 a[i] > a[j],则称这是一个反序。 给定一个数组，要求写一个函数，计算出这个数组里所有反序的个数。

**50**

有两个序列A和B,A=(a1,a2,...,ak),B=(b1,b2,...,bk),A和B都按升序排列，对于1<=i,j<=k，求k个最小的（ai+bj），要求算法尽量高效。

**51**

有20个数组，每个数组里面有500个数组，降序排列，每个数字是32位的unit,求出这10000个数字中最大的500个。

**52**

100个任务，100个工人每人可做一项任务，每个任务每个人做的的费用为t[100][100],求一个分配任务的方案使得总费用最少。

提示：匈牙利算法。

**53**

寻找3个数的中位数

提示：可以采用两两比较的思路。

**54**

给定一数组，输出满足2a=b（a，b代表数组中的数）的数对，要求时间复杂度尽量低。

**55**

1万个元素的数组，90%的元素都是1到100的数，10%的元素是101--10000的数，如何高效排序。

**56**

一个有序数组（从小到大排列），数组中的数据有正有负，求这个数组中的最小绝对值。

**57**

等价于n\*n的矩阵，填写0，1，要求每行每列的都有偶数个1 （没有1也是偶数个），问有多少种方法。

**58**

数组里找到和最接近于0的两个值。

**59**

N个数组，每个数组中的元素都是递增的顺序，现在要找出这N个数组中的公共元素部分，如何做? 注：不能用额外辅助空间。

**60**

二重歌德巴赫猜想

所有大于等于6的偶数都可以表示成两个（奇）素数之和。

给定1-10000，找到可以用两个素数之和表示每一个偶数的两个素数，然后输出这两个素数，如果有多对，则只需要输出其中之一对即可。

**61**

N个整数（数的大小为0-255）的序列，把它们加密为K个整数（数的大小为0-255）.再将K个整数顺序随机打乱，使得可以从这乱序的K个整数中解码出原序列。设计加密解密算法,且要求K<=15\*N.

如果是：

* N<=16,要求K<=16\*N.
* N<=16,要求K<=10\*N.
* N<=64,要求K<=15\*N.

**62**

两个无序数组分别叫A和B，长度分别是m和n，求中位数，要求时间复杂度O(m+n)，空间复杂度O(1) 。

**63**

假设一个大小为100亿个数据的数组，该数组是从小到大排好序的，现在该数组分成若干段，每个段的数据长度小于20「也就是说：题目并没有说每段数据的size 相同，只是说每个段的 size < 20 而已」，然后将每段的数据进行乱序（即：段内数据乱序），形成一个新数组。

请写一个算法，将所有数据从小到大进行排序，并说明时间复杂度。

**64**

20个排序好的数组，每个数组500个数，按照降序排序好的，让找出500个最大的数。

**65**

请自己用双向链表实现一个队列，队列里节点内存的值为int，要求实现入队，出队和查找指定节点的三个功能。

**66**

n个数字（0,1,…,n-1）形成一个圆圈，从数字0开始，每次从这个圆圈中删除第m个数字（第一个为当前数字本身，第二个为当前数字的下一个数字）。

当一个数字删除后，从被删除数字的下一个继续删除第m个数字。求出在这个圆圈中剩下的最后一个数字。

**67、在从1到n的正数中1出现的次数**

输入一个整数n，求从1到n这n个整数的十进制表示中1出现的次数。

例如输入12，从1到12这些整数中包含1 的数字有1，10，11和12，1一共出现了5次。

**68**

对于给定的整数集合S，求出最大的d，使得a+b+c=d。a,b,c,d互不相同，且都属于S。集合的元素个数小于等于2000个，元素的取值范围在[-2^28，2^28 - 1]，假定可用内存空间为100MB，硬盘使用空间无限大，试分析时间和空间复杂度，找出最快的解决方法。

提示：两两相加转为多项式乘法，比如(1 2 4 6) + (2 3 4 5) => (x + x^2 + x^4 + x^6)\*(x^2 + x^3 + x^4 + x^5) 。

**69**

长度为N的数组乱序存放着0带N-1.现在只能进行0与其他数的swap操作，请设计并实现排序，必须通过交换实现排序。

**70**

输入是两个整数数组，他们任意两个数的和又可以组成一个数组，求这个和中前k个数怎么做？

分析：假设两个整数数组为A和B，各有N个元素，任意两个数的和组成的数组C有N^2个元素。那么可以把这些和看成N个有序数列：

A[1]+B[1] <= A[1]+B[2] <= A[1]+B[3] <=…

A[2]+B[1] <= A[2]+B[2] <= A[2]+B[3] <=…

…

A[N]+B[1] <= A[N]+B[2] <= A[N]+B[3] <=…

问题转变成，在这N个有序数列里，找到前k小的元素”。

**71、求500万以内的所有亲和数**

如果两个数a和b，a的所有真因数之和等于b,b的所有真因数之和等于a,则称a,b是一对亲和数。 例如220和284，1184和1210，2620和2924。

**72、杨辉三角的变形**

1

1 1 1

1 2 3 2 1

1 3 6 7 6 3 1

以上三角形的数阵，第一行只有一个数1，以下每行的每个数，是恰好是它上面的数，左上的数和右上数等3个数之和（如果不存在某个数，认为该数就是0）。

求第n行第一个偶数出现的位置。如果没有偶数，则输出-1。例如输入3,则输出2，输入4则输出3。

**73、三元组的数量**

{5 3 1}和{7 5 3}是2组不同的等差三元组，除了等差的性质之外，还有个奇妙的地方在于：5^2 – 3^2 – 1^2 = 7^2 – 5^2 – 3^2 = N = 15。

{19 15 11}同{7 5 3}这对三元组也存在同样的性质：19^2 – 15^2 – 11^2 = 7^2 – 5^2 – 3^2 = N = 15。 这种成对的三元组还有很多。当N = 15时，有3对，分别是{5 3 1}和{7 5 3}，{5 3 1}和{19 15 11}，{7 5 3}和{19 15 11}。

现给出一个区间 [a,b]求a <= N <= b 范围内，共有多少对这样的三元组。（1 <= a <= b <= 5\*10^6）

例如：a = 1，b = 30，输出：4。（注：共有4对，{5 3 1}和{7 5 3}，{5 3 1}和{19 15 11}，{7 5 3}和{19 15 11}，{34 27 20}和{12 9 6}。

**74、格子涂色**

有一行方格，共有n个，编号为1-n,现在要用两种颜色（例如蓝色和黄色）给每个方格涂色，每个方格只能涂两种颜色之一，不能不涂。要求最终至少有m个连续的格子被涂成蓝色，问一共有多少种着色方法。例如n = 4, m = 3，有3种涂色的方法，分别为

* 蓝蓝蓝黄
* 蓝蓝蓝蓝
* 黄蓝蓝蓝

**75、寻找直方图中面积最大的矩形**

给定直方图，每一小块的height由N个非负整数所确定，每一小块的width都为1，请找出直方图中面积最大的矩形。

如下图所示，直方图中每一块的宽度都是1，每一块给定的高度分别是[2,1,5,6,2,3]：

那么上述直方图中，面积最大的矩形便是下图所示的阴影部分的面积，面积= 10单位。

# 3.0本章导读

想要更好地理解红黑树，可以先理解二叉查找树和2-3树。为何呢？首先，二叉查找树中的结点是2-结点（一个键两条链），引入3-结点（两个键三条链），即成2-3树；然后将2-3树中3-结点分解，即成红黑树，故结合二叉查找树易查找和2-3树易插入的特点，便成了红黑二叉查找树，简称红黑树。

进一步而言，理解了2-3树，也就理解了B树、B+树、B\*树，因为2-3树就是一棵3阶的B树，而一颗3阶的B树各个结点关键字数满足1-2，故当结点关键字数多于2时则达到饱和，此时需要分裂结点，而结点关键字数少于1时则从兄弟结点“借”关键字补充。

但为何有了红黑树，还要发明B树呢？原因是，当计算机要处理的数据量一大，便无法一次性装入内存进行处理，于此，计算机会把大部分备用的数据存在磁盘中，有需要的时候，就从磁盘中调取数据到在内存中处理，如果处理时修改了数据，则再次将数据写入磁盘，如此导致了不断的磁盘IO读写，而树的高度越高，查找文件所需要的磁盘IO读写次数越多，所以为了减少磁盘的IO读写，要想办法进一步降低树的高度。 因此，具有多个孩子的B树便应运而生，因为B树每一个结点可以有几个到几千个孩子，使得在结点数目一定的情况下，树的高度会大大降低，从而有效减少磁盘IO读写消耗。

此外，无论是B树，还是B+树、B树，由于根或者树的上面几层被反复查询，所以树上层几块的数据可以存在内存中。换言之，B树、B+树、B树的根结点和部分顶层数据存在内存中，大部分下层数据存在磁盘上。

# 3.1教你透彻了解红黑树

## 二叉查找树

由于红黑树本质上就是一棵二叉查找树，所以在了解红黑树之前，咱们先来看下二叉查找树。

二叉查找树（Binary Search Tree），也称有序二叉树（ordered binary tree）,排序二叉树（sorted binary tree），是指一棵空树或者具有下列性质的二叉树：

* 若任意结点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
* 若任意结点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
* 任意结点的左、右子树也分别为二叉查找树。
* 没有键值相等的结点（no duplicate nodes）。

因为，一棵由n个结点，随机构造的二叉查找树的高度为lgn，所以顺理成章，一般操作的执行时间为O（lgn）.（至于n个结点的二叉树高度为lgn的证明，可参考算法导论 第12章 二叉查找树 第12.4节）。

但二叉树若退化成了一棵具有n个结点的线性链后，则此些操作最坏情况运行时间为O（n）。后面我们会看到一种基于二叉查找树-红黑树，它通过一些性质使得树相对平衡，使得最终查找、插入、删除的时间复杂度最坏情况下依然为O（lgn）。

## 红黑树

前面我们已经说过，红黑树，本质上来说就是一棵二叉查找树，但它在二叉查找树的基础上增加了着色和相关的性质使得红黑树相对平衡，从而保证了红黑树的查找、插入、删除的时间复杂度最坏为O(log n)。

但它是如何保证一棵n个结点的红黑树的高度始终保持在h = logn的呢？这就引出了红黑树的5条性质：

1）每个结点要么是红的，要么是黑的。

2）根结点是黑的。

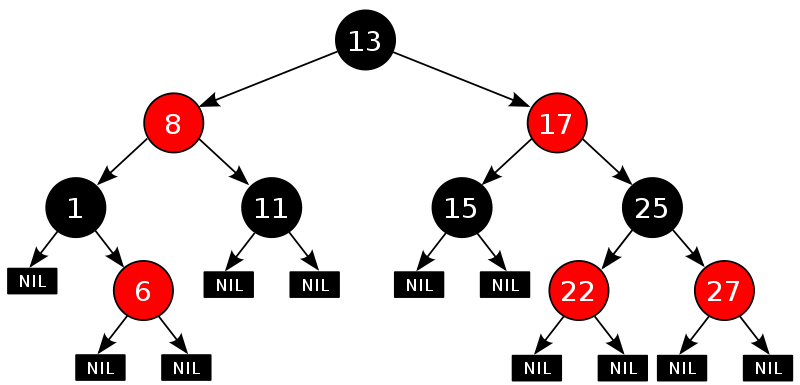
3）每个叶结点（叶结点即指树尾端NIL指针或NULL结点）是黑的。

4）如果一个结点是红的，那么它的俩个儿子都是黑的。

5）对于任一结点而言，其到叶结点树尾端NIL指针的每一条路径都包含相同数目的黑结点。

正是红黑树的这5条性质，使得一棵n个结点是红黑树始终保持了logn的高度，从而也就解释了上面我们所说的“红黑树的查找、插入、删除的时间复杂度最坏为O(log n)”这一结论的原因。

如下图所示，即是一颗红黑树(下图引自wikipedia：<http://t.cn/hgvH1l>)：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/1.png)

上文中我们所说的 "叶结点" 或"NULL结点"，它不包含数据而只充当树在此结束的指示，这些结点以及它们的父结点，在绘图中都会经常被省略。

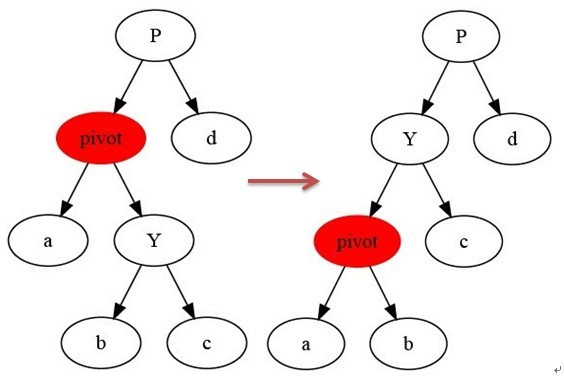
### 树的旋转知识

当我们在对红黑树进行插入和删除等操作时，对树做了修改，那么可能会违背红黑树的性质。

为了继续保持红黑树的性质，我们可以通过对结点进行重新着色，以及对树进行相关的旋转操作，即修改树中某些结点的颜色及指针结构，来达到对红黑树进行插入或删除结点等操作后，继续保持它的性质或平衡。

树的旋转，分为左旋和右旋，以下借助图来做形象的解释和介绍：

1.左旋

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/2.jpg)

如上图所示：

当在某个结点pivot上，做左旋操作时，我们假设它的右孩子y不是NIL[T]，pivot可以为任何不是NIL[T]的左孩子结点。

左旋以pivot到y之间的链为“支轴”进行，它使y成为该孩子树新的根，而y的左孩子b则成为pivot的右孩子。

左旋操作的参考代码如下所示（以x代替上述的pivot）：

LEFT-ROTATE(T, x)

1 y ← right[x] ▹ Set y.

2 right[x] ← left[y] ▹ Turn y's left subtree into x's right subtree.

3 p[left[y]] ← x

4 p[y] ← p[x] ▹ Link x's parent to y.

5 if p[x] = nil[T]

6 then root[T] ← y

7 else if x = left[p[x]]

8 then left[p[x]] ← y

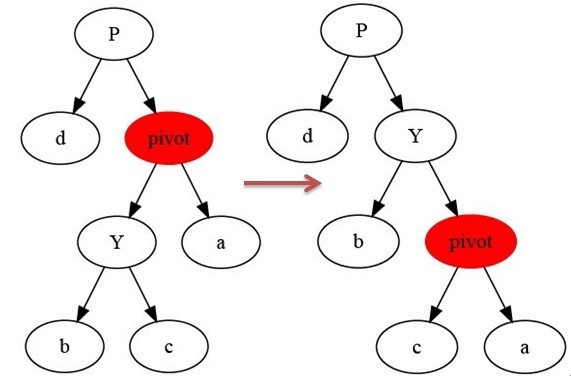
9 else right[p[x]] ← y

10 left[y] ← x ▹ Put x on y's left.

11 p[x] ← y

2.右旋

右旋与左旋差不多，再此不做详细介绍。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/3.jpg)

对于树的旋转，能保持不变的只有原树的搜索性质，而原树的红黑性质则不能保持，在红黑树的数据插入和删除后可利用旋转和颜色重涂来恢复树的红黑性质。

### 红黑树的插入

要真正理解红黑树的插入和删除，还得先理解二叉查找树的插入和删除。磨刀不误砍柴工，咱们再来分别了解下二叉查找树的插入和删除。

#### 二叉查找树的插入

如果要在二叉查找树中插入一个结点，首先要查找到结点插入的位置，然后进行插入，假设插入的结点为z的话，插入的伪代码如下：

TREE-INSERT(T, z)

1 y ← NIL

2 x ← root[T]

3 while x ≠ NIL

4 do y ← x

5 if key[z] < key[x]

6 then x ← left[x]

7 else x ← right[x]

8 p[z] ← y

9 if y = NIL

10 then root[T] ← z ⊹ Tree T was empty

11 else if key[z] < key[y]

12 then left[y] ← z

13 else right[y] ← z

可以看到，上述第3-7行代码即是在二叉查找树中查找z待插入的位置，如果插入结点z小于当前遍历到的结点，则到当前结点的左子树中继续查找，如果z大于当前结点，则到当前结点的右子树中继续查找，第9-13行代码找到待插入的位置，如果z依然比此刻遍历到的新的当前结点小，则z作为当前结点的左孩子，否则作为当前结点的右孩子。

#### 红黑树的插入和插入修复

现在我们了解了二叉查找树的插入，接下来，咱们便来具体了解红黑树的插入操作。红黑树的插入相当于在二叉查找树插入的基础上，为了重新恢复平衡，继续做了插入修复操作。

假设插入的结点为z，红黑树的插入伪代码具体如下所示：

RB-INSERT(T, z)

1 y ← nil[T]

2 x ← root[T]

3 while x ≠ nil[T]

4 do y ← x

5 if key[z] < key[x]

6 then x ← left[x]

7 else x ← right[x]

8 p[z] ← y

9 if y = nil[T]

10 then root[T] ← z

11 else if key[z] < key[y]

12 then left[y] ← z

13 else right[y] ← z

14 left[z] ← nil[T]

15 right[z] ← nil[T]

16 color[z] ← RED

17 RB-INSERT-FIXUP(T, z)

我们把上面这段红黑树的插入代码，跟我们之前看到的二叉查找树的插入代码，可以看出，RB-INSERT(T, z)前面的第1-13行代码基本就是二叉查找树的插入代码，然后第14-16行代码把z的左孩子、右孩子都赋为叶结点nil，再把z结点着为红色，最后为保证红黑性质在插入操作后依然保持，调用一个辅助程序RB-INSERT-FIXUP来对结点进行重新着色，并旋转。

换言之

* 如果插入的是根结点，因为原树是空树，此情况只会违反性质2，所以直接把此结点涂为黑色。
* 如果插入的结点的父结点是黑色，由于此不会违反性质2和性质4，红黑树没有被破坏，所以此时也是什么也不做。

但当遇到下述3种情况时：

* 插入修复情况1：如果当前结点的父结点是红色且祖父结点的另一个子结点（叔叔结点）是红色
* 插入修复情况2：当前结点的父结点是红色,叔叔结点是黑色，当前结点是其父结点的右子
* 插入修复情况3：当前结点的父结点是红色,叔叔结点是黑色，当前结点是其父结点的左子

又该如何调整呢？答案就是根据红黑树插入代码RB-INSERT(T, z)最后一行调用的RB-INSERT-FIXUP（T,z）所示操作进行，具体如下所示：

RB-INSERT-FIXUP（T,z）

1 while color[p[z]] = RED

2 do if p[z] = left[p[p[z]]]

3 then y ← right[p[p[z]]]

4 if color[y] = RED

5 then color[p[z]] ← BLACK ▹ Case 1

6 color[y] ← BLACK ▹ Case 1

7 color[p[p[z]]] ← RED ▹ Case 1

8 z ← p[p[z]] ▹ Case 1

9 else if z = right[p[z]]

10 then z ← p[z] ▹ Case 2

11 LEFT-ROTATE(T, z) ▹ Case 2

12 color[p[z]] ← BLACK ▹ Case 3

13 color[p[p[z]]] ← RED ▹ Case 3

14 RIGHT-ROTATE(T, p[p[z]]) ▹ Case 3

15 else (same as then clause

with "right" and "left" exchanged)

16 color[root[T]] ← BLACK

下面，咱们来分别处理上述3种插入修复情况。

**插入修复情况1：当前结点的父结点是红色且祖父结点的另一个子结点（叔叔结点）是红色。**

即如下代码所示：

1 while color[p[z]] = RED

2 do if p[z] = left[p[p[z]]]

3 then y ← right[p[p[z]]]

4 if color[y] = RED

此时父结点的父结点一定存在，否则插入前就已不是红黑树。  
与此同时，又分为父结点是祖父结点的左子还是右子，对于对称性，我们只要解开一个方向就可以了。

在此，我们只考虑父结点为祖父左子的情况。  
同时，还可以分为当前结点是其父结点的左子还是右子，但是处理方式是一样的。我们将此归为同一类。

对策：将当前结点的父结点和叔叔结点涂黑，祖父结点涂红，把当前结点指向祖父结点，从新的当前结点重新开始算法。即如下代码所示：

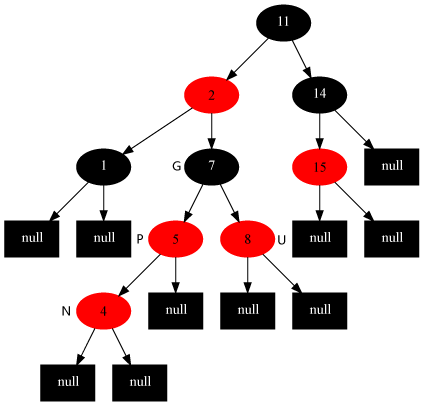
5 then color[p[z]] ← BLACK ▹ Case 1

6 color[y] ← BLACK ▹ Case 1

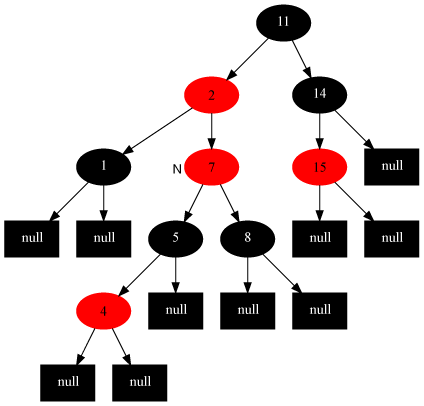
7 color[p[p[z]]] ← RED ▹ Case 1

8 z ← p[p[z]] ▹ Case 1

针对情况1，变化前（图片来源：saturnman）[当前结点为4结点]：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/4.png)

变化后：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/5.png)

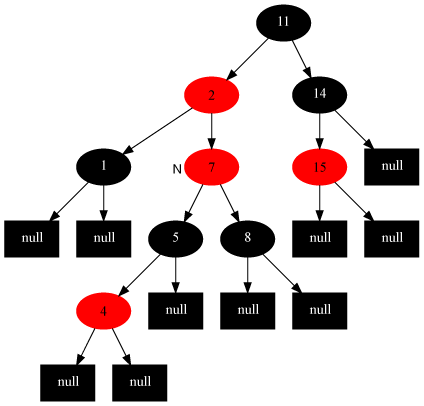
**插入修复情况2：当前结点的父结点是红色,叔叔结点是黑色，当前结点是其父结点的右子**  
对策：当前结点的父结点做为新的当前结点，以新当前结点为支点左旋。即如下代码所示：

9 else if z = right[p[z]]

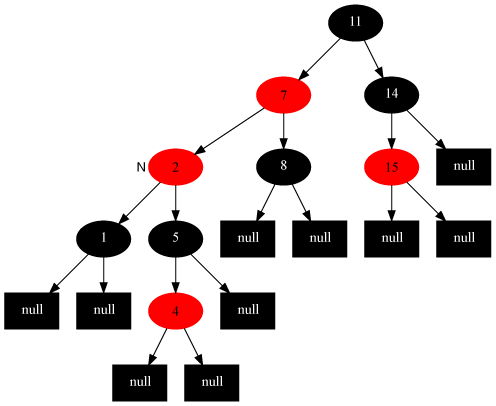
10 then z ← p[z] ▹ Case 2

11 LEFT-ROTATE(T, z) ▹ Case 2

如下图所示，变化前[当前结点为7结点]：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/6.png)

变化后：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/7.png)

**插入修复情况3：当前结点的父结点是红色,叔叔结点是黑色，当前结点是其父结点的左子**  
解法：父结点变为黑色，祖父结点变为红色，在祖父结点为支点右旋，操作代码为：

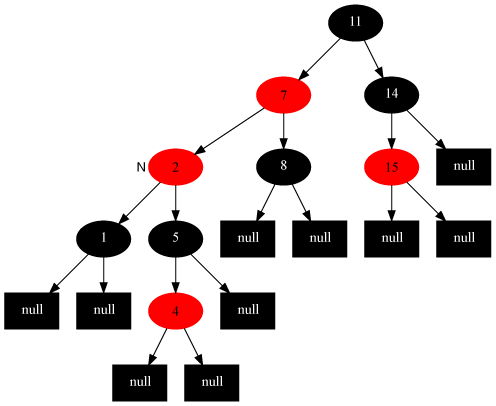
12 color[p[z]] ← BLACK ▹ Case 3

13 color[p[p[z]]] ← RED ▹ Case 3

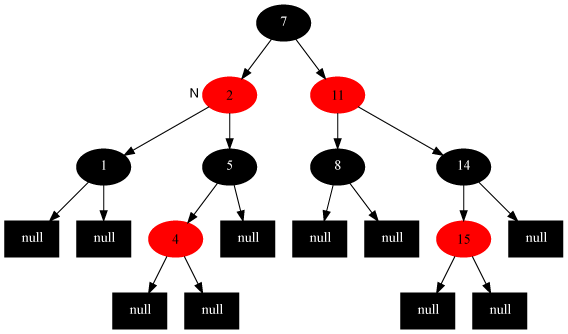
14 RIGHT-ROTATE(T, p[p[z]]) ▹ Case 3

最后，把根结点涂为黑色，整棵红黑树便重新恢复了平衡。

如下图所示[当前结点为2结点]

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/8.png)

变化后：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/9.png)

### 红黑树的删除

ok，接下来，咱们最后来了解，红黑树的删除操作。

"我们删除的结点的方法与常规二叉搜索树中删除结点的方法是一样的，如果被删除的结点不是有双非空子女，则直接删除这个结点，用它的唯一子结点顶替它的位置，如果它的子结点分是空结点，那就用空结点顶替它的位置，如果它的双子全为非空，我们就把它的直接后继结点内容复制到它的位置，之后以同样的方式删除它的后继结点，它的后继结点不可能是双子非空，因此此传递过程最多只进行一次。”

#### 二叉查找树的删除

继续讲解之前，补充说明下二叉树结点删除的几种情况，待删除的结点按照儿子的个数可以分为三种：

1. 没有儿子，即为叶结点。直接把父结点的对应儿子指针设为NULL，删除儿子结点就OK了。
2. 只有一个儿子。那么把父结点的相应儿子指针指向儿子的独生子，删除儿子结点也OK了。
3. 有两个儿子。这是最麻烦的情况，因为你删除结点之后，还要保证满足搜索二叉树的结构。其实也比较容易，我们可以选择左儿子中的最大元素或者右儿子中的最小元素放到待删除结点的位置，就可以保证结构的不变。当然，你要记得调整子树，毕竟又出现了结点删除。习惯上大家选择左儿子中的最大元素，其实选择右儿子的最小元素也一样，没有任何差别，只是人们习惯从左向右。这里咱们也选择左儿子的最大元素，将它放到待删结点的位置。左儿子的最大元素其实很好找，只要顺着左儿子不断的去搜索右子树就可以了，直到找到一个没有右子树的结点。那就是最大的了。

二叉查找树的删除代码如下所示：

TREE-DELETE(T, z)

1 if left[z] = NIL or right[z] = NIL

2 then y ← z

3 else y ← TREE-SUCCESSOR(z)

4 if left[y] ≠ NIL

5 then x ← left[y]

6 else x ← right[y]

7 if x ≠ NIL

8 then p[x] ← p[y]

9 if p[y] = NIL

10 then root[T] ← x

11 else if y = left[p[y]]

12 then left[p[y]] ← x

13 else right[p[y]] ← x

14 if y ≠ z

15 then key[z] ← key[y]

16 copy y's satellite data into z

17 return y

#### 红黑树的删除和删除修复

OK，回到红黑树上来，红黑树结点删除的算法实现是：

RB-DELETE(T, z) 单纯删除结点的总操作

1 if left[z] = nil[T] or right[z] = nil[T]

2 then y ← z

3 else y ← TREE-SUCCESSOR(z)

4 if left[y] ≠ nil[T]

5 then x ← left[y]

6 else x ← right[y]

7 p[x] ← p[y]

8 if p[y] = nil[T]

9 then root[T] ← x

10 else if y = left[p[y]]

11 then left[p[y]] ← x

12 else right[p[y]] ← x

13 if y ≠ z

14 then key[z] ← key[y]

15 copy y's satellite data into z

16 if color[y] = BLACK

17 then RB-DELETE-FIXUP(T, x)

18 return y

“在删除结点后，原红黑树的性质可能被改变，如果删除的是红色结点，那么原红黑树的性质依旧保持，此时不用做修正操作，如果删除的结点是黑色结点，原红黑树的性质可能会被改变，我们要对其做修正操作。那么哪些树的性质会发生变化呢，如果删除结点不是树唯一结点，那么删除结点的那一个支的到各叶结点的黑色结点数会发生变化，此时性质5被破坏。如果被删结点的唯一非空子结点是红色，而被删结点的父结点也是红色，那么性质4被破坏。如果被删结点是根结点，而它的唯一非空子结点是红色，则删除后新根结点将变成红色，违背性质2。”

RB-DELETE-FIXUP(T, x) 恢复与保持红黑性质的工作

1 while x ≠ root[T] and color[x] = BLACK

2 do if x = left[p[x]]

3 then w ← right[p[x]]

4 if color[w] = RED

5 then color[w] ← BLACK ▹ Case 1

6 color[p[x]] ← RED ▹ Case 1

7 LEFT-ROTATE(T, p[x]) ▹ Case 1

8 w ← right[p[x]] ▹ Case 1

9 if color[left[w]] = BLACK and color[right[w]] = BLACK

10 then color[w] ← RED ▹ Case 2

11 x ← p[x] ▹ Case 2

12 else if color[right[w]] = BLACK

13 then color[left[w]] ← BLACK ▹ Case 3

14 color[w] ← RED ▹ Case 3

15 RIGHT-ROTATE(T, w) ▹ Case 3

16 w ← right[p[x]] ▹ Case 3

17 color[w] ← color[p[x]] ▹ Case 4

18 color[p[x]] ← BLACK ▹ Case 4

19 color[right[w]] ← BLACK ▹ Case 4

20 LEFT-ROTATE(T, p[x]) ▹ Case 4

21 x ← root[T] ▹ Case 4

22 else (same as then clause with "right" and "left" exchanged)

23 color[x] ← BLACK

“上面的修复情况看起来有些复杂，下面我们用一个分析技巧：我们从被删结点后来顶替它的那个结点开始调整，并认为它有额外的一重黑色。这里额外一重黑色是什么意思呢，我们不是把红黑树的结点加上除红与黑的另一种颜色，这里只是一种假设，我们认为我们当前指向它，因此空有额外一种黑色，可以认为它的黑色是从它的父结点被删除后继承给它的，它现在可以容纳两种颜色，如果它原来是红色，那么现在是红+黑，如果原来是黑色，那么它现在的颜色是黑+黑。有了这重额外的黑色，原红黑树性质5就能保持不变。现在只要恢复其它性质就可以了，做法还是尽量向根移动和穷举所有可能性。"--saturnman。

如果是以下情况，恢复比较简单：

* a)当前结点是红+黑色  
  解法，直接把当前结点染成黑色，结束此时红黑树性质全部恢复。
* b)当前结点是黑+黑且是根结点， 解法：什么都不做，结束。

但如果是以下情况呢？：

* 删除修复情况1：当前结点是黑+黑且兄弟结点为红色(此时父结点和兄弟结点的子结点分为黑)
* 删除修复情况2：当前结点是黑加黑且兄弟是黑色且兄弟结点的两个子结点全为黑色
* 删除修复情况3：当前结点颜色是黑+黑，兄弟结点是黑色，兄弟的左子是红色，右子是黑色
* 删除修复情况4：当前结点颜色是黑-黑色，它的兄弟结点是黑色，但是兄弟结点的右子是红色，兄弟结点左子的颜色任意

此时，我们需要调用RB-DELETE-FIXUP(T, x)，来恢复与保持红黑性质的工作。

下面，咱们便来分别处理这4种删除修复情况。

**删除修复情况1：当前结点是黑+黑且兄弟结点为红色(此时父结点和兄弟结点的子结点分为黑)。**

解法：把父结点染成红色，把兄弟结点染成黑色，之后重新进入算法（我们只讨论当前结点是其父结点左孩子时的情况）。此变换后原红黑树性质5不变，而把问题转化为兄弟结点为黑色的情况(注：变化前，原本就未违反性质5，只是为了**把问题转化为兄弟结点为黑色的情况**)。 即如下代码操作：

//调用RB-DELETE-FIXUP(T, x) 的1-8行代码

1 while x ≠ root[T] and color[x] = BLACK

2 do if x = left[p[x]]

3 then w ← right[p[x]]

4 if color[w] = RED

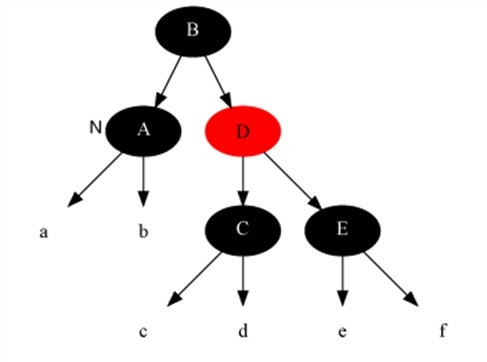
5 then color[w] ← BLACK ▹ Case 1

6 color[p[x]] ← RED ▹ Case 1

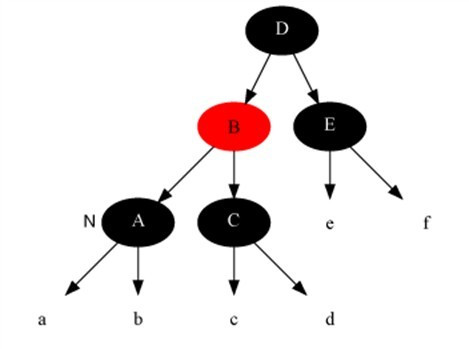
7 LEFT-ROTATE(T, p[x]) ▹ Case 1

8 w ← right[p[x]] ▹ Case 1

变化前：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/10.jpg)

变化后：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/11.jpg)

**删除修复情况2：当前结点是黑加黑且兄弟是黑色且兄弟结点的两个子结点全为黑色。**

解法：把当前结点和兄弟结点中抽取一重黑色追加到父结点上，把父结点当成新的当前结点，重新进入算法。（此变换后性质5不变），即调用RB-INSERT-FIXUP(T, z) 的第9-10行代码操作，如下：

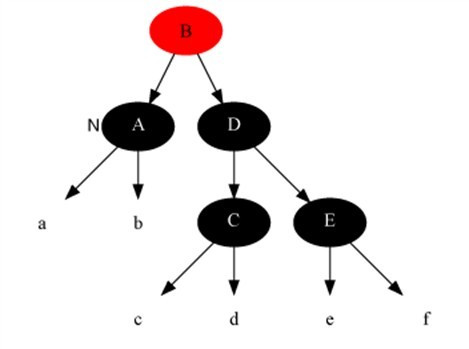
//调用RB-DELETE-FIXUP(T, x) 的9-11行代码

9 if color[left[w]] = BLACK and color[right[w]] = BLACK

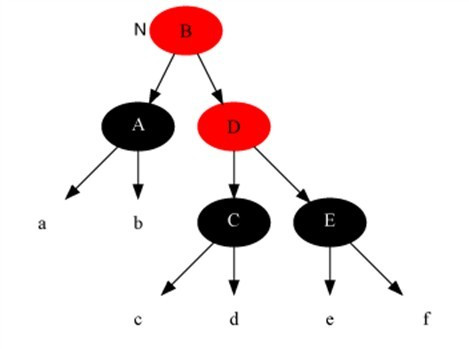
10 then color[w] ← RED ▹ Case 2

11 x p[x] ▹ Case 2

变化前：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/12.jpg)

变化后：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/13.jpg)

**删除修复情况3：当前结点颜色是黑+黑，兄弟结点是黑色，兄弟的左子是红色，右子是黑色。**

解法：把兄弟结点染红，兄弟左子结点染黑，之后再在兄弟结点为支点解右旋，之后重新进入算法。此是把当前的情况转化为情况4，而性质5得以保持，即调用RB-INSERT-FIXUP(T, z) 的第12-16行代码，如下所示：

//调用RB-DELETE-FIXUP(T, x) 的第12-16行代码

12 else if color[right[w]] = BLACK

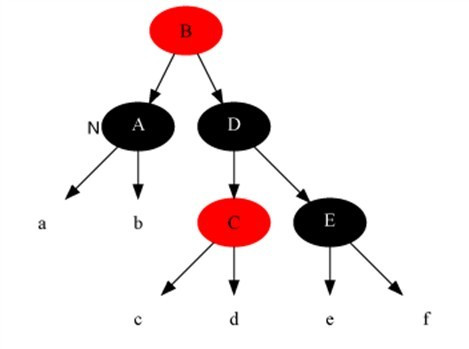
13 then color[left[w]] ← BLACK ▹ Case 3

14 color[w] ← RED ▹ Case 3

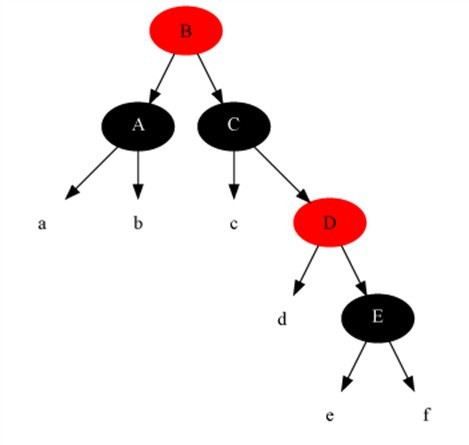
15 RIGHT-ROTATE(T, w) ▹ Case 3

16 w ← right[p[x]] ▹ Case 3

变化前：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/14.jpg)

变化后：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/15.jpg)

**删除修复情况4：当前结点颜色是黑-黑色，它的兄弟结点是黑色，但是兄弟结点的右子是红色，兄弟结点左子的颜色任意。**

解法：把兄弟结点染成当前结点父结点的颜色，把当前结点父结点染成黑色，兄弟结点右子染成黑色，之后以当前结点的父结点为支点进行左旋，此时算法结束，红黑树所有性质调整正确，即调用RB-INSERT-FIXUP(T, z)的第17-21行代码，如下所示：

//调用RB-DELETE-FIXUP(T, x) 的第17-21行代码

17 color[w] ← color[p[x]] ▹ Case 4

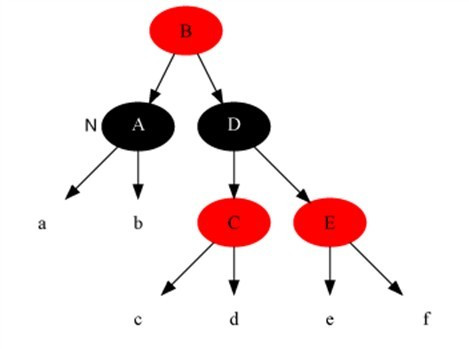
18 color[p[x]] ← BLACK ▹ Case 4

19 color[right[w]] ← BLACK ▹ Case 4

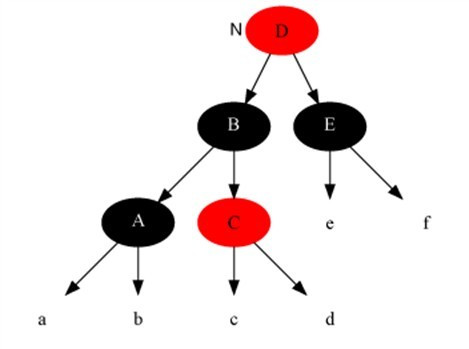
20 LEFT-ROTATE(T, p[x]) ▹ Case 4

21 x ← root[T] ▹ Case 4

变化前：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/16.jpg)

变化后：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/rbtree/17.jpg)

### 本文参考

本文参考了算法导论、STL源码剖析、计算机程序设计艺术等资料，并推荐阅读这个PDF：Left-Leaning Red-Black Trees, Dagstuhl Workshop on Data Structures, Wadern, Germany, February, 2008.  
下载地址：[http://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf](http://www.cs.princeton.edu/%7Ers/talks/LLRB/RedBlack.pdf)。

# 3.2 B树

#### 1.前言：

动态查找树主要有：二叉查找树（Binary Search Tree），平衡二叉查找树（Balanced Binary Search Tree），[红黑树](http://blog.csdn.net/v_JULY_v/article/category/774945)(Red-Black Tree )，B-tree/B+-tree/ B\*-tree (B~Tree)。前三者是典型的二叉查找树结构，其查找的时间复杂度O(log2N)与树的深度相关，那么降低树的深度自然会提高查找效率。

但是咱们有面对这样一个实际问题：就是大规模数据存储中，实现索引查询这样一个实际背景下，树节点存储的元素数量是有限的（如果元素数量非常多的话，查找就退化成节点内部的线性查找了），这样导致二叉查找树结构由于**树的深度过大而造成磁盘I/O读写过于频繁，进而导致查询效率低下**，因此我们该想办法降低树的深度，从而减少磁盘查找存取的次数。一个基本的想法就是：采用**多叉树**结构（由于树节点元素数量是有限的，自然该节点的子树数量也就是有限的）。

这样我们就提出了一个新的查找树结构——平衡多路查找树，即**B-tree（B-tree树即B树\**，B即Balanced，平衡的意思）\*，这棵神奇的树是在[Rudolf Bayer, Edward M. McCreight](http://academic.research.microsoft.com/Author/1008233/rudolf-bayer)(1970)写的一篇论文《Organization and Maintenance of Large Ordered Indices》中首次提出的。

后面我们会看到，B树的各种操作能使B树保持较低的高度，从而有效避免磁盘过于频繁的查找存取操作，达到有效提高查找效率的目的。然在开始介绍B~tree之前，先了解下相关的硬件知识，才能很好的了解为什么需要B~tree这种外存数据结构。

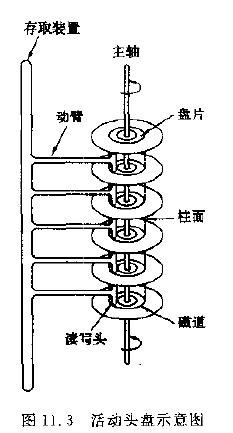
#### 2.外存储器—磁盘

计算机存储设备一般分为两种：内存储器(main memory)和外存储器(external memory)。 内存存取速度快，但容量小，价格昂贵，而且不能长期保存数据(在不通电情况下数据会消失)。

外存储器—磁盘是一种直接存取的存储设备(DASD)。它是以存取时间变化不大为特征的。可以直接存取任何字符组，且容量大、速度较其它外存设备更快。

##### 2.1 磁盘的构造

磁盘是一个扁平的圆盘(与电唱机的唱片类似)。盘面上有许多称为磁道的圆圈，数据就记录在这些磁道上。磁盘可以是单片的，也可以是由若干盘片组成的盘组，每一盘片上有两个面。如下图11.3中所示的6片盘组为例，除去最顶端和最底端的外侧面不存储数据之外，一共有10个面可以用来保存信息。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/1.jpg)

当磁盘驱动器执行读/写功能时。盘片装在一个主轴上，并绕主轴高速旋转，当磁道在读/写头(又叫磁头) 下通过时，就可以进行数据的读 / 写了。

一般磁盘分为固定头盘(磁头固定)和活动头盘。固定头盘的每一个磁道上都有独立的磁头，它是固定不动的，专门负责这一磁道上数据的读/写。

活动头盘 (如上图)的磁头是可移动的。每一个盘面上只有一个磁头(磁头是双向的，因此正反盘面都能读写)。它可以从该面的一个磁道移动到另一个磁道。所有磁头都装在同一个动臂上，因此不同盘面上的所有磁头都是同时移动的(行动整齐划一)。当盘片绕主轴旋转的时候，磁头与旋转的盘片形成一个圆柱体。各个盘面上半径相同的磁道组成了一个圆柱面，我们称为柱面 。因此，柱面的个数也就是盘面上的磁道数。

##### 2.2 磁盘的读/写原理和效率

磁盘上数据必须用一个三维地址唯一标示：柱面号、盘面号、块号(磁道上的盘块)。

读/写磁盘上某一指定数据需要下面3个步骤：

1. 首先移动臂根据柱面号使磁头移动到所需要的柱面上，这一过程被称为定位或查找 。
2. 如上图11.3中所示的6盘组示意图中，所有磁头都定位到了10个盘面的10条磁道上(磁头都是双向的)。这时根据盘面号来确定指定盘面上的磁道。
3. 盘面确定以后，盘片开始旋转，将指定块号的磁道段移动至磁头下。

经过上面三个步骤，指定数据的存储位置就被找到。这时就可以开始读/写操作了。

访问某一具体信息，由3部分时间组成：

* 查找时间(seek time) Ts: 完成上述步骤(1)所需要的时间。这部分时间代价最高，最大可达到0.1s左右。
* 等待时间(latency time) Tl: 完成上述步骤(3)所需要的时间。由于盘片绕主轴旋转速度很快，一般为7200转/分(电脑硬盘的性能指标之一, 家用的普通硬盘的转速一般有5400rpm(笔记本)、7200rpm几种)。因此一般旋转一圈大约0.0083s。
* 传输时间(transmission time) Tt: 数据通过系统总线传送到内存的时间，一般传输一个字节(byte)大概0.02us=2\*10^(-8)s

**磁盘读取数据是以盘块**(block)**为基本单位的。**位于同一盘块中的所有数据都能被一次性全部读取出来。而磁盘IO代价主要花费在查找时间Ts上。因此我们应该尽量将相关信息存放在同一盘块，同一磁道中。或者至少放在同一柱面或相邻柱面上，以求在**读/写信息时尽量减少磁头来回移动的次数，避免过多的查找时间Ts。**

所以，在大规模数据存储方面，大量数据存储在外存磁盘中，而在外存磁盘中读取/写入块(block)中某数据时，首先需要定位到磁盘中的某块，如何有效地查找磁盘中的数据，需要一种合理高效的外存数据结构，就是下面所要重点阐述的B-tree结构，以及相关的变种结构：B+-tree结构和B\*-tree结构。

#### 3.B- 树

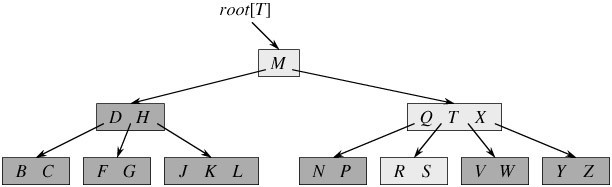
##### 3.1 什么是B-树

B-树，即为B树。顺便说句，因为B树的原英文名称为B-tree，而国内很多人喜欢把B-tree译作B-树，其实，这是个非常不好的直译，很容易让人产生误解。如人们可能会以为B-树是一种树，而B树又是另外一种树。而事实上是，**B-tree就是指的B树**。

我们知道，B 树是为了磁盘或其它存储设备而设计的一种多叉（下面你会看到，相对于二叉，B树每个内结点有多个分支，即多叉）平衡查找树。与之前介绍的红黑树很相似，但在降低磁盘I/0操作方面要更好一些。许多数据库系统都一般使用B树或者B树的各种变形结构，如下文即将要介绍的B+树，B\*树来存储信息。

B树与红黑树最大的不同在于，B树的结点可以有许多子女，从几个到几千个。不过B树与红黑树一样，一棵含n个结点的B树的高度也为O(lgn)，但可能比一棵红黑树的高度小许多，因为它的分支因子比较大。所以，B树可以在O（logn）时间内，实现各种如插入（insert），删除（delete）等动态集合操作。

如下图所示，即是一棵B树，一棵关键字为英语中辅音字母的B树，现在要从树中查找字母R（包含n[x]个关键字的内结点x，x有n[x]+1个子女（也就是说，一个内结点x若含有n[x]个关键字，那么x将含有n[x]+1个子女）。所有的叶结点都处于相同的深度，带阴影的结点为查找字母R时要检查的结点）：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/2.jpg)

**相信，从上图你能轻易的看到，一个内结点x若含有n[x]个关键字，那么x将含有n[x]+1个子女。如含有2个关键字D H的内结点有3个子女，而含有3个关键字Q T X的内结点有4个子女。**

**B树**的定义

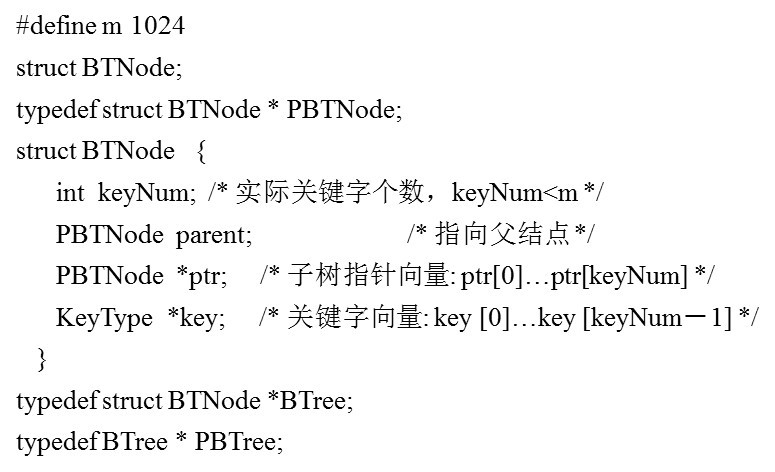
B 树又叫平衡多路查找树。**一棵m阶的B 树** (注：切勿简单的认为一棵m阶的B树是m叉树，虽然存在[四叉树](http://zh.wikipedia.org/wiki/%E5%9B%9B%E5%8F%89%E6%A0%91)，[八叉树](http://zh.wikipedia.org/wiki/%E5%85%AB%E5%8F%89%E6%A0%91)，[KD树](http://blog.csdn.net/v_july_v/article/details/8203674)，及vp/R树/R\*树/R+树/X树/M树/线段树/希尔伯特R树/优先R树等空间划分树，但与B树完全不等同)**的特性如下：**

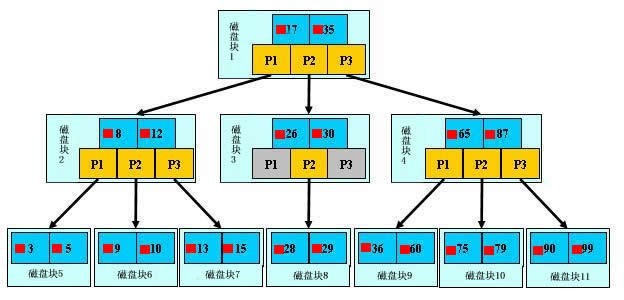
1. 树中每个结点最多含有m个孩子（m>=2）；
2. 除根结点和叶子结点外，其它每个结点至少有[ceil(m / 2)]个孩子（其中ceil(x)是一个取上限的函数）；
3. 根结点至少有2个孩子（除非B树只包含一个结点：根结点）；
4. 所有叶子结点都出现在同一层，叶子结点不包含任何关键字信息(可以看做是外部结点或查询失败的结点，指向这些结点的指针都为null)；（注：叶子节点只是没有孩子和指向孩子的指针，这些节点也存在，也有元素。类似红黑树中，每一个NULL指针即当做叶子结点，只是没画出来而已）。
5. 每个非终端结点中包含有n个关键字信息： (n，P0，K1，P1，K2，P2，......，Kn，Pn)。其中：  
   a) Ki (i=1...n)为关键字，且关键字按顺序升序排序K(i-1)< Ki。  
   b) Pi为指向子树根的结点，且指针P(i-1)指向子树种所有结点的关键字均小于Ki，但都大于K(i-1)。  
   c) 关键字的个数n必须满足： [ceil(m / 2)-1]<= n <= m-1。比如有j个孩子的非叶结点恰好有j-1个关键码。

B树中的每个结点根据实际情况可以包含大量的关键字信息和分支(当然是不能超过磁盘块的大小，根据磁盘驱动(disk drives)的不同，一般块的大小在1k~4k左右)；这样树的深度降低了，这就意味着查找一个元素只要很少结点从外存磁盘中读入内存，很快访问到要查找的数据。

##### 3.2 B树的类型和节点定义

B树的类型和节点定义如下图所示：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/5.gif)

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/6.jpg)

##### 3.3 文件查找的具体过程(涉及磁盘IO操作)

为了简单，这里用少量数据构造一棵3叉树的形式，实际应用中的B树结点中关键字很多的。上面的图中比如根结点，其中17表示一个磁盘文件的文件名；小红方块表示这个17文件内容在硬盘中的存储位置；p1表示指向17左子树的指针。

其结构可以简单定义为：

typedef struct {

/\*文件数\*/

int file\_num;

/\*文件名(key)\*/

char \* file\_name[max\_file\_num];

/\*指向子节点的指针\*/

BTNode \* BTptr[max\_file\_num+1];

/\*文件在硬盘中的存储位置\*/

FILE\_HARD\_ADDR offset[max\_file\_num];

}BTNode;

假如每个盘块可以正好存放一个B树的结点（正好存放2个文件名）。那么一个BTNODE结点就代表一个盘块，而子树指针就是存放另外一个盘块的地址。

下面，咱们来模拟下查找文件29的过程：

1. 根据根结点指针找到文件目录的根磁盘块1，将其中的信息导入内存。【磁盘IO操作 1次】
2. 此时内存中有两个文件名17、35和三个存储其他磁盘页面地址的数据。根据算法我们发现：17<29<35，因此我们找到指针p2。
3. 根据p2指针，我们定位到磁盘块3，并将其中的信息导入内存。【磁盘IO操作 2次】
4. 此时内存中有两个文件名26，30和三个存储其他磁盘页面地址的数据。根据算法我们发现：26<29<30，因此我们找到指针p2。
5. 根据p2指针，我们定位到磁盘块8，并将其中的信息导入内存。【磁盘IO操作 3次】
6. 此时内存中有两个文件名28，29。根据算法我们查找到文件名29，并定位了该文件内存的磁盘地址。

分析上面的过程，发现需要**3次磁盘IO操作和3次内存查找**操作。关于内存中的文件名查找，由于是一个有序表结构，可以利用折半查找提高效率。至于IO操作是影响整个B树查找效率的决定因素。

当然，如果我们使用平衡二叉树的磁盘存储结构来进行查找，磁盘4次，最多5次，而且文件越多，B树比平衡二叉树所用的磁盘IO操作次数将越少，效率也越高。

##### 3.4 B树的高度

根据上面的例子我们可以看出，对于辅存做IO读的次数取决于B树的高度。而B树的高度又怎么求呢？

对于一棵含有N个关键字，m阶的B树来说（据B树的定义可知：m满足：ceil(m/2)<=**m**<=m，m阶即代表树中任一结点最多含有m个孩子，如5阶代表每个结点最多5个孩子，或俗称5叉树），且从1开始计数的话，其高度h为：

h <= log\_ceil(m/2) (n + 1)/2 + 1

这个B树的高度公式从侧面显示了B树的查找效率是相当高的。为什么呢？因为底数m/2可以取很大，如m可以达到几千，从而在关键字数一定的情况下，使得最终的h值尽量比较小，树的高度比较低。

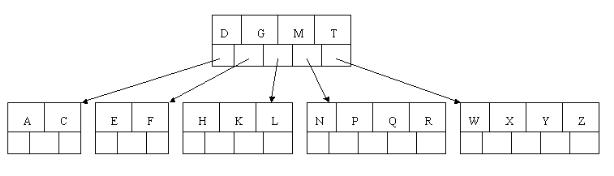
树的高度降低了，磁盘存取的次数也随着树高度的降低而减少，从而使得存取性能也相应提升。

#### 4、B树的插入、删除操作

根据B树的性质可知，如果是一棵m阶的B 树，那么有：

* 树中每个结点含有最多含有m个孩子，即m满足：ceil(m/2)<=**m**<=m。
* 除根结点和叶子结点外，其它每个结点至少有[ceil(m / 2)]个孩子（其中ceil(x)是一个取上限的函数）；
* 除根结点之外的结点的关键字的个数n必须满足： [ceil(m / 2)-1]<= n <= m-1（叶子结点也必须满足此条关于关键字数的性质）。

下面咱们通过另外一个实例来对这棵B树的插入（insert）,删除（delete）基本操作进行详细的介绍。以一棵**5阶**（即树中任一结点至多含有4个关键字，5棵子树）B树实例进行讲解(如下图所示)：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/11.jpg)

在上图所示的一棵5阶B树中，读者可以看到关键字数2-4个，内结点孩子数3-5个。**关键字数（2-4个）针对包括叶子结点在内的非根结点，孩子数（3-5个）则针对根结点和叶子结点之外的内结点。同时，根结点是必须至少有2个孩子的，不然就成直线型搜索树了。**且关键字为大写字母，顺序为字母升序。

结点定义如下：

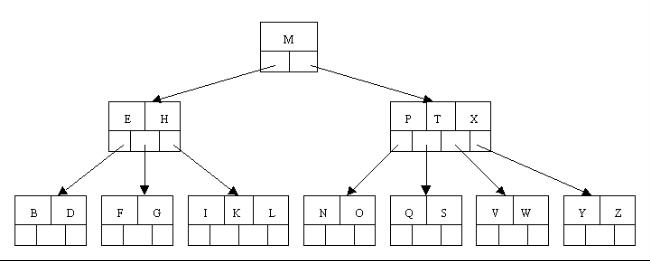
typedef struct{

int Count; // 当前节点中关键元素数目

ItemType Key[4]; // 存储关键字元素的数组

long Branch[5]; // 伪指针数组，(记录数目)方便判断合并和分裂的情况

} NodeType;

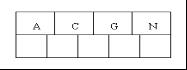
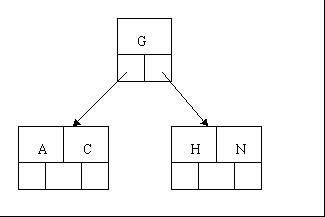
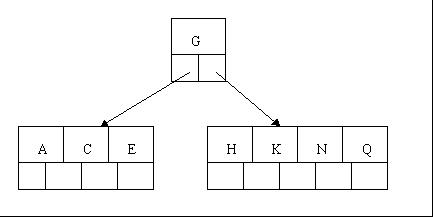
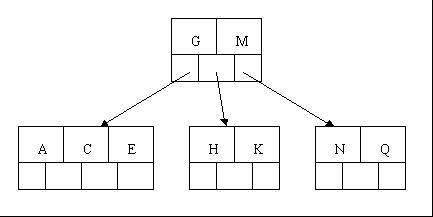
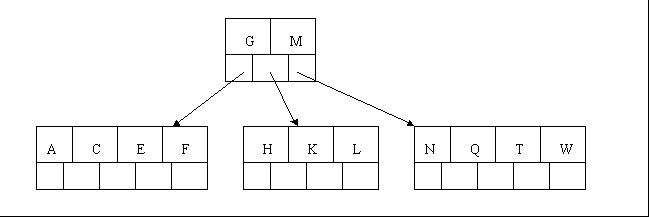
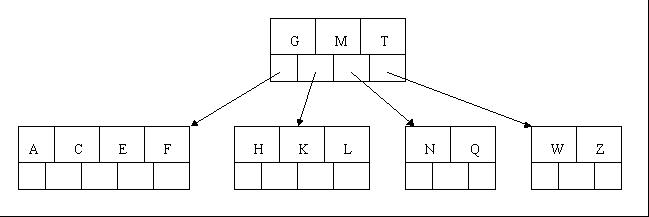
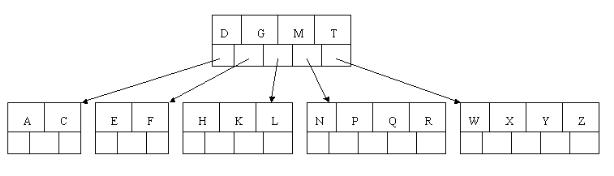
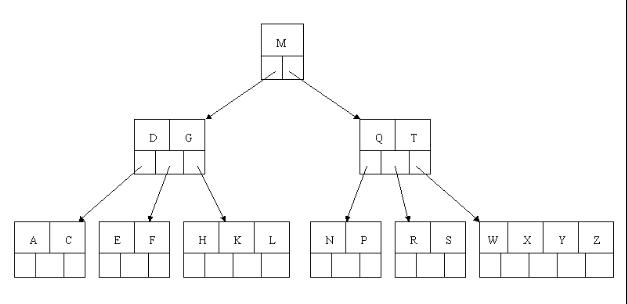
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/12.jpg)

##### 4.1 插入（insert）操作

针对一棵高度为h的m阶B树，插入一个元素时，首先在B树中是否存在，如果不存在，一般在叶子结点中插入该新的元素，此时分3种情况：

* 如果叶子结点空间足够，即该结点的关键字数小于m-1，则直接插入在叶子结点的左边或右边；
* 如果空间满了以致没有足够的空间去添加新的元素，即该结点的关键字数已经有了m个，则需要将该**结点**进行“分裂”，将一半数量的关键字元素分裂到新的其相邻右结点中，中间关键字元素上移到父结点中，而且当结点中关键元素向右移动了，相关的指针也需要向右移。
  + 此外，如果在上述中间关键字上移到父结点的过程中，导致根结点空间满了，那么根结点也要进行分裂操作，这样原来的根结点中的中间关键字元素向上移动到新的根结点中，因此导致树的高度增加一层。

下面咱们通过一个实例来逐步讲解下。插入以下字符字母到一棵空的5阶B 树中：C N G A H E K Q M F W L T Z D P R X Y S，而且，因为是5阶B树，所以必有非根结点**关键字数**小了（小于2个）就合并，大了（超过4个）就分裂。

1. 首先，结点空间足够，刚开始的4个字母可以直接到插入相同的结点中，如下图：  
   [](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/14.jpg)
2. 插入H结点时，发现结点空间不够，所以将其分裂成2个结点，移动中间元素G上移到新的根结点中，且把A和C留在当前结点中，而H和N放置在新的右邻居结点中。如下图：  
   [](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/15.jpg)
3. 当插入E,K,Q时，不需要任何分裂操作  
   [](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/16.jpg)
4. 插入M需要一次分裂，注意到M恰好是中间关键字元素，所以M向上移到父节点中  
   [](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/17.jpg)
5. 插入F,W,L,T不需要任何分裂操作  
   [](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/18.jpg)
6. 插入Z时，最右的叶子结点空间满了，需要进行分裂操作，中间元素T上移到父节点中[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/19.jpg)
7. 插入D时，导致最左边的叶子结点被分裂，D恰好也是中间元素，上移到父节点中，然后字母P,R,X,Y直接陆续插入，不需要任何分裂操作  
   [](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/20.jpg)
8. 最后，当插入S时，含有N,P,Q,R的结点需要分裂，把中间元素Q上移到父节点中，但是问题来了，因为Q上移导致父结点 “D G M T” 也满了，所以也要进行分裂，将父结点中的中间元素M上移到新形成的根结点中，从而致使树的高度增加一层。[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/21.jpg)

##### 4.2、删除(delete)操作

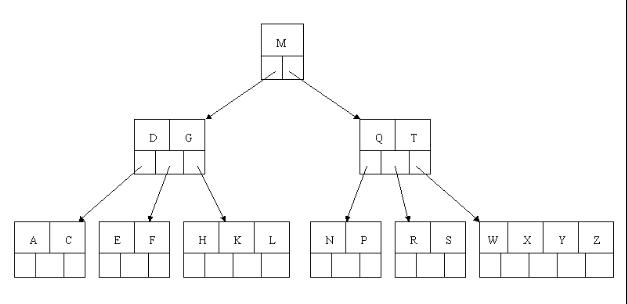
下面介绍删除操作，删除操作相对于插入操作要考虑的情况多点。

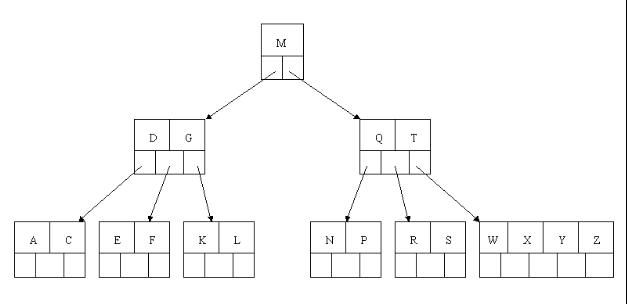
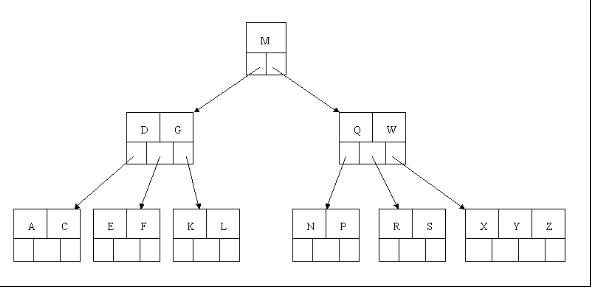
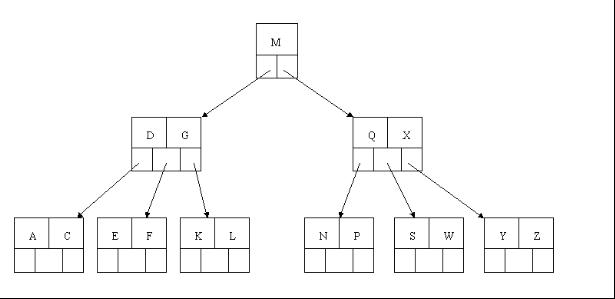
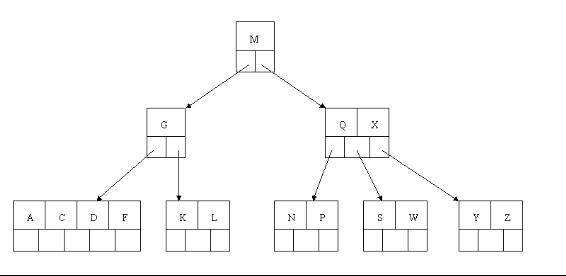
* 首先查找B树中需删除的元素,如果该元素在B树中存在，则将该元素在其结点中进行删除，如果删除该元素后，首先判断该元素是否有左右孩子结点
  + 如果有，则上移孩子结点中的某相近元素(“左孩子最右边的节点”或“右孩子最左边的节点”)到父节点中，然后是移动之后的情况；
  + 如果没有，直接删除后，移动之后的情况。

删除元素，移动相应元素之后，如果某结点中元素数目（即关键字数）小于**ceil(m/2)-1**，则需要看其某相邻兄弟结点是否丰满（结点中元素个数大于ceil(m/2)-1）

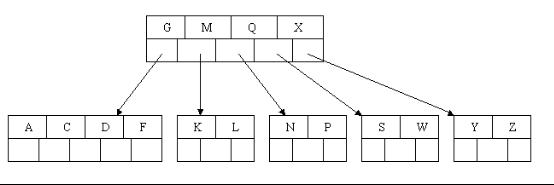
* 如果丰满，则向父节点借一个元素来满足条件；
* 如果其相邻兄弟都刚脱贫，即借了之后其结点数目小于ceil(m/2)-1，则该结点与其相邻的某一兄弟结点进行“合并”成一个结点，以此来满足条件。

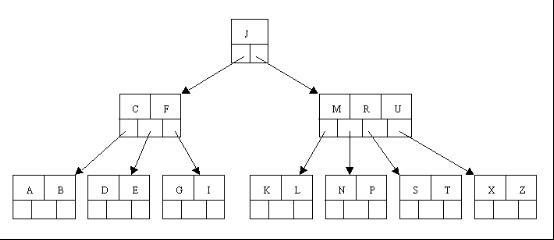
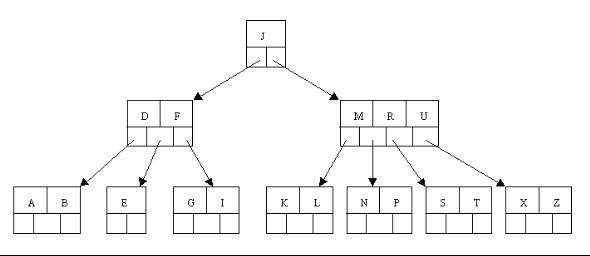
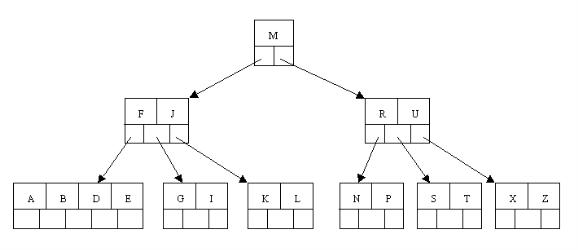
下面咱们还是以上述插入操作构造的一棵5阶B树（**树中除根结点和叶子结点外的任意结点的孩子数m满足3<=m<=5，除根结点外的任意结点的关键字数n满足：2<=n<=4，所以关键字数小于2个就合并，超过4个就分裂**）为例，依次删除H,T,R,E。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/22.jpg)

1. 首先删除元素H，当然首先查找H，H在一个叶子结点中，且该叶子结点元素数目3大于最小元素数目ceil(m/2)-1=2，则操作很简单，咱们只需要移动K至原来H的位置，移动L至K的位置（也就是结点中删除元素后面的元素向前移动）  
   [](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/23.jpg)
2. 下一步，删除T,因为T没有在叶子结点中，而是在中间结点中找到，咱们发现他的继承者W(字母升序的下个元素)，将W上移到T的位置，然后将原包含W的孩子结点中的W进行删除，这里恰好删除W后，该孩子结点中元素个数大于2，无需进行合并操作。  
   [](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/24.jpg)
3. 下一步删除R，R在叶子结点中,但是该结点中元素数目为2，删除导致只有1个元素，已经小于最小元素数目ceil(5/2)-1=2,而由前面我们已经知道：**如果其某个相邻兄弟结点中比较丰满（元素个数大于ceil(5/2)-1=2），则可以向父结点借一个元素，然后将最丰满的相邻兄弟结点中上移最后或最前一个元素到父节点中**（有没有看到红黑树中左旋操作的影子?）。 故在这个实例中，由于右相邻兄弟结点“X Y Z”比较丰满，而删除元素R后，导致“S”结点稀缺
   * 所以原来的的“R S”结点先向父节点借一个元素W下移到该叶子结点中，代替原来S的位置，S前移；
   * 然后相邻右兄弟结点中的X上移到父结点中；
   * 最后相邻右兄弟结点中元素Y和Z前移。  
     [](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/25.jpg)
4. 最后一步删除E， 删除后会导致很多问题，因为E所在的结点数目刚好达标，刚好满足最小元素个数（ceil(5/2)-1=2）,而相邻的兄弟结点也是同样的情况，删除一个元素都不能满足条件，所以需要**该节点与某相邻兄弟结点进行合并操作**；
   * 首先移动父结点中的元素（该元素在两个需要合并的两个结点元素之间）下移到其子结点中，
   * 然后将这两个结点进行合并成一个结点。所以在该实例中，咱们首先将父节点中的元素D下移到已经删除E而只有F的结点中，然后将含有D和F的结点和含有A,C的相邻兄弟结点进行合并成一个结点。  
     [](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/26.jpg)

也许你认为这样删除操作已经结束了，其实不然，在看看上图，对于这种特殊情况，你立即会发现父节点只包含一个元素G，没达标（因为非根节点包括叶子结点的关键字数n必须满足于2=<n<=4，而此处的n=1），这是不能够接受的。如果这个问题结点的相邻兄弟比较丰满，则可以向父结点借一个元素。假设这时右兄弟结点（含有Q,X）有一个以上的元素（Q右边还有元素），然后咱们将M下移到元素很少的子结点中，将Q上移到M的位置，这时，Q的左子树将变成M的右子树，也就是含有N，P结点被依附在M的右指针上。

所以在这个实例中，咱们没有办法去借一个元素，只能与兄弟结点进行合并成一个结点，而根结点中的唯一元素M下移到子结点，这样，树的高度减少一层。  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/27.jpg)

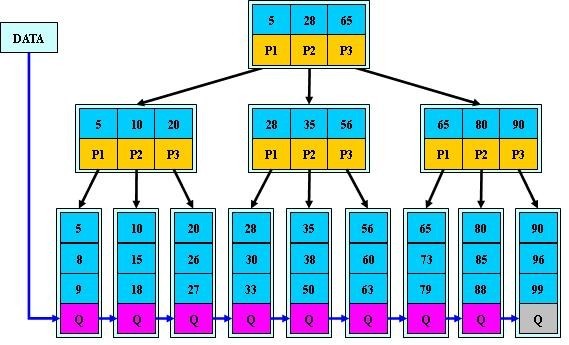
为了进一步详细讨论删除的情况，**再举另外一个实例**：  
这里是一棵不同的5序B树，那咱们试着删除C  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/28.jpg)  
于是将删除元素C的右子结点中的D元素上移到C的位置，但是出现上移元素后，只有一个元素的结点的情况。  
又因为含有E的结点，其相邻兄弟结点才刚脱贫（最少元素个数为2），不可能向父节点借元素，所以只能进行合并操作，于是这里将含有A,B的左兄弟结点和含有E的结点进行合并成一个结点。  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/29.jpg)  
这样又出现只含有一个元素F结点的情况，这时，其相邻的兄弟结点是丰满的（元素个数为3>最小元素个数2），这样就可以想父结点借元素了，把父结点中的J下移到该结点中，相应的如果结点中J后有元素则前移，然后相邻兄弟结点中的第一个元素（或者最后一个元素）上移到父节点中，后面的元素（或者前面的元素）前移（或者后移）；注意含有K，L的结点以前依附在M的左边，现在变为依附在J的右边。这样每个结点都满足B树结构性质。  
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/30.jpg)  
从以上操作可看出：除根结点之外的结点（包括叶子结点）的关键字的个数n满足：（ceil(m / 2)-1）<= n <= m-1，即2<=n<=4。这也佐证了咱们之前的观点。删除操作完。

#### 5.B+-tree

B+-tree：是应文件系统所需而产生的一种B-tree的变形树。

一棵m阶的B+树和m阶的B树的异同点在于：

1. 有n棵子树的结点中含有n-1 个关键字； (与B 树n棵子树有n-1个关键字 保持一致，参照：<http://en.wikipedia.org/wiki/B%2B_tree#Overview>，而下面**B+树的图可能有问题**，请读者注意)
2. 所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。 (而B 树的叶子节点并没有包括全部需要查找的信息)
3. **所有的非终端结点可以看成是索引部分**，结点中仅含有其子树根结点中最大（或最小）关键字。 (而B 树的非终节点也包含需要查找的有效信息)

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/8.jpg)

a) 为什么说B+-tree比B 树更适合实际应用中操作系统的文件索引和数据库索引？

1. B+-tree的磁盘读写代价更低B+-tree的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。

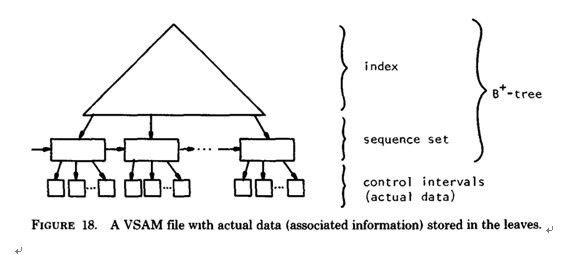
举个例子，假设磁盘中的一个盘块容纳16bytes，而一个关键字2bytes，一个关键字具体信息指针2bytes。一棵9阶B-tree(一个结点最多8个关键字)的内部结点需要2个盘块。而B+ 树内部结点只需要1个盘快。当需要把内部结点读入内存中的时候，B 树就比B+ 树多一次盘块查找时间(在磁盘中就是盘片旋转的时间)。

1. B+-tree的查询效率更加稳定

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

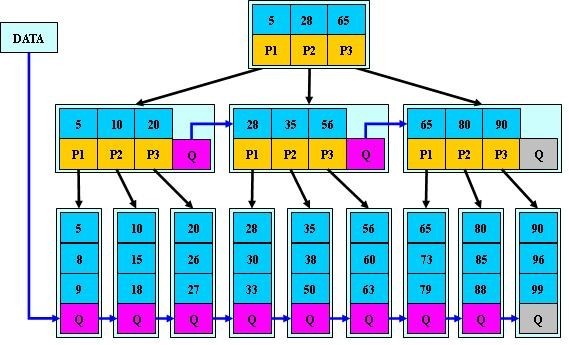
总而言之，B树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。正是为了解决这个问题，B+树应运而生。B+树只要遍历叶子节点就可以实现整棵树的遍历，支持基于范围的查询，而B树不支持range-query这样的操作（或者说效率太低）。

b) B+-tree的应用: VSAM(虚拟存储存取法)文件(来源论文 the ubiquitous Btree 作者：D COMER - 1979 )

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/9.jpg)

#### 6.B\*-tree

B\*-tree是B+-tree的变体，在B+树的基础上(所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针)，B\*树中非根和非叶子结点再增加指向兄弟的指针；B\*树定义了非叶子结点关键字个数至少为(2/3)\*M，即块的最低使用率为2/3（代替B+树的1/2）。给出了一个简单实例，如下图所示：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/7/10.jpg)

B+树的分裂：当一个结点满时，分配一个新的结点，并将原结点中1/2的数据复制到新结点，最后在父结点中增加新结点的指针；B+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针。

B\*树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制1/3的数据到新结点，最后在父结点增加新结点的指针。

所以，B\*树分配新结点的概率比B+树要低，空间使用率更高；

#### 7.总结

通过以上介绍，大致将B树，B+树，B\*树总结如下：

* B树：有序数组+平衡多叉树；
* B+树：有序数组链表+平衡多叉树；
* B\*树：一棵丰满的B+树。

顺便说一句，无论是B树，还是B+树、b树，由于根或者树的上面几层被反复查询，所以这几块可以存在内存中，换言之，B树、B+树、B树的根结点和部分顶层数据在内存中，大部分下层数据在磁盘上。

# 3.3 最近公共祖先LCA问题

## 问题描述

求有根树的任意两个节点的最近公共祖先。

## 分析与解法

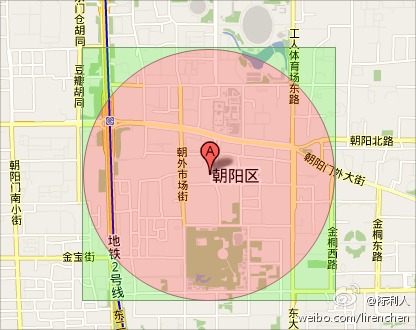
解答这个问题之前，咱们得先搞清楚到底什么是最近公共祖先。最近公共祖先简称LCA（Lowest Common Ancestor），所谓LCA，是当给定一个有根树T时，对于任意两个结点u、v，找到一个离根最远的结点x，使得x同时是u和v的祖先，x 便是u、v的最近公共祖先。（参见：<http://en.wikipedia.org/wiki/Lowest_common_ancestor> ）原问题涵盖一般性的有根树，本文为了简化，多使用二叉树来讨论。

举个例子，如针对下图所示的一棵普通的二叉树来讲：

# 3.10本章堆栈树图相关的习题

**1、附近地点搜索**

找一个点集中与给定点距离最近的点，同时，给定的二维点集都是固定的，查询可能有很多次，例如，坐标(39.91, 116.37)附近500米内有什么餐馆，那么让你来设计，该怎么做？

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/36~37/37.1.jpg)

提示：可以建立R树进行二维搜索，或使用GeoHash算法解决。

**2、最小操作数**

给定一个单词集合Dict，其中每个单词的长度都相同。现从此单词集合Dict中抽取两个单词A、B，我们希望通过若干次操作把单词A变成单词B，每次操作可以改变单词的一个字母，同时，新产生的单词必须是在给定的单词集合Dict中。求所有行得通步数最少的修改方法。

举个例子如下：

Given: A = "hit" B = "cog" Dict = ["hot","dot","dog","lot","log"] Return [ ["hit","hot","dot","dog","cog"], ["hit","hot","lot","log","cog"] ]

即把字符串A = "hit"转变成字符串B = "cog"，有以下两种可能：

"hit" -> "hot" -> "dot" -> "dog" -> "cog"；

"hit" -> "hot" -> "lot" -> "log" ->"cog"。

提示：建图然后搜索。

**3、最少操作次数的简易版**

给定两个字符串，仅由小写字母组成，它们包含了相同字符。 求把第一个字符串变成第二个字符串的最小操作次数，且每次操作只能对第一个字符串中的某个字符移动到此字符串中的开头。 例如给定两个字符串“abcd" "bcad" ，输出：2，因为需要操作2次才能把"abcd"变成“bcad" ，方法是：abcd->cabd->bcad。

**3、把二元查找树转变成排序的双向链表**

输入一棵二元查找树，将该二元查找树转换成一个排序的双向链表。要求不能创建任何新的结点，只调整指针的指向。例如把下述二叉查找树

10

/ /

6 14

/ / / /

4 8 12

转换成双向链表，即得：

4=6=8=10=12=14=16。

**4、在二元树中找出和为某一值的所有路径**

输入一个整数和一棵二元树。 从树的根结点开始往下访问一直到叶结点所经过的所有结点形成一条路径。 打印出和与输入整数相等的所有路径。

**5、判断整数序列是不是二元查找树的后序遍历结果**

输入一个整数数组，判断该数组是不是某二元查找树的后序遍历的结果，如果是返回true，否则返回false。

例如输入5、7、6、9、11、10、8，由于这一整数序列是如下树的后序遍历结果：

8

/ /

6 10

/ / / /

5 7 9 11因此返回true。

如果输入7、4、6、5，没有哪棵树的后序遍历的结果是这个序列，因此返回false。

**6、设计包含min函数的栈**

定义栈的数据结构，要求添加一个min函数，能够得到栈的最小元素。要求函数min、push以及pop的时间复杂度都是O(1)。

**7、求二叉树中节点的最大距离**

如果我们把二叉树看成一个图，父子节点之间的连线看成是双向的，我们姑且定义"距离"为两节点之间边的个数。

请写一个程序，求一棵二叉树中相距最远的两个节点之间的距离。

**8**

输入一颗二元树，从上往下按层打印树的每个结点，同一层中按照从左往右的顺序打印。

例如输入

8

/ /

6 10

/ / / /

5 7 9 11

输出8 6 10 5 7 9 11。

**9**

请用递归和非递归俩种方法实现二叉树的前序遍历。

**10、求树的深度**

输入一棵二元树的根结点，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

例如：输入二元树： 10

/ /

6 14

/ / /

4 12 16

输出该树的深度3。

实现简单的一个查找二叉树的深度的函数。

**11、用俩个栈实现队列**

某队列的声明如下：

template<typename T> class CQueue

{

public:

CQueue() {}

~CQueue() {}

void appendTail(const T& node); // append a element to tail

void deleteHead(); // remove a element from head

private:

T> m\_stack1;

T> m\_stack2;

};

提示：这道题实质上是要求我们用两个栈来实现一个队列。栈是一种后入先出的数据容器，因此对队列进行的插入和删除操作都是在栈顶上进行；队列是一种先入先出的数据容器，我们总是把新元素插入到队列的尾部，而从队列的头部删除元素。

**12**

假设有一颗二叉树，已知这棵树的节点上不均匀的分布了若干石头，石头数跟这棵二叉树的节点数相同，石头只可以在子节点和父节点之间进行搬运，每次只能搬运一颗石头。请问如何以最少的步骤将石头搬运均匀，使得每个节点上的石头上刚好为1。

**13**

对于一颗完全二叉树，要求给所有节点加上一个pNext指针，指向同一层的相邻节点；如果当前节点已经是该层的最后一个节点，则将pNext指针指向NULL；给出程序实现，并分析时间复杂度和空间复杂度。

**14**

两个用户之间可能互相认识，也可能是单向的认识，用什么数据结构来表示？如果一个用户不认识别人，而且别人也不认识他，那么他就是无效节点，如何找出这些无效节点？自定义数据接口并实现之，要求尽可能节约内存和空间复杂度。

**15**

有一个一亿节点的树，现在已知两个点，找这两个点的共同的祖先。

**16**

给一个二叉树，每个节点都是正或负整数，如何找到一个子树，它所有节点的和最大？

提示：后序遍历，每一个节点保存左右子树的和加上自己的值。额外一个空间存放最大值。

写完后序遍历，面试官可能接着与你讨论，

* a). 如果要求找出只含正数的最大子树，程序该如何修改来实现？
* b). 假设我们将子树定义为它和它的部分后代，那该如何解决？
* c). 对于b，加上正数的限制，方案又该如何？

总之，一道看似简单的面试题，可能能变换成各种花样。

比如，面试管可能还会再提两个要求：第一，不能用全局变量；第二，有个参数控制是否要只含正数的子树。

**17**

有一个排序二叉树，数据类型是int型，如何找出中间大的元素。

**18**

中序遍历二叉树，结果为ABCDEFGH，后序遍历结果为ABEDCHGF，那么前序遍历结果为?

**19**

写程序输出8皇后问题的所有排列，要求使用非递归的深度优先遍历。

**20**

在8X8的棋盘上分布着n个骑士，他们想约在某一个格中聚会。骑士每天可以像国际象棋中的马那样移动一次，可以从中间像8个方向移动（当然不能走出棋盘），请计算n个骑士的最早聚会地点和要走多少天。要求尽早聚会，且n个人走的总步数最少，先到聚会地点的骑士可以不再移动等待其他的骑士。

从键盘输入n（0<n<=64），然后一次输入n个骑士的初始位置xi,yi（0<=xi,yi<=7）。屏幕输出以空格分隔的三个数，分别为聚会点（x，y）以及走的天数。

提示：BFS。

**21、城市遍历**

某人家住北京，想去青海玩，可能会经过许多城市，现已知地图上的城市连接，求经过M个城市到达青海的路线种类。城市可以多次到达的，比如去了天津又回到北京，再去天津，即为3次。北京出发不算1次。

输入：

N M S N为城市总数，北京为0，青海为N-1； M为经过的城市数目； S为之后有S行i j 表示第i个城市可以去第j个城市，是有方向的。

输出：N 表示路径种类。

**22**

给定两个站点，如果没有直达的路线，如何找到换乘次数最少的路线？

**23**

有两座桥，其中一座可能是坏的，两个守桥人分别守在这两座桥的入口。他们一个总是会说实话，一个总是说谎话。 你现在需要找出哪一座桥可以通过。

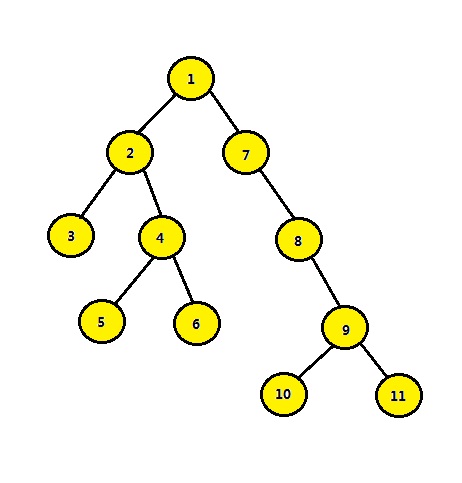
请问最少需要问守桥人几个问题，可以找出可以通过的桥？如何问？

**24**

一类似于蜂窝的结构的图，进行搜索最短路径（要求5分钟）。

**25**

对于一颗完全二叉树，要求给所有节点加上一个pNext指针，指向同一层的相邻节点；如果当前节点已经是该层的最后一个节点，则将pNext指针指向NULL；给出程序实现，并分析时间复杂度和空间复杂度。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.1.jpg)

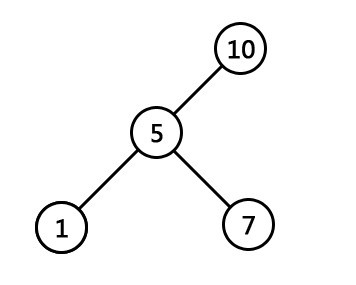
结点3和结点4的最近公共祖先是结点2，即LCA（3 4）=2 。在此，需要注意到当两个结点在同一棵子树上的情况，如结点3和结点2的最近公共祖先为2，即 LCA（3，2）=2。同理：LCA（5，6）=4，LCA（6，10）=1。

明确了题意，咱们便来试着解决这个问题。直观的做法，可能是针对是否为二叉查找树分情况讨论，这也是一般人最先想到的思路。除此之外，还有所谓的Tarjan算法、倍增算法、以及转换为RMQ问题（求某段区间的极值）。后面这几种算法相对高级，不那么直观，但思路比较有启发性，了解一下也有裨益。

### 解法一：暴力对待

#### 1.1、是二叉查找树

在当这棵树是二叉查找树的情况下，如下图：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.2.jpg)

那么从树根开始：

* 如果当前结点t 大于结点u、v，说明u、v都在t 的左侧，所以它们的共同祖先必定在t 的左子树中，故从t 的左子树中继续查找；
* 如果当前结点t 小于结点u、v，说明u、v都在t 的右侧，所以它们的共同祖先必定在t 的右子树中，故从t 的右子树中继续查找；
* 如果当前结点t 满足 u <t < v，说明u和v分居在t 的两侧，故当前结点t 即为最近公共祖先；
* 而如果u是v的祖先，那么返回u的父结点，同理，如果v是u的祖先，那么返回v的父结点。

代码如下所示：

//copyright@eriol 2011

//modified by July 2014

public int query(Node t, Node u, Node v) {

int left = u.value;

int right = v.value;

Node parent = null;

//二叉查找树内，如果左结点大于右结点，不对，交换

if (left > right) {

int temp = left;

left = right;

right = temp;

}

while (true) {

//如果t小于u、v，往t的右子树中查找

if (t.value < left) {

parent = t;

t = t.right;

//如果t大于u、v，往t的左子树中查找

} else if (t.value > right) {

parent = t;

t = t.left;

} else if (t.value == left || t.value == right) {

return parent.value;

} else {

return t.value;

}

}

}

#### 1.2、不是二叉查找树

但如果这棵树不是二叉查找树，只是一棵普通的二叉树呢？如果每个结点都有一个指针指向它的父结点，于是我们可以从任何一个结点出发，得到一个到达树根结点的单向链表。因此这个问题转换为两个单向链表的第一个公共结点。

此外，如果给出根节点，LCA问题可以用递归很快解决。而关于树的问题一般都可以转换为递归（因为树本来就是递归描述），参考代码如下：

//copyright@allantop 2014-1-22-20:01

node\* getLCA(node\* root, node\* node1, node\* node2)

{

if(root == null)

return null;

if(root== node1 || root==node2)

return root;

node\* left = getLCA(root->left, node1, node2);

node\* right = getLCA(root->right, node1, node2);

if(left != null && right != null)

return root;

else if(left != null)

return left;

else if (right != null)

return right;

else

return null;

}

然不论是针对普通的二叉树，还是针对二叉查找树，上面的解法有一个很大的弊端就是：如需N 次查询，则总体复杂度会扩大N 倍，故这种暴力解法仅适合一次查询，不适合多次查询。

接下来的解法，将不再区别对待是否为二叉查找树，而是一致当做是一棵普通的二叉树。总体来说，由于可以把LCA问题看成是询问式的，即给出一系列询问，程序对每一个询问尽快做出反应。故处理这类问题一般有两种解决方法：

* 一种是在线算法，相当于循序渐进处理；
* 另外一种则是离线算法，如Tarjan算法，相当于一次性批量处理，一开始就知道了全部查询，只待询问。

### 解法二：Tarjan算法

如上文末节所述，不论咱们所面对的二叉树是二叉查找树，或不是二叉查找树，都可以把求任意两个结点的最近公共祖先，当做是查询的问题，如果是只求一次，则是单次查询；如果要求多个任意两个结点的最近公共祖先，则相当于是批量查询。

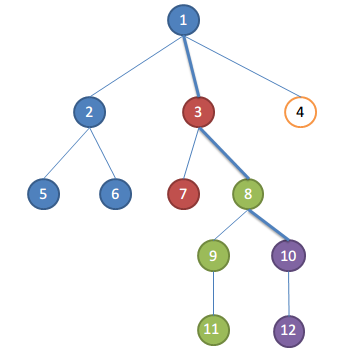
涉及到批量查询的时候，咱们可以借鉴离线处理的方式，这就引出了解决此LCA问题的Tarjan离线算法。

#### 2.1、什么是Tarjan算法

Tarjan算法 （以发现者Robert Tarjan命名）是一个在图中寻找强连通分量的算法。算法的基本思想为：任选一结点开始进行深度优先搜索dfs（若深度优先搜索结束后仍有未访问的结点，则再从中任选一点再次进行）。搜索过程中已访问的结点不再访问。搜索树的若干子树构成了图的强连通分量。

应用到咱们要解决的LCA问题上，则是：对于新搜索到的一个结点u，先创建由u构成的集合，再对u的每颗子树进行搜索，每搜索完一棵子树，这时候子树中所有的结点的最近公共祖先就是u了。

举一个例子，如下图（不同颜色的结点相当于不同的集合）：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.3.jpg)

假设遍历完10的孩子,要处理关于10的请求了，取根节点到当前正在遍历的节点的路径为关键路径,即1-3-8-10，集合的祖先便是关键路径上距离集合最近的点。

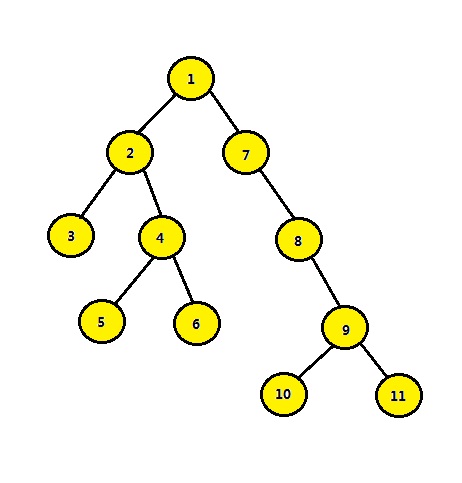
比如：

* 1，2，5，6为一个集合,祖先为1，集合中点和10的LCA为1
* 3，7为一个集合，祖先为3，集合中点和10的LCA为3
* 8，9，11为一个集合，祖先为8，集合中点和10的LCA为8
* 10，12为一个集合，祖先为10，集合中点和10的LCA为10

得出的结论便是：LCA(u,v)便是根至u的路径上到节点v最近的点。

#### 2.2、Tarjan算法如何而来

但关键是 Tarjan算法是怎么想出来的呢？再给定下图，你是否能看出来：分别从结点1的左右子树当中，任取一个结点，设为u、v，这两个任意结点u、v的最近公共祖先都为1。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.4.jpg)

于此，我们可以得知：若两个结点u、v分别分布于某节点t 的左右子树，那么此节点 t即为u和v的最近公共祖先。更进一步，考虑到一个节点自己就是LCA的情况，得知：

* 若某结点t 是两结点u、v的祖先之一，且这两结点并不分布于该结点t 的一棵子树中，而是分别在结点t 的左子树、右子树中，那么该结点t 即为两结点u、v的最近公共祖先。

这个定理就是Tarjan算法的基础。

一如上文1.1节我们得到的结论：“如果当前结点t 满足 u <t < v，说明u和v分居在t 的两侧，故当前结点t 即为最近公共祖先”。

而对于本节开头我们所说的“如果要求多个任意两个结点的最近公共祖先，则相当于是批量查询”，即在很多组的询问的情况下，或许可以先确定一个LCA。例如是根节点1，然后再去检查所有询问，看是否满足刚才的定理，不满足就忽视，满足就赋值，全部弄完，再去假设2号节点是LCA，再去访问一遍。

可此方法需要判断一个结点是在左子树、还是右子树，或是都不在，都只能遍历一棵树，而多次遍历的代价实在是太大了，所以我们需要找到更好的方法。这就引出了下面要阐述的Tarjan算法，即每个结点只遍历一次，怎么做到的呢，请看下文讲解。

### 2.3、Tarjan算法流程

Tarjan算法流程为：

Procedure dfs（u）；

begin

设置u号节点的祖先为u

若u的左子树不为空，dfs（u - 左子树）；

若u的右子树不为空，dfs（u - 右子树）；

访问每一条与u相关的询问u、v

-若v已经被访问过，则输出v当前的祖先t（t即u,v的LCA）

标记u为已经访问，将所有u的孩子包括u本身的祖先改为u的父亲

end

普通的dfs 不能直接解决LCA问题，故Tarjan算法的原理是dfs + [并查集](http://zh.wikipedia.org/zh-cn/%E5%B9%B6%E6%9F%A5%E9%9B%86)，它每次把两个结点对的最近公共祖先的查询保存起来，然后dfs 更新一次。如此，利用并查集优越的时空复杂度，此算法的时间复杂度可以缩小至O(n＋Q)，其中，n为数据规模，Q为询问个数。

### 解法三：转换为RMQ问题

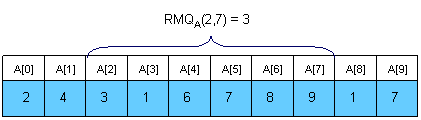
解决此最近公共祖先问题的还有一个算法，即转换为RMQ问题，用Sparse Table（简称ST）算法解决。

### 3.1、什么是RMQ问题

RMQ，全称为Range Minimum Query，顾名思义，则是区间最值查询，它被用来在数组中查找两个指定索引中最小值的位置。即RMQ相当于给定数组A[0, N-1]，找出给定的两个索引如 i、j 间的最小值的位置。

假设一个算法预处理时间为 f(n)，查询时间为g(n)，那么这个算法复杂度的标记为<f(n), g(n)>。我们将用RMQA(i, j) 来表示数组A 中索引i 和 j 之间最小值的位置。 u和v的离树T根结点最远的公共祖先用LCA T(u, v)表示。

如下图所示，RMQA(2,7 )则表示求数组A中从A[2]~A[7]这段区间中的最小值：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.31.jpg)

很显然，从上图中，我们可以看出最小值是A[3] = 1，所以也就不难得出最小值的索引值RMQA(2,7) = 3。

#### 3.2、如何解决RMQ问题

##### 3.2.1、Trivial algorithms for RMQ

下面，我们对对每一对索引(i, j)，将数组中索引i 和 j 之间最小值的位置 RMQA(i, j) 存储在M[0, N-1][0, N-1]表中。 RMQA(i, j) 有不同种计算方法，你会看到，随着计算方法的不同，它的时空复杂度也不同：

* 普通的计算将得到一个 <O(N^3), O(1)> 复杂度的算法。尽管如此，通过使用一个简单的动态规划方法，我们可以将复杂度降低到 <O(N^2), O(1)>。如何做到的呢？方法如下代码所示：

//copyright@

//modified by July 2014

void process1(int M[MAXN][MAXN], int A[MAXN], int N)

{

int i, j;

for (i =0; i < N; i++)

M[i][i] = i;

for (i = 0; i < N; i++)

for (j = i + 1; j < N; j++)

//若前者小于后者，则把后者的索引值付给M[i][j]

if (A[M[i][j - 1]] < A[j])

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

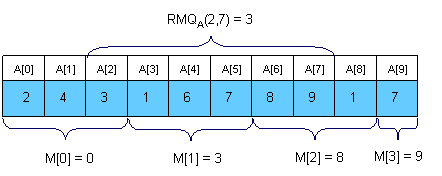
//否则前者的索引值付给M[i][j]

else

M[i][j] = j;

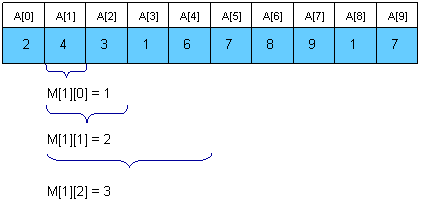
}

* 一个比较有趣的点子是把向量分割成sqrt(N)大小的段。我们将在M[0,sqrt(N)-1]为每一个段保存最小值的位置。如此，M可以很容易的在O(N)时间内预处理。

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.32.jpg)

* 一个更好的方法预处理RMQ 是对2^k 的长度的子数组进行动态规划。我们将使用数组M[0, N-1][0, logN]进行保存，其中M[ i ][ j ] 是以i 开始，长度为 2^j 的子数组的最小值的索引。这就引出了咱们接下来要介绍的Sparse Table (ST) algorithm。

##### 3.2.2、Sparse Table (ST) algorithm

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.33.jpg)

在上图中，我们可以看出：

* 在A[1]这个长度为2^0的区间内，最小值即为A[1] = 4，故最小值的索引M[1][0]为1；
* 在A[1]、A[2] 这个长度为2^1的区间内，最小值为A[2] = 3，故最小值的索引为M[1][1] = 2；
* 在A[1]、A[2]、A[3]、A[4]这个长度为2^2的区间内，最小值为A[3] = 1，故最小值的索引M[1][2] = 3。

为了计算M[i][j]我们必须找到前半段区间和后半段区间的最小值。很明显小的片段有着2^(j-1)长度，因此递归如下

[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/39/39.34.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.34.jpg)

根据上述公式，可以写出这个预处理的递归代码，如下：

void process2(int M[MAXN][LOGMAXN], int A[MAXN], int N)

{

int i, j;

//initialize M for the intervals with length 1

for (i = 0; i < N; i++)

M[i][0] = i;

//compute values from smaller to bigger intervals

for (j = 1; 1 << j <= N; j++)

for (i = 0; i + (1 << j) - 1 < N; i++)

if (A[M[i][j - 1]] < A[M[i + (1 << (j - 1))][j - 1]])

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

else

M[i][j] = M[i + (1 << (j - 1))][j - 1];

}

经过这个O(N logN)时间复杂度的预处理之后，让我们看看怎样使用它们去计算 RMQA(i, j)。思路是选择两个能够完全覆盖区间[i..j]的块并且找到它们之间的最小值。设k = [log(j - i + 1)]。

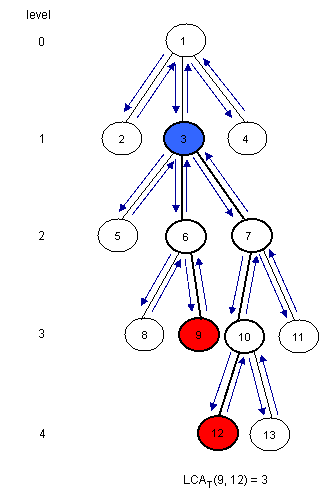
为了计算 RMQA(i, j)，我们可以使用下面的公式：

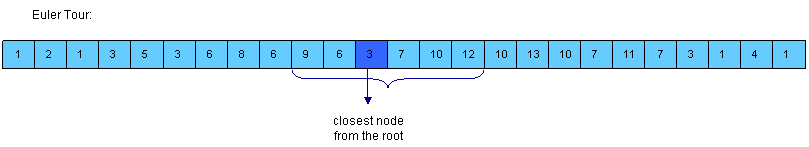
[https://github.com/julycoding/The-Art-Of-Programming-By-July/raw/master/ebook/images/39/39.35.jpg](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.35.jpg)

故，综合来看，咱们预处理的时间复杂度从O(N3)降低到了O(N logN)，查询的时间复杂度为O(1)，所以最终的整体复杂度为：<O(N logN), O(1)>。

#### 3.3、LCA与RMQ的关联性

现在，让我们看看怎样用RMQ来计算LCA查询。事实上，我们可以在线性时间里将LCA问题规约到RMQ问题，因此每一个解决RMQ的问题都可以解决LCA问题。让我们通过例子来说明怎么规约的：

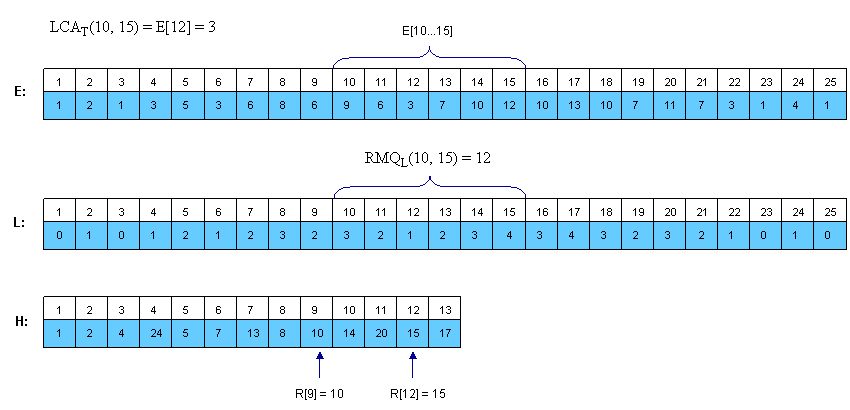
[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.37.jpg)

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.38.jpg)

注意LCAT(u, v)是在对T进行dfs过程当中在访问u和v之间离根结点最近的点。因此我们可以考虑树的欧拉环游过程u和v之间所有的结点，并找到它们之间处于最低层的结点。为了达到这个目的，我们可以建立三个数组：

* E[1, 2\*N-1] - 对T进行欧拉环游过程中所有访问到的结点;E[i]是在环游过程中第i个访问的结点
* L[1,2\*N-1] - 欧拉环游中访问到的结点所处的层数;L[i]是E[i]所在的层数
* H[1, N] - H[i] 是E中结点i第一次出现的下标(任何出现i的地方都行，当然选第一个不会错)

假定H[u]<H[v](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/%E5%90%A6%E5%88%99%E4%BD%A0%E8%A6%81%E4%BA%A4%E6%8D%A2u%E5%92%8Cv)。可以很容易的看到u和v第一次出现的结点是E[H[u]..H[v]]。现在，我们需要找到这些结点中的最低层。为了达到这个目的，我们可以使用RMQ。因此 LCAT(u, v) = E[RMQL(H[u], H[v])] ,RMQ返回的是索引，下面是E,L,H数组：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.39.jpg)

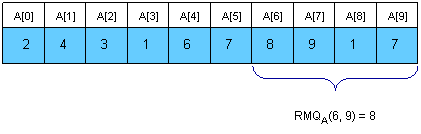
注意L中连续的元素相差为1。

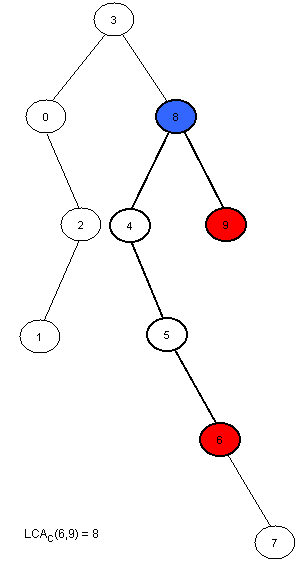
### 3.4、从RMQ到LCA

我们已经看到了LCA问题可以在线性时间规约到RMQ问题。现在让我们来看看怎样把RMQ问题规约到LCA。这个意味着我们实际上可以把一般的RMQ问题规约到带约束的RMQ问题(这里相邻的元素相差1)。为了达到这个目的，我们需要使用笛卡尔树。

对于数组A[0,N-1]的笛卡尔树C(A)是一个二叉树，根节点是A的最小元素，假设i为A数组中最小元素的位置。当i>0时，这个笛卡尔树的左子结点是A[0,i-1]构成的笛卡尔树，其他情况没有左子结点。右结点类似的用A[i+1,N-1]定义。注意对于具有相同元素的数组A，笛卡尔树并不唯一。在本文中，将会使用第一次出现的最小值，因此笛卡尔树看作唯一。可以很容易的看到RMQA(i, j) = LCAC(i, j)。

下面是一个例子：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.40.jpg)

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.41.jpg)

现在我们需要做的仅仅是用线性时间计算C(A)。这个可以使用栈来实现。

* 初始栈为空。
* 然后我们在栈中插入A的元素。
* 在第i步,A[i]将会紧挨着栈中比A[i]小或者相等的元素插入,并且所有较大的元素将会被移除。
* 在插入结束之前栈中A[i]位置前的元素将成为i的左儿子，A[i]将会成为它之后一个较小元素的右儿子。

在每一步中，栈中的第一个元素总是笛卡尔树的根。

如果使用栈来保存元素的索引而不是值，我们可以很轻松的建立树。由于A中的每个元素最多被增加一次和最多被移除一次，所以建树的时间复杂度为O(N)。最终查询的时间复杂度为O(1)，故综上可得，咱们整个问题的最终时间复杂度为：<O(N), O(1)>。

现在，对于询问 RMQA(i, j) 我们有两种情况:

* i和j在同一个块中,因此我们使用在P和T中计算的值
* i和j在不同的块中，因此我们计算三个值:从i到i所在块的末尾的P和T中的最小值，所有i和j中块中的通过与处理得到的最小值以及从j所在块i和j在同一个块中,因此我们使用在P和T中计算的值j的P和T的最小值；最后我们我们只要计算三个值中最小值的位置即可。

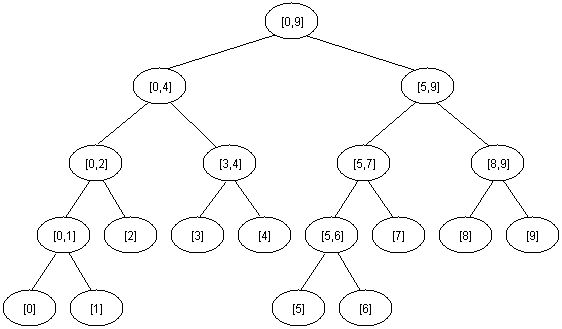
RMQ和LCA是密切相关的问题，因为它们之间可以相互规约。有许多算法可以用来解决它们，并且他们适应于一类问题。

### 解法四：线段树

解决RMQ问题也可以用所谓的线段树Segment trees。线段树是一个类似堆的数据结构，可以在基于区间数组上用对数时间进行更新和查询操作。我们用下面递归方式来定义线段树的[i, j]区间：

* 第一个结点将保存区间[i, j]区间的信息
* 如果i<j 左右的孩子结点将保存区间[i, (i+j)/2]和[(i+j)/2+1, j] 的信息

注意具有N个区间元素的线段树的高度为[logN] + 1。下面是区间[0,9]的线段树：

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/images/39/39.36.jpg)

线段树和堆具有相同的结构，因此我们定义x是一个非叶结点，那么左孩子结点为2x,而右孩子结点为2x+1。想要使用线段树解决RMQ问题，我们则要要使用数组 M[1, 2 \* 2[logN] + 1]，这里M[i]保存结点i区间最小值的位置。初始时M的所有元素为-1。树应当用下面的函数进行初始化(b和e是当前区间的范围)：

void initialize(int node, int b, int e, int M[MAXIND], int A[MAXN], int N)

{

if (b == e)

M[node] = b;

else

{

//compute the values in the left and right subtrees

initialize(2 \* node, b, (b + e) / 2, M, A, N);

initialize(2 \* node + 1, (b + e) / 2 + 1, e, M, A, N);

//search for the minimum value in the first and

//second half of the interval

if (A[M[2 \* node]] <= A[M[2 \* node + 1]])

M[node] = M[2 \* node];

else

M[node] = M[2 \* node + 1];

}

}

上面的函数映射出了这棵树建造的方式。当计算一些区间的最小值位置时，我们应当首先查看子结点的值。调用函数的时候使用 node = 1, b = 0和e = N-1。

现在我们可以开始进行查询了。如果我们想要查找区间[i, j]中的最小值的位置时，我们可以使用下一个简单的函数：

int query(int node, int b, int e, int M[MAXIND], int A[MAXN], int i, int j)

{

int p1, p2;

//if the current interval doesn't intersect

//the query interval return -1

if (i > e || j < b)

return -1;

//if the current interval is included in

//the query interval return M[node]

if (b >= i && e <= j)

return M[node];

//compute the minimum position in the

//left and right part of the interval

p1 = query(2 \* node, b, (b + e) / 2, M, A, i, j);

p2 = query(2 \* node + 1, (b + e) / 2 + 1, e, M, A, i, j);

//return the position where the overall

//minimum is

if (p1 == -1)

return M[node] = p2;

if (p2 == -1)

return M[node] = p1;

if (A[p1] <= A[p2])

return M[node] = p1;

return M[node] = p2;

}

你应该使用node = 1, b = 0和e = N - 1来调用这个函数，因为分配给第一个结点的区间是[0, N-1]。

可以很容易的看出任何查询都可以在O(log N)内完成。注意当我们碰到完整的in/out区间时我们停止了，因此数中的路径最多分裂一次。用线段树我们获得了<O(N),O(logN)>的算法

线段树非常强大，不仅仅是因为它能够用在RMQ上，还因为它是一个非常灵活的数据结构，它能够解决动态版本的RMQ问题和大量的区间搜索问题。

### 其余解法

除此之外，还有倍增法、重链剖分算法和后序遍历也可以解决该问题。其中，倍增思路相当于层序遍历，逐层或几层跳跃查，查询时间复杂度为O(log n)，空间复杂度为nlogn，对于每个节点先存储向上1层2层4层的节点，每个点有depth信息。