

```

430  FOR K%=0 TO 256
440  `
450      XX[I%] = XX[I%] + REX[K%] * COS(2 * PI * K% * I% / N%)
460      XX[I%] = XX[I%] + IMX[K%] * SIN(2 * PI * K% * I% / N%)
470  `
480  NEXT I%
490 NEXT K%

```

#### 四、 分解运算方法（DFT）

有三种完全不同的方法进行 DFT：一种方法是通过联立方程进行求解，从代数的角度看，要从  $N$  个已知值求  $N$  个未知值，需要  $N$  个联立方程，且  $N$  个联立方程必须是线性独立的，但这是这种方法计算量非常的大且极其复杂，所以很少被采用；第二种方法是利用信号的相关性（correlation）进行计算，这个是我们后面将要介绍的方法；第三种方法是快速傅立叶变换（FFT），这是一个非常具有创造性和革命性的方法，因为它大大提高了运算速度，使得傅立叶变换能够在计算机中被广泛应用，但这种算法是根据复数形式的傅立叶变换来实现的，它把  $N$  个点的信号分解成长度为  $N$  的频域，这个跟我们现在所进行的实域 DFT 变换不一样，而且这种方法也较难理解，这里我们先不去理解，等先理解了复数 DFT 后，再来看一下 FFT。有一点很重要，那就是这三种方法所得的变换结果是一样的，经过实践证明，当频域长度为 32 时，利用相关性方法进行计算效率最好，否则 FFT 算法效率较高。现在就让我们来看一下相关性算法。

利用第一种方法、信号的相关性(correlation)可以从噪声背景中检测出已知的信号，我们也可以利用这个方法检测信号波中是否含有某个频率的信号波：把一个待检测信号波乘以另一个信号波，得到一个新的信号波，再把这个新的信号波所有的点进行相加，从相加的结果就可以判断出这两个信号的相似程度。如下图：

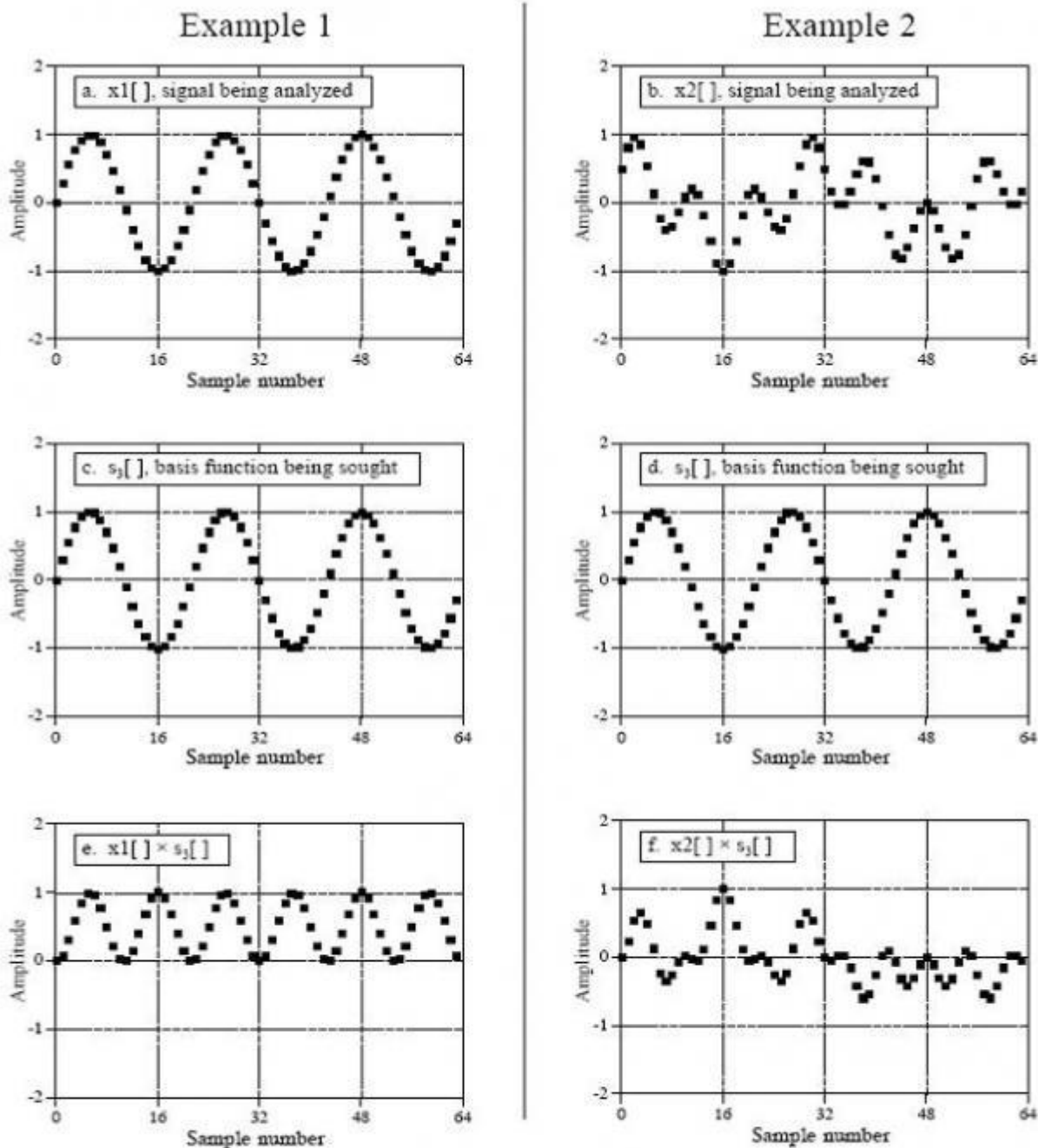


FIGURE 8-8  
 Two example signals, (a) and (b), are analyzed for containing the specific basis function shown in (c) and (d). Figures (e) and (f) show the result of multiplying each example signal by the basis function. Figure (e) has an average of 0.5, indicating that  $x1[]$  contains the basis function with an amplitude of 1.0. Conversely, (f) has a zero average, indicating that  $x2[]$  does not contain the basis function.

上面 a 和 b 两个图是待检测信号波, 图 a 很明显可以看出是个 3 个周期的正弦信号波, 图 b 的信号波则看不出是否含有正弦或余弦信号, 图 c 和 d 都是个 3 个周期的正弦信号波, 图 e 和 f 分别是 a、b 两图跟 c、d 两图相乘后的结果, 图 e 所有点的平均值是 0.5, 说明信号 a 含有振幅为 1 的正弦信号 c, 但图 f 所有点的平均值是 0, 则说明信号 b 不含有信号 d。这个就是通过信号相关性来检测是否含有某个信号的方法。

**第二种方法：**相应地，我也可以通过把输入信号和每一种频率的正余弦信号进行相乘（**关联操作**），从而得到原始信号与每种频率的关联程度（即总和大小），这个结果便是我们所要的傅立叶变换结果，下面两个等式便是我们所要的计算方法：

$$ReX[k] = \sum_{i=0}^{N-1} x[i] \cos(2\pi k i / N)$$

$$ImX[k] = - \sum_{i=0}^{N-1} x[i] \sin(2\pi k i / N)$$

第二个式子中加了个负号，是为了保持复数形式的一致，[前面我们知道在计算  \$Im \bar{X}\[k\]\$  时又加了个负号](#)，所以这只是个形式的问题，并没有实际意义，你也可以把负号去掉，并在计算  $Im \bar{X}[k]$  时也不加负号。

这里有一点必须明白一个正交的概念：两个函数相乘，如果结果中的每个点的总和为 0，则可认为这两个函数为正交函数。要确保关联性算法是正确的，则必须使得跟原始信号相乘的信号的函数形式是正交的，我们知道所有的正弦或余弦函数是正交的，这一点我们可以通过简单的高数知识就可以证明它，所以我们可以通过关联的方法把原始信号分离出正余弦信号。当然，其它的正交函数也是存在的，如：方波、三角波等形式的脉冲信号，所以原始信号也可被分解成这些信号，但这只是说可以这样做，却是没有用的。

下面是实域傅立叶变换的 BASIC 语言代码：

```

100 'THE DISCRETE FOURIER TRANSFORM
110 'The frequency domain signals, held in REX[ ] and IMX[ ], are calculated from
120 'the time domain signal, held in XX[ ].
130 '
140 DIM XX[511]           'XX[ ] holds the time domain signal
150 DIM REX[256]          'REX[ ] holds the real part of the frequency domain
160 DIM IMX[256]          'IMX[ ] holds the imaginary part of the frequency domain
170 '
180 PI = 3.14159265       'Set the constant, PI
190 N% = 512              'N% is the number of points in XX[ ]
200 '
210 GOSUB XXXX            'Mythical subroutine to load data into XX[ ]
220 '
230 '
240 FOR K% = 0 TO 256     'Zero REX[ ] & IMX[ ] so they can be used as accumulators
250   REX[K%] = 0
260   IMX[K%] = 0
270 NEXT K%
280 '
290 '                     'Correlate XX[ ] with the cosine and sine waves, Eq. 8-4
300 '
310 FOR K% = 0 TO 256     'K% loops through each sample in REX[ ] and IMX[ ]
320   FOR I% = 0 TO 511   'I% loops through each sample in XX[ ]
330     '
340     REX[K%] = REX[K%] + XX[I%] * COS(2*PI*K%*I%/N%)
350     IMX[K%] = IMX[K%] - XX[I%] * SIN(2*PI*K%*I%/N%)
360     '
370   NEXT I%
380 NEXT K%
390 '
400 END

```

到此为止，我们对傅立叶变换便有了感性的认识了吧。但要记住，这只是在实域上的离散傅立叶变换，其中虽然也用到了复数的形式，但那只是个替代的形式，并无实际意义，现实中一般使用的是复数形式的离散傅立叶变换，且**快速傅立叶变换**是根据复数离散傅立叶变换来设计算法的，在后面我们先来复习一下有关复数的内容，然后再在理解实域离散傅立叶变换的基础上理解复数形式的离散傅立叶变换。

**更多见下文：**[十、从头到尾彻底理解傅里叶变换算法、下](#)（July、dznlong）

本人 **July** 对本博客所有任何文章、内容和资料享有版权。

转载务必注明作者本人及出处，并通知本人。二零一一年二月二十一日。

# 十、从头到尾彻底理解傅里叶变换算法、下

作者: July、dznlong 二零一一年二月二十二日

推荐阅读: *The Scientist and Engineer's Guide to Digital Signal Processing*,  
By Steven W. Smith, Ph.D. 此书地址: <http://www.dspguide.com/pdfbook.htm>。

---

从头到尾彻底理解傅里叶变换算法、上

前言

第一部分、 DFT

第一章、傅立叶变换的由来

第二章、实数形式离散傅立叶变换 (Real DFT)

从头到尾彻底理解傅里叶变换算法、下

第三章、复数

第四章、复数形式离散傅立叶变换

前期回顾, 在上一篇: [十、从头到尾彻底理解傅里叶变换算法、上](#)里, 我们讲了傅立叶变换的由来、和实数形式离散傅立叶变换 (Real DFT) 俩个问题, 本文接上文, 着重讲下复数、和复数形式离散傅立叶变换等俩个问题。

## 第三章、复数

复数扩展了我们一般所能理解的数的概念, 复数包含了实数和虚数两部分, 利用复数的形式可以把由两个变量表示的表达式变成由一个变量(复变量)来表达, 使得处理起来更加自然和方便。

我们知道傅立叶变换的结果是由两部分组成的, 使用复数形式可以缩短变换表达式, 使得我们可以单独处理一个变量 (这个在后面的描述中我们就可以更加确切地知道), 而且快速傅立叶变换正是基于复数形式的, 所以几乎所有描述的傅立叶变换形式都是复数的形式。

但是复数的概念超过了我们日常生活中所能理解的概念, 要理解复数是较难的, 所以我们在理解复数傅立叶变换之前, 先来专门复习一下有关复数的知识, 这对后面的理解非常重要。

### 一、复数的提出

在此, 先让我们看一个物理实验: 把一个球从某点向上抛出, 然后根据初速度和时间来计算球所在高度, 这个方法可以根据下面的式子计算得出:

$$h = \frac{-gt^2}{2} + vt$$

其中  $h$  表示高度， $g$  表示重力加速度( $9.8\text{m/s}^2$ )， $v$  表示初速度， $t$  表示时间。现在反过来，假如知道了高度，要求计算到这个高度所需要的时间，这时我们又可以通过下式来计算：

$$t = 1 \pm \sqrt{1 - h/4.9}$$

(多谢 [JERRY\\_PRI](#) 提出：

1、根据公式  $h = -(gt^2/2) + vt$  ( $gt$  后面的 2 表示  $t$  的平方)，我们可以讨论最终情况，也就是说小球运动到最高点时， $v = gt$ ，所以，可以得到  $t = \sqrt{2h/g}$  且在您给的公式中，根号下为  $1 - (2h)/g$ ，化成分数形式为  $(g - 2h)/g$ ， $g$  和  $h$  不能直接做加减运算。

2、 $g$  是重力加速度，单位是  $\text{m/s}^2$ ， $h$  的单位是  $\text{m}$ ，他们两个相减的话在物理上没有意义，而且使用您给的那个公式反向回去的话推出的是  $h = -(gt^2/2) + gt$  啊 ( $gt$  后面的 2 表示  $t$  的平方)。

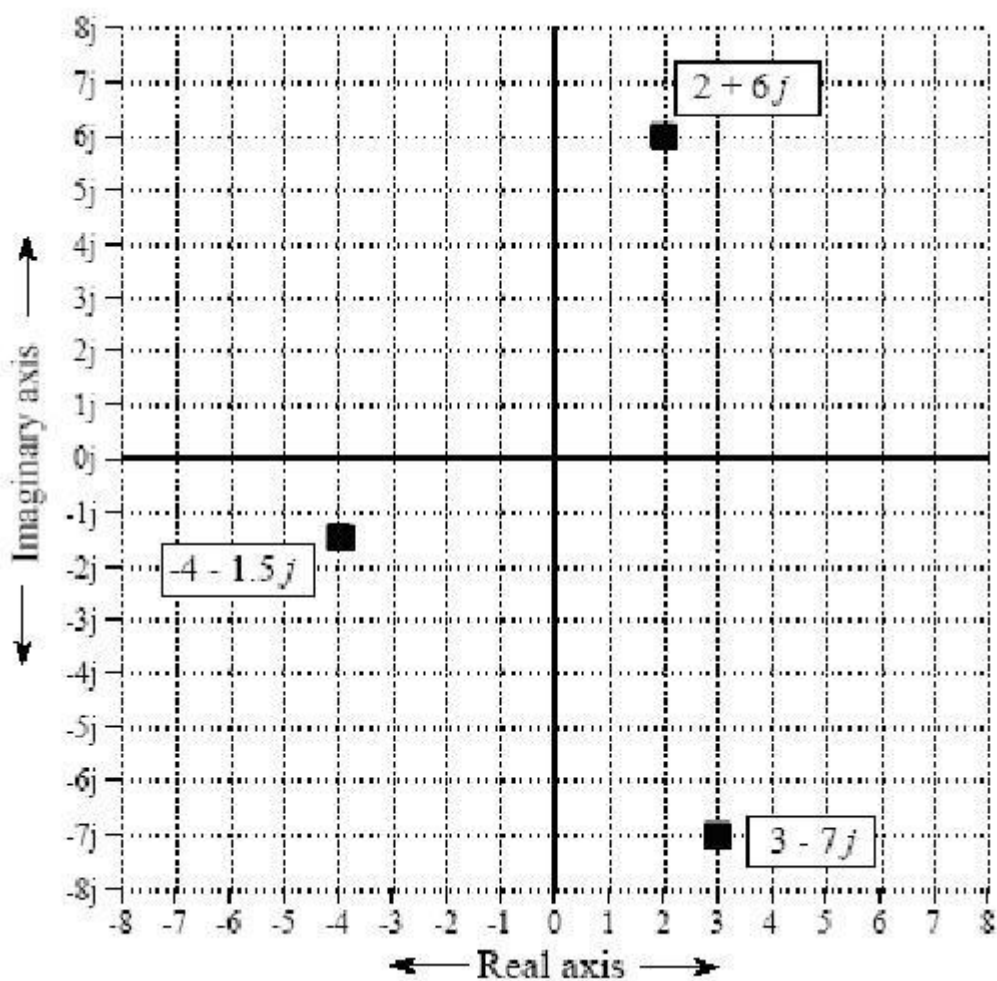
3、直接推到可以得出  $t = v/g \pm \sqrt{(v^2 - 2hg)/g^2}$  ( $v$  和  $g$  后面的 2 都表示平方)，那么也就是说当  $v^2 < 2hg$  时会产生复数，但是如果从实际的  $v^2$  是不可能小于  $2hg$  的，所以我感觉复数不能从实际出发去推到，只能从抽象的角度说明一下。

)

经过计算我们可以知道，当高度是 3 米时，有两个时间点到达该高度：球向上运动时的时间是 0.38 秒，球向下运动时的时间是 1.62 秒。但是如果高度等于 10 时，结果又是什么呢？根据上面的式子可以发现存在对负数进行开平方运算，我们知道这肯定是不现实的。

第一次使用这个不一般的式子的人是意大利数学家 **Girolamo Cardano** (1501-1576)，两个世纪后，德国伟大数学家 **Carl Friedrich Gauss** (1777-1855) 提出了复数的概念，为后来的应用铺平了道路，他对复数进行这样表示：复数由实数 (real) 和虚数 (imaginary) 两部分组成，虚数中的根号负 1 用  $i$  来表示 (在这里我们用  $j$  来表示，因为  $i$  在电力学中表示电流的意思)。

我们可以把横坐标表示成实数，纵坐标表示成虚数，则坐标中的每个点的向量就可以用复数来表示，如下图：



上图中的 **ABC** 三个向量可以表示成如下的式子：

$$A = 2 + 6j$$

$$B = -4 - 1.5j$$

$$C = 3 - 7j$$

这样子来表达方便之处在于运用一个符号就能把两个原来难以联系起来的数组合起来了，不方便的是我们要分辨哪个是实数和哪个是虚数，我们一般是用 **Re( )** 和 **Im( )** 来表示实数和虚数两部分，如：

$$\text{Re } A = 2 \quad \text{Im } A = 6$$

$$\text{Re } B = -4 \quad \text{Im } B = -1.5$$

$$\text{Re } C = 3 \quad \text{Im } C = -7$$

复数之间也可以进行加减乘除运算：

$$(a + bj) + (c + dj) = (a + c) + j(b + d)$$

$$(a + bj) - (c + dj) = (a - c) + j(b - d)$$

$$(a + bj)(c + dj) = (ac - bd) + j(bc + ad)$$

$$\frac{(a + bj)}{(c + dj)} = \left( \frac{ac + bd}{c^2 + d^2} \right) + j \left( \frac{bc - ad}{c^2 + d^2} \right)$$

这里有个特殊的地方是  $j^2$  等于-1，上面第四个式子的计算方法是把分子和分母同时乘以  $c - dj$ ，这样就可消去分母中的  $j$  了。

复数也符合代数运算中的交换律、结合律、分配律：

$$A B = B A$$

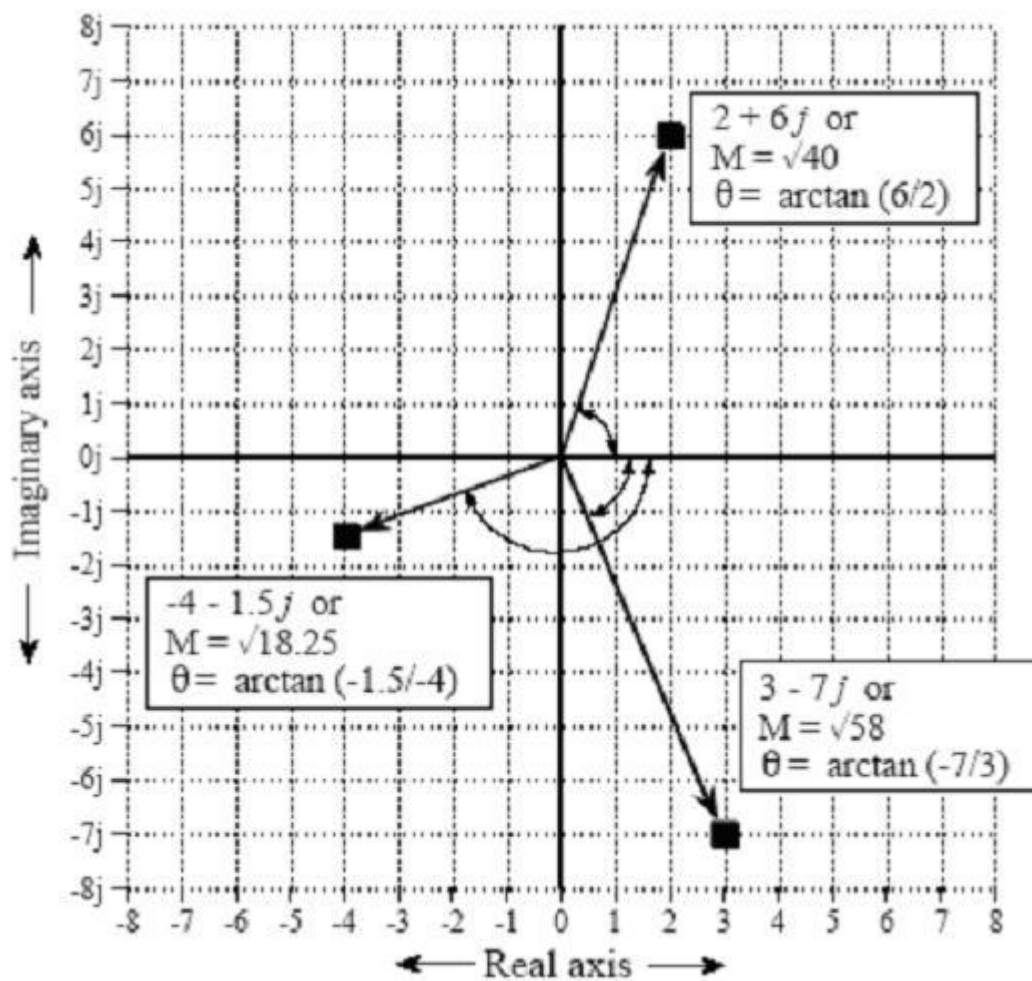
$$(A + B) + C = A + (B + C)$$

$$A(B + C) = AB + AC$$

## 二、复数的极坐标表示形式

前面提到的是运用直角坐标来表示复数，其实更为普遍应用的是极坐标的表示方法，如下图：





上图中的  $M$  即是数量积(magnitude)，表示从原点到坐标点的距离， $\theta$  是相位角(phase angle)，表示从 X 轴正方向到某个向量的夹角，下面四个式子是计算方法：

$$M = \sqrt{(Re A)^2 + (Im A)^2}$$

$$\theta = \arctan \left[ \frac{Im A}{Re A} \right]$$

$$Re A = M \cos(\theta)$$

$$Im A = M \sin(\theta)$$

我们还可以通过下面的式子进行极坐标到直角坐标的转换：

$$\mathbf{a + jb = M (cos\theta + j sin\theta)}$$

上面这个等式中左边是直角坐标表达式，右边是极坐标表达式。

还有一个更为重要的等式——欧拉等式（欧拉，瑞士的著名数学家，Leonhard Euler，1707-1783）：

$$\mathbf{e^{jx} = \cos x + j \sin x}$$

这个等式可以从下面的级数变换中得到证明：

$$e^{jx} = \sum_{n=0}^{\infty} \frac{(jx)^n}{n!} = \left[ \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!} \right] + j \left[ \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!} \right]$$

上面中右边的两个式子分别是  $\cos(x)$  和  $\sin(x)$  的泰勒(Taylor)级数。

这样子我们又可以把复数的表达式表示成指数的形式了：

$$\mathbf{a + jb = M e^{j\theta}} \quad (\text{这便是复数的两个表达式})$$

指数形式是数字信号处理中数学方法的支柱，也许是因为用指数形式进行复数的乘除运算极为简单的缘故吧：

$$M_1 e^{j\theta_1} M_2 e^{j\theta_2} = M_1 M_2 e^{j(\theta_1 + \theta_2)}$$

$$\frac{M_1 e^{j\theta_1}}{M_2 e^{j\theta_2}} = \left[ \frac{M_1}{M_2} \right] e^{j(\theta_1 - \theta_2)}$$

### 三、复数是数学分析中的一个工具

为什么要使用复数呢？其实它只是个工具而已，就如钉子和锤子的关系，复数就象那锤子，作为一种使用的工具。我们把要解决的问题表达成复数的形式（因为有些问题用复数的形式进行运算更加方便），然后对复数进行运算，最后再转换回来得到我们所需要的结果。

有两种方法使用复数，一种是用复数进行简单的替换，如前面所说的向量表达式方法和前一节中我们所讨论的实域 DFT，另一种是更高级的方法：数学等价(mathematical equivalence)，复数形式的傅立叶变换用的便是数学等价的方法，但在这里我们先不讨论这种方法，这里我们先来看一下用复数进行替换中的问题。

用复数进行替换的基本思想是：把所要分析的物理问题转换成复数的形式，其中只是简单地添加一个复数的符号  $j$ ，当返回到原来的物理问题时，则只是把符号  $j$  去掉就可以了。

有一点要明白的是并不是所有问题都可以用复数来表示，必须看用复数进行分析是否适用，有个例子可以看出用复数来替换原来问题的表达方式明显是谬误的：假设一箱的苹果是 5 美元，一箱的桔子是 10 美元，于是我们把它表示成  $5 + 10j$ ，有一个星期你买了 6 箱苹果和 2 箱桔子，我们又把它表示成  $6 + 2j$ ，最后计算总共花的钱是  $(5 + 10j)(6 + 2j) = 10 + 70j$ ，结果是买苹果花了 10 美元的，买桔子花了 70 美元，这样的结果明显是错了，所以复数的形式不适合运用于对这种问题的解决。

### 四、用复数来表示正余弦函数表达式

对于象  $M \cos(\omega t + \phi)$  和  $A \cos(\omega t) + B \sin(\omega t)$  表达式，用复数来表示，可以变得非常简洁，对于直角坐标形式可以按如下形式进行转换：

$$A \cos(\omega t) + B \sin(\omega t) \rightleftharpoons a + jb$$

*(conventional representation) (complex number)*

上式中余弦幅值  $A$  经变换生成  $a$ , 正弦幅值  $B$  的相反数经变换生成  $b$ :  $A \rightleftharpoons a, B \rightleftharpoons -b$ , 但要注意的是, 这不是个等式, 只是个替换形式而已。

对于极坐标形式可以按如下形式进行转换:

$$M \cos(\omega t + \phi) \rightleftharpoons M e^{j\theta}$$

*(conventional representation) (complex number)*

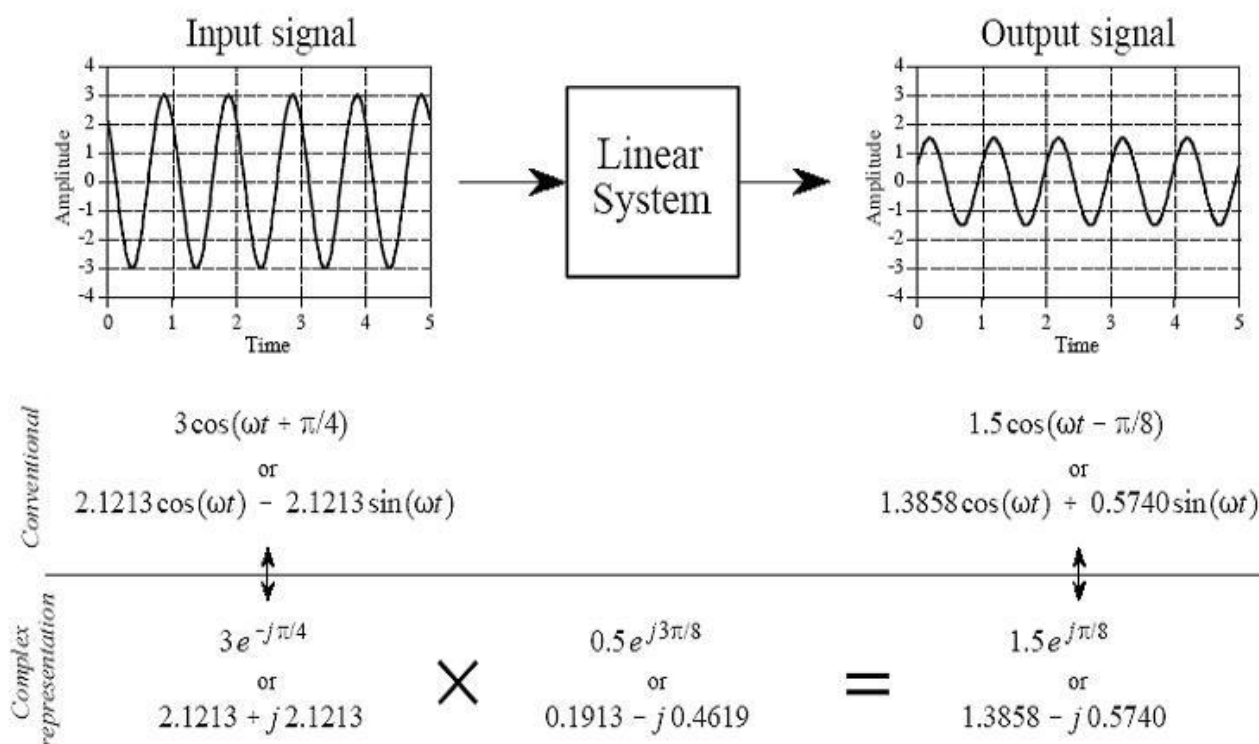
上式中,  $M \rightleftharpoons M, \theta \rightleftharpoons \phi$ 。

这里虚数部分采用负数的形式主要是为了跟复数傅立叶变换表达式保持一致, 对于这种替换的方法来表示正余弦, 符号的变换没有什么好处, 但替换时总会被改变掉符号以跟更高级的等价变换保持形式上的一致。

在离散信号处理中, 运用复数形式来表示正余弦波是个常用的技术, 这是因为利用复数进行各种运算得到的结果跟原来的正余弦运算结果是一致的, 但是, 我们要小心使用复数操作, 如加、减、乘、除, 有些操作是不能用的, 如两个正弦信号相加, 采用复数形式进行相加, 得到的结果跟替换前的直接相加的结果是一样的, 但是如果两个正弦信号相乘, 则采用复数形式来相乘结果是不一样的。幸运的是, 我们已严格定义了正余弦复数形式的运算操作条件:

- 1、参加运算的所有正余弦的频率必须是一样的;
- 2、运算操作必须是线性的, 如两个正弦信号可以进行相加减, 但不能进行乘除, 象信号的放大、衰减、高低通滤波等系统都是线性的, 象平方、缩短、取限等则不是线性的。要记住的是卷积和傅立叶分析也只有线性操作才可以进行。

下图是一个相量变换(我们把正弦或余弦波变成复数的形式称为相量变换, **Phasor transform**)的例子, 一个连续信号波经过一个线性处理系统生成另一个信号波, 从计算过程我们可以看出采用复数的形式使得计算变化十分的简洁:



在第二章中我们描述的实数形式傅立叶变换也是一种替换形式的复数变换，但要注意的是那还不是复数傅立叶变换，只是一种代替方式而已。下一章、即，第四章，我们就会知道复数傅立叶变换是一种更高级的变换，而不是这种简单的替换形式。

## 第四章、复数形式离散傅立叶变换

复数形式的离散傅立叶变换非常巧妙地运用了复数的方法，使得傅立叶变换更加自然和简洁，它并不是只是简单地运用替换的方法来运用复数，而是完全从复数的角度来分析问题，这一点跟实数 DFT 是完全不一样的。

### 一、把正余弦函数表示成复数的形式

通过欧拉等式可以把正余弦函数表示成复数的形式：

$$\begin{aligned}\cos(x) &= \frac{1}{2} e^{j(-x)} + \frac{1}{2} e^{jx} \\ \sin(x) &= j \left( \frac{1}{2} e^{j(-x)} - \frac{1}{2} e^{jx} \right)\end{aligned}$$

从这个等式可以看出，如果把正余弦函数表示成复数后，它们变成了由正负频率组成的正余弦波，相反地，一个由正负频率组成的正余弦波，可以通过复数的形式来表示。

我们知道，在实数傅立叶变换中，它的频谱是  $0 \sim \pi$  ( $0 \sim N/2$ ), 但无法表示  $-\pi \sim 0$  的频谱，可以预见，如果把正余弦表示成复数形式，则能够把负频率包含进来。

## 二、 把变换前后的变量都看成复数的形式

复数形式傅立叶变换把原始信号  $x[n]$  当成是一个用复数来表示的信号，其中实数部分表示原始信号值，虚数部分为 0，变换结果  $X[k]$  也是个复数的形式，但这里的虚数部分是有值的。

在这里要用复数的观点来看原始信号，是理解复数形式傅立叶变换的关键（如果有学过复变函数则可能更好理解，即把  $x[n]$  看成是一个复数变量，然后象对待实数那样对这个复数变量进行相同的变换）。

## 三、 对复数进行相关性算法（正向傅立叶变换）

从实数傅立叶变换中可以知道，我们可以通过原始信号乘以一个正交函数形式的信号，然后进行求总和，最后就能得到这个原始信号所包含的正交函数信号的分量。

现在我们的原始信号变成了复数，我们要得到的当然是复数的信号分量，我们是不是可以把它乘以一个复数形式的正交函数呢？答案是肯定的，正余弦函数都是正交函数，变成如下形式的复数后，仍旧还是正交函数（这个从正交函数的定义可以很容易得到证明）：

$$\cos x + j \sin x, \cos x - j \sin x, \dots$$

这里我们采用上面的第二个式子进行相关性求和，为什么用第二个式子呢？，我们在后面会知道，正弦函数在虚数中变换后得到的是负的正弦函数，这里我们再加上一个负号，使得最后的得到的是正的正弦波，根据这个于是我们很容易就可以得到了复数形式的 **DFT 正向变换等式**：

$$X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] \left( \cos(2\pi kn/N) - j \sin(2\pi kn/N) \right)$$

这个式子很容易可以得到欧拉变换式子：

$$X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N}$$

其实我们是为了表达上的方便才用到欧拉变换式，在解决问题时我们还是较多地用到正余弦表达式。

对于上面的等式，我们要清楚如下几个方面（也是区别于实数 DFT 的地方）：

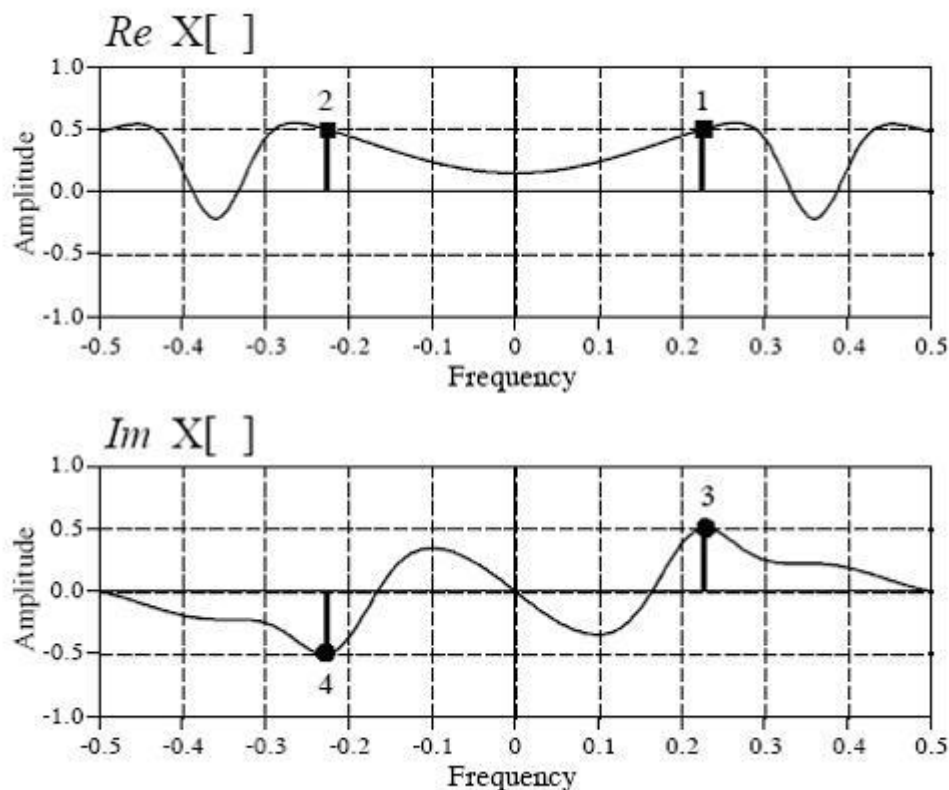
1、 $X[k]$ 、 $x[n]$  都是复数，但  $x[n]$  的虚数部分都是由 0 组成的，实数部分表示原始信号；

2、 $k$  的取值范围是  $0 \sim N-1$  (也可以表达成  $0 \sim 2\pi$ )，其中  $0 \sim N/2$  (或  $0 \sim \pi$ ) 是正频部分，

$N/2 \sim N-1$  ( $\pi \sim 2\pi$ ) 是负频部分，由于正余弦函数的对称性，所以我们把  $-\pi \sim 0$  表示成  $\pi \sim 2\pi$ ，这是出于计算上方便的考虑。

3、其中的  $j$  是一个不可分离的组成部分，就象一个等式中的变量一样，不能随便去掉，去掉之后意义就完全不一样了，但我们知道在实数 DFT 中， $j$  只是个符号而已，把  $j$  去掉，整个等式的意义不变；

4、下图是个连续信号的频谱，但离散频谱也是与此类似的，所以不影响我们对问题的分析：



上面的频谱图把负频率放到了左边，是为了迎合我们的思维习惯，但在实际实现中我们一般是把它移到正的频谱后面的。

从上图可以看出，时域中的正余弦波（用来组成原始信号的正余弦波）在复数 DFT 的频谱中被分成了正、负频率的两个组成部分，基于此等式中前面的比例系数是  $1/N$ （或  $1/2\pi$ ），而不是  $2/N$ ，这是因为现在把频谱延伸到了  $2\pi$ ，但把正负两个频率相加即又得到了  $2/N$ ，又还原到了实数 DFT 的形式，这个在后面的描述中可以更清楚地看到。

由于复数 DFT 生成的是一个完整的频谱，原始信号中的每一个点都是由正、负两个频率组合而成的，所以频谱中每一个点的带宽是一样的，都是  $1/N$ ，相对实数 DFT，两端带宽比其它点的带宽少了一半；复数 DFT 的频谱特征具有周期性： $-N/2 \sim 0$  与  $N/2 \sim N-1$

是一样的，实域频谱呈偶对称性（表示余弦波频谱），虚域频谱呈奇对称性（表示正弦波频谱）。

#### 四、 逆向傅立叶变换

假设我们已经得到了复数形式的频谱  $X[k]$ ，现在要把它还原到复数形式的原始信号  $x[n]$ ，当然应该是把  $X[k]$  乘以一个复数，然后再进行求和，最后得到原始信号  $x[n]$ ，这个跟  $X[k]$  相乘的复数首先让我们想到的应该是上面进行相关性计算的复数：

$$\cos(2\pi kn/N) - j \sin(2\pi kn/N),$$

但其中的负号其实是为了使得进行逆向傅立叶变换时把正弦函数变为正的符号，因为虚数  $j$  的运算特殊性，使得原来应该是正的正弦函数变为了负的正弦函数（我们从后面的推导会看到这一点），所以这里的负号只是为了纠正符号的作用，在进行逆向 DFT 时，我们可以把负号去掉，于是我们便得到了这样的**逆向 DFT 变换等式**：

$$x[n] = X[k] (\cos(2\pi kn/N) + j \sin(2\pi kn/N))$$

我们现在来分析这个式子，会发现这个式其实跟实数傅立叶变换是可以得到一样结果的。我们先把  $X[k]$  变换一下：

$$X[k] = \text{Re } X[k] + j \text{Im } X[k]$$

这样我们就可以对  $x[n]$  再次进行变换，如：

$$\begin{aligned} x[n] &= (\text{Re } X[k] + j \text{Im } X[k]) (\cos(2\pi kn/N) + j \sin(2\pi kn/N)) \\ &= (\text{Re } X[k] \cos(2\pi kn/N) + j \text{Im } X[k] \cos(2\pi kn/N) + j \text{Re } X[k] \sin(2\pi kn/N) - \text{Im } X[k] \sin(2\pi kn/N)) \\ &= (\text{Re } X[k] (\cos(2\pi kn/N) + j \sin(2\pi kn/N)) + \\ &\quad + \text{Im } X[k] (-\sin(2\pi kn/N) + j \cos(2\pi kn/N))) \end{aligned} \quad (1)$$

这时我们就把原来的等式分成了两个部分，第一个部分是跟实域中的频谱相乘，第二个部分是跟虚域中的频谱相乘，根据频谱图我们可以知道， $\text{Re } X[k]$  是个偶对称的变量， $\text{Im } X[k]$  是个奇对称的变量，即

$$\begin{aligned} \text{Re } X[k] &= \text{Re } X[-k] \\ \text{Im } X[k] &= -\text{Im } X[-k] \end{aligned}$$



但  $k$  的范围是  $0 \sim N-1$ ,  $0 \sim N/2$  表示正频率,  $N/2 \sim N-1$  表示负频率, 为了表达方便我们把  $N/2 \sim N-1$  用  $-k$  来表示, 这样在从  $0$  到  $N-1$  的求和过程中对于(1)和(2)式分别有  $N/2$  对的  $k$  和  $-k$  的和, 对于 (1) 式有:

$$\operatorname{Re} X[k] (\cos(2\pi kn/N) + j \sin(2\pi kn/N)) + \operatorname{Re} X[-k] (\cos(-2\pi kn/N) + j \sin(-2\pi kn/N))$$

根据偶对称性和三角函数的性质, 把上式化简得到:

$$\operatorname{Re} X[k] (\cos(2\pi kn/N) + j \sin(2\pi kn/N)) + \operatorname{Re} X[k] (\cos(2\pi kn/N) - j \sin(2\pi kn/N))$$

这个式子最后的结果是:

$$2 \operatorname{Re} X[k] \cos(2\pi kn/N)。$$

再考虑到求  $\operatorname{Re} X[k]$  等式中有个比例系数  $1/N$ , 把  $1/N$  乘以  $2$ , 这样的结果不就跟实数 DFT 中的式子一样了吗?

对于(2)式, 用同样的方法, 我们也可以得到这样的结果:

$$-2 \operatorname{Im} X[k] \sin(2\pi kn/N)$$

注意上式前面多了个负符号, 这是由于虚数变换的特殊性造成的, 当然我们肯定不能把负符号的正弦函数跟余弦来相加, 还好, 我们前面是用  $\cos(2\pi kn/N) - j \sin(2\pi kn/N)$  进行相关性计算, 得到的  $\operatorname{Im} X[k]$  中有个负的符号, 这样最后的结果中正弦函数就没有负的符号了, 这就是为什么在进行相关性计算时虚数部分要用到负符号的原因(我觉得这也许是复数形式 DFT 美中不足的地方, 让人有一种拼凑的感觉)。

从上面的分析中可以看出, 实数傅立叶变换跟复数傅立叶变换, 在进行逆变换时得到的结果是一样的, 只不过是殊途同归吧。本文完。(July、dznlong)

本人 **July** 对本博客所有任何文章、内容和资料享有版权。  
转载务必注明作者本人及出处, 并通知本人。二零一一年二月二十二日。

## 十一、从头到尾彻底解析 Hash 表算法

作者: July、wuliming、pkuoliver

出处: [http://blog.csdn.net/v\\_JULY\\_v](http://blog.csdn.net/v_JULY_v)。

说明: 本文分为三部分内容,

第一部分为一道百度面试题 Top K 算法的详解; 第二部分为关于 Hash 表算法的详细

阐述；第三部分为打造一个最快的 Hash 表算法。

---

## 第一部分：Top K 算法详解

### 问题描述

百度面试题：

搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为 1-255 字节。

假设目前有一千万个记录（这些查询串的重复度比较高，虽然总数是 1 千万，但如果除去重复后，不超过 3 百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。），请你统计最热门的 10 个查询串，要求使用的内存不能超过 1G。

必备知识：

什么是哈希表？

哈希表（Hash table，也叫散列表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

哈希表的做法其实很简单，就是把 Key 通过一个固定的算法函数既所谓的哈希函数转换成一个整型数字，然后就将该数字对数组长度进行取余，取余结果就当作数组的下标，将 value 存储在以该数字为下标的数组空间里。

而当使用哈希表进行查询的时候，就是再次使用哈希函数将 key 转换为对应的数组下标，并定位到该空间获取 value，如此一来，就可以充分利用到数组的定位性能进行数据定位（文章第二、三部分，会针对 Hash 表详细阐述）。

### 问题解析：

要统计最热门查询，首先就是要统计每个 Query 出现的次数，然后根据统计结果，找出 Top 10。所以我们可以基于这个思路分两步来设计该算法。

即，此问题的解决分为以下两个步骤：

### 第一步：Query 统计

Query 统计有以下两个方法，可供选择：

#### 1、直接排序法

首先我们最先想到的的算法就是排序了，首先对这个日志里面的所有 Query 都进行排序，然后再遍历排好序的 Query，统计每个 Query 出现的次数了。

但是题目中有明确要求，那就是内存不能超过 1G，一千万条记录，每条记录是 225Byte，很显然要占据 2.55G 内存，这个条件就不满足要求了。

让我们回忆一下数据结构课程上的内容，当数据量比较大而且内存无法装下的时候，我们可以采用外排序的方法来进行排序，这里我们可以采用归并排序，因为归并排序有一个比较好的时间复杂度  $O(N\lg N)$ 。

排完序之后我们再对已经有序的 Query 文件进行遍历，统计每个 Query 出现的次数，再次写入文件中。

综合分析一下，排序的时间复杂度是  $O(N\lg N)$ ，而遍历的时间复杂度是  $O(N)$ ，因此该算法的总体时间复杂度就是  $O(N+N\lg N)=O(N\lg N)$ 。

## 2、Hash Table 法

在第 1 个方法中，我们采用了排序的办法来统计每个 Query 出现的次数，时间复杂度是  $N\lg N$ ，那么能不能有更好的方法来存储，而时间复杂度更低呢？

题目中说明了，虽然有一千万个 Query，但是由于重复度比较高，因此事实上只有 300 万的 Query，每个 Query 255Byte，因此我们可以考虑把他们都放进内存中去，而现在只是需要一个合适的数据结构，在这里，Hash Table 绝对是我们优先的选择，因为 Hash Table 的查询速度非常的快，几乎是  $O(1)$  的时间复杂度。

那么，我们的算法就有了：维护一个 Key 为 Query 字符串，Value 为该 Query 出现次数的 HashTable，每次读取一个 Query，如果该字符串不在 Table 中，那么加入该字符串，并且将 Value 值设为 1；如果该字符串在 Table 中，那么将该字符串的计数加一即可。最终我们在  $O(N)$  的时间复杂度内完成了对该海量数据的处理。

本方法相比算法 1：在时间复杂度上提高了一个数量级，为  $O(N)$ ，但不仅仅是时间复杂度上的优化，该方法只需要 IO 数据文件一次，而算法 1 的 IO 次数较多的，因此该算法 2 比算法 1 在工程上有更好的可操作性。

## 第二步：找出 Top 10

### 算法一：普通排序

我想对于排序算法大家都已经不陌生了，这里不在赘述，我们要注意的是排序算法的时间复杂度是  $N\lg N$ ，在本题目中，三百万条记录，用 1G 内存是可以存下的。

### 算法二：部分排序

题目要求是求出 Top 10，因此我们没有必要对所有的 Query 都进行排序，我们只需要维护一个 10 个大小的数组，初始化放入 10 个 Query，按照每个 Query 的统计次数由大到小排序，然后遍历这 300 万条记录，每读一条记录就和数组最后一个 Query 对比，如果小于这个 Query，那么继续遍历，否则，将数组中最后一条数据淘汰，加入当前的 Query。最后当所有的数据都遍历完毕之后，那么这个数组中的 10 个 Query 便是我们要找的 Top 10 了。

不难分析出，这样，算法的最坏时间复杂度是  $N*K$ ，其中 K 是指 top 多少。

### 算法三：堆

在算法二中，我们已经将时间复杂度由  $N\log N$  优化到  $NK$ ，不得不说这是一个比较大的改进了，可是有没有更好的办法呢？

分析一下，在算法二中，每次比较完成之后，需要的操作复杂度都是  $K$ ，因为要把元素插入到一个线性表之中，而且采用的是顺序比较。这里我们注意一下，该数组是有序的，一次我们每次查找的时候可以采用二分的方法查找，这样操作的复杂度就降到了  $\log K$ ，可是，随之而来的问题就是数据移动，因为移动数据次数增多了。不过，这个算法还是比算法二有了改进。

基于以上的分析，我们想想，有没有一种既能快速查找，又能快速移动元素的数据结构呢？回答是肯定的，那就是堆。

借助堆结构，我们可以在  $\log$  量级的时间内查找和调整/移动。因此到这里，我们的算法可以改进为这样，维护一个  $K$  (该题目中是 10) 大小的小根堆，然后遍历 300 万的 Query，分别和根元素进行对比。

思想与上述算法二一致，只是算法在算法三，我们采用了最小堆这种数据结构代替数组，把查找目标元素的时间复杂度有  $O(K)$  降到了  $O(\log K)$ 。

那么这样，采用堆数据结构，算法三，最终的时间复杂度就降到了  $N'\log K$ ，和算法二相比，又有了比较大的改进。

### 总结：

至此，算法就完全结束了，经过上述第一步、先用 Hash 表统计每个 Query 出现的次数， $O(N)$ ；然后第二步、采用堆数据结构找出 Top 10， $N * O(\log K)$ 。所以，我们最终的时间复杂度是： $O(N) + N' * O(\log K)$ 。（ $N$  为 1000 万， $N'$  为 300 万）。如果各位有什么更好的算法，欢迎留言评论。第一部分，完。

## 第二部分、Hash 表 算法的详细解析

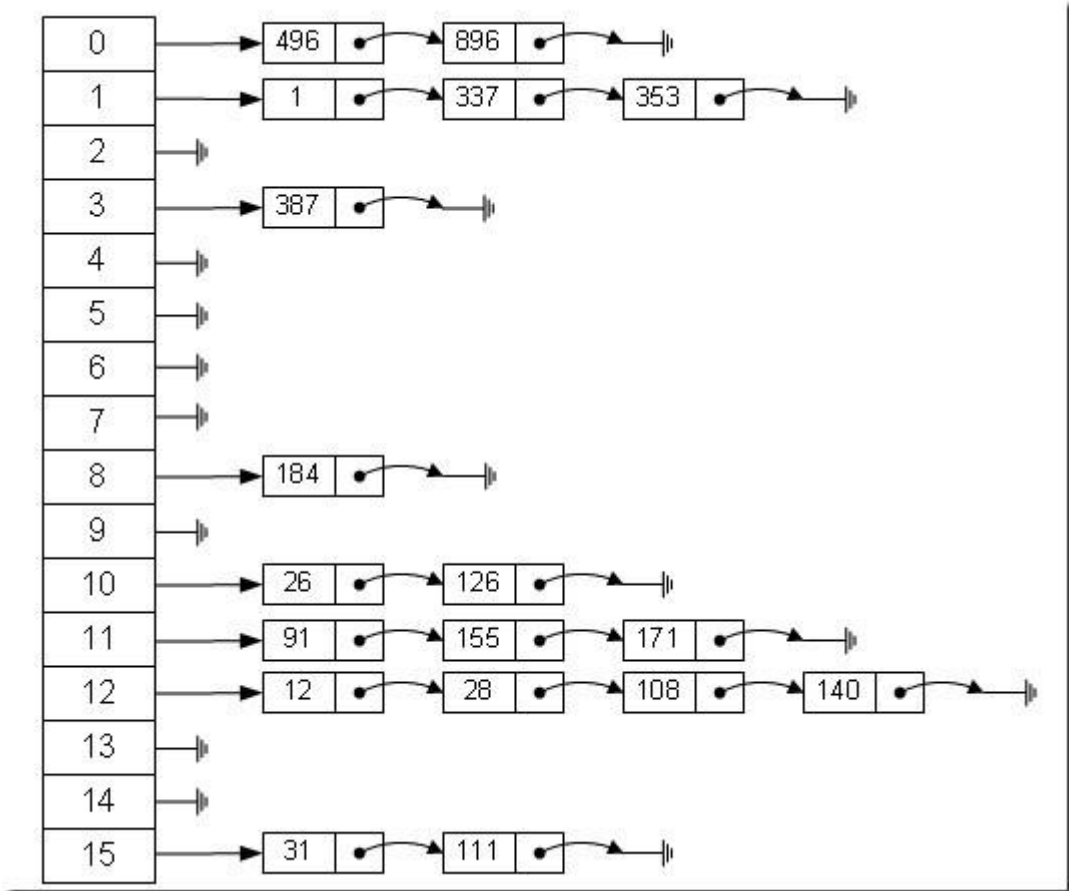
### 什么是 Hash

Hash，一般翻译做“散列”，也有直接音译为“哈希”的，就是把任意长度的输入（又叫做预映射，pre-image），通过散列算法，变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，而不可能从散列值来唯一的确定输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

HASH 主要用于信息安全领域中加密算法，它把一些不同长度的信息转化成杂乱的 128 位的编码，这些编码值叫做 HASH 值。也可以说，hash 就是找到一种数据内容和数据存放地址之间的映射关系。

数组的特点是：寻址容易，插入和删除困难；而链表的特点是：寻址困难，插入和删除容易。那么我们能不能综合两者的特性，做出一种寻址容易，插入删除也容易的数据结构？

答案是肯定的，这就是我们要提起的哈希表，哈希表有多种不同的实现方法，我接下来解释的是最常用的一种方法——拉链法，我们可以理解为“链表的数组”，如图：



左边很明显是个数组，数组的每个成员包括一个指针，指向一个链表的头，当然这个链表可能为空，也可能元素很多。我们根据元素的一些特征把元素分配到不同的链表中去，也是根据这些特征，找到正确的链表，再从链表中找出这个元素。

元素特征转变为数组下标的方法就是散列法。散列法当然不止一种，下面列出三种比较常用的：

### 1，除法散列法

最直观的一种，上图使用的就是这种散列法，公式：

$$\text{index} = \text{value} \% 16$$

学过汇编的都知道，求模数其实是通过一个除法运算得到的，所以叫“除法散列法”。

### 2，平方散列法

求 index 是非常频繁的操作，而乘法的运算要比除法来得省时（对现在的 CPU 来说，估计我们感觉不出来），所以我们考虑把除法换成乘法和一个位移操作。公式：

$$\text{index} = (\text{value} * \text{value}) \gg 28 \quad (\text{右移, 除以 } 2^{28} \text{。记法: 左移变大, 是乘。右移变小, 是除。})$$

如果数值分配比较均匀的话这种方法能得到不错的结果，但我上面画的那个图的各个元素的值算出来的 index 都是 0——非常失败。也许你还有个问题，value 如果很大，value

\* value 不会溢出吗？答案是会的，但我们这个乘法不关心溢出，因为我们根本不是为了获取相乘结果，而是为了获取 index。

### 3，斐波那契（Fibonacci）散列法

平方散列法的缺点是显而易见的，所以我们能不能找出一个理想的乘数，而不是拿 value 本身当作乘数呢？答案是肯定的。

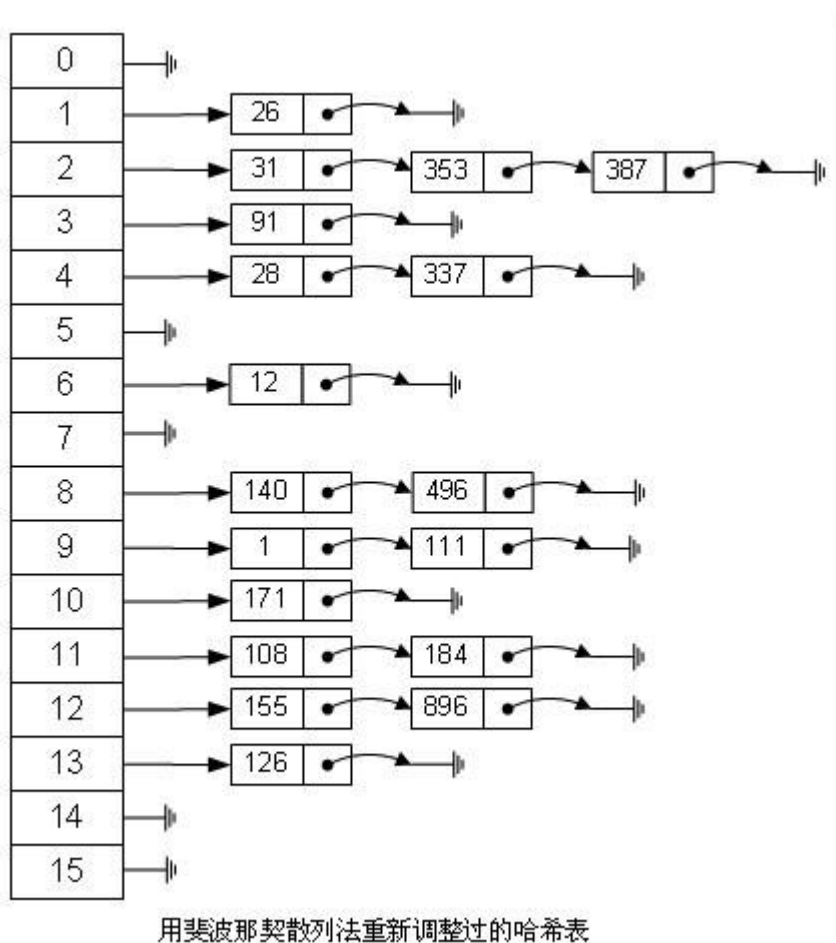
- 1，对于 16 位整数而言，这个乘数是 40503
- 2，对于 32 位整数而言，这个乘数是 2654435769
- 3，对于 64 位整数而言，这个乘数是 11400714819323198485

这几个“理想乘数”是如何得出来的呢？这跟一个法则有关，叫黄金分割法则，而描述黄金分割法则的最经典表达式无疑就是著名的斐波那契数列，即如此形式的序列：  
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, ...。另外，斐波那契数列的值和太阳系八大行星的轨道半径的比例出奇吻合。

对我们常见的 32 位整数而言，公式：

`index = (value * 2654435769) >> 28`

如果用这种斐波那契散列法的话，那上面的图就变成这样了：



很明显，用斐波那契散列法调整之后要比原来的取模散列法好很多。

## 适用范围

快速查找，删除的基本数据结构，通常需要总数据量可以放入内存。

## 基本原理及要点

hash 函数选择，针对字符串，整数，排列，具体相应的 hash 方法。

碰撞处理，一种是 open hashing，也称为拉链法；另一种就是 closed hashing，也称开地址法，opened addressing。

## 扩展

d-left hashing 中的 d 是多个的意思，我们先简化这个问题，看一看 2-left hashing。2-left hashing 指的是将一个哈希表分成长度相等的两半，分别叫做 T1 和 T2，给 T1 和 T2 分别配备一个哈希函数，h1 和 h2。在存储一个新的 key 时，同时用两个哈希函数进行计算，得出两个地址 h1[key]和 h2[key]。这时需要检查 T1 中的 h1[key]位置和 T2 中的 h2[key]位置，哪一个位置已经存储的（有碰撞的）key 比较多，然后将新 key 存储在负载少的位置。如果两边一样多，比如两个位置都为空或者都存储了一个 key，就把新

key 存储在左边的 T1 子表中, 2-left 也由此而来。在查找一个 key 时, 必须进行两次 hash, 同时查找两个位置。

## 问题实例（海量数据处理）

我们知道 hash 表在海量数据处理中有着广泛的应用, 下面, 请看另一道百度面试题:  
题目: 海量日志数据, 提取出某日访问百度次数最多的那个 IP。

方案: IP 的数目还是有限的, 最多  $2^{32}$  个, 所以可以考虑使用 hash 将 ip 直接存入内存, 然后进行统计。

## 第三部分、最快的 Hash 表算法

接下来, 咱们来具体分析一下一个最快的 Hash 表算法。

我们由一个简单的问题逐步入手: 有一个庞大的字符串数组, 然后给你一个单独的字符串, 让你从这个数组中查找是否有这个字符串并找到它, 你会怎么做? 有一个方法最简单, 老老实实从头查到尾, 一个一个比较, 直到找到为止, 我想只要学过程序设计的人都能把这样一个程序作出来, 但要是程序员把这样的程序交给用户, 我只能用无语来评价, 或许它真的能工作, 但...也只能如此了。

最合适的算法自然是使用 HashTable (哈希表), 先介绍介绍其中的基本知识, 所谓 Hash, 一般是一个整数, 通过某种算法, 可以把一个字符串"压缩" 成一个整数。当然, 无论如何, 一个 32 位整数是无法对应回一个字符串的, 但在程序中, 两个字符串计算出的 Hash 值相等的可能非常小, 下面看看在 MPQ 中的 Hash 算法:

函数一、以下的函数生成一个长度为 0x500 (合 10 进制数: 1280) 的 cryptTable[0x500]

```
void prepareCryptTable()
{
    unsigned long seed = 0x00100001, index1 = 0, index2 = 0, i;

    for( index1 = 0; index1 < 0x100; index1++ )
    {
        for( index2 = index1, i = 0; i < 5; i++, index2 += 0x100 )
        {
            unsigned long temp1, temp2;

            seed = (seed * 125 + 3) % 0x2AAAAB;
            temp1 = (seed & 0xFFFF) << 0x10;

            seed = (seed * 125 + 3) % 0x2AAAAB;
            temp2 = (seed & 0xFFFF);

            cryptTable[index2] = ( temp1 | temp2 );
        }
    }
}
```



```

    }
}
}

```

函数二、以下函数计算 lpszFileName 字符串的 hash 值，其中 dwHashType 为 hash 的类型，在下面的函数三、GetHashTablePos 函数中调用此函数二，其可以取的值为 0、1、2；该函数返回 lpszFileName 字符串的 hash 值：

```

unsigned long HashString( char *lpszFileName, unsigned long dwHashType )
{
    unsigned char *key = (unsigned char *)lpszFileName;
    unsigned long seed1 = 0x7FED7FED;
    unsigned long seed2 = 0xEEEEEEEE;
    int ch;

    while( *key != 0 )
    {
        ch = toupper(*key++);

        seed1 = cryptTable[(dwHashType << 8) + ch] ^ (seed1 + seed2);
        seed2 = ch + seed1 + seed2 + (seed2 << 5) + 3;
    }
    return seed1;
}

```

Blizzard 的这个算法是非常高效的，被称为"One-Way Hash"( A one-way hash is a an algorithm that is constructed in such a way that deriving the original string (set of strings, actually) is virtually impossible)。举个例子，字符串 "unitneutralacritter.grp"通过这个算法得到的结果是 0xA26067F3。

是不是把第一个算法改进一下，改成逐个比较字符串的 Hash 值就可以了呢，答案是，远远不够，要想得到最快的算法，就不能进行逐个的比较，通常是构造一个**哈希表**(Hash Table)来解决问题，哈希表是一个大数组，这个数组的容量根据程序的要求来定义，例如 1024，每一个 Hash 值通过取模运算 (mod) 对应到数组中的一个位置，这样，只要比较这个字符串的哈希值对应的位置有没有被占用，就可以得到最后的结果了，想想这是什么速度？是的，是最快的 O(1)，现在仔细看看这个算法吧：

```

typedef struct
{
    int nHashA;
    int nHashB;
    char bExists;
    .....
}

```

```
} SOMESTRUCTURE;
```

一种可能的结构体定义？

**函数三**、下述函数为在 Hash 表中查找是否存在目标字符串，有则返回要查找字符串的 Hash 值，无则，return -1.

```
int GetHashTablePos( har *lpszString, SOMESTRUCTURE *lpTable )
//lpszString 要在 Hash 表中查找的字符串，lpTable 为存储字符串 Hash 值的 Hash 表。
{
    int nHash = HashString(lpszString); //调用上述函数二，返回要查找字符串
    lpszString 的 Hash 值。
    int nHashPos = nHash % nTableSize;

    if ( lpTable[nHashPos].bExists && !strcmp( lpTable[nHashPos].pString,
    lpszString ) )
    { //如果找到的 Hash 值在表中存在，且要查找的字符串与表中对应位置的字符串相
    同，
        return nHashPos; //则返回上述调用函数二后，找到的 Hash 值
    }
    else
    {
        return -1;
    }
}
```

看到此，我想大家都在想一个很严重的问题：“如果两个字符串在哈希表中对应的位置相同怎么办？”，毕竟一个数组容量是有限的，这种可能性很大。解决该方法的方法很多，我首先想到的就是用“**链表**”，感谢大学里学的数据结构教会了这个百试百灵的法宝，我遇到的很多算法都可以转化成链表来解决，只要在哈希表的每个入口挂一个链表，保存所有对应的字符串就 OK 了。事情到此似乎有了完美的结局，如果是把问题独自交给我解决，此时我可能就要开始定义数据结构然后写代码了。

然而 Blizzard 的程序员使用的方法则是更精妙的方法。基本原理就是：他们在哈希表中不是用一个哈希值而是用**三个哈希值**来校验字符串。

MPQ 使用文件名哈希表来跟踪内部的所有文件。但是这个表的格式与正常的哈希表有一些不同。首先，它没有使用哈希作为下标，把实际的文件名存储在表中用于验证，实际上它根本就没有存储文件名。而是使用了 3 种不同的哈希：一个用于哈希表的下标，两个用于验证。这两个验证哈希替代了实际文件名。

当然了，这样仍然会出现 2 个不同的文件名哈希到 3 个同样的哈希。但是这种情况发生的概率平均是：**1:18889465931478580854784**，这个概率对于任何人来说应该都是足够小的。现在再回到数据结构上，Blizzard 使用的哈希表没有使用链表，而采用“**顺延**”的方式来解决这个问题，看看这个算法：

函数四、lpszString 为要在 hash 表中查找的字符串；lpTable 为存储字符串 hash 值的 hash 表；nTableSize 为 hash 表的长度：

```
int GetHashTablePos( char *lpszString, MPQHASHTABLE *lpTable, int
nTableSize )
{
    const int  HASH_OFFSET = 0, HASH_A = 1, HASH_B = 2;

    int  nHash = HashString( lpszString, HASH_OFFSET );
    int  nHashA = HashString( lpszString, HASH_A );
    int  nHashB = HashString( lpszString, HASH_B );
    int  nHashStart = nHash % nTableSize;
    int  nHashPos = nHashStart;

    while ( lpTable[nHashPos].bExists )
    {
        /*如果仅仅是判断在该表中时候存在这个字符串，就比较这两个 hash 值就可以了，不用对
        用对
        *结构体中的字符串进行比较。这样会加快运行的速度？减少 hash 表占用的空间？这种
        种
        *方法一般应用在什么场合？*/
        if (    lpTable[nHashPos].nHashA == nHashA
            && lpTable[nHashPos].nHashB == nHashB )
        {
            return nHashPos;
        }
        else
        {
            nHashPos = (nHashPos + 1) % nTableSize;
        }

        if (nHashPos == nHashStart)
            break;
    }
    return -1;
}
```

上述程序解释：

1. 计算出字符串的三个哈希值（一个用来确定位置，另外两个用来校验）
2. 察看哈希表中的这个位置
3. 哈希表中这个位置为空吗？如果为空，则肯定该字符串不存在，返回-1。
4. 如果存在，则检查其他两个哈希值是否也匹配，如果匹配，则表示找到了该字符串，返回其 Hash 值。
5. 移到下一个位置，如果已经移到了表的末尾，则反绕到表的开始位置起继续查询

6. 看看是不是又回到了原来的位置，如果是，则返回没找到
7. 回到 3

ok，这就是本文中所说的最快的 Hash 表算法。什么?不够快?:D。欢迎，各位批评指正。

-----

补充 1、一个简单的 hash 函数:

```
/*key 为一个字符串，nTableLength 为哈希表的长度
*该函数得到的 hash 值分布比较均匀*/
unsigned long getHashIndex( const char *key, int nTableLength )
{
    unsigned long nHash = 0;

    while (*key)
    {
        nHash = (nHash<<5) + nHash + *key++;
    }

    return ( nHash % nTableLength );
}
```

补充 2、一个完整测试程序:

哈希表的数组是定长的，如果太大，则浪费，如果太小，体现不出效率。合适的数组大小是哈希表的性能的关键。哈希表的尺寸最好是一个质数。当然，根据不同的数据量，会有不同的哈希表的大小。对于数据量时多时少的应用，最好的设计是使用动态可变尺寸的哈希表，那么如果你发现哈希表尺寸太小了，比如其中的元素是哈希表尺寸的 2 倍时，我们就需要扩大哈希表尺寸，一般是扩大一倍。

下面是哈希表尺寸大小的可能取值:

17,	37,	79,	163,	331,
673,	1361,	2729,	5471,	10949,
21911,	43853,	87719,	175447,	350899,
701819,	1403641,	2807303,	5614657,	11229331,
22458671,	44917381,	89834777,	179669557,	359339171,
718678369,	1437356741,	2147483647		

以下为该程序的完整源码，已在 linux 下测试通过：

[view plaincopy to clipboardprint?](#)

```
1. #include <stdio.h>
2. #include <ctype.h>      //多谢 citylove 指正。
3. //cryptTable[]里面保存的是 HashString 函数里面将会用到的一些数据，在
   prepareCryptTable
4. //函数里面初始化
5. unsigned long cryptTable[0x500];
6.
7. //以下的函数生成一个长度为 0x500（合 10 进制数：1280）的 cryptTable[0x500]
8. void prepareCryptTable()
9. {
10.     unsigned long seed = 0x00100001, index1 = 0, index2 = 0, i;
11.
12.     for( index1 = 0; index1 < 0x100; index1++ )
13.     {
14.         for( index2 = index1, i = 0; i < 5; i++, index2 += 0x100 )
15.         {
16.             unsigned long temp1, temp2;
17.
18.             seed = (seed * 125 + 3) % 0x2AAAAAB;
19.             temp1 = (seed & 0xFFFF) << 0x10;
20.
21.             seed = (seed * 125 + 3) % 0x2AAAAAB;
22.             temp2 = (seed & 0xFFFF);
23.
24.             cryptTable[index2] = ( temp1 | temp2 );
25.         }
26.     }
27. }
28.
29. //以下函数计算 lpzFileName 字符串的 hash 值，其中 dwHashType 为 hash 的类型，
30. //在下面 GetHashTablePos 函数里面调用本函数，其可以取的值为 0、1、2；该函数
31. //返回 lpzFileName 字符串的 hash 值；
32. unsigned long HashString( char *lpzFileName, unsigned long dwHashType )
33. {
34.     unsigned char *key = (unsigned char *)lpzFileName;
35.     unsigned long seed1 = 0x7FED7FED;
36.     unsigned long seed2 = 0xEEEEEEEE;
37.     int ch;
38.
39.     while( *key != 0 )
40.     {
```

```

41.         ch = toupper(*key++);
42.
43.         seed1 = cryptTable[(dwHashType << 8) + ch] ^ (seed1 + seed2);
44.         seed2 = ch + seed1 + seed2 + (seed2 << 5) + 3;
45.     }
46.     return seed1;
47. }
48.
49. //在 main 中测试 argv[1]的三个 hash 值:
50. //./hash "arr\units.dat"
51. //./hash "unit\netral\acritter.grp"
52. int main( int argc, char **argv )
53. {
54.     unsigned long ulHashValue;
55.     int i = 0;
56.
57.     if ( argc != 2 )
58.     {
59.         printf("please input two arguments\n");
60.         return -1;
61.     }
62.
63.     /*初始化数组: cryptTable[0x500]*/
64.     prepareCryptTable();
65.
66.     /*打印数组 cryptTable[0x500]里面的值*/
67.     for ( ; i < 0x500; i++ )
68.     {
69.         if ( i % 10 == 0 )
70.         {
71.             printf("\n");
72.         }
73.
74.         printf("%-12X", cryptTable[i] );
75.     }
76.
77.     ulHashValue = HashString( argv[1], 0 );
78.     printf("\n----%X ----\n", ulHashValue );
79.
80.     ulHashValue = HashString( argv[1], 1 );
81.     printf("----%X ----\n", ulHashValue );
82.
83.     ulHashValue = HashString( argv[1], 2 );
84.     printf("----%X ----\n", ulHashValue );

```

```
85.  
86.     return 0;  
87. }
```

致谢：

- 1、<http://blog.redfox66.com/>。
- 2、[http://blog.csdn.net/wuliming\\_sc/](http://blog.csdn.net/wuliming_sc/)。完。

版权所有，法律保护。转载，请以链接形式，注明出处。

## 十一、从头到尾彻底解析 Hash 表算法

作者：July、yansha。编程艺术室出品。

出处：结构之法算法之道。

### 前言

本文阐述两个问题，第二十三章是杨氏矩阵查找问题，第二十四章是有关倒排索引中关键词 Hash 编码的问题，主要要解决不重复以及追加的功能，同时也是经典算法研究系列十一、从头到尾彻底解析 Hash 表算法之续。

OK，有任何问题，也欢迎随时交流或批评指正。谢谢。

### 第二十三章、杨氏矩阵查找

#### 杨氏矩阵查找

先看一个来自算法导论习题里 6-3 与剑指 offer 的一道编程题（也被经常用作面试题，本人此前去搜狗二面时便遇到了）：

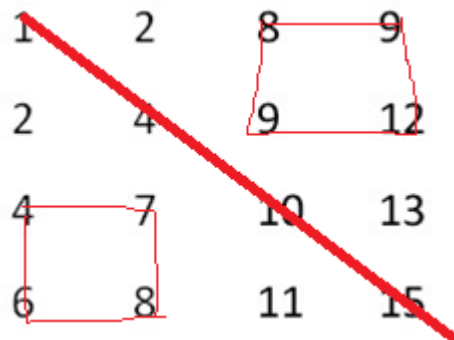
在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

例如下面的二维数组就是每行、每列都递增排序。如果在这个数组中查找数字 6，则返回 true；如果查找数字 5，由于数组不含有该数字，则返回 false。

1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

本 Young 问题解法有二（如查找数字 6）：

1、分治法，分为四个矩形，配以二分查找，如果要找的数是 6 介于对角线上相邻的两个数 4、10，可以排除掉左上和右下的两个矩形，而递归在左下和右上的两个矩形继续找，如下图所示：

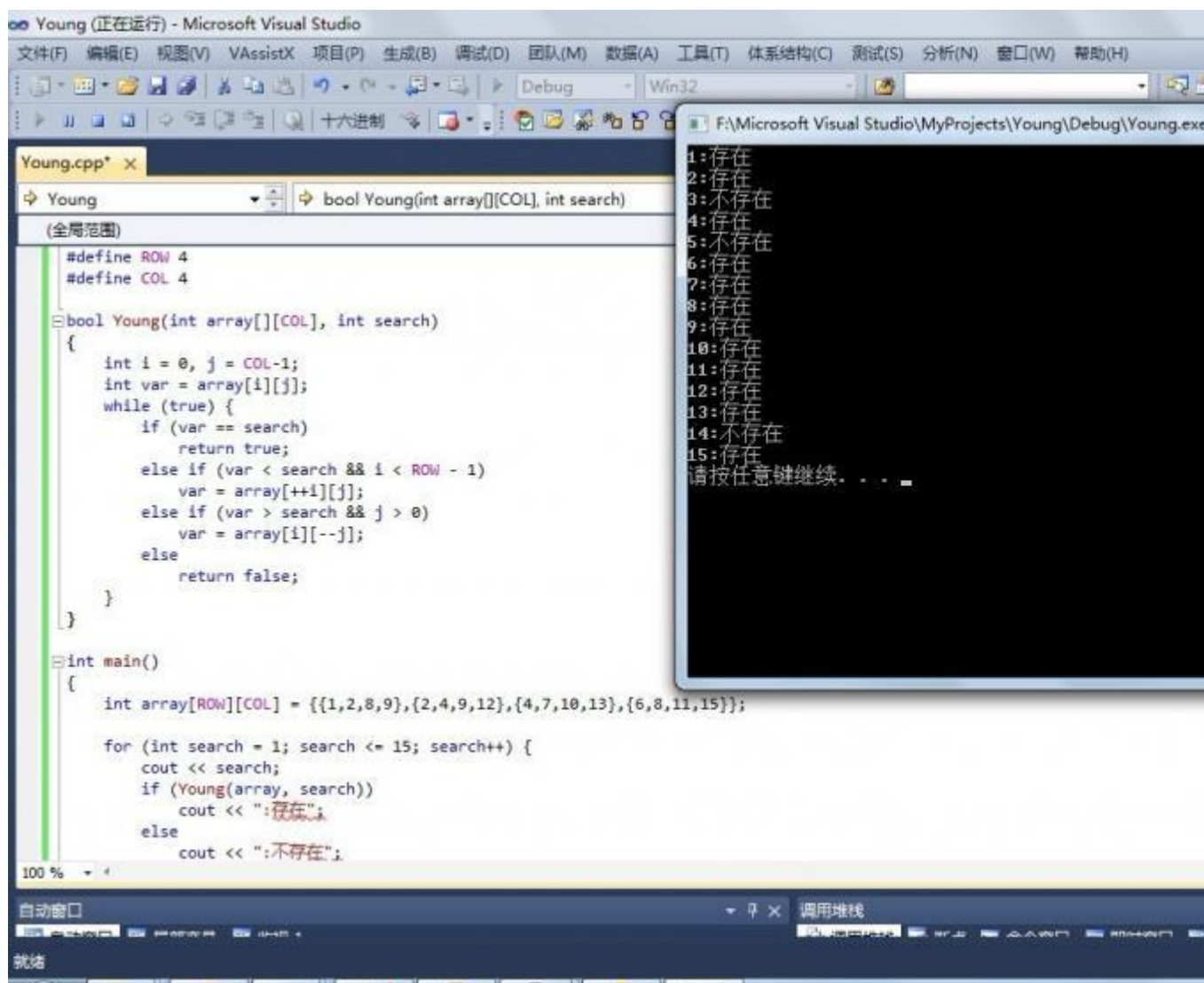


2、首先直接定位到最右上角的元素，再配以二分查找，比要找的数（6）大就往左走，比要找数（6）的小就往下走，直到找到要找的数字（6）为止，如下图所示：



1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

上述方法二的关键代码+程序运行如下图所示：



试问，上述算法复杂么？不复杂，只要稍微动点脑筋便能想到，还可以参看友人老梦的文章，Young 氏矩阵：<http://blog.csdn.net/zhanglei8893/article/details/6234564>，以及

IT 练兵场的：<http://www.jobcoding.com/array/matrix/young-tableau-problem/>，除此之外，何海涛先生一书剑指 offer 中也收集了此题，感兴趣的朋友也可以去看看。

## 十一（续）、倒排索引关键词 Hash 不重复编码实践

本章要介绍这样一个问题，对倒排索引中的关键词进行编码。那么，这个问题将分为两个个步骤：

1. 首先，要提取倒排索引内词典文件中的关键词；
2. 对提取出来的关键词进行编码。本章采取 hash 编码的方式。既然要用 hash 编码，那么最重要的就是要解决 hash 冲突的问题，下文会详细介绍。

有一点必须提醒读者的是，**倒排索引包含词典和倒排记录表两个部分**，词典一般有词项（或称为关键词）和词项频率（即这个词项或关键词出现的次数），倒排记录表则记录着上述词项（或关键词）所出现的位置，或出现的文档及网页 ID 等相关信息。

### 1、正排索引与倒排索引

咱们先来看什么是倒排索引，以及倒排索引与正排索引之间的区别：

我们知道，搜索引擎的关键步骤就是建立倒排索引，所谓倒排索引一般表示为一个关键词，然后是它的频度（出现的次数），位置（出现在哪一篇文章或网页中，及有关的日期，作者等信息），它相当于为互联网上几千亿页网页做了一个索引，好比一本书的目录、标签一般。读者想看哪一个主题相关的章节，直接根据目录即可找到相关的页面。不必再从书的第一页到最后一页，一页一页的查找。

接下来，阐述下正排索引与倒排索引的区别：

### 一般索引（正排索引）

正排表是以文档的 ID 为关键字，表中记录文档中每个字的位置信息，查找时扫描表中每个文档中字的信息直到找出所有包含查询关键字的文档。正排表结构如图 1 所示，这种组织方法在建立索引的时候结构比较简单，建立比较方便且易于维护；因为索引是基于文档建立的，若有新的文档假如，直接为该文档建立一个新的索引块，挂接在原来索引文件的后面。若有文档删除，则直接找到该文档号文档对应的索引信息，将其直接删除。但是在

查询的时候需对所有的文档进行扫描以确保没有遗漏，这样就使得检索时间大大延长，检索效率低下。

尽管正排表的工作原理非常的简单，但是由于其检索效率太低，除非在特定情况下，否则实用性价值不大。

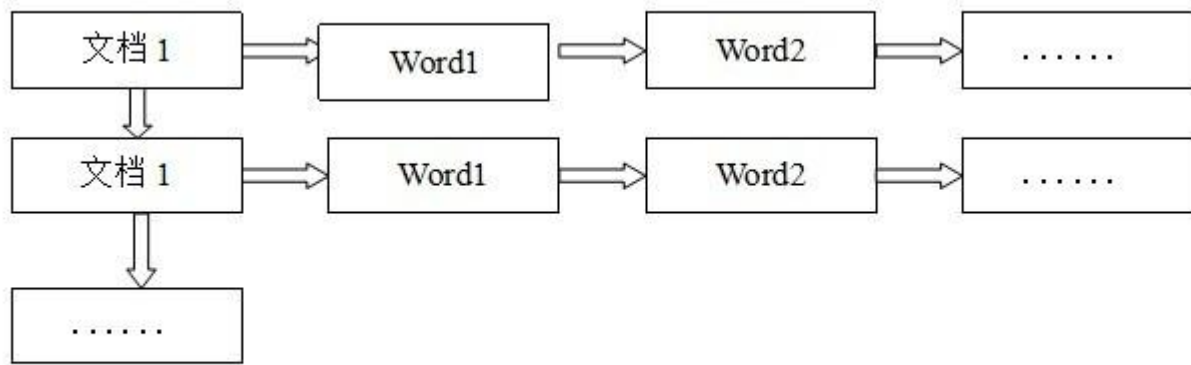


图 1 正排表结构图

## 倒排索引

倒排表以字或词为关键字进行索引，表中关键字所对应的记录表项记录了出现这个字或词的所有文档，一个表项就是一个字表段，它记录该文档的 ID 和字符在该文档中出现的位置情况。由于每个字或词对应的文档数量在动态变化，所以倒排表的建立和维护都较为复杂，但是在查询的时候由于可以一次得到查询关键字所对应的所有文档，所以效率高于正排表。在全文检索中，检索的快速响应是一个最为关键的性能，而索引建立由于在后台进行，尽管效率相对低一些，但不会影响整个搜索引擎的效率。

倒排表的结构图如图 2:

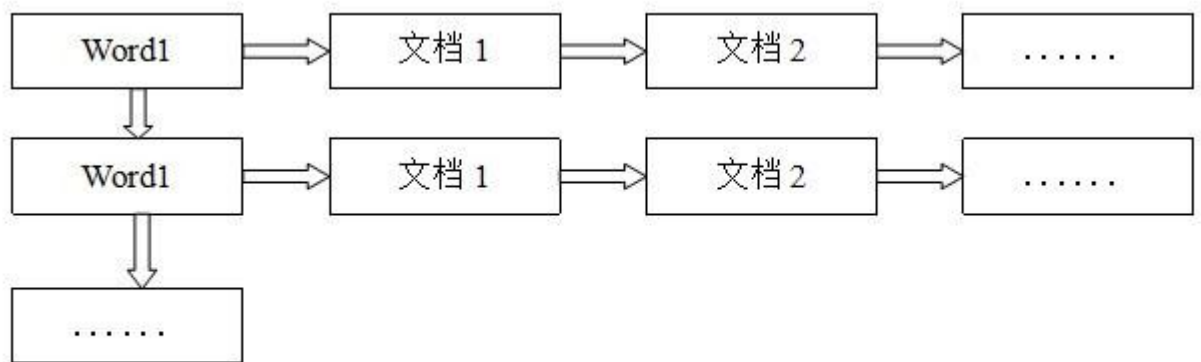


图 2 一倒排表结构图

倒排表的索引信息保存的是字或词后继数组模型、互关联后继数组模型条在文档内的位置，在同一篇文档内相邻的字或词条的前后关系没有被保存到索引文件内。

## 2、倒排索引中提取关键词

倒排索引是搜索引擎之基石。建成了倒排索引后，用户要查找某个 query，如在搜索框输入某个关键词：“结构之法”后，搜索引擎不会再次使用爬虫又一个一个去抓取每一个网页，从上到下扫描网页，看这个网页有没有出现这个关键词，而是会在它预先生成的倒排索引文件中查找和匹配包含这个关键词“结构之法”的所有网页。找到了之后，再按相关性度排序，最终把排序后的结果显示给用户。



搜索

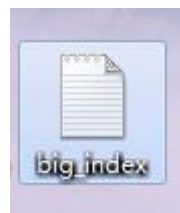
找到约 33,100,000 条结果 (用时 0.14 秒)

所有结果

[结构之法算法之道- 博客频道- CSDN.NET](#)  
[blog.csdn.net/v\\_JULY\\_v?utm\\_source=weibolife](#) - 网页快照  
 置顶]程序员面试、算法研究、编程艺术、红黑树4大系列集锦与总结. 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--[结构之法算法之 ...](#)

图片

地图



如下，即是一个倒排索引文件（不全），我们把它取名为 big\_index，

文件中每一较短的，不包含有“#####”符号的便是某个关键词，及这个关键词的出现次数。  
现在要从这个大索引文件中提取出这些关键词，--Firelff--，-11，-Winter-，..，007，007：  
天降杀机，02Chan.. 如何做到呢？一行一行的扫描整个索引文件么？

何意？之前已经说过：倒排索引包含词典和倒排记录表两个部分，词典一般有词项（或称为关键词）和词项频率（即这个词项或关键词出现的次数），倒排记录表则记录着上述词项（或关键词）所出现的位置，或出现的文档及网页 ID 等相关信息。

最简单的讲，就是要提取词典中的词项（关键词）：--Firelff--，-11，-Winter-，..，  
007，007：天降杀机，02Chan...

--Firelff--（关键词） 8（出现次数）

	1q	2q	3q	4q	5q	6q	7q	8q	9q	10q	11q	12q		
1	--Firelff--	8												
2	20111116	2	T0011111600004603	240	00	240	a2f6dc51c6f5231770cfd1ee8eaecc376	85358	1#####	T0011111600004604	240	00	240	219a54675
3	20111115	5	T0011111500008241	240	00	240	36c7627ea4eb9e6b31af9217435b55da	131221	1#####	T0011111500007324	240	00	240	201565a
4	20111114	1	T0011111400004894	240	00	240	6d6a715558a5e67b815724af6050bbec1	123028	1#####					
5	20111112	1	T0011111200001071	240	00	240	f11bbb2063fac37b08a87175f80c1e47	84124	1#####					
6	20111111	5	T0011111100003099	240	00	240	ed2a41c4011cfec8aa237e8f74234a6	94256	1#####	T0011111100003741	240	00	240	ad265e927
7	20111110	9	T0011111000011295	0	00	240	99778567b5c10ace0097195e0534c49b	160408	1#####	T0011111000010700	240	00	240	23037e2c5
8	20111102	4	T0011110200007030	48	00	240	c84be7f5dc7d58aacd6923a87cd33e20	140003	1#####	T0011110200003672	48	00	240	4776396
9	20111101	2	T0011110100010412	48	00	240	f0e4d2f08235b2dcd77cc7a20e45ebd1b	163759	1#####	T0011110100010411	48	00	240	e6c572a
10	-11	1												
11	20111115	3	PV011111500017163	156	00	156	e9cc79096791d0325457ada45479c574	204455	1#####	PV011111500017162	156	00	156	e9cc790
12	-Winter-	1												
13	20111109	2	PV0111110900013663	120	00	156	6be088cc820436aa5b37f5be6cfce0309	205723	1#####	PV0111110900008419	120	00	156	6be088c
14	14													
15	20111116	1	TH011111600017185	317	00	317	ec8485e3c1eb234edbd06b049b7a167a	143222	1#####					
16	20111115	1	TH011111500005035	317	00	317	c6db57636d17284704c9ce3bf377c91d	93353	1#####					
17	20111114	2	TH011111400025995	317	00	317	eb142b74e9d50755f9aa9ae190a23f8	224200	1#####	TH011111400024634	317	00	317	923e7da
18	20111113	2	TH011111300002493	317	00	317	25b70ae531a216410b1b3352c8855583	162256	1#####	TH011111300001507	317	00	317	6c2f90f
19	20111110	1	TH011111000006082	0	00	317	f8269a29d6e6d4209508f7a24c2e04cc	112537	1#####					
20	20111108	1	TH0111110800008740	274	00	317	adea53908a65ad0eed87a8e3cace69cfe	151951	1#####					
21	20111107	1	TH0111110800008493	274	00	317	a4f480042169709aa7646f06842d9dc4	175644	1#####					
22	20111104	2	TH0111110800008495	274	00	317	9f51b969f50c48e8223ad7ed2eaa15	223600	1#####	TH0111110800008496	274	00	317	85fb5f9
23	20111102	1	TH0111110800008497	274	00	317	8ae6ff54d5cf8a32a9ada14a0689173e	175400	1#####					
24	20111101	1	TH0111110800008498	274	00	317	45234e0436f68e613b9376a12e454918	145000	1#####					
25	20111031	3	TH0111110800008499	274	00	317	9216c116405317fd0465679696e378e6	165400	1#####	TH0111110800008500	274	00	317	5c16e1a
26	20111028	1	TH0111110800008502	274	00	317	554fc8284454da98c91cc7ad8cf3c7b1	171000	1#####					
27	20111024	1	TH0111110800008503	274	00	317	e242db640dd18b198d28acfd00347eaca	85800	1#####					
28	20111022	1	TH0111110800008504	274	00	317	9190867388f627e01ad4035e036e8b61	105200	1#####					
29	007	1												
30	20111127	1	TB1111112700002087	369	B1	172	809439b6900122369318fa3ae1e43e78	43100	1#####					
31	007：天降杀机	2												
32	20111127	1	TB1111112700000280	389	B1	2	724c5d3cca8e46a2a2a25d9a81fe5268	91342	1#####					
33	20111126	1	TB1111112600006933	282	B1	282	0384aaa4a0900912a845f1b5c905e89	142910	1#####					
34	02Chan	11												
35	20111118	5	T0011111600020843	225	00	225	d3aeec614bc0f8a4bb44dd220a03f	173011	1#####	T0011111600020844	225	00	225	ab7126f
36	20111115	7	T0011111500013121	225	00	225	ae51779e970ba3850c9b8b35a3eb2be	150157	1#####	T0011111500013122	225	00	225	ac5f54c
37	20111114	2	T0011111500008407	225	00	225	eb5cab7e26c95aa8e586b0fc73122f7	233516	1#####	T0011111500008408	225	00	225	e696f2a
38	20111111	2	T0011111100012702	225	00	225	a5e0212e66338f3d128a8de2c984445	191540	1#####	T0011111100003010	225	00	225	1d38ea7
39	20111109	2	T00111110900013032	108	00	225	2c496ae2226389db1e090a98e406de86	172640	1#####	T00111110900004817	108	00	225	4a292e2



我们可以试着这么解决：通过查找#####便可判断某一行出现的词是不是关键词，但如果这样做的话，便要扫描整个索引文件的每一行，代价实在巨大。如何提高速度呢？对了，关键词后面的那个出现次数为我们问题的解决起到了很好的作用，如下注释所示：

```
//      本身没有##### 的行判定为关键词行，后跟这个关键词的行数 N（即词项频率）
//      接下来，截取关键词--Firelf--，然后读取后面关键词的行数 N
//      再跳过 N 行（滤过和避免扫描中间的倒排记录表信息）
//      读取下一个关键词..
```

有朋友指出，上述方法虽然减少了扫描的行数，但并没有减少 IO 开销。读者是否有更好的办法？欢迎随时交流。

## 34.2、为提取出来的关键词编码

爱思考的朋友可能会问，上述从倒排索引文件中提取出那些关键词（词项）的操作是为了什么呢？其实如我个人微博上 12 月 12 日所述的 Hash 词典编码：

词典文件的编码：1、词典怎么生成（存储和构造词典）；2、如何运用 hash 对输入的汉字进行编码；3、如何更好的解决冲突，即不重复以及追加功能。具体例子为：事先构造好词典文件后，输入一个词，要求找到这个词的编码，然后将其编码输出。且要有不断能添加词的功能，不得重复。

步骤应该是如下：1、读索引文件；2、提取索引中的词出来；3、词典怎么生成，存储和构造词典；4、词典文件的编码：不重复与追加功能。编码比如，输入中国，他的编码可以为 10001，然后输入银行，他的编码可以为 10002。只要实现不断添加词功能，以及不重复即可，词典类的大文件，hash 最重要的是怎样避免冲突。

也就是说，现在我要对上述提取出来后的关键词进行编码，采取何种方式编码呢？暂时用 hash 函数编码。编码之后的效果将是每一个关键词都有一个特定的编码，如下图所示（与上文 big\_index 文件比较一下便知）：

```
--Firelf--    对应编码为：135942

-11           对应编码为：106101

....
```

```

0 10 20 30 40 50 60 70 80 90 100 110 120
1 --Fireelf-- 135942
2 --11 106101
3 --Winter-- 90114
4 . 140059
5 007 106205
6 007: 天降杀机 37634
7 02Chan 8200
8 08:30 194788
9 09:34 195980
10 09:43 195981
11 10+10 25270
12 10086 25452
13 10:33 25797
14 10月团购报告 193308
15 11:14 26987
16 11月8日 30982
17 12580生活播报 18158
18 139说客 97835
19 14:51 30587
20 15:19 31755
21 1626潮流双周刊 100430
22 163 106246
23 163邮箱 142794
24 17173 33830
25 178游戏网 119117
26 1881 138811
27 1967-liu 28210
28 1983组合龙飞龙 53522
29 199it 38544
30 lqing 104981
31 1号店 111818
32 2001国足十强赛全场 120106
33 2010年新疆gdp 101348
34 2011-11-01 41963
35 2011-11-02 41963
36 2011-11-03 41963
37 2011-11-04 41963
38 2011-11-05 41963
39 2011-11-07 41963

```

但细心的朋友一看上图便知，其中第 34~39 行显示，有重复的编码，那么如何解决这个不重复编码的问题呢？

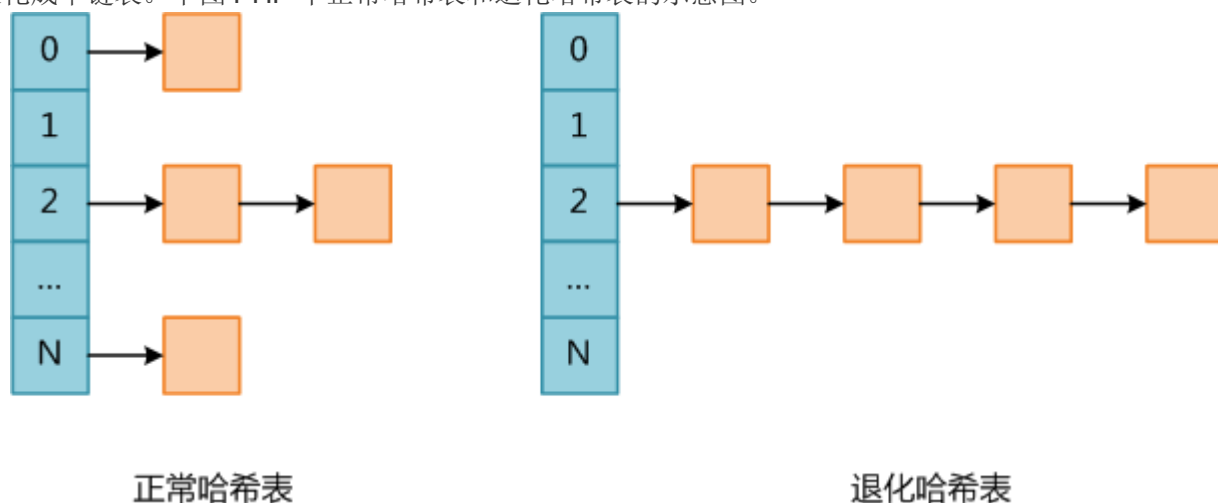
用 hash 表编码？但其极易产生冲突碰撞，为什么？请看：

哈希表是一种查找效率极高的数据结构，很多语言都在内部实现了哈希表。PHP 中的哈希表是一种极为重要的数据结构，不但用于表示 **Array** 数据类型，还在 **Zend** 虚拟机内部用于存储上下文环境信息（执行上下文的变量及函数均使用哈希表结构存储）。

理想情况下哈希表插入和查找操作的时间复杂度均为  $O(1)$ ，任何一个数据项可以在一个与哈希表长度无关的时间内计算出一个哈希值（**key**），然后在常量时间内定位到一个桶（术语 **bucket**，表示哈希表中的一个位置）。当然这是理想情况下，因为任何哈希表的长度都是有限的，所以一定存在不同的数据项具有相同哈希值的情况，此时不同数据项被定为到同一个桶，称为碰撞（**collision**）。哈希表的实现需要解决碰撞问题，碰撞解决大体有两种思路，第一种是根据某种原则将被碰撞数据定为到其它桶，例如线性探测——如果数据在插入时发生了碰撞，则顺序查找这个桶后面的桶，将其放入第一个没有被使用的桶；第二种策略是每个桶不是一个只能容纳单个数据项的位置，而是一个可容纳多个数据的数据结构（例如链表或红黑树），所有碰撞的数据以某种数据结构的形式组织起来。

不论使用了哪种碰撞解决策略，都导致插入和查找操作的时间复杂度不再是  $O(1)$ 。以查找为例，不能通过 **key** 定位到桶就结束，必须还要比较原始 **key**（即未做哈希之前的 **key**）是否相等，如果不相等，则要使用与插入相同的算法继续查找，直到找到匹配的值或确认数据不在哈希表中。

PHP 是使用单链表存储碰撞的数据，因此实际上 PHP 哈希表的平均查找复杂度为  $O(L)$ ，其中  $L$  为桶链表的平均长度；而最坏复杂度为  $O(N)$ ，此时所有数据全部碰撞，哈希表退化成单链表。下图 PHP 中正常哈希表和退化哈希表的示意图。



哈希表碰撞攻击就是通过精心构造数据，使得所有数据全部碰撞，人为将哈希表变成一个退化的单链表，此时哈希表各种操作的时间均提升了一个数量级，因此会消耗大量 CPU 资源，导致系统无法快速响应请求，从而达到拒绝服务攻击（DoS）的目的。

可以看到，进行哈希碰撞攻击的前提是哈希算法特别容易找出碰撞，如果是 MD5 或者 SHA1 那基本就没戏了，幸运的是（也可以说不幸的是）大多数编程语言使用的哈希算法都十分简单（这是为了效率考虑），因此可以不费吹灰之力构造出攻击数据（引自：<http://www.codinglabs.org/html/hash-collisions-attack-on-php.html>）。

### 3、暴雪的 Hash 算法

值得一提的是，在解决 Hash 冲突的时候，搞的焦头烂额，结果今天上午在自己的博客内的一篇文章（十一、从头到尾彻底解析 Hash 表算法）内找到了解决办法：网上流传甚广的暴雪的 Hash 算法。 OK，接下来，咱们回顾下暴雪的 hash 表算法：

“接下来，咱们来具体分析一下一个最快的 Hash 表算法。

我们由一个简单的问题逐步入手：有一个庞大的字符串数组，然后给你一个单独的字符串，让你从这个数组中查找是否有这个字符串并找到它，你会怎么做？

有一个方法最简单，老老实实从头查到尾，一个一个比较，直到找到为止，我想只要学过程序设计的人都能把这样一个程序作出来，但要是程序员把这样的程序交给用户，我只能用无语来评价，或许它真的能工作，但...也只能如此了。

最合适的算法自然是使用 HashTable(哈希表)，先介绍介绍其中的基本知识，所谓 Hash，一般是一个整数，通过某种算法，可以把一个字符串“压缩”成一个整数。当然，无论如何，一个 32 位整数是无法对应回一个字符串的，但在程序中，两个字符串计算出的 Hash 值相等的可能非常小，下面看看在 MPQ 中的 Hash 算法：



函数 prepareCryptTable 以下的函数生成一个长度为 0x500（合 10 进制数：1280）的 cryptTable[0x500]

```
1.      //函数 prepareCryptTable 以下的函数生成一个长度为 0x500（合 10 进制数：1280）的
      cryptTable[0x500]
2.      void prepareCryptTable()
3.      {
4.          unsigned long seed = 0x00100001, index1 = 0, index2 = 0, i;
5.
6.          for( index1 = 0; index1 < 0x100; index1++ )
7.          {
8.              for( index2 = index1, i = 0; i < 5; i++, index2 += 0x100 )
9.              {
10.                 unsigned long temp1, temp2;
11.
12.                 seed = (seed * 125 + 3) % 0x2AAAAB;
13.                 temp1 = (seed & 0xFFFF) << 0x10;
14.
15.                 seed = (seed * 125 + 3) % 0x2AAAAB;
16.                 temp2 = (seed & 0xFFFF);
17.
18.                 cryptTable[index2] = ( temp1 | temp2 );
19.             }
20.         }
21.     }
```

函数 HashString 以下函数计算 lpszFileName 字符串的 hash 值,其中 dwHashType 为 hash 的类型,

```
1.      //函数 HashString 以下函数计算 lpszFileName 字符串的 hash 值,其中 dwHashType 为 hash
      的类型,
2.      unsigned long HashString(const char *lpszkeyName, unsigned long dwHashType )
3.      {
4.          unsigned char *key = (unsigned char *)lpszkeyName;
5.          unsigned long seed1 = 0x7FED7FED;
6.          unsigned long seed2 = 0xEEEEEEEE;
7.          int ch;
8.
9.          while( *key != 0 )
10.         {
11.             ch = *key++;
12.             seed1 = cryptTable[(dwHashType<<8) + ch] ^ (seed1 + seed2);
13.             seed2 = ch + seed1 + seed2 + (seed2<<5) + 3;
```

```

14.         }
15.         return seed1;
16.     }

```

Blizzard 的这个算法是非常高效的，被称为"One-Way Hash"( A one-way hash is a an algorithm that is constructed in such a way that deriving the original string (set of strings, actually) is virtually impossible)。举个例子，字符串"unitneutralacritter.grp"通过这个算法得到的结果是 0xA26067F3。

是不是把第一个算法改进一下，改成逐个比较字符串的 Hash 值就可以了呢，答案是，远远不够，要想得到最快的算法，就不能进行逐个的比较，通常是构造一个哈希表(Hash Table)来解决问题，哈希表是一个大数组，这个数组的容量根据程序的要求来定义，

例如 1024，每一个 Hash 值通过取模运算 (mod) 对应到数组中的一个位置，这样，只要比较这个字符串的哈希值对应的位置有没有被占用，就可以得到最后的结果了，想想这是什么速度？是的，是最快的  $O(1)$ ，现在仔细看看这个算法吧：

```

1.     typedef struct
2.     {
3.         int nHashA;
4.         int nHashB;
5.         char bExists;
6.         .....
7.     } SOMESTRUCTURE;
8.     //一种可能的结构体定义？

```

函数 GetHashTablePos 下述函数为在 Hash 表中查找是否存在目标字符串，有则返回要查找字符串的 Hash 值，无则，return -1.

```

1.     //函数 GetHashTablePos 下述函数为在 Hash 表中查找是否存在目标字符串，有则返回要查找字符串的 Hash 值，无则，return -1.
2.     int GetHashTablePos( har *lpszString, SOMESTRUCTURE *lpTable )
3.     //lpszString 要在 Hash 表中查找的字符串，lpTable 为存储字符串 Hash 值的 Hash 表。
4.     {
5.         int nHash = HashString(lpszString); //调用上述函数 HashString, 返回要查找字符串 lpszString 的 Hash 值。
6.         int nHashPos = nHash % nTableSize;
7.
8.         if ( lpTable[nHashPos].bExists && !strcmp( lpTable[nHashPos].pString, lpszString ) )
9.         { //如果找到的 Hash 值在表中存在，且要查找的字符串与表中对应位置的字符串相同，
10.             return nHashPos; //返回找到的 Hash 值
11.         }
12.         else
13.         {
14.             return -1;
15.         }
16.     }

```

看到此，我想大家都在想一个很严重的问题：“如果两个字符串在哈希表中对应的位置相同怎么办？”，毕竟一个数组容量是有限的，这种可能性很大。解决该问题的方法很多，我首

先想到的就是用“链表”,感谢大学里学的数据结构教会了这个百试百灵的法宝,我遇到的很多算法都可以转化成链表来解决,只要在哈希表的每个入口挂一个链表,保存所有对应的字符串就 OK 了。事情到此似乎有了完美的结局,如果是把问题独自交给我解决,此时我可能就要开始定义数据结构然后写代码了。

然而 Blizzard 的程序员使用的方法则是更精妙的方法。基本原理就是:他们在哈希表中不是用一个哈希值而是用三个哈希值来校验字符串。”

“MPQ 使用文件名哈希表来跟踪内部的所有文件。但是这个表的格式与正常的哈希表有一些不同。首先,它没有使用哈希作为下标,把实际的文件名存储在表中用于验证,实际上它根本就没有存储文件名。而是使用了 3 种不同的哈希:一个用于哈希表的下标,两个用于验证。这两个验证哈希替代了实际文件名。

当然了,这样仍然会出现 2 个不同的文件名哈希到 3 个同样的哈希。但是这种情况发生的概率平均是:1:18889465931478580854784,这个概率对于任何人来说应该都是足够小的。现在再回到数据结构上,Blizzard 使用的哈希表没有使用链表,而采用“顺延”的方式来解决问题。”下面,咱们来看看这个网上流传甚广的暴雪 hash 算法:

函数 GetHashTablePos 中,lpzString 为要在 hash 表中查找的字符串;lpTable 为存储字符串 hash 值的 hash 表;nTableSize 为 hash 表的长度:

```
1. //函数 GetHashTablePos 中,lpzString 为要在 hash 表中查找的字符串;lpTable 为存储
   字符串 hash 值的 hash 表;nTableSize 为 hash 表的长度:
2. int GetHashTablePos( char *lpzString, MPQHASHTABLE *lpTable, int nTableSize
   )
3. {
4.     const int HASH_OFFSET = 0, HASH_A = 1, HASH_B = 2;
5.
6.     int nHash = HashString( lpzString, HASH_OFFSET );
7.     int nHashA = HashString( lpzString, HASH_A );
8.     int nHashB = HashString( lpzString, HASH_B );
9.     int nHashStart = nHash % nTableSize;
10.    int nHashPos = nHashStart;
11.
12.    while ( lpTable[nHashPos].bExists )
13.    {
14.        // 如果仅仅是判断在该表中时候存在这个字符串,就比较这两个 hash 值就可以了,不用对
           结构体中的字符串进行比较。
15.        // 这样会加快运行的速度? 减少 hash 表占用的空间? 这种方法一般应用在什么场
           合?
16.        if ( lpTable[nHashPos].nHashA == nHashA
17.            && lpTable[nHashPos].nHashB == nHashB )
18.        {
19.            return nHashPos;
20.        }
```

```

21.         else
22.         {
23.             nHashPos = (nHashPos + 1) % nTableSize;
24.         }
25.
26.         if (nHashPos == nHashStart)
27.             break;
28.     }
29.     return -1;
30. }

```

上述程序解释：

- 1、计算出字符串的三个哈希值（一个用来确定位置，另外两个用来校验）
- 2、察看哈希表中的这个位置
- 3、哈希表中这个位置为空吗？如果为空，则肯定该字符串不存在，返回-1。
- 4、如果存在，则检查其他两个哈希值是否也匹配，如果匹配，则表示找到了该字符串，返回其 Hash 值。
- 5、移到下一个位置，如果已经移到了表的末尾，则反绕到表的开始位置起继续查询
- 6、看看是不是又回到了原来的位置，如果是，则返回没找到
- 7、回到 3。

#### 4、不重复 Hash 编码

有了上面的暴雪 Hash 算法。咱们的问题便可解决了。不过，有两点必须先提醒读者：

- 1、Hash 表起初要初始化；
- 2、暴雪的 Hash 算法对于查询那样处理可以，但对插入就不能那么解决。

关键主体代码如下：

```

1.     //函数 prepareCryptTable 以下的函数生成一个长度为 0x500（合 10 进制数：1280）的
    cryptTable[0x500]
2.     void prepareCryptTable()
3.     {
4.         unsigned long seed = 0x00100001, index1 = 0, index2 = 0, i;
5.
6.         for( index1 = 0; index1 < 0x100; index1++ )

```

```

7.         {
8.             for( index2 = index1, i = 0; i < 5; i++, index2 += 0x100)
9.             {
10.                unsigned long temp1, temp2;
11.                seed = (seed * 125 + 3) % 0x2AAAAB;
12.                temp1 = (seed & 0xFFFF)<<0x10;
13.                seed = (seed * 125 + 3) % 0x2AAAAB;
14.                temp2 = (seed & 0xFFFF);
15.                cryptTable[index2] = ( temp1 | temp2 );
16.            }
17.        }
18.    }
19.
20.    //函数 HashString 以下函数计算 lpszFileName 字符串的 hash 值,其中 dwHashType 为 hash
    的类型,
21.    unsigned long HashString(const char *lpszkeyName, unsigned long dwHashType )
22.    {
23.        unsigned char *key = (unsigned char *)lpszkeyName;
24.        unsigned long seed1 = 0x7FED7FED;
25.        unsigned long seed2 = 0xEEEEEEEE;
26.        int ch;
27.
28.        while( *key != 0 )
29.        {
30.            ch = *key++;
31.            seed1 = cryptTable[(dwHashType<<8) + ch] ^ (seed1 + seed2);
32.            seed2 = ch + seed1 + seed2 + (seed2<<5) + 3;
33.        }
34.        return seed1;
35.    }
36.
37.    //////////////////////////////////////
38.    //function: 哈希词典 编码
39.    //parameter:
40.    //author: lei.zhou
41.    //time: 2011-12-14
42.    //////////////////////////////////////
43.    MPQHASHTABLE TestHashTable[nTableSize];
44.    int TestHashCTable[nTableSize];
45.    int TestHashDTable[nTableSize];
46.    key_list test_data[nTableSize];
47.
48.    //直接调用上面的 hashstring, nHashPos 就是对应的 HASH 值。

```

```

49.     int insert_string(const char *string_in)
50.     {
51.         const int HASH_OFFSET = 0, HASH_C = 1, HASH_D = 2;
52.         unsigned int nHash = HashString(string_in, HASH_OFFSET);
53.         unsigned int nHashC = HashString(string_in, HASH_C);
54.         unsigned int nHashD = HashString(string_in, HASH_D);
55.         unsigned int nHashStart = nHash % nTableSize;
56.         unsigned int nHashPos = nHashStart;
57.         int ln, ires = 0;
58.
59.         while (TestHashTable[nHashPos].bExists)
60.         {
61.             //      if (TestHashCTable[nHashPos] == (int) nHashC && TestHashDTable[nHas
62.             //      break;
63.             //      ...
64.             //      else
65.             //如之前所提示读者的那般，暴雪的 Hash 算法对于查询那样处理可以，但对插入就不
能那么解决
66.                 nHashPos = (nHashPos + 1) % nTableSize;
67.
68.                 if (nHashPos == nHashStart)
69.                     break;
70.         }
71.
72.         ln = strlen(string_in);
73.         if (!TestHashTable[nHashPos].bExists && (ln < nMaxStrLen))
74.         {
75.             TestHashCTable[nHashPos] = nHashC;
76.             TestHashDTable[nHashPos] = nHashD;
77.
78.             test_data[nHashPos] = (KEYNODE *) malloc (sizeof(KEYNODE) * 1);
79.             if(test_data[nHashPos] == NULL)
80.             {
81.                 printf("10000 EMS ERROR !!!!\n");
82.                 return 0;
83.             }
84.
85.             test_data[nHashPos]->pkey = (char *)malloc(ln+1);
86.             if(test_data[nHashPos]->pkey == NULL)
87.             {
88.                 printf("10000 EMS ERROR !!!!\n");
89.                 return 0;
90.             }

```

```

91.
92.         memset(test_data[nHashPos]->pkey, 0, ln+1);
93.         strncpy(test_data[nHashPos]->pkey, string_in, ln);
94.         *((test_data[nHashPos]->pkey)+ln) = 0;
95.         test_data[nHashPos]->weight = nHashPos;
96.
97.         TestHashTable[nHashPos].bExists = 1;
98.     }
99.     else
100.    {
101.        if(TestHashTable[nHashPos].bExists)
102.            printf("30000 in the hash table %s !!!\n", string_in);
103.        else
104.            printf("90000 strkey error !!!\n");
105.    }
106.    return nHashPos;
107. }

```

接下来要读取索引文件 big\_index 对其中的关键词进行编码（为了简单起见，直接一行一行扫描读写，没有跳过行数了）：

```

1.     void bigIndex_hash(const char *docpath, const char *hashpath)
2.     {
3.         FILE *fr, *fw;
4.         int len;
5.         char *pbuf, *p;
6.         char dockey[TERM_MAX LENG];
7.
8.         if(docpath == NULL || *docpath == '\0')
9.             return;
10.
11.        if(hashpath == NULL || *hashpath == '\0')
12.            return;
13.
14.        fr = fopen(docpath, "rb"); //读取文件 docpath
15.        fw = fopen(hashpath, "wb");
16.        if(fr == NULL || fw == NULL)
17.        {
18.            printf("open read or write file error!\n");
19.            return;
20.        }
21.
22.        pbuf = (char*)malloc(BUFF_MAX LENG);
23.        if(pbuf == NULL)

```

```

24.     {
25.         fclose(fr);
26.         return ;
27.     }
28.
29.     memset(pbuf, 0, BUFF_MAX LENG);
30.
31.     while(fgets(pbuf, BUFF_MAX LENG, fr))
32.     {
33.         len = GetRealString(pbuf);
34.         if(len <= 1)
35.             continue;
36.         p = strstr(pbuf, "#####");
37.         if(p != NULL)
38.             continue;
39.
40.         p = strstr(pbuf, " ");
41.         if (p == NULL)
42.         {
43.             printf("file contents error!");
44.         }
45.
46.         len = p - pbuf;
47.         dockey[0] = 0;
48.         strncpy(dockey, pbuf, len);
49.
50.         dockey[len] = 0;
51.
52.         int num = insert_string(dockey);
53.
54.         dockey[len] = ' ';
55.         dockey[len+1] = '\0';
56.         char str[20];
57.         itoa(num, str, 10);
58.
59.         strcat(dockey, str);
60.         dockey[len+strlen(str)+1] = '\0';
61.         fprintf (fw, "%s\n", dockey);
62.
63.     }
64.     free(pbuf);
65.     fclose(fr);
66.     fclose(fw);
67. }

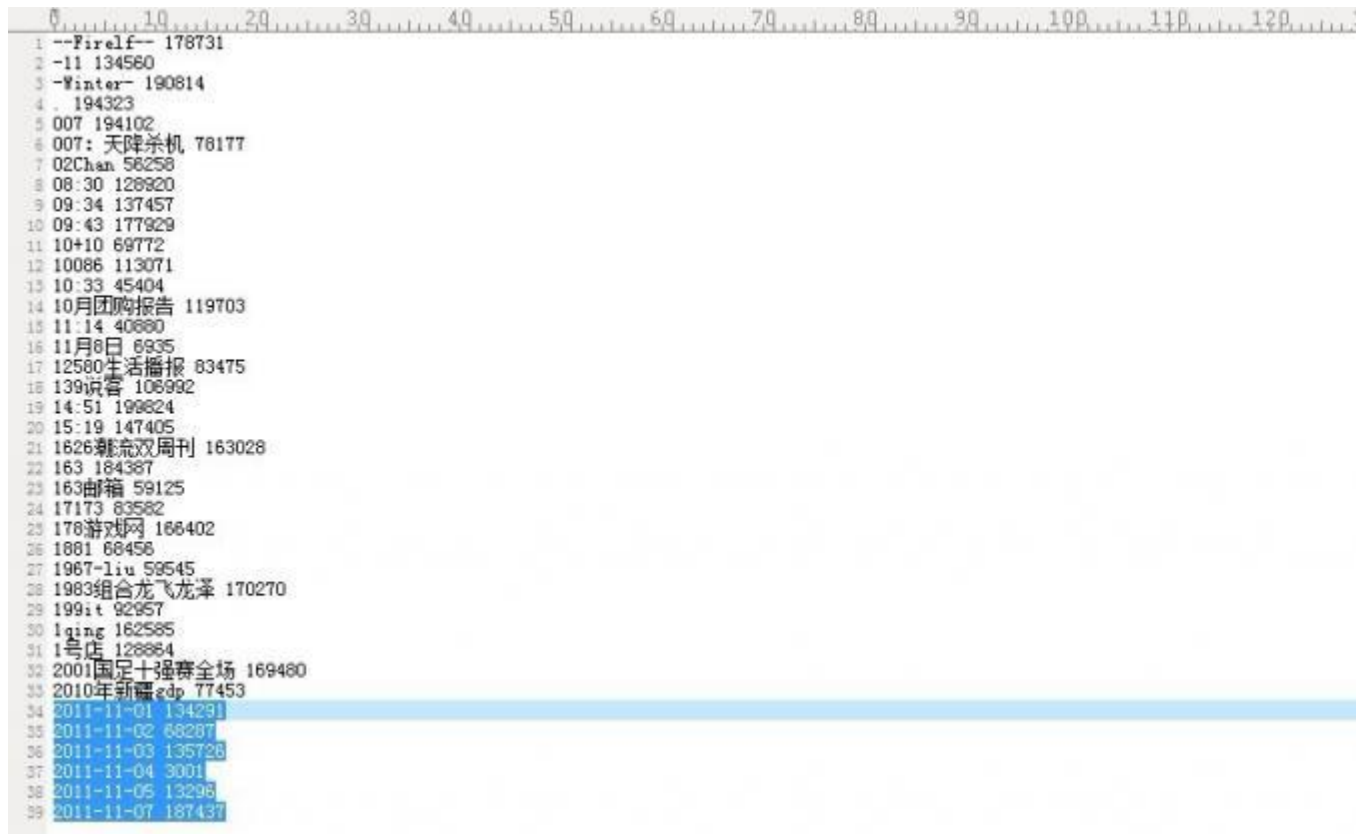
```



主函数已经很简单了，如下：

```
1.     int main()
2.     {
3.         prepareCryptTable(); //Hash 表起初要初始化
4.
5.         //现在要把整个 big_index 文件插入 hash 表，以取得编码结果
6.         bigIndex_hash("big_index.txt", "hashpath.txt");
7.         system("pause");
8.
9.         return 0;
10.    }
```

程序运行后生成的 hashpath.txt 文件如下：

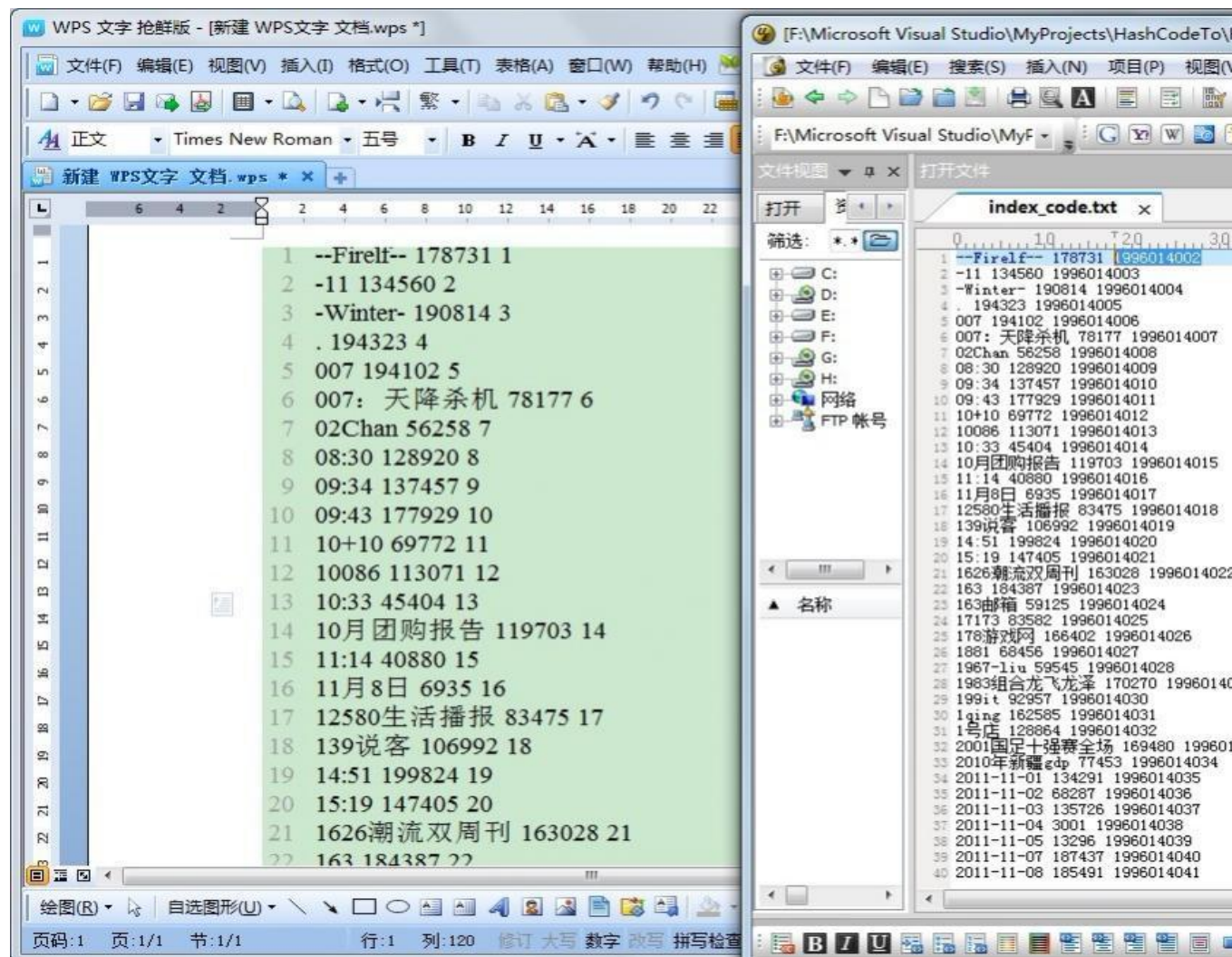


```
0 10 20 30 40 50 60 70 80 90 100 110 120
1 --Firelf-- 178731
2 -11 134580
3 -Winter- 190814
4 . 194323
5 007 194102
6 007: 天降杀机 76177
7 02Chan 56258
8 08:30 128920
9 09:34 137457
10 09:43 177929
11 10+10 69772
12 10086 113071
13 10:33 45404
14 10月团购报告 119703
15 11:14 40880
16 11月8日 6935
17 12580生活播报 83475
18 139说客 108992
19 14:51 199824
20 15:19 147405
21 1626潮流双周刊 163028
22 163 184387
23 163曲稿 59125
24 17173 83582
25 178游戏网 166402
26 1881 68458
27 1987-liu 58545
28 1983组合龙飞龙泽 170270
29 199it 92957
30 lqing 162585
31 1号店 128864
32 2001国足十强赛全场 169480
33 2010年新增sdp 77453
34 2011-11-01 134291
35 2011-11-02 68287
36 2011-11-03 135728
37 2011-11-04 3001
38 2011-11-05 13296
39 2011-11-07 187437
```

如上所示，采取暴雪的 Hash 算法并在插入的时候做适当处理，当再次对上文中的索引文件 big\_index 进行 Hash 编码后，冲突问题已经得到初步解决。当然，还有待更进一步更深入的测试。

后续添上数目索引 1~10000...

后来又为上述文件中的关键词编了码一个计数的内码，不过，奇怪的是，同样的代码，在 Dev C++ 与 VS2010 上运行结果却不同（左边 dev 上计数从“1”开始，VS 上计数从“1994014002”开始），如下图所示：



在上面的 bigIndex\_hashcode 函数的基础上，修改如下，即可得到上面的效果：

```

1.     void bigIndex_hashcode(const char *in_file_path, const char *out_file_path)
2.     {
3.         FILE *fr, *fw;
4.         int len, value;
5.         char *pbuf, *pleft, *p;
6.         char keyvalue[TERM_MAX LENG], str[WORD_MAX LENG];
7.
8.         if(in_file_path == NULL || *in_file_path == '\0') {
9.             printf("input file path error!\n");
10.            return;

```

```

11.     }
12.
13.     if(out_file_path == NULL || *out_file_path == '\0') {
14.         printf("output file path error!\n");
15.         return;
16.     }
17.
18.     fr = fopen(in_file_path, "r"); //读取 in_file_path 路径文件
19.     fw = fopen(out_file_path, "w");
20.
21.     if(fr == NULL || fw == NULL)
22.     {
23.         printf("open read or write file error!\n");
24.         return;
25.     }
26.
27.     pbuf = (char*)malloc(BUFF_MAX LENG);
28.     pleft = (char*)malloc(BUFF_MAX LENG);
29.     if(pbuf == NULL || pleft == NULL)
30.     {
31.         printf("allocate memory error!");
32.         fclose(fr);
33.         return ;
34.     }
35.
36.     memset(pbuf, 0, BUFF_MAX LENG);
37.
38.     int offset = 1;
39.     while(fgets(pbuf, BUFF_MAX LENG, fr))
40.     {
41.         if (--offset > 0)
42.             continue;
43.
44.         if(GetRealString(pbuf) <= 1)
45.             continue;
46.
47.         p = strstr(pbuf, "#####");
48.         if(p != NULL)
49.             continue;
50.
51.         p = strstr(pbuf, " ");
52.         if (p == NULL)
53.         {
54.             printf("file contents error!");

```

```

55.         }
56.
57.         len = p - pbuf;
58.
59.         // 确定跳过行数
60.         strcpy(pleft, p+1);
61.         offset = atoi(pleft) + 1;
62.
63.         strncpy(keyvalue, pbuf, len);
64.         keyvalue[len] = '\0';
65.         value = insert_string(keyvalue);
66.
67.         if (value != -1) {
68.
69.             // key value 中插入空格
70.             keyvalue[len] = ' ';
71.             keyvalue[len+1] = '\0';
72.
73.             itoa(value, str, 10);
74.             strcat(keyvalue, str);
75.
76.             keyvalue[len+strlen(str)+1] = ' ';
77.             keyvalue[len+strlen(str)+2] = '\0';
78.
79.             keysize++;
80.             itoa(keysize, str, 10);
81.             strcat(keyvalue, str);
82.
83.             // 将 key value 写入文件
84.             fprintf (fw, "%s\n", keyvalue);
85.
86.         }
87.     }
88.     free(pbuf);
89.     fclose(fr);
90.     fclose(fw);
91. }

```

## 小结

本文有一点值得一提的是，在此前的这篇文章（十一、从头到尾彻底解析 Hash 表算法）之中，只是对 Hash 表及暴雪的 Hash 算法有过学习和了解，但尚未真正运用过它，而今在本章中体现，证明还是之前写的文章，及之前对 Hash 表等算法的学习还是有一定作用的。

同时，也顺便对暴雪的 Hash 函数算是做了个测试，其的确能解决一般的冲突性问题，创造这个算法的人不简单呐。

## 后记

再次感谢老大 xiaoqi，以及艺术室内朋友 xiaolin，555，yansha 的指导。没有他们的帮助，我将寸步难行。日后，自己博客内的文章要经常回顾，好好体会。同时，写作本文时，刚接触倒排索引等相关问题不久，若有任何问题，欢迎随时交流或批评指正。

最后，基于本 blog 的分为程序语言，数据结构，算法讨论，面试题库，编程技巧五大板块交流的论坛正在加紧建设当中（总负责人：scott && yinhex && 网络骑士），相信不久以后便会与大家见面。谢谢。完。

## 十二、快速排序算法

作者 July 二零一一年一月四日

-----

写此八大排序算法系列之前，先说点题外话。

每写一篇文章，我都会遵循以下几点原则：

- 一、保持版面的尽量清晰，力保排版良好。
- 二、力争所写的东西，清晰易懂，图文并茂
- 三、尽最大可能确保所写的东西精准，有实用价值。

因为，我觉得，你既然要把你的文章，公布出来，那么你就一定要为你的读者负责。不然，就不要发表出来。一切，为读者服务。

ok，闲不多说。接下来，咱们立刻进入本文章的主题，排序算法。

众所周知，快速排序算法是排序算法中的重头戏。

因此，本系列，本文就从快速排序开始。

-----

### 一、快速排序算法的基本特性

时间复杂度： $O(n \lg n)$

最坏： $O(n^2)$

空间复杂度:  $O(n \lg n)$

不稳定。

快速排序是一种排序算法，对包含  $n$  个数的输入数组，平均时间为  $O(n \lg n)$ ，最坏情况是  $O(n^2)$ 。

通常是用于排序的最佳选择。因为，排序最快，也只能达到  $O(n \lg n)$ 。

## 二、快速排序算法的描述

算法导论，第 7 章

快速排序时基于分治模式处理的，

对一个典型子数组  $A[p..r]$  排序的分治过程为三个步骤：

1. 分解：

$A[p..r]$  被划分为两个（可能空）的子数组  $A[p..q-1]$  和  $A[q+1..r]$ ，使得

$A[p..q-1] \leq A[q] \leq A[q+1..r]$

2. 解决：通过递归调用快速排序，对子数组  $A[p..q-1]$  和  $A[q+1..r]$  排序。

3. 合并。

## 三、快速排序算法

版本一：

QUICKSORT( $A, p, r$ )

```
1 if  $p < r$ 
2   then  $q \leftarrow \text{PARTITION}(A, p, r)$  //关键
3     QUICKSORT( $A, p, q - 1$ )
4     QUICKSORT( $A, q + 1, r$ )
```

数组划分

快速排序算法的关键是 PARTITION 过程，它对  $A[p..r]$  进行就地重排：

PARTITION( $A, p, r$ )

```
1  $x \leftarrow A[r]$ 
2  $i \leftarrow p - 1$ 
3 for  $j \leftarrow p$  to  $r - 1$ 
4   do if  $A[j] \leq x$ 
5     then  $i \leftarrow i + 1$ 
```

```

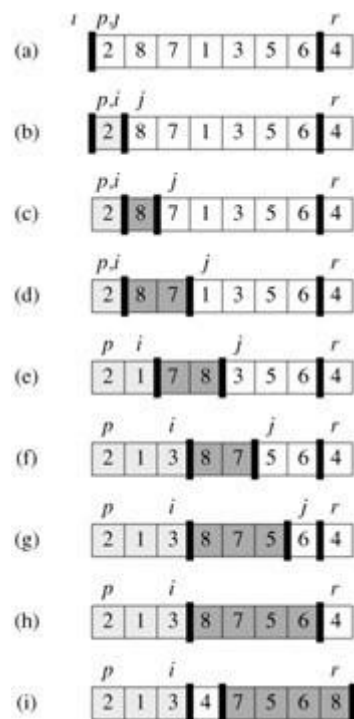
6         exchange A[i] <-> A[j]
7  exchange A[i + 1] <-> A[r]
8  return i + 1

```

ok，咱们来举一个具体而完整的例子。

来对以下数组，进行快速排序，

**2 8 7 1 3 5 6 4(主元)**



一、

i p/j

**2 8 7 1 3 5 6 4(主元)**

j 指的  $2 \leq 4$ ，于是  $i++$ ，i 也指到 2，2 和 2 互换，原数组不变。

j 后移，直到指向 1..

二、

j（指向 1） $\leq 4$ ，于是  $i++$

i 指向了 8，所以 8 与 1 交换。

数组变成了：

**i      j**

2 1 7 8 3 5 6 4

三、j 后移，指向了 3,  $3 \leq 4$ ，于是  $i++$

i 这是指向了 7，于是 7 与 3 交换。

数组变成了：

i j  
2 1 3 8 7 5 6 4

四、j 继续后移，发现没有再比 4 小的数，所以，执行到了最后一步，

即上述 PARTITION(A, p, r) 代码部分的 第 7 行。

因此，i 后移一个单位，指向了 8

i j  
2 1 3 8 7 5 6 4

$A[i + 1] \leftrightarrow A[r]$ ，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

ok，快速排序第一趟完成。

4 把整个数组分成了俩部分，2 1 3, 7 5 6 8，再递归对这俩部分分别快速排序。

i p/j

2 1 3(主元)

2 与 2 互换，不变，然后又是 1 与 1 互换，还是不变，最后，3 与 3 互换，不变，  
最终，3 把 2 1 3，分成了俩部分，2 1，和 3。

再对 2 1，递归排序，最终结果成为了 1 2 3。

7 5 6 8(主元)，7、5、6、都比 8 小，所以第一趟，还是 7 5 6 8，

不过，此刻 8 把 7 5 6 8，分成了 7 5 6，和 8。[7 5 6 -> 5 7 6 -> 5 6 7]

再对 7 5 6，递归排序，最终结果变成 5 6 7 8。

ok，所有过程，全部分析完成。

最后，看下我画的图：



## 快速排序

## QUICKSORT (A.P.T)

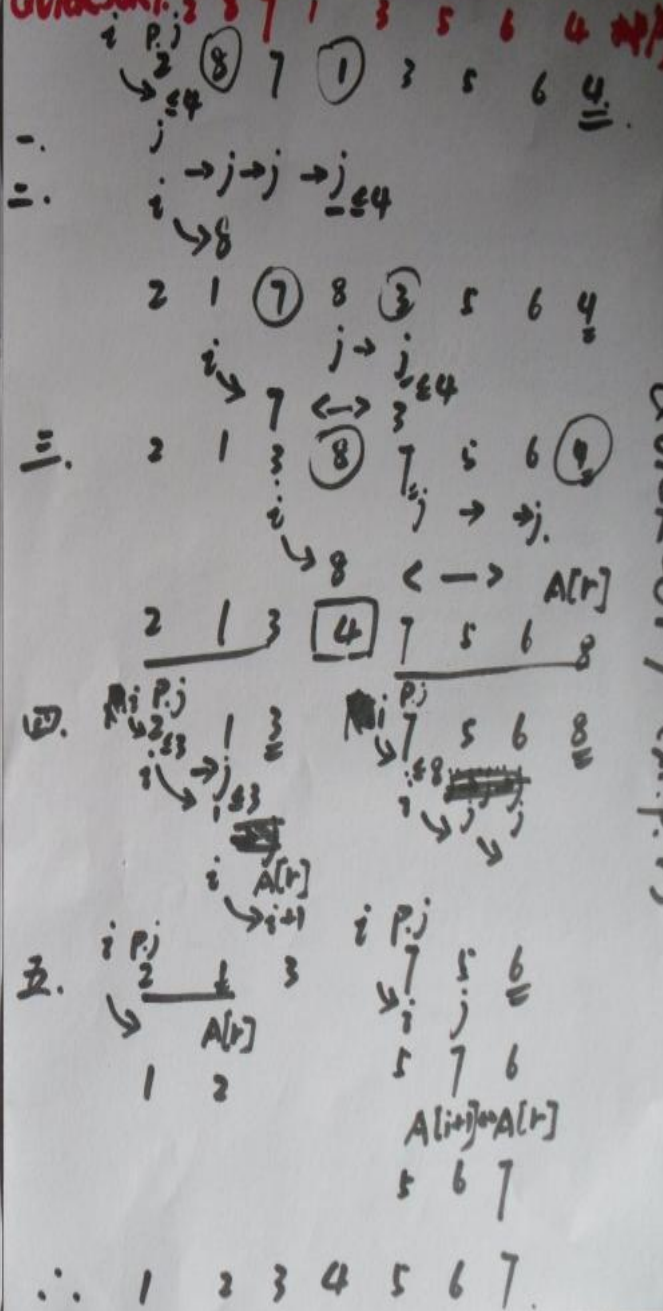
1. if  $p < r$
2. then  $q \leftarrow \text{PARTITION}(A, p, r)$
3.      $\text{QUICKSORT}(A, p, q-1)$
4.      $\text{QUICKSORT}(A, q+1, r)$

PARTITION(A, p, r)

1.  $x \leftarrow A[r]$
2.  $i \leftarrow p-1$
3. for  $j \leftarrow p$  to  $r-1$
4.     do if  $A[j] \leq x$
5.         then  $i \leftarrow i+1$
6.         exchange  $A[i] \leftrightarrow A[j]$
7. exchange  $A[i+1] \leftrightarrow A[r]$
8. return  $i+1$

July, 二〇一一年一月四日

QuickSort: 2 3 7 1 3 5 6 4 10



## 快速排序算法版本二

不过，这个版本不再选取（如上第一版本的）数组的最后一个元素为主元，而是选择，数组中的第一个元素为主元。

\*\*\*\*\*

```
/* 函数功能：快速排序算法
```

\*/

```

/* 函数参数：结构类型 table 的指针变量 tab      */
/*      整型变量 left 和 right 左右边界的下标  */
/* 函数返回值：空                                */
/* 文件名：quicksort.c 函数名：quicksort ()      */
/*****/
void quicksort(table *tab,int left,int right)
{
    int i,j;
    if(left<right)
    {
        i=left;j=right;
        tab->r[0]=tab->r[i]; //准备以本次最左边的元素值为标准进行划分，先保存其值
        do
        {
            while(tab->r[j].key>tab->r[0].key&& i<j)
                j--;      //从右向左找第 1 个小于标准值的位置 j
            if(i<j)        //找到了，位置为 j
            {
                tab->r[i].key=tab->r[j].key;i++;
            } //将第 j 个元素置于左端并重置 i
            while(tab->r[i].key<tab->r[0].key&& i<j)
                i++;      //从左向右找第 1 个大于标准值的位置 i
            if(i<j)        //找到了，位置为 i
            {
                tab->r[j].key=tab->r[i].key;j--;
            } //将第 i 个元素置于右端并重置 j
        }while(i!=j);

        tab->r[i]=tab->r[0]; //将标准值放入它的最终位置,本次划分结束
        quicksort(tab,left,i-1); //对标准值左半部递归调用本函数
        quicksort(tab,i+1,right); //对标准值右半部递归调用本函数
    }
}

```

-----

ok, 咱们, 还是以上述相同的数组, 应用此快排算法的版本二, 来演示此排序过程:  
这次, 以数组中的第一个元素 2 为主元。

**2(主) 8 7 1 3 5 6 4**

请细看:

**2 8 7 1 3 5 6 4**

i->                      <-j  
(找大)                      (找小)

一、j

j 找第一个小于 2 的元素 1,1 赋给(覆盖重置)i 所指元素 2

得到:

**1 8 7    3 5 6 4**  
i        j

二、i

i 找到第一个大于 2 的元素 8,8 赋给(覆盖重置)j 所指元素(NULL<-8)

**1    7 8 3 5 6 4**  
i   <-j

三、j

j 继续左移, 在与 i 碰头之前, 没有找到比 2 小的元素, 结束。

最后, 主元 2 补上。

第一趟快排结束之后, 数组变成:

**1 2 7 8 3 5 6 4**

第二趟,

**7 8 3 5 6 4**

i->                      <-j  
(找大)                      (找小)

一、j

j 找到 4, 比主元 7 小, 4 赋给 7 所处位置

得到:

**4 8 3 5 6**  
i->                      j

二、i

i 找比 7 大的第一个元素 8,8 覆盖 j 所指元素(NULL)

4 3 5 6 8

i j

4 6 3 5 8

i-> j

i 与 j 碰头，结束。

第三趟：

4 6 3 5 7 8

.....

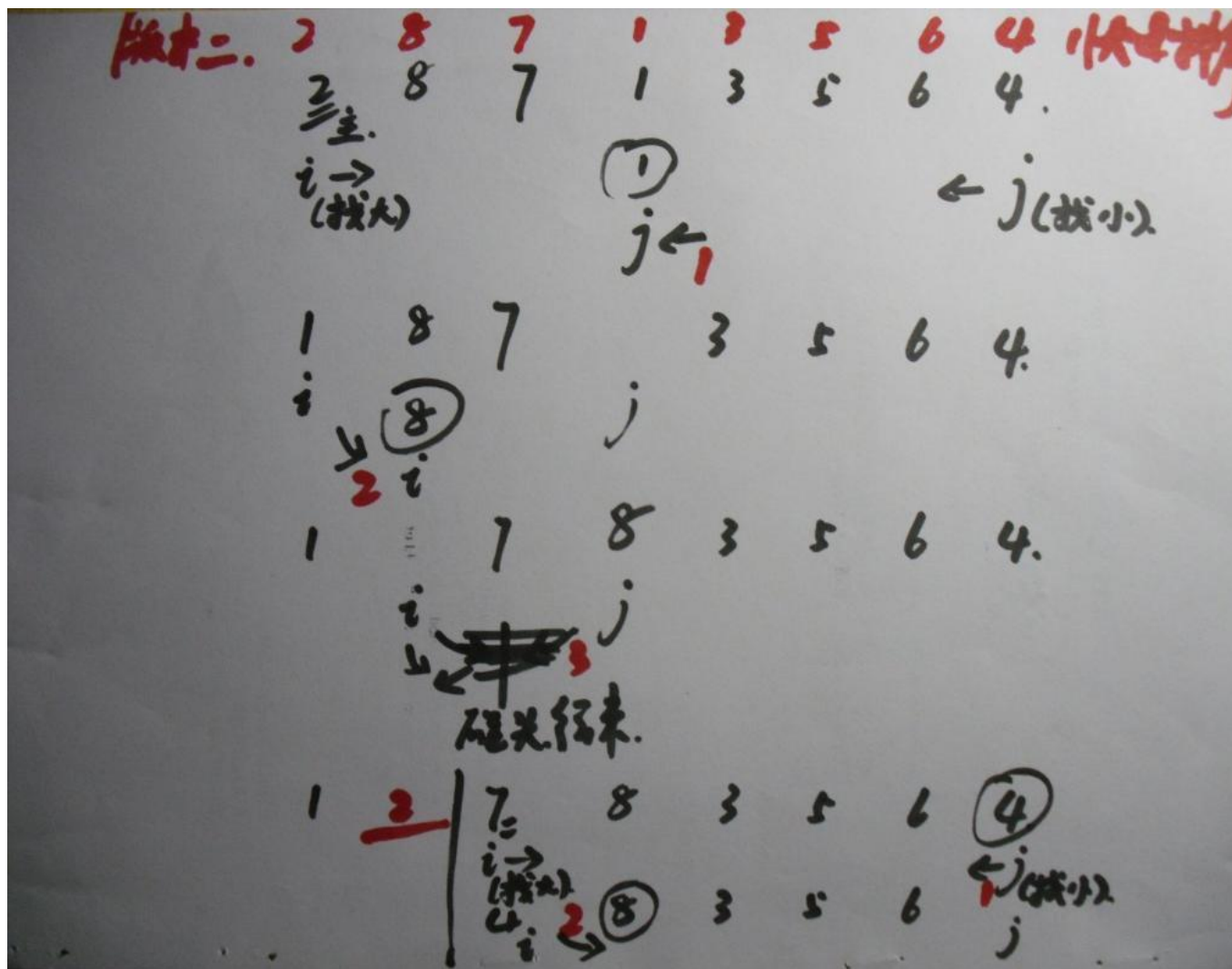
以下，分析原理，一致，略过。

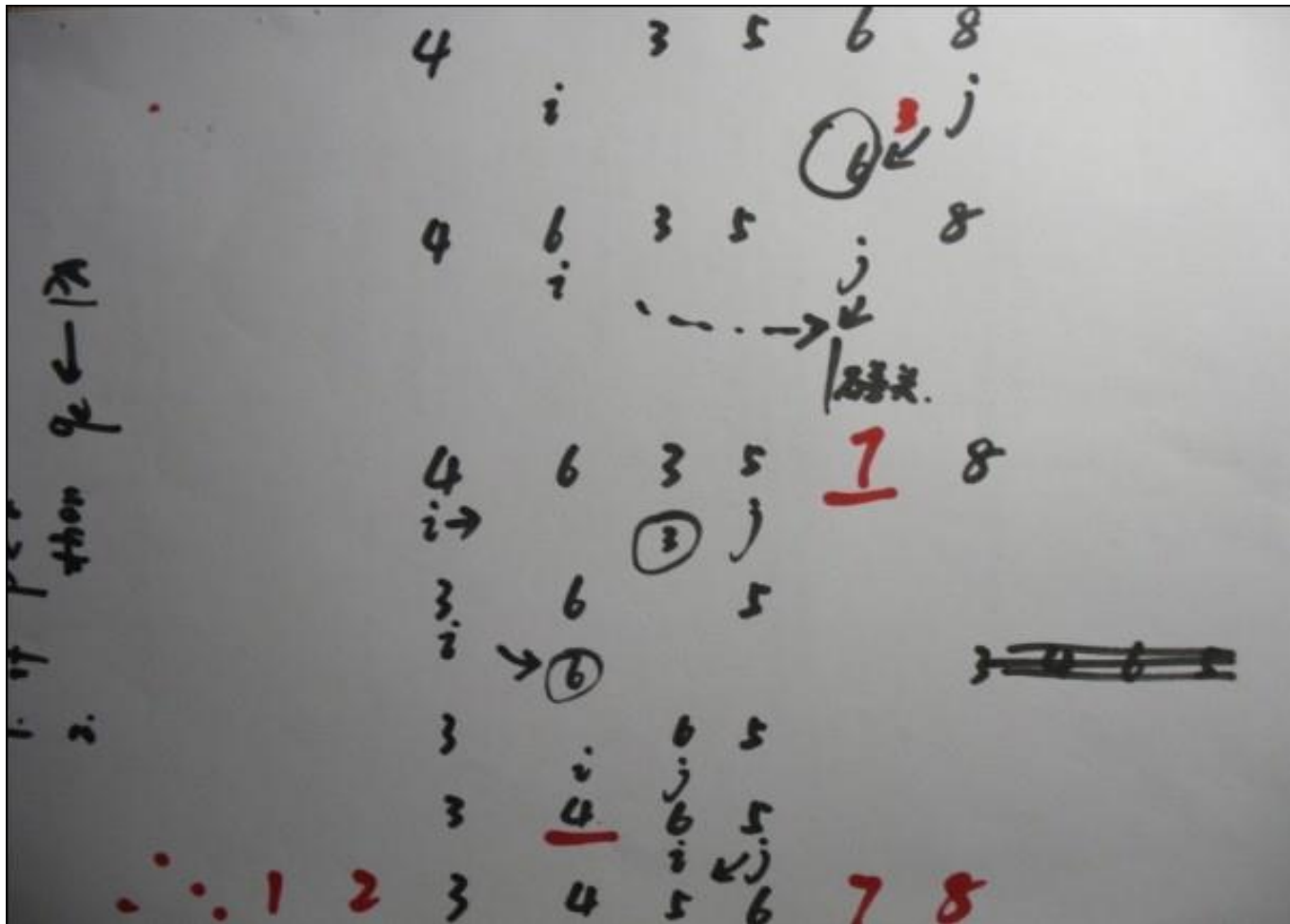
最后的结果，如下图所示：

1 2 3 4 5 6 7 8

相信，经过以上内容的具体分析，你一定明白了。

最后，贴一下我画的关于这个排序过程的图：





完。一月五日补充。

OK,上述两种算法，明白一种即可。

五、快速排序的最坏情况和最快情况。

**最坏情况**发生在划分过程产生的两个区域分别包含  $n-1$  个元素和一个 0 元素的时候，即假设算法每一次递归调用过程中都出现了，这种划分不对称。那么划分的代价为  $O(n)$ ，因为对一个大小为 0 的数组递归调用后，返回  $T(0) = O(1)$ 。

估算法的运行时间可以递归的表示为：

$$T(n) = T(n-1) + T(0) + O(n) = T(n-1) + O(n).$$

可以证明为  $T(n) = O(n^2)$ 。

因此，如果在算法的每一层递归上，划分都是最大程度不对称的，那么算法的运行时间就是  $O(n^2)$ 。

亦即，快速排序算法的最坏情况并不比插入排序的更好。

此外，当数组完全排好序之后，快速排序的运行时间为  $O(n^2)$ 。

而在同样情况下，插入排序的运行时间为  $O(n)$ 。

//注，请注意理解这句话。我们说一个排序的时间复杂度，是仅仅针对一个元素的。

//意思是，把一个元素进行插入排序，即把它插入到有序的序列里，花的时间为  $n$ 。

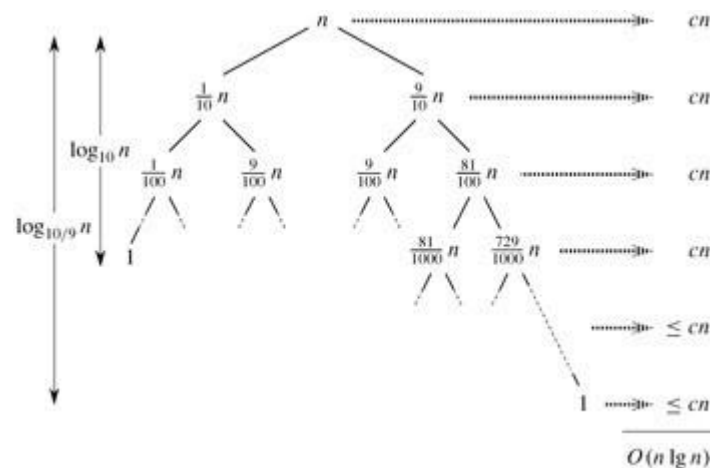
再来证明，最快情况下，即 PARTITION 可能做的最平衡的划分中，得到的每个子问题都不能大于  $n/2$ 。

因为其中一个子问题的大小为  $\lfloor n/2 \rfloor$ 。另一个子问题的大小为  $\lceil n/2 \rceil - 1$ 。

在这种情况下，快速排序的速度要快得多。为，

$$T(n) \leq 2T(n/2) + O(n) \text{ .可以证得, } T(n) = O(n \lg n) \text{ 。$$

直观上，看，快速排序就是一颗递归数，其中，PARTITION 总是产生 9:1 的划分，总的运行时间为  $O(n \lg n)$ 。各结点中示出了子问题的规模。每一层的代价在右边显示。每一层包含一个常数  $c$ 。



完。

July、二零一一年一月四日。

=====

请各位自行，思考以下这个版本，对应于上文哪个版本？

**HOARE-PARTITION(A, p, r)**

```

1  x ← A[p]
2  i ← p - 1
3  j ← r + 1
4  while TRUE
5      do repeat j ← j - 1
6          until A[j] ≤ x
7          repeat i ← i + 1
8              until A[i] ≥ x
9          if i < j
10             then exchange A[i] ↔ A[j]
11             else return j

```

我常常思考，为什么有的人当时明明读懂明白了一个算法，而一段时间过后，它又对此算法完全陌生而不了解了列？

我想，究其根本，还是没有彻底明白此快速排序算法的原理，与来龙去脉...  
那作何改进列，只能找发明那个算法的原作者了，从原作者身上，再多挖掘点有用的东西出来。

July、二零一一年二月十五日更新。

=====

最后，再给出一个快速排序算法的简洁示例：

```

Quicksort 函数
void quicksort(int l, int u)
{  int i, m;
   if (l >= u) return;
   swap(l, randint(l, u));
   m = l;
   for (i = l+1; i <= u; i++)
       if (x[i] < x[l])
           swap(++m, i);
   swap(l, m);
   quicksort(l, m-1);
}

```



```
    quicksort(m+1, u);  
}
```

如果函数的调用形式是 `quicksort(0, n-1)`，那么这段代码将对一个全局数组 `x[n]` 进行排序。函数的两个参数分别是将要进行排序的子数组的下标：`l` 是较低的下标，而 `u` 是较高的下标。

函数调用 `swap(i,j)` 将会交换 `x[i]` 与 `x[j]` 这两个元素。  
第一次交换操作将会按照均匀分布的方式在 `l` 和 `u` 之间随机地选择一个划分元素。

ok，更多请参考我写的关于快速排序算法的第二篇文章：[一之续、快速排序算法的深入分析](#)，第三篇文章：[十二、一之再续：快速排序算法之所有版本的 c/c++ 实现](#)。

[July、二零一一年二月二十日更新。](#)

## 十二（续）、快速排序算法的深入分析

作者:July 二零一一年二月二十七日

-----

前言

- 一、快速排序最初的版本
- 二、快速排序名字的由来
- 三、Hoare 版本的具体分析
- 四、快速排序的优化版本
- 五、快速排序的深入分析
- 六、Hoare 版本与优化后版本的比较
- 七、快速排序算法的时间复杂度
- 八、由快速排序所想到的

前言

之前，曾在本 BLOG 内写过一篇文章，十二、[快速排序算法](#)，不少网友反映此文好懂。然，后来有网友 [algorithm\\_\\_](#)，指出，“快速排序算法怎么一步一步想到的列？就如一个 P 与 NP 问题。知道了解，证明不难。可不知道解之前，要一点一点、一步一步推导出来，好难阿？”

其实，这个问题，我也想过很多次了。之前，也曾的博客里多次提到过。那么，到底为什么，有不少人看了我写的快速排序算法，过了一段时间后，又不清楚快排是怎么一回事了列？

以下是我在十、[从头到尾彻底理解傅里叶变换算法、下](#)，一文里回复 [algorithm\\_\\_](#) 的评论：

“很大一部分原因，就是只知其表，不知其里，只知其用，不知其本质。很多东西，都可以从本质看本质的。而大部分人没有做到这一点。从而看了又忘，忘了再看，如此，在对知识的一次一次重复记忆中，始终未能透析本质，从而，形成不好的循环。

所以，归根究底，学一个东西，不但要运用自如，还要通晓其原理，来龙去脉与本质。正如侯捷先生所言，只知一个东西的用法，却不知其原理，实在不算高明。你提出的问题，非常好。我会再写一篇文章，彻底阐述快速排序算法是如何设计的，以及怎么一步一步来的。”

ok，那么现在，我就来彻底分析下此快速排序算法，希望能让读者真正理解此算法，通晓其来龙去脉，明白其内部原理。本文着重分析快速排序算法的过程来源及其时间复杂度，要了解什么是快速排序算法，请参考此文：[精通八大排序算法系列：一、快速排序算法](#)。

## 一、快速排序最初的版本

快速排序的算法思想(此时，还不叫做快速排序)最初是由，一个名叫 [R.Sedegwick](#) 提出的，他的算法思想为：

I、取俩个指针 i, j，开始时， $i=2$ ， $j=n$ ，且我们确定，最终完成排序时，左边子序列的数  $\leq$  右边子序列。

II、矫正位置，不断交换

[i-->\(i 从左至右，右移，找比第一个元素要大的\)](#)

通过比较  $k_i$  与  $k_1$ ，如果  $R_i$  在分划之后最终要成为左边子序列的一部分，则  $i++$ ，且不断  $i++$ ，直到遇到一个该属于右边子序列  $R_i$ (较大)为止。

[<--j\(j 从右至左，左移，找比第一个元素要小的\)](#)

类似的， $j--$ ，直到遇到一个该属于左边子序列的较小元素  $R_j$ (较小)为止，

如此，当  $i < j$  时，交换  $R_i$  与  $R_j$ ，即摆正位置嘛，把较小的放左边，较大的放右边。

III、然后以同样的方式处理划分的记录，直到 i 与 j 碰头为止。这样，不断的通过交换 Ri, Rj 摆正位置，最终完成整个序列的排序。

举个例子，如下(2 为主元):

```
      i->                <-j(找小)
2  8  7  1  3  5  6  4
```

j 所指元素 4，大于 2，所以，j--，

```
      i                <--j
2  8  7  1  3  5  6  4
```

此过程中，若 j 没有遇到比 2 小的元素，则 j 不断--，直到 j 指向了 1，

```
      i      j
2  8  7  1  3  5  6  4
```

此刻，8 与 1 互换，

```
      i      j
2  1  7  8  3  5  6  4
```

此刻，i 所指元素 1，小于 2，所以 i 不动，j 继续--，始终没有遇到再比 2 小的元素，最终停至 7。

```
      i  j
2  1  7  8  3  5  6  4
```

最后，i 所指元素 1，与数列中第一个元素 k1，即 2 交换，

```
      i
[1] 2 [7 8 3 5 6 4]
```

这样，2 就把整个序列，排成了 2 个部分，接下来，再对剩余待排序的数递归进行第二趟、第三趟排序....。

由以上的过程，还是可以大致看出此算法的拙陋之处的。如一，在上述第一步过程中，j 没有找到比 2 小的元素时，需不断的前移，j--。二，当 i 所指元素交换后，无法确保此刻 i 所指的元素就一定小于 2，即 i 指针，还有可能继续停留在原处，不能移动。如此停留，势必应该会耗费不少时间。

## 二、快速排序名字的由来

后来，**C.A.R.Hoare** 根据上述方案，因其排序速率较快，便称之为"**快速排序**"，快速排序也因此而诞生了。并最终一举成名，成为了**二十世纪最伟大的 10 大算法**之一。

他给出的算法思想描述的具体版本，如下：

```
HOARE-PARTITION(A, p, r)
1  x ← A[p]
2  i ← p - 1
3  j ← r + 1
4  while TRUE
5      do repeat j ← j - 1
6          until A[j] ≤ x
7      repeat i ← i + 1
8          until A[i] ≥ x
9      if i < j
10         then exchange A[i] ↔ A[j]
11         else return j
```

此程序的原理，与上文中 **R.Sedegwick** 的思想，已明显不同。确切的说，是有了不小的完善。后来，此版本又有不少的类似变种。下面，会具体分析。

### 三、Hoare 版本的具体分析

在上面，我们已经知道，**Hoare** 的快速排序版本可以通过前后俩个指针，分别指向首尾，分别比较而进行排序。

下面，分析一下此版本，或其它变种问题：

I、 俩个指针，i 指向序列的首部，j 指着尾部，即  $i=1$ ， $j=n$ ，取数组中第一个元素  $k_i$  为主元，即  $key \leftarrow k_i$  (赋值)。

II、赋值操作（注，以下“ $\leftarrow$ ”，表示的是赋值）：

j(找小)，从右至左，不断--，直到遇到第一个比  $key$  小的元素  $k_j$ ， $k_i \leftarrow k_j$ 。

i(找大)，从左至右，不断++，直到遇到第一个比  $key$  大的元素  $k_i$ ， $k_j \leftarrow k_i$ 。

III、按上述方式不断进行，直到 i, j 碰头， $k_i=key$ ，第一趟排序完成接下来重复 II 步骤，递归进行。

再举一个例子：对序列 3 8 7 1 2 5 6 4，进行排序：

1、j--，直到遇到了序列中第一个比 key 值 3 小的元素 2，把 2 赋给 ki，j 此刻指向了空元素。

```
      i                j
    3  8  7  1  2  5  6  4
      i                j
=> 2  8  7  1    5  6  4
```

2、i++，指向 8，把 8 重置给 j 所指元素空白处，i 所指元素又为空：

```
      i                j
    2  8  7  1    5  6  4
      i                j
=> 2    7  1  8  5  6  4
```

3、j 继续--，遇到了 1，还是比 3(事先保存的 key 值)小，1 赋给 i 所指空白处：

```
      i    j
    2    7  1  8  5  6  4
=> 2  1  7    8  5  6  4
```

4、同理，i 又继续++，遇到了 7，比 key 大，7 赋给 j 所指空白处，此后，i，j 碰头。  
第一趟结束：

```
      i    j
    2  1  7    8  5  6  4
      i    j
=> 2  1    7  8  5  6  4
```

5、最后，事先保存的 key，即 3 赋给 ki，即 i 所指空白处，得：

[2 1] 3 [7 8 5 6 4]

所以，整趟下来，便是这样：

```
3  8  7  1  2  5  6  4
2  8  7  1  3  5  6  4
2  3  7  1  8  5  6  4
2  1  7  3  8  5  6  4
2  1  3  7  8  5  6  4
2  1  3  7  8  5  6  4
```

后续补充:

如果待排序的序列是逆序数列?ok, 为了说明的在清楚点, 再举个例子, 对序列 9 8 7 6 5 4 3 2 1 排序:

9 8 7 6 5 4 3 2 1 //9 为主元

1 8 7 6 5 4 3 2 //从右向左找小, 找到 1, 赋给第一个

1 8 7 6 5 4 3 2 //从左向右找大, 没有找到, 直到与 j 碰头

1 8 7 6 5 4 3 2 9 //最后, 填上 9.

如上, 当数组已经是逆序排列好的话, 我们很容易就能知道, 此时数组排序需要  $O(N^2)$  的时间复杂度。稍后下文, 会具体分析快速排序的时间复杂度。

最后, 写程序实现此算法, 如下, 相信, 不用我过多解释了:

```
int partition(int data[],int lo,int hi) //引自 whatever。
{
    int key=data[lo];
    int l=lo;
    int h=hi;
    while(l<h)
    {
        while(key<=data[h] && l<h) h--; //高位找小, 找到了, 就把它弄到前面去
        data[l]=data[h];
        while(data[l]<=key && l<h) l++; //低位找大, 找到了, 就把它弄到后面去
        data[h]=data[l];
    }
    data[l]=key;
    return l;
}
```

后续补充: 举个例子, 如下 (只说明了第一趟排序):

3 8 7 1 2 5 6 4

2 8 7 1 5 6 4

2 7 1 8 5 6 4

2 1 7 8 5 6 4

2 1 7 8 5 6 4

2 1 3 7 8 5 6 4 //最后补上，关键字 3

看到这，不知各位读者，有没有想到我上一篇文章里头，一、[快速排序算法](#)，的那快速排序的第二个版本？对，上述程序，即那篇文章里头的第二个版本。我把程序揪出来，你一看，就明白了：

```
void quicksort(table *tab,int left,int right)
{
    int i,j;
    if(left<right)
    {
        i=left;j=right;
        tab->r[0]=tab->r[i]; //准备以本次最左边的元素值为标准进行划分，先保存其值
        do
        {
            while(tab->r[j].key>tab->r[0].key&& i<j)
                j--; //从右向左找第 1 个小于标准值的位置 j
            if(i<j) //找到了，位置为 j
            {
                tab->r[i].key=tab->r[j].key;i++;
            } //将第 j 个元素置于左端并重置 i
            while(tab->r[i].key<tab->r[0].key&& i<j)
                i++; //从左向右找第 1 个大于标准值的位置 i
            if(i<j) //找到了，位置为 i
            {
                tab->r[j].key=tab->r[i].key;j--;
            } //将第 i 个元素置于右端并重置 j
        }while(i!=j);

        tab->r[i]=tab->r[0]; //将标准值放入它的最终位置,本次划分结束
        quicksort(tab,left,i-1); //对标准值左半部递归调用本函数
        quicksort(tab,i+1,right); //对标准值右半部递归调用本函数
    }
}
```

我想，至此，已经讲的足够明白了。如果，你还不懂的话，好吧，伸出你的手指，数数吧....ok，再问读者一个问题：像这种  $i$  从左至右找大，找到第一个比 **key** 大(数组中第一个元素置为 **key**)，便重置  $kj$ ， $j$  从右向左找小，找到第一个比 **key** 小的元素，则重置  $ki$ ，当然此过程中， $i, j$  都在不断的变化，通过 $++$ ，或 $--$ ，指向着不同的元素。

你是否联想到了，现实生活中，有什么样的情形，与此快速排序算法思想相似?ok，等你想到了，再告诉我，亦不迟。:D。

#### 四、快速排序的优化版本

再到后来，**N.Lomuto** 又提出了一种新的版本，此版本即为此文**快速排序算法**，中阐述的第一个版本，即优化了 **PARTITION** 程序，它现在写在了 **算法导论** 一书上，

快速排序算法的关键是 **PARTITION** 过程，它对  $A[p..r]$  进行就地重排：

```
PARTITION(A, p, r)
1   $x \leftarrow A[r]$       //以最后一个元素， $A[r]$ 为主元
2   $i \leftarrow p - 1$ 
3  for  $j \leftarrow p$  to  $r - 1$   //注， $j$  从  $p$  指向的是  $r-1$ ，不是  $r$ 。
4      do if  $A[j] \leq x$ 
5          then  $i \leftarrow i + 1$ 
6              exchange  $A[i] \leftrightarrow A[j]$ 
7  exchange  $A[i + 1] \leftrightarrow A[r]$ 
8  return  $i + 1$ 
```

然后，对整个数组进行递归排序：

```
QUICKSORT(A, p, r)
1  if  $p < r$ 
2      then  $q \leftarrow \text{PARTITION}(A, p, r)$  //关键
3          QUICKSORT(A, p,  $q - 1$ )
4          QUICKSORT(A,  $q + 1, r$ )
```

举最开头的那个例子：2 8 7 1 3 5 6 4，不过与上不同的是：它不再以第一个元素为主元，而是以最后一个元素 4 为主元，且  $i, j$  俩指针都从头出发， $j$  一前， $i$  一后。 $i$  指元素的前一个位置， $j$  指着待排序数列中的第一个元素。



一、

i p/j-->

2 8 7 1 3 5 6 4(主元)

j 指的  $2 \leq 4$ , 于是  $i++$ , i 也指到 2, 2 和 2 互换, 原数组不变。

j 后移, 直到指向 1..

二、

j (指向 1)  $\leq 4$ , 于是  $i++$ , i 指向了 8,

i j  
2 8 7 1 3 5 6 4

所以 8 与 1 交换, 数组变成了:

i j  
2 1 7 8 3 5 6 4

三、j 后移, 指向了 3,  $3 \leq 4$ , 于是  $i++$

i 这时指向了 7,

i j  
2 1 7 8 3 5 6 4

于是 7 与 3 交换, 数组变成了:

i j  
2 1 3 8 7 5 6 4

四、j 继续后移, 发现没有再比 4 小的数, 所以, 执行到了最后一步,  
即上述 PARTITION(A, p, r)代码部分的 第 7 行。

因此, i 后移一个单位, 指向了 8

i j  
2 1 3 8 7 5 6 4

$A[i + 1] \leftrightarrow A[r]$ , 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

ok, 快速排序第一趟完成。接下来的过程, 略, 详细, 可参考此文: [快速排序算法](#)。不过, 有个问题, 你能发现此版本与上述版本的优化之处么?

## 五、快速排序的深入分析

咱们，再具体分析下上述的优化版本，

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
```

咱们以下数组进行排序，每一步的变化过程是：

i p/j

2 8 7 1 3 5 6 4(主元)

i j  
2 1 7 8 3 5 6 4

i j  
2 1 3 8 7 5 6 4

i  
2 1 3 4 7 5 6 8

由上述过程，可看出，j 扫描了整个数组一遍，只要一旦遇到比 4 小的元素，i 就++，然后，kj、ki 交换。那么，为什么当 j 找到比 4 小的元素后，i 要++列？你想麻，如果 i 始终停在原地不动，与 kj 每次交换的 ki 不就是同一个元素了么？如此，还谈什么排序？。

所以，j 在前面开路，i 跟在 j 后，j 只要遇到比 4 小的元素，i 就向前前进一步，然后把 j 找到的比 4 小的元素，赋给 i，然后，j 才再前进。

打个比喻就是，你可以这么认为，i 所经过的每一步，都必须是比 4 小的元素，否则，i 就不能继续前行。好比 j 是先行者，为 i 开路搭桥，把小的元素作为跳板放到 i 跟前，为其铺路前行啊。

于此，j 扫描到最后，也已经完全排查出了比 4 小的元素，只有最后一个主元 4，则交给 i 处理，因为最后一步， $\text{exchange } A[i + 1] \leftrightarrow A[r]$ 。这样，不但完全确保了只要是比 4 小的元素，都被交换到了数组的前面，且 j 之前未处理的比较大的元素则被交换到了后面，而且还是  $O(N)$  的时间复杂度，你不得不佩服此算法设计的巧妙。

这样，我就有一个问题了，上述的  $\text{PARTITION}(A, p, r)$  版本，可不可以改成这样咧？  
望读者思考：

```
PARTITION(A, p, r) //请读者思考版本。
1  x ← A[r]
2  i ← p - 1
3  for j ← p to r    //让 j 从 p 指向了最后一个元素 r
4      do if A[j] ≤ x
5          then i ← i + 1
6              exchange A[i] ↔ A[j]
//7  exchange A[i + 1] ↔ A[r]  去掉此最后的步骤
8  return i    //返回 i，不再返回 i+1.
```

## 六、Hoare 版本与优化后版本的比较

现在，咱们来讨论一个问题，快速排序中，其中对于序列的划分，我们可以看到，已经有以上俩个版本，那么这两个版本孰优孰劣？ok，不急，咱们来比较下：

为了看着方便，再贴一下各自的算法，

**Hoare 版本：**

```
HOARE-PARTITION(A, p, r)
1  x ← A[p]    //以第一个元素为主元
2  i ← p - 1
3  j ← r + 1
4  while TRUE
5      do repeat j ← j - 1
6          until A[j] ≤ x
7          repeat i ← i + 1
8              until A[i] ≥ x
9      if i < j
```

```

10      then exchange A[i] ↔ A[j]
11      else return j

```

优化后的算法导论上的版本：

PARTITION(A, p, r)

```

1  x ← A[r]      //以最后一个元素，A[r]为主元
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

```

咱们，先举上述说明 Hoare 版本的这个例子，对序列 3 8 7 1 2 5 6 4，进行排序：

**Hoare 版本**（以 3 为主元，**红体**为主元）：

```

3 8 7 1 2 5 6 4
2 8 7 1 5 6 4 //交换 1 次，比较 4 次
2 7 1 8 5 6 4 //交换 1 次，比较 1 次
2 1 7 8 5 6 4 //交换 1 次，比较 1 次
2 1 7 8 5 6 4 //交换 1 次，比较 0 次
2 1 3 7 8 5 6 4 //总计交换 4 次，比较 6 次。
//移动了元素 3、8、7、1、2.移动范围为：2+3+1+2+4=12.

```

**优化版本**（以 **4** 为主元）：

```

3 8 7 1 2 5 6 4 //3 与 3 交换，不用移动元素，比较 1 次
3 1 7 8 2 5 6 4 //交换 1 次，比较 3 次
3 1 2 8 7 5 6 4 //交换 1 次，比较 1 次
3 1 2 4 7 5 6 8 //交换 1 次，比较 2 次。
//即完成，总计交换 4 次，比较 7 次。
//移动了元素 8、7、1、2、4.移动范围为：6+2+2+2+4=16.

```

再举一个例子：对序列 2 8 7 1 3 5 6 4 排序：

Hoare 版本:

2 8 7 1 3 5 6 4  
1 8 7 3 5 6 4 //交换 1 次, 比较 5 次  
1 7 8 3 5 6 4 //交换 1 次, 比较 1 次  
1 2 7 8 3 5 6 4 //交换 0 次, 比较 1 次。2 填上, 完成, 总计交换 2 次, 比较 7 次。

优化版本:

2 8 7 1 3 5 6 4 //2 与 2 交换, 比较 1 次  
2 1 7 8 3 5 6 4 //交换 1 次, 比较 3 次  
2 1 3 8 7 5 6 4 //交换 1 次, 比较 1 次  
2 1 3 4 7 5 6 8 //交换 1 次, 比较 2 次。完成, 总计交换 4 次, 比较 7 次。

各位, 已经看出来了, 这俩个例子说明不了任何问题。到底哪个版本效率更高, 还有待进一步验证或者数学证明。ok, 等我日后发现更有利的证据再来论证下。

## 七、快速排序算法的时间复杂度

ok, 我想你已经完全理解了此快速排序, 那么, 我想你应该也能很快的判断出: 快速排序算法的平均时间复杂度, 即为  $O(n \lg n)$ 。为什么列? 因为你看,  $j, i$  扫描一遍数组, 花费用时多少? 对了, 扫描一遍, 当然是  $O(n)$  了, 那样, 扫描多少遍列,  $\lg n$  到  $n$  遍, 最快  $\lg n$ , 最慢  $n$  遍。且可证得, 快速排序的平均时间复杂度即为  $O(n \lg n)$ 。

PARTITION 可能做的最平衡的划分中, 得到的每个子问题都不能大于  $n/2$ 。因为其中一个子问题的大小为  $\lfloor n/2 \rfloor$ 。另一个子问题的大小为  $\lfloor n/2 \rfloor - 1$ 。在这种情况下, 快速排序的速度要快得多。为:

$T(n) \leq 2T(n/2) + O(n)$ 。可以证得,  $T(n) = O(n \lg n)$ 。

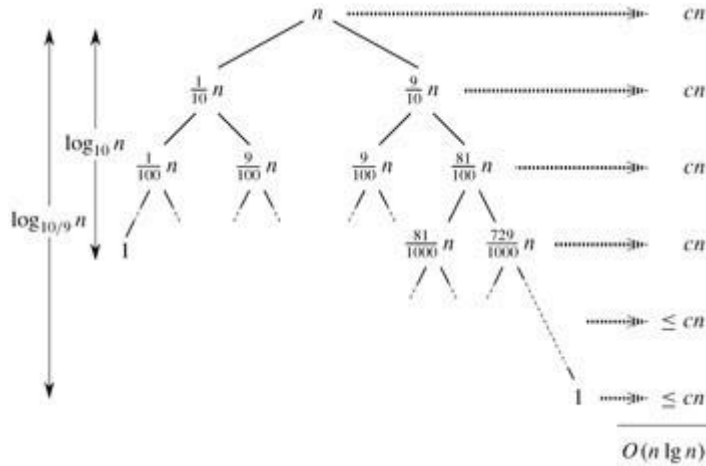
以下给出一个递归树的简单证明:

在分治算法中的三个步骤中, 我们假设分解和合并过程所用的时间分别为  $D(n), C(n)$ , 设  $T(n)$  为处理一个规模为  $n$  的序列所消耗的时间为子序列个数, 每一个子序列是原序列的  $1/b$ ,  $\alpha$  为把每个问题分解成  $\alpha$  个子问题, 则所消耗的时间为:

$$\begin{aligned} O(1) & \quad \text{如果 } n \leq c \\ T(n) &= \alpha T(n/b) + D(n) + C(n) \end{aligned}$$

在快速排序中, $\alpha$  是为2的,  $b$  也为2, 则分解(就是取参照点,可以认为是1), 合并(把数组合并,为  $n$ ), 因此  $D(n) + C(n)$  是一个线性时间  $O(n)$ .这样时间就变成了:

$$T(n) = 2T(n/2) + O(n).$$



如上图所示, 在每个层的时间复杂度为:  $O(n)$ , 一共有  $\lg n$  层, 每一层上都是  $cn$ , 所以共消耗时间为  $cn \cdot \lg n$ ; 则总时间:

$$cn \cdot \lg 2n + cn = cn(1 + \lg n) \quad \text{即 } O(n \lg n).$$

关于  $T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \lg n)$  的严格数学证明, 可参考[算法导论 第四章 递归式](#)。

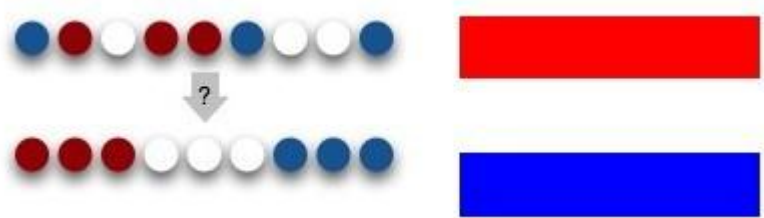
而后, 我想问读者一个问题, 由此快速排序算法的时间复杂度, 你想到了什么, 是否想到了归并排序的时间复杂度, 是否想到了二分查找, 是否想到了一棵  $n$  个结点的红黑树的高度  $\lg n$ , 是否想到了.....

## 八、由快速排序所想到的

上述提到的东西, 很早以前就想过了。此刻, 我倒是想到了前几天看到的一个荷兰国旗问题。当时, 和 `algorithm_`、`gnu_hpc` 简单讨论过这个问题。现在, 我也来具体解决下此问题:

问题描述:

我们将乱序的红白蓝三色小球排列成有序的红白蓝三色的同颜色在一起的小球组。这个问题之所以叫**荷兰国旗**, 是因为我们可以将红白蓝三色小球想象成条状物, 有序排列后正好组成荷兰国旗。如下图所示:



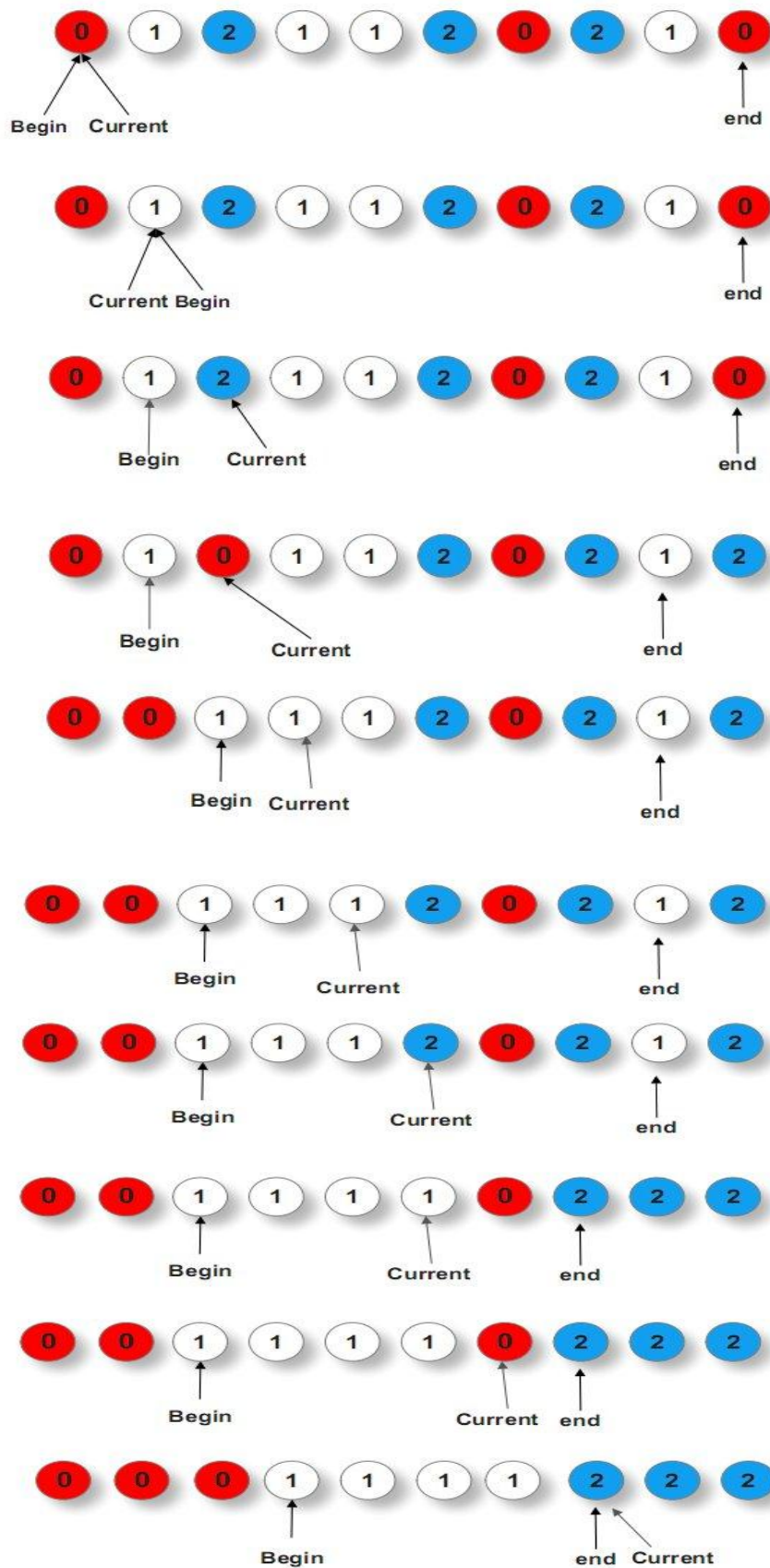
这个问题，类似快排中 `partition` 过程。不过，要用三个指针，一前 `begin`，一中 `current`，一后 `end`，俩俩交换。

- 1、`current` 遍历，整个数组序列，`current` 指 1 不动，
- 2、`current` 指 0，与 `begin` 交换，而后 `current++`，`begin++`，
- 3、`current` 指 2，与 `end` 交换，而后，`current` 不动，`end--`。

为什么，第三步，`current` 指 2，与 `end` 交换之后，`current` 不动了列，对的，正如 `algorithm__` 所说：`current` 之所以与 `begin` 交换后，`current++`、`begin++`，是因为此无后顾之忧。而 `current` 与 `end` 交换后，`current` 不动，`end--`，是因有后顾之忧。

为什么啊，因为你想想啊，你最终的目的无非就是为了让 0、1、2 有序排列，试想，如果第三步，`current` 与 `end` 交换之前，万一 `end` 之前指的是 0，而 `current` 交换之后，`current` 此刻指的是 0 了，此时，`current` 能动么？不能动啊，指的是 0，还得与 `begin` 交换列。

ok，说这么多，你可能不甚明了，直接引用下 `gnu`hpc 的图，就一目了然了：



本程序主体的代码是：



```

//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--;
    }
}

```

看似，此问题与本文关系不大，但是，一来因其余本文中快速排序 **partition** 的过程类似，二来因为此问题引发了一段小小的思考，并最终成就了本文。差点忘了，还没回答本文开头提出的问题。**所以，快速排序算法是如何想到的，如何一步一步发明的列？答案很简单：多观察，多思考。**

ok，测试一下，看看你平时有没有多观察、多思考的习惯：我不知道是否有人真正思考过冒泡排序，如果思考过了，你是否想过，怎样改进冒泡排序列？ok，其它的，我就不多说了，只贴以下这张图：

768	908	908	908	908	908	908	908
765	793	765	897	897	907	897	897
677	765	703	765	765	765	765	765
612	677	677	703	703	703	703	703
509	612	612	677	677	677	677	677
154	509	509	612	612	653	653	653
426	154	426	509	509	612	612	612
653	426	653	426	653	509	512	512
275	653	275	653	426	512	509	509
897	275	897	275	512	426	503	503
170	897	170	512	275	503	426	426
908	170	512	170	503	275	275	275
061	512	154	503	170	170	170	170
512	061	503	154	154	154	154	154
087	503	087	087	087	087	087	087
503	087	061	061	061	061	061	061

图16 “鸡尾混合排序”

更多，请参考：[十二、快速排序算法之所有版本的 c/c++实现](#)。本文完。

## 十二（再续）：快速排序算法之所有版本的 c/c++实现

作者：July、二零一一年三月二十日。

出处：[http://blog.csdn.net/v\\_JULY\\_v](http://blog.csdn.net/v_JULY_v)。

## 前言：

相信，经过本人之前写的前俩篇关于快速排序算法的文章：第一篇、[一、快速排序算法](#)，及第二篇、[一之续、快速排序算法的深入分析](#)，各位，已经对快速排序算法有了足够的了解与认识。但仅仅停留在对一个算法的认识层次上，显然是不够的，即便你认识的有多透彻与深入。最好是，编程实现它。

而网上，快速排序的各种写法层次不清，缺乏统一、整体的阐述与实现，即，没有个一锤定音，如此，我便打算自己去实现它了。

于是，今花了一个上午，把快速排序算法的各种版本全部都写程序一一实现了一下。包括网上有的，没的，算法导论上的，国内教材上通用的，随机化的，三数取中分割法的，递归的，非递归的，所有版本都用 `c/c++` 全部写了个遍。

鉴于时间仓促下，一个人考虑问题总有不周之处，以及水平有限等等，不正之处，还望各位不吝赐教。不过，以下，所有全部 `c/c++` 源码，都经本人一一调试，若有任何问题，恳请指正。

ok，本文主要分为以下几部分内容：

### 第一部分、递归版

#### 一、算法导论上的单向扫描版本

#### 二、国内教材双向扫描版

##### 2.1、Hoare 版本

##### 2.2、Hoare 的几个变形版本

#### 三、随机化版本

#### 四、三数取中分割法

### 第二部分、非递归版

好的，请一一细看。

## 第一部分、快速排序的递归版本

### 一、算法导论上的版本

在我写的第二篇文章中，我们已经知道：

“再到后来，N.Lomuto 又提出了一种新的版本，此版本....，即优化了 PARTITION 程序，它现在写在了 算法导论 一书上”：

快速排序算法的关键是 PARTITION 过程，它对  $A[p..r]$  进行就地重排：

```

PARTITION(A, p, r)
1  x ← A[r]      //以最后一个元素，A[r]为主元
2  i ← p - 1
3  for j ← p to r - 1  //注，j 从 p 指向的是 r-1，不是 r。
4      do if A[j] ≤ x
5          then i ← i + 1
6              exchange A[i] <-> A[j]
7  exchange A[i + 1] <-> A[r]  //最后，交换主元
8  return i + 1

```

然后，对整个数组进行递归排序：

```

QUICKSORT(A, p, r)
1  if p < r
2      then q ← PARTITION(A, p, r)  //关键
3      QUICKSORT(A, p, q - 1)
4      QUICKSORT(A, q + 1, r)

```

根据上述伪代码，我们不难写出以下的 c/c++ 程序：  
首先是，PARTITION 过程：

```

int partition(int data[],int lo,int hi)
{
    int key=data[hi]; //以最后一个元素，data[hi]为主元
    int i=lo-1;
    for(int j=lo;j<hi;j++)  ///注，j 从 p 指向的是 r-1，不是 r。
    {
        if(data[j]<=key)
        {
            i=i+1;
            swap(&data[i],&data[j]);
        }
    }
    swap(&data[i+1],&data[hi]); //不能改为 swap(&data[i+1],&key)
    return i+1;
}

```

补充说明：举个例子，如下为第一趟排序（更多详尽的分析请参考第二篇文章）：

第一趟(4步)：

a: 3 8 7 1 2 5 6 4 //以最后一个元素，data[hi]为主元

b: 3 1 7 8 2 5 6 4

c: 3 1 2 8 7 5 6 4

d: 3 1 2 4 7 5 6 8 //最后，swap(&data[i+1],&data[hi])

而其中 swap 函数的编写，是足够简单的：

```
void swap(int *a,int *b)
{
    int temp=*a;
    *a=*b;
    *b=temp;
}
```

然后是，调用 partition，对整个数组进行递归排序：

```
void QuickSort(int data[], int lo, int hi)
{
    if (lo<hi)
    {
        int k = partition(data, lo, hi);
        QuickSort(data, lo, k-1);
        QuickSort(data, k+1, hi);
    }
}
```

现在，我有一个问题要问各位了，保持其它的不变，稍微修改一下上述的 partition 过程，如下：

```
int partition(int data[],int lo,int hi) //请读者思考
{
    int key=data[hi]; //以最后一个元素，data[hi]为主元
    int i=lo-1;
```

```

for(int j=lo;j<=hi;j++) //现在，我让 j 从 lo 指向了 hi，不是 hi-1。
{
    if(data[j]<=key)
    {
        i=i+1;
        swap(&data[i],&data[j]);
    }
}
//swap(&data[i+1],&data[hi]); //去掉这行
return i;           //返回 i，非 i+1.
}

```

如上，其它的不变，请问，让 j 扫描到了最后一个元素，后与 data[i+1]交换，去掉最后的 swap(&data[i+1],&data[hi])，然后，再返回 i。请问，如此，是否可行？

其实这个问题，就是我第二篇文章中，所提到的：

“上述的 PARTITION(A, p, r)版本，可不可以改成这样咧？以下这样列”：

PARTITION(A, p, r) //请读者思考版本。

```

1  x ← A[r]
2  i ← p - 1
3  for j ← p to r    //让 j 从 p 指向了最后一个元素 r
4      do if A[j] ≤ x
5          then i ← i + 1
6          exchange A[i] <-> A[j]
//7  exchange A[i + 1] <-> A[r]  去掉此最后的步骤
8  return i    //返回 i，不再返回 i+1.

```

望读者思考，后把结果在评论里告知我。

我这里简单论述下：上述请读者思考版本，只是代码做了以下三处修改而已：1、j 从 p->r；2、去掉最后的交换步骤；3、返回 i。首先，无论是我的版本，还是算法导论上的原装版本，都是准确无误的，且我都已经编写程序测试通过了。但，其实这俩种写法，思路是完全一致的。

为什么这么说咧？请具体看以下的请读者思考版本，

```

int partition(int data[],int lo,int hi) //请读者思考
{
    int key=data[hi]; //以最后一个元素，data[hi]为主元
    int i=lo-1;
    for(int j=lo;j<=hi;j++) //....
    {
        if(data[j]<=key) //如果让 j 从 lo 指向 hi，那么当 j 指到 hi 时，是一定会有 A[j]<=x 的
        {
            i=i+1;
            swap(&data[i],&data[j]);
        }
    }
    //swap(&data[i+1],&data[hi]); //事实是，应该加上这句，直接交换，即可。
    return i; //
}

```

我们知道当 j 最后指到了 r 之后，是一定会有  $A[j] \leq x$  的（即=），所以这个 if 判断就有点多余，没有意义。所以应该如算法导论上的版本那般，最后直接交换 `swap(&data[i+1],&data[hi]);` 即可，返回 i+1。所以，总体说来，算法导论上的版本那样写，比请读者思考版本更规范，更合乎情理。ok，请接着往下阅读。

当然，上述 partition 过程中，也可以去掉 swap 函数的调用，直接写在分割函数里：

```

int partition(int data[],int lo,int hi)
{
    int i,j,t;
    int key = data[hi]; //还是以最后一个元素作为哨兵，即主元元素
    i = lo-1;
    for (j =lo;j<=hi;j++)
        if(data[j]<key)
        {
            i++;
            t = data[j];
            data[j] = data[i];

```

```

    data[i] = t;
}
data[hi] = data[i+1]; //先,data[i+1]赋给 data[hi]
data[i+1] = key;      //后, 把事先保存的 key 值, 即 data[hi]赋给 data[i+1]
//不可调换这两条语句的顺序。

return i+1;
}

```

#### 提醒:

- 1、程序中尽量不要有任何多余的代码。
- 2、你最好绝对清楚的知道, 程序的某一步, 是该用 **if**, 还是该用 **while**, 等任何细节的东西。

ok, 以上程序的测试用例, 可以简单编写如下:

```

int main()
{
    int a[8]={3,8,7,1,2,5,6,4};
    QuickSort(a,0,N-1);
    for(int i=0;i<8;i++)
        cout<<a[i]<<endl;
    return 0;
}

```

当然, 如果, 你如果对以上的测试用例不够放心, 可以采取 1~10000 的随机数进行极限测试, 保证程序的万无一失 (主函数中测试用的随机数例子, 即所谓的“**极限**”测试, 下文会给出)。

至于上述程序是什么结果, 相信, 不用我再啰嗦。:D。

补充一种写法:

```

void quickSort(int p, int q)
{
    if(p < q)
    {
        int x = a[p]; //以第一个元素为主元
        int i = p;
        for(int j = p+1; j < q; j++)

```



```

{
  if(a[j] < x)
  {
    i++;
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }
}
int temp = a[p];
a[p] = a[i];
a[i] = temp;
quickSort(p, i);
quickSort(i+1, q);
}
}

```

## 二、国内教材双向扫描版

国内教材上一般所用的通用版本，是我写的第二篇文章中所提到的霍尔排序或其变形，[而非上述所述的算法导论上的版本](#)。而且，现在网上一般的朋友，也是更倾向于采用此种思路来实现快速排序算法。ok，请看：

### 2.1、Hoare 版本

那么，什么是霍尔提出的快速排序版本呢？如下，即是：

HOARE-PARTITION(A, p, r)

```

1  x ← A[p]
2  i ← p - 1
3  j ← r + 1
4  while TRUE
5    do repeat j ← j - 1
6      until A[j] ≤ x
7    repeat i ← i + 1
8      until A[i] ≥ x
9    if i < j

```

```

10      then exchange A[i] <-> A[j]
11      else return j

```

同样，根据以上伪代码，不难写出以下的 c/c++代码：

```

1. //此处原来的代码有几点错误，后听从了 Joshua 的建议，现修改如下：
2. int partition(int data[],int lo,int hi) //。
3. {
4.     int key=data[lo];
5.     int l=lo-1;
6.     int h=hi+1;
7.     for(;;)
8.     {
9.         do{
10.             h--;
11.         }while(data[h]>key);
12.
13.         do{
14.             l++;
15.         }while(data[l]<key);
16.
17.         if(l<h)
18.         {
19.             swap(data[l],data[h]);
20.         }
21.         else
22.         {
23.             return h;
24.             //各位注意了，这里的返回值是 h。不是返回各位习以为常的枢纽元素，即不是 l 之类的。
25.         }
26.     }
27. }

```

或者原来的代码修改成这样（已经过测试，有误）：

```

int partition(int data[],int lo,int hi) //。
{
int key=data[lo];
int l=lo;
int h=hi;
for(;;)
{
while(data[h]>key) //不能加 “=”

```

```

h--;
while(data[l]<key) //不能加 “=”
l++;
if(l<h)
{
swap(data[l],data[h]);
}
else
{
return h; //各位注意了，这里的返回值是 h。不是返回各位习以为常的枢纽元素，即不是
l之类的。
}
}
} //这个版本，已经证明有误，因为当 data[l] == data[h] == key 的时候，将会进入死循环，所以淘汰。因此，使用上面的 do-while 形式吧。

```

读者可以试下，对这个序列进行排序，用上述淘汰版本将立马进入死循环：int data[16]={ 1000, 0, 6, 5, 4, 3, 2, 1, 7, 156, 44, 23, 123, 11, 5 }；。

或者，如朋友颜沙所说：

如果 data 数组有相同元素就可能陷入死循环，比如：

```

2 3 4 5 6 2
l->|      |<-h

```

交换 l 和 h 单元后重新又回到：

```

2 3 4 5 6 2
l->|      |<-h

```

而第一个程序不存在这种情况，因为它总是在 l 和 h 调整后比较，也就是 l 终究会大于等于 h。

.

相信，你已经看出来了，上述的第一个程序中 partition 过程的返回值 h 并不是枢纽元的位置，但是仍然保证了  $A[p..j] \leq A[j+1...q]$ 。

这种方法在效率上与以下将要介绍的 Hoare 的几个变形版本差别甚微，只不过是上述代码相对更为紧凑点而已。

## 2.2、Hoare 的几个变形版本

ok, 可能, 你对上述的最初的霍尔排序 partition 过程, 理解比较费力, 没关系, 我再写几种变形, 相信, 你立马就能了解此双向扫描是怎么一回事了。

```
int partition(int data[],int lo,int hi) //双向扫描。
{
    int key=data[lo]; //以第一个元素为主元
    int l=lo;
    int h=hi;
    while(l<h)
    {
        while(key<=data[h] && l<h)
            h--;
        data[l]=data[h];
        while(data[l]<=key && l<h)
            l++;
        data[h]=data[l];
    }
    data[l]=key; //1.key。只有出现要赋值的的情况, 才事先保存好第一个元素的值。
    return l; //这里和以下所有的 Hoare 的变形版本都是返回的是枢纽元素, 即主元元素 l。
}
```

补充说明: 同样, 还是举上述那个例子, 如下为第一趟排序 (更多详尽的分析请参考第二篇文章):

第一趟(五步曲):

```
a: 3  8  7  1  2  5  6  4 //以第一个元素为主元
    2  8  7  1    5  6  4
b: 2    7  1  8  5  6  4
c: 2  1  7    8  5  6  4
d: 2  1    7  8  5  6  4
e: 2  1  3  7  8  5  6  4 //最后补上, 关键字 3
```

然后, 对整个数组进行递归排序:

```
void QuickSort(int data[], int lo, int hi)
{
```

```

    if (lo<hi)
    {
        int k = partition(data, lo, hi);
        QuickSort(data, lo, k-1);
        QuickSort(data, k+1, hi);
    }
}

```

当然，你也可以这么写，把递归过程写在同一个排序过程里：

```

void QuickSort(int data[],int lo,int hi)
{
    int i,j,temp;
    temp=data[lo]; //还是以第一个元素为主元。
    i=lo;
    j=hi;
    if(lo>hi)
        return;
    while(i!=j)
    {
        while(data[j]>=temp && j>i)
            j--;
        if(j>i)
            data[i++]=data[j];
        while(data[i]<=temp && j>i)
            i++;
        if(j>i)
            data[j--]=data[i];
    }
    data[i]=temp; //2.temp。同上，返回的是枢纽元素，即主元元素。
    QuickSort(data,lo,i-1); //递归左边
    QuickSort(data,i+1,hi); //递归右边
}

```

或者，如下：

```

1. void quicksort (int[] a, int lo, int hi)

```

```

2. {
3. // lo is the lower index, hi is the upper index
4. // of the region of array a that is to be sorted
5.     int i=lo, j=hi, h;
6.
7.     // comparison element x
8.     int x=a[(lo+hi)/2];
9.
10.    // partition
11.    do
12.    {
13.        while (a[i]<x) i++;
14.        while (a[j]>x) j--;
15.        if (i<=j)
16.        {
17.            h=a[i]; a[i]=a[j]; a[j]=h;
18.            i++; j--;
19.        }
20.    } while (i<=j);
21.
22.    // recursion
23.    if (lo<j) quicksort(a, lo, j);
24.    if (i<hi) quicksort(a, i, hi);
25. }

```

另，本人在一本国内的数据结构教材上（[注](#)，此处非指**严**那本），看到的一种写法，发现如下问题：一、冗余繁杂，二、错误之处无所不在，除了会犯一些注释上的错误，一些最基本的代码，都会弄错。详情，如下：

```
void QuickSort(int data[],int lo,int hi)
```

```
{
```

```
int i,j,key;
```

```
if(lo<hi)
```

```
{
```

```
    i=lo;
```

```
    j=hi;
```

```
    key=data[lo];
```

[//已经测试：原教材上，原句为“data\[0\]=data\[lo\];”，有误。](#)

[//因为只能用一个临时变量 key 保存着主元，data\[lo\]，而若为以上，则相当于覆盖原元素 data\[0\]的值了。](#)

```
    do
```

```

    {
while(data[j]>=key&& i<j)
    j--;
if(i<j)
{
    data[i]=data[j];
    //i++; 这是教材上的语句，为使代码简洁，我特意去掉。
}
while(data[i]<=key&& i<j)
    i++;
if(i<j)
{
    data[j]=data[i];
    //j--; 这是教材上的语句，为使代码简洁，我特意去掉。
}
    }while(i!=j);
    data[i]=key;    //3.key。
//已经测试：原教材上，原句为“data[i]=data[0];”，有误。
    QuickSort(data,lo,i-1);    //对标准值左半部递归调用本函数
    QuickSort(data,i+1,hi);    //对标准值右半部递归调用本函数
}
}

```

然后，你能很轻易的看到，这个写法，与上是同一写法，之所以写出来，是希望各位慎看国内的教材，多多质疑+思考，勿轻信。

ok，再给出一种取中间元素为主元的实现：

```

void QuickSort(int data[],int lo,int hi)
{
    int pivot,l,r,temp;
    l = lo;
    r = hi;
    pivot=data[(lo+hi)/2]; //取中位值作为分界值
    while(l<r)
    {

```

```

while(data[l]<pivot)
    ++l;
while(data[r]>pivot)
    --r;
if(l>=r)
    break;
temp = data[l];
data[l] = data[r];
data[r] = temp;
++l;
--r;
}
if(l==r)
    l++;
if(lo<r)
    QuickSort(data,lo,l-1);
if(l<hi)
    QuickSort(data,r+1,hi);
}

```

或者，这样写：

```

void quickSort(int arr[], int left, int right)
{
    int i = left, j = right;
    int tmp;
    int pivot = arr[(left + right) / 2]; //取中间元素为主元

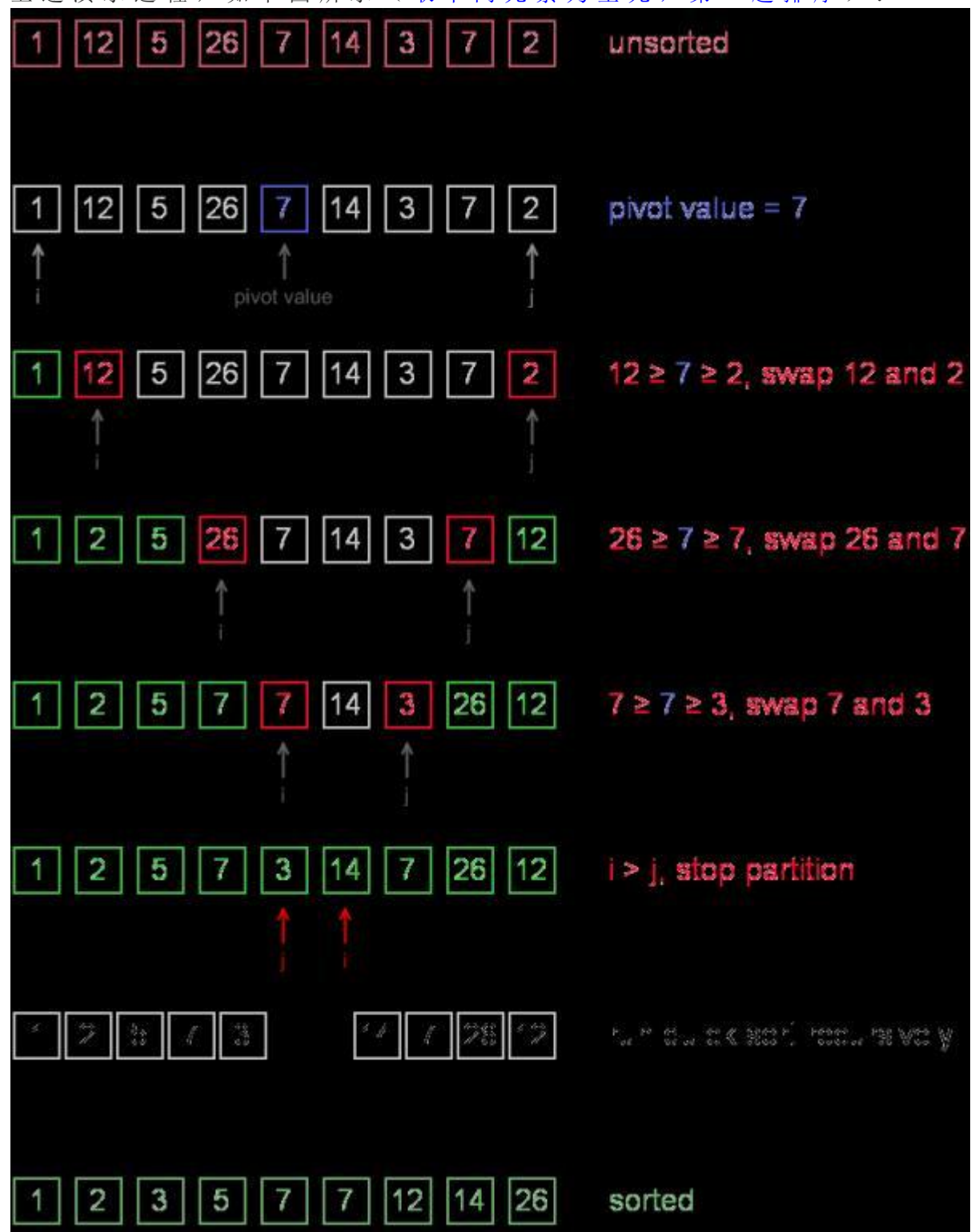
    /* partition */
    while (i <= j)
    {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i <= j)

```



```
{  
    tmp = arr[i];  
    arr[i] = arr[j];  
    arr[j] = tmp;  
    i++;  
    j--;  
}  
}  
}
```

上述演示过程，如下图所示（取中间元素为主元，第一趟排序）：



### 三、快速排序的随机化版本

以下是完整测试程序，由于给的注释够详尽了，就再做多余的解释了：

//交换两个元素值，咱们换一种方式，采取引用“&”

```
void swap(int& a , int& b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

//返回属于[lo,hi)的随机整数

```
int rand(int lo,int hi)
{
    int size = hi-lo+1;
    return lo+ rand()%size;
}
```

//分割，换一种方式，采取指针 a 指向数组中第一个元素

```
int RandPartition(int* data, int lo , int hi)
{
    //普通的分割方法和随机化分割方法的区别就在于下面三行
    swap(data[rand(lo,hi)], data[lo]);
    int key = data[lo];
    int i = lo;

    for(int j=lo+1; j<=hi; j++)
    {
        if(data[j]<=key)
        {
            i = i+1;
            swap(data[i], data[j]);
        }
    }
    swap(data[i],data[lo]);
}
```

```

    return i;
}

//逐步分割排序
void RandQuickSortMid(int* data, int lo, int hi)
{
    if(lo<hi)
    {
        int k = RandPartition(data,lo,hi);
        RandQuickSortMid(data,lo,k-1);
        RandQuickSortMid(data,k+1,hi);
    }
}

int main()
{
    const int N = 100; //此就是上文说所的“极限”测试。为了保证程序的准确无误，你也可以让
    N=10000。
    int *data = new int[N];
        for(int i =0; i<N; i++)
            data[i] = rand(); //同样，随机化的版本，采取随机输入。
        for(i=0; i<N; i++)
            cout<<data[i]<<" ";
        RandQuickSortMid(data,0,N-1);
    cout<<endl;
    for(i=0; i<N; i++)
        cout<<data[i]<<" ";
    cout<<endl;
        return 0;
}

```

#### 四、三数取中分割法

我想，如果你爱思考，可能你已经在想一个问题了，那就是，像上面的程序版本，其中算法导论上采取单向扫描中，是以最后一个元素为枢纽元素，即主元，而在 **Hoare** 版本及其几个变形中，都是以第一个元素、或中间元素为主元，最后，上述给的快速排序算法的随

机化版本，则是以序列中任一个元素作为主元。

那么，枢纽元素的选取，即主元元素的选取是否决定快速排序最终的效率列？

答案是肯定的，当我们采取 `data[lo],data[mid],data[hi]`三者之中的那个第二大的元素为主元时，便能尽最大限度保证快速排序算法不会出现  $O(N^2)$  的最坏情况。这就是所谓的三数取中分割方法。当然，针对的还是那个 **Partition** 过程。

ok，直接写代码：

//三数取中分割方法

```
int RandPartition(int* a, int p , int q)
```

```
{
```

```
//三数取中方法的关键就在于下述六行，
```

```
int m=(p+q)/2;
```

```
if(a[p]<a[m])
```

```
    swap(a[p],a[m]);
```

```
if(a[q]<a[m])
```

```
    swap(a[q],a[m]);
```

```
if(a[q]<a[p])
```

```
    swap(a[q],a[p]);
```

```
int key = a[p];
```

```
int i = p;
```

```
for(int j = p+1; j <= q; j++)
```

```
{
```

```
    if(a[j] <= key)
```

```
    {
```

```
        i = i+1;
```

```
        if(i != j)
```

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

```
    }
```

```
}
```

```
swap(a[i],a[p]);
```

```
return i;
```

```
}
```

```

void QuickSort(int data[], int lo, int hi)
{
    if (lo<hi)
    {
        int k = RandPartition(data, lo, hi);
        QuickSort(data, lo, k-1);
        QuickSort(data, k+1, hi);
    }
}

```

经过测试，这种方法可行且有效，不过到底其性能、效率有多好，还有待日后进一步的测试。

## 第二部分、快速排序的非递归版

ok，相信，您已经看到，上述所有的快速排序算法，都是递归版本的，那还有什么办法可以实现此快速排序算法列？对了，递归，与之相对的，就是非递归了。

以下，就是快速排序算法的非递归实现：

```

template <class T>
int RandPartition(T data[],int lo,int hi)
{
    T v=data[lo];
    while(lo<hi)
    {
        while(lo<hi && data[hi]>=v)
            hi--;
        data[lo]=data[hi];
        while(lo<hi && data[lo]<=v)
            lo++;
        data[hi]=data[lo];
    }
    data[lo]=v;
    return lo;
}

```

//快速排序的非递归算法

```
template <class T>
```

```
void QuickSort(T data[],int lo,int hi)
```

```
{
```

```
    stack<int> st;
```

```
    int key;
```

```
    do{
```

```
        while(lo<hi)
```

```
        {
```

```
            key=partition(data,lo,hi);
```

```
            //递归的本质是什么?对了，就是借助栈，进栈，出栈来实现的。
```

```
            if( (key-lo)<(key-hi) )
```

```
            {
```

```
                st.push(key+1);
```

```
                st.push(hi);
```

```
                hi=key-1;
```

```
            }
```

```
            else
```

```
            {
```

```
                st.push(lo);
```

```
                st.push(key-1);
```

```
                lo=key+1;
```

```
            }
```

```
        }
```

```
    if(st.empty())
```

```
        return;
```

```
    hi=st.top();
```

```
    st.pop();
```

```
    lo=st.top();
```

```
    st.pop();
```

```
    }while(1);
```

```
}
```

```
void QuickSort(int data[], int lo, int hi)
```

```
{
```

```
    if (lo<hi)
```

```

{
    int k = RandPartition(data, lo, hi);
    QuickSort(data, lo, k-1);
    QuickSort(data, k+1, hi);
}
}

```

如果你还尚不知道快速排序算法的原理与算法思想，请参考本人写的关于快速排序算法的前俩篇文章：[一之续、快速排序算法的深入分析](#)，及[二、快速排序算法](#)。如果您看完了此篇文章后，还是不知如何从头实现快速排序算法，那么好吧，伸出手指，数数，1,2,3,4,5....数到 100 之后，再来看此文。

-----

据本文评论里头网友 ybt631 的建议，表示非常感谢，并补充阐述下所谓的**并行快速排序**：

Intel Threading Building Blocks(简称 TBB)是一个 C++的并行编程模板库，它能使你的程序充分利用多核 CPU 的性能优势，方便使用，效率很高。

以下是，**parallel\_sort.h** 头文件中的关键代码：

```

1. 00039 template<typename RandomAccessIterator, typename Compare>
2. 00040 class quick_sort_range: private no_assign {
3. 00041
4. 00042     inline size_t median_of_three(const RandomAccessIterator &array, s
size_t l, size_t m, size_t r) const {
5. 00043         return comp(array[l], array[m]) ? ( comp(array[m], array[r]) ?
m : ( comp( array[l], array[r]) ? r : l ) )
6. 00044             : ( comp(array[r], array[m]) ?
m : ( comp( array[r], array[l] ) ? r : l ) );
7. 00045     }
8. 00046
9. 00047     inline size_t pseudo_median_of_nine( const RandomAccessIterator &a
rray, const quick_sort_range &range ) const {
10. 00048         size_t offset = range.size/8u;
11. 00049         return median_of_three(array,
12. 00050                                 median_of_three(array, 0, offset, offse
t*2),
13. 00051                                 median_of_three(array, offset*3, offset
*4, offset*5),
14. 00052                                 median_of_three(array, offset*6, offset
*7, range.size - 1) );
15. 00053

```



```

16. 00054     }
17. 00055
18. 00056 public:
19. 00057
20. 00058     static const size_t grainsize = 500;
21. 00059     const Compare &comp;
22. 00060     RandomAccessIterator begin;
23. 00061     size_t size;
24. 00062
25. 00063     quick_sort_range( RandomAccessIterator begin_, size_t size_, const
        Compare &comp_ ) :
26. 00064         comp(comp_), begin(begin_), size(size_) {}
27. 00065
28. 00066     bool empty() const {return size==0;}
29. 00067     bool is_divisible() const {return size>=grainsize;}
30. 00068
31. 00069     quick_sort_range( quick_sort_range& range, split ) : comp(range.co
        mp) {
32. 00070         RandomAccessIterator array = range.begin;
33. 00071         RandomAccessIterator key0 = range.begin;
34. 00072         size_t m = pseudo_median_of_nine(array, range);
35. 00073         if (m) std::swap ( array[0], array[m] );
36. 00074
37. 00075         size_t i=0;
38. 00076         size_t j=range.size;
39. 00077         // Partition interval [i+1,j-1] with key *key0.
40. 00078         for(;;) {
41. 00079             __TBB_ASSERT( i<j, NULL );
42. 00080             // Loop must terminate since array[1]==*key0.
43. 00081             do {
44. 00082                 --j;
45. 00083                 __TBB_ASSERT( i<=j, "bad ordering relation?" );
46. 00084             } while( comp( *key0, array[j] ));
47. 00085             do {
48. 00086                 __TBB_ASSERT( i<=j, NULL );
49. 00087                 if( i==j ) goto partition;
50. 00088                 ++i;
51. 00089             } while( comp( array[i],*key0 ));
52. 00090             if( i==j ) goto partition;
53. 00091             std::swap( array[i], array[j] );
54. 00092         }
55. 00093 partition:
56. 00094         // Put the partition key were it belongs
57. 00095         std::swap( array[j], *key0 );

```

```

58. 00096        // array[l..j) is less or equal to key.
59. 00097        // array(j..r) is greater or equal to key.
60. 00098        // array[j] is equal to key
61. 00099        i=j+1;
62. 00100        begin = array+i;
63. 00101        size = range.size-i;
64. 00102        range.size = j;
65. 00103    }
66. 00104 };
67. 00105
68. ....
69. 00218 #endif

```

再贴一下插入排序、快速排序之其中的俩种版本、及插入排序与快速排序结合运用的实现代码，如下：

```

1.  /// 插入排序,最坏情况下为  $O(n^2)$ 
2.  template< typename InPos, typename ValueType >
3.  void _isort( InPos posBegin_, InPos posEnd_, ValueType* )
4.  {
5.  /*****
6.  *      伪代码如下:
7.  *          for i = [1, n)
8.  *              t = x
9.  *              for( j = i; j > 0 && x[j-1] > t; j-- )
10. *                  x[j] = x[j-1]
11. *                  x[j] = t
12. *****/
13. if( posBegin_ == posEnd_ )
14. {
15.     return;
16. }
17.
18. /// 循环迭代, 将每个元素插入到合适的位置
19. for( InPos pos = posBegin_; pos != posEnd_; ++pos )
20. {
21.     ValueType Val = *pos;
22.     InPos posPrev = pos;
23.     InPos pos2 = pos;
24.     /// 当元素比前一个元素大时, 交换
25.     for( ; pos2 != posBegin_ && *(--posPrev) > Val ; --pos2 )
26.     {

```

```

27.     *pos2 = *posPrev;
28. }
29. *pos2 = Val;
30. }
31. }
32.
33. /// 快速排序 1, 平均情况下需要 O(nlogn)的时间
34. template< typename InPos >
35. inline void qsort1( InPos posBegin_, InPos posEnd_ )
36. {
37. /*****
    *
38. *     伪代码如下:
39. *         void qsort(l, n)
40. *             if(l >= u)
41. *                 return;
42. *             m = l
43. *             for i = [l+1, u]
44. *                 if( x < x[l]
45. *                     swap(++m, i)
46. *             swap(l, m)
47. *             qsort(l, m-1)
48. *             qsort(m+1, u)
49. *****/
    */
50. if( posBegin_ == posEnd_ )
51. {
52.     return;
53. }
54.
55. /// 将比第一个元素小的元素移至前半部
56. InPos pos = posBegin_;
57. InPos posLess = posBegin_;
58. for( ++pos; pos != posEnd_; ++pos )
59. {
60.     if( *pos < *posBegin_ )
61.     {
62.         swap( *pos, *(++posLess) );
63.     }
64. }
65.
66. /// 把第一个元素插到两快元素中央
67. swap( *posBegin_, *(posLess) );
68.

```

```

69. /// 对前半部、后半部执行快速排序
70. qsort1(posBegin_, posLess);
71. qsort1(++posLess, posEnd_);
72. };
73.
74. /// 快速排序 2, 原理与 1 基本相同, 通过两端同时迭代加快平均速度
75. template<typename InPos>
76. void qsort2( InPos posBegin_, InPos posEnd_ )
77. {
78. if( distance(posBegin_, posEnd_) <= 0 )
79. {
80. return;
81. }
82.
83. InPos posL = posBegin_;
84. InPos posR = posEnd_;
85.
86. while( true )
87. {
88. /// 找到不小于第一个元素的数
89. do
90. {
91. ++posL;
92. }while( *posL < *posBegin_ && posL != posEnd_ );
93.
94. /// 找到不大于第一个元素的数
95. do
96. {
97. --posR;
98. } while ( *posR > *posBegin_ );
99.
100. /// 两个区域交叉时跳出循环
101. if( distance(posL, posR) <= 0 )
102. {
103. break;
104. }
105. /// 交换找到的元素
106. swap(*posL, *posR);
107. }
108.
109. /// 将第一个元素换到合适的位置
110. swap(*posBegin_, *posR);
111. /// 对前半部、后半部执行快速排序 2
112. qsort2(posBegin_, posR);

```

```

113. qsort2(++posR, posEnd_);
114. }
115.
116. /// 当元素个数小与 g_iSortMax 时使用插入排序, g_iSortMax 是根据 STL 库选取的
117. const int g_iSortMax = 32;
118. /// 该排序算法是快速排序与插入排序的结合
119. template<typename InPos>
120. void qsort3( InPos posBegin_, InPos posEnd_ )
121. {
122.     if( distance(posBegin_, posEnd_) <= 0 )
123.     {
124.         return;
125.     }
126.
127.     /// 小与 g_iSortMax 时使用插入排序
128.     if( distance(posBegin_, posEnd_) <= g_iSortMax )
129.     {
130.         return isort(posBegin_, posEnd_);
131.     }
132.
133.     /// 大与 g_iSortMax 时使用快速排序
134.     InPos posL = posBegin_;
135.     InPos posR = posEnd_;
136.
137.     while( true )
138.     {
139.         do
140.         {
141.             ++posL;
142.         } while( *posL < *posBegin_ && posL != posEnd_ );
143.
144.         do
145.         {
146.             --posR;
147.         } while ( *posR > *posBegin_ );
148.
149.         if( distance(posL, posR) <= 0 )
150.         {
151.             break;
152.         }
153.         swap(*posL, *posR);
154.     }
155.     swap(*posBegin_, *posR);
156.     qsort3(posBegin_, posR);

```

```
157. qsort3(++posR, posEnd_);
158. }
```

## 十三、通过浙大上机复试试题学 SPFA 算法

作者: July、sunbaigui。二零一一年三月二十五日。

出处: [http://blog.csdn.net/v\\_JULY\\_v](http://blog.csdn.net/v_JULY_v)。

---

### 前言:

本人不喜欢写诸如“[如何学算法](#)”此类文章，一来怕被人认为是自以为是，二来话题太泛，怕扯得太远，反而不着边际。所以，一直不打算写怎么学习算法此类文章。

不过，鉴于读者的热心支持与关注，给出以下几点小小的建议，仅供参考：

1、算法，浩如烟海，找到自己感兴趣的那个分支，或那个点来学习，然后，一往无前的深入探究下去。

2、兴趣第一，一切，由着你的兴趣走，忌浮躁。

3、思维敏捷。给你一道常见的题目，你的头脑中应该立刻能冒出解决这道问题的最适用的数据结构，以及算法。

4、随兴趣，多刷题。ACM 题。poj，面试题，包括下文将出现的研究生复试上机考试题，都可以作为你的编程练习题库。

5、多实践，多思考。学任何一个算法，反复研究，反复思考，反复实现。

6、数据结构是一切的基石。不必太过专注于算法，一切算法的实现，原理都是依托数据结构来实现的。弄懂了一个数据结构，你也就通了一大片算法。

7、学算法，重优化。

8、学习算法的高明之处不在于某个算法运用得有多自如，而在于通晓一个算法的内部原理，运作机制，及其来龙去脉。

ok，话不再多。希望，对你有用。

接下来，咱们来通过最近几年的浙大研究生复试上机试题，来学习或巩固常用的算法。

### 浙大研究生复试 2010 年上机试题-最短路径问题

问题描述：

给你  $n$  个点， $m$  条无向边，每条边都有长度  $d$  和花费  $p$ ，给你起点  $s$  终点  $t$ ，要求输出起点到终点的最短距离及其花费，如果最短距离有多条路线，则输出花费最少的。

输入：输入  $n, m$ ，点的编号是  $1 \sim n$ ，然后是  $m$  行，每行 4 个数  $a, b, d, p$ ，表示  $a$  和  $b$  之间有一条边，且其长度为  $d$ ，花费为  $p$ 。最后一行是两个数  $s, t$ ；起点  $s$ ，终点  $t$ 。 $n$  和  $m$  为 0 时输入结束。

( $1 < n \leq 1000$ ,  $0 < m < 100000$ ,  $s \neq t$ )

输出：一行有两个数，最短距离及其花费。（下文，会详细解决）

## 几个最短路径算法的比较

ok, 怎么解决上述的最短路径问题? 提到最短路径问题, 想必大家会立马想到 Dijkstra 算法, 但 Dijkstra 算法的效率如何?

我们知道, Dijkstra 算法的运行时间, 依赖于其最小优先队列的采取何种具体实现决定, 而最小优先队列可有以下三种实现方法:

1、利用从 1 至  $|V|$  编好号的顶点, 简单地将每一个  $d[v]$  存入一个数组中对应的第  $v$  项, 如上述 DIJKSTRA ( $G, w, s$ ) 所示, Dijkstra 算法的运行时间为  $O(V^2 + E)$ 。

2、如果是二叉/项堆实现最小优先队列的话, EXTRACT-MIN( $Q$ ) 的运行时间为  $O(V \lg V)$ ,

所以, Dijkstra 算法的运行时间为  $O(V \lg V + E \lg V)$ ,

若所有顶点都是从源点可达的话,  $O((V + E) \lg V) = O(E \lg V)$ 。

当是稀疏图时, 则  $E = O(V^2 / \lg V)$ , 此 Dijkstra 算法的运行时间为  $O(V^2)$ 。

3、采用斐波那契堆实现最小优先队列的话, EXTRACT-MIN( $Q$ ) 的运行时间为  $O(V \lg V)$ ,

所以, 此 Dijkstra 算法的运行时间即为  $O(V \lg V + E)$ 。

综上所述, 此最小优先队列的三种实现方法比较如下:

EXTRACT-MIN + RELAX

I、简单方式:  $O(V^2 + E \cdot 1)$

II、二叉/项堆:  $O(V \lg V + |E| \lg V)$

源点可达:  $O(E \lg V)$

稀疏图时, 有  $E = O(V^2 / \lg V)$ ,

$\Rightarrow O(V^2)$

III、斐波那契堆:  $O(V \lg V + E)$

是的, 由上, 我们已经看出来了, Dijkstra 算法最快的实现是, 采用斐波那契堆作最小优先队列, 算法时间复杂度, 可达到  $O(V \lg V + E)$ 。

但是? 如果题目有时间上的限制?  $V \lg V + E$  的时间复杂度, 能否一定满足要求? 我们试图寻找一种解决此最短路径问题更快的算法。

这个时候, 我们想到了 Bellman-Ford 算法: 求单源最短路, 可以判断有无负权回路 (若有, 则不存在最短路), 时效性较好, 时间复杂度  $O(VE)$ 。不仅时效性好于上述的 Dijkstra 算法, 还能判断回路中是否有无负权回路。

既然, 想到了 Bellman-Ford 算法, 那么时间上, 是否还能做进一步的突破。对了, 我们中国人自己的算法--SPFA 算法: SPFA 算法, Bellman-Ford 的队列优化, 时效性相对好, 时间复杂度  $O(kE)$ 。(  $k < V$  )。

是的, 线性的时间复杂度, 我想, 再苛刻的题目, 或多或少, 也能满足要求了。

## 什么是 SPFA 算法

在上一篇文章[二之三续、Dijkstra 算法+Heap 堆的完整 c 实现源码](#)中，我们给出了 Dijkstra+Heap 堆的实现。其实，在稀疏图中对单源问题来说 SPFA 的效率略高于 Heap+Dijkstra；对于无向图上的 APSP(All Pairs Shortest Paths)问题，SPFA 算法加上优化后效率更是远高于 Heap+Dijkstra。

那么，究竟什么是 SPFA 算法呢？

**SPFA 算法**，Shortest Path Faster Algorithm，是由西南交通大学段凡丁于 1994 年发表的。正如上述所说，Dijkstra 算法无法用于负权回路，很多时候，如果给定的图存在负权边，这时类似 Dijkstra 等算法便没有了用武之地，而 Bellman-Ford 算法的复杂度又过高，SPFA 算法便派上用场了。

简洁起见，我们约定有向加权图  $G$  不存在负权回路，即最短路径一定存在。当然，我们可以在执行该算法前做一次拓扑排序，以判断是否存在负权回路。

我们用数组  $d$  记录每个结点的最短路径估计值，而且用邻接表来存储图  $G$ 。

我们采取的方法是动态逼近法：设立一个先进先出的队列用来保存待优化的结点，优化时每次取出队首结点  $u$ ，并且用  $u$  点当前的最短路径估计值对离开  $u$  点所指向的结点  $v$  进行松弛操作，如果  $v$  点的最短路径估计值有所调整，且  $v$  点不在当前的队列中，就将  $v$  点放入队尾。

这样不断从队列中取出结点来进行松弛操作，直至队列空为止。

定理：只要最短路径存在，上述 SPFA 算法必定能求出最小值。

期望的时间复杂度  $O(ke)$ ，其中  $k$  为所有顶点进队的平均次数，可以证明  $k$  一般小于等于 2。

SPFA 实际上是 Bellman-Ford 基础上的优化，以下，是此算法的伪代码：

//这里的  $q$  数组表示的是节点是否在队列中，如  $q[v]=1$  则点  $v$  在队列中

SPFA( $G, w, s$ )

1. INITIALIZE-SINGLE-SOURCE( $G, s$ )

2.  $Q \leftarrow \&\text{Oslash}$ ;

3. for each vertex  $v \in V[G]$

4.  $q[v]=0$

5. ENQUEUE( $Q, s$ )

6.  $q[s]=1$

7. while  $Q \neq \&\text{Oslash}$ ;

8. do  $u \leftarrow \text{DEQUEUE}(Q)$

9.  $q[u]=0$

10. for each edge  $(u, v) \in E[G]$

11. do  $t \leftarrow d[v]$

12. RELAX( $u, v, w$ )



```
13.  $\pi[v] \leftarrow u$ 
14. if ( $d[v] < t$ )
15. ENQUEUE( $Q, v$ )
16.  $q[v]=1$ 
```

SPFA 是 Bellman-Ford 的队列优化，时效性相对好，时间复杂度  $O(kE)$ 。 $(k \ll V)$ 。与 Bellman-ford 算法类似，SPFA 算法采用一系列的松弛操作以得到从某一个节点出发到达图中其它所有节点的最短路径。所不同的是，SPFA 算法通过维护一个队列，使得一个节点的当前最短路径被更新之后没有必要立刻去更新其他的节点，从而大大减少了重复的操作次数。

与 Dijkstra 算法与 Bellman-ford 算法不同，SPFA 的算法时间效率是不稳定的，即它对于不同的图所需要的时间有很大的差别。在最好情形下，每一个节点都只入队一次，则算法实际上变为广度优先遍历，其时间复杂度仅为  $O(E)$ 。另一方面，存在这样的例子，使得每一个节点都被入队  $(V-1)$  次，此时算法退化为 Bellman-ford 算法，其时间复杂度为  $O(VE)$ 。有研究指出在随机情形下平均一个节点入队的次数不超过 2 次，因此算法平均的时间复杂度为  $O(E)$ ，甚至优于使用堆优化过的 Dijkstra 算法。

## 最短路径问题的解决

浙大研究生复试 2010 年上机试题-最短路径问题

问题描述：

给你  $n$  个点， $m$  条无向边，每条边都有长度  $d$  和花费  $p$ ，给你起点  $s$  终点  $t$ ，要求输出起点到终点的最短距离及其花费，如果最短距离有多条路线，则输出花费最少的。

输入：输入  $n, m$ ，点的编号是  $1 \sim n$ ，然后是  $m$  行，每行 4 个数  $a, b, d, p$ ，表示  $a$  和  $b$  之间有一条边，且其长度为  $d$ ，花费为  $p$ 。最后一行是两个数  $s, t$ ；起点  $s$ ，终点  $t$ 。 $n$  和  $m$  为 0 时输入结束。

$(1 < n \leq 1000, 0 < m < 100000, s \neq t)$

输出：一行有两个数，最短距离及其花费。

接下来，咱们便利用 SPFA 算法来解决此最短路径问题。

以下，便引用 sunbaigui 的代码来说明此问题：

声明几个变量：

[view plaincopy to clipboardprint?](#)

```
1. int d[1005][2];
2. bool used[1005];
3. vector<Node>map[1005];
4. int N, M, S, T;
```

建个数据结构：

[view plaincopy to clipboardprint?](#)

```

1. struct Node
2. {
3.     int x,y,z;
4.     Node(int a=0,int b=0,int c=0):x(a),y(b),z(c){}
5. };

```

以下是关键代码

[view plaincopy to clipboardprint?](#)

```

1. //sunbaigui:
2. //首先将起点弹入队列，用 used 数组标记 i 节点是否在队列中，
3. //然后从队列中弹出节点，判断从这个弹出节点能到达的每个节点的距离是否小于已得到的距离，
4. //如果是则更新距离，然后将其弹入队列，修改 used 数组。
5. void spfa()
6. {
7.     queue<int>q;    //构造一个队列
8.     q.push(S);
9.     memset(used,false,sizeof(used));
10.    int i;
11.    for(i=1;i<=N;i++)
12.        d[i][0]=d[i][1]=-1;
13.    d[S][0]=d[S][1]=0;    //初始化
14.    while(!q.empty())
15.    {
16.        int node=q.front();
17.        q.pop();
18.        used[node]=false;
19.        int t,dis,p;
20.        for(i=0;i<map[node].size();i++)    //遍历
21.        {
22.            t=map[node][i].x;
23.            dis=map[node][i].y;
24.            p=map[node][i].z;
25.            if(d[t][0]==-1||d[t][0]>d[node][0]+dis)
26.            {
27.                d[t][0]=d[node][0]+dis;    //松弛操作
28.                d[t][1]=d[node][1]+p;
29.                if(!used[t])
30.                {
31.                    used[t]=true;
32.                    q.push(t);    //t 点不在当前队列中，放入队尾。

```

```

33.         }
34.     }
35.     else if(d[t][0]!=-1&& d[t][0]==d[node][0]+dis)
36.     {
37.         if(d[t][1]>d[node][1]+p)
38.         {
39.             d[t][1]=d[node][1]+p;
40.             if(!used[t])
41.             {
42.                 q.push(t);
43.                 used[t]=true;
44.             }
45.         }
46.     }
47. }
48.
49. }
50. }

```

主函数测试用例:

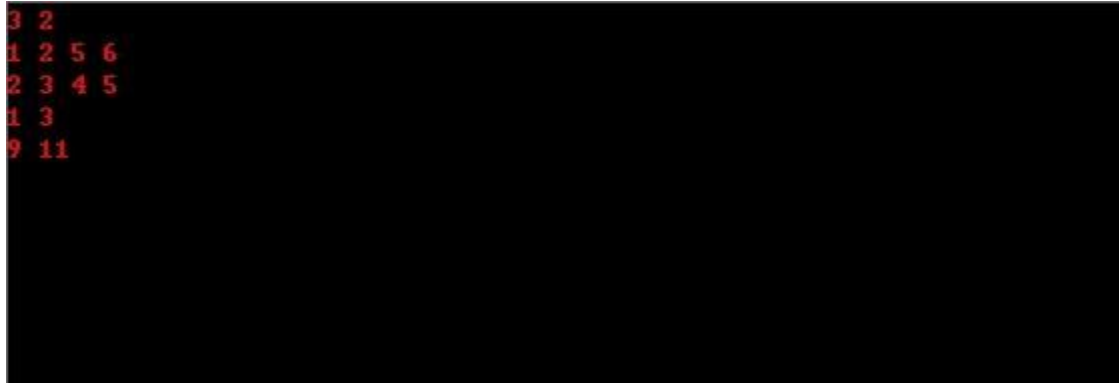
[view plaincopy to clipboardprint?](#)

```

1. int main()
2. {
3.     while(scanf("%d %d",&N,&M)!=EOF)
4.     {
5.         if(N==0&&M==0)break;
6.         int s,t,dis,p;
7.         int i;
8.         for(i=1;i<=N;i++)
9.             map[i].clear();
10.        while(M--)
11.        {
12.            scanf("%d %d %d %d",&s,&t,&dis,&p);
13.            map[s].push_back(Node(t,dis,p));
14.            map[t].push_back(Node(s,dis,p));
15.        }
16.        scanf("%d %d",&S,&T);
17.        spfa();
18.        printf("%d %d\n",d[T][0],d[T][1]);
19.    }
20.    return 0;
21. }

```

运行结果，是：



### 最后，总结一下（Lunatic Princess）：

(1)对于稀疏图，当然是 SPFA 的天下，不论是单源问题还是 APSP 问题，SPFA 的效率都是最高的，写起来也比 Dijkstra 简单。对于无向图的 APSP 问题还可以加入优化使效率提高 2 倍以上。

(2)对于稠密图，就得分情况讨论了。单源问题明显还是 Dijkstra 的势力范围，效率比 SPFA 要高 2-3 倍。APSP 问题，如果对时间要求不是那么严苛的话简简单单的 Floyd 即可满足要求，又快又不容易写错；否则就得使用 Dijkstra 或其他更高级的算法了。如果是无向图，则可以把 Dijkstra 扔掉了，加上优化的 SPFA 绝对是必然的选择。

	稠密图	稀疏图	有负权边
单源问题	Dijkstra+heap	SPFA(或 Dijkstra+heap,根据稀疏程度)	SPFA
APSP(无向图)	SPFA(优化)/Floyd	SPFA(优化)	SPFA(优化)
APSP(有向图)	Floyd	SPFA (或 Dijkstra+heap,根据稀疏程度)	SPFA

完。

## 十四：快速选择 SELECT 算法的深入分析与实现

作者：July。

出处：[http://blog.csdn.net/v\\_JULY\\_v](http://blog.csdn.net/v_JULY_v)。

## 前言

经典算法研究系列已经写了十三个算法，共计 22 篇文章（详情，见这：[十三个经典算法研究与总结、目录+索引](#)），我很怕我自己不再把这个算法系列给继续写下去了。沉思良久，到底是不想因为要创作狂想曲系列而耽搁这个经典算法研究系列，何况它，至今反响还不错。

ok，狂想曲第三章提出了一个算法，就是快速选择 **SELECT** 算法，关于这个 **SELECT** 算法通过选取数组中中位数的中位数作为枢纽元能保证在最坏情况下，亦能做到线性  $O(N)$  的时间复杂度的证明，在狂想曲第三章也已经给出。

本文咱们从快速排序算法分析开始（因为如你所知，快速选择算法与快速排序算法在 **partition** 划分过程上是类似的），参考 Mark 的数据结构与算法分析-c 语言描述一书，而后逐步深入分析快速选择 **SELECT** 算法，最后，给出 **SELECT** 算法的程序实现。

同时，本文有部分内容来自狂想曲系列第三章，也算是对[第三章、寻找最小的 k 个数](#)的一个总结。yeah，有任何问题，欢迎各位批评指正，如果你挑出了本文章或[本 blog](#) 任何一个问题或错误，当即免费给予单独赠送本 [blog](#) 最新一期第 6 期的博文集锦 CHM 文件，谢谢。

## 第一节、快速排序

### 1.1、快速排序算法的介绍

关于快速排序算法，本人已经写了 3 篇文章（可参见其中的两篇：1、[十二、快速排序算法之所有版本的 c/c++实现](#)，2、[一之续、快速排序算法的深入分析](#)），为何又要旧事重提？正如很多事物都有相似的地方，而咱们面临的问题--快速选择算法中的划分过程等同于快速排序，所以，在分析快速选择 **SELECT** 算法之前，咱们先再来简单回顾和分析下快速排序，ok，今天看到 Mark 的数据结构与算法分析-c 语言描述一书上对快速排序也有不错的介绍，所以为了增加点新鲜感，就不用自己以前的文章而改为直接引用 Mark 的叙述了：

As its name implies, quicksort is the fastest known sorting algorithm in practice. Its average running time is  $O(n \log n)$ （快速排序是实践中已知的最快的排序算法，他的平均运行时间为  $O(N \log N)$ ）。It is very fast, mainly due to a very tight and highly optimized inner loop. It has

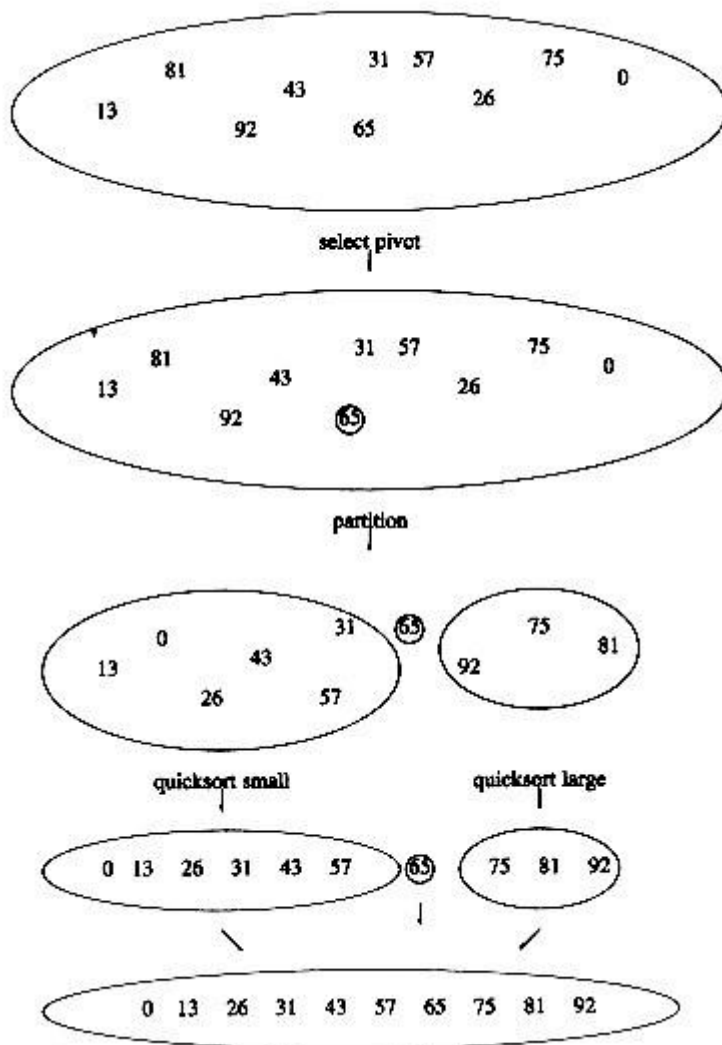
$O(n^2)$  worst-case performance (最坏情形的性能为  $O(N^2)$ ), but this can be made exponentially unlikely with a little effort.

The quicksort algorithm is simple to understand and prove correct, although for many years it had the reputation of being an algorithm that could in theory be highly optimized but in practice was impossible to code correctly (no doubt because of FORTRAN).

Like mergesort, quicksort is a divide-and-conquer recursive algorithm (像归并排序一样, 快速排序也是一种采取分治方法的递归算法). The basic algorithm to sort an array  $S$  consists of the following four easy steps (通过下面的 4 个步骤将数组  $S$  排序的算法如下):

1. If the number of elements in  $S$  is 0 or 1, then return (如果  $S$  中元素个数是 0 或 1, 则返回).
2. Pick any element  $v$  in  $S$ . This is called the pivot (取  $S$  中任一元素  $v$ , 作为枢纽元).
3. Partition  $S - \{v\}$  (the remaining elements in  $S$ ) into two disjoint groups (枢纽元  $v$  将  $S$  中其余的元素分成两个不相交的集合):  $S_1 = \{x \in S - \{v\} \mid x \leq v\}$ , and  $S_2 = \{x \in S - \{v\} \mid x > v\}$ .
4. Return { quicksort( $S_1$ ) followed by  $v$  followed by quicksort( $S_2$ )}.

下面依据上述步骤对序列 13,81,92,43,65,31,57,26,75,0 进行第一趟划分处理, 可得到如下图所示的过程:



## 1.2、选取枢纽元的几种方法

### 1、糟糕的方法

通常的做法是选择数组中第一个元素作为枢纽元，如果输入是随机的，那么这是可以接受的。但是，如果输入序列是预排序的或者是反序的，那么依据这样的枢纽元进行划分则会出现相当糟糕的情况，因为可能所有的元素不是被划入 S1，就是都被划入 S2 中。

### 2、较好的方法

一个比较好的做法是随机选取枢纽元，一般来说，这种策略是比较妥当的。

### 3、三数取中值方法

例如，输入序列为 8, 1, 4, 9, 6, 3, 5, 2, 7, 0，它的左边元素为 8，右边元素为 0，中间位置  $\lfloor \frac{\text{left} + \text{right}}{2} \rfloor$  上的元素为 6，于是枢纽元为 6。显然，使用三数中值分割法消除了预

排序输入的坏情形，并且减少了快速排序大约 5%（此为前人实验所得数据，无法具体证明）的运行时间。

### 1.3、划分过程

下面，我们再对序列 8, 1, 4, 9, 6, 3, 5, 2, 7, 0 进行第一趟划分，我们要达到的划分目的就是为把所有小于枢纽元（据三数取中分割法取元素 6 为枢纽元）的元素移到数组的左边，而把所有大于枢纽元的元素全部移到数组的右边。

此过程，如下述几个图所示：

8 1 4 9 0 3 5 2 7 6  
i j

8 1 4 9 0 3 5 2 7 6  
i j

After First Swap:  
-----  
2 1 4 9 0 3 5 8 7 6  
i j

Before Second Swap:  
-----  
2 1 4 9 0 3 5 8 7 6  
i j

After Second Swap:  
-----  
2 1 4 5 0 3 9 8 7 6  
i j

Before Third Swap  
-----  
2 1 4 5 0 3 9 8 7 6  
j i //i, j 在元素 3 处碰头之后，i++指向了 9，最后与 6 交换后，得到：



2 1 4 5 0 3 6 8 7 9  
          i      pivot

至此，第一趟划分过程结束，枢纽元 6 将整个序列划分成了左小右大两个部分。

#### 1.4、四个细节

下面，是 4 个值得你注意的细节问题：

1、我们要考虑一下，就是如何处理那些等于枢纽元的元素，问题在于当 i 遇到第一个等于枢纽元的关键字时，是否应该停止移动 i，或者当 j 遇到一个等于枢纽元的元素时是否应该停止移动 j。

答案是：如果 i, j 遇到等于枢纽元的元素，那么我们就让 i 和 j 都停止移动。

2、对于很小的数组，如数组的大小  $N \leq 20$  时，快速排序不如插入排序好。

3、只通过元素间进行比较达到排序目的的任何排序算法都需要进行  $O(N \log N)$  次比较，如快速排序算法（最坏  $O(N^2)$ ，最好  $O(N \log N)$ ），归并排序算法（最坏  $O(N \log N)$ ，不过归并排序的问题在于合并两个待排序的序列需要附加线性内存，在整个算法中，还要将数据拷贝到临时数组再拷贝回来这样一些额外的开销，放慢了归并排序的速度）等。

4、下面是实现三数取中的划分方法的程序：

//三数取中分割法

input\_type median3( input\_type a[], int left, int right )

//下面的快速排序算法实现之一，及通过三数取中分割法寻找最小的 k 个数的快速选择  
SELECT 算法都要调用这个 median3 函数

```
{
    int center;
    center = (left + right) / 2;

    if( a[left] > a[center] )
        swap( &a[left], &a[center] );
    if( a[left] > a[right] )
        swap( &a[left], &a[right] );
    if( a[center] > a[right] )
        swap( &a[center], &a[right] );

    /* invariant: a[left] <= a[center] <= a[right] */
    swap( &a[center], &a[right-1] );    /* hide pivot */
```

```

return a[right-1];          /* return pivot */
}

```

下面的程序是利用上面的三数取中分割法而运行的快速排序算法：

//快速排序的实现之一

```

void q_sort( input_type a[], int left, int right )
{
    int i, j;
    input_type pivot;
    if( left + CUTOFF <= right )
    {
        pivot = median3( a, left, right ); //调用上面的实现三数取中分割法的 median3 函数
        i=left; j=right-1; //第 8 句
        for(;;)
        {
            while( a[++i] < pivot );
            while( a[--j] > pivot );
            if( i < j )
                swap( &a[i], &a[j] );
            else
                break; //第 16 句
        }
        swap( &a[i], &a[right-1] ); /*restore pivot*/
        q_sort( a, left, i-1 );
        q_sort( a, i+1, right );
    }
}

```

//如上所见，在划分过程（[partition](#)）后，快速排序需要两次递归，一次对左边递归  
 //一次对右边递归。下面，你将看到，快速选择 [SELECT](#) 算法始终只对一边进行递归。  
 //这从直观上也能反应出：此快速排序算法（ $O(N \cdot \log N)$ ）明显会比  
 //下面第二节中的快速选择 [SELECT](#) 算法（ $O(N)$ ）平均花费更多的运行时间。

```

}
}

```

如果上面的第 8-16 句，改写成以下这样：

```

i=left+1; j=right-2;
for(;;)
{
    while( a[i] < pivot ) i++;
    while( a[j] > pivot ) j--;
    if( i < j )
        swap( &a[i], &a[j] );
    else
        break;
}

```

那么，当  $a[i] = a[j] = \text{pivot}$  则会产生无限，即死循环（相信，不用我多余解释，:D）。ok，接下来，咱们将进入正题--快速选择 **SELECT** 算法。

## 第二节、线性期望时间的快速选择 **SELECT** 算法

### 2.1、快速选择 **SELECT** 算法的介绍

Quicksort can be modified to solve the selection problem, which we have seen in chapters 1 and 6. Recall that by using a priority queue, we can find the  $k$ th largest (or smallest) element in  $O(n + k \log n)$  (以用最小堆初始化数组，然后取这个优先队列前  $k$  个值，复杂度  $O(n) + k \cdot O(\log n)$ )。实际上，最好采用最大堆寻找最小的  $k$  个数，那样，此时复杂度为  $n \cdot \log k$ 。更多详情，请参见：狂想曲系列[第三章、寻找最小的  \$k\$  个数](#)）。For the special case of finding the median, this gives an  $O(n \log n)$  algorithm.

Since we can sort the file in  $O(n \log n)$  time, one might expect to obtain a better time bound for selection. The algorithm we present to find the  $k$ th smallest element in a set  $S$  is almost identical to quicksort. In fact, the first three steps are the same. We will call this algorithm quickselect (叫做快速选择). Let  $|S_i|$  denote the number of elements in  $S_i$  (令  $|S_i|$  为  $S_i$  中元素的个数). The steps of quickselect are:

1. If  $|S| = 1$ , then  $k = 1$  and return the elements in  $S$  as the answer. If a cutoff for small files is being used and  $|S| \leq \text{CUTOFF}$ , then sort  $S$  and return the  $k$ th smallest element.

2. Pick a pivot element,  $v$  ( $v \in S$ ). (选取一个枢纽元  $v$  属于  $S$ )

3. Partition  $S - \{v\}$  into  $S_1$  and  $S_2$ , as was done with quicksort.

(将集合  $S - \{v\}$  分割成  $S_1$  和  $S_2$ ，就像我们在快速排序中所作的那样)

4. If  $k \leq |S1|$ , then the  $k$ th smallest element must be in  $S1$ . In this case, return `quickselect (S1, k)`. If  $k = 1 + |S1|$ , then the pivot is the  $k$ th smallest element and we can return it as the answer. Otherwise, the  $k$ th smallest element lies in  $S2$ , and it is the  $(k - |S1| - 1)$ st smallest element in  $S2$ . We make a recursive call and return `quickselect (S2, k - |S1| - 1)`.

（如果  $k \leq |S1|$ ，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)`）（下面几节的程序关于  $k$  的表述可能会有所出入，但无碍，抓住原理即 **ok**）。

In contrast to quicksort, quickselect makes only one recursive call instead of two. The worst case of quickselect is identical to that of quicksort and is  $O(n^2)$ . Intuitively, this is because quicksort's worst case is when one of  $S1$  and  $S2$  is empty; thus, quickselect（快速选择） is not really saving a recursive call. The average running time, however, is  $O(n)$ （不过，其平均运行时间为  $O(N)$ 。看到了没，就是平均复杂度为  $O(N)$  这句话）。The analysis is similar to quicksort's and is left as an exercise.

The implementation of quickselect is even simpler than the abstract description might imply. The code to do this shown in Figure 7.16. When the algorithm terminates, the  $k$ th smallest element is in position  $k$ . This destroys the original ordering; if this is not desirable, then a copy must be made.

## 2.2、三数中值分割法寻找第 $k$ 小的元素

第一节，已经介绍过此三数中值分割法，有个细节，你要注意，即数组元素索引是从“0...i”开始计数的，所以第  $k$  小的元素应该是返回 `a[i]=a[k-1]`.即  $k-1=i$ 。换句话说就是说，第  $k$  小元素，实际上应该在数组中对应下标为  $k-1$ 。ok，下面给出三数中值分割法寻找第  $k$  小的元素的程序的两个代码实现：

```
1. //代码实现一
2. //copyright@ mark allen weiss
3. //July、 updated, 2011.05.05 凌晨.
4.
5. //三数中值分割法寻找第 k 小的元素的快速选择 SELECT 算法
6. void q_select( input_type a[], int k, int left, int right )
7. {
8.     int i, j;
9.     input_type pivot;
10.    if( left /*+ CUTOFF*/ <= right ) //去掉 CUTOFF 常量，无用
11.    {
```

```

12.     pivot = median3( a, left, right );    //调用 1、4 节里的实现三数取中分割法
      的 median3 函数
13.     //取三数中值作为枢纽元，可以消除最坏情况而保证此算法是  $O(N)$  的。不过，这还
      只局限在理论意义上。
14.     //稍后，您将看到另一种选取枢纽元的方法。
15.
16.     i=left; j=right-1;
17.     for(;;)    //此句到下面的九行代码，即为快速排序中的 partition 过程的实现之
      一
18.     {
19.         while( a[++i] < pivot ){}
20.         while( a[--j] > pivot ){}
21.         if ( i < j )
22.             swap( &a[i], &a[j] );
23.         else
24.             break;
25.     }
26.     swap( &a[i], &a[right-1] ); /* restore pivot */
27.     if( k < i)
28.         q_select( a, k, left, i-1 );
29.     else
30.         if( k-1 > i )    //此条语句相当于: if(k>i+1)
31.             q-select( a, k, i+1, right );
32.         //1、希望你已经看到，通过上面的 if-else 语句表明，此快速选择 SELECT 算法
      始终只对数组的一边进行递归，
33.         //这也是其与第一节中的快速排序算法的本质性区别。
34.
35.         //2、这个区别则直接决定了：快速排序算法最快能达到  $O(N \log N)$ ，
36.         //而快速选择 SELECT 算法则最坏亦能达到  $O(N)$  的线性时间复杂度。
37.         //3、而确保快速选择算法最坏情况下能做到  $O(N)$  的根本保障在于枢纽元元素的
      选取，
38.         //即采取稍后的 2.3 节里的五分法中项的中项，或 2.4 节里的中位数的中外位数的
      枢纽元选择方法达到  $O(N)$  的目的。
39.         //后天老爸生日，孩儿深深祝福。July、updated, 2011.05.19。
40.     }
41.     else
42.         insert_sort(a, left, right-left+1 );
43. }
44.
45.
46. //代码实现二
47. //copyright @ 飞羽
48. //July、updated, 2011.05.11。
49. //三数中值分割法寻找第 k 小的元素

```

```

50. bool median_select(int array[], int left, int right, int k)
51. {
52.     //第 k 小元素，实际上应该在数组中下标为 k-1
53.     if (k-1 > right || k-1 < left)
54.         return false;
55.
56.     //三数中值作为枢纽元方法，关键代码就是下述六行：
57.     int midIndex=(left+right)/2;
58.     if(array[left]<array[midIndex])
59.         swap(array[left],array[midIndex]);
60.     if(array[right]<array[midIndex])
61.         swap(array[right],array[midIndex]);
62.     if(array[right]<array[left])
63.         swap(array[right],array[left]);
64.     swap(array[midIndex], array[right]);
65.
66.     int pos = partition(array, left, right);
67.
68.     if (pos == k-1)    //第 k 小元素，实际上应该在数组中下标为 k-1
69.         return true;
70.     else if (pos > k-1)
71.         return median_select(array, left, pos-1, k);
72.     else return median_select(array, pos+1, right, k);
73. }

```

上述程序使用三数中值作为枢纽元的方法可以使得最坏情况发生的概率几乎可以忽略不计。然而，稍后，您将看到：通过一种更好的方法，如“五分化中项的中项”，或“中位数的中位数”等方法选取枢纽元，我们将能彻底保证在最坏情况下依然是线性  $O(N)$  的复杂度。即，如稍后 2.3 节所示。

## 2.3、五分化中项的中项，确保 $O(N)$

The selection problem requires us to find the  $k$ th smallest element in a list  $S$  of  $n$  elements (要求我们找出含  $N$  个元素的表  $S$  中的第  $k$  个最小的元素). Of particular interest is the special case of finding the median. This occurs when  $k = \lceil n/2 \rceil$  (向上取整). (我们对找出中间元素的特殊情况有着特别的兴趣，这种情况发生在  $k = \lceil n/2 \rceil$  的时候)

In Chapters 1, 6, 7 we have seen several solutions to the selection problem. The solution in Chapter 7 uses a variation of quicksort and runs in  $O(n)$  average time (第 7 章中的解法，即本文上面第 1 节所述的思路 4，用到快速排序的变体并以平均时间  $O(N)$  运行). Indeed, it is described in Hoare's original paper on quicksort.

Although this algorithm runs in linear average time, it has a worst case of  $O(n^2)$  (但它有一个  $O(N^2)$  的最快情况). Selection can easily be solved in  $O(n \log n)$  worst-case time by sorting the elements, but for a long time it was unknown whether or not selection could be accomplished in  $O(n)$  worst-case time. The quickselect algorithm outlined in Section 7.7.6 is quite efficient in practice, so this was mostly a question of theoretical interest.

Recall that the basic algorithm is a simple recursive strategy. Assuming that  $n$  is larger than the cutoff point where elements are simply sorted, an element  $v$ , known as the pivot, is chosen. The remaining elements are placed into two sets,  $S_1$  and  $S_2$ .  $S_1$  contains elements that are guaranteed to be no larger than  $v$ , and  $S_2$  contains elements that are no smaller than  $v$ . Finally, if  $k \leq |S_1|$ , then the  $k$ th smallest element in  $S$  can be found by recursively computing the  $k$ th smallest element in  $S_1$ . If  $k = |S_1| + 1$ , then the pivot is the  $k$ th smallest element. Otherwise, the  $k$ th smallest element in  $S$  is the  $(k - |S_1| - 1)$ st smallest element in  $S_2$ . The main difference between this algorithm and quicksort is that there is only one subproblem to solve instead of two (这个快速选择算法与快速排序之间的主要区别在于, 这里求解的只有一个子问题, 而不是两个子问题)。

#### 定理 10.9

The running time of quickselect using median-of-median-of-five partitioning is  $O(n)$ .

The basic idea is still useful. Indeed, we will see that we can use it to improve the expected number of comparisons that quickselect makes. To get a good worst case, however, the key idea is to use one more level of indirection. Instead of finding the median from a sample of random elements, we will find the median from a sample of medians.

The basic pivot selection algorithm is as follows:

1. Arrange the  $n$  elements into  $\lfloor n/5 \rfloor$  groups of 5 elements, ignoring the (at most four) extra elements.
2. Find the median of each group. This gives a list  $M$  of  $\lfloor n/5 \rfloor$  medians.
3. Find the median of  $M$ . Return this as the pivot,  $v$ .

We will use the term **median-of-median-of-five partitioning** to describe the quickselect algorithm that uses the pivot selection rule given above. (我们将用术语“五分化中项的中项”来描述使用上面给出的枢纽元选择法的快速选择算法)。We will now show that median-of-median-of-five partitioning guarantees that each recursive subproblem is at most roughly 70 percent as large as the original (现在我们要证明, “五分化中项的中项”, 得保证每个递归子问题的大小最多为原问题的大约 70%)。We will also show that the pivot can be

computed quickly enough to guarantee an  $O(n)$  running time for the entire selection algorithm (我们还要证明, 对于整个选择算法, 枢纽元可以足够快的算出, 以确保  $O(N)$  的运行时间。看到了没, 这再次佐证了我们的类似快速排序的 partition 过程的分治方法为  $O(N)$  的观点) (更多详细的证明, 请参考: [第三章、寻找最小的 k 个数](#))。

## 2.4、中位数的中位数, $O(N)$ 的再次论证

以下内容来自算法导论第九章第 9.3 节全部内容 (最坏情况线性时间的选择), 如下 (我酌情对之参考原中文版做了翻译, 下文中括号内的中文解释, 为我个人添加):

### 9.3 Selection in worst-case linear time (最坏情况下线性时间的选择算法)

We now examine a selection algorithm whose running time is  $O(n)$  in the worst case (现在来看, 一个最坏情况运行时间为  $O(N)$  的选择算法 SELECT). Like RANDOMIZED-SELECT, the algorithm SELECT finds the desired element by recursively partitioning the input array. The idea behind the algorithm, however, is to *guarantee* a good split when the array is partitioned. SELECT uses the deterministic partitioning algorithm PARTITION from quicksort (see Section 7.1), modified to take the element to partition around as an input parameter (像 RANDOMIZED-SELECT 一样, SELECT 通过输入数组的递归划分来找出所求元素, 但是, 该算法的基本思想是要保证对数组的划分是个好的划分。SELECT 采用了取自快速排序的确定性划分算法 partition, 并做了修改, 把划分主元元素作为其参数)。

The SELECT algorithm determines the  $i$ th smallest of an input array of  $n > 1$  elements by executing the following steps. (If  $n = 1$ , then SELECT merely returns its only input value as the  $i$ th smallest.) (算法 SELECT 通过执行下列步骤来确定一个有  $n > 1$  个元素的输入数组中的第  $i$  小的元素。(如果  $n = 1$ , 则 SELECT 返回它的唯一输入数值作为第  $i$  个最小值。))

1. Divide the  $n$  elements of the input array into  $\lfloor n/5 \rfloor$  groups of 5 elements each and at most one group made up of the remaining  $n \bmod 5$  elements.
2. Find the median of each of the  $\lfloor n/5 \rfloor$  groups by first insertion sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
3. Use SELECT recursively to find the median  $x$  of the  $\lfloor n/5 \rfloor$  medians found in step 2. (If there are an even number of medians, then by our convention,  $x$  is the lower median.)
4. Partition the input array around the median-of-medians  $x$  using the modified version of PARTITION. Let  $k$  be one more than the number of elements on the low side of the partition, so that  $x$  is the  $k$ th smallest element and there are  $n-k$  elements on the high side of the partition. (利用修改过的 partition 过程, 按中位数的中位数  $x$  对输入数组进行划分, 让  $k$  比划低去的元素数目多 1, 所以,  $x$  是第  $k$  小的元素, 并且有  $n-k$  个元素在划分的高区)



5. If  $i = k$ , then return  $x$ . Otherwise, use SELECT recursively to find the  $i$ th smallest element on the low side if  $i < k$ , or the  $(i - k)$ th smallest element on the high side if  $i > k$ . (如果要找的第  $i$  小的元素等于程序返回的  $k$ , 即  $i=k$ , 则返回  $x$ 。否则, 如果  $i < k$ , 则在低区递归调用 SELECT 以找出第  $i$  小的元素, 如果  $i > k$ , 则在高区间找第  $(i-k)$  个最小元素)

1) 将输入数组的  $n$  个元素划分为  $\lfloor n/5 \rfloor$  组, 每组 5 个元素, 且至多只有一个组由剩下的  $n \bmod 5$  个元素组成。

2) 寻找  $\lfloor n/5 \rfloor$  个组中每一组的中位数。首先对每组中的元素(至多为 5 个)进行插入排序, 然后从排序过的序列中选出中位数。

3) 对第 2 步中找出的  $\lfloor n/5 \rfloor$  个中位数, 递归调用 SELECT 以找出其中位数  $x$ 。(如果有偶数个中位数, 根据约定,  $x$  是下中位数。)

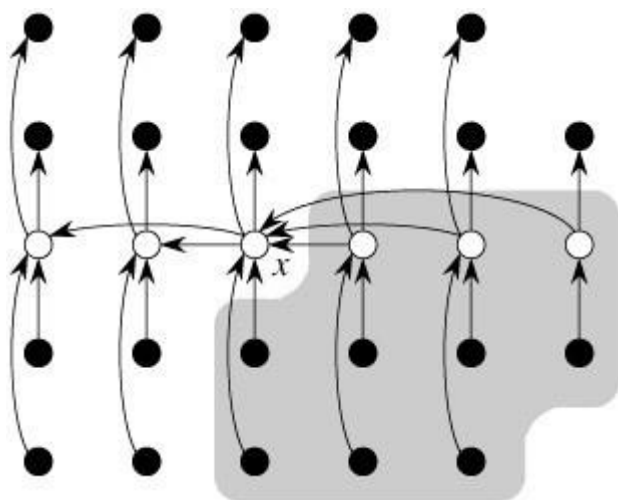
4) 利用修改过的 PARTITION 过程, 按中位数的中位数  $x$  对输入数组进行划分。让  $k$  比划分低区的元素数目多 1, 所以  $x$  是第  $k$  小的元素, 并且有  $n-k$  个元素在划分的高区。

5) 如果  $i=k$ , 则返回  $x$ 。否则, 如果  $i < k$ , 则在低区递归调用 SELECT 以找出第  $i$  小的元素, 如果  $i > k$ , 则在高区找第  $(i-k)$  个最小元素。

(以上五个步骤, 即本文上面的第四节末中所提到的所谓“五分化中项的中项”的方法。)

To analyze the running time of SELECT, we first determine a lower bound on the number of elements that are greater than the partitioning element  $x$ . (为了分析 SELECT 的运行时间, 先来确定大于划分主元元素  $x$  的元素数的一个下界) Figure 9.1 is helpful in visualizing this bookkeeping. At least half of the medians found in step 2 are greater than<sup>[1]</sup> the median-of-medians  $x$ . Thus, at least half of the  $\lfloor n/5 \rfloor$  groups contribute 3 elements that are greater than  $x$ , except for the one group that has fewer than 5 elements if 5 does not divide  $n$  exactly, and the one group containing  $x$  itself. Discounting these two groups, it follows that the number of elements greater than  $x$  is at least:

$$3 \left( \left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6.$$



(Figure 9.1: 对上图的解释或称对 SELECT 算法的分析:  $n$  个元素由小圆圈来表示, 并且每一个组占一纵列。组的中位数用白色表示, 而各中位数的中位数  $x$  也被标出。(当寻找偶数数目元素的中位数时, 使用下中位数)。箭头从比较大的元素指向较小的元素, 从中可以看出, 在  $x$  的右边, 每一个包含 5 个元素的组中都有 3 个元素大于  $x$ , 在  $x$  的左边, 每一个包含 5 个元素的组中有 3 个元素小于  $x$ 。大于  $x$  的元素以阴影背景表示。)

Similarly, the number of elements that are less than  $x$  is at least  $3n/10 - 6$ . Thus, in the worst case, SELECT is called recursively on at most  $7n/10 + 6$  elements in step 5.

We can now develop a recurrence for the worst-case running time  $T(n)$  of the algorithm SELECT. Steps 1, 2, and 4 take  $O(n)$  time. (Step 2 consists of  $O(n)$  calls of insertion sort on sets of size  $O(1)$ .) Step 3 takes time  $T(\lceil n/5 \rceil)$ , and step 5 takes time at most  $T(7n/10 + 6)$ , assuming that  $T$  is monotonically increasing. We make the assumption, which seems unmotivated at first, that any input of 140 or fewer elements requires  $O(1)$  time; the origin of the magic constant 140 will be clear shortly. We can therefore obtain the recurrence:

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \leq 140, \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n > 140. \end{cases}$$

We show that the running time is linear by substitution. More specifically, we will show that  $T(n) \leq cn$  for some suitably large constant  $c$  and all  $n > 0$ . We begin by assuming that  $T(n) \leq cn$  for some suitably large constant  $c$  and all  $n \leq 140$ ; this assumption holds if  $c$  is large enough. We also pick a constant  $a$  such that the function described by the  $O(n)$  term above (which describes the non-recursive component of the running time of the algorithm) is bounded above by  $an$  for all  $n > 0$ . Substituting this inductive hypothesis into the right-hand side of the recurrence yields

$$\begin{aligned} T(n) &\leq c \lceil n/5 \rceil + c(7n/10 + 6) \\ &\quad + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an), \end{aligned}$$

which is at most  $cn$  if

$$-cn/10 + 7c + an \leq 0.$$

Inequality (9.2) is equivalent to the inequality  $c \geq 10a(n/(n - 70))$  when  $n > 70$ . Because we assume that  $n \geq 140$ , we have  $n/(n - 70) \leq 2$ , and so choosing  $c \geq 20a$  will satisfy inequality (9.2). (Note that there is nothing special about the constant 140; we could replace it by any integer strictly greater than 70 and then choose  $c$  accordingly.) The worst-case running time of SELECT is therefore linear (因此, 此 SELECT 的最坏情况的运行时间是线性的)。

As in a comparison sort (see Section 8.1), SELECT and RANDOMIZED-SELECT determine

information about the relative order of elements only by comparing elements. Recall from Chapter 8 that sorting requires  $\Omega(n \lg n)$  time in the comparison model, even on average (see Problem 8-1). The linear-time sorting algorithms in Chapter 8 make assumptions about the input. In contrast, the linear-time selection algorithms in this chapter do not require any assumptions about the input. They are not subject to the  $\Omega(n \lg n)$  lower bound because they manage to solve the selection problem without sorting.

（与比较排序（算法导论 8.1 节）中的一样，SELECT 和 RANDOMIZED-SELECT 仅通过元素间的比较来确定它们之间的相对次序。在算法导论第 8 章中，我们知道在比较模型中，即使在平均情况下，排序仍然要  $O(n \lg n)$  的时间。第 8 章得线性时间排序算法在输入上做了假设。相反地，本节提到的此类似 partition 过程的 SELECT 算法不需要关于输入的任何假设，它们不受下界  $O(n \lg n)$  的约束，因为它们没有使用排序就解决了选择问题（看到了没，道出了此算法的本质阿））

Thus, the running time is linear because these algorithms do not sort; the linear-time behavior is not a result of assumptions about the input, as was the case for the sorting algorithms in Chapter 8. Sorting requires  $\Omega(n \lg n)$  time in the comparison model, even on average (see Problem 8-1), and thus the method of sorting and indexing presented in the introduction to this chapter is asymptotically inefficient.（所以，本节中的选择算法之所以具有线性运行时间，是因为这些算法没有进行排序；线性时间的结论并不需要在输入上所任何假设，即可得到。.....）

### 第三节、快速选择 SELECT 算法的实现

本节，咱们将依据下图所示的步骤，采取中位数的中位数选取枢纽元的方法来实现此

SELECT 算 法 ，

- 1) 将输入数组的  $n$  个元素划分为  $\lfloor n/5 \rfloor$  组，每组 5 个元素，且至多只有一个组由剩下的  $n \bmod 5$  个元素组成。
- 2) 寻找  $\lfloor n/5 \rfloor$  个组中每一组的中位数。首先对每组中的元素（至多为 5 个）进行插入排序，然后从排序过的序列中选出中位数。
- 3) 对第 2 步中找出的  $\lfloor n/5 \rfloor$  个中位数，递归调用 SELECT 以找出其中位数  $x$ 。（如果有偶数个中位数，根据约定， $x$  是下中位数。）
- 4) 利用修改过的 PARTITION 过程，按中位数的中位数  $x$  对输入数组进行划分。让  $k$  比划分低区的元素数目多 1，所以  $x$  是第  $k$  小的元素，并且有  $n-k$  个元素在划分的高区。
- 5) 如果  $i=k$ ，则返回  $x$ 。否则，如果  $i < k$ ，则在低区递归调用 SELECT 以找出第  $i$  小的元素，如果  $i > k$ ，则在高区找第  $(i-k)$  个最小元素。

不过，在实现之前，有个细节我还是必须要提醒你，即上文中 2.2 节开头处所述，“数组元素索引是从“0...i”开始计数的，所以第  $k$  小的元素应该是返回  $a[i]=a[k-1]$ 。即  $k-1=i$ 。换句话说就是，第  $k$  小元素，实际上应该在数组中对应下标为  $k-1$ ”这句话，我想，你应该明白了：

返回数组中第  $k$  小的元素，实际上就是返回数组中的元素  $array[i]$ ，即  $array[k-1]$ 。ok，最后请看此快速选择 **SELECT** 算法的完整代码实现（据我所知，在此之前，从没有人采取中位数的中位数选取枢纽元的方法来实现过这个 **SELECT** 算法）：

```

1. //copyright@ yansha && July && 飞羽
2. //July、updated, 2011.05.19.清晨。
3. //版权所有，引用必须注明出处：http://blog.csdn.net/v_JULY_v。
4. #include <iostream>
5. #include <time.h>
6. using namespace std;
7.
8. const int num_array = 13;
9. const int num_med_array = num_array / 5 + 1;
10. int array[num_array];
11. int midian_array[num_med_array];
12.
13. //冒泡排序（早些时候将修正为插入排序）
14. /*void insert_sort(int array[], int left, int loop_times, int compare_times)
15. {
16.     for (int i = 0; i < loop_times; i++)
17.     {
18.         for (int j = 0; j < compare_times - i; j++)
19.         {
20.             if (array[left + j] > array[left + j + 1])
21.                 swap(array[left + j], array[left + j + 1]);
22.         }
23.     }
24. }*/
25.
26. /*
27. //插入排序算法伪代码
28. INSERTION-SORT(A)                                cost    times
29. 1  for j ← 2 to length[A]                          c1      n
30. 2      do key ← A[j]                                c2      n - 1
31. 3          Insert A[j] into the sorted sequence A[1 .. j - 1].    0...n - 1
32. 4          i ← j - 1                                c4      n - 1
33. 5          while i > 0 and A[i] > key                  c5
34. 6              do A[i + 1] ← A[i]                    c6
35. 7              i ← i - 1                              c7
36. 8          A[i + 1] ← key                            c8      n - 1
37. */
38. //已修正为插入排序，如下：

```

```

39. void insert_sort(int array[], int left, int loop_times)
40. {
41.     for (int j = left; j < left+loop_times; j++)
42.     {
43.         int key = array[j];
44.         int i = j-1;
45.         while ( i>left && array[i]>key )
46.         {
47.             array[i+1] = array[i];
48.             i--;
49.         }
50.         array[i+1] = key;
51.     }
52. }
53.
54. int find_median(int array[], int left, int right)
55. {
56.     if (left == right)
57.         return array[left];
58.
59.     int index;
60.     for (index = left; index < right - 5; index += 5)
61.     {
62.         insert_sort(array, index, 4);
63.         int num = index - left;
64.         midian_array[num / 5] = array[index + 2];
65.     }
66.
67.     // 处理剩余元素
68.     int remain_num = right - index + 1;
69.     if (remain_num > 0)
70.     {
71.         insert_sort(array, index, remain_num - 1);
72.         int num = index - left;
73.         midian_array[num / 5] = array[index + remain_num / 2];
74.     }
75.
76.     int elem_aux_array = (right - left) / 5 - 1;
77.     if ((right - left) % 5 != 0)
78.         elem_aux_array++;
79.
80.     // 如果剩余一个元素返回，否则继续递归
81.     if (elem_aux_array == 0)
82.         return midian_array[0];

```

```

83.     else
84.         return find_median(median_array, 0, elem_aux_array);
85. }
86.
87. // 寻找中位数的所在位置
88. int find_index(int array[], int left, int right, int median)
89. {
90.     for (int i = left; i <= right; i++)
91.     {
92.         if (array[i] == median)
93.             return i;
94.     }
95.     return -1;
96. }
97.
98. int q_select(int array[], int left, int right, int k)
99. {
100.    // 寻找中位数的中位数
101.    int median = find_median(array, left, right);
102.
103.    // 将中位数的中位数与最右元素交换
104.    int index = find_index(array, left, right, median);
105.    swap(array[index], array[right]);
106.
107.    int pivot = array[right];
108.
109.    // 申请两个移动指针并初始化
110.    int i = left;
111.    int j = right - 1;
112.
113.    // 根据枢纽元素的值对数组进行一次划分
114.    while (true)
115.    {
116.        while(array[i] < pivot)
117.            i++;
118.        while(array[j] > pivot)
119.            j--;
120.        if (i < j)
121.            swap(array[i], array[j]);
122.        else
123.            break;
124.    }
125.    swap(array[i], array[right]);
126.

```

```

127.     /* 对三种情况进行处理: (m = i - left + 1)
128.     1、如果 m=k, 即返回的主元即为我们要找的第 k 小的元素, 那么直接返回主元 a[i]即可;
129.     2、如果 m>k, 那么接下来要到低区间 A[0...m-1]中寻找, 丢掉高区间;
130.     3、如果 m<k, 那么接下来要到高区间 A[m+1...n-1]中寻找, 丢掉低区间。
131.     */
132.     int m = i - left + 1;
133.     if (m == k)
134.         return array[i];
135.     else if(m > k)
136.         //上条语句相当于 if( (i-left+1) >k), 即 if( (i-left) > k-1 ), 于此就与
        2.2 节里的代码实现一、二相对应起来了。
137.         return q_select(array, left, i - 1, k);
138.     else
139.         return q_select(array, i + 1, right, k - m);
140. }
141.
142. int main()
143. {
144.     //srand(unsigned(time(NULL)));
145.     //for (int j = 0; j < num_array; j++)
146.     //array[j] = rand();
147.
148.     int array[num_array]={0,45,78,55,47,4,1,2,7,8,96,36,45};
149.     // 寻找第 k 最小数
150.     int k = 4;
151.     int i = q_select(array, 0, num_array - 1, k);
152.     cout << i << endl;
153.
154.     return 0;
155. }

```

## 十五、多项式乘法与快速傅里叶变换

**经典算法研究**系列，已经写到第十五章了，本章，咱们来介绍多项式的乘法以及快速傅里叶变换算法。本博客之前也已详细介绍过离散傅里叶变换（请参考：[十、从头到尾彻底理解傅里叶变换算法、上](#)，及[十、从头到尾彻底理解傅里叶变换算法、下](#)），这次咱们从多项式乘法开始，然后介绍 FFT 算法的原理与实现。同时，本文虽涉及到不少数学公式和定理（当然，我会尽量舍去一些与本文咱们要介绍的中心内容无关的定理或证明，只为保证能让读者易于接受或理解），但尽量保证通俗易懂，以让读者能看个明白。

有朋友建议，算法专一种，就 ok，没必要各个都学习。但个人实在抑制不住自己的兴趣，就是想写，当没法做到强迫自己不写时，这个经典算法研究系列便一直这么写下来了。

## 第一节、多项式乘法

我们知道，有两种表示多项式的方法，即系数表示法和点值表示法。什么是系数表示法？所谓的系数表示法，举个例子如下图所示， $A(x)=6x^3+7x^2-10x+9$ ， $B(x)=-2x^3+4x-5$ ，则  $C(x)=A(x)*B(x)$  就是普通的多项式相乘的算法，系数与系数相乘，这就是所谓的系数表示法。

$$\begin{array}{r}
 6x^3 + 7x^2 - 10x + 9 \\
 - 2x^3 \qquad \qquad + 4x - 5 \\
 \hline
 - 30x^3 - 35x^2 + 50x - 45 \\
 24x^4 + 28x^3 - 40x^2 + 36x \\
 - 12x^6 - 14x^5 + 20x^4 - 18x^3 \\
 \hline
 - 12x^6 - 14x^5 + 44x^4 - 20x^3 - 75x^2 + 86x - 45
 \end{array}$$

那么，何谓点值表示法？。一个次数为  $n$  次的多项式  $A(x)$  的点值表示就是  $n$  个点值所形成的集合： $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ 。其中所有  $x_k$  各不相同，且当  $k=0, 1, \dots, n-1$  时，有  $y(k)=A(x_k)$ 。

一个多项式可以由很多不同的点值表示，这是由于任意  $n$  个相异点  $x_0, x_1, \dots, x_{n-1}$  组成的集合，都可以看做是这种点值法的表示方法。对于一个用系数形式表示的多项式来说，在原则上计算其点值表示是简单易行的，我们只需要选取  $n$  个相异点  $x_0, x_1, \dots, x_{n-1}$ ，然后对  $k=0, 1, \dots, n-1$ ，求出  $A(x_k)$ ，然后根据霍纳法则，求出这  $n$  个点的值所需要的时间为  $O(n^2)$ 。

然，稍后，你将看到，如果巧妙的选取  $x_k$  的话，适当的利用点值表示可以使多项式的乘法可以在线性时间内完成。所以，如果我们能恰到好处的利用系数表示法与点值表示法的相互转化，那么我们可以加速多项式乘法（下面，将详细阐述这个过程），达到  $O(n \log n)$  的最佳时间复杂度。



前面说了，当用系数表示法，即用一般的算法表示多项式时，两个  $n$  次多项式的乘法需要用  $O(n^2)$  的时间才能完成。但采用点值表示法时，多项式乘法的运行时间仅为  $O(n)$ 。接下来，咱们要做的是，如何利用系数表示法与点值表示法的相互转化来实现多项式系数表示法的  $O(n \log n)$  的快速乘法。

## 第二节、多项式的快速乘法

在此之前，我们得明白两个概念，**求值与插值**。通俗的讲，所谓求值（系数 $\rightarrow$ 点值），是指系数形式的多项式转换为点值形式的表示方式。而插值（点值 $\rightarrow$ 系数）正好是求值的逆过程，即反过来，它是已知点值表示法，而要你转换其为多项式的系数表示法（用  $n$  个点值对也可以唯一确定一个不超过  $n-1$  次多项式，这个过程称之为插值）。而这个系数表示法与点值表示法的相互转化，即无论是从系数表示法转化到点值表示法所谓的求值，还是从点值表示法转化到系数表示法所谓的插值，求值和插值两种过程的时间复杂度都是  $O(n \log n)$ 。

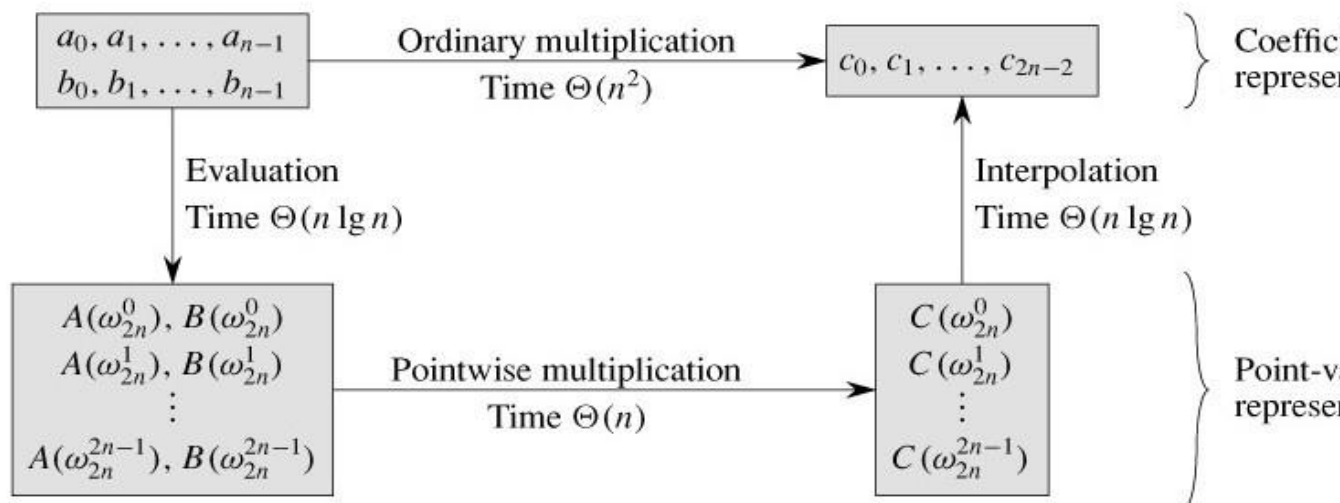
前面，我们已经说了，在多项式乘法中，如果用系数表示法表示多项式，那么多项式乘法的复杂度将为  $O(n^2)$ ，而用点值表示法表示的多项式，那么多项式的乘法的复杂度将是线性复杂度为  $O(n)$ ，即：适当的利用点值表示可以使多项式的乘法可以在线性时间内完成。此时读者可以发挥你的想象，**假设我们以下面这样一种过程来计算多项式的乘法**（不过在此之前，你得先把两个要相乘的多项式  $A(x)$  和  $B(x)$  扩充为次数或者说系数为  $2n$  次的多项式），**我们将会得到我们想要的结果：**

1. 系数表示法转化为点值表示法。先用系数表示法表示一个多项式，然后对这个多项式进行求值操作，即多项式从系数表示法变成了点值表示法，时间复杂度为  $O(n \log n)$ ；
2. 点值表示法计算多项式乘法。用点值表示法表示多项式后，计算多项式的乘法，线性复杂度为  $O(n)$ ；
3. 点值表示法转化为系数表示法。最终再次将点值表示法表示的多项式转变为系数表示法表示的多项式，这一过程，为  $O(n \log n)$ 。

那么，综上所述三项操作，系数表示法表示的多项式乘法总的时间复杂度已被我们降到了  $O(n \log n + n + n \log n) = O(N \log N)$ 。

如下图所示，从左至右，看过去，这个过程即是完成**多项式乘法的计算过程**。不过，完成这个过程有两种方法，一种就是前面第一节中所说的普通方法，即直接对系数表示法表示的多项式进行乘法运算，复杂度为  $O(n^2)$ ，它体现在下图中得 Ordinary multiplication 过程。还有一种就是本节上文处所述的三个步骤：1、将多项式由系数表示法转化为点值表

示法(点值过程)；2、利用点值表示法完成多项式乘法；3、将点值表示法再转化为系数表示法(插值过程)。三个步骤下来，总的时间复杂度为  $O(N \cdot \log N)$ 。它体现在下图中的 Evaluation + Pointwise multiplication + Interpolation 三个合过程。



由上图中，你也已经看到了。我们巧妙的利用了关于点值形式的多项式的线性时间乘法算法，来加快了系数形式表示的多项式乘法的运算速度。在此过程中，我们的工作最关键的就在于以  $O(n \cdot \log n)$  的时间快速把一个多项式从系数形式转换为点值形式（求值，我们即称之为 **FFT**），然后再以  $O(n \cdot \log n)$  的时间从点值形式转换为系数形式（插值，我们即称之为 **FFT 逆**）。最终达到了我们以最终的系数形式表示的多项式的快速乘法为  $O(N \cdot \log N)$  的时间复杂度。好不令人心生快意。

对上图，有一点必须说明，项  $\omega_{2n}$  是  $2n$  次单位复根。且不容忽视的是，在上述两种表示法即系数表示法和点值表示法相互转换的过程中，都使用了单位复根，才得以在  $O(n \cdot \log n)$  的时间内完成求值与插值运算（至于何谓单位复根，请参考相关资料。因为为了保证本文的通俗易懂性，无意引出一大堆公式或定理）。

### 第三节、FFT

注：本节有相当部分文字来自算法导论中文版第二版以及维基百科。

在具体介绍 **FFT** 之前，我们得熟悉知道折半定理是怎么回事儿：如果  $n > 0$  且  $n$  为偶数，那么， $n$  个  $n$  次单位复根的平方等于  $n/2$  个  $n/2$  个单位复根。我们已经知道，通过使用一种称为快速傅里叶变换（**FFT**）的方法，可以在  $O(n \cdot \log n)$  的时间内计算出  $DFT_n(a)$ ，而若采用直接的方法复杂度为  $O(n^2)$ 。**FFT** 就是利用了单位复根的特殊性质。

你将看到，FFT 方法运用的策略为分治策略，所谓分治，即分而治之。两个多项式  $A(x)$  与  $B(x)$  相乘的过程中，FFT 用  $A(x)$  中偶数下标的系数与奇数下标的系数，分别定义了两个新的次数界为  $n/2$  的多项式  $A[0](x)$  和  $A[1](x)$ ：

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1},$$

$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1}.$$

注意， $A[0]$  包含了  $A$  中所有偶数下标的系数（下标的相应二进制数的最后一位为 0），而  $A[1]$  包含  $A$  中所有奇数下标的系数（下标的相应二进制数的最后一位为 1）。有下式：

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2),$$

这样，求  $A(x)$  在  $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$  处的问题就转换为：

- 1) 求次数界为  $n/2$  的多项式  $A[0]$  与  $A[1]$  在点  $(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2$  的值，然后
- 2) 将上述结果进行组合。

下面，我们用 **N 次单位根**  $W_N$  来表示  $e^{-j\frac{2\pi}{N}}$ 。

$W_N$  的性质：

1. 周期性， $W_N$  具有周期  $N$ 。
2. 对称性：  $W_N^{k+\frac{N}{2}} = -W_N^k$ 。
3.  $W_N^{ikn} = W_{\frac{N}{i}}^{kn}$

为了简单起见，我们下面设待变换序列长度  $n = 2^r$ 。根据上面单位根的对称性，求级数

$$y_k = \sum_{n=0}^{N-1} W_N^{kn} x_n$$

时，可以将求和区间分为两部分：

$$\begin{aligned} y_k &= \sum_{n=2i} W_N^{kn} x_n + \sum_{n=2i+1} W_N^{kn} x_n \\ &= \sum_i W_{\frac{N}{2}}^{ki} x_{2i} + W_N^k \sum_i W_{\frac{N}{2}}^{ki} x_{2i+1} \\ &= F_{\text{even}}(k) + W_N^k F_{\text{odd}}(k) \quad (i \in \mathbb{Z}) \end{aligned}$$

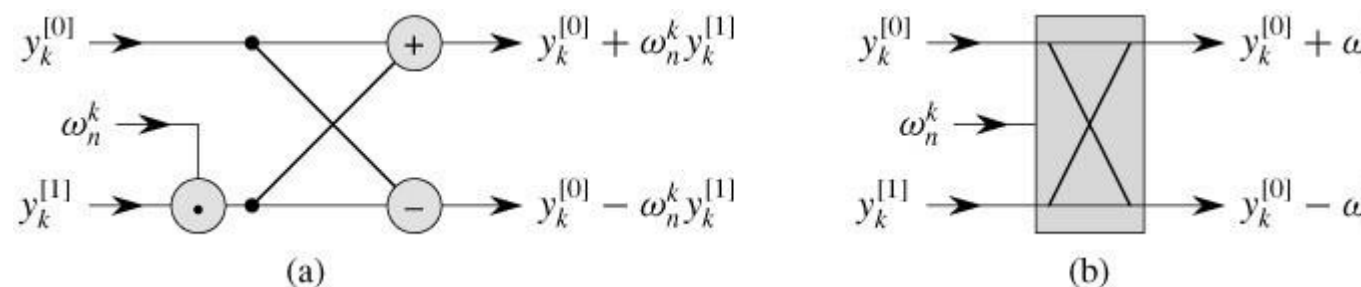
$F_{odd}(k)$  和  $F_{even}(k)$  是两个分别关于序列  $\{x_n\}_0^{N-1}$  奇数号和偶数号序列  $N/2$  点变换。由此式只能计算出  $y_k$  的前  $N/2$  个点，对于后  $N/2$  个点，注意  $F_{odd}(k)$  和  $F_{even}(k)$  都是周期为  $N/2$  的函数，由单位根的对称性，于是有以下变换公式：

- $y_{k+\frac{N}{2}} = F_{even}(k) - W_N^k F_{odd}(k)$
- $y_k = F_{even}(k) + W_N^k F_{odd}(k)$ 。

这样，一个  $N$  点变换就分解成了两个  $N/2$  点变换，照这样可继续分解下去。这就是**库利-图基快速傅里叶变换**算法的基本原理。此时，我们已经不难分析出此时算法的时间复杂度将为  $O(M\log N)$ 。

ok，没想到，本文之中还是出现了这么多的公式（没办法，搞这个 FFT 就是个纯数学活儿。之前买的一本小波与傅里叶分析基础也是如此，就是一本数学公式和定理的聚集书。不过，能看懂更好，实在无法弄懂也只权当做个稍稍了解）。

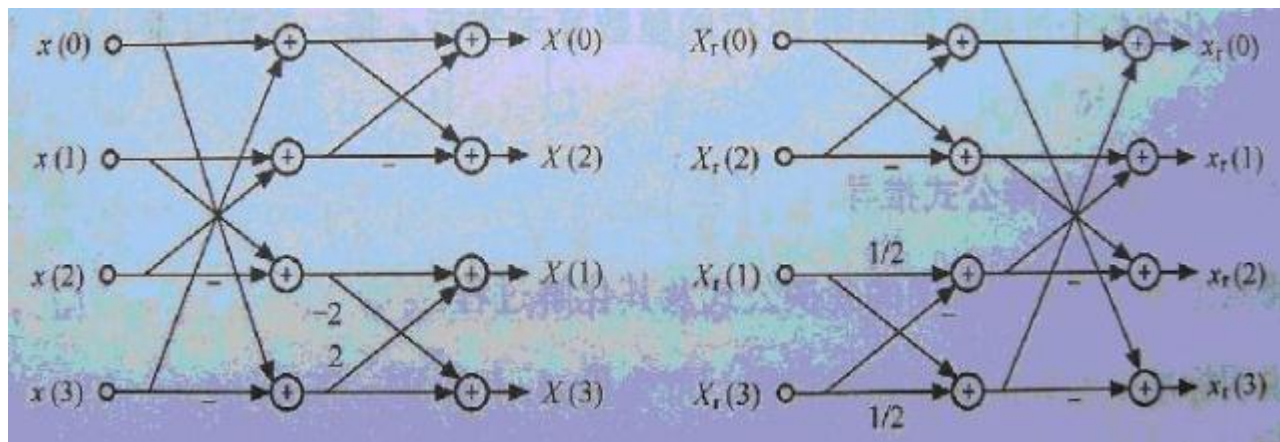
好了，下面，咱们来简单了解下 FFT 中的蝶形算法，本文将告结束。如下图所示：



有人这样解释**蝶形算法**：对于  $N$  ( $2$  的  $x$  次方) 点的离散信号，把它按索引位置分成两个序列，分别为  $0, 2, 4, \dots, 2K$  (记为  $A$ ) 和  $1, 3, 5, \dots, 2K-1$  (记为  $B$ )，由傅立叶变换可以推出  $N$  点的傅立叶变换前半部分的结果为  $A+B*$ 旋转因子，后半部分的结果为  $A-B*$ 旋转因子。于是求  $N$  点的傅立叶变换就变成分别求两个  $N/2$  点序列的傅立叶变换，对每一个  $N/2$  点的序列，递归前面的步骤，直到只有两点的序列，就只变成简单的加减关系了。把这些点的加减关系用线连接，看上去就是个蝶形。ok，更多可参考算法导论第 30 章。

举一个例子，我们知道， $4 \times 4$  的矩阵运算如果按常规算法的话仍要进行 64 次乘法运算和 48 次加法，这将耗费较多的时间，于是在 H.264 中，有一种改进的算法（蝶形算法）可

以减少运算的次数。这种矩阵运算算法构造非常巧妙，利用构造的矩阵的整数性质和对称性，可完全将乘法运算转化为加法运算。如下图所示：



下面的代码来自一本数字图像处理的书上的源代码：

```

1.     VOID WINAPI FFT(complex<double> * TD, complex<double> * FD, int r)
2.     {
3.         // 付立叶变换点数
4.         LONG count;
5.
6.         // 循环变量
7.         int i,j,k;
8.
9.         // 中间变量
10.        int bffsize,p;
11.
12.        // 角度
13.        double angle;
14.
15.        complex<double> *W,*X1,*X2,*X;
16.
17.        // 计算付立叶变换点数
18.        count = 1 << r;
19.
20.        // 分配运算所需存储器
21.        W = new complex<double>[count / 2];
22.        X1 = new complex<double>[count];
23.        X2 = new complex<double>[count];
24.
25.        // 计算加权系数
26.        for(i = 0; i < count / 2; i++)
27.        {

```

```

28.         angle = -i * PI * 2 / count;
29.         W[i] = complex<double> (cos(angle), sin(angle));
30.     }
31.
32.     // 将时域点写入 X1
33.     memcpy(X1, TD, sizeof(complex<double>) * count);
34.
35.     // 采用蝶形算法进行快速付立叶变换
36.     for(k = 0; k < r; k++)
37.     {
38.         for(j = 0; j < 1 << k; j++)
39.         {
40.             bffsize = 1 << (r-k);
41.             for(i = 0; i < bffsize / 2; i++)
42.             {
43.                 p = j * bffsize;
44.                 X2[i + p] = X1[i + p] + X1[i + p + bffsize / 2];
45.                 X2[i + p + bffsize / 2] = (X1[i + p] - X1[i + p + bffsize / 2]
    ) * W[i * (1<<k)];
46.             }
47.         }
48.         X = X1;
49.         X1 = X2;
50.         X2 = X;
51.     }
52.
53.     // 重新排序
54.     for(j = 0; j < count; j++)
55.     {
56.         p = 0;
57.         for(i = 0; i < r; i++)
58.         {
59.             if (j&(1<<i))
60.             {
61.                 p+=1<<(r-i-1);
62.             }
63.         }
64.         FD[j]=X1[p];
65.     }
66.
67.     // 释放内存
68.     delete W;
69.     delete X1;
70.     delete X2;

```

71. }

**updated:** 关于快速傅立叶变换(FFT)的 C++实现与 Matlab 实验, 这里有一篇不错的文章, 读者可以看看: <http://blog.csdn.net/rappy/article/details/1700829>。全系列, 完。2012.03.03。

---

## 后记

### 关于制作者一花明月暗 & [有鱼网](#) CEO 吴超

朋友花明月暗此前全权负责了此经典算法研究系列 V0.1 版(十三个经典算法)的制作, 另一朋友吴超则负责了 V0.2 版本(十五个经典算法)的更新, V0.2 版最后的目录+标签工作仍由花明月暗完成。

而在此之前, 就在昨天, 朋友吴超刚刚完成[程序员编程艺术](#) PDF 电子版本的制作, 同样也是 400 多页, 由此, 特别感谢这两位朋友的奉献, 希望我们可以帮助到更多的人。

### 联系作者

本经典算法研究系列文档中有任何一处错误, bug, 或漏洞, 读者朋友一经发现, 欢迎随时来信指导, 或 blog 内留言, 评论, 我将感激不尽。

我的联系方式如下:

邮箱: [zhoulei0907@yahoo.cn](mailto:zhoulei0907@yahoo.cn)

微博: <http://weibo.com/julyweibo>

Blog: [http://blog.csdn.net/v\\_JULY\\_v](http://blog.csdn.net/v_JULY_v)。

本结构之法算法之道 blog, 若无意外, 永久更新, 永久勘误, 谢谢。

*July*、2012 年 4 月 4 日, 于有鱼网公司。

---

版权所有, 侵权必究。严禁用于任何商业用途, 违者定究法律责任。