

# 程序员编程艺术第一~三十七章集锦

作者: July、编程艺术创作组

时间: 二零一一年四月~二零一三年十二月

出处: [http://blog.csdn.net/v\\_july\\_v](http://blog.csdn.net/v_july_v)

本 PDF 制作者: 吴新隆

## 前言

从 2011 年 4 月写下第一篇至今, 编程艺术系列已经写了 37 章, 也就是说详细阐述了 37 个编程问题, 在创作的过程当中, 得到了很多朋友的支持, 特别是博客上随时都会有朋友不断留言, 或提出改进建议, 或 show 出自己的思路、代码, 或指正 bug, 非常感激。

本系列越写到最后, 越会发觉无论是面试, 还是编程当中遇到的绝大部分问题, 都是有规律可循的, 而且可以不断优化, 这也是自己愿一直写下去的原因。再者, 能给每一年找工作的毕业生带去或多或少的参考, 给早已参加工作的人提供思维锻炼的机会, 何尝不是一种思考与编程的双重乐趣!

编程艺术的继续创作仍需要得到广大读者的更多支持, 最近, 正在 review 和优化编程艺术系列, 即在徐徐创作新的章节的同时, 不断回顾已写的 37 章, 希望能找出所有显而易见的 bug, 包括优化相关代码, 希望有更多的朋友可以随我一起加入 review 当中。

如你发现任何 bug、问题, 或有任何建议, 欢迎随时在博客上留言反馈, 或联系我, 我的微博: <http://weibo.com/julyweibo>, 异常感谢。

愿你享受旅途, 不断思考, 不断收获, have fun !

# 目录

程序员编程艺术第一~三十七章集锦.....	1
前言.....	1
目录.....	2
第一章、左旋转字符串.....	3
第二章、字符串是否包含问题.....	23
第三章、寻找最小的 k 个数.....	37
第三章续、Top K 算法问题的实现.....	84
十四、亦第三章再续：快速选择 SELECT 算法的深入分析与实现.....	125
第三章三续、求数组中给定下标区间内的第 K 小（大）元素.....	145
第四章、现场编写类似 strstr/strcpy/strpbrk 的函数.....	156
第五章、寻找和为定值的两个或多个数.....	173
第六章、亲和数问题-求解 500 万以内的亲和数.....	185
第七章、求连续子数组的最大和.....	191
第八章、从头至尾漫谈虚函数.....	199
第九章、闲话链表追赶问题.....	215
第十章、如何给 $10^7$ 个数据量的磁盘文件排序.....	225
第十一章：最长公共子序列(LCS)问题.....	259
第十二~十五章：中签概率，IP 访问次数，回文等问题（初稿）.....	271
第十六~第二十章：全排列，跳台阶，奇偶排序，第一个只出现一次等问题.....	285
第二十一~二十二章：出现次数超过一半的数字，最短摘要的生成.....	303
第二十三、四章：杨氏矩阵查找，倒排索引关键词 Hash 不重复编码实践.....	323
第二十五章：二分查找实现（Jon Bentley：90%程序员无法正确实现）.....	345
第二十六章：基于给定的文档生成倒排索引的编码与实践.....	349
第二十七章：不改变正负数之间相对顺序重新排列数组.时间 $O(N)$ ，空间 $O(1)$ .....	374
第二十八~二十九章：最大连续乘积子串、字符串编辑距离.....	382
第三十~三十一章：字符串转换成整数，带通配符的字符串匹配.....	403
第三十二~三十三章：最小操作数，木块砌墙问题.....	434
第三十四~三十五章：格子取数，完美洗牌算法.....	468
第三十六~三十七章、搜索智能提示 suggestion，附近地点搜索.....	496
后记.....	513

# 第一章、左旋转字符串

作者: July, yansha、caopengcs。

时间: 二零一一年四月十四日。

## 题目描述:

定义字符串的左旋转操作：把字符串前面的若干个字符移动到字符串的尾部，如把字符串 abcdef 左旋转 2 位得到字符串 cdefab。请实现字符串左旋转的函数，要求对长度为 n 的字符串操作的时间复杂度为 O(n)，空间复杂度为 O(1)。

## 思路一、暴力移位法

初看此题，咱们最先想到的笨方法可能就是一位一位移动，故咱们写一个函数叫做 leftshiftone(char \*s, int n) 完成左移动一位的功能

```
void leftshiftone(char *s, int n) {
    char t = s[0];      //保存第一个字符
    for (int i = 1; i < n; ++i) {
        s[i - 1] = s[i];
    }
    s[n - 1] = t;
}
```

如此，左移 m 位的话，可以如下实现：

```
void leftshift(char *s, int n, int m) {
    while (m--) {
        leftshiftone(s, n);
    }
}
```

## 思路二、指针翻转法

咱们先来看个例子，如下：abc defghi，若要让 abc 移动至最后的过程可以是：abc defghi->def abcghi->def ghiabc

如此，我们可定义俩指针， $p1$  指向  $ch[0]$ ， $p2$  指向  $ch[m]$ ；  
一下过程循环  $m$  次，交换  $p1$  和  $p2$  所指元素，然后  $p1++$ ， $p2++$ ；。

1. 第一步，交换 abc 和 def， $abc\ defghi \rightarrow def\ abcghi$
2. 第二步，交换 abc 和 ghi， $def\ abcghi \rightarrow def\ ghiabc$

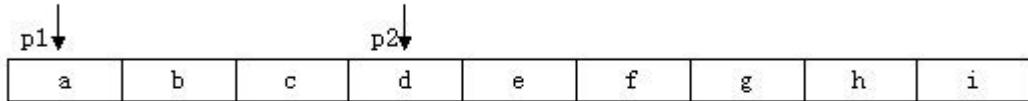
整个过程，看起来，就是 abc 一步一步 向后移动

- abc defghi
- def abcghi
- def ghi abc

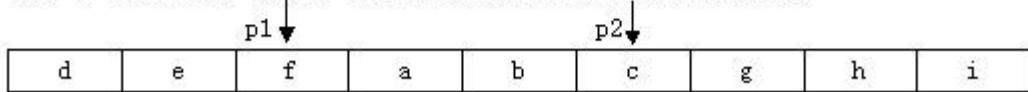
//最后的 复杂度是  $O(m+n)$

图解如下：

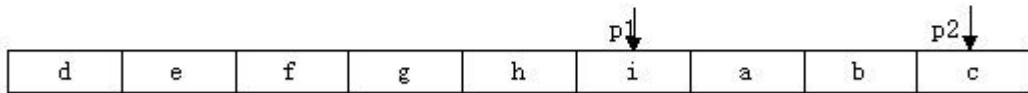
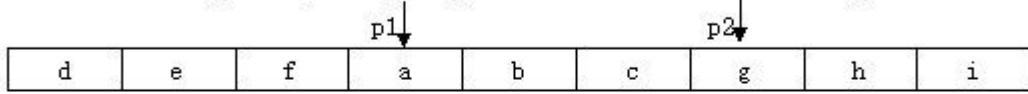
第一步：指针处于初始位置，如下所示：



第二步：交换 abc 和 def，指针  $p1$  和  $p2$  移动距离  $m$ ，如下所示：



第三步：首先， $p1++$ ， $p2++$ ， $p1$  和  $p2$  还是相距  $m$ ，然后交换 abc 和 ghi，如下所示：



至此，整个过程结束，得到最终结果  $defghi\ abc$ 。

由上述例子九个元素的序列 abcdefghi，您已经看到， $m=3$  时， $p_2$  恰好指到了数组最后一个元素，于是，上述思路没有问题。[但如果上面例子中 i 的后面还有元素列？](#)

即，如果是要左旋十个元素的序列：abcdefhij，ok，下面，就举这个例子，对 abcdefghij 序列进行左旋转操作：

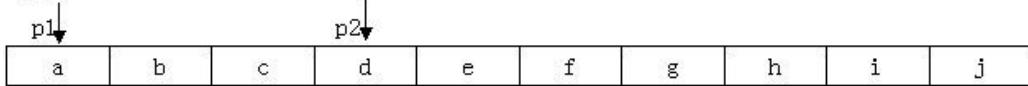
如果 abcdef ghij 要变成 defghij abc：

abcdef ghij

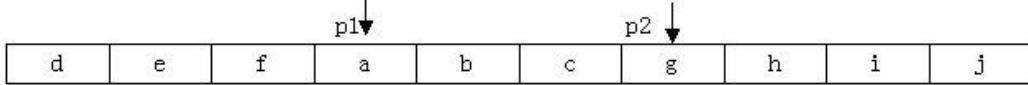
1. def abc ghij
2. def ghi abc j //接下来，j 步步前移
3. def ghi ab jc
4. def ghi a j bc
5. def ghi j abc

下面，再针对上述过程，画个图清晰说明下，如下所示：

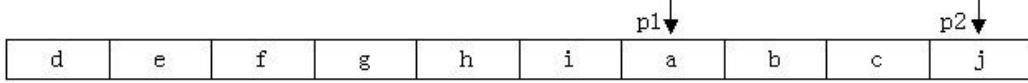
第一步：指针处于初始位置，其中 p1 指向首地址，p2 指向  $p1+m$ （本例  $m$  为 3），如下图所示：



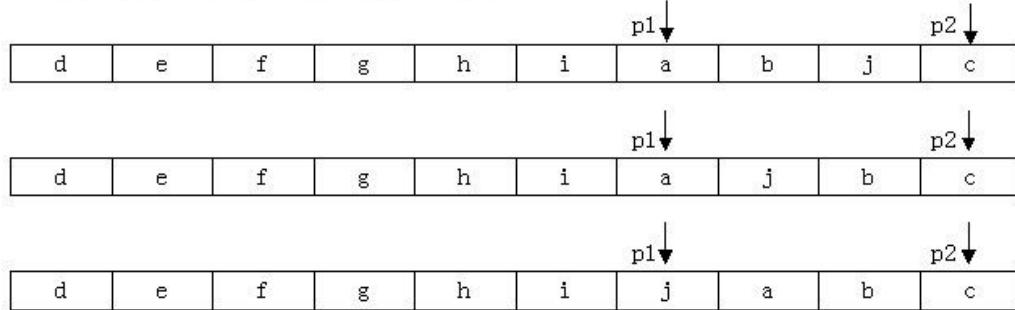
第二步：交换 p1 和 p2 所指元素，循环  $m$  次，abc 与 def 交换，然后  $p1++$ ,  $p2++$ ，如下图所示：



第三步：重复第二步，abc 与 ghi 交换，然后  $p1++$ , 在 a,  $p2++$ , 在 j:



第四步：如果  $p2+m-1$  不越界，说明 p2 到数组末尾之间所包含的元素为  $m$ ，即为上例（指 abcdefghi，九个元素）讨论的情况。否则，说明 p2 到数组末尾之间所包含的元素小于  $m$ ，保持 p1, p2 不变，将这些元素向左移动  $m$  个单元即可，如下图所示（下图的情况，就是 j 前移  $m$  个单位，移到 abc 的前面去，就 ok 了）：



至此，整个过程结束，得到最终结果。

ok，咱们来好好彻底总结一下此思路二：（就 4 点，请仔细阅读）：

- 1、首先让  $p1=ch[0]$ ,  $p2=ch[m]$ ，即让 p1, p2 相隔  $m$  的距离；
- 2、判断  $p2+m-1$  是否越界，如果没有越界转到 3，否则转到 4（abcdefg 这 8 个字母的字符串，以 4 左旋，那么初始时 p2 指向 e,  $p2+4$  越界了，但事实上 p2 至  $p2+m-1$  是  $m$  个字符，可以再做一个交换）。
- 3、不断交换  $*p1$  与  $*p2$ ，然后  $p1++$ ,  $p2++$ ，循环  $m$  次，然后转到 2。
- 4、此时  $p2+m-1$  已经越界，在此只需处理尾巴。过程如下：
  4. 1 通过  $n-p2$  得到  $p2$  与尾部之间元素个数  $r$ ，即我们要前移的元素个数。
  4. 2 以下过程执行  $r$  次:  $ch[p2] \leftarrow ch[p2-1]$ ,  $ch[p2-1] \leftarrow ch[p2-2]$ , ...,  $ch[p1+1] \leftarrow ch[p1]$ ;  $p1++$ ;  $p2++$ ;

所以，之前最初的那个左旋转九个元素 abcdefghi 的思路在末尾会出现问题的（如果 p2 后面有元素就不能这么变，例如，如果是处理十个元素，abcdefg hij 列？对的，就是这个意思），解决办法有两个：

方法一（即如上述思路总结所述）：

```
def ghi abc jk
```

当 p1 指向 a, p2 指向 j 时，由于  $p2+m$  越界，那么此时 p1, p2 不要变。这里 p1 之后 (abcjk) 就是尾巴，处理尾巴只需将 j, k 移到 abc 之前，得到最终序列，代码编写如下：

```
//copyright@July、颜沙
//最终代码，July, updated again, 2011.04.17.
#include <iostream>
#include <string>
using namespace std;

void rotate(string &str, int m)
{
    if (str.length() == 0 || m <= 0)
        return;

    int n = str.length();

    if (m % n <= 0)
        return;

    int p1 = 0, p2 = m;
    int k = (n - m) - n % m;

    // 交换 p1, p2 指向的元素，然后移动 p1, p2
    while (k--)
    {
        swap(str[p1], str[p2]);
        p1++;
        p2++;
    }

    // 重点，都在下述几行。
    // 处理尾部，r 为尾部左移次数
    int r = n - p2;
    while (r--)
    {
        int i = p2;
```

```

while (i > p1)
{
    swap(str[i], str[i-1]);
    i--;
}
p2++;
p1++;
}

//比如一个例子, abcdefghijk
//          p1   p2
//当执行到这里时, defghi a b c j k
//p2+m 出界了,
//r=n-p2=2, 所以下过程, 要执行循环俩次。

//第一次: j 步步前移, abcjk->abjck->ajbck->jabck
//然后, p1++, p2++, p1 指 a, p2 指 k。
//          p1   p2
//第二次: defghi j a b c k
//同理, 此后, k 步步前移, abck->abkc->akbc->kabc。
}

int main()
{
    string ch="abcdefghijkl";
    rotate(ch,3);
    cout<<ch<<endl;
    return 0;
}

```

## 方法二:

def ghi abc jk

当 p1 指向 a, p2 指向 j 时, 那么交换 p1 和 p2,

此时为:

def ghi jbc ak

p1++, p2++, p1 指向 b, p2 指向 k, 继续上面步骤得:

def ghi jkc ab

p1++, p2 不动, p1 指向 c, p2 指向 b, p1 和 p2 之间 (cab) 也就是尾巴,

那么处理尾巴 (cab) 需要循环左移一定次数 (而后的具体操作步骤已在下述程序的注释中已详细给出)。

根据方案二, 不难写出下述代码 (已测试正确) :

```

#include <iostream>
#include <string>

```

```

using namespace std;

//颜沙, 思路二之方案二,
//July、updated, 2011.04.16。
void rotate(string &str, int m)
{
    if (str.length() == 0 || m < 0)
        return;

    //初始化 p1, p2
    int p1 = 0, p2 = m;
    int n = str.length();

    // 处理 m 大于 n
    if (m % n == 0)
        return;

    // 循环直至 p2 到达字符串末尾
    while(true)
    {
        swap(str[p1], str[p2]);
        p1++;
        if (p2 < n - 1)
            p2++;
        else
            break;
    }

    // 处理尾部, r 为尾部循环左移次数
    int r = m - n % m; // r = 1.
    while (r--) //外循环执行一次
    {
        int i = p1;
        char temp = str[p1];
        while (i < p2) //内循环执行俩次
        {
            str[i] = str[i+1];
            i++;
        }
        str[p2] = temp;
    }

    //举一个例子
    //abcdefghijkl
    //当执行到这里的时候, defghijkl
}

```

```

//      p1      p2
//defghi a b c j k, a 与 j 交换, jbcak, 然后, p1++, p2++
//      p1      p2
//      j b c a k, b 与 k 交换, jkcab, 然后, p1++, p2 不动,
//r = m - n % m= 3-11%3=1, 即循环移位1次。
//      p1      p2
//      j k c a b
//p1 所指元素 c 实现保存在 temp 里,
//然后执行此条语句: str[i] = str[i+1]; 即 a 跑到 c 的位置处, a_b_
//i++, 再次执行: str[i] = str[i+1], ab_
//最后, 保存好的 c 填入, 为 abc, 所以, 最终序列为 defghi jk abc。
//July、updated, 2011.04.17 晚, 送走了她。
}

int main()
{
    string ch="abcdefghijkl";
    rotate(ch,3);
    cout<<ch<<endl;
    return 0;
}

```

**注意：**上文中都是假设  $m < n$ ，且如果鲁棒点的话令  $m=m \% n$ ，这样  $m$  允许大于  $n$ 。另外，各位要记得处理指针为空的情况。

还可以看下这段代码：

```

/*
 * myinvert2.cpp
 *
 * Created on: 2011-5-11
 *      Author: BigPotato
 */

#include<iostream>
#include<string>
#define positiveMod(m,n) ((m) % (n) + (n)) % (n)

/*
 *左旋字符串 str, m 为负数时表示右旋 abs (m) 个字母
 */
void rotate(std::string &str, int m) {
    if (str.length() == 0)
        return;

```

```

int n = str.length();
//处理大于 str 长度及 m 为负数的情况,positiveMod 可以取得 m 为负数时对 n 取余得到正数
m = positiveMod(m,n);
if (m == 0)
    return;
//    if (m % n <= 0)
//        return;
int p1 = 0, p2 = m;
int round;
//p2 当前所指和之后的 m-1 个字母共 m 个字母, 就可以和 p2 前面的 m 个字母交换。
while (p2 + m - 1 < n) {
    round = m;
    while (round--) {
        std::swap(str[p1], str[p2]);
        p1++;
        p2++;
    }
}
//剩下的不足 m 个字母逐个交换
int r = n - p2;
while (r--) {
    int i = p2;
    while (i > p1) {
        std::swap(str[i], str[i - 1]);
        i--;
    }
    p2++;
    p1++;
}
}

//测试
int main(int argc, char **argv) {
//    std::cout << ((-15) % 7 + 7) % 7 << std::endl;
//    std::cout << (-15) % 7 << std::endl;
std::string ch = "abcdefg";
int len = ch.length();
for (int m = -2 * len; m <= len * 2; m++) {
    //由于传给 rotate 的是 string 的引用, 所以这里每次调用都用了一个新的字符串
    std::string s = "abcdefg";
    rotate(s, m);
    std::cout << positiveMod(m,len) << ":" << s << std::endl;
}
}

```

```
    return 0;  
}
```

### 思路三、递归转换法

本文最初发布时，网友留言 bluesmic 说：楼主，谢谢你提出的研讨主题，很有学术和实践价值。关于思路二，本人提一个建议：思路二的代码，如果用递归的思想去简化，无论代码还是逻辑都会更加简单明了。

就是说，把一个规模为 N 的问题化解为规模为 M( $M < N$ ) 的问题。

举例来说，设字符串总长度为 L，左侧要旋转的部分长度为 s1，那么当从左向右循环交换长度为 s1 的小段，直到最后，由于剩余的部分长度为 s2 ( $s2 == L \% s1$ ) 而不能直接交换。

该问题可以递归转化成规模为  $s1 + s2$  的，方向相反（从右向左）的同一个问题。随着递归的进行，左右反复回荡，直到某一次满足条件  $L \% s1 == 0$  而交换结束。

举例解释一下：

设原始问题为：将“123abcdefg”左旋转为“abcdefg123”，即总长度为 10，旋转部（“123”）长度为 3 的左旋转。按照思路二的运算，演变过程为

“123abcdefg” $\rightarrow$ “abc123defg” $\rightarrow$ “abcdef123g”。这时，“123”无法和“g”作对调，该问题递归转化为：将“123g”右旋转为“g123”，即总长度为 4，旋转部（“g”）长度为 1 的右旋转。

**updated:**

Ys:

Bluesmic 的思路没有问题，他的思路以前很少有人提出。思路是通过递归将问题规模变小。当字符串总长度为 n，左侧要旋转的部分长度为 m，那么当从左向右循环交换长度为 m 的小段直到剩余部分为  $m' (n \% m)$ ，此时  $m' < m$ ，已不能直接交换了。

此后，我们换一个思路，把该问题递归转化成规模大小为  $m' + m$ ，方向相反的同一问题。随着递归的进行，直到满足结束条件  $n \% m == 0$ 。

举个具体事例说明，如下：

1、对于字符串 abc def ghi gk，

将 abc 右移到 def ghi gk 后面，此时  $n = 11$ ,  $m = 3$ ,  
 $m' = n \% m = 2$ ;

abc def ghi gk -> def ghi abc gk

2、问题变成 gk 左移到 abc 前面，此时  $n = m' + m = 5$ ,  $m = 2$ ,  
 $m' = n \% m = 1$ ;

abc gk -> a gk bc

3、问题变成 a 右移到 gk 后面，此时  $n = m' + m = 3$ ,  $m = 1$ ,  
 $m' = n \% m = 0$ ;

a gk bc -> gk a bc。由于此刻,  $n \% m = 0$ , 满足结束条件, 返回结果。

即从左至右，后从右至左，再从左至右，如此反反复复，直到满足条件，返回退出。

代码如下，已测试正确（有待优化）：

```
//递归,
//感谢网友 Bluesmic 提供的思路

//copyright@ yansha 2011.04.19
//July, updated, 2011.04.20.
#include <iostream>
using namespace std;

void rotate(string &str, int n, int m, int head, int tail, bool flag)
{
    //n 待处理部分的字符串长度, m: 待处理部分的旋转长度
    //head: 待处理部分的头指针, tail: 待处理部分的尾指针
    //flag = true 进行左旋, flag = false 进行右旋

    // 返回条件
    if (head == tail || m <= 0)
        return;

    if (flag == true)
    {
        int p1 = head;
        int p2 = head + m; //初始化 p1, p2

        //1、左旋: 对于字符串 abc def ghi gk,
        //将 abc 右移到 def ghi gk 后面, 此时 n = 11, m = 3, m' = n \% m = 2;
        //abc def ghi gk -> def ghi abc gk
    }
}
```

```

// (相信, 经过上文中那么多繁杂的叙述, 此类的转换过程, 你应该是了如指掌了。)

int k = (n - m) - n % m; //p1, p2 移动距离, 向右移六步

/*
解释下上面的 k = (n - m) - n % m 的由来:
yansha:
以 p2 为移动的参照系:
n-m 是开始时 p2 到末尾的长度, n%m 是尾巴长度
(n-m)-n%m 就是 p2 移动的距离
比如 abc def efg hi
开始时 p2->d, 那么 n-m 为 def efg hi 的长度 8,
n%m 为尾巴 hi 的长度 2,
因为我知道 abc 要移动到 hi 的前面, 所以移动长度是
(n-m)-n%m = 8-2 = 6。
*/
for (int i = 0; i < k; i++, p1++, p2++)
    swap(str[p1], str[p2]);

rotate(str, n - k, n % m, p1, tail, false); //flag 标志变为 false, 结束左旋,
下面, 进入右旋
}

else
{
    //2、右旋: 问题变成 gk 左移到 abc 前面, 此时 n = m' + m = 5, m = 2, m' = n % m - 1;
    //abc gk -> a gk bc

    int p1 = tail;
    int p2 = tail - m;

    // p1, p2 移动距离, 向左移俩步
    int k = (n - m) - n % m;

    for (int i = 0; i < k; i++, p1--, p2--)
        swap(str[p1], str[p2]);

    rotate(str, n - k, n % m, head, p1, true); //再次进入上面的左旋部分,
    //3、左旋: 问题变成 a 右移到 gk 后面, 此时 n = m' + m = 3, m = 1, m' = n % m = 0;
    //a gk bc-> gk a bc。 由于此刻, n % m = 0, 满足结束条件, 返回结果。
}
}

```

```

int main()
{
    int i=3;
    string str = "abcdefghijkl";
    int len = str.length();
    rotate(str, len, i % len, 0, len - 1, true);
    cout << str.c_str() << endl; //转化成字符数组的形式输出
    return 0;
}

```

非常感谢。

稍后，由下文，您将看到，其实上述思路二的本质即是下文将要阐述的 stl **rotate 算法**，详情，请继续往下阅读。

## 思路四、循环移位法

下面，我将再具体深入阐述下此 STL 里的 rotate 算法，由于 stl 里的 rotate 算法，用到了 gcd 的原理，下面，我将先介绍辗转相除法(又称欧几里得算法、gcd 算法) 的算法思路及原理。

**gcd**，即辗转相除法，又称欧几里得算法，是求最大公约数的算法，即求两个正整数之最大公因子的算法。此算法作为 TAOCP 第一个算法被阐述，足见此算法被重视的程度。

**gcd 算法：**给定俩个正整数  $m, n (m>=n)$ ，求它们的最大公约数。（注意，一般要求  $m>=n$ ，若  $m<n$ ，则要先交换  $m<->n$ 。下文，会具体解释）。

用数学定理表示即为：“定理： $\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$  ( $a>b$  且  $a \bmod b$  不为 0)”。以下，是此算法的具体流程：

- 1、[求余数]，令  $r=m \% n$ ， $r$  为  $n$  除  $m$  所得余数 ( $0<=r<n$ )；
- 2、[余数为 0?]，若  $r=0$ ，算法结束，此刻， $n$  即为所求答案，否则，继续，转到 3；
- 3、[重置]，置  $m<-n$ ， $n<-r$ ，返回步骤 1.

此算法的证明，可参考计算机程序设计艺术第一卷：基本算法。证明，此处略。

ok，下面，举一个例子，你可能看的更明朗点。

比如，给定  $m=544$ ,  $n=119$ ,

则余数  $r=m \% n = 544 \% 119 = 68$ ; 因  $r \neq 0$ , 所以跳过上述步骤 2, 执行步骤 3。;

置  $m < -119$ ,  $n < -68$ ,  $\Rightarrow r = m \% n = 119 \% 68 = 51$ ;

置  $m < -68$ ,  $n < -51$ ,  $\Rightarrow r = m \% n = 68 \% 51 = 17$ ;

置  $m < -51$ ,  $n < -17$ ,  $\Rightarrow r = m \% n = 51 \% 17 = 0$ , 算法结束,

此时的  $n=17$ , 即为  $m=544$ ,  $n=119$  所求的俩个数的最大公约数。

再解释下上述  $\text{gcd}(m, n)$  算法开头处的，要求  $m >= n$  的原因：举这样一个例子，如  $m < n$ , 即  $m=119$ ,  $n=544$  的话，那么  $r=m \% n = 119 \% 544 = 119$ ,

因为  $r \neq 0$ , 所以执行上述步骤 3, 注意，看清楚了： $m < -544$ ,  $n < -119$ 。看到了没，尽管刚开始给的  $m < n$ , 但最终执行  $\text{gcd}$  算法时，还是会把  $m$ ,  $n$  的值交换过来，以保证  $m >= n$ 。

ok，我想，现在，你已经彻底明白了此  $\text{gcd}$  算法，下面，咱们进入主题，stl 里的 `rotate` 算法的具体实现。//待续。

熟悉 stl 里的 `rotate` 算法的人知道，对长度为  $n$  的数组  $(ab)$  左移  $m$  位，可以用 stl 的 `rotate` 函数（stl 针对三种不同的迭代器，提供了三个版本的 `rotate`）。但在某些情况下，用 stl 的 `rotate` 效率极差。

对数组循环移位，可以采用的方法有（[也算是对上文思路一，和思路二的总结](#)）：

flyinghearts:

- ① 动态分配一个同样长度的数组，将数据复制到该数组并改变次序，再复制回原数组。（最普通的方法）
- ② 利用  $ba = (br)^T (ar)^T = (arbr)^T$ , 通过三次反转字符串。（即上述思路一，首先对序列前部分逆序，再对序列后部分逆序，再对整个序列全部逆序）
- ③ 分组交换（尽可能使数组的前面连续几个数为所要结果）：

若  $a$  长度大于  $b$ , 将  $ab$  分成  $a_0a_1b$ , 交换  $a_0$  和  $b$ , 得  $ba_1a_0$ , 只需再交换  $a_1$  和  $a_0$ 。

若  $a$  长度小于  $b$ , 将  $ab$  分成  $ab_0b_1$ , 交换  $a$  和  $b_0$ , 得  $b_0ab_1$ , 只需再交换  $a$  和  $b_0$ 。

通过不断将数组划分, 和交换, 直到不能再划分为止。分组过程与求最大公约数很相似。

④ 所有序号为  $(j+i * m) \% n$  ( $j$  表示每个循环链起始位置,  $i$  为计数变量,  $m$  表示左旋转位数,  $n$  表示字符串长度), 会构成一个循环链(共有  $\text{gcd}(n, m)$  个,  $\text{gcd}$  为  $n$ 、 $m$  的最大公约数), 每个循环链上的元素只要移动一个位置即可, 最后整个过程总共交换了  $n$  次 (每一次循环链, 是交换  $n/\text{gcd}(n, m)$  次, 总共  $\text{gcd}(n, m)$  个循环链。所以, 总共交换  $n$  次)。

stl 的 rotate 的三种迭代器, 即是, 分别采用了后三种方法。

在给出 stl rotate 的源码之前, 先来看下我的朋友 ys 对上述第 4 种方法的评论:

ys: 这条思路个人认为绝妙, 也正好说明了数学对算法的重要影响。

通过前面思路的阐述, 我们知道对于循环移位, 最重要的是指针所指单元不能重复。例如要使  $abcd$  循环移位变成  $dabc$  (这里  $m=3, n=4$ ), 经过以下一系列眼花缭乱的赋值过程就可以实现:  $ch[0] \rightarrow \text{temp}$ ,  $ch[3] \rightarrow ch[0]$ ,  $ch[2] \rightarrow ch[3]$ ,  $ch[1] \rightarrow ch[2]$ ,  $\text{temp} \rightarrow ch[1]$ ; (\*)

字符串变化为:  $abcd \rightarrow bcd \rightarrow dbc \rightarrow db_c \rightarrow d_bc \rightarrow dabc$ ;  
是不是很神奇? 其实这是有规律可循的。

请先看下面的说明再回过头来看。

对于左旋转字符串, 我们知道每个单元都需要且只需要赋值一次, 什么样的序列能保证每个单元都只赋值一次呢?

1、对于正整数  $m$ 、 $n$  互为质数的情况, 通过以下过程得到序列的满足上面的要求:

```
for i = 0: n-1
```

```
k = i * m % n;  
end
```

举个例子来说明一下，例如对于  $m=3, n=4$  的情况，

1、我们得到的序列：即通过上述式子求出来的  $k$  序列，是 0, 3, 2, 1。

2、然后，你只要只需按这个顺序赋值一遍就达到左旋 3 的目的了：

```
ch[0]->temp, ch[3]->ch[0], ch[2]->ch[3], ch[1]->ch[2],  
temp->ch[1];      (*)
```

ok，这是不是就是按上面 (\*) 式子的顺序所依次赋值的序列阿？哈哈，很巧妙吧。当然，以上只是特例，作为一个循环链，相当于 rotate 算法的一次内循环。

2、对于正整数  $m, n$  不是互为质数的情况（因为不可能所有的  $m, n$  都是互质整数对），那么我们把它分成一个个互不影响的循环链，正如 flyinghearts 所言，所有序号为  $(j + i * m) \% n$  ( $j$  为 0 到  $\gcd(n, m)-1$  之间的某一整数， $i = 0:n-1$ ) 会构成一个循环链，一共有  $\gcd(n, m)$  个循环链，对每个循环链分别进行一次内循环就行了。

综合上述两种情况，可简单编写代码如下：

```
//④ 所有序号为 (j+i *m) % n (j 表示每个循环链起始位置, i 为计数变量, m 表示左旋转位数, n 表示字符串长度),  
//会构成一个循环链 (共有 gcd(n,m)个, gcd 为 n、m 的最大公约数) ,  
  
//每个循环链上的元素只要移动一个位置即可, 最后整个过程总共交换了 n 次  
// (每一次循环链, 是交换 n/gcd(n,m)次, 共有 gcd(n,m)个循环链, 所以, 总共交换 n 次) 。  
  
void rotate(string &str, int m)  
{  
    int lenOfStr = str.length();  
    int numofGroup = gcd(lenOfStr, m);  
    int elemInSub = lenOfStr / numofGroup;  
  
    for(int j = 0; j < numofGroup; j++)  
        //对应上面的文字描述, 外循环次数 j 为循环链的个数, 即 gcd(n, m)个循环链  
    {  
        char tmp = str[j];
```

```

    for (int i = 0; i < elemInSub - 1; i++)
        //内循环次数 i 为, 每个循环链上的元素个数, n/gcd(m,n)次
        str[(j + i * m) % lenOfStr] = str[(j + (i + 1) * m) % lenOfStr];
        str[(j + i * m) % lenOfStr] = tmp;
    }
}

```

后来有网友针对上述的思路④，给出了下述的证明：

1、首先，直观的看肯定是有循环链，关键是有几条以及每条有多长，根据 $(i+j*m) \% n$ 这个表达式可以推出一些东东，一个 j 对应一条循环链，现在要证明 $(i+j*m) \% n$ 有  $n/\gcd(n, m)$  个不同的数。

2、假设 j 和 k 对应的数字是相同的，即  $(i+j*m) \% n = (i+k*m) \% n$ ，可以推出  $n | (j-k)*m$ ,  $m=m' * \gcd(n, m)$ ,  $n=n' * \gcd(n, m)$ , 可以推出  $n' | (j-k)*m'$ ，而  $m'$  和  $n'$  互素，于是  $n' | (j-k)$ ，即  $(n/\gcd(n, m)) | (j-k)$ ，

3、所以  $(i+j*m) \% n$  有  $n/\gcd(n, m)$  个不同的数。则总共有  $\gcd(n, m)$  个循环链。符号“|”是整除的意思。

以上的 3 点关于为什么一共有  $\gcd(n, m)$  个循环链的证明，应该是来自 qq3128739xx 的，非常感谢这位朋友。

由于上述 stl rotate 源码中，方案④ 的代码，较复杂，难以阅读，下面是对上述第④ 方案的简单改写：

```

//对上述方案 4 的改写。
//④ 所有序号为 (i+t*k) % n (i 为指定整数, t 为任意整数), ....
//copyright@ hplonline && July 2011.04.18.
//July、sahala、yansha, updated, 2011.06.02.
void my_rotate(char *begin, char *mid, char *end)
{
    int n = end - begin;
    int k = mid - begin;
    int d = gcd(n, k);
    int i, j;
    for (i = 0; i < d; i++)
    {
        int tmp = begin[i];
        int last = i;

```

```

//i+k 为 i 右移 k 的位置, %n 是当 i+k>n 时从左重新开始。
for (j = (i + k) % n; j != i; j = (j + k) % n)    //多谢 laocpp 指正。
{
    begin[last] = begin[j];
    last = j;
}
begin[last] = tmp;
}
}

```

对上述程序的解释：关于第二个 for 循环中，j 初始化为  $(i+k) \% n$ ，程序注释中已经说了， $i+k$  为  $i$  右移  $k$  的位置， $\% n$  是当  $i+k > n$  时从左重新开始。为什么要这么做呢？很简单， $n$  个数的数组不管循环左移多少位，用上述程序的方法一共需要交换  $n$  次。当  $i+k >= n$  时  $i+k$  表示的位置在数组中不存在了，所以又从左边开始的  $(i+k)\%n$  是下一个交换的位置。

1. 好比 5 个学生，，编号从 0 开始，即 0 1 2 3 4，老师说报数，规则是从第一个学生开始，中间隔一个学生报数。报数的学生编号肯定是 0 2 4 1 3。这里就相当于  $i$  为 0， $k$  为 2， $n$  为 5；
2. 然后老师又说，编号为 0 的学生出列，其他学生到在他前一个报数的学生位置上去，那么学生从 0 1 2 3 4 => 2 3 4 \_ 1，最后老师说，编号 0 到剩余空位去，得到最终排位 2 3 4 0 1。此时的结果，实际上就是相当于上述程序中左移  $k=2$  个位置了。而至于为什么让 编号为 0 的学生 出列。实际是这句：int last = i；因为要达到这样的效果 0 1 2 3 4 => 2 3 4 0 1，那么 2 3 4 必须要移到前面去。怎么样，明白了么？。

关于本题，不少网友也给出了他们的意见，具体请参见此帖子[微软 100 题，维护地址](#)。

## 思路五、三步翻转法

对于这个问题，咱们换一个角度，可以这么做：

将一个字符串分成两部分，X 和 Y 两个部分，在字符串上定义反转的操作  $X^T$ ，即把 X 的所有字符反转（如， $X="abc"$ ，那么  $X^T="cba"$ ），那么我们可以得到下面的结论： $(X^T Y^T)^T=YX$ 。显然我们这就可以转化为字符串的反转的问题了。

不是么?ok, 就拿 abcdef 这个例子来说, 若要让 def 翻转到 abc 的前头, 那么只要按下述 3 个步骤操作即可:

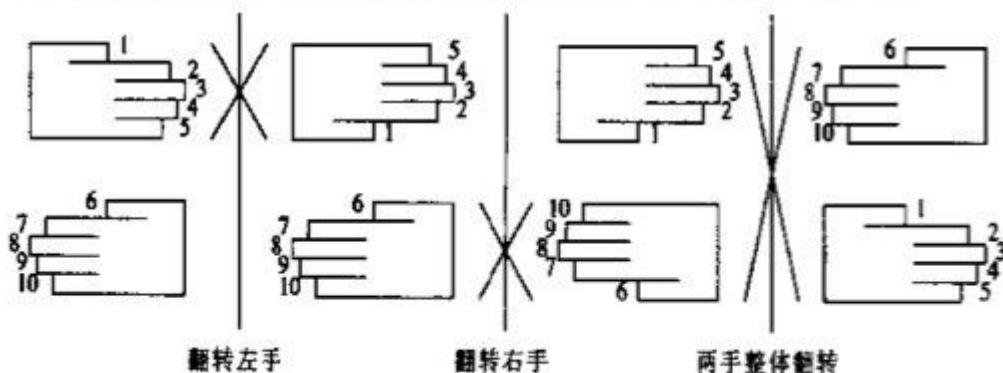
- 1、首先分为俩部分, X:abc, Y:def;
- 2、 $X \rightarrow X^T$ , abc->cba,  $Y \rightarrow Y^T$ , def->fed。
- 3、 $(X^T Y^T)^T = YX$ , cbafed->defabc, 即整个翻转。

我想, 这下, 你应该一目了然了。

其次, 在《编程珠玑》上也有这样一个类似的问题, 它的解法同本思路一致, 如下图所示:

#### \*. 可操作的证明方法-手摇法

如果要将一个具有10个元素(我们只有10个手指啊)的数组向上旋转5个位置, 先让两只手的掌你自己, 左右放在右手上面(其实两只手是同意平面上的), 看下图:



然后, 代码可以这么写:

```
//Copyright@ 小桥流水 && July
//c 代码实现, 已测试正确。
//http://www.smallbridge.co.cc/2011/03/13/100%E9%A2%98
//_21-%E5%B7%A6%E6%97%8B%E8%BD%AC%E5%AD%97%E7%AC%A6%E4%B8%B2.html
//July、updated, 2011.04.17。
char * invert(char *start, char *end)
{
    char tmp, *ptmp = start;
    while (start != NULL && end != NULL && start < end)
    {
```

```

        tmp = *start;
        *start = *end;
        *end = tmp;
        start++;
        end--;
    }
    return ptmp;
}

char *left(char *s, int pos) //pos 为要旋转的字符个数, 或长度, 下面主函数测试中,
pos=3。
{
    int len = strlen(s);
    invert(s, s + (pos - 1)); //如上, X->X^T, 即 abc->cba
    invert(s + pos, s + (len - 1)); //如上, Y->Y^T, 即 def->fed
    invert(s, s + (len - 1)); //如上, 整个翻转, (X^TY^T)^T=YX,
即 cbafed->defabc。
    return s;
}

```

完。

## 第二章、字符串是否包含问题

作者: July, yansha, caopengcs。

时间: 二零一一年四月二十三日。

致谢: 老梦, nossiac, Hession, Oliver, *Iuuillu*, 啊菜, 雨翔, 及微软 100 题实现小组所有成员。

### 题目描述:

假设这有一个各种字母组成的字符串 A, 和另外一个字符串 B, 字符串里 B 的字母数相对少一些。什么方法能最快的查出所有小字符串 B 里的字母在大字符串 A 里都有?

比如, 如果是下面两个字符串:

String 1: ABCDEFGHLMNOPQRS

String 2: DCGSRQP0

答案是 true, 所有在 string2 里的字母 string1 也都有。

如果是下面两个字符串:

String 1: ABCDEFGHLMNOPQRS

String 2: DCGSRQPZ

答案是 false, 因为第二个字符串里的 Z 字母不在第一个字符串里。

### 点评:

1、题目描述虽长, 但题意简单明了, 就是给定一长一短的俩个字符串 A, B, 假设 A 长 B 短, 现在, 要你判断 B 是否包含在字符串 A 中, 即  $B \subseteq A$ 。

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

ok，在继续往下阅读之前，您最好先想个几分钟，看你能想到的最好方案是什么，是否与本文最后实现的方法一致。

## 第一节、几种常规解法

### 1.1、 $O(n*m)$ 的轮询方法

判断 string2 中的字符是否在 string1 中?:

String 1: ABCDEFGHLMNOPQRS

String 2: DCGSRQPO

判断一个字符串是否在另一个字符串中，最直观也是最简单的思路是，针对第二个字符串 string2 中每一个字符，一一与第一个字符串 string1 中每个字符依次轮询比较，看它是否在第一个字符串 string1 中。

代码可如下编写：

```
//copyright@啊菜 2011
//updated@July&Image 、时光 2013
#include <iostream>
#include <string>
using namespace std;

int CompareString(string LongString,string ShortString)
{
    int i,j;
    for (i=0; i<ShortString.length(); i++)
    {
        for (j=0; j<LongString.length(); j++) //O(n*m)
        {
            if (LongString[j] == ShortString[i]) //一一比较
            {
                break;
            }
        }
        if (j==LongString.length())
        {
            cout << "false" << endl;
            return 0;
        }
    }
}
```

```

        }
    }

    cout << "true" << endl;
    return 1;
}

int main()
{
    string LongString="ABCDEFGHIJKLMNOPQRS";
    string ShortString="DCGSRQPO";
    CompareString(LongString,ShortString);
    return 0;
}

```

假设  $n$  是字符串  $string1$  的长度,  $m$  是字符串  $string2$  的长度, 那么此算法, 需要  $O(n*m)$  次操作, 拿上面的例子来说, 最坏的情况下将会有  $16*8 = 128$  次操作。显然, 时间开销太大, 我们需要找到一种更好的办法。

## 1.2、 $O(m\log m) + O(n \log n) + O(m+n)$ 的排序方法

一个稍微好一点的方案是先对这两个字符串的字母进行排序, 然后同时对两个字串依次轮询。两个字串的排序需要(常规情况) $O(m \log m) + O(n \log n)$  次操作, 之后的线性扫描需要  $O(m+n)$  次操作。

同样拿上面的字串做例子, 将会需要  $16*4 + 8*3 = 88$ , 再加上对两个字串线性扫描的  $16 + 8 = 24$  的操作。(随着字串长度的增长, 你会发现这个算法的效果会越来越好)

关于采用何种排序方法, 我们采用最常用的快速排序, 下面的[快速排序](#)的代码用的是以前写的, 比较好懂, 并且, 我执意不用库函数的 `qsort` 代码。唯一的问题是, 此前写的代码是针对整数进行排序的, 不过, 难不倒我们, 稍微改一下参数, 即可, 如下:

```

//copyright@ 2011 July && yansha
//July, updated, 2011.04.23.
#include <iostream>
#include <string>
using namespace std;

```

```

//以前的注释，还让它保留着

int partition(string &str, int lo, int hi)
{
    int key = str[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(str[j] <= key)
        {
            i++;
            swap(str[i], str[j]);
        }
    }
    swap(str[i+1], str[hi]);    //不能改为 swap(&data[i+1],&key)
    return i + 1;
}

//递归调用上述 partition 过程, 完成排序。

void quicksort(string &str, int lo, int hi)
{
    if (lo < hi)
    {
        int k = partition(str, lo, hi);
        quicksort(str, lo, k - 1);
        quicksort(str, k + 1, hi);
    }
}

//比较, 上述排序 O(m log m) + O(n log n), 加上下面的 O(m+n),
//时间复杂度总计为: O(mlogm)+O(nlogn)+O(m+n)。
void compare(string str1, string str2)
{
    int posOne = 0;
    int posTwo = 0;
    while (posTwo < str2.length() && posOne < str1.length())
    {
        while (str1[posOne] < str2[posTwo] && posOne < str1.length() - 1)
            posOne++;
        //如果和 str2 相等, 那就不能动。只有比 str2 小, 才能动。

        if (str1[posOne] != str2[posTwo])
            break;
    }
}

```

```

        //posOne++;
        //归并的时候, str1[str1Pos] == str[str2Pos]的时候, 只能 str2Pos++, str1Pos 不可以
        自增。
        //多谢 helloworld 指正。

    posTwo++;
}

if (posTwo == str2.length())
    cout << "true" << endl;
else
    cout << "false" << endl;
}

int main()
{
    string str1 = "ABCDEFGHLMNOPQRS";
    string str2 = "DCGDSRQPOM";
    //之前上面加了那句 posOne++之所以有 bug, 是因为, @helloworld:
    //因为 str1 如果也只有一个 D, 一旦 posOne++, 就到了下一个不是'D'的字符上去了,
    //而 str2 有俩 D, posTwo++后, 下一个字符还是'D', 就不等了, 出现误判。

    quicksort(str1, 0, str1.length() - 1);
    quicksort(str2, 0, str2.length() - 1); //先排序
    compare(str1, str2); //后线性扫描
    return 0;
}

```

### 1.3、 $O(n+m)$ 的计数排序方法

此方案与上述思路相比，就是在排序的时候采用线性时间的计数排序方法，排序  $O(n+m)$ ，线性扫描  $O(n+m)$ ，总计时间复杂度为： $O(n+m) + O(n+m) = O(n+m)$ 。

代码如下：

```

#include <iostream>
#include <string>
using namespace std;

// 计数排序, O(n+m)
void CounterSort(string str, string &help_str)

```

```

{
    // 辅助计数数组
    int help[26] = {0};

    // help[index]存放了等于 index + 'A'的元素个数
    for (int i = 0; i < str.length(); i++)
    {
        int index = str[i] - 'A';
        help[index]++;
    }

    // 求出每个元素对应的最终位置
    for (int j = 1; j < 26; j++)
        help[j] += help[j-1];

    // 把每个元素放到其对应的最终位置
    for (int k = str.length() - 1; k >= 0; k--)
    {
        int index = str[k] - 'A';
        int pos = help[index] - 1;
        help_str[pos] = str[k];
        help[index]--;
    }
}

//线性扫描 O (n+m)
void Compare(string long_str,string short_str)
{
    int pos_long = 0;
    int pos_short = 0;
    while (pos_short < short_str.length() && pos_long < long_str.length())
    {
        // 如果 pos_long 递增直到 long_str[pos_long] >= short_str[pos_short]
        while (long_str[pos_long] < short_str[pos_short] && pos_long < long_str.length()
            () - 1)
            pos_long++;

        // 如果 short_str 有连续重复的字符， pos_short 递增
        while (short_str[pos_short] == short_str[pos_short+1])
            pos_short++;

        if (long_str[pos_long] != short_str[pos_short])

```

```

        break;

    pos_long++;
    pos_short++;
}

if (pos_short == short_str.length())
    cout << "true" << endl;
else
    cout << "false" << endl;
}

int main()
{
    string strOne = "ABCDLK";
    string strTwo = "A";
    string long_str = strOne;
    string short_str = strTwo;

    // 对字符串进行计数排序
    CounterSort(strOne, long_str);
    CounterSort(strTwo, short_str);

    // 比较排序好的字符串
    Compare(long_str, short_str);
    return 0;
}

```

不过上述方法，空间复杂度为  $O(n+m)$ ，即消耗了一定的空间。有没有在线性时间，且空间复杂度较小的方案列？

## 第二节、寻求线性时间的解法

### 2.1、 $O(n+m)$ 的 hashtable 的方法

上述方案中，较好的方法是先对字符串进行排序，然后再线性扫描，总的时间复杂度已经优化到了： $O(m+n)$ ，貌似到了极限，还有没有更好的办法列？

我们可以对短字串进行轮询（此思路的叙述可能与网上的一些叙述有出入，因为我们最好是应该把短的先存储，那样，会降低题目时间复杂度），把其中的每个字母都放入一个 **Hashtable** 里（我们始终设  $m$  为短字符串的长度，那么此项操作成本是  $O(m)$  或 8 次操作）。然后轮询长字符串，在 **Hashtable** 里查询短字符串的每个字符，看能否找到。如果找不到，说明没有匹配成功，轮询长字符串将消耗掉 16 次操作，这样两项操作加起来一共只有  $8+16=24$  次。

当然，理想情况是如果长字串的前缀就为短字串，只需消耗 8 次操作，这样总共只需  $8+8=16$  次。

或如梦想天窗所说： 我之前用散列表做过一次，算法如下：

- 1、 $\text{hash}[26]$ ，先全部清零，然后扫描短的字符串，若有相应的置 1，
- 2、计算  $\text{hash}[26]$  中 1 的个数，记为  $m$
- 3、扫描长字符串的每个字符  $a$ ；若原来  $\text{hash}[a] == 1$ ，则修改  $\text{hash}[a] = 0$ ，并将  $m$  减 1；若  $\text{hash}[a] == 0$ ，则不做处理
- 4、若  $m == 0$  or 扫描结束，退出循环。

代码实现，也不难，如下：

```
//copyright@ 2011 yansha
//July、updated, 2011.04.25。
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string str1="ABCDEFGHIJKLMOPQRS";
    string str2="DCGSRQPOM";

    // 开辟一个辅助数组并清零
    int hash[26] = {0};

    // num 为辅助数组中元素个数
    int num = 0;

    // 扫描短字符串
```

```

for (int j = 0; j < str2.length(); j++)
{
    // 将字符转换成对应辅助数组中的索引
    int index = str1[j] - 'A';

    // 如果辅助数组中该索引对应元素为 0，则置 1，且 num++;
    if (hash[index] == 0)
    {
        hash[index] = 1;
        num++;
    }
}

// 扫描长字符串
for (int k = 0; k < str1.length(); k++)
{
    int index = str1[k] - 'A';

    // 如果辅助数组中该索引对应元素为 1，则 num--;为零的话，不作处理（不写语句）。
    if(hash[index] ==1)
    {
        hash[index] = 0;
        num--;
        if(num == 0)      //m==0, 即退出循环。
            break;
    }
}

// num 为 0 说明长字符串包含短字符串内所有字符
if (num == 0)
    cout << "true" << endl;
else
    cout << "false" << endl;
return 0;
}

```

## 2.2、O (n+m) 的数组存储方法

有两个字符串 short\_str 和 long\_str。

第一步：你标记 short\_str 中有哪些字符，在 store 数组中标记为 true。  
(store 数组起一个映射的作用，如果有 A，则将第 1 个单元标记 true，如果有 B，则将第 2 个单元标记 true，... 如果有 Z，则将第 26 个单元标记 true)

第二步：遍历 long\_str，如果 long\_str 中的字符包括 short\_str 中的字符，则将 store 数组中对应位置标记为 false。(如果有 A，则将第 1 个单元标记 false，如果有 B，则将第 2 个单元标记 false，... 如果有 Z，则将第 26 个单元标记 false)，如果没有，则不作处理。

第三步：此后，遍历 store 数组，如果所有的元素都是 false，也就说明 store\_str 中字符都包含在 long\_str 内，输出 true。否则，输出 false。

举个简单的例子好了，如 abcd，abcdefg 两个字符串，

1、先遍历短字符串 abcd，在 store 数组中相对应的 abcd 的位置上的单元元素置为 true，

2、然后遍历 abcdefg，在 store 数组中相应的 abcd 位置上，发现已经有了 abcd，则前 4 个的单元元素都置为 false，当我们已经遍历了 4 个元素，等于了短字符串 abcd 的 4 个数目，所以，满足条件，退出。

(不然，继续遍历的话，我们会发现 efg 在 store 数组中没有元素，不作处理。最后，自然，就会发现 store 数组中的元素单元都是 false 的。)

3、遍历 store 数组，发现所有的元素都已被置为 false，所以程序输出 true。

其实，这个思路和上一节中， $O(n+m)$  的 hashtable 的方法代码，原理是完全一致的，且本质上都采用的数组存储（hash 表也是一个数组），但我并不认为此思路多此一举，所以仍然贴出来。ok，代码如下：

```
//copyright@ 2011 Hession
//July、updated, 2011.04.23.
#include<iostream>
#include<string.h>
using namespace std;

int main()
{
    char long_ch[]="ABCDEFGHIJKLMNPQRS";
```

```

char short_ch[]="DEFGHXLMNOPQ";
int i;
bool store[58];
memset(store, false, 58);

//前两个 是 遍历 两个字符串，后面一个是 遍历 数组
for(i=0;i<sizeof(short_ch)-1;i++)
    store[short_ch[i]-65]=true;

for(i=0;i<sizeof(long_ch)-1;i++)
{
    if(store[long_ch[i]-65]!=false)
        store[long_ch[i]-65]=false;
}

for(i=0;i<58;i++)
{
    if(store[i]!=false)
    {
        cout<<"short_ch is not in long_ch"<<endl;
        break;
    }
    if(i==57)
        cout<<"short_ch is in long_ch"<<endl;
}

return 0;
}

```

### 第三节、O(n) 到 O(n+m) 的素数方法

我想问的是，还有更好的方案么？

你可能会这么想：O(n+m) 是你能得到的最好的结果了，至少要对每个字母至少访问一次才能完成这项操作，而上一节最后的俩个方案是刚好是对每个字母只访问一次。

ok，下面给出一个更好的方案：

假设我们有一个一定个数的字母组成字串，我给每个字母分配一个素数，从2开始，往后类推。这样A将会是2，B将会是3，C将会是5，等等。现在我遍

历第一个字串，把每个字母代表的素数相乘。你最终会得到一个很大的整数，对吧？

然后——轮询第二个字符串，用每个字母除它。如果除的结果有余数，这说明有不匹配的字母。如果整个过程中没有余数，你应该知道它是第一个字串恰好子集了。

思路总结如下：

1. 定义最小的 26 个素数分别与字符' A' 到' Z' 对应。
2. 遍历长字符串，求得每个字符对应素数的乘积。
3. 遍历短字符串，判断乘积能否被短字符串中的字符对应的素数整除。
4. 输出结果。

至此，如上所述，上述算法的时间复杂度为  $O(m+n)$ ，时间复杂度最好的情况为  $O(n)$ （[遍历短的字符串的第一个数，与长字符串素数的乘积相除，即出现余数，便可退出程序，返回 false](#)）， $n$  为长字串的长度，空间复杂度为  $O(1)$ 。如你所见，我们已经优化到了最好的程度。

不过，正如原文中所述：“现在我想告诉你 —— Guy 的方案在算法上并不能说就比我的好。而且在实际操作中，你很可能仍会使用我的方案，因为它更通用，无需跟麻烦的大型数字打交道。但从”巧妙水平“上讲，Guy 提供的是一种更、更、更有趣的方案。”

ok，如果你有更好的思路，欢迎在本文的评论中给出，非常感谢。

```
#include <iostream>
#include <string>
#include "BigInt.h"
using namespace std;

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

int main()
```

```

{
    string strOne = "ABCDEFGHIJKLMNPQRS";
    string strTwo = "DCGSRQPOM";

    // 这里需要用到大整数
    CBigInt product = 1;    // 大整数除法的代码，下头给出。

    // 遍历长字符串，得到每个字符对应素数的乘积
    for (int i = 0; i < strOne.length(); i++)
    {
        int index = strOne[i] - 'A';
        product = product * primeNumber[index];
    }

    // 遍历短字符串
    for (int j = 0; j < strTwo.length(); j++)
    {
        int index = strTwo[j] - 'A';

        // 如果余数不为 0，说明不包括短字符串中的字符，跳出循环
        if (product % primeNumber[index] != 0)
            break;
    }

    // 如果积能整除短字符串中所有字符则输出"true"，否则输出"false"。
    if (strTwo.length() == j)
        cout << "true" << endl;
    else
        cout << "false" << endl;
    return 0;
}

```

上述程序待改进的地方

1. 只考虑大写字符，如果考虑小写字符和数组的话，素数数组需要更多素数
2. 没有考虑重复的字符，可以加入判断重复字符的辅助数组。

大整数除法的代码，后续公布下载地址。

说明：此次的判断字符串是否包含问题，来自一位外国网友提供的 gofish、google 面试题，这个题目出自此篇文章：

<http://www.aqee.net/2011/04/11/google-interviewing-story/>, 文章记录了整个面试的过程，比较有趣，值得一读。

**扩展：**正如网友安逸所说：其实这个问题还可以转换为：a 和 b 两个字符串，求 b 串包含 a 串的最小长度。包含指的就是 b 的字串包含 a 中每个字符。

**updated：**我们假设字母都由大写字母组成……，我们先对小字符串预处理，可以得到 B 里包含哪些字符，这里可以用位运算，或者用 bool 数组。位运算简单些，用一个 int 中的 26bit 表示其是否在 B 中出现即可。

```
//copyright@ caopengcs 2013
bool AcontainsB(char *A,char *B) {
    int have = 0;
    while (*B) {
        have |= 1 << (*B++) - 'A'; // 把A..Z 对应为 0..26
    }
    while (*A) {
        if ((have & (1 << (*(A++) - 'A')))) == 0) {
            return false;
        }
    }
    return true;
}
```

完。

## 第三章、寻找最小的 k 个数

作者：July。

时间：二零一一年四月二十八日。

致谢：litaoye，strugglever，yansha，luuillu，Sorehead，及狂想曲创作组。

微博：<http://weibo.com/julyweibo>。

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

---

### 前奏

**@July\_\_\_\_\_**：1、当年明月：“我写文章有个习惯，由于早年读了太多学究书，所以很痛恨那些故作高深的文章，其实历史本身很精彩，所有的历史都可以写得很好看，....。”2、IT 技术文章，亦是如此，可以写得很通俗，很有趣，而非故作高深。希望，我可以做到。

下面，我试图用最清晰易懂，最易令人理解的思维或方式阐述有关寻找最小的 k 个数这个问题（这几天一直在想，除了计数排序外，这题到底还有没有其它的 O(n) 的算法？）。希望，有任何问题，欢迎不吝指正。谢谢。

### 寻找最小的 k 个数

题目描述：5. 查找最小的 k 个元素

题目：输入 n 个整数，输出其中最小的 k 个。

例如输入 1, 2, 3, 4, 5, 6, 7 和 8 这 8 个数字，则最小的 4 个数字为 1, 2, 3 和 4。

### 第一节、各种思路，各种选择

- 0、咱们先简单的理解，要求一个序列中最小的 k 个数，按照惯有的思维方式，很简单，先对这个序列从小到大排序，然后输出前面的最小的 k 个数即可。

- 1、至于选取什么的排序方法，我想你可能会第一时间想到快速排序，我们知道，快速排序平均所费时间为  $n \log n$ ，然后再遍历序列中前  $k$  个元素输出，即可，总的时间复杂度为  $O(n \log n + k) = O(n \log n)$ 。
- 2、咱们再进一步想想，题目并没有要求要查找的  $k$  个数，甚至后  $n-k$  个数是有序的，既然如此，咱们又何必对所有的  $n$  个数都进行排序列？

这时，咱们想到了用选择或交换排序，即遍历  $n$  个数，先把最先遍历到得  $k$  个数存入大小为  $k$  的数组之中，对这  $k$  个数，利用选择或交换排序，找到  $k$  个数中的最大数  $k_{max}$  ( $k_{max}$  设为  $k$  个元素的数组中最大元素)，用时  $O(k)$  (你应该知道，插入或选择排序查找操作需要  $O(k)$  的时间)，后再继续遍历后  $n-k$  个数， $x$  与  $k_{max}$  比较：如果  $x < k_{max}$ ，则  $x$  代替  $k_{max}$ ，并再次重新找出  $k$  个元素的数组中最大元素  $k_{max}$  (多谢 [kk791159796 提醒修正](#))；如果  $x > k_{max}$ ，则不更新数组。这样，每次更新或不更新数组的所用的时间为  $O(k)$  或  $O(0)$ ，整趟下来，总的时间复杂度平均下来为： $n * O(k) = O(n * k)$ 。

- 3、当然，更好的办法是维护  $k$  个元素的最大堆，原理与上述第 2 个方案一致，即用容量为  $k$  的最大堆存储最先遍历到的  $k$  个数，并假设它们即是最小的  $k$  个数，建堆费时  $O(k)$  后，有  $k_1 < k_2 < \dots < k_{max}$  ( $k_{max}$  设为大顶堆中最大元素)。继续遍历数列，每次遍历一个元素  $x$ ，与堆顶元素比较， $x < k_{max}$ ，更新堆（用时  $\log k$ ），否则不更新堆。这样下来，总费时  $O(k + (n-k) * \log k) = O(n * \log k)$ 。此方法得益于在堆中，查找等各项操作时间复杂度均为  $\log k$  (不然，就如上述思路 2 所述：直接用数组也可以找出前  $k$  个大的元素，用时  $O(n * k)$ )。
- 4、按编程之美第 141 页上解法二的所述，类似快速排序的划分方法， $N$  个数存储在数组  $S$  中，再从数组中随机选取一个数  $X$  (随机选取枢纽元，可做到线性期望时间  $O(N)$  的复杂度，在第二节论述)，把数组划分为  $S_a$  和  $S_b$  俩部分， $S_a \leq X \leq S_b$ ，如果要查找的  $k$  个元素小于  $S_a$  的元素个数，则返回  $S_a$  中较小的  $k$  个元素，否则返回  $S_a$  中所有元素+ $S_b$  中的  $k - |S_a|$  个元素。像上述过程一样，这个运用类似快速排序的 `partition` 的快速选择 `SELECT` 算法寻找最小的  $k$  个元素，在最坏情况下亦能做到  $O(N)$  的复杂度。不过值得一提的是，这个快速选择 `SELECT` 算法是选取数组中“中位数的中位数”作为枢纽元，而非随机选取枢纽元。
- 5、RANDOMIZED-SELECT，每次都是随机选取数列中的一个元素作为主元，在  $O(n)$  的时间内找到第  $k$  小的元素，然后遍历输出前面的  $k$  小的元

素。如果能的话，那么总的时间复杂度为线性期望时间： $O(n+k) = O(n)$ （当  $k$  比较小）。

Ok，稍后第二节中，我会具体给出 RANDOMIZED-SELECT(A, p, r, i) 的整体完整伪码。在此之前，要明确一个问题：我们通常所熟知的快速排序是以固定的第一个或最后一个元素作为主元，每次递归划分都是不均等的，最后的平均时间复杂度为： $O(n \log n)$ ，但 RANDOMIZED-SELECT 与普通的快速排序不同的是，每次递归都是随机选择序列从第一个到最后一个元素中任一个作为主元。

- 6、线性时间的排序，即计数排序，时间复杂度虽能达到  $O(n)$ ，但限制条件太多，不常用。

- 7、[updated](#): huaye502 在本文的评论下指出：“可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  $O(n) + k * O(\log n)$ ”。

huaye502 的意思是针对整个数组序列建最小堆，建堆所用时间为  $O(n)$ （算法导论一书上第 6 章第 6.3 节已经论证，在线性时间内，能将一个无序的数组建成一个最小堆），然后取堆中的前  $k$  个数，总的时间复杂度即为： $O(n+k \log n)$ 。

关于上述第 7 点思路的继续阐述：至于思路 7 的  $O(n+k \log n)$  是否小于上述思路 3 的  $O(n \log k)$ ，即  $O(n+k \log n) < O(n \log k)$ 。粗略数学证明可参看如下第一幅图，我们可以这么解决：当  $k$  是常数， $n$  趋向于无穷大时，求  $(n \log k) / (n+k \log n)$  的极限  $T$ ，如果  $T > 1$ ，那么可得  $O(n \log k) > O(n+k \log n)$ ，也就是  $O(n+k \log n) < O(n \log k)$ 。虽然这有违我们惯常的思维，然事实最终证明的确如此，这个极值  $T = \log k > 1$ ，即采取建立  $n$  个元素的最小堆后取其前  $k$  个数的方法的复杂度小于采取常规的建立  $k$  个元素最大堆后通过比较寻找最小的  $k$  个数的方法的复杂度。但，最重要的是，如果建立  $n$  个元素的最小堆的话，那么其空间复杂度势必为  $O(N)$ ，而建立  $k$  个元素的最大堆的空间复杂度为  $O(k)$ 。所以，综合考虑，我们一般还是选择用建立  $k$  个元素的最大堆的方法解决此类寻找最小的  $k$  个数的问题。

思路 3 准确的时间复杂度表述为： $O(k + (n-k) \log k)$ ，思路 7 准确的时间复杂度表述为： $O(n+k \log n)$ ，也就是如 gbb21 所述粗略证明：[要证原式  \$k+n \log k - n - k \log n > 0\$ ，等价于证  \$\(\log k - 1\)n - k \log n + k > 0\$](#) ，要证思路 3 的时间复杂度大于思路 7 的时间复杂度，等价于要证原式  $k+n \log k - k \log k - n - k \log n > 0$ ，即证  $(\log k - 1)n - k \log n + k - k \log k > 0$ 。[当  \$n \rightarrow +/\inf\$  \( \$n\$  趋向于正无穷大\) 时， \$\log k - 1 - 0 - 0 > 0\$ ，即只要满足  \$\log k - 1 > 0\$  即可。原式得证。即  \$O\(n+k \log n\) < O\(n \log k\)\$](#)

$(k+n*\log k) > O(n+k*\log n) \Rightarrow O(n+k*\log n) < O(n*\log k)$ ，与上面得到的结论一致。

事实上，是建立最大堆还是建立最小堆，其实际的程序运行时间相差并不大，运行时间都在一个数量级上。因为后续，我们还专门写了个程序进行测试，即针对 1000w 的数据寻找其中最小的 k 个数的问题，采取两种实现，一是采取常规的建立 k 个元素最大堆后通过比较寻找最小的 k 个数的方案，一是采取建立 n 个元素的最小堆，然后取其前 k 个数的方法，发现两相比较，运行时间实际上相差无几。结果可看下面的第二幅图。

~~Heaps~~ =  $y$

$k$  是常数

$$\lim_{n \rightarrow \infty} \frac{k + n \lg k}{n + k \lg n} = \lim_{n \rightarrow \infty} \frac{\frac{k}{n}}{1 + \frac{k \lg n}{n}} + \lim_{n \rightarrow \infty} \frac{n \lg k}{n + k \lg n} \quad ①$$

$$\lim_{n \rightarrow \infty} \frac{k}{n + k \lg n} = \lim_{n \rightarrow \infty} \frac{\frac{k}{n}}{1 + \frac{k \lg n}{n}} = ②$$

$$\lim_{n \rightarrow \infty} \frac{n \lg k}{n + k \lg n} = \lim_{n \rightarrow \infty} \frac{\lg k}{1 + \frac{k \lg n}{n}} = \frac{\lg k}{1 + \lim_{n \rightarrow \infty} \frac{k \lg n}{n}} \approx \lg k \quad ③$$

$$① \approx ② + ③ = \lg k \quad \text{BP 且 } k > 2 \text{ 时}$$

$$k + n \lg k > n + k \lg n$$

```

choose the min 256 numbers in 10000000 directly: 407ms
choose the min 256 numbers in 10000000 by max heap: 375ms
choose the min 256 numbers in 10000000 by min heap: 391ms
Press any key to continue.

```

- 8、@lingyun310：与上述思路 7 类似，不同的是在对元素数组原地建最小堆  $O(n)$  后，然后提取  $K$  次，但是每次提取时，换到顶部的元素只需要下移顶多  $k$  次就足够了，下移次数逐次减少（而上述思路 7 每次提取都需要

[要  \$\log n\$ , 所以提取 k 次, 思路 7 需要  \$k \* \log n\$ 。而本思路 8 只需要  \$K^2\$ \)。](#)

此种方法的复杂度为  $O(n+k^2)$ 。[@July: 对于这个  \$O\(n+k^2\)\$  的复杂度, 我相当怀疑。因为据我所知, n 个元素的堆, 堆中任何一项操作的复杂度皆为  \$\log n\$ , 所以按理说, lingsyun310 方法的复杂度应该跟下述思路 8 一样, 也为  \$O\(n+k \* \log n\)\$ , 而非  \$O\(n+k \* k\)\$ 。](#) ok, 先放到这, 待时间考证。

[06. 02.](#)

### **updated:**

经过和几个朋友的讨论, 已经证实, 上述思路 7 lingsyun310 所述的思路应该是完全可以的。下面, 我来具体解释下他的这种方法。

我们知道, n 个元素的最小堆中, 可以先取出堆顶元素得到我们第 1 小的元素, 然后把堆中最后一个元素(较大的元素)上移至堆顶, 成为新的堆顶元素(取出堆顶元素之后, 把堆中下面的最后一个元素送到堆顶的过程可以参考下面的第一幅图。至于为什么是这样做, 为什么是把最后一个元素送到堆顶成为堆顶元素, 而不是把原来堆顶元素的儿子送到堆顶呢?具体原因可参考相关书籍)。

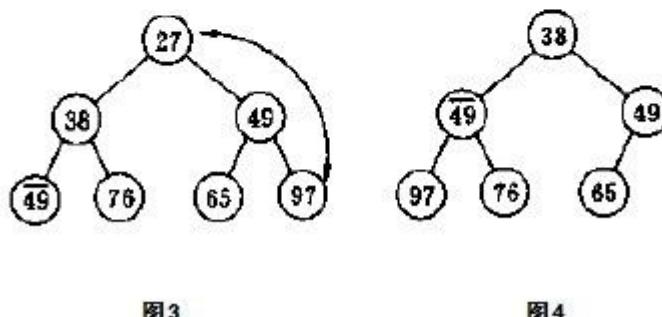
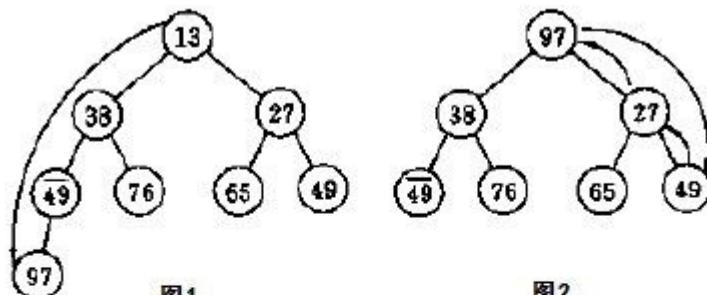
此时, 堆的性质已经被破坏了, 所以此后要调整堆。怎么调整呢?就是一般人所说的针对新的堆顶元素 shiftdown, 逐步下移(因为新的堆顶元素由最后一个元素而来, 比较大嘛, 既然是最小堆, 当然大的元素就要下沉到堆的下部了)。下沉多少步呢?即如 lingsyun310 所说的, 下沉 k 次就足够了。

下移 k 次之后, 此时的堆顶元素已经是我们要找的第 2 小的元素。然后, 取出这个第 2 小的元素(堆顶元素), 再次把堆中的最后一个元素送到堆顶, 又经过  $k-1$  次下移之后(此后下移次数逐步减少,  $k-2, k-3, \dots, k=0$  后算法中断)...., 如此重复  $k-1$  趟操作, 不断取出的堆顶元素即是我们要找的最小的 k 个数。虽然上述算法中断后整个堆已经不是最小堆了, 但是求得的 k 个最小元素已经满足我们题目所要求的了, 就是说已经找到了最小的 k 个数, 那么其它的咱们不管了。

我可以再举一个形象易懂的例子。你可以想象在一个水桶中, 有很多的气泡, 这些气泡从上到下, 总体的趋势是逐渐增大的, 但却不是严格的逐次大(正好这也符合最小堆的性质)。ok, 现在我们取出第一个气泡, 那这个气泡一定是水桶中所有气泡中最小的, 把它拿出来, 然后把最下面的那个大气泡(但不一定是最大的气泡)移到最上面去, 此时违反了气泡从上到下总体上逐步变大的趋势, 所以, 要把这个大气泡往下沉, 下沉到哪个位置呢?就是下沉 k 次。下沉 k 次后,

最上面的气泡已经肯定是最小的气泡了，把他再次取出。然后又将最下面最后的那个气泡移至最上面，移到最上面后，再次让它逐次下沉，下沉  $k-1$  次...，如此循环往复，最终取到最小的  $k$  个气泡。

ok，所以，上面方法所述的过程，更进一步来说，其实是第一趟调整保持第 0 层到第  $k$  层是最小堆，第二趟调整保持第 0 层到第  $k-1$  层是最小堆...，依次类推。但这个思路只是下述思路 8 中正规的最小堆算法（因为它最终对全部元素都进行了调整，算法结束后，整个堆还是一个最小堆）的调优，时间复杂度  $O(n+k^2)$  没有量级的提高，空间复杂度为  $O(N)$  也不会减少。



原理理解透了，那么写代码，就不难了，完整粗略代码如下（有问题烦请批评指正）：

```
//copyright@ 泡泡鱼
//July、2010. 06. 02.

//@lingyun310: 先对元素数组原地建最小堆，O(n)。然后提取 K 次，但是每次
//提取时，
//换到顶部的元素只需要下移顶多 k 次就足够了，下移次数逐次减少。此种方
//法的复杂度为 O(n+k^2)。
#include <stdio.h>
#include <stdlib.h>
```

```

#define MAXLEN 123456
#define K 100

// 
void HeapAdjust(int array[], int i, int Length)
{
    int child, temp;
    for(temp=array[i];2*i+1<Length;i=child)
    {
        child = 2*i+1;
        if(child<Length-1 && array[child+1]<array[child])

            child++;
        if (temp>array[child])
            array[i]=array[child];
        else
            break;
        array[child]=temp;
    }
}

void Swap(int* a, int* b)
{
    *a=*a^*b;
    *b=*a^*b;
    *a=*a^*b;
}

int GetMin(int array[], int Length, int k)
{
    int min=array[0];
    Swap(&array[0],&array[Length-1]);

    int child, temp;
    int i=0, j=k-1;
    for (temp=array[0]; j>0 && 2*i+1<Length; --j, i=child)

    {
        child = 2*i+1;
        if(child<Length-1 && array[child+1]<array[child])

            child++;
        if (temp>array[child])
            array[i]=array[child];
    }
}

```

```

        else
            break;
        array[child]=temp;
    }

    return min;
}

void Kmin(int array[] , int Length , int k)
{
    for(int i=Length/2-1;i>=0;--i)
        //初始建堆，时间复杂度为 O(n)
        HeapAdjust(array, i, Length);

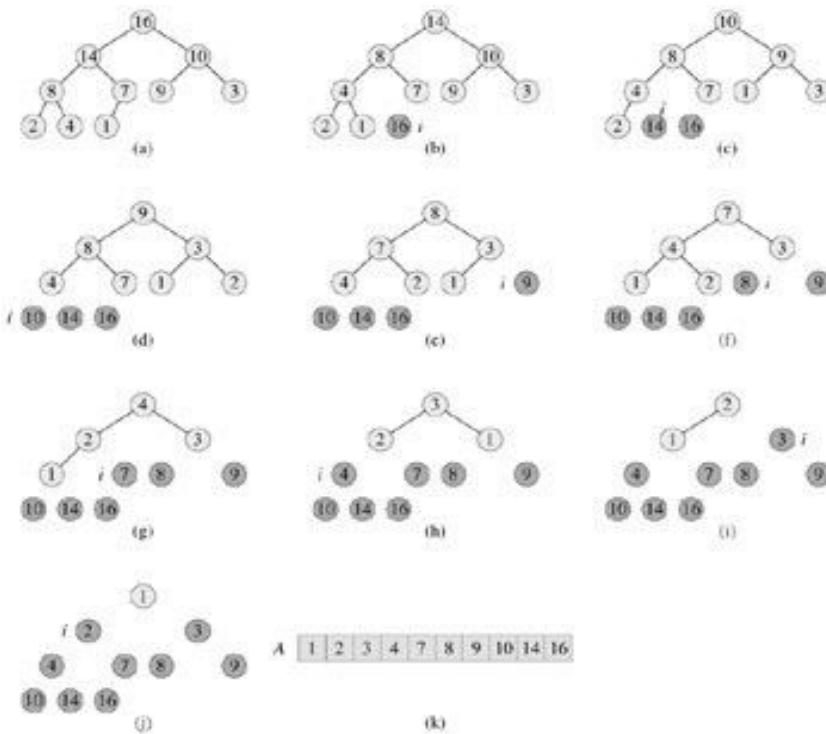
    int j=Length;
    for(i=k;i>0;--i,--j)
        //k 次循环，每次循环的复杂度最多为 k 次交换，复杂度为
        o(k^2)
    {
        int min=GetMin(array, j, i);
        printf("%d, ", min);
    }
}

int main()
{
    int array[MAXLEN];
    for(int i=MAXLEN;i>0;--i)
        array[MAXLEN-i] = i;

    Kmin(array, MAXLEN, K);
    return 0;
}

```

在算法导论第 6 章有下面这样一张图，因为开始时曾一直纠结过这个问题，“取出堆顶元素之后，把堆中下面的最后一个元素送到堆顶”。因为算法导论上下面这张图给了我一个假象，从 a)  $\rightarrow$  b) 中，让我误以为是取出堆顶元素之后，是把原来堆顶元素的儿子送到堆顶。而事实上不是这样的。因为在下面的图中，16 被删除后，堆中最后一个元素 1 替代 16 成为根结点，然后 1 下沉（注意下图所示的过程是最大堆的堆排序过程，不再是上面的最小堆了，所以小的元素当然要下移），14 上移到堆顶。所以，图中小图图 b) 是已经在小图 a) 之和被调整过的最大堆了，只是调整了 logn 次，非上面所述的 k 次。



ok, 接下来, 咱们再着重分析下上述思路 4。或许, 你不会相信上述思路 4 的观点, 但我马上将用事实来论证我的观点。这几天, 我一直在想, 也一直在找资料查找类似快速排序的 partition 过程的分治算法(即上述在编程之美上提到的第 4 点思路), 是否能做到  $O(N)$  的论述或证明,

然找了三天, 不但在算法导论上找到了 RANDOMIZED-SELECT, 在平均情况下为线性期望时间  $O(N)$  的论证(请参考本文第二节), 还在 mark allen weiss 所著的数据结构与算法分析--c 语言描述一书(还得多谢朋友 sheguang 提醒)中, 第 7 章第 7.7.6 节(本文下面的第 4 节末, 也有关此问题的阐述)也找到了在最坏情况下, 为线性时间  $O(N)$ (是的, 不含期望, 是最坏情况下为  $O(N)$ ) 的快速选择算法(此算法, 本文文末, 也有阐述), 请看下述文字(括号里的中文解释为本人添加):

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  $O(n + k \log n)$  (即上述思路 7) . 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 (快速选择, 即上述编程之美一书上的, 思路 4, 步骤如下) :

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,  $v \in S$ . (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  $k \leq |S_1|$ , then the  $k$ th smallest element must be in  $S_1$ . In this case, return quickselect ( $S_1, k$ ). If  $k = 1 + |S_1|$ , 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  $S_2$ , and it is the  $(k - |S_1| - 1)$ st smallest element in  $S_2$ . We make a recursive call and return quickselect ( $S_2, k - |S_1| - 1$ ).

(如果  $k \leq |S_1|$ , 那么第  $k$  个最小元素必然在  $S_1$  中。在这种情况下, 返回 quickselect ( $S_1, k$ )。如果  $k = 1 + |S_1|$ , 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  $S_2$  中, 即  $S_2$  中的第  $(k - |S_1| - 1)$  (多谢王洋提醒修正) 个最小元素, 我们递归调用并返回 quickselect ( $S_2, k - |S_1| - 1$ ) )。

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  $S_1$  and  $S_2$  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 kth smallest element is in position k. This destroys the original ordering; if this is not desirable, then a copy must be made.

并给出了代码示例：

```
//copyright@ mark allen weiss
//July、updated, 2011.05.05 凌晨.

//q_select places the kth smallest element in a[k]
void q_select( input_type a[], int k, int left, int right )
{
    int i, j;
    input_type pivot;
    if( left + CUTOFF <= right )
    {
        pivot = median3( a, left, right );
        //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。
        //稍后，除了下文的第二节的随机选取枢纽元，在第四节末，
        //您将看到另一种选取枢纽元的方法。
        i=left; j=right-1;
        for(;;)
        {
            while( a[++i] < pivot );
            while( a[--j] > pivot );
            if (i < j)
                swap( &a[i], &a[j] );
            else
                break;
        }
        swap( &a[i], &a[right-1] ); /* restore pivot
    */
    if ( k < i)
        q_select( a, k, left, i-1 );
    else
        if ( k > i )
            q-select( a, k, i+1, right );
}
```

```

    }
    else
        insert_sort(a,  left,  right );
}

```

### 结论：

1. 与快速排序相比，快速选择只做了一次递归调用而不是两次。快速选择的最坏情况和快速排序的相同，也是  $O(N^2)$ ，最坏情况发生在枢纽元的选取不当，以致  $S_1$ ，或  $S_2$  中有一个序列为空。
2. 这就好比快速排序的运行时间与划分是否对称有关，划分的好或对称，那么快速排序可达最佳的运行时间  $O(n \log n)$ ，划分的不好或不对称，则会有最坏的运行时间为  $O(N^2)$ 。而枢纽元的选取则完全决定快速排序的 partition 过程是否划分对称。
3. 快速选择也是一样，如果枢纽元的选取不当，则依然会有最坏的运行时间为  $O(N^2)$  的情况发生。那么，怎么避免这个最坏情况的发生，或者说就算是最坏情况下，亦能保证快速选择的运行时间为  $O(N)$  列?对了，关键，还是看你的枢纽元怎么选取。
4. 像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，在本文第四节末，及本文文末，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。

至于编程之美上所述：从数组中随机选取一个数  $X$ ，把数组划分为  $S_a$  和  $S_b$  俩部分，那么这个问题就转到了下文第二节 RANDOMIZED-SELECT，以线性期望时间做选择，无论如何，编程之美的解法二的复杂度为  $O(n \log k)$  都是有待商榷的。至于最坏情况下一种全新的，为  $O(N)$  的快速选择算法，直接[跳转到本文第四节末，或文末部分吧](#)）。

不过，为了公正起见，把编程之美第 141 页上的源码贴出来，由大家来评判：

```

Kbig(S,  k):
    if(k  <=  0):
        return  []
    if(length  S  <=  k):
        return  S
    (Sa,  Sb)  =  Partition(S)
    return  Kbig(Sa,  k).Append(Kbig(Sb,  k  -  length  Sa))

```

```

Partition(S):
    Sa = []           // 初始化为空数组
    Sb = []           // 初始化为空数组
    Swap(s[1], S[Random()%length S])
                    // 随机选择一个数作为分组标准，以
                    // 避免特殊数据下的算法退化，也可
                    // 通过对整个数据进行洗牌预处理
                    // 实现这个目的
    p = S[1]
    for i in [2: length S]:
        S[i] > p ? Sa.Append(S[i]) : Sb.Append(S[i])

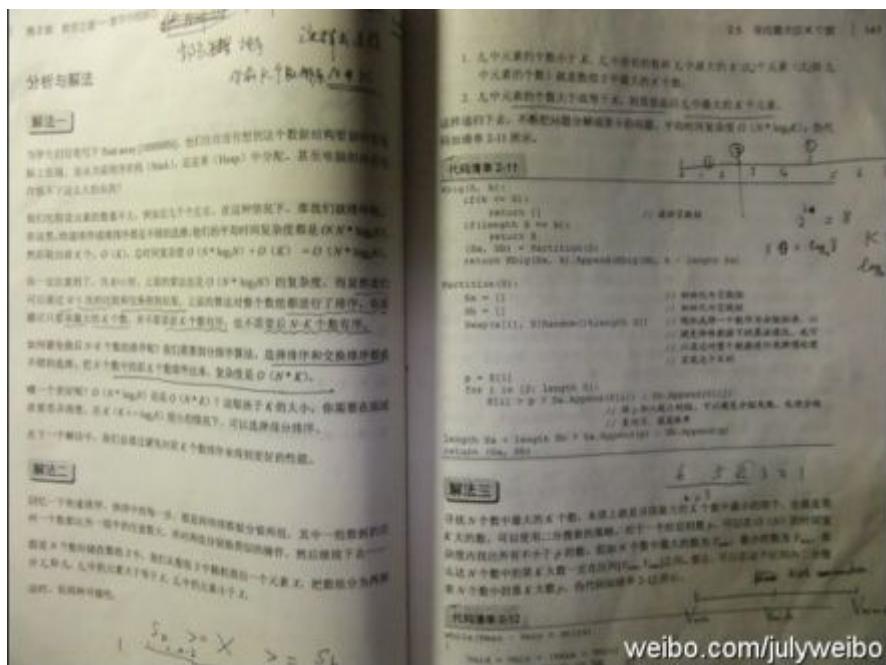
                    // 将 p 加入较小的组，可以避免分组失败，也使分组
                    // 更均匀，提高效率
    length Sa < length Sb ? Sa.Append(p) : Sb.Append(p)
return (Sa, Sb)

```

你已经看到，它是随机选取数组中的任一元素为枢纽的，这就是本文下面的第二节 RANDOMIZED-SELECT 的问题了，只是要修正的是，此算法的平均时间复杂度为线性期望  $O(N)$  的时间。而，稍后在本文的第四节或本文文末，您还将会看到此问题的进一步阐述（SELECT 算法，即快速选择算法），此 SELECT 算法能保证即使在最坏情况下，依然是线性  $O(N)$  的复杂度。

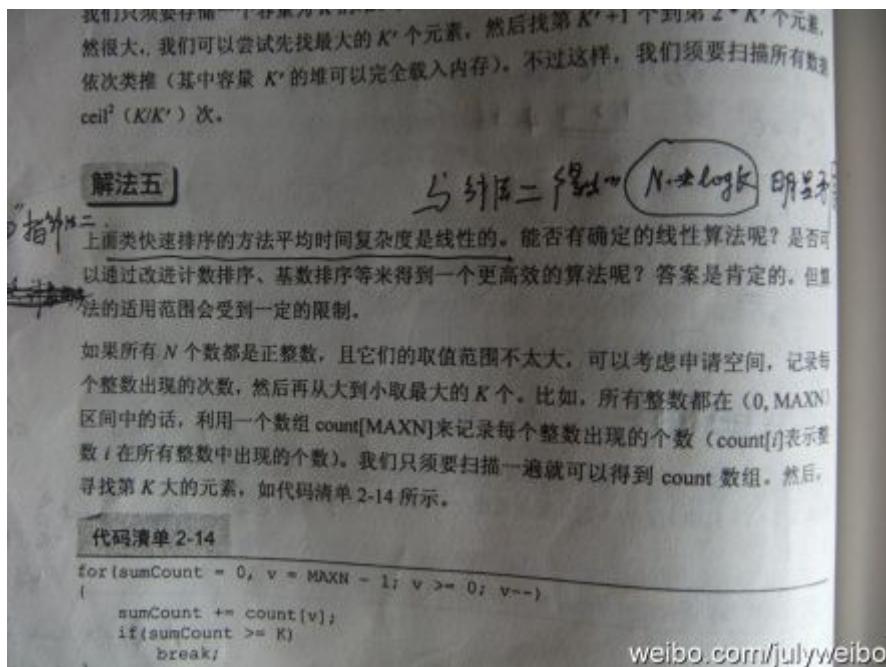
**updated:**

1、为了照顾手中没编程之美这本书的 friends，我拍了张照片，现贴于下供参考（提醒：1、书上为寻找最大的  $k$  个数，而我们面对的问题是寻找最小的  $k$  个数，两种形式，一个本质（该修改的地方，上文已经全部修改）。2、书中描述与上文思路 4 并无原理性出入，不过，勿被图中记的笔记所误导，因为之前也曾被书中的这个  $n*\log k$  复杂度所误导过。ok，相信，看完本文后，你不会再有此疑惑）：



[weibo.com/julyweibo](http://weibo.com/julyweibo)

2、同时，在编程之美原书上此节的解法五的开头提到，“上面类似快速排序的方法平均时间复杂度是线性的”，我想上面的类似快速排序的方法，应该是指解法（即如上所述的类似快速排序 partition 过程的方法），但解法二得出的平均时间复杂度却为  $O(N \log k)$ ，明摆着前后矛盾（参见下图）。



[weibo.com/julyweibo](http://weibo.com/julyweibo)

3、此文创作后的几天，已把本人意见反馈给邹欣等人，下面是编程之美 bop1 的改版修订地址的页面截图（本人也在参加其改版修订的工作），下面的文字是我的记录（同时，本人声明，此狂想曲系列文章系我个人独立创作，与其它的事不相干）：

July 用类似快速排序的partition过程的分治方法，即快选选择算法，平均用时 $O(N)$ ，而非编程之美上所说的 $O(n\log k)$ 。分析请看[http://blog.csdn.net/u\\_JULY/archive/2011/04/28/6370650.aspx](http://blog.csdn.net/u_JULY/archive/2011/04/28/6370650.aspx)

“ok，继续全文，根据选取不同的元素作为主元（枢纽）的情况，可简单总结如下：

1. RANDOMIZED-SELECT：以序列中随机选取一个元素作为主元，可达到线性期望时间 $O(N)$ 的复杂度。
2. SELECT：快速选择算法，以序列中“五划分项中的中项”，或“中位数的中位数”作为主元（枢纽元），则不容易保证的可信度在最坏情况下亦为 $O(N)$ 的复杂度。
3. 本文结论：至此，可以毫无保留的确定此问题之结论：选用类似快速排序的partition的快速选择SELECT算法找出最小的k个元素能做到 $O(N)$ 的复杂度。RANDOMIZED-SELECT可能会有 $O(N^2)$ 的最坏的时间复杂度，但上面的SELECT算法，采用如上所述的“中位数的中位数”的取元方法，则可保证此快速选择算法在最坏情况下是线性时间 $O(N)$ 的复杂度。
4. 关于编程之美第141页，第2章第2.6节“寻找最大的k个数问题的再次讨论”的第三个证据：我再次在编程之初将15章第15.2节找到了，关于SELECT算法能在平均时间 $O(N)$ 内找出第k小元素的第三个证据。同时，该章节上所说，由于SELECT算法采取partition过程划分整个数组元素，所以在找到第k小的元素*x*之后，*x*元素*>x*前面的*k*个元素即为所要查找的*k*个元素。”

以上所有内容皆出自：[http://blog.csdn.net/u\\_JULY/archive/2011/04/28/6370650.aspx](http://blog.csdn.net/u_JULY/archive/2011/04/28/6370650.aspx)。

5、最后一点，在编程之美原书上此节的解法的开头提到：“上面类似快速排序的方法平均时间复杂度是线性的”。我搞上面的类似快速排序的方法，应该是理解法二（即如我上过所述的类似快速排序partition过程的方法）。但解法二得出的平均时间复杂度却为 $O(N^2\log k)$ ，这不是明摆着前后矛盾么？

本人July对博客内任何内容享有版权和著作权，July, updated: 2011.06.07

2.6 精确表达浮点数 ★

[weibo.com/julyweibo](http://weibo.com/julyweibo)

## 第二节、Randomized-Select，线性期望时间

下面是 RANDOMIZED-SELECT(A, p, r) 完整伪码（来自算法导论），我给了注释，或许能给你点启示。在下结论之前，我还需要很多的时间去思量，以确保结论之完整与正确。

```
PARTITION(A, p, r) //partition 过程 p 为第一个数, r 为最后一个数
1   x ← A[r]           //以最后一个元素作为主元
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
```

RANDOMIZED-PARTITION(A, p, r) //随机快排的 partition 过程

```
1   i ← RANDOM(p, r)      //i 随机取 p 到 r 中一个值
2   exchange A[r] <-> A[i] //以随机的 i 作为主元
3   return PARTITION(A, p, r) //调用上述原来的 partition 过程
```

RANDOMIZED-SELECT(A, p, r, i) //以线性时间做选择，目的是返回数组 A[p..r] 中的第 i 小的元素

```
1   if p = r           //p=r, 序列中只有一个元素
```

```

2           then return A[p]
3   q ← RANDOMIZED-PARTITION(A, p, r) //随机选取的元素 q 作为主元
4   k ← q - p + 1           //k 表示子数组 A[p…q] 内的元素个数，处于划分
低区的元素个数加上一个主元元素
5   if i == k //检查要查找的 i 等于子数组中 A[p…q] 中的元素个数 k
6       then return A[q] //则直接返回 A[q]
7   else if i < k
8       then return RANDOMIZED-SELECT(A, p, q - 1, i)
//得到的 k 大于要查找的 i 的大小，则递归到低区间 A[p, q-1] 中去查找
9   else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
//得到的 k 小于要查找的 i 的大小，则递归到高区间 A[q+1, r] 中去查
找。

```

写此文的目的，在于起一个抛砖引玉的作用。希望，能引起你的重视及好的思路，直到有个彻底明白的结果。

updated: 算法导论原英文版有关于 RANDOMIZED-SELECT(A, p, r) 为  $O(n)$  的证明。为了一个彻底明白的阐述，我现将其原文的证明自个再翻译加工后，阐述如下：

此 RANDOMIZED-SELECT 最坏情况下时间复杂度为  $\Theta(n^2)$ ，即使是要选择最小元素也是如此，因为在划分时可能极不走运，总是按余下元素中的最大元素进行划分，而划分操作需要  $O(n)$  的时间。

然而此算法的平均情况性能极好，因为它是随机化的，故没有哪一种特别的输入会导致其最坏情况的发生。

算法导论上，针对此 RANDOMIZED-SELECT 算法平均时间复杂度为  $O(n)$  的证明，引用如下，或许，能给你我多点的启示（本来想直接引用第二版中文版的翻译文字，但在中英文对照阅读的情况下，发现第二版中文版的翻译实在不怎么样，所以，得自己一个一个字的敲，最终敲完修正如下），分 4 步证明：

1、当 RANDOMIZED-SELECT 作用于一个含有  $n$  个元素的输入数组  $A[p \dots r]$  上时，所需时间是一个随机变量，记为  $T(n)$ ，我们可以这样得到线性期望值  $E[T(n)]$  的下界：程序 RANDOMIZED-PARTITION 会以等同的可能性返回数组中任何一个元素为主元，因此，对于每一个  $k$ ， $(1 \leq k \leq n)$ ，子数组  $A[p \dots q]$  有  $k$  个元素，它们全部小于或等于主元元素的概率为  $1/n$ 。对  $k = 1, 2, \dots, n$ ，我们定指示器  $X_k$ ，为：

$$X_k = I\{\text{子数组 } A[p \dots q] \text{ 恰有 } k \text{ 个元素}\},$$

我们假定元素的值不同，因此有

$$E[X_k] = 1/n$$

当调用 RANDOMIZED-SELECT 并且选择  $A[q]$  作为主元元素的时候，我们事先不知道是否会立即找到我们所想要的第  $i$  小的元素，因为，我们很有可能需要在子数组  $A[p \dots q - 1]$ ，或  $A[q + 1 \dots r]$  上递归继续进行寻找。具体在哪一个子数组上递归寻找，视第  $i$  小的元素与  $A[q]$  的相对位置而定。

2、假设  $T(n)$  是单调递增的，我们可以将递归所需时间的界限限定在输入数组时可能输入的所需递归调用的最大时间（此句话，原中文版的翻译也是有问题的）。换言之，我们断定，为得到一个上界，我们假定第  $i$  小的元素总是在划分的较大的一边，对一个给定的 RANDOMIZED-SELECT，指示器  $X_k$  刚好在一个  $k$  值上取 1，在其它的  $k$  值时，都是取 0。当  $X_k = 1$  时，可能要递归处理的两个子数组的大小分别为  $k-1$ ，和  $n-k$ ，因此可得到递归式为

$$\begin{aligned} T(n) &\leq \sum_{k=1}^n X_k \cdot (T(\max(k-1, n-k)) + O(n)) \\ &= \sum_{k=1}^n (X_k \cdot T(\max(k-1, n-k)) + O(n)). \end{aligned}$$

取期望值为：

$$\begin{aligned} E[T(n)] &\leq E\left[\sum_{k=1}^n X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \\ &= \sum_{k=1}^n E[X_k \cdot T(\max(k-1, n-k))] + O(n) \quad (\text{by linearity of expectation}) \\ &= \sum_{k=1}^n E[X_k] \cdot E[T(\max(k-1, n-k))] + O(n) \quad (\text{by equation (C.23)}) \\ &= \sum_{k=1}^n \frac{1}{n} \cdot E[T(\max(k-1, n-k))] + O(n) \quad (\text{by equation (9.1)}). \end{aligned}$$

为了能应用等式 [\(C.23\)](#)，我们依赖于  $X_k$  和  $T(\max(k-1, n-k))$  是独立的随机变量（这个可以证明，证明此处略）。

3、下面，我们来考虑下表达式  $\max(k-1, n-k)$  的结果。我们有：

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil, \\ n-k & \text{if } k \leq \lceil n/2 \rceil. \end{cases}$$

如果  $n$  是偶数，从  $T(1)$  到  $T(n-1)$  每个项在总和中刚好出现两次， $T(j)$  出

现一次。因此，有

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + O(n).$$

我们可以用替换法来解上面的递归式。假设对满足这个递归式初始条件的某个常数  $c$ ，有  $T(n) \leq cn$ 。我们假设对于小于某个常数  $c$ （稍后再来说明如何选取这个常数）的  $n$ ，有  $T(n) = O(1)$ 。同时，还要选择一个常数  $a$ ，使得对于所有的  $n > 0$ ，由上式中  $O(n)$  项（用来描述这个算法的运行时间中非递归的部分）所描述的函数，可由  $an$  从上方限界得到（这里，原中文版的翻译的确是有点含糊）。利用这个归纳假设，可以得到：

（此段原中文版翻译有点问题，上述文字已经修正过来，对应的此段原英文为：We solve the recurrence by substitution. Assume that  $T(n) \leq cn$  for some constant  $c$  that satisfies the initial conditions of the recurrence. We assume that  $T(n) = O(1)$  for  $n$  less than some constant; we shall pick this constant later. 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 from above by  $an$  for all  $n > 0$ . Using this inductive hypothesis, we have）

$$\begin{aligned} E[T(n)] &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\ &= \frac{2c}{n} \left( \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an \\ &= \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\lfloor n/2 \rfloor}{2} \right) + an \\ &\leq \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{(n/2-2)(n/2-1)}{2} \right) + an \\ &= \frac{2c}{n} \left( \frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) + an \\ &= \frac{c}{n} \left( \frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an \\ &= c \left( \frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an \\ &\leq \frac{3cn}{4} + \frac{c}{2} + an \\ &= cn - \left( \frac{cn}{4} - \frac{c}{2} - an \right). \end{aligned}$$

4、为了完成证明，还需要证明对足够大的  $n$ ，上面最后一个表达式最大为  $cn$ ，即要证明： $cn/4 - c/2 - an \geq 0$ . 如果在俩边加上  $c/2$ ，并且提取因子  $n$ ，就可以得到  $n(c/4 - a) \geq c/2$ . 只要我们选择的常数  $c$  能满足  $c/4 - a > 0$ , i. e., 即  $c > 4a$ , 我们就可以将俩边同时除以  $c/4 - a$ , 最终得到：

$$n \geq \frac{c/2}{c/4 - a} = \frac{2c}{c - 4a}.$$

综上，如果假设对  $n < 2c/(c - 4a)$ , 有  $T(n) = O(1)$ ，我们就能得到  $E[T(n)] = O(n)$ 。所以，最终我们可以得出这样的结论，并确认无疑：在平均情况下，任何顺序统计量（特别是中位数）都可以在线性时间内得到。

结论：如你所见，RANDOMIZED-SELECT 有线性期望时间  $O(N)$  的复杂度，但此 RANDOMIZED-SELECT 算法在最坏情况下有  $O(N^2)$  的复杂度。所以，我们得找出一种在最坏情况下也为线性时间的算法。稍后，在本文的第四节末，及本文文末部分，你将看到一种在最坏情况下是线性时间  $O(N)$  的复杂度的快速选择 SELECT 算法。

### 第三节、各执己见，百家争鸣

[updated](#)：本文昨晚发布后，现在朋友们之间，主要有以下几种观点（在彻底弄清之前，最好不要下结论）：

1. [luuillu](#): 我不认为随机快排比直接快排的时间复杂度小。使用快排处理数据前，我们是不知道数据的排列规律的，因此一般情况下，被处理的数据本来就是一组随机数据，对于随机数据再多进行一次随机化处理，数据仍然保持随机性，对排序没有更好的效果。对一组数据采用随选主元的方法，在极端的情况下，也可能出现每次选出的主元恰好是从大到小排列的，此时时间复杂度为  $O(N^2)$ . 当然这个概率极低。随机选主元的好处在于，由于在现实中常常需要把一些数据保存为有序数据，因此，快速排序碰到有序数据的概率就会高一些，使用随机快排可以提高对这些数据的处理效率。这个概率虽然高一些，但仍属于特殊情况，不影响一般情况的时间复杂度。我觉得楼主上面提到的思路 4 和思路 5 的时间复杂度是一样的。
2. 571 楼 得分: 0 Sorehead 回复于: 2011-03-09 16:29:58  
关于第五题：

**Sorehead:** 这两天我总结了一下，有以下方法可以实现：

- 1、第一次遍历取出最小的元素，第二次遍历取出第二小的元素，依次直到第

$k$  次遍历取出第  $k$  小的元素。这种方法最简单，时间复杂度是  $O(k*n)$ 。看上去效率很差，但当  $k$  很小的时候可能是最快的。

2、对这  $n$  个元素进行排序，然后取出前  $k$  个数据即可，可以采用比较普遍的堆排序或者快速排序，时间复杂度是  $O(n*\log n)$ 。这种方法有着很大的弊端，题目并没有要求这最小的  $k$  个数是排好序的，更没有要求对其它数据进行排序，对这些数据进行排序某种程度上来讲完全是一种浪费。而且当  $k=1$  时，时间复杂度依然是  $O(n*\log n)$ 。

3、可以把快速排序改进一下，应该和楼主的 `kth_elem` 一样，这样的好处是不用对所有数据都进行排序。平均时间复杂度应该是  $O(n*\log k)$ 。（[在本文最后一节，你或将看到，复杂度可能应该为  \$O\(n\)\$](#) ）

4、使用我开始讲到的平衡二叉树或红黑树，树只用来保存  $k$  个数据即可，这样遍历所有数据只需要一次。时间复杂度为  $O(n*\log k)$ 。后来我发现这个思路其实可以再改进，使用堆排序中的堆，堆中元素数量为  $k$ ，这样堆中最大元素就是头节点，遍历所有数据时比较次数更少，当然时间复杂度并没有变化。

5、使用计数排序的方法，创建一个数组，以元素值为该数组下标，数组的值为该元素在数组中出现的次数。这样遍历一次就可以得到这个数组，然后查询这个数组就可以得到答案了。时间复杂度为  $O(n)$ 。如果元素值没有重复的，还可以使用位图方式。这种方式有一定局限性，元素必须是正整数，并且取值范围不能太大，否则就造成极大的空间浪费，同时时间复杂度也未必就是  $O(n)$  了。当然可以再次改进，使用一种比较合适的哈希算法来代替元素值直接作为数组下标。

3. **litaoye**: 按照算法导论上所说的，最坏情况下线性时间找第  $k$  大的数。证明一下：把数组中的元素，5 个分为 1 组排序，排序需要进行 7 次比较( $2^7 > 5!$ )，这样需要  $1.4 * n$  次比较，可以完成所有组的排序。取所有组的中位数，形成一个新的数组，有  $n/5$  个元素，5 个分为 1 组排序，重复上面的操作，直到只剩下小于 5 个元素，找出中位数。根据等比数列求和公式，求出整个过程的比较次数： $7/5 + 7/25 + 7/125 + \dots = 7/4$ ，用  $7/4 * n$  次比较可以找出中位数的中位数  $M$ 。能够证明，整个数组中  $> M$  的数超过  $3*n / 10 - 6$ ， $\leq M$  的数超过  $3*n / 10 - 6$ 。以  $M$  为基准，执行上面的 PARTITION，每次至少可以淘汰  $3*n / 10 - 6$ ，约等于  $3/10 * n$  个数，也就是说是用  $(7/4 + 1) * n$  次比较之后，最坏情况下可以让数据量变为原来的  $7/10$ ，同样根据等比

数列求和公式，可以算出最坏情况下找出第  $k$  大的数需要的比较次数， $1 + \frac{7}{10} + \frac{49}{100} + \dots = \frac{10}{3}$ ,  $\frac{10}{3} * \frac{11}{4} * n = \frac{110}{12} * n$ , 也就是说整个过程是  $O(n)$  的，尽管隐含的常数比较大。

总结：关于 RANDOMIZED-SELECT( $A, q + 1, r, i - k$ )，期望运行时间为  $O(n)$  已经没有疑问了，更严格的论证在上面的第二节也已经给出来了。

ok，现在，咱们剩下的问题是，除了此 RANDOMIZED-SELECT( $A, q + 1, r, i - k$ ) 方法（实用价值并不大）和计数排序，都可以做到  $O(n)$  之外，还有类似快速排序的 partition 过程，是否也能做到  $O(n)$ ？

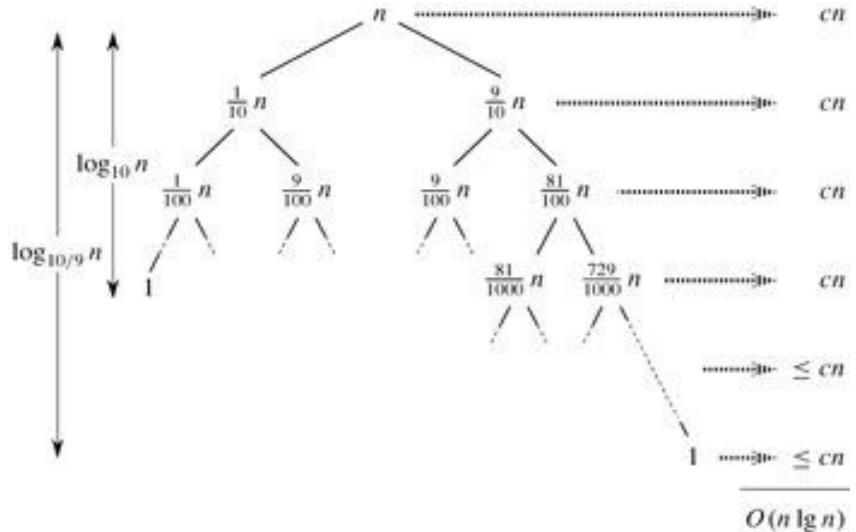
## 第四节、类似 **partition** 过程，最坏亦能做到 **O(n)**？

我想，经过上面的各路好汉的思路轰炸，您的头脑和思维肯定有所混乱了。ok，下面，我尽量以通俗易懂的方式来继续阐述咱们的问题。上面第三节的总结提出了一个问题，即类似快速排序的 partition 过程，是否也能做到  $O(n)$ ？

我们说对  $n$  个数进行排序，快速排序的平均时间复杂度为  $O(n \log n)$ ，这个  $n \log n$  的时间复杂度是如何得来的呢？

经过之前我的有关快速排序的三篇文章，相信您已经明了了以下过程：快速排序每次选取一个主元  $X$ ，依据这个主元  $X$ ，每次把整个序列划分为  $A, B$  两个部分，且有  $Ax < X < Bx$ 。

假如我们每次划分总是产生 9:1 的划分，那么，快速排序运行时间的递归式为： $T(n) = T(9n/10) + T(n/10) + cn$ 。形成的递归树，（注：最后同样能推出  $T(n) = n \log n$ ，即如下图中，每一层的代价为  $cn$ ，共有  $\log n$  层（深度），所以，最后的时间复杂度为  $O(n) \log n$ ）如下：



而我们知道，如果我们每次划分都是平衡的，即每次都划分为均等的两部分元素（对应上图，第一层  $1/2, 1/2, \dots$ ，第二层  $1/4, 1/4, \dots$ ），那么，此时快速排序的运行时间的递归式为：

$$T(n) \leq 2T(n/2) + \Theta(n), \text{ 同样, 可推导出: } T(n) = O(n \lg n).$$

这就是快速排序的平均时间复杂度的由来。

那么，咱们要面对的问题是什么，要寻找  $n$  个数的序列中前  $k$  个元素。如何找列？假设咱们首先第一次对  $n$  个数运用快速排序的 partition 过程划分，主元为  $X_m$ ，此刻找到的主元元素  $X_m$  肯定为序列中第  $m$  小的元素，此后，分为三种情况：

- 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  $X_m$  即可，然后直接输出  $X_m$  前面的  $m-1$  个元素，这  $m$  个元素，即为所求的前  $k$  个最小的元素。
- 2、如果  $m>k$ ，那么接下来要到低区间  $A[0 \dots m-1]$  中寻找，丢掉高区间。
- 3、如果  $m<k$ ，那么接下来要到高区间  $A[m+1 \dots n-1]$  中寻找，丢掉低区间。

当  $m$  一直  $>k$  的时候，好说，区间总是被不断的均分为两个区间（理想情况），那么最后的时间复杂度如 luluillu 所说， $T(n)=n + T(n/2) = n + n/2 + n/4 + n/8 + \dots + 1$ 。式中一共  $\log n$  项。可得出： $T(n)$  为  $O(n \lg n)$ 。

但当  $m < k$  的时候，上述情况，就不好说了。正如 luuillu 所述：当  $m < k$ ，那么接下来要到高区间  $A[m+1 \dots n-1]$  中寻找，新区间的长度为  $n-m-1$ ，需要寻找  $k-m$  个数。此时可令： $k=k-m$ ,  $m=n-m-1$ , 递归调用原算法处理，本次执行次数为  $m$ , 当  $m$  减到 1 算法停止（当  $m < k$  时， $k=m-k$ . 这个判断过程实际上相当于对  $m$  取模运算，即： $k=k \% m;$ ）。

最终在高区间找到的  $k-m$  个数，加上在低区间的  $k$  个数，即可找到最小的  $k$  个数，是否也能得出  $T(n) = O(n)$ ，则还有待验证（本文已经全面更新，所有的论证，都已经给出，确认无误的是：类似快速排序的 partition 过程，明确的可以做到  $O(N)$ ）。

Ok，如果在评论里回复，有诸多不便，欢迎到此帖子上回复：[微软 100 题维护地址](#)，我会随时追踪这个帖子。谢谢。

```
//求取无序数组中第 K 个数，本程序枢纽元的选取有问题，不作推荐。
//copyright@ 飞羽
//July、yansha, updated, 2011.05.18.
#include <iostream>
#include <time.h>
using namespace std;

int kth_elem(int a[], int low, int high, int k)
{
    int pivot = a[low];
    //这个程序之所以做不到 O(N) 的最最重要的原因，就在于这个枢纽元的选取。
    //而这个程序直接选取数组中第一个元素作为枢纽元，是做不到平均时间复杂度为 O(N) 的。

    //要 做到，就必须 把上面选取枢纽元的 代码改掉，要么是随机选择数组中某一元素作为枢纽元，能达到线性期望的时间
    //要么是选取数组中中位数的中位数作为枢纽元，保证最坏情况下，依然为线性 O(N) 的平均时间复杂度。
    int low_temp = low;
    int high_temp = high;
    while (low < high)
    {
        while (low < high && a[high] >= pivot)
            --high;
        a[low] = a[high];
        while (low < high && a[low] < pivot)
            ++low;
    }
}
```

```

        a[high] = a[low];
    }
    a[low] = pivot;

    //以下就是主要思想中所述的内容
    if(low == k - 1)
        return a[low];
    else if(low > k - 1)
        return kth_elem(a, low_temp, low - 1, k);
    else
        return kth_elem(a, low + 1, high_temp, k);
}

int main() //以后尽量不再用随机产生的数组进行测试，没多大必要。
{
    for (int num = 5000; num < 50000001; num *= 10)
    {
        int *array = new int[num];

        int j = num / 10;
        int acc = 0;
        for (int k = 1; k <= num; k += j)
        {
            // 随机生成数据
            srand(unsigned(time(0)));
            for(int i = 0; i < num; i++)
                array[i] = rand() * RAND_MAX + rand();

            //”如果数组本身就是利用随机化产生的话，那么选择其中任何一个元素作为枢轴都可以看作等价于随机选择枢轴，“
            //（虽然这不叫随机选择枢纽）”，这句话，是完全不成立的，是错误的。
        }

        //“因为你总是选择 随机数组中第一个元素 作为枢纽元，不是 随机选择枢纽元”
        //相当于把上面这句话中前面的 “随机” 两字去掉，就是：
        //因为 你总是选择数组中第一个元素作为枢纽元，不是 随机选择枢纽元。
        //所以，这个程序，始终做不到平均时间复杂度为 O(N)。

        //随机数组和给定一个非有序而随机手动输入的数组，是一个道理。稍后，还将就程序的运行结果继续解释这个问题。
        //July、updated, 2011.05.18。
    }
}

```

```

        // 计算一次查找所需的时钟周期数
        clock_t start = clock();
        int data = kth_elem(array, 0, num - 1, k);
        clock_t end = clock();
        acc += (end - start);
    }
    cout << "The average time of searching a date in the
array size of " << num << " is " << acc / 10 << endl;
}
return 0;
}

```

关于上述程序的更多阐述，请参考此文[第三章续、Top K 算法问题的实现中，第一节有关实现三的说明。](#)

updated:

近日，再次在 Mark Allen Weiss 的数据结构与算法分析一书上，第 10 章，第 10.2.3 节看到了关于此分治算法的应用，平均时间复杂度为  $O(N)$  的阐述与证明，可能本文之前的叙述将因此而改写（[July, updated, 2011.05.05](#)）：

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）。

.....

证明从略，更多，请参考 Mark Allen Weiss 的数据结构与算法分析--c 语言描述一书上，第 10 章，第 10.2.3 节。

updated again:

为了给读者一个彻彻底底、明明白白的论证，我还是决定把书上面的整个论证过程全程贴上来，下面，接着上面的内容，然后直接从其中文译本上截两张图来说明好了（更清晰明了）：

现在让我们假设  $N$  可以被 5 整除，因此不存在多余的元素。再设  $N/5$  为奇数，这样  $M$  就包含奇数个元素。我们将要看到，这将提供某种对称性。因此为方便起见我们假设  $N$  为  $10k+5$  的形式。我们还要假设所有的元素都是互异的。实际的算法必须保证能够处理该假设不成立的情况。图 10-36 指出当  $N=45$  时，枢纽元如何能够选出。

在图 10-36 中， $v$  代表该算法选出作为枢纽元的元素。由于  $v$  是 9 个元素的中项，而我们假设所有元素互异，因此必然存在 4 个中项大于  $v$  以及 4 个小于  $v$ 。我们分别用  $L$  和  $S$  表示这些中项。考虑具有一个大中项( $L$  型)的五元素组。该组的中项小于组中的另两个元素且大于组中的另两个元素。我们将令  $H$  代表那些巨型元素。存在一些已知大于一个大中项的元素，类似地， $T$  代表那些小于一个小中项的元素。存在 10 个  $H$  型的元素：具有  $L$  型中项的每组中有两个、 $v$  所在的组中有两个。类似地，存在 10 个  $T$  型元素。

bbs.theithome.com

示这些中项。考虑具有一个大中项( $L$  型)的五元素组。该组的中项小于组中的另两个元素且大于组中的另两个元素。我们将令  $H$  代表那些巨型元素。存在一些已知大于一个大中项的元素，类似地， $T$  代表那些小于一个小中项的元素。存在 10 个  $H$  型的元素：具有  $L$  型中项的每组中有两个、 $v$  所在的组中有两个。类似地，存在 10 个  $T$  型元素。

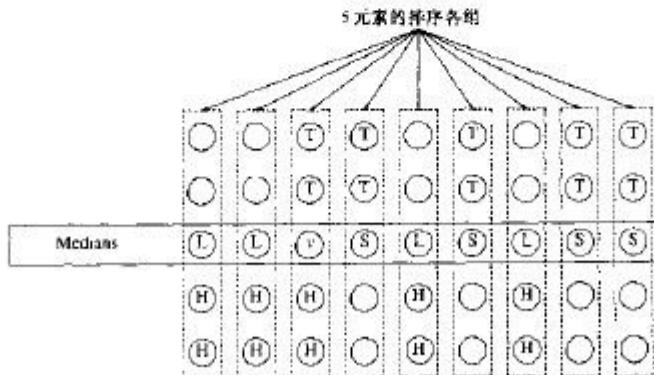


图 10-36 枢纽元的选择

374

$L$  型元素或  $H$  型元素保证大于  $v$ , 而  $S$  型元素或  $T$  型元素保证小于  $v$ 。于是在我们的问题中保证有 14 个大元素和 14 个小元素。因此，递归调用最多可以对  $45 - 14 - 1 = 30$  个元素进行。

让我们把分析推广到对形如  $10k + 5$  的一般的  $N$  的情形。在这种情况下，存在  $k$  个  $L$  型元素和  $k$  个  $S$  型元素。存在  $2k + 2$  个  $H$  型元素，还有  $2k + 2$  个  $T$  型元素。因此，有  $3k + 2$  个元素保证大于  $v$  以及  $3k + 2$  个元素保证小于  $v$ 。于是在这种情况下递归调用最多可以包含  $7k + 2 < 0.7N$  个元素。如果  $N$  不是  $10k + 5$  的形式，类似的论证仍可进行而不影响基本结果。

剩下的问题是确定得到枢纽元的运行时间的界。有两个基本的步骤。我们可以以常数时间找到 5 元素的中项。例如，不难用 8 次比较将 5 个元素排序。我们必须进行  $\lfloor N/5 \rfloor$  次这样的运算，因此这一步花费  $O(N)$  时间。然后我们必须计算  $\lfloor N/5 \rfloor$  元素组的中项。明显的方法是将该组排序并返回中间的元素。但这需要花费  $O(\lfloor N/5 \rfloor \log \lfloor N/5 \rfloor) = O(N \log N)$  的时间，因此不能这么做。解决方法是对这  $\lfloor N/5 \rfloor$  个元素递归调用选择算法。

现在对基本算法的描述已经完成。如果想有一个实际的实现方法，那么还有某些细节仍然需要填补。例如，重复元必须要正确地处理，该算法需要截止点足够大以确保递归调用能够进行。由于涉及到相当大量的系统开销，而且该算法根本不实用，因此我们将不再描述任何细节。即使如此，该算法从理论的角度来看仍然是一种突破，因为其运行时间在最坏情形下是线性的，正如下面的定理所述。

#### 定理 10.9

使用“五分化中项的中项”的快速选择算法的运行时间为  $O(N)$ 。

证明：

该算法由大小为  $0.7N$  和  $0.2N$  的两个递归调用以及线性附加工作组成。根据定理

bbs.theithome.com

284

第 10 章

10.8、其运行时间是线性的。

关于上图提到的定理 10.8，如下图所示，至于证明，留给读者练习（可参考本文第二节关于 RANDOMIZED-SELECT 为线性时间的证明）：

#### 定理 10.8

如果  $\sum_{i=1}^k \alpha_i < 1$ ，则方程  $T(N) = \sum_{i=1}^k T(\alpha_i N) + O(N)$  的解为  $T(N) = O(N)$ 。

ok，第四节，有关此问题的更多论述，请参见下面的[本文文末 updated again 部分](#)。

## 第五节、堆结构实现，处理海量数据

文章，可不能这么完了，咱们还得实现一种靠谱的方案，从整个文章来看，处理这个寻找最小的 k 个数，最好的方案是第一节中所提到的思路 3：当然，更好的办法是维护 k 个元素的最大堆，原理与上述第 2 个方案一致，即用容量为 k 的最大堆存储最小的 k 个数，此时， $k_1 < k_2 < \dots < k_{\max}$  ( $k_{\max}$  设为大顶堆中最大元素)。遍历一次数列，n，每次遍历一个元素 x，与堆顶元素比较， $x < k_{\max}$ ，更新堆（用时  $\log k$ ），否则不更新堆。这样下来，总费时  $O(n * \log k)$ 。

为什么？道理很简单，如果要处理的序列 n 比较小，思路 2（选择排序）的  $n * k$  的复杂度还能说得过去，但当 n 很大的时候呢？同时，别忘了，如果选择思路 1（快速排序），还得在数组中存储 n 个数。当面对海量数据处理的时候呢？n 还能全部存放于电脑内存中么？（或许可以，或许很难）。

ok，相信你已经明白了我的意思，下面，给出借助堆（思路 3）这个数据结构，来寻找最小的 k 个数的完整代码，如下：

```
//借助堆，查找最小的 k 个数
//copyright@ yansha &&July
//July、updated, 2011. 04. 28.

#include <iostream>
#include <assert.h>
using namespace std;
void MaxHeap(int heap[], int i, int len);
/*
BUILD-MIN-HEAP(A)
1   heap-size[A] ← length[A]
2   for i ← ⌊length[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)
*/
// 建立大根堆
void BuildHeap(int heap[], int len)
{
    if (heap == NULL)
        return;

    int index = len / 2;
    for (int i = index; i >= 1; i--)
        MaxHeap(heap, i, len);
}
```

```

PARENT(i)
    return |_i/2_
LEFT(i)
    return 2i
RIGHT(i)
    return 2i + 1
MIN-HEAPIFY(A, i)
1 l ← LEFT(i)
2 r ← RIGHT(i)
3 if l ≤ heap-size[A] and A[l] < A[i]
4     then smallest ← l
5     else smallest ← i
6 if r ≤ heap-size[A] and A[r] < A[smallest]
7     then smallest ← r
8 if smallest ≠ i
9     then exchange A[i] <-> A[smallest]
10            MIN-HEAPIFY(A, smallest)
*/
//调整大根堆
void MaxHeap(int heap[], int i, int len)
{
    int largeIndex = -1;
    int left = i * 2;
    int right = i * 2 + 1;

    if (left <= len && heap[left] > heap[i])
        largeIndex = left;
    else
        largeIndex = i;

    if (right <= len && heap[right] > heap[largeIndex])
        largeIndex = right;

    if (largeIndex != i)
    {
        swap(heap[i], heap[largeIndex]);
        MaxHeap(heap, largeIndex, len);
    }
}
int main()
{
    // 定义数组存储堆元素
    int k;
    cin >> k;
}

```

```

int *heap = new int [k+1];      //注, 只需申请存储 k 个数的数组
FILE *fp = fopen("data.txt", "r"); //从文件导入海量数据（便于
测试, 只截取了 9M 的数据大小）
assert(fp);

for (int i = 1; i <= k; i++)
    fscanf(fp, "%d ", &heap[i]);

BuildHeap(heap, k);           //建堆

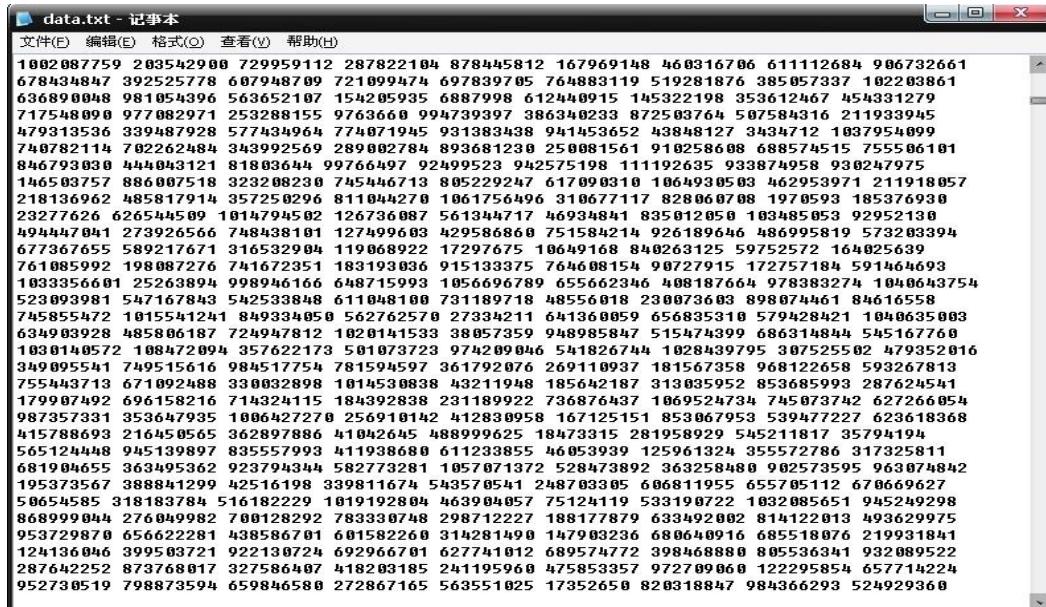
int newData;
while (fscanf(fp, "%d", &newData) != EOF)
{
    if (newData < heap[1])      //如果遇到比堆顶元素 kmax 更小的,
则更新堆
    {
        heap[1] = newData;
        MaxHeap(heap, 1, k);   //调整堆
    }
}

for (int j = 1; j <= k; j++)
    cout << heap[j] << " ";
cout << endl;

fclose(fp);
return 0;
}

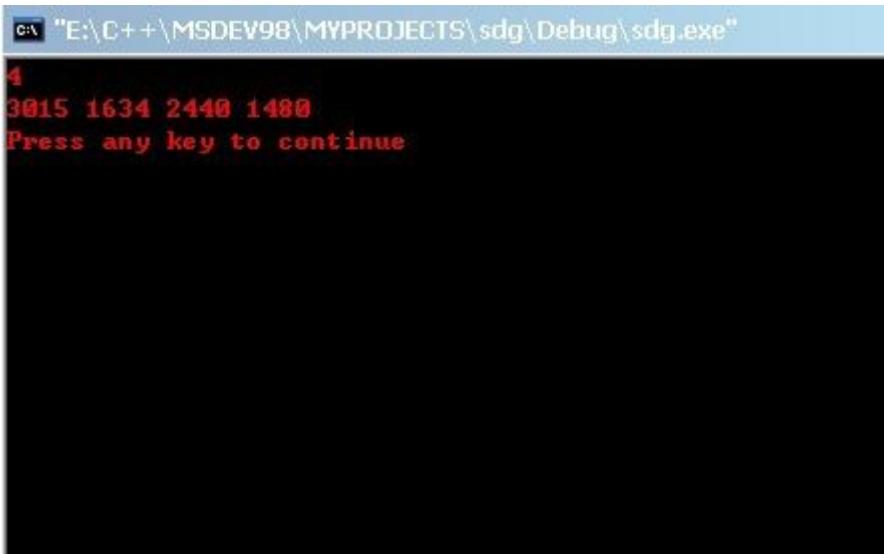
```

咱们用比较大量的数据文件测试一下，如这个数据文件：



The screenshot shows a Windows Notepad window titled "data.txt - 记事本". The window contains a large amount of numerical data, likely integers, listed one per line. The data spans from approximately line 1000 to line 3500. The numbers are mostly four digits long, with some exceptions. The Notepad interface includes standard menu options like File, Edit, Format, View, and Help.

输入 k=4，即要从这大量的数据中寻找最小的 k 个数，可得到运行结果，如下图所示：



The screenshot shows a terminal window with the command prompt "C:\>". The window title is "E:\C++\MSDEV98\MYPROJECTS\sdg\Debug\sdg.exe". The output of the program is displayed in red text:  
4  
3015 1634 2440 1480  
Press any key to continue

至于，这 4 个数，到底是不是上面大量数据中最小的 4 个数，这个，咱们就无从验证了，非人力之所能及也。毕。

## 第六节、stl 之 nth\_element ，逐步实现

以下代码摘自 stl 中 nth\_element 的实现，且逐步追踪了各项操作，其完整代码如下：

```
//_nth_element(...).实现
template <class RandomAccessIterator, class T>
void __nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                    RandomAccessIterator last, T*) {
    while (last - first > 3) {
        RandomAccessIterator cut = __unguarded_partition //下面追踪
__unguarded_partition
        (first, last, T(__median(*first, *(first + (last - first)/2),
                                *(last - 1))));
        if (cut <= nth)
            first = cut;
        else
            last = cut;
    }
    __insertion_sort(first, last); //下面追踪__insertion_sort(first, last)
}

//__unguarded_partition().实现
template <class RandomAccessIterator, class T>
RandomAccessIterator __unguarded_partition(RandomAccessIterator first,
                                            RandomAccessIterator last,
                                            T pivot) {
    while (true) {
        while (*first < pivot) ++first;
        --last;
        while (pivot < *last) --last;
        if (!(first < last)) return first;
        iter_swap(first, last);
        ++first;
    }
}

//__insertion_sort(first, last).实现
template <class RandomAccessIterator>
void __insertion_sort(RandomAccessIterator first, RandomAccessIterator last) {
    if (first == last) return;
    for (RandomAccessIterator i = first + 1; i != last; ++i)
        __linear_insert(first, i, value_type(first)); //下面追踪__linear_insert
```

```

}

//_linear_insert()的实现
template <class RandomAccessIterator, class T>
inline void __linear_insert(RandomAccessIterator first,
                           RandomAccessIterator last, T*) {
    T value = *last;
    if (value < *first) {
        copy_backward(first, last, last + 1); //这个追踪，待续
        *first = value;
    }
    else
        __unguarded_linear_insert(last, value); //最后，再追踪
__unguarded_linear_insert
}

//_unguarded_linear_insert()的实现
template <class RandomAccessIterator, class T>
void __unguarded_linear_insert(RandomAccessIterator last, T value) {
    RandomAccessIterator next = last;
    --next;
    while (value < *next) {
        *last = *next;
        last = next;
        --next;
    }
    *last = value;
}

```

## 第七节、再探 Selection\_algorithm，类似 partition 方法 O(n) 再次求证

网友反馈：

stupidcat：用类似快排的 partition 的方法，只求 2 边中的一边，在 O(N) 时间得到第 k 大的元素 v；

弄完之后，`vector<int> &data` 的前 k 个元素，就是最小的 k 个元素了。时间复杂度是 O(N)，应该是最优的算法了。并给出了代码示例：

```

//copyright@ stupidcat
//July, updated, 2011.05.08
int Partition(vector<int> &data, int headId, int tailId)
//这里，采用的是算法导论上的 partition 过程方法

```

```

{
    int posSlow = headId - 1, posFast = headId;      //一前一后，俩个指针
    for (; posFast < tailId; ++posFast)
    {
        if (data[posFast] < data[tailId])      //以最后一个元素作为主元
        {
            ++posSlow;
            swap(data[posSlow], data[posFast]);
        }
    }
    ++posSlow;
    swap(data[posSlow], data[tailId]);
    return posSlow;      //写的不错，命名清晰
}

void FindKLeast(vector<int> &data, int headId, int tailId, int k)
//寻找第 k 小的元素
{
    if (headId < tailId)
    {
        int midId = Partition(data, headId, tailId);
        //可惜这里，没有随机或中位数的方法选取枢纽元（主元），使得本程序思路虽对，却达不到 O(N) 的目标

        if (midId > k)
        {
            FindKLeast(data, headId, midId - 1, k);      //k < midid, 直接在低区间找
        }

        else
        {
            if (midId < k)
            {
                FindKLeast(data, midId + 1, tailId, k);      //k > midid, 递归到高区间找
            }
        }
    }
}

void FindKLeastNumbers(vector<int> &data, unsigned int k)
{
    int len = data.size();
    if (k > len)
    {
}

```

```

        throw new std::exception("Invalid argument!");
    }
    FindKLeast(data, 0, len - 1, k);
}

看来，这个问题，可能会因此纠缠不清了，近日，在维基百科的英文页面上，  

找到有关 Selection_algorithm 的资料，上面给出的示例代码为：

function partition(list, left, right, pivotIndex)
    pivotValue := list[pivotIndex]
    swap list[pivotIndex] and list[right] // Move pivot to end
    storeIndex := left
    for i from left to right
        if list[i] < pivotValue
            swap list[storeIndex] and list[i]
            increment storeIndex
    swap list[right] and list[storeIndex] // Move pivot to its final place
    return storeIndex

function select(list, left, right, k)
    if left = right
        return list[left]
    select pivotIndex between left and right
    pivotNewIndex := partition(list, left, right, pivotIndex)
    pivotDist := pivotNewIndex - left + 1
    if pivotDist = k
        return list[pivotNewIndex]
    else if k < pivotDist
        return select(list, left, pivotNewIndex - 1, k)
    else
        return select(list, pivotNewIndex + 1, right, k - pivotDist)

```

这个算法，其实就是在本人这篇文章：[当今世界最受人们重视的十大经典算法](#)里提到的：第三名：BFPRT 算法：

A worst-case linear algorithm for the general case of selecting the  $k$ th largest element was published by Blum, Floyd, Pratt, Rivest and Tarjan in their 1973 paper "Time bounds for selection", sometimes called BFPRT after the last names of the authors.

It is based on the quickselect algorithm and is also known as the median-of-medians algorithm.

同时据维基百科上指出，若能选取一个好的 pivot，则此算法能达到  $O(n)$  的最佳时间复杂度。

The median-calculating recursive call does not exceed worst-case linear behavior because the list of medians is 20% of the size of the list, while the other recursive call recurs on at most 70% of the list, making the running time

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

The  $O(n)$  is for the partitioning work (we visited each element a constant number of times, in order to form them into  $O(n)$  groups and take each median in  $O(1)$  time).

From this, one can then show that  $T(n) \leq c*n*(1 + (9/10) + (9/10)^2 + \dots) = O(n)$ .

当然，上面也提到了用堆这个数据结构，扫描一遍数组序列，建  $k$  个元素的堆  $O(k)$  的同时，调整堆 ( $\log k$ )，然后再遍历剩下的  $n-k$  个元素，根据其与堆顶元素的大小比较，决定是否更新堆，更新一次  $\log k$ ，所以，最终的时间复杂度为  $O(k*\log k + (n-k)*\log k) = O(n*\log k)$ 。

Another simple method is to add each element of the list into an ordered set data structure, such as a heap or self-balancing binary search tree, with at most  $k$  elements. Whenever the data structure has more than  $k$  elements, we remove the largest element, which can be done in  $O(\log k)$  time. Each insertion operation also takes  $O(\log k)$  time, resulting in  $O(n \log k)$  time overall.

而如果上述类似快速排序的 partition 过程的 BFPRT 算法成立的话，则将最大限度的优化了此寻找第  $k$  个最小元素的算法复杂度（[经过第 1 节末+第二节+第 4 节末的 updated](#)，以及本节的论证，现最终确定，运用类似快速排序的 partition 算法寻找最小的  $k$  个元素能做到  $O(N)$  的复杂度，并确认无疑。July、updated, 2011.05.05.凌晨）。

updated again:

为了再次佐证上述论证之不可怀疑的准确性，我再原文引用下第九章第 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  $\Theta(n)$  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  $\lceil \frac{n}{5} \rceil$  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  $\lceil \frac{n}{5} \rceil$  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  $\lceil \frac{n}{5} \rceil$  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  $\lceil n/5 \rceil$  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  $\lceil n/5 \rceil$  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  $\lceil n/5 \rceil$  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。

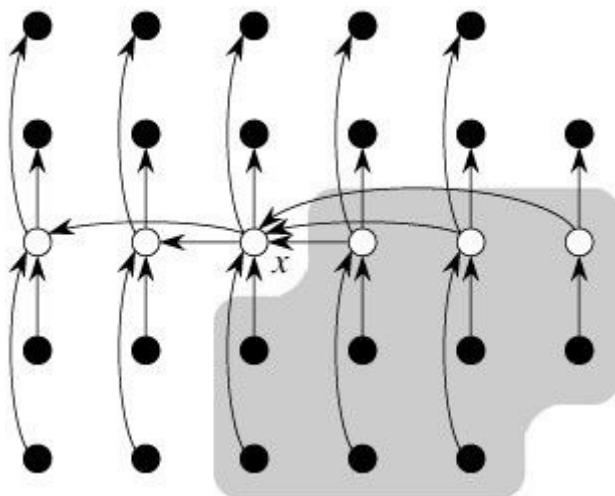
(以上五个步骤, 即本文上面的第四节末中所提到的所谓“五分化中项的中项”的方法。)

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  $\lceil \frac{n}{5} \rceil$  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  $\Theta(n)$  time. (Step 2 consists of  $\Theta(n)$  calls of insertion sort on sets of size  $\Theta(1)$ .) Step 3 takes

time  $T(1)$ , 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  $\mathcal{O}(1)$  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

$$c \lceil n/5 \rceil + c(7n/10 + 6) + an$$

$$\leq cn/5 + c + 7cn/10 + 6c + an$$

$$9cn/10 + 7c + an$$

$$cn + (-cn/10 + 7c + an),$$

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  $\Omega(n \lg n)$  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。……)

ok，综述全文，根据选取不同的元素作为主元（或枢纽）的情况，可简单总结如下：

1、RANDOMIZED-SELECT，以序列中随机选取一个元素作为主元，可达到线性期望时间  $O(N)$  的复杂度。

这个在本文第一节有关编程之美第 2.5 节关于寻找最大的  $k$  个元素（但其  $n \lg k$  的复杂度是严重错误的，待勘误，应以算法导论上的为准，随机选取主元，可达线性期望时间的复杂度），及本文第二节中涉及到的算法导论上第九章第 9.2 节中（以线性期望时间做选择），都是以随机选取数组中任一元素作为枢纽元的。

2、SELECT，快速选择算法，以序列中“五分化中项的中项”，或“中位数的中位数”作为主元（枢纽元），则不容置疑的可保证在最坏情况下亦为  $O(N)$  的复杂度。

这个在本文第四节末，及本文第七节，本文文末中都有所阐述，具体涉及到算法导论一书中第九章第 9.3 节的最快情况线性时间的选择，及 Mark Allen Weiss 所著的数据结构与算法分析--c 语言描述一书的第 10 章第 10.2.3 节（选择问题）中，都有所阐述。

本文结论：至此，可以毫无保留的确定此问题之结论：**运用类似快速排序的 partition 的快速选择 SELECT 算法寻找最小的 k 个元素能做到 O (N) 的复杂度。 RANDOMIZED-SELECT 可能会有 O (N^2) 的最坏的时间复杂度，但上面的 SELECT 算法，采用如上所述的“中位数的中位数”的取元方法，则可保证此快速选择算法在最坏情况下是线性时间 O (N) 的复杂度。**

最终验证：

1、我想，我想，是的，仅仅是我猜想，你可能会有这样的疑问：经过上文大量严谨的论证之后，利用 SELECT 算法，以序列中“五分化中项的中项”，或“中位数的中位数”作为主元（枢纽元），的的确确在最坏情况下 O (N) 的时间复杂度内找到第 k 小的元素，但是，但是，咱们的要面对的问题是什么？咱们是要找最小的 k 个数阿！不是找第 k 小的元素，而是找最小的 k 个数（即不是要你找 1 个数，而是要你找 k 个数）？哈哈，问题提的非常之好阿。

2、事实上，在最坏情况下，能在 O (N) 的时间复杂度内找到第 k 小的元素，那么，亦能保证最坏情况下在 O (N) 的时间复杂度内找到前最小的 k 个数，咱们得找到一个理论依据，即一个证明（我想，等你看到找到前 k 个数的时间复杂度与找第 k 小的元素，最坏情况下，同样是 O (N) 的时间复杂度后，你便会 100% 的相信本文的结论了，然后可以通告全世界，你找到了这个世界上最靠谱的中文算法 blog，ok，这是后话）。

算法导论第 9 章第 9.3 节练习里，有 2 个题目，与我们将要做的证明是一个道理，请看：

#### Exercises 9.3-4: ★

Suppose that an algorithm uses only comparisons to find the  $i^{\text{th}}$  smallest element in a set of  $n$  elements. Show that it can also find the  $i - 1$  smaller elements and the  $n - i$  larger elements without performing any additional comparisons. (假设对一个含有 n 个元素的集合，某算法只需比较来确定第  $i$  小的元素。证明：无需另外的比较操作，它也能找到比  $i$  小的  $i-1$  个元素和比  $i$  大的  $n-i$  个元素)。

### Exercises 9.3-7

Describe an  $O(n)$ -time algorithm that, given a set  $S$  of  $n$  distinct numbers and a positive integer  $k \leq n$ , determines the  $k$  numbers in  $S$  that are closest to the median of  $S$ . (给出一个  $O(N)$  时间的算法，在给定一个有  $n$  个不同数字的集合  $S$  以及一个正整数  $K \leq n$  后，它能确定出  $S$  中最接近其中位数的  $k$  个数。)

怎么样，能证明么？既然通过本文，咱们已经证明了上述的 SELECT 算法在最坏情况下  $O(N)$  的时间内找到第  $k$  小的元素，那么距离咱们确切的问题：寻找最小的  $k$  个数的证明，只差一步之遥了。

#### 给点提示：

1、找到了第  $K$  小的数  $X_k$  为  $O(n)$ ，再遍历一次数组，找出所有比  $k$  小的元素  $O(N)$ （比较  $X_k$  与数组中各数的大小，凡是比  $X_k$  小的元素，都是我们要找的元素），最终时间复杂度即为： $O(N)$ （找到第  $k$  小的元素）+ 遍历整个数组  $O(N) = O(N)$ 。这个结论非常之简单，也无需证明（但是，正如上面的算法导论练习题 9.3-7 所述，能否在找到第  $k$  小的元素后，能否不需要再比较元素列？）。

2、我们的问题是，找到 第  $k$  小的元素后  $X_k$ ，是否  $X_k$  之前的元素就是我们 要找的最小的  $k$  个数，即， $X_k$  前面的数，是否都  $\leq X_k$ ？因为 那样的话，复杂度则变为： $O(N) + O(K)$ （遍历找到的第  $k$  小元素 前面的  $k$  个元素） $= O(N+K) = O(N)$ ，最坏情况下，亦是线性时间。

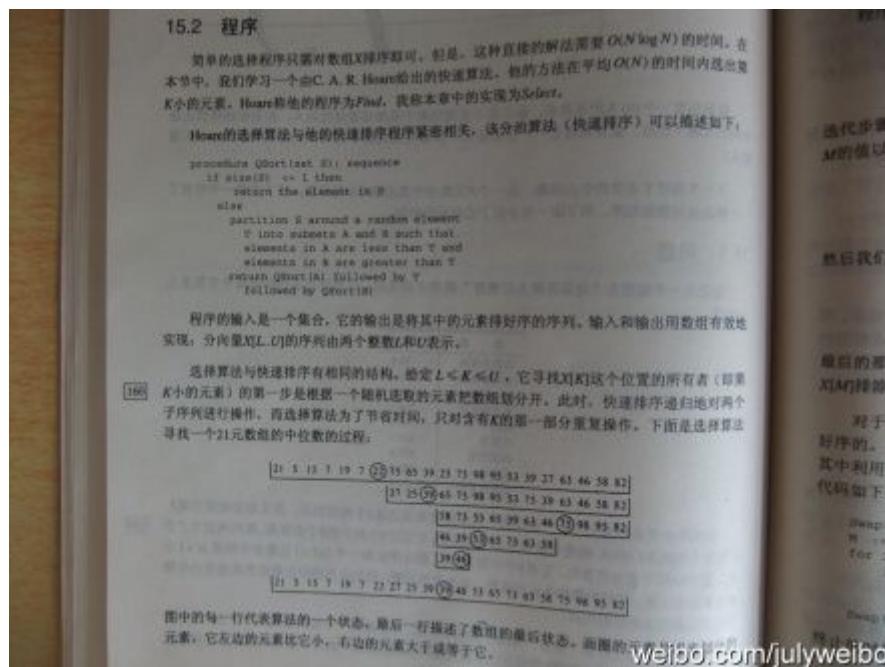
**终极结论：**证明只有一句话：因为本文我们所有的讨论都是基于快速排序的 **partition** 方法，而这个方法，每次划分之后，都保证了 枢纽元  $X_k$  的前边元素统统小于  $X_k$ ，后边元素统统大于  $X_k$ （当然，如果你是属于那种打破沙锅问到底的人，你可能还想要我证明 partition 过程中枢纽元素为何能把整个序列分成左小右大两个部分。但这个不属于本文讨论范畴。读者可参考算法导论第 7 章第 7.1 节关于 partition 过程中循环不变式的证明）。所以，正如本文第一节思路 5 所述在  $O(n)$  的时间内找到第  $k$  小的元素，然后遍历输出前面的  $k$  个小的元素。如此，再次验证了咱们之前得到的结论：运用类似快速排序的 **partition** 的快速选择 SELECT 算法寻找最小的  $k$  个元素，在最坏情况下亦能做到  $O(N)$  的复杂度。

5、RANDOMIZED-SELECT，每次都是随机选取数列中的一个元素作为主元，在 $O(n)$ 的时间内找到第 $k$ 小的元素，然后遍历输出前面的 $k$ 个小的元素。如果能的话，那么总的时间复杂度为线性期望时间： $O(n+k) = O(n)$ （当 $k$ 比较小时）。

所以列，所以，恭喜你，你找到了这个世界上最靠谱的中文算法 blog。

updated:

我假设，你并不认为并赞同上述那句话：你找到了这个世界上最靠谱的中文算法 blog。ok，我再给你一个证据：我再次在编程珠玑 II 上找到了 SELECT 算法能在平均时间  $O(N)$  内找出第  $k$  小元素的第三个证据。同时，依据书上所说，由于 SELECT 算法采取 partition 过程划分整个数组元素，所以在找到第  $k$  小的元素  $X_k$  之后， $X_k+X_k$  前面的  $k$  个元素即为所要查找的  $k$  个元素（下图为编程珠玑 II 第 15 章第 15.2 节的截图，同时各位还可看到，快速排序是递归的对俩个子序列进行操作，而选择算法只对含有  $K$  的那一部分重复操作）。



再多余的话，我不想说了。我知道我的确是一个庸人自扰的 P 民，即没有问题的事情却硬要弄出一堆问题出来，然后再矢志不渝的论证自己的观点不容置疑之正确性。ok，毕。

备注：

- 快速选择 SELECT 算法，虽然复杂度平均是  $O(n)$ ，但这个系数比较大，与用一个最大堆  $O(n \log k)$  不见得就有优势）
  - 当  $K$  很小时， $O(N \log K)$  与  $O(N)$  等价，当  $K$  很大时，当然也就不能忽略掉了。也就是说，在我们这个具体寻找  $k$  个最小的数的问题中，当我们无法确定  $K$  的具体值时（是小是大），咱们便不能简单的从表面上忽略。也就是说： $O(N \log K)$  就是  $O(N \log K)$ ，非  $O(N)$ 。
1. 如果  $n=1024, k=n-1$ , 最差情况下需比较  $2n$  次, 而  $n \log(k-1)=10n$ , 所以不相同。实际上, 这个算法时间复杂度与  $k$  没有直接关系。且只在第一次划分的时候用到了  $K$ , 后面几次划分, 是根据实际情况确定的, 与  $K$  无关了。
  2. 但  $k=n/2$  时也不是  $n \log k$ , 因为只在第一次划分的时候用到了  $K$ , 后面几次划分, 是根据实际情况确定的, 与  $K$  无关了。比如  $a[1001].k=500$ , 第一次把  $a$  划分成两部分,  $b$  和  $c$ , 不妨设  $b$  元素个数为 400 个,  $c$  中元素为 600 个, 则下一步应该舍掉  $a$ , 然后在  $c$  中寻找 top100, 此时  $k$  已经变成了 100, 因此与  $k$  无关。
- 所以, 咱们在表述快速选择算法的平均时间复杂度时, 还是要写成  $O(N)$  的, 断不可写成  $O(N \log K)$  的。

参考文献:

- 1、Mark Allen Weiss 的数据结构与算法分析--c 语言描述, 第 7 章第 7.7.6 节, 线性期望时间的选择算法, 第 10 章第 10.2.3 节, 选择问题
- 2、算法导论, 第九章第 9.2 节, 以线性期望时间做选择, 第九章第 9.3 节, 最快情况线性时间的选择
- 3、编程之美第一版, 第 141 页, 第 2.5 节 寻找最大的  $k$  个数 (找最大或最小, 一个道理)
- 4、维基百科, [http://en.wikipedia.org/wiki/Selection\\_algorithm](http://en.wikipedia.org/wiki/Selection_algorithm)。
- 5、M. Blum, R. W. Floyd, V. Pratt, R. Rivest and R. Tarjan, "Time bounds for selection,"  
J. Comput. System Sci. 7 (1973) 448–461.
- 6、**当今世界最受人们重视的十大经典算法**里提到的, BFPRT 算法。
- 7、编程珠玑 II 第 15 章第 15.2 节程序。顺便大赞此书。July、updated, 2011.05.07。

**预告:** 程序员面试题狂想曲、第四章 ([更多有关海量数据处理, 及 Top K 算法问题 \(此问题已作为第三章续\)](#), [第四章, 择日发布。](#)) , 五月份发布 (近期

内事情较多，且昨夜因修正此文足足熬到了凌晨 4 点（但室内并无海棠花），写一篇文章太耗精力和时间，见谅。有关本人动态，可关注本人微博：  
<http://weibo.com/julyweibo>。谢谢。July、updated, 2011. 05. 05）。

ok，有任何问题，欢迎随时指出。谢谢。完。

## 第三章续、Top K 算法问题的实现

作者: July, zhouzhenren, yansha。

致谢: 微软 100 题实现组, 狂想曲创作组。

时间: 2011 年 05 月 08 日

微博: <http://weibo.com/julyweibo>。

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

wiki: <http://tctop.wikispaces.com/>。

---

### 前奏

在上一篇文章, [程序员面试题狂想曲: 第三章、寻找最小的 k 个数](#)中, 后来为了论证类似快速排序中 `partition` 的方法在最坏情况下, 能在  $O(N)$  的时间复杂度内找到最小的  $k$  个数, 而前前后后 `updated` 了 10 余次。所谓功夫不负苦心人, 终于得到了一个想要的结果。

简单总结如下(详情, 请参考原文第三章) :

1、**RANDOMIZED-SELECT**, 以序列中随机选取一个元素作为主元, 可达到线性期望时间  $O(N)$  的复杂度。

2、**SELECT**, 快速选择算法, 以序列中“五分化中项的中项”, 或“中位数的中位数”作为主元(枢纽元), 则不容置疑的可保证在最坏情况下亦为  $O(N)$  的复杂度。

本章, 咱们来阐述寻找最小的  $k$  个数的反面, 即寻找最大的  $k$  个数, 但此刻可能就有读者质疑了, 寻找最大的  $k$  个数和寻找最小的  $k$  个数, 原理不是一样的么?

是的, 的确是一样, 但这个寻找最大的  $k$  个数的问题的实用范围更广, 因为它牵扯到了一个 **Top K 算法问题**, 以及有关搜索引擎, 海量数据处理等广泛的问题, 所以本文特意对这个 **Top K 算法问题**, 进行阐述以及实现(侧重实现, 因为那样看起来, 会更令人激动人心), 算是第三章的续。**ok**, 有任何问题, 欢迎随时不吝指正。谢谢。

## 说明

关于寻找最小  $K$  个数能做到最坏情况下为  $O(N)$  的算法及证明, 请参考原第三章, [寻找最小的  \$k\$  个数](#), 本文的代码不保证  $O(N)$  的平均时间复杂度, 只是根据第三章有办法可以做到而已(如上面总结的, 2、SELECT, 快速选择算法, 以序列中“五分化中项的中项”, 或“中位数的中位数”作为主元或枢纽元的方法, 原第三章已经严格论证并得到结果)。

## 第一节、寻找最小的第 $k$ 个数

在进入寻找最大的  $k$  个数的主题之前, 先补充下关于寻找最  $k$  小的数的三种简单实现。由于堆的完整实现, 第三章: 第五节, 堆结构实现, 处理海量数据中已经给出, 下面主要给出类似快速排序中 `partition` 过程的代码实现:

**寻找最小的  $k$  个数, 实现一** (下段代码经本文评论下多位读者指出有问题: 当  $a[i]=a[j]=pivot$  时, 则会产生一个无限循环, 在 Mark Allen Weiss 的数据结构与算法分析 C++ 描述中文版的 P209-P210 有描述, 读者可参看之。特此说明, 因本文代码存在问题的地方还有几处, 故请待后续统一修正.2012.08.21) :

```
//copyright@ mark allen weiss && July && yansha
//July, yansha, updated, 2011.05.08.

//本程序, 后经飞羽找出错误, 已经修正。
//随机选取枢纽元, 寻找最小的第 k 个数
#include <iostream>
#include <stdlib.h>
using namespace std;

int my_rand(int low, int high)
{
    int size = high - low + 1;
    return low + rand() % size;
}

//q_select places the kth smallest element in a[k]
int q_select(int a[], int k, int left, int right)
{
    if(k > right || k < left)
    {
        //cout<<"-----"<<endl; //为了处理当 k 大于数组中元素个数的异常情况
    }
}
```

```

    return false;
}

//真正的三数中值作为枢组元方法，关键代码就是下述六行
int midIndex = (left + right) / 2;
if(a[left] < a[midIndex])
    swap(a[left], a[midIndex]);
if(a[right] < a[midIndex])
    swap(a[right], a[midIndex]);
if(a[right] < a[left])
    swap(a[right], a[left]);
swap(a[left], a[right]);

int pivot = a[right]; //之前是 int pivot = right, 特此，修正。

// 申请两个移动指针并初始化
int i = left;
int j = right-1;

// 根据枢组元素的值对数组进行一次划分
for (;;)
{
    while(a[i] < pivot)
        i++;
    while(a[j] > pivot)
        j--;
    //a[i] >= pivot, a[j] <= pivot
    if (i < j)
        swap(a[i], a[j]); //a[i] <= a[j]
    else
        break;
}
swap(a[i], a[right]);

/* 对三种情况进行处理
1、如果 i=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；
2、如果 i>k，那么接下来要到低区间 A[0...m-1]中寻找，丢掉高区间；
3、如果 i<k，那么接下来要到高区间 A[m+1...n-1]中寻找，丢掉低区间。
*/
if (i == k)
    return true;
else if (i > k)
    return q_select(a, k, left, i-1);

```

```

    else return q_select(a, k, i+1, right);
}

int main()
{
    int i;
    int a[] = {7, 8, 9, 54, 6, 4, 11, 1, 2, 33};
    q_select(a, 4, 0, sizeof(a) / sizeof(int) - 1);
    return 0;
}

```

寻找最小的第  $k$  个数，实现二：

```

//copyright@ July
//yansha、updated, 2011.05.08.
// 数组中寻找第  $k$  小元素，实现二
#include <iostream>
using namespace std;

const int numOfArray = 10;

// 这里并非真正随机
int my_rand(int low, int high)
{
    int size = high - low + 1;
    return low + rand() % size;
}

// 以最末元素作为主元对数组进行一次划分
int partition(int array[], int left, int right)
{
    int pos = right;
    for(int index = right - 1; index >= left; index--)
    {
        if(array[index] > array[right])
            swap(array[--pos], array[index]);
    }
    swap(array[pos], array[right]);
    return pos;
}

// 随机快排的 partition 过程
int random_partition(int array[], int left, int right)
{

```

```

// 随机从范围 left 到 right 中取一个值作为主元
int index = my_rand(left, right);
swap(array[right], array[index]);

// 对数组进行划分，并返回主元在数组中的位置
return partition(array, left, right);
}

// 以线性时间返回数组 array[left...right]中第 k 小的元素
int random_select(int array[], int left, int right, int k)
{
    // 处理异常情况
    if (k < 1 || k > (right - left + 1))
        return -1;

    // 主元在数组中的位置
    int pos = random_partition(array, left, right);

    /* 对三种情况进行处理: (m = i - left + 1)
     * 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 array[i] 即可；
     * 2、如果 m>k，那么接下来要到低区间 array[left....pos-1] 中寻找，丢掉高区间；
     * 3、如果 m<k，那么接下来要到高区间 array[pos+1...right] 中寻找，丢掉低区间。
     */
    int m = pos - left + 1;
    if(m == k)
        return array[pos];
    else if (m > k)
        return random_select(array, left, pos - 1, k);
    else
        return random_select(array, pos + 1, right, k - m);
}

int main()
{
    int array[numOfArray] = {7, 8, 9, 54, 6, 4, 2, 1, 12, 33};
    cout << random_select(array, 0, numOfArray - 1, 4) << endl;
    return 0;
}

```

寻找最小的第 k 个数，实现三：

```

//求取无序数组中第 K 个数，本程序枢纽元的选取有问题，不作推荐。
//copyright@ 飞羽
//July、yansha, updated, 2011.05.18.

```

```

#include <iostream>
#include <time.h>
using namespace std;

int kth_elem(int a[], int low, int high, int k)
{
    int pivot = a[low];
    //这个程序之所以做不到 O(N) 的最最重要的原因，就在于这个枢纽元的选取。
    //而这个程序直接选取数组中第一个元素作为枢纽元，是做不到平均时间复杂度为 O(N) 的。

    //要 做到，就必须 把上面选取枢纽元的 代码改掉，要么是随机选择数组中某一元素作为枢纽元，能达到线性期望的时间
    //要么是选取数组中中位数的中位数作为枢纽元，保证最坏情况下，依然为线性 O(N) 的平均时间复杂度。
    int low_temp = low;
    int high_temp = high;
    while(low < high)
    {
        while(low < high && a[high] >= pivot)
            --high;
        a[low] = a[high];
        while(low < high && a[low] < pivot)
            ++low;
        a[high] = a[low];
    }
    a[low] = pivot;

    //以下就是主要思想中所述的内容
    if(low == k - 1)
        return a[low];
    else if(low > k - 1)
        return kth_elem(a, low_temp, low - 1, k);
    else
        return kth_elem(a, low + 1, high_temp, k);
}

int main() //以后尽量不再用随机产生的数组进行测试，没多大必要。
{
    for (int num = 5000; num < 50000001; num *= 10)
    {
        int *array = new int[num];

        int j = num / 10;
        int acc = 0;

```

```

for (int k = 1; k <= num; k += j)
{
    // 随机生成数据
    srand(unsigned(time(0)));
    for(int i = 0; i < num; i++)
        array[i] = rand() * RAND_MAX + rand();
    /*如果数组本身就是利用随机化产生的话，那么选择其中任何一个元素作为枢轴都可以看
    作等价于随机选择枢轴，
    //（虽然这不叫随机选择枢纽）”，这句话，是完全不成立的，是错误的。

    //“因为你总是选择 随机数组中第一个元素 作为枢纽元，不是 随机选择枢纽元”
    //相当于把上面这句话中前面的 “随机” 两字去掉，就是：
    //因为 你总是选择数组中第一个元素作为枢纽元，不是 随机选择枢纽元。
    //所以，这个程序，始终做不到平均时间复杂度为 O (N) 。

//随机数组和给定一个非有序而随机手动输入的数组，是一个道理。稍后，还将就程序的运行
结果继续解释这个问题。
//July、updated, 2011.05.18.

// 计算一次查找所需的时钟周期数
clock_t start = clock();
int data = kth_elem(array, 0, num - 1, k);
clock_t end = clock();
acc += (end - start);
}
cout << "The average time of searching a date in the array size of " << num
<< " is " << acc / 10 << endl;
}
return 0;
}

```

### 测试：

```

The average time of searching a date in the array size of 5000 is 0
The average time of searching a date in the array size of 50000 is 1
The average time of searching a date in the array size of 500000 is 12
The average time of searching a date in the array size of 5000000 is 114
The average time of searching a date in the array size of 50000000 is 1159
Press any key to continue

```

通过测试这个程序，我们竟发现这个程序的运行时间是线性的？

或许，你还没有意识到这个问题，ok，听我慢慢道来。

我们之前说，要保证这个算法是线性的，就一定要在枢纽元的选取上下足功夫：

- 1、要么是随机选取枢纽元作为划分元素
- 2、要么是取中位数的中位数作为枢纽元划分元素

现在，这程序直接选取了数组中第一个元素作为枢纽元

竟然，也能做到线性  $O(N)$  的复杂度，这不是自相矛盾么？

你觉得这个程序的运行时间是线性  $O(N)$ ，是巧合还是确定会是如此？

哈哈，且看 1、@well：根据上面的运行结果不能判断线性，如果人家是  $O(n^{1.1})$  也有可能啊，而且部分数据始终是拟合，还是要数学证明才可靠。2、@July：同时，随机数组中选取一个元素作为枢纽元！  
=> 随机数组中随机选取一个元素作为枢纽元（如果是随机选取随机数组中的一个元素作为主元，那就不同了，跟随机选取数组中一个元素作为枢纽元一样了）。3、@飞羽：正是因为数组本身是随机的，所以选择第一个元素和随机选择其它的数是等价的（由等概率产生保证），这第 3 点，我与飞羽有分歧，至于谁对谁错，待时间让我考证。

关于上面第 3 点我和飞羽的分歧，在我们进一步讨论之后，一致认定（不过，相信，你看到了上面程序更新的注释之后，你应该有几分领会了）：

1. 我们说输入一个数组的元素，不按其顺序输入：如，1,2,3,4,5,6,7，而是这样输入：5,7,6,4,3, 1,2，这就叫随机输入，而这种情况就相当于上述程序主函数中所产生的随机数组。然而选取随机输入的数组或随机数组中第一个元素作为主元，我们不能称之为说是随机选取枢纽元。
2. 因为，随机数产生器产生的数据是随机的，没错，但你要知道，你总是选取随机数组的第一个元素作为枢纽元，这不叫随机选取枢纽元。
3. 所以，上述程序的主函数中随机产生的数组对这个程序的算法而言，没有任何意义，就是帮忙产生了一个随机数组，帮助我们完成了测试，且方便我们测试大数据量而已，就这么简单。
4. 且一般来说，我们看一个程序的时间复杂度，是不考虑其输入情况的，即不考虑主函数，正如这个 `kth number` 的程序所见，你每次都是随机选取数组中第一个元素作为枢纽元，而并不是随机选择枢纽元，所以，做不到平均时间复杂度为  $O(N)$ 。

所以：想要保证此快速选择算法为  $O(N)$  的复杂度，只有两种途径，那就是保证划分的枢纽元元素的选取是：

- 1、随机的（注，此枢纽元随机不等同于数组随机）
- 2、五分化中项的中项，或中位数的中位数。

所以，虽然咱们对于一切心知肚明，但上面程序的运行结果说明不了任何问题，这也从侧面再次佐证了咱们第三章中观点的正确无误性。

**updated:**

非常感谢飞羽等人的工作，将上述三个版本综合到了一起（待进一步测试）：

```

///下面的代码对 July 博客中的三个版本代码进行重新改写。欢迎指出错误。
///先把它们贴在这里，还要进行随机化数据测试。待发...

//modified by 飞羽 at 2011.5.11
////Top_K_test

//修改了下命名规范，July、updated, 2011.05.12。
#include <iostream>
#include <stdlib.h>
using namespace std;

inline int my_rand(int low, int high)
{
    int size = high - low + 1;
    return low + rand() % size;
}

int partition(int array[], int left, int right)
{
    int pivot = array[right];
    int pos = left-1;
    for(int index = left; index < right; index++)
    {
        if(array[index] <= pivot)
            swap(array[++pos], array[index]);
    }
    swap(array[++pos], array[right]);
    return pos;//返回 pivot 所在位置
}

bool median_select(int array[], int left, int right, int k)
{
    //第 k 小元素，实际上应该在数组中下标为 k-1
    if (k-1 > right || k-1 < left)
        return false;

    //真正的三数中值作为枢纽元方法，关键代码就是下述六行
    int midIndex=(left+right)/2;
    if(array[left]<array[midIndex])
        swap(array[left],array[midIndex]);
    if(array[right]<array[midIndex])
        swap(array[right],array[midIndex]);
    if(array[right]<array[left])
        swap(array[right],array[left]);
}

```

```

    swap(array[left], array[right]);

    int pos = partition(array, left, right);

    if (pos == k-1)
        return true;
    else if (pos > k-1)
        return median_select(array, left, pos-1, k);
    else return median_select(array, pos+1, right, k);
}

bool rand_select(int array[], int left, int right, int k)
{
    //第 k 小元素，实际上应该在数组中下标为 k-1
    if (k-1 > right || k-1 < left)
        return false;

    //随机从数组中选取枢纽元元素
    int Index = my_rand(left, right);
    swap(array[Index], array[right]);

    int pos = partition(array, left, right);

    if (pos == k-1)
        return true;
    else if (pos > k-1)
        return rand_select(array, left, pos-1, k);
    else return rand_select(array, pos+1, right, k);
}

bool kth_select(int array[], int left, int right, int k)
{
    //直接取最原始的划分操作
    if (k-1 > right || k-1 < left)
        return false;

    int pos = partition(array, left, right);
    if(pos == k-1)
        return true;
    else if(pos > k-1)
        return kth_select(array, left, pos-1, k);
    else return kth_select(array, pos+1, right, k);
}

```

```

int main()
{
    int array1[] = {7, 8, 9, 54, 6, 4, 11, 1, 2, 33};
    int array2[] = {7, 8, 9, 54, 6, 4, 11, 1, 2, 33};
    int array3[] = {7, 8, 9, 54, 6, 4, 11, 1, 2, 33};

    int numFromArray = sizeof(array1) / sizeof(int);
    for(int i=0; i<numFromArray; i++)
        printf("%d/t",array1[i]);

    int K = 9;
    bool flag1 = median_select(array1, 0, numFromArray-1, K);
    bool flag2 = rand_select(array2, 0, numFromArray-1, K);
    bool flag3 = kth_select(array3, 0, numFromArray-1, K);
    if(!flag1)
        return 1;
    for(i=0; i<K; i++)
        printf("%d/t",array1[i]);
    printf("/n");

    if(!flag2)
        return 1;
    for(i=0; i<K; i++)
        printf("%d/t",array2[i]);
    printf("/n");

    if(!flag3)
        return 1;
    for(i=0; i<K; i++)
        printf("%d/t",array3[i]);
    printf("/n");

    return 0;
}

```

说明：@飞羽：因为预先设定了 K，经过分割算法后，数组肯定被划分为 array[0...k-1] 和 array[k...length-1]，注意到经过 Select\_K\_Version 操作后，数组是被不断地分割的，使得比 array[k-1] 的元素小的全在左边，题目要求的是最小的 K 个元素，当然也就是 array[0...k-1]，所以输出的结果就是前 k 个最小的数：

7	8	9	54	6	4	11	1	2	33
4	1	2	6	7	8	9	11	33	
7	6	4	1	2	8	9	11	33	

7    8    9    6    4    11    1    2    33

Press any key to continue

(更多, 请参见: 此狂想曲系列 tctop 修订 wiki 页面: <http://tctop.wikispaces.com/>)

## 第二节、寻找最大的 k 个数

把之前第三章的问题, 改几个字, 即成为寻找最大的 k 个数的问题了, 如下所述:

### 查找最大的 k 个元素

题目描述: 输入 n 个整数, 输出其中最大的 k 个。

例如输入 1, 2, 3, 4, 5, 6, 7 和 8 这 8 个数字, 则最大的 4 个数字为 8, 7, 6 和 5。

分析: 由于寻找最大的 k 个数的问题与之前的寻找最小的 k 个数的问题, 本质是一样的, 所以, 这里就简单阐述下思路, ok, 考验你举一反三能力的时间到了:

1、排序, 快速排序。我们知道, 快速排序平均所费时间为  $n \log n$ , 从小到大排序这 n 个数, 然后再遍历序列中后 k 个元素输出, 即可, 总的时间复杂度为  $O(n \log n + k) = O(n \log n)$ 。

2、排序, 选择排序。用选择或交换排序, 即遍历 n 个数, 先把最先遍历到得 k 个数存入大小为 k 的数组之中, 对这 k 个数, 利用选择或交换排序, 找到 k 个数中的最小数  $k_{min}$  ( $k_{min}$  设为 k 个元素的数组中最小元素), 用时  $O(k)$  (你应该知道, 插入或选择排序查找操作需要  $O(k)$  的时间), 后再继续遍历后  $n-k$  个数,  $x$  与  $k_{min}$  比较: 如果  $x > k_{min}$ , 则  $x$  代替  $k_{min}$ , 并再次重新找出 k 个元素的数组中最大元素  $k_{min}'$  (多谢 [jiyeyuran](#) 提醒修正); 如果  $x < k_{min}$ , 则不更新数组。这样, 每次更新或不更新数组的所用的时间为  $O(k)$  或  $O(0)$ , 整趟下来, 总的时间复杂度平均下来为:  $n \cdot O(k) = O(n \cdot k)$ 。

3、维护 k 个元素的最小堆, 原理与上述第 2 个方案一致, 即用容量为 k 的最小堆存储最先遍历到的 k 个数, 并假设它们即是最大的 k 个数, 建堆费时  $O(k)$ , 并调整堆 (费时  $O(\log k)$ ) 后, 有  $k_1 > k_2 > \dots > k_{min}$  ( $k_{min}$  设为小顶堆中最大元素)。继续遍历数列, 每次遍历一个元素  $x$ , 与堆顶元素比较, 若  $x > k_{min}$ , 则更新堆 (用时  $\log k$ ), 否则不更新堆。这样下来, 总费时  $O(k \log k + (n-k) \cdot \log k) = O(n \log k)$ 。此方法得益于在堆中, 查找等各项操作时间复杂度均为  $\log k$  (不然, 就如上述思路 2 所述: 直接用数组也可以找出最大的 k 个元素, 用时  $O(n \cdot k)$ )。

4、按编程之美第 141 页上解法二的所述, 类似快速排序的划分方法, N 个数存储在数组 S 中, 再从数组中随机选取一个数 X, 把数组划分为  $S_a$  和  $S_b$  俩部分,  $S_a \geq X \geq S_b$ , 如果要查找的 k 个元素小于  $S_a$  的元素个数, 则返回  $S_a$  中较大的 k 个元素, 否则返回  $S_a$  中所有的元素 +  $S_b$  中最大的  $k - |S_a|$  个元素。不断递归下去, 把问题分解成更小的问题, 平均时间复杂度为  $O(N)$  (编程之美所述的  $n \log k$  的复杂度有误, 应为  $O(N)$ ), 特此订正。其严格证明, 请参考第三章: [程序员面试题狂想曲: 第三章、寻找最小的 k 个](#)

数、updated 10 次)。

.....

其它的方法，在此不再重复了，同时，寻找最小的  $k$  个数借助堆的实现，代码在上一篇文章第三章已有给出，更多，可参考第三章，只要把最大堆改成最小堆，即可。

## 第三节、Top K 算法问题

### 3.1、搜索引擎热门查询统计

#### 题目描述：

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。

分析：这个问题在之前的这篇文章[十一、从头到尾彻底解析 Hash 表算法](#)里，已经有所解答。方法是：

**第一步**、先对这批海量数据预处理，在  $O(N)$  的时间内用 Hash 表完成统计（之前写成了排序，特此订正。July、2011.04.27）；

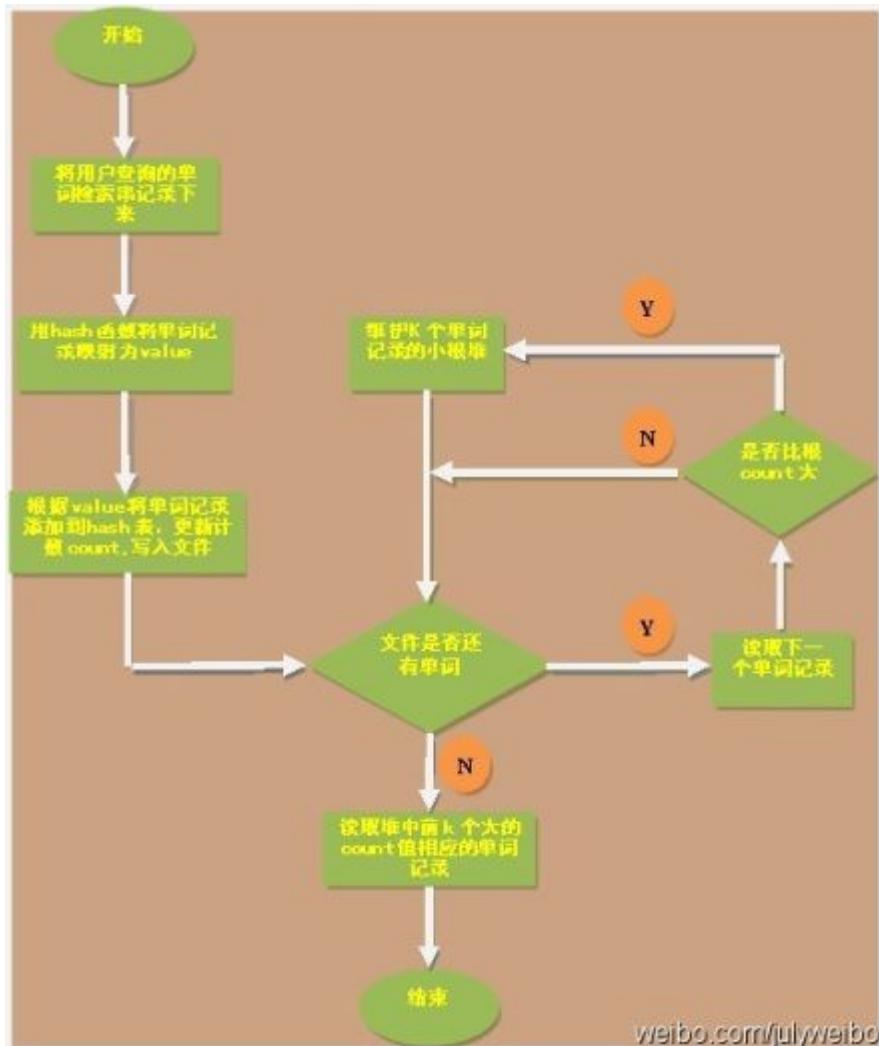
**第二步**、借助堆这个数据结构，找出 Top K，时间复杂度为  $N \log K$ 。

即，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，维护一个  $K$ (该题目中是 10)大小的小根堆 ( $K_1 > K_2 > \dots > K_{\min}$ ,  $K_{\min}$  设为堆顶元素)，然后遍历 300 万的 Query，分别和根元素  $K_{\min}$  进行对比比较（如上第 2 节思路 3 所述，若  $X > K_{\min}$ ，则更新并调整堆，否则，不更新），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)。ok，更多，详情，请参考原文。

或者：采用 trie 树，关键字域存该查询串出现的次数，没有出现为 0。最后用 10 个元素的最小堆来对出现频率进行排序。

ok，本章里，咱们来实现这个问题，为了降低实现上的难度，假设这些记录全部是一些英文单词，即用户在搜索框里敲入一个英文单词，然后查询搜索结果，最后，要你统计输入单词中频率最大的前  $K$  个单词。ok，复杂问题简单化了之后，编写代码实现也相对轻松多

了，画的简单示意图（绘制者， yansha），如下：



完整源码：

```
//copyright@yansha &&July  
//July、updated, 2011.05.08  
  
//题目描述：  
//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。  
  
#include <iostream>  
#include <string>
```

```
#include <assert.h>
using namespace std;

#define HASHLEN 2807303
#define WORDLEN 30

// 结点指针
typedef struct node_no_space *ptr_no_space;
typedef struct node_has_space *ptr_has_space;
ptr_no_space head[HASHLEN];

struct node_no_space
{
    char *word;
    int count;
    ptr_no_space next;
};

struct node_has_space
{
    char word[WORDLEN];
    int count;
    ptr_has_space next;
};

// 最简单 hash 函数
int hash_function(char const *p)
{
    int value = 0;
    while (*p != '/0')
    {
        value = value * 31 + *p++;
        if (value > HASHLEN)
            value = value % HASHLEN;
    }
    return value;
}

// 添加单词到 hash 表
void append_word(char const *str)
{
    int index = hash_function(str);
    ptr_no_space p = head[index];
    while (p != NULL)
```

```

{
    if (strcmp(str, p->word) == 0)
    {
        (p->count)++;
        return;
    }
    p = p->next;
}

// 新建一个结点
ptr_no_space q = new node_no_space;
q->count = 1;
q->word = new char [strlen(str)+1];
strcpy(q->word, str);
q->next = head[index];
head[index] = q;
}

// 将单词处理结果写入文件
void write_to_file()
{
    FILE *fp = fopen("result.txt", "w");
    assert(fp);

    int i = 0;
    while (i < HASHLEN)
    {
        for (ptr_no_space p = head[i]; p != NULL; p = p->next)
            fprintf(fp, "%s %d\n", p->word, p->count);
        i++;
    }
    fclose(fp);
}

// 从上往下筛选，保持小根堆
void sift_down(node_has_space heap[], int i, int len)
{
    int min_index = -1;
    int left = 2 * i;
    int right = 2 * i + 1;

    if (left <= len && heap[left].count < heap[i].count)
        min_index = left;
}

```

```

    else
        min_index = i;

    if (right <= len && heap[right].count < heap[min_index].count)
        min_index = right;

    if (min_index != i)
    {
        // 交换结点元素
        swap(heap[i].count, heap[min_index].count);

        char buffer[WORDLEN];
        strcpy(buffer, heap[i].word);
        strcpy(heap[i].word, heap[min_index].word);
        strcpy(heap[min_index].word, buffer);

        sift_down(heap, min_index, len);
    }
}

// 建立小根堆
void build_min_heap(node_has_space heap[], int len)
{
    if (heap == NULL)
        return;

    int index = len / 2;
    for (int i = index; i >= 1; i--)
        sift_down(heap, i, len);
}

// 去除字符串前后符号
void handle_symbol(char *str, int n)
{
    while (str[n] < '0' || (str[n] > '9' && str[n] < 'A') || (str[n] > 'Z' && str[n]
< 'a') || str[n] > 'z')
    {
        str[n] = '/0';
        n--;
    }

    while (str[0] < '0' || (str[0] > '9' && str[0] < 'A') || (str[0] > 'Z' && str[0]
< 'a') || str[0] > 'z')
    {
}

```

```

int i = 0;
while (i < n)
{
    str[i] = str[i+1];
    i++;
}
str[i] = '/0';
n--;
}

int main()
{
    char str[WORDLEN];
    for (int i = 0; i < HASHLEN; i++)
        head[i] = NULL;

    // 将字符串用 hash 函数转换成一个整数并统计出现频率
    FILE *fp_passage = fopen("string.txt", "r");
    assert(fp_passage);
    while (fscanf(fp_passage, "%s", str) != EOF)
    {
        int n = strlen(str) - 1;
        if (n > 0)
            handle_symbol(str, n);
        append_word(str);
    }
    fclose(fp_passage);

    // 将统计结果输入文件
    write_to_file();

    int n = 10;
    ptr_has_space heap = new node_has_space [n+1];

    int c;

    FILE *fp_word = fopen("result.txt", "r");
    assert(fp_word);
    for (int j = 1; j <= n; j++)
    {
        fscanf(fp_word, "%s %d", &str, &c);
        heap[j].count = c;
        strcpy(heap[j].word, str);
    }
}

```

```

}

// 建立小根堆
build_min_heap(heap, n);

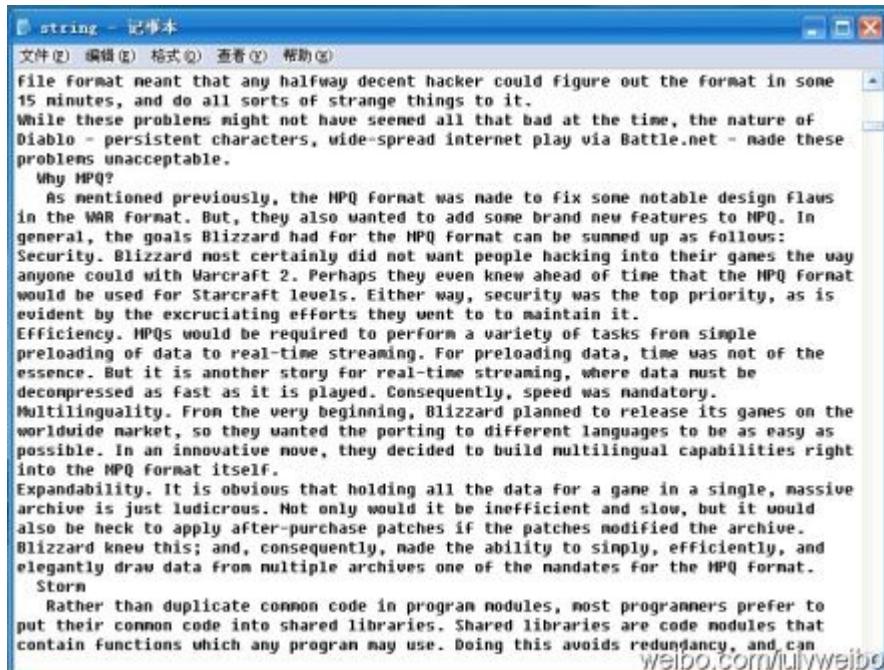
// 查找出现频率最大的 10 个单词
while (fscanf(fp_word, "%s %d", &str, &c) != EOF)
{
    if (c > heap[1].count)
    {
        heap[1].count = c;
        strcpy(heap[1].word, str);
        sift_down(heap, 1, n);
    }
}
fclose(fp_word);

// 输出出现频率最大的单词
for (int k = 1; k <= n; k++)
    cout << heap[k].count << " " << heap[k].word << endl;

return 0;
}

```

**程序测试：**咱们接下来，来对下面的通过用户输入单词后，搜索引擎记录下来，“大量”单词记录进行统计（同时，令 K=10，即要你找出 10 个最热门查询的单词）：



**运行结果：**根据程序的运行结果，可以看到，搜索引擎记录下来的查询次数最多的 10 个单词为（注，并未要求这 10 个数要有序输出）：in (312 次)，it (384 次)，a (432)，that (456)，MPQ (408)，of (504)，and (624)，is (456)，the (1008)，to (936)。

```
312 in
384 it
432 a
456 that
408 MPQ
504 of
624 and
456 is
1008 the
936 to
Press any key to continue...
```

**读者反馈 from 杨忠胜：**3.1 节的代码第 38 行 `hash_function(char const *p)` 有误吧，这样的话，不能修改 `p` 的值（但是函数需要修改指针的值），要想不修改 `*p` 指向的内容，应该是 `const char *p;` 此外，您程序中的 `\t, \n` 有误，C 语言是 `\t,\n`。

感谢这位读者的来信，日后统一订正。谢谢。

## 3.2、统计出现次数最多的数据

### 题目描述：

给你上千万或上亿数据（有重复），统计其中出现次数最多的前 N 个数据。

分析：上千万或上亿的数据，现在的机器的内存应该能存下（也许可以，也许不可以）。所以考虑采用 `hash_map`/搜索二叉树/红黑树等来进行统计次数。然后就是取出前 N 个出现次数最多的数据了。当然，也可以堆实现。

ok，此题与上题类似，最好的方法是用 `hash_map` 统计出现的次数，然后再借用堆找出出现次数最多的 N 个数据。不过，上一题统计搜索引擎最热门的查询已经采用过 `hash` 表统计单词出现的次数，特此，本题咱们改用红黑树取代之前的用 `hash` 表，来完成最初的统计，然后用堆更新，找出出现次数最多的前 N 个数据。

同时，正好个人此前用 c && c++ 语言实现过红黑树，那么，代码能借用就借用吧。

完整代码：

```
//copyright@ zhouchenren &&July
//July、updated, 2011.05.08.

//题目描述:
//上千万或上亿数据（有重复），统计其中出现次数最多的前 N 个数据

//解决方案:
//1、采用红黑树（本程序中有关红黑树的实现代码来源于@July）来进行统计次数。
//2、然后遍历整棵树，同时采用最小堆更新前 N 个出现次数最多的数据。

//声明：版权所有，引用必须注明出处。
#define PARENT(i) (i)/2
#define LEFT(i) 2*(i)
#define RIGHT(i) 2*(i)+1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef enum rb_color{ RED, BLACK }RB_COLOR;
typedef struct rb_node
{
    int key;
    int data;
    RB_COLOR color;
    struct rb_node* left;
    struct rb_node* right;
    struct rb_node* parent;
}RB_NODE;

RB_NODE* RB_CreatNode(int key, int data)
{
    RB_NODE* node = (RB_NODE*)malloc(sizeof(RB_NODE));
    if (NULL == node)
    {
        printf("malloc error!");
        exit(-1);
    }

    node->key = key;
    node->data = data;
```

```

node->color = RED;
node->left = NULL;
node->right = NULL;
node->parent = NULL;

return node;
}

/**
* 左旋
*
*   node           right
*   / / ==>      / /
* a  right      node  y
*   / /          / /
*   b  y      a  b
*/
RB_NODE* RB_RotateLeft(RB_NODE* node, RB_NODE* root)
{
    RB_NODE* right = node->right; // 指定指针指向 right<--node->right

    if ((node->right = right->left))
        right->left->parent = node; // 好比上面的注释图, node 成为 b 的父母

    right->left = node; // node 成为 right 的左孩子

    if ((right->parent = node->parent))
    {
        if (node == node->parent->right)
            node->parent->right = right;
        else
            node->parent->left = right;
    }
    else
        root = right;

    node->parent = right; // right 成为 node 的父母

    return root;
}

/**
* 右旋
*

```

```

*      node          left
*      / /          / /
*      left y    ==>    a   node
*      / /          / /
*      a   b          b   y
*/
RB_NODE* RB_RotateRight(RB_NODE* node, RB_NODE* root)
{
    RB_NODE* left = node->left;

    if ((node->left == left->right))
        left->right->parent = node;

    left->right = node;

    if ((left->parent == node->parent))
    {
        if (node == node->parent->right)
            node->parent->right = left;
        else
            node->parent->left = left;
    }
    else
        root = left;

    node->parent = left;

    return root;
}

/**
* 红黑树的 3 种插入情况
* 用 z 表示当前结点，p[z] 表示父母、p[p[z]] 表示祖父，y 表示叔叔。
*/
RB_NODE* RB_Insert_Rebalance(RB_NODE* node, RB_NODE* root)
{
    RB_NODE *parent, *gparent, *uncle, *tmp; // 父母 p[z]、祖父 p[p[z]]、叔叔 y、临时结点 *tmp

    while ((parent = node->parent) && parent->color == RED)
    { // parent 为 node 的父母，且当父母的颜色为红时
        gparent = parent->parent; // gparent 为祖父

```

```

if (parent == gparent->left) // 当祖父的左孩子即为父母时,其实上述几行语句,无非
就是理顺孩子、父母、祖父的关系。
{
    uncle = gparent->right; // 定义叔叔的概念,叔叔y就是父母的右孩子。
    if (uncle && uncle->color == RED) // 情况 1: z 的叔叔 y 是红色的
    {
        uncle->color = BLACK; // 将叔叔结点 y 着为黑色
        parent->color = BLACK; // z 的父母 p[z] 也着为黑色。解决 z, p[z] 都是红
        的问题。
        gparent->color = RED;
        node = gparent; // 将祖父当做新增结点 z, 指针 z 上移俩层, 且着为红
        色。
        // 上述情况 1 中, 只考虑了 z 作为父母的右孩子的情况。
    }
    else // 情况 2: z 的叔叔 y 是黑色的,
    {
        if (parent->right == node) // 且 z 为右孩子
        {
            root = RB_RotateLeft(parent, root); // 左旋[结点 z, 与父母结
            点]
            tmp = parent;
            parent = node;
            node = tmp; // parent 与 node 互换角色
        }
        // 情况 3: z 的叔叔 y 是黑色的, 此时 z 成为了左孩子。
        // 注意, 1: 情况 3 是由上述情况 2 变化而来的。
        // .....2: z 的叔叔总是黑色的, 否则就是情况 1 了。
        parent->color = BLACK; // z 的父母 p[z] 着为黑色
        gparent->color = RED; // 原祖父结点着为红色
        root = RB_RotateRight(gparent, root); // 右旋[结点 z, 与祖父结点]
    }
}

else
{
    // 这部分是特别为情况 1 中, z 作为左孩子情况, 而写的。
    uncle = gparent->left; // 祖父的左孩子作为叔叔结点。[原理还是与上部分一样
    的]
    if (uncle && uncle->color == RED) // 情况 1: z 的叔叔 y 是红色的
    {
        uncle->color = BLACK;
        parent->color = BLACK;
        gparent->color = RED;
        node = gparent; // 同上
    }
}

```

```

    }

    else // 情况 2: z 的叔叔 y 是黑色的,
    {

        if (parent->left == node) // 且 z 为左孩子
        {
            root = RB_RotateRight(parent, root); // 以结点 parent、root 右
            旋
            tmp = parent;
            parent = node;
            node = tmp; // parent 与 node 互换角色
        }

        // 经过情况 2 的变化, 成为了情况 3.
        parent->color = BLACK;
        gparent->color = RED;
        root = RB_RotateLeft(gparent, root); // 以结点 gparent 和 root 左
        旋
    }

}

return root; // 返回根结点。
}

/***
* 红黑树查找结点
* rb_search_auxiliary: 查找
* rb_node_t* rb_search: 返回找到的结点
*/
RB_NODE* RB_SearchAuxiliary(int key, RB_NODE* root, RB_NODE** save)
{
    RB_NODE* node = root;
    RB_NODE* parent = NULL;
    int ret;

    while (node)
    {
        parent = node;
        ret = node->key - key;
        if (0 < ret)
            node = node->left;
        else if (0 > ret)
            node = node->right;
        else

```

```

        return node;
    }

    if (save)
        *save = parent;

    return NULL;
}

/**
* 返回上述 rb_search_auxiliary 查找结果
*/
RB_NODE* RB_Search(int key, RB_NODE* root)
{
    return RB_SearchAuxiliary(key, root, NULL);
}

/**
* 红黑树的插入
*/
RB_NODE* RB_Insert(int key, int data, RB_NODE* root)
{
    RB_NODE* parent = NULL;
    RB_NODE* node = NULL;

    parent = NULL;
    if ((node = RB_SearchAuxiliary(key, root, &parent))) // 调用 RB_SearchAuxiliary
找到插入结点的地方
    {
        node->data++; // 节点已经存在 data 值加 1
        return root;
    }

    node = RB_CreatNode(key, data); // 分配结点
    node->parent = parent;

    if (parent)
    {
        if (parent->key > key)
            parent->left = node;
        else
            parent->right = node;
    }
    else

```

```

{
    root = node;
}

    return RB_Insert_Rebalance(node, root); // 插入结点后，调用 RB_Insert_Rebalance
修复红黑树的性质
}

typedef struct rb_heap
{
    int key;
    int data;
}RB_HEAP;
const int heapSize = 10;
RB_HEAP heap[heapSize+1];

/**
* MAX_HEAPIFY 函数对堆进行更新，使以 i 为根的子树成最大堆
*/
void MIN_HEAPIFY(RB_HEAP* A, const int& size, int i)
{
    int l = LEFT(i);
    int r = RIGHT(i);
    int smallest = i;

    if (l <= size && A[l].data < A[i].data)
        smallest = l;
    if (r <= size && A[r].data < A[smallest].data)
        smallest = r;

    if (smallest != i)
    {
        RB_HEAP tmp = A[i];
        A[i] = A[smallest];
        A[smallest] = tmp;
        MIN_HEAPIFY(A, size, smallest);
    }
}

/**
* BUILD_MINHEAP 函数对数组 A 中的数据建立最小堆
*/
void BUILD_MINHEAP(RB_HEAP* A, const int& size)
{

```

```

    for (int i = size/2; i >= 1; --i)
        MIN_HEAPIFY(A, size, i);
}

/*
3、维护 k 个元素的最小堆，原理与上述第 2 个方案一致，
即用容量为 k 的最小堆存储最先在红黑树中遍历到的 k 个数，并假设它们即是最大的 k 个数，建堆费时 O
(k) ,
然后调整堆（费时 O (logk) ）后，有 k1>k2>...kmin (kmin 设为小顶堆中最小元素) 。
继续中序遍历红黑树，每次遍历一个元素 x，与堆顶元素比较，若 x>kmin，则更新堆（用时 logk），否
则不更新堆。
这样下来，总费时 O (k*logk+ (n-k) *logk) =O (n*logk) 。
此方法得益于在堆中，查找等各项操作时间复杂度均为 logk) 。
*/

```

//中序遍历 RBTree

```

void InOrderTraverse(RB_NODE* node)
{
    if (node == NULL)
    {
        return;
    }
    else
    {
        InOrderTraverse(node->left);
        if (node->data > heap[1].data) // 当前节点 data 大于最小堆的最小元素时，更新堆数
据
        {
            heap[1].data = node->data;
            heap[1].key = node->key;
            MIN_HEAPIFY(heap, heapSize, 1);
        }
        InOrderTraverse(node->right);
    }
}

void RB_Destroy(RB_NODE* node)
{
    if (NULL == node)
    {
        return;
    }
    else

```

```

{
    RB_Destroy(node->left);
    RB_Destroy(node->right);
    free(node);
    node = NULL;
}
}

int main()
{
    RB_NODE* root = NULL;
    RB_NODE* node = NULL;

    // 初始化最小堆
    for (int i = 1; i <= 10; ++i)
    {
        heap[i].key = i;
        heap[i].data = -i;
    }
    BUILD_MINHEAP(heap, heapSize);

    FILE* fp = fopen("data.txt", "r");
    int num;
    while (!feof(fp))
    {
        fscanf(fp, "%d", &num);
        root = RB_Insert(num, 1, root);
    }
    fclose(fp);

    InOrderTraverse(root);      // 递归遍历红黑树
    RB_Destroy(root);

    for (i = 1; i <= 10; ++i)
    {
        printf("%d\t%d\n", heap[i].key, heap[i].data);
    }
    return 0;
}

```

**程序测试:** 咱们来对下面这个小文件进行测试:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
1  
1  
1  
5  
2  
4  
5  
2  
3  
4  
5
```

weibo.com/julyweibo

运行结果：如下图所示，

```
10 1  
9 1  
7 1  
8 1  
6 1  
3 2  
4 3  
5 4  
1 4  
2 4  
Press any key to continue...
```

weibo.com/julyweibo

问题补遗：

ok，由于在遍历红黑树采用的是递归方式比较耗内存，下面给出一个非递归遍历的程序（下述代码若要运行，需贴到上述程序之后，因为其它的代码未变，只是在遍历红黑树的时候，采取非递归遍历而已，同时，主函数的编写也要稍微修改下）：

```
//copyright@ zhouzhenren
```

```

//July、updated, 2011.05.08.

#define STACK_SIZE 1000

typedef struct
{
    RB_NODE** top;                                // 栈的结点定义
    RB_NODE** base;
}*PStack, Stack;

bool InitStack(PStack& st)                      // 初始化栈
{
    st->base = (RB_NODE**)malloc(sizeof(RB_NODE*) * STACK_SIZE);
    if (!st->base)
    {
        printf("InitStack error!");
        exit(1);
    }
    st->top = st->base;
    return true;
}

bool Push(PStack& st, RB_NODE*& e)                // 入栈
{
    if (st->top - st->base >= STACK_SIZE)
        return false;
    *st->top = e;
    st->top++;
    return true;
}

bool Pop(PStack& st, RB_NODE*& e)                  // 出栈
{
    if (st->top == st->base)
    {
        e = NULL;
        return false;
    }
    e = *--st->top;
    return true;
}

bool StackEmpty(PStack& st)                         // 栈是否为空
{
    if (st->base == st->top)
        return true;
}

```

```

    else
        return false;
}

bool InOrderTraverse_Stack(RB_NODE*& T) // 中序遍历
{
    PStack S = (PStack)malloc(sizeof(Stack));
    RB_NODE* P = T;
    InitStack(S);
    while (P != NULL || !StackEmpty(S))
    {
        if (P != NULL)
        {
            Push(S, P);
            P = P->left;
        }
        else
        {
            Pop(S, P);
            if (P->data > heap[1].data) // 当前节点 data 大于最小堆的最小元素时，更新堆
数据
            {
                heap[1].data = P->data;
                heap[1].key = P->key;
                MIN_HEAPIFY(heap, heapSize, 1);
            }
            P = P->right;
        }
    }
    free(S->base);
    S->base = NULL;
    free(S);
    S = NULL;

    return true;
}

bool PostOrderTraverse_Stack(RB_NODE*& T) //后序遍历
{
    PStack S = (PStack)malloc(sizeof(Stack));
    RB_NODE* P = T;
    RB_NODE* Pre = NULL;
    InitStack(S);
    while (P != NULL || !StackEmpty(S))

```

```

{
    if (P != NULL) // 非空直接入栈
    {
        Push(S, P);
        P = P->left;
    }
    else
    {
        Pop(S, P); // 弹出栈顶元素赋值给 P
        if (P->right == NULL || P->right == Pre) // P 的右子树空或是右子树是刚访问
过的
        {
            // 节点，则释放当前节点内存
            free(P);
            Pre = P;
            P = NULL;
        }
        else // 反之，当前节点重新入栈，接着判断右子树
        {
            Push(S, P);
            P = P->right;
        }
    }
    free(S->base);
    S->base = NULL;
    free(S);
    S = NULL;

    return true;
}

//主函数稍微修改如下：
int main()
{
    RB_NODE* root = NULL;
    RB_NODE* node = NULL;

    // 初始化最小堆
    for (int i = 1; i <= 10; ++i)
    {
        heap[i].key = i;
        heap[i].data = -i;
    }
    BUILD_MINHEAP(heap, heapSize);
}

```

```

FILE* fp = fopen("data.txt", "r");
int num;
while (!feof(fp))
{
    fscanf(fp, "%d", &num);
    root = RB_Insert(num, 1, root);
}
fclose(fp);

//若上面的程序后面加上了上述的非递归遍历红黑树的代码，那么以下几行代码，就得修改如下：
//InOrderTraverse(root); //此句去掉（递归遍历树）
InOrderTraverse_Stack(root); // 非递归遍历树

//RB_Destroy(root); //此句去掉（通过递归释放内存）
PostOrderTraverse_Stack(root); // 非递归释放内存

for (i = 1; i <= 10; ++i)
{
    printf("%d\t%d\n", heap[i].key, heap[i].data);
}
return 0;
}

```

### updated:

后来，我们狂想曲创作组中的 3 又用 hash+堆实现了上题，很明显比采用上面的红黑树，整个实现简洁了不少，其完整源码如下：

### 完整源码：

```

//Author: zhouchenren
//Description: 上千万或上亿数据（有重复），统计其中出现次数最多的钱 N 个数据

//Algorithm: 采用 hash_map 来进行统计次数+堆（找出 Top K）。
//July, 2011.05.12。纪念汶川地震三周年，默哀三秒。

#define PARENT(i) (i)/2
#define LEFT(i) 2*(i)
#define RIGHT(i) 2*(i)+1

#define HASHTABLESIZE 2807303
#define HEAPSIZE 10

```

```

#define A 0.6180339887
#define M 16384      //m=2^14

#include <stdio.h>
#include <stdlib.h>

typedef struct hash_node
{
    int data;
    int count;
    struct hash_node* next;
}HASH_NODE;
HASH_NODE* hash_table[HASHTABLESIZE];

HASH_NODE* creat_node(int& data)
{
    HASH_NODE* node = (HASH_NODE*)malloc(sizeof(HASH_NODE));

    if (NULL == node)
    {
        printf("malloc node failed!\n");
        exit(EXIT_FAILURE);
    }

    node->data = data;
    node->count = 1;
    node->next = NULL;
    return node;
}

/**
* hash 函数采用乘法散列法
* h(k)=int(m*(A*k mod 1))
*/
int hash_function(int& key)
{
    double result = A * key;
    return (int)(M * (result - (int)result));
}

void insert(int& data)
{
    int index = hash_function(data);
    HASH_NODE* pnode = hash_table[index];

```

```

while (NULL != pnode)
{
    // 以存在 data, 则 count++
    if (pnode->data == data)
    {
        pnode->count += 1;
        return;
    }
    pnode = pnode->next;
}

// 建立一个新的节点, 在表头插入
pnode = creat_node(data);
pnode->next = hash_table[index];
hash_table[index] = pnode;
}

/***
* destroy_node 释放创建节点产生的所有内存
*/
void destroy_node()
{
    HASH_NODE* p = NULL;
    HASH_NODE* tmp = NULL;
    for (int i = 0; i < HASHTABLESIZE; ++i)
    {
        p = hash_table[i];
        while (NULL != p)
        {
            tmp = p;
            p = p->next;
            free(tmp);
            tmp = NULL;
        }
    }
}

typedef struct min_heap
{
    int count;
    int data;
}MIN_HEAP;
MIN_HEAP heap[HEAPSIZE + 1];

/**

```

```

* min_heapify 函数对堆进行更新，使以 i 为跟的子树成最大堆
*/
void min_heapify(MIN_HEAP* H, const int& size, int i)
{
    int l = LEFT(i);
    int r = RIGHT(i);
    int smallest = i;

    if (l <= size && H[l].count < H[i].count)
        smallest = l;
    if (r <= size && H[r].count < H[smallest].count)
        smallest = r;

    if (smallest != i)
    {
        MIN_HEAP tmp = H[i];
        H[i] = H[smallest];
        H[smallest] = tmp;
        min_heapify(H, size, smallest);
    }
}

/**
* build_min_heap 函数对数组 A 中的数据建立最小堆
*/
void build_min_heap(MIN_HEAP* H, const int& size)
{
    for (int i = size/2; i >= 1; --i)
        min_heapify(H, size, i);
}

/**
* traverse_hashtable 函数遍历整个 hashtable，更新最小堆
*/
void traverse_hashtable()
{
    HASH_NODE* p = NULL;
    for (int i = 0; i < HASHTABLESIZE; ++i)
    {
        p = hash_table[i];
        while (NULL != p)
        {
            // 如果当前节点的数量大于最小堆的最小值，则更新堆
            if (p->count > heap[1].count)
            {

```

```

        heap[1].count = p->count;
        heap[1].data = p->data;
        min_heapify(heap, HEAPSIZE, 1);
    }
    p = p->next;
}
}

int main()
{
    // 初始化最小堆
    for (int i = 1; i <= 10; ++i)
    {
        heap[i].count = -i;
        heap[i].data = i;
    }
    build_min_heap(heap, HEAPSIZE);

    FILE* fp = fopen("data.txt", "r");
    int num;
    while (!feof(fp))
    {
        fscanf(fp, "%d", &num);
        insert(num);
    }
    fclose(fp);

    traverse_hashtable();

    for (i = 1; i <= 10; ++i)
    {
        printf("%d/t%d/n", heap[i].data, heap[i].count);
    }

    return 0;
}

```

**程序测试：**对 65047kb 的数据量文件，进行测试统计（不过，因其数据量实在太大，半天没打开）：



运行结果：如下，

```
ca *D:\Program Files\Microsoft Visual Studio\MyProjects\jjkk\Debug\jjkk.... - □ x
13939 367
28788 368
29322 368
11635 374
6283 368
11086 378
30869 373
5352 375
5878 377
2720 372
Press any key to continue...
```

## 第四节、海量数据处理问题一般总结

关于海量数据处理的问题，一般有 Bloom filter, Hashing, bit-map, 堆, trie 树等方法来处理。更详细的介绍，请查看此文：[十道海量数据处理面试题与十个方法大总结](#)。

### 余音

**反馈：**此文发布后，走进搜索引擎的作者&&深入搜索引擎-海量信息的压缩、索引和查询的译者，梁斌老师，对此文提了点意见，如下：1、首先 TopK 问题，肯定需要有并发的，否则串行搞肯定慢，IO 和计算重叠度不高。其次在 IO 上需要一些技巧，当然可能只是验证算法，在实践中 IO 的提升会非常明显。最后上文的代码可读性虽好，但机器的感觉可能就会差，这样会影响性能。2、同时，TopK 可以看成从地球上选拔 k 个跑的最快的，参加奥林匹克比赛，各个国家自行选拔，各个大洲选拔，层层选拔，最后找出最快的 10 个。发挥多机多核的优势。

**预告：**程序员面试题狂想曲、第四章，本月月底之前发布（尽最大努力）。

### 修订

程序员面试题狂想曲-tctop (the crazy thingking of programers) 的修订 wiki (<http://tctop.wikispaces.com/>) 已于今天建立，我们急切的想得到读者的反馈，意见，建议，以及更好的思路，算法，和代码优化的建议。所以，

- 如果你发现了狂想曲系列中的任何一题，任何一章 (<http://t.cn/hgVPmH>) 中的错误，问题，与漏洞，欢迎告知给我们，我们将感激不尽，同时，免费赠送本 blog 内的全部博文集锦的 CHM 文件 1 期；
- 如果你能对狂想曲系列的创作提供任何建设性意见，或指导，欢迎反馈给我们，并真诚邀请您加入到狂想曲的 wiki 修订工作中；
- 如果你是编程高手，对狂想曲的任何一章有自己更好的思路，或算法，欢迎加入狂想曲的创作组，以为千千万万的读者创造更多的价值，更好的服务。

Ps: 狂想曲 tctop 的 wiki 修订地址为: <http://tctop.wikispaces.com/>。欢迎围观，更欢迎您加入到狂想曲的创作或 wiki 修订中。

### 联系 July

- email, [zhoulei0907@yahoo.cn](mailto:zhoulei0907@yahoo.cn)
- blog, [http://blog.csdn.net/v\\_JULY\\_v](http://blog.csdn.net/v_JULY_v)。
- weibo, <http://weibo.com/julyweibo>。

作者按：有任何问题，或建议，欢迎以上述联系方式 call me，真诚的谢谢各位。

July、狂想曲创作组，二零一一年五月十日。

## 十四、亦第三章再续：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](#)），为何又要旧事重提列？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：

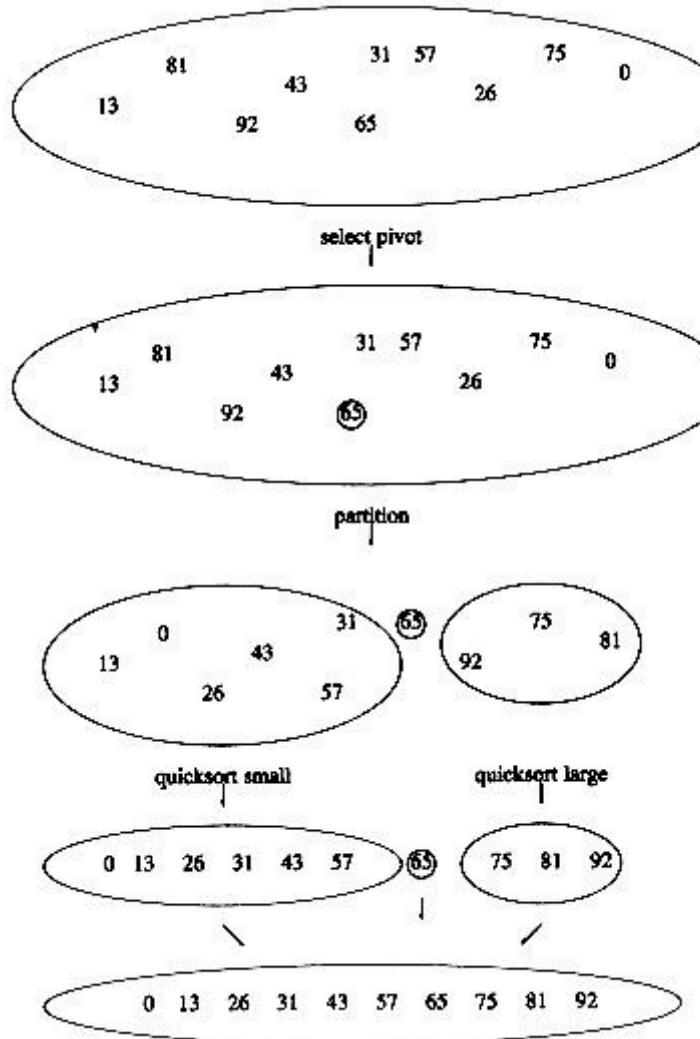
As its name implies, quicksort is the fastest known sorting algorithm in practice. Its average running time is  $O(n \log n)$  (快速排序是实践中已知的最快的排序算法，他的平均运行时间为  $O(N \log N)$ ) . It is very fast, mainly due to a very tight and highly optimized inner loop. It has  $O(n^2)$  worst-case performance (最坏情形的性能为  $O(N^2)$ ) , 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) :  $S_1 = \{x | x \in S \text{ and } x \leq v\}$ , and  $S_2 = \{x | x \in S \text{ and } x > v\}$ .
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、三数选取中值方法

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 $\lfloor \frac{\left| \text{left}+\text{right} \right|}{2} \rfloor$ 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 ( $O(N \log N)$ ) 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  $O(n)+k*O(\log n)$ )。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  $n*\log k$ 。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  $|S| \leq CUTOFF$ , then sort  $S$  and return the  $k$ th smallest element.

2. Pick a pivot element,  $v$  (-  $S$ . (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  $k \leq |S_1|$ , then the  $k$ th smallest element must be in  $S_1$ . In this case, return quickselect ( $S_1, k$ ). If  $k = 1 + |S_1|$ , 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  $S_2$ , and it is the  $(k - |S_1| - 1)$ st smallest element in  $S_2$ . We make a recursive call and return quickselect ( $S_2, k - |S_1| - 1$ ).

(如果  $k \leq |S_1|$ , 那么第  $k$  个最小元素必然在  $S_1$  中。在这种情况下, 返回 quickselect ( $S_1, k$ )。如果  $k = 1 + |S_1|$ , 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  $S_2$  中, 即  $S_2$  中的第  $(k - |S_1| - 1)$  个最小元素, 我们递归调用并返回 quickselect ( $S_2, k - |S_1| - 1$ ) ) (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  $S_1$  and  $S_2$  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“ $0 \dots i$ ”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:

```
//代码实现一  
//copyright@ mark allen weiss  
//July、updated, 2011.05.05 凌晨.
```

```

//三数中值分割法寻找第 k 小的元素的快速选择 SELECT 算法
void q_select( input_type a[], int k, int left, int right )
{
    int i, j;
    input_type pivot;
    if( left /*+ CUTOFF*/ <= right ) //去掉 CUTOFF 常量, 无用
    {
        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
        //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。
        //稍后, 您将看到另一种选取枢纽元的方法。

        i=left; j=right-1;
        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
        {
            while( a[++i] < pivot ){}
            while( a[--j] > pivot ){}
            if ( i < j )
                swap( &a[i], &a[j] );
            else
                break;
        }
        swap( &a[i], &a[right-1] ); /* restore pivot */
        if ( k < i )
            q_select( a, k, left, i-1 );
        else
            if ( k-1 > i ) //此条语句相当于: if(k>i+1)
                q-select( a, k, i+1, right );
        //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,
        //这也是其与第一节中的快速排序算法的本质性区别。

        //2、这个区别则直接决定了: 快速排序算法最快能达到 O (N*logN) ,
        //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。
        //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,
        //即采取稍后的 2.3 节里的五分化中项的中项, 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。
        //后天老爸生日, 孩儿深深祝福。July、updated, 2011.05.19。
    }
    else
        insert_sort(a, left, right-left+1 );
}

```

```

//代码实现二
//copyright @ 飞羽
//July、updated, 2011.05.11。
//三数中值分割法寻找第 k 小的元素
bool median_select(int array[], int left, int right, int k)
{
    //第 k 小元素，实际上应该在数组中下标为 k-1
    if (k-1 > right || k-1 < left)
        return false;

    //三数中值作为枢纽元方法，关键代码就是下述六行：
    int midIndex=(left+right)/2;
    if(array[left]<array[midIndex])
        swap(array[left],array[midIndex]);
    if(array[right]<array[midIndex])
        swap(array[right],array[midIndex]);
    if(array[right]<array[left])
        swap(array[right],array[left]);
    swap(array[midIndex], array[right]);

    int pos = partition(array, left, right);

    if (pos == k-1) //第 k 小元素，实际上应该在数组中下标为 k-1
        return true;
    else if (pos > k-1)
        return median_select(array, left, pos-1, k);
    else return median_select(array, pos+1, right, k);
}

```

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  $\mathcal{O}(n)$  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.

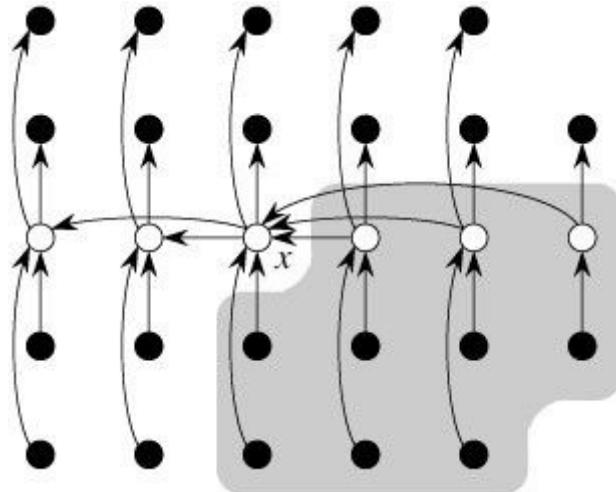
2. Find the median of each of the  $\lceil n/5 \rceil$  groups by first insertion sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
  3. Use SELECT recursively to find the median  $x$  of the  $\lceil n/5 \rceil$  medians 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  $\lceil n/5 \rceil$  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  $\lceil n/5 \rceil$  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  $\lceil n/5 \rceil$  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  $\lceil n/5 \rceil$  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  $\Theta(n)$  time. (Step 2 consists of  $\Theta(n)$  calls of insertion sort on sets of size  $\Theta(1)$ .) 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  $\Theta(1)$  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

$$\begin{aligned} [n/5] + c(7n/10 + 6) + an \\ /5 + c + 7cn/10 + 6c + an \\ n/10 + 7c + an \\ + (-cn/10 + 7c + an), \end{aligned}$$

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  $\Omega(n \lg n)$  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  $\Omega(n \lg n)$  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  $\lceil n/5 \rceil$  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  $\lceil n/5 \rceil$  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  $k-1=i$ ”。换句话就是说，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）：

```
//copyright@ yansha && July && 飞羽
//July、updated, 2011.05.19.清晨。
//版权所有，引用必须注明出处: http://blog.csdn.net/v_JULY_v。
#include <iostream>
#include <time.h>
using namespace std;

const int num_array = 13;
const int num_med_array = num_array / 5 + 1;
int array[num_array];
int median_array[num_med_array];

//冒泡排序（晚些时候将修正为插入排序）
/*void insert_sort(int array[], int left, int loop_times, int compare_times)
```

```

{
    for (int i = 0; i < loop_times; i++)
    {
        for (int j = 0; j < compare_times - i; j++)
        {
            if (array[left + j] > array[left + j + 1])
                swap(array[left + j], array[left + j + 1]);
        }
    }
}*/


/*
//插入排序算法伪代码
INSERTION-SORT(A)                      cost      times
1   for j ← 2 to length[A]              c1          n
2       do key ← A[j]                  c2          n - 1
3           Insert A[j] into the sorted sequence A[1 .. j - 1].    0...n - 1
4           i ← j - 1                  c4          n - 1
5           while i > 0 and A[i] > key      c5
6               do A[i + 1] ← A[i]          c6
7               i ← i - 1                  c7
8           A[i + 1] ← key              c8          n - 1
*/
//已修正为插入排序，如下：
void insert_sort(int array[], int left, int loop_times)
{
    for (int j = left; j < left+loop_times; j++)
    {
        int key = array[j];
        int i = j-1;
        while ( i>left && array[i]>key )
        {
            array[i+1] = array[i];
            i--;
        }
        array[i+1] = key;
    }
}

int find_median(int array[], int left, int right)
{
    if (left == right)
        return array[left];
}

```

```

int index;
for (index = left; index < right - 5; index += 5)
{
    insert_sort(array, index, 4);
    int num = index - left;
    median_array[num / 5] = array[index + 2];
}

// 处理剩余元素
int remain_num = right - index + 1;
if (remain_num > 0)
{
    insert_sort(array, index, remain_num - 1);
    int num = index - left;
    median_array[num / 5] = array[index + remain_num / 2];
}

int elem_aux_array = (right - left) / 5 - 1;
if ((right - left) % 5 != 0)
    elem_aux_array++;

// 如果剩余一个元素返回，否则继续递归
if (elem_aux_array == 0)
    return median_array[0];
else
    return find_median(median_array, 0, elem_aux_array);
}

// 寻找中位数的所在位置
int find_index(int array[], int left, int right, int median)
{
    for (int i = left; i <= right; i++)
    {
        if (array[i] == median)
            return i;
    }
    return -1;
}

int q_select(int array[], int left, int right, int k)
{
    // 寻找中位数的中位数
    int median = find_median(array, left, right);
}

```

```

// 将中位数的中位数与最右元素交换
int index = find_index(array, left, right, median);
swap(array[index], array[right]);

int pivot = array[right];

// 申请两个移动指针并初始化
int i = left;
int j = right - 1;

// 根据数组元素的值对数组进行一次划分
while (true)
{
    while(array[i] < pivot)
        i++;
    while(array[j] > pivot)
        j--;
    if (i < j)
        swap(array[i], array[j]);
    else
        break;
}
swap(array[i], array[right]);

/* 对三种情况进行处理: (m = i - left + 1)
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；
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；
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。
*/
int m = i - left + 1;
if (m == k)
    return array[i];
else if(m > k)
    //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。
    return q_select(array, left, i - 1, k);
else
    return q_select(array, i + 1, right, k - m);
}

int main()
{
    //srand(unsigned(time(NULL)));
    //for (int j = 0; j < num_array; j++)
}

```

```
//array[j] = rand();\n\nint array[num_array]={0,45,78,55,47,4,1,2,7,8,96,36,45};\n// 寻找第 k 最小数\nint k = 4;\nint i = q_select(array, 0, num_array - 1, k);\ncout << i << endl;\n\nreturn 0;\n}
```

# 第三章三续、求数组中给定下标区间内的第 K 小（大）元素

作者：July、上善若水、编程艺术室。

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

## 前奏

原狂想曲系列已更名为：[程序员编程艺术系列](#)。原狂想曲创作组更名为[编程艺术室](#)。编程艺术室致力于以下三点工作：1、针对一个问题，不断寻找更高效的算法，并予以编程实现。2、解决实际中会碰到的应用问题，如第十章、如何给磁盘文件排序。3、经典算法的研究与实现。总体突出一点：编程，如何高效的编程解决实际问题。欢迎有志者加入。

ok，扯远了。在上一章，我们介绍了[第十章、如何给 10^7 个数据量的磁盘文件排序](#)，下面介绍下本章的主题。我们知道，通常来讲，寻找给定区间内的第 k 小（大）的元素的问题是 ACM 中一类常用的数据结构的一个典型例题，即划分树/逆向归并树，通常用线段树的结构存储。

当然这里暂且不表，尚不说划分树思想的神奇，就是线段树的结构，一般没有 ACM 基础的人也都觉得难以理解。所以，这里提供一个时间效率尚可，空间代价还要略小的巧妙解法—伴随数组。

如果看过此前程序员编程艺术：[第六章、求解 500 万以内的亲和数](#)中，有关亲和数的那个题目的伴随数组的解法，也就是利用数组下标作为伴随数组，相信就会对这个方法有一定程度的理解。

## 第一节、寻找给定区间内的第 k 小（大）的元素

给定数组，给定区间，求第 K 小的数如何处理？求最小的 k 个元素用最大堆，求最大的 k 的元素用最小堆。OK，常规方法请查阅：程序员编程艺术：[第三章、寻找最小的 k 个数](#)。

1、排序，快速排序。我们知道，快速排序平均所费时间为  $n \log n$ ，从小到大排序这 n 个数，然后再遍历序列中后 k 个元素输出，即可，总的时间复杂度为  $O(n \log n + k) = O(n \log n)$ 。

2、排序，选择排序。用选择或交换排序，即遍历  $n$  个数，先把最先遍历到得  $k$  个数存入大小为  $k$  的数组之中，对这  $k$  个数，利用选择或交换排序，找到  $k$  个数中的最大数  $kmax$  ( $kmax$  设为  $k$  个元素的数组中最大元素)，用时  $O(k)$  (你应该知道，插入或选择排序查找操作需要  $O(k)$  的时间)，后再继续遍历后  $n-k$  个数， $x$  与  $kmax$  比较：如果  $x < kmax$ ，则  $x$  替代  $kmax$ ，并再次重新找出  $k$  个元素的数组中最大元素  $kmax'$  (多谢 jiyeyuran 提醒修正)；如果  $x > kmax$ ，则不更新数组。这样，每次更新或不更新数组的所用的时间为  $O(k)$  或  $O(0)$ ，整趟下来，总的时间复杂度平均下来为： $n \cdot O(k) = O(n \cdot k)$ 。

3、维护  $k$  个元素的最大堆，原理与上述第 2 个方案一致，即用容量为  $k$  的最大堆存储最先遍历到的  $k$  个数，并假设它们即是最大的  $k$  个数，建堆费时  $O(k)$ ，有  $k_1 < k_2 < \dots < k_{max}$  ( $kmax$  设为最大堆中的最小元素)。继续遍历数列，每次遍历一个元素  $x$ ，与堆顶元素比较，若  $x < kmax$ ，则更新堆（用时  $\log k$ ），否则不更新堆。这样下来，总费时  $O(k + (n-k) \cdot \log k) = O(N \cdot \log k)$ 。此方法得益于在堆中，查找等各项操作时间复杂度均为  $\log k$  (不然，就如上述思路 2 所述：直接用数组也可以找出最大的  $k$  个元素，用时  $O(n \cdot k)$ )。

4、按编程之美上解法二的所述，类似快速排序的划分方法， $N$  个数存储在数组  $S$  中，再从数组中随机选取一个数  $X$ ，把数组划分为  $S_a$  和  $S_b$  俩部分， $S_a <= X <= S_b$ ，如果要查找的  $k$  个元素小于  $S_a$  的元素个数，则返回  $S_a$  中较小的  $k$  个元素，否则返回  $S_a$  中所有的元素 +  $S_b$  中较小的  $k - |S_a|$  个元素。不断递归下去，把问题分解成更小的问题，平均时间复杂度为  $O(N)$  (编程之美所述的  $n \cdot \log k$  的复杂度有误，应为  $O(N)$ ，特此订正。其严格证明，请参考第三章：程序员面试题狂想曲：[第三章、寻找最小的 k 个数](#)、updated 10 次) .....

下面我们给出伴随数组解法，首先，定义一个结构体，一个是数组元素，另一个是数组原来的标号，记录每个数在数组的原顺序。

我们以下面的测试数据举例（红体部分表示下标为 2~5 之间的数 5,2,6,3，浅色部分表示数组中的数各自对应的数组下标，淡蓝色部分为给定的下标区间，注，这里，我们让数组下标从 1 开始）：

```
a[i].data  1 5 2 6 3 7 4  
a[i].num   1 2 3 4 5 6 7
```

现在，题目给定了下标区间，如在原序列中下标 2~5 (即下标为 2、3、4、5) 区间找到第 3 小的数。问题亦相当于要你找原序列里给定下标区间即第 2 个数到第 5 个数之中 (5 2 6 3) 第 3 小的数 (当然，答案很明显，第 3 小的数就是 5)。

那么对原数组进行排序，然后得到的序列应该是（注：原下标始终保持不变）：

```
a[i].data 1 2 3 4 5 6 7  
a[i].num 1 3 5 7 2 4 6
```

如上，既然数据现在已经从小到大排好了，那么，我们只需要进行一次检索，从最小的数到最大的数，我们找第  $k(k=3)$  小的数，当我们发现下标  $a[i].num$  等于原给定下标区间在  $2\sim 5$  中，即  $a[i].num==2 \parallel 3 \parallel 4 \parallel 5$  的时候， $k--$ ，那么当  $k==0$  的时候，我们也就找到了第  $k(3)$  小的数了。如下（红色部分表示原给定下标区间中的数，浅色部分依然是原各数对应的下标，淡蓝色部分为原来给定的下标区间所对应的索引）：

```
a[i].data 1 2 3 4 5 6 7  
a[i].num 1 3 5 7 2 4 6  
k      3 2 1 1 0
```

故下标索引为  $2\sim 5$  之间第  $k(3)$  小的数是 5。

程序的构造与解释：由于排序后，我们能保证原序列已经从小到大的排好序了，所以，当遍历或扫描到原序列给定下标区间中的数时，则  $k--$ ，最终能在  $k==0$  时，找到第  $k$  小的数，且这个数是在原来给定下标区间中的某一个数。

而这个伴随数组，或者说原序列各数的索引则帮我们或者说是帮电脑记下了原来的数，已让我们后来遍历时能识别排序后序列中的数是否是给定下标区间中的某一个数。如果是原给定下标区间中的数，则  $k--$ ，否则  $k$  不变。

## 第二节、采用伴随数组方案的实现

上述采用伴随数组的方法巧妙且简单，也很好理解和实现，关键 就是在于题目要求是在给定下标区间中找寻第  $k$  小（大）的元素，所以，基本上在排序  $n*\log n$  完了之后，总能在  $O(n)$  的时间内找到想找的数。源代码如下：

```
//copyright@ 水 && July  
//总的时间复杂度为 O (N*logN+N) =O (N*logN)。  
//July、updated, 2011.05.28.凌晨。  
  
#include<iostream>  
#include<algorithm>  
using namespace std;  
  
struct node{  
    int num,data;  
    bool operator < (const node &p) const
```

```

{
    return data < p.data;
}
};

node p[100001];

int main()
{
    int n=7;
    int i,j,a,b,c;//c: flag;

    for(i=1;i<=n;i++)
    {
        scanf("%d",&p[i].data);
        p[i].num = i;
    }
    sort(p+1,p+1+n);      //调用库函数 sort 完成排序, 复杂度 n*logn

    scanf("%d %d %d",&a,&b,&c);
    for(i=1;i<=n;i++)    //扫描一遍, 复杂度 n
    {
        if(p[i].num>=a && p[i].num<=b)
            c--;
        if(c == 0)
            break;
    }
    printf("%d\n",p[i].data);
    return 0;
}

```

**程序测试:** 输入的第 1 行数字 1526374 代表给定的数组, 第二行的数字中, 25 代表给定的下标区间 2~5, 3 表示要在给定的下标区间 2~5 中寻找第 3 小的数, 第三行的 5 表示找到的第 3 小的数。程序运行结果如下:

```
1 5 2 6 3 7 4  
2 5 3  
5  
Press any key to continue...
```

水原来写的代码（上面我的改造，是为了达到后来扫描时  $O(N)$  的视觉效果）：

```
//copyright@ 水  
#include<iostream>  
#include<algorithm>  
using namespace std;  
  
struct node{  
    int num,data;  
    bool operator < (const node &p) const  
    {  
        return data < p.data;  
    }  
};  
node p[100001];  
  
int main()  
{  
    int n,m,i,j,a,b,c;//c: flag;  
    while(scanf("%d %d",&n,&m)!=EOF)  
    {  
        for(i=1;i<=n;i++)  
        {  
            scanf("%d",&p[i].data);  
            p[i].num = i;  
        }  
    }  
}
```

```

    }

    sort(p+1,p+1+n);

    for(j=1;j<=m;j++)
    {
        scanf("%d %d %d",&a,&b,&c);
        for(i=1;i<=n;i++)
        {
            if(p[i].num>=a && p[i].num<=b)
                c--;
            if(c == 0)
                break;
        }
        printf("%d\n",p[i].data);
    }
}

return 0;
}

```

### 第三节、直接排序给定下标区间的数

你可能会忽略一个重要的事实，不知读者是否意识到。题目是要求我们在数组中求给定下标区间内某一第  $k$  小的数，即我们只要找到这个第  $k$  小的数，就够了。但上述程序显示的一个弊端，就是它先对整个数组进行了排序，然后采用伴随数组的解法寻找到了第  $k$  小的数。而事实是，我们不需要对整个数组进行排序，我们只需要对我们要寻找的那个数的数组中给定下标区间的数进行部分排序，即可。

对，事情就是这么简单。我们摒弃掉伴随数组的方法，只需要直接对数组中给定的那部分下标区间中的数进行排序，而不是对整个数组进行排序。如此的话，算法的时间复杂度降到了  $L \cdot \log K$ 。其中， $L = |b-a+1|$ ， $L$  为给定下标区间的长度，相对整个数组的程度  $n$ ， $L \leq n$ 。程序代码如下。

```

//copyright@ 苍狼
//直接对给定区间的数进行排序，没必要用伴随数组。
#include<iostream>
#include<algorithm>
using namespace std;

struct node{
    int data;
    bool operator < (const node &p) const
    {

```

```

        return data < p.data;
    }
};

node p[100001];

int main()
{
    int n=7;
    int i,a,b,c;//c: flag;

    for(i=1;i<=n;i++)
    {
        scanf("%d",&p[i].data);
    }

    scanf("%d%d%d", &a, &b, &c); //b, a 为原数组的下标索引
    sort(p+a, p+b+1); //直接对给定区间进行排序, |b-a+1|*log (b-a+1)

    printf("The number is %d/n", p[a-1+c].data);
    return 0;
}

```

**程序测试：**我们同样采取第二节的测试用例。输入的第 1 行数字 1 5 2 6 3 7 4 代表给定的数组，第二行的数字中，2 5 代表给定的下标区间 2~5，3 表示要在给定的下标区间 2~5 中的数，即从 a[2]~a[5] 中寻找第 3 小的数，第三行的 5 表示找到的第 3 小的数。程序运行结果如下。

```

1 5 2 6 3 7 4
2 5 3
The number is 5
Press any key to continue

```

貌似上述直接对给定区间内的数进行排序，效率上较第二节的伴随数组方案更甚一筹。既然如此，那么伴随数组是不是多此一举呢？其实不然，@水：假如，我对 2-5 之间进行了排序，那么数据就被摧毁了，怎么进行 2 次的操作？就是现在的 2 位置已经不是初始的 2 位置的数据了。也就是说，快排之后下标直接定位的方法明显只能用一次。

ok，更多请看下文第四节中的“百家争鸣”与“经典对白”。

## 第四节、伴随数组的优势所在

### 百家争鸣

- @雨翔：伴随数组这种方式确实比较新颖，伴随数组的前提是在排序后的，但总的复杂度还是  $O(N \log N + N) = O(N \log N)$ ，找第  $K$  大的数的此类面试题都是有这几点限制：1、数很多，让你在内存中放不下，2、复杂度严格要求，即不能用排序。当然，即便第三节中，直接对给定下标区间进行排序，复杂度同样为  $L \log L$ ， $L$  为给定区间的长度。事实上，我们在解决“从给定下标区间中的数找寻第  $k$  小（大）的元素”这个问题，还是选择堆为好，在之前的基础上：入堆的时候 只需检测这个元素的下标是否是给定下标区间内的，不是则不入这样的复杂度会低，不需要排序。然后便是平均时间复杂度虽为  $O(N)$ ，但并不常用的快速选择 SELECT 算法，参考：[第三章再续：快速选择 SELECT 算法的深入分析与实现](#)。
- @水：伴随数组的解法是为了达到预处理开销换查找开销目的。直接对给定不同的下标区间的数进行排序，在小数据量处理时复杂度还可以接受，但当面临大数据量，即海量数据处理时，比如 10G 的数据量，每次取 1G 的段的问题，则使用伴随数组的方法会凸显优势，只不过预处理的开销的确是大了点。伴随数组的精髓就是稳定的时间之内解决对相同数据的多次访问查找。说白了，就是同一个数组，要不断查找数组中给定的不同下标区间中的第  $k$  小的数时优势明显。具体，还可以看看这道题：<http://poj.org/problem?id=2104>。
- @July：不用看我了，基本上同意上述水的观点。雨翔之所以认为伴随数组不可取，是因为没有考虑到水提出的问题，即如果要多次或不断的从数组中不同的下标区间中寻找第  $k$  小的数的情况。这时，伴随数组的优势就体现出来了。ok，读者还可以继续看下面的经典对白。相信，你能找到你想要的答案。

### 经典对白

- 查找  $a[0] \sim a[n-1]$  内第  $K$  小，然后再找  $a[1] \sim a[n]$  内第  $K$  小，依次往复，找个几次就优势明显了。其实是比较采取伴随数组解法  $n \log n + m * n$  的代价（ $m$  为给定不同区间的个数）和直接排序  $m * (L * \log L)$  ( $L$  为给定下标区间的长度) 的代价，哪个更低。其中，采用伴随数组查找最差情况是  $n \log n + m(n-1)$ ，而直接排序代价，最差情况为  $m * ((n-1) * \log (n-1))$ 。当  $m >> 0$  且  $n >> 0$  时，排序时间-伴随时间  $= m * n * \log n - n * \log n - mn = (m-1) n * \log n - mn$  恒正，**结论：**即在需要不断的从不同给定下标区间中寻找第  $k$  小数的情况下，当数据规模大的时候伴随数组效果**恒优于每次都直接对给定的下标区间的部分数进行排序。**
- 是的，好比我现在给定不同的另外一个下标区间，要你从中查找第  $k$  小的数，你总不能每次都排序吧。而采取伴随数组的方案的话，由于伴随数组记下了各自给定的下标区间对应的数。所以，第二次在不同的下标区间中查找第  $k$  小的数时，还是只要扫描一遍即可找到，复杂度还是  $O(N)$ 。从而，给定不同的下标区间查找第  $k$  小的数，复杂度为  $m * N$  加上之前排序预处理的复杂度， $N * \log N$ ，总的时间复杂度为  $O(N * \log N + m * N)$  ( $m$  为给定不同区间的个数)。而直接对给定下标区间中的数进行排序的代价则为  $|l_1 * \log l_1 + l_2 * \log l_2 + \dots + l_i * \log l_i|$ 。当  $m >> 0$  且  $n >> 0$  时，哪个复杂度谁大谁小，一眼就看出来了伴随数组所体现的巨大优势。
- 恩，实际样例是这样的，我们有每天超过 100 万次点击的网页，我们常见的来源有  $n$  种，然后，我们要确定每天的每个时段和一周乃至整个月的点击来源地分析。数据库的库存数据量庞大，`copy` 花销很大，内排序花销更大，如果要做出这样的统计图，我擦泪，如果每次都排序，玩死了。

## 原例重现

ok，说了这么多，你可能还根本就不明白到底是怎么一回事。让我们从第一节举的那个例子说起。我们要找给定下标区间 **2~5** 的数中第 **3** 小的数，诚然，此时，我们有两种选择，**1**、如上第一节、第二节所述的伴随数组，**2**、直接对下标区间 **2-5** 的数进行排序。下面，只回顾下伴随数组的方案。

### 伴随数组

`a[i].data 1 5 2 6 3 7 4`

`a[i].num 1 2 3 4 5 6 7`

第一次排序后：

`a [i].data 1 2 3 4 5 6 7`

`a [i].num 1 3 5 7 2 4 6`

伴随数组方案查找：

a[i].data 1 2 3 4 5 6 7

a[i].num 1 3 5 7 2 4 6

k 3 2 1 1 0

好的，那么现在，如果题目要求你在之前数组的下标区间 **3~6** 的数中找第 3 小的数呢（答案很明显，为 6）？

a[i].data 1 5 2 6 3 7 4

a[i].num 1 2 3 4 5 6 7

1. 直接排序么?ok，退万一步讲，假设有的读者可能还是会依然选择直接排序下标 3~6 之间的数。但你是否可曾想到，每次对不同的下标区间所对应的数进行排序，你不但破坏了原有的数据，而且如果区间有覆盖的话，那么将使得我们无法再能依靠原有的直接的下标定位找到原来的数据，且每进行一次排序，都要花费平均时间复杂度为  $N \log N$  的时间开销。如上面的经典对白所述，这样下去的开销将非常大，将为  $I_1 \log I_1 + I_2 \log I_2 + \dots + I_i \log I_i$ 。
2. 那么，如果是采取伴随数组的方法，我们要怎么做呢?如下所示，我们在  $k=0$  的时候，同样找到了第 3 小的数 6，如此是不是只要在之前的一次排序，以后不论是换各种不同的下标区间时都能扫描一遍  $O(N)$  搞定?复杂度为  $O(N \log N + m \cdot N)$  ( $m$  为给定不同的下标区间的区间数)。
3. 由上面的经典对白里面的内容，我们已经知道，当  $m \gg 0$  且  $n \gg 0$  时 ( $m$  为给定不同的下标区间的区间数,  $n$  为数组大小)，排序时间-伴随时间 =  $m \cdot n \cdot \log n - n \cdot \log n - mn = (m-1) n \cdot \log n - mn$  恒正。yeah，相信，你已经明白了。

### 伴随数组

原第一次排序后：

a[i].data 1 2 3 4 5 6 7

a[i].num 1 3 5 7 2 4 6

再次扫描，直接  $O(N)$  搞定：

```
a[i].data 1 2 3 4 5 6 7  
a[i].num 1 3 5 7 2 4 6  
k      3 2 1 1 1 0
```

(而之前有的读者意识到伴随数组的意义，是因为一般的人只考虑找一次，不会想到第二次或多次查找)

## 编程独白

给你 40 分钟的时间，你可以思考十分钟，然后用三十分钟的时间来写代码，最后浪费在无谓的调试上；你也可以思考半个小时，彻底弄清问题的本质与程序的脉络，然后用十分钟的时间来编写代码，体会代码如行云流水而出的感觉。

本章完。

# 第四章、现场编写类似 strstr/strcpy/strpbrk 的函数

作者：July。

说明：如果在博客中代码使用了\n，csdn blog 系统将会自动回给我变成/n。据后续验证，可能是原来旧 blog 版本的 bug，新版已不存在此问题。至于，本文代码，日后统一修正。

July、2012.05.02。

微博：<http://weibo.com/julyweibo>。

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

wiki：<http://tctop.wikispaces.com/>。

---

## 前奏

有网友向我反应，之前三章（<http://t.cn/hgVPmH>）的面试题目，是否有点太难了。诚如他所说，绝大部分公司的面试题不会像微软等公司的面试题目出的那么变态，或复杂。

面试考察的是你对基础知识的掌握程度，及编程能力是否过硬的一种检测，所以，扎实基础知识，提高编程能力，比去看什么所谓的面经，或去背面试题目的答案强多了。

很多中、小型公司自己的创造能力，包括人力，物力资源都有限，所以，他们的面试题目除了 copy 一些大公司的题库之外（当然，考察你对基础知识的掌握情况，是肯定不会放过的），还有一个途径就是让你在限定时间内（如十分钟），当场实现一些类似 strcpy/strcat/strpbrk 等库函数，这个主要看你对细节的把握，以及编程能力是否之扎实了。

同时，本章里出现的代码（除了第 4 节的 c 标准库部分源码）都是个人限定在短时间内（正好，突出现场感）编写的，很多问题，难免有所考虑不周。所以，如果你发现本章任何一段代码有任何问题，恳请不吝指正。

## 第一节、字符串查找

### 1.1 题目描述：

给定一个字符串 A，要求在 A 中查找一个子串 B。

如 A="ABCDF"，要你在 A 中查找子串 B="CD"。

分析：比较简单，相当于实现 strstr 库函数，主体代码如下：

```
//在字符串中查找指定字符串的第一次出现，不能找到则返回-1
int strstr(char *string, char *substring)
{
    if (string == NULL || substring == NULL)
        return -1;

    int lenstr = strlen(string);
    int lensub = strlen(substring);

    if (lenstr < lensub)
        return -1;

    int len = lenstr - lensub;
    for (int i = 0; i <= len; i++) //复杂度为 O(m*n)
    {
        for (int j = 0; j < lensub; j++)
        {
            if (string[i+j] != substring[j])
                break;
        }
        if (j == lensub)
            return i + 1;
    }
    return -1;
}
```

读者反馈@xiaohui5319：楼主啊，对于你那个 strstr 的函数，我觉得有点小问题。我查了一下 C 标准库的源码，它给的声明是这样的，两个参数都有 const。

char \*

STRSTR (const char \*haystack\_start, const char \*needle\_start)

而且标准库中没有调用 strlen 函数，因为假如你是标准库的设计者，strlen()函数还没设计出来，你怎么去计算两个字符串的长度？是不是只能通过指针移动来实现，我觉得这些都是微软要考察的地方。

此外：还有 int lenstr=strlen(string);这是不安全的？

`strlen` 函数的返回类型是 `size_t` 型，也就是无符号整型，假如我的数组长度很长（假如是用堆分配的，可以很大很大），长过  $2^{31}$  次方减 1 的话，会发生一处，你这 `lenstr` 就会变成负值了，用 `size_t` 类型最保险。

以后，[本编程艺术系列中](#)有任何问题，暂未来得及及时修正，请读者多加思考，多加辨明。

上述程序已经实现了在字符串中查找第一个子串的功能，时间复杂度为  $O(n*m)$ ，也可以用 **KMP** 算法，复杂度为  $O(m+n)$ 。为人打通思路，提高他人创造力，我想，这是狂想曲与其它的面试解答所不同的地方，也是我们写狂想曲系列文章的意义与价值之所在。

## 1.2、题目描述

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

代码则可以如下编写：

```
//查找第一个只出现一次的字符,
//copyright@ yansha
//July、updated, 2011.04.24.
char FirstNotRepeatChar(char* pString)
{
    if(!pString)
        return '/0';

    const int tableSize = 256;
    //有点要提醒各位注意，一般常数的空间消耗，如这里的 256，我们也认为此空间复杂度为 O(1)。
    int hashTable[tableSize] = {0}; //存入数组，并初始化为 0

    char* pHashKey = pString;
    while(*pHashKey != '/0')
        hashTable[*pHashKey]++;
}

while(*pString != '/0')
{
    if(hashTable[*pString] == 1)
        return *pString;

    pString++;
}
return '/0'; //没有找到满足条件的字符，退出
}
```

代码二， bitmap：

```
# include<stdio.h>
# include<string.h>

const int N = 26;
int bit_map[N];

void findNoRepeat(char *src)
{
    int pos;
    char *str = src;
    int i ,len = strlen(src);

    //统计
    for(i = 0 ; i < len ;i++)
        bit_map[str[i]-'a']++;

    //从字符串开始遍历 其 bit_map==1 那么就是结果
    for(i = 0 ; i < len ; i++)
    {
        if(bit_map[str[i]-'a'] == 1)
        {
            printf("%c",str[i]);
            return ;
        }
    }
}

int main()
{
    char *src = "abaccdeff";
    findNoRepeat(src);
    printf("/n");
    return 0;
}
```

## 第二节、字符串拷贝

题目描述：

要求实现库函数 strcpy，

原型声明： **extern char \*strcpy(char \*dest,char \*src);**

功能：把 `src` 所指由 `NULL` 结束的字符串复制到 `dest` 所指的数组中。

说明：`src` 和 `dest` 所指内存区域不可以重叠且 `dest` 必须有足够的空间来容纳 `src` 的字符串。  
返回指向 `dest` 的指针。

分析：如果编写一个标准 `strcpy` 函数的总分值为 10，下面给出几个不同得分的答案：

```
//得 2 分
void strcpy( char *strDest, char *strSrc )
{
    while( (*strDest++ = * strSrc++) != '/0' );
}

//得 4 分
void strcpy( char *strDest, const char *strSrc )
{
    //将源字符串加 const，表明其为输入参数，加 2 分
    while( (*strDest++ = * strSrc++) != '/0' );
}

//得 7 分
void strcpy(char *strDest, const char *strSrc)
{
    //对源地址和目的地址加非 0 断言，加 3 分
    assert( (strDest != NULL) && (strSrc != NULL) );
    while( (*strDest++ = * strSrc++) != '/0' );
}

//得 9 分
//为了实现链式操作，将目的地址返回，加 2 分！
char * strcpy( char *strDest, const char *strSrc )
{
    assert( (strDest != NULL) && (strSrc != NULL) );
    char *address = strDest;
    while( (*strDest++ = * strSrc++) != '/0' );
    return address;
}

//得 10 分，基本上所有的情况，都考虑到了
//如果有考虑到源目所指区域有重叠的情况，加 1 分！
char * strcpy( char *strDest, const char *strSrc )
{
    if(strDest == strSrc) { return strDest; }
    assert( (strDest != NULL) && (strSrc != NULL) );
    char *address = strDest;
```

```

    while( (*strDest++ = * strSrc++) != '/0' );
    return address;
}

```

### 第三节、小部分库函数的实现

考察此类编写同库函数一样功能的函数经常见于大大小小的 IT 公司的面试题目中，以下是常见的字符串库函数的实现，希望，对你有所帮助，有任何问题，欢迎不吝指正：

```

//@yansha:字串末尾要加结束符'/0'，不然输出错位结果
char *strncpy(char *strDes, const char *strSrc, unsigned int count)
{
    assert(strDes != NULL && strSrc != NULL);
    char *address = strDes;
    while (count-- && *strSrc != '/0')
        *strDes++ = *strSrc++;
    *strDes = '/0';
    return address;
}

//查找字符串 s 中首次出现字符 c 的位置
char *strchr(const char *str, int c)
{
    assert(str != NULL);
    for (; *str != (char)c; ++ str)
        if (*str == '/0')
            return NULL;
    return str;
}

int strcmp(const char *s, const char *t)
{
    assert(s != NULL && t != NULL);
    while (*s && *t && *s == *t)
    {
        ++ s;
        ++ t;
    }
    return (*s - *t);
}

char *strcat(char *strDes, const char *strSrc)

```

```

{
    assert((strDes != NULL) && (strSrc != NULL));
    char *address = strDes;
    while (*strDes != '/0')
        ++ strDes;
    while ((*strDes ++ = *strSrc++) != '/0')
        NULL;
    return address;
}

int strlen(const char *str)
{
    assert(str != NULL);
    int len = 0;
    while (*str ++ != '/0')
        ++ len;
    return len;
}

//此函数，梦修改如下
char *strdup_(char *strSrc)
//将字符串拷贝到新的位置
{
    if(strSrc!=NULL)
    {
        char *start=strSrc;
        int len=0;
        while(*strSrc++!='/0')
            len++;

        char *address=(char *)malloc(len+1);
        assert(address != NULL);

        while((*address++=*start++)!='/0');
        return address-(len+1);
    }
    return NULL;
}

//多谢 laoyi19861011 指正
char *strstr(const char *strSrc, const char *str)
{
    assert(strSrc != NULL && str != NULL);
    const char *s = strSrc;

```

```

const char *t = str;
for (; *strSrc != '\0'; ++ strSrc)
{
    for (s = strSrc, t = str; *t != '\0' && *s == *t; ++s, ++t)
        NULL;
    if (*t == '\0')
        return (char *) strSrc;
}
return NULL;
}

char *strncat(char *strDes, const char *strSrc, unsigned int count)
{
    assert((strDes != NULL) && (strSrc != NULL));
    char *address = strDes;
    while (*strDes != '\0')
        ++ strDes;
    while (count -- && *strSrc != '\0' )
        *strDes ++ = *strSrc++;
    *strDes = '\0';
    return address;
}

int strncmp(const char *s, const char *t, unsigned int count)
{
    assert((s != NULL) && (t != NULL));
    while (*s && *t && *s == *t && count --)
    {
        ++ s;
        ++ t;
    }
    return (*s - *t);
}

char *strupr(const char *strSrc, const char *str)
{
    assert((strSrc != NULL) && (str != NULL));
    const char *s;
    while (*strSrc != '\0')
    {
        s = str;
        while (*s != '\0')
        {
            if (*strSrc == *s)

```

```

        return (char *) strSrc;
    ++ s;
}
++ strSrc;
}
return NULL;
}

int strcspn(const char *strSrc, const char *str)
{
    assert((strSrc != NULL) && (str != NULL));
    const char *s;
    const char *t = strSrc;
    while (*t != '/0')
    {
        s = str;
        while (*s != '/0')
        {
            if (*t == *s)
                return t - strSrc;
            ++ s;
        }
        ++ t;
    }
    return 0;
}

int strspn(const char *strSrc, const char *str)
{
    assert((strSrc != NULL) && (str != NULL));
    const char *s;
    const char *t = strSrc;
    while (*t != '/0')
    {
        s = str;
        while (*s != '/0')
        {
            if (*t == *s)
                break;
            ++ s;
        }
        if (*s == '/0')
            return t - strSrc;
        ++ t;
    }
}
```

```

    }
    return 0;
}

char *strrchr(const char *str, int c)
{
    assert(str != NULL);
    const char *s = str;
    while (*s != '/0')
        ++ s;
    for (-- s; *s != (char) c; -- s)
        if (s == str)
            return NULL;
    return (char *) s;
}

char* strrev(char *str)
{
    assert(str != NULL);
    char *s = str, *t = str, c;
    while (*t != '/0')
        ++ t;
    for (-- t; s < t; ++ s, -- t)
    {
        c = *s;
        *s = *t;
        *t = c;
    }
    return str;
}

char *strnset(char *str, int c, unsigned int count)
{
    assert(str != NULL);
    char *s = str;
    for (; *s != '/0' && s - str < count; ++ s)
        *s = (char) c;
    return str;
}

char *strset(char *str, int c)
{
    assert(str != NULL);
    char *s = str;

```

```

    for (; *s != '/\0'; ++ s)
        *s = (char) c;
    return str;
}

//@heyaming
//对原 strtok 的修改, 根据 MSDN,strToken 可以为 NULL. 实际上第一次 call strtok 给定一字符串,
//再 call strtok 时可以输入 NULL 代表要接着处理给定字符串。
//所以需要用一 static 保存没有处理完的字符串。同时也需要处理多个分隔符在一起的情况。
char *strtok(char *strToken, const char *str)
{
    assert(str != NULL);
    static char *last;

    if (strToken == NULL && (strToken = last) == NULL)
        return (NULL);

    char *s = strToken;
    const char *t = str;
    while (*s != '/\0')
    {
        t = str;
        while (*t != '/\0')
        {
            if (*s == *t)
            {
                last = s + 1;
                if (s - strToken == 0) {
                    strToken = last;
                    break;
                }
                *(strToken + (s - strToken)) = '/\0';
                return strToken;
            }
            ++ t;
        }
        ++ s;
    }
    return NULL;
}

char *strupr(char *str)
{
    assert(str != NULL);

```

```

char *s = str;
while (*s != '/0')
{
    if (*s >= 'a' && *s <= 'z')
        *s -= 0x20;
    s++;
}
return str;
}

char *strlwr(char *str)
{
    assert(str != NULL);
    char *s = str;
    while (*s != '/0')
    {
        if (*s >= 'A' && *s <= 'Z')
            *s += 0x20;
        s++;
    }
    return str;
}

void *memcpy(void *dest, const void *src, unsigned int count)
{
    assert((dest != NULL) && (src != NULL));
    void *address = dest;
    while (count --)
    {
        *(char *) dest = *(char *) src;
        dest = (char *) dest + 1;
        src = (char *) src + 1;
    }
    return address;
}

void *memccpy(void *dest, const void *src, int c, unsigned int count)
{
    assert((dest != NULL) && (src != NULL));
    while (count --)
    {
        *(char *) dest = *(char *) src;
        if (* (char *) src == (char) c)
            return ((char *) dest + 1);
    }
}

```

```

        dest = (char *) dest + 1;
        src = (char *) src + 1;
    }
    return NULL;
}

void *memchr(const void *buf, int c, unsigned int count)
{
    assert(buf != NULL);
    while (count--)
    {
        if (*(char *) buf == c)
            return (void *) buf;
        buf = (char *) buf + 1;
    }
    return NULL;
}

int memcmp(const void *s, const void *t, unsigned int count)
{
    assert((s != NULL) && (t != NULL));
    while (*((char *) s && *((char *) t && *((char *) s == *((char *) t) && count --))
    {
        s = (char *) s + 1;
        t = (char *) t + 1;
    }
    return (*((char *) s - *((char *) t));
}

//@big:
//要处理 src 和 dest 有重叠的情况，不是从尾巴开始移动就没问题了。
//一种情况是 dest 小于 src 有重叠，这个时候要从头开始移动，
//另一种是 dest 大于 src 有重叠，这个时候要从尾开始移动。
void *memmove(void *dest, const void *src, unsigned int count)
{
    assert(dest != NULL && src != NULL);
    char* pdest = (char*) dest;
    char* psrc = (char*) src;

    //pdest 在 psrc 后面，且两者距离小于 count 时，从尾部开始移动。其他情况从头部开始移动
    if (pdest > psrc && pdest - psrc < count)
    {
        while (count--)
        {

```

```

        *(pdest + count) = *(psrc + count);
    }
} else
{
    while (count--)
    {
        *pdest++ = *psrc++;
    }
}
return dest;
}

void *memset(void *str, int c, unsigned int count)
{
    assert(str != NULL);
    void *s = str;
    while (count --)
    {
        *(char *) s = (char) c;
        s = (char *) s + 1;
    }
    return str;
}

```

**测试：**以上所有的函数，都待进一步测试，有任何问题，欢迎任何人随时不吝指出。

## 第四节、c 标准库部分源代码

为了给各位一个可靠的参考，以下，我摘取一些 c 标准框里的源代码，以飨各位：

```

char * __cdecl strcat (char * dst,const char * src)
{
    char * cp = dst;

    while( *cp )
        cp++; /* find end of dst */

    while( *cp++ = *src++ ) ; /* Copy src to end of dst */

    return( dst ); /* return dst */
}

```

```

int __cdecl strcmp (const char * src,const char * dst)
{
    int ret = 0 ;

    while( ! (ret = *(unsigned char *)src - *(unsigned char *)dst) && *dst)
        ++src, ++dst;

    if ( ret < 0 )
        ret = -1 ;
    else if ( ret > 0 )
        ret = 1 ;

    return( ret );
}

size_t __cdecl strlen (const char * str)
{
    const char *eos = str;

    while( *eos++ ) ;

    return( (int)(eos - str - 1) );
}

char * __cdecl strncat (char * front,const char * back,size_t count)
{
    char *start = front;

    while (*front++)
        ;
    front--;

    while (count--)
        if (!(*front++ = *back++))
            return(start);

    *front = '/0';
    return(start);
}

int __cdecl strncmp (const char * first,const char * last,size_t count)
{
    if (!count)
        return(0);
}

```

```

    while (--count && *first && *first == *last)
    {
        first++;
        last++;
    }

    return( *(unsigned char *)first - *(unsigned char *)last );
}

/* Copy SRC to DEST.  */
char *
strcpy (dest, src)
char *dest;
const char *src;
{
    reg_char c;
    char *__unbounded s = (char *__unbounded) CHECK_BOUNDS_LOW (src);
    const ptrdiff_t off = CHECK_BOUNDS_LOW (dest) - s - 1;
    size_t n;

    do
    {
        c = *s++;
        s[off] = c;
    }
    while (c != '/0');

    n = s - src;
    (void) CHECK_BOUNDS_HIGH (src + n);
    (void) CHECK_BOUNDS_HIGH (dest + n);

    return dest;
}

char * __cdecl strncpy (char * dest,const char * source,size_t count)
{
    char *start = dest;

    while (count && (*dest++ = *source++)) /* copy string */
        count--;

    if (count) /* pad out with zeroes */
        while (--count)

```

```
*dest++ = '/0';  
  
return(start);  
}
```

## 有关狂想曲的修订

程序员面试题狂想曲-tctop (the crazy thinking of programmers) 的修订 wiki (<http://tctop.wikispaces.com/>) 已于今天建立，我们急切的想得到读者的反馈，意见，建议，以及更好的思路，算法，和代码优化的建议。所以，

- 如果你发现了狂想曲系列中的任何一题，任何一章 (<http://t.cn/hgVPmH>) 中的错误，问题，与漏洞，欢迎告知给我们，我们将感激不尽，同时，免费赠送本 blog 内的全部博文集锦的 CHM 文件 1 期；
- 如果你能对狂想曲系列的创作提供任何建设性意见，或指导，欢迎反馈给我们，并真诚邀请您加入到狂想曲的 wiki 修订工作中；
- 如果你是编程高手，对狂想曲的任何一章有自己更好的思路，或算法，欢迎加入狂想曲的创作组，以为千千万万的读者创造更多的价值，更好的服务。

Ps: 狂想曲 tctop 的 wiki 修订地址为: <http://tctop.wikispaces.com/>。欢迎围观，更欢迎您加入到狂想曲的创作或 wiki 修订中。

# 第五章、寻找和为定值的两个或多个数

作者: July, yansha, zhouzhenren。

致谢: 微软 100 题实现组, 编程艺术室。

微博: <http://weibo.com/julyweibo>。

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

wiki: <http://tctop.wikispaces.com/>。

---

## 前奏

希望此编程艺术系列能给各位带来的是一种方法, 一种创造力, 一种举一反三的能力。本章依然同第四章一样, 选取比较简单的面试题, 恭祝各位旅途愉快。同样, 有任何问题, 欢迎不吝指正。谢谢。

## 第一节、寻找和为定值的两个数

第 14 题 (数组) :

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

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

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

分析:

咱们试着一步一步解决这个问题 (注意阐述中数列有序无序的区别) :

1. 直接穷举, 从数组中任意选取两个数, 判定它们的和是否为输入的那个数字。此举复杂度为  $O(N^2)$ 。很显然, 我们要寻找效率更高的解法。
2. 题目相当于, 对每个  $a[i]$ , 查找  $sum-a[i]$  是否也在原始序列中, 每一次要查找的时间都要花费为  $O(N)$ , 这样下来, 最终找到两个数还是需要  $O(N^2)$  的复杂度。那如何提高查找判断的速度呢? 答案是二分查找, 可以将  $O(N)$  的查找时间提高到  $O(\log N)$ , 这样对于  $N$  个  $a[i]$ , 都要花  $\log N$  的时间去查找相对应的  $sum-a[i]$  是否

在原始序列中，总的时间复杂度已降为  $O(N \log N)$ ，且空间复杂度为  $O(1)$ 。

（如果有序，直接二分  $O(N \log N)$ ，如果无序，先排序后二分，复杂度同样为  $O(N \log N + N \log N) = O(N \log N)$ ，空间总为  $O(1)$ ）。

3. 有没有更好的办法呢？咱们可以依据上述思路 2 的思想， $a[i]$  在序列中，如果  $a[i]+a[k]=sum$  的话，那么  $sum-a[i] (a[k])$  也必然在序列中，举个例子，如下：  
原始序列：1、2、4、7、11、15 用输入数字 15 减一下各个数，得到对应的序列为：

对应序列：14、13、11、8、4、0

第一个数组以一指针  $i$  从数组最左端开始向右扫描，第二个数组以一指针  $j$  从数组最右端开始向左扫描，如果下面出现了和上面一样的数，即  $a[*i]=a[*j]$ ，就找出这两个数来了。如上， $i, j$  最终在第一个，和第二个序列中找到了相同的数 4 和 11，所以符合条件的两个数，即为  $4+11=15$ 。怎么样，两端同时查找，时间复杂度瞬间缩短到了  $O(N)$ ，但却同时需要  $O(N)$  的空间存储第二个数组（@飞羽：要达到  $O(N)$  的复杂度，第一个数组以一指针  $i$  从数组最左端开始向右扫描，第二个数组以一指针  $j$  从数组最右端开始向左扫描，首先初始  $i$  指向元素 1， $j$  指向元素 0，谁指的元素小，谁先移动，由于  $1(i) > 0(j)$ ，所以  $i$  不动， $j$  向左移动。然后  $j$  移动到元素 4 发现大于元素 1，故而停止移动  $j$ ，开始移动  $i$ ，直到  $i$  指向 4，这时  $i$  指向的元素与  $j$  指向的元素相等，故而判断 4 是满足条件的第一个数；然后同时移动  $i, j$  再进行判断，直到它们到达边界）。

4. 当然，你还可以构造  $hash$  表，正如编程之美上的所述，给定一个数字，根据  $hash$  映射查找另一个数字是否也在数组中，只需用  $O(1)$  的时间，这样的话，总体的算法通上述思路 3 一样，也能降到  $O(N)$ ，但有个缺陷，就是构造  $hash$  额外增加了  $O(N)$  的空间，此点同上述思路 3。不过，空间换时间，仍不失为在时间要求较严格的情况下的一种好办法。
5. 如果数组是无序的，先排序 ( $n \log n$ )，然后用两个指针  $i, j$ ，各自指向数组的首尾两端，令  $i=0, j=n-1$ ，然后  $i++, j--$ ，逐次判断  $a[i]+a[j]?=sum$ ，如果某一刻  $a[i]+a[j]>sum$ ，则要想办法让  $sum$  的值减小，所以此刻  $i$  不动， $j--$ ，如果某一刻  $a[i]+a[j]<sum$ ，则要想办法让  $sum$  的值增大，所以此刻  $i++$ ， $j$  不动。所以，数组无序的时候，时间复杂度最终为  $O(n \log n + n) = O(n \log n)$ ，若原数组是有序的，则不需要事先的排序，直接  $O(n)$  搞定，且空间复杂度还是  $O(1)$ ，此思路是相对于上述所有思路的一种改进。（如果有序，直接两个指针两端扫描，时间  $O(N)$ ，如果无序，先排序后两端扫描，时间  $O(N \log N + N) = O(N \log N)$ ，空间始终都为  $O(1)$ ）。  
(与上述思路 2 相比，排序后的时间开销由之前的二分的  $n \log n$  降到了扫描的  $O(N)$ ）。

总结：

- 不论原序列是有序还是无序，解决这类题有以下三种办法：1、二分（若无序，先排序后二分），时间复杂度总为  $O(n \log n)$ ，空间复杂度为  $O(1)$ ；2、扫描一遍  $X-S[i]$  映射到一个数组或构造 hash 表，时间复杂度为  $O(n)$ ，空间复杂度为  $O(n)$ ；3、两个指针两端扫描（若无序，先排序后扫描），时间复杂度最后为：有序  $O(n)$ ，无序  $O(n \log n + n) = O(n \log n)$ ，空间复杂度都为  $O(1)$ 。
- 所以，要想达到时间  $O(N)$ ，空间  $O(1)$  的目标，除非原数组是有序的（指针扫描法），不然，当数组无序的话，就只能先排序，后指针扫描法或二分（时间  $n \log n$ ，空间  $O(1)$ ），或映射或 hash（时间  $O(n)$ ，空间  $O(n)$ ）。时间或空间，必须牺牲一个，自个权衡吧。
- 综上，若是数组有序的情况下，优先考虑两个指针两端扫描法，以达到最佳的时  $(O(N))$ ，空  $(O(1))$  效应。否则，如果要排序的话，时间复杂度最快当然是只能达到  $N \log N$ ，空间  $O(1)$  则是不在话下。

代码：

ok，在进入第二节之前，咱们先来实现思路 5（这里假定数组已经是有序的），代码可以如下编写（两段代码实现）：

```
//代码一
//O(N)
Pair findSum(int *s,int n,int x)
{
    //sort(s,s+n);    如果数组非有序的，那就事先排好序 O(N*logN)

    int *begin=s;
    int *end=s+n-1;

    while(begin<end)      //俩头夹逼，或称两个指针两端扫描法，很经典的方法，O(N)
    {
        if(*begin+*end>x)
        {
            --end;
        }
        else if(*begin+*end<x)
        {
            ++begin;
        }
        else
        {
            return Pair(*begin,*end);
        }
    }
}
```

```

    }

    return Pair(-1,-1);
}

//或者如下编写,
//代码二
//copyright@ zhedahht && yansha
//July、updated, 2011.05.14.
bool find_num(int data[], unsigned int length, int sum, int& first_num, int& second
_num)
{
    if(length < 1)
        return true;

    int begin = 0;
    int end = length - 1;

    while(end > begin)
    {
        long current_sum = data[begin] + data[end];

        if(current_sum == sum)
        {
            first_num = data[begin];
            second_num = data[end];
            return true;
        }
        else if(current_sum > sum)
            end--;
        else
            begin++;
    }
    return false;
}

```

## 扩展:

- 1、如果在返回找到的两个数的同时，还要求你返回这两个数的位置列？
- 2、如果把题目中的要你寻找的两个数改为“多个数”，或任意个数列？（请看下面第二节）
- 3、二分查找时： left <= right, right = middle - 1;left < right, right = middle;

//算法所操作的区间,是左闭右开区间,还是左闭右闭区间,这个区间,需要在循环初始化,  
//循环体是否终止的判断中,以及每次修改 left,right 区间值这三个地方保持一致,否则就可能  
出错.

//二分查找实现一

```
int search(int array[], int n, int v)
{
```

```
    int left, right, middle;
```

```
    left = 0, right = n - 1;
```

```
    while (left <= right)
```

```
{
```

```
    middle = left + (right-left)/2;
```

```
    if (array[middle] > v)
```

```
{
```

```
        right = middle - 1;
```

```
}
```

```
    else if (array[middle] < v)
```

```
{
```

```
        left = middle + 1;
```

```
}
```

```
    else
```

```
{
```

```
        return middle;
```

```
}
```

```
}
```

```
    return -1;
```

```
}
```

//二分查找实现二

```
int search(int array[], int n, int v)
```

```
{
```

```
    int left, right, middle;
```

```

left = 0, right = n;

while (left < right)
{
    middle = left + (right-left)/2;

    if (array[middle] > v)
    {
        right = middle;
    }
    else if (array[middle] < v)
    {
        left = middle + 1;
    }
    else
    {
        return middle;
    }
}

return -1;
}

```

## 第二节、寻找和为定值的多个数

第 21 题（数组）

2010 年中兴面试题

编程求解：

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

### 解法一

我想，稍后给出的程序已经足够清楚了，就是要注意到放  $n$ ，和不放  $n$  个区别，即可，代码如下：

```

// 21 题递归方法
//copyright@ July && yansha
//July、yansha, updated.
#include<list>
#include<iostream>
using namespace std;

list<int>list1;

void find_factor(int sum, int n)
{
    // 递归出口
    if(n <= 0 || sum <= 0)
        return;

    // 输出找到的结果
    if(sum == n)
    {
        // 反转 list
        list1.reverse();
        for(list<int>::iterator iter = list1.begin(); iter != list1.end(); iter++)
            cout << *iter << " + ";
        cout << endl;
        list1.reverse();
    }

    list1.push_front(n);      //典型的 01 背包问题
    find_factor(sum-n, n-1); //放 n, n-1 个数填满 sum-n
    list1.pop_front();
    find_factor(sum, n-1);   //不放 n, n-1 个数填满 sum
}

int main()
{
    int sum, n;
    cout << "请输入你要等于多少的数值 sum:" << endl;
    cin >> sum;
    cout << "请输入你要从 1.....n 数列中取值的 n: " << endl;
    cin >> n;
    cout << "所有可能的序列, 如下: " << endl;
    find_factor(sum,n);
    return 0;
}

```

## 解法二

@zhouzhenren:

这个问题属于子集和问题（也是背包问题）。本程序采用 回溯法+剪枝  
 $X$  数组是解向量， $t = \sum(1,..,k-1)Wi * Xi$ ,  $r = \sum(k,..,n)Wi$   
若  $t + W_k + W_{k+1} \leq M$ , 则  $X_k = true$ , 递归左儿子( $X_1, X_2, .., X_{(k-1)}, 1$ ); 否则剪枝；  
若  $t + r - W_k \geq M \ \&\& \ t + W_{k+1} \leq M$ , 则置  $X_k = 0$ , 递归右儿子( $X_1, X_2, .., X_{(k-1)}, 0$ ); 否则剪枝；  
本题中  $W$  数组就是  $(1, 2, .., n)$ , 所以直接用  $k$  代替  $W_K$  值。

代码编写如下：

```
//copyright@ 2011 zhouzhenren

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

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>

/**
 * 输入 t, r, 尝试 W_k
 */
void sumofsub(int t, int k, int r, int& M, bool& flag, bool* X)
{
    X[k] = true; // 选第 k 个数
    if (t + k == M) // 若找到一个和为 M, 则设置解向量的标志位, 输出解
    {
        flag = true;
        for (int i = 1; i <= k; ++i)
        {
            if (X[i] == 1)
            {
                printf("%d ", i);
            }
        }
        printf("\n");
    }
    else
    { // 若第 k+1 个数满足条件, 则递归左子树
        if (t + k + (k+1) <= M)
        {
            sumofsub(t + k, k + 1, r - k, M, flag, X);
        }
    }
}
```

```

        // 若不选第 k 个数, 选第 k+1 个数满足条件, 则递归右子树
        if ((t + r - k >= M) && (t + (k+1) <= M))
        {
            X[k] = false;
            sumofsub(t, k + 1, r - k, M, flag, X);
        }
    }

void search(int& N, int& M)
{
    // 初始化解空间
    bool* X = (bool*)malloc(sizeof(bool) * (N+1));
    memset(X, false, sizeof(bool) * (N+1));
    int sum = (N + 1) * N * 0.5f;
    if (1 > M || sum < M) // 预先排除无解情况
    {
        printf("not found\n");
        return;
    }
    bool f = false;
    sumofsub(0, 1, sum, M, f, X);
    if (!f)
    {
        printf("not found\n");
    }
    free(X);
}

int main()
{
    int N, M;
    printf("请输入整数 N 和 M/n");
    scanf("%d%d", &N, &M);
    search(N, M);
    return 0;
}

```

### 扩展:

1、从一列数中筛除尽可能少的数使得从左往右看, 这些数是从小到大再从大到小的(网易)。

2、有两个序列 a,b, 大小都为 n, 序列元素的值任意整数, 无序;

要求: 通过交换 a,b 中的元素, 使[序列 a 元素的和]与[序列 b 元素的和]之间的差最小。

例如：

```
var a=[100,99,98,1,2, 3];
var b=[1, 2, 3, 4,5,40]; (微软 100 题第 32 题)。
```

@well: [fairywell]:

给出扩展问题 1 的一个解法：

1、从一列数中筛选尽可能少的数使得从左往右看，这些数是从小到大再从大到小的（网易）。

双端 LIS 问题，用 DP 的思想可解，目标规划函数  $\max\{ b[i] + c[i] - 1 \}$ ，其中  $b[i]$  为从左到右， $0 \sim i$  个数之间满足递增的数字个数； $c[i]$  为从右到左， $n-1 \sim i$  个数之间满足递增的数字个数。最后结果为  $n - \max + 1$ 。其中 DP 的时候，可以维护一个  $inc[]$  数组表示递增数字序列， $inc[i]$  为从小到大第  $i$  大的数字，然后在计算  $b[i] c[i]$  的时候使用二分查找在  $inc[]$  中找出区间  $inc[0] \sim inc[i-1]$  中小于  $a[i]$  的元素个数（low）。

源代码如下：

```
/*
* The problem:
* 从一列数中筛选尽可能少的数使得从左往右看，这些数是从小到大再从大到小的（网易）。
* use binary search, perhaps you should compile it with -std=c99
* fairywell 2011
*/
#include <stdio.h>

#define MAX_NUM      (1U<<31)

int
main()
{
    int i, n, low, high, mid, max;

    printf("Input how many numbers there are: ");
    scanf("%d\n", &n);

    /* a[] holds the numbers, b[i] holds the number of increasing numbers
     * from a[0] to a[i], c[i] holds the number of increasing numbers
     * from a[n-1] to a[i]
     * inc[] holds the increasing numbers
     * VLA needs c99 features, compile with -std=c99
     */
    double a[n], b[n], c[n], inc[n];

    printf("Please input the numbers:/n");
    for (i = 0; i < n; ++i) scanf("%lf", &a[i]);
```

```

// update array b from left to right
for (i = 0; i < n; ++i) inc[i] = (unsigned) MAX_NUM;
//b[0] = 0;
for (i = 0; i < n; ++i) {
    low = 0; high = i;
    while (low < high) {
        mid = low + (high-low)*0.5;
        if (inc[mid] < a[i]) low = mid + 1;
        else high = mid;
    }
    b[i] = low + 1;
    inc[low] = a[i];
}

// update array c from right to left
for (i = 0; i < n; ++i) inc[i] = (unsigned) MAX_NUM;
//c[0] = 0;
for (i = n-1; i >= 0; --i) {
    low = 0; high = i;
    while (low < high) {
        mid = low + (high-low)*0.5;
        if (inc[mid] < a[i]) low = mid + 1;
        else high = mid;
    }
    c[i] = low + 1;
    inc[low] = a[i];
}

max = 0;
for (i = 0; i < n; ++i)
    if (b[i]+c[i] > max) max = b[i] + c[i];
printf("%d number(s) should be erased at least./n", n+1-max);
return 0;
}

```

@yansha: fairywell 的程序很赞，时间复杂度  $O(n \log n)$ ，这也是我能想到的时间复杂度最优值了。不知能不能达到  $O(n)$ 。

## 扩展题第 2 题

当前数组  $a$  和数组  $b$  的和之差为

$$A = \text{sum}(a) - \text{sum}(b)$$

$a$  的第  $i$  个元素和  $b$  的第  $j$  个元素交换后， $a$  和  $b$  的和之差为

$$\begin{aligned} A' &= \text{sum}(a) - a[i] + b[j] - (\text{sum}(b) - b[j] + a[i]) \\ &= \text{sum}(a) - \text{sum}(b) - 2(a[i] - b[j]) \\ &= A - 2(a[i] - b[j]) \end{aligned}$$

设  $x = a[i] - b[j]$ , 得

$$|A| - |A'| = |A| - |A - 2x|$$

假设  $A > 0$ ,

当  $x$  在  $(0, A)$  之间时, 做这样的交换才能使得交换后的  $a$  和  $b$  的和之差变小,  $x$  越接近  $A/2$  效果越好, 如果找不到在  $(0, A)$  之间的  $x$ , 则当前的  $a$  和  $b$  就是答案。

所以算法大概如下:

在  $a$  和  $b$  中寻找使得  $x$  在  $(0, A)$  之间并且最接近  $A/2$  的  $i$  和  $j$ , 交换相应的  $i$  和  $j$  元素, 重新计算  $A$  后, 重复前面的步骤直至找不到  $(0, A)$  之间的  $x$  为止。

接上, @yuan:

$a[i]-b[j]$  要接近  $A/2$ , 则可以这样想,

我们可以对于  $a$  数组的任意一个  $a[k]$ , 在数组  $b$  中找出与  $a[k]-C$  最接近的数 ( $C$  就是常数, 也就是  $0.5*A$ ) 这个数要么就是  $a[k]-C$ , 要么就是比他稍大, 要么比他稍小, 所以可以要二分查找。

查找最后一个小于等于  $a[k]-C$  的数和第一个大于等于  $a[k]-C$  的数,

然后看哪一个与  $a[k]-C$  更加接近, 所以  $T(n) = n\log n$ 。

除此之外, 受本文读者 xiafei1987128 启示, 有朋友在 stackoverflow 上也问过一个类似的题, :-), 见此:

<http://stackoverflow.com/questions/9047908/swap-the-elements-of-two-sequences-such-that-the-difference-of-the-element-sums>。感兴趣的可以看看。

本章完。

# 第六章、亲和数问题--求解 500 万以内的亲和数

作者：上善若水、July、yansha。

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

## 前奏

本章陆续开始，除了继续保持原有的字符串、数组等面试题之外，会有意识的间断性节选一些有关数字趣味小而巧的面试题目，重在突出思路的“巧”，和“妙”。本章亲和数问题之关键字，“500 万”，“线性复杂度”。

## 第一节、亲和数问题

题目描述：

求 500 万以内的所有亲和数

如果两个数  $a$  和  $b$ ,  $a$  的所有真因数之和等于  $b$ , $b$  的所有真因数之和等于  $a$ ,则称  $a,b$  是一对亲和数。

例如 220 和 284, 1184 和 1210, 2620 和 2924。

分析：

首先得明确到底是什么是亲和数？

亲和数问题最早是由毕达哥拉斯学派发现和研究的。他们在研究数字的规律的时候发现有以下性质特点的两个数：

220 的真因子是：1、2、4、5、10、11、20、22、44、55、110;

284 的真因子是：1、2、4、71、142。

而这两个数恰恰等于对方的真因子各自加起来的和（ $\text{sum}[i]$ 表示数  $i$  的各个真因子的和），即

$$220=1+2+4+5+10+11+20+22+44+55+110=\text{sum}[220],$$

$$284=1+2+4+71+142=\text{sum}[284].$$

得 284 的真因子之和  $\text{sum}[284]=220$ , 且 220 的真因子之和  $\text{sum}[220]=284$ , 即有  $\text{sum}[220]=\text{sum}[\text{sum}[284]]=284$ 。

如此, 是否已看出丝毫端倪?

如上所示, 考虑到 1 是每个整数的因子, 把出去整数本身之外的所有因子叫做这个数的“真因子”。如果两个整数, 其中每一个真因子的和都恰好等于另一个数, 那么这两个数, 就构成一对“亲和数”(有关亲和数的更多讨论, 可参考这: <http://t.cn/hesH09>)。

### 求解:

了解了什么是亲和数, 接下来咱们一步一步来解决上面提出的问题(以下内容大部引自水的原话, 同时水哥有一句原话, “在你真正弄懂这个范例之前, 你不配说你懂数据结构和算法”)。

1. 看到这个问题后, 第一想法是什么? 模拟搜索+剪枝? 回溯? 时间复杂度有多大? 其中  $bn$  为  $an$  的伪亲和数, 即  $bn$  是  $an$  的真因数之和大约是多少? 至少是  $10^{13}$  (@iicup:  $N^{1.5}$  对于  $5 \times 10^6$ , 次数大致  $10^{10}$  而不是  $10^{13}$ .) 的数量级的。那么对于每秒千万次运算的计算机来说, 大概在 1000 多天也就是 3 年内就可以搞定了(iicup 的计算:  $10^{13} / 10^7 = 1000000$ (秒) 大约 278 小时.)。如果是基于这个基数在优化, 你无法在一天内得到结果的。
2. 一个不错的算法应该在半小时之内搞定这个问题, 当然这样的算法有很多。节约时间的做法是可以生成伴随数组, 也就是空间换时间, 但是那样, 空间代价太大, 因为数据规模庞大。
3. 在稍后的算法中, 依然使用的伴随数组, 只不过, 因为题目的特殊性, 只是它方便和巧妙地利用了下标作为伴随数组, 来节约时间。同时, 将回溯的思想换成递推的思想(预处理数组的时间复杂度为  $\log N$  (调和级数) \*  $N$ , 扫描数组的时间复杂度为线性  $O(N)$ )。所以, 总的时间复杂度为  $O(N * \log N + N)$  (其中  $\log N$  为调和级数)。

## 第二节、伴随数组线性遍历

依据上文中的第 3 点思路, 编写如下代码:

```
//求解亲和数问题

//第一个 for 和第二个 for 循环是 logn (调和级数) * N 次遍历, 第三个 for 循环扫描 O (N)。
//所以总的时间复杂度为 O (n * logn) + O (n) = O (N * logN) (其中 logN 为调和级数)。
```

```

//关于第一个 for 和第二个 for 寻找中，调和级数的说明：
//比如给 2 的倍数加 2，那么应该是 n/2 次，3 的倍数加 3 应该是 n/3 次，...
//那么其实都是 n * (1+1/2+1/3+1/4+...1/(n/2)) =n* (调和级数) =n*logn。

//copyright@ 上善若水
//July、updated, 2011.05.24。
#include<stdio.h>

int sum[5000010]; //为防越界

int main()
{
    int i, j;
    for (i = 0; i <= 5000000; i++)
        sum[i] = 1; //1 是所有数的真因数所以全部置 1

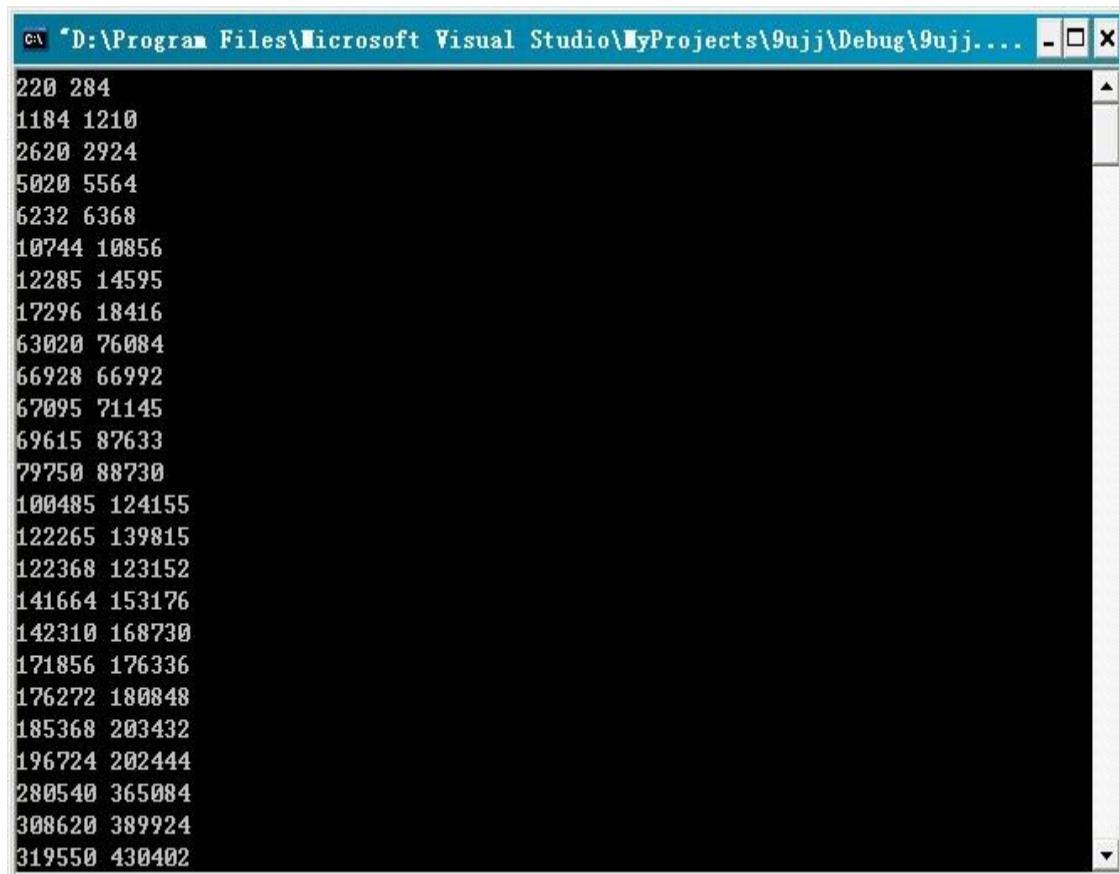
    for (i = 2; i + i <= 5000000; i++) //预处理，预处理是 logN (调和级数) *N。
        //@litaoye: 调和级数 1/2 + 1/3 + 1/4.....的和近似为 ln(n)，
        //因此 O(n * (1/2 + 1/3 + 1/4.....)) = O(n * ln(n)) = O(N*log(N))。
    {

        //5000000 以下最大的真因数是不超过它的一半的
        j = i + i; //因为真因数，所以不能算本身，所以从它的 2 倍开始
        while (j <= 5000000)
        {
            //将所有 i 的倍数的位置上加 i
            sum[j] += i;
            j += i;
        }
    }

    for (i = 220; i <= 5000000; i++) //扫描，O (N)。
    {
        // 一次遍历，因为知道最小是 220 和 284 因此从 220 开始
        if (sum[i] > i && sum[i] <= 5000000 && sum[sum[i]] == i)
        {
            //去重，不越界，满足亲和
            printf("%d %d\n", i, sum[i]);
        }
    }
    return 0;
}

```

运行结果：



The screenshot shows a command-line window with the title bar "D:\Program Files\Microsoft Visual Studio\MyProjects\9ujj\Debug\9ujj....". The window contains a list of 26 pairs of numbers, each pair consisting of two consecutive integers. The numbers are listed vertically, separated by a space. The last number, 319550, is highlighted with a black rectangle.

220	284
1184	1210
2620	2924
5020	5564
6232	6368
10744	10856
12285	14595
17296	18416
63020	76084
66928	66992
67095	71145
69615	87633
79750	88730
100485	124155
122265	139815
122368	123152
141664	153176
142310	168730
171856	176336
176272	180848
185368	203432
196724	202444
280540	365084
308620	389924
319550	430402

@上善若水：

1、可能大家理解的还不是很清晰，我们建立一个 5 000 000 的数组，从 1 到 2 500 000 开始，在每一个下标是  $i$  的倍数的位置上加上  $i$ ，那么在循环结束之后，我们得到的是什么？是 类似埃斯托拉晒求素数的数组（当然里面有真的亲和数），然后只需要一次遍历就可以轻松找到所有的亲和数了。时间复杂度，线性。

2、我们可以清晰的发现连续数据的映射可以通过数组结构本身的特点替代，用来节约空间，这是数据结构的艺术。在大规模连续数据的回溯处理上，可以通过转化为递推生成的方法，逆向思维操作，这是算法的艺术。

3、把最简单的东西运用的最巧妙的人，要比用复杂方法解决复杂问题的人要头脑清晰。

### 第三节、程序的构造与解释

我再来具体解释下上述程序的原理，ok，举个例子，假设是求 10 以内的亲和数，求解步骤如下：

因为所有数的真因数都包含 1，所以，先在各个数的下方全部置 1

1. 然后取  $i=2,3,4,5$  ( $i \leq 10/2$ )， $j$  依次对应的位置为  $j=(4、6、8、10)$ ， $(6、9)$ ， $(8)$ ， $(10)$  各数所对应的位置。
2. 依据  $j$  所找到的位置，在  $j$  所指的各个数的下面加上各个真因子  $i$  ( $i=2、3、4、5$ )。整个过程，即如下图所示（如  $\text{sum}[6]=1+2+3=6$ ,  $\text{sum}[10]=1+2+5=8$ .）：

1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1
	2		2		2		2		
		3			3				
			4						
				5					

3. 然后一次遍历  $i$  从 220 开始到 5000000， $i$  每遍历一个数后，将  $i$  对应的数下面的各个真因子加起来得到一个和  $\text{sum}[i]$ ，如果这个和  $\text{sum}[i]==$  某个  $i'$ ，且  $\text{sum}[i']=i$ ，那么这两个数  $i$  和  $i'$ ，即为一对亲和数。
4.  $i=2$ ;  $\text{sum}[4]+=2$ ,  $\text{sum}[6]+=2$ ,  $\text{sum}[8]+=2$ ,  $\text{sum}[10]+=2$ ,  $\text{sum}[12]+=2\dots$   
 $i=3$ ,  $\text{sum}[6]+=3$ ,  $\text{sum}[9]+=3\dots$   
.....
5.  $i=220$  时,  $\text{sum}[220]=284$ ,  $i=284$  时,  $\text{sum}[284]=220$ ; 即  
 $\text{sum}[220]=\text{sum}[\text{sum}[284]]=284$ ,  
得出 220 与 284 是一对亲和数。所以，最终输出 220、284, ...

## 特别鸣谢

litaoye 专门为本亲和数问题开帖子继续阐述，有兴趣的朋友可继续参见：  
<http://topic.csdn.net/u/20110526/21/129c2235-1f44-42e9-a55f-878920c21e19.html>。同时，任何人对本亲和数问题有任何问题，也可以回复到上述帖子上。

```
//求解亲和数问题
//copyright@ litaoye
//July、胡滨, updated, 2011.05.26。
using System;
using System.Collections.Generic;

namespace CSharpTest
{
    class Program
    {
```

```

public static void Main()
{
    int max = 5000000;
    DateTime start = DateTime.Now;
    int[] counter = CreateCounter(max);

    for (int i = 0; i < counter.Length; i++)
    {
        int num = counter[i] - i;
        //if (num < counter.Length && num > i && counter[num] == counter[i])

        // Console.WriteLine("{0} {1}", i, num);
    }
    Console.WriteLine((DateTime.Now - start).TotalSeconds);

    Console.ReadKey();
}

static int[] CreateCounter(int n)
{
    List<int> primes = new List<int>();
    int[] counter = new int[n + 1];
    counter[1] = 1;

    for (int i = 2; i <= n; i++)
    {
        if (counter[i] == 0)
        {
            counter[i] = i + 1;
            primes.Add(i);
        }

        for (int j = 0; j < primes.Count; j++)
        {
            if (primes[j] * i > n)
                break;

            if (i % primes[j] == 0)
            {
                int k = i;
                int l = primes[j] * primes[j];

                while (k % primes[j] == 0)
                {
                    k /= primes[j];
                }
                counter[k] = l;
            }
        }
    }
}

```

```

        l *= primes[j];
        k /= primes[j];
    }

    counter[primes[j] * i] = counter[k] * (l - 1) / (primes[j]
- 1);
    break;
}
else
    counter[primes[j] * i] = counter[i] * (primes[j] + 1);
}
}

return counter;
}
}

/*
测试结果:
0.484375
0.484375
0.46875
单位 second。
*/

```

本章完。

## 面试题征集令

1. 十三个经典算法研究系列+附、红黑树系列（国内有史以来最为经典的红黑树教程），共计 20+6=26 篇文章，带目录+标签的 PDF 文档，耗时近一个星期，足足 346 页（够一本书的分量了），已在花明月暗的帮助下，正式制作完成。
2. 想要的，发一道你自认为较好的面试题（C, C++, 数据结构，算法，智力题，数字逻辑或运算题）至我的邮箱：[zhoulei0907@yahoo.cn](mailto:zhoulei0907@yahoo.cn)，即可。我收到后，三天之内传送此 PDF 文件。July、2011.0.5.24.此声明永久有效。

## 第七章、求连续子数组的最大和

作者：July。

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

# 前奏

- 希望更多的人能和我一样，把本狂想曲系列中的任何一道面试题当做一道简单的编程题或一个实质性的问题来看待，在阅读本狂想曲系列的过程中，希望你能尽量暂时放下所有有关面试的一切包袱，潜心攻克每一道“编程题”，在解决编程题的过程中，好好享受编程带来的无限乐趣，与思考带来的无限激情。--By@July\_\_\_\_\_。
- 原狂想曲系列已更名为：**程序员编程艺术系列**。原狂想曲创作组更名为**编程艺术室**。编程艺术室致力于以下三点工作：1、针对一个问题，不断寻找更高效的算法，并予以编程实现。2、解决实际中会碰到的应用问题，如**第十章、如何给 10^7 个数据量的磁盘文件排序**。3、经典算法的研究与实现。总体突出一点：编程，如何高效的编程解决实际问题。欢迎有志者加入。

## 第一节、求子数组的最大和

### 3.求子数组的最大和

题目描述：

输入一个整形数组，数组里有正数也有负数。

数组中连续的一个或多个整数组成一个子数组，每个子数组都有一个和。

求所有子数组的和的最大值。要求时间复杂度为  $O(n)$ 。

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

**分析：**这个问题在各大公司面试中出现频率之频繁，被人引用次数之多，非一般面试题可与之匹敌。单凭这点，就没有理由不入选狂想曲系列中了。此题曾作为本人之前整理的微软 100 题中的第 3 题，至今反响也很大。ok，下面，咱们来一步一步分析这个题：

1、求一个数组的最大子数组和，如此序列 1, -2, 3, 10, -4, 7, 2, -5，我想最最直观也是最野蛮的办法便是，三个 for 循环三层遍历，求出数组中每一个子数组的和，最终求出这些子数组的最大的一个值。

记  $\text{Sum}[i, \dots, j]$  为数组 A 中第 i 个元素到第 j 个元素的和（其中  $0 \leq i \leq j < n$ ），遍历所有可能的  $\text{Sum}[i, \dots, j]$ ，那么时间复杂度为  $O(N^3)$ ：

//本段代码引自编程之美

```
int MaxSum(int* A, int n)
{
    int maximum = -INF;
```

```

int sum=0;
for(int i = 0; i < n; i++)
{
    for(int j = i; j < n; j++)
    {
        for(int k = i; k <= j; k++)
        {
            sum += A[k];
        }
        if(sum > maximum)
            maximum = sum;
    }
}

sum=0; //这里要记得清零，否则的话 sum 最终存放的是所有子数组的和。也就是编程之美上所说的 bug。多谢苍狼。
}
}
return maximum;
}

```

**2、**其实这个问题，在我之前上传的微软 100 题，答案 V0.2 版[第 1-20 题答案]，便直接给出了以下  $O(N)$  的算法：

```

//copyright@ July 2010/10/18
//updated, 2011.05.25.
#include <iostream.h>

int maxSum(int* a, int n)
{
    int sum=0;
    //其实要处理全是负数的情况，很简单，如稍后下面第 3 点所见，直接把这句改成："int sum=a[0]" 即可
    //也可以不改，当全是负数的情况，直接返回 0，也不见得不行。
    int b=0;

    for(int i=0; i<n; i++)
    {
        if(b<0)          //...
            b=a[i];
        else
            b+=a[i];
    }
}

```

```

    if(sum<b)
        sum=b;
    }
    return sum;
}

int main()
{
    int a[10]={1, -2, 3, 10, -4, 7, 2, -5};
    //int a[]={-1,-2,-3,-4}; //测试全是负数的用例
    cout<<maxSum(a,8)<<endl;
    return 0;
}

```

/\*-----

解释下：

例如输入的数组为 1, -2, 3, 10, -4, 7, 2, -5,

那么最大的子数组为 3, 10, -4, 7, 2,

因此输出为该子数组的和 18。

所有的东西都在以下俩行，

即：

b :	0	1	-1	3	13	9	16	18	13
sum:	0	1	1	3	13	13	16	18	18

其实算法很简单，当前面的几个数，加起来后， $b < 0$  后，

把  $b$  重新赋值，置为下一个元素， $b=a[i]$ 。

当  $b>sum$ ，则更新  $sum=b$ ；

若  $b<sum$ ，则  $sum$  保持原值，不更新。。July、10/31。

-----\*/

**3、**不少朋友看到上面的答案之后，认为上述思路 2 的代码，没有处理全是负数的情况，当全是负数的情况时，我们可以让程序返回 0，也可以让其返回最大的那个负数，下面便是前几日重写的，修改后的处理全是负数情况（返回最大的负数）的代码：

```

//copyright@ July
//July、updated, 2011.05.25。
#include <iostream.h>
#define n 4           //多定义了一个变量

int maxsum(int a[n])
//于此处，你能看到上述思路 2 代码（指针）的优势
{
    int max=a[0];      //全负情况，返回最大数

```

```

int sum=0;
for(int j=0;j<n;j++)
{
    if(sum>=0)      //如果加上某个元素, sum>=0 的话, 就加
        sum+=a[j];
    else
        sum=a[j];   //如果加上某个元素, sum<0 了, 就不加
    if(sum>max)
        max=sum;
}
return max;
}

int main()
{
    int a[]={-1,-2,-3,-4};
    cout<<maxsum(a)<<endl;
    return 0;
}

```

4、DP 解法的具体方程: @\_flyinghearts: 设  $sum[i]$  为前  $i$  个元素中, 包含第  $i$  个元素且和最大的连续子数组,  $result$  为已找到的子数组中和最大的。对第  $i+1$  个元素有两种选择: 做为新子数组的第一个元素、放入前面找到的子数组。

```

sum[i+1] = max(a[i+1], sum[i] + a[i+1])
result = max(result, sum[i])

```

扩展:

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

## 第二节、Data structures and Algorithm analysis in C

下面给出《Data structures and Algorithm analysis in C》中 4 种实现。

```

//感谢网友 firo
//July、2010.06.05。

```

```

//Algorithm 1:时间效率为 O(n*n*n)
int MaxSubsequenceSum1(const int A[],int N)
{
    int ThisSum=0 ,MaxSum=0,i,j,k;
    for(i=0;i<N;i++)
        for(j=i;j<N;j++)
    {
        ThisSum=0;
        for(k=i;k<j;k++)
            ThisSum+=A[k];

        if(ThisSum>MaxSum)
            MaxSum=ThisSum;
    }
    return MaxSum;
}

//Algorithm 2:时间效率为 O(n*n)
int MaxSubsequenceSum2(const int A[],int N)
{
    int ThisSum=0,MaxSum=0,i,j,k;
    for(i=0;i<N;i++)
    {
        ThisSum=0;
        for(j=i;j<N;j++)
        {
            ThisSum+=A[j];
            if(ThisSum>MaxSum)
                MaxSum=ThisSum;
        }
    }
    return MaxSum;
}

//Algorithm 3:时间效率为 O(n*log n)
//算法3的主要思想：采用二分策略，将序列分成左右两份。
//那么最长子序列有三种可能出现的情况，即
//【1】只出现在左部分。
//【2】只出现在右部分。
//【3】出现在中间，同时涉及到左右两部分。
//分情况讨论之。
static int MaxSubSum(const int A[],int Left,int Right)
{

```

```

int MaxLeftSum,MaxRightSum;           //左、右部分最大连续子序列值。对应情况【1】。
【2】
int MaxLeftBorderSum,MaxRightBorderSum; //从中间分别到左右两侧的最大连续子序列值,
对应 case 【3】。
int LeftBorderSum,RightBorderSum;
int Center,i;
if(Left == Right)Base Case
    if(A[Left]>0)
        return A[Left];
    else
        return 0;
Center=(Left+Right)/2;
MaxLeftSum=MaxSubSum(A,Left,Center);
MaxRightSum=MaxSubSum(A,Center+1,Right);
MaxLeftBorderSum=0;
LeftBorderSum=0;
for(i=Center;i>=Left;i--)
{
    LeftBorderSum+=A[i];
    if(LeftBorderSum>MaxLeftBorderSum)
        MaxLeftBorderSum=LeftBorderSum;
}
MaxRightBorderSum=0;
RightBorderSum=0;
for(i=Center+1;i<=Right;i++)
{
    RightBorderSum+=A[i];
    if(RightBorderSum>MaxRightBorderSum)
        MaxRightBorderSum=RightBorderSum;
}
int max1=MaxLeftSum>MaxRightSum?MaxLeftSum:MaxRightSum;
int max2=MaxLeftBorderSum+MaxRightBorderSum;
return max1>max2?max1:max2;
}

//Algorithm 4:时间效率为 O(n)
//同上述第一节中的思路 3、和 4。
int MaxSubsequenceSum(const int A[],int N)
{
    int ThisSum,MaxSum,j;
    ThisSum=MaxSum=0;
    for(j=0;j<N;j++)
    {
        ThisSum+=A[j];

```

```
    if(ThisSum>MaxSum)
        MaxSum=ThisSum;
    else if(ThisSum<0)
        ThisSum=0;
    }
    return MaxSum;
}
```

本章完。

# 第八章、从头至尾漫谈虚函数

作者：July。

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

## 前奏

有关虚函数的问题层出不穷，有关虚函数的文章千篇一律，那为何还要写这一篇有关虚函数的文章呢？看完本文后，相信能懂其意义之所在。同时，原狂想曲系列已经更名为程序员编程艺术系列，因为不再只专注于“面试”，而在“编程”之上了。**ok**，如果有不正之处，望不吝赐教。谢谢。

## 第一节、一道简单的虚函数的面试题

题目要求：写出下面程序的运行结果？

```
//谢谢董天喆提供的这道百度的面试题
#include <iostream>
using namespace std;
class A{
public:virtual void p()
{
    cout << "A" << endl;
}
};

class B : public A
{
public:virtual void p()
{ cout << "B" << endl;
}
};

int main()
{
    A * a = new A;
```

```
A * b = new B;  
a->p();  
b->p();  
delete a;  
delete b;  
return 0;  
}
```

我想，这道面试题应该是考察虚函数相关知识的相对简单的一道题目了。然后，希望你碰到此类有关虚函数的面试题，不论其难度是难是易，都能够举一反三，那么本章的目的也就达到了。**ok**，请跟着我的思路，咱们步步深入（上面程序的输出结果为**A B**）。

## 第二节、有无虚函数的区别

1、当上述程序中的函数 **p()**不是虚函数，那么程序的运行结果是如何？即如下代码所示：

```
class A  
{  
public:  
void p()  
{  
cout << "A" << endl;  
}  
  
};  
  
class B : public A  
{  
public:  
void p()  
{  
cout << "B" << endl;  
}  
};
```

对的，程序此时将输出两个 **A**，**A**。为什么？

我们知道，在构造一个类的对象时，如果它有基类，那么首先将构造基类的对象，然后才构造派生类自己的对象。如上，`A* a=new A;` 调用默认构造函数构造基类 A 对象，然后调用函数 `p()`, `a->p();`输出 A，这点没有问题。

然后，`A * b = new B;` 构造了派生类对象 B，B 由于是基类 A 的派生类对象，所以会先构造基类 A 对象，然后再构造派生类对象，但由于当程序中函数是非虚函数调用时，B 类对象对函数 p() 的调用时在编译时就已静态确定了，所以，不论基类指针 b 最终指向的是基类对象还是派生类对象，只要后面的对象调用的函数不是虚函数，那么就直接无视，而调用基类 A 的 p() 函数。

**2、那如果加上虚函数呢？即如最开始的那段程序那样，程序的输出结果，将是什么？**  
在此之前，我们还得明确以下两点：

**a、通过基类引用或指针调用基类中定义的函数时，我们并不知道执行函数的对象的确切类型，执行函数的对象可能是基类类型的，也可能是派生类型的。**

**b、如果调用非虚函数，则无论实际对象是什么类型，都执行基类类型所定义的函数（如上述第 1 点所述）。如果调用虚函数，则直到运行时才能确定调用哪个函数，运行的虚函数是引用所绑定的或指针所指向的对象所属类型定义的版本。**

根据上述 b 的观点，我们知道，如果加上虚函数，如上面这道面试题，

```
class A
```

```
{
```

```
public:
```

```
virtual void p()
```

```
{
```

```
    cout << "A" << endl;
```

```
}
```

```
};
```

```
class B : public A
```

```
{
```

```
public:
```

```
virtual void p()
```

```
{
```

```
cout << "B" << endl;
}

};

int main()
{
    A * a = new A;
    A * b = new B;
    a->p();
    b->p();
    delete a;
    delete b;
    return 0;
}
```

那么程序的输出结果将是 A B。

所以，至此，咱们的这道面试题已经解决。但虚函数的问题，还没有解决。

### 第三节、虚函数的原理与本质

我们已经知道，虚(**virtual**)函数的一般实现模型是：每一个类(**class**)有一个虚表(**virtual table**)，内含该**class**之中有用的虚(**virtual**)函数的地址，然后每个对象有一个 **vptr**，指向虚表(**virtual table**)的所在。

请允许我援引自深度探索 C++ 对象模型一书上的一个例子：

```
class Point {
public:
    virtual ~Point();

    virtual Point& mult( float ) = 0;

    float x() const { return _x; }    //非虚函数，不作存储
    virtual float y() const { return 0; }
```

```
virtual float z() const { return 0; }
// ...
```

protected:

```
Point( float x = 0.0 );
float _x;
};
```

**1、在 Point 的对象 pt 中，有两个东西，一个是数据成员\_x，一个是\_vptr\_Point。其中\_vptr\_Point 指向着 virtual table point，而 virtual table (虚表) point 中存储着以下东西：**

- virtual ~Point() 被赋值 slot 1,
- mult() 将被赋值 slot 2.
- y() is 将被赋值 slot 3
- z() 将被赋值 slot 4.

```
class Point2d : public Point {
public:
    Point2d( float x = 0.0, float y = 0.0 )
        : Point( x ), _y( y ) {}
    ~Point2d(); //1

    //改写 base class virtual functions
    Point2d& mult( float ); //2
    float y() const { return _y; } //3
```

protected:

```
float _y;
};
```

**2、在 Point2d 的对象 pt2d 中，有三个东西，首先是继承自基类 pt 对象的数据成员\_x，然后是 pt2d 对象本身的数据成员\_y，最后是\_vptr\_Point。其中\_vptr\_Point 指向着 virtual table point2d。由于 Point2d 继承自 Point，所以在 virtual table point2d 中存储着：改写的其中的~Point2d()、Point2d& mult( float )、float y() const，以及未被改写的 Point::z() 函数。**

```
class Point3d: public Point2d {
public:
```

```

Point3d( float x = 0.0,
          float y = 0.0, float z = 0.0 )
: Point2d( x, y ), _z( z ) {}
~Point3d();

// overridden base class virtual functions
Point3d& mult( float );
float z() const { return _z; }

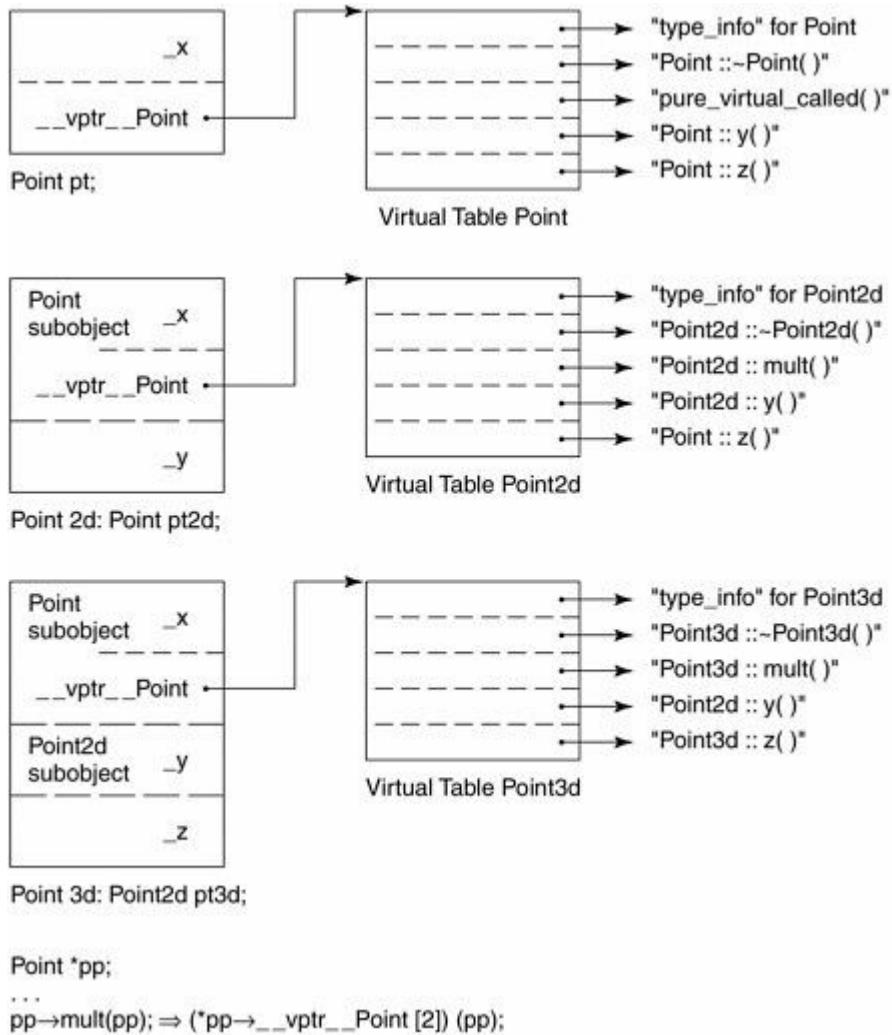
// ... other operations ...

protected:
    float _z;
};

```

**3、**在 Point3d 的对象 pt3d 中，则有四个东西，一个是\_x，一个是\_vptr\_Point，一个是\_y，一个是\_z。其中\_vptr\_Point 指向着 virtual table point3d。由于 point3d 继承自 point2d，所以在 virtual table point3d 中存储着：已经改写了的 point3d 的~Point3d()，point3d::mult() 的函数地址，和 z()函数的地址，以及未被改写的 point2d 的 y()函数地址。

ok，上述 1、2、3 所有情况的详情，请参考下图。



(图: virtual table (虚表) 的布局: 单一继承情况)

本文，日后可能会酌情考虑增补有关内容。ok，更多，可参考深度探索 C++ 对象模型一书第四章。

最近几章难度都比较小，是考虑到狂想曲有深有浅的原则，后续章节会逐步恢复到相应难度。

## 第四节、虚函数的布局与汇编层面的考察

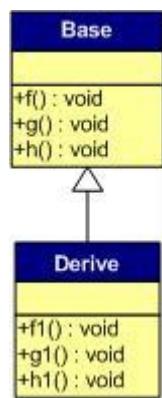
ivan、老梦的两篇文章继续对虚函数进行了一番深入，我看他们已经写得很好了，我不饶舌了。ok，请看：1、VC 虚函数布局引发的问题，2、从汇编层面深度剖析 C++ 虚函数、<http://blog.csdn.net/linty/archive/2011/04/20/6336762.aspx>。

## 第五节、虚函数表的详解

本节全部内容来自淄博的共享，非常感谢。注@molixiaogemao：只有发生继承的时候且父类子类都有 **virtual** 的时候才会出现虚函数指针，请不要忘了虚函数出现的目的是为了实现多态。

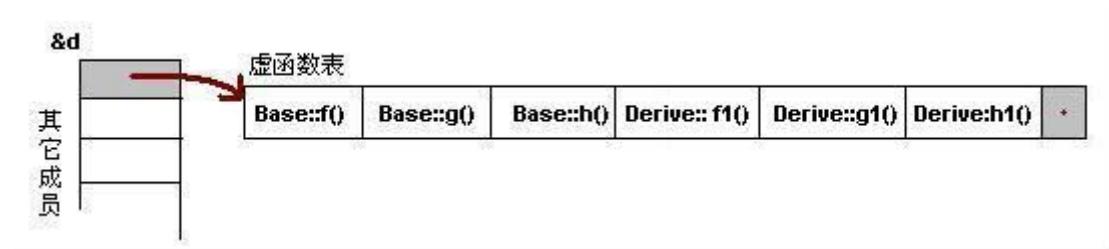
### 一般继承（无虚函数覆盖）

下面，再让我们来看看继承时的虚函数表是什么样的。假设有如下所示的一个继承关系：



请注意，在这个继承关系中，子类没有重载任何父类的函数。那么，在派生类的实例中，

对于实例：Derive d; 的虚函数表如下：



我们可以看到下面几点：

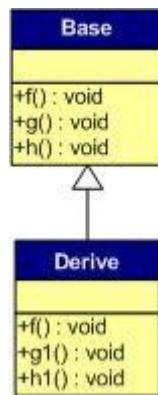
- 1) 虚函数按照其声明顺序放于表中。
- 2) 父类的虚函数在子类的虚函数前面。

我相信聪明的你一定可以参考前面的那个程序，来编写一段程序来验证。

## 一般继承（有虚函数覆盖）

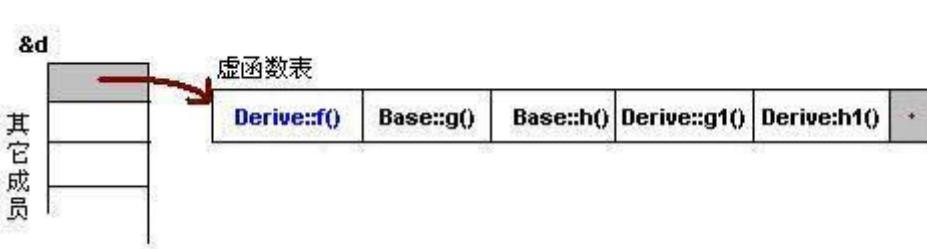
覆盖父类的虚函数是很显然的事情，不然，虚函数就变得毫无意义。

下面，我们来看一下，如果子类中有虚函数重载了父类的虚函数，会是一个什么样子？假设，我们有下面这样的一个继承关系。



为了让大家看到被继承过后的效果，在这个类的设计中，我只覆盖了父类的一个函数：**f()**。

那么，对于派生类的实例，其虚函数表会是下面的一个样子：



我们从表中可以看到下面几点，

- 1) 覆盖的 **f()** 函数被放到了虚表中原来父类虚函数的位置。
- 2) 没有被覆盖的函数依旧。

这样，我们就可以看到对于下面这样的程序，

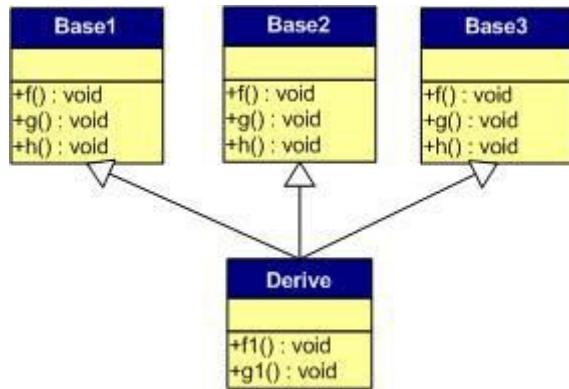
```
Base *b = new Derive();
```

```
b->f();
```

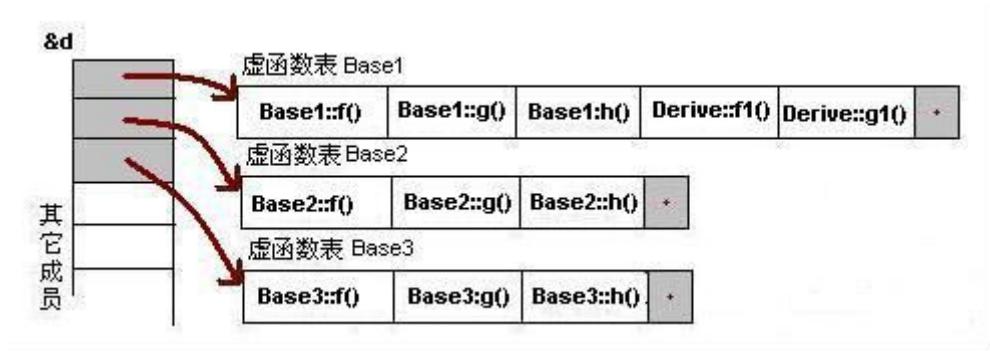
由 **b** 所指的内存中的虚函数表的 **f()** 的位置已经被 **Derive::f()** 函数地址所取代，于是在实际调用发生时，是 **Derive::f()** 被调用了。这就实现了多态。

## 多重继承（无虚函数覆盖）

下面，再让我们来看看多重继承中的情况，假设有下面这样一个类的继承关系（注意：子类并没有覆盖父类的函数）：



对于子类实例中的虚函数表，是下面这个样子：



我们可以看到：

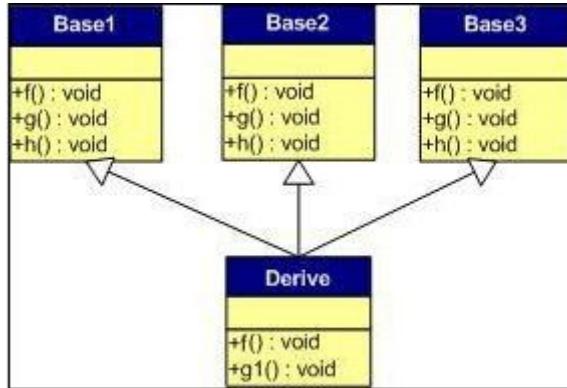
- 1) 每个父类都有自己的虚表。
- 2) 子类的成员函数被放到了第一个父类的表中。（所谓的第一个父类是按照声明顺序来判断的）

这样做就是为了解决不同的父类类型的指针指向同一个子类实例，而能够调用到实际的函数。

### 多重继承（有虚函数覆盖）

下面我们再来看看，如果发生虚函数覆盖的情况。

下图中，我们在子类中覆盖了父类的 f() 函数。



下面是对于子类实例中的虚函数表的图：



我们可以看见，三个父类虚函数表中的 f() 的位置被替换成了子类的函数指针。

这样，我们就可以任一静态类型的父类来指向子类，并调用子类的 f() 了。如：

```

Derive d;
Base1 *b1 = &d;
Base2 *b2 = &d;
Base3 *b3 = &d;
b1->f(); //Derive::f()
b2->f(); //Derive::f()
b3->f(); //Derive::f()
b1->g(); //Base1::g()
b2->g(); //Base2::g()
b3->g(); //Base3::g()

```

## 安全性

每次写 C++ 的文章，总免不了要批判一下 C++。

这篇文章也不例外。通过上面的讲述，相信我们对虚函数表有一个比较细致的了解了。

水可载舟，亦可覆舟。下面，让我们来看看我们可以用虚函数表来干点什么坏事吧。

### 一、通过父类型的指针访问子类自己的虚函数

我们知道，子类没有重载父类的虚函数是一件毫无意义的事情。因为多态也是要基于函数重载的。

虽然在上面的图中我们可以看到 Base1 的虚表中有 Derive 的虚函数，但我们根本不可能使用下面的语句来调用子类的自有虚函数：

```
Base1 *b1 = new Derive();
b1->g1(); //编译出错
```

任何妄图使用父类指针想调用子类中的未覆盖父类的成员函数的行为都会被编译器视为非法，即基类指针不能调用子类自己定义的成员函数。所以，这样的程序根本无法编译通过。

但在运行时，我们可以通过指针的方式访问虚函数表来达到违反 C++ 语义的行为。

（关于这方面的尝试，通过阅读后面附录的代码，相信你可以做到这一点）

### 二、访问 non-public 的虚函数

另外，如果父类的虚函数是 private 或是 protected 的，但这些非 public 的虚函数同样会存在于虚函数表中，

所以，我们同样可以使用访问虚函数表的方式来访问这些 non-public 的虚函数，这是很容易做到的。

如：

```
class Base {
private:
    virtual void f() { cout << "Base::f" << endl; }
};

class Derive : public Base{
};

typedef void(*Fun)(void);
void main() {
    Derive d;
    Fun pFun = (Fun)*((int*)*(int*)(&d)+0);
```

```
pFun();  
}
```

对上面粗体部分的解释 (@a && x) :

1. (int\*)(&d) 取 vptr 地址，该地址存储的是指向 vtbl 的指针
2. (int\*)\*(int\*)(&d) 取 vtbl 地址，该地址存储的是虚函数表数组
3. (Fun)\*((int\*)\*(int\*)(&d)+0)，取 vtbl 数组的第一个元素，即 Base 中第一个虚函数 f 的地址
4. (Fun)\*((int\*)\*(int\*)(&d)+1)，取 vtbl 数组的第二个元素（这第 4 点，如下图所示）。

下图也能很清晰的说明一些东西 (@5) :

```
#include <iostream>  
using namespace std;  
class Base {  
private:  
    virtual void f() { cout << "Base::f" << endl; }  
    virtual void g() { cout << "Base::g" << endl; }  
};  
class Derive : public Base{  
};  
class DeriveChild : public Derive{  
};  
typedef void(*Fun)(void);  
void main() {  
    DeriveChild d;  
    Fun pFun = (Fun)*((int*)*(int*)(int*)(&d)+0);  
    pFun();  
    pFun = (Fun)*((int*)*(int*)(int*)(&d)+1);  
    pFun();  
}
```

C:\WINDOWS\system32  
Base::f  
Base::g  
请按任意键继续... . .

ok，再来看一个问题，如果一个子类重载的虚拟函数为 private，那么通过父类的指针可以访问到它吗？

```
#include <Iostream>  
class B
```

```

{
public:
    virtual void fun()
    {
        std::cout << "base fun called";
    };
};

class D : public B
{
private:
    virtual void fun()
    {
        std::cout << "driver fun called";
    };
};

int main(int argc, char* argv[])
{
    B* p = new D();
    p->fun();
    return 0;
}

```

运行时会输出 **driver fun called**

从这个实验，可以更深入的了解虚拟函数编译时的一些特征：

在编译虚拟函数调用的时候，例如 `p->fun();` 只是按其静态类型来处理的，在这里 `p` 的类型就是 `B`，不会考虑其实际指向的类型（动态类型）。

也就是说，碰到 `p->fun();` 编译器就当作调用 `B` 的 `fun` 来进行相应的检查和处理。

因为在 `B` 里 `fun` 是 `public` 的，所以这里在“访问控制检查”这一关就完全可以通过了。

然后就会转换成`(*p->vptr[1])(p)`这样的方式处理，`p` 实际指向的动态类型是 `D`，

所以 `p` 作为参数传给 `fun` 后（类的非静态成员函数都会编译加一个指针参数，指向调用该函数的对象，我们平常用的 `this` 就是该指针的值），实际运行时 `p->vptr[1]` 则获取到的是 `D::fun()` 的地址，也就调用了该函数，这也就是动态运行的机理。

为了进一步的实验，可以将 B 里的 fun 改为 private 的，D 里的改为 public 的，则编译就会出错。

C++的注意条款中有一条“绝不重新定义继承而来的缺省参数值”  
(Effective C++ Item37, never redefine a function's inherited default parameter value) 也是同样的道理。

可以再做个实验

```
class B
{
public:
    virtual void fun(int i = 1)
    {
        std::cout << "base fun called, " << i;
    }
};

class D : public B
{
private:
    virtual void fun(int i = 2)
    {
        std::cout << "driver fun called, " << i;
    }
};
```

则运行会输出 **driver fun called, 1**

关于这一点，Effective 上讲的很清楚“virtual 函数系动态绑定，而缺省参数却是静态绑定”，也就是说在编译的时候已经按照 p 的静态类型处理其默认参数了，转换成了(\*p->vptr[1])(p, 1)这样的方式。

## 补遗

一个类如果有虚函数，不管是几个虚函数，都会为这个类声明一个虚函数表，这个虚表是一个含有虚函数的类的，不是说是类对象的。一个含有虚函数的类，不管有多少个数据成员，每个对象实例都有一个虚指针，在内存中，存放每个类对象的内存区，在内存区的头部都是先存放这个指针变量的（准确的说，应该是：视编译器具体情况而定），从第  $n$  ( $n$  视实际情况而定) 个字节才是这个对象自己的东西。

下面再说下通过基类指针，调用虚函数所发生的一切：

```
One *p;  
p->disp();
```

1、上来要取得类的虚表的指针，就是要得到，虚表的地址。存放类对象的内存区的前四个字节其实就是用来存放虚表的地址的。

2、得到虚表的地址后，从虚表那知道你调用的那个函数的入口地址。根据虚表提供的你要找的函数的地址。并调用函数；你要知道，那个虚表是一个存放指针变量的数组，并不是说，那个虚表中就是存放的虚函数的实体。

本章完。

# 第九章、闲话链表追趕问题

作者：July、狂想曲创作组。

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

## 前奏

有这样一个问题：在一条左右水平放置的直线轨道上任选两个点，放置两个机器人，请用如下指令系统为机器人设计控制程序，使这两个机器人能够在直线轨道上相遇。（注意两个机器人用你写的同一个程序来控制）。

指令系统：只包含 4 条指令，向左、向右、条件判定、无条件跳转。其中向左（右）指令每次能控制机器人向左（右）移动一步；条件判定指令能对机器人所在的位置进行条件测试，测试结果是如果对方机器人曾经到过这里就返回 `true`，否则返回 `false`；无条件跳转，类似汇编里面的跳转，可以跳转到任何地方。

ok，这道很有意思的趣味题是去年微软工程院的题，文末将给出解答（如果急切想知道此问题的答案，可以直接跳到本文第三节）。同时，我们看到其实这个题是一个典型的追趕问题，那么追趕问题在哪种面试题中比较常见？对了，链表追趕。本章就来阐述这个问题。有不正之处，望不吝指正。

## 第一节、求链表倒数第 k 个结点

### 第 13 题、题目描述：

输入一个单向链表，输出该链表中倒数第 k 个结点，  
链表的倒数第 0 个结点为链表的尾指针。

分析：此题一出，相信，稍微有点 经验的同志，都会说到：设置两个指针 `p1,p2`，首先 `p1` 和 `p2` 都指向 `head`，然后 `p2` 向前走 `k` 步，这样 `p1` 和 `p2` 之间就间隔 `k` 个节点，最后 `p1` 和 `p2` 同时向前移动，直至 `p2` 走到链表末尾。

前几日有朋友提醒我说，让我讲一下此种求链表倒数第 `k` 个结点的问题。我想，这种问题，有点经验的人恐怕都已了解过，无非是利用两个指针一前一后逐步前移。但他提醒我说，如果参加面试的人没有这个意识，它怎么也想不到那里去。

那在平时准备面试的过程中如何加强这一方面的意识呢？我想，除了平时遇到一道面试题，尽可能用多种思路解决，以延伸自己的视野之外，便是平时有意注意观察生活。因为，相信，你很容易了解到，其实这种链表追赶的问题来源于生活中长跑比赛，如果平时注意多多思考，多多积累，多多发现并体味生活，相信也会对面试有所帮助。

ok，扯多了，下面给出这个题目的主体代码，如下：

```
struct ListNode
{
    char data;
    ListNode* next;
};

ListNode* head,*p,*q;
ListNode *pone,*ptwo;

//@heyaming, 第一节,求链表倒数第 k 个结点应该考虑 k 大于链表长度的 case。
ListNode* fun(ListNode *head,int k)
{
    assert(k >= 0);
    pone = ptwo = head;
    for( ; k > 0 && ptwo != NULL; k--)
        ptwo=ptwo->next;
    if (k > 0) return NULL;

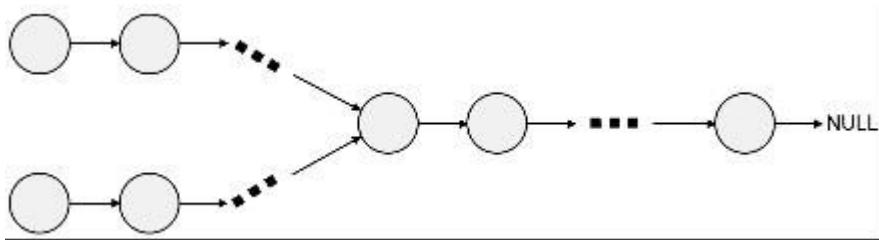
    while(ptwo!=NULL)
    {
        pone=pone->next;
        ptwo=ptwo->next;
    }
    return pone;
}
```

### 扩展：

这是针对链表单项链表查找其中倒数第  $k$  个结点。试问，如果链表是双向的，且可能存在环呢？请看第二节、编程判断两个链表是否相交。

## 第二节、编程判断两个链表是否相交

题目描述：给出两个单向链表的头指针（如下图所示），



比如  $h1$ 、 $h2$ ，判断这两个链表是否相交。这里为了简化问题，我们假设两个链表均不带环。

分析：这是来自编程之美上的微软亚院的一道面试题目。请跟着我的思路步步深入（部分文字引自编程之美）：

1. 直接循环判断第一个链表的每个节点是否在第二个链表中。但，这种方法的时间复杂度为  $O(\text{Length}(h1) * \text{Length}(h2))$ 。显然，我们得找到一种更为有效的方法，至少不能是  $O(N^2)$  的复杂度。
2. 针对第一个链表直接构造 **hash** 表，然后查询 **hash** 表，判断第二个链表的每个结点是否在 **hash** 表出现，如果所有的第二个链表的结点都能在 **hash** 表中找到，即说明第二个链表与第一个链表有相同的结点。时间复杂度为线性： $O(\text{Length}(h1) + \text{Length}(h2))$ ，同时为了存储第一个链表的所有节点，空间复杂度为  $O(\text{Length}(h1))$ 。是否还有更好的方法呢，既能够以线性时间复杂度解决问题，又能减少存储空间？
3. 进一步考虑“如果两个没有环的链表相交于某一节点，那么在这个节点之后的所有节点都是两个链表共有的”这个特点，我们可以知道，如果它们相交，则最后一个节点一定是共有的。而我们很容易能得到链表的最后一个节点，所以这成了我们简化解法的一个主要突破口。那么，我们只要判断两个链表的尾指针是否相等。相等，则链表相交；否则，链表不相交。

所以，先遍历第一个链表，记住最后一个节点。然后遍历第二个链表，到最后一个节点时和第一个链表的最后一个节点做比较，如果相同，则相交，否则，不相交。这样我们就得到了一个时间复杂度，它为  $O(\text{Length}(h1) + \text{Length}(h2))$ ，而且只用了一个额外的指针来存储最后一个节点。这个方法时间复杂度为线性  $O(N)$ ，空间复杂度为  $O(1)$ ，显然比解法三更胜一筹。

4. 上面的问题都是针对链表无环的，那么如果现在，链表是有环的呢？还能找到最后一个结点进行判断么？上面的方法还同样有效么？显然，这个问题的本质已经转化为判断链表是否有环。那么，如何来判断链表是否有环呢？

## 总结：

所以，事实上，这个判断两个链表是否相交的问题就转化成了：

- 1.先判断带不带环
  - 2.如果都不带环，就判断尾节点是否相等
  - 3.如果都带环，判断一链表上俩指针相遇的那个节点，在不在另一条链表上。
- 如果在，则相交，如果不相交，则不相交。

**1、**那么，如何编写代码来判断链表是否有环呢？因为很多时候，你给出了问题的思路后，面试官可能还要追加你的代码，**ok**，如下（设置两个指针(**p1, p2**)，初始值都指向头，**p1**每次前进一步，**p2**每次前进二步，如果链表存在环，则**p2**先进入环，**p1**后进入环，两个指针在环中走动，必定相遇）：

```
//copyright@ Kurtwang
//July、2011.05.27。
struct Node
{
    int value;
    Node * next;
};

//1.先判断带不带环
//判断是否有环，返回 bool，如果有环，返回环里的节点
//思路：用两个指针，一个指针步长为 1，一个指针步长为 2，判断链表是否有环
bool isCircle(Node * head, Node *& circleNode, Node *& lastNode)
{
    Node * fast = head->next;
    Node * slow = head;
    while(fast != slow && fast && slow)
    {
        if(fast->next != NULL)
            fast = fast->next;

        if(fast->next == NULL)
            lastNode = fast;
        if(slow->next == NULL)
            lastNode = slow;

        fast = fast->next;
        slow = slow->next;
    }
}
```

```

    }
    if(fast == slow && fast && slow)
    {
        circleNode = fast;
        return true;
    }
    else
        return false;
}

```

**2&3**、如果都不带环，就判断尾节点是否相等，如果都带环，判断一链表上俩指针相遇的那个节点，在不在另一条链表上。下面是综合解决这个问题的代码：

```

//判断带环不带环时链表是否相交
//2.如果都不带环，就判断尾节点是否相等
//3.如果都带环，判断一链表上俩指针相遇的那个节点，在不在另一条链表上。
bool detect(Node * head1, Node * head2)
{
    Node * circleNode1;
    Node * circleNode2;
    Node * lastNode1;
    Node * lastNode2;

    bool isCircle1 = isCircle(head1,circleNode1, lastNode1);
    bool isCircle2 = isCircle(head2,circleNode2, lastNode2);

    //一个有环，一个无环
    if(isCircle1 != isCircle2)
        return false;
    //两个都无环，判断最后一个节点是否相等
    else if(!isCircle1 && !isCircle2)
    {
        return lastNode1 == lastNode2;
    }
    //两个都有环，判断环里的节点是否能到达另一个链表环里的节点
    else
    {
        Node * temp = circleNode1->next; //updated, 多谢苍狼 and hyy.
        while(temp != circleNode1)
        {
            if(temp == circleNode2)
                return true;
            temp = temp->next;
        }
    }
}

```

```
    return false;
}

return false;
}
```

## 扩展 2：求两个链表相交的第一个节点

思路：在判断是否相交的过程中要分别遍历两个链表，同时记录下各自长度。

@Joshua：这个算法需要处理一种特殊情况，即：其中一个链表的头结点在另一个链表的环中，且不是环入口结点。这种情况有两种意思：1)如果其中一个链表是循环链表，则另一个链表必为循环链表，即两个链表重合但头结点不同；2)如果其中一个链表存在环(除去循环链表这种情况)，则另一个链表必在此环中与此环重合，其头结点为环中的一个结点，但不是入口结点。在这种情况下我们约定，如果链表 B 的头结点在链表 A 的环中，且不是环入口结点，那么链表 B 的头结点即作为 A 和 B 的第一个相交结点；如果 A 和 B 重合(定义方法时形参 A 在 B 之前)，则取 B 的头结点作为 A 和 B 的第一个相交结点。

@风过无痕：读《程序员编程艺术》，补充代码 2012 年 7 月 18 日 周三下午 10:15

发件人："風過無痕" <luxiaoxun001@qq.com>将发件人添加到联系人

收件人："zhoulei0907" <zhoulei0907@yahoo.cn>

你好

看到你在 csdn 上博客，学习了很多，看到下面一章，有个扩展问题没有代码，发现自己有个，发给你吧，思路和别人提出来的一样，感觉有代码更加完善一些，呵呵

## 扩展 2：求两个链表相交的第一个节点

思路：如果两个尾结点是一样的，说明它们有重合；否则两个链表没有公共的结点。

在上面的思路中，顺序遍历两个链表到尾结点的时候，我们不能保证在两个链表上同时到达尾结点。这是因为两个链表不一定长度一样。但如果假设一个链表比另一个长 L 个结点，我们先在长的链表上遍历 L 个结点，之后再同步遍历，这个时候我们就能保证同时到达最后一个结点了。由于两个链表从第一个公共结点开始到链表的尾结点，这一部分是重合的。因此，它们肯定也是同时到达第一公共结点的。于是在遍历中，第一个相同的结点就是第一个公共的结点。

在这个思路中，我们先要分别遍历两个链表得到它们的长度，并求出两个长度之差。在长的链表上先遍历若干次之后，再同步遍历两个链表，直到找到相同的结点，或者一直到链表结束。PS：没有处理一种特殊情况：就是一个是循环链表，而另一个也是，只是头结点所在位置不一样。

代码如下：

```
ListNode* FindFirstCommonNode( ListNode *pHead1, ListNode *pHead2)
{
    // Get the length of two lists
    unsigned int nLength1 = ListLength(pHead1);
    unsigned int nLength2 = ListLength(pHead2);
    int nLengthDif = nLength1 - nLength2;

    // Get the longer list
    ListNode *pListHeadLong = pHead1;
    ListNode *pListHeadShort = pHead2;
    if(nLength2 > nLength1)
    {
        pListHeadLong = pHead2;
        pListHeadShort = pHead1;
        nLengthDif = nLength2 - nLength1;
    }

    // Move on the longer list
    for(int i = 0; i < nLengthDif; ++ i)
        pListHeadLong = pListHeadLong->m_pNext;

    // Move on both lists
    while((pListHeadLong != NULL) && (pListHeadShort != NULL) && (pListHeadLong != pListHeadShort))
    {
        pListHeadLong = pListHeadLong->m_pNext;
        pListHeadShort = pListHeadShort->m_pNext;
    }

    // Get the first common node in two lists
    ListNode *pFirstCommonNode = NULL;
    if(pListHeadLong == pListHeadShort)
        pFirstCommonNode = pListHeadLong;

    return pFirstCommonNode;
}

unsigned int ListLength(ListNode* pHead)
{
    unsigned int nLength = 0;
    ListNode* pNode = pHead;
    while(pNode != NULL)
```

```

{
    ++ nLength;
    pNode = pNode->m_pNext;
}
return nLength;
}

```

关于判断单链表是否相交的问题，还可以看看此篇文章：

<http://www.cppblog.com/humanchao/archive/2008/04/17/47357.html>。ok，下面，回到本章前奏部分的那道非常有趣味的智力题。

## 第三节、微软工程院面试智力题

### 题目描述：

在一条左右水平放置的直线轨道上任选两个点，放置两个机器人，请用如下指令系统为机器人设计控制程序，使这两个机器人能够在直线轨道上相遇。（注意两个机器人用你写的同一个程序来控制）

指令系统：只包含 4 条指令，向左、向右、条件判定、无条件跳转。其中向左（右）指令每次能控制机器人向左（右）移动一步；条件判定指令能对机器人所在的位置进行条件测试，测试结果是如果对方机器人曾经到过这里就返回 true，否则返回 false；无条件跳转，类似汇编里面的跳转，可以跳转到任何地方。

**分析：**我尽量以最清晰的方式来说明这个问题（大部分内容来自 ivan, big 等人的讨论）：

1、首先题目要求很简单，就是要想办法让 A 最终能赶上 B，A 在后，B 在前，都向右移动，如果它们的速度永远一致，那 A 是永远无法追赶上 B 的。但题目给出了一个条件判断指令，即如果 A 或 B 某个机器人向前移动时，若是某个机器人经过的点是第二个机器人曾经经过的点，那么程序返回 true。对的，就是抓住这一点，A 到达曾经 B 经过的点后，发现此后的路是 B 此前经过的，那么 A 开始提速两倍，B 一直保持原来的一倍速度不变，那样的话，A 势必会在 $|AB|/move_right$  个单位时间内，追上 B。ok，简单伪代码如下：

```

start:
if(at the position other robots have not reached)
    move_right
if(at the position other robots have reached)
    move_right
    move_right
goto start

```

再简单解释下上面的伪代码（@big）：

A-----B

| |

在 A 到达 B 点前，两者都只有第一条 if 为真，即以相同的速度向右移动，在 A 到达 B 后，A 只满足第二个 if，即以两倍的速度向右移动，B 依然只满足第一个 if，则速度保持不变，经过  $|AB|/move\_right$  个单位时间，A 就可以追上 B。

**2、**有个细节又出现了，正如 ivan 所说，

```
if(at the position other robots have reached)
    move_right
    move_right
```

上面这个分支不一定能提速的。why? 因为如果 if 条件花的时间很少，而 move 指令发的时间很大（实际很可能是这样），那么两个机器人的速度还是基本是一样的。

那作如何修改呢？：

```
start:
if(at the position other robots have not reached)
    move_right
    move_left
    move_right
if(at the position other robots have reached)
    move_right
goto start
```

-----

这样改后，A 的速度应该比 B 快了。

**3、**然要是说每个指令处理速度都很快，AB 岂不是一直以相同的速度右移了？那到底该作何修改呢？请看：

```
go_step()
{
    向右
    向左
```

```
    向右  
}  
-----
```

三个时间单位才向右一步

```
go_2step()  
{  
    向右  
}
```

一个时间单向右一步向左和向右花的时间是同样的，并且会占用一定时间。如果条件判定指令时间比移令花的时间较少的话，应该上面两种步法，后者比前者快。至此，咱们的问题已经得到解决。

最后，感谢蜜蜂提供的这道有意思的面试题：



hi,

看到了你在群里的悬赏，我有一个面试题，是去年微软工程院的题

描述：

在一条左右放置的直线轨道上任选两个点，放置两个机器人，请用如下指令系统为机器人设计控制程序，使这两个机器人能够在直线轨道上相遇。  
(注意两个机器人用你写的同一个程序来控制)

指令系统：只包含4条指令，向左、向右、条件判定、无条件跳转。其中向左(右)指令每次能控制机器人向左(右)移动一步；条件判定指令能对机器人所在的位置进行条件测试，测试结果是如果对方机器人曾经到过这里就返回true，否则返回false；无条件跳转，类似汇编里面的跳转，可以跳转到任何地方。

解决提示：利用变速，有问题可以随时与我联系。

本章完。

# 第十章、如何给 $10^7$ 个数据量的磁盘文件排序

作者:July, yansha, 5, 编程艺术室。

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

## 前奏

经过几天的痛苦沉思，最终决定，把原程序员面试题狂想曲系列正式更名为程序员编程艺术系列，同时，狂想曲创作组更名为编程艺术室。之所以要改名，我们考虑到三点：1、为面试服务不能成为我们最终或最主要的目的，2、我更愿把解答一道道面试题，ACM 题等各类程序设计题目过程，当做一种艺术来看待，3、艺术的提炼本身是一个非常非常艰难的过程，但我们乐意接受这个挑战。

ok，如果任何人对本编程艺术系列有任何意见，或发现了本编程艺术系列任何问题，漏洞，bug，欢迎随时提出，我们将虚心接受并感激不尽，以为他人创造更好的价值，更好的服务。

## 第一节、如何给磁盘文件排序

**问题描述:**

**输入：**给定一个文件，里面最多含有  $n$  个不重复的正整数（也就是说可能含有少于  $n$  个不重复正整数），且其中每个数都小于等于  $n$ ， $n=10^7$ 。

**输出：**得到按从小到大升序排列的包含所有输入的整数的列表。

**条件：**最多有大约 1MB 的内存空间可用，但磁盘空间足够。且要求运行时间在 5 分钟以下，10 秒为最佳结果。

**分析：**下面咱们来一步一步的解决这个问题，

**1、归并排序。**你可能会想到把磁盘文件进行归并排序，但题目要求你只有 1MB 的内存空间可用，所以，归并排序这个方法不行。

**2、位图方案。**熟悉位图的朋友可能会想到用位图来表示这个文件集合。例如正如编程珠

机一书上所述，用一个 20 位长的字符串来表示一个所有元素都小于 20 的简单的非负整数集合，边框用如下字符串来表示集合{1,2,3,5,8,13}：

0 1 1 1 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0

上述集合中各数对应的位置则置 1，没有对应的数的位置则置 0。

参考编程珠玑一书上的位图方案，针对我们的  $10^7$  个数据量的磁盘文件排序问题，我们可以这么考虑，由于每个 7 位十进制整数表示一个小于 1000 万的整数。我们可以使用一个具有 1000 万个位的字符串来表示这个文件，其中，当且仅当整数  $i$  在文件中存在时，第  $i$  位为 1。采取这个位图的方案是因为我们面对的这个问题的特殊性：1、输入数据限制在相对较小的范围内，2、数据没有重复，3、其中的每条记录都是单一的整数，没有任何其它与之关联的数据。

所以，此问题用位图的方案分为以下三步进行解决：

- 第一步，将所有的位都置为 0，从而将集合初始化为空。
- 第二步，通过读入文件中的每个整数来建立集合，将每个对应的位都置为 1。
- 第三步，检验每一位，如果该位为 1，就输出对应的整数。

经过以上三步后，产生有序的输出文件。令  $n$  为位图向量中的位数（本例中为 1000 0000），程序可以用伪代码表示如下：

```
//磁盘文件排序位图方案的伪代码
//copyright@ Jon Bentley
//July、updated, 2011.05.29.

//第一步，将所有的位都初始化为 0
for i ={0,...n}
    bit[i]=0;

//第二步，通过读入文件中的每个整数来建立集合，将每个对应的位都置为 1。
for each i in the input file
    bit[i]=1;

//第三步，检验每一位，如果该位为 1，就输出对应的整数。
for i={0...n}
    if bit[i]==1
        write i on the output file
```

上面只是为了简单介绍下位图算法的伪代码之抽象级描述。显然，咱们面对的问题，可不是这么简单。下面，我们试着针对这个要分两趟给磁盘文件排序的具体问题编写完整代码，如下。

```
//copyright@ yansha
//July、2010.05.30.
//位图方案解决 10^7 个数据量的文件的排序问题
//如果有重复的数据，那么只能显示其中一个 其他的将被忽略
#include <iostream>
#include <bitset>
#include <assert.h>
#include <time.h>
using namespace std;

const int max_each_scan = 5000000;

int main()
{
    clock_t begin = clock();
    bitset<max_each_scan> bit_map;
    bit_map.reset();

    // open the file with the unsorted data
    FILE *fp_unsort_file = fopen("data.txt", "r");
    assert(fp_unsort_file);
    int num;

    // the first time scan to sort the data between 0 - 4999999
    while (fscanf(fp_unsort_file, "%d ", &num) != EOF)
    {
        if (num < max_each_scan)
            bit_map.set(num, 1);
    }

    FILE *fp_sort_file = fopen("sort.txt", "w");
    assert(fp_sort_file);
    int i;

    // write the sorted data into file
    for (i = 0; i < max_each_scan; i++)
    {
        if (bit_map[i] == 1)
            fprintf(fp_sort_file, "%d ", i);
    }
}
```

```

// the second time scan to sort the data between 5000000 - 9999999
int result = fseek(fp_unsort_file, 0, SEEK_SET);
if (result)
    cout << "fseek failed!" << endl;
else
{
    bit_map.reset();
    while (fscanf(fp_unsort_file, "%d ", &num) != EOF)
    {
        if (num >= max_each_scan && num < 10000000)
        {
            num -= max_each_scan;
            bit_map.set(num, 1);
        }
    }
    for (i = 0; i < max_each_scan; i++)
    {
        if (bit_map[i] == 1)
            fprintf(fp_sort_file, "%d ", i + max_each_scan);
    }
}

clock_t end = clock();
cout<<"用位图的方法，耗时："<<endl;
cout << (end - begin) / CLK_TCK << "s" << endl;
fclose(fp_sort_file);
fclose(fp_unsort_file);
return 0;
}

```

而后测试了一下上述程序的运行时间，采取位图方案耗时 14s，即 14000ms：



```
用位图的方法，耗时：  
14000ms  
Press any key to continue...
```

本章中，生成大数据量（1000w）的程序如下，下文第二节的多路归并算法的 c++ 实现和第三节的磁盘文件排序的编程实现中，生成的 1000w 数据量也是用本程序产生的，且本章内生成的 1000w 数据量的数据文件统一命名为“**data.txt**”。

```
//purpose: 生成随机的不重复的测试数据
//copyright@ 2011.04.19 yansha
//1000w 数据量，要保证生成不重复的数据量，一般的程序没有做到。
//但，本程序做到了。
//July、2010.05.30。
#include <iostream>
#include <time.h>
#include <assert.h>
using namespace std;

const int size = 10000000;
int num[size];

int main()
{
    int n;
    FILE *fp = fopen("data.txt", "w");
    assert(fp);

    for (n = 1; n <= size; n++)
        //之前此处写成了 n=0;n<size。导致下面有一段小程序的测试数据出现了 0，特此订
        //正。
        num[n] = n;
    srand((unsigned)time(NULL));
    int i, j;
```

```

for (n = 0; n < size; n++)
{
    i = (rand() * RAND_MAX + rand()) % 10000000;
    j = (rand() * RAND_MAX + rand()) % 10000000;
    swap(num[i], num[j]);
}

for (n = 0; n < size; n++)
    fprintf(fp, "%d ", num[n]);
fclose(fp);
return 0;
}

```

—不过很快，我们就将意识到，用此位图方法，严格说来还是不太行，空间消耗  $10^{7/8}$  还是大于  $1M$  ( $1M=1024*1024$  空间，小于  $10^{7/8}$ )。

—既然如果用位图方案的话，我们需要约  $1.25MB$  (若每条记录是 8 位的正整数的话，则  $10000000/(1024*1024*8) \approx 1.2M$ ) 的空间，而现在只有  $1MB$  的可用存储空间，那么究竟该作何处理呢？

#### **updated && correct:**

@yansha: 上述的位图方案，共需要扫描输入数据两次，具体执行步骤如下：

- 第一次，只处理  $1-4999999$  之间的数据，这些数都是小于  $5000000$  的，对这些数进行位图排序，只需要约  $5000000/8=625000Byte$ ，也就是  $0.625M$ ，排序后输出。
  - 第二次，扫描输入文件时，只处理  $4999999-10000000$  的数据项，也需要  $0.625M$  (可以使用第一次处理申请的内存)。
- 因此，总共也需要  $0.625M$

位图的方法有必要强调一下，就是位图的适用范围为针对不重复的数据进行排序，若数据有重复，位图方案就不适用了。

**3、多路归并。**诚然，在面对本题时，还可以通过计算分析出可以用如 2 的位图法解决，但实际上，很多的时候，我们都面临着这样一个问题，文件太大，无法一次性放入内存中计算处理，那这个时候咋办呢？分而治之，大而化小，也就是把整个大文件分为若干大小的几块，然后分别对每一块进行排序，最后完成整个过程的排序。 $k$  趟算法可以在  $kn$  的时间开销内和  $n/k$  的空间开销内完成对最多  $n$  个小于  $n$  的无重复正整数的排序。

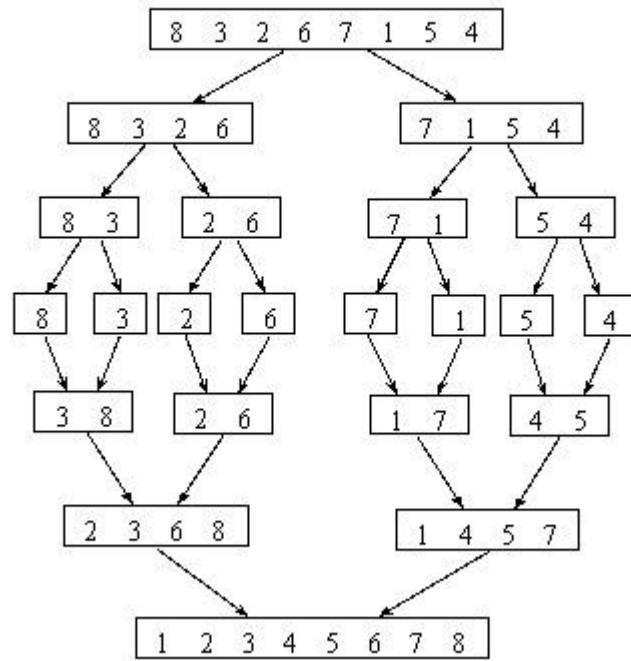
比如可分为 2 块 ( $k=2$ , 1 趟反正占用的内存只有  $1.25/2M$ ) , 1~4999999, 和 5000000~9999999。先遍历一趟, 首先排序处理 1~4999999 之间的整数 (用  $5000000/8=625000$  个字的存储空间来排序 0~4999999 之间的整数) , 然后再第二趟, 对 5000001~1000000 之间的整数进行排序处理。在稍后的第二节、第三节、第四节, 我们将详细阐述并实现这种多路归并排序磁盘文件的方案。

**4、读者思考。** 经过上述思路 3 的方案之后, 现在有两个局部有序的数组了, 那么要得到一个完整的排序的数组, 接下来改怎么做呢?或者说, 如果是  $K$  路归并, 得到  $k$  个排序的子数组, 把他们合并成一个完整的排序数组, 如何优化? 或者, 我再问你一个问题,  $K$  路归并用败者树 和 胜者树 效率有什么差别?这些问题, 请读者思考。

## 第二节、多路归并算法的 c++ 实现

本节咱们暂抛开咱们的问题, 阐述下有关多路归并算法的 c++ 实现问题。在稍后的第三节, 咱们再来具体针对咱们的磁盘文件排序问题阐述与实现。

在了解多路归并算法之前, 你还得了解归并排序的过程, 因为下面的多路归并算法就是基于这个流程的。其实归并排序就是 2 路归并, 而多路归并算法就是把 2 换成了  $k$ , 即多 ( $k$ ) 路归并。下面, 举个例子来说明下此归并排序算法, 如下图所示, 我们对数组 8 3 2 6 7 1 5 4 进行归并排序:



归并排序算法简要介绍：

### 一、思路描述：

设两个有序的子文件(相当于输入堆)放在同一向量中相邻的位置上:  $R[low..m]$ ,  $R[m+1..high]$ , 先将它们合并到一个局部的暂存向量  $R1$ (相当于输出堆)中, 待合并完成后将  $R1$  复制回  $R[low..high]$  中。

二路归并排序的过程是：

- (1) 把无序表中的每一个元素都看作是一个有序表, 则有  $n$  个有序子表;
- (2) 把  $n$  个有序子表按相邻位置分成若干对 (若  $n$  为奇数, 则最后一个子表单独作为一组), 每对中的两个子表进行归并, 归并后子表数减少一半;
- (3) 反复进行这一过程, 直到归并为一个有序表为止。

二路归并排序过程的核心操作是将一维数组中相邻的两个有序表归并为一个有序表。

### 二、分类：

归并排序可分为：多路归并排序、两路归并排序。

若归并的有序表有两个, 叫做二路归并。一般地, 若归并的有序表有  $k$  个, 则称为  $k$  路归并。二路归并最为简单和常用, 既适用于内部排序, 也适用于外部排序。本文着重讨论外部排序下的多( $K$ )路归并算法。

### 三、算法分析：

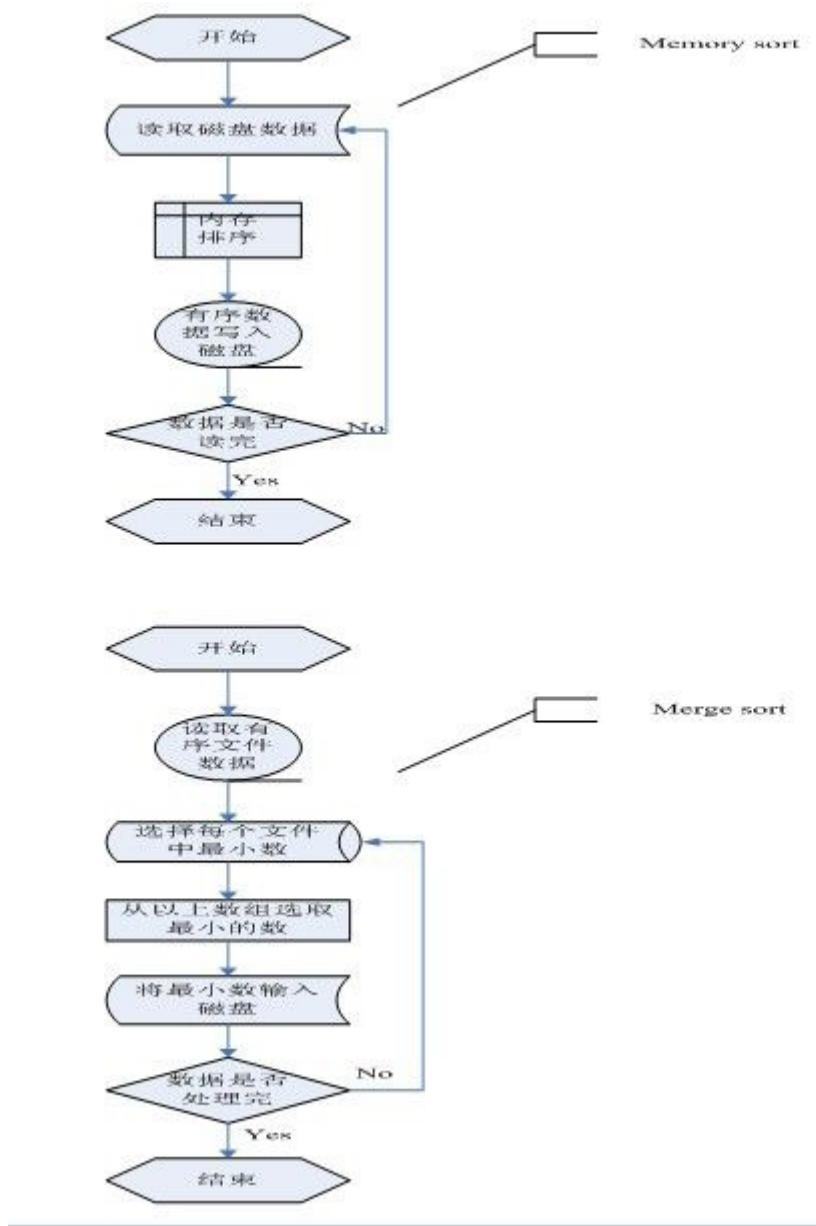
- 1、稳定性：归并排序是一种稳定的排序。
- 2、存储结构要求：可用顺序存储结构。也易于在链表上实现。
- 3、时间复杂度：对长度为  $n$  的文件，需进行  $\lg n$  趟二路归并，每趟归并的时间为  $O(n)$ ，故其时间复杂度无论是在最好情况下还是在最坏情况下均是  $O(n \lg n)$ 。
- 4、空间复杂度：需要一个辅助向量来暂存两有序子文件归并的结果，故其辅助空间复杂度为  $O(n)$ ，显然它不是就地排序。

注意：若用单链表做存储结构，很容易给出就地的归并排序。

总结：与快速排序相比，归并排序的最大特点是，它是一种稳定的排序方法。归并排序一般多用于外排序。但它在内排方面也占有重要地位，因为它是基于比较的时间复杂度为  $O(N * \log(N))$  的排序算法中唯一稳定的排序，所以在需要稳定内排序时通常会选择归并排序。归并排序不要求对序列可以很快地进行随机访问，所以在链表排序的实现中很受欢迎。

好的，介绍完了归并排序后，回到咱们的问题。由第一节，我们已经知道，当数据量大到不适合在内存中排序时，可以利用多路归并算法对磁盘文件进行排序。

我们以一个包含很多个整数的大文件为例，来说明多路归并的外排序算法基本思想。假设文件中整数个数为  $N$  ( $N$  是亿级的)，整数之间用空格分开。首先分多次从该文件中读取  $M$  (十万级) 个整数，每次将  $M$  个整数在内存中使用快速排序之后存入临时文件，然后使用多路归并将各个临时文件中的数据再次整体排好序后存入输出文件。显然，该排序算法需要对每个整数做 2 次磁盘读和 2 次磁盘写。以下是本程序的流程图：



本程序是基于以上思想对包含大量整数文件的从小到大排序的一个简单实现，这里没有使用内存缓冲区，在归并时简单使用一个数组来存储每个临时文件的第一个元素。下面是多路归并排序算法的 c++ 实现代码（在第四节，将给出多路归并算法的 c 实现）：

```

//copyright@ 纯净的天空 && yansha
//5、July, updated, 2010.05.28。
#include <iostream>
#include <ctime>
#include <fstream>
//#include "ExternSort.h"using namespace std;
//使用多路归并进行外排序的类
//ExternSort.h

```

```

/** 大数据量的排序* 多路归并排序* 以千万级整数从小到大排序为例* 一个比较简单的例子，没有建立内存缓冲区*/
#ifndef EXTERN_SORT_H
#define EXTERN_SORT_H

#include <cassert>
class ExternSort
{
public:
    void sort()
    {
        time_t start = time(NULL);
        //将文件内容分块在内存中排序，并分别写入临时文件
        int file_count = memory_sort();
        //归并临时文件内容到输出文件
        merge_sort(file_count);
        time_t end = time(NULL); printf("total time:%f/n", (end - start) * 1000.0/ CLOCKS_PER_SEC);
    }

    //input_file:输入文件名
    //out_file:输出文件名
    //count: 每次在内存中排序的整数个数
    ExternSort(const char *input_file, const char * out_file, int count)
    {
        m_count = count;
        m_in_file = new char[strlen(input_file) + 1];
        strcpy(m_in_file, input_file);
        m_out_file = new char[strlen(out_file) + 1];
        strcpy(m_out_file, out_file);
    }
    virtual ~ExternSort()
    {
        delete [] m_in_file;
        delete [] m_out_file;
    }
private:
    int m_count;
    //数组长度 char *m_in_file;
    //输入文件的路径
    char *m_out_file;
    //输出文件的路径
protected:
    int read_data(FILE* f, int a[], int n)
    {

```

```

int i = 0;
while(i < n && (fscanf(f, "%d", &a[i]) != EOF))
    i++;
printf("read:%d integer/n", i);
return i;
}

void write_data(FILE* f, int a[], int n)
{
    for(int i = 0; i < n; ++i)
        fprintf(f, "%d ", a[i]);
}

char* temp_filename(int index)
{
    char *tempfile = new char[100];
    sprintf(tempfile, "temp%d.txt", index);
    return tempfile;
}

static int cmp_int(const void *a, const void *b)
{
    return *(int*)a - *(int*)b;
}

int memory_sort()
{
    FILE* fin = fopen(m_in_file, "rt");
    int n = 0, file_count = 0; int *array = new int[m_count];

    //每读入 m_count 个整数就在内存中做一次排序，并写入临时文件
    while((n = read_data(fin, array, m_count)) > 0)
    {
        qsort(array, n, sizeof(int), cmp_int);      //这里，调用了库函数阿，在第四节的
c 实现里，不再调 qsort。
        char *fileName = temp_filename(file_count++);
        FILE *tempFile = fopen(fileName, "w");
        free(fileName);
        write_data(tempFile, array, n);
        fclose(tempFile);
    }
    delete [] array;
    fclose(fin);
    return file_count;
}

void merge_sort(int file_count)

```

```

{
    if(file_count <= 0)
        return;
    //归并临时文件 FILE *fout = fopen(m_out_file, "wt");
    FILE* *farray = new FILE*[file_count];
    int i;
    for(i = 0; i < file_count; ++i)
    {
        char* fileName = temp_filename(i);
        farray[i] = fopen(fileName, "rt");
        free(fileName);
    }
    int *data = new int[file_count];
    //存储每个文件当前的一个数字
    bool *hasNext = new bool[file_count];
    //标记文件是否读完
    memset(data, 0, sizeof(int) * file_count);
    memset(hasNext, 1, sizeof(bool) * file_count);
    for(i = 0; i < file_count; ++i)
    {
        if(fscanf(farray[i], "%d", &data[i]) == EOF)
            //读每个文件的第一个数到 data 数组
            hasNext[i] = false;
    }

    while(true)
    {
        //求 data 中可用的最小的数字，并记录对应文件的索引
        int min = data[0];
        int j = 0;
        while (j < file_count && !hasNext[j])
            j++;
        if (j >= file_count)
            //没有可取的数字，终止归并
            break;
        for(i = j + 1; i < file_count; ++i)
        {
            if(hasNext[i] && min > data[i])
            {
                min = data[i];
                j = i;
            }
        }
        if(fscanf(farray[j], "%d", &data[j]) == EOF)

```

```

        //读取文件的下一个元素
        hasNext[j] = false;
        fprintf(fout, "%d ", min);
    }

    delete [] hasNext;
    delete [] data;
    for(i = 0; i < file_count; ++i)
    {
        fclose(farray[i]);
    }
    delete [] farray;
    fclose(fout);
}

};

#endif

//测试主函数文件
/** 大文件排序* 数据不能一次性全部装入内存* 排序文件里有多个整数，整数之间用空格隔开*/

const unsigned int count = 10000000;
// 文件里数据的行数 const unsigned int number_to_sort = 1000000;
//在内存中一次排序的数量
const char *unsort_file = "unsort_data.txt";
//原始未排序的文件名
const char *sort_file = "sort_data.txt";
//已排序的文件名
void init_data(unsigned int num);

//随机生成数据文件

int main(int argc, char* argv)
{
    srand(time(NULL));
    init_data(count);
    ExternSort extSort(unsort_file, sort_file, number_to_sort);
    extSort.sort();
    system("pause");
    return 0;
}

void init_data(unsigned int num)
{
    FILE* f = fopen(unsort_file, "wt");

```

```
    for(int i = 0; i < num; ++i)
        fprintf(f, "%d ", rand());
    fclose(f);
}
```

程序测试：读者可以继续用小文件小数据量进一步测试。

```
//10趟排序
const unsigned int count = 10000000; // 文件里数据的行数
const unsigned int number_to_sort = 1000000; //在内存中一次排序的数量
const char *unsort_file = "unsort_data.txt"; //原始未排序的文件名
const char *sort_file = "sort_data.txt"; //已排序的文件名
void init_data(unsigned int num); //随机生成数据文件
```

```
D:\Program Files\Microsoft Visual Studio\MyProjects\uu\Debug\uu.exe
read:1000000 integer
read:0 integer
total time:20.000000
请按任意键继续... ■
```

### 第三节、磁盘文件排序的编程实现

ok，接下来，我们来编程实现上述磁盘文件排序的问题，本程序由两部分构成：

#### 1、内存排序

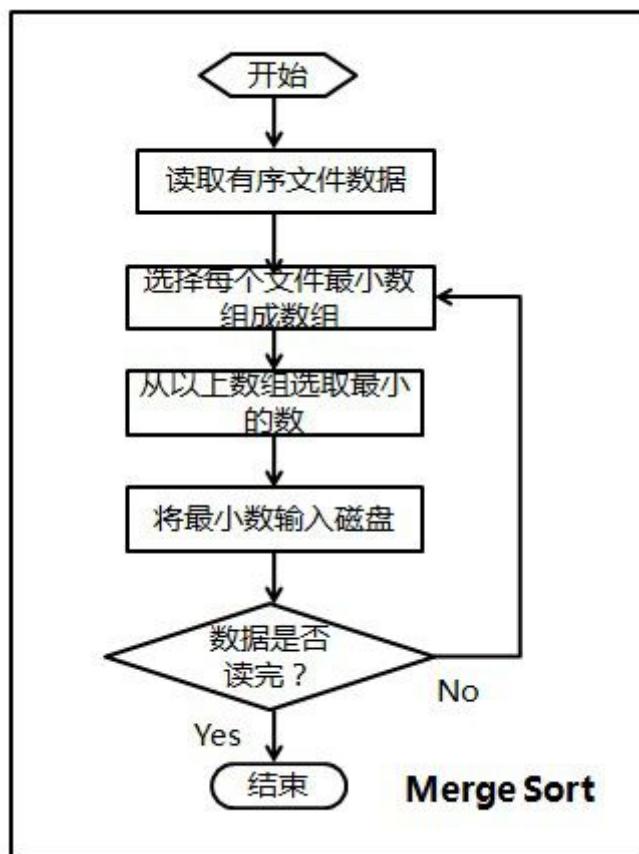
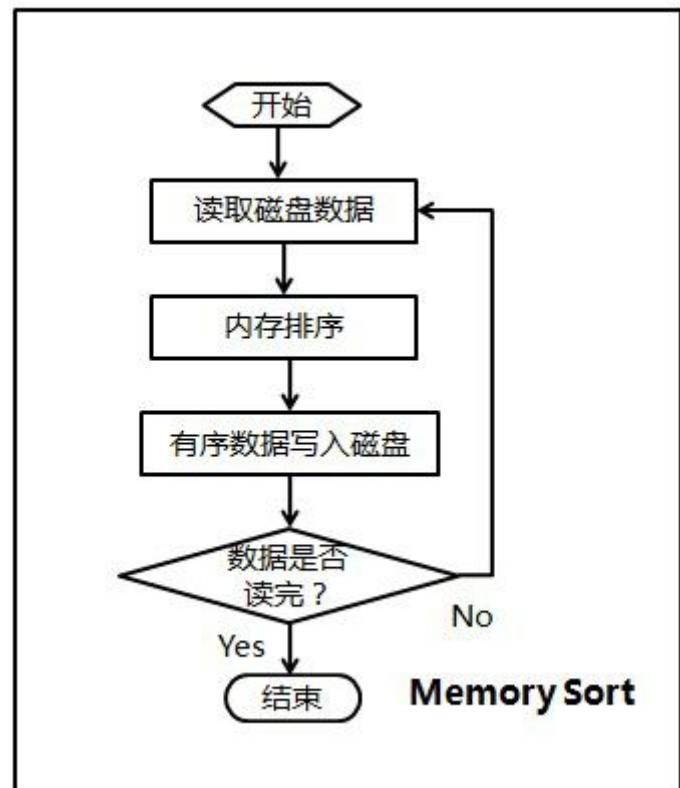
由于要求的可用内存为 1MB，那么每次可以在内存中对 250K 的数据进行排序，然后将有序的数写入硬盘。

那么 10M 的数据需要循环 40 次，最终产生 40 个有序的文件。

## 2、归并排序

1. 将每个文件最开始的数读入(由于有序，所以为该文件最小数)，存放在一个大小为 40 的 `first_data` 数组中；
2. 选择 `first_data` 数组中最小的数 `min_data`，及其对应的文件索引 `index`；
3. 将 `first_data` 数组中最小的数写入文件 `result`，然后更新数组 `first_data`(根据 `index` 读取该文件下一个数代替 `min_data`)；
4. 判断是否所有数据都读取完毕，否则返回 2。

所以，本程序按顺序分两步，第一步、Memory Sort，第二步、Merge Sort。程序的流程图，如下图所示（感谢 F 的绘制）。



然后，编写的完整代码如下：

```

//copyright@ yansha
//July、updated, 2011.05.28.
#include <iostream>
#include <string>
#include <algorithm>
#include <time.h>
using namespace std;

int sort_num = 10000000;
int memory_size = 250000;

//每次只对 250k 个小数据量进行排序
int read_data(FILE *fp, int *space)
{
    int index = 0;
    while (index < memory_size && fscanf(fp, "%d ", &space[index]) != EOF)
        index++;
    return index;
}

void write_data(FILE *fp, int *space, int num)
{
    int index = 0;
    while (index < num)
    {
        fprintf(fp, "%d ", space[index]);
        index++;
    }
}

// check the file pointer whether valid or not.
void check_fp(FILE *fp)
{
    if (fp == NULL)
    {
        cout << "The file pointer is invalid!" << endl;
        exit(1);
    }
}

int compare(const void *first_num, const void *second_num)
{
    return *(int *)first_num - *(int *)second_num;
}

```

```

string new_file_name(int n)
{
    char file_name[20];
    sprintf(file_name, "data%d.txt", n);
    return file_name;
}

int memory_sort()
{
    // open the target file.
    FILE *fp_in_file = fopen("data.txt", "r");
    check_fp(fp_in_file);
    int counter = 0;
    while (true)
    {
        // allocate space to store data read from file.
        int *space = new int[memory_size];
        int num = read_data(fp_in_file, space);
        // the memory sort have finished if not numbers any more.
        if (num == 0)
            break;

        // quick sort.
        qsort(space, num, sizeof(int), compare);
        // create a new auxiliary file name.
        string file_name = new_file_name(++counter);
        FILE *fp_aux_file = fopen(file_name.c_str(), "w");
        check_fp(fp_aux_file);

        // write the orderly numbers into auxiliary file.
        write_data(fp_aux_file, space, num);
        fclose(fp_aux_file);
        delete []space;
    }
    fclose(fp_in_file);

    // return the number of auxiliary files.
    return counter;
}

void merge_sort(int file_num)
{
    if (file_num <= 0)

```

```

    return;
// create a new file to store result.
FILE *fp_out_file = fopen("result.txt", "w");
check_fp(fp_out_file);

// allocate a array to store the file pointer.
FILE **fp_array = new FILE *[file_num];
int i;
for (i = 0; i < file_num; i++)
{
    string file_name = new_file_name(i + 1);
    fp_array[i] = fopen(file_name.c_str(), "r");
    check_fp(fp_array[i]);
}

int *first_data = new int[file_num];
//new 出个大小为 0.1 亿/250k 数组, 由指针 first_data 指示数组首地址
bool *finish = new bool[file_num];
memset(finish, false, sizeof(bool) * file_num);

// read the first number of every auxiliary file.
for (i = 0; i < file_num; i++)
    fscanf(fp_array[i], "%d ", &first_data[i]);
while (true)
{
    int index = 0;
    while (index < file_num && finish[index])
        index++;

    // the finish condition of the merge sort.
    if (index >= file_num)
        break;
    //主要的修改在上面两行代码, 就是 merge sort 结束条件。
    //要保证所有文件都读完, 必须使得 finish[0]...finish[40]都为真
    //July、 yansha, 555, 2011.05.29.

    int min_data = first_data[index];
    // choose the relative minimum in the array of first_data.
    for (i = index + 1; i < file_num; i++)
    {
        if (min_data > first_data[i] && !finish[i])
            //一旦发现比 min_data 更小的数据 first_data[i]
        {
            min_data = first_data[i];

```

```

        //则置 min_data<-first_data[i]index = i;
        //把下标 i 赋给 index。
    }

}

// write the orderly result to file.
fprintf(fp_out_file, "%d ", min_data);
if (fscanf(fp_array[index], "%d ", &first_data[index]) == EOF)
    finish[index] = true;
}

fclose(fp_out_file);
delete []finish;
delete []first_data;
for (i = 0; i < file_num; i++)
    fclose(fp_array[i]);
delete [] fp_array;
}

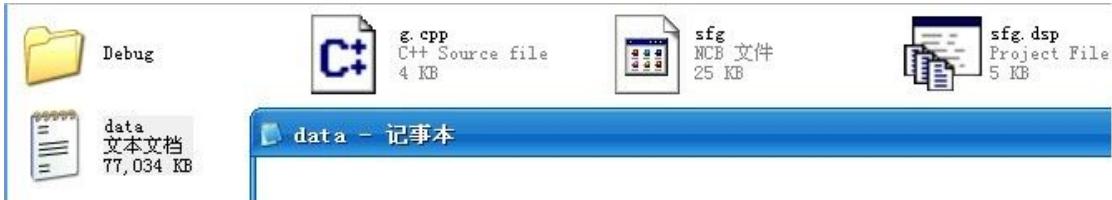
int main()
{
    clock_t start_memory_sort = clock();
    int aux_file_num = memory_sort();
    clock_t end_memory_sort = clock();
    cout << "The time needs in memory sort: " << end_memory_sort - start_memory_sort << endl;
    clock_t start_merge_sort = clock();
    merge_sort(aux_file_num);
    clock_t end_merge_sort = clock();
    cout << "The time needs in merge sort: " << end_merge_sort - start_merge_sort << endl;
    system("pause");
    return 0;
}

```

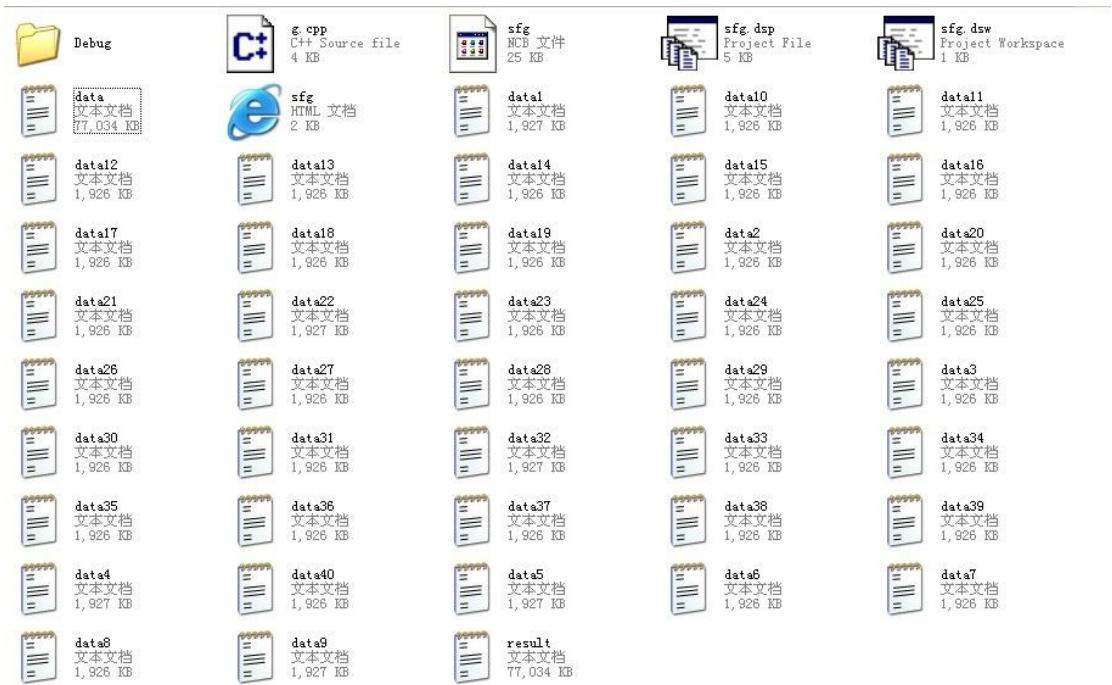
其中，生成数据文件 `data.txt` 的代码在第一节已经给出。

**程序测试：**

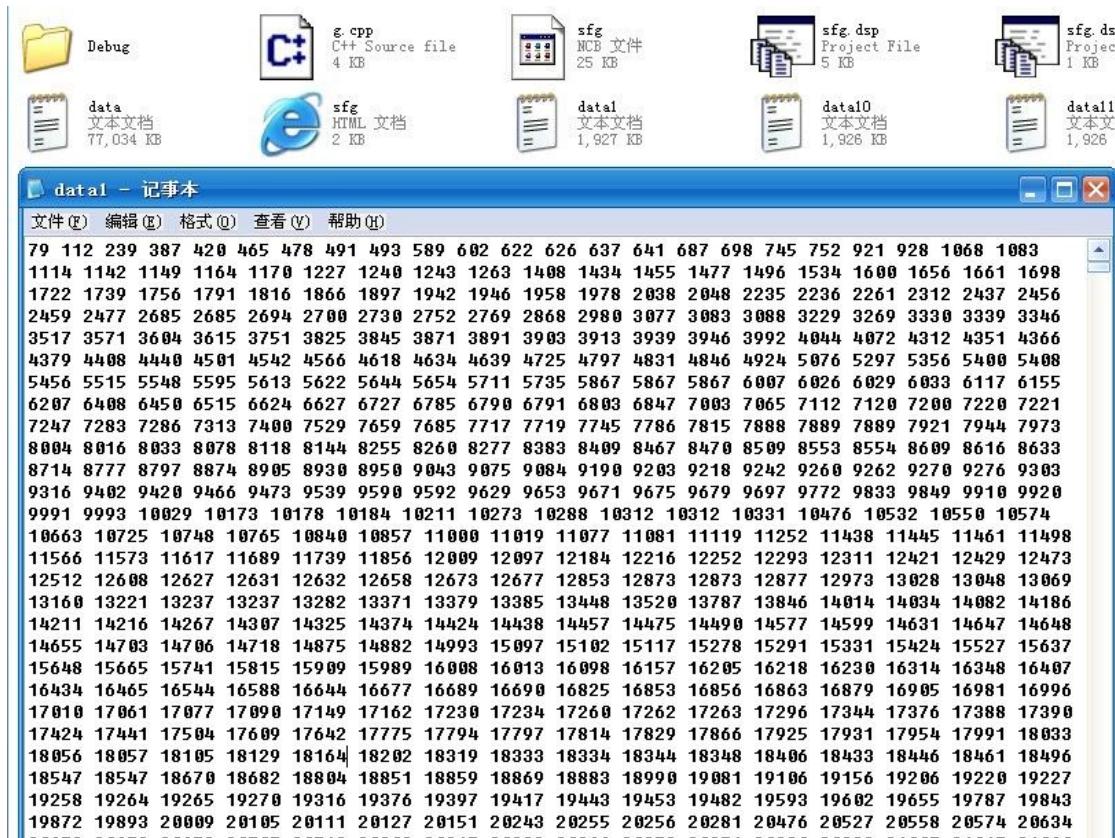
1、咱们对 1000W 数据进行测试，打开半天没看到数据，



2、编译运行上述程序后，**data** 文件先被分成 40 个小文件 **data[1....40]**，然后程序再对这 40 个小文件进行归并排序，排序结果最终生成在 **result** 文件中，自此 **result** 文件中便是由 **data** 文件的数据经排序后得到的数据。



3、且，我们能看到，**data[i]**, **i=1...40** 的每个文件都是有序的，如下图：



4、最终的运行结果，如下，单位统一为 ms:

```
The time needs in memory sort: 14671ms
The time needs in merge sort: 10625ms
请按任意键继续. . . =
```

由上观之，我们发现，第一节的位图方案的程序效率是最快的，约为 14s，而采用上述的多路归并算法的程序运行时间约为 25s。时间主要浪费在读写磁盘 IO 上，且程序中用的库函数 `qsort` 也耗费了不少时间。所以，总的来说，采取位图方案是最佳方案。

### 小数据量测试：

我们下面针对小数据量的文件再测试一次，针对 20 个小数据，每趟对 4 个数据进行排序，即 5 路归并，程序的排序结果如下图所示。

The screenshot displays six windows, each titled "data" or "result" followed by a number and "- 记事本".

- data - 记事本:** Contains the initial array of 20 numbers: 5 11 0 18 4 14 9 7 6 8 12 17 16 13 19 10 2 1 3 15.
- data1 - 记事本:** Contains the first partition: 0 5 11 18.
- data2 - 记事本:** Contains the second partition: 4 7 9 14.
- data3 - 记事本:** Contains the third partition: 6 8 12 17.
- data4 - 记事本:** Contains the fourth partition: 10 13 16 19.
- result - 记事本:** Shows the sorted array: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 |.

运行时间：

0ms，可以忽略不计了，毕竟是对 20 个数的小数据量进行排序：

```
The time needs in memory sort: 0
The time needs in merge sort: 0
请按任意键继续. . .
```

沙海拾贝：

我们不在乎是否能把一个软件产品或一本书最终完成，我们更在乎的是，在完成这个产品或创作这本书的过程中，读者学到了什么，能学到什么？所以，不要一味的马上就想得到一道题目的正确答案，请跟着我们一起逐步走向山巅。

## 第四节、多路归并算法的 c 实现

本多路归并算法的c实现原理与上述c++实现一致，不同的地方体现在一些细节处理上，且对临时文件的排序，不再用系统提供的快排，即上面的 `qsort` 库函数，是采用的三数中值的快速排序（个数小于 3 用插入排序）的。而我们知道，纯正的归并排序其实就是比较排序，在归并过程中总是不断的比较，为了从两个数中挑小的归并到最终的序列中。`ok`，此程序的详情请看：

```
//copyright@ 555
//July、2011.05.29。
#include <assert.h>
#include <time.h>
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>

void swap_int(int* a,int* b)
{
    int c;
```

```

    c = *a;
    *a = *b;
    *b = c;
}

//插入排序
void InsertionSort(int A[],int N)
{
    int j,p;
    int tmp;
    for(p = 1; p < N; p++)
    {
        tmp = A[p];
        for(j = p;j > 0 && A[j - 1] >tmp;j--)
        {
            A[j] = A[j - 1];
        }

        A[j] = tmp;
    }
}

//三数取中分割法
int Median3(int A[],int Left,int Right)
{
    int Center = (Left + Right) / 2;
    if (A[Left] > A[Center])
        swap_int(&A[Left],&A[Center]);
    if (A[Left] > A[Right])
        swap_int(&A[Left],&A[Right]);
    if (A[Center] > A[Right])
        swap_int(&A[Center],&A[Right]);
    swap_int(&A[Center],&A[Right - 1]);
    return A[Right - 1];
}

//快速排序
void QuickSort(int A[],int Left,int Right)
{
    int i,j;
    int Pivot;
    const int Cutoff = 3;
    if (Left + Cutoff <= Right)
    {

```

```

Pivot = Median3(A,Left,Right);

i = Left;
j = Right - 1;
while (1)
{
    while(A[++i] < Pivot){;}
    while(A[--j] > Pivot){;}
    if (i < j)
        swap_int(&A[i],&A[j]);
    else
        break;
}
swap_int(&A[i],&A[Right - 1]);

QuickSort(A,Left,i - 1);
QuickSort(A,i + 1,Right);
}

else
{
    InsertionSort(A+Left,Right - Left + 1);
}
}

//const int KNUM = 40;
//分块数
const int NUMBER = 10000000;
//输入文件最大读取的整数的个数
//为了便于测试，我决定改成小文件小数据量进行测试。
const int KNUM = 4;
//分块数 const int NUMBER = 100;
//输入文件最大读取的整数的个数
const char *in_file = "infile.txt";
const char *out_file = "outfile.txt";
#define OUTPUT_OUT_FILE_DATA
//数据量大的时候，没必要把所有的数全部打印出来，所以可以把上面这句注释掉。
void gen_infile(int n)
{
    int i;
    FILE *f = fopen(in_file, "wt");
    for(i = 0;i < n; i++)
        fprintf(f,"%d ",rand());
    fclose(f);
}

```

```

int  read_data(FILE *f,int a[],int n)
{
    int i = 0;
    while ((i < n) && (fscanf(f,"%d",&a[i]) != EOF))
        i++;
    printf("read: %d integer/n",i);
    return i;
}

void  write_data(FILE *f,int a[],int n)
{
    int i;for(i = 0; i< n;i++)
        fprintf(f,"%d ",a[i]);
}

char* temp_filename(int index)
{
    char *tempfile = (char*) malloc(64*sizeof(char));
    assert(tempfile);
    sprintf(tempfile, "temp%d.txt", index);
    return tempfile;
}

//K路串行读取
void k_num_read(void)
{
    char* filename;
    int i,cnt,*array;
    FILE* fin;
    FILE* tmpfile;
    //计算 knum,每路应读取的整数个数 int n = NUMBER/KNUM;
    if (n * KNUM < NUMBER)n++;

    //建立存储分块读取的数据的数组
    array = (int*)malloc(n * sizeof(int));assert(array);
    //打开输入文件
    fin = fopen(in_file,"rt");
    i = 0;

    //分块循环读取数据,并写入硬盘上的临时文件
    while ( (cnt = read_data(fin,array,n))>0)
    {
        //对每次读取的数据,先进行快速排序,然后写入硬盘上的临时文件
        QuickSort(array,0,cnt - 1);
    }
}

```

```

        filename = temp_filename(i++);
        tmpfile = fopen(filename, "w");
        free(filename);
        write_data(tmpfile, array, cnt);
        fclose(tmpfile);
    }
    assert(i == KNUM);
    //没有生成 K 路文件时进行诊断
    //关闭输入文件句柄和临时存储数组
    fclose(fin);
    free(array);
}

//k 路合并(败者树)
void k_num_merge(void)
{
    FILE *fout;
    FILE **farray;
    char *filename;
    int *data;
    char *hasNext;
    int i, j, m, min;
#ifdef OUTPUT_OUT_FILE_DATA
int id;
#endif
    //打开输出文件
    fout = fopen(out_file, "wt");
    //打开各路临时分块文件
    farray = (FILE**)malloc(KNUM*sizeof(FILE*));
    assert(farray);
    for(i = 0; i < KNUM; i++)
    {
        filename = temp_filename(i);
        farray[i] = fopen(filename, "rt");
        free(filename);
    }

    //建立 KNUM 个元素的 data,hasNext 数组,存储 K 路文件的临时数组和读取结束状态
    data = (int*)malloc(KNUM*sizeof(int));
    assert(data);
    hasNext = (char*)malloc(sizeof(char)*KNUM);
    assert(hasNext);
    memset(data, 0, sizeof(int) * KNUM);
    memset(hasNext, 1, sizeof(char) * KNUM);
}

```

```

//读K路文件先读取第一组数据，并对读取结束的各路文件设置不可再读状态
for(i = 0; i < KNUM; i++)
{
    if(fscanf(farray[i], "%d", &data[i]) == EOF)
    {
        hasNext[i] = 0;
    }
}

//读取各路文件，利用败者树从小到大输出到输出文件
#ifndef OUTPUT_OUT_FILE_DATAid = 0;
#endif

j = 0; F_LOOP:
if (j < KNUM)
    //以下这段代码嵌套过深，日后应尽量避免此类问题。
{
    while(1==1)
    {
        min = data[j];
        m = j;
        for(i = j+1; i < KNUM; i++)
        {
            if(hasNext[i] == 1 && min > data[i])
            {
                min = data[i];m = i;
            }
        }

        if(fscanf(farray[m], "%d", &data[m]) == EOF)
        {
            hasNext[m] = 0;
        }
        fprintf(fout, "%d ", min);
#endif OUTPUT_OUT_FILE_DATAprintf("fout :%d %d/n", ++id,min);
#endif
        if (m == j && hasNext[m] == 0)
        {
            for (i = j+1; i < KNUM; i++)
            {
                if (hasNext[m] != hasNext[i])
                {
                    m = i;
                    //第i个文件未读完，从第i个继续往下读
                }
            }
        }
    }
}

```

```

                break;
            }
        }
        if (m != j)
        {
            j = m;
            goto F_LOOP;
        }
        break;
    }
}

//关闭分配的数据和数组
free(hasNext);
free(data);
for(i = 0; i < KNUM; ++i)
{
    fclose(farray[i]);
}
free(farray);
fclose(fout);
}

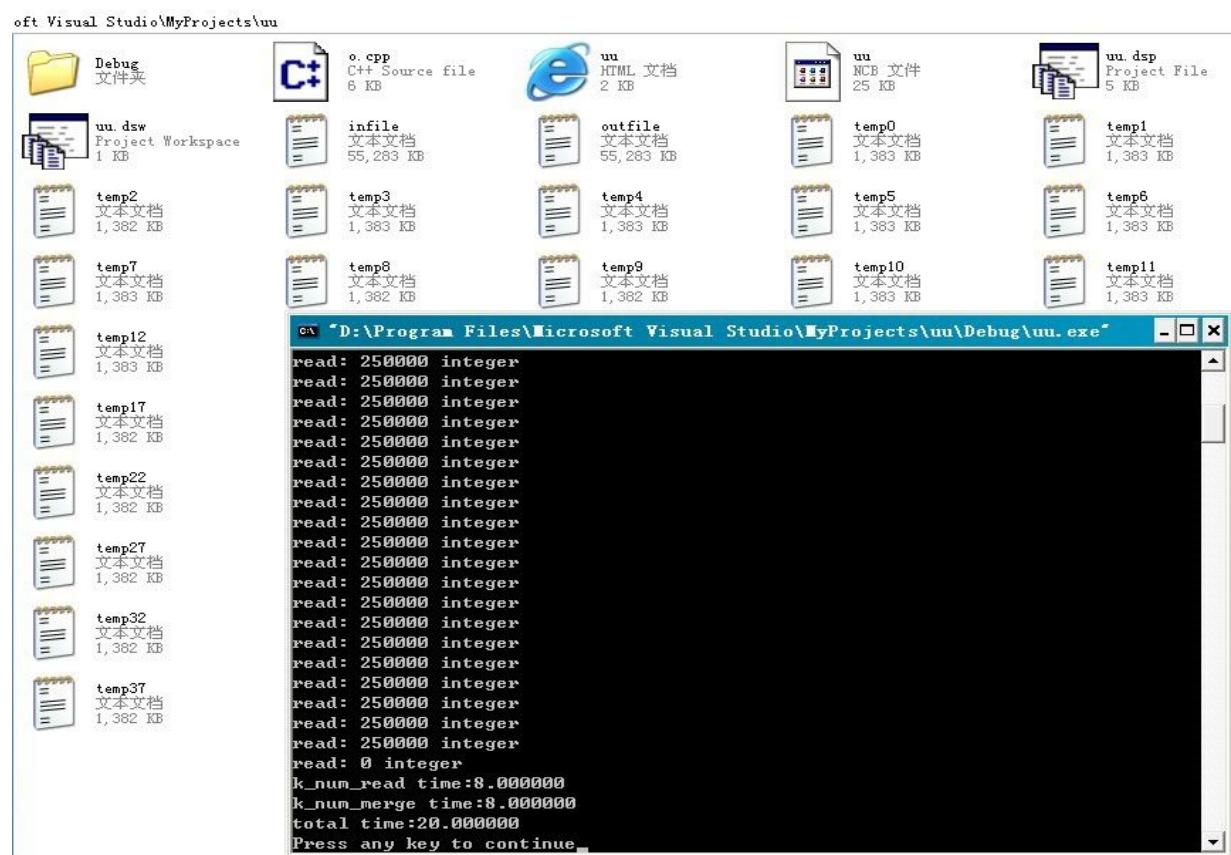
int main()
{
    time_t start = time(NULL),end,start_read,end_read,start_merge,end_merge;
    gen_infile(NUMBER);
    end = time(NULL);
    printf("gen_infile data time:%f/n", (end - start) * 1000.0/ CLOCKS_PER_SEC);
    start_read = time(NULL);k_num_read();
    end_read = time(NULL);
    printf("k_num_read time:%f/n", (end_read - start_read) * 1000.0/ CLOCKS_PER_SEC)
    ;
    start_merge = time(NULL);
    k_num_merge();
    end_merge = time(NULL);
    printf("k_num_merge time:%f/n", (end_merge - start_merge) * 1000.0/ CLOCKS_PER_SEC);
    end = time(NULL);
    printf("total time:%f/n", (end - start) * 1000.0/ CLOCKS_PER_SEC);
    return 0;
}

```

程序测试：

在此，我们先测试下对 10000000 个数据的文件进行 40 趟排序，然后再对 100 个数据的文件进行 4 趟排序（读者可进一步测试）。如弄几组小点的数据，输出 ID 和数据到屏幕，再看程序运行效果。

1. 10 个数, 4 组
  2. 40 个数, 5 组
  3. 55 个数, 6 组
  4. 100 个数, 7 组



```

infile - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
41 18467 6334 26500 19169 15724 11478 29358 26962 24464 5705 28145 23281 16827 9961 491 2995
11942 4827 5436 32391 14604 3902 153 292 12382 17421 18716 19718 19895 5447 21726 14771 11538
1869 19912 25667 26299 17035 9894 28703 23811 31322 30333 17673 4664 15141 7711 28253 6868 25547
27644 32662 32757 20037 12859 8723 9741 27529 778 12316 3035 22190 1842 288 30106 9040 8942
19264 22648 27446 23805 15890 6729 24370 15350 15006 31101 24393 3548 19629 12623 24084 19954
18756 11840 4966 7376 13931 26308 16944 32439 24626 11323 5537 21538 16118 2082 22929 16541

outfile - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
41 153 288 292 491 778 1842 1869 2082 2995 3035 3548 3902 4664 4827 4966 5436 5447 5537 5705
6334 6729 6868 7376 7711 8723 8942 9040 9741 9894 9961 11323 11478 11538 11840 11942 12316 12382
12623 12859 13931 14604 14771 15006 15141 15350 15724 15890 16118 16541 16827 16944 17035 17421
17673 18467 18716 18756 19169 19264 19629 19718 19895 19912 19954 20037 21538 21726 22190 22648
22929 23281 23805 23811 24084 24370 24393 24464 24626 25547 25667 26299 26308 26500 26962 27446
27529 27644 28145 28253 28703 29358 30106 30333 31101 31322 32391 32439 32662 32757

```

(备注：1、以上所有各节的程序运行环境为 windows xp + vc6.0 + e5200 cpu 2.5g 主频，2、感谢 5 为本文程序所作的大量测试工作)

## 全文总结：

1、关于本章中位图和多路归并两种方案的时间复杂度及空间复杂度的比较，如下：

	<u>时间复杂度</u>	<u>空间复杂度</u>
位图	$O(N)$	0.625M
多位归并	$O(N \log n)$	1M

(多路归并，时间复杂度为  $O(k^*n/k^*\log n/k)$ ，严格来说，还要加上读写磁盘的时间，而此算法绝大部分时间也是浪费在这上面)

## 2、bit-map

适用范围：可进行数据的快速查找，判重，删除，一般来说数据范围是 int 的 10 倍以下

基本原理及要点：使用 bit 数组来表示某些元素是否存在，比如 8 位电话号码

扩展：bloom filter 可以看做是对 bit-map 的扩展

问题实例：

1)已知某个文件内包含一些电话号码，每个号码为 8 位数字，统计不同号码的个数。

8 位最多 99 999 999，大概需要 99m 个 bit，大概 10 几 m 字节的内存即可。

2)2.5 亿个整数中找出不重复的整数的个数，内存空间不足以容纳这 2.5 亿个整数。

将 bit-map 扩展一下，用 2bit 表示一个数即可，0 表示未出现，1 表示出现一次，2 表示出现 2 次及以上。或者我们不用 2bit 来进行表示，我们用两个 bit-map 即可模拟实现这个 2bit-map。

3、[外排序适用范围]大数据的排序，去重基本原理及要点：外排序的归并方法，置换选择败者树原理，最优归并树扩展。问题实例：1).有一个 1G 大小的一个文件，里面每一行是一个词，词的大小不超过 16 个字节，内存限制大小是 1M。返回频数最高的 100 个词。这个数据具有很明显的特点，词的大小为 16 个字节，但是内存只有 1m 做 hash 有些不够，所以可以用来排序。内存可以当输入缓冲区使用。

#### 4、海量数据处理

有关海量数据处理的方法或面试题可参考此文，[十道海量数据处理面试题与十个方法大总结](#)。日后，会逐步实现这十个处理海量数据的方法。同时，送给各位一句话，解决问题的关键在于熟悉一个算法，而不是某一个问题。熟悉了一个算法，便通了一片题目。

本章完。

**updated:** 有一读者朋友针对本文写了一篇文章为，海量数据多路归并排序的 c++ 实现

(归并时利用了败者树)，地址为：

<http://www.cnblogs.com/harryshayne/archive/2011/07/02/2096196.html>。谢谢，欢迎参考。

# 第十一章：最长公共子序列(LCS)问题

## 前言

程序员编程艺术系列重新开始创作了（前十章，请参考[程序员编程艺术第一~十章集锦与总结](#)）。回顾之前的前十章，有些代码是值得商榷的，因当时的代码只顾阐述算法的原理或思想，所以，很多的与代码规范相关的问题都未能做到完美。日后，会着力修缮之。

搜遍网上，讲解这个 LCS 问题的文章不计其数，但大多给读者一种并不友好的感觉，稍感晦涩，且代码也不够清晰。本文力图避免此些情况。力保通俗，阐述详尽。同时，经典算法研究系列的第三章（[三、dynamic programming](#)）也论述了此 LCS 问题。有任何问题，欢迎不吝赐教。

## 第一节、问题描述

什么是最长公共子序列呢？好比一个数列  $S$ ，如果分别是两个或多个已知数列的子序列，且是所有符合此条件序列中最长的，则  $S$  称为已知序列的最长公共子序列。

举个例子，如：有两条随机序列，如 1 3 4 5 5，and 2 4 5 5 7 6，则它们的最长公共子序列便是：4 5 5。

注意最长公共子串 (Longest Common Substring) 和最长公共子序列 (Longest Common Subsequence, LCS) 的区别：子串 (Substring) 是串的一个连续的部分，子序列 (Subsequence) 则是从不改变序列的顺序，而从序列中去掉任意的元素而获得的新序列；更简略地说，前者（子串）的字符的位置必须连续，后者（子序列 LCS）则不必。比如字符串 acdfg 同 akdfc 的最长公共子串为 df，而他们的最长公共子序列是 adf。LCS 可以使用动态规划法解决。下文具体描述。

## 第二节、LCS 问题的解决思路

### 穷举法

解最长公共子序列问题时最容易想到的算法是穷举搜索法，即对  $X$  的每一个子序列，检查它是否也是  $Y$  的子序列，从而确定它是否为  $X$  和  $Y$  的公共子序列，并且在检查过程中选出最长的公共子序列。 $X$  和  $Y$  的所有子序列都检查过后即可求出  $X$  和  $Y$  的最长公共子序列。

$X$  的一个子序列相应于下标序列 $\{1, 2, \dots, m\}$ 的一个子序列，因此， $X$  共有  $2^m$  个不同子序列  
( $Y$  亦如此，如为  $2^n$ )，从而穷举搜索法需要指数时间 ( $2^m * 2^n$ )。

## 动态规划算法

事实上，最长公共子序列问题也有最优子结构性质。

记：

$X_i = < x_1, \dots, x_i >$  即  $X$  序列的前  $i$  个字符 ( $1 \leq i \leq m$ ) (前缀)

$Y_j = < y_1, \dots, y_j >$  即  $Y$  序列的前  $j$  个字符 ( $1 \leq j \leq n$ ) (前缀)

假定  $Z = < z_1, \dots, z_k > \in LCS(X, Y)$ 。

- 若  $x_m = y_n$  (最后一个字符相同)，则不难用反证法证明：该字符必是  $X$  与  $Y$  的任一最长公共子序列  $Z$  (设长度为  $k$ ) 的最后一个字符，即有  $z_k = x_m = y_n$  且显然有  $Z_{k-1} \in LCS(X_{m-1}, Y_{n-1})$  即  $Z$  的前缀  $Z_{k-1}$  是  $X_{m-1}$  与  $Y_{n-1}$  的最长公共子序列。此时，问题化归成求  $X_{m-1}$  与  $Y_{n-1}$  的  $LCS$  ( $LCS(X, Y)$  的长度等于  $LCS(X_{m-1}, Y_{n-1})$  的长度加 1)。
- 若  $x_m \neq y_n$ ，则亦不难用反证法证明：要么  $Z \in LCS(X_{m-1}, Y)$ ，要么  $Z \in LCS(X, Y_{n-1})$ 。由于  $z_k \neq x_m$  与  $z_k \neq y_n$  其中至少有一个必成立，若  $z_k \neq x_m$  则有  $Z \in LCS(X_{m-1}, Y)$ ，类似的，若  $z_k \neq y_n$  则有  $Z \in LCS(X, Y_{n-1})$ 。此时，问题化归成求  $X_{m-1}$  与  $Y$  的  $LCS$  及  $X$  与  $Y_{n-1}$  的  $LCS$ 。 $LCS(X, Y)$  的长度为： $\max\{LCS(X_{m-1}, Y)$  的长度， $LCS(X, Y_{n-1})$  的长度 $\}$ 。

由于上述当  $x_m \neq y_n$  的情况中，求  $LCS(X_{m-1}, Y)$  的长度与  $LCS(X, Y_{n-1})$  的长度，这两个问题不是相互独立的：两者都要求  $LCS(X_{m-1}, Y_{n-1})$  的长度。另外两个序列的  $LCS$  中包含了两个序列的前缀的  $LCS$ ，故问题具有最优子结构性质考虑用动态规划法。

也就是说，解决这个  $LCS$  问题，你要求三个方面的东西：1、 $LCS(X_{m-1}, Y_{n-1}) + 1$ ；  
2、 $LCS(X_{m-1}, Y)$ ， $LCS(X, Y_{n-1})$ ；3、 $\max\{LCS(X_{m-1}, Y), LCS(X, Y_{n-1})\}$ 。

行文至此，其实对这个  $LCS$  的动态规划解法已叙述殆尽，不过，为了成书的某种必要性，下面，我试着再多加详细阐述这个问题。

## 第三节、动态规划算法解 LCS 问题

### 3.1、最长公共子序列的结构

最长公共子序列的结构有如下表示：

设序列  $X = \langle x_1, x_2, \dots, x_m \rangle$  和  $Y = \langle y_1, y_2, \dots, y_n \rangle$  的一个最长公共子序列  $Z = \langle z_1, z_2, \dots, z_k \rangle$ ，则：

1. 若  $x_m = y_n$ ，则  $z_k = x_m = y_n$  且  $Z_{k-1}$  是  $X_{m-1}$  和  $Y_{n-1}$  的最长公共子序列；
2. 若  $x_m \neq y_n$  且  $z_k \neq x_m$ ，则  $Z$  是  $X_{m-1}$  和  $Y$  的最长公共子序列；
3. 若  $x_m \neq y_n$  且  $z_k \neq y_n$ ，则  $Z$  是  $X$  和  $Y_{n-1}$  的最长公共子序列。

其中  $X_{m-1} = \langle x_1, x_2, \dots, x_{m-1} \rangle$ ,  $Y_{n-1} = \langle y_1, y_2, \dots, y_{n-1} \rangle$ ,  $Z_{k-1} = \langle z_1, z_2, \dots, z_{k-1} \rangle$ 。

### 3.2、子问题的递归结构

由最长公共子序列问题的最优子结构性质可知，要找出  $X = \langle x_1, x_2, \dots, x_m \rangle$  和  $Y = \langle y_1, y_2, \dots, y_n \rangle$  的最长公共子序列，可按以下方式递归地进行：当  $x_m = y_n$  时，找出  $X_{m-1}$  和  $Y_{n-1}$  的最长公共子序列，然后在其尾部加上  $x_m (=y_n)$  即可得  $X$  和  $Y$  的一个最长公共子序列。当  $x_m \neq y_n$  时，必须解两个子问题，即找出  $X_{m-1}$  和  $Y$  的一个最长公共子序列及  $X$  和  $Y_{n-1}$  的一个最长公共子序列。这两个公共子序列中较长者即为  $X$  和  $Y$  的一个最长公共子序列。

由此递归结构容易看到最长公共子序列问题具有子问题重叠性质。例如，在计算  $X$  和  $Y$  的最长公共子序列时，可能要计算出  $X$  和  $Y_{n-1}$  及  $X_{m-1}$  和  $Y$  的最长公共子序列。而这两个子问题都包含一个公共子问题，即计算  $X_{m-1}$  和  $Y_{n-1}$  的最长公共子序列。

与矩阵连乘积最优计算次序问题类似，我们来建立子问题的最优值的递归关系。用  $c[i,j]$  记录序列  $X_i$  和  $Y_j$  的最长公共子序列的长度。其中  $X_i = \langle x_1, x_2, \dots, x_i \rangle$ ,  $Y_j = \langle y_1, y_2, \dots, y_j \rangle$ 。当  $i=0$  或  $j=0$  时，空序列是  $X_i$  和  $Y_j$  的最长公共子序列，故  $c[i,j]=0$ 。其他情况下，由定理可建立递归关系如下：

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

### 3.3、计算最优值

直接利用上节节末的递归式，我们将很容易就能写出一个计算  $c[i,j]$  的递归算法，但其计算时间是随输入长度指数增长的。由于在所考虑的子问题空间中，总共只有  $\Theta(m \cdot n)$  个不同的子问题，因此，用动态规划算法自底向上地计算最优值能提高算法的效率。

计算最长公共子序列长度的动态规划算法  $LCS\_LENGTH(X, Y)$  以序列  $X = \langle x_1, x_2, \dots, x_m \rangle$  和  $Y = \langle y_1, y_2, \dots, y_n \rangle$  作为输入。输出两个数组  $c[0..m, 0..n]$  和  $b[1..m, 1..n]$ 。其中  $c[i,j]$  存储  $X_i$  与  $Y_j$  的最长公共子序列的长度， $b[i,j]$  记录指示  $c[i,j]$  的值是由哪一个子问题的解达到的，这在构造最长公共子序列时要用到。最后， $X$  和  $Y$  的最长公共子序列的长度记录于  $c[m,n]$  中。

```
Procedure LCS_LENGTH(X, Y);
begin
    m:=length[X];
    n:=length[Y];
    for i:=1 to m do c[i,0]:=0;
    for j:=1 to n do c[0,j]:=0;
    for i:=1 to m do
        for j:=1 to n do
            if x[i]=y[j] then
                begin
                    c[i,j]:=c[i-1,j-1]+1;
                    b[i,j]:="↑";
                end
            else
                begin
                    c[i,j]:=c[i,j-1];
                    b[i,j]:="<span style='color:green; font-size:2em; vertical-align:middle;">←";
                end;
            return(c,b);
    end;
```

由算法  $LCS\_LENGTH$  计算得到的数组  $b$  可用于快速构造序列  $X = \langle x_1, x_2, \dots, x_m \rangle$  和  $Y = \langle y_1, y_2, \dots, y_n \rangle$  的最长公共子序列。首先从  $b[m,n]$  开始，沿着其中的箭头所指的方向在数组  $b$  中搜索。

- 当  $b[i,j]$  中遇到 " $\nwarrow$ " 时 (意味着  $x_i=y_j$  是 LCS 的一个元素), 表示  $X_i$  与  $Y_j$  的最长公共子序列是由  $X_{i-1}$  与  $Y_{j-1}$  的最长公共子序列在尾部加上  $x_i$  得到的子序列;
- 当  $b[i,j]$  中遇到 " $\uparrow$ " 时, 表示  $X_i$  与  $Y_j$  的最长公共子序列和  $X_{i-1}$  与  $Y_j$  的最长公共子序列相同;
- 当  $b[i,j]$  中遇到 " $\leftarrow$ " 时, 表示  $X_i$  与  $Y_j$  的最长公共子序列和  $X_i$  与  $Y_{j-1}$  的最长公共子序列相同。

这种方法是按照反序来找 LCS 的每一个元素的。由于每个数组单元的计算耗费  $O(1)$  时间, 算法  $LCS\_LENGTH$  耗时  $O(mn)$ 。

### 3.4、构造最长公共子序列

下面的算法  $LCS(b, X, i, j)$  实现根据  $b$  的内容打印出  $X_i$  与  $Y_j$  的最长公共子序列。通过算法的调用  $LCS(b, X, \text{length}[X], \text{length}[Y])$ , 便可打印出序列  $X$  和  $Y$  的最长公共子序列。

```
Procedure LCS(b, X, i, j);
begin
  if i=0 or j=0 then return;
  if b[i, j] = " $\nwarrow$ " then
    begin
      LCS(b, X, i-1, j-1);
      print(x[i]); { 打印 x[i] }
    end
  else if b[i, j] = " $\uparrow$ " then LCS(b, X, i-1, j)
    else LCS(b, X, i, j-1);
end;
```

在算法  $LCS$  中, 每一次的递归调用使  $i$  或  $j$  减 1, 因此算法的计算时间为  $O(m+n)$ 。

例如, 设所给的两个序列为  $X=<A, B, C, B, D, A, B>$  和  $Y=<B, D, C, A, B, A>$ 。由算法  $LCS\_LENGTH$  和  $LCS$  计算出的结果如下图所示:

	$j$	0	1	2	3	4	5	6
$i$	$y_j$	B	D	C	A	B	A	
0	$x_i$	0	0	0	0	0	0	0
1	A	0	0	0	0	1	-1	1
2	B	0	1	-1	-1	1	2	-2
3	C	0	1	1	2	-2	2	2
4	B	0	1	1	2	2	3	-3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

我来说明下此图（参考算法导论）。在序列  $X=\{A, B, C, B, D, A, B\}$  和  $Y=\{B, D, C, A, B, A\}$  上，由  $LCS\_LENGTH$  计算出的表  $c$  和  $b$ 。第  $i$  行和第  $j$  列中的方块包含了  $c[i, j]$  的值以及指向  $b[i, j]$  的箭头。在  $c[7,6]$  的项 4，表的右下角为  $X$  和  $Y$  的一个  $LCS<B, C, B, A>$  的长度。对于  $i, j > 0$ ，项  $c[i, j]$  仅依赖于是否有  $x_i = y_j$ ，及项  $c[i-1, j]$  和  $c[i, j-1]$  的值，这几个项都在  $c[i, j]$  之前计算。为了重构一个  $LCS$  的元素，从右下角开始跟踪  $b[i, j]$  的箭头即可，这条路径标示为阴影，这条路径上的每一个“↖”对应于一个使  $x_i = y_j$  为一个  $LCS$  的成员的项（高亮标示）。

所以根据上述图所示的结果，程序将最终输出：“B C B A”。

### 3.5、算法的改进

对于一个具体问题，按照一般的算法设计策略设计出的算法，往往在算法的时间和空间需求上还可以改进。这种改进，通常是利用具体问题的一些特殊性。

例如，在算法  $LCS\_LENGTH$  和  $LCS$  中，可进一步将数组  $b$  省去。事实上，数组元素  $c[i, j]$  的值仅由  $c[i-1, j-1]$ ,  $c[i-1, j]$  和  $c[i, j-1]$  三个值之一确定，而数组元素  $b[i, j]$  也只是用来指示  $c[i, j]$  究竟由哪个值确定。因此，在算法  $LCS$  中，我们可以不借助于数组  $b$  而借助于数组  $c$  本身临时判断  $c[i, j]$  的值是由  $c[i-1, j-1]$ ,  $c[i-1, j]$  和  $c[i, j-1]$  中哪一个数值元素所确定，代价是  $O(1)$  时间。既然  $b$  对于算法  $LCS$  不是必要的，那么算法  $LCS\_LENGTH$  便不必保存它。这一来，可节省  $\Theta(mn)$  的空间，而  $LCS\_LENGTH$  和  $LCS$  所需要的时间分别仍然是  $O(mn)$  和  $O(m+n)$ 。

不过，由于数组  $c$  仍需要  $O(mn)$  的空间，因此这里所作的改进，只是在空间复杂性的常数因子上的改进。

另外，如果只需要计算最长公共子序列的长度，则算法的空间需求还可大大减少。事实上，在计算  $c[i,j]$  时，只用到数组  $c$  的第  $i$  行和第  $i-1$  行。因此，只要用 2 行的数组空间就可以计算出最长公共子序列的长度。更进一步的分析还可将空间需求减至  $\min(m, n)$ 。

## 第四节、编码实现 LCS 问题

动态规划的一个计算最长公共子序列的方法如下，以两个序列  $X$ 、 $Y$  为例子：

设有二维数组  $f[i][j]$  表示  $X$  的  $i$  位和  $Y$  的  $j$  位之前的最长公共子序列的长度，则有：

$$\begin{aligned} f[1][1] &= \text{same}(1,1) \\ f[i][j] &= \max\{f[i-1][j-1] + \text{same}(i,j), f[i-1][j], f[i][j-1]\} \end{aligned}$$

其中， $\text{same}(a,b)$  当  $X$  的第  $a$  位与  $Y$  的第  $b$  位完全相同时为“1”，否则为“0”。

此时， $f[i][j]$  中最大的数便是  $X$  和  $Y$  的最长公共子序列的长度，依据该数组回溯，便可找出最长公共子序列。

该算法的空间、时间复杂度均为  $O(n^2)$ ，经过优化后，空间复杂度可为  $O(n)$ ，时间复杂度为  $O(n \log n)$ 。

以下是此算法的 java 代码：

```
import java.util.Random;

public class LCS{
    public static void main(String[] args){

        //设置字符串长度
        int substringLength1 = 20;
        int substringLength2 = 20; //具体大小可自行设置

        // 随机生成字符串
        String x = GetRandomStrings(substringLength1);
        String y = GetRandomStrings(substringLength2);

        Long startTime = System.nanoTime();
        // 构造二维数组记录子问题 x[i] 和 y[i] 的 LCS 的长度
```

```

int[][] opt = new int[substringLength1 + 1][substringLength2 + 1];

// 动态规划计算所有子问题
for (int i = substringLength1 - 1; i >= 0; i--) {
    for (int j = substringLength2 - 1; j >= 0; j--) {
        if (x.charAt(i) == y.charAt(j))
            opt[i][j] = opt[i + 1][j + 1] + 1;
        //参考上文我给的公式。
        else
            opt[i][j] = Math.max(opt[i + 1][j], opt[i][j + 1]);           //参考
    考上文我给的公式。
}

```

---

理解上段，参考上文我给的公式：

根据上述结论，可得到以下公式，

如果我们记字符串  $X_i$  和  $Y_j$  的 LCS 的长度为  $c[i, j]$ ，我们可以递归地求  $c[i, j]$ ：

$$c[i, j] = \begin{cases} 0 & \text{if } i < 0 \text{ or } j < 0 \\ c[i-1, j-1] + 1 & \text{if } i, j \geq 0 \text{ and } x_i = x_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j \geq 0 \text{ and } x_i \neq x_j \end{cases}$$

```

System.out.println("substring1:" + x);
System.out.println("substring2:" + y);
System.out.print("LCS:");

int i = 0, j = 0;
while (i < substringLength1 && j < substringLength2) {
    if (x.charAt(i) == y.charAt(j)) {
        System.out.print(x.charAt(i));
        i++;
        j++;
    } else if (opt[i + 1][j] >= opt[i][j + 1])
        i++;
    else
        j++;
}
Long endTime = System.nanoTime();

```

```

        System.out.println(" Total time is " + (endTime - startTime) + " ns");

    }

    //取得定长随机字符串
    public static String GetRandomStrings(int length){
        StringBuffer buffer = new StringBuffer("abcdefghijklmnopqrstuvwxyz");

        StringBuffer sb = new StringBuffer();
        Random r = new Random();
        int range = buffer.length();
        for (int i = 0; i < length; i++){
            sb.append(buffer.charAt(r.nextInt(range)));
        }
        return sb.toString();
    }
}

```

## 第五节、改进的算法

下面咱们来了解一种不同于动态规划法的一种新的求解最长公共子序列问题的方法,该算法主要是把求解公共字符串问题转化为求解矩阵  $L(p,m)$  的问题, 在利用定理求解矩阵的元素过程中 (1) while( $i < k$ ), $L(k,i)=null$ ,

(2) while( $L(k,i)=k$ ), $L(k,i+1)=L(k,i+2)=\dots=L(k,m)=k$ ;

求出每列元素, 一直到发现第  $p+1$  行都为  $null$  时退出循环, 得出矩阵  $L(k,m)$  后,  
 $B[L(1,m-p+1)]B[L(2,m-p+2)]\dots B[L(p,m)]$  即为 A 和 B 的 LCS, 其中 p 为 LCS 的长度。

### 5.1 主要定义及定理

- 定义 1 子序列(Subsequence): 给定字符串  $A=A[1]A[2]\dots A[m]$ , ( $A[i]$  是  $A$  的第  $i$  个字母,  $A[i] \in \Sigma$ ,  $1 \leq i \leq m$ ,  $A$  表示字符串  $A$  的长度), 字符串  $B$  是  $A$  的子序列是指  $B=A[1:i]A[2:i]\dots A[k:i]$ , 其中  $1 \leq i < 2 \leq \dots < k \leq m$ .
- 定义 2 公共子序列(Common Subsequence): 给定字符串  $A$ 、 $B$ 、 $C$ ,  $C$  称为  $A$  和  $B$  的公共子序列是指  $C$  既是  $A$  的子序列, 又是  $B$  的子序列。
- 定义 3 最长公共子序列(Longest Common Subsequence 简称 LCS): 给定字符串  $A$ 、 $B$ 、 $C$ ,  $C$  称为  $A$  和  $B$  的最长公共子序列是指  $C$  是  $A$  和  $B$  的公共子序列, 且对于  $A$  和  $B$  的任意公共子序列  $D$ , 都有  $D \leq C$ 。给定字符串  $A$  和  $B$ ,  $A=m$ ,  $B=n$ , 不妨设  $m \leq n$ , LCS 问题就是要求出  $A$  和  $B$  的 LCS。

- 定义 4 给定字符串  $A=A[1]A[2]\dots A[m]$  和字符串  $B=B[1]B[2]\dots B[n]$ ,  $A(1:i)$  表示  $A$  的连续子序列  $A[1]A[2]\dots A[i]$ , 同样  $B(1:j)$  表示  $B$  的连续子序列  $B[1]B[2]\dots B[j]$ 。  $L_i(k)$  表示所有与字符串  $A(1:i)$  有长度为  $k$  的 LCS 的字符串  $B(l:j)$  中  $j$  的最小值。用公式表示就是  $L_i(k)=\min_j(LCS(A(1:i), B(l:j)))=k$  [3]。

定理 1  $\forall i \in [1, m]$ , 有  $L_i(1) < L_i(2) < L_i(3) < \dots < L_i(m)$  .

定理 2  $\forall i \in [l, m-1], \forall k \in [l, m]$ , 有  $i \leq L(i+k) \leq i+L(k)$ .

定理 3  $\forall i \in [l, m-1], \forall k \in [l, m-1]$ , 有  $i \leq L(k) < i+L(k+l)$ .

以上三个定理都不考虑  $L_i(k)$  无定义的情况。

定理 4[3]  $i \leq L(i+k)$  如果存在, 那么它的取值必为:  $i \leq L(i+k)=\min(j, i \leq L(k))$ 。这里  $j$  是满足以下条件的最小整数:  $A[i+j]=B[j]$  且  $j > i \leq L(k-1)$ 。

	$i=1$	2	3	...	$m$
$k=1$	$L(1,1)$	<del><math>L(1,2)</math></del>	$L(1,3)$	...	$L(1,m)$
2	null	$L(2,2)$	$L(2,3)$	...	$L(2,m)$
3	null	null	$L(3,3)$	...	$L(3,m)$
...	...	...	...	...	...
p	null	null	null	null	$L(p,m)$
$p+1$	null	null	null	null	null

图 1 求解 LCS 的矩阵  $L(p,m)$   
Fig. 1  $L(p,m)$  of solving matrix LCS

矩阵中元素  $L(k, i)=L_i(k)$ , 这里( $1 \leq i \leq m$ ,  $1 \leq k \leq m$ ), null 表示  $L(k,i)$  不存在。当  $i < k$  时, 显然  $L(k, i)$  不存在。

设  $p=\text{Max}_k(L(k, m) \neq \text{null})$ , 可以证明  $L$  矩阵中  $L(p,m)$  所在的对角线,  $L(1,m-p+1), L(2,m-p+2) \dots L(p-1,m-1), L(p,m)$  所对应的子序列  $B[L(1,m-p+1)]B[L(2,m-p+2)] \dots B[L(p,m)]$  即为  $A$  和  $B$  的 LCS,  $p$  为该 LCS 的长度。这样, LCS 问题的求解就转化为对  $m \times m$   $L \times L$  矩阵的求解。

## 5.2 算法思想

根据定理, 第一步求出第一行元素,  $L(1,1), L(1,2), \dots, L(1,m)$ , 第二步求第二行, 一直到发现第  $p+1$  行都为 null 为止。在计算过程中遇到  $i < k$  时,  $L(k,i)=\text{null}$ , 及  $L(k,i)=k$  时,  $L(k,i+1)=L(k,i+2)=\dots=L(k,m)=k$ 。这样, 计算每行的时间复杂度为  $O(n)$ , 则整个时间复杂度为  $O(pn)$ 。在求  $L$  矩阵的过程中不用存储整个矩阵, 只需存储当前行和上一行即可。空间复杂度为  $O(m+n)$ 。

下面给出一个例子来说明: 给定字符串  $A$  和  $B$ ,  $A=acdabbc$ ,  $B=cddbacaba$ , ( $m=A=7$ ,  $n=B=9$ )。按照定理给出的递推公式, 求出  $A$  和  $B$  的  $L$  矩阵如图 2, 其中的\$表示 NULL。

```
c:\Documents and Settings\USER\桌面\s1串\Del
please enter A's number:7
please enter A[]:acdabbc
please enter B's number:9
please enter B[]:cddbacaba
      1   1   1   1   1   1
$   6   2   2   2   2   2
$   $   $   5   4   4   4
$   $   $   $   8   8   6
$   $   $   $   $   $   $
$   $   $   $   $   $   $
$   $   $   $   $   $   $
LCS=4$cdbc
```

图 2  $L(p,m)$   
Fig. 2  $L(p,m)$

则  $A$  和  $B$  的 LCS 为  $B[1]B[2]B[4]B[6]=cdbc$ , LCS 的长度为 4。

### 5.3 算法伪代码

算法  $L(A, B, L)$

输入 长度分别为  $m, n$  的字符串  $A, B$

输出  $A, B$  的最长公共子序列 LCS

```
L(A,B,L){//字符串 A, B, 所求矩阵 L
    for(k=1;k<=m;k++){
        for(i=1;i<=m;i++){
            if(i<k) L[k][i]=N;//i<k 时,L(k,i)=null, N 代表无穷大
            if(L[k][i]==k)//L(k,i)=k 时,L(k,i+1)=L(k,i+2)=...L(k,m)=k
                for(l=i+1;l<=m;l++){
                    { L[k][l]=k;
                    Break;}
                }
            for(j=1;j<=n;j++){//定理 4 的实现
                if(A[i+1]==B[j]&&j>L[k-1][i]){
                    L[k][i+1]=(j<L[k][i]?j:L[k][i]);
                    break;
                }
                if(L[k][i+1]==0)
                    L[k][i]=N;
            }
            if(L[k][m]==N)
                {p=k-1;break;}
        }
        p=k-1;
    }
}
```

### 5.4 结语

本节主要描述区别于动态规划法的一种新的求解最长公共子序列问题的方法，在不影响精确度的前提下，提高序列匹配的速度，根据定理  $i \leq L + (k) = \min(j, i \leq (k))$  得出矩阵，在求解矩阵的过程中对最耗时的  $L(p, m)$  进行条件约束优化。我们在 Intel(R) Core(TM)2 Quad 双核处理器、1G 内存，软件环境：Windows XP 下试验结果证明，本文算法与其他经典的比对算法相比，不但能够取得准确的结果，而且速度有了较大的提高（本节参考了刘佳梅女士的论文）。

若有任何问题，恳请不吝指正。谢谢各位。完。

## 第十二~十五章：中签概率，IP 访问次数，回文等问题（初稿）

作者：上善若水.qinyu, BigPotato, luuillu, well, July。编程艺术室出品。

### 前言

本文的全部稿件是由我们编程艺术室的部分成员：上善若水.qinyu, BigPotato, luuillu, well, July 共同完成，共分 4 个部分，即 4 道题：

- 第一部分、从一道题，漫谈数据结构、以及压缩、位图算法，由上善若水.qinyu 完成，
- 第二部分、遍历  $n$  个元素取出等概率随机取出其中之一元素，由 BigPotato 完成，
- 第三部分、提取出某日访问百度次数最多的那个 IP，由 luuillu 完成，
- 第四部分、回文判断，由 well 完成。全文由 July 统稿完成。

由于本人在这周时间上实在是过于仓促，来不及过多整理，所以我尽量保持上述我的几位伙伴的原话原文，基本没做多少改动。因此，标明为初稿，以后会更加详尽细致的进行修补完善。

前十一章请见这：[程序员编程艺术第一~十章集锦与总结](#)。吾以咱们编程艺术室的朋友为傲，以能识尽天下各路朋友为傲，以诸君为傲。

### 第一部分、从一道题，漫谈数据结构、以及压缩、位图算法

海量数据处理往往会很有趣，有趣在什么地方呢？

- 空间，`available` 的内存不够，需要反复交换内存
- 时间，速度太慢不行，毕竟那是海量数据
- 处理，数据是一次调用还是反复调用，因为针对时间和空间，通常来说，多次调用的话，势必会增加预处理以减少每次调用的时候的时间代价。

#### 题目如下

7、腾讯面试题：给 40 亿个不重复的 `unsignedint` 的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那 40 亿个数当中？

分析：1个 `unsigned int` 占用 4 字节，40亿大约是 4G 个数不到，那么一共大约要用 16G 的内存空间，如果内存不够大，反复和硬盘交换数据的话，后果不堪设想。

那么怎么储存这么多的数据呢？还记得伴随数组么？还是那种思想，利用内存地址代替下标。

先举例，在内存中应该是 1 个 `byte=8bit`，那么明显有

0 = 0000 0000

255 = 1111 1111

69 = 0100 0101

那么 69 可以表示 0.2.6 三个数存在，其余的 7 以下的数不存在，0 表示 0-7 都不存在，255 表示 0-7 都存在，这就是位图算法：通过全部置 0，存在置 1，这样一种模式来通过连续的地址存贮数据，和检验数据的方法。

那么 1 个 `unsigned int` 代表多少个数呢？1 个 `unsigned int` 是一个  $2^{32}$  以内的数，那么也就是这样的 1 个数，可以表示 32 个数是否存在。同理申请一个 `unsigned int` 的数组 `a[n]` 则可以表示连续的  $(n+1) * 32$  的数。也就是 `a[0]` 表示 0-31 的数是否存在，`a[1]` 表示 32-63 的数是否存在，依次类推。

这时候需要用多大的内存呢？

$16g/32=512M$

512M 和 16G 之间的区别，却是是否一个 32 位寻址的 CPU 能否办得到的事儿了，众所周知，32 位 CPU 最大寻址不超过 4G，固然，你会说，现在都是 64 位的 CPU 之类的云云，但是，对于底层的设计者来说，寻址范围越小越好操控的事实是不争的。

问题到这里，其实基本上已经完事了，判断本身，在位图算法这里就是找到对应的内存位置是否为 1 就可以了。

当数据超出可接受范围之后...

当然，下面就要开始说一说，当数据超出了可以接受的范围之后的事情了。比如， $2^{66}$  范围的数据检索，也会是一个问题

4 倍于 64 位 CPU 寻址范围，如果加上 CPU 本身的偏移寄存器占用的资源，可能应该是 6-8 个 64 位 U 的寻址范围，如果反复从内存到硬盘的读写，过程本身就是可怕的。

算法，更多的是用来解决瓶颈的，就想现在，根本不用考虑内存超出 8M 的问题，但是 20 年前，8086 的年代，内存 4M，或者内存 8M，你怎么处理？固然做软件的不需要完全考虑摩尔定律，但是摩尔定律绝对是影响软件和算法编写者得想法的。

再比如，乌克兰俄罗斯的一批压缩高手，比如国内有名的 R 大，为什么压缩会出现？就是因为，要么存不下，要么传输时间过长。网络再好，64G 的高清怎么的也得下载个一段时间吧。海量数据处理，永远是考虑超过了当前硬件条件的时候，该怎么办？！

那么我们可以发现一个更加有趣的问题，如果存不下，但是还要存，怎么办！

压缩！这里简单的说一嘴，无损压缩常见的为 Huffman 算法和 LZW(Lenpel-Ziv & Welch) 压缩算法，前者研究不多，后者却经常使用。

因为上面提到了位图算法，我就用常见的位图类的数据举例：

以下引自我的摘抄出处忘记了，请作者见谅：

“对原始数据 ABCCAABCDDAACCDB 进行 LZW 压缩

原始数据中，只包括 4 个字符(Character), A,B,C,D, 四个字符可以用一个 2bit 的数表示，0-A,1-B,2-C,3-D, 从最直观的角度看，原始字符串存在重复字符：ABCCAABCDDAACCDB，用 4 代表 AB,5 代表 CC，上面的字符串可以替代表示为：45A4CDDAA5DB, 这样是不是就比原数据短了一些呢！

### LZW 算法的适用范围

为了区别代表串的值(Code)和原来的单个的数据值(String)，需要使它们的数值域不重合，上面用 0-3 来代表 A-D, 那么 AB 就必须用大于 3 的数值来代替，再举另外一个例子，原来的数值范围可以用 8bit 来表示，那么就认为原始的数的范围是 0~255，压缩程序生成的标号的范围就不能为 0~255（如果是 0-255，就重复了）。只能从 256 开始，但是这样一来就超过了 8 位的表示范围了，所以必须要扩展数据的位数，至少扩展一位，但是这样不是增加了 1 个字符占用的空间了么？但是却可以用一个字符代表几个字符，比如原来 255 是 8bit, 但是现在用 256 来表示 254, 255 两个数，还是划得来的。从这个原理可以看出 LZW 算法的适用范围是原始数据串最好是有大量的子串多次重复出现，重复的越多，压缩效果越好。反之则越差，可能真的不减反增了。

伪代码如下

```
1 STRING = get input character
2 WHILE there are still input characters DO
3     CHARACTER = get input character
4     IF STRING+CHARACTER is in the string table then
5         STRING = STRING+character
6     ELSE
7         output the code for STRING
8         add STRING+CHARACTER to the string table
9         STRING = CHARACTER
10    END of IF
11 END of WHILE
12 output the code for STRING
```

看过上面的适用范围在联想本题，数据有多少种，根据同余模的原理，可以惊人的发现，其实真的非常适合压缩，但是压缩之后，尽管存下了，在查找的时候，势必又需要解码，那么又回到了我们当初学习算法时候，的那句经典话，算法本身，就是为了解决时间和空间的均衡问题，要么时间换空间，要么空间换时间。

更多的，请读者自行思考，因为，压缩本身只是想引起读者思考，已经是题外话了~本部分完--上善若水.qinyu。

## 第二部分、遍历 $n$ 个元素取出等概率随机取出其中之一元素

### 问题描述

1. 一个文件中含有  $n$  个元素，只能遍历一遍，要求等概率随机取出其中之一。

先讲一个例子，5个人抽5个签，只有一个签意味着“中签”，轮流抽签，那么这种情况，估计这5个人都不会有异议，都觉得这种方法是公平的，这确实也是公平的，“抓阄”的方法已经有很长的历史了，要是不公平的话老祖先们就不干了。

或许有人觉得先抓的人中签的概率会大一些，因为要是前面的人中了，后面的中签概率就是0了，也可能有人会觉得后面抓的人更有优势，因为前面拿去了不中的签，后面中签的概率就大，那么我们就计算一下吧。

### 问题分析

第一个人中签的概率是  $1/5$ ,

第二个人中签的情况只能在第一个人未中时才有可能,所以他中的概率是  $4/5 \times 1/4 = 1/5$  ( $4/5$  表示第一个人未中,  $1/4$  表示在剩下的 4 个签里中签的概率), 所以, 第二个人最终的中签概率也是  $1/5$ ,

同理, 第三个人中签的概率为: 第一个人未中的概率  $\times$  第二个人未中的概率  $\times$  第三个人中的概率, 即为:  $4/5 \times 3/4 \times 1/3 = 1/5$ ,

一样的可以求出第四和第五个人的概率都为  $1/5$ , 也就是说先后顺序不影响公平性。

说这个问题是要说明这种前后有关联的事件的概率计算的方式, 我们回到第 1 个问题。前几天我的一个同学面试百度是被问到这个问题, 他想了想回答说, 依次遍历, 遇到每一个元素都生成一个随机数作为标记, 如果当前生成的随机数大于为之前保留的元素生成的随机数就替换, 这样操作直到文件结束。

但面试官问到: 如果生成的随机数和之前保留的元素的随机数一样大的话, 要不要替换呢?

你也许会想, 一个 `double` 的范围可以是  $-1.79E+308 \sim +1.79E+308$ , 要让两个随机生成的 `double` 相等的概率不是一般的微乎其微啊! 但计算机世界里有条很让人伤心的“真理”: 可能发生的事件, 总会发生!

那我们遇到这种情况, 是换还是不换? `To be or not to be, that's a question!`

就好比, 两个人百米赛跑, 测出来的时间一样, 如果只能有一个人得冠军的话, 对于另一个人始终是不公平的, 那么只能再跑一次, 一决雌雄了!

## 我的策略

下面, 说一个个人认为比较满足要求的选取策略:

- 顺序遍历, 当前遍历的元素为第  $L$  个元素, 变量  $e$  表示之前选取了的某一个元素, 此时生成一个随机数  $r$ , 如果  $r \% L == 0$ (当然 0 也可以是  $0 \sim L-1$  中的任何一个, 概率都是一样的), 我们将  $e$  的值替换为当前值, 否则扫描下一个元素直到文件结束。

你要是给面试官说明了这样一个策略后, 面试官百分之一千会问你这样做是等概率吗? 那我们来证明一下。

## 证明

在遍历到第 1 个元素的时候，即  $L$  为 1，那么  $r\%L$  必然为 0，所以  $e$  为第一个元素， $p=100\%$ ，

遍历到第 2 个元素时， $L$  为 2， $r\%L==0$  的概率为  $1/2$ ，这个时候，第 1 个元素不被替换的概率为  $1 \times (1-1/2) = 1/2$ ，

第 1 个元素被替换，也就是第 2 个元素被选中的概率为  $1/2 = 1/2$ ，你可以看到，只有 2 时，这两个元素是等概率的机会被选中的。

继续，遍历到第 3 个元素的时候， $r\%L==0$  的概率为  $1/3$ ，前面被选中的元素不被替换的概率为  $1/2 \times (1-1/3) = 1/3$ ，前面被选中的元素被替换的概率，即第 3 个元素被选中的概率为  $1/3$

归纳法证明，这样走到第  $L$  个元素时，这  $L$  个元素中任一被选中的概率都是  $1/L$ ，那么走到  $L+1$  时，第  $L+1$  个元素选中的概率为  $1/(L+1)$ ，之前选中的元素不被替换，即继续被选中的概率为  $1/L \times (1-1/(L+1)) = 1/(L+1)$ 。证毕。

也就是说，走到文件最后，每一个元素最终被选出的概率为  $1/n$ ， $n$  为文件中元素的总数。好歹我们是一个技术博客，看不到一丁点代码多少有点遗憾，给出一个选取策略的伪代码，如下：

## 伪代码

Element RandomPick(file):

Int length = 1;

While( length <= file.size )

If( rand() % length == 0 )

Picked = File[length];

Length++;

Return picked

近日，看见我的一些同学在他们的面经里面常推荐结构之法算法之道这个博客，感谢东南大学计算机学院即将找工作的同学们对本博的关注，欢迎批评指正！--**BigPotato**。

## 第三部分、提取出某日访问百度次数最多的那个 IP

问题描述：海量日志数据，提取出某日访问百度次数最多的那个 IP。

方法：计数法

假设一天之内某个 IP 访问百度的次数不超过 40 亿次，则访问次数可以用 `unsigned` 表示。用数组统计出每个 IP 地址出现的次数，即可得到访问次数最大的 IP 地址。

IP 地址是 32 位的二进制数，所以共有  $N=2^{32}=4G$  个不同的 IP 地址，创建一个 `unsigned count[N]` 的数组，即可统计出每个 IP 的访问次数，而 `sizeof(count) == 4G*4=16G`，远远超过了 32 位计算机所支持的内存大小，因此不能直接创建这个数组。下面采用划分法解决这个问题。

假设允许使用的内存是 512M， $512M/4=128M$  即 512M 内存可以统计 128M 个不同的 IP 地址的访问次数。而  $N/128M = 4G/128M = 32$ ，所以只要把 IP 地址划分成 32 个不同的区间，分别统计出每个区间中访问次数最大的 IP，然后就可以计算出所有 IP 地址中访问次数最大的 IP 了。

因为  $2^5=32$ ，所以可以把 IP 地址的最高 5 位作为区间编号，剩下的 27 位作为区间内的值，建立 32 个临时文件，代表 32 个区间，把相同区间的 IP 地址保存到同一的临时文件中。

例如：

`ip1=0x1f4e2342`

`ip1` 的高 5 位是 `id1 = ip1 >> 27 = 0x11 = 3`

`ip1` 的其余 27 位是 `value1 = ip1 & 0x07ffff = 0x074e2342`

所以把 `value1` 保存在 `tmp3` 文件中。

由 `id1` 和 `value1` 可以还原成 `ip1`，即 `ip1 = (id1 << 27) | value1`

按照上面的方法可以得到 32 个临时文件，每个临时文件中的 IP 地址的取值范围属于 [0-128M)，因此可以统计出每个 IP 地址的访问次数。从而找到访问次数最大的 IP 地址。

程序源码：

test.cpp 是 c++ 源码。

```
#include <fstream>
#include <iostream>
#include <ctime>

using namespace std;
#define N 32           //临时文件数

#define ID(x)  (x>>27)          //x 对应的文件编号
#define VALUE(x) (x&0x07fffffff) //x 在文件中保存的值
#define MAKE_IP(x,y) ((x<<27)|y) //由文件编号和值得到 IP 地址.

#define MEM_SIZE 128*1024*1024    //需分配内存的大小
为 MEM_SIZE*sizeof(unsigned)

char* data_path="D:/test/ip.dat";      //ip 数据

//产生 n 个随机 IP 地址
void make_data(const int& n)
{
    ofstream out(data_path,ios::out|ios::binary);
    srand((unsigned)(time(NULL)));
    if (out)
    {
        for (int i=0; i<n; ++i)
        {
            unsigned val=unsigned(rand());
            val = (val<<24)|val;          //产生 unsigned 类型的随机数

            out.write((char *)&val,sizeof (unsigned));
        }
    }
}

//找到访问次数最大的 ip 地址
int main()
{
    //make_data(100);      //
    make_data(100000000); //产生测试用的 IP 数据
    fstream arr[N];
```

```

for (int i=0; i<N; ++i) //创建 N 个临时文件
{
    char tmp_path[128]; //临时文件路径
    sprintf(tmp_path, "D:/test/tmp%d.dat", i);
    arr[i].open(tmp_path, ios::trunc|ios::in|ios::out|ios::binary); //打开第 i
    个文件

    if( !arr[i])
    {
        cout<<"open file"<<i<<"error"<<endl;
    }
}

ifstream infile(data_path,ios::in|ios::binary); //读入测试用的 IP 数据
unsigned data;

while(infile.read((char*)(&data), sizeof(data)))
{
    unsigned val=VALUE(data);
    int key=ID(data);
    arr[ID(data)].write((char*)(&val), sizeof(val)); //保存到临时文件
    件中
}

for(unsigned i=0; i<N; ++i)
{
    arr[i].seekg(0);
}
unsigned max_ip = 0; //出现次数最多的 ip 地址
unsigned max_times = 0; //最大只出现的次数

//分配 512M 内存,用于统计每个数出现的次数
unsigned *count = new unsigned[MEM_SIZE];

for (unsigned i=0; i<N; ++i)
{
    memset(count, 0, sizeof(unsigned)*MEM_SIZE);

    //统计每个临时文件件中不同数字出现的次数
    unsigned data;
    while(arr[i].read((char*)(&data), sizeof(unsigned)))
    {
        ++count[data];
    }
}

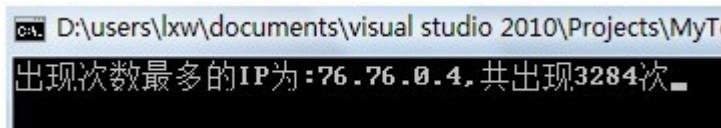
```

```

//找出出现次数最多的 IP 地址
for(unsigned j=0; j<MEM_SIZE; ++j)
{
    if(max_times<count[j])
    {
        max_times = count[j];
        max_ip = MAKE_IP(i,j);           // 恢复成原 ip 地址.
    }
}
delete[] count;
unsigned char *result=(unsigned char *)(&max_ip);
printf("出现次数最多的 IP 为:%d.%d.%d.%d,共出现%d 次",
      result[0], result[1], result[2], result[3], max_times);
}

```

执行结果.



--luuillu。

## 第四部分、回文判断

(初稿，写的比较急，请审阅、增补、修改)

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

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

### 一、回文判断

那么，我们的第一个问题就是：判断一个字串是否是回文

通过对回文字符串的考察，最直接的方法显然是将字符串逆转，存入另外一个字符串，然后比较原字符串和逆转后的字符串是否一样，一样就是回文，这个方法的时空复杂度都是  $O(n)$ 。

我们还很容易想到只要从两头开始同时向中间扫描字串，如果直到相遇两端的字符都一样，那么这个字串就是一个回文。我们只需要维护头部和尾部两个扫描指针即可，代码如下：

```
/*
 *check weather s is a palindrome, n is the length of string s
 *Copyright(C) fairywell 2011
 */
bool IsPalindrome(const char *s, int n)
{
    if (s == 0 || n < 1) return false; // invalid string
    char *front, *back;
    front = s; back = s + n - 1; // set front and back to the begin and endof the string
    while (front < back) {
        if (*front != *back) return false; // not a palindrome
        ++front; --back;
    }
    return true; // check over, it's a palindrome
}
```

这是一个直白且效率不错的实现，只需要附加 2 个额外的指针，在  $O(n)$  时间内我们可以判断出字符串是否是回文。

是否还有其他方法？呵呵，聪明的读者因该想到了不少变种吧，不过时空复杂度因为不会有显著提升了（为什么？），下面再介绍一种回文判断方法，先上代码：

```
/*
 *check weather s is a palindrome, n is the length of string s
 *Copyright(C) fairywell 2011
 */
bool IsPalindrome2(const char *s, int n)
{
    if (s == 0 || n < 1) return false; // invalid string
    char *first, *second;
    int m = ((n>>1) - 1) >= 0 ? (n>>1) - 1 : 0; // m is themiddle point of s
    first = s + m; second = s + n - 1 - m;
    while (first >= s)
```

```
    if (s[first--] != s[second++]) return false; // not equal, so it's not a palindrome
}
return true; // check over, it's a palindrome
```

代码略有些小技巧，不过相信我们聪明的读者已经看清了意思，这里就是从中间开始、向两边扩展查看字符是否相等的一种方法，时空复杂度和上一个方法是一模一样的，既然一样，那么我们为什么还需要这种方法呢？首先，世界的美存在于它的多样性；其次，我们很快会看到，在某些回文问题里面，这个方法有着自己的独到之处，可以方便的解决一类问题。

那么除了直接用数组，我们还可以采用其他的数据结构来判断回文吗呢？请读者朋友稍作休息想想看。相信我们聪明的读者肯定想到了不少好方法吧，也一定想到了经典的单链表和栈这两种方法吧，这也是面试中常常出现的两种回文数据结构类型。

对于单链表结构，处理的思想不难想到：用两个指针从两端或者中间遍历并判断对应字符是否相等。所以这里的关键就是如何朝两个方向遍历。单链表是单向的，所以要向两个方向遍历不太容易。一个简单的方法是，用经典的快慢指针的方法，定位到链表的中间位置，将链表的后半逆置，然后用两个指针同时从链表头部和中间开始同时遍历并比较即可。

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

## 二、回文的应用

我们已经了解了回文的判断方法，接下来可以来尝试回文的其他应用了。回文不是很简单的东西吗，还有其他应用？是的，比如：[查找一个字符串中的最长回文字串](#)

Hum，还是请读者朋友们先自己想想看看。嗯，有什么好方法了吗？枚举所有的子串，分别判断其是否为回文？这个思路是正确的，但却做了很多无用功，如果一个长的子串包含另一个短一些的子串，那么对子串的回文判断其实是不需要的。

那么如何高效的进行判断呢？既然对短的子串的判断和包含它的长的子串的判断重复了，我们何不复用下短的子串的判断呢（哈，算法里也跑出软件工程了），让短的子串的判断成为长的子串的判断的一个部分！想到怎么做了吗？Aha，没错，扩展法。从一个字符开始，向两边扩展，看看最多能到多长，使其保持为回文。这也就是为什么我们在上一节里面要提出 [IsPalindrome2](#) 的原因。

具体而言，我们可以枚举中心位置，然后再在该位置上用扩展法，记录并更新得到的最长的回文长度，即为所求。代码如下：

```
/*
*find the longest palindrome in a string, n is the length of string s
*Copyright(C) fairywell 2011
*/
int LongestPalindrome(const char *s, int n)
{
    int i, j, max;
    if (s == 0 || n < 1) return 0;
    max = 0;
    for (i = 0; i < n; ++i) { // i is the middle point of the palindrome
        for (j = 0; (i-j >= 0) && (i+j < n); ++j) // if the length of the palindrome
is odd
            if (s[i-j] != s[i+j]) break;
        if (j*2+1 > max) max = j * 2 + 1;
        for (j = 0; (i-j >= 0) && (i+j+1 < n); ++j) // for the even case
            if (s[i-j] != s[i+j+1]) break;
        if (j*2+2 > max) max = j * 2 + 2;
    }
    return max;
}
```

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

当然，还有更先进但也更复杂的方法，比如用 `s` 和逆置 `s'` 的组合 `s$s'` 来建立后缀树的方法也能找到最长回文，但构建的过程比较复杂，所以在实践中用的比较少，感兴趣的朋友可以参考相应资料。

回文的内容还有不少，但主要的部分通过上面的内容相信大家已经掌握，希望大家能抓住实质，在实践中灵活运用，回文的内容我就暂时介绍到这里了，谢谢大家--well。

#### 附注：

- 如果读者对本文的内容或形式，语言和表达不甚满意。完全理解。我之前也跟编程艺术室内的朋友们开玩笑的说：我暂不做任何修改，题目会标明为初稿。这样的话，你们才会相信或者知晓，你们的才情只有通过我的语言和表达才能最大限度的发挥

出来，被最广泛的人轻而易举的所认同和接受（不过，以上 4 位兄弟的思维灵活度都在本人之上）。呵呵，开个玩笑。

- 本文日后会抽取时间再做修补和完善。若有任何问题，欢迎随时不吝指正。谢谢。

完。

## 第十六~第二十章：全排列，跳台阶，奇偶排序，第一个只出现一次等问题

作者：July、2011.10.16。

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

### 引言

最近这几天闲职在家，一忙着投简历，二为准备面试而搜集整理各种面试题。故常常关注个人所建的 Algorithms1-14 群内朋友关于笔试，面试，宣讲会，offer，薪资的讨论以及在群内发布的各种笔/面试题，常感言道：咱们这群人之前已经在学校受够了学校的那种应试教育，如今出来找工作又得东奔西走去参加各种笔试/面试，着实亦不轻松。幻想，如果在企业与求职者之间有个中间面试服务平台就更好了。

ok，闲话少扯。在上一篇文章中，已经说过，“个人正在针对那 100 题一题一题的写文章，多种思路，不断优化，即成程序员编程艺术系列。”现本编程艺术系列继续开始创作，你而后自会和我有同样的感慨：各种面试题千变万化，层出不穷，但基本类型，解决问题的思路基本一致。

本文为程序员编程艺术第十六章~第二十章，包含以下 5 个问题：

1. 全排列；
2. 跳台阶；
3. 奇偶排序；
4. 第一个只出现一次的字符；
5. 一致性哈希算法。

同时，本文会在解答去年微软面试 100 题的部分题目时，尽量结合今年最近各大 IT 公司最新的面试题来讲解，两相对比，彼此对照，相信你会更加赞同我上面的话。且本文也不奢望读者能从中学到什么高深技术之类的东西，只求读者看此文看着舒服便可，通顺流畅以致一口气读完而无任何压力。ok，有任何问题，欢迎不吝指正。谢谢。

## 第十六章、全排列问题

### 53. 字符串的排列。

题目：输入一个字符串，打印出该字符串中字符的所有排列。

例如输入字符串 abc，则输出由字符 a、b、c 所能排列出来的所有字符串  
abc、acb、bac、bca、cab 和 cba。

分析：此题最初整理于去年的微软面试 100 题中第 53 题，第二次整理于微软、Google 等公司非常好的面试题及解答[第 61-70 题]第 67 题。无独有偶，这个问题今年又出现于今年的 2011.10.09 百度笔试题中。ok，接下来，咱们先好好分析这个问题。

## 一、递归实现

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

固定 a，求后面 bc 的排列：abc，acb，求好后，a 和 b 交换，得到 bac

固定 b，求后面 ac 的排列：bac，bca，求好后，c 放到第一位置，得到 cba

固定 c，求后面 ba 的排列：cba，cab。代码可如下编写所示：

```
template <typename T>
void CalcAllPermutation_R(T perm[], int first, int num)
{
    if (num <= 1) {
        return;
    }

    for (int i = first; i < first + num; ++i) {
        swap(perm[i], perm[first]);
        CalcAllPermutation_R(perm, first + 1, num - 1);
        swap(perm[i], perm[first]);
    }
}
```

或者如此编写，亦可：

```
void Permutation(char* pStr, char* pBegin);

void Permutation(char* pStr)
{
    Permutation(pStr, pStr);
}

void Permutation(char* pStr, char* pBegin)
```

```

{
    if(!pStr || !pBegin)
        return;

    if(*pBegin == '\0')
    {
        printf("%s\n", pStr);
    }
    else
    {
        for(char* pCh = pBegin; *pCh != '\0'; ++ pCh)
        {
            // swap pCh and pBegin
            char temp = *pCh;
            *pCh = *pBegin;
            *pBegin = temp;

            Permutation(pStr, pBegin + 1);
            // restore pCh and pBegin
            temp = *pCh;
            *pCh = *pBegin;
            *pBegin = temp;
        }
    }
}

```

## 二、字典序排列

把升序的排列（当然，也可以实现为降序）作为当前排列开始，然后依次计算当前排列的下一个字典序排列。

对当前排列从后向前扫描，找到一对为升序的相邻元素，记为  $i$  和  $j$  ( $i < j$ )。如果不存在这样一对为升序的相邻元素，则所有排列均已找到，算法结束；否则，重新对当前排列从后向前扫描，找到第一个大于  $i$  的元素  $k$ ，交换  $i$  和  $k$ ，然后对从  $j$  开始到结束的子序列反转，则此时得到的新排列就为下一个字典序排列。这种方式实现得到的所有排列是按字典序有序的，这也是 C++ STL 算法 `next_permutation` 的思想。算法实现如下：

```

template <typename T>
void CalcAllPermutation(T perm[], int num)
{
    if (num < 1)
        return;

```

```

while (true) {
    int i;
    for (i = num - 2; i >= 0; --i) {
        if (perm[i] < perm[i + 1])
            break;
    }

    if (i < 0)
        break; // 已经找到所有排列

    int k;
    for (k = num - 1; k > i; --k) {
        if (perm[k] > perm[i])
            break;
    }

    swap(perm[i], perm[k]);
    reverse(perm + i + 1, perm + num);

}
}

```

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

## 第十七章、跳台阶问题

### 27.跳台阶问题

题目：一个台阶总共有  $n$  级，如果一次可以跳 1 级，也可以跳 2 级。

求总共有多少总跳法，并分析算法的时间复杂度。

分析：在九月腾讯，创新工场，淘宝等公司最新面试十三题中第 23 题又出现了这个问题，题目描述如下：**23、人人笔试 1：**一个人上台阶可以一次上 1 个，2 个，或者 3 个，问这个人上  $n$  层的台阶，总共有几种走法？咱们先撇开这个人人笔试的问题（其实差别就在于人人笔试题中多了一次可以跳三级的情况而已），先来看这个第 27 题。

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

现在我们再来讨论一般情况。我们把  $n$  级台阶时的跳法看成是  $n$  的函数，记为  $f(n)$ 。当  $n > 2$  时，第一次跳的时候就有两种不同的选择：一是第一次只跳 1 级，此时跳法数目等于后面剩下的  $n-1$  级台阶的跳法数目，即为  $f(n-1)$ ；另外一种选择是第一次跳 2 级，此时跳法数目等于后面剩下的  $n-2$  级台阶的跳法数目，即为  $f(n-2)$ 。因此  $n$  级台阶时的不同跳法的总数  $f(n) = f(n-1) + f(n-2)$ 。

我们把上面的分析用一个公式总结如下：

$$f(n) = \begin{cases} 1 & n=1 \\ 2 & n=2 \\ f(n-1) + f(n-2) & n>2 \end{cases}$$

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

```
long long Fibonacci_Solution1(unsigned int n)
{
    int result[2] = {0, 1};
    if(n < 2)
        return result[n];

    return Fibonacci_Solution1(n - 1) + Fibonacci_Solution1(n - 2);
}
```

那么，如果是人人笔试那道题呢？一个人上台阶可以一次上 1 个，2 个，或者 3 个，岂不是可以轻而易举的写下如下公式：

$$f(n) = \begin{cases} 1 & n=1 \\ 2 & n=2 \\ 4 & n=3 \\ f(n-1)+f(n-2)+f(n-3) & n>3 \end{cases}$$

行文至此，你可能会认为问题已经解决了，但事实上没有：

1. 用递归方法计算的时间复杂度是以  $n$  的指数的方式递增的，我们可以尝试用递推方法解决。具体如何操作，读者自行思考。
2. 有一种方法，能在  $O(\log n)$  的时间复杂度内求解 Fibonacci 数列问题，你能想到么？

3. 同时，有朋友指出对于这个台阶问题只需求幂就可以了（求复数幂 C++库里有），不用任何循环且复杂度为  $O(1)$ ，如下图所示，是否真如此？：

$$f(n) = f(n-1) + f(n-2) + f(n-3)$$

$$f(1) = 1, f(2) = 2, f(3) = 4;$$

$$x_1 = 1.839286755214161$$

$$x_2 = -0.419643377607081 + 0.606290729207199i$$

$$x_3 = -0.419643377607081 - 0.606290729207199i$$

$$A = 0.618419922319393$$

$$B = 0.190790038840305 - 0.018700583111741i$$

$$C = 0.190790038840304 + 0.018700583111740i$$

$$f(n) = Ax_1^n + Bx_2^n + Cx_3^n, n=1,2,3\dots$$

## 第十八章、奇偶调序

54. 调整数组顺序使奇数位于偶数前面。

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

分析：

1. 你当然可以从头扫描这个数组，每碰到一个偶数时，拿出这个数字，并把位于这个数字后面的所有数字往前挪动一位。挪完之后在数组的末尾有一个空位，这时把该偶数放入这个空位。由于碰到一个偶数，需要移动  $O(n)$  个数字，只是这种方法总的时间复杂度是  $O(n^2)$ ，不符合要求，pass。
2. 很简单，维护两个指针，一个指针指向数组的第一个数字，向后移动；一个指针指向最后一个数字，向前移动。如果第一个指针指向的数字是偶数而第二个指针指向的数字是奇数，我们就交换这两个数字。

思路有了，接下来，写代码实现：

```
//思路，很简答，俩指针，一首一尾
```

```

//如果第一个指针指向的数字是偶数而第二个指针指向的数字是奇数,
//我们就交换这两个数字

// 2 1 3 4 6 5 7
// 7 1 3 4 6 5 2
// 7 1 3 5 6 4 2

//如果限制空间复杂度为 O (1), 时间为 O (N), 且奇偶数之间相对顺序不变, 就相当于正负数间顺序调整的那道题了。

//copyright@2010 zhedahht.

void Reorder(int *pData, unsigned int length, bool (*func)(int));
bool isEven(int n);
void ReorderOddEven(int *pData, unsigned int length)
{
    if(pData == NULL || length == 0)
        return;

    Reorder(pData, length, isEven);
}

void Reorder(int *pData, unsigned int length, bool (*func)(int))
{
    if(pData == NULL || length == 0)
        return;
    int *pBegin = pData;
    int *pEnd = pData + length - 1;
    while(pBegin < pEnd)
    {
        // if *pBegin does not satisfy func, move forward
        if(!func(*pBegin)) //偶数
        {
            pBegin++;
            continue;
        }

        // if *pEnd does not satisfy func, move backward
        if(func(*pEnd)) //奇数
        {
            pEnd--;
            continue;
        }
        // if *pBegin satisfy func while *pEnd does not,
        // swap these integers
        int temp = *pBegin;

```

```

        *pBegin = *pEnd;
        *pEnd = temp;
    }
}

bool isEven(int n)
{
    return (n & 1) == 0;
}

```

细心的读者想必注意到了上述程序注释中所说的“如果限制空间复杂度为  $O(1)$ ，时间为  $O(N)$  就相当于正负数间顺序调整的那道题了”，没错，它与个人之前整理的一文中的第 5 题极其类似：5、一个未排序整数数组，有正负数，重新排列使负数排在正数前面，并且要求不改变原来的正负数之间相对顺序 比如： input: 1,7,-5,9,-12,15 ans: -5,-12,1,7,9,15 要求时间复杂度  $O(N)$ ,空间  $O(1)$ 。(此题一直没看到令我满意的答案，一般达不到题目所要求的：时间复杂度  $O(N)$ ，空间  $O(1)$ ，且保证原来正负数之间的相对位置不变)。

如果你想到了绝妙的解决办法，不妨在本文评论下告知于我，或者来信指导(zhoulei0907@yahoo.cn)，谢谢。

## 第十九章、第一个只出现一次的字符

第 17 题：题目：在一个字符串中找到第一个只出现一次的字符。如输入 abaccdeff，则输出 b。

分析：这道题是 2006 年 google 的一道笔试题。它在今年又出现了，不过换了一种形式。即最近的搜狐笔试大题：数组非常长，如何找到第一个只出现一次的数字，说明算法复杂度。此问题已经在程序员编程艺术系列第二章中有所阐述，在此不再作过多讲解。

代码，可编写如下：

```

#include <iostream>
using namespace std;

//查找第一个只出现一次的字符，第 1 个程序
//copyright@ Sorehead && July
//July、updated, 2011.04.24.
char find_first_unique_char(char *str)
{
    int data[256];
    char *p;

    if (str == NULL)

```

```

    return '\0';

    memset(data, 0, sizeof(data)); //数组元素先全部初始化为 0
    p = str;
    while (*p != '\0')
        data[(unsigned char)*p++]++; //遍历字符串，在相应位置++，(同时，下标强制转换)

    while (*str != '\0')
    {
        if (data[(unsigned char)*str] == 1) //最后，输出那个第一个只出现次数为 1 的字
符
            return *str;

        str++;
    }

    return '\0';
}

int main()
{
    char *str = "afaccde";
    cout << find_first_unique_char(str) << endl;
    return 0;
}

```

当然，代码也可以这么写（测试正确）：

```

//查找第一个只出现一次的字符，第 2 个程序
//copyright@ yansha
//July、updated, 2011.04.24.
char FirstNotRepeatChar(char* pString)
{
    if(!pString)
        return '\0';

    const int tableSize = 256;
    int hashTable[tableSize] = {0}; //存入数组，并初始化为 0

    char* pHshKey = pString;
    while(*(pHshKey) != '\0')
        hashTable[*pHshKey]++;

    while(*pString != '\0')
    {
        if(hashTable[*pString] == 1)

```

```
    return *pString;

    pString++;
}

return '\0'; //没有找到满足条件的字符，退出
}
```

## 第二十章、一致性哈希算法

tencent2012 笔试题附加题

问题描述： 例如手机朋友网有  $n$  个服务器，为了方便用户的访问会在服务器上缓存数据，因此用户每次访问的时候最好能保持同一台服务器。

已有的做法是根据  $\text{ServerIPIndex}[\text{QQNUM}\%n]$  得到请求的服务器，这种方法很方便将用户分到不同的服务器上去。但是如果一台服务器死掉了，那么  $n$  就变为了  $n-1$ ，那么  $\text{ServerIPIndex}[\text{QQNUM}\%n]$  与  $\text{ServerIPIndex}[\text{QQNUM}\%(n-1)]$  基本上都不一样了，所以大多数用户的请求都会转到其他服务器，这样会发生大量访问错误。

问： 如何改进或者换一种方法，使得：

- (1) 一台服务器死掉后，不会造成大面积的访问错误；
- (2) 原有的访问基本还是停留在同一台服务器上；
- (3) 尽量考虑负载均衡。（思路：往分布式一致哈希算法方面考虑。）

1. 最土的办法还是用模余方法：做法很简单，假设有  $N$  台服务器，现在完好的是  $M$  ( $M \leq N$ )，先用  $N$  求模，如果不落在完好的机器上，然后再用  $N-1$  求模，直到  $M$ 。这种方式对于坏的机器不多的情况下，具有更好的稳定性。
2. 一致性哈希算法。

下面，本文剩下部分重点来讲讲这个一致性哈希算法。

## 应用场景

在做服务器负载均衡时候可供选择的负载均衡的算法有很多，包括： 轮循算法 (Round Robin)、哈希算法 (HASH)、最少连接算法 (Least Connection)、响应速度算法 (Response Time)、加权法 (Weighted) 等。其中哈希算法是最为常用的算法。

典型的应用场景是： 有  $N$  台服务器提供缓存服务，需要对服务器进行负载均衡，将请求平均分发到每台服务器上，每台机器负责  $1/N$  的服务。

常用的算法是对 `hash` 结果取余数 (`hash() mod N`): 对机器编号从 0 到  $N-1$ , 按照自定义的 `hash()` 算法, 对每个请求的 `hash()` 值按  $N$  取模, 得到余数  $i$ , 然后将请求分发到编号为  $i$  的机器。但这样的算法方法存在致命问题, 如果某一台机器宕机, 那么应该落在该机器的请求就无法得到正确的处理, 这时需要将当掉的服务器从算法中去除, 此时会有  $(N-1)/N$  的服务器的缓存数据需要重新进行计算; 如果新增一台机器, 会有  $N/(N+1)$  的服务器的缓存数据需要进行重新计算。对于系统而言, 这通常是不可接受的颠簸 (因为这意味着大量缓存的失效或者数据需要转移)。那么, 如何设计一个负载均衡策略, 使得受到影响的请求尽可能的少呢?

在 Memcached、Key-Value Store、BitTorrent DHT、LVS 中都采用了 Consistent Hashing 算法, 可以说 Consistent Hashing 是分布式系统负载均衡的首选算法。

## Consistent Hashing 算法描述

下面以 Memcached 中的 Consistent Hashing 算法为例说明。

consistent hashing 算法早在 1997 年就在论文 [Consistent hashing and random trees](#) 中被提出, 目前在 cache 系统中应用越来越广泛;

### 1 基本场景

比如你有  $N$  个 cache 服务器 (后面简称 `cache`), 那么如何将一个对象 `object` 映射到  $N$  个 `cache` 上呢, 你很可能会采用类似下面的通用方法计算 `object` 的 `hash` 值, 然后均匀的映射到  $N$  个 `cache`:

`hash(object)%N`

一切都运行正常, 再考虑如下的两种情况:

1. 一个 `cache` 服务器  $m$  down 掉了 (在实际应用中必须要考虑这种情况), 这样所有映射到 `cache m` 的对象都会失效, 怎么办, 需要把 `cache m` 从 `cache` 中移除, 这时候 `cache` 是  $N-1$  台, 映射公式变成了 `hash(object)%(N-1)`;
2. 由于访问加重, 需要添加 `cache`, 这时候 `cache` 是  $N+1$  台, 映射公式变成了 `hash(object)%(N+1)`;

1 和 2 意味着什么? 这意味着突然之间几乎所有的 `cache` 都失效了。对于服务器而言, 这是一场灾难, 洪水般的访问都会直接冲向后台服务器; 再来考虑第三个问题, 由于硬件能力越来越强, 你可能想让后面添加的节点多做点活, 显然上面的 `hash` 算法也做不到。

有什么方法可以改变这个状况呢，这就是 **consistent hashing**。

## 2 hash 算法和单调性

Hash 算法的一个衡量指标是单调性（Monotonicity），定义如下：

单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。

容易看到，上面的简单 hash 算法  $\text{hash}(\text{object}) \% N$  难以满足单调性要求。

## 3 consistent hashing 算法的原理

consistent hashing 是一种 hash 算法，简单的说，在移除 / 添加一个 cache 时，它能够尽可能小的改变已存在 key 映射关系，尽可能的满足单调性的要求。

下面就来按照 5 个步骤简单讲讲 consistent hashing 算法的基本原理。

### 3.1 环形 hash 空间

考虑通常的 hash 算法都是将 value 映射到一个 32 位的 key 值，也即是  $0 \sim 2^{32}-1$  次方的数值空间；我们可以将这个空间想象成一个首（0）尾（ $2^{32}-1$ ）相接的圆环，如下面图 1 所示的那样。

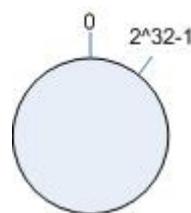


图 1 环形 hash 空间

### 3.2 把对象映射到 hash 空间

接下来考虑 4 个对象  $\text{object1} \sim \text{object4}$ ，通过 hash 函数计算出的 hash 值 key 在环上的分布如图 2 所示。

$\text{hash}(\text{object1}) = \text{key1};$

.....

hash(object4) = key4;

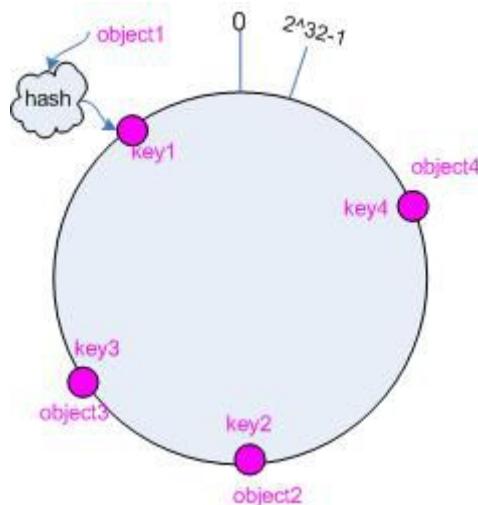


图 2 4 个对象的 key 值分布

### 3.3 把 cache 映射到 hash 空间

Consistent hashing 的基本思想就是将对象和 cache 都映射到同一个 hash 数值空间中，并且使用相同的 hash 算法。

假设当前有 A,B 和 C 共 3 台 cache，那么其映射结果将如图 3 所示，他们在 hash 空间中，以对应的 hash 值排列。

hash(cache A) = key A;

.....

hash(cache C) = key C;

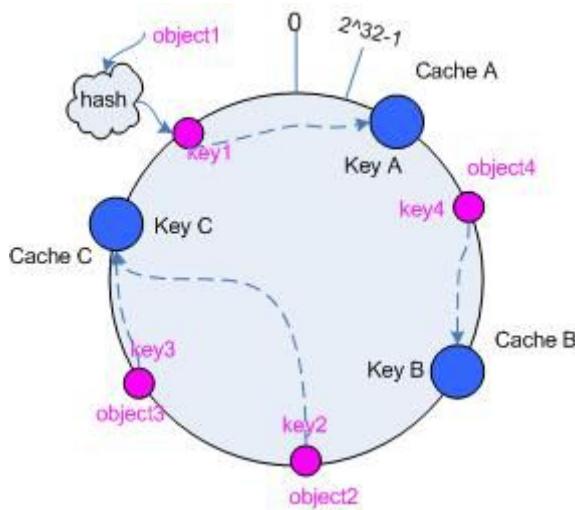


图 3 cache 和对象的 key 值分布

说到这里，顺便提一下 cache 的 hash 计算，一般的方法可以使用 cache 机器的 IP 地址或者机器名作为 hash 输入。

### 3.4 把对象映射到 cache

现在 cache 和对象都已经通过同一个 hash 算法映射到 hash 数值空间中了，接下来要考虑的就是如何将对象映射到 cache 上面了。

在这个环形空间中，如果沿着顺时针方向从对象的 key 值出发，直到遇见一个 cache，那么就将该对象存储在这个 cache 上，因为对象和 cache 的 hash 值是固定的，因此这个 cache 必然是唯一和确定的。这样不就找到了对象和 cache 的映射方法了吗？！

依然继续上面的例子(参见图 3)，那么根据上面的方法，对象 object1 将被存储到 cache A 上； object2 和 object3 对应到 cache C； object4 对应到 cache B；

### 3.5 考察 cache 的变动

前面讲过，通过 hash 然后求余的方法带来的最大问题就在于不能满足单调性，当 cache 有所变动时，cache 会失效，进而对后台服务器造成巨大的冲击，现在就来分析分析 consistent hashing 算法。

#### 3.5.1 移除 cache

考慮假设 cache B 挂掉了，根据上面讲到的映射方法，这时受影响的将仅是那些沿 cache B 顺时针遍历直到下一个 cache (cache C) 之间的对象，也即是本来映射到 cache B 上的那些对象。

因此这里仅需要变动对象 object4，将其重新映射到 cache C 上即可；参见图 4。

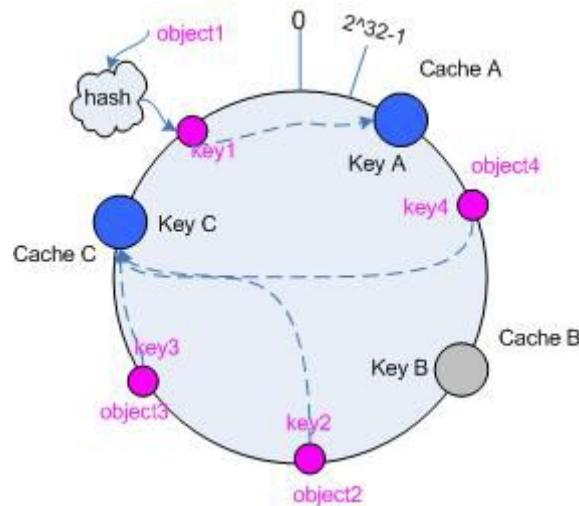


图 4 Cache B 被移除后的 cache 映射

### 3.5.2 添加 cache

再考虑添加一台新的 cache D 的情况，假设在这个环形 hash 空间中，cache D 被映射在对象 object2 和 object3 之间。这时受影响的将仅是那些沿 cache D 逆时针遍历直到下一个 cache (cache B) 之间的对象（它们是也本来映射到 cache C 上对象的一部分），将这些对象重新映射到 cache D 上即可。

因此这里仅需要变动对象 object2，将其重新映射到 cache D 上；参见图 5。

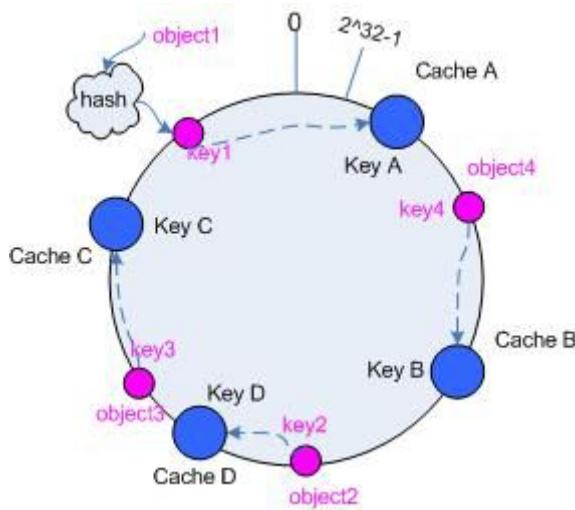


图 5 添加 cache D 后的映射关系

## 4 虚拟节点

考量 Hash 算法的另一个指标是平衡性 (Balance)， 定义如下：

平衡性

平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。

hash 算法并不是保证绝对的平衡，如果 cache 较少的话，对象并不能被均匀的映射到 cache 上，比如在上面的例子中，仅部署 cache A 和 cache C 的情况下，在 4 个对象中，cache A 仅存储了 object1，而 cache C 则存储了 object2、object3 和 object4；分布是很不均衡的。

为了解决这种情况，consistent hashing 引入了“虚拟节点”的概念，它可以如下定义：

“虚拟节点”（virtual node）是实际节点在 hash 空间的复制品（replica），一实际个节点对应了若干个“虚拟节点”，这个对应个数也成为“复制个数”，“虚拟节点”在 hash 空间中以 hash 值排列。

仍以仅部署 cache A 和 cache C 的情况为例，在图 4 中我们已经看到，cache 分布并不均匀。现在我们引入虚拟节点，并设置“复制个数”为 2，这就意味着一共会存在 4 个“虚拟节点”，cache A1, cache A2 代表了 cache A；cache C1, cache C2 代表了 cache C；假设一种比较理想的情况，参见图 6。

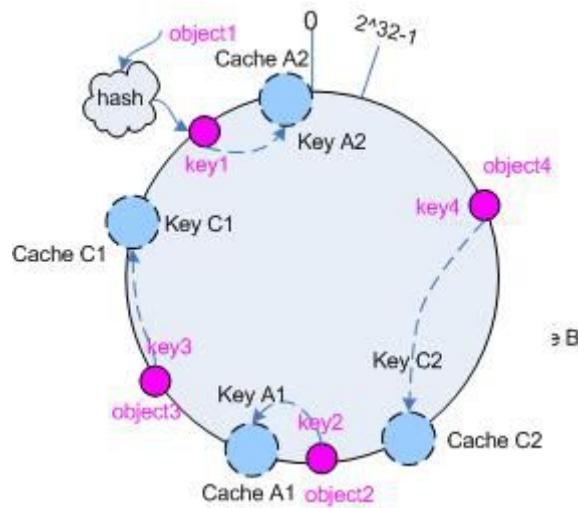


图 6 引入“虚拟节点”后的映射关系

此时，对象到“虚拟节点”的映射关系为：

objec1->cache A2 ; objec2->cache A1 ; objec3->cache C1 ; objec4->cache C2 ;

因此对象 object1 和 object2 都被映射到了 cache A 上，而 object3 和 object4 映射到了 cache C 上；平衡性有了很大提高。

引入“虚拟节点”后，映射关系就从 { 对象 -> 节点 } 转换到了 { 对象 -> 虚拟节点 }。查询物体所在 cache 时的映射关系如图 7 所示。

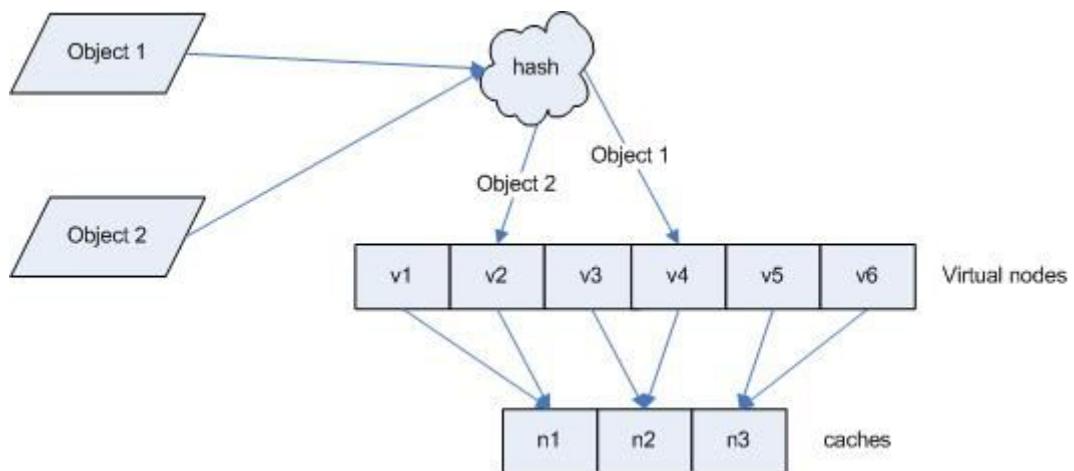


图 7 查询对象所在 cache

“虚拟节点”的 hash 计算可以采用对应节点的 IP 地址加数字后缀的方式。例如假设 cache A 的 IP 地址为 202.168.14.241。

引入“虚拟节点”前，计算 cache A 的 hash 值：

```
Hash("202.168.14.241");
```

引入“虚拟节点”后，计算“虚拟节点” cache A1 和 cache A2 的 hash 值：

```
Hash("202.168.14.241#1"); // cache A1
```

```
Hash("202.168.14.241#2"); // cache A2
```

## 后记

1. 以上部分代码思路有参考自此博客：<http://zhedahht.blog.163.com/blog/>。特此注明下。
2. 上文第五部分来源：<http://blog.csdn.net/sparkliang/article/details/5279393>;
3. 行文仓促，若有任何问题或漏洞，欢迎不吝指正或赐教。谢谢。转载，请注明出处。完。

## 第二十一~二十二章：出现次数超过一半的数字，最短摘要的生成

### 前言

咱们先来看两个问题：

第一个问题来自编程之美上，Tango 是微软亚洲研究院的一个试验项目，如图 1 所示。研究院的员工和实习生们都很喜欢在 Tango 上面交流灌水。传说，Tango 有一大“水王”，他不但喜欢发帖，还会回复其他 ID 发的每个帖子。坊间风闻该“水王”发帖数目超过了帖子总数的一半。如果你有一个当前论坛上所有帖子（包括回帖）的列表，其中帖子作者的 ID 也在表中，你能快速找出这个传说中的 Tango 水王吗？



图 1 Tango

第二个问题来自各位读者的手中，你我在百度或谷歌搜索框中敲入本博客名称的前 4 个字“结构之法”，便能在第一个选项看到本博客的链接，如下图 2 所示：

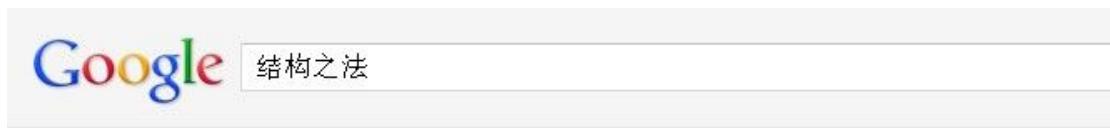


图 2 谷歌中搜索关键字“结构之法”

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

ok，看本文之前，你尚不知道怎么解决上述两个问题的话不要紧，本文即要阐述上述两个问题。若有任何问题，欢迎随时不吝指正。谢谢。

## 第二十一章、发帖水王及其扩展

### 第一节、74. 数组中超过出现次数超过一半的数字

题目：数组中有一个数字出现的次数超过了数组长度的一半，找出这个数字。

分析：编程之美上也有这道题，不过它变换了题目的表述形式，即是如本文前言所述的寻找发帖水王的问题。

ok，咱们来解决上述这道题，以微软面试 100 题第 74 题的阐述为准（本程序员编程艺术系列就是按照之前整理的微软 100 题一题一题展开而来的）。

一个数组中有很多数，现在我们要找出这个数组中那个超过出现次数一半的数字，怎么找呢？大凡当我们碰到某一个杂乱无序的东西时，我们人的内心本质期望是希望把它梳理成有序的。所以，我们得分两种情况来讨论，无序和有序：

1. 如果无序，那么我们是不是可以先把数组中所有这些数字先进行排序，至于选取什么排序方法则不在话下，最常用的快速排序  $O(N \log N)$  即可。排完序呢，直接遍

历。在遍历整个数组的同时统计每个数字的出现次数，然后把那个出现次数超过一半的数字直接输出，题目便解答完成了。总的时间复杂度为  $O(N \log N + N)$ 。

2. 但各位再想想，**如果是有序的数组呢**或者经过上述由无序的数组变成有序后的数组呢？是否在排完序  $O(N \log N)$  后，真的还需要再遍历一次整个数组么？我们知道，既然是数组的话，那么我们可以根据数组索引支持直接定向到某一个数。我们发现，一个数字在数组中的出现次数超过了一半，那么在已排好序的数组索引的  $N/2$  处（从零开始编号），就一定是这个数字。自此，我们只需要对整个数组排完序之后，然后直接输出数组中的第  $N/2$  处的数字即可，这个数字即是整个数组中出现次数超过一半的数字，总的时间复杂度由于少了最后一次整个数组的遍历，缩小到  $O(N \log N)$ 。
3. 然不论是上述思路一的  $O(N \log N + N)$ ，还是思路二的  $O(N \log N)$ ，时间复杂度并无本质性的改变。我们需要找到一种更为有效的思路或方法。既要缩小总的时间复杂度，那么就用查找时间复杂度为  $O(1)$ ，事先预处理时间复杂度为  $O(N)$  的**hash 表**。哈希表的键值（Key）为数组中的数字，值（Value）为该数字对应的次数。然后直接遍历整个 hash 表，找出每一个数字在对应的位置处出现的次数，输出那个出现次数超过一半的数字即可。
4. **Hash 表**需要  $O(N)$  的开销空间，且要设计 hash 函数，还有没有更好的办法呢？我们可以试着这么考虑，如果**每次删除两个不同的数**（不管是不是我们要查找的那个出现次数超过一半的数字），那么，在剩下的数中，我们要查找的数（出现次数超过一半）出现的次数仍然超过总数的一半。通过不断重复这个过程，不断排除掉其它的数，最终找到那个出现次数超过一半的数字。这个方法，免去了上述思路一、二的排序，也避免了思路三空间  $O(N)$  的开销，总得说来，时间复杂度只有  $O(N)$ ，空间复杂度为  $O(1)$ ，不失为最佳方法。

或许，你还没有明白上述思路 4 的意思，举个简单的例子吧，如数组  $a[5] = \{0, 1, 2, 1, 1\}$ ；

很显然，若我们要找出数组  $a$  中出现次数超过一半的数字，这个数字便是 1，若根据上述思路 4 所述的方法来查找，我们应该怎么做呢？通过一次性遍历整个数组，然后每次删除不相同的两个数字，过程如下简单表示：

**0 1 2 1 1 => 2 1 1 => 1**, 最终，1 即为所找。

但是如果是 **5, 5, 5, 5, 1**，还能运用上述思路么？额，别急，请看下文思路 5。

5. 咱们根据数组的特性进一步考虑@zhedahht：数组中有个数字出现的次数超过了数组长度的一半。也就是说，有个数字出现的次数比其他所有数字出现次数的和还要多。

因此我们可以考虑在遍历数组的时候保存两个值：一个是数组中的一个数字，一个是次数。当我们遍历到下一个数字的时候，如果下一个数字和我们之前保存的数字相同，则次数加 1。如果下一个数字和我们之前保存的数字不同，则次数减 1。如果次数为零，我们需要保存下一个数字，并把次数重新设为 1。

下面，举二个例子：

- 第一个例子，[5, 5, 5, 5, 1](#)：

不同的相消，相同的累积。遍历到第四个数字时，candidate 是 5, nTimes 是 4；遍历到第五个数字时，candidate 是 5, nTimes 是 3；nTimes 不为 0，那么 candidate 就是超过半数的。

- 第二个例子，[0, 1, 2, 1, 1](#)：

开始时，保存 candidate 是数字 0, ntimes 为 1，遍历到数字 1 后，与数字 0 不同，则 ntime 减 1 变为零；接下来，遍历到数字 2, 2 与 1 不同，candidate 保存数字 2，且 ntimes 重新设为 1；继续遍历到第 4 个数字 1 时，与 2 不同，ntimes 减一为零，同时 candidate 保存为 1；最终遍历到最后一个数字还是 1，与我们之前 candidate 保存的数字 1 相同，ntime 加一为 1。最后返回的是之前保存的 candidate 为 1。

```
//copyright@zhedahht
//July,updated,
//2011.04.16.
#include <iostream>
using namespace std;

//改自编程之美 2010
int Find(int* a, int N) //a代表数组, N代表数组长度
{
    int candidate;
    int nTimes, i;
    for(i = nTimes = 0; i < N; i++)
    {
        if(nTimes == 0)
        {
            candidate = a[i], nTimes = 1;
        }
        else
        {
            if(candidate == a[i])
                nTimes++;
            else
                nTimes--;
        }
    }
    return candidate; //返回出现次数超过一半的那个数字
}

int main()
{
    int a[5]={0,1,2,1,1};
    int* n=a;
    cout<<Find(a,5)<<endl;
    return 0;
}
```

针对上述程序，我再说详细点，**0, 1, 2, 1, 1**:

1. i=0, candidate=0, nTimes=1;
2. i=1, a[1]=0 != candidate, nTimes--, =0;
3. i=2, candidate=2, nTimes=1;
4. i=3, a[3] != candidate, nTimes--, =0;
5. i=4, candidate=1, nTimes=1;
6. 如果是 **0, 1, 2, 1, 1, 1** 的话，那么 i=5, a[5]=1=candidate, nTimes++, =2; .....

Ok，思路清楚了，完整的代码如下：

```
//改自编程之美 2010
Type Find(Type* a, int N) //a 代表数组, N 代表数组长度
{
    Type candidate;
    int nTimes, i;
    for(i = nTimes = 0; i < N; i++)
    {
        if(nTimes == 0)
        {
            candidate = a[i], nTimes = 1;
        }
        else
        {
            if(candidate == a[i])
                nTimes++;
            else
                nTimes--;
        }
    }
    return candidate;
}
```

或者：

```
//copyright@zhedahht
//July,updated,
//2011.04.16.
#include <iostream>
using namespace std;

bool g_Input = false;

int Num(int* numbers, unsigned int length)
```

```

{
    if(numbers == NULL && length == 0)
    {
        g_Input = true;
        return 0;
    }
    g_Input = false;

    int result = numbers[0];
    int times = 1;
    for(int i = 1; i < length; ++i)
    {
        if(numbers[i] == result)
            times++;
        else
            times--;
        if(times == 0)
        {
            result = numbers[i];
            times = 1;
        }
    }

    //检测输入是否有效。
    times = 0;
    for(i = 0; i < length; ++i)
    {
        if(numbers[i] == result)
            times++;
    }
    if(times * 2 <= length)
        //检测的标准是：如果数组中并不包含这么一个数字，那么输入将是无效的。
    {
        g_Input = true;
        result = 0;
    }
    return result;
}

int main()
{
    int a[10]={1,2,3,4,6,6,6,6,6};
    int* n=a;
    cout<<Num(a,9)<<endl;
}

```

```
    return 0;  
}
```

这段代码与上段代码本质上并无二致，不过有几个问题，还是需要我们注意：

1. 当输入无效性时，要处理。比如数组长度为 0。
2. 最后，上述代码加了一个判断，如果数组中并不包含这么一个数字，那么输入也是无效的。因此在函数结束前还加了一段代码来验证输入是不是有效的。

## 第二节、加强版水王：找出出现次数刚好是一半的数字

### 1、问题扩展@BrainDeveloper

我们知道，水王问题：有  $N$  个数，其中有一个数出现超过一半，要求在线性时间求出这个数。那么，我的问题是，加强版水王：有  $N$  个数，其中有一个数刚好出现一半次数，要求在线性时间内求出这个数。

因为，很明显，如果是刚好出现一半的话，如此例：[0, 1, 2, 1](#):

遍历到 0 时 `candidate` 为 0, `times` 为 1

遍历到 1 时 与 `candidate` 不同, `times` 减为 0

遍历到 2 时, `times` 为 0, 则 `candidate` 更新为 2, `times` 加 1

遍历到 1 时, 与 `candidate` 不同, 则 `times` 减为 0; 我们需要返回所保存 `candidate` (数字 2) 的下一个数字即数字 1。

所以，如果还运用上面的程序的话，那么只能返回我们要找的数字 1 的前一个数字，即遍历到 1 之前所保存的 `candidate2`，试问如何让程序能返回我们需要的数字 1 呢？望读者思考之：

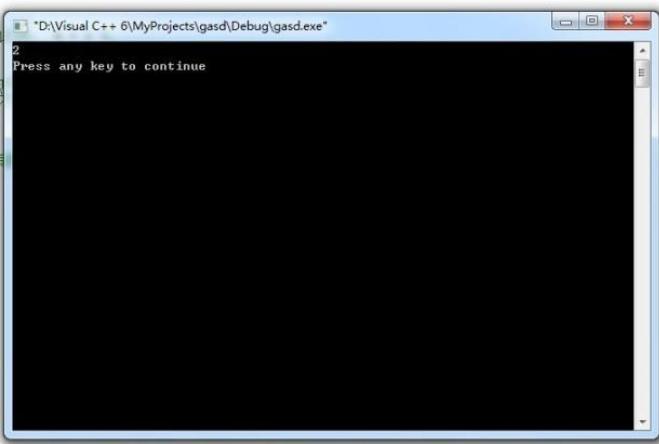
```

//copyright@zhedahht
//July, updated,
//2011.04.16.
#include <iostream>
using namespace std;

/*因为，很明显，如果是刚好出现一半的话，如
遍历到0时 candidate为0, times为1
遍历到1时，与candidate不同，times为0
遍历到2时， times为0，则candidate更新为1
遍历到1时，与candidate不同，则times减1
*/
//改自编程之美 2010
int Find(int* a, int N) //a代表数组，N代表数组的长度
{
    int candidate;
    int nTimes, i;
    for(i = nTimes = 0; i < N; i++)
    {
        if(nTimes == 0)
        {
            candidate = a[i], nTimes = 1;
        }
        else
        {
            if(candidate == a[i])
                nTimes++;
            else
                nTimes--;
        }
    }
    return candidate;
    //此处只能返回我们要找的数字1的前一个数字，即遍历到1之前所保存的candidate2，如何返回出现次数刚好是一半的那个数字1呢？
}

int main()
{
    int a[4]={0,1,2,1};
    int* p=a;
    cout<<Find(a,4)<<endl;
    return 0;
}

```



程序经过修改后，如下（未经严格测试，如有误恳请指正）：

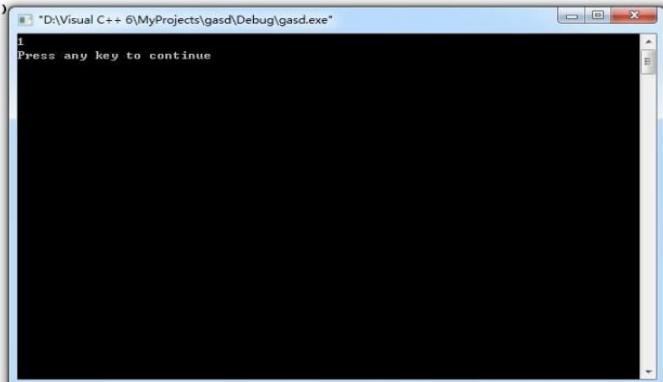
```

//copyright@2011 赤血红狼(&&July
//July, updated,
//2011.12.7日凌晨。
#include <iostream>
using namespace std;

//适用于发帖水王及其发帖水王的扩展问题
//即不论是出现次数刚好出现一半，还是超过一半的数字都能找到。
int Find(int* a, int N)
{
    int candidate;
    int nTimes, i;
    for(i = nTimes = 0; i < N; i++)
    {
        if(nTimes == 0)
        {
            candidate = a[i], nTimes = 1;
        }
        else
        {
            if(candidate == a[i])
                nTimes++;
            else
                nTimes--;
        }
    }
    int j;
    for(j=1;j<N-2;)
    {
        if(a[j]==a[j+2])
            j=j+2;
        else
            break;
    }
    if(j==N-1)
        return a[1];
    else
        return candidate;
}

int main()
{
    int a[4]={0,1,2,1};
    int* p=a;
    cout<<Find(a,4)<<endl;
    return 0;
}

```



## 2、bug 出现！

据本文读者 tenger\_lee 评论反应，上述程序有 bug，因为若数组序列是：1 0 2 1 2 1，则无法得到正确结果。现修正如下（只是程序复杂度高达为 O (N^2)，还待后续改进与优

化):

```
#include <iostream>
using namespace std;

int Find(int* a,int N)
{
    int candidate;
    int nTimes=0;
    int i=0;
    for (;i<N;i++)
    {
        if(nTimes==0)
        {
            int j;
            for(j=0;j<N-2;)
            {
                if(a[N-1]==a[j]||a[N-1]==a[j+1])
                    j=j+2;
                else
                    break;
            }
            if(j==N-2)
                candidate=a[N-1];
        }
        return candidate;
    }

    int main()
    {
        int a[6]={1,0,2,1,2,1};
```

### 3、持续改进

赤血红狐&&pointersky 用了两个变量来记录水王，最终完整代码如下（测试正确）：

```
#include<iostream>
using namespace std;

int Find(int* a, int N)
{
    int candidate1,candidate2;
    int nTimes1, nTimes2, i;

    for(i = nTimes1 = nTimes2 =0; i < N; i++)
    {
        if(nTimes1 == 0)
        {
            candidate1 = a[i], nTimes1 = 1;
        }
        else if(nTimes2 == 0 && candidate1 != a[i])
            //注意：这里的判断条件加上第二个变量是否等于第一个变量的判断
        {
            candidate2 = a[i], nTimes2 = 1;
```

```

    }
    else
    {
        if(candidate1 == a[i])
            nTimes1++;
        else if(candidate2 == a[i])
            nTimes2++;
        else
        {
            nTimes1--;
            nTimes2--;
        }
    }
    return nTimes1>nTimes2?candidate1:candidate2;
}

int main()
{
    int a[4]={0,1,2,1};
    cout<<Find(a,4)<<endl;
//    int a[6]={1,0,2,1,2,1};
//    cout<<Find(a,6)<<endl;
}

```

## 4、读者反馈

当然也可以如本文评论下第 16 楼 dalong7 所述：

易知总数必定是偶数，同时删除不同数字，最后剩余的两个数字必有其一为水王，只需简单判断一下即可，

代码如下（测试正确）：

```

int Find(int* a, int N) //a 代表数组, N 代表数组长度
{
    int candidate;
    int nTimes, i;
    for(i = nTimes = 0; i < N; i++)
    {
        if(nTimes == 0)
        {
            candidate = a[i], nTimes = 1;
        }
        else
        {

```

```

        if(candidate == a[i])
            nTimes++;
        else
            nTimes--;
    }

}

int cTimes = 0;
int candidate2 = a[N-1];
for(i = 0; i < N; i++)
{
    if(a[i] == candidate)
    {
        cTimes++;
    }
}

return cTimes == N/2 ? candidate : candidate2;
}

```

其实上述代码与本文评论下第 19 楼 xiaoyinghao999 所述思路基本一致，

关于加强版水王的题我有个想法可以扫描一遍数组就解决问题：

首先，水王占总数的一半，说明总数必为偶数；其次，最后一个元素或者是水王，或者不是水王，因此只要在扫描数组的时候每一个元素都与最后一个元素做比较，如果相等则最后一个元素的个数加 1，否则不处理。如果最后一个元素的个数为  $N/2$ , ( $N$  为数组元素个数) 则它就是水王，否则水王就是前面  $N-1$  个元素中选出的 `candidate`。

代码如下（暂未测试）：

```

int MoreThanHalf(int a[], int N)
{
    int sum1 = 0;//最后一个元素的个数
    int sum2 = 0;
    int candidate;
    int i;
    for(i=0;i<N-1;i++)//扫描前 N-1 个元素
    {
        if(a == a[N-1])//判断当前元素与最后一个是否相等
            sum1++;
        if(sum2 == 0)
        {
            candidate = a;
            sum2++;
        }
        else
        {

```

```

        if(a == candidate)
            sum2++;
        else
            sum2--;
    }

    if((sum1+1) == N/2)
        return a[N-1];
    else
        return candidate;
}

```

特别感谢诸位朋友们的指正，与贡献代码!July、二零一二年九月二十八日。

## 第二十二章、最短摘要的生成

**Alibaba 笔试题：**给定一段产品的英文描述，包含 M 个英文字母，每个英文单词以空格分隔，无其他标点符号；再给定 N 个英文单词关键字，请说明思路并编程实现方法

String extractSummary(String description, String[] key words)

目标是找出此产品描述中包含 N 个关键字（每个关键词至少出现一次）的长度最短的子串，作为**产品简介**输出。（不限编程语言）20 分。

这题是来自此篇文章十月百度，阿里巴巴，迅雷搜狗最新面试十一题中整理的阿里巴巴的笔试题，之前已经给出了这样一种思路，如下：

**@owen:** 扫描过程始终保持一个[`left,right`]的 range，初始化确保[`left,right`]的 range 里包含所有关键字则停止。然后每次迭代：

1. 试图右移动 `left`，停止条件为再移动将导致无法包含所有关键字。
2. 比较当前 range's length 和 best length，更新最优值。
3. 右移 `right`，停止条件为使任意一个关键字的计数+1。
4. 重复迭代。

在那篇文章中也提到了编程之美有最短摘要生成的问题，与此问题类似。下面，我将介绍这种方法。首先，咱们来看一个问题。读者可以在百度或谷歌中搜索本博客名称的前 4 个字，“结构之法”，便会在第一个搜索结果中看到如下图所示的搜索项：

### 结构之法 算法之道 - 博客频道 - CSDN.NET

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

算法之道blog之博主。时间：2010年10月-2011年6月。出处：<http://...>

[blog.csdn.net/v\\_JULY\\_v/2011-10-3](http://blog.csdn.net/v_JULY_v/2011-10-3) - 百度快照

上图中，那段大致介绍本博客结构之法算法之道的文字：“程序员面试、算法研究、编程艺术、红黑树 4 大经典原创系列集锦与总结 作者：July--结构之法算法之道 blog 之博主。时间：2010 年 10 月-2011 年 6 月。出处：<http://...>”这段介于搜索关键词与最底下的 URL 便是我们所称之为的摘要。那么，这段摘要是怎么产生的呢？可以对问题进行如下的简化。

1. 假设给定的已经是经过网页分词之后的结果，词语序列数组为  $W$ 。其中  $W[0], W[1], \dots, W[N]$  为一些已经分好的词语。
2. 假设用户输入的搜索关键词为数组  $Q$ 。其中  $Q[0], Q[1], \dots, Q[m]$  为所有输入的搜索关键词。

这样，生成的最短摘要实际上就是一串相互联系的分词序列。比如从  $W[i]$  到  $W[j]$ ，其中， $0 < i < j \leq N$ 。例如上图所示的摘要“程序员面试、算法研究、编程艺术、红黑树 4 大经典原创集锦与总结 作者：July--结构之法算法之道 blog 之博主.....”中包含了关键字——“**结构之法**”。

那么，我们该怎么做呢？

#### 思路一：

在分析问题之前，先通过一个实际的例子来探讨。比如在本博客第一篇置顶文章的开头，有这么一段话：

“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--结构之法算法之道 blog 之博主。

时间：2010 年 10 月-2011 年 6 月。

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

声明：版权所有，侵犯必究。”

那么，我们可以猜想一下可能的分词结果：

“[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/[道](#)/blog/之/[博](#)主/....”（网页的分词效果  $W$  数组）

这也就是我们期望的 W 数组序列。

之前的 Q 数组序列为：“[结构之法](#)”（用户输入的关键字 Q 数组）

再看下下面这个 W-Q 序列：

w0,w1,w2,w3,[q0](#),w4,w5,[q1](#),w6,w7,w8,[q0](#),w9,[q1](#)

上述序列上面的是 W 数组（经过网页分词之后的结果），W[0], W[1], …, W[N]为一些已经分好的词语，

上述序列下面的是 Q 数组（用户输入的搜索关键词）。其中 Q[0], Q[1], …, Q[m]为所有输入的搜索关键词。

ok，如果你不甚明白，我说的通俗点：如上 W-Q 序列中，我们可以把，[q0](#), w4, w5, [q1](#)作为摘要，[q0](#), w9, [q1](#)的也可以作为摘要，同样都包括了所有的关键词 q0, q1，那么选取哪个是最短摘要呢？答案很明显，后一个更短，选取 [q0](#), w9, [q1](#)的作为最短摘要，这便是[最短摘要的生成](#)。

我们可以进一步可以想象，如下：

从用户的角度看：当我们在百度的搜索框中输入“结构之法”4个字时，搜索引擎将在索引数据库中（关于搜索引擎原理的大致介绍，可参考本博客中这篇文章：[搜索引擎技术之概要预览](#)）查找和匹配这4个字的网页，最终第一个找到了本博客的置顶的第一篇文章：[\[置顶\]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](#)；

从搜索引擎的角度看：搜索引擎经过把上述网页分词后，便得到了上述的分词效果，然后在这些分词中查找“结构之法”4个关键字，但这4个关键字不一定只会出现一遍，它可能会在这篇文章中出现多次，就如上面的 W-Q 序列一般。咱们可以假想出下面的结果（[结构之法](#)便出现了两次）：

“[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/道/blog/之/博主/././.转/载/请/注明/出处/：/结构/之/法/算法/之/道/CSDN/博客/././.”](#)

由此，我们可以得出解决此问题的思路，如下：

1. 从 W 数组的第一个位置开始查找出一段包含所有关键词数组 Q 的序列（第一个位置“开始：[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/道/blog/之/博主/././.转/载/请/注明/出处/：/结构/之/法/算法/之/道/CSDN/博客/././.”](#)”）

- 集锦/与/总结//作者/: /July/-/结构/之/法/查找包含关键字“**结构之法**”所有关键词的序列)。计算当前的最短长度，并更新 Seq 数组。
2. 对目标数组 W 进行遍历，从第二个位置开始，重新查找包含所有关键词数组 Q 的序列 (第二个位置“处开始：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/查找包含关键字“**结构之法**”所有关键词的序列)，同样计算出其最短长度，以及更新包含所有关键词的序列 Seq，然后求出最短距离。
  3. 依次操作下去，一直到遍历至目标数组 W 的最后一个位置为止。

最终，通过比较，咱们确定如下分词序列作为最短摘要，即搜索引擎给出的分词效果：

”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--结构之法算法之道 blog 之博主。时间：2010 年 10 月-2011 年 6 月。出处：<http://...>“

那么，这个算法的时间复杂度如何呢？

要遍历所有其他的关键词 (M)，对于每个关键词，要遍历整个网页的词 (N)，而每个关键词在整个网页中的每一次出现，要遍历所有的 Seq，以更新这个关键词与所有其他关键词的最小距离。所以算法复杂度为：O (N^2 \* M)。

## 思路二：

我们试着降低此问题的复杂度。因为上述思路一再进行查找的时候，总是重复地循环，效率不高。那么怎么简化呢？先来看看这些序列：

w0,w1,w2,w3,q0,w4,w5,q1,w6,w7,w8,q0,w9,q1

问题在于，如何一次把所有的关键词都扫描到，并且不遗漏。扫描肯定是无法避免的，但是如何把两次扫描的结果联系起来呢？这是一个值得考虑的问题。

沿用前面的扫描方法，再来看看。第一次扫描的时候，假设需要包含所有的关键词，从第一个位置 w0 处将扫描到 w6 处：

w0,w1,w2,w3,q0,w4,w5,q1,w6,w7,w8,q0,w9,q1

那么，下次扫描应该怎么办呢？先把第一个被扫描的位置挪到 q0 处。

w0,w1,w2,w3,q0,w4,w5,q1,w6,w7,w8,q0,w9,q1

然后把第一个被扫描的位置继续往后面移动一格，这样包含的序列中将减少了关键词 q0。那么，我们便可以把第二个扫描位置往后移，这样就可以找到下一个包含所有关键词的序列。即从 w4 扫描到 w9 处，便包含了 q1, q0:

w0,w1,w2,w3,q0,w4,w5,q1,w6,w7,w8,q0,w9,q1

这样，问题就和第一次扫描时碰到的情况一样了。依次扫描下去，在 w 中找出所有包含 q 的序列，并且找出其中的最小值，就可得到最终的结果。编程之美上给出了如下参考代码：

//July、updated, 2011.10.21。

```
int nTargetLen = N + 1;           // 设置目标长度为总长度+1
int pBegin = 0;                  // 初始指针
int pEnd = 0;                    // 结束指针
int nLen = N;                   // 目标数组的长度为 N
int nAbstractBegin = 0;          // 目标摘要的起始地址
int nAbstractEnd = 0;            // 目标摘要的结束地址

while(true)
{
    // 假设未包含所有的关键词，并且后面的指针没有越界，往后移动指针
    while(!isAllExisted() && pEnd < nLen)
    {
        pEnd++;
    }

    // 假设找到一段包含所有关键词信息的字符串
    while(isAllExisted())
    {
        if(pEnd - pBegin < nTargetLen)
        {
            nTargetLen = pEnd - pBegin;
            nAbstractBegin = pBegin;
            nAbstractEnd = pEnd - 1;
        }
        pBegin++;
    }
    if(pEnd >= N)
        Break;
}
```

小结：上述思路二相比于思路一，很明显提高了不小效率。我们在匹配的过程中利用了可以省去其中某些死板的步骤，这让我想到了 KMP 算法的匹配过程。同样是经过观察，比较，

最后总结归纳出的高效算法。我想，一定还有更好的办法，只是我们目前还没有看到，想到，待我们去发现，创造。

### 思路三：

以下是读者 **jiaotao1983** 回复于本文评论下的反馈，非常感谢。

关于最短摘要的生成，我觉得 July 的处理有些简单，我以 July 的想法为基础，提出了自己的一些想法，这个问题分以下几步解决：

1，将传入的 key words[] 生成哈希表，便于以后的字符串比较。结构为 KeyHash，如下：

```
struct KeyHash
{
    int cnt;
    char key[];
    int hash;
}
```

结构体中的 hash 代表了关键字的哈希值，key 代表了关键字，cnt 代表了在当前的扫描过程中，扫描到的该关键字的个数。

当然，作为哈希表结构，该结构体中还会有其它值，这里不赘述。

初始状态下，所有哈希结构的 cnt 字段为 0。

2，建立一个 KeyWord 结构，结构体如下：

```
struct KeyWord
{
    int start;
    KeyHash* key;
    KeyWord* next;
    KeyWord* prev;
}
```

key 字段指向了建立的一个 KeyWord 代表了当前扫描到的一个关键字，扫描到的多个关键字组成一个双向链表。

start 字段指向了关键字在文章中的起始位置。

3，建立几个全局变量：

KeyWord\* head，指向了双向链表的头，初始为 NULL。

KeyWord\* tail, 指向了双向链表的尾, 初始为 NULL。  
int minLen, 当前扫描到的最短的摘要的长度, 初始为 0。  
int minStartPos, 当前扫描到的最短摘要的起始位置。  
int needKeyCnt, 还需要几个关键字才能够包括全部的关键字, 初始为关键字的个数。

**4,** 开始对文章进行扫描。每扫描到一个关键字时, 就建立一个 KeyWord 的结构并且将其连入到扫描到的双向链表中, 更新 head 和 tail 结构, 同时将对应的 KeyHash 结构中的 cnt 加 1, 表示扫描到了关键字。如果 cnt 由 0 变成了 1, 表示扫描到一个新的关键字, 因此 needKeyCnt 减 1。

**5,** 当 needKeyCnt 变成 0 时, 表示扫描到了全部的关键字了。此时要进行一个操作: 链表头优化。

链表头指向的 word 是摘要的起始点, 可是如果对应的 KeyHash 结构中的 cnt 大于 1, 表示扫描到的摘要中还有该关键字, 因此可以跳过该关键字。因此, 此时将链表头更新为下一个关键字, 同时, 将对应的 KeyHash 中的结构中的 cnt 减 1, 重复这样的检查, 直至某个链表头对应的 KeyHash 结构中的 cnt 为 1, 此时该结构不能够少了。

**6,** 如果找到更短的 minLength, 则更新 minLength 和 minStartPos。

**7,** 开始新一轮的搜索。此时摘除链表的第一个节点, 将 needKeyCnt 加 1, 将下一个节点作为链表头, 同样的开始链表头优化措施。搜索从上一次的搜索结束处开始, 不用回溯。就是所, 搜索在整个算法的过程中是一直沿着文章向下的, 不会回溯。, 直至文章搜索完毕。

这样的算法的复杂度初步估计是  $O(M+N)$ 。

**8,** 另外, 我觉得该问题不具备实际意义, 要具备实际意义, 摘要应该包含完整的句子, 所以摘要的起始和结束点应该以句号作为分隔。

这里, 新建立一个结构: Sentence, 结构体如下:

```
struct Sentence
{
    int start; //句子的起始位置
    int end; //句子的结束位置
    KeyWord* startKey; //句子包含的起始关键字
    KeyWord* endKey; //句子包含的结束关键字
    Sentence* prev; //下一个句子结构
```

```
Sentence* next; //前一个句子结构  
}
```

扫描到的多个句子结构组成一个链表。增加两个全局变量，分别指向了 Sentence 链表的头和尾。

扫描时，建立关键字链表时，也要建立 Sentence 链表。当扫描到包含了所有的关键字时，必须要扫描到一个完整句子的结束。开始做 Sentence 头节点优化。做法是：查看 Sentence 结构中的全部 key 结构，如果全部的 key 对应的 KeyHash 结构的 cnt 属性全部大于 1，表明该句子是多余的，去掉它，去掉它的时候更新对应的 HashKey 结构的关键字，因为减去了很多的关键字。然后对下一个 Sentence 结构做同样的操作，直至某个 Sentence 结构是必不可少的，就是说它包含了当前的摘要中只出现过一次的关键字！

扫描到了一个摘要后，在开始新的扫描。更新 Sentence 链表的头结点为下一个节点，同时更新对应的 KeyHash 结构中的 cnt 关键字，当某个 cnt 变成 0 时，就递增 needKeycnt 变量。再次扫描时仍然是从当前的结束位置开始扫描。

初步估计时间也是  $O(M+N)$ 。

ok，留下一个编程之美一书上的扩展问题：当搜索一索一个词语后，有许多的相似页面出现，如何判断两个页面相似，从而在搜索结果中隐去这类结果？

本文参考：

1. 编程之美第二章第 2.3 节寻找发帖水王；
2. 编程之美第三章第 3.5 节最短摘要的生成；
3. <http://zhedahht.blog.163.com/blog/static/25411174201085114733349/>。

## 后记

编程艺术系列从今年 4 月开始创作，已写了二十二章。此系列最初是我一个人写，后来我的一些朋友加入进来了，便成立了程序员编程艺术室，是我和一些朋友们一起写了，但到如今一直在坚持的又只剩下自己了。近些天，常常发呆胡乱思考一些东西，如个人写博刚过一年，有时候也看得一些有关互联网创业的文章，便有了下面写博 VS 创业这个话题：

1. 读者第一（用户至上）；
2. 站在读者角度和思维方式阐述问题，文不易懂死不休（重视用户体验，用户不喜欢不会用的产品便是废品）；

3. 只写和创作读者最需要的文章，东西（别人不需要，便没有市场，没有市场，一切免谈）；
4. 写博贵在坚持（创业贵在坚持）。

编程艺术系列一如之前早已说过，“因为编程艺术系列最后可能要写到第六十章”（语出自：[程序员编程艺术第一~十章集锦与总结--面试、算法、编程](#)）。期待，编程艺术室的朋友能早日继续加入共同创作。以诸君为傲。ok，若有任何问题，欢迎随时不吝指正。转载请注明出处。完。July、2011.10。

## 第二十三、四章：杨氏矩阵查找，倒排索引关键词 Hash 不重复编码实践

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）：

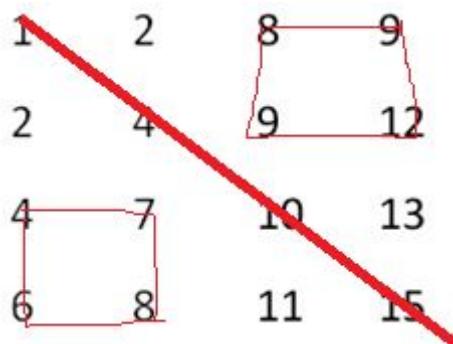
在一个  $m$  行  $n$  列二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

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。

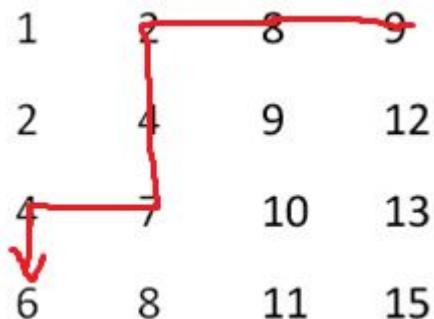
1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

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、定位法，时间复杂度  $O(m+n)$ 。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：



上述方法二的关键代码+程序运行如下图所示：

The screenshot shows a Microsoft Visual Studio interface with the following details:

- Title Bar:** Young (正在运行) - Microsoft Visual Studio
- Menu Bar:** 文件(F) 编辑(E) 视图(V) VAssistX 项目(P) 生成(B) 调试(D) 团队(M) 数据(A) 工具(T) 体系结构(C) 测试(S) 分析(N) 窗口(W)
- Toolbars:** Standard, Debug, Win32
- Code Editor:** Young.cpp\* (Young)
- Code Content:**

```
#define ROW 4
#define COL 4

bool Young(int array[][COL], int search)
{
    int i = 0, j = COL-1;
    int var = array[i][j];
    while (true) {
        if (var == search)
            return true;
        else if (var < search && i < ROW - 1)
            var = array[++i][j];
        else if (var > search && j > 0)
            var = array[i][--j];
        else
            return false;
    }
}

int main()
{
    int array[ROW][COL] = {{1,2,8,9},{2,4,9,12},{4,7,10,13},{6,8,11,15}};

    for (int search = 1; search <= 15; search++) {
        cout << search;
        if (Young(array, search))
            cout << ":存在";
        else
            cout << ":不存在";
    }
}
```
- Output Window:** 显示了16次调用Young函数的结果，输出为15行文本，每行包含一个搜索值和一个“存在”或“不存在”的判断结果。
- Status Bar:** 100 %
- Bottom Navigation:** 自动窗口, 调用堆栈

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。

## 第二十四章、经典算法十一 Hash 表算法（续）、倒排索引关键词不重复 Hash 编码

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步骤：

- 首先，要提取倒排索引内词典文件中的关键词；

- 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。

## 24.1、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，若是有新的文档假如，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。若是有文档删除，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。

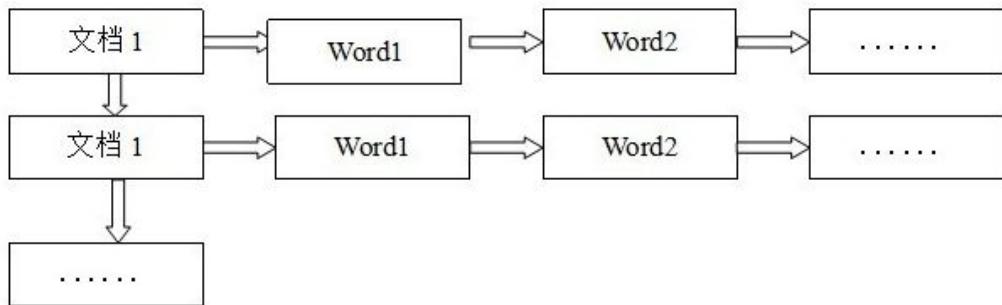


图 1 正排表结构图

## 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：

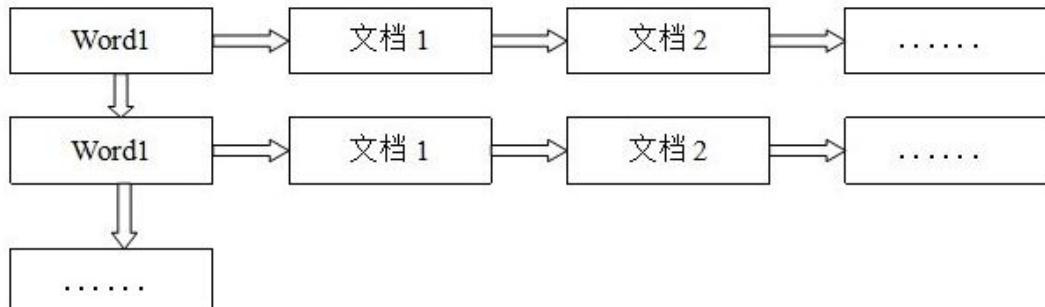


图 2 倒排表结构图

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。

## 24.2、倒排索引中提取关键词

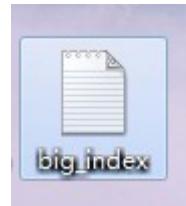
倒排索引是搜索引擎之基石。建成了倒排索引后，用户要查找某个 query，如在搜索框输入某个关键词：“结构之法”后，搜索引擎不会再次使用爬虫又一个一个去抓取每一个网页，从上到下扫描网页，看这个网页有没有出现这个关键词，而是会在它预生成的倒排索引文件中查找和匹配包含这个关键词“结构之法”的所有网页。找到了之后，再按相关性度排序，最终把排序后的结果显示给用户。



Google 搜索 结构之法

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  
图片 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  
地图 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 ...



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，--Firelf--，-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）：--Firelf--，-11，-Winter-，..，007，007：天降杀机，02Chan..。

--Firelf-- (关键词) 8 (出现次数)

		0	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190				
1	-11	1	8																						
2	20111108	5	T0011111600004693	240	00	240	36#d5f1#6f231770#d1#e8***#376	85368	1#####T0011111600004694	240	00	240	210#e46f73#b4d#e5#b1#f1#d176	84539	1#####										
3	20111108	5	T0011111600004693	240	00	240	36#7627#e4#9e#b3#1#92#1743#b5#da	131221	1#####T0011111500007324	240	00	240	2015#5#5#5#2#b1#f1#33#e#80#e#074#974	113059	1#####T0011111500005758	240	00								
4	20111114	1	T0011111400004894	240	00	240	6#8#7#1#5#5#5#e#5#b1#5#7#24#e#0#5#b#1	123028	1#####																
5	20111112	1	T0011111200001071	240	00	240	4#1#b#2#3#4#f#3#7#0#d#7#1#5#b#1#e#7	84124	1#####																
6	20111108	9	T0011111000011295	00	240	9#9#0#8#7#1#5#d#1#e#0#9#7#1#9#5#d#4#9#	160409	1#####T0011111000010700	240	230#7#e#9#6#1#e#d#1#b#3#7#d#f#4#8#	195151	1#####T0011111000010701	240	00	240										
7	20111102	4	T0011102000007030	48	00	240	c#4#4#7#5#4#d#7#5#9#e#d#5#2#d#3#3#2#0	140003	1#####T0011102000036#72	48	00	240	4#7#7#6#9#e#5#7#o#3#4#7#d#e#0#b#b#4#e#e#3#1#1	100006	1#####T001111020000823	48	00	240							
8	20111101	2	T001110100010412	48	00	240	f#d#2#e#0#8#2#5#2#d#2#d#7#7#e#7#2#0#4#5#e#b#1#	163759	1#####T001110100010411	48	00	240	e#5#7#e#d#9#e#3#0#0#e#f#5#b#b#f#d#3#8#1#2	163758	1#####										
9	20111105	3	PW011111500017183	156	00	156	e#9#e#7#9#0#9#6#7#9#1#4#3#5#7#d#4#5#7#9#e#5#7#4	204455	1######T0011111500017162	156	00	156	e#9#e#7#9#0#9#6#7#9#1#4#3#5#7#d#4#5#7#9#e#5#7#4	204454	1######T0011111500017161	156	00	156							
10	20111109	2	PW011110900013663	156	00	156	b#5#b#0#b#e#8#0#4#3#6#e#3#6#e#3#7#f#5#b#e#f#e#0#3#0#	205723	1######P#011110900008419	120	00	156	b#5#b#0#b#e#8#0#4#3#6#e#3#6#e#3#7#f#5#b#e#f#e#0#3#0#	122538	1#####										
11	20111109	2	PW011110900013663	156	00	156	b#5#b#0#b#e#8#0#4#3#6#e#3#6#e#3#7#f#5#b#e#f#e#0#3#0#	205723	1######P#011110900008419	120	00	156	b#5#b#0#b#e#8#0#4#3#6#e#3#6#e#3#7#f#5#b#e#f#e#0#3#0#	122538	1#####										
12	“Kinter”																								
13	20111109	2	PW011110900013663	156	00	156	b#5#b#0#b#e#8#0#4#3#6#e#3#6#e#3#7#f#5#b#e#f#e#0#3#0#	205723	1######P#011110900008419	120	00	156	b#5#b#0#b#e#8#0#4#3#6#e#3#6#e#3#7#f#5#b#e#f#e#0#3#0#	122538	1#####										
14	14																								
15	20111108	1	T#0111116000017105	317	00	317	e#9#4#5#3#d#1#b#2#4#e#0#b#6#0#9#3#7#1#0#7#	143222	1#####																
16	20111105	1	T#0111116000017105	317	00	317	e#6#b#7#5#3#6#4#1#7#2#0#7#0#4#9#e#3#7#1#3#7#	93353	1#####																
17	20111114	2	T#011111400025995	317	00	317	e#1#4#2#7#4#e#4#5#7#5#5#9#e#9#e#1#9#0#2#3#8#	224200	1######T#011111400025434	317	00	317	9#2#7#d#4#e#6#4#3#b#e#1#7#t#b#1#9#4#9#7#0#4#0#4#2#	183116	1#####										
18	20111108	2	T#011111300002493	317	00	317	2#5#7#0#e#5#1#2#1#4#1#0#1#b#3#3#e#8#6#5#5#8#3	162256	1######T#011111300001507	317	00	317	6#e#2#f#9#0#4#9#e#5#9#f#0#b#e#5#8#0#b#f#7#3#	114656	1#####										
19	20111110	1	T#011111000000882	0	00	0	317	e#5#2#9#e#2#9#e#4#4#5#e#5#0#5#7#1#2#e#2#4#0#e#e#1#	112357	1#####															
20	20111109	1	T#011111000000882	0	00	0	317	e#5#2#9#e#2#9#e#4#4#5#e#5#0#5#7#1#2#e#2#4#0#e#e#1#	112357	1#####															
21	20111107	1	T#011110800008495	274	00	317	9#5#1#9#e#9#5#0#4#8#e#2#2#3#d#7#e#2#e#1#5	223600	1######T#011110800008496	274	00	317	8#5#f#9#8#4#1#3#d#8#e#8#0#6#4#6#7#8#2#9#3#f#1#9#0#9#	92700	1#####										
22	20111108	2	T#011110800008495	274	00	317	9#5#1#9#e#9#5#0#4#8#e#2#2#3#d#7#e#2#e#1#5	223600	1######T#011110800008495	274	00	317	8#5#f#9#8#4#1#3#d#8#e#8#0#6#4#6#7#8#2#9#3#f#1#9#0#9#	92700	1#####										
23	20111108	2	T#011110800008497	274	00	317	8#e#6#f#5#4#e#5#e#5#9#3#2#9#d#1#4#e#8#9#1#7#3#	175400	1#####																
24	20111108	2	T#011110800008497	274	00	317	8#e#6#f#5#4#e#5#e#5#9#3#2#9#d#1#4#e#8#9#1#7#3#	175400	1#####																
25	20111101	3	T#011110800008497	274	00	317	8#e#6#f#5#4#e#5#e#5#9#3#2#9#d#1#4#e#8#9#1#7#3#	175400	1######T#011110800008500	274	00	317	5#e#1#e#1#7#d#e#3#e#7#0#2#1#0#d#3#8#1#2#5#9#8#1	143200	1######T#011110800008501	274	00	317							
26	20111108	2	T#011110800008502	274	00	317	5#4#4#8#2#0#4#4#5#4#8#9#1#e#7#e#2#7#b#1#	171000	1#####																
27	20111104	1	T#011110800008503	274	00	317	e#2#4#2#b#6#4#0#d#1#b#1#9#d#2#e#0#3#4#7#e#ca#	058000	1#####																
28	20111102	1	T#011110800008504	274	00	317	9#1#9#0#6#7#3#8#6#2#7#0#1#4#3#5#e#8#3#6#1#	105200	1#####																
29	20111127	1	T#111112700002087	369	B1	172	8#0#9#4#9#6#0#1#2#2#3#9#3#1#f#3#3#e#1#4#3#7#	43100	1#####																
30	007: 天降手机	1	T#111112700002089	369	B1	2#7#2#5#d#c#e#4#2#4#2#5#4#9#e#1#f#2#6#8#	91342	1#####																	
31	20111127	1	T#111112700002089	369	B1	2#7#2#5#d#c#e#4#2#4#2#5#4#9#e#1#f#2#6#8#	91342	1#####																	
32	20111128	1	T#111112600000933	282	B1	2#8#2#0#9#4#4#4#4#4#4#5#f#5#b#e#f#e#0#3#0#	142910	1#####																	
33	20111127	1	T#111112600000933	282	B1	2#8#2#0#9#4#4#4#4#4#5#f#5#b#e#f#e#0#3#0#	142910	1#####																	
34	20111109	2	T#001111600020943	225	00	225	d#3#a#e#6#1#b#0#f#4#4#b#4#4#4#2#2#0#0#3#f#	173011	1######T#001111600020944	225	00	225	a#7#1#6#7#e#6#d#9#2#1#6#5#2#8#2#d#1#5#e#3#2#0#e#	162902	1######T#001111600015428	225	00	225							
35	20111105	7	T#001111500013121	225	00	225	a#5#1#7#9#e#9#7#0#3#8#9#0#9#3#e#3#b#e#	150157	1######T#001111500013122	225	00	225	a#f#5#4#e#5#5#5#0#3#f#1#7#9#9#8#f#1#3#4#b#2#7	140340	1######T#001111500013124	225	00	225							
36	20111114	2	T#001111500008407	225	00	225	e#5#5#b#e#7#e#2#9#5#a#w#5#b#6#0#f#7#3#1#2#2#7	233516	1######T#001111500008408	225	00	225	e#9#8#E#2#3#0#9#4#E#d#A#3#5#0#5#2#4#d#1#5#	192057	1#####										
37	20111111	2	T#001111100012702	225	00	225	a#5#0#2#1#d#8#3#3#f#1#2#8#4#d#2#e#2#9#4#4#5	191540	1######T#001111100003010	225	00	225	1#4#8#e#7#A#b#1#d#1#e#9#5#7#3#6#4#3#2#e#e#C#	80527	1#####										
38	20111109	2	T#001110900013032	108	00	225	2#4#9#e#2#2#2#5#3#9#d#1#4#0#d#8#	172640	1######T#00111090004817	108	00	225	4#2#9#e#1#8#b#6#d#4#f#4#5#d#7#2#5#1#	100047	1#####										

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。读者是否有更好地办法？欢迎随时交流。

### 24.3、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
....	
	

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）。

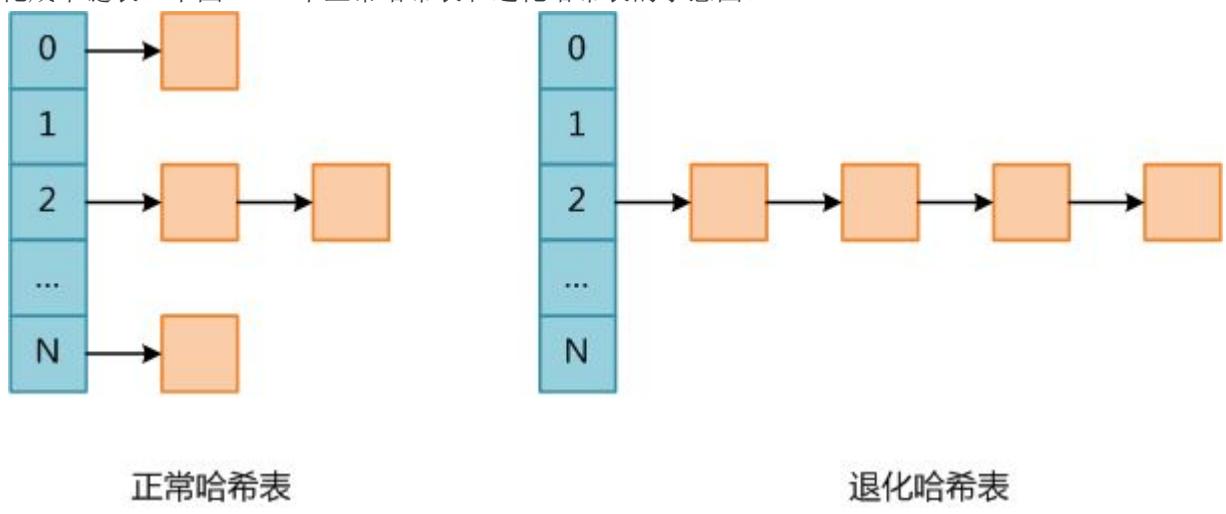
理想情况下哈希表插入和查找操作的时间复杂度均为  $O(1)$ ，任何一个数据项可以在一个与哈希表长度无关的时间内计算出一个哈希值（**key**），然后在常量时间内定位到一个桶（术语 **bucket**，表示哈希表中的一个位置）。当然这是理想情况下，因为任何哈希表的长度都是有限的，所以一定存在不同的数据项具有相同哈希值的情况，此时不同数据项被定为到同一个桶，称为碰撞（**collision**）。

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，

1. 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；
2. 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。（上述五段文字引自：<http://www.codinglabs.org/html/hash-collisions-attack-on-php.html>）。

## 24.4、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，但要是有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]

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

```

//函数 HashString 以下函数计算 lpszFileName 字符串的 hash 值，其中 dwHashType 为 hash 的类型，
unsigned long HashString(const char *lpszkeyName, unsigned long dwHashType )
{
    unsigned char *key = (unsigned char *)lpszkeyName;
    unsigned long seed1 = 0x7FED7FED;
    unsigned long seed2 = 0xFFFFFFFF;
    int ch;

    while( *key != 0 )
    {
        ch = *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 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 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.

```

//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.
int GetHashTablePos( char *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 HashString，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;      //返回找到的 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，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：

```

//函数 GetHashTablePos 中，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。

## 24.5、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：

```

//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]
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 HashString 以下函数计算 lpszFileName 字符串的 hash 值, 其中 dwHashType 为 hash 的类
型,
unsigned long HashString(const char *lpszkeyName, unsigned long dwHashType )
{
    unsigned char *key = (unsigned char *)lpszkeyName;
    unsigned long seed1 = 0x7FED7FED;
    unsigned long seed2 = 0xEEEEEEEE;
    int ch;

    while( *key != 0 )
    {
        ch = *key++;
        seed1 = cryptTable[(dwHashType<<8) + ch] ^ (seed1 + seed2);
        seed2 = ch + seed1 + seed2 + (seed2<<5) + 3;
    }
    return seed1;
}

///////////////////////////////
//function: 哈希词典 编码
//parameter:
//author: lei.zhou
//time: 2011-12-14
/////////////////////////////

```

```

MPQHASHTABLE TestHashTable[nTableSize];
int TestHashCTable[nTableSize];
int TestHashDTable[nTableSize];
key_list test_data[nTableSize];

//直接调用上面的 hashstring, nHashPos 就是对应的 HASH 值。
int insert_string(const char *string_in)
{
    const int HASH_OFFSET = 0, HASH_C = 1, HASH_D = 2;
    unsigned int nHash = HashString(string_in, HASH_OFFSET);
    unsigned int nHashC = HashString(string_in, HASH_C);
    unsigned int nHashD = HashString(string_in, HASH_D);
    unsigned int nHashStart = nHash % nTableSize;
    unsigned int nHashPos = nHashStart;
    int ln, ires = 0;

    while (TestHashTable[nHashPos].bExists)
    {
        //      if (TestHashCTable[nHashPos] == (int) nHashC && TestHashDTable[nHashPos] ==
        //          (int) nHashD)
        //          break;
        //      //...
        //      else
        //          //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
        nHashPos = (nHashPos + 1) % nTableSize;

        if (nHashPos == nHashStart)
            break;
    }

    ln = strlen(string_in);
    if (!TestHashTable[nHashPos].bExists && (ln < nMaxStrLen))
    {
        TestHashCTable[nHashPos] = nHashC;
        TestHashDTable[nHashPos] = nHashD;

        test_data[nHashPos] = (KEYNODE *) malloc (sizeof(KEYNODE) * 1);
        if(test_data[nHashPos] == NULL)
        {
            printf("10000 EMS ERROR !!!!\n");
            return 0;
        }
    }
}

```

```

    test_data[nHashPos]->pkey = (char *)malloc(ln+1);
    if(test_data[nHashPos]->pkey == NULL)
    {
        printf("10000 EMS ERROR !!!!\n");
        return 0;
    }

    memset(test_data[nHashPos]->pkey, 0, ln+1);
    strncpy(test_data[nHashPos]->pkey, string_in, ln);
    *((test_data[nHashPos]->pkey)+ln) = 0;
    test_data[nHashPos]->weight = nHashPos;

    TestHashTable[nHashPos].bExists = 1;
}
else
{
    if(TestHashTable[nHashPos].bExists)
        printf("30000 in the hash table %s !!!\n", string_in);
    else
        printf("90000 strkey error !!!\n");
}
return nHashPos;
}

```

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）：

```

void bigIndex_hash(const char *docpath, const char *hashpath)
{
    FILE *fr, *fw;
    int len;
    char *pbuf, *p;
    char dockey[TERM_MAX LENG];

    if(docpath == NULL || *docpath == '\0')
        return;

    if(hashpath == NULL || *hashpath == '\0')
        return;

    fr = fopen(docpath, "rb"); //读取文件 docpath
    fw = fopen(hashpath, "wb");
    if(fr == NULL || fw == NULL)
    {

```

```

        printf("open read or write file error!\n");
        return;
    }

    pbuf = (char*)malloc(BUFF_MAX LENG);
    if(pbuf == NULL)
    {
        fclose(fr);
        return ;
    }

    memset(pbuf, 0, BUFF_MAX LENG);

    while(fgets(pbuf, BUFF_MAX LENG, fr))
    {
        len = GetRealString(pbuf);
        if(len <= 1)
            continue;
        p = strstr(pbuf, "#####");
        if(p != NULL)
            continue;

        p = strstr(pbuf, " ");
        if (p == NULL)
        {
            printf("file contents error!");
        }

        len = p - pbuf;
        dockey[0] = 0;
        strncpy(dockey, pbuf, len);

        dockey[len] = 0;

        int num = insert_string(dockey);

        dockey[len] = ' ';
        dockey[len+1] = '\0';
        char str[20];
        itoa(num, str, 10);

        strcat(dockey, str);
        dockey[len+strlen(str)+1] = '\0';
        fprintf (fw, "%s\n", dockey);
    }
}

```

```
    }
    free(pbuf);
    fclose(fr);
    fclose(fw);
}
```

主函数已经很简单了，如下：

```
int main()
{
    prepareCryptTable(); //Hash 表起初要初始化

    //现在要把整个 big_index 文件插入 hash 表，以取得编码结果
    bigIndex_hash("big_index.txt", "hashpath.txt");
    system("pause");

    return 0;
}
```

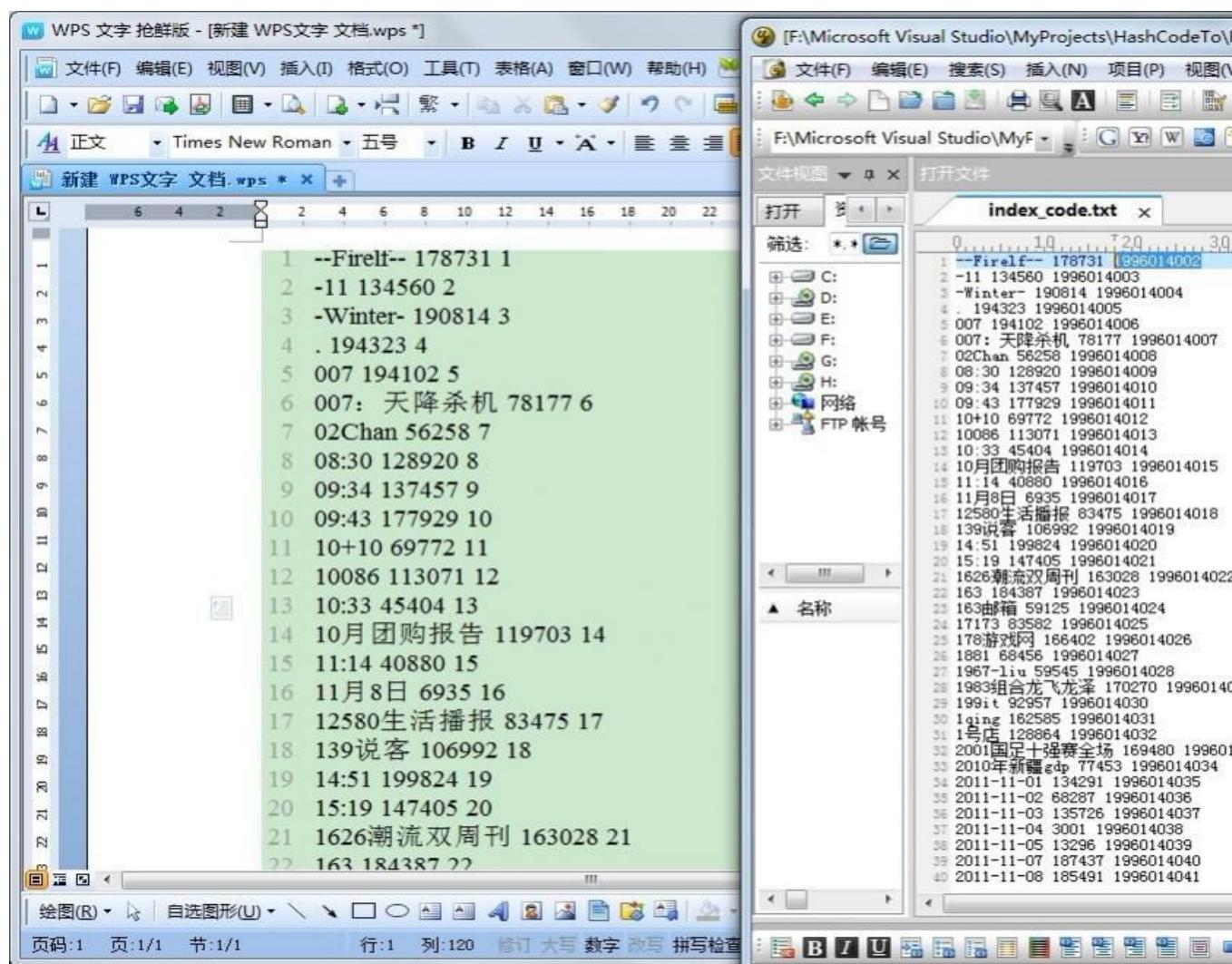
程序运行后生成的 hashpath.txt 文件如下：

```
1 --Firelf-- 178731
2 -11 134560
3 -Winter- 190814
4 . 194323
5 007 194102
6 007: 天降杀机 78177
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 106992
19 14:51 199824
20 15:19 147405
21 1626潮流双周刊 163028
22 163 184387
23 163邮箱 59125
24 17173 83582
25 178游戏网 166402
26 1881 68456
27 1967-liu 59545
28 1983组合龙飞龙泽 170270
29 199it 92957
30 1qing 162585
31 1号店 128884
32 2001国足十强赛全场 169480
33 2010年春晚.adp 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：

```
void bigIndex_hashcode(const char *in_file_path, const char *out_file_path)
{
    FILE *fr, *fw;
```

```

int len, value;
char *pbuf, *pleft, *p;
char keyvalue[TERM_MAX LENG], str[WORD_MAX LENG];

if(in_file_path == NULL || *in_file_path == '\0') {
    printf("input file path error!\n");
    return;
}

if(out_file_path == NULL || *out_file_path == '\0') {
    printf("output file path error!\n");
    return;
}

fr = fopen(in_file_path, "r"); //读取 in_file_path 路径文件
fw = fopen(out_file_path, "w");

if(fr == NULL || fw == NULL)
{
    printf("open read or write file error!\n");
    return;
}

pbuf = (char*)malloc(BUFF_MAX LENG);
pleft = (char*)malloc(BUFF_MAX LENG);
if(pbuf == NULL || pleft == NULL)
{
    printf("allocate memory error!");
    fclose(fr);
    return ;
}

memset(pbuf, 0, BUFF_MAX LENG);

int offset = 1;
while(fgets(pbuf, BUFF_MAX LENG, fr))
{
    if (--offset > 0)
        continue;

    if(GetRealString(pbuf) <= 1)
        continue;

    p = strstr(pbuf, "#####");

```

```

    if(p != NULL)
        continue;

    p = strstr(pbuf, "  ");
    if (p == NULL)
    {
        printf("file contents error!");
    }

    len = p - pbuf;

    // 确定跳过行数
    strcpy(pleft, p+1);
    offset = atoi(pleft) + 1;

    strncpy(keyvalue, pbuf, len);
    keyvalue[len] = '\0';
    value = insert_string(keyvalue);

    if (value != -1) {

        // key value 中插入空格
        keyvalue[len] = ' ';
        keyvalue[len+1] = '\0';

        itoa(value, str, 10);
        strcat(keyvalue, str);

        keyvalue[len+strlen(str)+1] = ' ';
        keyvalue[len+strlen(str)+2] = '\0';

        keyszie++;
        itoa(keyszie, str, 10);
        strcat(keyvalue, str);

        // 将 key value 写入文件
        fprintf (fw, "%s\n", keyvalue);

    }
    free(pbuf);
    fclose(fr);
    fclose(fw);
}

```

## 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。谢谢。完。

## 第二十五章：二分查找实现（Jon Bentley：90%程序员无法正确实现）

作者：July

出处：结构之法算法之道

### 引言

Jon Bentley：90%以上的程序员无法正确无误的写出二分查找代码。也许很多人都早已听说过这句话，但我还是想引用《编程珠玑》上的如下几段文字：

“二分查找可以解决（**预排序数组的查找**）问题：只要数组中包含  $T$ （即要查找的值），那么通过不断缩小包含  $T$  的范围，最终就可以找到它。一开始，范围覆盖整个数组。将数组的中间项与  $T$  进行比较，可以排除一半元素，范围缩小一半。就这样反复比较，反复缩小范围，最终就会在数组中找到  $T$ ，或者确定原以为  $T$  所在的范围实际为空。对于包含  $N$  个元素的表，整个查找过程大约要经过  $\log(2)N$  次比较。

多数程序员都觉得只要理解了上面的描述，写出代码就不难了；但事实并非如此。如果你不认同这一点，最好的办法就是放下书本，自己动手写一写。试试吧。

我在贝尔实验室和 IBM 的时候都出过这道考题。那些专业的程序员有几个小时的时间，可以用他们选择的语言把上面的描述写出来；写出高级伪代码也可以。考试结束后，差不多所有程序员都认为自己写出了正确的程序。于是，我们花了半个钟头来看他们编写的代码经过测试用例验证的结果。几次课，一百多人的结果相差无几：90%的程序员写的程序中有 bug（我并不认为没有 bug 的代码就正确）。

我很惊讶：在足够的时间内，只有大约 10%的专业程序员可以把这个小程序写对。但写不对这个小程序的还不止这些人：高德纳在《计算机程序设计的艺术 第 3 卷 排序和查找》第 6.2.1 节的“历史与参考文献”部分指出，虽然早在 1946 年就有人将二分查找的方法公诸于世，但直到 1962 年才有人写出没有 bug 的二分查找程序。”——乔恩·本特利，《编程珠玑（第 1 版）》第 35-36 页。

你能正确无误的写出二分查找代码么？不妨一试。

## 二分查找代码

二分查找的原理想必不用多解释了，不过有一点必须提醒读者的是，二分查找是针对的排好序的数组。OK，纸上读来终觉浅，觉知此事要躬行。我先来写一份，下面是我写的一份二分查找的实现（之前去某一家公司面试也曾被叫当场实现二分查找，不过结果可能跟你一样，当时就未能完整无误写出），有任何问题或错误，恳请不吝指正：

```
//二分查找 v0.1 实现版
//copyright@2011 July
//随时欢迎读者找 bug, email: zhoulei0907@yahoo.cn。

//首先要把握以下几个要点：
//right=n-1 => while(left <= right) => right=middle-1;
//right=n    => while(left < right) => right=middle;
//middle 的计算不能写在 while 循环外，否则无法得到更新。

int binary_search(int array[], int n, int value)
{
    int left=0;
    int right=n-1;
    //如果这里是 int right = n 的话，那么下面有两处地方需要修改，以保证一一对应：
    //1、下面循环的条件则是 while(left < right)
    //2、循环内当 array[middle]>value 的时候，right = mid

    while (left<=right)          //循环条件，适时而变
    {
        int middle=left + ((right-left)>>1); //防止溢出，移位也更高效。同时，每次循环
       都需要更新。

        if (array[middle]>value)
        {
            right =middle-1; //right 赋值，适时而变
        }
        else if(array[middle]<value)
        {
            left=middle+1;
        }
        else
            return middle;
        //可能会有读者认为刚开始时就要判断相等，但毕竟数组中不相等的情况更多
        //如果每次循环都判断一下是否相等，将耗费时间
    }
    return -1;
```

}

简单测试下，运行结果如下所示（当然，一次测试正确不代表程序便 0 bug 了，且测试深度远远不够）：



The screenshot shows the Microsoft Visual Studio IDE. On the left is the code editor with a file named 'HashStruct.cpp'. The code implements a binary search algorithm. The code is annotated with several green comments explaining specific parts of the algorithm. On the right is a terminal window titled 'F:\Microsoft Vis' with the text '2 请按任意键继续' (Press any key to continue).

```
HashStruct.cpp
binary_search.while.if    else if(array[middle]<value)
(全局范围)
//right=n => while(left < right) => right=middle;
//middle的计算不能写在while循环外，否则无法得到更新。
int binary_search(int array[],int n,int value)
{
    int left=0;
    int right=n-1;
    //如果这里是int right = n 的话，那么下面有两处地方需要修改，以保证一一对应：
    //1. 下面循环的条件则是while(left < right)
    //2. 循环内当array[middle]>value 的时候，right = mid

    while (left<=right)      //循环条件 适时而变
    {
        int middle=left + ((right-left)>>1); //防止溢出 移位也要
        if (array[middle]>value)
        {
            right =middle-1; //right赋值 适时而变
        }
        else if(array[middle]<value)
        {
            left=middle+1;
        }
        else
            return middle;
        //可能会有读者认为刚开始时就要判断相等，但毕竟数组中不相等的值
        //如果每次循环都判断一下是否相等，将耗费时间
    }
    return -1;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int array[4]={1,4,5,7};
    cout<<binary_search(array,4,5)<<endl;
    system("pause");
    return 0;
}
```

## 测试

也许你之前已经把二分查找实现过很多次了，但现在不妨再次测试一下。关闭所有网页，窗口，打开记事本，或者编辑器，或者直接在本文评论下，不参考上面我写的或其他任何人的程序，给自己十分钟到 N 个小时不等的时间，立即编写一个二分查找程序。独立一次性正确写出来后，可以留下代码和邮箱地址，我给你传一份本 blog 的博文集锦 CHM 文件 && 十三个经典算法研究带标签+目录的 PDF 文档（你也可以去我的资源下载处下载：

[http://download.csdn.net/user/v\\_july\\_v](http://download.csdn.net/user/v_july_v)）。

当然，能正确写出来不代表任何什么，不能正确写出来亦不代表什么，仅仅针对 Jon Bentley 的言论做一个简单的测试而已。下一章，请见第二十六章：基于给定的文档生成倒排索引的编码与实践。谢谢。

## 总结

本文发表后，马上就有很多朋友自己尝试了。根据从朋友们在本文评论下留下的代码，发现出错率最高的在以下这么几个地方：

1. 注释里已经说得很明白了，可还是会有很多朋友犯此类的错误：

```
1.      //首先要把握下面几个要点:  
2.      //right=n-1 => while(left <= right) => right=middle-1;  
3.      //right=n    => while(left < right) => right=middle;  
4.      //middle 的计算不能写在 while 循环外，否则无法得到更新。
```

2. 还有一个最常犯的错误是@土豆：

`middle= (left+right)>>1;` 这样的话 left 与 right 的值比较大的时候，其和可能溢出。

各位继续努力。

**updated:** 各位，可以到此处 0 积分下载本 blog 最新博文集锦第 6 期 CHM 文件：

[http://download.csdn.net/detail/v\\_july\\_v/4020172](http://download.csdn.net/detail/v_july_v/4020172)。

# 第二十六章：基于给定的文档生成倒排索引的编码与实践

作者：July、yansha。

出处：结构之法算法之道

## 引言

本周实现倒排索引。实现过程中，寻找资料，结果发现找份资料诸多不易：1、网上搜倒排索引实现，结果千篇一律，例子都是那几个同样的单词；2、到谷歌学术上想找点稍微有价值水平的资料，结果下篇论文还收费或者要求注册之类；3、大部分技术书籍只有理论，没有实践。于是，朋友戏言：网上一般有价值的东西不多。希望，本 blog 的出现能改变此现状。

在第二十四章、倒排索引关键词不重复 Hash 编码中，我们针对一个给定的倒排索引文件，提取出其中的关键词，然后针对这些关键词进行 Hash 不重复编码。本章，咱们再倒退一步，即给定一个正排文档（暂略过文本解析，分词等步骤，日后会慢慢考虑这些且一并予以实现），要求生成对应的倒排索引文件。同时，本章还是基于 Hash 索引之上（运用暴雪的 Hash 函数可以比较完美的解决大数据量下的冲突问题），日后自会实现 B+树索引。

与此同时，本编程艺术系列逐步从为面试服务而转到实战性的编程当中了，教初学者如何编程，如何运用高效的算法解决实际应用中的编程问题，将逐步成为本编程艺术系列的主旨之一。

OK，接下来，咱们针对给定的正排文档一步一步来生成倒排索引文件，有任何问题，欢迎随时不吝赐教或批评指正。谢谢。

## 第一节、索引的构建方法

根据信息检索导论（Christopher D.Manning 等著，王斌译）一书给的提示，我们可以选择两种构建索引的算法：BSBI 算法，与 SPIMI 算法。

**BSBI 算法，基于磁盘的外部排序算法**，此算法首先将词项映射成其 ID 的数据结构，如 Hash 映射。而后将文档解析成词项 ID-文档 ID 对，并在内存中一直处理，直到累积至放满一个固定大小的块空间为止，我们选择合适的块大小，使之能方便加载到内存中并允许在内存中快速排序，快速排序后的块转换成倒排索引格式后写入磁盘。

建立倒排索引的步骤如下：

1. 将文档分割成几个大小相等的部分；
2. 对词项 ID-文档 ID 进行排序；
3. 将具有同一词项 ID 的所有文档 ID 放到倒排记录表中，其中每条倒排记录仅仅是一个文档 ID；
4. 将基于块的倒排索引写到磁盘上。

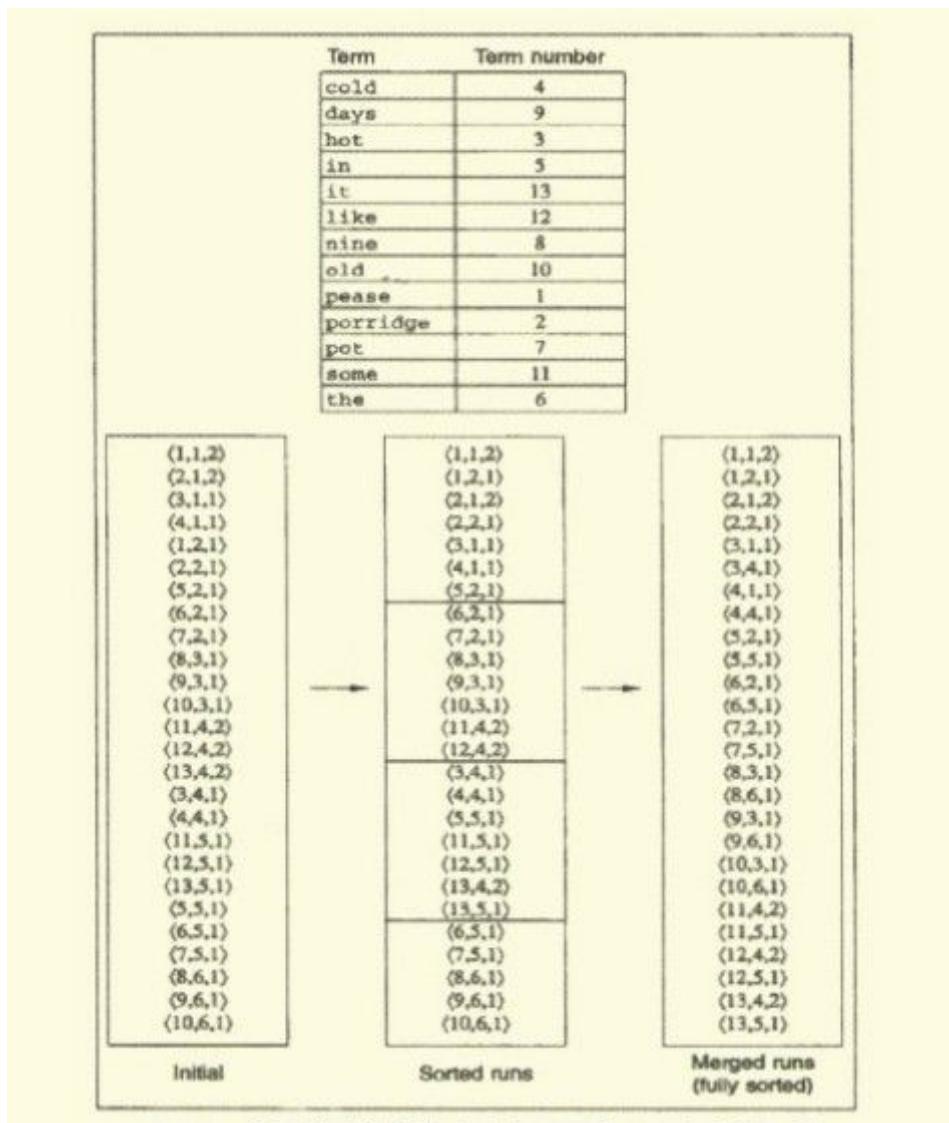
此算法假如说最后可能会产生 10 个块。其伪码如下：

```
BSBI_NDEXCONSTRUCTION()
n <- 0
while(all documents have not been processed)
    do n<-n+1
        block <- PARSENEXTBLOCK()      //文档分析
        BSBI-INVERT(block)
        WRITEBLOCKTODISK(block,fn)
        MERGEBLOCKS(f1,...,fn;fmerged)
```

(基于块的排序索引算法，该算法将每个块的倒排索引文件存入文件 `f1,...,fn` 中，最后合并成 `fmerged`

如果该算法应用最后一步产生了 10 个块，那么接下来便会将 10 个块索引同时合并成一个索引文件。)

合并时，同时打开所有块对应的文件，内存中维护了为 10 个块准备的读缓冲区和一个为最终合并索引准备的写缓冲区。每次迭代中，利用优先级队列（如堆结构或类似的数据结构）选择最小的未处理的词项 ID 进行处理。如下图所示（图片引自深入搜索引擎--海里信息的压缩、索引和查询，梁斌译），分块索引，分块排序，最终全部合并（说实话，跟 MapReduce 还是有些类似的）：



读入该词项的倒排记录表并合并，合并结果写回磁盘中。需要时，再次从文件中读入数据到每个读缓冲区（基于磁盘的外部排序算法的更多可以参考：程序员编程艺术第十章、如何给  $10^7$  个数据量的磁盘文件排序）。

BSBI 算法主要的时间消耗在排序上，选择什么排序方法呢，简单的快速排序足矣，其时间复杂度为  $O(N \log N)$ ，其中  $N$  是所需要排序的项（词项 ID-文档 ID 对）的数目的上界。

### SPIMI 算法，内存式单遍扫描索引算法

与上述 BSBI 算法不同的是：SPIMI 使用词项而不是其 ID，它将每个块的词典写入磁盘，对于写一块则重新采用新的词典，只要硬盘空间足够大，它能索引任何大小的文档集。

倒排索引 = 词典（关键词或词项+词项频率）+倒排记录表。建倒排索引的步骤如下：

1. 从头开始扫描每一个词项-文档 ID（信息）对，遇一词，构建索引；

2. 继续扫描，若遇一新词，则再建一新索引块（加入词典，通过 Hash 表实现，同时，建一新的倒排记录表）；若遇一旧词，则找到其倒排记录表的位置，添加其后
3. 在内存内基于分块完成排序，后合并分块；
4. 写入磁盘。

其伪码如下：

```

SPIMI-Invert(Token_stream)
output.file=NEWFILE()
dictionary = NEWHASH()
while (free memory available)
    do token <-next(token_stream)      //逐一处理每个词项-文档 ID 对
        if term(token) !(- dictionary
            then postings_list = AddToDictionary(dictionary,term(token))      //如果
词项是第一次出现，那么加入 hash 词典，同时，建立一个新的倒排索引表
            else postings_list = GetPostingList(dictionary,term(token))          //如果
不是第一次出现，那么直接返回其倒排记录表，在下面添加其后
            if full(postings_list)
                then postings_list =DoublePostingList(dictionary,term(token))
                AddToPostingsList (postings_list,docID(token))           //SPIMI 与 BSBI 的区别就在
于此，前者直接在倒排记录表中增加此项新纪录
sorted_terms <- SortTerms(dictionary)
WriteBlockToDisk(sorted_terms,dictionary,output_file)
return output_file

```

#### SPIMI 与 BSBI 的主要区别：

SPIMI 当发现关键词是第一次出现时，会直接在倒排记录表中增加一项（与 BSBI 算法不同）。同时，与 BSBI 算法一开始就整理出所有的词项 ID-文档 ID，并对它们进行排序的做法不同（而这恰恰是 BSBI 的做法），这里的每个倒排记录表都是动态增长的（也就是说，倒排记录表的大小会不断调整），同时，扫描一遍就可以实现全体倒排记录表的收集。

SPIMI 这样做有两点好处：

1. 由于不需要排序操作，因此处理的速度更快，
2. 由于保留了倒排记录表对词项的归属关系，因此能节省内存，词项的 ID 也不需要保存。这样，每次单独的 SPIMI-Invert 调用能够处理的块大小可以非常大，整个倒排索引的构建过程也可以非常高效。

但不得不提的是，由于事先并不知道每个词项的倒排记录表大小，算法一开始只能分配一个较小的倒排记录表空间，每次当该空间放满的时候，就会申请加倍的空间，

与此同时，自然而然便会浪费一部分空间（当然，此前因为不保存词项 ID，倒也省下一点空间，总体而言，算作是抵消了）。

不过，至少 SPIMI 所用的空间会比 BSBI 所用空间少。当内存耗尽后，包括词典和倒排记录表的块索引将被写到磁盘上，但在此之前，为使倒排记录表按照词典顺序来加快最后的

合并操作，所以要对词项进行排序操作。

## 小数据量与大数据量的区别

在小数据量时，有足够的内存保证该创建过程可以一次完成；

数据规模增大后，可以采用分组索引，然后再归并索引的策略。该策略是，

1. 建立索引的模块根据当时运行系统所在的计算机的内存大小，将索引分为 k 组，使得每组运算所需内存都小于系统能够提供的最大使用内存的大小。
2. 按照倒排索引的生成算法，生成 k 组倒排索引。
3. 然后将这 k 组索引归并，即将相同索引词对应的数据合并到一起，就得到了以索引词为主键的最终的倒排文件索引，即反向索引。

为了测试的方便，本文针对小数据量进行从正排文档到倒排索引文件的实现。而且针对大量的 K 路归并算法或基于磁盘的外部排序算法本编程艺术系列第十章中已有详细阐述。

## 第二节、Hash 表的构建与实现

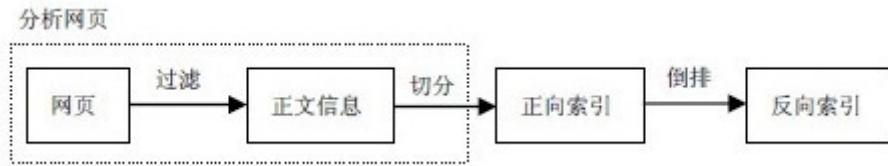
如下，给定如下图所示的正排文档，每一行的信息分别为（中间用#####隔开）：

文档 ID、订阅源（子频道）、频道分类、网站类 ID（大频道）、时间、md5、文档权值、关键词、作者等等。

0	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190																																																								
T0011122300017685#####27#####00#####45#####20111223133703#####31e1a42c8fd4cf5c4e4363e3d4a86#####1#####人人网####果壳##世纪佳缘#####	2	T0011122300017685#####27#####00#####45#####2011122313345#####4951901420a15b48515816545d53#####1#####人人网####果壳##世纪佳缘#####	3	TJ011122300017687#####27#####00#####45#####2011122310200#####3190b9f8d4e78774a0d39505db278#####1#####人人网####果壳##世纪佳缘#####	4	T0011122300017687#####27#####00#####45#####2011122313302#####781247694bb0b084106cbfb9315#####1#####人人网####果壳##世纪佳缘#####	5	TF011122300017694#####27#####00#####45#####20111223133414#####3da4x22e45093m4f1e+e=96262d2#####1#####人人网####果壳##世纪佳缘#####	6	T0011122300017694#####27#####00#####45#####2011122313157#####31a2c2e4f0b1e699f74027d640#####1#####人人网####果壳##世纪佳缘#####	7	T0011122300017702#####27#####00#####45#####2011122313309#####31a533e0f1e+e=98f12775575d2578787857932#####1#####人人网####果壳##世纪佳缘#####	8	TB111122300017702#####27#####00#####45#####2011122313309#####31a533e0f1e+e=98f12775575d2578787857932#####1#####人人网####果壳##世纪佳缘#####	9	TZ011122300017705#####27#####00#####45#####20111223132900#####31a4e4d7f018940d8c1+e03d9-94361#####1#####人人网####果壳##世纪佳缘#####	10	TZ011122300017705#####27#####00#####45#####2011122313437#####31a4e4d7f018940d8c1+e03d9-94361#####1#####人人网####果壳##世纪佳缘#####	11	TZ011122300017707#####27#####00#####45#####2011122313437#####31a4e4d7f018940d8c1+e03d9-94361#####1#####人人网####果壳##世纪佳缘#####	12	TX011122300017709#####27#####00#####45#####20111223133100#####1f22719447458314e278acelce906#####1#####人人网####果壳##世纪佳缘#####	13	TX011122300017710#####27#####00#####45#####20111223133100#####1f22719447458314e278acelce906#####1#####人人网####果壳##世纪佳缘#####	14	TX011122300017710#####27#####00#####45#####20111223133439#####5922708a313382b2b2d6973bae52#####1#####人人网####果壳##世纪佳缘#####	15	TX011122300017712#####27#####00#####45#####20111223132621#####31448a312ade52+ff75f445784fc#####1#####人人网####果壳##世纪佳缘#####	16	TX011122300017712#####27#####00#####45#####20111223132621#####31448a312ade52+ff75f445784fc#####1#####人人网####果壳##世纪佳缘#####	17	TX011122300017711#####27#####00#####45#####20111223132451#####547950e30d176e021967700ee#####1#####人人网####果壳##世纪佳缘#####	18	TX011122300017714#####27#####00#####45#####20111223132451#####547950e30d176e021967700ee#####1#####人人网####果壳##世纪佳缘#####	19	TX011122300017715#####27#####00#####45#####20111223134340#####61d23bf9fb4e17f503733310313#####1#####人人网####果壳##世纪佳缘#####	20	TX011122300017717#####27#####00#####45#####20111223134340#####61d23bf9fb4e17f503733310313#####1#####人人网####果壳##世纪佳缘#####	21	TX011122300017717#####27#####00#####45#####20111223134340#####61d23bf9fb4e17f503733310313#####1#####人人网####果壳##世纪佳缘#####	22	TX011122300017724#####27#####00#####45#####20111223133005#####75d651281f4e+e59+4521+3d765#####1#####人人网####果壳##世纪佳缘#####	23	TX011122300017724#####27#####00#####45#####20111223133005#####75d651281f4e+e59+4521+3d765#####1#####人人网####果壳##世纪佳缘#####	24	TX011122300017724#####27#####00#####45#####20111223133134#####4551a3b0071bb59309528d149#####1#####人人网####果壳##世纪佳缘#####	25	TB011122300017724#####27#####00#####45#####20111223133600#####455234881a0869+e859+2f7743#####1#####人人网####果壳##世纪佳缘#####	26	TB011122300017724#####27#####00#####45#####20111223133700#####4c4ef42e57494247e30a2437a950#####1#####人人网####果壳##世纪佳缘#####	27	TB011122300017724#####27#####00#####45#####20111223133700#####4c4ef42e57494247e30a2437a950#####1#####人人网####果壳##世纪佳缘#####	28	TX011122300017724#####27#####00#####45#####2011122314652#####-893a1e425019630e1b51293n0774#####1#####人人网####果壳##世纪佳缘#####	29	TX011122300017724#####27#####00#####45#####20111223131705#####193557401164e5+745+267694-e724#####1#####人人网####果壳##世纪佳缘#####	30	TZ011122300017730#####27#####00#####45#####20111223132600#####4133155a+b38951309+e0853109#####1#####人人网####果壳##世纪佳缘#####	31	T0011122300017732#####27#####00#####45#####2011122312800#####a1178204e44b57c4b078e484097#####1#####人人网####果壳##世纪佳缘#####	32	T0011122300017732#####27#####00#####45#####2011122312800#####a1178204e44b57c4b078e484097#####1#####人人网####果壳##世纪佳缘#####	33	TB011122300017734#####27#####00#####45#####20111223132900#####5154e1fc2e4b3937300ea105#####1#####人人网####果壳##世纪佳缘#####	34	TB011122300017734#####27#####00#####45#####20111223132900#####5154e1fc2e4b3937300ea105#####1#####人人网####果壳##世纪佳缘#####	35	TB011122300017735#####27#####00#####45#####20111223132900#####53526a317af45b6aa850b2a1045#####1#####人人网####果壳##世纪佳缘#####	36	TB011122300017735#####27#####00#####45#####20111223132900#####53526a317af45b6aa850b2a1045#####1#####人人网####果壳##世纪佳缘#####	37	TB011122300017736#####27#####00#####45#####201112231329#####52d9783de130099594a863b7cf-b#####1#####人人网####果壳##世纪佳缘#####	38	TB011122300017736#####27#####00#####45#####201112231329#####52d9783de130099594a863b7cf-b#####1#####人人网####果壳##世纪佳缘#####	39

要求基于给定的上述正排文档。生成如第二十四章所示的倒排索引文件（注，关键词所在的文章如果是同一个日期的话，是挨在同一行的，用“#”符号隔开）：

我们知道：为网页建立全文索引是网页预处理的核心部分，包括分析网页和建立倒排文件。二者是顺序进行，先分析网页，后建立倒排文件（也称为反向索引），如图所示：



正如上图粗略所示，我们知道倒排索引创建的过程如下：

1. 写爬虫抓取相关的网页，而后提取相关网页或文章中所有的关键词；
  2. 分词，找出所有单词；
  3. 过滤不相干的信息（如广告等信息）；
  4. 构建倒排索引，关键词=>（文章 ID 出现次数 出现的位置）
  5. 生成词典文件 频率文件 位置文件
  6. 压缩。

因为已经给定了正排文档，接下来，咱们跳过一系列文本解析、分词等中间步骤，直接根据正排文档生成倒排索引文档（幸亏有 **yansha** 相助，不然，寸步难行，其微博地址为：<http://weibo.com/yanshazi>，欢迎关注他）。

OK，闲不多说，咱们来一步一步实现吧。

建相关的数据结构

根据给定的正排文档，我们可以建立如下的两个结构体表示这些信息：文档 ID、订阅源（子频道）、频道分类、网站类 ID（大频道）、时间、md5、文档权值、关键词、作者等等。如下所示：

```
typedef struct key_node
{
    char *pkey;      // 关键词实体
    int count;       // 关键词出现次数
    int pos;         // 关键词在 hash 表中位置
    struct doc_node *next; // 指向文档结点
}KEYNODE, *key_list;

key_list key_array[TABLE_SIZE];

typedef struct doc_node
{
    char id[WORD_MAX_LEN]; // 文档 ID
    int classOne;          // 订阅源（子频道）
    char classTwo[WORD_MAX_LEN]; // 频道分类
    int classThree;         // 网站类 ID（大频道）
    char time[WORD_MAX_LEN]; // 时间
    char md5[WORD_MAX_LEN]; // md5
    int weight;             // 文档权值
    struct doc_node *next;
}DOCNODE, *doc_list;
```

我们知道，通过第二十四章的暴雪的 Hash 表算法，可以比较好的避免相关冲突的问题。下面，我们再次引用其代码：

### 基于暴雪的 Hash 之上的改造算法

```
//函数 prepareCryptTable 以下的函数生成一个长度为 0x100 的 cryptTable[0x100]
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 HashString 以下函数计算 lpszFileName 字符串的 hash 值，其中 dwHashType 为 hash 的类型,
unsigned long HashString(const char *lpszkeyName, unsigned long dwHashType )
{
    unsigned char *key = (unsigned char *)lpszkeyName;
    unsigned long seed1 = 0x7FED7FED;
    unsigned long seed2 = 0xFFFFFFFF;
    int ch;

    while( *key != 0 )
    {
        ch = *key++;
        seed1 = cryptTable[(dwHashType<<8) + ch] ^ (seed1 + seed2);
        seed2 = ch + seed1 + seed2 + (seed2<<5) + 3;
    }
    return seed1;
}

//按关键字查询，如果成功返回 hash 表中索引位置
key_list SearchByString(const char *string_in)
{
    const int HASH_OFFSET = 0, HASH_C = 1, HASH_D = 2;
    unsigned int nHash = HashString(string_in, HASH_OFFSET);
    unsigned int nHashC = HashString(string_in, HASH_C);
    unsigned int nHashD = HashString(string_in, HASH_D);
    unsigned int nHashStart = nHash % TABLE_SIZE;
    unsigned int nHashPos = nHashStart;

    while (HashTable[nHashPos].bExists)
    {
        if (HashTable[nHashPos] == (int) nHashC && HashBTable[nHashPos] == (int) nHashD)
        {
            break;
            //查询与插入不同，此处不需修改
        }
        else
        {
            nHashPos = (nHashPos + 1) % TABLE_SIZE;
        }
    }
}

```

```

    }

    if (nHashPos == nHashStart)
    {
        break;
    }
}

if( key_array[nHashPos] && strlen(key_array[nHashPos]->pkey))
{
    return key_array[nHashPos];
}

return NULL;
}

//按索引查询，如果成功返回关键字（此函数在本章中没有被用到，可以忽略）
key_list SearchByIndex(unsigned int nIndex)
{
    unsigned int nHashPos = nIndex;
    if (nIndex < TABLE_SIZE)
    {
        if(key_array[nHashPos] && strlen(key_array[nHashPos]->pkey))
        {
            return key_array[nHashPos];
        }
    }

    return NULL;
}

//插入关键字，如果成功返回 hash 值
int InsertString(const char *str)
{
    const int HASH_OFFSET = 0, HASH_A = 1, HASH_B = 2;
    unsigned int nHash = HashString(str, HASH_OFFSET);
    unsigned int nHashA = HashString(str, HASH_A);
    unsigned int nHashB = HashString(str, HASH_B);
    unsigned int nHashStart = nHash % TABLE_SIZE;
    unsigned int nHashPos = nHashStart;
    int len;

    while (HashTable[nHashPos].bExists)
    {

```

```

nHashPos = (nHashPos + 1) % TABLE_SIZE;

if (nHashPos == nHashStart)
    break;
}

len = strlen(str);
if (!HashTable[nHashPos].bExists && (len < WORD_MAX_LEN))
{
    HashATable[nHashPos] = nHashA;
    HashBTable[nHashPos] = nHashB;

    key_array[nHashPos] = (KEYNODE *) malloc (sizeof(KEYNODE) * 1);
    if(key_array[nHashPos] == NULL)
    {
        printf("10000 EMS ERROR !!!!\n");
        return 0;
    }

    key_array[nHashPos]->pkey = (char *)malloc(len+1);
    if(key_array[nHashPos]->pkey == NULL)
    {
        printf("10000 EMS ERROR !!!!\n");
        return 0;
    }

    memset(key_array[nHashPos]->pkey, 0, len+1);
    strncpy(key_array[nHashPos]->pkey, str, len);
    *((key_array[nHashPos]->pkey)+len) = 0;
    key_array[nHashPos]->pos = nHashPos;
    key_array[nHashPos]->count = 1;
    key_array[nHashPos]->next = NULL;
    HashTable[nHashPos].bExists = 1;
    return nHashPos;
}

if(HashTable[nHashPos].bExists)
    printf("30000 in the hash table %s !!!\n", str);
else
    printf("90000 strkey error !!!\n");
return -1;
}

```

有了这个 Hash 表，接下来，我们就可以把词插入 Hash 表进行存储了。

### 第三节、倒排索引文件的生成与实现

Hash 表实现了（存于 HashSearch.h 中），还得编写一系列的函数，如下所示（所有代码还只是初步实现了功能，稍后在第四部分中将予以改进与优化）：

```
//处理空白字符和空自行
int GetRealString(char *pbuf)
{
    int len = strlen(pbuf) - 1;
    while (len > 0 && (pbuf[len] == (char)0x0d || pbuf[len] == (char)0x0a || pbuf[len] == ' ' || pbuf[len] == '\t'))
    {
        len--;
    }

    if (len < 0)
    {
        *pbuf = '\0';
        return len;
    }
    pbuf[len+1] = '\0';
    return len + 1;
}

//重新 strcoll 字符串比较函数
int strcoll(const void *s1, const void *s2)
{
    char *c_s1 = (char *)s1;
    char *c_s2 = (char *)s2;
    while (*c_s1 == *c_s2++)
    {
        if (*c_s1++ == '\0')
        {
            return 0;
        }
    }

    return *c_s1 - *--c_s2;
}

//从行缓冲中得到各项信息，将其写入 items 数组
void GetItems(char *&move, int &count, int &wordnum)
{
```

```

char *front = move;
bool flag = false;
int len;
move = strstr(move, "#####");
if (*(move + 5) == '#')
{
    flag = true;
}

if (move)
{
    len = move - front;
    strncpy(items[count], front, len);
}
items[count][len] = '\0';
count++;

if (flag)
{
    move = move + 10;
} else
{
    move = move + 5;
}
}

//保存关键字相应的文档内容
doc_list SaveItems()
{
    doc_list infolist = (doc_list) malloc(sizeof(DOCNODE));
    strcpy_s(infolist->id, items[0]);
    infolist->classOne = atoi(items[1]);
    strcpy_s(infolist->classTwo, items[2]);
    infolist->classThree = atoi(items[3]);
    strcpy_s(infolist->time, items[4]);
    strcpy_s(infolist->md5, items[5]);
    infolist->weight = atoi(items[6]);
    return infolist;
}

//得到目录下所有文件名
int GetFileName(char filename[][FILENAME_MAX_LEN])
{
    _finddata_t file;

```

```

long handle;
int filenum = 0;
//C:\Users\zhangxu\Desktop\CreateInvertedIndex\data
if ((handle = _findfirst("C:\\\\Users\\\\zhangxu\\\\Desktop\\\\CreateInvertedIndex\\\\data\\*.txt", &file)) == -1)
{
    printf("Not Found\\n");
}
else
{
    do
    {
        strcpy_s(filename[filenum++], file.name);
    } while (!_findnext(handle, &file));
}
_findclose(handle);
return filenum;
}

//以读方式打开文件，如果成功返回文件指针
FILE* OpenReadFile(int index, char filename[][FILENAME_MAX_LEN])
{
    char *abspath;
    char dirpath[] = {"data\\\"};
    abspath = (char *)malloc(ABSPATH_MAX_LEN);
    strcpy_s(abspath, ABSPATH_MAX_LEN, dirpath);
    strcat_s(abspath, FILENAME_MAX_LEN, filename[index]);

    FILE *fp = fopen (abspath, "r");
    if (fp == NULL)
    {
        printf("open read file error!\\n");
        return NULL;
    }
    else
    {
        return fp;
    }
}

//以写方式打开文件，如果成功返回文件指针
FILE* OpenWriteFile(const char *in_file_path)
{
    if (in_file_path == NULL)

```

```

{
    printf("output file path error!\n");
    return NULL;
}

FILE *fp = fopen(in_file_path, "w+");
if (fp == NULL)
{
    printf("open write file error!\n");
}
return fp;
}

```

最后，主函数编写如下：

```

int main()
{
    key_list keylist;
    char *pbuf, *move;
    int filenum = GetFileName(filename);
    FILE *fr;
    pbuf = (char *)malloc(BUF_MAX_LEN);
    memset(pbuf, 0, BUF_MAX_LEN);

    FILE *fw = OpenWriteFile("index.txt");
    if (fw == NULL)
    {
        return 0;
    }

    PrepareCryptTable(); //初始化 Hash 表

    int wordnum = 0;
    for (int i = 0; i < filenum; i++)
    {
        fr = OpenReadFile(i, filename);
        if (fr == NULL)
        {
            break;
        }

        // 每次读取一行处理
        while (fgets(pbuf, BUF_MAX_LEN, fr))
        {

```

```

int count = 0;
move = pbuf;
if (GetRealString(pbuf) <= 1)
    continue;

while (move != NULL)
{
    // 找到第一个非'#'的字符
    while (*move == '#')
        move++;

    if (!strcmp(move, ""))
        break;

    GetItems(move, count, wordnum);
}

for (int i = 7; i < count; i++)
{
    // 将关键字对应的文档内容加入文档结点链表中
    if (keylist = SearchByString(items[i]))      //到 hash 表内查询
    {
        doc_list infolist = SaveItems();
        infolist->next = keylist->next;
        keylist->count++;
        keylist->next = infolist;
    }
    else
    {
        // 如果关键字第一次出现，则将其加入 hash 表
        int pos = InsertString(items[i]);          //插入 hash 表
        keylist = key_array[pos];
        doc_list infolist = SaveItems();
        infolist->next = NULL;
        keylist->next = infolist;
        if (pos != -1)
        {
            strcpy_s(words[wordnum++], items[i]);
        }
    }
}
}

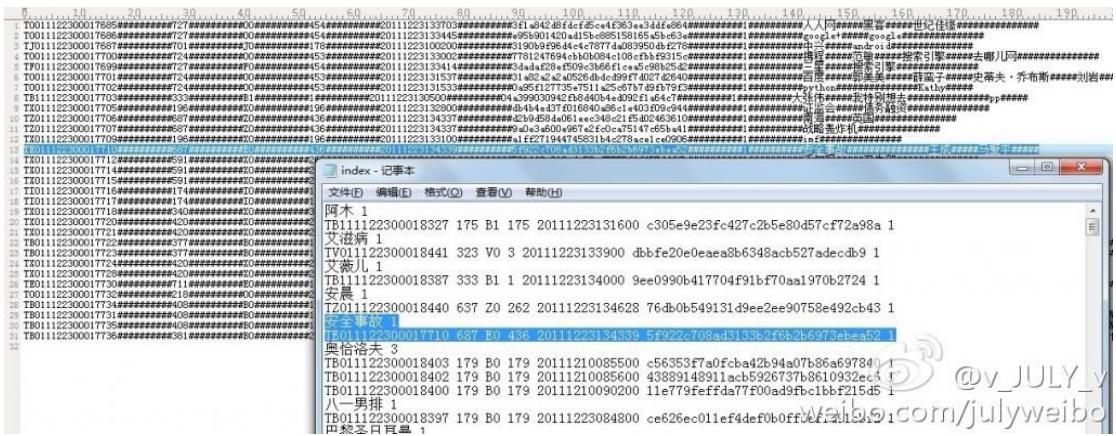
```

```
// 通过快排对关键字进行排序
qsort(words, WORD_MAX_NUM, WORD_MAX_LEN, strcoll);

// 遍历关键字数组，将关键字及其对应的文档内容写入文件中
for (int i = 0; i < WORD_MAX_NUM; i++)
{
    keylist = SearchByString(words[i]);
    if (keylist != NULL)
    {
        fprintf(fw, "%s %d\n", words[i], keylist->count);
        doc_list infolist = keylist->next;
        for (int j = 0; j < keylist->count; j++)
        {
            // 文档 ID, 订阅源 (子频道) 频道分类 网站类 ID (大频道) 时间 md5, 文档权
            值
            fprintf(fw, "%s %d %s %d %s %s %d\n", infolist->id, infolist->class
One,
            infolist->classTwo, infolist->classThree, infolist->time, infol
ist->md5, infolist->weight);
            infolist = infolist->next;
        }
    }
}

free(pbuf);
fclose(fr);
fclose(fw);
system("pause");
return 0;
}
```

程序编译运行后，生成的倒排索引文件为 `index.txt`，其与原来给定的正排文档对照如下：



有没有发现关键词奥恰洛夫出现在现在的三篇文章是同一个日期 1210 的，貌似与本文开头指定的倒排索引格式要求不符？因为第二部分开头中，已明确说明：“注，关键词所在的文章如果是同一个日期的话，是挨在同一行的，用“#”符号隔开”。OK，有疑问是好事，代表你思考了，请直接转至下文第 4 部分。

## 第四节、程序需求功能的改进

### 4.1、对相同日期与不同日期的处理

细心的读者可能还是会注意到：在第二部分开头中，要求基于给定的上述正排文档。生成如第二十四章所示的倒排索引文件是下面这样子的，即是：

	0	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190		
1	-1111116	2	T0011111600004603	240	00	240	a2f4d51e6f231770fd1e0e**e76	85358	1####T0011111600004604	240	00	240	2194a6752ab6d6d6d45bbfb**d176	84539	1####							
2	20111115	5	T0011111500008241	240	00	240	367e627e**e9e5831x9217435b594	131221	1####T0011111500007324	240	00	240	201586e55f52fb1f33d960e674974	113059	1####T0011111500007578	240	00					
3	20111114	1	T0011111400004045	240	00	240	6d8a715558a5e97b15724ef050b8c1	123028	1####													
4	20111113	5	T001111130000399	240	00	240	a2d41e40e**e9efea2374e74234	9428	1####T00111113000039741	240	00	240	a2659e2734d4022a7e5d3633af672	94032	1####T0011111300002535	240	00	240	71			
5	20111110	9	T0011111000011295	0	00	240	9977856715c10e**e009719505495	160408	1####T0011111000010700	240	00	240	23037e29694a6e**e1d49537cf0f492	155151	1####T0011111000010701	240	00	240				
6	20111109	4	T0011110900009703	0	00	240	c94b47f5c7e59a**e65923a67c33a20	140003	1####T0011110900009672	48	00	240	47763967e0347c9d36b6c**e3111	100008	1####T001111090000823	48	00	24				
7	20111108	1	T0011110800010412	48	00	240	e0a120256c51e0a10411	48	163199	1####T0011110800010411	48	00	240	e0a120256c51e0a10411	163199	1####						
8	20111107	11	P0011111500017183	156	UD	156	+9ecf9096791a0325457a4e45479e74	204459	1####P0011111500017182	156	UD	156	+9ecf9096791a0325457a4e45479e74	204454	1####P0011111500017181	156	UD	15				
9	20111106	2	P0011110900013663	120	UD	156	b5a88e8e020436a**e537f5b6cf0e0309	205723	1####P00111109000108419	120	UD	156	b5a88e8e020436a**e537f5b6cf0e0309	122538	1####							
10	20111105	14																				
11	20111116	1	TM011111600017185	317	ND	317	e+840543c1e234a**d0604977a157a	143222	1####													
12	20111115	1	TM011111500017184	317	ND	317	c8d7675e**e294703a**e31bf7c1514	9335	1####													
13	20111114	1	TM011111400004045	317	ND	317	1145a144e**e1f4a1e40e**e040500	183120	1####T0011111400004034	317	ND	317	9237d4e5843bc1e37b19434970402	183116	1####							
14	20111113	2	TM011111300002493	317	ND	317	253704e6512164101b13352c0655833	162256	1####T0011111300001507	317	ND	317	6-e2f901049ca59fe8b50b85e3	114656	1####							
15	20111110	1	TM01111100000682	0	ND	317	f526929a8e424209505f7a24e**e04cc	112537	1####													
16	20111109	1	TM011110900008493	274	ND	317	e14480042167098764656e**e04cc	112551	1####													
17	20111108	107	P001110800008495	274	ND	317	9511969105048a822314d7e**e2aa15	223000	1####P001110800008495	274	ND	317	85f5f98413de880646748293fc190+9	92700	1####							
18	20111107	2	P001110800008495	274	ND	317	8a6e5f5e5c**e32a9414d4889173a	175400	1####													
19	20111106	1	TM011110800008497	274	ND	317	317a144e**e1f4a1e40e**e040500	175400	1####T0011110800008500	274	ND	317	5c16e1a7cdcf3c7021b0c43d612a5981	143200	1####T001110800008501	274	ND	31				
20	20111105	3	TM011110800008499	274	ND	317	9216e11645317a404659656957e9a	165400	1####T001110800008500	274	ND	317	5c16e1a7cdcf3c7021b0c43d612a5981	143200	1####T001110800008501	274	ND	31				
21	20111103	1	TM01111080000502	274	ND	317	594e62834544a9891c7e**e037b1	171000	1####													
22	20111102	1	TM01111080000502	274	ND	317	e24246404d10b198428ac**e0347eac	85080	1####													
23	20111101	2	P001111080000504	274	ND	317	9190867388e827a01a035e03e8961	105200	1####P001111080000504	274	ND	317	9190867388e827a01a035e03e8961	105200	1####							
24	20111100	2	P001111080000504	274	ND	317	9190867388e827a01a035e03e8961	105200	1####P001111080000504	274	ND	317	9190867388e827a01a035e03e8961	105200	1####							
25	20111102	1	TB111112700002087	369	B1	27245f5c**e46a2e25491f**e2083	91342	1####														
26	20111101	2	TB111112700002089	369	B1	27245f5c**e46a2e25491f**e2083	91342	1####														
27	20111100	126	TB11111260000933	282	B1	282c034aaaae090912485f1c5e05489	142910	1####														
28	02Chan	11																				
29	20111116	5	T001111600020945	225	00	225	d3aae1f1a1e1b143a1e444220593e	175011	1####T001111600020944	225	00	225	a71827d4e20165201c8415c320a8	162802	1####T0011111600015428	225	00	225				
30	20111115	11	T001111500013121	225	00	225	051177989708a398335e3b3a5	150157	1####T001111500013122	225	00	225	a554e6585820d1749581c34a287	160340	1####T0011111500013124	225	00	225				
31	20111114	2	T001111500008407	225	00	225	a5c5a7e2895a8a598634d731227	233516	1####T001111500008408	225	00	225	a962e2340949dcda30350542d91a15	162057	1####							
32	20111113	1	T0011111000012702	225	00	225	a5e0212e6533e3d128a4d2e09445	191540	1####T001111100003100	225	00	225	1428e7a8b1d1c895737643c2e8c	80527	1####							
33	20111112	9	T0011110900013032	108	00	225	c496e2226394b1a90498e488	172640	1####T0011110900004817	108	00	225	4e29e1fc98b880dd4a895d5672225145	100047	1####							

也就是说，上面建索引的过程本该是如下的：

- 倒排索引 = 词典{关键词或词项+词项频率} + 倒排记录表  
 1、从头开始扫描每一个词项-文档ID(信息)对，遇一词，构建索引  
 2、继续扫描  
     若遇一新词，则再建一新索引块(加入词典，通过Hash实现)，同时，建一新的倒排记录表  
     -相同日期，添加到之前同一日期的记录之后(第一个记录的后面记下同一日期的记录数目)；不同日期，另起一行新增记录  
 3、在内存内基于分块完成排序，后合并分块(这里可以参考程序员编程艺术第十章)  
 4、写入磁盘

与第一部分所述的 SMIPI 算法有什么区别？对的，就在于对在同一个日期的出现的关键词的处理。如果是遇一旧词，则找到其倒排记录表的位置：相同日期，添加到之前同一日期的记录之后(第一个记录的后面记下同一日期的记录数目)；不同日期，另起一行新增记录。

相同（单个）日期，根据文档权值排序

不同日期，根据时间排序

代码主要修改如下：

```

//function: 对链表进行冒泡排序
void ListSort(key_list keylist)
{
    doc_list p = keylist->next;
    doc_list final = NULL;
    while (true)
    {
        bool isfinish = true;
        while (p->next != final) {
            if (strcmp(p->time, p->next->time) < 0)
            {
                SwapDocNode(p);
                isfinish = false;
            }
            p = p->next;
        }
        final = p;
        p = keylist->next;
        if (isfinish || p->next == final) {
            break;
        }
    }
}

int main()
{
    key_list keylist;
    char *pbuff, *move;
    int filenum = GetFileName(filename);
    FILE *frp;
    pbuf = (char *)malloc(BUF_MAX_LEN);
    memset(pbuf, 0, BUF_MAX_LEN);

    fwp = OpenWriteFile("index.txt");
    if (fwp == NULL) {
        return 0;
    }

    PrepareCryptTable();

    int wordnum = 0;
    for (int i = 0; i < filenum; i++)
    {
        frp = OpenReadFile(i, filename);

```

```

    if (frp == NULL) {
        break;
    }

    // 每次读取一行处理
    while (fgets(pbuf, BUF_MAX_LEN, frp))
    {
        int count = 0;
        move = pbuf;
        if (GetRealString(pbuf) <= 1)
            continue;

        while (move != NULL)
        {
            // 找到第一个非'#'的字符
            while (*move == '#')
                move++;

            if (!strcmp(move, ""))
                break;

            GetItems(move, count, wordnum);
        }

        for (int i = 7; i < count; i++) {
            // 将关键字对应的文档内容加入文档结点链表中
            // 如果关键字第一次出现，则将其加入 hash 表
            if (keylist = SearchByString(items[i])) {
                doc_list infolist = SaveItems();
                infolist->next = keylist->next;
                keylist->count++;
                keylist->next = infolist;
            } else {
                int pos = InsertString(items[i]);
                keylist = key_array[pos];
                doc_list infolist = SaveItems();
                infolist->next = NULL;
                keylist->next = infolist;
                if (pos != -1) {
                    strcpy_s(words[wordnum++], items[i]);
                }
            }
        }
    }
}

```

```

}

// 通过快排对关键字进行排序
qsort(words, WORD_MAX_NUM, WORD_MAX_LEN, strcoll);

// 遍历关键字数组，将关键字及其对应的文档内容写入文件中
int rownum = 1;
for (int i = 0; i < WORD_MAX_NUM; i++) {
    keylist = SearchByString(words[i]);
    if (keylist != NULL) {
        doc_list infolist = keylist->next;

        char date[9];

        // 截取年月日
        for (int j = 0; j < keylist->count; j++)
        {
            strncpy_s(date, infolist->time, 8);
            date[8] = '\0';
            strncpy_s(infolist->time, date, 9);
            infolist = infolist->next;
        }

        // 对链表根据时间进行排序
        ListSort(keylist);

        infolist = keylist->next;
        int *count = new int[WORD_MAX_NUM];
        memset(count, 0, WORD_MAX_NUM);
        strcpy_s(date, infolist->time);
        int num = 0;
        // 得到单个日期的文档数目
        for (int j = 0; j < keylist->count; j++)
        {
            if (strcmp(date, infolist->time) == 0) {
                count[num]++;
            } else {
                count[++num]++;
            }
            strcpy_s(date, infolist->time);
            infolist = infolist->next;
        }
        fprintf(fwp, "%s %d %d\n", words[i], num + 1, rownum);
        WriteFile(keylist, num, fwp, count);
    }
}

```

```

        rounum++;
    }

}

free(pbuf);
// fclose(frp);
fclose(fwp);
system("pause");
return 0;
}

```

修改后编译运行，生成的 index.txt 文件如下：

```

阿木 1
20111223 1 TB111122300018327 175 B1 175 c305e9e23fc427c2b5e80d57cf72a98a 1#####
艾滋病 1
20111223 1 TV011122300018441 323 V0 3 dbbfe20e0eaaa8b6348acb527adecdb9 1#####
艾薇儿 1
20111223 1 TB111122300018387 333 B1 1 9ee0990b417704f91bf70aa1970b2724 1#####
安晨 1
20111223 1 TZ011122300018440 637 Z0 262 76db0b549131d9ee2ee90758e492cb43 1#####
安全事故 1
20111223 1 TE011122300017710 687 E0 436 5f922c708ad3133b2f6b2b6973eba52 1#####
奥恰洛夫 1
20111210 3 TB011122300018403 179 B0 179 c56353f7a0fcba42b94a07b86a697840 1#####TB011122300018402 179 B0 179 43889148911acb5926737b8610932ec
179 B0 179 11e779feff7a77f00ad9fb1bbf215d5 1#####
八一男排 1
20111223 1 TB011122300018397 179 B0 179 ce626ec011ef4def0b0ff5ef1d91ebf2 1#####
巴黎圣母 1
20111223 1 TB011122300017734 408 B0 178 5154e6elcfe2cd636b7d300aaf541054 1#####
百度 1
20111223 1 TO011122300017701 724 00 453 31a82a2a2a0526dbdc99f7d027d2640 1#####
贝克汉姆 1
20111223 1 TB011122300017734 408 B0 178 5154e6elcfe2cd636b7d300aaf541054 1#####
本田 1
20111223 1 TI011122300017717 174 I0 174 5bbd509ddeced74af8ec6c3d5598016 1#####
冰球 1
20111223 1 TB011122300018430 381 B0 2 e260d29a312fe9dc88833099976b47d0 1#####
波尔 1
20111210 2 TB011122300018402 179 B0 179 43889148911acb5926737b8610932ec6 1#####TB011122300018399 179 B0 179 aa78f9b7ad6b50ad27eb0f1a797baaa
波什 1
20111223 1 TB011122300017736 381 B0 2 e32d97783deb130099594a863b7cfbc7 1#####
博物馆奇妙夜 1
20111223 1 TB111122300018424 704 B1 262 2cb105e259508fdc64344253fb17106f 1#####
不速之约 1
20111223 1 TB111122300018327 175 B1 175 c305e9e23fc427c2b5e80d57cf72a98a 1#####
超人 1
20111223 5 TB011122300018412 179 B0 179 42dc49ad014ffc4460eff6c9b54cec52 1#####TB011122300018411 179 B0 179 1b2cd35059e507361d1927a57c2d718
179 B0 179 7a63502df9a3f9db0928ad1136c65ca3 1#####TB011122300018397 179 B0 179 ce626ec011ef4def0b0ff5ef1d91ebf2 1#####TB011122300018396 179
7fdac5720a1029f98a5c27fb06ac5c 1#####
陈光标 1
20111223 1 TO011122300017701 724 00 453 31a82a2a0526dbdc99f7d027d2640 1#####
陈洁 1
20111223 2 TX011122300018373 340 X0 3 7f4459a5ef3da2e7cabdcdf8281bc5d6 1#####TX011122300018371 340 X0 3 f4487368f4b2fe349558ce60128635b 1#
传世群英传 1

```

## 4.2、为关键词添上编码

如上图所示，已经满足需求了。但可以再在每个关键词的背后添加一个计数表示索引到了第多少个关键词：

```

网木 1_1
文进海 1_2 TB111122300018327 175 B1 175 c305e9e23fc427c2b5e80457cf72a98a 1#####
20111223 1 TV011122300018444 322 VO 3 dbbfe20e0eae8b6348ac527adecdb9 1#####
文进海 1_3
20111223 1 TB111122300018387 333 B1 1 9ee0990b417704f91bf70a1970b2724 1#####
安惠 1_d
20111223 1 TZO11122300018449 637 YO 262 76db05e491319ee2ee90758e492cb43 1#####
安惠 1_e
20111223 1 TE011122300017170 687 EO 438 5f922c708ad3133a2f62b697sebea52 1#####
奥怡洛夫 1_f
20111223 1 TB011122300018403 179 BO 179 43889148911acb5926737b8610932ec6 1#####TB011122300018400
179 BO 179 11e7759effda77f00a49fb1bbf215d5 1#####
八一男排 1_g
20111223 1 TB011122300018397 179 BO 179 ce626e011ef4def0b0ffef1491ebf2 1#####
巴黎圣日耳曼 1_h
20111223 1 TB011122300017734 408 BO 178 5154e6e1fe2cd63674300aa5f41054 1#####
百度 1_i
20111223 1 T0011122300017701 724 00 453 31a82a2a2a0526db-dcd99f74027d2640 1#####
贝克汉姆 1_j
20111223 1 TB011122300017733 408 BO 178 5154e6e1fe2cd63674300aa5f41054 1#####
朱由检 1_k
20111223 1 TB011122300017717 174 IO 174 5bb-d509ddceeed74af8ec6c3d598016 1#####
冰球 1_l
20111223 1 TB011122300018430 381 BO 2 e260a29a312fe94e8883309976b47d0 1#####
20111223 1 TB011122300018402 178 BO 179 43889148911acb5926737b8610932ec6 1#####TB011122300018399 179 BO 179 aa78f9b7ad6b50ad27eb0f1a797baaa1 1#####
波什 1_m
20111223 1 TB011122300017736 381 BO 2 e32d97783debl30099594a863b7cfbc7 1#####
博物诙谐妙语 1_n
20111223 1 TB111122300018424 704 B1 262 2cb105e259508f6e64344253fb17106f 1#####
不速之客 1_o
20111223 1 TB111122300018327 175 B1 175 c305e9e23fc427c2b5e80457cf72a98a 1#####
超人 1_p
20111223 5 TB011122300018412 173 BO 179 424c49a-e014ff-e469e0ff5-9b54-c-e52 1#####TB011122300018411 179 BO 179 152-c435059e507361d1927a57c2d7199 1#####TB011122300018410
7fdac5720a1029fb9aa5c27bfb0928ad1136-65c83 1#####TB011122300018397 179 BO 179 ce626e011ef4def0b0ffef1491ebf2 1#####TB011122300018396 179 BO 179
陈光标 1_q
20111223 1 T0011122300017701 724 00 453 31a82a2a2a0526db-dcd99f74027d2640 1#####
陈洁 1_r
20111223 2 TX011122300018373 340 X0 3 7f4459a5ef3da2e7cabdcdf8281bc5d6 1#####TX011122300018371 340 X0 3 f4487368fb2fe349558cce60128635b 1#####
伟世环球英传 1_s

```

## 第五节、算法的二次改进

### 5.1、省去二次 Hash

针对本文评论下读者的留言，做了下思考，自觉可以省去二次 hash:

```

for (int i = 7; i < count; i++)
{
    // 将关键字对应的文档内容加入文档结点链表中
    //也就是说当查询到 hash 表中没有某个关键词之,后便会插入
    //而查询的时候, search 会调用 hashstring, 得到了 nHashC , nHashD
    //插入的时候又调用了一次 hashstring, 得到了 nHashA, nHashB
    //而如果查询的时候, 是针对同一个关键词查询的, 所以也就是说 nHashC&nHashD, 与
nHashA&nHashB 是相同的, 无需二次 hash

    //所以, 若要改进, 改的也就是下面这个 if~else 语句里头。July, 2011.12.30。
    if (keylist = SearchByString(items[i]))          //到 hash 表内查询
    {
        doc_list infolist = SaveItems();
        infolist->next = keylist->next;
        keylist->count++;
        keylist->next = infolist;
    }
    else
    {
        // 如果关键字第一次出现, 则将其加入 hash 表
        int pos = InsertString(items[i]);           //插入 hash 表
        keylist = key_array[pos];
        doc_list infolist = SaveItems();
        infolist->next = NULL;
        keylist->next = infolist;
        if (pos != -1)

```

```

        {
            strcpy_s(words[wordnum++], items[i]);
        }
    }
}

// 通过快排对关键字进行排序
qsort(words, WORD_MAX_NUM, WORD_MAX_LEN, strcoll);

```

## 5.2、除去排序，针对不同日期的记录直接插入

```

//对链表进行冒泡排序。这里可以改成快速排序：等到统计完所有有关这个关键词的文章之后，才能对他
集体快排。
//但其实完全可以用插入排序，不同日期的，根据时间的先后找到插入位置进行插入：
//假如说已有三条不同日期的记录 A B C
//来了 D 后，发现 D 在 C 之前，B 之后，那么就必须为它找到 B C 之间的插入位置，
//A B D C。July、2011.12.31。
void ListSort(key_list keylist)
{
    doc_list p = keylist->next;
    doc_list final = NULL;
    while (true)
    {
        bool isfinish = true;
        while (p->next != final) {
            if (strcmp(p->time, p->next->time) < 0) //不同日期的按最早到最晚排序
            {
                SwapDocNode(p);
                isfinish = false;
            }
            p = p->next;
        }
        final = p;
        p = keylist->next;
        if (isfinish || p->next == final) {
            break;
        }
    }
}

```

综上 5.1、5.2 两节免去冒泡排序和，省去二次 hash 和免去冒泡排序，修改后如下：

```

for (int i = 7; i < count; i++) {
    // 将关键字对应的文档内容加入文档结点链表中
    // 如果关键字第一次出现，则将其加入 hash 表
    InitHashValue(items[i], hashvalue);
    if (keynode = SearchByString(items[i], hashvalue)) {
        doc_list infonode = SaveItems();
        doc_list p = keynode->next;
        // 根据时间由早到晚排序
        if (strcmp(infonode->time, p->time) < 0) {
            //考虑 infonode 插入 keynode 后的情况
            infonode->next = p;
            keynode->next = infonode;
        } else {
            //考虑其他情况
            doc_list pre = p;
            p = p->next;
            while (p)
            {
                if (strcmp(infonode->time, p->time) > 0) {
                    p = p->next;
                    pre = pre->next;
                } else {
                    break;
                }
            }
            infonode->next = p;
            pre->next = infonode;
        }
        keynode->count++;
    } else {
        int pos = InsertString(items[i], hashvalue);
        keynode = key_array[pos];
        doc_list infolist = SaveItems();
        infolist->next = NULL;
        keynode->next = infolist;
        if (pos != -1) {
            strcpy_s(words[wordnum++], items[i]);
        }
    }
}
}

// 通过快排对关键字进行排序

```

```
qsort(words, WORD_MAX_NUM, WORD_MAX_LEN, strcoll);
```

修改后编译运行的效果图如下（用了另外一份更大的数据文件进行测试）：



```
丁子高 1
20111217 4 TB111121700003444 570 B1 389 2456c31c499544001716d1286402947 #####TB111121700002089 350 B1 3 4a0ae42f1222c016bd19e6d94271ab19 1#####TB11112170000348 333
B1 3 4a0ae42f1222c016bd19e6d94271ab19 1#####
丁建庭 1 2
2011104 1 TA011121800009460 687 A0 436 defb6dde1967739a2571dac3336bed5a 1#####
丁志军 2 3
2011107 1 TA011121800009347 687 A0 436 3e127cd4262f2e7a6200ff961499ab6c 1#####
2011031 1 TA011121800009594 687 A0 436 0249925c174daba41c970cac6f8027dc 1#####
丁文杰 1 4
2011101 1 TA01112180000986 687 A0 436 c3f94743dcf77f6f275074919b5a5284 1#####
丁步翠 1 5
2011020 1 TA011121800009762 687 A0 436 04f05c3691af61bb21b41ddbd9ec30 1#####
丁秀玉 1 6
2011102 2 TA011121800010368 687 A0 436 c19077b7b1a17fd9f76109e064670df3 1#####
丁伟魁 1 7
2011216 5 TB011121700005695 165 B0 165 b222384d4c98c6110749cf0a0164543cc 1#####TB011121700002713 381 B0 2 fb10beb3c788143a906f5670c5bf3b3d 1#####TB011121700002761 566
B0 162 de575d30d524d2ba57e190a007728355 1#####TB011121700002546 408 B0 178 c7d2713043094e7943d4612c4eb2dc0c 1#####TB011121700001539 377 B0 172
2>8c398054866a4f3949b5a0e230celce 1#####
七彩虹 1 8
20111210 1 TJ011121700004141 216 J0 216 5a17b8a877498a35648830095db-d9e4d 1#####
万兵 1 9
2011217 1 TV011121700002878 323 V0 3 d69fe6fa8ce53793c8e2ea8e35ace940 1#####
万凰之王 1 10
2011121 2 TB111121700002077 704 B1 262 d3c2634d6bb665f1deebbbb8e65dc16cb 1#####TB111121700000106 175 B1 175 d3c2634d6bb665f1deebbbb8e65dc16cb 1#####
万力达 1 11
2011121 1 TX011121700003870 591 X0 262 294fd48c89abc6d09a9df614c003f1 1#####
万里 3 12
20111217 1 TX011121700001753 420 X0 2 5819e1f6b57e74bb5208c066cd21fbfc 1#####
2011116 1 TA011121600068393 577 A0 394 28846795b4d57296a3735dcace4365e 1#####
2011028 1 TA011121800010162 687 A0 436 eb939c7c95a1d2000a413fad7c1326 1#####
三星 1 13
20111217 1 TX011121700001805 420 X0 2 4852fdbcc038a03089c1ef1e066fd 1#####
三星 7 17
20111217 2 TJ011121700004040 460 J0 172 a5e5627ce9a80c61406f7658d77e211e 1#####TJ011121700004013 653 J0 214 860650c35f9312c7ad896f36e678277 1#####TJ011121700003167
607 J0 402 a8a9a078b9de3b83bf5a2e896c8ddc0d 1#####TJ011121700003172 607 J0 402 96cff05b236194c0b077096894dd86 1#####TJ011121700002844 460 J0 172
c14e4dc0b018e07aab8979eb27e838c99 1#####TJ011121700002806 460 J0 172 5342cc247684b3f39cd840903bed7166 1#####TJ011121700002297 701 J0 178
```

本章全部源码请到以下两处任一处下载（欢迎读者朋友们继续优化，若能反馈于我，则幸甚不过了）：

1. [http://download.csdn.net/detail/v\\_july\\_v/4012605](http://download.csdn.net/detail/v_july_v/4012605) (csdn 下载处)
2. <https://github.com/fuxiang90/CreateInvertedIndex>. (github 下载处)

## 后记

本文代码还有很多的地方可以改进和优化，请待后续更新。当然，代码看起来也很青嫩，亟待提高阿。

近几日后，准备编程艺术室内 38 位兄弟的靓照和 blog 或空间地址公布在博客内，给读者一个联系他们的方式，顺便还能替他们征征友 招招婚之类的。ys，土豆，水哥，老梦，3，飞羽，风清扬，well，weedge，xiaolin，555 等等三十八位兄弟皆都对编程艺术系列贡献卓著。

最后说一句，读者朋友们中如果是初学编程的话切勿跟风学算法，夯实编程基础才是最重要的。预祝各位元旦快乐。谢谢，本章完。

## 第二十七章：不改变正负数之间相对顺序重新排列数组.时间 $O(N)$ , 空间 $O(1)$

### 前言

在这篇文章：[九月腾讯，创新工场，淘宝等公司最新面试十三题](#)的第 5 题(一个未排序整数数组，有正负数，重新排列使负数排在正数前面，并且要求不改变原来的正负数之间相对顺序)，自从去年九月收录了此题至今，一直未曾看到令人满意的答案，为何呢？

因为一般达不到题目所要求的：时间复杂度  $O(N)$ ,空间  $O(1)$ ，且保证原来正负数之间的相对位置不变。本编程艺术系列第 27 章就来阐述这个问题，若有任何漏洞，欢迎随时不吝指正。谢谢。

#### 重新排列使负数排在正数前面

原题是这样的：

一个未排序整数数组，有正负数，重新排列使负数排在正数前面，并且要求不改变原来的正负数之间相对顺序。

比如：**input: 1,7,-5,9,-12,15 , ans: -5,-12,1,7,9,15** 。且要求时间复杂度  $O(N)$ ,  
空间  $O(1)$  。

OK，下面咱们就来试着一步一步解这道题，如下 5 种思路（从复杂度  $O(N^2)$  到  $O(N*\log N)$ ，从不符合题目条件到一步步趋近于条件）：

1. 最简单的，如果不考虑时间复杂度，最简单的思路是从头扫描这个数组，每碰到一个正数时，拿出这个数字，并把位于这个数字后面的所有数字往前挪动一位。挪完之后在数组的末尾有一个空位，这时把该正数放入这个空位。由于碰到一个正，需要移动  $O(n)$  个数字，因此总的时间复杂度是  $O(n^2)$ 。
2. 既然题目要求的是把负数放在数组的前半部分，正数放在数组的后半部分，因此所有的负数应该位于正数的前面。也就是说我们在扫描这个数组的时候，如果发现有正数出现在负数的前面，我们可以交换他们的顺序，交换之后就符合要求了。  
因此我们可以维护两个指针，第一个指针初始化为数组的第一个数字，它只向后移动；第二个指针初始化为数组的最后一个数字，它只向前移动。在两个指针相遇之

前，第一个指针总是位于第二个指针的前面。如果第一个指针指向的数字是正而第二个指针指向的数字是负数，我们就交换这两个数字。

但遗憾的是上述方法改变了原来正负数之间的相对顺序。所以，咱们得另寻良策。

3. 首先，定义这样一个过程为“翻转”：  $(a_1, a_2, \dots, a_m, b_1, b_2, \dots, b_n) \rightarrow (b_1, b_2, \dots, b_n, a_1, a_2, \dots, a_m)$ 。其次，对于待处理的未排序整数数组，从头到尾进行扫描，寻找(正正...正负...负负)串；每找到这样一个串，则计数器加 1；若计数为奇数，则对当前串做一个“翻转”；反复扫描，直到再也找不到(正正...正负...负负)串。

此思路来自朋友胡果果，空间复杂度虽为  $O(1)$ ，但其时间复杂度  $O(N * \log N)$ 。更多具体细节参看原文：

<http://qing.weibo.com/1570303725/5d98e0ed33000hcb.html>。故，不符合题目要求，继续寻找。

4. 我们可以这样，设置一个起始点  $j$ ，一个翻转点  $k$ ，一个终止点  $L$ ，从右侧起，起始点在第一个出现的负数，翻转点在起始点后第一个出现的正数，终止点在翻转点后出现的第一个负数(或结束)。

如果无翻转点，则不操作，如果有翻转点，则待终止点出现后，做翻转，即  $ab \Rightarrow ba$  这样的操作。翻转后，负数串一定在左侧，然后从负数串的右侧开始记录起始点，继续往下找下一个翻转点。

例子中的就是(下划线代表要交换顺序的两个数字)：

1, 7, -5, 9, -12, 15

第一次翻转: 1, 7, -5, -12, 9, 15  $\Rightarrow$  1, -12, -5, 7, 9, 15

第二次翻转: -5, -12, 1, 7, 9, 15

此思路 2 果真解决了么？NO，用下面这个例子试一下，我们就能立马看出了漏洞：

1, 7, -5, -6, 9, -12, 15 (此种情况未能处理)

1 7 -5 -6 -12 9 15

1 -12 -5 -6 7 9 15

-6 -12 -5 1 7 9 15 (此时，正负数之间的相对顺序已经改变，本应该是-5, -6, -12, 而现在是-6 -12 -5)

5. 看来这个问题的确有点麻烦，不过我们最终貌似还是找到了另外一种解决办法，正如朋友超越神所说的：从后往前扫描，遇到负数，开始记录负数区间，然后遇到正数，记录前面的正数区间，然后把整个负数区间与前面的正数区间进行交换，交换区间但保序的算法类似  $(a, bc \rightarrow bc, a)$  的字符串原地翻转算法。交换完之后要继续

向前一直扫描下去，每次碰到负数区间在正数区间后面，就翻转区间。下面，将详细阐述此思路 4。

## 思路 5 之区间翻转

其实上述思路 5 非常简单，既然单个翻转无法解决问题，那么咱们可以区间翻转阿。什么叫区间翻转？不知读者朋友们是否还记得本 blog 之前曾经整理过这样一道题，微软面试 100 题系列第 10 题，如下：

10、翻转句子中单词的顺序。

题目：输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。句子中单词以空格符隔开。为简单起见，标点符号和普通字母一样处理。例如输入“*I am a student.*”，则输出“**student. a am I**”。而此题可以在  $O(N)$  的时间复杂度内解决：

由于本题需要翻转句子，我们先颠倒句子中的所有字符。这时，不但翻转了句子中单词的顺序，而且单词内字符也被翻转了。我们再颠倒每个单词内的字符。由于单词内的字符被翻转两次，因此顺序仍然和输入时的顺序保持一致。

以上的输入为例：翻转“`I am a student.`”中所有字符得到“`.tneduts a ma I`”，再翻转每个单词中字符的顺序得到“`students. a am I`”，正是符合要求的输出(编码实现，可以参看此文：<http://zhedahht.blog.163.com/blog/static/254111742007289205219/>)。

对的，上述思路 3 就是这个意思，单词翻转便相当于区间翻转，既如此，咱们来验证下上述思路 2 中那个测试用例，如下：

$$1, 7, -5, -6, \underline{9}, \underline{-12}, 15$$

17 -5 -6 -12 9 15

-12 -6 -5 7 19 15 (借用单词翻转的方法, 先逐个数字翻转, 后正负数整体原地翻转)

-5 -6 -12 1 7 9 15

### 思路 5 再次被质疑

但是，我还想再问，问题至此被解决了么？真的被 KO 了么？NO，咱们来看这样一种情况，正如威士忌所说：

看看这个数据，`+-+-+-+-----+`，假如  $N_{minus}$  等于  $n/2$ ，由于前面都是`+-+-+-`，区间交换需要  $n/2/2 = n/4$  次，每次交换是  $T(2*(N_{minus} + N_{plus})) \geq T(n)$ ， $n/4 * T(n) = T(n * n/4) = O(n^2)$ 。

还有一种更坏的情况，就是 $+-+-+-+-----+$ 这种数据可能，后面一大堆的负数，前面正负交替。所以，咱们的美梦再次破灭，路漫漫其修远兮，此题仍然未找到一个完全解决了的方案。

## 公开征集本题思路

下面公开征集解决思路：如果你能想到完全满足和符合题目三个条件的思路（不改变相对顺序&时间  $O(N)$ &空间  $O(1)$ ），欢迎随时在本文下留言或评论，或者发至我邮箱：[zhoulei0907@yahoo.cn](mailto:zhoulei0907@yahoo.cn)。

如果验证完全正确属实，或者要么你就证明在那三个条件下此题无解，当然，你若提出了本文内没有的思路，虽然严格论证下可能并不符合题目三个条件，但我将依然邀请您作为听众来参加读书会第2期(到场嘉宾可能包括为 [pongba](#), [xlvector](#), [penny](#), 以届时现场为准)，再或者，你能指出本文文末 **updated** 部分：本题思路征集中的那些思路的不符要求与错误，也行，**限前10位**，额满为止。

读书会第2期主讲人初步确定：最新消息，[@刘未鹏pongba](#) 与[项亮@xlvector\\_Hulu](#) (推荐系统实践作者)，初步定为将于4月中旬举办的读书会第2期的主讲分享人，[@梁斌penny](#) 亦可能会到场助阵，相信与第1期相比，不逊精彩，令人期待，:-)。安。

具体时间、场地待后续确定(微博上和本 blog 内会届时通知)。

## **updated：本题思路集中之一解一点评**

关于本题不改变正负数相对顺序重新排列数组，陆陆续续有不少朋友或发来了邮件，或在本文评论下提供了他们自己的思路或解法，思维理性碰撞，共同享受思考的乐趣，我觉挺有意思，精选其中一些解答贴出来，让大家评判、讨论，如下：

**第1解：** from Muqi: Hi July,

很高兴看到你的问题，真的很有意思！

我这里想到一种解法，主要的思路是通过改变数字内容来实现保留数字之间相对的顺序：

比方说数列 3 4 -1 -3 5 2 -7 6 1

那负数相对顺序是： -1 -3 -7， 正数是 3 4 5 2 6 1

我们可以做变形：

负数成为 -1.1 -2.3, -3.7 (整数部分为相对顺序，小数部分为原来的数字)

同理 正数为： 1.3 2.4 3.5 4.2 5.6 7.1

现在数组变成： 1.3 2.4 -1.1 -2.3 3.5 4.2 -3.7 4.6 5.1

接下去 先通过置换把所有负数排到前面：具体方法为从前往后扫描数组，每碰到一个负数 就和数组最前面的正数交换， 结果如下：

-1.1 -2.3 -3.7 2.4 3.5 4.2 1.3 4.6 5.1

可以看到负数部分已经完成题目要求（只需要把整数部分去掉即可），接下去对于正数数列 2.4 3.5 4.2 1.3 4.6 5.1， 也用类似的方法还原先前的顺序：具体方法为一次遍历每个数字，检查其整数部分是否与其所在的位置相同，如不相同，将该数字与位置为该数字整数部分的交换，比如说 2.4 整数部分为 2，但是现在位于数列第一位，所以与位于第二位的 3.5 交换，得到：3.5 2.4 4.2 1.3 4.6 5.1（最多只需要  $O(n)$  因为每次交换都保证一个数字回到原来的位置，而总共有  $n$  个数字），最后和前面负数的处理相同，即去掉整数部分(+1， 邀请来参加读书会第 2 期)。

**点评：**但此方法在本文评论下马上有人指出：不过，整数变成浮点数，存储空间要扩大一倍，跟申请一个大小为  $n$  的数组一个道理，空间复杂度  $O(N)$  不符要求。更多请看本文评论下第 18 楼。(zj060607 & topskycen, +2)。

**第 2 解： form 立宋(+7)：**

July 巨巨，

由于在 csdn 上那贴删改留言次数有点多，csdn 不让留言了，就发邮件给您吧。应该是最终稿了。

稍微改动下 Muqi 的方法，可以得到平均时间  $O(n)$ ，最坏时间  $O(n^2)$ ，空间复杂度  $O(1)$  的。当把负数放到数列前半部分操作时，这个负数是和前面的一个正数交换的。交换过后，把这个正数变成他的相反数(5 变成-5 这种)。那么当第一轮把负数放到前面过后，剩下的部分又形成了一个相同的子问题。当然，后面几轮把负数放到前面后，得把他们重新恢复成相应的正数。

还是用 3 4 -1 -3 5 2 -7 6 1 为例子：

第一轮： -1 -3 -7 [-3 -4 2 -5 6 1].

第二轮： -1 -3 -7 -(-3) -(-4) -(-5) [-2 6 1].

第三轮： 1 3 7 3 4 5 2 6 1

最终添上-号， -1 -3 -7 3 4 5 2 6 1

平均时间复杂度（假设数组是随机的）：

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

如果遇到++++...+-这种情况，就会导致最坏的时间复杂度。

这也不算是完美的解法。有点怀疑完美的解是不存在的，但不知道怎么证明。

谢谢，

mouris|

**点评:** from litaoye: 我的想法(见下文之综合点评)也许同上述解法 2 类似，但我确实没看明白解法 2 的操作过程，并且我认为解法 2 十有八九是错的。举个例子来说，如

1,2,-4,-5,3,-6

-4,2,1,-5,3,-6

-4,-5,1,-2,3,-6

-4,-5,-6,-2,3,-1

这样的话-2 同-1 的顺序就乱了。更多请看本文评论下第 29 楼。

**第 3 解:** jiangbin00cn 在其 blog 中提出了一种新的思路：假设全体数据为  $n$  个，正数  $m$  个分别映射到  $1--m$ ，这  $m$  个数是分散分布在空间  $n$  中，利用桶排序使得其排列在  $n-m-n$  中，这个过程用到了桶排序的思想，只不过每个桶中只有一个元素。具体步骤如下：

(1) 桶排序能够在 时间  $O(N)$ ，空间  $O(1)$  实现，那么能否利用桶排序解决该问题，即如何将该问题转换为桶排序问题

(2) 通过可逆的修改元素使得数组满足桶排序要求

(3) 利用桶排序实现

(4) 恢复元素

假设原数组中的全体正数按顺序依次为： $a[0], \dots, a[n]$

$(a[0], a[1], \dots, a[n]) = f(x) \Rightarrow (b[0], b[1], \dots, b[n]) = g(x) \Rightarrow (0, 1, \dots, n) \Rightarrow$  桶排序

原始正数(可能相同) (修改为全不相同正数)

$(0, 1, \dots, n) = g'(x) \Rightarrow (b[0], b[1], \dots, b[n]) = f'(x) \Rightarrow (a[0], a[1], \dots, a[n])$

可逆运算恢复数据

可逆运算恢复数据

结论：

由于桶排序能够在 时间  $O(N)$ ，空间  $O(1)$  实现，若可逆函数  $f(x), g(x)$  能够在 时间  $O(N)$ ，空间  $O(1)$  中找到并实现，那么就能够解决该问题(+3)。具体代码实现，请参见原文：

<http://blog.csdn.net/jiangbin00cn/article/details/7331387>。然而，本文评论下第 64 楼有读者反应：这段桶代码段有问题，当查询的数字比如是 1, 7, -5, -6, 9, -12, 15 这样没问题 但是 最后一个数字是负数 这个程序就死循环了 1, 7, -5, -6, 9, -12, 15 -16。

**点评:** from 威士忌(+5), jiangbin00cn 和 Muqi 的方法都很取巧。其实他们的方法都是压缩了整数值域或者扩大值域来保存附加信息，虽然符合时间  $O(N)$  和空间  $O(1)$  的要求，但是并不适用所有 int 值。

这些方法的思路其实很简单，比如：

```
num[] = {1,7,-5,9,-12,15};
```

```
pos[] = {2,3,0,4,1,5};
```

pos 的计算扫描 2 遍 num 数组即可，有了 pos 数组当然排序不成问题。

关键解决 pos 空间问题时，两位做法分别是，Muqi 保存到 double 浮点域，jiangbin00cn 是利用进制方法保存到 int 高位（其实根本不需桶排了），更明显的做法就是 flag(num[i])\*pos[i]\*1000+num[i] 转换为 3007,-0005,-1012。

很高兴看到如此让人眼前一亮的方法，但是仔细想想的话，就觉得还是不符合要求。

#### 第 4 解：from topskychen & acmerfight(+6)：

首先明确题目的题意要求空间复杂度是 O(1), 我的理解就是只能有一个空间来存储数据，其他的任何临时变量都不能出现，包括循环变量和临时开辟的空间（例如数字交换时）。

下边我的解法是在 允许自己输入数据，可以利用数组大小 n 的情况下产生的，只包含一个额外的变量。

用 pos 记录正数的最最左位置减一， a[pos] 记录负数最右的位置加一  
基本步骤：

- 1 让 pos 代表最后一个数据的位置
- 2 然后输入一个数据存储在最后的位置 a[pos]
- 3 如果输入的数据 a[pos] 是正数，我们就让 pos = pos - 1；如果输入的是负数就把这个负数挪到前边，让 a[pos]-1 来记录负数最右的位置。
- 4 发生相应交换

代码在本文评论下第 28 楼。

## 综合点评

1. **from 威士忌**，感觉最近的几种解法越来越倒退了，还不如之前 nlogn 的来的有价值。
2. **from litaoye**: 只想到了  $n \cdot \log(n)$ ,  $O(1)$  的方法。双指针分别指向头和尾，头指针找到的正数同尾指针找到的负数交换，直到 2 个指针相遇。交换过程中将所有交换元素 \* -1，也就是正数变负数，负数变正数。此时被换到尾部的正数 (\*-1 后已经变为负数)，顺序正好倒过来了，把这部分反转一下。整个过程  $O(n)$ ，把原问题转化为两个规模为  $n/2$  的子问题。因此根据主定理，整个过程应当是  $n \log(n)$  的，即最坏情况下是  $n^2$  的，不过平均情况下也只是  $n \log(n)$  的，达不到  $O(n)$ 。用迭代的方法写，应该可以做到  $O(1)$  (用递归，空间复杂度就是  $\log(n)$  了)，感觉这个问题很难找到  $O(n), O(1)$  的解法，类似的问题有完美洗牌问题，LZ 可以看一下，解法比较复杂，是通过原根构造置换群来解的。本来还有个原地归并的思路，后来发现有问题，没有继续深入。
3. **from sbwwkmyd**: 除非也能找到划分固定环的方法，一直没找到办法将原根的特性应用到这个问题上，完美洗牌问题 是这个问题的一个很小的子集。这个问题应该无解。

4. **from July**: 有 friends 反应, 算法导论第 8 章线性时间排序思考题 8-2: 以线性时间原地置换排序, 是此题原型。说运行时间为  $O(n)$ 、稳定、不使用额外空间原地排序, 这 3 个条件中, 三者只能满足其二, 由此推出第 27 章此题无解? 果真如此么? 如何证明? 此题作为面试题, 能当场 K 掉 99% 的面试者/面试官。

也有朋友反应, 根据算导第 8 章中定理 8.1: 任意一个比较排序算法在最坏情况下, 都需要  $n \lg n$  次比较。即给定  $n$  个不同的输入元素, 对于任何确定或随机的比较排序算法, 其期望运行时间都有下界  $O(n \lg n)$ 。由此推出此题无解。但他们忽略了: 不一定非要排序非要比较。

也就是说, 尽管:

1. 插入.归并.堆.快速排序皆是基于比较排序, 且除归并排序外, 皆是原地排序算法。
2. 堆/归并排序运行时间上界皆为  **$O(n \lg n)$** 。
3. 计数排序非基于比较, 非原地排序, 但稳定, 是基数排序算法的一个子过程。
4. 计数/基数等非原地(需借助外部空间)排序, 空间换时间。

但本题统统与这些无关, 因为追根究底, 本题实质性上只是一个排列, 重新组合问题, 与排序无关。

更多还可参考此论文: 《STABLE MINIMUM SPACE PARTITIONING IN LINEAR TIME》。待后续验证。

## 第二十八~二十九章：最大连续乘积子串、字符串编辑距离

### 前言

时间转瞬即逝，一转眼，又有 4 个多月没来更新 blog 了，过去 4 个月都在干嘛呢？对的，今 2013 年元旦和朋友利用业余时间一起搭了个方便朋友们找工作的编程面试算法论坛：[为学论坛 `http://www.51weixue.com/`](http://www.51weixue.com/)（因为后边的 hero，论坛已逐步废弃）。最近则开始负责一款在线编程挑战平台：英雄会 <http://hero.pongo.cn/>，包括其产品运营，出题审题，写代码测试，制定比赛规则等等。

前几天跟百度的几个朋友线下闲聊，听他们说，百度校招群内的不少朋友在找工作的时候都看过我的 blog，一听当即便激起了自己重写此 blog 的欲望，恰巧眼下阳春三月（虽说已是 3 月，奇妙的是，前两天北京还下了一场大雪），又是找工作的季节（相对于每年的 9 月份来说，3 月则是一个小高潮），那就从继续更新专为 IT 人员找工作时准备笔试面试的程序员编程艺术系列开始吧。

再者从去年 4 月份上传的编程艺术前 27 章的 PDF 文档的 1.3 万下载量来看  
[http://download.csdn.net/detail/v\\_july\\_v/4256339](http://download.csdn.net/detail/v_july_v/4256339)，此系列确确实实帮助了成千上万的人。

Yeah，本文讲两个问题，

- 第二十八章、最大连续乘积子串，
- 第二十九章、字符串编辑距离，

这两个问题皆是各大 IT 公司最喜欢出的笔试面试题，比如说前者是小米 2013 年校招笔试原题，而后者则更是反复出现，如去年 9 月 26 日百度一二面试题，10 月 9 日腾讯面试题第 1 小题，10 月 13 日百度 2013 校招北京站笔试题第二 大道题第 3 小题，及去年 10 月 15 日 2013 年 Google 校招笔试最后一道大题皆是考察的这个字符串编辑距离问题。

OK，欢迎朋友们在这篇文章下参与讨论，如果在线编译自己的代码（编程语言任选 C/C++/Java/C#），可以上英雄会提交你的代码，有任何问题，欢迎随时不吝批评或指正，感谢。

## 第二十八章、最大连续乘积子串

题目描述：

给一个浮点数序列，取最大乘积连续子串的值，例如 -2.5, 4, 0, 3, 0.5, 8, -1，则取出的最大乘积连续子串为 3, 0.5, 8。也就是说，上述数组中，3 0.5 8 这 3 个数的乘积  $3 \times 0.5 \times 8 = 12$  是最大的，而且是连续的。

提醒：此最大乘积连续子串与最大乘积子序列不同，请勿混淆，前者子串要求连续，后者子序列不要求连续。也就是说：最长公共子串（Longest Common Substring）和最长公共子序列（Longest Common Subsequence, LCS）的区别：

- 子串（Substring）是串的一个连续的部分，
- 子序列（Subsequence）则是从不改变序列的顺序，而从序列中去掉任意的元素而获得的新序列；

更简略地说，前者（子串）的字符的位置必须连续，后者（子序列 LCS）则不必。比如字符串“acdfg”同“akdfc”的最长公共子串为“df”，而它们的最长公共子序列 LCS 是“adf”，LCS 可以使用动态规划法解决。

解答：

解法一、穷举，所有的计算组合：

或许，读者初看此题，自然会想到最大乘积子序列问题类似于最大子数组和问题：

[http://blog.csdn.net/v\\_JULY\\_v/article/details/6444021](http://blog.csdn.net/v_JULY_v/article/details/6444021)，可能立马会想到用最简单粗暴的方式：两个 for 循环直接轮询。

```
double max=0;
double start=0;
double end=0;
for (int i=0;i<num;i++) {
    double x=arrs[i];
    for (int j = i+1; j < num; j++) {
        x*=arrs[j];
        if(x>max){
            max=x;
            start=arrs[i];
            end=arrs[j];
        }
    }
}
```

```
    }  
}
```

**解法二**、虽说类似于最大子数组和问题，但实际上具体处理起来诸多不同。为什么呢，因为乘积子序列中有正有负也还可能有 0。我们可以把问题简化成这样：数组中找一个子序列，使得它的乘积最大；同时找一个子序列，使得它的乘积最小（负数的情况）。因为虽然我们只要一个最大积，但由于负数的存在，我们同时找这两个乘积做起来反而方便。也就是说，不但记录最大乘积，也要记录最小乘积。So，我们让

- `maxCurrent` 表示当前最大乘积的 `candidate`,
- `minCurrent` 反之，表示当前最小乘积的 `candidate`,
- 而 `maxProduct` 则记录到目前为止所有最大乘积 `candidates` 的最大值。

(以上用 `candidate` 这个词是因为只是可能成为新一轮的最大/最小乘积)

由于空集的乘积定义为 1，在搜索数组前，`maxCurrent`, `minCurrent`, `maxProduct` 都赋为 1。

假设在任何时刻你已经有了 `maxCurrent` 和 `minCurrent` 这两个最大/最小乘积的 `candidates`，新读入数组的元素 `x(i)` 后，新的最大乘积 `candidate` 只可能是 `maxCurrent` 或者 `minCurrent` 与 `x(i)` 的乘积中的较大者，如果 `x(i)<0` 导致 `maxCurrent<minCurrent`，需要交换这两个 `candidates` 的值。

当任何时候 `maxCurrent<1`，由于 1（空集）是比 `maxCurrent` 更好的 `candidate`，所以更新 `maxCurrent` 为 1，类似的可以更新 `minCurrent`。任何时候 `maxCurrent` 如果比最好的 `maxProduct` 大，更新 `maxProduct`。

代码一：

```
template <typename Comparable>  
Comparable maxprod( const vector<Comparable>&v)  
{  
    int i;  
    Comparable maxProduct = 1;  
    Comparable minProduct = 1;  
    Comparable maxCurrent = 1;  
    Comparable minCurrent = 1;  
    //Comparable t;  
  
    for( i=0; i< v.size(); i++)  
    {  
        maxCurrent *= v[i];  
        minCurrent *= v[i];  
        if(maxCurrent > maxProduct)  
            maxProduct = maxCurrent;  
        if(minCurrent > maxProduct)
```

```

        maxProduct = minCurrent;
        if(maxCurrent < minProduct)
            minProduct = maxCurrent;
        if(minCurrent < minProduct)
            minProduct = minCurrent;
        if(minCurrent > maxCurrent)
            swap(maxCurrent,minCurrent);
        if(maxCurrent<1)
            maxCurrent = 1;
        //if(minCurrent>1)
        //    minCurrent =1;
    }
    return maxProduct;
}

```

代码二：思路，记录以第  $i$  个结尾的最大乘积  $M$  和最小乘积  $m$ ，并且记录这两个区间的起点（终点都是  $i$ ），不断更新，来源 <http://www.51weixue.com/thread-246-1-1.html>:

```

pair<int,int> maxproduct(double *f,int n) { //返回最大乘积的起点终点
int R = 0, r = 0; //最大最小区间的 起点
pair<int,int> ret = make_pair(0, 0); //最大 最小的区间下标
double M = f[0], m = f[0], answer = f[0]; // 最大 最小值
for (int i = 1; i < n; ++i) {
    double t0 = f[i] * M, t1 = f[i] * m;
    if (t0 > t1) {
        M = t0;
        m = t1;
    }
    else {
        int t = R;
        R = r;
        r = t;
        M = t1;
        m = t0;
    }
    if (M < f[i]) {
        M = f[i];
        R = i;
    }
    if (m > f[i]) {
        m = f[i];
        r = i;
    }
}

```

```

        if (answer < M) {
            answer = M;
            ret = make_pair(R, i);
        }

    }
    return ret;
}

```

### 解法三、

本题除了上述类似最大子数组和的解法，也可以直接用动态规划求解（其实，上述的解法一本质上也是动态规划，只是解题所表现出来的具体形式与接下来的解法二不同罢了。这个不同就在于下面的解法二会写出动态规划问题中经典常见的 DP 方程，而解法一是直接求解）。具体解法如下：

假设数组为  $a[]$ ，直接利用动归来求解，考虑到可能存在负数的情况，我们用  $\text{Max}$  来表示以  $a$  结尾的最大连续子串的乘积值，用  $\text{Min}$  表示以  $a$  结尾的最小的子串的乘积值，那么状态转移方程为：

$\text{Max}=\max\{a, \text{Max}[i-1]*a, \text{Min}[i-1]*a\};$

$\text{Min}=\min\{a, \text{Max}[i-1]*a, \text{Min}[i-1]*a\};$

初始状态为  $\text{Max}[1]=\text{Min}[1]=a[1]$ 。

**C/C++** 代码一，很简洁的一小段代码：

```

double func(double *a,const int n)
{
    double *maxA = new double[n];
    double *minA = new double[n];
    maxA[0] = minA[0] = a[0];
    double value = maxA[0];
    for(int i = 1 ; i < n ; ++i)
    {
        maxA[i] = max(max(a[i],maxA[i-1]*a[i]),minA[i-1]*a[i]);
        minA[i] = min(min(a[i],maxA[i-1]*a[i]),minA[i-1]*a[i]);
        value=max(value,maxA[i]);
    }
    return value;
}

```

**C/C++** 代码二：

```

/*
给定一个浮点数数组，有正有负数，0，正数组成，数组下标从 1 算起

```

求最大连续子序列乘积，并输出这个序列，如果最大子序列乘积为负数，那么就输出-1  
 用 Max[i] 表示以 a[i] 结尾乘积最大的连续子序列  
 用 Min[i] 表示以 a[i] 结尾乘积最小的连续子序列 因为有复数，所以保存这个是必须的

```

/*
void longest_multiple(double *a,int n){
    double *Min=new double[n+1]();
    double *Max=new double[n+1]();
    double *p=new double[n+1]();
    //初始化
    for(int i=0;i<=n;i++){
        p[i]=-1;
    }
    Min[1]=a[1];
    Max[1]=a[1];
    double max_val=Max[1];
    for(int i=2;i<=n;i++){
        Max[i]=max(Max[i-1]*a[i],Min[i-1]*a[i],a[i]);
        Min[i]=min(Max[i-1]*a[i],Min[i-1]*a[i],a[i]);
        if(max_val<Max[i])
            max_val=Max[i];
    }
    if(max_val<0)
        printf("%d",-1);
    else
        printf("%d",max_val);
    //内存释放
    delete [] Max;
    delete [] Min;
}

```

C#版完整代码(代码来自参加英雄会在线编程挑战之 1019、最大乘积连续子串：

<http://hero.pongo.cn/Question/Details?ID=19&ExamID=19> 的在线提交代码的用户)：

```

//答题英雄: danielqkj
using System;
public class Test
{
    void Max(double a, double b, double c)
    {
        double d = (a>b)?a:b;
        return (d>c)?d:c;
    }

    void Min(double a, double b, double c)

```

```

{
    double d = (a>b)?b:a;
    return (d>c)?c:d;
}

public static void Main()
{
    int n = Int32.parse(Console.readline());
    double[] a = new double[n];
    double maxvalue = a[0];
    double[] max = new double[n];
    double[] min = new double[n];
    double start, end;

    String[] s = Console.readline().split(' ');
    for (int i = 0; i < n; i++)
    {
        a[i] = Double.parseDouble(s[i])
    }

    max[0] = a[0];
    min[0] = a[0];
    start = 0, end = 0;

    for (int i = 1; i < n; i++)
    {
        max[i]=Max(a[i], a[i]*max[i-1], a[i]*min[i-1]);
        min[i]=Min(a[i], a[i]*max[i-1], a[i]*min[i-1]);

        if (max[i] > maxvalue)
        {
            maxvalue = max[i];
            end = i;
        }
    }

    double mmm = maxvalue;
    while ( (mmm - 0.0) > 0.00001 )
    {
        start = end;
        mmm = mmm / a[start];
    }
}

```

```

        Console.WriteLine(a[start] + " " + a[end] + " " + maxvalue);

    }
}

```

## 变种

此外，此题还有另外的一个变种形式，即给定一个长度为  $N$  的整数数组，只允许用乘法，不能用除法，计算任意  $(N-1)$  个数的组合中乘积最大的一组，并写出算法的时间复杂度。

我们可以把所有可能的  $(N-1)$  个数的组合找出来，分别计算它们的乘积，并比较大小。由于总共有  $N$  个  $(N-1)$  个数的组合，总的时间复杂度为  $O(N^2)$ ，显然这不是最好的解法。

OK，以下解答来自编程之美

### 解法 1

在计算机科学中，时间和空间往往是一对矛盾体，不过，这里有一个折中方法。可以通过“空间换时间”或“时间换空间”的策略来达到优化某一方面的目的。在这里，是否可以通过“空间换时间”来降低时间复杂度呢？<sup>4</sup>

计算  $(N-1)$  个数的组合乘积，假设第  $i$  个  $(0 \leq i \leq N-1)$  元素被排除在乘积之外（如图 2-16 所示）。<sup>4</sup>

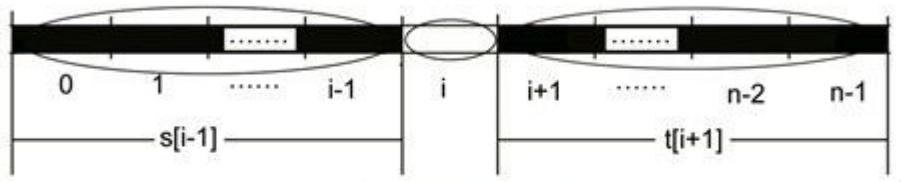


图 2-16 组合示意图

设  $array[]$  为初始数组， $s[i]$  表示数组前  $i$  个元素的乘积  $s[i] = \prod_{j=0}^{i-1} array[j]$ ，其中  $1 \leq i \leq N$ ， $s[0] = 1$ （边界条件），那么  $s[i] = s[i-1] \times array[i-1]$ ，其中  $i = 1, 2, \dots, N-1, N$ ；<sup>4</sup>

设  $t[i]$  表示数组后  $(N-i)$  个元素的乘积  $t[i] = \prod_{j=i}^{N-1} array[j]$ ，其中  $1 \leq i \leq N$ ， $t[N+1] = 1$ （边界条件），那么  $t[i] = t[i+1] \times array[i]$ ，其中  $i = 1, 2, \dots, N-1, N$ ；<sup>4</sup>

那么设  $p[i]$  为数组除第  $i$  个元素外，其他  $N-1$  个元素的乘积，即有：<sup>4</sup>

$$p[i] = s[i-1] \times t[i+1]$$

由于只需要从头至尾，和从尾至头扫描数组两次即可得到数组  $s[]$  和  $t[]$ ，进而线性时间可以得到  $p[]$ 。所以，很容易就可以得到  $p[]$  的最大值（只需遍历  $p[]$  一次）。总的时间复杂度等于计算数组  $s[]$ 、 $t[]$ 、 $p[]$  的时间复杂度加上查找  $p[]$  最大值的时间复杂度等于  $O(N)$ 。<sup>4</sup>

### 解法 2

此外，还可以通过分析，进一步减少解答问题的计算量。假设  $N$  个整数的乘积为  $P$ ，针对  $P$  的正负性进行如下分析（其中， $AN-1$  表示  $N-1$  个数的组合， $PN-1$  表示  $N-1$  个数的组

合的乘积)。

#### 1. $P$ 为 0

那么，数组中至少包含有一个 0。假设除去一个 0 之外，其他  $N-1$  个数的乘积为  $Q$ ，根据  $Q$  的正负性进行讨论：

$Q$  为 0

说明数组中至少有两个 0，那么  $N-1$  个数的乘积只能为 0，返回 0；

$Q$  为正数

返回  $Q$ ，因为如果以 0 替换此时  $A_{N-1}$  中的任一个数，所得到的  $P_{N-1}$  为 0，必然小于  $Q$ ；

$Q$  为负数

如果以 0 替换此时  $A_{N-1}$  中的任一个数，所得到的  $P_{N-1}$  为 0，大于  $Q$ ，乘积最大值为 0。

#### 2. $P$ 为负数

根据“负负得正”的乘法性质，自然想到从  $N$  个整数中去掉一个负数，使得  $P_{N-1}$  为一个正数。而要使这个正数最大，这个被去掉的负数的绝对值必须是数组中最小的。我们只需要扫描一遍数组，把绝对值最小的负数给去掉就可以了。

#### 3. $P$ 为正数

类似地，如果数组中存在正数值，那么应该去掉最小的正数值，否则去掉绝对值最大的负数值。

上面的解法采用了直接求  $N$  个整数的乘积  $P$ ，进而判断  $P$  的正负性的办法，但是直接求乘积在编译环境下往往会有溢出的危险（这也就是本题要求不使用除法的潜在用意），事实上可做一个小的转变，不需要直接求乘积，而是求出数组中正数 (+)、负数 (-) 和 0 的个数，从而判断  $P$  的正负性，其余部分与以上面的解法相同。

在时间复杂度方面，由于只需要遍历数组一次，在遍历数组的同时就可得到数组中正数 (+)、负数 (-) 和 0 的个数，以及数组中绝对值最小的正数和负数，时间复杂度为  $O(N)$ 。

## 第二十九章、字符串编辑距离

题目描述：

给定一个源串和目标串，能够对源串进行如下操作：

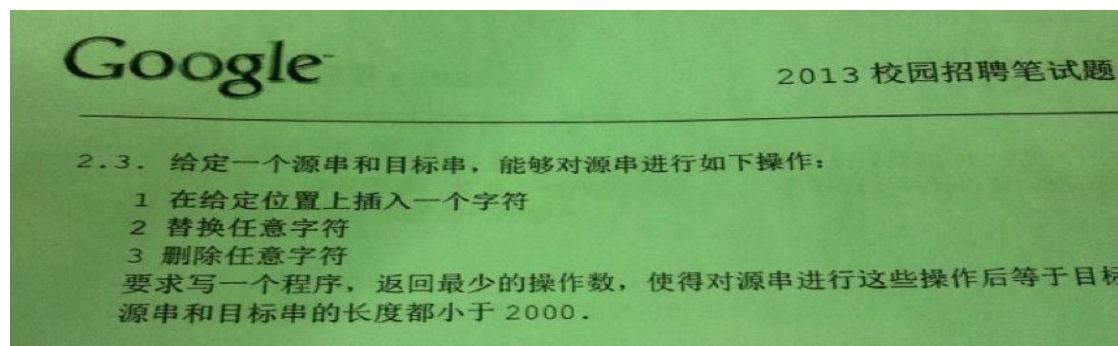
1. 在给定位置上插入一个字符

2. 替换任意字符

3. 删除任意字符

写一个程序，返回最小操作数，使得对源串进行这些操作后等于目标串，源串和目标串的长度都小于 2000。

**提醒：**上文前言中已经说过了，此题反复出现，最近考的最多的是百度和 Google 的笔试面试经常考察。下图则是 2013 年 Google 的校招试题原景重现：



解答：

**解法一、** 此题跟上面的最大连续乘积子串类似，常见的思路是动态规划，下面是简单的 DP 状态方程：

```
//动态规划:  
  
//f[i,j]表示 s[0...i]与 t[0...j]的最小编辑距离。  
f[i,j] = min { f[i-1,j]+1, f[i,j-1]+1, f[i-1,j-1]+(s[i]==t[j]?0:1) }  
  
//分别表示：添加 1 个，删除 1 个，替换 1 个（相同就不用替换）。
```

**解法二、** 本解法来自为学论坛：<http://www.51weixue.com/thread-482-1-1.html>。

编辑距离的定义和计算方法如下：

Given two strings A and B, edit A to B with the minimum number of edit operations:

- a) Replace a letter with another letter
- b) Insert a letter
- c) Delete a letter

E.g.

A = interestingly    \_i\_\_nterestingly

B = bioinformatics    bioinformatics  
                      1011011011001111

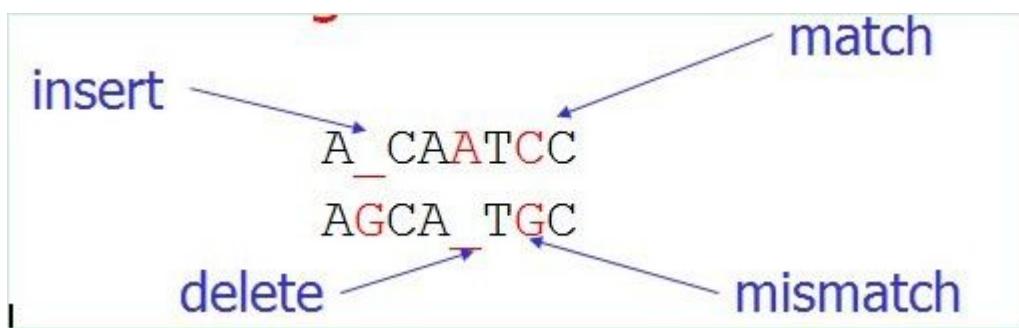
Edit distance = 11

Instead of minimizing the number of edge operations, we can associate a cost function to the operations and minimize the total cost. Such cost is called edit distance. Instead of using string edit, in computational biology, people like to use string alignment. We use similarity function, instead of cost function, to evaluate the goodness of the alignment.

E.g. of similarity function: match – 2, mismatch, insert, delete – -1.

Consider two strings ACAATCC and AGCATGC.

One of their alignment is



In the above alignment, space ('\_') is introduced to both strings. There are 5 matches, 1 mismatch, 1 insert, and 1 delete. The alignment has similarity score 7.

A\_CAAATCC

AGCA\_TGC

Note that the above alignment has the maximum score. Such alignment is called optimal

alignment. String alignment problem tries to find the alignment with the maximum similarity

score! String alignment problem is also called global alignment problem.

Needleman-Wunsch algorithm

Consider two strings  $S[1..n]$  and  $T[1..m]$ . Define  $V(i, j)$  be the score of the optimal alignment

between  $S[1..i]$  and  $T[1..j]$ .

Basis:

$$V(0, 0) = 0$$

$$V(0, j) = V(0, j-1) + d(\_, T[j]): \text{Insert } j \text{ times}$$

$$V(i, 0) = V(i-1, 0) + d(S, \_): \text{Delete } i \text{ times}$$

that is:

$V(i, j) = \max \begin{cases} V(i-1, j-1) + \delta(S[i], T[j]) & \text{Match/mis} \\ V(i-1, j) + \delta(S[i], \_) & \text{Delete} \\ V(i, j-1) + \delta(\_, T[j]) & \text{Ins} \end{cases}$

Example :



	-	A	G	C	A	T	G	C
-	0	-1	-2	-3	-4	-5	-6	-7
A	-1	2	1	0	-1	-2	-3	-4
C	-2	1	1	3	2	1	0	-1
A	-3	0	0	2	5	4	3	2
A	-4	-1	-1	1	4	4	3	2
T	-5	-2	-2	0	3	6	5	4
C	-6	-3	-3	0	2	5	5	7
C	-7	-4	-4	-1	1	4	4	7

下面是代码，测试数据比较少，若有问题请指正：

```

//copyright@ peng_weida
//实现代码如下:
//头文件 StrEditDistance.h
#pragma once
#include <string>
class CStrEditDistance

```

```

{
public:
    CStrEditDistance(std::string& vStrRow, std::string& vStrColumn);
    ~CStrEditDistance(void);
    int getScore() { return m_Score; }
    int getEditDis() { return m_EditDis; }
    void setEditDis(int vDis) { m_EditDis = vDis; }
    void setScore(int vScore) { m_Score = vScore; }
private:
    void process(const std::string& vStrRow, const std::string& vStrColumn);
    int getMaxValue(int a, int b, int c)
    {
        if (a < b){ if (b < c) return c; return b; }
        else { if (b > c) return a; return a < c ? c : a; }
    }
private:
    int m_EditDis;
    int m_Score;
};

//源文件 StrEditDistance.cpp
#include "StrEditDistance.h"
#include <iostream>
#include <iomanip>
#define MATCH 2
#define MISS_MATCH -1
#define INSERT -1
#define DELETE -1
CStrEditDistance::CStrEditDistance(std::string& vStrRow, std::string& vStrColumn)
{
    process(vStrRow, vStrColumn);
}
CStrEditDistance::~CStrEditDistance(void)
{
}
//FUNCTION:
void CStrEditDistance::process(const std::string& vStrRow, const std::string& vStrColumn)
{
    int editDis = 0; //编辑距离
    int row = vStrColumn.length();
    int column = vStrRow.length();
    const int sizeR = row + 1;
    const int sizeC = column + 1;
}

```

```

int **pScore = new int*[sizeR]; //二维指针
for (int i = 0; i <= row; i++)
    pScore = new int[sizeC];

//初始化第一行和第一列
for (int c = 0; c <= column; c++)
    pScore[0][c] = 0 - c;
for (int r = 0; r <= row; r++)
    pScore[r][0] = 0 - r;

//从 v(1,1)开始每列计算
for (int c = 1; c <= column; c++)
{
    for (int r = 1; r <= row; r++)
    {
        //计算 v(i,j)
        int valueMatch;
        if (vStrColumn[r-1] == vStrRow[c-1])
            valueMatch = MATCH;
        else
            valueMatch = MISS_MATCH;
        int A = pScore[r-1][c] + INSERT;
        int B = pScore[r][c-1] + DELETE;
        int C = pScore[r-1][c-1] + valueMatch;
        pScore[r][c] = getMaxValue(A, B, C);
    }
}

//计算编辑距离
int r = row, c = column;
while(r > 0 && c > 0)
{
    if (pScore[r][c]+1 == pScore[r-1][c])      { editDis++; r--; continue; }
    else if (pScore[r][c]+1 == pScore[r][c-1]) { editDis++; c--; continue; }
    else if (pScore[r][c]+1 == pScore[r-1][c-1]) { editDis++; r--; c--; continue; }
    else { r--; c--; }
}
if (r > 0 && c == 0) editDis += r;
else if (c > 0 && r == 0) editDis += c;

std::cout << std::endl;
//-----DEBUG-----//
//打印数据

```

```

for (int i = 0; i <= row; i++)
{
    for (int j = 0; j <= column; j++)
        std::cout << std::setw(2) << pScore[j] << " ";
    std::cout << std::endl;
}
std::cout << std::endl;

//设置编辑距离和得分
setEditDis(editDis);
setScore(pScore[row][column]);

for (int i = 0; i <= row) //释放内存
{
    delete pScore;
    pScore = NULL;
}
delete[] pScore;
}

```

## 类似

上述问题类似于编程之美上的下述一题「以下内容摘自编程之美第 3.3 节」：

许多程序会大量使用字符串。对于不同的字符串，我们希望能够有办法判断其相似程度。我们定义了一套操作方法来把两个不相同的字符串变得相同，具体的操作方法为：

1. 修改一个字符（如把“a”替换为“b”）；
2. 增加一个字符（如把“abdd”变为“aebdd”）；
3. 删一个字符（如把“travelling”变为“traveling”）。

比如，对于“abcdefg”和“abcdef”两个字符串来说，我们认为可以通过增加/减少一个“g”的方式来达到目的。上面的两种方案，都仅需要一次操作。把这个操作所需要的次数定义为两个字符串的距离，而相似度等于“距离+1”的倒数。也就是说，“abcdefg”和“abcdef”的距离为 1，相似度为  $1 / 2 = 0.5$ 。

给定任意两个字符串，你是否能写出一个算法来计算出它们的相似度呢？

不难看出，两个字符串的距离肯定不超过它们的长度之和（我们可以通过删除操作把两个串都转化为空串）。虽然这个结论对结果没有帮助，但至少可以知道，任意两个字符串的距离都是有限的。 ↶

考虑如何才能把这个问题转化成规模较小的同样的问题。如果有两个串  $A = xabcdae$  和  $B = xfdfa$ ，它们的第一个字符是相同的，只要计算  $A[2, \dots, 7] = abcdae$  和  $B[2, \dots, 5] = fdfa$  的距离就可以了。但是如果两个串的第一个字符不相同，那么可以进行如下的操作（ $\text{len}_A$  和  $\text{len}_B$  分别是  $A$  串和  $B$  串的长度）。 ↶

1. 删除  $A$  串的第一个字符，然后计算  $A[2, \dots, \text{len}_A]$  和  $B[1, \dots, \text{len}_B]$  的距离。 ↶
2. 删除  $B$  串的第一个字符，然后计算  $A[1, \dots, \text{len}_A]$  和  $B[2, \dots, \text{len}_B]$  的距离。 ↶
3. 修改  $A$  串的第一个字符为  $B$  串的第一个字符，然后计算  $A[2, \dots, \text{len}_A]$  和  $B[2, \dots, \text{len}_B]$  的距离。 ↶
4. 修改  $B$  串的第一个字符为  $A$  串的第一个字符，然后计算  $A[2, \dots, \text{len}_A]$  和  $B[2, \dots, \text{len}_B]$  的距离。 ↶
5. 增加  $B$  串的第一个字符到  $A$  串的第一个字符之前，然后计算  $A[1, \dots, \text{len}_A]$  和  $B[2, \dots, \text{len}_B]$  的距离。 ↶
6. 增加  $A$  串的第一个字符到  $B$  串的第一个字符之前，然后计算  $A[2, \dots, \text{len}_A]$  和  $B[1, \dots, \text{len}_B]$  的距离。 ↶

在这个题目中，我们并不在乎两个字符串变得相等之后的字符串是怎样的。所以，可以将上面 6 个操作合并为： ↶

1. 一步操作之后，再将  $A[2, \dots, \text{len}_A]$  和  $B[1, \dots, \text{len}_B]$  变成相同字符串； ↶
2. 一步操作之后，再将  $A[1, \dots, \text{len}_A]$  和  $B[2, \dots, \text{len}_B]$  变成相同字符串； ↶
3. 一步操作之后，再将  $A[2, \dots, \text{len}_A]$  和  $B[2, \dots, \text{len}_B]$  变成相同字符串。 ↶

这样，很快就可以完成一个递归程序，如下所示：

```
Int CalculateStringDistance(string strA, int pABegin, int pAEnd,
    string strB, int pBBegin, int pPEnd)
{
    if(pABegin > pAEnd)
    {
        if(pBBegin > pPEnd)
            return 0;
        else

            return pPEnd - pBBegin + 1;
    }

    if(pBBegin > pPEnd)
    {
        if(pABegin > pAEnd)
            return 0;
    }
```

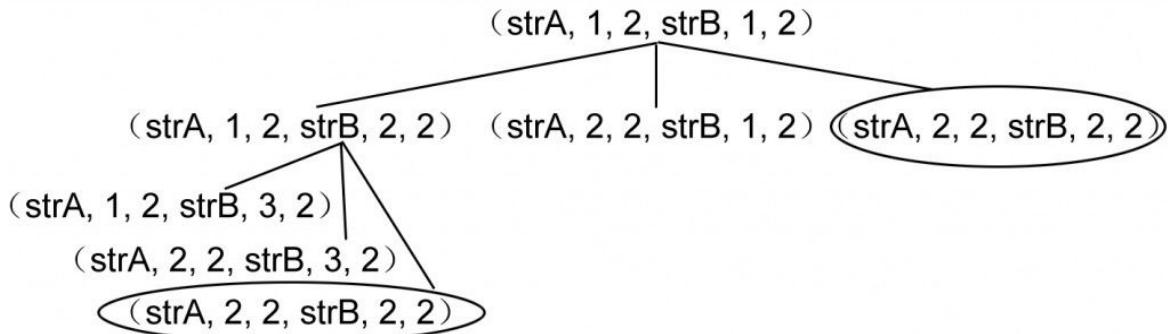
```

        else
            return pPEnd - pABegin + 1;
    }

    if(strA[pABegin] == strB[pBBegin])
    {
        return CalculateStringDistance(strA, pABegin + 1, pPEnd,
            strB, pBBegin + 1, pBEnd);
    }
    else
    {
        int t1 = CalculateStringDistance(strA, pABegin, pPEnd, strB,
            pBBegin + 1, pBEnd);
        int t2 = CalculateStringDistance(strA, pABegin + 1, pPEnd,
            strB, pBBegin, pBEnd);
        int t3 = CalculateStringDistance(strA, pABegin + 1, pPEnd,
            strB, pBBegin + 1, pBEnd);
        return minValue(t1,t2,t3) + 1;
    }
}

```

上面的递归程序，有什么地方需要改进呢？在递归的过程中，有些数据被重复计算了。比如，如果开始我们调用 `CalculateStringDistance(strA, 1, 2, strB, 1, 2)`，下图是部分展开的递归调用。



可以看到，圈中的两个子问题被重复计算了。为了避免这种不必要的重复计算，可以把子问题计算后的解存储起来。如何修改递归程序呢？还是 DP！请看此链接：  
<http://www.cnblogs.com/yujunyong/articles/2004724.html>。

## 深入

1. 详细读者朋友们也已经看到了，百度/Google 经常喜欢出这个字符串编辑距离，实际上，关于这个“编辑距离”问题在搜索引擎中有着重要的作用，如搜索引擎关键字

查询中拼写错误的提示，如下图所示，当你输入“**Jult**”后，因为没有这个单词“**Jult**”，所以搜索引擎猜测你可能是输入错误，进而会提示你是不是找“**July**”：

Google Jult

网页 图片 地图 更多 搜索工具

找到约 4,560,000,000 条结果 (用时 0.29 秒)

显示以下查询字词的结果: [July](#)  
仍然搜索: [Jult](#)

将“[July](#)”从英语翻译成您想要的语言  
[translate.google.cn](#)

July - 七月

[July是什么意思](#) [July在线翻译](#) [英语](#) [读音](#) [用法](#) [例句](#) [海词词典](#)  
[dict.cn/July](#) - 网页快照

中国最权威最专业的海量词典,海词词典为您提供July的在线翻译,July是什么意思,July的真人发音,权威用法和精选例句等。

释义 - 用例 - 讲解 - 问答

[July歌曲July专辑在线试听mp3下载](#)  
[www.xiami.com/artist/63901](#) - 网页快照

풀라이(본명: 장정우) 가수, 작곡가 출생 1984년 7월 20일(서울특별시) 소속사 H2엔터테인먼트 소속소울리스트(CEO, 총괄프로듀서) 학력 국민대학교 경영학학사 ...

结构之法算法之道-博客频道-[CSDN.NET](#)  
[blog.csdn.net\\_v\\_JULY\\_v](#) - 网页快照

支持向量机通俗导论(理解SVM的三层境界) 作者: July、pluskid; 致谢: 白石、... 程

但这个拼写错误检查的原理是什么呢？**Google** 是基于贝叶斯统计推断的方法，相关原理详情可以看下 **Google** 的研发总监 **Peter Norvig** 写的这篇文章：

<http://norvig.com/spell-correct.html>, 以及 fuanyif 写的这篇：

[http://www.ruanyifeng.com/blog/2012/10/spelling\\_corrector.html](http://www.ruanyifeng.com/blog/2012/10/spelling_corrector.html)。

2. 关于什么是“编辑距离”：一个快速、高效的 **Levenshtein** 算法实现，这个是计算两个字符串的算法，**Levenshtein** 距离又称为“编辑距离”，是指两个字符串之间，由一个转换成另一个所需的最少编辑操作次数。当然，次数越小越相似。这里有一个 BT 树的数据结构，挺有意思的：

<http://blog.notdot.net/2007/4/Damn-Cool-Algorithms-Part-1-BK-Trees>;

3. 最后，Lucene 中也有这个算法的实现(我想，一般的搜索引擎一般都应该会有此项拼写错误检查功能的实现)，下面是 lucene 的源码(并没有太多优化，与实际工程中 java 注重实用性原则并不背离)：

```
public final class LevensteinDistance {  
  
    public LevensteinDistance () {  
    }  
  
    // Compute Levenshtein distance:  
    // see org.apache.commons.lang.StringUtils#getLevenshteinDistance(String, String)  
  
    public float getDistance (String target, String other) {  
        char[] sa;
```

```

int n;
int p[];
//'previous' cost array, horizontally
int d[];
// cost array, horizontally
int _d[];
//placeholder to assist in swapping p and d

sa = target.toCharArray();
n = sa.length;
p = new int[n+1];
d = new int[n+1];

final int m = other.length();
if (n == 0 || m == 0) {
    if (n == m) {
        return 1;
    }
    else {
        return 0;
    }
}

// indexes into strings s and t
int i;
// iterates through s
int j;
// iterates through t

char t_j;
// jth character of t

int cost;
// cost

for (i = 0; i<=n; i++) {
    p[i] = i;
}

for (j = 1; j<=m; j++) {
    t_j = other.charAt(j-1);
    d[0] = j;

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

```

```

        cost = sa[i-1]==t_j ? 0 : 1;

    // minimum of cell to the left+1, to the top+1, diagonally left and up +cost
    d[i] = Math.min(Math.min(d[i-1]+1, p[i]+1), p[i-1]+cost);
}

// copy current distance counts to 'previous row' distance counts
_d = p;
p = d;
d = _d;
}

// our last action in the above loop was to switch d and p, so p now
// actually has the most recent cost counts
return 1.0f - ((float) p[n] / Math.max(other.length(), sa.length()));
}
}

```

## 扩展

当然，面试官还可以继续问下去，如请问，如何设计一个比较两篇文章相似性的算法？这个问题的讨论可以看看这里：<http://t.cn/zl82CAH>。OK，字符串编辑距离这个问题实用性很强，限于篇幅，详情读者自己深入吧。

## 参考文献及推荐阅读

1. 参与最大乘积连续子串问题的讨论：<http://www.51weixue.com/thread-246-1-1.html>，在线提交你的代码：<http://hero.pongo.cn/Question/Details?ID=19&ExamID=19>；
2. 参与字符串编辑距离问题的讨论：<http://www.51weixue.com/thread-482-1-1.html>，在线提交你的代码：<http://hero.pongo.cn/Question/Details?ID=20&ExamID=20>；
3. 杨氏矩阵查找、最大乘积连续子串、字符串循环右移、社区很忙等 5 题集中讨论地址：<http://bbs.csdn.net/topics/390398519>；
4. 编程之美；

5. 程序员第一~二十九章集锦: [http://blog.csdn.net/v\\_JULY\\_v/article/details/6460494](http://blog.csdn.net/v_JULY_v/article/details/6460494);
6. <http://www.bjwilly.com/archives/395.html>;

## 后记

有种人喜欢远观代码，以欣赏代码的角度阅读代码，所谓如镜中美女只可远观不可亵玩焉，发现自己也陷入了这种境地。昨天买的一本《提高 C++ 性能的编程技术》，书不错，推荐给大家。

想看编程面试算法题的讲解，就进本 **blog**；想参与笔试面试题的讨论，就上[为学论坛](#)；想在线刷笔试面试题，就上[英雄会](#)，祝各位好运，享受生活，享受工作，享受思考和编码的乐趣。

July、二零一三年三月二十一日。

## 第三十~三十一章：字符串转换成整数，带通配符的字符串匹配

### 前言

之前本一直想写写神经网络算法和 EM 算法，但写这两个算法实在需要大段大段的时间，而平时上班，周末则跑去北大教室自习看书（顺便以时间为序，说下过去半年看过的自觉还不错的数学史方面的书：《数理统计学简史》《微积分概念发展史》《微积分的历程：从牛顿到勒贝格》《数学恩仇录》《数学与知识的探求》《古今数学思想》《素数之恋》），故一直未曾有时间写。

然最近在负责一款在线编程挑战平台：<http://hero.pongo.cn/>（简称 hero，通俗理解是中国的 topcoder，当然，一直在不断完善中，与一般 OJ 不同点在于，OJ 侧重为参与 ACM 竞赛者提供刷题练习的场所，而 hero 则着重为企业招聘面试服务），在上面出了几道编程面试题，有些题目看似简单，但一 coding，很多问题便立马都在 hero 上给暴露出来了，故就从 hero 上的编程挑战题切入，继续更新本程序员编程艺术系列吧。

况且，前几天与一朋友聊天，他说他认识的今年 360 招进来的三四十人应届生包括他自己找工作时基本都看过我的博客，则更增加了更新此编程艺术系列的动力。

OK，本文讲两个问题：

- 第三十章、字符串转换成整数，从确定思路，到写出有瑕疵的代码，继而到 `microsoft & linux` 的 `atoi` 实现，再到第一份比较完整的代码，最后以 Net/OS 中的实现结尾，看似很简单的一个问题，其实非常不简单；
- 第三十一章、字符串匹配问题

还是这句老话，有问题恳请随时批评指正，感谢。

### 第三十章、字符串转换成整数

先看题目：

输入一个表示整数的字符串，把该字符串转换成整数并输出，例如输入字符串"345"，则输出整数 345。

给定函数原型 `int StrToInt(const char *str)`，完成函数 `StrToInt`，实现字符串转换成整数的功能，不得用库函数 `atoi`（即便准许使用，其对于溢出情况的处理也达不到题目的要求，详情请参看下文第 7 节末）。

我们来一步一步分析（共 9 小节，重点在下文第 8 小节及后续内容），直至写出第一份准确的代码：

**1、**本题考查的实际上就是字符串转换成整数的问题，或者说是要你自行实现 `atoi` 函数。那如何实现把表示整数的字符串正确地转换成整数呢？以"345"作为例子：

1. 当我们扫描到字符串的第一个字符'3'时，由于我们知道这是第一位，所以得到数字 3。
2. 当扫描到第二个数字'4'时，而之前我们知道前面有一个 3，所以便在后面加上一个数字 4，那前面的 3 相当于 30，因此得到数字： $3*10+4=34$ 。
3. 继续扫描到字符'5'，'5'的前面已经有了 34，由于前面的 34 相当于 340，加上后面扫描到的 5，最终得到的数是： $34*10+5=345$ 。

因此，此题的思路便是：每扫描到一个字符，我们便把在之前得到的数字乘以 10，然后再加上当前字符表示的数字。

**2、**思路有了，有一些细节需要注意，如 zhedahht 所说：

1. “由于整数可能不仅仅之含有数字，还有可能以'+'或者'-'开头，表示整数的正负。因此我们需要把这个字符串的第一个字符做特殊处理。如果第一个字符是'+'号，则不需要做任何操作；如果第一个字符是'-'号，则表明这个整数是个负数，在最后的时候我们要把得到的数值变成负数。”
2. 接着我们试着处理非法输入。由于输入的是指针，在使用指针之前，我们要做的第一件是判断这个指针是不是为空。如果试着去访问空指针，将不可避免地导致程序崩溃。
3. 另外，输入的字符串中可能含有不是数字的字符。每当碰到这些非法的字符，我们就没有必要再继续转换。

4. 最后一个需要考虑的问题是溢出问题。由于输入的数字是以字符串的形式输入，因此有可能输入一个很大的数字转换之后会超过能够表示的最大的整数而溢出。”

比如，当给的字符串是如左边图片所示的时候，有考虑到么？当然，它们各自对应的正确输出如右边图片所示（假定你是在 32 位系统下，且编译环境是 VS2008 以上）：

""	0
"1"	1
"+1"	1
"-1"	-1
"123"	123
"-123"	-123
"010"	10
"+00131204"	131204
"-01324000"	-1324000
"2147483647"	2147483647
"-2147483647"	-2147483647
"-2147483648"	-2147483648
"2147483648"	2147483647
"-2147483649"	-2147483648
"abc"	0
"-abc"	0
"1a"	1
"23a8f"	23
"-3924x8fc"	-3924
" 321"	321
" -321"	-321
"123 456"	123
"123 "	123
" - 321"	0
" +4488"	4488
" + 413"	0
" ++c"	0
" ++1"	0
" --2"	0
" -2"	-2

3、很快，可能你就会写下如下代码：

```
//copyright@zhedahht 2007
enum Status {kValid = 0, kInvalid};
int g_nStatus = kValid;

// Convert a string into an integer
int StrToInt(const char* str)
{
    g_nStatus = kInvalid;
    long long num = 0;

    if(str != NULL)
    {
```

```

const char* digit = str;

// the first char in the string maybe '+' or '-'
bool minus = false;
if(*digit == '+')
    digit++;
else if(*digit == '-')
{
    digit++;
    minus = true;
}

// the remaining chars in the string
while(*digit != '\0')
{
    if(*digit >= '0' && *digit <= '9')
    {
        num = num * 10 + (*digit - '0');

        // overflow
        if(num > std::numeric_limits<int>::max())
        {
            num = 0;
            break;
        }
    }

    digit++;
}
// if the char is not a digit, invalid input
else
{
    num = 0;
    break;
}
}

if(*digit == '\0')
{
    g_nStatus = kValid;
    if(minus)
        num = 0 - num;
}
}

return static_cast<int>(num);

```

```
}
```

run 下上述程序，会发现当输入字符串是下图中红叉部分所对应的时候，程序结果出错：

“-2147483648”	0	-2147483648	✗
“2147483648”	0	2147483647	✗
“-2147483649”	0	-2147483648	✗
“abc”	0	0	✓
“-abc”	0	0	✓
“1a”	0	1	✗
“23a8f”	0	23	✗
“-3924x8fc”	0	-3924	✗
“ 321”	0	321	✗
“ -321”	0	-321	✗
“123 456”	0	123	✗
“123 ”	0	123	✗
“ - 321”	0	0	✓
“ +4488”	0	4488	✗
“ + 413”	0	0	✓
“ ++c”	0	0	✓
“ ++1”	0	0	✓
“ --2”	0	0	✓
“ -2”	0	-2	✗

两个问题：

1. 当输入的字符串不是数字，而是字符的时候，比如“1a”，上述程序直接返回了 0（而正确的结果应该是得到 1）：

```
// if the char is not a digit, invalid input
    else
    {
        num = 0;
        break;
    }
```

2. 处理溢出时，有问题。因为它遇到溢出情况时，直接返回了 0：

```
// overflow
    if(num > std::numeric_limits<int>::max())
    {
        num = 0;
        break;
    }
```

**4、**把代码做下微调，如下（注：库函数 atoi 规定超过 int 值，按最大值 maxint: 2147483647 来，

超过-int 按最小值 minint: -2147483648 来）：

```
//copyright@SP_daiyq 2013/5/29
```

```

int StrToInt(const char* str)
{
    int res = 0; // result
    int i = 0; // index of str
    int signal = '+'; // signal '+' or '-'
    int cur; // current digit

    if (!str)
        return 0;

    // skip backspace
    while (isspace(str[i]))
        i++;

    // skip signal
    if (str[i] == '+' || str[i] == '-')
    {
        signal = str[i];
        i++;
    }

    // get result
    while (str[i] >= '0' && str[i] <= '9')
    {
        cur = str[i] - '0';

        // judge overlap or not
        if ( (signal == '+') && (cur > INT_MAX - res*10) )
        {
            res = INT_MAX;
            break;
        }
        else if ( (signal == '-') && (cur -1 > INT_MAX - res*10) )
        {
            res = INT_MIN;
            break;
        }

        res = res * 10 + cur;
        i++;
    }

    return (signal == '-') ? -res : res;
}

```

此时会发现，上面第 3 小节末所述的第 1 个小问题（当输入的字符串不是数字，而是字符的时候）解决了：

"-2147483648"	-2147483648	-2147483648	✓
"2147483648"	2147483647	2147483647	✓
"-2147483649"	-2147483648	-2147483648	✓
"abc"	0	0	✓
"-abc"	0	0	✓
"1a"	1	1	✓
"23a8f"	23	23	✓
"-3924x8fc"	-3924	-3924	✓
" 321"	321	321	✓
" -321"	-321	-321	✓
"123 456"	123	123	✓
"123 "	123	123	✓
" - 321"	0	0	✓
" +4488"	4488	4488	✓
" + 413"	0	0	✓
" ++c"	0	0	✓
" ++1"	0	0	✓
" --2"	0	0	✓
" -2"	-2	-2	✓

但，上文第 3 小节末所述的第 2 个小问题：溢出问题却没有解决。即当给定下述测试数据的时候，问题就来了：

需要转换的字符串	代码运行结果	理应得到的正确结果	
" 10522545459"	1932610867	2147483647	✗
" +10523538441s"	1933603849	2147483647	✗
" +10432359437"	1842424845	2147483647	✗

什么问题呢？比如说用上述代码转换这个字符串：" 10522545459"，它本应得到的正确结果应该是 2147483647，但程序实际得到的结果却是：1932610867。故很明显，程序没有解决好上面的第 2 个小问题：溢出问题。原因是什么呢？咱们来分析下代码，看是如何具体处理溢出情况的：

```
// judge overlap or not
if ( (signal == '+') && (cur > INT_MAX - res*10) )
{
    res = INT_MAX;
    break;
}
else if ( (signal == '-') && (cur -1 > INT_MAX - res*10) )
{
    res = INT_MIN;
    break;
}
```

接着上面的例子来，比如给定字符串" 10522545459"，除去空格有 11 位，而 MAX\_INT，即 2147483647 是 10 位数，当扫描到最后一个字符'9'的时候，程序会比较 9 和 2147483647 - 1052254545\*10 的大小。

问题立马就暴露出来了，因为此时让 `res*10`，即让  $1052254545 * 10 > \text{MAX\_INT}$ ，溢出无疑，程序已经出错，再执行下面这行代码已无意义：

```
1.     cur > INT_MAX - res*10
```

也就是说，对于字符串"10522545459"，当扫描到最后一个字符'9'时，根据上文第1小节的字符串转换成整数的思路：“每扫描到一个字符，我们便把在之前得到的数字乘以10，然后再加上当前字符表示的数字”，为了得到最终的整数，我们得如此计算：

$1052254545 * 10 + 4$ ，

然实际上当程序计算到  $1052254545 * 10$  时，

$1052254545 * 10 >$

2147483647

此时已经溢出了，若再执意计算，则程序逻辑将出错，故此后也就不能再判断字串的最后一位 4 是否大于  $2147483647 \% 10$  了（耐不得烦想尽快看到最终正确代码的读者可以直接跳到下文第 8 节）。

**5、** 上面说给的程序没有“很好的解决溢出问题。由于输入的数字是以字符串的形式输入，因此有可能输入一个很大的数字转换之后会超过能够表示的最大的整数而溢出”。那么，到底代码该如何写呢？

像下面这样？：

```
//copyright@fuwutu 2013/5/29
int StrToInt(const char* str)
{
    bool negative = false;
    long long result = 0;
    while (*str == ' ' || *str == '\t')
    {
        ++str;
    }
    if (*str == '-')
    {
        negative = true;
        ++str;
    }
    else if (*str == '+')
    {
        ++str;
    }
```

```
while (*str != '\0')
{
    int n = *str - '0';
    if (n < 0 || n > 9)
    {
        break;
    }

    if (negative)
    {
        result = result * 10 - n;
        if (result < -2147483648LL)
        {
            result = -2147483648LL;
        }
    }
    else
    {
        result = result * 10 + n;
        if (result > 2147483647LL)
        {
            result = 2147483647LL;
        }
    }
    ++str;
}

return result;
}
```

run 下程序，看看运行结果：

" +1q0384297885"	1	1	✓
" -1034946q1019"	-1034946	-1034946	✓
" 11810097701"	2147483647	2147483647	✓
"115784825v67"	115784825	115784825	✓
" 1030q2849284"	1030	1030	✓
" -120m15417222"	-120	-120	✓
"1095502006p8"	1095502006	1095502006	✓
" r11384376420"	0	0	✓
" 10522545459"	2147483647	2147483647	✓
" +1u0557196150"	1	1	✓
" 10315g546111"	10315	10315	✓
" +o11950655481"	0	0	✓
" 1052223016k4"	1052223016	1052223016	✓
" 1206z8519909"	1206	1206	✓
" -115a64905859"	-115	-115	✓
" 1191597x7433"	1191597	1191597	✓
" 1094m3066812"	1094	1094	✓

上图所示程序貌似通过了，然实际上它还是未能处理数据溢出的问题，因为它只是做了个取巧，即把返回的值 `esult` 定义成了 `long long`，如下所示：

```
long long result = 0;
```

故严格说来，我们依然未写出准确的规范代码。

**6、那到底该如何解决这个数据溢出的问题呢？咱们先来看看 Microsoft 是如何实现 `atoi` 的吧：**

```
//atol 函数
//Copyright (c) 1989-1997, Microsoft Corporation. All rights reserved.

long __cdecl atol(
    const char *nptr
)
{
    int c; /* current char */
    long total; /* current total */
    int sign; /* if '-' , then negative, otherwise positive */

    /* skip whitespace */
    while ( isspace((int)(unsigned char)*nptr) )
        ++nptr;

    c = (int)(unsigned char)*nptr++;
    sign = c; /* save sign indication */
    if (c == '-' || c == '+')
        c = (int)(unsigned char)*nptr++; /* skip sign */
```

```

total = 0;

while (isdigit(c)) {
    total = 10 * total + (c - '0'); /* accumulate digit */
    c = (int)(unsigned char)*nptr++; /* get next char */
}

if (sign == '-')
    return -total;
else
    return total; /* return result, negated if necessary */
}

```

其中，`isspace` 和 `isdigit` 函数的实现代码为：

```

isspace(int x)
{
    if(x==' '||x=='\t'||x=='\n'||x=='\f'||x=='\b'||x=='\r')
        return 1;
    else
        return 0;
}

isdigit(int x)
{
    if(x<='9'&&x>='0')
        return 1;
    else
        return 0;
}

```

然后 `atoi` 调用上面的 `atol` 函数，如下所示：

```

//atoi 调用上述的 atol
int __cdecl atoi(
    const char *nptr
)
{
    //Overflow is not detected. Because of this, we can just use
    return (int)atol(nptr);
}

```

但很遗憾的是，上述 `atoi` 标准代码依然返回的是 `long`：

```

long total; /* current total */
if (sign == '-')
    return -total;
else

```

```
    return total; /* return result, negated if necessary */
```

再者，下面这里定义成 long 的 total 与 10 相乘，即 total\*10 很容易溢出：

```
long total; /* current total */
total = 10 * total + (c - '0'); /* accumulate digit */
```

最后，根据本文评论下的读者 meiyuli 反应：“测试数据是字符串"-21474836480"，api 算出来的是-2147483648，用上述代码算出来的结果是 0”，如此，上述微软的这个 atoi 源码是有问题的。

7、microsoft 既然不行，读者想必很自然的想到 linux。So，咱们接下来便看看 linux 内核中是如何实现此字符串转换为整数的问题的。linux 内核中提供了以下几个函数：

1. simple\_strtol，把一个字符串转换为一个有符号长整数；
2. simple strtoll，把一个字符串转换为一个有符号长长整数；
3. simple strtoul，把一个字符串转换为一个无符号长整数；
4. simple strtoull，把一个字符串转换为一个无符号长长整数

相关源码及分析如下。

首先，atoi 调下面的 strtol：

```
//linux/lib/vsprintf.c
//Copyright (C) 1991, 1992 Linus Torvalds
//simple_strtol - convert a string to a signed long
long simple_strtol(const char *cp, char **endp, unsigned int base)
{
    if (*cp == '-')
        return -simple strtoul(cp + 1, endp, base);

    return simple strtoul(cp, endp, base);
}
EXPORT_SYMBOL(simple_strtol);
```

然后，上面的 strtol 调下面的 strtoul：

```
//simple strtoul - convert a string to an unsigned long
unsigned long simple strtoul(const char *cp, char **endp, unsigned int base)
{
    return simple strtoull(cp, endp, base);
}
EXPORT_SYMBOL(simple strtoul);
```

接着，上面的 strtoul 调下面的 strtoull：

```
//simple strtoull - convert a string to a signed long long
```

```

long long simple_strtoll(const char *cp, char **endp, unsigned int base)
{
    if (*cp == '-')
        return -simple strtoull(cp + 1, endp, base);

    return simple strtoull(cp, endp, base);
}
EXPORT_SYMBOL(simple_strtoll);

```

最后，`strtoull` 调用 `_parse_integer_fixup_radix` 和 `_parse_integer` 来处理相关逻辑：

```

//simple strtoull - convert a string to an unsigned long long
unsigned long long simple strtoull(const char *cp, char **endp, unsigned int base)
{
    unsigned long long result;
    unsigned int rv;

    cp = _parse_integer_fixup_radix(cp, &base);
    rv = _parse_integer(cp, base, &result);
    /* FIXME */
    cp += (rv & ~KSTRTOX_OVERFLOW);

    if (endp)
        *endp = (char *)cp;

    return result;
}
EXPORT_SYMBOL(simple strtoull);

```

重头戏来了。接下来，我们来看上面 `strtoull` 函数中的 `_parse_integer_fixup_radix` 和 `_parse_integer` 两段代码。如鲨鱼所说

- “真正的处理逻辑主要是在 `_parse_integer` 里面，关于溢出的处理，`_parse_integer` 处理的很优美，
- 而 `_parse_integer_fixup_radix` 是用来自动根据字符串判断进制的”。

先来看 `_parse_integer` 函数：

```

//lib/kstrtox.c, line 39
//Convert non-negative integer string representation in explicitly given radix to a
n integer.
//Return number of characters consumed maybe or-ed with overflow bit.
//If overflow occurs, result integer (incorrect) is still returned.
unsigned int _parse_integer(const char *s, unsigned int base, unsigned long long
*p)
{

```

```

    unsigned long long res;
    unsigned int rv;
    int overflow;

    res = 0;
    rv = 0;
    overflow = 0;
    while (*s) {
        unsigned int val;

        if ('0' <= *s && *s <= '9')
            val = *s - '0';
        else if ('a' <= _tolower(*s) && _tolower(*s) <= 'f')
            val = _tolower(*s) - 'a' + 10;
        else
            break;

        if (val >= base)
            break;
        /*
         * Check for overflow only if we are within range of
         * it in the max base we support (16)
         */
        if (unlikely(res & (~0ull << 60))) {
            if (res > div_u64(ULLONG_MAX - val, base))
                overflow = 1;
        }
        res = res * base + val;
        rv++;
        s++;
    }
    *p = res;
    if (overflow)
        rv |= KSTRTOX_OVERFLOW;
    return rv;
}

```

解释下两个小细节：

1. 上头出现了个 unlikely，其实 unlikely 和 likely 经常出现在 linux 相关内核源码中

```

if(likely(value)){
    //等价于 if(likely(value)) == if(value)
}
else{

```

```
}
```

likely 表示 value 为真的可能性更大，而 unlikely 表示 value 为假的可能性更大，这两个宏被定义成：

```
//include/linux/compiler.h
#ifndef likely
#define likely(x) (_builtin_constant_p(x) ? !!x : __branch_check__(x, 1))

#endif
#ifndef unlikely
#define unlikely(x)  (_builtin_constant_p(x) ? !!x : __branch_check__(x,
0))
#endif
```

2. 呈现下 div\_u64 的代码：

```
//include/linux/math64.h
//div_u64
static inline u64 div_u64(u64 dividend, u32 divisor)
{
    u32 remainder;
    return div_u64_rem(dividend, divisor, &remainder);
}

//div_u64_rem
static inline u64 div_u64_rem(u64 dividend, u32 divisor, u32 *remainder)
{
    *remainder = dividend % divisor;
    return dividend / divisor;
}
```

最后看下\_parse\_integer\_fixup\_radix 函数：

```
//lib/kstrtox.c, line 23
const char *_parse_integer_fixup_radix(const char *s, unsigned int *base)
{
    if (*base == 0) {
        if (s[0] == '0') {
            if (_tolower(s[1]) == 'x' && isxdigit(s[2]))
                *base = 16;
            else
                *base = 8;
        } else
            *base = 10;
    }
}
```

```
if (*base == 16 && s[0] == '0' && _tolower(s[1]) == 'x')
    s += 2;
return s;
}
```

读者 MJN 君在我的建议下，对上述 linux 内核中的 atoi 函数进行了测试，咱们来看下测试结果如何。

```
2147483647 : 2147483647
2147483648 : -2147483648
10522545459 : 1932610867
-2147483648 : -2147483648
-2147483649 : -2147483647
-10522545459 : 1932610867
```

如上，根据程序的输出结果可以看出，对于某些溢出的情况，atoi 程序的处理并不符合本题的要求。

也就是说，atoi 程序对溢出的处理是一个标准，而本题要求对溢出的处理则是另外一个标准，所以说直接用 atoi 程序达不到本题的要求，但你不能因为本题的标准而否认 atoi 程序的正确性。

既然直接借用 atoi 的源码（原理是 parseXXX, int i=Integer.parseInt(String str)，把 str 转换成 int 的方法），不符合题目要求，则咱们另寻他路。

路漫漫其修远兮，吾等将上下而求索，但与此同时，我们已渐入佳境。

**8、** 根据我们第 1 小节达成一致的字符串转换成整数的思路：“每扫描到一个字符，我们便把在之前得到的数字乘以 10，然后再加上当前字符表示的数字”，相信读者已经觉察到，在扫描到最后一个字符的时候，如果之前得到的数比较大，此时若再让其扩大 10 倍，相对来说是比较容易溢出的。

但车到山前必有路，既然让一个比较大的 int 整型数据大 10 倍，比较容易溢出，那么在不好判断是否溢出的情况下，可以尝试使用除法。即如 MJN 所说：

1. 与其将 n 扩大 10 倍，冒着溢出的风险，再与 MAX\_INT 进行比较（如果已经溢出，则比较的结果没有意义），

2. 不如未雨绸缪先用 n 与 MAX\_INT/10 进行比较：若  $n > MAX\_INT/10$ （当然同时还要考虑  $n = MAX\_INT/10$  的情况），说明最终得到的整数一定会溢出，故此时可以当即进行溢出处理，直接返回最大值 MAX\_INT，从而也就免去了计算  $n * 10$  这一步骤。

也就是说，计算  $n * 10$  前，先比较 n 与 MAX\_INT/10 大小，若  $n > MAX\_INT/10$ ，那么  $n * 10$  肯定大于 MAX\_INT，即代表最后得到的整数 n 肯定溢出，既然溢出，不能再计算  $n * 10$ ，直接提前返回 MAX\_INT 就行了。

一直以来，我们努力的目的归根结底是为了更好的处理溢出，但上述做法最重要的是巧妙的规避了计算  $n * 10$  这一乘法步骤，转换成计算除法 MAX\_INT/10 代替，不能不说此法颇妙。

他的代码如下，如有问题请指出：

```
//copyright@njnu_mjn 2013
int StrToInt(const char* str)
{
    static const int MAX = (int)((unsigned)~0 >> 1);
    static const int MIN = -(int)((unsigned)~0 >> 1) - 1;
    unsigned int n = 0;
    int sign = 1;
    int c;

    while (isspace(*str))
        ++str;
    if (*str == '+' || *str == '-')
    {
        if (*str == '-')
            sign = -1;
        ++str;
    }
    while (isdigit(*str))
    {
        c = *str - '0';
        if (sign > 0 && (n > MAX/10 || (n == MAX/10 && c > MAX%10)))
        {
            n = MAX;
            break;
        }
        else if (sign < 0 && (n > (unsigned)MIN/10
                               || (n == (unsigned)MIN/10 && c > (unsigned)MIN%10)))
        {
            n = MIN;
            break;
        }
        n *= 10;
        n += c;
    }
}
```

```

{
    n = MIN;
    break;
}
n = n * 10 + c;
++str;
}
return sign > 0 ? n : -n;
}

```

上述代码从测试结果来看，暂未发现什么问题

输入	输出
10522545459 :	2147483647
-10522545459 :	-2147483648

咱们再来总结下上述代码是如何处理溢出情况的。对于正数来说，它溢出的可能性有两种：

1. 一种是诸如 2147483650，即  $n > \text{MAX}/10$  的；
2. 一种是诸如 2147483649，即  $n == \text{MAX}/10 \ \&\& c > \text{MAX}\%10$ 。

故咱们上面处理溢出情况的代码便是：

```

c = *str - '0';
if (sign > 0 && (n > MAX/10 || (n == MAX/10 && c > MAX%10)))
{
    n = MAX;
    break;
}
else if (sign < 0 && (n > (unsigned)MIN/10
    || (n == (unsigned)MIN/10 && c > (unsigned)MIN%10)))

{
    n = MIN;
    break;
}

```

不过，即便如此，有些细节是改进的，如他自己所说：

1.  $n$  的声明及定义应该为

```
int n = 0;
```

2. 将 MAX/10,MAX%10,(unsigned)MIN/10 及(unsigned)MIN%10 保存到变量中，防止重复计算

这样，优化后的代码为：

```
//copyright@njnu_mjn 2013
int StrToInt(const char* str)
{
    static const int MAX = (int)((unsigned)~0 >> 1);
    static const int MIN = -(int)((unsigned)~0 >> 1) - 1;
    static const int MAX_DIV = (int)((unsigned)~0 >> 1) / 10;
    static const int MIN_DIV = (int)((((unsigned)~0 >> 1) + 1) / 10);
    static const int MAX_R = (int)((unsigned)~0 >> 1) % 10;
    static const int MIN_R = (int)((((unsigned)~0 >> 1) + 1) % 10);
    int n = 0;
    int sign = 1;
    int c;

    while (isspace(*str))
        ++str;
    if (*str == '+' || *str == '-')
    {
        if (*str == '-')
            sign = -1;
        ++str;
    }
    while (isdigit(*str))
    {
        c = *str - '0';
        if (sign > 0 && (n > MAX_DIV || (n == MAX_DIV && c >= MAX_R)))
        {
            n = MAX;
            break;
        }
        else if (sign < 0 && (n > MIN_DIV
                                || (n == MIN_DIV && c >= MIN_R)))
    )
    {
        n = MIN;
        break;
    }
    n = n * 10 + c;
    ++str;
}
return sign > 0 ? n : -n;
```

```
}
```

部分数据的测试结果如下图所示：

输入	输出
10522545459	: 2147483647
-10522545459	: -2147483648
2147483648	: 2147483647
-2147483648	: -2147483648

是否已是完美？如 MJN 君本人所说“我的实现与 linux 内核的 atoi 函数的实现，都有一个共同的问题：即使出错，函数也返回了一个值，导致调用者误认为自己传入的参数是正确的，但是可能会导致程序的其他部分产生莫名的错误且很难调试”。

9、最后看下 Nut/OS 中 atoi 的实现，同时，本小节内容主要来自参考文献条目 9，即 MJN 的博客：

```
#include <compiler.h>

#include <stdlib.h>

int atoi(CONST char *str)

{

    return ((int) strtol(str, (char **) NULL, 10));

}
```

上述代码中 strtol 实现的思想跟上文第 7 节所述的 MJN 君的思路类似，也是除法代替乘法。加上测试函数后的具体代码如下：

```
#include <errno.h>
#include <stdio.h>
#include <ctype.h>
#include <limits.h>

#define CONST      const

long mstrtol(CONST char *nptr, char **endptr, int base)
{
    register CONST char *s;
```

```

register long acc, cutoff;
register int c;
register int neg, any, cutlim;

/*
 * Skip white space and pick up leading +/- sign if any.
 * If base is 0, allow 0x for hex and 0 for octal, else
 * assume decimal; if base is already 16, allow 0x.
 */
s = nptr;
do {
    c = (unsigned char) *s++;
} while (isspace(c));
if (c == '-') {
    neg = 1;
    c = *s++;
} else {
    neg = 0;
    if (c == '+')
        c = *s++;
}
if ((base == 0 || base == 16) && c == '0' && (*s == 'x' || *s == 'X')) {
    c = s[1];
    s += 2;
    base = 16;
}
if (base == 0)
    base = c == '0' ? 8 : 10;

/*
 * Compute the cutoff value between legal numbers and illegal
 * numbers. That is the largest legal value, divided by the
 * base. An input number that is greater than this value, if
 * followed by a legal input character, is too big. One that
 * is equal to this value may be valid or not; the limit
 * between valid and invalid numbers is then based on the last
 * digit. For instance, if the range for longs is
 * [-2147483648..2147483647] and the input base is 10,
 * cutoff will be set to 214748364 and cutlim to either
 * 7 (neg==0) or 8 (neg==1), meaning that if we have accumulated
 * a value > 214748364, or equal but the next digit is > 7 (or 8),
 * the number is too big, and we will return a range error.
 *
 * Set any if any `digits' consumed; make it negative to indicate

```

```

* overflow.

*/
cutoff = neg ? LONG_MIN : LONG_MAX;
cutlim = cutoff % base;
cutoff /= base;
if (neg) {
    if (cutlim > 0) {
        cutlim -= base;
        cutoff += 1;
    }
    cutlim = -cutlim;
}
for (acc = 0, any = 0;; c = (unsigned char) *s++) {
    if (isdigit(c))
        c -= '0';
    else if (isalpha(c))
        c -= isupper(c) ? 'A' - 10 : 'a' - 10;
    else
        break;
    if (c >= base)
        break;
    if (any < 0)
        continue;
    if (neg) {
        if ((acc < cutoff || acc == cutoff) && c > cutlim) {
            any = -1;
            acc = LONG_MIN;
            errno = ERANGE;
        } else {
            any = 1;
            acc *= base;
            acc -= c;
        }
    } else {
        if ((acc > cutoff || acc == cutoff) && c > cutlim) {
            any = -1;
            acc = LONG_MAX;
            errno = ERANGE;
        } else {
            any = 1;
            acc *= base;
            acc += c;
        }
    }
}

```

```

    }

    if (endptr != 0)
        *endptr = (char *) (any ? s - 1 : nptr);
    return (acc);
}

int matoi2(CONST char *str)
{
    return ((int) strtol(str, (char **) NULL, 10));
}

int mgetline(char* buf, size_t n) {
    size_t idx = 0;
    int c;

    while (--n > 0 && (c = getchar()) != EOF && c != '\n') {
        buf[idx++] = c;
    }
    buf[idx] = '\0';
    return idx;
}

#define MAX_LINE 200

int main() {
    char buf[MAX_LINE];
    while (mgetline(buf, MAX_LINE) >= 0) {
        if (strcmp(buf, "quit") == 0) break;
        printf("matoi2=%d\n", matoi2(buf));
    }
    return 0;
}

```

同样，MJN 对上述实现测试了下，结果如下：

```

10522545459
matoi2=2147483647
-10522545459
matoi2=-2147483648

```

程序貌似对溢出的处理是正确的，真的吗？再把测试数据换成"10522545454"（与"10522545459"的区别在于最后一个字符）

```

10522545454
matoi2=1932610862
-10522545454

```

```
matoi2=-1932610862
```

症结就在于下面这段代码：

```
if (neg) {
    if ((acc < cutoff || acc == cutoff) && c > cutlim) {
        any = -1;
        acc = LONG_MIN;
        errno = ERANGE;
    } else {
        any = 1;
        acc *= base;
        acc -= c;
    }
} else {
    if ((acc > cutoff || acc == cutoff) && c > cutlim) {
        any = -1;
        acc = LONG_MAX;
        errno = ERANGE;
    }
}
```

要想得到正确的输出结果，需要改动两个地方：

① 其中这行：

```
if ((acc > cutoff || acc == cutoff) && c > cutlim)
```

应该改为：

```
if ( acc > cutoff || (acc == cutoff) && c > cutlim) )
```

② 与此同时，这行：

```
if ((acc < cutoff || acc == cutoff) && c > cutlim) {
```

改为：

```
if (acc < cutoff || (acc == cutoff && c > cutlim)) {
```

为何要这样修改呢？细心的读者相信还是会记得上文第 8 节中关于正数的两种溢出情况的可能性：“对于正数来说，它溢出的可能性有两种：

1. 一种是诸如 2147483650，即  $n > \text{MAX}/10$  的；
2. 一种是诸如 2147483649，即  $n == \text{MAX}/10 \&& c > \text{MAX}\%10$ 。 ”

也就是说无论是"10522545459"，还是"10522545454"，都是属于第 1 种情况，即“诸如 2147483650，即  $n > \text{MAX}/10$  的”，此时直接返回 `MAX_INT` 即可，所以不需要也不能再去判断  $n == \text{MAX}/10$  的情况。

这个处理思路类似于上文第 8 节处理溢出情况的代码：

```
if (sign > 0 && (n > MAX/10 || (n == MAX/10 && c > MAX%10)))
{
    n = MAX;
    break;
}
else if (sign < 0 && (n > (unsigned)MIN/10
                           || (n == (unsigned)MIN/10 && c > (unsigned)MIN%10)))
{
    n = MIN;
    break;
}
```

So，修改过后的代码测试正常：

```
10522545459
matoi2=2147483647
-10522545459\
matoi2=-2147483648
10522545454
matoi2=2147483647
-10522545454
matoi2=-2147483648
quit
```

OK，字符串转换成整数这一问题已基本解决。但如果面试官继续问你，如何把整数转换成字符串呢？欢迎于本文评论下或 `hero` 上 `show` 出你的思路或代码。

## 第三十一章、带通配符的字符串匹配问题

字符串匹配问题，给定一串字符串，按照指定规则对其进行匹配，并将匹配的结果保存至 `output` 数组中，多个匹配项用空格间隔，最后一个不需要空格。

要求：

1. 匹配规则中包含通配符? 和\*, 其中? 表示匹配任意一个字符, \*表示匹配任意多个( $\geq 0$ )字符。
2. 匹配规则要求匹配最大的字符子串, 例如  $a^*d$ , 匹配  $abbdd$  而非  $abbd$ , 即最大匹配子串。
3. 匹配后的输入串不再进行匹配, 从当前匹配后的字符串重新匹配其他字符串。

请实现函数: `char* my_find(char input[], char rule[])`

举例说明

`input:abcadefg`

`rule:a?c`

`output:abc`

`input :newsadfanewfdadsf`

`rule: new`

`output: new new`

`input :breakfastfood`

`rule: f*d`

`output:fastfood`

注意事项:

1. 自行实现函数 `my_find`, 勿在 `my_find` 函数里夹杂输出, 且不准用 C、C++ 库, 和 Java 的 String 对象;
2. 请注意代码的时间, 空间复杂度, 及可读性, 简洁性;
3. `input=aaa, rule=aa` 时, 返回一个结果 `aa`, 即可。

**1.** 本题与上述第三十章的题不同, 上题字符串转换成整数更多考察对思维的全面性和对细节的处理, 本题则更多的是编程技巧。闲不多说, 直接上代码:

```
//copyright@cao_peng 2013/4/23
int str_len(char *a) { //字符串长度
    if (a == 0) {
        return 0;
    }
    char *t = a;
    for (; *t; ++t)
```

```

        ;
    return (int) (t - a);
}

void str_copy(char *a,const char *b,int len) { //拷贝字符串 a = b
    for (;len > 0; --len, ++b,++a) {
        *a = *b;
    }
    *a = 0;
}

char *str_join(char *a,const char *b,int lenb) { //连接字符串 第一个字符串被回收
    char *t;
    if (a == 0) {
        t = (char *) malloc(sizeof(char) * (lenb + 1));
        str_copy(t, b, lenb);
        return t;
    }
    else {
        int lena = str_len(a);
        t = (char *) malloc(sizeof(char) * (lena + lenb + 2));
        str_copy(t, a, lena);
        *(t + lena) = ' ';
        str_copy(t + lena + 1, b, lenb);
        free(a);
        return t;
    }
}

int canMatch(char *input, char *rule) { // 返回最长匹配长度 -1 表示不匹配
    if (*rule == 0) { //已经到 rule 尾端
        return 0;
    }
    int r = -1 ,may;
    if (*rule == '*') {
        r = canMatch(input, rule + 1); // *匹配 0 个字符
        if (*input) {
            may = canMatch(input + 1, rule); // *匹配非 0 个字符
            if ((may >= 0) && (++may > r)) {
                r = may;
            }
        }
    }
    if (*input == 0) { //到尾端

```

```

    return r;
}
if ((*rule == '?') || (*rule == *input)) {
    may = canMatch(input + 1, rule + 1);
    if ((may >= 0) && (++may > r)) {
        r = may;
    }
}
return r;
}

char * my_find(char input[], char rule[]) {
    int len = str_len(input);
    int *match = (int *) malloc(sizeof(int) * len); //input 第 i 位最多能匹配多少
位 匹配不上是-1
    int i,max_pos = - 1;
    char *output = 0;

    for (i = 0; i < len; ++i) {
        match[i] = canMatch(input + i, rule);
        if ((max_pos < 0) || (match[i] > match[max_pos])) {
            max_pos = i;
        }
    }
    if ((max_pos < 0) || (match[max_pos] <= 0)) { //不匹配
        output = (char *) malloc(sizeof(char));
        *output = 0; // \0
        return output;
    }
    for (i = 0; i < len;) {
        if (match[i] == match[max_pos]) { //找到匹配
            output = str_join(output, input + i, match[i]);
            i += match[i];
        }
        else {
            ++i;
        }
    }
    free(match);
    return output;
}

```

2、本题也可以直接写出 DP 方程，如下代码所示：

```

//copyright@chpeih 2013/4/23
char* my_find(char input[], char rule[])
{
    //write your code here
    int len1,len2;
    for(len1 = 0;input[len1];len1++);
    for(len2 = 0;rule[len2];len2++);
    int MAXN = len1>len2?(len1+1):(len2+1);
    int **dp;

    //dp[i][j]表示字符串1和字符串2分别以 i j 结尾匹配的最大长度
    //记录 dp[i][j] 是由之前那个节点推算过来 i*MAXN+j
    dp = new int *[len1+1];
    for (int i = 0;i<=len1;i++)
    {
        dp[i] = new int[len2+1];
    }

    dp[0][0] = 0;
    for(int i = 1;i<=len2;i++)
        dp[0][i] = -1;
    for(int i = 1;i<=len1;i++)
        dp[i][0] = 0;

    for (int i = 1;i<=len1;i++)
    {
        for (int j = 1;j<=len2;j++)
        {
            if(rule[j-1]=='*'){
                dp[i][j] = -1;
                if (dp[i-1][j-1]!=-1)
                {
                    dp[i][j] = dp[i-1][j-1]+1;
                }
                if (dp[i-1][j]!=-1 && dp[i][j]<dp[i-1][j]+1)
                {
                    dp[i][j] = dp[i-1][j]+1;
                }
            }else if (rule[j-1]=='?')
            {
                if(dp[i-1][j-1]!=-1){

```

```

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

    }else dp[i][j] = -1;
}
else
{
    if(dp[i-1][j-1]!=-1 && input[i-1]==rule[j-1]){
        dp[i][j] = dp[i-1][j-1]+1;
    }else dp[i][j] = -1;
}
}

int m = -1;//记录最大字符串长度
int *ans = new int[len1];
int count_ans = 0;//记录答案个数
char *returnans = new char[len1+1];
int count = 0;
for(int i = 1;i<=len1;i++)
{
    if (dp[i][len2]>m){
        m = dp[i][len2];
        count_ans = 0;
        ans[count_ans++] = i-m;
    }else if(dp[i][len2]!=-1 && dp[i][len2]==m){
        ans[count_ans++] = i-m;
    }
}

if (count_ans!=0)
{
    int len = ans[0];
    for (int i = 0;i<m;i++)
    {
        printf("%c",input[i+ans[0]]);
        returnans[count++] = input[i+ans[0]];
    }
    for (int j = 1;j<count_ans;j++)
    {
        printf(" ");
        returnans[count++] = ' ';
        len = ans[j];
        for (int i = 0;i<m;i++)
        {
            printf("%c",input[i+ans[j]]);
            returnans[count++] = input[i+ans[j]];
        }
    }
}

```

```

        }
    }

    printf("\n");
    returnans[count++] = '\0';
}

return returnans;
}

```

欢迎于本文评论下或 hero 上 show your code。

## 参考文献及推荐阅读

1. <http://zhedahht.blog.163.com/blog/static/25411174200731139971/>;
2. <http://hero.pongo.cn/>, 本文大部分代码都取自左边 hero 上参与答题者提交的代码, 欢迎你也去挑战;
3. 字符串转换成整数题目完整描述:  
<http://hero.pongo.cn/Question/Details?ID=47&ExamID=45;>
4. 字符串匹配问题题目完整描述:  
<http://hero.pongo.cn/Question/Details?ID=28&ExamID=28;>
5. linux3.8.4 版本下的相关字符串整数转换函数概览:  
<https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/lib/vsprintf.c?id.refs/tags/v3.9.4>;
6. 关于 linux 中的 likely 和 unlikely:  
<http://blog.21ic.com/user1/5593/archives/2010/68193.html>;
7. 如果你喜欢编程挑战, 除了 topcoder 和 hero, 你应该还多去 leetcode 上逛逛:  
<http://leetcode.com/onlinejudge>;
8. atoi 函数的实现: [http://blog.csdn.net/njnu\\_mjn/article/details/9099405](http://blog.csdn.net/njnu_mjn/article/details/9099405);
9. atoi 函数的实现: linux 内核 atoi 函数的测试:  
[http://blog.csdn.net/njnu\\_mjn/article/details/9104143](http://blog.csdn.net/njnu_mjn/article/details/9104143);
10. Nut/OS 中 atoi 函数的实现: [http://www.ethernut.de/api/atoi\\_8c\\_source.html](http://www.ethernut.de/api/atoi_8c_source.html);
11. 一读者写的 hero 上“字符串转换成整数”一题的解题报告 (测试正确):  
<http://blog.csdn.net/u011070134/article/details/9116831>;

# 第三十二~三十三章：最小操作数，木块砌墙问题

作者：July、caopengcs、红色标记。致谢：fuwutu、demo。

时间：二零一三年八月十二日

## 题记

再过一两月，便又到了每年的九月十月校招高峰期，在此依次推荐：

1. 程序员编程艺术 <http://blog.csdn.net/column/details/taopp.html>;
2. 秒杀 99% 的海量数据处理面试题  
[http://blog.csdn.net/v\\_july\\_v/article/details/7382693](http://blog.csdn.net/v_july_v/article/details/7382693);
3. 《编程之美》;
4. 微软面试 100 题系列 <http://blog.csdn.net/column/details/ms100.html>;
5. 《剑指 offer》

一年半前在学校的那会，我曾经无比疯狂的创作程序员编程艺术这个系列，因为当时我坚信它能帮到更多的人找到更好的工作，此刻今后，我更加无比坚信这点。

同时，相信你也已看到，编程艺术系列的创作原则是把受众定位为一个编程初学者，从看到问题后最先想到的思路开始讲解，一点一点改进，不断优化。

而本文主要讲下述两个问题：

- 第三十二章：最小操作数问题，主要由 caopengcs 完成；
- 第三十三章：木块砌墙问题，主要由红色标记和 caopengcs 完成。

全文由 July 统一整理修订完成。OK，还是很真诚的那句话：有任何问题，欢迎读者随时批评指正，感谢。

## 第三十二章、最小操作数

题目详情如下：

给定一个单词集合 **Dict**, 其中每个单词的长度都相同。现从此单词集合 **Dict** 中抽取两个单词 **A**、**B**, 我们希望通过若干次操作把单词 **A** 变成单词 **B**, 每次操作可以改变单词的一个字母, 同时, 新产生的单词必须是在给定的单词集合 **Dict** 中。求所有行得通步数最少的修改方法。

举个例子如下：

Given:

**A** = "hit"

**B** = "cog"

**Dict** = ["hot", "dot", "dog", "lot", "log"]

Return

```
[  
    ["hit", "hot", "dot", "dog", "cog"],  
    ["hit", "hot", "lot", "log", "cog"]  
]
```

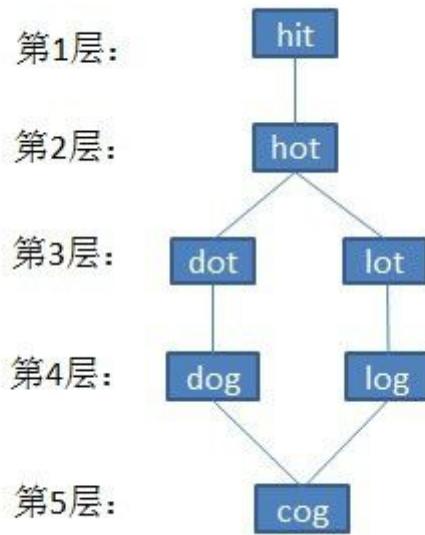
即把字符串 **A** = "hit" 转变成字符串 **B** = "cog", 有以下两种可能：

"hit" -> "hot" -> "dot" -> "dog" -> "cog";

"hit" -> "hot" -> "lot" -> "log" -> "cog"。

**详解：**本题是一个典型的图搜索算法问题。此题看似跟本系列的第 29 章的字符串编辑距离相似, 但其实区别特别大, 原因是最短编辑距离是让某个单词增加一个字符或减少一个字符或修改一个字符达到目标单词, 来求变换的最少次数, 但此最小操作数问题就只是改变一个字符。

通过此文：[http://blog.csdn.net/v\\_JULY\\_v/article/details/6111353](http://blog.csdn.net/v_JULY_v/article/details/6111353), 我们知道, 在图搜索算法中, 有深度优先遍历 **DFS** 和广度优先遍历 **BFS**, 而题目中并没有给定图, 所以需要我们自己建立图。



涉及到图就有这么几个问题要思考，节点是什么？边如何建立？图是有方向的还是无方向的？包括建好图之后，如何记录单词序列等等都是我们要考虑的问题。

### 解法一、单向 BFS 法

#### 1、建图

对于本题，我们的图的节点就是字典里的单词，两个节点有连边，对应着我们可以把一个单词按照规则变为另外一个单词。比如我们有单词 `hat`，它应该与单词 `cat` 有一条连边，因为我们可以把 `h` 变为 `c`，反过来我们也可以把 `c` 变为 `h`，所以我们建立的连边应该是无向的。

如何建图？有两种办法，

- 第一种方法是：我们可以把字典里的任意两个单词，通过循环判断一下这两个单词是否只有一个位置上的字母不同。即假设字典里有  $n$  个单词，我们遍历任意两个单词的复杂度是  $O(n^2)$ ，如果每个单词长度为 `length`，我们判断两个单词是否连边的复杂度是  $O(length)$ ，所以这个建图的总复杂度是  $O(n^2 * length)$ 。但当  $n$  比较大时，这个复杂度非常高，有没有更好的方法呢？
- 第二种方法是：我们把字典里的每个单词的每个位置的字母修改一下，从字典里查找一下（若用基于 `red-black tree` 的 `map` 查找，其查找复杂度为  $O(log n)$ ，若用基于 `hashmap` 的 `unordered_map`，则查找复杂度为  $O(1)$ ），修改后的单词是否在字典里出现过。即我们需要遍历字典里的每一个单词  $O(n)$ ，尝试修改每个位置的每个字母，对每个位置我

们需要尝试 26 个字母（其实是 25 个，因为要改得和原来不同），因此这部分复杂度是  $O(26 * \text{length})$ ，总复杂度是  $O(26 * n * \text{length})$  （第二种方法优化版：这第二种方法能否更优？在第二种方法中，我们对每个单词每个位置尝试了 26 次修改，事实上我们可以利用图是无向的这一特点，我们对每个位置试图把该位置的字母变到字典序更大的字母。例如，我们只考虑 cat 变成 hat，而不考虑 hat 变成 cat，因为再之前已经把无向边建立了。这样，只进行一半的修改次数，从而减少程序的运行时间。当然这个优化从复杂度上来讲是常数的，因此称为常数优化，此虽算是一种改进，但不足以成为第三种方法，原因是经常忽略 O 背后隐藏的常数）。

OK，上面两种方法孰优孰劣呢？直接比较  $n^2 * \text{length}$  与  $26 * n * \text{length}$  的大小。很明显，通常情况下，字典里的单词个数非常多，也就是  $n$  比较大，因此第二种方法效果会好一些，稍后的参考代码也会选择上述第二种方法的优化。

## 2、记录单词序列

对于最简单的 bfs，我们是如何记录路径的？如果只需要记录一条最短路径的话，我们可以对每个走到的位置，记录走到它的前一个位置。这样到终点后，我们可以不断找到它的前一个位置。我们利用了最短路径的一个特点：即第二次经过一个节点的时候，路径长度不比第一次经过它时短。因此这样的路径是没有圈的。

但是本题需要记录全部的路径，我们第二次经过一个节点时，路径长度可能会和第一次经过一个节点时路径长度一样。这是因为，我们可能在第  $i$  层中有多个节点可以到达第  $(i + 1)$  层的同一个位置，这样那个位置有多条路径都是最短路径。

如何解决呢？——我们记录经过这个位置的前面所有位置的集合。这样一个节点的前驱不是一个节点，而是一个节点的集合。如此，当我们第二次经过一个第  $(i + 1)$  层的位置时，我们便保留前面那第  $i$  层位置的集合作为前驱。

## 3、遍历

解决了以上两个问题，我们最终得到的是什么？如果有解的话，我们最终得到的是从终点开始的前一个可能单词的集合，对每个单词，我们都有能得到它的上一个单词的集合，直到起点。这就是 bfs 分层之后的图，我们从终点开始遍历这个图的到起点的所有路径，就得到了所有的解，这个遍历我们可以采用之前介绍的 dfs 方法（路径的数目可能非常多）。

其实，为了简单起见，我们可以从终点开始 bfs，因为记录路径记录的是之前的节点，也就是反向的。这样最终可以按顺序从起点遍历到终点的所有路径。

参考代码如下：

```
//copyright@caopengcs
//updated@July 08/12/2013
class Solution
{
public:
    // help 函数负责找到所有的路径
    void help(int x, vector<int> &d, vector<string> &word, vector<vector<int>> &next,
    vector<string> &path, vector<vector<string>> &answer) {
        path.push_back(word[x]);
        if (d[x] == 0) { //已经到达终点了
            answer.push_back(path);
        }
        else {
            int i;
            for (i = 0; i < next[x].size(); ++i) {
                help(next[x][i], d, word, next, path, answer);
            }
        }
        path.pop_back(); //回溯
    }

    vector<vector<string>> findLadders(string start, string end, set<string>& dict)
    {

        vector<vector<string>> answer;
        if (start == end) { //起点终点恰好相等
            return answer;
        }
        //把起点终点加入字典的 map
        dict.insert(start);
        dict.insert(end);
        set<string>::iterator dt;
        vector<string> word;
        map<string, int> allword;
        //把 set 转换为 map, 这样每个单词都有编号了。
        for (dt = dict.begin(); dt != dict.end(); ++dt) {
            word.push_back(*dt);
            allword.insert(make_pair(*dt, allword.size()));
        }

        //建立连边 邻接表
```

```

vector<vector<int>> con;
int i,j,n = word.size(), temp, len = word[0].length();
con.resize(n);
for (i = 0; i < n; ++i) {
    for (j = 0; j < len; ++j) {
        char c;
        for (c = word[i][j] + 1; c <= 'z'; ++c) { //根据上面第二种方法的优化
版的思路，让每个单词每个位置变更大
            char last = word[i][j];
            word[i][j] = c;
            map<string,int>::iterator t = allword.find(word[i]);
            if (t != allword.end()) {
                con[i].push_back(t->second);
                con[t->second].push_back(i);
            }
            word[i][j] = last;
        }
    }
}

//以下是标准 bfs 过程
queue<int> q;
vector<int> d;
d.resize(n, -1);
int from = allword[start], to = allword[end];
d[to] = 0; //d 记录的是路径长度, -1 表示没经过
q.push(to);
vector<vector<int>> next;
next.resize(n);
while (!q.empty()) {
    int x = q.front(), now = d[x] + 1;
    //now 相当于路径长度
    //当 now > d[from] 时，则表示所有解都找到了
    if ((d[from] >= 0) && (now > d[from])) {
        break;
    }
    q.pop();
    for (i = 0; i < con[x].size(); ++i) {
        int y = con[x][i];
        //第一次经过 y
        if (d[y] < 0) {
            d[y] = now;
            q.push(y);
        }
    }
}

```

```

        next[y].push_back(x);
    }
    //非第一次经过 y
    else if (d[y] == now) { //是从上一层经过的，所以要保存
        next[y].push_back(x);
    }

}
if (d[from] >= 0) { //有解
    vector<string>path;
    help(from, d, word, next, path, answer);
}
return answer;
}
};


```

## 解法二、双向 BFS 法

BFS 需要把每一步搜到的节点都存下来，很有可能每一步的搜到的节点个数越来越多，但最后的目的节点却只有一个。后半段的很多搜索都是白耗时间了。

上面给出了单向 BFS 的解法，但看过此前 blog 中的这篇文章“[A\\*、Dijkstra、BFS 算法性能比较演示](#)”可知：[http://blog.csdn.net/v\\_JULY\\_v/article/details/6238029](http://blog.csdn.net/v_JULY_v/article/details/6238029)，双向 BFS 性能优于单向 BFS。

举个例子如下，第 1 步，是起点，1 个节点，第 2 步，搜到 2 个节点，第 3 步，搜到 4 个节点，第 4 步搜到 8 个节点，第 5 步搜到 16 个节点，并且有一个是终点。那这里共出现了 31 个节点。从起点开始广搜的同时也从终点开始广搜，就有可能在两头各第 3 步，就相遇了，出现的节点数不超过 $(1+2+4)*2=14$  个，如此就节省了一半以上的搜索时间。

下面给出双向 BFS 的解法，参考代码如下：

```

//copyright@fuwutu 6/26/2013
class Solution
{
public:
    vector<vector<string>> findLadders(string start, string end, set<string>& dict)

    {
        vector<vector<string>> result, result_temp;
        if (dict.erase(start) == 1 && dict.erase(end) == 1)
        {

```

```

map<string, vector<string>> kids_from_start;
map<string, vector<string>> kids_from_end;

set<string> reach_start;
reach_start.insert(start);
set<string> reach_end;
reach_end.insert(end);

set<string> meet;
while (meet.empty() && !reach_start.empty() && !reach_end.empty())
{
    if (reach_start.size() < reach_end.size())
    {
        search_next_reach(reach_start, reach_end, meet, kids_from_start,
dict);
    }
    else
    {
        search_next_reach(reach_end, reach_start, meet, kids_from_end,
dict);
    }
}

if (!meet.empty())
{
    for (set<string>::iterator it = meet.begin(); it != meet.end(); ++it)
    {
        vector<string> words(1, *it);
        result.push_back(words);
    }

    walk(result, kids_from_start);
    for (size_t i = 0; i < result.size(); ++i)
    {
        reverse(result[i].begin(), result[i].end());
    }
    walk(result, kids_from_end);
}
}

return result;
}

```

```

private:
    void search_next_reach(set<string>& reach, const set<string>& other_reach, se
t<string>& meet, map<string, vector<string>>& path, set<string>& dict)
    {
        set<string> temp;
        reach.swap(temp);

        for (set<string>::iterator it = temp.begin(); it != temp.end(); ++it)
        {
            string s = *it;
            for (size_t i = 0; i < s.length(); ++i)
            {
                char back = s[i];
                for (s[i] = 'a'; s[i] <= 'z'; ++s[i])
                {
                    if (s[i] != back)
                    {
                        if (reach.count(s) == 1)
                        {
                            path[s].push_back(*it);
                        }
                        else if (dict.erase(s) == 1)
                        {
                            path[s].push_back(*it);
                            reach.insert(s);
                        }
                        else if (other_reach.count(s) == 1)
                        {
                            path[s].push_back(*it);
                            reach.insert(s);
                            meet.insert(s);
                        }
                    }
                }
                s[i] = back;
            }
        }

        void walk(vector<vector<string>>& all_path, map<string, vector<string>> kids)
        {
            vector<vector<string>> temp;
            while (!kids[all_path.back().back()].empty())

```

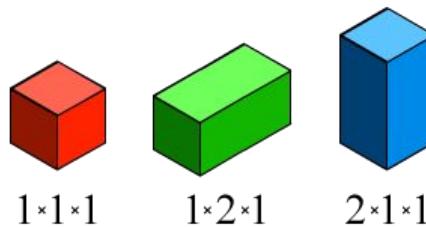
```

{
    all_path.swap(temp);
    all_path.clear();
    for (vector<vector<string>>::iterator it = temp.begin(); it != temp.end(); ++it)
    {
        vector<string>& one_path = *it;
        vector<string>& p = kids[one_path.back()];
        for (size_t i = 0; i < p.size(); ++i)
        {
            all_path.push_back(one_path);
            all_path.back().push_back(p[i]);
        }
    }
}
};

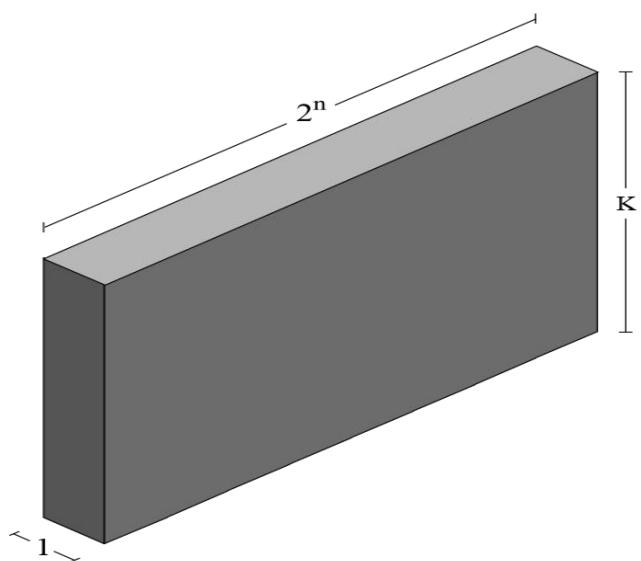

```

## 第三十三章、木块砌墙

题目：用  $1 \times 1 \times 1$ ,  $1 \times 2 \times 1$  以及  $2 \times 1 \times 1$  的三种木块（横绿竖蓝，且绿蓝长度均为 2），



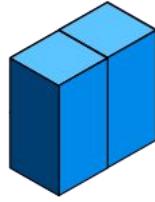
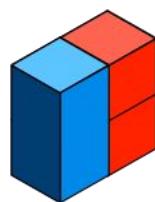
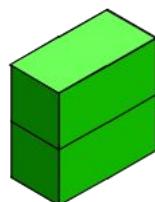
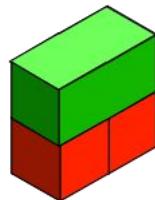
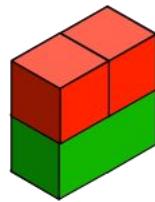
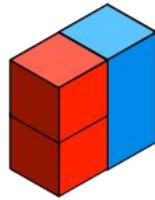
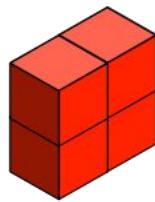
搭建高长宽分别为  $K \times 2^N \times 1$  的墙，不能翻转、旋转（其中， $0 \leq N \leq 1024$ ,  $1 \leq K \leq 4$ ）



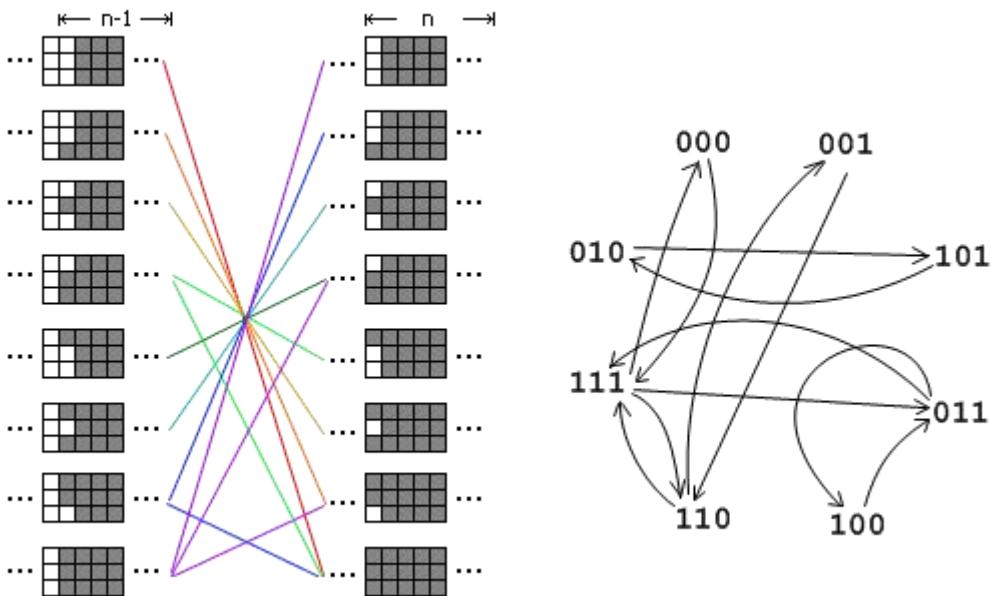
有多少种方案，输出结果

对 1000000007 取模。

举个例子如给定高度和长度：N=1 K=2，则答案是 7，即有 7 种搭法，如下图所示：



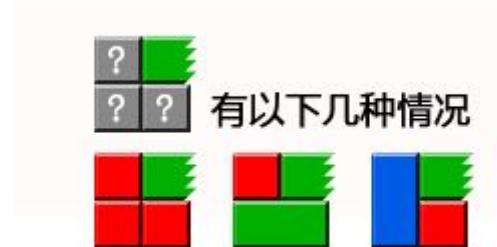
**详解:** 此题很有意思，涉及的知识点也比较多，包括动态规划，快速矩阵幂，状态压缩，排列组合等等都一一考察了个遍。而且跟一个比较经典的矩阵乘法问题类似：即用  $1 \times 2$  的多米诺骨牌填满  $M \times N$  的矩形有多少种方案， $M \leq 5$ ,  $N \leq 2^{31}$ ，输出答案  $\text{mod } p$  的结果



OK, 回到正题。下文使用的图示说明(所有看到的都是横切面):

	该位置是空位置		该位置摆放2x1横条
	该位置摆放1x1方格		该位置摆放2x1左边部分
	该位置摆放1x2方格		该位置摆放2x1右边部分

首先说明“? 方块”的作用



“? 方块”，表示这个位置是空位置，可以任意摆放。

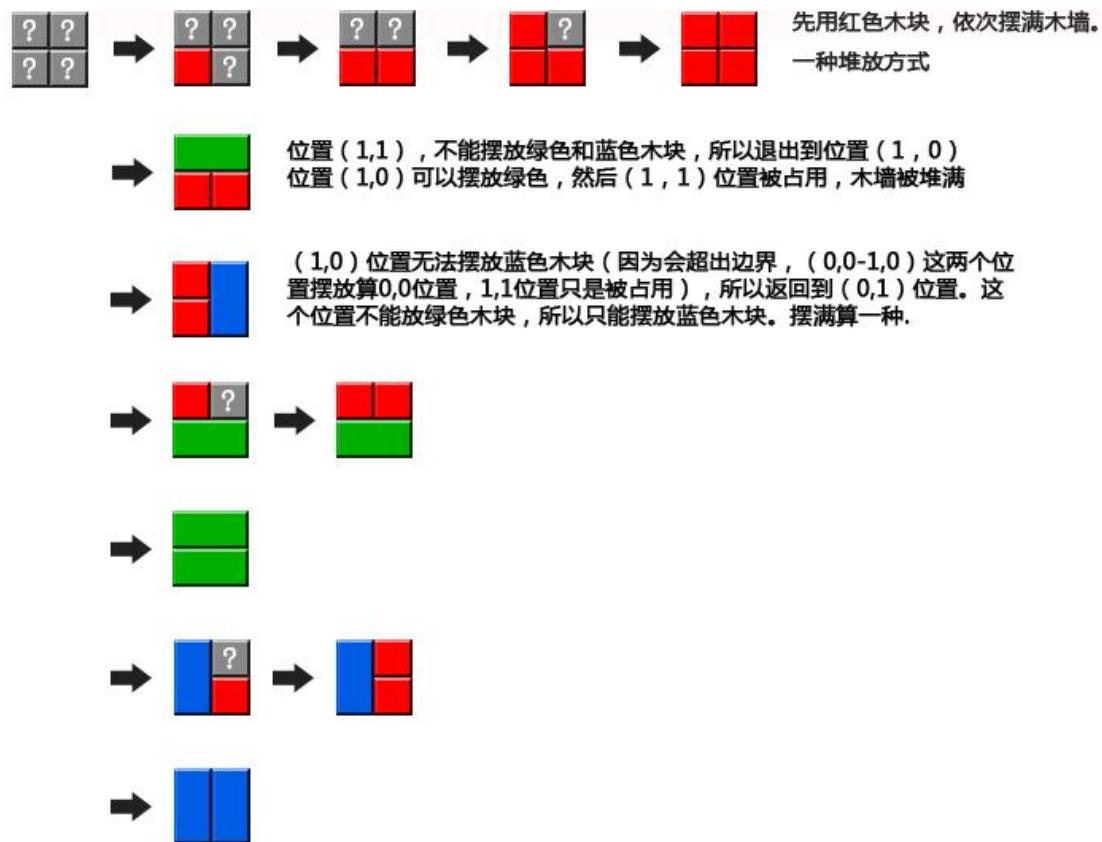
上图的意思就是，当右上角被绿色木块占用，此位置固定不变，其他位置任意摆放，在这种情况下的堆放方案数。

## 解法一、穷举遍历

初看此题，你可能最先想到的思路便是穷举：用二维数组模拟墙，从左下角开始摆放，从左往右，从下往上，最后一个格子是右上角那个位置；每个格子把每种可以摆放木块都摆放一次，每堆满一次算一种用摆放方法。为了便于描述，为木块的每个格子进行编号：



下面演示当  $n=1, k=2$  的算法过程（7 种情况）：



穷举遍历在数据规模比较小的情况下还撑得住，但在  $0 \leq N \leq 1024$  这样的数据规模下，此方法则立刻变得有心无力，因此我们得寻找更优化的解法。

## 解法二、递归分解

递归求解就是把一个大问题，分解成小问题，逐个求解，然后再解决大问题。

### 2.1、算法演示

假如有墙规模为  $(n, k)$ ，如果从中间切开，被分为规模问  $(n-1, k)$  的两堵墙，那么被分开的墙和原墙有什么关系呢？我们首先来看一下几组演示。

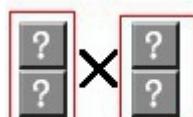
#### 2.1.1、 $n=1, k=2$ 的情况

首先演示， $n=1, k=2$  时的情况，如下图 2-1：

$$\begin{aligned} \begin{array}{|c|c|} \hline ? & ? \\ \hline ? & ? \\ \hline \end{array} & = \begin{array}{|c|} \hline ? \\ \hline ? \\ \hline \end{array} \times \begin{array}{|c|} \hline ? \\ \hline ? \\ \hline \end{array} + \begin{array}{|c|} \hline ? \\ \hline ? \\ \hline \end{array} \times \begin{array}{|c|} \hline ? \\ \hline ? \\ \hline \end{array} + \begin{array}{|c|} \hline ? \\ \hline ? \\ \hline \end{array} \times \begin{array}{|c|} \hline ? \\ \hline ? \\ \hline \end{array} + \begin{array}{|c|} \hline ? \\ \hline ? \\ \hline \end{array} \times \begin{array}{|c|} \hline ? \\ \hline ? \\ \hline \end{array} \\ & = 2 \times 2 + 1 \times 1 + 1 \times 1 + 1 \times 1 \\ & = 7 \end{aligned}$$

图 2-1

上图 2-1 中：



表示，左边墙的所有堆放方案数 \* 右边墙所有堆放方案数  $= 2 * 2 = 4$



表示，当切开处有一个横条的时候，空位置存在的堆放方案数。左边\*右边  $= 1 * 1 = 2$ ；剩余两组以此类推。

这个是排列组合的知识。

### 2.1.2、n=2, k=3 的情况

其次，我们再来演示下面更具一般性的计算分解，即当 n=2,k=3 的情况，如下图 2-2：

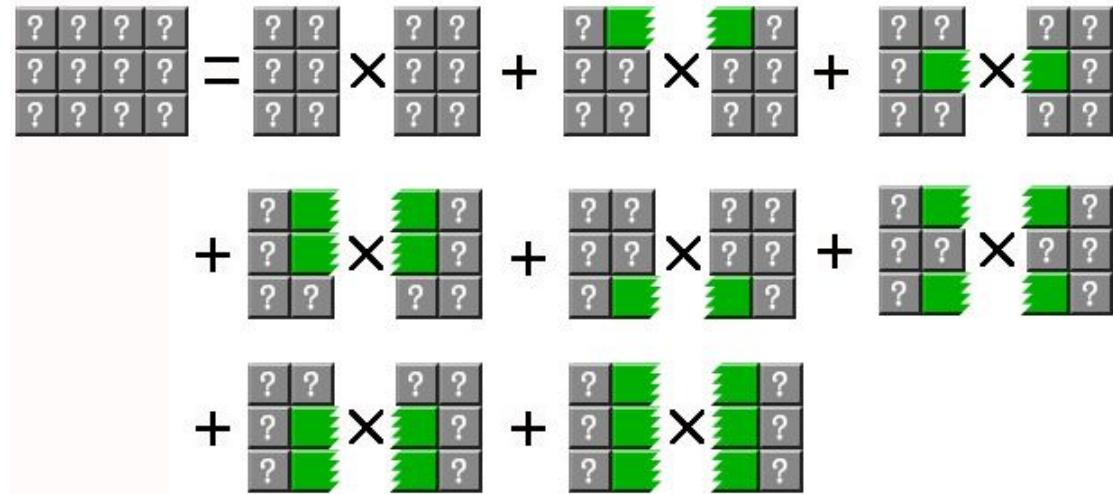


图 2-2

再从分解的结果中，挑选一组进行分解演示：

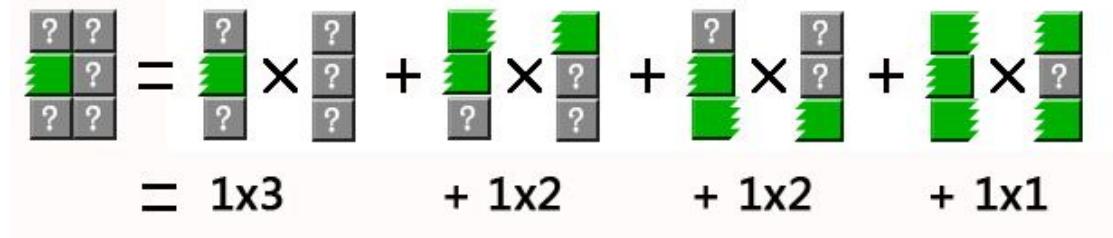


图 2-3

通过图 2-2 和图 2-3 的分解演示，可以说明，最终都是分解成一列求解。在逐级向上汇总。

### 2.1.3、n=4, k=3 的情况

我们再假设一堵墙 n=4, k=3，也就是说，宽度是 16，高度是 3 时，会有以下分解：

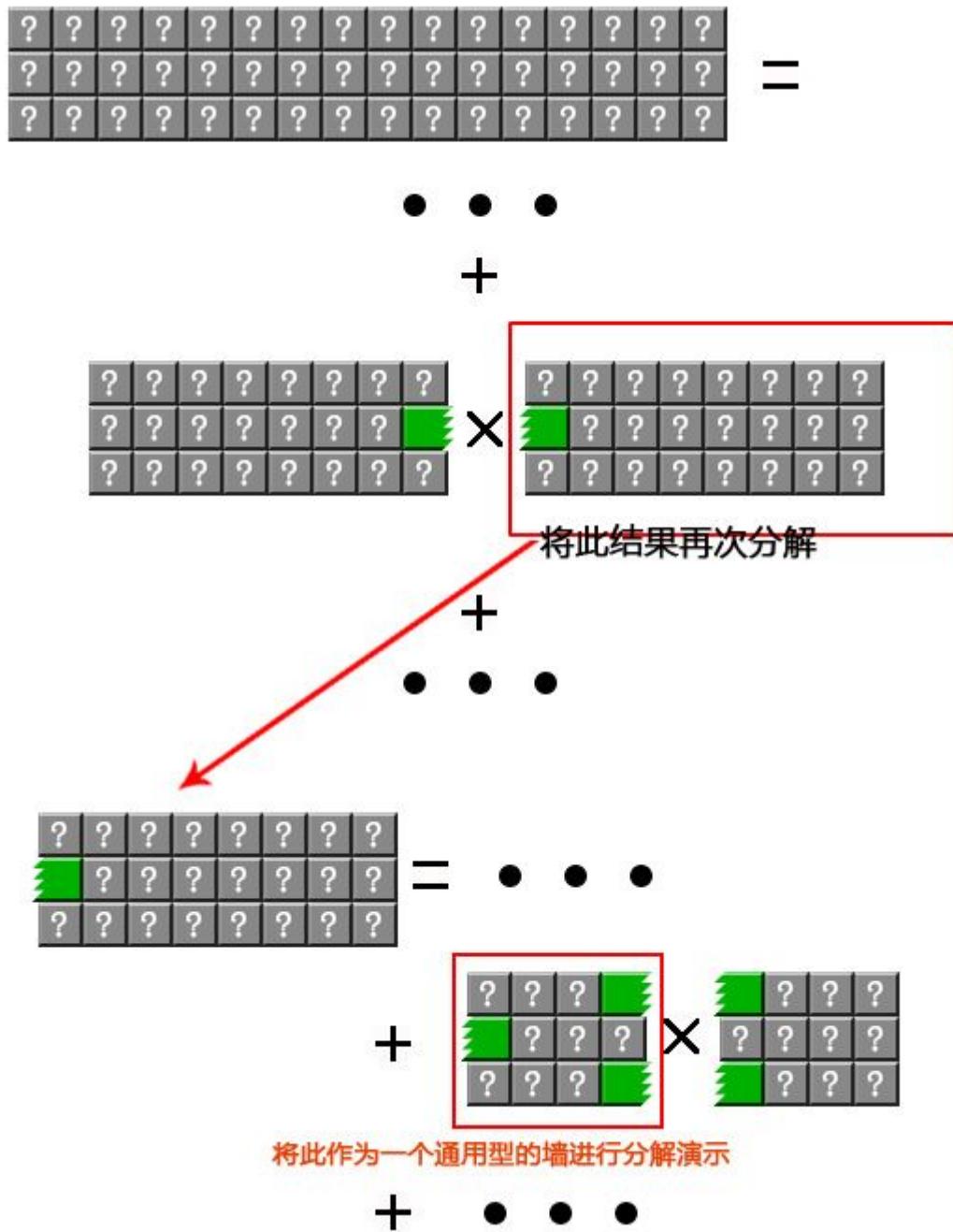


图 2-4

根据上面的分解的一个中间结果，再进行分解，如下：

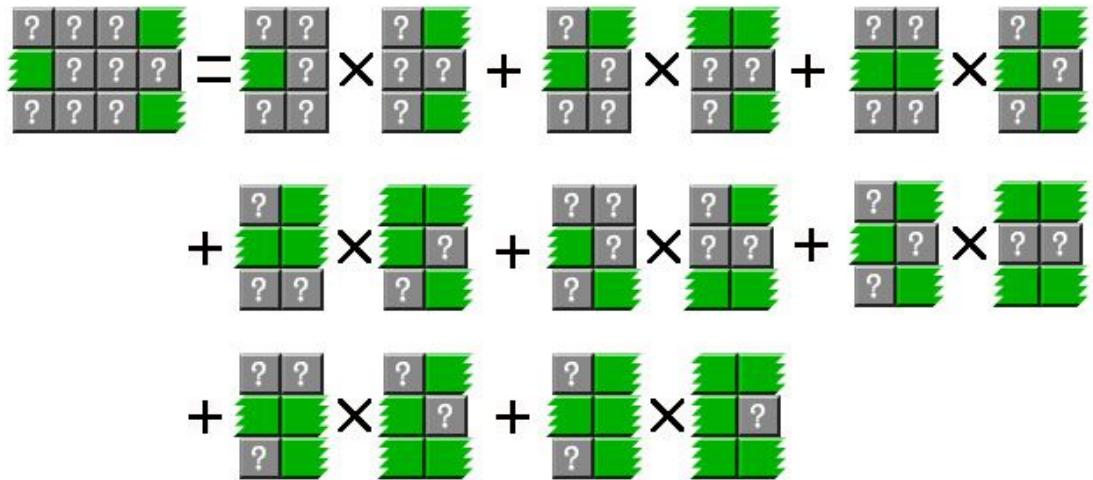


图 2-5

通过上面图 2-1~图 2-5 的演示可以明确如下几点：

1. 假设  $f(n)$  用于计算问题，那么  $f(n)$  依赖于  $f(n-1)$  的多种情况。
2. 切开处有什么特殊的地方呢？通过上面的演示，我们得知被切开的两堵墙从没有互相嵌入的木块（绿色木块）到全是互相连接的木块，相当于切口绿色木块的全排列（即有绿色或者没有绿色的所有排列），即有  $2^k$  种状态（比如  $k=2$ ，且有绿色用 1 表示，没有绿色用 0 表示，那么就有 00、01、10、11 这 4 种状态）。根据排列组合的性质，把每一种状态下左右木墙堆放方案数相乘，再把所有乘积求和，就得到木墙的堆放结果数。以此类推，将问题逐步往下分解即可。
3. 此外，从图 2-5 中可以看出，除了需要考虑切口绿色木块的状态，还需要考虑最左边一列和最右边一列的绿色木块状态。我们把这两种边界状态称为左边界状态和右边界状态，分别用 `leftState` 和 `rightState` 表示。

且在观察图 2-5 被切分后，所有左边的墙，他们的左边界 `ls` 状态始终保持不变，右边界 `rs` 状态从  $0 \sim \text{maxState}$ ,  $\text{maxState} = 2^k - 1$  (有绿色方块表示 1, 没有表示 0; `ls` 表示左边界状态, `rs` 表示右边界状态) :

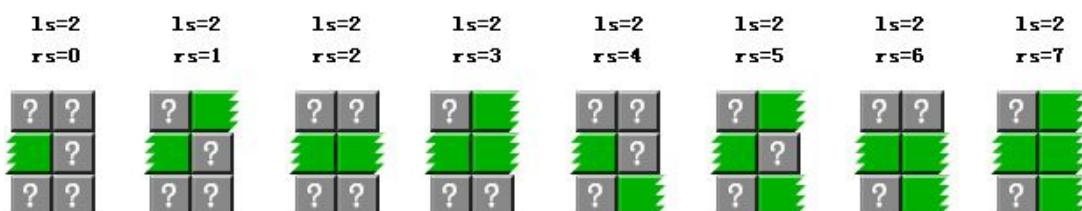


图 2-6

同样可以看出右边的墙的右边界状态保持不变，而左边界状态从 0~maxState。要堆砌的木墙可以看做是左边界状态=0，和右边界状态=0 的一堵墙。

有一点可能要特别说明下，即上文中说，有绿色方块的状态表示标为 1，无绿色方块的状态表示标为 0，特意又拿上图 2-6 标记了一些数字，以让绝大部分读者能看得一目了然，如下所示：

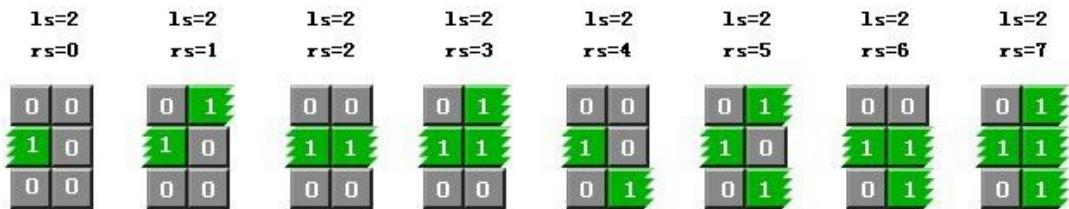


图 2-7

这下，你应该很清楚的看到，在上图中，左边木块的状态表示一律为 010，右边木块的状态表示则是 000~111（即从下至上开始计数，右边木块 rs 的状态用二进制表示为：000 001 010 011 100 101 110 111，它们各自分别对应整数则是：0 1 2 3 4 5 6 7）。

## 2.2、计算公式

通过图 2-4、图 2-5、图 2-6 的分解过程，我们可以总结出下面公式（`leftState`=最左边边界状态，`rightState`=最右边边界状态）：

$$f(n, k, leftState, rightState) = \sum_{state=0}^{2^k-1} f(n-1, k, leftState, state) * f(n-1, k, state, rightState)$$

即：

红色标记 20:33:29

- 左边墙的状态在一次分解中，左边界的 state 始终不会改变，右边从 0 ~ maxState

再看

- 右边墙的状态再一次分解中，右边界的 state 是终不会改变，左边从 0 ~ maxState

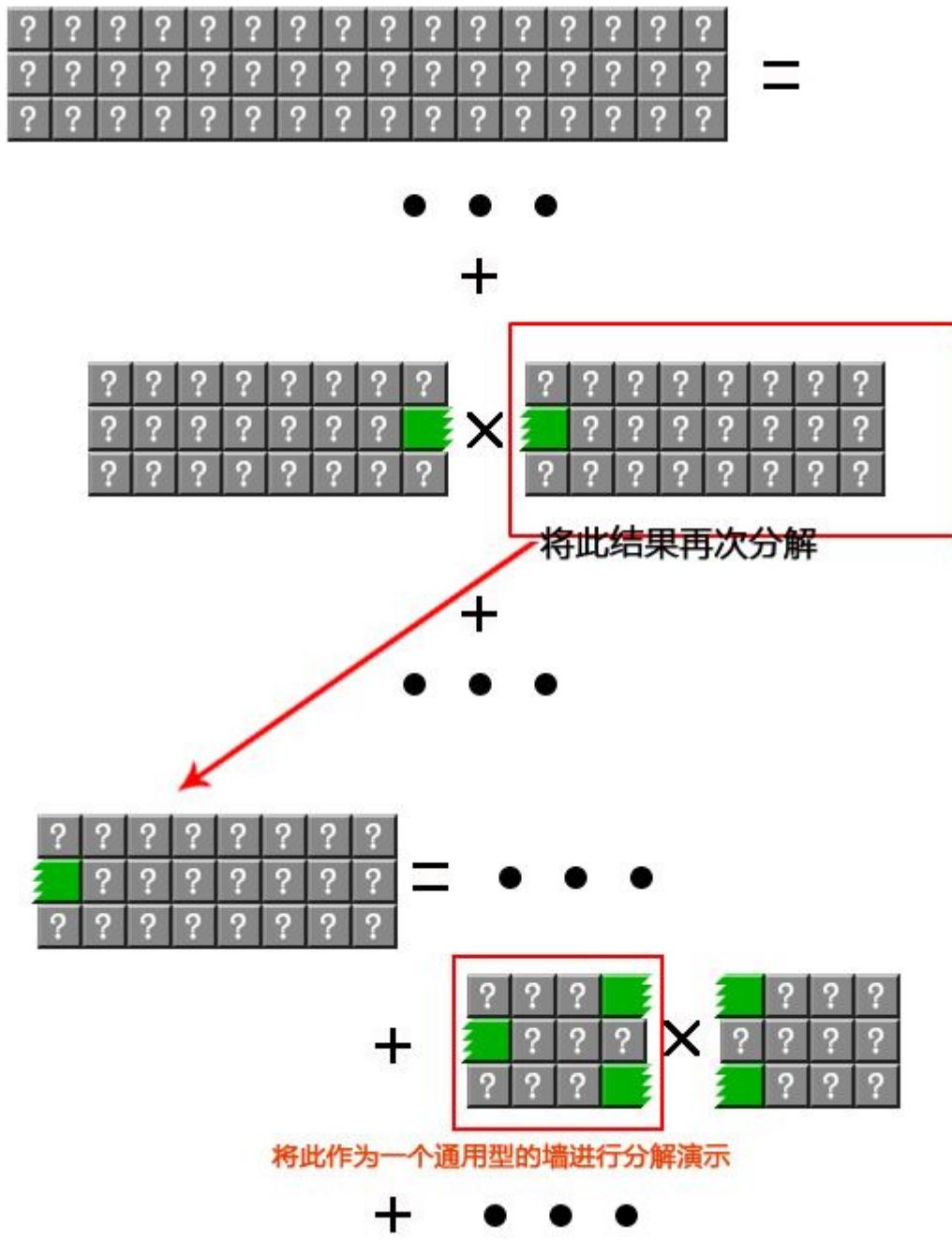
- 更具图示，可以得到公式。

红色标记 20:35:03

$$= \sum_{state=0}^{2^{k-1}} f(n-1, k, leftState, state) * f(n-1, k, state, rightState)$$

接下来，分 3 点解释下上述公式：

- 1、上述函数返回结果是当左边状态为 `=leftState`，右边状态 `=rightState` 时木墙的堆砌方案数，相当于直接分解的左右状态都为 0 的情况，即直接分解  $f(n, k, 0, 0)$  即可。看到这，读者可能便有疑问了，既然直接分解  $f(n, k, 0, 0)$  即可，为何还要加 `leftState` 和 `rightState` 两个变量呢？回顾下 2.1.3 节中  $n=4, k=3$  的演示例子，即当  $n=4, k=3$  时，其分解过程即如下图（上文 2.1.3 节中的图 2-4）

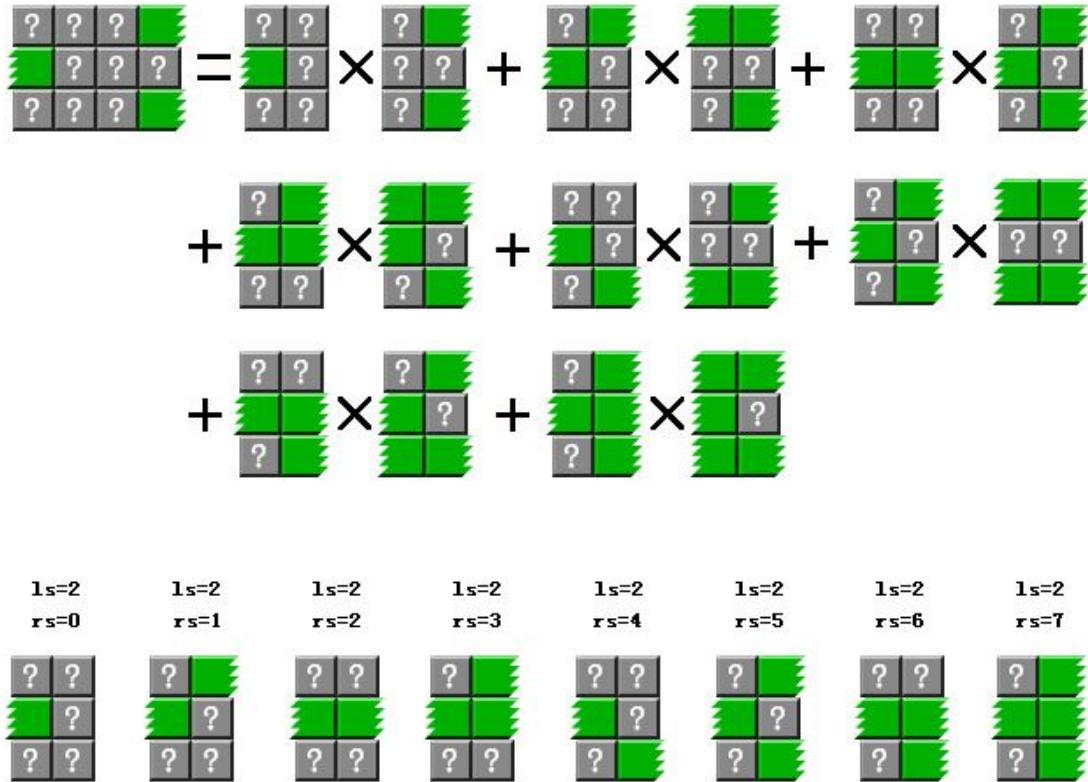


也就是说，刚开始直接分解  $f(4,3,0,0)$ ，即  $n=4$ ,  $k=3$ ,  $\text{leftstate}=0$ ,  $\text{rightstate}=0$ ，但分解过程中  $\text{leftstate}$  和  $\text{rightstate}$  皆从 0 变化到了  $\text{maxstate}$ ，故才让函数的第 3 和第 4 个参数采用  $\text{leftstate}$  和  $\text{rightstate}$  这两个变量的形式，公式也就理所当然的写成了  $f(n,k,\text{leftstate},\text{rightstate})$ 。

2、然后我们再看下当  $n=4$ ,  $k=3$  分解的一个中间结果，即给定如上图最下面部分中红色框框所框住的木块时：



它用方程表示即为  $f(2,3,2,5)$ , 怎么得来的呢? 其实还是又回到了上文 2.1.3 节中, 当  $n=2$ ,  $k=3$  时 (下图即为上文 2.1.3 节中的图 2-5 和图 2-6)



左边界  $ls$  状态始终保持不变时, 右边界  $rs$  状态从  $0 \sim maxState$ ; 右边界状态保持不变时, 而左边界状态从  $0 \sim maxState$ 。

故上述分解过程用方程式可表示为:

$$\begin{aligned}
 f(2,3,2,5) &= f(1,3,2,0) * f(1,3,0,5) \\
 &\quad + f(1,3,2,1) * f(1,3,1,5) \\
 &\quad + f(1,3,2,2) * f(1,3,2,5) \\
 &\quad + f(1,3,2,3) * f(1,3,3,5) \\
 &\quad + f(1,3,2,4) * f(1,3,4,5) \\
 &\quad + f(1,3,2,5) * f(1,3,5,5) \\
 &\quad + f(1,3,2,6) * f(1,3,6,5) \\
 &\quad + f(1,3,2,7) * f(1,3,7,5)
 \end{aligned}$$

说白了, 我们曾在 2.1 节中从图 2-2 到图 2-6 正推推导出了公式, 然上述过程中, 则又再倒推推了一遍公式进行了说明。

3、最后, 作者是怎么想到引入 `leftstate` 和 `rightstate` 这两个变量的呢? 如红色标记所

说：“因为切开后，发现绿色条，在分出不断的变化，当时也进入了死胡同，我就在想，蓝色的怎么办。后来才想明白，与蓝色无关。每一种变化就是一种状态，所以就想到了引入 leftstate 和 rightstate 这两个变量。”

## 2.3、参考代码

下面代码就是根据上面函数原理编写的。最终执行效率，n=1024, k=4 时，用时 0.2800160 秒（之前代码用的是字典作为缓存，用时在 1.3 秒左右，后来改为数组结果，性能大增）。

```
//copyright@红色标记 12/8/2013
//updated@July 13/8/2013

using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;

namespace HeapBlock
{
    public class WoolWall
    {
        private int n;
        private int height;
        private int maxState;
        private int[, ,] resultCache; //结果缓存数组

        public WoolWall(int n, int height)
        {
            this.n = n;
            this.height = height;
            maxState = (1 << height) - 1;
            resultCache = new int[n + 1, maxState + 1, maxState + 1]; //构建缓存
数组，每个值默认为 0;
        }

        /// <summary>
        /// 静态入口。计算堆放方案数。
        /// </summary>
        /// <param name="n"></param>
        /// <param name="k"></param>
        /// <returns></returns>
        public static int Heap(int n, int k)
        {
```

```

        return new WoolWall(n, k).Heap();
    }

    /// <summary>
    /// 计算堆放方案数。
    /// </summary>
    /// <returns></returns>
    public int Heap()
    {
        return (int)Heap(n, 0, 0);
    }

    private long Heap(int n, int lState, int rState)
    {
        //如果缓存数组中的值不为 0，则表示该结果已经存在缓存中。
        //直接返回缓存结果。
        if (resultCache[n, lState, rState] != 0)
        {
            return resultCache[n, lState, rState];
        }

        //在只有一列的情况，无法再进行切分
        //根据列状态计算一列的堆放方案
        if (n == 0)
        {
            return CalcOneColumnHeapCount(lState);
        }

        long result = 0;
        for (int state = 0; state <= maxState; state++)
        {
            if (n == 1)
            {
                //在只有两列的情况，判断当前状态在切分之后是否有效
                if (!StateIsAvailable(n, lState, rState, state))
                {
                    continue;
                }
                result += Heap(n - 1, state | lState, state | lState) //合并状
            }
            * Heap(n - 1, state | rState, state | rState);
        }
        else
        {
    
```

态。因为只有一列，所以 lState 和 rState 相同。

```

        result += Heap(n - 1, lState, state) * Heap(n - 1, state, rStat
e);
    }
    result %= 1000000007; //为了防止结果溢出，根据题目要求求模。
}

resultCache[n, lState, rState] = (int)result; //将结果写入缓存数组
}
}

resultCache[n, rState, lState] = (int)result; //对称的墙结果相同，所以直
接写入缓存。
return result;
}

/// <summary>
/// 根据一列的状态，计算列的堆放方案数。
/// </summary>
/// <param name="state">状态</param>
/// <returns></returns>
private int CalcOneColumnHeapCount(int state)
{
    int sn = 0; //连续计数
    int result = 1;
    for (int i = 0; i < height; i++)
    {
        if ((state & 1) == 0)
        {
            sn++;
        }
        else
        {
            if (sn > 0)
            {
                result *= CalcAllState(sn);
            }
            sn = 0;
        }
        state >>= 1;
    }
    if (sn > 0)
    {
        result *= CalcAllState(sn);
    }

    return result;
}

```

```
}

/// <summary>
/// 类似于斐波那契序列。
/// f(1)=1
/// f(2)=2
/// f(n) = f(n-1)*f(n-2);
/// 只是初始值不同。
/// </summary>
/// <param name="k"></param>
/// <returns></returns>
private static int CalcAllState(int k)
{
    return k <= 2 ? k : CalcAllState(k - 1) + CalcAllState(k - 2);
}

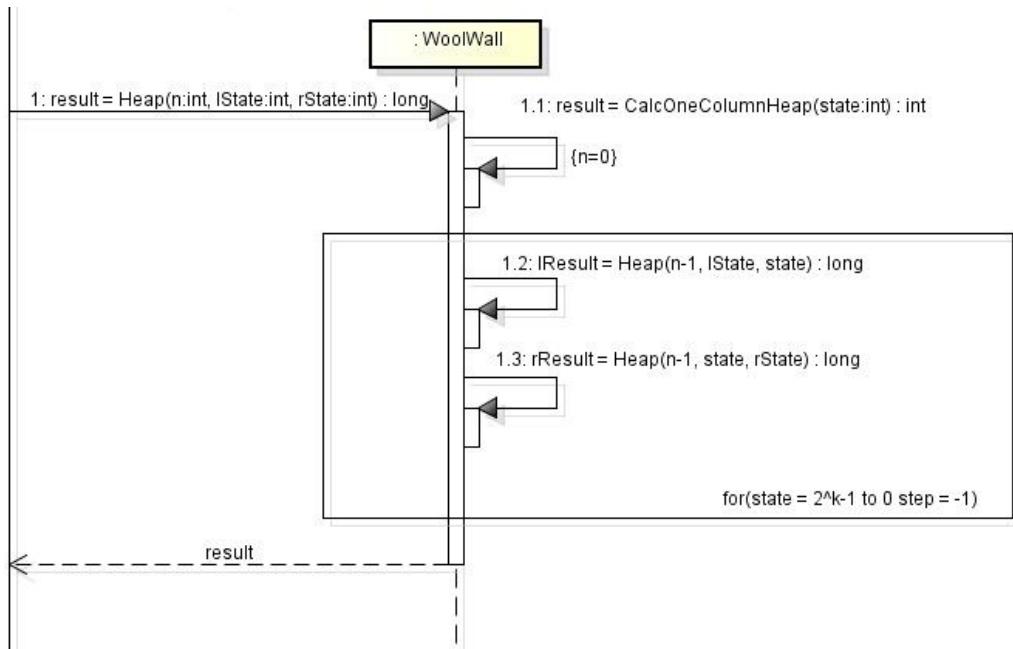
/// <summary>
/// 判断状态是否可用。
/// 当 n=1 时，分割之后，左墙和右边墙只有一列。
/// 所以 state 的状态码可能会覆盖原来的边缘状态。
/// 如果有覆盖，则该状态不可用；没有覆盖则可用。
/// 当 n>1 时，不存在这种情况，都返回状态可用。
/// </summary>
/// <param name="n"></param>
/// <param name="lState">左边界状态</param>
/// <param name="rState">右边界状态</param>
/// <param name="state">切开位置的当前状态</param>
/// <returns>状态有效返回 true, 状态不可用返回 false</returns>
private bool StateIsAvailable(int n, int lState, int rState, int state)
{
    return (n > 1) || ((lState | state) == lState + state && (rState | state) == rState + state);
}
```

上述程序中，

- `WoolWall.Heap(1024,4);` //直接通过静态方法获得结果
  - `new WoolWall(n, k).Heap();`//通过构造对象获得结果

### 2.3.1、核心算法讲解

因为它最终都是分解成一列的情况进行处理，这就会导致很慢。为了提高速度，本文使用了缓存机制来提高性能。缓存原理就是，`n,k,leftState,rightState` 相同的墙，返回的结果肯定相同。利用这个特性，每计算一种结果就放入到缓存中，如果下次计算直接从缓存取出。刚开始缓存用字典类实现，有网友给出了更好的缓存方法——数组。这样性能好了很多，也更加简单。程序结构如下图所示：



上图反应了 `Heep` 调用的主要方法调用，在循环中，`result` 累加 `lResult` 和 `rResult`。

①在实际代码中，首先是从缓存中读取结果，如果没有缓存中读取结果在进行计算。

分解法分解到一列时，不在分解，直接计算机过

```
if (n == 0)
{
    return CalcOneColumnHeap(lState);
}
```

②下面是整个程序的核心代码，通过 `for` 循环，求和 `state=0` 到 `state=2^k-1` 的两边木墙乘积：

```
for (int state = 0; state <= maxState; state++)
{
    if (n == 1)
    {
        if (!StateIsAvailable(n, lState, rState, state))
```

```

    {
        continue;
    }
    result += Heap(n - 1, state | lState, state | lState) *
        Heap(n - 1, state | rState, state | rState);
}
else
{
    result += Heap(n - 1, lState, state)
        * Heap(n - 1, state, rState);
}
result %= 1000000007;
}

```

当  $n=1$  切分时，需要特殊考虑。如下图：

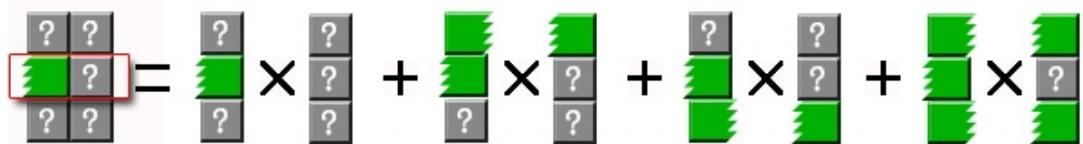


图 2-8

看上图中，因为左边墙中间被绿色方块占用，所以在  $(1,0) - (1,1)$  这个位置（位置的标记方法同解法一）不能再放绿色方块。所以一些状态需要排除，如  $\text{state}=2$  需要排除。同时在还需要合并状态，如  $\text{state}=1$  时，左边墙的状态=3。

**特别说明下：**依据我们上文 2.2 节中的公式，如果第  $i$  行有这种木块， $\text{state}$  对应  $2^{(i-1)}$ ，加上所有行的贡献就得到  $\text{state}$  ( $0$  就是没有这种横跨木块， $2^k-1$  就是所有行都是横跨木块)，然后遍历  $\text{state}$ ，还记得上文中的图 2-7 么？

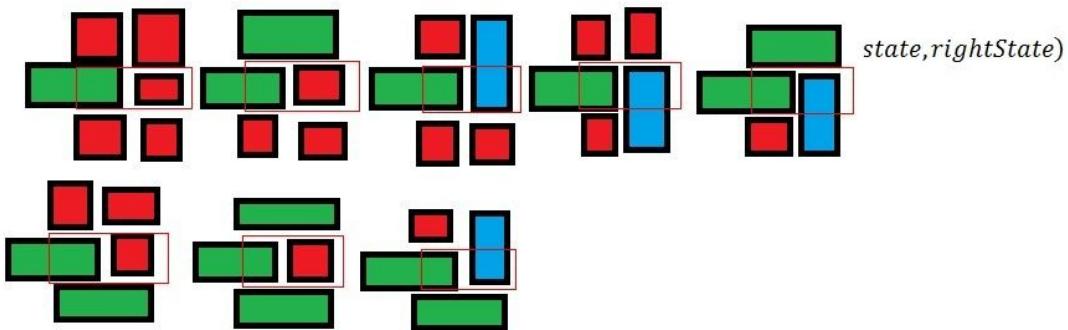
| $ls=2$ |
|--------|--------|--------|--------|--------|--------|--------|--------|
| $rs=0$ | $rs=1$ | $rs=2$ | $rs=3$ | $rs=4$ | $rs=5$ | $rs=6$ | $rs=7$ |
|        |        |        |        |        |        |        |        |
|        |        |        |        |        |        |        |        |

当第  $i$  行被这样的木块 或这样的木块 占据时，其各自对应的  $\text{state}$  值分别为：

1. 当第 1 行被占据, state=1;
2. 当第 2 行被占据, state=2;
3. 当第 1 和第 2 行都被占据, state=3;
4. 当第 3 行被占据, state=4;
5. 当第 1 和第 3 行被占据, state=5;
6. 当第 2 和第 3 行被占据, state=6;
7. 当第 1、2、3 行全部都被占据, state=7。

至于原因, 即如 2.1.3 节末所说: 二进制表示为: 000 001 010 011 100 101 110 111, 它们各自分别对应整数则是: 0 1 2 3 4 5 6 7。

具体来说, 下面图中所有框出来的位置, 不能有绿色的:



③CalcOneColumnHeap(int state)函数用于计算一列时摆放方案数。

计算方法是, 求和被绿色木块分割开的每一段连续方格的摆放方案数。每一段连续的方格的摆放方案通过 CalcAllState 方法求得。经过分析, 可以得知 CalcAllState 是类似斐波那契序列的函数。

举个例子如下 (分步骤讲述) :

1. 令 state = 4546 (state=2^k-1, k 最大为 4, 故本题中 state 最大在 15, 而这里取 state=4546 只是为了演示如何计算), 二进制是: 1000111000010。位置上为 1, 表示被绿色木块占用, 0 表示空着, 可以自由摆放。
2. 1000111000010 被分割后 1 000 111 0000 1 0, 那么就有 000=3 个连续位置, 0000=4 个连续位置, 0=1 个连续位置。
3. 堆放结果=CalcAllState(3) + CalcAllState(4) + CalcAllState(1) = 3 + 5 + 1 = 9。

## 2.4、再次优化

上面程序因为调用性能的树形结构，形成了大量的函数调用和缓存查找，所以其性能不是很高。为了得到更高的性能，可以让所有的运算直接依赖于上一次运算的结果，以防止更多的调用。即如果每次运算都算出所有边界状态的结果，那么就能为下一次运算提供足够的信息。后续优化请查阅此文第3节：

<http://blog.csdn.net/dw14132124/article/details/9038417#t2>。

## 解法三、动态规划

相信读到上文，不少读者都已经意识到这个问题其实就是一个动态规划问题，接下来咱们换一个角度来分析此问题。

### 3.1、暴力搜索不可行

首先，因为木块的宽度都是1，我们可以想成2维的问题。也就是说三种木板的规格分别为 $1 \times 1, 1 \times 2, 2 \times 1$ 。

通过上文的解法一，我们已经知道这个问题最直接的想法就是暴力搜索，即对每个空格尝试放置哪种木板。但是看看数据规模就知道，这种思路是不可行的。因为有一条边范围长度高达 $2^{1024}$ ，普通的电脑， $2^{30}$ 左右就到极限了。于是我们得想想别的方法。

### 3.2、另辟蹊径

为了方便，我们把墙看做有 $2^n$ 行， $k$ 列的矩形。这是因为虽然矩形木块不能翻转，但是我们同时拥有 $1 \times 2$ 和 $2 \times 1$ 的两种木块。

假设我们从上到下，从左到右考虑每个 $1 \times 1$ 的格子是如何被覆盖的。显然，我们每个格子都要被覆盖住。木块的特点决定了我们覆盖一个格子最多只会影响到下一行的格子。这就可以让我们暂时只考虑两行。

假设现我们已经完全覆盖了前 $(i-1)$ 行。那么由于覆盖前 $(i-1)$ 行导致第*i*行也不“完整”了。如下图：

```
XXXXXXXXX  
OOXOOXOXO
```

我们用 $X$ 表示已经覆盖的格子， $O$ 表示没覆盖的格子。为了方便，我们使用9列。

我们考虑第*i*行的状态，上图中，第1列我们可以用 $1 \times 1$ 的覆盖掉，也可以用 $1 \times 2$ 的覆盖前两列。第4、5列的覆盖方式和第1、2列是同样的情况。第7列需要覆盖也有两种方

式，即用  $1 \times 1$  的覆盖或者用  $2 \times 1$  的覆盖，但是这样会导致第  $(i+1)$  行第 7 列也被覆盖。第 9 列和第 7 列的情况是一样的。这样把第  $i$  行覆盖满了之后，我们再根据第  $(i+1)$  行被影响的状态对下一行进行覆盖。

那么每行有多少种状态呢？显然有  $2^k$ ，由于  $k$  很小，我们只有大约 16 种状态。如果我们对于这些状态之间的转换制作一个矩阵，矩阵的第  $i$  行第  $j$  列的数表示的是我们第  $m$  行是状态  $i$ ，我们把它完整覆盖掉，并且使得第  $(m+1)$  行变成状态  $j$  的可能的方法数，这个矩阵我们可以暴力搜索出来，搜索的方式就是枚举第  $m$  行的状态，然后尝试放木板，用所有的方法把第  $m$  行覆盖掉之后，下一行的状态。当然，我们也可以认为只有两行，并且第一行是  $2^k$  种状态的一种，第二行起初是空白的，求使得第一行完全覆盖掉，第二行的状态有多少种类型以及每种出现多少次。

### 3.3、动态规划

这个矩阵作用很大，其实我们覆盖的过程可以认为是这样：第一行是空的，我们看看把它覆盖了，第 2 行是什么样子的。根据第二行的状态，我们把它覆盖掉，看看第 3 行是什么样子的。

如果我们知道第  $i$  行的状态为  $s$ ，怎么考虑第  $i$  行完全覆盖后，第  $(i+1)$  行的状态？那只要看那个矩阵的状态  $s$  对应的行就可以了。我们可以考虑一下，把两个这样的方阵相乘得到得结果是什么。这个方阵的第  $i$  行第  $j$  个元素是这样得到的，是第  $i$  行第  $k$  个元素与第  $k$  行第  $j$  个元素的对  $k$  的叠加。它的意义是上一行是第  $m$  行是状态  $i$ ，把第  $m$  行和第  $(m+1)$  行同时覆盖住，第  $(m+2)$  行的状态是  $j$  的方法数。这是因为中间第  $(m+1)$  行的所有状态  $k$ ，我们已经完全遍历了。

于是我们发现，每做一次方阵的乘法，我们相当于把状态推动了一行。那么我们要坐多少次方阵乘法呢？就是题目中墙的长度  $2^n$ ，这个数太大了。但是事实上，我们可以不断地平方  $n$  次。也就是说我们可以算出  $A^2, A^4, A^8, A^{16} \dots$  方法就是不断用结果和自己相乘，这样乘  $n$  次就可以了。

因此，我们最关键的问题就是建立矩阵  $A$ 。我们可以这样表示一行的状态，从左到右分别叫做第 0 列，第 1 列……覆盖了我们认为是 1，没覆盖我们认为是 0，这样一行的状态可以表示维一个整数。某一列的状态我们可以用为运算来表示。例如，状态  $x$  第  $i$  列是否被覆盖，我们只需要判断  $x \& (1 << i)$  是否非 0 即可，或者判断  $(x >> i) \& 1$ ，用右移位的目的是防止溢出，但是本题不需要考虑溢出，因为  $k$  很小。接下来的任务就是递归尝试放置方案了。

### 3.4、参考代码

最终结果，我们最初的行是空得，要求最后一行之后也不能被覆盖，所以最终结果是矩阵的第[0][0]位置的元素。另外，本题在乘法过程中会超出 32 位 int 的表示范围，需要临时用 C/C++ 的 long long，或者 java 的 long。

参考代码如下：

```
//copyright@caopengcs 12/08/2013
#ifndef WIN32
#define ll __int64
#else
#define ll long long
#endif

// 1 covered 0 uncovered

void cal(int a[6][32][32],int n,int col,int laststate,int nowstate) {
    if (col >= n) {
        ++a[n][laststate][nowstate];
        return;
    }
    //不填 或者用 1*1 的填
    cal(a,n, col + 1, laststate, nowstate);
    if (((laststate >> col) & 1) == 0) {
        cal(a,n, col + 1, laststate, nowstate | (1 << col));
        if ((col + 1 < n) && (((laststate >> (col + 1)) & 1) == 0)) {
            cal(a,n, col + 2, laststate, nowstate);
        }
    }
}

inline int mul(ll x, ll y) {
    return x * y % 1000000007;
}

void multiply(int n,int a[][32],int b[][32]) { // b = a * a
    int i,j, k;
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            for (k = b[i][j] = 0; k < n; ++k) {
                if ((b[i][j] += mul(a[i][k],a[k][j])) >= 1000000007) {
                    b[i][j] -= 1000000007;
                }
            }
        }
    }
}
```

```

        }
    }
}

int calculate(int n,int k) {
    int i, j;
    int a[6][32][32],mat[2][32][32];
    memset(a,0,sizeof(a));
    for (i = 1; i <= 5; ++i) {
        for (j = (1 << i) - 1; j >= 0; --j) {
            cal(a,i, 0, j, 0);
        }
    }
    memcpy(mat[0], a[k],sizeof(mat[0]));
    k = (1 << k);
    for (i = 0; n; --n) {
        multiply(k, mat[i], mat[i ^ 1]);
        i ^= 1;
    }
    return mat[i][0][0];
}

```

## 参考链接及推荐阅读

1. caopengcs, 最小操作数: <http://blog.csdn.net/caopengcs/article/details/9919341>;
2. caopengcs, 木块砌墙: <http://blog.csdn.net/caopengcs/article/details/9928061>;
3. 红色标记, 木块砌墙: <http://blog.csdn.net/dw14132124/article/details/9038417>;
4. LoveHarvy, 木块砌墙: [http://blog.csdn.net/wangyan\\_boy/article/details/9131501](http://blog.csdn.net/wangyan_boy/article/details/9131501);
5. 在线编译测试木块砌墙问题:  
<http://hero.pongo.cn/Question/Details?ID=36&ExamID=36>;
6. 编程艺术第 29 章字符串编辑距离: [http://blog.csdn.net/v\\_july\\_v/article/details/8701148#t4](http://blog.csdn.net/v_july_v/article/details/8701148#t4);
7. matrix67, 十个利用矩阵乘法解决的经典题目:  
<http://www.matrix67.com/blog/archives/276>;
8. leetcode 上最小操作数一题: [http://leetcode.com/onlinejudge#question\\_126](http://leetcode.com/onlinejudge#question_126);
9. hero 上木块砌墙一题:  
[http://hero.pongo.cn/Question/Details?ExamID=36&ID=36&bsh\\_bid=273040296](http://hero.pongo.cn/Question/Details?ExamID=36&ID=36&bsh_bid=273040296);

10. 超然烟火, <http://blog.csdn.net/sunnianzhong/article/details/9326289>;

## 后记

在本文的创作过程中, caopengcs 开始学会不再自以为是了, 意识到文章应该尽量写的详细点, 很不错; 而红色标记把最初的关于木块砌墙问题的原稿给我后, 被我拉着来来回回修改了几十遍才罢休, 尤其画了不少的图, 辛苦感谢。

此外, 围绕"编程""面试""算法"3 大主题的程序员编程艺术系列  
[http://blog.csdn.net/v\\_JULY\\_v/article/details/6460494](http://blog.csdn.net/v_JULY_v/article/details/6460494), 始创作于 2011 年 4 月, 那会还在学校, 如今已写了 33 章, 今年内我会 Review 已写的这 33 章, 且继续更新, 朋友们若有发现任何问题, 欢迎随时评论于原文下或向我反馈, 我会迅速修正, 感激不尽。

July、二零一三年八月十四日。

# 第三十四~三十五章：格子取数，完美洗牌算法

作者：July、caopengcs、绿色夹克衫。致谢：西芹\_new，陈利人，Peiyush Jain，白石，zinking。  
时间：二零一三年八月二十三日。

## 题记

再过一个半月，即到 2013 年 10 月 11 日，便是本博客开通 3 周年之际，巧的是，那天刚好也是我的 25 岁生日。写博近 3 年，访问量趋近 500 万，无法确切知道帮助了多少人影响了多少人，但有些文章和一些系列是我比较喜欢的，如这三篇：从 B 树、B+树、B\*树谈到 R 树；教你如何迅速秒杀掉：99% 的海量数据处理面试题；支持向量机通俗导论（理解 SVM 的三层境界）。

以及这 2 个系列：数据挖掘十大算法系列，程序员编程艺术。

当然，还有很多文章或系列自己也比较喜欢（如微软面试 100 题系列，经典算法研究系列等等），只是上面的文章或系列更具代表性。

但若论在上述文章或系列中，哪篇文章或系列对人找工作的帮助最大，则应该是：

- 程序员编程艺术 <http://blog.csdn.net/column/details/taopp.html>,
- 秒杀 99% 的海量数据处理面试题  
[http://blog.csdn.net/v\\_july\\_v/article/details/7382693](http://blog.csdn.net/v_july_v/article/details/7382693),
- 微软面试 100 题系列 <http://blog.csdn.net/column/details/ms100.html>,

其中，尤以编程艺术系列更佳。

OK，话休絮烦，本文讲解此文 [http://blog.csdn.net/v\\_july\\_v/article/details/7974418](http://blog.csdn.net/v_july_v/article/details/7974418) 中的第 75 题、第 79 题：

- 第三十四章：格子取数问题；
- 第三十五章：完美洗牌算法的变形

若有任何问题，欢迎读者随时批评指正，感谢。

## 第三十四章、格子取数问题

**题目详情：**有  $n \times n$  个格子，每个格子里有正数或者 0，从最左上角往最右下角走，只能向下和向右，一共走两次（即从左上角走到右下角走两趟），把所有经过的格子的数加起来，求最大值  $SUM$ ，且两次如果经过同一个格子，则最后总和  $SUM$  中该格子的计数只加一次。

		向右							
		1	2	3	4	5	6	7	8
向下	1	0	0	0	0	0	0	0	0
	2	0	0	13	0	0	6	0	0
	3	0	0	0	0	7	0	0	0
	4	0	0	0	14	0	0	0	0
	5	0	21	0	0	0	4	0	0
	6	0	0	15	0	0	0	0	0
	7	0	14	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0

**题目分析：**此题是去年 2013 年搜狗的校招笔试题。初看到此题，因为要让两次走下来的路径总和最大，读者可能最初想到的思路可能是让每一次的路径都是最优的，即不顾全局，只看局部，让第一次和第二次的路径都是最优。

但问题马上就来了，虽然这一算法保证了连续的两次走法都是最优的，但却不能保证总体最优，相应的反例也不难给出，请看下图：

	3	2	
	3		
	3		
		4	
		4	
	3		

图一

3	3	2	
	3		
	3		
		4	
		4	
	3		

图二

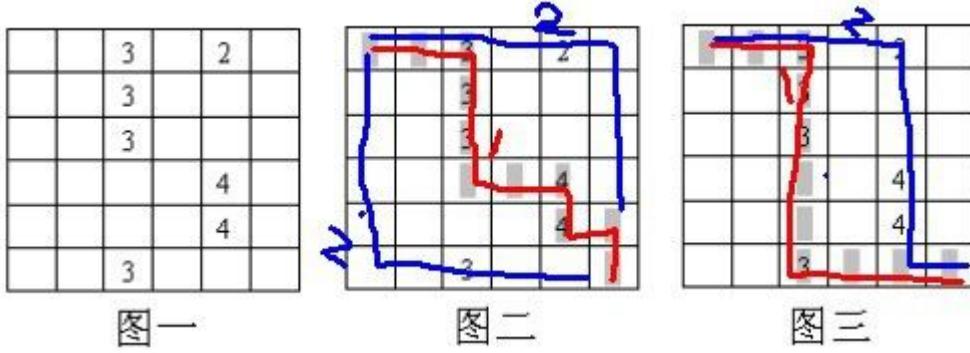
3	3	2	
	3		
	3		
		4	
		4	
	3		

图三

上图中，图一是原始图，那么我们有两种走法可供我们选择：

- 如果按照上面的局部贪优走法，那么第一次势必会如图二那样走，导致的结果是第二次要么取到 2，要么取到 3，
- 但若不按照上面的局部贪优走法，那么第一次可以如图三那样走，从而第二次走的时候能取到 2 4 4，很显然，这种走法求得的最终  $SUM$  值更大；

为了便于读者理解，我把上面的走法在图二中标记出来，而把应该正确的走法在上图三中标示出来，如下图所示：



也就是说，上面图二中的走法太追求每一次最优，所以第一次最优，导致第二次将是很差；而图三第一次虽然不是最优，但保证了第二次不差，所以图三的结果优于图二。由此可知不要只顾局部而贪图一时最优，而丧失了全局最优。

## 解法一、直接搜索

局部贪优不行，我们可以考虑穷举，但最终将导致复杂度过高，所以咱们得另寻良策。

@西芹\_new，针对此题，可以使用直接搜索法，一共搜 $(2n-2)$ 步，每一步有四种走法，考虑不相交等条件可以剪去很多枝，代码如下：

```
//copyright@西芹_new 2013
#include "stdafx.h"
#include <iostream>
using namespace std;

#define N 5
int map[5][5]={
    {2,0,8,0,2},
    {0,0,0,0,0},
    {0,3,2,0,0},
    {0,0,0,0,0},
    {2,0,8,0,2}};
int sumMax=0;
int p1x=0;
int p1y=0;
int p2x=0;
int p2y=0;
int curMax=0;

void dfs( int index){
```

```

if( index == 2*N-2){
    if( curMax>sumMax)
        sumMax = curMax;
    return;
}

if( !(p1x==0 && p1y==0) && !(p2x==N-1 && p2y==N-1))
{
    if( p1x>= p2x && p1y >= p2y )
        return;
}

//right right
if( p1x+1<N && p2x+1<N ){
    p1x++;p2x++;
    int sum = map[p1x][p1y]+map[p2x][p2y];
    curMax += sum;
    dfs(index+1);
    curMax -= sum;
    p1x--;p2x--;
}
}

//down down
if( p1y+1<N && p2y+1<N ){
    p1y++;p2y++;
    int sum = map[p1x][p1y]+map[p2x][p2y];
    curMax += sum;
    dfs(index+1);
    curMax -= sum;
    p1y--;p2y--;
}
}

//rd
if( p1x+1<N && p2y+1<N ) {
    p1x++;p2y++;
    int sum = map[p1x][p1y]+map[p2x][p2y];
    curMax += sum;
    dfs(index+1);
    curMax -= sum;
    p1x--;p2y--;
}
}

//dr
if( p1y+1<N && p2x+1<N ) {

```

```

    p1y++;p2x++;
    int sum = map[p1x][p1y]+map[p2x][p2y];
    curMax += sum;
    dfs(index+1);
    curMax -= sum;
    p1y--;p2x--;
}
}

int _tmain(int argc, _TCHAR* argv[])
{
    curMax = map[0][0];
    dfs(0);
    cout <<sumMax-map[N-1][N-1]<<endl;
    return 0;
}

```

## 解法二、动态规划

上述解法一的搜索解法是的时间复杂度是指数型的，如果是只走一次的话，是经典的 dp。

### 2.1、DP 思路详解

故正如@绿色夹克衫所说：此题也可以用动态规划求解，主要思路就是同时 DP 2 次所走的状态。

**1、先来分析一下这个问题，为了方便讨论，先对矩阵做一个编号，且以 5\*5 的矩阵为例（给这个矩阵起个名字叫 M1）：**

M1
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

从左上(0)走到右下(8)共需要走 8 步 ( $2 \times 5 - 2$ )。我们设所走的步数为 s。因为限定了只能向右和向下走，因此无论如何走，经过 8 步后 ( $s = 8$ ) 都将走到右下。而 DP 的状态也是依据所走的步数来记录的。

再来分析一下经过其他 s 步后所处的位置，根据上面的讨论，可以知道：

- 经过 8 步后，一定处于右下角(8)；
- 那么经过 5 步后( $s = 5$ )，肯定会处于编号为 5 的位置；
- 3 步后肯定处于编号为 3 的位置；
- $s = 4$  的时候，处于编号为 4 的位置，此时对于方格中，共有 5 (相当于  $n$ ) 个不同的位置，也是所有编号中最多的。

故推广来说，对于  $n \times n$  的方格，总共需要走  $2n - 2$  步，且当  $s = n - 1$  时，编号为  $n$  个，也是编号数最多的。

如果用  $DP[s, i, j]$  来记录 2 次所走的状态获得的最大值，其中  $s$  表示走  $s$  步， $i$  和  $j$  分别表示在  $s$  步后第 1 趟走的位置和第 2 趟走的位置。

**2、**为了方便描述，再对矩阵做一个编号（给这个矩阵起个名字叫  $M2$ ）：

M2
0 0 0 0 0
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4

把之前定的  $M1$  矩阵也再贴下：

M1
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

我们先看  $M1$ ，在经过 6 步后，肯定处于  $M1$  中编号为 6 的位置。而  $M1$  中共有 3 个编号为 6 的，它们分别对应  $M2$  中的 2 3 4。故对于  $M2$  来说，假设第 1 次经过 6 步走到了  $M2$  中的 2，第 2 次经过 6 步走到了  $M2$  中的 4， $DP[s, i, j]$  则对应  $DP[6, 2, 4]$ 。由于  $s = 2n - 2, 0 <= i <= j <= n$ ，所以这个  $DP$  共有  $O(n^3)$  个状态。

M1
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

再来分析一下状态转移，以  $DP[6,2,3]$  为例(就是上面 M1 中加粗的部分)，可以到达  $DP[6,2,3]$  的状态包括  $DP[5,1,2]$ ,  $DP[5,1,3]$ ,  $DP[5,2,2]$ ,  $DP[5,2,3]$ 。

**3、**下面，我们就来看看这几个状态： $DP[5,1,2]$ ,  $DP[5,1,3]$ ,  $DP[5,2,2]$ ,  $DP[5,2,3]$ , 用加粗表示位置  $DP[5,1,2]$   $DP[5,1,3]$   $DP[5,2,2]$   $DP[5,2,3]$  (加红表示要达到的状态  $DP[6,2,3]$ )

0 1 2 3 4	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4
1 2 3 4 <b>5</b>	1 2 3 4 <b>5</b>	1 2 3 4 5	1 2 3 4 5
<b>2 3 4 5 6</b>	2 3 4 5 <b>6</b>	2 3 4 <b>5 6</b>	2 3 4 <b>5 6</b>
3 4 5 <b>6</b> 7	3 4 <b>5 6</b> 7	3 4 5 <b>6</b> 7	3 4 <b>5 6</b> 7
4 5 6 7 8	4 5 6 7 8	4 5 6 7 8	4 5 6 7 8

因此：

$$DP[6,2,3] = \text{Max}(DP[5,1,2], DP[5,1,3], DP[5,2,2], DP[5,2,3]) + 6, 2 \text{ 和 } 6, 3 \text{ 格子中对应的数值} \quad (\text{式一})$$

上面 (式一) 所示的这个递推看起来没有涉及：“如果两次经过同一个格子，那么该数只加一次的这个条件”，讨论这个条件需要换一个例子，以  $DP[6,2,2]$  为例： $DP[6,2,2]$  可以由  $DP[5,1,1]$ ,  $DP[5,1,2]$ ,  $DP[5,2,2]$  到达，但由于  $i = j$ ，也就是 2 次走到同一个格子，那么数值只能加 1 次。

所以当  $i = j$  时，

$$DP[6,2,2] = \text{Max}(DP[5,1,1], DP[5,1,2], DP[5,2,2]) + 6, 2 \text{ 格子中对应的数值}$$

(式二)

**4、**故，综合上述的 (式一), (式二) 最后的递推式就是

if( $i \neq j$ )

$$DP[s, i, j] = \text{Max}(DP[s - 1, i - 1, j - 1], DP[s - 1, i - 1, j], DP[s - 1, i, j - 1], DP[s - 1, i, j]) + W[s, i] + W[s, j]$$

else

$$DP[s, i, j] = \text{Max}(DP[s - 1, i - 1, j - 1], DP[s - 1, i - 1, j], DP[s - 1, i, j]) + W[s, i]$$

其中  $W[s, i]$  表示经过  $s$  步后，处于  $i$  位置，位置  $i$  对应的方格中的数字。下一节我们将根据上述 DP 方程编码实现。

## 2.2、DP 方法实现

为了便于实现，我们认为所有不能达到的状态的得分都是负无穷，参考代码如下：

```
//copyright@caopengcs 2013
```

```

const int N = 202;
const int inf = 1000000000; //无穷大
int dp[N * 2][N][N];
bool isValid(int step, int x1, int x2, int n) { //判断状态是否合法
    int y1 = step - x1, y2 = step - x2;
    return ((x1 >= 0) && (x1 < n) && (x2 >= 0) && (x2 < n) && (y1 >= 0) && (y1 < n)
        && (y2 >= 0) && (y2 < n));
}

int getValue(int step, int x1, int x2, int n) { //处理越界 不存在的位置 给负无穷的值
    return isValid(step, x1, x2, n)?dp[step][x1][x2]:(-inf);
}

//状态表示 dp[step][i][j] 并且 i <= j, 第 step 步 两个人分别在第 i 行和第 j 行的最大得分 时
间复杂度 O(n^3) 空间复杂度 O(n^3)
int getAnswer(int a[N][N], int n) {
int P = n * 2 - 2; //最终的步数
int i, j, step;

    //不能到达的位置 设置为负无穷大
    for (i = 0; i < n; ++i) {
        for (j = i; j < n; ++j) {
            dp[0][i][j] = -inf;
        }
    }

    dp[0][0][0] = a[0][0];

    for (step = 1; step <= P; ++step) {
        for (i = 0; i < n; ++i) {
            for (j = i; j < n; ++j) {
                dp[step][i][j] = -inf;
                if (!isValid(step, i, j, n)) { //非法位置
                    continue;
                }
                //对于合法的位置进行 dp
                if (i != j) {
                    dp[step][i][j] = max(dp[step][i][j], getValue(step - 1, i - 1,
j - 1, n));
                    dp[step][i][j] = max(dp[step][i][j], getValue(step - 1, i - 1,
j, n));
                    dp[step][i][j] = max(dp[step][i][j], getValue(step - 1, i, j -
1, n));
                }
            }
        }
    }
}

```

```

        dp[step][i][j] = max(dp[step][i][j], getValue(step - 1, i, j, n))
    ;
        dp[step][i][j] += a[i][step - i] + a[j][step - j]; //不在同一个
格子，加两个数
    }
    else {
        dp[step][i][j] = max(dp[step][i][j], getValue(step - 1, i - 1,
j - 1, n));
        dp[step][i][j] = max(dp[step][i][j], getValue(step - 1, i - 1,
j, n));
        dp[step][i][j] = max(dp[step][i][j], getValue(step - 1, i, j,
n));
        dp[step][i][j] += a[i][step - i]; // 在同一个格子里，只能加一次
    }

}
}
}
return dp[P][n - 1][n - 1];
}

```

复杂度分析：状态转移最多需要统计 4 个变量的情况，看做是  $O(1)$  的，共有  $O(n^3)$  个状态，所以总的时间复杂度是  $O(n^3)$  的，且  $dp$  数组开了  $N^3$  大小，故其空间复杂度亦为  $O(n^3)$ 。

## 2.3、DP 实现优化版

如上节末所说，2.2 节实现的代码的复杂度空间复杂度是  $O(n^3)$ ，事实上，空间上可以利用滚动数组优化，由于每一步的递推只跟上 1 步的情况有关，因此可以循环利用数组，将空间复杂度降为  $O(n^2)$ 。

即我们在推算  $dp[step]$  的时候，只依靠它上一次的状态  $dp[step - 1]$ ，所以  $dp$  数组的第一维，我们只开到 2 就可以了。即  $step$  为奇数时，我们用  $dp[1][i][j]$  表示状态， $step$  为偶数我们用  $dp[0][i][j]$  表示状态，这样我们只需要  $O(n^2)$  的空间，这就是滚动数组的方法。滚动数组写起来并不复杂，只需要对上面的代码稍作修改即可，优化后的代码如下：

```

//copyright@caopengcs 8/24/2013
int dp[2][N][N];

bool isValid(int step, int x1, int x2, int n) { //判断状态是否合法
int y1 = step - x1, y2 = step - x2;
    return ((x1 >= 0) && (x1 < n) && (x2 >= 0) && (x2 < n) && (y1 >= 0) && (y1 < n)
&& (y2 >= 0) && (y2 < n));
}

```

```

}

int getValue(int step, int x1,int x2,int n) { //处理越界 不存在的位置 给负无穷的值
    return isValid(step, x1, x2, n)?dp[step % 2][x1][x2]:(-inf);
}

//状态表示 dp[step][i][j] 并且 i <= j, 第 step 步 两个人分别在第 i 行和第 j 行的最大得分 时
间复杂度 O(n^3) 使用滚动数组 空间复杂度 O(n^2)
int getAnswer(int a[N][N],int n) {
int P = n * 2 - 2; //最终的步数
int i,j,step,s;

//不能到达的位置 设置为负无穷大
for (i = 0; i < n; ++i) {
    for (j = i; j < n; ++j) {
        dp[0][i][j] = -inf;
    }
}

dp[0][0][0] = a[0][0];

for (step = 1; step <= P; ++step) {
    for (i = 0; i < n; ++i) {
        for (j = i; j < n; ++j) {
            dp[step][i][j] = -inf;
            if (!isValid(step, i, j, n)) { //非法位置
                continue;
            }
            s = step % 2; //状态下表标
            //对于合法的位置进行 dp
            if (i != j) {
                dp[s][i][j] = max(dp[s][i][j], getValue(step - 1, i - 1, j - 1,
n));
                dp[s][i][j] = max(dp[s][i][j], getValue(step - 1, i - 1, j, n));

                dp[s][i][j] = max(dp[s][i][j], getValue(step - 1, i, j - 1, n));

                dp[s][i][j] = max(dp[s][i][j], getValue(step - 1, i, j,n));
                dp[s][i][j] += a[i][step - i] + a[j][step - j]; //不在同一个格
子, 加两个数
            }
            else {
                dp[s][i][j] = max(dp[s][i][j], getValue(step - 1, i - 1, j - 1,
n));
            }
        }
    }
}

```

```

        dp[s][i][j] = max(dp[s][i][j], getValue(step - 1, i - 1, j, n))
    ;
        dp[s][i][j] = max(dp[s][i][j], getValue(step - 1, i, j, n));
        dp[s][i][j] += a[i][step - i]; // 在同一个格子里，只能加一次
    }

}
}

return dp[P % 2][n - 1][n - 1];
}

```

本第 34 章分析完毕。

## 第三十五章、完美洗牌算法

**题目详情：**有个长度为  $2n$  的数组  $\{a_1, a_2, a_3, \dots, a_n, b_1, b_2, b_3, \dots, b_n\}$ ，希望排序后  $\{a_1, b_1, a_2, b_2, \dots, a_n, b_n\}$ ，请考虑有无时间复杂度  $O(n)$ ，空间复杂度  $O(1)$  的解法。

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

### 解法一、蛮力变换

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

**a1, a2, a3, a4, b1, b2, b3, b4**

变成

**a1, b1, a2, b2, a3, b3, a4, b4**

#### 1.1、步步前移

仔细观察变换前后两个序列的特点，我们可做如下一系列操作：

第①步、确定  $b_1$  的位置，即让  $b_1$  跟它前面的  $a_2, a_3, a_4$  交换：

**a1, b1, a2, a3, a4, b2, b3, b4**

第②步、接着确定 **b2** 的位置，即让 **b2** 跟它前面的 **a3, a4** 交换：

**a1, b1, a2, b2, a3, a4, b3, b4**

第③步、**b3** 跟它前面的 **a4** 交换位置：

**a1, b1, a2, b2, a3, b3, a4, b4**

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

## 1.2、中间交换

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

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

**a1, a2, a3, b1, a4, b2, b3, b4**

第②步，让最中间的两对元素各自交换：

**a1, a2, b1, a3, b2, a4, b3, b4**

第③步，交换最中间的三对元素，序列变成：

**a1, b1, a2, b2, a3, b3, a4, b4**

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

## 解法二、完美洗牌算法

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



如果这副牌用  $a_1\ a_2\ a_3\ a_4\ b_1\ b_2\ b_3\ b_4$  表示（为简化问题，假设这副牌只有 8 张牌），然后一分为二之后，左手上的牌可能是  $a_1\ a_2\ a_3\ a_4$ ，右手上的牌是  $b_1\ b_2\ b_3\ b_4$ ，那么在如上图那样的洗牌之后，得到的牌就可能是  $b_1\ a_1\ b_2\ a_2\ b_3\ a_3\ b_4\ a_4$ 。

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

这个算法解决一个什么问题呢？跟本题有什么联系呢？

Yeah，顾名思义，完美洗牌算法解决的就是一个完美洗牌问题。什么是完美洗牌问题呢？即给定一个数组  $a_1, a_2, a_3, \dots, a_n, b_1, b_2, b_3, \dots, b_n$ ，最终把它置换成  $b_1, a_1, b_2, a_2, \dots, b_n, a_n$ 。读者可以看到，这个完美洗牌问题本质上与本题完全一致，只要在完美洗牌问题的基础上对它最后的序列 swap 两两相邻元素即可。

即：

$a_1, a_2, a_3, \dots, a_n, b_1, b_2, b_3, \dots, b_n$

通过完美洗牌问题，得到：

$b_1, a_1, b_2, a_2, b_3, a_3, \dots, b_n, a_n$

再让上面相邻的元素两两 swap，即可达到本题的要求：

$a_1, b_1, a_2, b_2, a_3, b_3, \dots, a_n, b_n$

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

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

## 2.1、位置置换 `perfect_shuffle1` 算法

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

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

从上面的例子我们能看到，前 n 个元素中，

- 第 1 个元素 a1 到了原第 2 个元素 a2 的位置，即 1->2;
- 第 2 个元素 a2 到了原第 4 个元素 a4 的位置，即 2->4;
- 第 3 个元素 a3 到了原第 6 个元素 b2 的位置，即 3->6;
- 第 4 个元素 a4 到了原第 8 个元素 b4 的位置，即 4->8;

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

上面是针对前 n 个元素，那么针对后 n 个元素，可以看出：

- 第 5 个元素 b1 到了原第 1 个元素 a1 的位置，即 5->1;
- 第 6 个元素 b2 到了原第 3 个元素 a3 的位置，即 6->3;
- 第 7 个元素 b3 到了原第 5 个元素 b1 的位置，即 7->5;
- 第 8 个元素 b4 到了原第 7 个元素 b3 的位置，即 8->7;

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

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

为何呢？因为：

1. 当  $0 < i < n$  时，原式  $= (2i) \% (2 * n + 1) = 2i$ ;
2. 当  $i \geq n$  时，原式  $(2 * i) \% (2 * n + 1)$  保持不变。

因此，如果题目允许我们再用一个数组的话，我们直接把每个元素放到该放得位置就好了。

也就产生了最简单的方法 `perfect_shuffle1`，参考代码如下：

```
// 时间 O(n)，空间 O(n) 数组下标从 1 开始
void perfect_shuffle1(int *a, int n) {
    int n2 = n * 2, i, b[N];
    for (i = 1; i <= n2; ++i) {
        b[(i * 2) % (n2 + 1)] = a[i];
```

```
    }  
  
    for (i = 1; i <= n2; ++i) {  
        a[i] = b[i];  
    }  
}
```

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

与此同时，我也提醒下读者，根据上面变换的节奏，我们可以看出有两个圈，

- 一个是 **1 -> 2 -> 4 -> 8 -> 7 -> 5 -> 1**;
  - 一个是 **3 -> 6 -> 3**。

下文 2.3.1、走圈算法 cycle\_leader 将再次提到这两个圈。

## 2.2、分而治之 perfect\_shuffle2 算法

熟悉分治法的朋友，包括若看了此文的读者肯定知道，当一个问题规模比较大时，则大而化小，分而治之。对于本题，假设  $n$  是偶数，我们试着把数组从中间拆分成两半(为了方便描述，只看数组下标就够了)：

原始数组的下标: 1...2n, 即 (1 .. n/2, n/2+1..n) (n+1 .. n+n/2, n+n/2+1 .. 2n)

前半段 ( $1 \dots n/2$ ,  $n/2+1 \dots n$ ) 和后半段 ( $n+1 \dots n+n/2$ ,  $n+n/2+1 \dots 2n$ ) 的长度皆为  $n$ 。

接下来，我们把前半段的后  $n/2$  个元素 ( $n/2+1 \dots n$ ) 和后半段的前  $n/2$  个元素 ( $n+1 \dots n+n/2$ ) 交换，得到

新的前  $n$  个元素 A:  $(1..n/2)$   $n+1..n+n/2)$

新的后  $n$  个元素 B:  $(n/2+1 \dots n \quad n+n/2+1 \dots 2n)$

换言之，当  $n$  是偶数的时候，我们把原问题拆分成了  $A$ ,  $B$  两个子问题，继而原  $n$  的求解转换成了  $n' = n/2$  的求解。

可当  $n$  是奇数的时候呢？我们可以把前半段多出来的那个元素  $a$  先拿出来放到末尾，后面所有元素前移，于此，新数列的最后两个元素满足已满足要求，只需考虑前  $2*(n-1)$  个元素即可，继而转换成了  $n-1$  的问题。

针对上述  $n$  分别为偶数和奇数的情况，下面举  $n=4$  和  $n=5$  两个例子来说明下。

① $n=4$  时，原始数组即为

a1 a2 a3 a4 b1 b2 b3 b4

按照之前  $n$  为偶数时的思路，把前半段的后 2 个元素  $a_3 a_4$  同后半段的前 2 个元素  $b_1 b_2$  交换，可得：

$a_1 a_2 b_1 b_2 a_3 a_4 b_3 b_4$

因此，我们只要用 `perfect_shuffle1` 算法继续求解 A ( $a_1 a_2 b_1 b_2$ ) 和 B ( $a_3 a_4 b_3 b_4$ ) 两个子问题就可以了。

②当  $n=5$  时，原始数组则为

$a_1 a_2 a_3 a_4 a_5 b_1 b_2 b_3 b_4 b_5$

还是按照之前  $n$  为奇数时的思路，先把  $a_5$  先单独拎出来放在最后，然后所有剩下的元素全部前移，变为：

$a_1 a_2 a_3 a_4 b_1 b_2 b_3 b_4 b_5 a_5$

此时，最后的两个元素  $b_5 a_5$  已经是我们想要的结果，只要跟之前  $n=4$  的情况一样考虑即可。

参考代码如下：

```
//copyright@caopengcs 8/23/2013
//时间 O(nlogn) 空间 O(1) 数组下标从 1 开始
void perfect_shuffle2(int *a,int n) {
    int t,i;
    if (n == 1) {
        t = a[1];
        a[1] = a[2];
        a[2] = t;
        return;
    }
    int n2 = n * 2, n3 = n / 2;
    if (n % 2 == 1) { //奇数的处理
        t = a[n];
        for (i = n + 1; i <= n2; ++i) {
            a[i - 1] = a[i];
        }
        a[n2] = t;
        --n;
    }
    //到此 n 是偶数

    for (i = n3 + 1; i <= n; ++i) {
        t = a[i];
        a[i] = a[i + n3];
        a[i + n3] = t;
    }
}
```

```

    }

    // [1.. n /2]
    perfect_shuffle2(a, n3);
    perfect_shuffle2(a + n, n3);
}

```

分析下此算法的复杂度：每次，我们交换中间的  $n$  个元素，需要  $O(n)$  的时间， $n$  是奇数的话，我们还需要  $O(n)$  的时间先把后两个元素调整好，但这不影响总体时间复杂度。

事实上，当我们采用分治算法的时候，其时间复杂度的计算公式为： $T(n) = 2*T(n/2) + O(n)$ ，这个就是跟归并排序一样的复杂度式子，由《算法导论》中文第二版 44 页的主定理，可最终解得  $T(n) = O(n\log n)$ 。至于空间，此算法在数组内部折腾的，所以是  $O(1)$ （在不考虑递归的栈的空间的前提下）。

## 2.3、完美洗牌算法 `perfect_shuffle3`

### 2.3.1、走圈算法 `cycle_leader`

因为之前无论是 `perfect_shuffle1`，还是 `perfect_shuffle2`，这两个算法的均未达到时间复杂度  $O(N)$  并且空间复杂度  $O(1)$  的要求，所以我们必须得再找一种新的方法，以期能完美的解决本节开头提出的完美洗牌问题。

让我们先来回顾一下 2.1 节位置置换 `perfect_shuffle1` 算法，还记得我之前提醒读者的关于当  $n=4$  时，通过位置置换让每一个元素到了最后的位置时，所形成的两个圈么？我引用下 2.1 节的相关内容：

当  $n=4$  的情况：

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

即通过置换，我们得到如下结论：

“于此同时，我也提醒下读者，根据上面变换的节奏，我们可以看出有两个圈，

1. 一个是 **1 -> 2 -> 4 -> 8 -> 7 -> 5 -> 1**;
2. 一个是 **3 -> 6 -> 3**。”

这两个圈可以表示为  $(1,2,4,8,7,5)$  和  $(3,6)$ ，且 `perfect_shuffle1` 算法也已经告诉了我们，不管你  $n$  是奇数还是偶数，每个位置的元素都将变为第  $(2^i) \% (2n+1)$  个元素：

$$i \rightarrow 2i \bmod (2n+1), i \in \{1, 2, \dots, 2n\}$$

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

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

第一个圈:  $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 7 \rightarrow 5 \rightarrow 1$

第二个圈:  $3 \rightarrow 6 \rightarrow 3$ :

原始数组: **1 2 3 4 5 6 7 8**

数组小标: **1 2 3 4 5 6 7 8**

走第一圈: **5 1 3 2 7 6 8 4**

走第二圈: **5 1 6 2 7 3 8 4**

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

```
//数组下标从 1 开始, from 是圈的头部, mod 是要取模的数 mod 应该为 2 * n + 1, 时间复杂度 O(圈长)
void cycle_leader(int *a,int from, int mod) {
int last = a[from],t,i;

for (i = from * 2 % mod;i != from; i = i * 2 % mod) {
    t = a[i];
    a[i] = last;
    last = t;

}
a[from] = last;
}
```

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

下面我要引用此论文“*A Simple In-Place Algorithm for In-Shuffle*”的一个结论了，即

- 对于  $2^n = (3^k - 1)$  这种长度的数组，恰好只有  $k$  个圈，且每个圈头部的起始位置分别是 **1,3,9, ... $3^{k-1}$** 。

论文原文部分为：

This means that in an in-shuffle permutation of order  $3^k - 1$ , we have exactly  $k$  cycles with  $1, 3, 3^2, \dots, 3^{k-1}$  each belonging to a different cycle. Thus for these permutations, it becomes easy to pick the 'next' cycle in order to apply the cycle leader algorithm. Note that the length of the cycle containing  $3^s$  is  $\varphi(3^k)/3^s$ , which helps us implement the cycle leader algorithm more efficiently.

也就是说，利用上述这个结论，我们可以解决这种特殊长度  $2^*n = (3^k-1)$  的数组问题，那么若给定的长度  $n$  是任意的咋办呢？此时，我们可以借鉴 2.2 节、分而治之算法的思想，把整个数组一分为二，即拆分成两个部分：

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

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

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

参照之前 2.2 节、分而治之算法的思路，且更为了能让前部分的序列满足神级结论  $2^*m = (3^k-1)$ ，我们可以把中间那两段长度为  $n-m$  和  $m$  的段交换位置，即相当于把  $m+1..n$ ,  $n+1..n+m$  的段循环右移  $m$  次（为什么要这么做？因为如此操作后，数组的前部分的长度为  $2m$ ，而根据神级结论：当  $2m=3^k-1$  时，可知这长度  $2m$  的部分恰好有  $k$  个圈）。

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

这个翻转的代码如下：

```
//翻转字符串时间复杂度 O(to - from)
void reverse(int *a, int from, int to) {
    int t;
    for (; from < to; ++from, --to) {
        t = a[from];
        a[from] = a[to];
        a[to] = t;
    }
}
```

```

    a[to] = t;
}

}

//循环右移 num 位 时间复杂度 O(n)
void right_rotate(int *a, int num, int n) {
    reverse(a, 1, n - num);
    reverse(a, n - num + 1, n);
    reverse(a, 1, n);
}

```

翻转后，得到的目标数组的下标为：

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

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

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

既然  $m=4$ ，即让上述数组中有下划线的两个部分交换，得到：

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

继而目标数组中的前半部分 a1 a2 a3 a4 b1 b2 b3 b4 部分可以用 2.3.1、走圈算法 `cycle_leader` 搞定，于此我们最终求解的  $n$  长度变成了  $n'=3$ ，即  $n$  的长度减小了 4，单独再解决后半部分 a5 a6 a7 b5 b6 b7 即可。

### 2.3.3、完美洗牌算法 perfect\_shuffle3

从上文的分析过程中也就得出了我们的完美洗牌算法，其算法流程为：

- 输入数组 A[1..2 \* n]
1. step 1 找到  $2 * m = 3^k - 1$  使得  $3^k \leq 2 * n < 3^{k+1}$
  2. step 2 把  $a[m + 1..n + m]$  那部分循环移  $m$  位
  3. step 3 对每个  $i = 0, 1, 2..k - 1$ ， $3^i$  是个圈的头部，做 `cycle_leader` 算法，数组长度为  $m$ ，所以对  $2 * m + 1$  取模。

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

上述算法流程对应的论文原文为：

Thus we have the following ***In-shuffle Algorithm:***

**Input:** An array  $A[1, \dots, 2n]$

**Step 1.** Find a  $2m = 3^k - 1$  such that  $3^k \leq 2n < 3^{k+1}$

**Step 2.** Do a right cyclic shift of  $A[m + 1, \dots, n + m]$  by a distance  $m$

**Step 3.** For each  $i \in \{0, 1, \dots, k - 1\}$ , starting at  $3^i$ , do the cycle leader algorithm for the in-shuffle permutation of order  $2m$

**Step 4.** Recursively do the in-shuffle algorithm on  $A[2m + 1, \dots, 2n]$ .

以上各个步骤对应的时间复杂度分析如下：

1. 因为循环不断乘 3 的，所以时间复杂度  $O(\log n)$
2. 循环移位  $O(n)$
3. 每个圈，每个元素只走了一次，一共  $2^*m$  个元素，所以复杂度  $\Omega(m)$ ，而  $m < n$ ，所以 也在  $O(n)$  内。
4.  $T(n - m)$

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

此完美洗牌算法实现的参考代码如下：

```
//copyright@caopengcs 8/24/2013
//时间 O(n)，空间 O(1)
void perfect_shuffle3(int *a, int n) {
    int n2, m, i, k, t;
    for (; n > 1;) {
        // step 1
        n2 = n * 2;
        for (k = 0, m = 1; n2 / m >= 3; ++k, m *= 3)
        ;
        m /= 2;
        // 2m = 3^k - 1, 3^k <= 2n < 3^(k + 1)

        // step 2
```

```

    right_rotate(a + m, m, n);

    // step 3

    for (i = 0, t = 1; i < k; ++i, t *= 3) {
        cycle_leader(a, t, m * 2 + 1);

    }

    //step 4
    a += m * 2;
    n -= m;

}

// n = 1
t = a[1];
a[1] = a[2];
a[2] = t;
}

```

### 2.3.4、perfect\_shuffle3 算法解决其变形问题

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

```

//copyright@caopengcs 8/24/2013
//时间复杂度 O(n)，空间复杂度 O(1)，数组下标从 1 开始，调用 perfect_shuffle3
void shuffle(int *a,int n) {
    int i,t,n2 = n * 2;
    perfect_shuffle3(a,n);
    for (i = 2; i <= n2; i += 2) {
        t = a[i - 1];
        a[i - 1] = a[i];
        a[i] = t;

    }
}

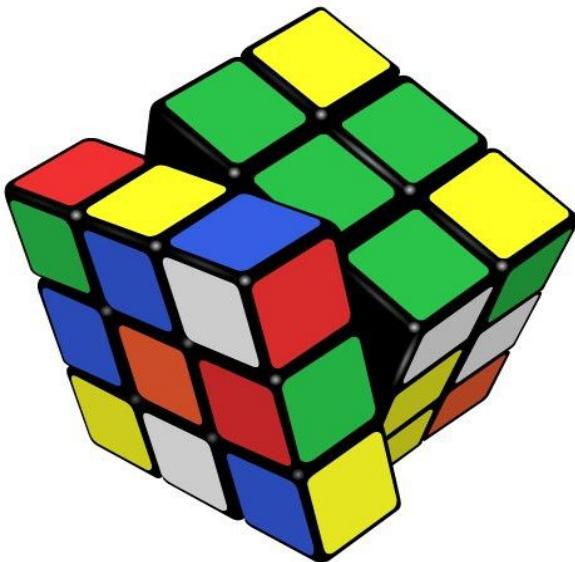
```

上述的这个“在完美洗牌问题的基础上对它最后的序列 swap 两两相邻元素”的操作（当然，你也可以让原数组第一个和最后一个不变，中间的  $2 * (n - 1)$  项用原始的标准完美洗牌算法做），只是在完美洗牌问题时间复杂度  $O(N)$  空间复杂度  $O(1)$  的基础上再增加  $O(N)$  的时间复杂度，故总

的时间复杂度  $O(N)$  不变，且理所当然的保持了空间复杂度  $O(1)$ 。至此，咱们的问题得到了圆满解决！

### 2.3.5、神级结论是如何来的？

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



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

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

- 概念 1  $\mod$  表示对一个数取余数，比如  $3 \mod 5 = 3, 5 \mod 3 = 2$ ；
- 定义 1 欧拉函数  $\phi(m)$  表示为不超过  $m$  (即小于等于  $m$ ) 的数中，与  $m$  互素的正整数个数
- 定义 2 若  $\phi(m) = \text{Ord}_m(a)$  则称  $a$  为  $m$  的原根，其中  $\text{Ord}_m(a)$  定义为： $a^d \equiv 1 \pmod{m}$ ，其中  $d=0, 1, 2, 3, \dots$ ，但取让等式成立的最小的那个  $d$ 。

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

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

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

- 定义 3 对模指数， $a$  对模  $m$  的原根定义为  $Ord_m(a) = d$ , s.t.  $a^d \equiv 1 \pmod{m}$   
中最小的正整数  $d$

再比如，2 是 9 的原根，因为  $2^d \equiv 1 \pmod{9}$ ，为了让  $2^d$  除以 9 的余数恒等于 1，可知最小的正整数  $d=6$ ，而  $\phi(9)=6$ ，满足原根的定义。

- 定理 1 同余定理：两个整数  $a, b$ ，若它们除以正整数  $m$  所得的余数相等，则称  $a \equiv b \pmod{m}$  同余，记作  $a \equiv b \pmod{m}$ ，读做  $a$  与  $b$  关于模  $m$  同余。

- 定理 2 当  $p$  为奇素数且  $a$  是  $p^2$  的原根时  $\Rightarrow a$  也是  $p^k$  的原根

- 定理 3 费马小定理：如果  $a$  和  $m$  互质，那么  $a^{\phi(m)} \bmod m = 1$

- 定理 4 若  $(a, m) = 1$  且  $a$  为  $m$  的原根，那么  $a$  是  $(\mathbb{Z}/m\mathbb{Z})^*$  的生成元。

取  $a = 2, m = 3$ 。

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

所以：

$$S(1) = \{1, 2\}$$

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

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

那与  $3^k$  不互质的数怎么办？如果  $0 < i < 3^k$  与  $3^k$  不互质，那么  $i$  与  $3^k$  的最大公约数一定是  $3^t$  的形式（只包含约数 3），并且  $t < k$ 。即  $\gcd(i, 3^k) =$

**3^t**, 等式两边除以个  $3^t$ , 即得  $\gcd(i/(3^t), 3^{(k-t)}) = 1$ ,  $i/(3^t)$  都与  $3^{(k-t)}$  互质了, 并且  $i/(3^t) < 3^{(k-t)}$ , 根据  $S(k)$  的定义, 可见 **i/(3^t)** 在集合 **S(k-t)** 中。

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

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

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

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

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

所以  $S(1) = \{1, 2\}$

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

比如  $k = 3$ 。 我们有:

1.  $S(3) = \{1, 2, 4, 8, 16, 5, 10, 20, 13, 26, 25, 23, 19, 11, 22, 17, 7, 14\}$  包含了小于 **27** 且与 **27** 互质的所有数, 圈的首部为 **1**, 这是原根定义决定的。
2. 那么与 **27** 最大公约数为 **3** 的数, 我们用  $S(2)$  中的数乘以 **3** 得到。  $S(2) * 3 = \{3, 6, 12, 24, 21, 15\}$ , 圈中元素的顺序没变化, 圈的首部是 **3**。
3. 与 **27** 最大公约数为 **9** 的数, 我们用  $S(1)$  中的数乘以 **9** 得到。  $S(1) * 9 = \{9, 18\}$ , 圈中得元素的顺序没变化, 圈的首部是 **9**。

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

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

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

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

- 取头为 1,  $(\mathbb{Z}/27)^* = \{1, 2, 4, 8, 16, 5, 10, 20, 13, 26, 25, 23, 19, 11, 22, 17, 7, 14\}$ , 也就是说, 与  $27$  互素且小于  $27$  的正整数集合为  $\{1, 2, 4, 8, 16, 5, 10, 20, 13, 26, 25, 23, 19, 11, 22, 17, 7, 14\}$ , 因此  $\phi(m) = \phi(27) = 18$ , 从而满足  $a^d \equiv 1 \pmod{m}$  的最小  $d = 18$ , 故得出  $2$  为  $27$  的原根;
- 取头为 3, 就可以得到  $\{3, 6, 12, 24, 21, 15\}$ , 这就是以 3 为头的环, 这个圈的特点是所有的数都是 3 的倍数, 且都不是 9 的倍数。为什么呢? 因为  $2^k$  和  $27$  互素。

具体点则是: 如果  $3 \times 2^k$  除  $27$  的余数能够被 9 整除, 则有一个  $n$  使得  $3 \times 2^k - 9n \equiv 0 \pmod{27}$ , 即  $3 \times 2^k - 9n$  能够被  $27$  整除, 从而  $3 \times 2^k - 9n = 27m$ , 其中  $n, m$  为整数, 这样一来, 式子约掉一个 3, 我们便能得到  $2^k = 9m + 3n$ , 也就是说,  $2^k$  是 3 的倍数, 这与  $2^k$  与  $27$  互素是矛盾的, 所以,  $3 \times 2^k$  除  $27$  的余数不可能被 9 整除。

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

式子两边同时乘以 3 得到:  $3 \times 2^k - 3 = 81m$  是  $27$  的倍数, 从而  $3 \times 2^k$  除以  $27$  的余数为 3;

同理, 当余数为 2 时,  $2^k - 2 = 27m$ ,  $\Rightarrow 3 \times 2^k - 6 = 81m$ , 从而  $3 \times 2^k$  除以  $27$  的余数为 6;

当余数为 4 时,  $2^k - 4 = 37m$ ,  $\Rightarrow 3 \times 2^k - 12 = 81m$ , 从而  $3 \times 2^k$  除以  $27$  的余数为 12;

同理, 可以取到 15, 21, 24。从而也就印证了上面的结论: 取头为 3, 就可以得到  $\{3, 6, 12, 24, 21, 15\}$ 。

- 取 9 为头, 这就很简单了, 这个圈就是  $\{9, 18\}$

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

**最后**, 咱们也再验证下上述过程:

因为  $i \in \{1, 2, \dots, 2n\}$ , 故:

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

26 27

由于  $n=13$ ,  $2n+1=27$ , 据此  $i \rightarrow 2i \pmod{2n+1}$  公式可知, 上面第 i 位置的数将分别变成下述位置的:

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

25 0

根据 i 和  $i'$  前后位置的变动, 我们将得到 3 个圈:

- $1 \rightarrow 248165102013262523191122177141$ ;
- $3 \rightarrow 6 \rightarrow 12 \rightarrow 24 \rightarrow 21 \rightarrow 15 \rightarrow 3$
- $9 \rightarrow 18 \rightarrow 9$

没错, 这 3 个圈的数字与咱们之前得到的 3 个圈一致吻合, 验证完毕。

### 2.3.6、完美洗牌问题的几个扩展

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

1. 既然完美洗牌问题是给定输入:  $a_1, a_2, a_3, \dots, a_N, b_1, b_2, b_3, \dots, b_N$ , 要求输出:  $b_1, a_1, b_2, a_2, \dots, b_N, a_N$ ; 那么有无考虑过它的逆问题: 即给定  $b_1, a_1, b_2, a_2, \dots, b_N, a_N$ , 要求输出  $a_1, a_2, a_3, \dots, a_N, b_1, b_2, b_3, \dots, b_N$  ?
2. 完美洗牌问题是两手洗牌, 假设有三只手同时洗牌呢? 那么问题将变成: 输入是  $a_1, a_2, \dots, a_N, b_1, b_2, \dots, b_N, c_1, c_2, \dots, c_N$ , 要求输出是  $c_1, b_1, a_1, c_2, b_2, a_2, \dots, c_N, b_N, a_N$ , 这个时候, 怎么处理?

以上两个完美洗牌问题的几个扩展请读者思考, 具体解答请参看参考链接第 15 条。

本第 35 章完。

## 参考链接

1. huangxy10, <http://blog.csdn.net/huangxy10/article/details/8071242>;
2. @绿色夹克衫, <http://www.51nod.com/answer/index.html#!answerId=598>;
3. 格子取数的蛮力穷举法:  
<http://wenku.baidu.com/view/681c853b580216fc700af9a.html>;
4. @陈立人,  
[http://mp.weixin.qq.com/mp/appmsg/show?\\_\\_biz=MjM5ODIzNDQ3Mw==&appmsgid=10000141&itemidx=1&sign=4f1aa1a2269a1fac88be49c8cba21042](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5ODIzNDQ3Mw==&appmsgid=10000141&itemidx=1&sign=4f1aa1a2269a1fac88be49c8cba21042);
5. caopengcs, <http://blog.csdn.net/caopengcs/article/details/10196035>;
6. 完美洗牌算法的原始论文“*A Simple In-Place Algorithm for In-Shuffle*”,  
<http://att.newsmth.net/att.php?p.1032.47005.1743.pdf>;
7. 原始根模: [http://en.wikipedia.org/wiki/Primitive\\_root\\_modulo\\_n](http://en.wikipedia.org/wiki/Primitive_root_modulo_n);
8. 洗牌的学问: <http://www.thecodeway.com/blog/?p=680>;
9. 关于完美洗牌算法:  
<http://cs.stackexchange.com/questions/332/in-place-algorithm-for-interleaving-an-array/400#400>;
10. 关于完美洗牌算法中圈的说明:  
<http://www.emis.de/journals/DMTCS/pdfpapers/dm050111.pdf>;
11. 关于神级结论的讨论:  
<http://math.stackexchange.com/questions/477125/how-to-prove-algebraic-structure-of-the-perfect-shuffle> (左边链接中的讨论中有错误, 以在本文 2.3.5 节进行了相关修正) ;
12. caopengcs 关于神级结论的证明:  
<http://blog.csdn.net/caopengcs/article/details/10429013>;
13. 同余的概念: <http://zh.wikipedia.org/wiki/%E5%90%8C%E9%A4%98>;
14. 神奇的费马小定理:  
[http://www.xieguofang.cn/Maths/Number\\_Theory/Fermat's\\_Little\\_Theorem\\_1.htm](http://www.xieguofang.cn/Maths/Number_Theory/Fermat's_Little_Theorem_1.htm);
15. 完美洗牌问题的几个扩展: <http://blog.csdn.net/caopengcs/article/details/10521603>;
16. 原根与指数的介绍: <http://wenku.baidu.com/view/bbb88ffc910ef12d2af9e738>;
17. 《数论概论》Joseph H. Silverman 著, 推荐理由: 因写上文中的完美洗牌算法遇到了一堆数论定理受了刺激, 故推荐此书;

## 后记

以上第 35 章可能是整个系列迄今为止我最满意的一篇，不仅仅是因为此章思路清晰，过渡自然，代码风格良好，更因为有了@曹鹏博士 的加入，编程艺术如虎添翼，质量更上一层！

：编程艺术通过解决一个个实际的编程面试题，让广大初学者一步步学会分析问题解决问题优化问题的能力，且每个问题的讲解足够通俗，希望后续我(们)做得越来越好！July、二零一三年八月二十四日凌晨零点三十七分。

## 第三十六~三十七章、搜索智能提示 suggestion，附近地点搜索

作者：July。致谢：caopengcs、胡果果。

时间：二零一三年九月七日。

### 题记

写博的近三年，整理了太多太多的笔试面试题，如微软面试 100 题系列，和眼下这个程序员编程艺术系列，真心觉得题目年年变，但解决问题的方法永远都是那几种，用心准备后，自会发现一切有迹可循。

故为更好的帮助人们找到工作，特准备在北京举办一系列**面试&算法讲座**。时间定为周末，每次一个上午或下午，受众对象为要找工作或换工作或对算法感兴趣的朋友，费用前期暂愿交就交，交多少全由自己决定。主讲人：我和目前 zoj 排名第一的 caopengcs 博士。9 月 15 日为第 1 次讲座：[http://blog.csdn.net/v\\_july\\_v/article/details/7237351#t22](http://blog.csdn.net/v_july_v/article/details/7237351#t22)。

OK，切入正题。上面说整理过很多笔试面试题，但好的笔试面试题真心难求，包括在编程艺术系列每一章的选题，越到后面越难挑，而本文写两个跟实际挂钩的问题，它们来自此文 [http://blog.csdn.net/v\\_july\\_v/article/details/7974418](http://blog.csdn.net/v_july_v/article/details/7974418) 的第 3.6 题，和第 87 题，即

- 第三十六章、搜索引擎中中的关键词智能提示 suggestion；
- 第三十七章、附近地点的搜索；

本文的两个选题都是比较开放的，没有固定标准的答案。读者若有何意见，或是发现了任何问题，欢迎随时于本文评论下留言或指正，感谢。

## 第三十六章、搜索关键词智能提示 suggestion

**题目详情：**百度搜索框中，输入“北京”，搜索框下面会以北京为前缀，展示“北京爱情故事”、“北京公交”、“北京医院”等等搜索词，输入“结构之”，会提示“结构之法”，“结构之法 算法之道”等搜索词。

请问，如何设计此系统，使得空间和时间复杂度尽量低。



**题目分析：**本题来源于去年 2012 年百度的一套实习生笔试题中的系统设计题（为尊重原题，本章主要使用百度搜索引擎展开论述，而不是 google 等其它搜索引擎，但原理不会差太多。然脱离本题，平时搜的时候，鼓励用...），题目比较开放，考察的目的在于看应聘者解决问题的思路是否清晰明确，其次便是看能考虑到多少细节。

我去年整理此题的时候，曾简单解析过，提出的方法是：

- 直接上 **Trie 树** 「Trie 树的介绍见：从 Trie 树（字典树）谈到后缀树」 + **TOP K** 「hashmap+堆，hashmap+堆 统计出如 10 个近似的热词，也就是说，只存与关键词近似的比如 10 个热词」

方法就是这样子的：**Trie 树+TOP K 算法**，但在实际中，真的只要 **Trie 树 + TOP K 算法就够了么**，有什么需要考虑的细节？OK，请看下文娓娓道来。

# 解法一、Trie 树 + TOP K

## 步骤一、trie 树存储前缀后缀

若看过博客内这篇介绍 Trie 树和后缀树的文章

[http://blog.csdn.net/v\\_july\\_v/article/details/6897097](http://blog.csdn.net/v_july_v/article/details/6897097) 的话，应该就能对 trie 树有个大致的了解，为示本文完整性，引用下原文内容，如下：

### 1.1、什么是 Trie 树

Trie 树，即字典树，又称单词查找树或键树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：最大限度地减少无谓的字符串比较，查询效率比哈希表高。

Trie 的核心思想是空间换时间。利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。它有 3 个基本性质：

1. 根节点不包含字符，除根节点外每一个节点都只包含一个字符。
2. 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。
3. 每个节点的所有子节点包含的字符都不相同。

### 1.2、树的构建

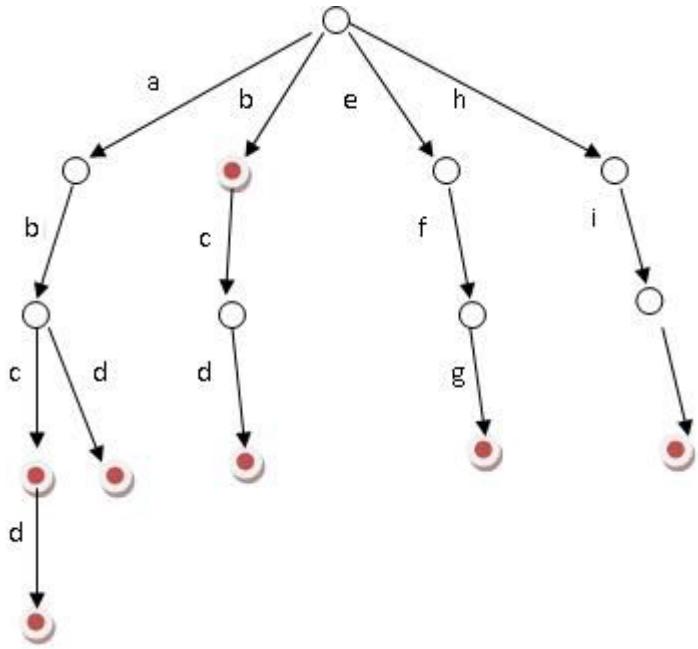
举个在网上流传颇广的例子，如下：

题目：给你 100000 个长度不超过 10 的单词。对于每一个单词，我们要判断他出没出现过，如果出现了，求第一次出现在第几个位置。

分析：这题当然可以用 hash 来解决，但是本文重点介绍的是 trie 树，因为在某些方面它的用途更大。比如说对于某一个单词，我们要询问它的前缀是否出现过。这样 hash 就不好搞了，而用 trie 还是很简单。

现在回到例子中，如果我们用最傻的方法，对于每一个单词，我们都要去查找它前面的单词中是否有它。那么这个算法的复杂度就是  $O(n^2)$ 。显然对于 100000 的范围难以接受。现在我们换个思路想。假设我要查询的单词是 abcd，那么在他前面的单词中，以 b, c, d, f 之类开头的我显然不必考虑。而只要找以 a 开头的中是否存在 abcd 就可以了。同样的，在以 a 开头中的单词中，我们只要考虑以 b 作为第二个字母的，一次次缩小范围和提高针对性，这样一个树的模型就渐渐清晰了。

好比假设有 b, abc, abd, bcd, abcd, efg, hii 这 6 个单词，我们构建的树就是如下图这样的：



当时第一次看到这幅图的时候，便立马感到此树之不凡构造了。单单从上幅图便可窥知一二，好比大海搜人，立马就能确定东南西北中的到底哪个方位，如此迅速缩小查找的范围和提高查找的针对性，不失为一创举。

ok，如上图所示，对于每一个节点，从根遍历到他的过程就是一个单词，如果这个节点被标记为红色，就表示这个单词存在，否则不存在。

那么，对于一个单词，我只要顺着它从根走到对应的节点，再看这个节点是否被标记为红色就可以知道它是否出现过了。把这个节点标记为红色，就相当于插入了这个单词。“”

借用上面的图，当用户输入前缀 **a** 的时候，搜索框可能会展示以 **a** 为前缀的“**abcd**”，“**abd**”等关键词，再当用户输入前缀 **b** 的时候，搜索框下面可能会提示以 **b** 为前缀的“**bcd**”等关键词，如此，实现搜索引擎智能提示 **suggestion** 的第一个步骤便清晰了，即用 **trie** 树存储大量字符串，当前缀固定时，存储相对来说比较热的后缀。那又如何统计热词呢？请看下文步骤二、**TOP K** 算法统计热词。

## 步骤二、**TOP K** 算法统计热词

当每个搜索引擎输入一个前缀时，下面它只会展示 0~10 个候选词，但若是碰到那种候选词很多的时候，如何取舍，哪些展示在前面，哪些展示在后面？这就是一个搜索热度的问题。

如本题描述所说，在去年的这个时候，当我在搜索框内搜索“北京”时，它下面会提示以“北京”为前缀的诸如“北京爱情故事”，“北京公交”，“北京医院”，且“北京爱情故事”展示在第一个：

**【东南早报】#北京爱情故事#：**不仅是一个爱情故事<http://url.cn/4eFMN3>，与超高收视相伴的，是《北京爱情故事》引发了一场“网络风暴”。在国内最火搜索引擎百度搜索中引发热议，搜索输入关键词“北京”，“北京爱情故事”已成为提示首选，这一现象足以说明《北京爱情故事》在网络的影响力。

为何输入“北京”，会首先提示“北京爱情故事”呢？因为去年的这个时候，正是《北京爱情故事》这部电视剧上映正火的时候（其上映日期为2012年1月8日，火了至少一年），那个时候大家都一个劲的搜索这部电视剧的相关信息，当10个人中输入“北京”后，其中有8个人会继续敲入“爱情故事”（连起来就是“北京爱情故事”）的时候，搜索引擎对此当然不会无动于衷。

也就是说，搜索引擎知道了这个时间段，大家都在疯狂查找北京爱情故事，故当用户输入以“北京”为前缀的时候，搜索引擎猜测用户有80%的机率是要查找“北京爱情故事”，故把“北京爱情故事”在下面提示出来，并放在第一个位置上。

但为何今年这个时候再次搜索“北京”的时候，它展示出来的词不同了呢？



原因在于随着时间变化，人们对《北京爱情故事》这部电视剧的关注度逐渐下降，与此同时，又出现了新的热词，或新的电影，故现在虽然同样是输入“北京”，后面提示的词也相应跟着起了变化。那解决这个问题的办法是什么呢？如开头所说：定期分析某段时间内的人们搜索的关键词，统计出搜索次数比较多的热词，继而当用户输入某个前缀时，优先展示热词。

故说白了，这个问题的第二个步骤便是统计热词，我们把统计热词的方法称为 TOP K 算法，此算法的应用场景便是此文 [http://blog.csdn.net/v\\_july\\_v/article/details/7382693](http://blog.csdn.net/v_july_v/article/details/7382693) 中的第 2 个问题，再次原文引用：

#### “寻找热门查询，300 万个查询字符串中统计最热门的 10 个查询”

原题：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。

解答：由上面第 1 题，我们知道，数据大则划为小的，如一亿个 IP 求 Top 10，可先 %1000 将 IP 分到 1000 个小文件中去，并保证一种 IP 只出现在一个文件中，再对每个小文件中的 IP 进行 hashmap 计数统计并按数量排序，最后归并或者最小堆依次处理每个小文件的 top10 以得到最后的结果。

但如果数据规模本身就比较小，能一次性装入内存呢？比如这第 2 题，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（300 万个字符串假设没有重复，都是最大长度，那么最多占用内存  $3M * 1K / 4 = 0.75G$ 。所以可以将所有字符串都存放在内存中进行处理），而现在只是需要一个合适的数据结构，在这里，HashTable 绝对是我们优先的选择。

所以我们放弃分而治之/hash 映射的步骤，直接上 hash 统计，然后排序。So，针对此类典型的 TOP K 问题，采取的对策往往是：hashmap + 堆。如下所示：

1. **hashmap 统计：**先对这批海量数据预处理。具体方法是：维护一个 Key 为 Query 字串，Value 为该 Query 出现次数的 HashTable，即 `hash_map(Query, Value)`，每次读取一个 Query，如果该字串不在 Table 中，那么加入该字串，并且将 Value 值设为 1；如果该字串在 Table 中，那么将该字串的计数加一即可。最终我们在  $O(N)$  的时间复杂度内用 Hash 表完成了统计；
2. **堆排序：**第二步、借助堆这个数据结构，找出 Top K，时间复杂度为  $N \log K$ 。即借助堆结构，我们可以在  $\log$  量级的时间内查找和调整/移动。因此，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。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)。

别忘了这篇文章中所述的堆排序思路：‘维护 k 个元素的最小堆，即用容量为 k 的最小堆存储最先遍历到的 k 个数，并假设它们即是最大的 k 个数，建堆费时  $O(k)$ ，并调整堆(费时  $O(\log k)$ )后，有  $k_1 > k_2 > \dots > k_{\min}$  ( $k_{\min}$  设为小顶堆中最小元素)。继续遍历数列，每次遍历一个元素 x，与堆顶元素比较，若  $x > k_{\min}$ ，则更新堆 (x 入堆，用时  $\log k$ )，否则不更新堆。这样下来，总费时  $O(k * \log k + (n - k) * \log k) = O(n * \log k)$ 。

此方法得益于在堆中，查找等各项操作时间复杂度均为  $\log k$ 。--[第三章续、Top K 算法问题的实现](#)。

当然，你也可以采用 trie 树，关键字域存该查询串出现的次数，没有出现为 0。最后用 10 个元素的最小堆来对出现频率进行排序。

相信，如此，也就不难理解开头所提出的方法了：Trie 树+ TOP K 「hashmap+堆，hashmap+堆 统计出如 10 个近似的热词，也就是说，只存与关键词近似的比如 10 个热词」。

而且你以后就可以告诉你身边的伙伴们，为何输入“结构之”，会提示出来一堆以“结构之”为前缀的词了：

结构之|

结构之美

结构之法 算法之道

结构之法 算法

结构之美 算法之道

结构之法 算法知道

结构之法 算法之道 第一期博文chm文件集锦

结构之道 算法之道

结构之法 红黑树

结构之法 july

结构之法 博客

Google 搜索

手气不错

方法貌似成型了，但有哪些需要注意的细节呢？如[@江申\\_Johnson](#) 所说：“实际工作里，比如当前缀很短的时候，候选词很多的时候，查询和排序性能可能有问题，也许可以加一层索引 trie（这层索引可以只索引频率高于某一个阈值的词，很短的时候查这个就可以了。数量不够的话再去查索引了全部词的 trie 树）；而且有时候不能根据 query 频率来排，而要引导用户输入信息量更全面的 query，或者或不仅仅是前缀匹配这么简单。”

## 扩展阅读

除了上文提到的 trie 树，三叉树或许也是一个不错的解决方案：

<http://igoro.com/archive/efficient-auto-complete-with-a-ternary-search-tree/>。此外，

StackOverflow 上也有两个讨论帖子，大家可以看看：①

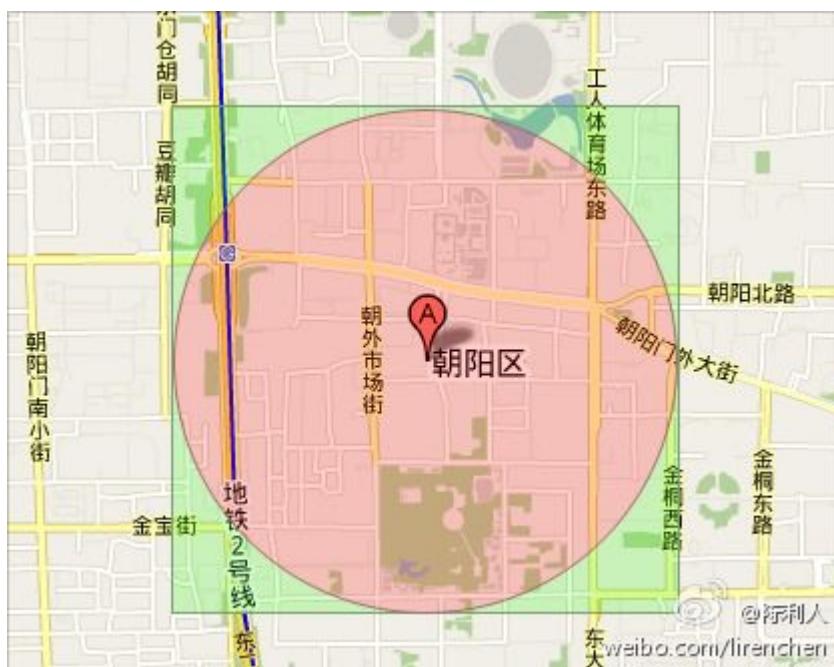
<http://stackoverflow.com/questions/2901831/algorithm-for-autocomplete>, ②

<http://stackoverflow.com/questions/1783652/what-is-the-best-autocomplete-suggest-algorithm-datastructure-c-c>.

## 第三十七章、附近地点搜索

**题目详情：**找一个点集中与给定点距离最近的点，同时，给定的二维点集都是固定的，查询可能有很多次，时间复杂度  $O(n)$  无法接受，请设计数据结构和相应的算法。

**题目分析：**此题是去年微软的三面题，类似于一朋友@陈利人 出的这题：附近地点搜索，就是搜索用户附近有哪些地点。随着 GPS 和带有 GPS 功能的移动设备的普及，附近地点搜索也变得炙手可热。在庞大的地理数据库中搜索地点，索引是很重要的。但是，我们的需求是搜索附近地点，例如，坐标(39.91, 116.37)附近 500 米内有什么餐馆，那么让你来设计，该怎么做？



## 解法一、R 树二维搜索

假定只允许你初中数学知识，那么你可能建一个 X-Y 坐标系，即以坐标(39.91, 116.37)为圆心，以 500 的长度为半径，画一个园，然后一个一个坐标点的去查找。此法看似可行，但复杂度可想而知，即便你自以为聪明的说把整个平面划分为四个象限，一个一个象限的查找，此举虽然优化程度不够，但也说明你一步步想到点子上去了。

即不一个一个坐标点的查找，而是一个一个区域的查找，相对来说，其平均查找速度和效率会显著提升。如此，便自然而然的想到了有没有一种一次查找定位于一个区域的数据结构呢？

若看过博客内之前介绍 R 树的这篇文章

[http://blog.csdn.net/v\\_JULY\\_v/article/details/6530142#t2](http://blog.csdn.net/v_JULY_v/article/details/6530142#t2) 的读者立马便能意识到，R 树就是解决这个区域查找继而不断缩小规模的问题。特直接引用原文：

## R 树的数据结构

R 树是 B 树在高维空间的扩展，是一棵平衡树。每个 R 树的叶子结点包含了多个指向不同数据的指针，这些数据可以是存放在硬盘中的，也可以是存在内存中。根据 R 树的这种数据结构，当我们需要进行一个高维空间查询时，我们只需要遍历少数几个叶子结点所包含的指针，查看这些指针指向的数据是否满足要求即可。这种方式使我们不必遍历所有数据即可获得答案，效率显著提高。下图 1 是 R 树的一个简单实例：

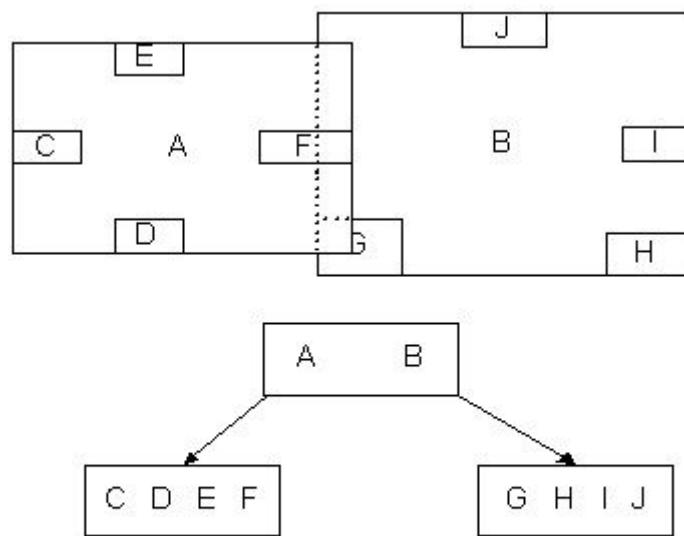
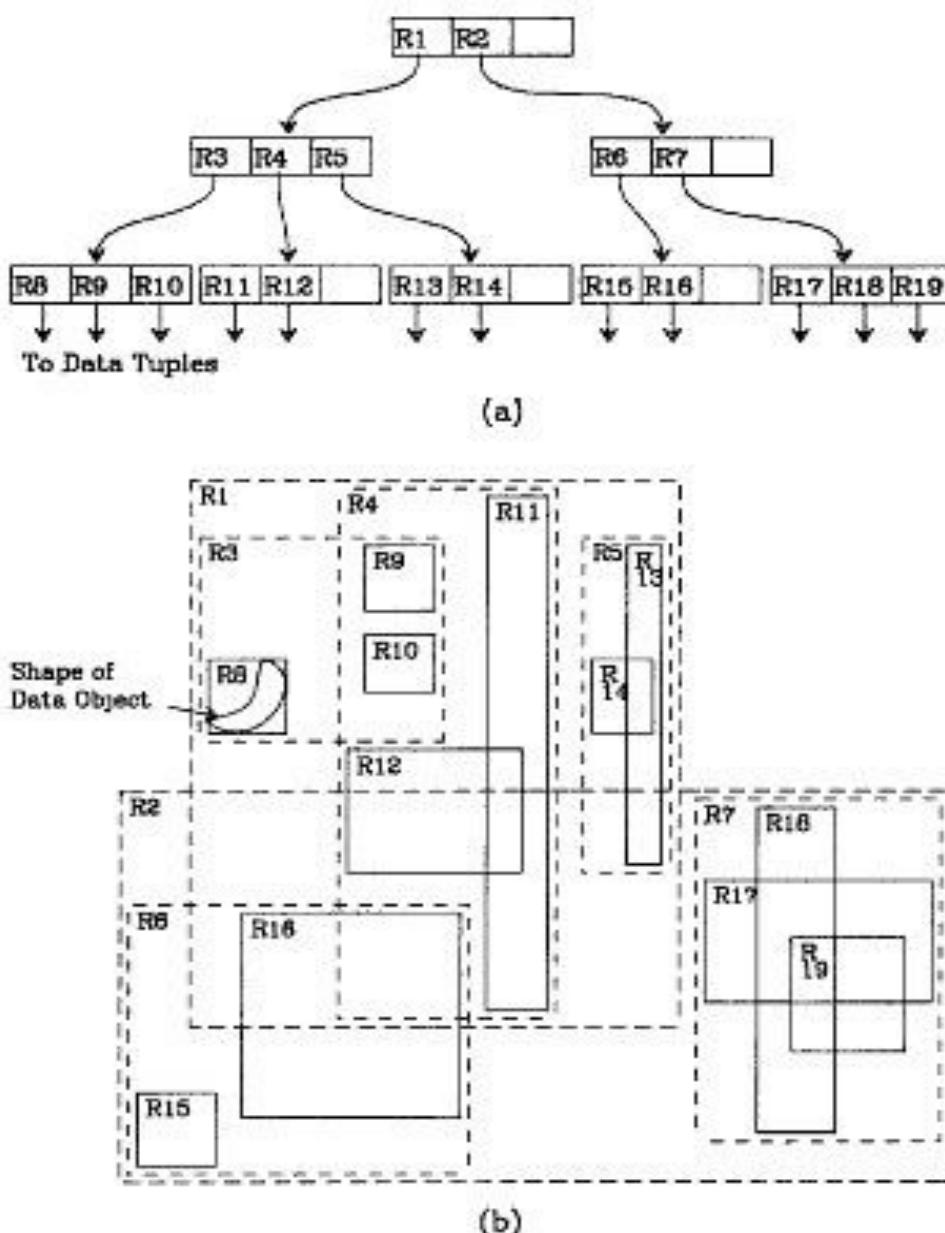


图 1：一个 R 树 实例

我们在上面说过，R 树运用了空间分割的理念，这种理念是如何实现的呢？R 树采用了一种称为 MBR(Minimal Bounding Rectangle)的方法，在此我把它译作“最小边界矩形”。从叶子结点开始用矩形(rectangle)将空间框起来，结点越往上，框住的空间就越大，以此对空间进行分割。有点不懂？没关系，继续往下看。在这里我还想提一下，R 树中的 R 应该代表的是 Rectangle（此处参考 wikipedia 上关于 [R 树](#) 的介绍），而不是大多数国内教材中所说的 Region（很多书把 R 树称为区域树，这是有误的）。我们就拿二维空间来举例。下图是 Guttman 论文中的一幅图：



我来详细解释一下这张图。

1. 先来看图 (b)，首先我们假设所有数据都是二维空间下的点，图中仅仅标志了 R8 区域中的数据，也就是那个 **shape of data object**。别把那一块不规则图形看成一个数据，我们把它看作是多个数据围成的一个区域。为了实现 R 树结构，我们用一个最小边界矩形恰好框住这个不规则区域，这样，我们就构造出了一个区域：R8。R8 的特点很明显，就是正正好框住所有在此区域中的数据。
2. 其他实线包围住的区域，如 R9, R10, R12 等都是同样的道理。这样一来，我们一共得到了 12 个最最基本的最小矩形。这些矩形都将被存储在子结点中。
3. 下一步操作就是进行高一层次的处理。我们发现 R8, R9, R10 三个矩形距离最为靠近，因此就可以用一个更大的矩形 R3 恰好框住这 3 个矩形。
4. 同样道理，R15, R16 被 R6 恰好框住，R11, R12 被 R4 恰好框住，等等。所有最基本的最小边界矩形被框入更大的矩形中之后，再次迭代，用更大的框去框住这些矩形。

我想大家都应该理解这个数据结构的特征了。用地图的例子来解释，就是所有的数据都是餐厅所对应的地点，先把相邻的餐厅划分到同一块区域，划分好所有餐厅之后，再把邻近的区域划分到更大的区域，划分完毕后再次进行更高层次的划分，直到划分到只剩下两个最大的区域为止。要查找的时候就方便了。

下面就可以把这些大大小小的矩形存入我们的 R 树中去了。根结点存放的是两个最大的矩形，这两个最大的矩形框住了所有的剩余的矩形，当然也就框住了所有的数据。下一层的结点存放了次大的矩形，这些矩形缩小了范围。每个叶子结点都是存放的最小的矩形，这些矩形中可能包含有 n 个数据。

## 地图查找的实例

讲完了基本的数据结构，我们来讲个实例，如何查询特定的数据。又以餐厅为例，假设我要查询广州市天河区天河城附近一公里的所有餐厅地址怎么办？

1. 打开地图（也就是整个 R 树），先选择国内还是国外（也就是根结点）；
2. 然后选择华南地区（对应第一层结点），选择广州市（对应第二层结点）；
3. 再选择天河区（对应第三层结点）；
4. 最后选择天河城所在的那个区域（对应叶子结点，存放有最小矩形）；

遍历所有在此区域内的结点，看是否满足我们的要求即可。怎么样，其实 R 树的查找规则跟查地图很像吧？对应下图：



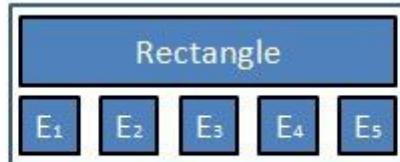
一棵 R 树满足如下的性质：

1. 除非它是根结点之外，所有叶子结点包含有  $m$  至  $M$  个记录索引（条目）。作为根结点的叶子结点所具有的记录个数可以少于  $m$ 。通常， $m=M/2$ 。
2. 对于所有在叶子中存储的记录（条目）， $i$  是最小的可以在空间中完全覆盖这些记录所代表的点的矩形（注意：此处所说的“矩形”是可以扩展到高维空间的）。
3. 每一个非叶子结点拥有  $m$  至  $M$  个孩子结点，除非它是根结点。
4. 对于在非叶子结点上的每一个条目， $i$  是最小的可以在空间上完全覆盖这些条目所代表的点的矩形（同性质 2）。
5. 所有叶子结点都位于同一层，因此 R 树为平衡树。

## 叶子结点的结构

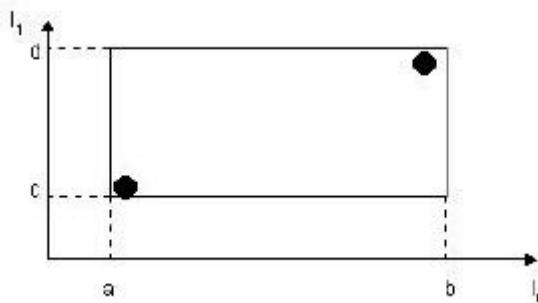
先来探究一下叶子结点的结构。叶子结点所保存的数据形式为：(l, tuple-identifier)。

其中，tuple-identifier 表示的是一个存放于数据库中的 tuple，也就是一条记录，它是 n 维的。I 是一个 n 维空间的矩形，并可以恰好框住这个叶子结点中所有记录代表的 n 维空间中的点。 $I=(l_0, l_1, \dots, l_{n-1})$ 。其结构如下图所示：



R树中结点的存储结构。E代表Entry，即指向孩子结点的条目

下图描述的就是在二维空间中的叶子结点所要存储的信息。



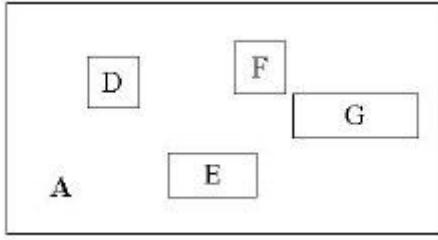
在这张图中，I 所代表的就是图中的矩形，其范围是  $a \leq l_0 \leq b$ ,  $c \leq l_1 \leq d$ 。有两个 tuple-identifier，在图中即表示为那两个点。这种形式完全可以推广到高维空间。大家简单想想三维空间中的样子就可以了。这样，叶子结点的结构就介绍完了。

## 非叶子结点

非叶子结点的结构其实与叶子结点非常类似。想象一下 B 树就知道了，B 树的叶子结点存放的是真实存在的数据，而非叶子结点存放的是这些数据的“边界”，或者说也算是一种索引（有疑问的读者可以回顾一下上述第一节中讲解 B 树的部分）。

同样道理，R 树的非叶子结点存放的数据结构为：(I, child-pointer)。

其中，child-pointer 是指向孩子结点的指针，I 是覆盖所有孩子结点对应矩形的矩形。这边有点拗口，但我想不是很难懂？给张图：



D,E,F,G 为孩子结点所对应的矩形。A 为能够覆盖这些矩形的更大的矩形。这个 A 就是这个非叶子结点所对应的矩形。这时候你应该悟到了吧？无论是叶子结点还是非叶子结点，它们都对应着一个矩形。树形结构上层的结点所对应的矩形能够完全覆盖它的孩子结点所对应的矩形。根结点也唯一对应一个矩形，而这个矩形是可以覆盖所有我们拥有的数据信息在空间中代表的点的。

我个人感觉这张图画的不那么精确，应该是矩形 A 要恰好覆盖 D,E,F,G，而不应该再留出这么多没用的  
空间了。但为尊重原图的绘制者，特不作修改。<sup>”</sup>

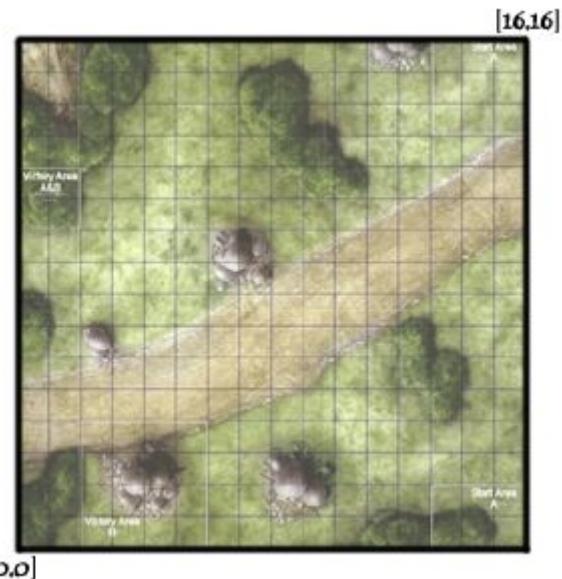
但 R 树有些什么问题呢？如@宋枭\_CD 所说：“单纯用 R 树来作索引，搜索附近的地点，可能会遍历树的很多个分支。而且当全国的地图或者全省的地图时候，树的叶节点数目很多，树的深度也会是一个问题。一般会把地理位置上附近的节点（二维地图中点线面）预处理成 page(大小为 4K 的倍数)，在这些 page 上建立 R 树的索引。”

## 解法二、GeoHash 算法索引地理位置信息

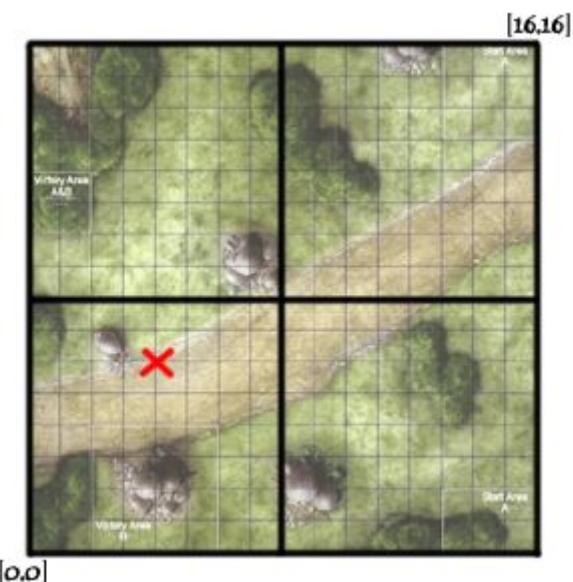
我在微博上跟一些朋友讨论这个附近点搜索的问题时，除了谈到 R 树，有几个朋友都指出 GeoHash 算法可以解决，故才了解了下 GeoHash 算法，此文

<http://blog.nosqlfan.com/html/1811.html> 清晰阐述了 MongoDB 借助 GeoHash 算法实现地理位置索引的原理，特引用其内容加以说明，如下：

“支持地理位置索引是 MongoDB 的一大亮点，这也是全球最流行的 LBS 服务 foursquare 选择 MongoDB 的原因之一。我们知道，通常的数据库索引结构是 B+ Tree，如何将地理位置转化为可建立 B+Tree 的形式。首先假设我们将需要索引的整个地图分成 16×16 的方格，如下图（左下角为坐标 0,0 右上角为坐标 16,16）：



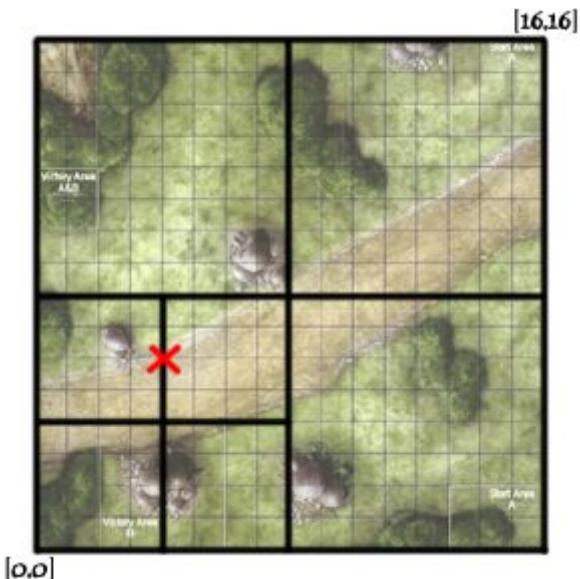
单纯的  $[x, y]$  的数据是无法建立索引的，所以 MongoDB 在建立索引的时候，会根据相应字段的坐标计算一个可以用来做索引的 **hash** 值，这个值叫做 **geohash**，下面我们将以地图上坐标为  $[4, 6]$  的点（图中红叉位置）为例。我们第一步将整个地图分成等大小的四块，如下图：



划分成四块后我们可以定义这四块的值，如下（左下为 00，左上为 01，右下为 10，右上为 11）：

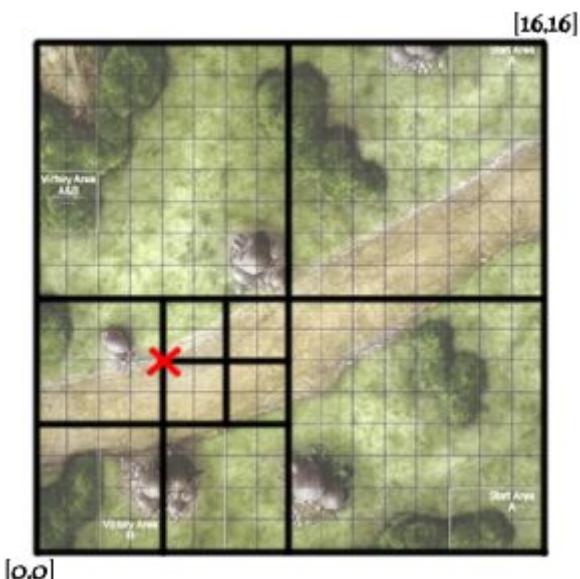
01	11
00	10

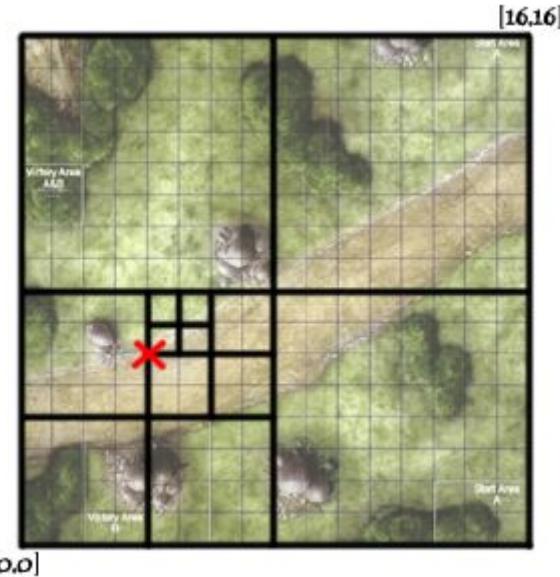
这样 [4, 6] 点的 geohash 值目前为 00 然后再将四个小块每一块进行切割，如下：



这时 [4, 6] 点位于右上区域，右上的值为 11，这样 [4, 6] 点的 geohash 值变为：

0011 继续往下做两次切分：





最终得到 [4, 6] 点的 geohash 值为: 00110100

这样我们用这个值来做索引，则地图上点相近的点就可以转化成有相同前缀的 geohash 值了。

我们可以看到，这个 geohash 值的精确度是与划分地图的次数成正比的，上例对地图划分了四次。而 MongoDB 默认是进行 26 次划分，这个值在建立索引时是可控的。具体建立二维地理位置索引的命令如下：

```
1. db.map.ensureIndex({point : "2d"}, {min : 0, max : 16, bits : 4})
```

其中的 bits 参数就是划分几次，默认为 26 次。 ”

读者点评@yuotulck: 首先多谢博主的文章，不过如果是新手（例如我）看到 geohash 那里可能会有误解：是否相邻可以靠前缀来比较？其实这是错的，例如边界那一块的相邻区域编码的前缀从第一个就不一样了，也就是说在 geohash 里相近的点 hash 值不一定相近。

上面的知识点了解自：

<http://www.cnblogs.com/step1/archive/2009/04/22/1441689.html>，而 geohash 的进一步用法在这里可以了解到：

<http://tech.idv2.com/2011/07/05/geohash-intro/>。

本章完。

## 参考链接及推荐阅读

1. 2012 年九月十月笔试面试八十题：  
[http://blog.csdn.net/v\\_july\\_v/article/details/7974418](http://blog.csdn.net/v_july_v/article/details/7974418);
2. 从 Trie 树（字典树）谈到后缀树：  
[http://blog.csdn.net/v\\_july\\_v/article/details/6897097](http://blog.csdn.net/v_july_v/article/details/6897097);
3. 教你如何迅速秒杀掉：99% 的海量数据处理面试题：  
[http://blog.csdn.net/v\\_july\\_v/article/details/7382693](http://blog.csdn.net/v_july_v/article/details/7382693);
4. 从 B 树、B+树、B\*树谈到 R 树：[http://blog.csdn.net/v\\_july\\_v/article/details/6530142](http://blog.csdn.net/v_july_v/article/details/6530142);
5. 图解 MongoDB 地理位置索引的实现原理：<http://blog.nosqlfan.com/html/1811.html>;
6. 《Hbase 实战》第 8 章、在 HBase 上查询地理信息系统；

## 后记

感谢制作本 PDF 的吴新隆，非常耐心细致，当然也不忘去年 4 月制作了第 1~27 章的第一版 PDF 的吴超同学，要知道，第一版到如今可是已有接近 2 万人下载，一千多条热心留言：[http://download.csdn.net/detail/v\\_july\\_v/4256339](http://download.csdn.net/detail/v_july_v/4256339)。

本编程艺术系列仍会不断创作下去，敬请大家继续保持关注。同时，依然还是开头那句话：发现任何 bug、问题，和可以优化的代码，或任何改进建议或意见，欢迎随时在博客上留言反馈，或私信联系我：<http://weibo.com/julyweibo>，感谢。

编程之美来源于算法，算法之美来源于数学，愿与君共同追求美之所在。

博客上第 38 章再见。

July、二零一三年十二月十日晚于北京天通苑。