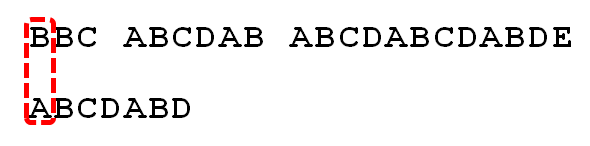
# 常用算法分析

## KMP算法：

https://www.cnblogs.com/c-cloud/p/3224788.html

许多算法可以完成这个任务，[Knuth-Morris-Pratt算法](http://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm)（简称KMP）是最常用的之一。它以三个发明者命名，起头的那个K就是著名科学家Donald Knuth。



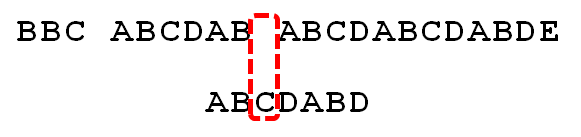
KMP算法的想法是，设法利用这个已知信息，不要把"搜索位置"移回已经比较过的位置，继续把它向后移，这样就提高了效率。

移动位数 = 已匹配的字符数 - 对应的部分匹配值

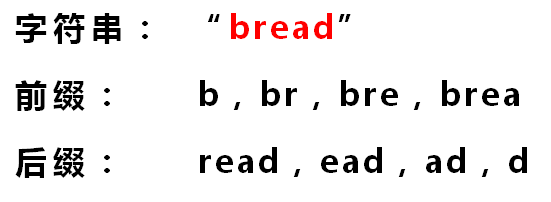


已知空格与D不匹配时，前面六个字符"ABCDAB"是匹配的。查表可知，最后一个匹配字符B对应的"部分匹配值"为2，因此按照下面的公式算出向后移动的位数：

因为 6 - 2 等于4，所以将搜索词向后移动4位。









"部分匹配值"就是"前缀"和"后缀"的最长的共有元素的长度。以"ABCDABD"为例，

　　－　"A"的前缀和后缀都为空集，共有元素的长度为0；

　　－　"AB"的前缀为[A]，后缀为[B]，共有元素的长度为0；

　　－　"ABC"的前缀为[A, AB]，后缀为[BC, C]，共有元素的长度0；

　　－　"ABCD"的前缀为[A, AB, ABC]，后缀为[BCD, CD, D]，共有元素的长度为0；

　　－　"ABCDA"的前缀为[A, AB, ABC, ABCD]，后缀为[BCDA, CDA, DA, A]，共有元素为"A"，长度为1；

　　－　"ABCDAB"的前缀为[A, AB, ABC, ABCD, ABCDA]，后缀为[BCDAB, CDAB, DAB, AB, B]，共有元素为"AB"，长度为2；

　　－　"ABCDABD"的前缀为[A, AB, ABC, ABCD, ABCDA, ABCDAB]，后缀为[BCDABD, CDABD, DABD, ABD, BD, D]，共有元素的长度为0。

"部分匹配"的实质是，有时候，字符串头部和尾部会有重复。比如，"ABCDAB"之中有两个"AB"，那么它的"部分匹配值"就是2（"AB"的长度）。搜索词移动的时候，第一个"AB"向后移动4位（字符串长度-部分匹配值），就可以来到第二个"AB"的位置。

**nextval**数组的求解方法是：nextval[1]=0。从第二位开始，若要求nextval[i]，将next[i]的值对应的位的值与i的值进行比较（例如，第i为的值为'b'，next[i]=3,则将i的值'b'与第三位的值进行比较），若相等，nextval[i]=nextval【next[i]】（例，nextval[i]=nextval[3]）；若不相等，则nextval[i]=next[i]（例，nextval[i]=next[i]=3）。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 模式串 | a | b | a | a | b | c | a | c |
| next值 | 0 | 1 | 1 | 2 | 2 | 3 | 1 | 2 |
| nextval值 | 0 | 1 | 0 | 2 | 1 | 3 | 0 | 2 |

1.第一位的nextval值必定为0，第二位如果与第一位相同则为0，如果不同则为1。

 2.第三位的next值为1，那么将第三位和第一位进行比较，均为a，相同，则，第三位的nextval值为0。

3.第四位的next值为2，那么将第四位和第二位进行比较，不同，则第四位的nextval值为其next值，为2。

4.第五位的next值为2，那么将第五位和第二位进行比较，相同，第二位的next值为1，则继续将第二位与第一位进行比较，不同，则第五位的nextval值为第二位的next值，为1。

5.第六位的next值为3，那么将第六位和第三位进行比较，不同，则第六位的nextval值为其next值，为3。

6.第七位的next值为1，那么将第七位和第一位进行比较，相同，则第七位的nextval值为0。

7.第八位的next值为2，那么将第八位和第二位进行比较，不同，则第八位的nextval值为其next值，为2。

可在“aaaab”内进行验证：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 模式串 | a | a | a | a | b |
| next值 | 0 | 1 | 2 | 3 | 4 |
| nextval值 | 0 | 0 | 0 | 0 | 4 |

## 稀疏矩阵

1：稀疏矩阵的背景

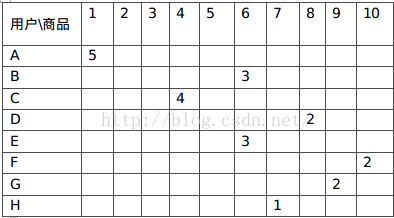
2：什么是稀疏矩阵？

3：为什么要对稀疏矩阵进行压缩存储以及压缩存储的方式？

4：稀疏矩阵的相关运算

**一：背景**

        第一此介绍稀疏矩阵是在数据结构学习时，然后当时并没有多么用心的去学习它，因为，感觉它在实际应用中很少遇见，直到后来自己看了基于用户的协同过滤推荐算法时，才有了较大的感触，在协同过滤中稀疏矩阵产生的背景是，例如下表是某宝N个用户对购买商品的评分，因为某宝的商品特别多，所以各个用户之间的交集就小了，此时便产生了稀疏矩阵



        那么下面我们针对稀疏矩阵做以下总结和讨论

**二：什么是稀疏矩阵？**

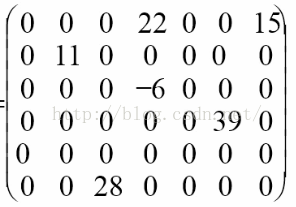
        数值为0的元素数目远远多于非0元素的数目，并且非零元素的分布没有规律的矩阵称为稀疏矩阵（sparse），

        其实往往对于稀疏矩阵的定义并没有明确的规则或者标准，更大程度上是根据人的经验准则来进行判断的。

**三：为什么要对稀疏矩阵进行压缩存储以及主要的压缩存储的方式？**

        由于稀疏矩阵中存在大量的“空”值，占据了大量的存储空间，而真正有用的数据却少之又少，且在计算时浪费资源，所以要进行压缩存储以节省存储空间和计算方便。

        拿下面这个图来举例



        这里我们首先采用三元组表示方法来表示稀疏矩阵，例如上边的稀疏矩阵可以表示为：

        (  (1,4,22),(1,7,15),(2,2,11),(3,4,-6),(4,6,39),(6,3,28)  )

       接下来我们讨论存储方式

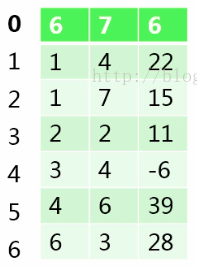
**1：顺序存储**

             若把稀疏矩阵的三元组线性表按顺序存储结构存储，则称为稀疏矩阵的三元组顺序表。 　　     顺序表中除了存储三元组外，还应该存储矩阵行数、列数和总的非零元素数目，这样才能唯一的确定一个矩阵。

             (1)用一个二维数组A[0..m,1..3]:Integer

             (2)存储方法：a[0,1]——总行数，a[0,2]——总列数，a[0,3]——存放非零元素个数

             (3)按行存放：每个非零元素所在行，列数以及值



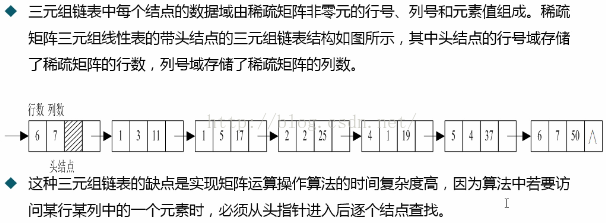
         顺序存储的缺点：

         与用二维数组存储稀疏矩阵比较，用三元组表表示的稀疏矩阵不仅节约了空间，而且使得矩阵某些运算的时间比经典算法还少，但是在进行矩阵加法，减法和乘法等运算时，有时矩阵中的非零元素的位置和个数会发生很大的变化，如A = A+ B，将矩阵B加到矩阵A上，此时若还用三元组顺序表，势必会为了保持三元组表 “ 以行序为主序”而移动大量的元素

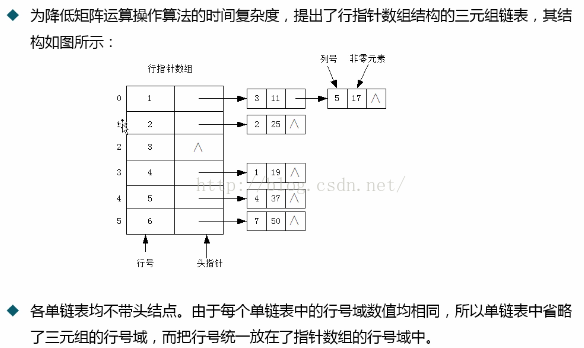
**2：链式存储（即稀疏矩阵的三元链表）**

       链式存储又可以分为三类：

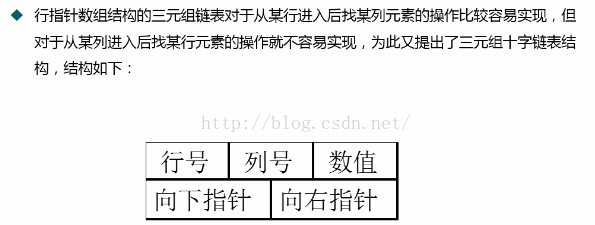
**(1)三元组链表：用链表存储的三元线性表**



**(2)  行指针数组结果的三元组链表：把每行非零元素三元组组织乘一个单链表，再设计一个指针类型的数组存储所有单链表的头指针**

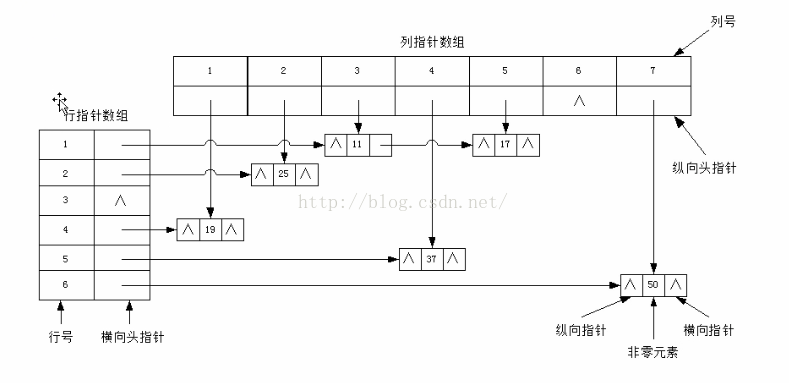


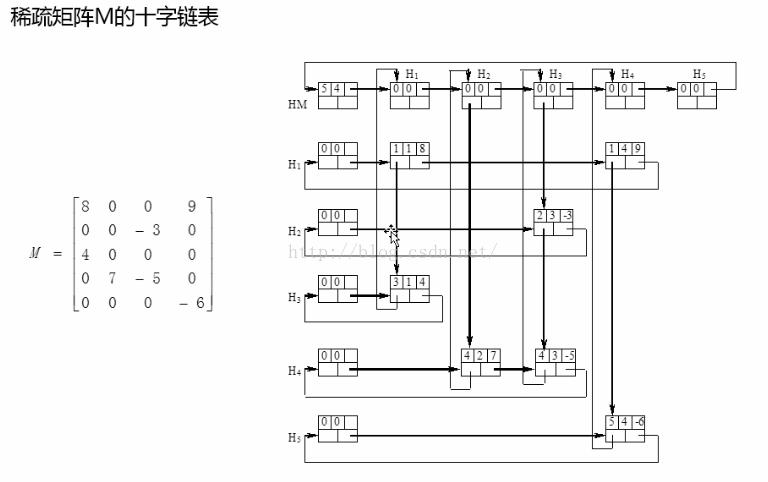
**(3)三元组十字链表：用的最多的形式，把非零元素三元组按行和按列组织乘单链表，这样稀疏矩阵的每个非零元素三元组节点都将即勾链在行单链表上，又都勾链在列单链表上，形成十字链表。**



                                 表结点， 行头结点和列结点，总表头结点







**3：两种存储方式的比较**

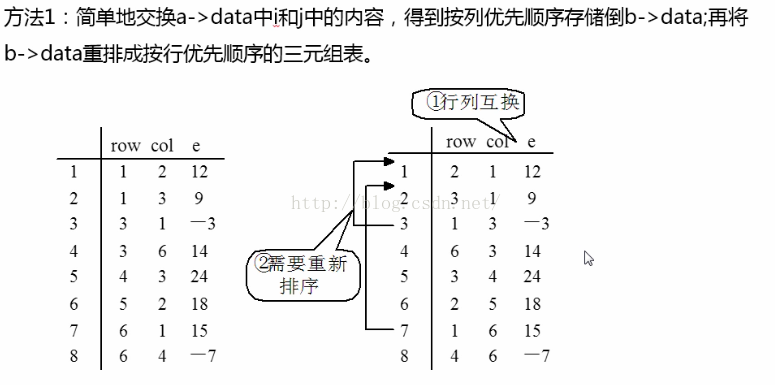
             三元组顺序表：非零元素在表中按行序有序存储，因此便于进行依行顺序处理的矩阵运算，但是，若需按行号存取某一行的非零元素，则需从头开始进行查找。（时间复杂度高）

             行逻辑连接的顺序表：便于随机存取任意一行的非零元素

             十字链表：当家族很的非零元素个数和位置操作过程中变化较大时，就不适宜采用顺序存储结构来表示三元组的线性表

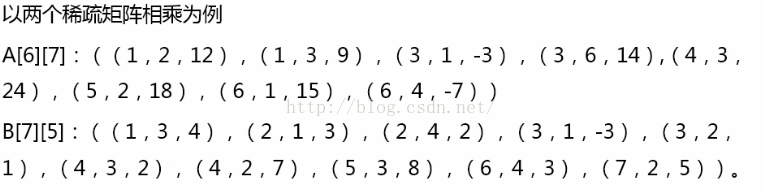
**四：稀疏矩阵的相关运算**

**1：转置**





**2：基于顺序表存储的稀疏矩阵乘法的实现**

前提条件是：前者矩阵的列和后者矩阵的行数目相同，即m\*n  n\*p，如下两个矩阵，进行矩阵相乘

其遵循的主要规则是：

                                                C[ i ][ j ] = sum（A[ i ][ k ] \* B[ k ][ j ]） (k从1到n)

每一次从A矩阵的第一个开始进行遍历（1，2，12）：

在B中发现（2，1，3），可以计算得到C[ 1 ][ 1 ] = 36

B中继续后移发现（2，4，2），可以计算得到C[ 1 ][ 4 ]=24

知道B遍历完毕

从A中第二个（1，3，9）开始遍历：

在B中发现（3，1，-3），可以计算得到C[ 1 ][ 1 ] = -27，由于上一次遍历以及得到C[ 1 ][ 1 ]=36 ，所以两者相加，结果为C[ 1 ][ 1 ]=9

B中继续后移发现（3，2，1），可以计算得到C[ 1 ][ 2 ] = 9，由于之前遍历没有得到这个结果，所以不用相加，继续遍历

.......

直到所有遍历结束

得到的计算结果为：

C[ 1 ][ 1 ] = 9，C[ 1 ][ 2 ] = 9，C[ 3 ][ 3 ] = -12，C[ 3 ][ 4 ] = 42，C[ 4 ][ 1 ] = -42，C[ 4 ][ 2 ] = 24，

C[ 5 ][ 1 ] = 54，C[ 5 ][ 4 ] = 36，C[ 6 ][ 2 ] = -14，C[ 6 ][ 3 ]=46

其三元组表示为：

（（1，1，9），（1，2，9），（3，3，-12），（3，4，42），（4，1，-42），（4，2，24），（5，1，54），（5，4，36），（6，2，-14），（6，3，46））

**3：基于两个十字链表存储的稀疏矩阵的加法**

该部分参考：[点击查看](http://c.biancheng.net/cpp/html/970.html)

首先矩阵A和B满足矩阵相加的条件即两者的行列数相同

已知两个稀疏矩阵A 和B，分别采用十字链表存储，计算C=A+B，C 也采用十字链表方式存储，并且在A 的基础上形成C。  由矩阵的加法规则知，只有A 和B 行列对应相等，二者才能相加。C 中的非零元素cij 只可能有３种情况：或者是aij+bij，或者是aij (bij=0)，或者是bij (aij=0)，因此当B 加到A 上时，对A 十字链表的当前结点来说，对应下列四种情况：或者改变结点的值（aij+bij≠０），或者不变（bij＝０），或者插入一个新结点（aij＝０），还可能是删除一个结点（aij+bij＝０）。整个运算从矩阵的第一行起逐行进行。对每一行都从行表的头结点出发，分别找到A 和B 在该行中的第一个非零元素结点后开始比较，然后按４种不同情况分别处理。

设pa和pb 分别指向A 和B 的十字链表中行号相同的两个结点，４种情况如下：  (1) 若pa->col=pb->col 且pa->v+pb->v≠0，则只要用aij+bij 的值改写pa 所指结点的值域即可。  (2) 若pa->col=pb->col 且pa->v+pb->v=0，则需要在矩阵A 的十字链表中删除pa 所指结点，此时需改变该行链表中前趋结点的right 域，以及该列链表中前趋结点的down 域。  (3) 若pa->col < pb->col 且pa->col≠0（即不是表头结点），则只需要将pa 指针向右推进一步，并继续进行比较。  (4) 若pa->col > pb->col 或pa->col＝0（即是表头结点），则需要在矩阵A 的十字链表中插入一个pb 所指结点。

N NP NPC 复杂度

计算机复杂度并不是表示一个程序解决问题需要花多少时间，而是当问题规模扩大后，程序需要的时间长度增长得有多快。

具有o(1)的时间复杂度，也就是常数复杂度，数据规模变得有多大，花的时间也更着变得有多长，这个时间复杂度就是o（n）。

甚至O(n!)的阶乘级复杂度。不会存在O(2\*n^2)的复杂度，因为前面的那个“2”是系数，根本不会影响到整个程序的时间增长。同样地，O (n^3+n^2)的复杂度也就是O(n^3)的复杂度。

前面的几类复杂度被分为两种级别，其中后者的复杂度无论如何都远远大于前者：一种是O(1),O(log(n)),O(n^a)等，我们把它叫做多项式级的复杂度，因为它的规模n出现在底数的位置

另一种是O(a^n)和O(n!)型复杂度，它是非多项式级的，其复杂度计算机往往不能承受。

## 分治算法

分治算法设计思想是：将一个难以解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

分支策略：对于一个规模为n的问题，若该问题可以容易得到解决（比如说规模较小n）则直接解决，否者将其分解为规模较小的字问题，这些子问题互相独立且与原问题形式相同，递归这些字问题，让后将这些子问题解的原问题的解。

使用情况：

1该问题的规模缩小到一定的程度可以容易地解决

2该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构。

3 利用该问题的分解出的子问题可以合并为该问题的解

4该问题的所分解的各个子问题是相互独立的，即子问题之间不包含公共的子问题

# 009-矩阵乘法-分治法-《算法设计技巧与分析》M.H.A学习笔记

A、B是两个n\*n的矩阵，计算C=A\*B。

### 传统算法：

按照下面公式计算，需要n3次乘法和n3-n2次加法，时间复杂度为Θ(n3)。

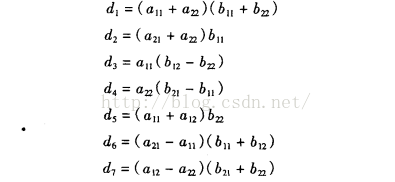
http://img.blog.csdn.net/20160627165702159?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

### 递归算法：

假定n为2的幂，将A、B、C分成4个大小为(n/2)\*(n/2)的子矩阵。

http://img.blog.csdn.net/20160627165706353?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center  
用分治法来计算C。  
http://img.blog.csdn.net/20160627165710784?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center  
需要8次(n/2)\*(n/2)矩阵的乘法和4次(n/2)\*(n/2)矩阵的加法，其中乘法是原来的1/8倍消费，加法是原来的1/4倍耗费。用m表示n=1是乘法的耗费，用a表示加法的耗费。  
于是有了下面的递推式：  
http://img.blog.csdn.net/20160627165715494?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center  
可以推出：  
http://img.blog.csdn.net/20160627165719628?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center  
同样需要n3次乘法和n3-n2次加法，与传统方法相比，时间复杂度没有改进，反而还增加了递归带来的管理开销。

### Strassen算法：

复杂度为o(n3)，运行时间渐进少于n3。  
像递归方法一样划分矩阵，但在计算C的时候有一些不同。  
首先计算出一些中间值：  
  
再由这些中间值得出C：  
http://img.blog.csdn.net/20160627165729322?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center  
Strassen算法进行了18次加法和7次乘法。对于运行时间有如下的递推式：  
http://img.blog.csdn.net/20160627165733681?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center  
经过计算可得，运行时间为Θ(nlog7)=O(n2.81)。

### 三个算法的比较：

