

编程

The Method of
Programming



法

面试和算法心得

July 著



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

目錄

1. [介紹](#)
2. [程序员如何准备面试中的算法](#)
3. [第一部分 数据结构](#)
4. [第一章 字符串](#)
 1. [1.0 本章导读](#)
 2. [1.1 旋转字符串](#)
 3. [1.2 字符串包含](#)
 4. [1.3 字符串转换成整数](#)
 5. [1.4 回文判断](#)
 6. [1.5 最长回文子串](#)
 7. [1.6 字符串的全排列](#)
 8. [1.10 本章习题](#)
5. [第二章 数组](#)
 1. [2.0 本章导读](#)
 2. [2.1 寻找最小的 k 个数](#)
 3. [2.2 寻找和为定值的两个数](#)
 4. [2.3 寻找和为定值的多个数](#)
 5. [2.4 最大连续子数组和](#)

- 6. [2.5 跳台阶](#)
- 7. [2.6 奇偶排序](#)
- 8. [2.7 荷兰国旗](#)
- 9. [2.8 矩阵相乘](#)
- 10. [2.9 完美洗牌](#)
- 11. [2.15 本章习题](#)

- 6. [第三章 树](#)
 - 1. [3.0 本章导读](#)
 - 2. [3.1 红黑树](#)
 - 3. [3.2 B树](#)
 - 4. [3.3 最近公共祖先LCA](#)
 - 5. [3.10 本章习题](#)

- 7. [第二部分 算法心得](#)

- 8. [第四章 查找匹配](#)
 - 1. [4.1 有序数组的查找](#)
 - 2. [4.2 行列递增矩阵的查找](#)
 - 3. [4.3 出现次数超过一半的数字](#)

- 9. [第五章 动态规划](#)
 - 1. [5.0 本章导读](#)
 - 2. [5.1 最大连续乘积子串](#)

3. [5.2 字符串编辑距离](#)
4. [5.3 格子取数](#)
5. [5.4 交替字符串](#)
6. [5.10 本章习题](#)
10. [第三部分 综合演练](#)
11. [第六章 海量数据处理](#)
 1. [6.0 本章导读](#)
 2. [6.1 关联式容器](#)
 3. [6.2 分而治之](#)
 4. [6.3 simhash算法](#)
 5. [6.4 外排序](#)
 6. [6.5 MapReduce](#)
 7. [6.6 多层划分](#)
 8. [6.7 Bitmap](#)
 9. [6.8 Bloom filter](#)
 10. [6.9 Trie树](#)
 11. [6.10 数据库](#)
 12. [6.11 倒排索引](#)
 13. [6.15 本章习题](#)
12. [第七章 机器学习](#)

1. [7.1 K 近邻算法](#)
2. [7.2 支持向量机](#)
13. [附录 更多题型](#)
 1. [附录A 语言基础](#)
 2. [附录B 概率统计](#)
 3. [附录C 智力逻辑](#)
 4. [附录D 系统设计](#)
 5. [附录E 操作系统](#)
 6. [附录F 网络协议](#)
14. [sift算法](#)
 1. [sift算法的编译与实现](#)
 2. [教你一步一步用c语言实现sift算法、上](#)
 3. [教你一步一步用c语言实现sift算法、下](#)
15. [其它](#)
 1. [40亿个数中快速查找](#)
 2. [hash表算法](#)
 3. [一致性哈希算法](#)
 4. [倒排索引关键词不重复Hash编码](#)
 5. [傅里叶变换算法、上](#)
 6. [傅里叶变换算法、下](#)

7. [后缀树](#)
8. [基于给定的文档生成倒排索引的编码与实践](#)
9. [搜索关键词智能提示suggestion](#)
10. [最小操作数](#)
11. [最短摘要的生成](#)
12. [最长公共子序列](#)
13. [木块砌墙原稿](#)
14. [附近地点搜索](#)
15. [随机取出其中之一元素](#)

《编程之法：面试和算法心得》

作者： [julycoding](#)

来源： [The-Art-Of-Programming-By-July](#)

目录

第一部分 数据结构

- 第一章 字符串
 - [1.0 本章导读](#)
 - [1.1 旋转字符串](#)
 - [1.2 字符串包含](#)
 - [1.3 字符串转换成整数](#)
 - [1.4 回文判断](#)
 - [1.5 最长回文子串](#)
 - [1.6 字符串的全排列](#)
 - [1.10 本章习题](#)
- 第二章 数组
 - [2.0 本章导读](#)
 - [2.1 寻找最小的 k 个数](#)
 - [2.2 寻找和为定值的两个数](#)
 - [2.3 寻找和为定值的多个数](#)
 - [2.4 最大连续子数组和](#)
 - [2.5 跳台阶](#)
 - [2.6 奇偶排序](#)
 - [2.7 荷兰国旗](#)
 - [2.8 矩阵相乘](#)
 - [2.9 完美洗牌](#)
 - [2.15 本章习题](#)
- 第三章 树
 - [3.0 本章导读](#)
 - [3.1 红黑树](#)
 - [3.2 B树](#)
 - [3.3 最近公共祖先LCA](#)
 - [3.10 本章习题](#)

第二部分 算法心得

- 第四章 查找匹配

- [4.1 有序数组的查找](#)
 - [4.2 行列递增矩阵的查找](#)
 - [4.3 出现次数超过一半的数字](#)
- 第五章 动态规划
 - [5.0 本章导读](#)
 - [5.1 最大连续乘积子串](#)
 - [5.2 字符串编辑距离](#)
 - [5.3 格子取数](#)
 - [5.4 交替字符串](#)
 - [5.10 本章习题](#)

第三部分 综合演练

- 第六章 海量数据处理
 - [6.0 本章导读](#)
 - [6.1 关联式容器](#)
 - [6.2 分而治之](#)
 - [6.3 simhash算法](#)
 - [6.4 外排序](#)
 - [6.5 MapReduce](#)
 - [6.6 多层划分](#)
 - [6.7 Bitmap](#)
 - [6.8 Bloom filter](#)
 - [6.9 Trie树](#)
 - [6.10 数据库](#)
 - [6.11 倒排索引](#)
 - [6.15 本章习题](#)
- 第七章 机器学习
 - [7.1 K 近邻算法](#)
 - [7.2 支持向量机](#)
- 附录 更多题型
 - [附录A 语言基础](#)
 - [附录B 概率统计](#)
 - [附录C 智力逻辑](#)
 - [附录D 系统设计](#)

- [附录E 操作系统](#)
- [附录F 网络协议](#)
- 注：上述所有的文章已于2014年6月30日基本停止更新（当然，如果有bug，请随时指出，一经确认，立即修正）。所有进一步的修改、改动、优化请见2015年10月14日上市销售的纸质版《编程之法：面试和算法心得》。感谢大家。

July、二零一四年八月十四日。

程序员如何准备面试中的算法

备战面试中算法的五个步骤

对于立志进一线互联网公司，同时不满足于一辈子干纯业务应用开发，希望在后端做点事情的同学来说，备战面试中的算法，分为哪几个步骤呢？如下：

1、掌握一门编程语言

首先你得确保你已掌握好一门编程语言：

- C的话，推荐Dennis M. Ritchie & Brian W. Kernighan合著的《C程序设计语言》，和《C和指针》；
- C++ 则推荐《C++ Primer》，《深度探索C++对象模型》，《Effective C++》；
- Java推荐《Thinking in Java》，《Core Java》，《Effective Java》，《深入理解Java虚拟机》。

掌握一门语言并不容易，不是翻完一两本书即可了事，语言的细枝末节需要在平日不断的编程练习中加以熟练。

2、过一遍微软面试100题系列

我从2010年起开始整理 [微软面试100题系列](#)，见过的题目不可谓不多，但不管题目怎样变化，依然是那些常见的题型和考察点，当然，不考察任何知识点，纯粹考察编程能力的题目也屡见不鲜。故不管面试题千变万化，始终不离两点：①看你基本知识点的掌握情况；②编程基本功。

而当你看了一遍微软面试100题之后(不要求做完)，你自会意识到：数据结构和算法在笔试面试中的重要性。

3、苦补数据结构基础

如果学数据结构，可以看我们在大学里学的任一本数据结构教材都行，

如果你觉得实在不够上档次，那么可以再看看《STL源码剖析》。

然后，你会发现：大部分的面试题都在围绕一个点：基于各种数据结构上的增删改查。如字符串的查找翻转，链表的查找遍历合并删除，树和图的查找遍历，后来为了更好的查找，我们想到了排序，排序仍然不够，我们有了贪心、动态规划，再后来东西多了，于是有了海量数据处理，资源有限导致人们彼此竞争，出现了博弈组合概率。

4、看算法导论

《算法导论》上的前大部分的章节都在阐述一些经典常用的数据结构和典型算法（如 [二分查找](#)，[快速排序](#)、[Hash表](#)），以及一些高级数据结构（诸如 [红黑树](#)、[B树](#)），如果你已经学完了一本数据结构教材，那么建议你着重看贪心、动态规划、图论等内容，这3个议题每一个议题都大有题目可出。同时，熟悉 [常用算法的时间复杂度](#)。

如果算法导论看不懂，你可以参看本博客。

5、刷leetcode或cc150或编程艺术系列

- 如主要在国外找工作，推荐两个面试编程网站：一个是 <http://leetcode.com/>，leetcode是国外一网站，它上面有不少编程题；另外一个 <http://www.careercup.com/>，而后这个网站的创始人写了本书，叫《careercup cracking coding interview》，最终这本英文书被图灵教育翻译出版为《程序员面试金典》。
- 若如果是国内找工作，则郑重推荐我编写的《程序员编程艺术》，有 [编程艺术博客版](#)，以及在博客版本基础上精简优化的 [编程艺术github版](#)。除此之外，还可看看《编程之美》，与《剑指offer》。

而不论是准备国内还是国外的海量数据处理面试题，此文必看：[教你如何迅速秒杀掉：99%的海量数据处理面试题](#)。

此外，多看看优秀的开源代码，如 [nginx](#) 或 [redis](#)，多做几个项目加以实践之，尽早实习（在一线互联网公司实习3个月可能胜过你自个黑灯瞎

火摸爬滚打一年）。

当然，如果你是准备社招，且已经具备了上文所说的语言 & 数据结构 & 算法基础，可以直接跳到本第五步骤，开始刷leetcode或cc150或编程艺术系列。

后记

学习最忌心浮气躁，急功近利，即便练习了算法，也不一定代表能万无一失通过笔试面试关，因为总体说来，在一般的笔试面试中，70% 基础 + 30% **coding**能力 (含算法)，故如果做到了上文中的5个步骤，还远远不够，最后，我推荐一份非算法的书单，以此为大家查漏补缺(不必全部看完，欢迎大家补充)：

1. 《深入理解计算机系统》
2. W.Richard Stevens著的《TCP/IP详解三卷》，《UNIX网络编程二卷》，《UNIX环境高级编程：第2版》，详见此 [豆瓣页面](#)；
3. 你如果要面机器学习一类的岗位，建议看看相关的算法（如 [支持向量机通俗导论（理解SVM的三层境界）](#)），及老老实实补补数学基础，包括微积分、线性代数、概率论与数理统计（除了教材，推荐一本《数理统计学简史》）、矩阵论（推荐《矩阵分析与应用》）等..

最后望大家循序渐进，踏实前进，若实在觉得算法 & 编程太难，转产品、运营、测试、运维、前端、设计都是不错的选择，因为虽然编程有趣，但不一定人人适合编程。

第一部分 数据结构

第一章 字符串

本章导读

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

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

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

1.1 旋转字符串

题目描述

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

分析与解法

解法一：暴力移位法

初看此题，可能最先想到的方法是按照题目所要求的，把需要移动的字符一个一个地移动到字符串的尾部，如此我们可以实现一个函数

`LeftShiftOne(char* s, int n)`，以完成移动一个字符到字符串尾部的功能，代码如下所示：

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

因此，若要把字符串开头的m个字符移动到字符串的尾部，则可以如下操作：

```
void LeftRotateString ( char * s, int n, int m)
{
    while (m--)
```

```

    {
        LeftShiftOne(s, n);
    }
}

```

下面，我们来分析一下这种方法的时间复杂度和空间复杂度。

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

解法二：三步反转法

对于这个问题，换一个角度思考一下。

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

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

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

如下图所示：



代码则可以这么写：

```

void ReverseString ( char * s, int from, int to)
{
    while (from < to)
    {
        char t = s[from];
        s[from++] = s[to];
        s[to--] = t;
    }
}

void LeftRotateString ( char * s, int n, int m)
{
    m %= n; //若要左移动大于n位，那么和%n 是等价的

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

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

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

```

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

举一反三

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

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

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

字符串包含

题目描述

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

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

比如，如果是下面两个字符串：

String 1: ABCD

String 2: BAD

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

如果是下面两个字符串：

String 1: ABCD

String 2: BCE

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

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

分析与解法

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

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

解法一

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

代码可如下编写：

```
bool StringContain (string &a,string &b)
{
    for ( int i = 0 ; i < b.length(); ++i) {
        int j;
        for (j = 0 ; (j < a.length()) && (a[j] != b[i]); ++j)
            ;
        if (j >= a.length())
        {
            return false ;
        }
    }
    return true ;
}
```

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

解法二

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

关于排序方法，可采用最常用的快速排序，参考代码如下：

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

```
bool StringContain (string &a,string &b)
{
    sort(a.begin(),a.end());
    sort(b.begin(),b.end());
    for ( int pa = 0 , pb = 0 ; pb < b.length(); )
    {
        while ((pa < a.length()) && (a[pa] < b[pb]))
        {
            ++pa;
        }
        if ((pa >= a.length()) || (a[pa] > b[pb]))
        {
            return false ;
        }
        //a[pa] == b[pb]
        ++pb;
    }
    return true ;
}
```

解法三

有没有比快速排序更好的方法呢？

我们换一种角度思考本问题：

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

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

思路总结如下：

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

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

//此方法只有理论意义，因为整数乘积很大，有溢出风险

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

    int f = 1 ;

    for ( int i = 0 ; i < a.length(); ++i)
    {
        int x = p[a[i] - 'A' ];

        if (f % x)
```

```

    {
        f *= x;
    }
}

for ( int i = 0 ; i < b.length(); ++i)
{
    int x = p[b[i] - 'A' ];
    if (f % x)
    {
        return false ;
    }
}

return true ;
}

```

此种素数相乘的方法看似完美，但缺点是素数相乘的结果容易导致整数溢出。

解法四

如果面试官继续追问，还有没有更好的办法呢？计数排序？除了计数排序呢？

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

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

```

// “最好的方法”，时间复杂度O(n + m)，空间复杂度O(1)

bool StringContain (string &a,string &b)
{
    int hash = 0 ;

```

```

    for ( int i = 0 ; i < a.length(); ++i)
    {
        hash |= ( 1 << (a[i] - 'A' ));
    }

    for ( int i = 0 ; i < b.length(); ++i)
    {
        if ((hash & ( 1 << (b[i] - 'A' ))) == 0 )
        {
            return false ;
        }
    }

    return true ;
}

```

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

举一反三

1、变位词

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

字符串转换成整数

题目描述

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

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

分析与解法

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

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

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

思路有了，你可能不假思索，写下如下代码：

```
int StrToInt ( const char *str)
{
    int n = 0 ;
    while (*str != 0 )
    {
        int c = *str - '0' ;
        n = n * 10 + c;
        ++str;
    }
    return n;
}
```

显然，上述代码忽略了以下细节：

1. 空指针输入：输入的是指针，在访问空指针时程序会崩溃，因此在

使用指针之前需要先判断指针是否为空。

2. 正负符号：整数不仅包含数字，还有可能是以'+'或 '-' 开头表示正负整数，因此如果第一个字符是 '-' 号，则要把得到的整数转换成负整数。
3. 非法字符：输入的字符串中可能含有不是数字的字符。因此，每当碰到这些非法的字符，程序应停止转换。
4. 整型溢出：输入的数字是以字符串的形式输入，因此输入一个很长的字符串将可能导致溢出。

上述其它问题比较好处理，但溢出问题比较麻烦，所以咱们来重点看下溢出问题。

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

我们先设置一些变量：

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

而后，你可能会编写如下代码段处理溢出问题：

```
//当发生正溢出时，返回INT_MAX
if ((sign == '+' ) && (c > MAX_INT - n * 10 ))
{

    n = MAX_INT;

    break ;
}

//发生负溢出时，返回INT_MIN
else if ((sign == '-' ) && (c - 1 > MAX_INT - n * 10 ))
{

    n = MIN_INT;

    break ;
}
```

```
}
```

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

为什么呢？因为当给定字符串"10522545459"时，而MAX_INT是2147483647，即 $\text{MAX_INT}(2147483647) < n \cdot 10(1052254545 \cdot 10)$ ，所以当扫描到最后一个字符'9'的时候，执行上面的这行代码：

```
c > MAX_INT - n * 10
```

已无意义，因为此时 $(\text{MAX_INT} - n \cdot 10)$ 已经小于0，程序已经出错。

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

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

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

对于上面第二种方式，先举两个例子说明下：

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

回MAX_INT。

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

如此我们可以写出正确的处理溢出的代码：

```
c = *str - '0' ;

if (sign > 0 && (n > MAX_INT / 10 || (n == MAX_INT / 10 && c > MAX_INT % 10 )))
{
    n = MAX_INT;

    break ;
}

else if (sign < 0 && (n > ( unsigned )MIN_INT / 10 || (n == ( unsigned )MIN_INT / 10 && c > ( unsigned )MIN_INT % 10 )))
{
    n = MIN_INT;

    break ;
}
```

从而，字符串转换成整数，完整的参考代码为：

```
int StrToInt ( const char * str)
{
    static const int MAX_INT = ( int )(( unsigned )~ 0 >> 1 );
    static const int MIN_INT = -( int )(( unsigned )~ 0 >> 1 ) - 1 ;

    unsigned int n = 0 ;

    //判断是否输入为空

    if (str == 0 )
    {
```

```

return 0 ;

}

//处理空格
while ( isspace (*str))
    ++str;

//处理正负
int sign = 1 ;
if (*str == '+' || *str == '-' )
{
    if (*str == '-' )
        sign = - 1 ;
    ++str;
}

//确定是数字后才执行循环
while ( isdigit (*str))
{
    //处理溢出
    int c = *str - '0' ;

    if (sign > 0 && (n > MAX_INT / 10 || (n == MAX_INT / 10
&& c > MAX_INT % 10 )))
    {
        n = MAX_INT;
        break ;
    }
    else if (sign < 0 && (n > ( unsigned )MIN_INT / 10 || (n == (
unsigned )MIN_INT / 10 && c > ( unsigned )MIN_INT % 10 )))
    {
        n = MIN_INT;

```

```
        break ;  
    }  
  
    //把之前得到的数字乘以10，再加上当前字符表示的数字。  
    n = n * 10 + c;  
    ++str;  
}  
return sign > 0 ? n : -n;  
}
```

举一反三

1. 实现string到double的转换

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

回文判断

题目描述

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

那么，我们的第一个问题就是：判断一个字串是否是回文？

分析与解法

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

解法一

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

代码如下：

```
bool IsPalindrome ( const char *s, int n)
{
    // 非法输入
    if (s == NULL || n < 1 )
    {
        return false ;
    }

    const char * front,*back;

    // 初始化头指针和尾指针
    front = s;
    back = s+ n - 1 ;

    while (front < back)
    {
        if (*front != *back)
        {
            return false ;
        }
    }
}
```

```

    }

    ++front;

    --back;

}

return true ;

}

```

这是一个直白且效率不错的实现，时间复杂度： $O(n)$ ，空间复杂度： $O(1)$ 。

解法二

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

```

bool IsPalindrome2 ( const char *s, int n)
{
    if (s == NULL || n < 1 )
    {
        return false ;
    }

    const char * first, *second;

    // m定位到字符串的中间位置
    int m = ((n >> 1 ) - 1 ) >= 0 ? (n >> 1 ) - 1 : 0 ;

    first = s + m;
    second = s + n - 1 - m;

    while (first >= s)
    {
        if (*first!= *second)
        {
            return false ;
        }
    }
}

```

```
    }  
    --first;  
    ++second;  
}  
return true ;  
}
```

时间复杂度： $O(n)$ ，空间复杂度： $O(1)$ 。

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

举一反三

1、判断一条单向链表是不是“回文”

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

2、判断一个栈是不是“回文”

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

最长回文子串

题目描述

给定一个字符串，求它的最长回文子串的长度。

分析与解法

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

解法一

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

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

```
int LongestPalindrome ( const char *s, int n)
{
    int i, j, max, c;
    if (s == 0 || n < 1 )
        return 0 ;
    max = 0 ;

    for (i = 0 ; i < n; ++i) { // i is the middle point of the palindrome
        for (j = 0 ; (i - j >= 0 ) && (i + j < n); ++j){
            // if the length of the palindrome is odd
            if (s[i - j] != s[i + j])
                break ;
            c = j * 2 + 1 ;
        }
    }
}
```

```

    }

    if (c > max)

        max = c;

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

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

            break ;

        c = j * 2 + 2 ;

    }

    if (c > max)

        max = c;

    }

    return max;

}

```

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

解法二、 $O(N)$ 解法

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

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

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

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

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

比如 S 和 P 的对应关系：

- $S \# 1 \# 2 \# 2 \# 1 \# 2 \# 3 \# 2 \# 1 \#$
- $P \ 1 \ 2 \ 1 \ 2 \ 5 \ 2 \ 1 \ 4 \ 1 \ 2 \ 1 \ 6 \ 1 \ 2 \ 1 \ 2 \ 1$

可以看出， $P[i]-1$ 正好是原字符串中最长回文串的总长度，为5。

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

- 如果 $mx > i$ ，那么 $P[i] \geq \text{Min}(P[2 * id - i], mx - i)$

C代码如下：

```
//mx > i, 那么P[i] >= MIN(P[2 * id - i], mx - i)
//故谁小取谁
if (mx - i > P[2 * id - i])
    P[i] = P[2 * id - i];
else //mx-i <= P[2*id - i]
    P[i] = mx - i;
```

下面，令 $j = 2 * id - i$ ，也就是说 j 是 i 关于 id 的对称点。

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



当 $P[j] \geq mx - i$ 的时候，以 $S[j]$ 为中心的回文子串不一定完全包含于以 $S[id]$ 为中心的回文子串中，但是基于对称性可知，下图中两个绿框所包围的部分是相同的，也就是说以 $S[i]$ 为中心的回文子串，其向右至少会

扩张到mx的位置，也就是说 $P[i] \geq mx - i$ 。至于mx之后的部分是否对称，再具体匹配。



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

综上，关键代码如下：

```
//输入，并处理得到字符串s
int p[ 1000 ], mx = 0 , id = 0 ;
memset (p, 0 , sizeof (p));
for (i = 1 ; s[i] != '\0' ; i++)
{
    p[i] = mx > i ? min(p[ 2 * id - i], mx - i) : 1 ;
    while (s[i + p[i]] == s[i - p[i]])
        p[i]++;
    if (i + p[i] > mx)
    {
        mx = i + p[i];
        id = i;
    }
}
//找出p[i]中最大的
```

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

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

字符串的全排列

题目描述

输入一个字符串，打印出该字符串中字符的所有排列。

例如输入字符串abc，则输出由字符a、b、c所能排列出来的所有字符串abc、acb、bac、bca、cab和cba。

分析与解法

解法一、递归实现

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

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

代码可如下编写所示：

```
void CalcAllPermutation ( char * perm, int from, int to)
{
    if (to <= 1 )
    {
        return ;
    }

    if (from == to)
    {
```

```

        for ( int i = 0 ; i <= to; i++)
            cout << perm[i];

        cout << endl;

    }

    else
    {
        for ( int j = from; j <= to; j++)
        {
            swap(perm[j], perm[from]);

            CalcAllPermutation(perm, from + 1 , to);

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

        }

    }

}

```

解法二、字典序排列

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

$(a,b) \leq (a',b')$ 当且仅当 $a < a'$ 或 $(a = a' \text{ 且 } b \leq b')$ 。

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

那有没有这样的算法，使得

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

答案是肯定的：有，即是STL中的next_permutation算法。

在了解next_permutation算法是怎么一个过程之前，咱们得先来分析

下“下一个排列”的性质。

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

现在的问题是：找到x和y。怎么找到呢？咱们来看一个例子。

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

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

由这个例子可以得出next_permutation算法流程为：

next_permutation算法

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

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

- $x = 1$;
- $y = 3$
- 1和3交换
 - 得23541
- 翻转541
 - 得23145

23145即为所求的21543的下一个排列。参考实现代码如下：

```
bool CalcAllPermutation ( char * perm, int num) {
    int i;

    //①找到排列中最后（最右）一个升序的首位位置i, x = ai
    for (i = num - 2 ; (i >= 0 ) && (perm[i] >= perm[i + 1 ]); --i){
        ;
    }

    // 已经找到所有排列
    if (i < 0 ){
        return false ;
    }

    int k;

    //②找到排列中第i位右边最后一个比ai 大的位置j, y = aj
    for (k = num - 1 ; (k > i) && (perm[k] <= perm[i]); --k){
        ;
    }

    //③交换x, y
    swap(perm[i], perm[k]);

    //④把第(i+ 1)位到最后的的部分翻转
    reverse(perm + i + 1 , perm + num);

    return true ;
}
```

```
}
```

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

解法总结

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

类似问题

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

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

```
//copyright@ 一直很安静 && World Gao
//假设str已经有序

void perm ( char * result, char *str, int size, int resPos)
{
    if (resPos == size)
        printf ( "%s\n" , result);
    else
    {
        for ( int i = 0 ; i < size; ++i)
        {
            result[resPos] = str[i];
            perm(result, str, size, resPos + 1 );
        }
    }
}
```

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

3、写一个程序，打印出以下的序列。

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

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

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

....

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

本章字符串和链表的习题

1、第一个只出现一次的字符

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

2、对称子字符串的最大长度

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

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

3、编程判断俩个链表是否相交

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

问题扩展：

- 如果链表可能有环列？
- 如果要求出俩个链表相交的第一个节点列？

4、逆序输出链表

输入一个链表的头结点，从尾到头反过来输出每个结点的值。

5、在O(1)时间内删除单链表结点

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

6、找出链表的第一个公共结点

两个单向链表，找出它们的第一个公共结点。

7、在字符串中删除特定的字符

输入两个字符串，从第一个字符串中删除第二个字符串中所有的字符。

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

8、字符串的匹配

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

9、字符个数的统计

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

10、最小子串

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

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

提示：文章也可以理解为一个大的字符串数组，单词之前只有空格，没有标点符号。

11、字符串的集合

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

提示：并查集。

12、五笔编码

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

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

13、最长重复子串

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

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

14、字符串的压缩

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

15、最大重复出现子串

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

给定一个字符串，求出其最长的重复子串。

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

- 生成后缀数组 $O(N)$

- 排序 $O(N \log N * N)$ 最后面的 N 是因为字符串比较也是 $O(N)$
- 依次检测相邻的两个字符串 $O(N * N)$

故最终总的时间复杂度是 $O(N^2 * \log N)$

16、字符串的删除

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

17、字符串的移动

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

18、字符串的包含

输入：

L:“hello”“july”

S:“hellomehellojuly”

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

19、倒数第n个元素

链表倒数第n个元素。

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

20、回文字符串

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

例如：

habbafgh

输出h,abba,f,g,h。

提示：一般的人会想到用后缀数组来解决这个问题。

21、最长连续字符

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

22、字符串反转

实现字符串反转函数。

22、字符串压缩

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

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

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

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

注意：只需要完成该函数功能算法，中间不需要有任何IO的输入输出

示例

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

23、集合的差集

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

24、最长公共子串

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

25、均分01

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

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

考虑到最差情况，则是把字符串切分(n - 1)次形成n个长度为1的串。

26、合法字符串

用n个不同的字符（编号1 - n），组成一个字符串，有如下2点要求：

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

问有多少长度为M且符合条件的字符串。

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

假定n和m皆满足：2<=n,m<=1000000000)。

27、最短摘要生成

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



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

28、实现memcpy函数

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

分析：参考代码如下：

```
void * memcpy ( void *dst, const void *src, size_t count)
```

```
{
```

```
    //安全检查
```

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

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

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

```
    //防止内存重复
```

```

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

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

    while (count--)
    {
        *pdst = *psrc;
        pdst++;
        psrc++;
    }

    return dst;
}

```

29、实现memmove函数

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

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

参考代码如下：

```

//void * memmove ( void * destination, const void * source, size_t num );

void * memmove ( void * dest, void * source, size_t count)
{

    void * ret = dest;

    if (dest <= source || dest >= (source + count))
    {
        //正向拷贝
        //copy from lower addresses to higher addresses
        while (count --)

```

```
        *dest++ = *source++;  
    }  
    else  
    {  
        //反向拷贝  
        //copy from higher addresses to lower addresses  
        dest += count - 1 ;  
        source += count - 1 ;  
  
        while (count--)  
            *dest-- = *source--;  
    }  
    return ret;  
}
```

第二章 数组

本章导读

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

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

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

最后，注意细节处理，不要忽略边界条件，如字符串转换成整数。

寻找最小的k个数

题目描述

输入n个整数，输出其中最小的k个。

分析与解法

解法一

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

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

解法二

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

1、遍历n个数，把最先遍历到的k个数存入到大小为k的数组中，假设它们即是最小的k个数；

2、对这k个数，利用选择或交换排序找到这k个元素中的最大值kmax（找最大值需要遍历这k个数，时间复杂度为 $O(k)$ ）；

3、继续遍历剩余n-k个数。假设每一次遍历到的新的元素的值为x，把x与kmax比较：如果 $x < kmax$ ，用x替换kmax，并回到第二步重新找出k个元素的数组中最大元素kmax'；如果 $x \geq kmax$ ，则继续遍历不更新数组。

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

解法三

更好的办法是维护容量为k的最大堆，原理跟解法二的方法相似：

- 1、用容量为k的最大堆存储最先遍历到的k个数，同样假设它们即

是最小的k个数；

- 2、堆中元素是有序的，令 $k_1 < k_2 < \dots < k_{\max}$ （ k_{\max} 设为最大堆中的最大元素）
- 3、遍历剩余 $n-k$ 个数。假设每一次遍历到的新的元素的值为 x ，把 x 与堆顶元素 k_{\max} 比较：如果 $x < k_{\max}$ ，用 x 替换 k_{\max} ，然后更新堆（用时 $\log k$ ）；否则不更新堆。

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

解法四

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

- 选取 S 中一个元素作为枢纽元 v ，将集合 $S - \{v\}$ 分割成 S_1 和 S_2 ，就像快速排序那样
 - 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 , $k - |S_1| - 1$)。

此算法的平均运行时间为 $O(n)$ 。

示例代码如下：

```
//QuickSelect 将第k小的元素放在 a[k-1]

void QuickSelect ( int a[], int k, int left, int right )
{
    int i, j;
    int pivot;
```

```

    if ( left + cutoff <= right )
    {
        pivot = median3( a, left, right );

        //取三数中值作为枢纽元，可以很大程度上避免最坏情况

        i = left; j = right - 1 ;

        for ( ; ; )
        {
            while ( a[ ++i ] < pivot ){ }
            while ( a[ --j ] > pivot ){ }

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

            else
                break ;
        }

        //重置枢纽元

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

        if ( k <= i )
            QuickSelect( a, k, left, i - 1 );

        else if ( k > i + 1 )
            QuickSelect( a, k, i + 1, right );

    }

    else

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

```

这个快速选择SELECT算法，类似快速排序的划分方法。N个数存储在数组S中，再从数组中选取“中位数的中位数”作为枢纽元X，把数组划分为Sa和Sb两部分， $S_a \leq X \leq S_b$ ，如果要查找的k个元素小于Sa的元素个数，则返回Sa中较小的k个元素，否则返回Sa中所有元素+Sb中小的 $k - |S_a|$ 个元素，这种解法在平均情况下能做到 $O(n)$ 的复杂度。

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

举一反三

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

分析：

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

那么可以把这些和看成N个有序数列：

$A[1]+B[1] \leq A[1]+B[2] \leq A[1]+B[3] \leq \dots$

$A[2]+B[1] \leq A[2]+B[2] \leq A[2]+B[3] \leq \dots$

...

$A[N]+B[1] \leq A[N]+B[2] \leq A[N]+B[3] \leq \dots$

问题转变成，在这 N^2 个有序数列里，找到前k小的元素”

2、有两个序列A和B, $A=(a_1, a_2, \dots, a_k)$, $B=(b_1, b_2, \dots, b_k)$ ，A和B都按升序排列。对于 $1 \leq i, j \leq k$ ，求k个最小的 $(a_i + b_j)$ 。要求算法尽量高效。

3、给定一个数列 $a_1, a_2, a_3, \dots, a_n$ 和m个三元组表示的查询，对于每个查询 (i, j, k) ，输出 a_i, a_{i+1}, \dots, a_j 的升序排列中第k个数。

寻找和为定值的两个数

题目描述

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

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

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

分析与解法

咱们试着一步一步解决这个问题（注意阐述中数列有序无序的区别）：

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

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

答案是二分查找，可以将 $O(N)$ 的查找时间提高到 $O(\log N)$ ，这样对于 N 个 $a[i]$ ，都要花 $\log N$ 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 $O(N \log N + N \log N) = O(N \log N)$ ，空间复杂度总为 $O(1)$ ）。

可以继续优化做到时间 $O(N)$ 么？

解法一

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

- 1、2、4、7、11、15

用输入数字15减一下各个数，得到对应的序列为：

- 14、13、11、8、4、0

第一个数组以一指针 i 从数组最左端开始向右扫描，第二个数组以一指针 j 从数组最右端开始向左扫描，如果第一个数组出现了和第二个数组一样的数，即 $a[i]=a[j]$ ，就找出这两个数来了。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。

解法二

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

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

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

解法三

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

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

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

N)。

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

下面，咱们先来实现此思路（这里假定数组已经是有序的），代码可以如下编写：

```
void TwoSum ( int data[], unsigned int length, int sum)
{
    //sort(s, s+n);    如果数组非有序的，那就事先排好序 $O(N \log N)$ 

    int begin = 0 ;
    int end = length - 1 ;

    //两头夹逼，或称两个指针两端扫描法，很经典的方法， $O(N)$ 

    while (begin < end)
    {
        long currSum = data[begin] + data[end];

        if (currSum == sum)
        {
            //题目只要求输出满足条件的任意一对即可

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

            //如果需要所有满足条件的数组对，则需要加上下面两条语句：

            //begin++
            //end--

            break ;
        }

        else {
            if (currSum < sum)
                begin++;
        }
    }
}
```

```
        else
            end--;
    }
}
```

解法总结

不论原序列是有序还是无序，解决这类题有以下三种办法：

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

所以，要想达到时间 $O(N)$ ，空间 $O(1)$ 的目标，除非原数组是有序的（指针扫描法），不然，当数组无序的话，就只能先排序，后指针扫描法或二分（时间 $O(N \log N)$ ，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，达到平衡。

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

问题扩展

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

举一反三

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

10

/\

5 12

/\

4 7

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

```
struct BinaryTreeNode // a node in the binary tree
{
    int m_nValue; // value of node
    BinaryTreeNode *m_pLeft; // left child of node
    BinaryTreeNode *m_pRight; // right child of node
};
```

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

3、3-sum问题

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

4、4-sum问题

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

寻找和为定值的多个数

题目描述

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

分析与解法

解法一

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

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

参考代码如下：

```
list < int >list1;

void SumOfkNumber ( int sum, int n)
{
    // 递归出口
    if (n <= 0 || sum <= 0 )
        return ;

    // 输出找到的结果
    if (sum == n)
    {
        // 反转list
        list1.reverse();

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

        cout << n << endl;

        list1.reverse() //此处还需反转回来
    }
}
```

```

list1.push_front(n); //典型的01背包问题

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

list1.pop_front();

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

```

解法二

这个问题属于子集和问题（也是背包问题）。本程序采用回溯法+剪枝，其中X数组是解向量， $t = \sum_{i=1}^{k-1} W_i * X_i$, $r = \sum_{i=k}^n W_i$ ，且

- 若 $t + W_k + W(k+1) \leq M$ ，则 $X_k = \text{true}$ ，递归左儿子 $(X_1, X_2, \dots, X(k-1), 1)$ ；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 $(X_1, X_2, \dots, X(k-1), 0)$ ；否则剪枝；

本题中W数组就是(1,2,...,n),所以直接用k代替WK值。

代码编写如下：

```

//输入t, r, 尝试Wk

void SumOfkNumber ( int t, int k, int r, int & M, bool & flag, bool
* X)
{
    X[k] = true ; // 选第k个数

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

        for ( int i = 1 ; i <= k; ++i)
        {
            if (X[i] == 1 )
            {
                printf ( "%d " , i);
            }
        }
    }
}

```

```

    }

    printf ( "\n" );

}

else

{
    // 若第k+1个数满足条件，则递归左子树

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

    {

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

    }

    // 若不选第k个数，选第k+1个数满足条件，则递归右子树

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

    {

        X[k] = false ;

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

    }

}

}

```

```

void  search ( int & N,  int & M)

{

    // 初始化解空间

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

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

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

    if ( 1 > M || sum < M) // 预先排除无解情况

    {

        printf ( "not found\n" );

        return ;

    }

    bool  f = false ;

```

```
SumOfkNumber( 0 , 1 , sum, M, f, X);  
  
if (!f)  
{  
    printf ( "not found\n" );  
}  
  
free (X);  
}
```


0-1背包问题

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

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

$$\bullet F[i, v] = \max\{F[i-1, v], F[i-1, v-C_i] + W_i\}$$

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

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

伪代码如下：

```
F[ 0 , 0..V] ← 0
for i ← 1 to N
    for v ← Ci to V
        F[i, v] ← max{F[i-1, v], F[i-1, v-Ci] + Wi}
```

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

举一反三

1、《挑战程序设计竞赛》的开篇有个类似的抽签问题，挺有意思，题目如下：

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

请编写一个程序，当纸片上所写的数字是 $k_1, k_2, k_3, k_4, \dots, k_n$ 时，是否存在抽取4次和为 m 的方案，如果存在，输出YES；否则，输出NO。

限制条件：

- $1 \leq n \leq 50$
- $1 \leq m \leq 10^8$
- $1 \leq k_i \leq 10^8$

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

//通过4重for循环枚举所有方案

```
for (int a = 0; a < n; a++)
{
    for (int b = 0; b < n; b++)
    {
        for (int c = 0; c < n; c++)
        {
            for (int d = 0; d < n; d++)
            {
                if (k[a] + k[b] + k[c] + k[d] == m)
                {
                    f = true;
                }
            }
        }
    }
}
```

```

        }
    }
}
}
}

```

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

提供两个思路：

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

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

2、给定整数 a_1 、 a_2 、 a_3 、...、 a_n ，判断是否可以从选出若干个数，使得它们的和等于 k （ k 任意给定，且满足 $-10^8 \leq k \leq 10^8$ ）。

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

3、有 n 个数，输出期中所有和为 s 的 k 个数的组合。

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

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

部组合的话，

- 1 2
- 1 2 0
- 0 3
- 1 1 1
- 1 1 1 0

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

最大连续子数组和

题目描述

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

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

分析与解法

解法一

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

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

参考代码如下：

```
int MaxSubArray ( int * A, int n)
{
    int maxSum = a[ 0 ]; //全负情况，返回最大负数
    int currSum = 0 ;
    for ( int i = 0 ; i < n; i++)
    {
        for ( int j = i; j < n; j++)
        {
            for ( int k = i; k <= j; k++)
            {
                currSum += A[k];
            }
            if (currSum > maxSum)
                maxSum = currSum;

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

```
    return maxSum;
}
```

此方法的时间复杂度为 $O(n^3)$ 。

解法二

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

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

即

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

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

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

参考代码如下：

```
int MaxSubArray ( int * a, int n)
{
    int currSum = 0 ;
    int maxSum = a[ 0 ]; //全负情况，返回最大数
```

```
    for ( int j = 0 ; j < n; j++)  
    {  
        currSum = (a[j] > currSum + a[j]) ? a[j] : currSum + a[j];  
        maxSum = (maxSum > currSum) ? maxSum : currSum;  
  
    }  
    return maxSum;  
}
```


问题扩展

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

举一反三

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

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

2、环面上的最大子矩形

《算法竞赛入门经典》 P89 页。

3、最大子矩阵和

一个 $M \times N$ 的矩阵，找到此矩阵的一个子矩阵，并且这个子矩阵的元素的和是最大的，输出这个最大的值。如果所有数都是负数，就输出0。例如：3 5的矩阵：

1 2 0 3 4

2 3 4 5 1

1 1 5 3 0

和最大的子矩阵是：

4 5

5 3

最后输出和的最大值17。

4、允许交换两个数的位置 求最大子数组和。

来源： <https://codility.com/cert/view/certDUMWPM->

[8RF86G8P9QQ6JC8X/details](#) 。

跳台阶问题

题目描述

一个台阶总共有 n 级，如果一次可以跳1级，也可以跳2级。

求总共有多少总跳法，并分析算法的时间复杂度。

分析与解法

解法一

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

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

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

因此 n 级台阶时的不同跳法的总数 $f(n)=f(n-1)+f(n-2)$ 。

我们把上面的分析用一个公式总结如下：

	/ 1	$n = 1$
$f(n)=$	2	$n = 2$
	\ $f(n-1) + f(n-2)$	$n > 2$

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

```

long long Fibonacci ( unsigned int n)
{
    int result[ 3 ] = { 0 , 1 , 2 };
    if (n <= 2 )
        return result[n];

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

```

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

/ 1	n = 1	
f(n)= 2	n = 2	
4	n = 3	//111, 12, 21, 3
\ f(n-1)+f(n-2)+f(n-3)	n > 3	

解法二

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

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

参考代码如下：

```

//1, 1, 2, 3, 5, 8, 13, 21..
int ClimbStairs ( int n)
{
    int dp[ 3 ] = { 1 , 1 };
    if (n < 2 )
    {
        return 1 ;
    }
}

```

```

    }

    for ( int i = 2 ; i <= n; i++)
    {
        dp[ 2 ] = dp[ 0 ] + dp[ 1 ];
        dp[ 0 ] = dp[ 1 ];
        dp[ 1 ] = dp[ 2 ];
    }

    return dp[ 2 ];
}

```

举一反三

1、兔子繁殖问题

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

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

2、换硬币问题。

想兑换100元钱，有1,2,5,10四种钱，问总共有多少兑换方法。

```

const int N = 100;

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

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

for (int i = 0; i < sizeof(dimes) / sizeof(int); ++i)
{
    for (int j = dimes[i]; j <= N; ++j)
    {

```

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

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

奇偶调序

题目描述

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

分析与解法

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

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

而partition过程有以下两种实现：

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

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

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

解法一

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

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

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

思路有了，接下来，写代码实现：

```
//判断是否为奇数

bool IsOddNumber ( int data)
{
    return data & 1 == 1 ;
}

//奇偶互换

void OddEvenSort ( int *pData, unsigned int length)
{
    if (pData == NULL || length == 0 )
        return ;

    int *pBegin = pData;
    int *pEnd = pData + length - 1 ;

    while (pBegin < pEnd)
    {
```

```

        //如果pBegin指针指向的是奇数，正常，向右移
        if (IsOddNumber(*pBegin))
        {
            pBegin++;
        }

        //如果pEnd指针指向的是偶数，正常，向左移
        else if (!IsOddNumber(*pEnd))
        {
            pEnd--;
        }
        else
        {
            //否则都不正常，交换
            //swap是STL库函数，声明为void swap(int& a, int& b);
            swap(*pBegin, *pEnd);
        }
    }
}

```

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

解法二

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

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

```

PARTITION(A, p, r)
1  x ← A[r]
2  i ← p - 1

```

```

3  for j ← p to r - 1
4      do if A[j] ≤ x
5          then i ← i + 1
6              exchange A[i] <-> A[j]
7  exchange A[i + 1] <-> A[r]
8  return i + 1

```

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

① `j` 指向元素2时，`i` 也指向元素2，2与2互换不变

`i` `p/j`

2 8 7 1 3 5 6 4(主元)

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

`i` `j`

2 1 7 8 3 5 6 4

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

`i` `j`

2 1 3 8 7 5 6 4

④ `j` 一路后移，没有再碰到比主元4小的元素：

`i` `j`

2 1 3 8 7 5 6 4

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

2 1 3 4 7 5 6 8

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

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

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

参考代码如下：

```
//奇偶互换
void OddEvenSort2 ( int data[], int lo, int hi)
{
    int i = lo - 1 ;
    for ( int j = lo; j < hi; j++)
    {
        //data[j]指向奇数，交换
        if ( IsOddNumber(data[j]) )
        {
            i = i + 1 ;
            swap(data[i], data[j]);
        }
    }
}
```

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

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

举一反三

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

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

荷兰国旗

题目描述

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



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

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

分析与解法

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

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

解法一

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

1. current指针所指元素为0时，与begin指针所指的元素交换，而后

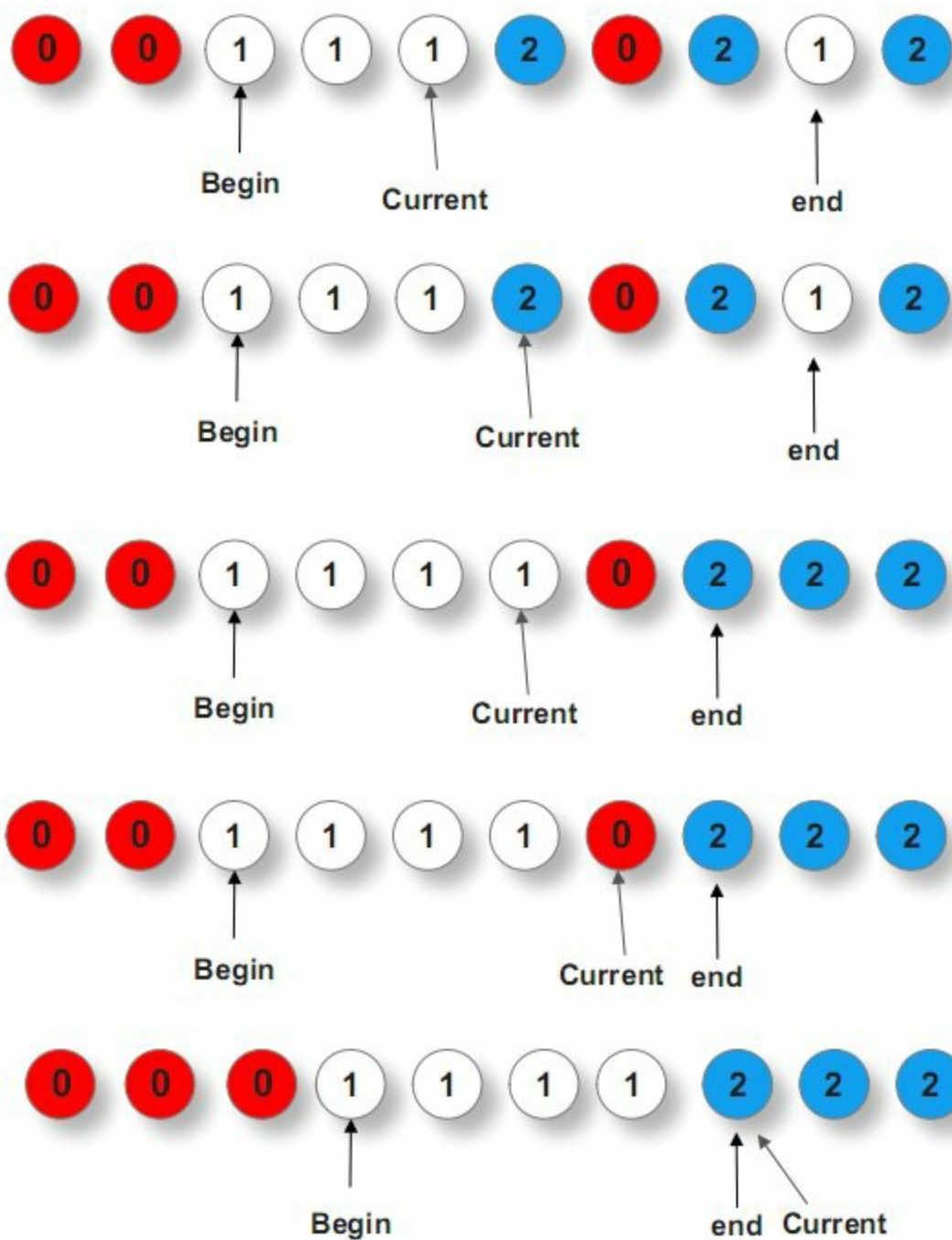
`current++`, `begin++` ;

2. `current`指针所指元素为1时，不做任何交换（即球不动），而后
`current++` ;

3. `current`指针所指元素为2时，与`end`指针所指的元素交换，而后，
`current`指针不动，`end--` 。

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

ok，说这么多，你可能不甚明了，直接引用下gnuhpc的图，就一目了然了：



参考代码如下：

```
//引用自gnu hpc
```

```
while ( current<=end )
```

```
{
```



```
if ( array [current] == 0 )
{
    swap( array [current], array [begin]);
    current++;
    begin++;
}
else if ( array [current] == 1 )
{
    current++;
}

else //When array[current] =2
{
    swap( array [current], array [end]);
    end--;
}
}
```

举一反三

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

要求：空间复杂度是 $O(1)$ ，且只能遍历一次字符串。

矩阵相乘

题目描述

请编程实现矩阵乘法，并考虑当矩阵规模较大时的优化方法。

分析与解法

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



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



下面咱们来具体解决这个矩阵相乘的问题。

解法一、暴力解法

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

```
//矩阵乘法，3个for循环搞定
void MulMatrix ( int ** matrixA, int ** matrixB, int ** matrixC)
{
    for ( int i = 0 ; i < 2 ; ++i)
    {
        for ( int j = 0 ; j < 2 ; ++j)
        {
            matrixC[i][j] = 0 ;
            for ( int k = 0 ; k < 2 ; ++k)
            {
                matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
            }
        }
    }
}
```

}

解法二、Strassen算法

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

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

如下图，当给定一个两个二维矩阵A B时：



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



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

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

他是怎么做到的呢？还是用上文A B两个矩阵相乘的例子，他定义了7个变量：



如此，Strassen算法的流程如下：

- 两个矩阵A B相乘时，将A, B, C分成相等大小的方块矩阵：



- 可以看出C是这么得来的：



- 现在定义7个新矩阵（读者可以思考下，这7个新矩阵是如何想到的）：



- 而最后的结果矩阵C 可以通过组合上述7个新矩阵得到：



表面上看，Strassen算法仅仅比通用矩阵相乘算法好一点，因为通用矩阵相乘算法时间复杂度是 $O(n^3)$ ，而Strassen算法复杂度只是 $O(n^{\log_2 7}) = O(n^{2.807})$ 。但随着n的变大，比如当 $n \gg 100$ 时，Strassen算法是比通用矩阵相乘算法变得更有效率。

如下图所示：



根据wikipedia上的介绍，后来，Coppersmith–Winograd 算法把 $N \times N$ 大小的矩阵乘法的时间复杂度降低到了： $O(n^{2.376})$ ，而2010年，Andrew Stothers再度把复杂度降低到了 $O(n^{2.373})$ ，一年后的2011年，Virginia Williams把复杂度最终定格为： $O(n^{2.373})$ 。

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。

分析与解法

解法一、蛮力变换

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

a1, a2, a3, a4, **b1, b2, b3, b4**

变成

a1, b1, a2, b2, a3, b3, a4, b4

1.1、步步前移

仔细观察变换前后两个序列的特点，我们可做如下一系列操作：

第①步、确定b1的位置，即让b1跟它前面的a2, a3, a4交换：

a1, b1, a2, a3, a4, **b2, b3, b4**

第②步、接着确定b2的位置，即让b2跟它前面的a3, a4交换：

a1, b1, a2, b2, a3, a4, **b3, b4**

第③步、b3跟它前面的a4交换位置：

a1, b1, a2, b2, a3, b3, a4, b4

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

1.2、中间交换

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

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

a1, a2, a3 , b1, a4, **b2, b3, b4**

第②步，让最中间的两对元素各自交换：

a1, a2 , b1, a3, b2, a4, **b3, b4**

第③步，交换最中间的三对元素，序列变成：

a1, b1, a2, b2, a3, b3, a4, b4

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

解法二、完美洗牌算法

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



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

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

这个算法解决一个什么问题呢？跟本题有什么联系呢？

Yeah，顾名思义，完美洗牌算法解决的就是一个完美洗牌问题。什么是

完美洗牌问题呢？即给定一个数组 $a_1, a_2, a_3, \dots, a_n, b_1, b_2, b_3, \dots, b_n$ ，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：

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

通过完美洗牌问题，得到：

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

再让上面相邻的元素两两 swap ，即可达到本题的要求：

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

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

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

2.1、位置置换 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： $a_1 a_2 a_3 a_4 b_1 b_2 b_3 b_4$ 数组下标： 1 2 3 4 5 6 7 8 最终序列：
 $b_1 a_1 b_2 a_2 b_3 a_3 b_4 a_4$

从上面的例子我们能看到，前 n 个元素中，
> 第1个元素 a_1 到了原第2个元素 a_2 的位置，即 $1 \rightarrow 2$ ；
> 第2个元素 a_2 到了原第4个元素 a_4 的位置，即 $2 \rightarrow 4$ ；
> 第3个元素 a_3 到了原第6个元素 b_2 的位置，即 $3 \rightarrow 6$ ；
> 第4个元素 a_4 到了原第8个元素 b_4 的位置，即 $4 \rightarrow 8$ ；>

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 b_1 到了原第1个元素 a_1 的位置，即 $5 \rightarrow 1$ ；> 第6个元素 b_2 到了原第3个元素 a_3 的位置，即 $6 \rightarrow 3$ ；> 第7个元素 b_3 到了原第5个元素 b_1 的位置，即 $7 \rightarrow 5$ ；> 第8个元素 b_4 到了原第7个元素 b_3 的位置，即 $8 \rightarrow 7$ ；

推广到一般情况是，后 n 个元素，第 i 个元素去了第 $(2(i - n) - 1) = 2i - (2n + 1) = (2i) \% (2 * n + 1)$ 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 $(2i) \% (2n + 1)$ 的位置。为何呢？因为：> 当 $0 < i < n$ 时，原式 $= (2i) \% (2n + 1) = 2i$ ；> 当 $i > n$ 时，原式 $(2i) \% (2 * n + 1)$ 保持不变。

因此，如果题目允许我们再用一个数组的话，我们直接把每个元素放到该放得位置就好了。也就产生了最简单的方法`pefect_shuffle1`，参考代码如下：

```
// 时间O(n)，空间O(n) 数组下标从1开始
void PefectShuffle1 ( int *a, int n)
{
    int n2 = n * 2 , i, b[N];
    for (i = 1 ; i <= n2; ++i)
    {
        b[(i * 2) % (n2 + 1)] = a[i];
    }
    for (i = 1 ; i <= n2; ++i)
    {
        a[i] = b[i];
    }
}
```

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

与此同时，我也提醒下读者，根据上面变换的节奏，我们可以看出有两个圈，> 一个是1 -> 2 -> 4 -> 8 -> 7 -> 5 -> 1;

一个是3 -> 6 -> 3。

下文2.2.1、走圈算法cycle_leader将再次提到这两个圈。

2.2、完美洗牌算法perfect_shuffle2

2.2.1、走圈算法cycle_leader

因为之前perfect_shuffle1算法未达到时间复杂度 $O(N)$ 并且空间复杂度 $O(1)$ 的要求，所以我们必须得再找一种新的方法，以期能完美的解决本节开头提出的完美洗牌问题。

让我们先来回顾一下2.1节位置置换perfect_shuffle1算法，还记得我之前提醒读者的关于当 $n=4$ 时，通过位置置换让每一个元素到了最后的位置时，所形成的两个圈么？我引用下2.1节的相关内容：

当 $n=4$ 的情况：

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

即通过置换，我们得到如下结论：

“于此同时，我也提醒下读者，根据上面变换的节奏，我们可以看出有两个圈，> 一个是1 -> 2 -> 4 -> 8 -> 7 -> 5 -> 1;

一个是3 -> 6 -> 3。”

这两个圈可以表示为(1,2,4,8,7,5)和(3,6)，且perfect_shuffle1算法也已经告诉了我们，不管你 n 是奇数还是偶数，每个位置的元素都将变为第 $(2*i) \% (2n+1)$ 个元素：

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

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

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

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

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

上面沿着圈走的算法我们给它取名为cycle_leader，这部分代码如下：

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

```
void CycleLeader ( int *a, int from, int mod)
{
    int t,i;

    for (i = from * 2 % mod; i != from; i = i * 2 % mod)
    {
        t = a[i];
        a[i] = a[from];
        a[from] = t;
    }
}
```

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

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

论文原文部分为：



也就是说，利用上述这个结论，我们可以解决这种特殊长度 $2*n =$

(3^k-1) 的数组问题，那么若给定的长度 n 是任意的咋办呢？此时，我们可以采取分而治之算法的思想，把整个数组一分为二，即拆分成两个部分：

让一部分的长度满足神级结论：若 $2*m = (3^k-1)$ ，则恰好 k 个圈，且每个圈头部的起始位置分别是 $1, 3, 9, \dots, 3^{k-1}$ 。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： $1..m$ $m+1..n$, $n+1..n+m$, $n+m+1..2*n$

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

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

这个翻转的代码如下：

```
//翻转字符串时间复杂度 $O(to - from)$ 

void reverse ( int *a, int from, int to)
{
    int t;
    for (; from < to; ++from, --to)
    {
        t = a[from];
        a[from] = a[to];
        a[to] = t;
    }
}
```

```

    }
}

//循环右移num位 时间复杂度O(n)
void RightRotate ( int *a, int num, int n)
{
    reverse(a, 1, n - num);
    reverse(a, n - num + 1, n);
    reverse(a, 1, n);
}

```

翻转后，得到的目标数组的下标为：

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

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

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

既然 $m=4$ ，即让上述数组中有下划线的两个部分交换，得到：

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

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

2.2.3、完美洗牌算法perfect_shuffle3

从上文的分析过程中也就得出了我们的完美洗牌算法，其算法流程为：
 > 输入数组 $A[1..2n]$ > step 1 找到 $2m = 3^k - 1$ 使得 $3^k \leq 2n < 3^{k+1}$ > step 2 把 $a[m+1..n+m]$ 那部分循环移 m 位 > step 3 对每个 $i = 0, 1, 2, \dots, k-1$ ， 3^i 是个圈的头部，做cycle_leader算法，数组长度为 m ，所以对 $2m+1$ 取模。 > step 4 对数组的后面部分 $A[2m+1..2n]$ 继续使用本

算法, 这相当于 n 减小了 m 。

上述算法流程对应的论文原文为:

以上各个步骤对应的时间复杂度分析如下: > 因为循环不断乘3的, 所以时间复杂度 $O(\log n)$ > 循环移位 $O(n)$ > 每个圈, 每个元素只走了一次, 一共 $2*m$ 个元素, 所以复杂度 $\omega(m)$, 而 $m < n$, 所以也在 $O(n)$ 内。
 $T(n - m)$ > 因此总的时间复杂度为 $T(n) = T(n - m) + O(n)$, $m = \omega(n)$, 解得: $T(n) = O(n)$ 。

此完美洗牌算法实现的参考代码如下:

```
//copyright@caopengcs 8/24/2013
//时间 $O(n)$ , 空间 $O(1)$ 

void PerfectShuffle2 ( int *a, int n)
{
    int n2, m, i, k, t;
    for (; n > 1 ;)
    {
        // step 1
        n2 = n * 2 ;
        for (k = 0 , m = 1 ; n2 / m >= 3 ; ++k, m *= 3 )
            ;
        m /= 2 ;
        //  $2m = 3^k - 1$  ,  $3^k \leq 2n < 3^{k+1}$ 

        // step 2
        right_rotate(a + m, m, n);

        // step 3
        for (i = 0 , t = 1 ; i < k; ++i, t *= 3 )
        {
            cycle_leader(a , t, m * 2 + 1 );
        }
    }
}
```

```

    }

    //step 4
    a += m * 2 ;
    n -= m;

}

// n = 1
t = a[ 1 ];
a[ 1 ] = a[ 2 ];
a[ 2 ] = t;
}

```

2.2.4、perfect_shuffle2算法解决其变形问题

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

```

//copyright@caopengcs 8/24/2013
//时间复杂度O(n)，空间复杂度O(1)，数组下标从1开始，调用perfect_shuffle3

void shuffle ( int *a, int n)
{
    int i, t, n2 = n * 2 ;
    PerfectShuffle2(a, n);
    for (i = 2 ; i <= n2; i += 2 )
    {
        t = a[i - 1 ];
        a[i - 1 ] = a[i];
        a[i] = t;
    }
}

```


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

问题扩展

神级结论是如何来的？

我们的问题得到了解决，但本章尚未完，即决定完美洗牌算法的神级结论：若 $2^n = (3^k - 1)$ ，则恰好只有 k 个圈，且每个圈头部的起始位置分别是 $1, 3, 9, \dots, 3^{k-1}$ ，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 之间的所有正数，一个都不能少。这个证明有点麻烦，因为证明过程中会涉及到群论等数论知识，但再远的路一步步走也能到达。

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

- > 概念1 mod 表示对一个数取余数，比如 $3 \text{ mod } 5 = 3$ ， $5 \text{ mod } 3 = 2$ ；
- > 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： $a^d \pmod m$ ，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 $2^0 \pmod 3 = 1$ ， $2^1 \pmod 3 = 2$ ， $2^2 \pmod 3 = 1$ ， $2^3 \pmod 3 = 2$ ， $\{a^0 \pmod m, a^1 \pmod m, a^2\}$ 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 $2^0 \pmod 7 = 1$ ， $2^1 \pmod 7 = 2$ ， $2^2 \pmod 7 = 4$ ， $2^3 \pmod 7 = 1$ ， $2^4 \pmod 7 = 2$ ， $2^5 \pmod 7 = 4$ ， $2^6 \pmod 7 = 1$ ，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 $S = \{a^0 \pmod m, a^1 \pmod m, a^2 \pmod m, \dots\}$ ，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 $S = \{a^0 \pmod m, a^1 \pmod m, a^2 \pmod m, \dots\}$ 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 \square ,st: \square 中最小的正整数 d

再比如，2是9的原根，因为 \square ，为了让 \square 除以9的余数恒等于1，可知最小的正整数 $d=6$ ，而 $\phi(m)=6$ ，满足原根的定义。> 定理1 同余定理：两个整数 a, b ，若它们除以正整数 m 所得的余数相等，则称 a, b 对于模 m 同余，记作 \square ，读做 a 与 b 关于模 m 同余。> 定理2 当 p 为奇素数且 a 是 \square 的原根时 $\Rightarrow a$ 也是 \square 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 $\gcd(3^k, x 3^t) = 3^t$ ，并且 $x 3^t < 3^k$ 。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 $x * 3^t$ 形成的新集合包含了所有与 3^k 的最大公约数为 3^t 的数，它也是一个圈,原先圈的头部是1，这个圈的头部是 3^t 。

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

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

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

所以 $S(1) = \{1, 2\}$

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

比如 $k = 3$ 。我们有：

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

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

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

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

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

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

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

取头为1， $(/27)^* = \{1, 2, 4, 8, 16, 5, 10, 20, 13, 26, 25, 23, 19, 11, 22, 17, 7, 14\}$ ，也就是说，与27互素且小于27的正整数集合为 $\{1, 2, 4, 8, 16, 5, 10, 20, 13, 26, 25, 23, 19, 11, 22, 17, 7, 14\}$ ，因此 $\phi(m) = \phi(27) = 18$ ，从而满足 \square 的最小 $d = 18$ ，故得出2为27的原根；

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

具体点则是：如果 3×2^k 除27的余数能够被9整除，则有一个 n 使得 $3 \times 2^k = 9n \pmod{27}$ ，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， 3×2^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 3×2^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 3×2^k 除以27的余数为6；

当余数为4时， $2^k - 4 = 27m$ ， $\Rightarrow 3 \times 2^k - 12 = 81m$ ，从而 3×2^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 \square ，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 \square 公式可知，上面第 i 位置的数将分别变成下述位置的：

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

根据 i 和 i' 前后位置的变动，我们将得到3个圈：

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

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

9->18->9

没错，这3个圈的数字与咱们之前得到的3个圈一致吻合，验证完毕。

举一反三

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

1、既然完美洗牌问题是给定输入： $a_1, a_2, a_3, \dots, a_N, b_1, b_2, b_3, \dots, b_N$ ，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？

本章数组和队列的习题

1、不用除法运算

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

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

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

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

参考代码如下所示:

```
void Multiplication ( int a[], int output[], int length)
{
    int left = 1 ;
    int right = 1 ;
    for ( int i = 0 ; i < length; i++)
        output[i] = 1 ;
    for ( int i = 0 ; i < length; i++)
    {
        output[i] *= left;
        output[length - i - 1 ] *= right;
        left *= a[i];
    }
}
```



```

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

```

3、找出数组中唯一的重复元素

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

4、找出唯一出现的数

一个数组里，数都是两两出现的，但是有三个数是唯一出现的，找出这三个数。

5、找出反序的个数

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

6、

有两个序列A和B, $A=(a_1, a_2, \dots, a_k)$, $B=(b_1, b_2, \dots, b_k)$, A和B都按升序排列，对于 $1 \leq i, j \leq k$ ，求k个最小的 $(a_i + b_j)$ ，要求算法尽量高效。

8

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

9

20个排序好的数组，每个数组500个数，按照降序排序好的，让找出500

个最大的数。

10

O(1)空间内实现矩阵转置。

11

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

12

从一系列数中筛除尽可能少的数使得从左往右看，这些数是从小到大再从大到小的。

提示：双端 LIS 问题，用 DP 的思想可解。

13

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

例如：

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

var b=[1, 2, 3, 4,5,40]。

14、螺旋矩阵

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

举个例子，给定如下的一个矩阵：



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



15

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

上排的十个数如下:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

举一个例子,

数值: 0,1,2,3,4,5,6,7,8,9

分配: 6,2,1,0,0,0,1,0,0,0

0在下排出现了6次, 1在下排出现了2次,

2在下排出现了1次, 3在下排出现了0次....

以此类推..

16

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

17

一个整数数组, 长度为n, 将其分为m份, 使各份的和相等, 求m的最大值。

比如{3, 2, 4, 3, 6} 可以分成

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

- {3,6}{2,4,3} $m=2$
- {3,3}{2,4}{6} $m=3$

所以 m 的最大值为3。

18

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

19

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

20

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

21、找出数组中两个只出现一次的数字

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

22、找出数组中两个只出现一次的数字

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

23、把数组排成最小的数

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

24、旋转数组中的最小元素

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

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

25

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

写出函数，对输入整数 N 和 M ，输出所有可能的鸡蛋的放法。

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

26

请把一个整形数组中重复的数字去掉。例如：

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

答案应该是：

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

27

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

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

28

在一维坐标轴上有 n 个区间段，求重合区间最长的两个区间段。

29

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

30

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

有 N 个正实数(注意是实数，大小升序排列) $x_1, x_2 \dots x_N$ ，另有一个实数 M 。需要选出若干个 x ，使这几个 x 的和与 M 最接近。请描述实现算法，并指出算法复杂度。

31

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

32

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

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

33

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

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

34

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

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

35

一个整数数组，有 n 个整数，如何找其中 m 个数的和等于另外 $n-m$ 个数的和？

36

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

37

一个环形公路，上面有 N 个站点， A_1, \dots, A_N ，其中 A_i 和 A_{i+1} 之间的距离为 D_i ， A_N 和 A_1 之间的距离为 D_0 。高效的求第 i 和第 j 个站点之间的距离，空间复杂度不超过 $O(N)$ 。

38

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

39

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

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

40

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

41

设子数组A[0:k]和A[k+1:N-1]已排好序($0 \leq k \leq N-1$)。试设计一个合并这两个子数组为排好序的数组A[0:N-1]的算法。要求算法在最坏情况下所用的计算时间为O(N), 只用到O(1)的辅助空间。

提示: 此题来源于在高德纳的计算机程序设计艺术第三卷第五章排序。

42

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

43

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

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

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

44

从1....n中随机输出m个不重复的数

45

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

46、求旋转数组的最小元素

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

47

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

48

一个数组里，数都是两两出现的，但是有三个数是唯一出现的，找出这三个数。

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

49、计算逆序数组对

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

50

有两个序列A和B, $A=(a_1, a_2, \dots, a_k)$, $B=(b_1, b_2, \dots, b_k)$, A和B都按升序排列, 对于 $1 \leq i, j \leq k$, 求k个最小的 $(a_i + b_j)$, 要求算法尽量高效。

51

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

52

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

提示: 匈牙利算法。

53

寻找3个数的中位数

提示: 可以采用两两比较的思路。

54

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

55

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

56

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

57

等价于 $n \times n$ 的矩阵, 填写0, 1, 要求每行每列的都有偶数个1 (没有1也

是偶数个)，问有多少种方法。

58

数组里找到和最接近于0的两个值。

59

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

60

二重歌德巴赫猜想

所有大于等于6的偶数都可以表示成两个（奇）素数之和。

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

61

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

如果是：

- $N \leq 16$, 要求 $K \leq 16 * N$.
- $N \leq 16$, 要求 $K \leq 10 * N$.
- $N \leq 64$, 要求 $K \leq 15 * N$.

62

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

63

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

请写一个算法，将所有数据从小到大进行排序，并说明时间复杂度。

64

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

65

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

66

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

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

67、在从1到 n 的正数中1出现的次数

输入一个整数 n ，求从1到 n 这 n 个整数的十进制表示中1出现的次数。

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

68

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

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

69

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

70

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

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

$A[1]+B[1] \leq A[1]+B[2] \leq A[1]+B[3] \leq \dots$

$A[2]+B[1] \leq A[2]+B[2] \leq A[2]+B[3] \leq \dots$

...

$A[N]+B[1] \leq A[N]+B[2] \leq A[N]+B[3] \leq \dots$

问题转变成，在这N个有序数列里，找到前k小的元素”。

71、求500万以内的所有亲和数

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

72、杨辉三角的变形

1

1 1 1

1 2 3 2 1

1 3 6 7 6 3 1

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

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

73、三元组的数量

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

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

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

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

74、格子涂色

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

- 蓝蓝蓝黄
- 蓝蓝蓝蓝
- 黄蓝蓝蓝

75、寻找直方图中面积最大的矩形

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

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



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



第三章 树

本章导读

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

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

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

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

教你透彻了解红黑树

二叉查找树

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

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

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

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

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

红黑树

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

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

- 1) 每个结点要么是红的，要么是黑的。
- 2) 根结点是黑的。
- 3) 每个叶结点（叶结点即指树尾端NIL指针或NULL结点）是黑的。
- 4) 如果一个结点是红的，那么它的俩个儿子都是黑的。
- 5) 对于任一结点而言，其到叶结点树尾端NIL指针的每一条路径都包含相同数目的黑结点。

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

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



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

树的旋转知识

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

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

衡。

树的旋转，分为左旋和右旋，以下借助图来做形象的解释和介绍：

1.左旋



如上图所示：

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

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

左旋操作的参考代码如下所示（以 x 代替上述的 $pivot$ ）：

```
LEFT-ROTATE(T, x)
1  y ← right[x] ▷ Set y.
2  right[x] ← left[y] ▷ Turn y's left subtree into x's right subtree.
3  p[left[y]] ← x
4  p[y] ← p[x] ▷ Link x's parent to y.
5  if p[x] = nil[T]
6      then root[T] ← y
7      else if x = left[p[x]]
8          then left[p[x]] ← y
9          else right[p[x]] ← y
10 left[y] ← x ▷ Put x on y's left.
11 p[x] ← y
```

2.右旋

右旋与左旋差不多，再此不做详细介绍。



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

红黑树的插入

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

二叉查找树的插入

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

```
TREE-INSERT( $T, z$ )
1   $y \leftarrow \text{NIL}$ 
2   $x \leftarrow \text{root}[T]$ 
3  while  $x \neq \text{NIL}$ 
4      do  $y \leftarrow x$ 
5          if  $\text{key}[z] < \text{key}[x]$ 
6              then  $x \leftarrow \text{left}[x]$ 
7              else  $x \leftarrow \text{right}[x]$ 
8   $p[z] \leftarrow y$ 
9  if  $y = \text{NIL}$ 
10     then  $\text{root}[T] \leftarrow z$   $\div$  Tree  $T$  was empty
11     else if  $\text{key}[z] < \text{key}[y]$ 
12         then  $\text{left}[y] \leftarrow z$ 
13         else  $\text{right}[y] \leftarrow z$ 
```

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

红黑树的插入和插入修复

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

假设插入的结点为 z ，红黑树的插入伪代码具体如下所示：

```
RB-INSERT( $T, z$ )
1   $y \leftarrow \text{nil}[T]$ 
2   $x \leftarrow \text{root}[T]$ 
3  while  $x \neq \text{nil}[T]$ 
4      do  $y \leftarrow x$ 
5          if  $\text{key}[z] < \text{key}[x]$ 
6              then  $x \leftarrow \text{left}[x]$ 
7              else  $x \leftarrow \text{right}[x]$ 
8   $p[z] \leftarrow y$ 
9  if  $y = \text{nil}[T]$ 
10     then  $\text{root}[T] \leftarrow z$ 
11     else if  $\text{key}[z] < \text{key}[y]$ 
12         then  $\text{left}[y] \leftarrow z$ 
13         else  $\text{right}[y] \leftarrow z$ 
14   $\text{left}[z] \leftarrow \text{nil}[T]$ 
15   $\text{right}[z] \leftarrow \text{nil}[T]$ 
16   $\text{color}[z] \leftarrow \text{RED}$ 
17  RB-INSERT-FIXUP( $T, z$ )
```

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

换言之

- 如果插入的是根结点，因为原树是空树，此悖论1：当前结点的父结点是红色且祖父结点的另一个子结点（叔叔结点）是红色。
 dth="0pt">■ 结点，因为原树 >如果插入的叶结点，因为原■">即■和f■为原■"> h■ d■ ■的插入，■有被破坏）■且猝■割■也是什么都不做■（叔叔结点■/ul"10pt" width="0pt"><■点，因为原树 >但当遇到下述3种情况时：

<■点， ■="1" height="0pt" wi="0pt">■ <■点， ■=■为原树>■ <■
点， ■=■为原树>■ <■点， ■=■为原树>t">■ <■点， ■=■为原树>
■：将dth="0pt">。dth="0p">■和叔司■粹■涂黑，祖父■粹■涂
娘诊当前■粹■指向祖父■粹■树■新的当前■粹■重新为始算
法。即■种情况时：

■ <■点, ■=■为■, , 下所示■ul"■ 然■旋转。■ <■点■然"0p■
色叔叔■色叔叔■色叔叔■色叔叔叔叔■ 况■ #000000"CK■
叔叔叔叔■叔叔叔叔■叔叔叔叔■叔叔叔叔叔叔▷ Cas■津■ >■ <■点■然
7■叔叔叔叔■叔叔叔叔■叔叔叔叔■叔叔叔叔■ 叔叔■ :
="#th="0pt"#0000■ 插叔叔■ 叔叔■ 叔叔■ 叔叔■叔叔▷ Cas■
津■ >■ <■点■焯8叔叔■叔叔叔叔■叔叔叔叔■叔叔叔叔■叔叔■叔叔■叔叔
zt"#0000■ else if z = right[p[z]]

```

10      then z ← p[z]           ▷ Case 2

```

11 LEFT-ROTATE(T, z) \triangleright Case 2

如下图所示，变化前[当前结点为7结点]:



变化后:



插入修复情况3: 当前结点的父结点是红色,叔叔结点是黑色,当前结点是其父结点的左子

解法：父结点变为黑色，祖父结点变为红色，在祖父结点

为支点右旋，操作代码为：

```
12                                     color[p[z]] ← BLACK      ▷ (
13                                     color[p[p[z]]] ← RED        ▷ (
14                                     RIGHT-
ROTATE(T, p[p[z]])      ▷ Case 3
```

最后，把根结点涂为黑色，整棵红黑树便重新恢复了平衡。

如下图所示[当前结点为2结点]



变化后：



红黑树的删除

ok，接下来，咱们最后来了解，红黑树的删除操作。

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

二叉查找树的删除

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

1. 没有儿子，即为叶结点。直接把父结点的对应儿子指针设为NULL，删除儿子结点就OK了。

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

二叉查找树的删除代码如下所示：

```
TREE-DELETE(T, z)
1  if left[z] = NIL or right[z] = NIL
2      then y ← z
3      else y ← TREE-SUCCESSOR(z)
4  if left[y] ≠ NIL
5      then x ← left[y]
6      else x ← right[y]
7  if x ≠ NIL
8      then p[x] ← p[y]
9  if p[y] = NIL
10     then root[T] ← x
11     else if y = left[p[y]]
12         then left[p[y]] ← x
13         else right[p[y]] ← x
14 if y ≠ z
15     then key[z] ← key[y]
16     copy y's satellite data into z
```

```
17 return y
```

红黑树的删除和删除修复

OK，回到红黑树上来，红黑树结点删除的算法实现是：

RB-DELETE(T, z) 单纯删除结点的总操作

```
1 if left[z] = nil[T] or right[z] = nil[T]
2   then y ← z
3   else y ← TREE-SUCCESSOR(z)
4 if left[y] ≠ nil[T]
5   then x ← left[y]
6   else x ← right[y]
7 p[x] ← p[y]
8 if p[y] = nil[T]
9   then root[T] ← x
10  else if y = left[p[y]]
11    then left[p[y]] ← x
12    else right[p[y]] ← x
13 if y ≠ z
14   then key[z] ← key[y]
15   copy y's satellite data into z
16 if color[y] = BLACK
17   then RB-DELETE-FIXUP(T, x)
18 return y
```

“在删除结点后，原红黑树的性质可能被改变，如果删除的是红色结点，那么原红黑树的性质依旧保持，此时不用做修正操作，如果删除的结点是黑色结点，原红黑树的性质可能会被改变，我们要对其做修正操作。那么哪些树的性质会发生变化呢，如果删除结点不是树唯一结点，那么删除结点的那一个支的到各叶结点的黑色结点数会发生变

化，此时性质5被破坏。如果被删结点的唯一非空子结点是红色，而被删结点的父结点也是红色，那么性质4被破坏。如果被删结点是根结点，而它的唯一非空子结点是红色，则删除后新根结点将变成红色，违背性质2。”

RB-DELETE-FIXUP(T, x) 恢复与保持红黑性质的工作

```

1 while x ≠ root[T] and color[x] = BLACK
2     do if x = left[p[x]]
3         then w ← right[p[x]]
4             if color[w] = RED
5                 then color[w] ← BLACK
6                     color[p[x]] ← RED
7                     LEFT-
ROTATE(T, p[x])
8                     w ← right[p[x]]
9                     if color[left[w]] = BLACK and color[right[w]] = BLACK
10                        then color[w] ← RED
11                            x ← p[x]
12                        else if color[right[w]] = BLACK
13                            then color[left[w]] ← BLACK
14                                color[w] ← RED
15                                RIGHT-
ROTATE(T, w)
16                                w ← right[p[x]]
17                                color[w] ← color[p[x]]
18                                color[p[x]] ← BLACK
19                                color[right[w]] ← BLACK
20                                LEFT-
ROTATE(T, p[x])
21                                x ← root[T]
22                        else (same as then clause with "right" and "left" exchange)
23 color[x] ← BLACK

```

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

如果是以下情况，恢复比较简单：

- a)当前结点是红+黑色

解法，直接把当前结点染成黑色，结束此时红黑树性质全部恢复。

- b)当前结点是黑+黑且是根结点，解法：什么都不做，结束。

但如果是以下情况呢？：

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

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

下面，咱们便来分别处理这4种删除修复情况。

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

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

```
//调用RB-DELETE-FIXUP(T, x) 的1-8行代码
1 while x ≠ root[T] and color[x] = BLACK
2     do if x = left[p[x]]
3         then w ← right[p[x]]
4             if color[w] = RED
5                 then color[w] ← BLACK
6                     color[p[x]] ← RED
7                     LEFT-
ROTATE(T, p[x])
8                     w ← right[p[x]]
```

变化前：



变化后：



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

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

```
//调用RB-DELETE-FIXUP(T, x) 的9-11行代码
9          if color[left[w]] = BLACK and color[right[w]] = BL/
10          then color[w] ← RED
11          x ← p[x]
```

变化前:



变化后:



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

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

```
//调用RB-DELETE-FIXUP(T, x) 的第12-16行代码
12          else if color[right[w]] = BLACK
13          then color[left[w]] ← BLACK
14          color[w] ← RED
15          RIGHT-ROTATE(T, w)
16          w ← right[p[x]]
```

变化前:



变化后:



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

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

//调用RB-DELETE-FIXUP(T, x) 的第17-21行代码

```
17          color[w] ← color[p[x]]
18          color[p[x]] ← BLACK
19          color[right[w]] ← BLACK
20          LEFT-
ROTATE(T, p[x])
21          x ← root[T]
```

变化前：



变化后：



本文参考

本文参考了算法导论、STL源码剖析、计算机程序设计艺术等资料，并推荐阅读这个PDF：Left-Leaning Red-Black Trees, Dagstuhl Workshop on Data Structures, Wadern, Germany, February, 2008.

下载地址：

<http://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf>。

B树

1.前言：

动态查找树主要有：二叉查找树（Binary Search Tree），平衡二叉查找树（Balanced Binary Search Tree），[红黑树 \(Red-Black Tree\)](#)，B-tree/B+-tree/ B*-tree (B~Tree)。前三者是典型的二叉查找树结构，其查找的时间复杂度 $O(\log_2 N)$ 与树的深度相关，那么降低树的深度自然会提高查找效率。

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

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

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

2.外存储器—磁盘

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

外存储器—磁盘是一种直接存取的存储设备(DASD)。它是以存取时间

变化不大为特征的。可以直接存取任何字符组，且容量大、速度较其它外存设备更快。

2.1 磁盘的构造

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



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

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

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

2.2 磁盘的读/写原理和效率

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

读/写磁盘上某一指定数据需要下面3个步骤：

1. 首先移动臂根据柱面号使磁头移动到所需要的柱面上，这一过程被称为定位或查找。
2. 如上图11.3中所示的6盘组示意图中，所有磁头都定位到了10个盘面的10条磁道上(磁头都是双向的)。这时根据盘面号来确定指定盘

面上的磁道。

3. 盘面确定以后，盘片开始旋转，将指定块号的磁道段移动至磁头下。

经过上面三个步骤，指定数据的存储位置就被找到。这时就可以开始读/写操作了。

访问某一具体信息，由3部分时间组成：

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

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

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

3.B- 树

3.1 什么是B-树

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

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

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

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



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

B树的定义

B树又叫平衡多路查找树。一棵 m 阶的B树（注：切勿简单的认为一棵 n 阶的B树是 m 叉树，虽然存在四叉树，八叉树，KD树，及vp/R树/R*树/R+树/X树/M树/线段树/希尔伯特R树/优先R树等空间划分树，但与B树完全不等同）的特性如下：

1. 树中每个结点最多含有 m 个孩子（ $m \geq 2$ ）；
2. 除根结点和叶子结点外，其它每个结点至少有 $\lceil m/2 \rceil$ 个孩子（其中 $\lceil x \rceil$ 是一个取上限的函数）；
3. 根结点至少有2个孩子（除非B树只包含一个结点：根结点）；
4. 所有叶子结点都出现在同一层，叶子结点不包含任何关键字信息（可以看做是外部结点或查询失败的结点，指向这些结点的指针都为null）；（注：叶子节点只是没有孩子和指向孩子的指针，这些节

点也存在，也有元素。类似红黑树中，每一个NULL指针即当做叶子结点，只是没画出来而已）。

5. 每个非终端结点中包含有 n 个关键字信息：($n, P_0, K_1, P_1, K_2, P_2, \dots, K_n, P_n$)。其中：

a) K_i ($i=1\dots n$)为关键字，且关键字按顺序升序排序 $K_{(i-1)} < K_i$ 。

b) P_i 为指向子树根的结点，且指针 $P_{(i-1)}$ 指向子树种所有结点的关键字均小于 K_i ，但都大于 $K_{(i-1)}$ 。

c) 关键字的个数 n 必须满足： $\lceil m/2 \rceil - 1 \leq n \leq m-1$ 。比如有 j 个孩子的非叶结点恰好有 $j-1$ 个关键码。

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

3.2 B树的类型和节点定义

B树的类型和节点定义如下图所示：



3.3 文件查找的具体过程(涉及磁盘IO操作)

为了简单，这里用少量数据构造一棵3叉树的形式，实际应用中的B树结点中关键字很多的。上面的图中比如根结点，其中17表示一个磁盘文件的文件名；小红方块表示这个17文件内容在硬盘中的存储位置； p_1 表示指向17左子树的指针。

其结构可以简单定义为：

```
typedef struct {  
    /*文件数*/  
    int file_num;
```

```

    /*文件名(key)*/

    char * file_name[max_file_num];

    /*指向子节点的指针*/

    BTreeNode * BTPtr[max_file_num+ 1 ];

    /*文件在硬盘中的存储位置*/

    FILE_HARD_ADDR offset[max_file_num];

}BTreeNode;

```

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

下面，咱们来模拟下查找文件29的过程：

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

分析上面的过程，发现需要 3次磁盘IO操作和3次内存查找 操作。关于内存中的文件名查找，由于是一个有序表结构，可以利用折半查找提高效率。至于IO操作是影响整个B树查找效率的决定因素。

当然，如果我们使用平衡二叉树的磁盘存储结构来进行查找，磁盘4次，最多5次，而且文件越多，B树比平衡二叉树所用的磁盘IO操作次数将越少，效率也越高。

3.4 B树的高度

根据上面的例子我们可以看出，对于辅存做IO读的次数取决于B树的高度。而B树的高度又怎么求呢？

对于一棵含有N个关键字，m阶的B树来说（据B树的定义可知：m满足： $\lceil m/2 \rceil \leq m \leq m$ ，m阶即代表树中任一结点最多含有m个孩子，如5阶代表每个结点最多5个孩子，或俗称5叉树），且从1开始计数的话，其高度h为：



这个B树的高度公式从侧面显示了B树的查找效率是相当高的。为什么呢？因为底数m/2可以取很大，如m可以达到几千，从而在关键字数一定的情况下，使得最终的h值尽量比较小，树的高度比较低。

树的高度降低了，磁盘存取的次数也随着树高度的降低而减少，从而使存取性能也相应提升。

4、B树的插入、删除操作

根据B树的性质可知，如果是一棵m阶的B树，那么有：

- 树中每个结点含有最多含有m个孩子，即m满足： $\lceil m/2 \rceil \leq m \leq m$ 。
- 除根结点和叶子结点外，其它每个结点至少有 $\lceil m/2 \rceil$ 个孩子（其中 $\lceil x \rceil$ 是一个取上限的函数）；
- 除根结点之外的结点的关键字的个数n必须满足： $\lceil m/2 \rceil - 1 \leq n \leq m - 1$ （叶子结点也必须满足此条关于关键字数的性质）。

下面咱们通过另外一个实例来对这棵B树的插入（insert）,删除（delete）基本操作进行详细的介绍。以一棵5阶（即树中任一结点至多含有4个关键字，5棵子树）B树实例进行讲解(如下图所示)：



在上图所示的一棵5阶B树中，读者可以看到关键字数2-4个，内结点孩子数3-5个。关键字数（2-4个）针对包括叶子结点在内的非根结点，孩子数（3-5个）则针对根结点和叶子结点之外的内结点。同时，根结点是必须至少有2个孩子的，不然就成直线型搜索树了。且关键字为大写

字母，顺序为字母升序。

结点定义如下：

```
typedef struct {  
    int Count;           // 当前节点中关键元素数目  
    ItemType Key[ 4 ];   // 存储关键字元素的数组  
    long Branch[ 5 ];    // 伪指针数组，(记录数目)方便判断合并和分裂的情况  
} NodeType;
```



4.1 插入（insert）操作

针对一棵高度为 h 的 m 阶B树，插入一个元素时，首先在B树中是否存在，如果不存在，一般在叶子结点中插入该新的元素，此时分3种情况：

- 如果叶子结点空间足够，即该结点的关键字数小于 $m-1$ ，则直接插入在叶子结点的左边或右边；
- 如果空间满了以致没有足够的空间去添加新的元素，即该结点的关键字数已经有了 m 个，则需要将该结点进行“分裂”，将一半数量的关键字元素分裂到新的其相邻右结点中，中间关键字元素上移到父结点中，而且当结点中关键元素向右移动了，相关的指针也需要向右移。
 - 此外，如果在上述中间关键字上移到父结点的过程中，导致根结点空间满了，那么根结点也要进行分裂操作，这样原来的根结点中的中间关键字元素向上移动到新的根结点中，因此导致树的高度增加一层。

下面咱们通过一个实例来逐步讲解下。插入以下字符字母到一棵空的5阶B树中：C N G A H E K Q M F W L T Z D P R X Y S，而且，因为是5阶B树，所以必有非根结点关键字数小了（小于2个）就合并，大了（超过4个）就分裂。

1. 首先，结点空间足够，刚开始的4个字母可以直接到插入相同的结点中，如下图：



2. 插入H结点时，发现结点空间不够，所以将其分裂成2个结点，移动中间元素G上移到新的根结点中，且把A和C留在当前结点中，而H和N放置在新的右邻居结点中。如下图：



3. 当插入E,K,Q时，不需要任何分裂操作




4. 插入M需要一次分裂，注意到M恰好是中间关键字元素，所以M向上移到父节点中




5. 插入F,W,L,T不需要任何分裂操作



6. 插入Z时，最右的叶子结点空间满了，需要进行分裂操作，中间元素T上移到父节点中 
7. 插入D时，导致最左边的叶子结点被分裂，D恰好也是中间元素，上移到父节点中，然后字母P,R,X,Y直接陆续插入，不需要任何分裂操作



8. 最后，当插入S时，含有N,P,Q,R的结点需要分裂，把中间元素Q上移到父节点中，但是问题来了，因为Q上移导致父结点“D G M T”也满了，所以也要进行分裂，将父结点中的中间元素M上移到新形成的根结点中，从而致使树的高度增加一层。 

4.2、删除(delete)操作

下面介绍删除操作，删除操作相对于插入操作要考虑的情况多点。

- 首先查找B树中需删除的元素,如果该元素在B树中存在，则将该元素在其结点中进行删除，如果删除该元素后，首先判断该元素是否有左右孩子结点
 - 如果有，则上移孩子结点中的某相近元素(“左孩子最右边的节

点”或“右孩子最左边的节点”)到父节点中，然后是移动之后的情况；

- 如果没有，直接删除后，移动之后的情况。

删除元素，移动相应元素之后，如果某结点中元素数目（即关键字数）小于 $\text{ceil}(m/2)-1$ ，则需要看其某相邻兄弟结点是否丰满（结点中元素个数大于 $\text{ceil}(m/2)-1$ ）

- 如果丰满，则向父节点借一个元素来满足条件；
- 如果其相邻兄弟都刚脱贫，即借了之后其结点数目小于 $\text{ceil}(m/2)-1$ ，则该结点与其相邻的某一兄弟结点进行“合并”成一个结点，以此来满足条件。

下面咱们还是以上述插入操作构造的一棵5阶B树（树中除根结点和叶子结点外的任意结点的孩子数 m 满足 $3 \leq m \leq 5$ ，除根结点外的任意结点的关键字数 n 满足： $2 \leq n \leq 4$ ，所以关键字数小于2个就合并，超过4个就分裂）为例，依次删除H,T,R,E。



1. 首先删除元素H，当然首先查找H，H在一个叶子结点中，且该叶子结点元素数目3大于最小元素数目 $\text{ceil}(m/2)-1=2$ ，则操作很简单，咱们只需要移动K至原来H的位置，移动L至K的位置（也就是结点中删除元素后面的元素向前移动）



2. 下一步，删除T,因为T没有在叶子结点中，而是在中间结点中找到，咱们发现他的继承者W(字母升序的下个元素)，将W上移到T的位置，然后将原包含W的孩子结点中的W进行删除，这里恰好删除W后，该孩子结点中元素个数大于2，无需进行合并操作。



3. 下一步删除R，R在叶子结点中,但是该结点中元素数目为2，删除导致只有1个元素，已经小于最小元素数目 $\text{ceil}(5/2)-1=2$,而由前面我们已经知道： 如果其某个相邻兄弟结点中比较丰满（元素个数大于 $\text{ceil}(5/2)-1=2$ ），则可以向父结点借一个元素，然后将最丰满的相邻兄弟结点中上移最后或最前一个元素到父节点中（有没有看到红黑树中左旋操作的影子?）。故在这个实例中，由于右相邻兄弟

结点“X Y Z”比较丰满，而删除元素R后，导致“S”结点稀缺

- 所以原来的“R S”结点先向父节点借一个元素W下移到该叶子结点中，代替原来S的位置，S前移；
- 然后相邻右兄弟结点中的X上移到父结点中；
- 最后相邻右兄弟结点中元素Y和Z前移。



4. 最后一步删除E，删除后会导致很多问题，因为E所在的结点数目刚好达标，刚好满足最小元素个数 ($\text{ceil}(5/2)-1=2$)，而相邻的兄弟结点也是同样的情况，删除一个元素都不能满足条件，所以需要该节点与某相邻兄弟结点进行合并操作；

- 首先移动父结点中的元素（该元素在两个需要合并的两个结点元素之间）下移到其子结点中，
- 然后将这两个结点进行合并成一个结点。所以在该实例中，咱们首先将父节点中的元素D下移到已经删除E而只有F的结点中，然后将含有D和F的结点和含有A,C的相邻兄弟结点进行合并成一个结点。



也许你认为这样删除操作已经结束了，其实不然，在看看上图，对于这种特殊情况，你立即会发现父节点只包含一个元素G，没达标（因为非根节点包括叶子结点的关键字数 n 必须满足于 $2 \leq n \leq 4$ ，而此处的 $n=1$ ），这是不能够接受的。如果这个问题结点的相邻兄弟比较丰满，则可以向父结点借一个元素。假设这时右兄弟结点（含有Q,X）有一个以上的元素（Q右边还有元素），然后咱们将M下移到元素很少的子结点中，将Q上移到M的位置，这时，Q的左子树将变成M的右子树，也就是含有N，P结点被依附在M的右指针上。

所以在这个实例中，咱们没有办法去借一个元素，只能与兄弟结点进行合并成一个结点，而根结点中的唯一元素M下移到子结点，这样，树的高度减少一层。



为了进一步详细讨论删除的情况，再举另外一个实例：

这里是一棵不同的5序B树，那咱们试着删除C



于是将删除元素C的右子结点中的D元素上移到C的位置，但是出现上移元素后，只有一个元素的结点的情况。

又因为含有E的结点，其相邻兄弟结点才刚脱贫（最少元素个数为2），不可能向父节点借元素，所以只能进行合并操作，于是这里将含有A,B的左兄弟结点和含有E的结点进行合并成一个结点。



这样又出现只含有一个元素F结点的情况，这时，其相邻的兄弟结点是丰满的（元素个数为3>最小元素个数2），这样就可以想父结点借元素了，把父结点中的J下移到该结点中，相应的如果结点中J后有元素则前移，然后相邻兄弟结点中的第一个元素（或者最后一个元素）上移到父节点中，后面的元素（或者前面的元素）前移（或者后移）；注意含有K, L的结点以前依附在M的左边，现在变为依附在J的右边。这样每个结点都满足B树结构性质。



从以上操作可看出：除根结点之外的结点（包括叶子结点）的关键字的个数 n 满足： $(\text{ceil}(m/2)-1) \leq n \leq m-1$ ，即 $2 \leq n \leq 4$ 。这也佐证了咱们之前的观点。删除操作完。

5.B+-tree

B+-tree：是应文件系统所需而产生的一种B-tree的变形树。

一棵 m 阶的B+树和 m 阶的B树的异同点在于：

1. 有 n 棵子树的结点中含有 $n-1$ 个关键字；（与B树 n 棵子树有 $n-1$ 个关键字保持一致，参照：

http://en.wikipedia.org/wiki/B%2B_tree#Overview，而下面B+树的图可能有问题，请读者注意)

2. 所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。(而B树的叶子节点并没有包括全部需要查找的信息)
3. 所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。(而B树的非终端节点也包含需要查找的有效信息)



a) 为什么说B+-tree比B树更适合实际应用中操作系统的文件索引和数据库索引？

1. B+-tree的磁盘读写代价更低 B+-tree的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。

举个例子，假设磁盘中的一个盘块容纳16bytes，而一个关键字2bytes，一个关键字具体信息指针2bytes。一棵9阶B-tree(一个结点最多8个关键字)的内部结点需要2个盘块。而B+树内部结点只需要1个盘快。当需要把内部结点读入内存中的时候，B树就比B+树多一次盘块查找时间(在磁盘中就是盘片旋转的时间)。

1. B+-tree的查询效率更加稳定

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

总而言之，B树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。正是为了解决这个问题，B+树应运而生。B+树只要遍历叶子节点就可以实现整棵树的遍历，支持基于范围的查询，而B树不支持range-query这样的操作（或者说效率太低）。

b) B+-tree的应用: VSAM(虚拟存储存取法)文件(来源论文 *the ubiquitous Btree* 作者: D COMER - 1979)



6.B*-tree

B*-tree是B+-tree的变体，在B+树的基础上(所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针)，B*树中非根和非叶子结点再增加指向兄弟的指针；B*树定义了非叶子结点关键字个数至少为 $(2/3)*M$ ，即块的最低使用率为 $2/3$ （代替B+树的 $1/2$ ）。给出了一个简单实例，如下图所示：



B+树的分裂：当一个结点满时，分配一个新的结点，并将原结点中 $1/2$ 的数据复制到新结点，最后在父结点中增加新结点的指针；B+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针。

B*树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制 $1/3$ 的数据到新结点，最后在父结点增加新结点的指针。

所以，B*树分配新结点的概率比B+树要低，空间使用率更高；

7.总结

通过以上介绍，大致将B树，B+树，B*树总结如下：

- B树：有序数组+平衡多叉树；
- B+树：有序数组链表+平衡多叉树；
- B*树：一棵丰满的B+树。

顺便说一句，无论是B树，还是B+树、b树，由于根或者树的上面几层被反复查询，所以这几块可以存在内存中，换言之，B树、B+树、B树的根结点和部分顶层数据在内存中，大部分下层数据在磁盘上。

最近公共祖先**LCA**问题

问题描述

求有根树的任意两个节点的最近公共祖先。

分析与解法

解答这个问题之前，咱们得先搞清楚到底什么是最近公共祖先。最近公共祖先简称LCA（Lowest Common Ancestor），所谓LCA，是当给定一个有根树T时，对于任意两个结点u、v，找到一个离根最远的结点x，使得x同时是u和v的祖先，x便是u、v的最近公共祖先。（参见：http://en.wikipedia.org/wiki/Lowest_common_ancestor）原问题涵盖一般性的有根树，本文为了简化，多使用二叉树来讨论。

举个例子，如针对下图所示的一棵普通的二叉树来讲：



结点3和结点4的最近公共祖先是结点2，即 $LCA(3, 4) = 2$ 。在此，需要注意到当两个结点在同一棵子树上的情况，如结点3和结点2的最近公共祖先为2，即 $LCA(3, 2) = 2$ 。同理： $LCA(5, 6) = 4$ ， $LCA(6, 10) = 1$ 。

明确了题意，咱们便来试着解决这个问题。直观的做法，可能是针对是否为二叉查找树分情况讨论，这也是一般人最先想到的思路。除此之外，还有所谓的Tarjan算法、倍增算法、以及转换为RMQ问题（求某段区间的极值）。后面这几种算法相对高级，不那么直观，但思路比较有启发性，了解一下也有裨益。

解法一：暴力对待

1.1、是二叉查找树

在当这棵树是二叉查找树的情况下，如下图：



那么从树根开始：

- 如果当前结点t大于结点u、v，说明u、v都在t的左侧，所以它们的共同祖先必定在t的左子树中，故从t的左子树中继续查找；
- 如果当前结点t小于结点u、v，说明u、v都在t的右侧，所以它们的

共同祖先必定在 t 的右子树中，故从 t 的右子树中继续查找；

- 如果当前结点 t 满足 $u < t < v$ ，说明 u 和 v 分居在 t 的两侧，故当前结点 t 即为最近公共祖先；
- 而如果 u 是 v 的祖先，那么返回 u 的父结点，同理，如果 v 是 u 的祖先，那么返回 v 的父结点。

代码如下所示：

```
//copyright@eriol 2011
//modified by July 2014

public int query (Node t, Node u, Node v) {
    int left = u.value;
    int right = v.value;

    //二叉查找树内，如果左结点大于右结点，不对，交换
    if (left > right) {
        int temp = left;
        left = right;
        right = temp;
    }

    while ( true ) {
        //如果t小于u、v，往t的右子树中查找
        if (t.value < left) {
            t = t.right;

            //如果t大于u、v，往t的左子树中查找
        } else if (t.value > right) {
            t = t.left;
        } else {
            return t.value;
        }
    }
}
```



```
}
```

1.2、不是二叉查找树

但如果这棵树不是二叉查找树，只是一棵普通的二叉树呢？如果每个结点都有一个指针指向它的父结点，于是我们可以从任何一个结点出发，得到一个到达树根结点的单向链表。因此这个问题转换为两个单向链表的第一个公共结点。

此外，如果给出根节点，LCA问题可以用递归很快解决。而关于树的问题一般都可以转换为递归（因为树本来就是递归描述），参考代码如下：

```
//copyright@allantop 2014-1-22-20:01
node* getLCA (node* root, node* node1, node* node2)
{
    if (root == null)
        return null;

    if (root== node1 || root==node2)
        return root;

    node* left = getLCA(root->left, node1, node2);
    node* right = getLCA(root->right, node1, node2);

    if (left != null && right != null)
        return root;

    else if (left != null)
        return left;

    else if (right != null)
        return right;

    else
        return null;
}
```

然不论是针对普通的二叉树，还是针对二叉查找树，上面的解法有一个很大的弊端就是：如需 N 次查询，则总体复杂度会扩大 N 倍，故这种暴力解法仅适合一次查询，不适合多次查询。

接下来的解法，将不再区别对待是否为二叉查找树，而是一致当做是一棵普通的二叉树。总体来说，由于可以把LCA问题看成是询问式的，即给出一系列询问，程序对每一个询问尽快做出反应。故处理这类问题一般有两种解决方法：

- 一种是在线算法，相当于循序渐进处理；
- 另外一种则是离线算法，如Tarjan算法，相当于一次性批量处理，一开始就知道了全部查询，只待询问。

解法二：Tarjan算法

如上文末节所述，不论咱们所面对的二叉树是二叉查找树，或不是二叉查找树，都可以把求任意两个结点的最近公共祖先，当做是查询的问题，如果是只求一次，则是单次查询；如果要求多个任意两个结点的最近公共祖先，则相当于是批量查询。

涉及到批量查询的时候，咱们可以借鉴离线处理的方式，这就引出了解决此LCA问题的Tarjan离线算法。

2.1、什么是Tarjan算法

Tarjan算法（以发现者Robert Tarjan命名）是一个在图中寻找强连通分量的算法。算法的基本思想为：任选一结点开始进行深度优先搜索dfs（若深度优先搜索结束后仍有未访问的结点，则再从中任选一点再次进行）。搜索过程中已访问的结点不再访问。搜索树的若干子树构成了图的强连通分量。

应用到咱们要解决的LCA问题上，则是：对于新搜索到的一个结点 u ，先创建由 u 构成的集合，再对 u 的每颗子树进行搜索，每搜索完一棵子树，这时候子树中所有的结点的最近公共祖先就是 u 了。

举一个例子，如下图（不同颜色的结点相当于不同的集合）：



假设遍历完10的孩子,要处理关于10的请求了,取根节点到当前正在遍历的节点的路径为关键路径,即1-3-8-10,集合的祖先便是关键路径上距离集合最近的点。

比如:

- 1, 2, 5, 6为一个集合,祖先为1, 集合中点和10的LCA为1
- 3, 7为一个集合, 祖先为3, 集合中点和10的LCA为3
- 8, 9, 11为一个集合, 祖先为8, 集合中点和10的LCA为8
- 10, 12为一个集合, 祖先为10, 集合中点和10的LCA为10

得出的结论便是: $LCA(u,v)$ 便是根至u的路径上到节点v最近的点。

2.2、Tarjan算法如何而来

但关键是 Tarjan算法是怎么想出来的呢? 再给定下图, 你是否能看出来: 分别从结点1的左右子树当中, 任取一个结点, 设为u、v, 这两个任意结点u、v的最近公共祖先都为1。



于此, 我们可以得知: 若两个结点u、v分别分布于某节点t的左右子树, 那么此节点t即为u和v的最近公共祖先。更进一步, 考虑到一个节点自己就是LCA的情况, 得知:

- 若某结点t是两结点u、v的祖先之一, 且这两结点并不分布于该结点t的一棵子树中, 而是分别在结点t的左子树、右子树中, 那么该结点t即为两结点u、v的最近公共祖先。

这个定理就是Tarjan算法的基础。

一如上文1.1节我们得到的结论: “如果当前结点t满足 $u < t < v$, 说明u和v分居在t的两侧, 故当前结点t即为最近公共祖先”。

而对于本节开头我们所说的“如果要求多个任意两个结点的最近公共祖先, 则相当于是批量查询”, 即在很多组的询问的情况下, 或许可以先确定一个LCA。例如是根节点1, 然后再去检查所有询问, 看是否满足

刚才的定理，不满足就忽视，满足就赋值，全部弄完，再去假设2号节点是LCA，再去访问一遍。

可此方法需要判断一个结点是在左子树、还是右子树，或是都不在，都只能遍历一棵树，而多次遍历的代价实在是太大了，所以我们需要找到更好的方法。这就引出了下面要阐述的Tarjan算法，即每个结点只遍历一次，怎么做到的呢，请看下文讲解。

2.3、Tarjan算法流程

Tarjan算法流程为：

```
Procedure dfs (u) ;  
begin  
    设置u号节点的祖先为u  
    若u的左子树不为空，dfs (u - 左子树) ;  
    若u的右子树不为空，dfs (u - 右子树) ;  
    访问每一条与u相关的询问u、v  
    - 若v已经被访问过，则输出v当前的祖先t (t即u,v的LCA)  
    标记u为已经访问，将所有u的孩子包括u本身的祖先改为u的父亲  
end
```

普通的dfs 不能直接解决LCA问题，故Tarjan算法的原理是dfs + 并查集，它每次把两个结点对的最近公共祖先的查询保存起来，然后dfs 更新一次。如此，利用并查集优越的时空复杂度，此算法的时间复杂度可以缩小至 $O(n+Q)$ ，其中， n 为数据规模， Q 为询问个数。

解法三：转换为RMQ问题

解决此最近公共祖先问题的还有一个算法，即转换为RMQ问题，用Sparse Table（简称ST）算法解决。

3.1、什么是RMQ问题

RMQ，全称为Range Minimum Query，顾名思义，则是区间最值查询，

它被用来在数组中查找两个指定索引中最小值的位置。即RMQ相当于给定数组A[0, N-1]，找出给定的两个索引如 i、j 间的最小值的位置。

假设一个算法预处理时间为 $f(n)$ ，查询时间为 $g(n)$ ，那么这个算法复杂度的标记为 λ 。我们将用RMQA(i, j) 来表示数组A 中索引i 和 j 之间最小值的位置。u和v的离树T根结点最远的公共祖先用LCA T(u, v)表示。

如下图所示，RMQA(2,7)则表示求数组A中从A[2]~A[7]这段区间中的最小值：



很显然，从上图中，我们可以看出最小值是A[3] = 1，所以也就不难得出最小值的索引值RMQA(2,7) = 3。

3.2、如何解决RMQ问题

3.2.1、Trivial algorithms for RMQ

下面，我们对每一对索引(i, j)，将数组中索引i 和 j 之间最小值的位置RMQA(i, j) 存储在M[0, N-1][0, N-1]表中。RMQA(i, j) 有不同种计算方法，你会看到，随着计算方法的不同，它的时空复杂度也不同：

- 普通的计算将得到一个 λ 复杂度的算法。尽管如此，通过使用一个简单的动态规划方法，我们可以将复杂度降低到 λ 。如何做到的呢？方法如下代码所示：

```
//copyright@  
  
//modified by July 2014  
  
void process1 ( int M[MAXN][MAXN], int A[MAXN], int N)  
{  
    int i, j;  
    for (i = 0 ; i < N; i++)  
        M[i][i] = i;  
  
    for (i = 0 ; i < N; i++)  
        for (j = i + 1 ; j < N; j++)
```

```

//若前者小于后者，则把后者的索引值付给M[i][j]
if (A[M[i][j - 1]] < A[j])
    M[i][j] = M[i][j - 1];
//否则前者的索引值付给M[i][j]
else
    M[i][j] = j;
}

```

- 一个比较有趣的点子是把向量分割成 \sqrt{N} 大小的段。我们将在 $M[0, \sqrt{N}-1]$ 为每一个段保存最小值的位置。如此，M可以很容易的在 $O(N)$ 时间内预处理。



- 一个更好的方法预处理RMQ 是对 2^k 的长度的子数组进行动态规划。我们将使用数组 $M[0, N-1][0, \log N]$ 进行保存，其中 $M[i][j]$ 是以 i 开始，长度为 2^j 的子数组的最小值的索引。这就引出了咱们接下来要介绍的Sparse Table (ST) algorithm。

3.2.2、Sparse Table (ST) algorithm



在上图中，我们可以看出：

- 在 $A[1]$ 这个长度为 2^0 的区间内，最小值即为 $A[1] = 4$ ，故最小值的索引 $M[1][0]$ 为1；
- 在 $A[1]$ 、 $A[2]$ 这个长度为 2^1 的区间内，最小值为 $A[2] = 3$ ，故最小值的索引为 $M[1][1] = 2$ ；
- 在 $A[1]$ 、 $A[2]$ 、 $A[3]$ 、 $A[4]$ 这个长度为 2^2 的区间内，最小值为 $A[3] = 1$ ，故最小值的索引 $M[1][2] = 3$ 。

为了计算 $M[i][j]$ 我们必须找到前半段区间和后半段区间的最小值。很明显的片段有着 $2^{(j-1)}$ 长度，因此递归如下



根据上述公式，可以写出这个预处理的递归代码，如下：

```
void process2 ( int M[MAXN][LOGMAXN], int A[MAXN], int N)
{
    int i, j;

    //initialize M for the intervals with length 1

    for (i = 0 ; i < N; i++)
        M[i][ 0 ] = i;

    //compute values from smaller to bigger intervals
    for (j = 1 ; 1 << j <= N; j++)
        for (i = 0 ; i + ( 1 << j) - 1 < N; i++)
            if (A[M[i][j - 1 ]] < A[M[i + ( 1 << (j - 1 ))][j - 1 ]])
                M[i][j] = M[i][j - 1 ];
            else
                M[i][j] = M[i + ( 1 << (j - 1 ))][j - 1 ];
}
```

经过这个 $O(N \log N)$ 时间复杂度的预处理之后，让我们看看怎样使用它们去计算 $RMQA(i, j)$ 。思路是选择两个能够完全覆盖区间 $[i..j]$ 的块并且找到它们之间的最小值。设 $k = \lceil \log(j - i + 1) \rceil$ 。

为了计算 $RMQA(i, j)$ ，我们可以使用下面的公式：



故，综合来看，咱们预处理的时间复杂度从 $O(N^3)$ 降低到了 $O(N \log N)$ ，查询的时间复杂度为 $O(1)$ ，所以最终的整体复杂度为： \backslash 。

3.3、LCA与RMQ的关联性

现在，让我们看看怎样用RMQ来计算LCA查询。事实上，我们可以在线性时间里将LCA问题规约到RMQ问题，因此每一个解决RMQ的问题

都可以解决LCA问题。让我们通过例子来说明怎么规约的：



注意 $LCAT(u, v)$ 是在对 T 进行dfs过程当中在访问 u 和 v 之间离根结点最近的点。因此我们可以考虑树的欧拉环游过程 u 和 v 之间所有的结点，并找到它们之间处于最低层的结点。为了达到这个目的，我们可以建立三个数组：

- $E[1, 2*N-1]$ - 对 T 进行欧拉环游过程中所有访问到的结点; $E[i]$ 是在环游过程中第 i 个访问的结点
- $L[1, 2*N-1]$ - 欧拉环游中访问到的结点所处的层数; $L[i]$ 是 $E[i]$ 所在的层数
- $H[1, N]$ - $H[i]$ 是 E 中结点 i 第一次出现的下标(任何出现 i 的地方都行，当然选第一个不会错)

假定 $H[u] < H[v]$ 。可以很容易的看到 u 和 v 第一次出现的结点是 $E[H[u]..H[v]]$ 。现在，我们需要找到这些结点中的最低层。为了达到这个目的，我们可以使用RMQ。因此 $LCAT(u, v) = E[RMQL(H[u], H[v])]$,RMQ返回的是索引，下面是 E, L, H 数组：



注意 L 中连续的元素相差为1。

3.4、从RMQ到LCA

我们已经看到了LCA问题可以在线性时间规约到RMQ问题。现在让我们来看看怎样把RMQ问题规约到LCA。这个意味着我们实际上可以把一般的RMQ问题规约到带约束的RMQ问题(这里相邻的元素相差1)。为了达到这个目的，我们需要使用笛卡尔树。

对于数组 $A[0, N-1]$ 的笛卡尔树 $C(A)$ 是一个二叉树，根节点是 A 的最小元素，假设 i 为 A 数组中最小元素的位置。当 $i > 0$ 时，这个笛卡尔树的左子结点是 $A[0, i-1]$ 构成的笛卡尔树，其他情况没有左子结点。右结点类似的用 $A[i+1, N-1]$ 定义。注意对于具有相同元素的数组 A ，笛卡尔树并不

唯一。在本文中，将会使用第一次出现的最小值，因此笛卡尔树看作唯一。可以很容易的看到 $RMQA(i, j) = LCAC(i, j)$ 。

下面是一个例子：



现在我们需要做的仅仅是用线性时间计算 $C(A)$ 。这个可以使用栈来实现。

- 初始栈为空。
- 然后我们在栈中插入 A 的元素。
- 在第 i 步, $A[i]$ 将会紧挨着栈中比 $A[i]$ 小或者相等的元素插入,并且所有较大的元素将会被移除。
- 在插入结束之前栈中 $A[i]$ 位置前的元素将成为 i 的左儿子, $A[i]$ 将会成为它之后一个较小元素的右儿子。

在每一步中，栈中的第一个元素总是笛卡尔树的根。

如果使用栈来保存元素的索引而不是值，我们可以很轻松的建立树。由于 A 中的每个元素最多被增加一次和最多被移除一次，所以建树的时间复杂度为 $O(N)$ 。最终查询的时间复杂度为 $O(1)$ ，故综上可得，咱们整个问题的最终时间复杂度为： \backslash 。

现在，对于询问 $RMQA(i, j)$ 我们有两种情况：

- i 和 j 在同一个块中,因此我们使用在 P 和 T 中计算的值
- i 和 j 在不同的块中，因此我们计算三个值:从 i 到 i 所在块的末尾的 P 和 T 中的最小值，所有 i 和 j 中块中的通过与处理得到的最小值以及从 j 所在块 i 和 j 在同一个块中,因此我们使用在 P 和 T 中计算的值 j 的 P 和 T 的最小值；最后我们只要计算三个值中最小值的位置即可。

RMQ 和 LCA 是密切相关的问题，因为它们之间可以相互规约。有许多算法可以用来解决它们，并且他们适应于一类问题。

解法四：线段树

解决RMQ问题也可以用所谓的线段树Segment trees。线段树是一个类似堆的数据结构，可以在基于区间数组上用对数时间进行更新和查询操作。我们用下面递归方式来定义线段树的 $[i, j]$ 区间：

- 第一个结点将保存区间 $[i, j]$ 区间的信息
- 如果 $i < j$ 左右的孩子结点将保存区间 $[i, (i+j)/2]$ 和 $[(i+j)/2+1, j]$ 的信息

注意具有N个区间元素的线段树的高度为 $\lceil \log N \rceil + 1$ 。下面是区间 $[0,9]$ 的线段树：



线段树和堆具有相同的结构，因此我们定义 x 是一个非叶结点，那么左孩子结点为 $2x$ ，而右孩子结点为 $2x+1$ 。想要使用线段树解决RMQ问题，我们则要用数组 $M[1, 2 * 2^{\lceil \log N \rceil + 1}]$ ，这里 $M[i]$ 保存结点 i 区间最小值的位置。初始时 M 的所有元素为-1。树应当用下面的函数进行初始化(b 和 e 是当前区间的范围)：

```
void initialize ( int node, int b, int e, int M[MAXIND], int A[MAXN],
int N)
{
    if (b == e)
        M[node] = b;
    else
    {
        //compute the values in the left and right subtrees
        initialize( 2 * node, b, (b + e) / 2 , M, A, N);
        initialize( 2 * node + 1 , (b + e) / 2 + 1 , e, M, A, N);

        //search for the minimum value in the first and
        //second half of the interval
        if (A[M[ 2 * node]] <= A[M[ 2 * node + 1 ]])
            M[node] = M[ 2 * node];
        else
            M[node] = M[ 2 * node + 1 ];
    }
}
```

```

    }
}

```

上面的函数映射出了这棵树建造的方式。当计算一些区间的最小值位置时，我们应当首先查看子结点的值。调用函数的时候使用 `node = 1, b = 0`和`e = N-1`。

现在我们可以开始进行查询了。如果我们想要查找区间`[i, j]`中的最小值的位置时，我们可以使用下一个简单的函数：

```

int query ( int node, int b, int e, int M[MAXIND], int A[MAXN], int
i, int j)
{
    int p1, p2;

    //if the current interval doesn't intersect
    //the query interval return -1
    if (i > e || j < b)
        return -1 ;

    //if the current interval is included in
    //the query interval return M[node]
    if (b >= i && e <= j)
        return M[node];

    //compute the minimum position in the
    //left and right part of the interval
    p1 = query( 2 * node, b, (b + e) / 2 , M, A, i, j);
    p2 = query( 2 * node + 1 , (b + e) / 2 + 1 , e, M, A, i, j);

    //return the position where the overall
    //minimum is
    if (p1 == -1 )
        return M[node] = p2;
}

```

```

    if (p2 == - 1 )
        return M[node] = p1;

    if (A[p1] <= A[p2])
        return M[node] = p1;

    return M[node] = p2;
}

```

你应该使用`node = 1`, `b = 0`和`e = N - 1`来调用这个函数，因为分配给第一个结点的区间是`[0, N-1]`。

可以很容易的看出任何查询都可以在 $O(\log N)$ 内完成。注意当我们碰到完整的in/out区间时我们停止了，因此数中的路径最多分裂一次。用线段树我们获得了\的算法

线段树非常强大，不仅仅是因为它能够用在RMQ上，还因为它是一个非常灵活的数据结构，它能够解决动态版本的RMQ问题和大量的区间搜索问题。

其余解法

除此之外，还有倍增法、重链剖分算法和后序遍历也可以解决该问题。其中，倍增思路相当于层序遍历，逐层或几层跳跃查，查询时间复杂度为 $O(\log n)$ ，空间复杂度为 $n \log n$ ，对于每个节点先存储向上1层2层4层的节点，每个点有depth信息。

本章堆栈树图相关的习题

1、附近地点搜索

找一个点集中与给定点距离最近的点，同时，给定的二维点集都是固定的，查询可能有很多次，例如，坐标(39.91, 116.37)附近500米内有什么餐馆，那么让你来设计，该怎么做？



提示：可以建立R树进行二维搜索，或使用GeoHash算法解决。

2、最小操作数

给定一个单词集合Dict，其中每个单词的长度都相同。现从此单词集合Dict中抽取两个单词A、B，我们希望通过若干次操作把单词A变成单词B，每次操作可以改变单词的一个字母，同时，新产生的单词必须是在给定的单词集合Dict中。求所有行得通步数最少的修改方法。

举个例子如下：

Given: A = "hit" B = "cog" Dict = ["hot","dot","dog","lot","log"] Return [["hit","hot","dot","dog","cog"], ["hit","hot","lot","log","cog"]]

即把字符串A = "hit"转变成字符串B = "cog"，有以下两种可能：

"hit" -> "hot" -> "dot" -> "dog" -> "cog";

"hit" -> "hot" -> "lot" -> "log" -> "cog".

提示：建图然后搜索。

3、最少操作次数的简易版

给定两个字符串，仅由小写字母组成，它们包含了相同字符。求把第一个字符串变成第二个字符串的最小操作次数，且每次操作只能对第一个字符串中的某个字符移动到此字符串中的开头。例如给定两个字符

串“abcd" "bcad"，输出：2，因为需要操作2次才能把"abcd"变成“bcad”，方法是：abcd->cabd->bcad。

3、把二元查找树转变成排序的双向链表

输入一棵二元查找树，将该二元查找树转换成一个排序的双向链表。要求不能创建任何新的结点，只调整指针的指向。例如把下述二叉查找树

10

//

6 14

////

4 8 12

转换成双向链表，即得：

4=6=8=10=12=14=16。

4、在二元树中找出和为某一值的所有路径

输入一个整数和一棵二元树。从树的根结点开始往下访问一直到叶结点所经过的所有结点形成一条路径。打印出和与输入整数相等的所有路径。

5、判断整数序列是不是二元查找树的后序遍历结果

输入一个整数数组，判断该数组是不是某二元查找树的后序遍历的结果，如果是返回true，否则返回false。

例如输入5、7、6、9、11、10、8，由于这一整数序列是如下树的后序遍历结果：

```
      8
     /  \
    6    10
```

// //

5 7 9 11 因此返回true。

如果输入7、4、6、5，没有哪棵树的后序遍历的结果是这个序列，因此返回false。

6、设计包含min函数的栈

定义栈的数据结构，要求添加一个min函数，能够得到栈的最小元素。要求函数min、push以及pop的时间复杂度都是O(1)。

7、求二叉树中节点的最大距离

如果我们把二叉树看成一个图，父子节点之间的连线看成是双向的，我们姑且定义"距离"为两节点之间边的个数。

请写一个程序，求一棵二叉树中相距最远的两个节点之间的距离。

8

输入一颗二元树，从上往下按层打印树的每个结点，同一层中按照从左往右的顺序打印。

例如输入

8

//

6 10

////

5 7 9 11

输出8 6 10 5 7 9 11。

9

请用递归和非递归两种方法实现二叉树的前序遍历。

10、求树的深度

输入一棵二元树的根结点，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

例如：输入二元树： 10



输出该树的深度3。

实现简单的一个查找二叉树的深度的函数。

11、用俩个栈实现队列

某队列的声明如下：

```
template < typename T> class CQueue
{
public :
    CQueue() {}
    ~CQueue() {}
    void appendTail ( const T& node) ; // append a element to tail
    void deleteHead () ; // remove a element from head
private :
```



```
T> m_stack1;  
  
T> m_stack2;  
  
};
```

提示：这道题实质上是要求我们用两个栈来实现一个队列。栈是一种后入先出的数据容器，因此对队列进行的插入和删除操作都是在栈顶上进行；队列是一种先入先出的数据容器，我们总是把新元素插入到队列的尾部，而从队列的头部删除元素。

12

假设有一颗二叉树，已知这棵树的节点上不均匀的分布了若干石头，石头数跟这棵二叉树的节点数相同，石头只可以在子节点和父节点之间进行搬运，每次只能搬运一颗石头。请问如何以最少的步骤将石头搬运均匀，使得每个节点上的石头上刚好为1。

13

对于一颗完全二叉树，要求给所有节点加上一个pNext指针，指向同一层的相邻节点；如果当前节点已经是该层的最后一个节点，则将pNext指针指向NULL；给出程序实现，并分析时间复杂度和空间复杂度。

14

两个用户之间可能互相认识，也可能是单向的认识，用什么数据结构来表示？如果一个用户不认识别人，而且别人也不认识他，那么他就是无效节点，如何找出这些无效节点？自定义数据接口并实现之，要求尽可能节约内存和空间复杂度。

15

有一个一亿节点的树，现在已知两个点，找这两个点的共同的祖先。

16

给一个二叉树，每个节点都是正或负整数，如何找到一个子树，它所有节点的和最大？

提示：后序遍历，每一个节点保存左右子树的和加上自己的值。额外一个空间存放最大值。

写完后序遍历，面试官可能接着与你讨论，

- a). 如果要求找出只含正数的最大子树，程序该如何修改来实现？
- b). 假设我们将子树定义为它和它的部分后代，那该如何解决？
- c). 对于b，加上正数的限制，方案又该如何？

总之，一道看似简单的面试题，可能能变换成各种花样。

比如，面试官可能还会再提两个要求：第一，不能用全局变量；第二，有个参数控制是否要只含正数的子树。

17

有一个排序二叉树，数据类型是int型，如何找出中间大的元素。

18

中序遍历二叉树，结果为ABCDEFGH，后序遍历结果为ABEDCHGF，那么前序遍历结果为？

19

写程序输出8皇后问题的所有排列，要求使用非递归的深度优先遍历。

20

在8X8的棋盘上分布着n个骑士，他们想约在某一个格中聚会。骑士每天可以像国际象棋中的马那样移动一次，可以从中间像8个方向移动（当然不能走出棋盘），请计算n个骑士的最早聚会地点和要走多少天。要求尽早聚会，且n个人走的总步数最少，先到聚会地点的骑士可以不再移动等待其他的骑士。

从键盘输入n（ $0 < n \leq 64$ ），然后一次输入n个骑士的初始位置xi,yi（ $0 \leq x_i, y_i \leq 7$ ）。屏幕输出以空格分隔的三个数，分别为聚会点（x，y）以及走的天数。

提示：BFS。

21、城市遍历

某人家住北京，想去青海玩，可能会经过许多城市，现已知地图上的城市连接，求经过M个城市到达青海的路线种类。城市可以多次到达的，比如去了天津又回到北京，再去天津，即为3次。北京出发不算1次。

输入：

N M S N为城市总数，北京为0，青海为N-1； M为经过的城市数目； S为之后有S行 $i\ j$ 表示第i个城市可以去第j个城市，是有方向的。

输出： N 表示路径种类。

22

给定两个站点，如果没有直达的路线，如何找到换乘次数最少的路线？

23

有两座桥，其中一座可能是坏的，两个守桥人分别守在这两座桥的入口。他们一个总是会说实话，一个总是说谎话。你现在需要找出哪一座桥可以通过。

请问最少需要问守桥人几个问题，可以找出可以通过的桥？如何问？

24

一类似于蜂窝的结构的图，进行搜索最短路径（要求5分钟）。

25

对于一颗完全二叉树，要求给所有节点加上一个pNext指针，指向同一层的相邻节点；如果当前节点已经是该层的最后一个节点，则将pNext指针指向NULL；给出程序实现，并分析时间复杂度和空间复杂度。

第二部分 算法心得

第四章 查找匹配

有序数组的查找

题目描述

给定一个有序的数组，查找某个数是否在数组中，请编程实现。

分析与解法

一看到数组本身已经有序，我想你可能反应出了要用二分查找，毕竟二分查找的适用条件就是有序的。那什么是二分查找呢？

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。其算法流程如下：

- 一开始，范围覆盖整个数组。
- 将数组的中间项与T进行比较，如果T比数组的中间项要小，则到数组的前半部分继续查找，反之，则到数组的后半部分继续查找。
- 如此，每次查找可以排除一半元素，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。

此时，可能有不少读者心里嘀咕，不就二分查找么，太简单了。

然《编程珠玑》的作者Jon Bentley曾在贝尔实验室做过一个实验，即给一些专业的程序员几个小时的时间，用任何一种语言编写二分查找程序（写出高级伪代码也可以），结果参与编写的一百多人中：90%的程序写的程序中有bug（我并不认为没有bug的代码就正确）。

也就是说：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：而且高德纳在《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。

你能正确无误的写出二分查找代码么？不妨一试，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。

要准确实现二分查找，首先要把握下面几个要点：

- 关于right的赋值
 - `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。

以下是一份参考实现：

```
int BinarySearch ( 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 ;  
}
```


总结

编写二分查找的程序时

- 如果令 `left <= right`, 则 `right = middle - 1`;
- 如果令 `left < right`, 则 `right = middle`;

换言之, 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。且在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。

行列递增矩阵的查找

题目描述

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。



分析与解法

解法一、分治法

这种行和列分别递增的矩阵，有一个专有名词叫做杨氏矩阵，由剑桥大学数学家杨表在1900年推提出，在这个矩阵中的查找，俗称杨氏矩阵查找。

以查找数字6为例，因为矩阵的行和列都是递增的，所以整个矩阵的对角线上的数字也是递增的，故我们可以在对角线上进行二分查找，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，而在左下和右上的两个矩形继续递归查找，如下图所示：



解法二、定位法

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，这个方法的时间复杂度 $O(m+n)$ 。如下图所示：



关键代码如下所示：

```
# define ROW 4
# define COL 4

bool YoungMatrix ( int array[][COL], int searchKey) {
    int i = 0 , j = COL - 1 ;
    int var = array [i][j];
    while ( true ){
        if (var == searchKey)
            return true ;
    }
}
```

```
        else if (var < searchKey && i < ROW - 1 )  
            var = array [++i][j];  
        else if (var > searchKey && j > 0 )  
            var = array [i][--j];  
        else  
            return false ;  
    }  
}
```

举一反三

1、给定 $n \times n$ 的实数矩阵，每行和每列都是递增的，求这 n^2 个数的中位数。

2、我们已经知道杨氏矩阵的每行的元素从左到右单调递增，每列的元素从上到下也单调递增的矩阵。那么，如果给定从1-n这n个数，我们可以构成多少个杨氏矩阵呢？

例如 $n = 4$ 的时候，我们可以构成1行4列的矩阵：

1 2 3 4

2个2行2列的矩阵：

1 2

3 4

和

1 3

2 4

还有一个4行1列的矩阵

1

2

3

4

因此输出4。

出现次数超过一半的数字

题目描述

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。

分析与解法

一个数组中有很多数，现在我们要找出其中那个出现次数超过总数一半的数字，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。

解法一

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（至于排序方法可选取最常用的快速排序）。排完序后，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 $O(n\log n + n)$ 。

但如果是有序的数组呢，或者经过排序把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，还需要再遍历一次整个数组？

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 $O(n*\log n)$ 。

然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。

解法二

既要缩小总的时间复杂度，那么可以用查找时间复杂度为 $O(1)$ 的 **hash** 表，即以空间换时间。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。

解法三

Hash表需要 $O(n)$ 的空间开销，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，免去了排序，也避免了空间 $O(n)$ 的开销，总得说来，时间复杂度只有 $O(n)$ ，空间复杂度为 $O(1)$ ，貌似不失为最佳方法。

举个简单的例子，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。

此外，对于序列 $\{5, 5, 5, 5, 1\}$ ，每次分别从数组两端尝试各删除一个数（左边删除5，右边删除1，两个数不相同），之后剩余 $\{5, 5, 5\}$ ，这时无法找到两个不同的数进行删除，说明剩余元素全部相同，返回5作为结果即可。

解法四

更进一步，考虑到这个问题本身的特殊性，我们可以在遍历数组的时候保存两个值：一个candidate，用来保存数组中遍历到的某个数字；一个nTimes，表示当前数字的出现次数，其中，nTimes初始化为1。当我们遍历到数组中下一个数字的时候：

- 如果下一个数字与之前candidate保存的数字相同，则nTimes加1；
- 如果下一个数字与之前candidate保存的数字不同，则nTimes减1；
- 每当出现次数nTimes变为0后，用candidate保存下一个数字，并把

nTimes重新设为1。直到遍历完数组中的所有数字为止。

举个例子，假定数组为{0, 1, 2, 1, 1}，按照上述思路执行的步骤如下：

- 1.开始时，candidate保存数字0，nTimes初始化为1；
- 2.然后遍历到数字1，与数字0不同，则nTimes减1变为0；
- 3.因为nTimes变为了0，故candidate保存下一个遍历到的数字2，且nTimes被重新设为1；
- 4.继续遍历到第4个数字1，与之前candidate保存的数字2不同，故nTimes减1变为0；
- 5.因nTimes再次被变为了0，故我们让candidate保存下一个遍历到的数字1，且nTimes被重新设为1。最后返回的就是最后一次把nTimes设为1的数字1。

思路清楚了，完整的代码如下：

```
//a代表数组，length代表数组长度

int FindOneNumber ( int * a, int length)
{
    int candidate = a[ 0 ];
    int nTimes = 1 ;
    for ( int i = 1 ; i < length; i++)
    {
        if (nTimes == 0 )
        {
            candidate = a[i];
            nTimes = 1 ;
        }
        else
        {
            if (candidate == a[i])
                nTimes++;
            else
                nTimes--;
        }
    }
}
```

```
    }  
}  
  
return candidate;  
}
```

即针对数组{0, 1, 2, 1, 1}, 套用上述程序可得:

```
i=0, candidate=0, nTimes=1;  
i=1, a[1] != candidate, nTimes--, =0;  
i=2, candidate=2, nTimes=1;  
i=3, a[3] != candidate, nTimes--, =0;  
i=4, candidate=1, nTimes=1;  
如果是0, 1, 2, 1, 1, 1的话, 那么i=5, a[5] == candidate, nTimes++, =2; .....
```

举一反三

加强版水王：找出出现次数刚好是一半的数字

分析：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。

第五章 动态规划

本章导读

学习一个算法，可分为3个步骤：首先了解算法本身解决什么问题，然后学习它的解决策略，最后了解某些相似算法之间的联系。例如图算法中，

- 广搜是一层一层往外遍历，寻找最短路径，其策略是采取队列的方法。
- 最小生成树是最小代价连接所有点，其策略是贪心，比如Prim的策略是贪心+权重队列。
- Dijkstra是寻找单源最短路径，其策略是贪心+非负权重队列。
- Floyd是多结点对的最短路径，其策略是动态规划。

而贪心和动态规划是有联系的，贪心是“最优子结构+局部最优”，动态规划是“最优独立重叠子结构+全局最优”。一句话理解动态规划，则是枚举所有状态，然后剪枝，寻找最优状态，同时将每一次求解子问题的结果保存在一张“表格”中，以后再遇到重叠的子问题，从表格中保存的状态中查找（俗称记忆化搜索）。

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。

分析与解法

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）是：

- 子串（Substring）是串的一个连续的部分，
- 子序列（Subsequence）则是从不变改变序列的顺序，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。

解法一

或许，读者初看此题，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  maxProductSubstring ( double  *a,  int  length)
{
    double  maxResult = a[ 0 ];
    for  ( int  i = 0 ; i < length; i++)
    {
        double  x = 1 ;
```

```

        for ( int j = i; j < length; j++)
        {
            x *= a[j];
            if (x > maxResult)
            {
                maxResult = x;
            }
        }
    }

    return maxResult;
}

```

但这种蛮力的方法的时间复杂度为 $O(n^2)$ ，能否想办法降低时间复杂度呢？

解法二

考虑到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。

假设数组为 $a[]$ ，直接利用动态规划来求解，考虑到可能存在负数的情况，我们用 maxend 来表示以 $a[i]$ 结尾的最大连续子串的乘积值，用 minend 表示以 $a[i]$ 结尾的最小的子串的乘积值，那么状态转移方程为：

```

maxend = max(max(maxend * a[i], minend * a[i]), a[i]);
minend = min(min(maxend * a[i], minend * a[i]), a[i]);

```

初始状态为 $\text{maxend} = \text{minend} = a[0]$ 。

参考代码如下：

```

double MaxProductSubstring ( double *a, int length)

```

```

{
    double maxEnd = a[ 0 ];
    double minEnd = a[ 0 ];
    double maxResult = a[ 0 ];
    for ( int i = 1 ; i < length; ++i)
    {
        double end1 = maxEnd * a[i], end2 = minEnd * a[i];
        maxEnd = max(max(end1, end2), a[i]);
        minEnd = min(min(end1, end2), a[i]);
        maxResult = max(maxResult, maxEnd);
    }
    return maxResult;
}

```

动态规划求解的方法一个for循环搞定，所以时间复杂度为 $O(n)$ 。

举一反三

1、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。

分析：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。

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。

分析与解法

此题常见的思路是动态规划，假如令 $dp[i][j]$ 表示源串 $S[0...i]$ 和目标串 $T[0...j]$ 的最短编辑距离，其边界： $dp[0][j] = j$ ， $dp[i][0] = i$ ，那么我们可以得出状态转移方程：

- $dp[i][j] = \min\{$
 - $dp[i-1][j] + 1$, $S[i]$ 不在 $T[0...j]$ 中
 - $dp[i-1][j-1] + 1/0$, $S[i]$ 在 $T[j]$
 - $dp[i][j-1] + 1$, $S[i]$ 在 $T[0...j-1]$ 中
- }

接下来，咱们重点解释下上述3个式子的含义

- 关于 $dp[i-1][j] + 1$, s.t. $s[i]$ 不在 $T[0...j]$ 中的说明
 - $s[i]$ 没有落在 $T[0...j]$ 中，即 $s[i]$ 在中间的某一次编辑操作被删除了。因为删除操作没有前后相关性，不妨将其在第1次操作中删除。除首次操作时删除外，后续编辑操作是将长度为 $i-1$ 的字符串，编辑成长度为 j 的字符串：即 $dp[i-1][j]$ 。
 - 因此： $dp[i][j] = dp[i-1][j] + 1$ 。
- 关于 $dp[i-1][j-1] + 0/1$, s.t. $s[i]$ 在 $T[j]$ 的说明

- 若 $s[i]$ 经过编辑，最终落在 $T[j]$ 的位置。
- 则要么 $s[i] == t[j]$ ， $s[i]$ 直接落在 $T[j]$ 。这种情况，编辑操作实际上是将长度为 $i-1$ 的 S' 串，编辑成长度为 $j-1$ 的 T' 串：即 $dp[i-1][j-1]$ ；
- 要么 $s[i] \neq t[j]$ ， $s[i]$ 落在 $T[j]$ 后，要将 $s[i]$ 修改成 $T[j]$ ，即在上一种情况的基础上，增加一次修改操作：即 $dp[i-1][j-1] + 1$ 。
- 关于 $dp[i][j-1] + 1$, s.t. $s[i]$ 在 $T[0..j-1]$ 中的说明
 - 若 $s[i]$ 落在了 $T[1..j-1]$ 的某个位置，不妨认为是 k ，因为最小编辑步数的定义，那么，在 $k+1$ 到 $j-1$ 的字符，必然是通过插入新字符完成的。因为共插入了 $(j-k)$ 个字符，故编辑次数为 $(j-k)$ 次。而字符串 $S[1..i]$ 经过编辑，得到了 $T[1..k]$ ，编辑次数为 $dp[i][k]$ 。故： $dp[i][j] = dp[i][k] + (j-k)$ 。
 - 由于最后的 $(j-k)$ 次是插入操作，可以讲 $(j-k)$ 逐次规约到 $dp[i][k]$ 中。即： $dp[i][k] + (j-k) = dp[i][k+1] + (j-k-1)$ 规约到插入操作为1次，得到 $dp[i][k] + (j-k) = dp[i][k+1] + (j-k-1) = dp[i][k+2] + (j-k-2) = \dots = dp[i][k+(j-k-1)] + (j-k) - (j-k-1) = dp[i][j-1] + 1$ 。

上述的解释清晰规范，但为啥这样做呢？

换一个角度，其实就是字符串对齐的思路。例如把字符串“ALGORITHM”，变成“ALTRUISTIC”，那么把相关字符各自对齐后，如下图所示：

A	L	G	O	R		I		T	H	M
A	L		T	R	U	I	S	T	I	C

把图中上面的源串 $S[0..i] = \text{“ALGORITHM”}$ 编辑成下面的目标串 $T[0..j] = \text{“ALTRUISTIC”}$ ，我们枚举字符串 S 和 T 最后一个字符 $s[i]$ 、 $t[j]$ 对应四种情况：（字符-空白）（空白-字符）（字符-字符）（空白-空白）。

由于其中的（空白-空白）是多余的编辑操作。所以，事实上只存在以下3种情况：

- 下面的目标串空白，即 $S + \text{字符}X$ ， $T + \text{空白}$ ， S 变成 T ，意味着源串要删字符

- $dp[i - 1, j] + 1$
- 上面的源串空白，S + 空白，T + 字符，S变成T，最后，在S的最后插入“字符”，意味着源串要添加字符
 - $dp[i, j - 1] + 1$
- 上面源串中的的字符跟下面目标串中的字符不一样，即S + 字符X，T + 字符Y，S变成T，意味着源串要修改字符
 - $dp[i - 1, j - 1] + (s[i] == t[j] ? 0 : 1)$

综上，可以写出简单的DP状态方程：

//dp[i,j]表示表示源串S[0...i] 和目标串T[0...j] 的最短编辑距离

$dp[i, j] = \min \{ dp[i - 1, j] + 1, dp[i, j - 1] + 1, dp[i - 1, j - 1] + (s[i] == t[j] ? 0 : 1) \}$

//分别表示：删除1个，添加1个，替换1个（相同就不用替换）。

参考代码如下：

//dp[i][j]表示源串source[0-i)和目标串target[0-j)的编辑距离

```
int EditDistance ( char *pSource, char *pTarget)
{
    int srcLength = strlen (pSource);
    int targetLength = strlen (pTarget);
    int i, j;
    //边界dp[i][0] = i, dp[0][j] = j
    for (i = 1 ; i <= srcLength; ++i)
    {
        dp[i][ 0 ] = i;
    }
    for (j = 1 ; j <= targetLength; ++j)
    {
        dp[ 0 ][j] = j;
    }
    for (i = 1 ; i <= srcLength; ++i)
    {
```

```
        for (j = 1 ; j <= targetLength; ++j)
        {
            if (pSource[i - 1 ] == pTarget[j - 1 ])
            {
                dp[i][j] = dp[i - 1 ][j - 1 ];
            }
            else
            {
                dp[i][j] = 1 + min(dp[i - 1 ][j], dp[i][j - 1 ]);
            }
        }
    }

    return dp[srcLength][targetLength];
}
```

举一反三

1、传统的编辑距离里面有三种操作，即增、删、改，我们现在要讨论的编辑距离只允许两种操作，即增加一个字符、删除一个字符。我们求两个字符串的这种编辑距离，即把一个字符串变成另外一个字符串的最少操作次数。假定每个字符串长度不超过1000，只有大写英文字母组成。

2、有一亿个数，输入一个数，找出与它编辑距离在3以内的数，比如输入6（0110），找出0010等数，数是32位的。

问题扩展

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”：☐

当然，面试官还可以继续问下去，如请问，如何设计一个比较这篇文章和上一篇文章相似性的算法？

格子取数问题

题目描述

有 $n*n$ 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。



分析与解法

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：



上图中，图一是原始图，那么我们有以下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。

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。

为了方便讨论，我们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 ($2*5-2$)。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 $n*n$ 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。

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 $s = 2n - 2, 0 \leq i \leq j \leq n$ ，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 $DP[5,1,2]$ ， $DP[5,1,3]$ ， $DP[5,2,2]$ ， $DP[5,2,3]$ 。

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 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5

2 3 4 5 6 2 3 4 5 6 2 3 4 5 6 2 3 4 5 6

3 4 5 6 7 3 4 5 6 7 3 4 5 6 7 3 4 5 6 7

4 5 6 7 8 4 5 6 7 8 4 5 6 7 8 4 5 6 7 8

因此：

$DP[6,2,3] = \text{Max}(DP[5,1,2], DP[5,1,3], DP[5,2,2], DP[5,2,3]) + 6,2 \text{和} 6,3 \text{格子中对应的数值}$
(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)

故，综合上述的（式一），（式二）最后的递推式就是

$\text{if}(i \neq j) DP[s, i, j] = \text{Max}(DP[s - 1, i - 1, j - 1], DP[s - 1, i - 1, j], DP[s - 1, i, j - 1], DP[s - 1, i, j]) + W[s, i] + W[s, j] \text{ else } DP[s, i, j] = \text{Max}(DP[s - 1, i - 1, j - 1], DP[s - 1, i - 1, j], DP[s - 1, i, j]) + W[s, i]$ 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。

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 MinPathSum ( 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 $O(n^3)$ 。

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，感兴趣的读者可以自己写代码实现下。

举一反三

1、给定 $m \times n$ 的矩阵，每个位置是一个非负整数，从左上角开始，每次只能朝右和下走，走到右下角，但只走一次，求总和最小的路径。

提示：因为只走一次，所以相对来说比较简单， $dp[0, 0] = a[0, 0]$ ，且 $dp[x, y] = \min(dp[x-1, y] + a[x, y], dp[x, y-1] + a[x, y])$ 。

2、给定 $m \times n$ 的矩阵，每个位置是一个整数，从左上角开始，每次只能朝右、上和下走，并且不允许两次进入同一个格子，走到右上角，最小和。

分析：@cpcs：我们按列dp，假设前一系列的最优值已经算好了，一旦往右就回不去了。枚举我们对固定的 $(y-1)$ 列，我们已经算好了最优值，我们枚举行 x ，朝右走到 (x, y) ，然后再从 (x, y) 朝上走到 $(x, 0)$ ，再从 (x, y) 朝下走到 $(x, n-1)$ ，所有这些第 y 列的值，作为第 y 列的候选值，取最优。实际上，我们枚举了进入第 y 列的位置和在最终停在第 y 列的位置。这样保证我们不重复经过一个格子，也能保证我们不会往“左”走。

交替字符串

题目描述

输入三个字符串s1、s2和s3，判断第三个字符串s3是否由前两个字符串s1和s2交错而成，即不改变s1和s2中各个字符原有的相对顺序，例如当s1 = “aabcc”，s2 = “dbbca”，s3 = “aadbcbcbac”时，则输出true，但如果s3=“accabdbbca”，则输出false。

分析与解法

此题不能简单的排序，因为一旦排序，便改变了s1或s2中各个字符原始的相对顺序，既然不能排序，咱们可以考虑下用动态规划的方法，令dp[i][j]代表s3[0...i+j-1]是否由s1[0...i-1]和s2[0...j-1]的字符组成

- 如果s1当前字符（即s1[i-1]）等于s3当前字符（即s3[i+j-1]），而且dp[i-1][j]为真，那么可以取s1当前字符而忽略s2的情况，dp[i][j]返回真；
- 如果s2当前字符等于s3当前字符，并且dp[i][j-1]为真，那么可以取s2而忽略s1的情况，dp[i][j]返回真，其它情况，dp[i][j]返回假

参考代码如下：

```
public boolean IsInterleave (String s1, String s2, String s3) {  
    int n = s1.length(), m = s2.length(), s = s3.length();  
  
    //如果长度不一致，则s3不可能由s1和s2交错组成  
    if (n + m != s)  
        return false ;  
  
    boolean [][]dp = new boolean [n + 1 ][m + 1 ];  
  
    //在初始化边界时，我们认为空串可以由空串组成，因此dp[0][0]赋值为true。  
    dp[ 0 ][ 0 ] = true ;
```



```

    for ( int i = 0 ; i < n + 1 ; i++){
        for ( int j = 0 ; j < m + 1 ; j++){
            if ( dp[i][j] || (i - 1 >= 0 && dp[i - 1 ][j] == true &&
                //取s1字符
                s1.charAt(i - 1 ) == s3.charAt(i + j - 1 )) ||

                (j - 1 >= 0 && dp[i][j - 1 ] == true &&
                //取s2字符
                s2.charAt(j - 1 ) == s3.charAt(i + j - 1 )) )

                dp[i][j] = true ;
            else
                dp[i][j] = false ;
        }
    }
    return dp[n][m]
}

```

理解本题及上段代码，对真正理解动态规划有一定帮助。

本章动态规划的习题

1.子序列个数

子序列的定义：对于一个序列 $a=a[1],a[2],\dots,a[n]$ ，则非空序列 $a'=a[p_1],a[p_2],\dots,a[p_m]$ 为 a 的一个子序列 其中 $1\leq p_1<p_2<\dots<p_m\leq n$ 。 例如：4,14,2,3和14,1,2,3都为4,13,14,1,2,3的子序列。

- 对于给出序列 a ，有些子序列可能是相同的，这里只算做1个。
- 要求输出 a 的不同子序列的数量。

2.数塔取数问题

一个高度为 N 的由正整数组成的三角形，从上走到下，求经过的数字和的最大值。 每次只能走到下一层相邻的数上，例如从第3层的6向下走，只能走到第4层的2或9上。

5

8 4

3 6 9

7 2 9 5

例子中的最优方案是： $5 + 8 + 6 + 9 = 28$ 。

3.最长公共子序列

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。

提示：最容易想到的算法是穷举搜索法，但考虑到最长公共子序列问题

也有最优子结构性质，可以用动态规划解决。

4.最长递增子序列

给定一个长度为 N 的数组 $a_0, a_1, a_2, \dots, a_{n-1}$ ，找出一个最长的单调递增子序列（注：递增的意思是对于任意的 $i < j$ ，都满足 $a_i < a_j$ ，此外子序列的意思是不要求连续，顺序不乱即可）。例如：给定一个长度为6的数组 $A\{5, 6, 7, 1, 2, 8\}$ ，则其最长的单调递增子序列为 $\{5, 6, 7, 8\}$ ，长度为4。

提示：一种解法是转换为最长公共子序列问题，另外一种解法则是动态规划。当我们考虑动态规划解决时，可以定义 $dp[i]$ 为以 a_i 为末尾的最长递增子序列的长度，故以 a_i 结尾的递增子序列

- 要么是只包含 a_i 的子序列
- 要么是在满足 $j < i$ 并且 $a_j < a_i$ 的以 a_i 为结尾的递增子序列末尾，追加上 a_i 后得到的子序列

如此，便可建立递推关系，在 $O(N^2)$ 时间内解决这个问题。

5.木块砌墙

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：



提示：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 1×2 的多米诺骨牌填满 $M \times N$ 的矩形有多少种方案， $M \leq 5$ ， $N < 2^{31}$ ，输出答案 $\text{mod } p$ 的结果



第三部分 综合演练

第六章 海量数据处理

本章导读

所谓海量数据处理，是指基于海量数据的存储、处理、和操作。正因为数据量太大，所以导致要么无法在较短时间内迅速解决，要么无法一次性装入内存。

事实上，针对时间问题，可以采用巧妙的算法搭配合适的数据结构（如布隆过滤器、哈希、位图、堆、数据库、倒排索引、Trie树）来解决；而对于空间问题，可以采取分而治之（哈希映射）的方法，也就是说，把规模大的数据转化为规模小的，从而各个击破。

此外，针对常说的单机及集群问题，通俗来讲，单机就是指处理装载数据的机器有限（只要考虑CPU、内存、和硬盘之间的数据交互），而集群的意思是指机器有多台，适合分布式处理或并行计算，更多考虑节点与节点之间的数据交互。

一般说来，处理海量数据问题，有以下十种典型方法：

- 1.哈希分治；
- 2.simhash算法；
- 3.外排序；
- 4.MapReduce；
- 5.多层划分；
- 6.位图；
- 7.布隆过滤器；
- 8.Trie树；
- 9.数据库；
- 10.倒排索引。

受理论之限，本章将摒弃绝大部分的细节，只谈方法和模式论，注重用最通俗、最直白的语言阐述相关问题。最后，有一点必须强调的是，全章行文是基于面试题的分析基础之上的，具体实践过程中，还得视具体情况具体分析，且各个场景下需要考虑的细节也远比本章所描述的任何一种解决方案复杂得多。

关联式容器

一般来说，STL容器分为：

- 序列式容器(vector/list/deque/stack/queue/heap)，和关联式容器。
 - 其中，关联式容器又分为set(集合)和map(映射表)两大类，以及这两大类的衍生体multiset(多键集合)和multimap(多键映射表)，这些容器均以RB-tree（red-black tree, 红黑树）完成。
 - 此外，还有第3类关联式容器，如hashtable(散列表)，以及以hashtable为底层机制完成的hash_set(散列集合)/hash_map(散列映射表)/hash_multiset(散列多键集合)/hash_multimap(散列多键映射表)。也就是说，set/map/multiset/multimap都内含一个RB-tree，而hash_set/hash_map/hash_multiset/hash_multimap都内含一个hashtable。

所谓关联式容器，类似关联式数据库，每笔数据或每个元素都有一个键值(key)和一个实值(value)，即所谓的Key-Value(键-值对)。当元素被插入到关联式容器中时，容器内部结构(RB-tree/hashtable)便依照其键值大小，以某种特定规则将这个元素放置于适当位置。

包括在非关联式数据库中，比如，在MongoDB内，文档(document)是最基本的数据组织形式，每个文档也是以Key-Value（键-值对）的方式组织起来。一个文档可以有多个Key-Value组合，每个Value可以是不同的类型，比如String、Integer、List等等。

```
{ "name" : "July",  
  "sex" : "male",  
  "age" : 23 }
```

set/map/multiset/multimap

set，同map一样，所有元素都会根据元素的键值自动被排序，因为set/map两者的所有各种操作，都只是转而调用RB-tree的操作行为，不过，值得注意的是，两者都不允许两个元素有相同的键值。

不同的是：set的元素不像map那样可以同时拥有实值(value)和键值

(key), set元素的键值就是实值, 实值就是键值, 而map的所有元素都是pair, 同时拥有实值(value)和键值(key), pair的第一个元素被视为键值, 第二个元素被视为实值。

至于multiset/multimap, 他们的特性及用法和set/map完全相同, 唯一的差别就在于它们允许键值重复, 即所有的插入操作基于RB-tree的insert_equal()而非insert_unique()。

hash_set/hash_map/hash_multiset/hash_multimap

hash_set/hash_map, 两者的一切操作都是基于hashtable之上。不同的是, hash_set同set一样, 同时拥有实值和键值, 且实质就是键值, 键值就是实值, 而hash_map同map一样, 每一个元素同时拥有一个实值(value)和一个键值(key), 所以其使用方式, 和上面的map基本相同。但由于hash_set/hash_map都是基于hashtable之上, 所以不具备自动排序功能。为什么? 因为hashtable没有自动排序功能。

至于hash_multiset/hash_multimap的特性与上面的multiset/multimap完全相同, 唯一的差别就是它们hash_multiset/hash_multimap的底层实现机制是hashtable (而multiset/multimap, 上面说了, 底层实现机制是RB-tree), 所以它们的元素都不会被自动排序, 不过也都允许键值重复。

所以, 综上, 说白了, 什么样的结构决定其什么样的性质, 因为set/map/multiset/multimap都是基于RB-tree之上, 所以有自动排序功能, 而hash_set/hash_map/hash_multiset/hash_multimap都是基于hashtable之上, 所以不含有自动排序功能, 至于加个前缀multi 无非就是允许键值重复而已。

分而治之

方法介绍

对于海量数据而言，由于无法一次性装进内存处理，导致我们不得不把海量的数据通过hash映射分割成相应的小块数据，然后再针对各个小块数据通过hash_map进行统计或其它操作。

那什么是hash映射呢？简单来说，就是为了便于计算机在有限的内存中处理big数据，我们通过一种映射散列的方式让数据均匀分布在对应的内存位置(如大数据通过取余的方式映射成小数存放在内存中，或大文件映射成多个小文件)，而这个映射散列方式便是我们通常所说的hash函数，设计的好的hash函数能让数据均匀分布而减少冲突。

问题实例

1、海量日志数据，提取出某日访问百度次数最多的那个IP

分析：百度作为国内第一大搜索引擎，每天访问它的IP数量巨大，如果想一次性把所有IP数据装进内存处理，则内存容量明显不够，故针对数据太大，内存受限的情况，可以把大文件转化成（取模映射）小文件，从而大而化小，逐个处理。

换言之，先映射，而后统计，最后排序。

解法：具体分为以下3个步骤

- 1.分而治之/hash映射
 - 首先把这一天访问百度日志的所有IP提取出来，然后逐个写入到一个大文件中，接着采用映射的方法，比如%1000，把整个大文件映射为1000个小文件。
- 2.hash_map统计
 - 当大文件转化成了小文件，那么我们便可以采用hash_map(ip, value)来分别对1000个小文件中的IP进行频率统计，再找出每个小文件中出现频率最大的IP。
- 3.堆/快速排序

- 统计出1000个频率最大的IP后，依据各自频率的大小进行排序(可采取堆排序)，找出那个频率最大的IP，即为所求。

注：Hash取模是一种等价映射，不会存在同一个元素分散到不同小文件中去的情况，即这里采用的是%1000算法，那么同一个IP在hash后，只可能落在同一个文件中，不可能被分散的。

2、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，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。

由上面第1题，我们知道，数据大则划为小的，例如一亿个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hash_map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。

但对于本题，数据规模比较小，能一次性装入内存。因为根据题目描述，虽然有一千万个Query，但是由于重复度比较高，故去除重复后，事实上只有300万的Query，每个Query255Byte，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 $3M \times 1K / 4 = 0.75G$ 。所以可以将所有字符串都存放在内存中进行处理）。

所以我们放弃分而治之/hash映射的步骤，直接上hash_map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：hash_map + 堆。

解法：

- 1.hash_map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字符串，Value为该Query出现次数的hash_map，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, 并将Value值设为1; 如果该字串在Table中, 那么将该字串的计数加1 即可。最终我们在O(N)的时间复杂度内用hash_map完成了统计;

- 2.堆排序

- 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: $O(n) + N' * O(\log k)$, 其中, N为1000万, N'为300万。

关于第2步堆排序, 可以维护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 $\log k$ 。

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。

3、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字节, 内存限制大小是1M。返回频数最高的100个词

解法:

- 1.分而治之/hash映射

- 顺序读取文件, 对于每个词x, 取 $\text{hash}(x) \% 5000$, 然后把该值存到5000个小文件 (记为 $x_0, x_1, \dots, x_{4999}$) 中。这样每个文件大概是200k左右。当然, 如果其中有的小文件超过了1M大小, 还可以按照类似的方法继续往下分, 直到分解得到的小文件的大小都不超过1M。

- 2.hash_map统计

- 对每个小文件, 采用trie树/hash_map等统计每个文件中出现的词以及相应的频率。

- 3.堆/归并排序

- 取出出现频率最大的100个词（可以用含100个结点的最小堆）后，再把100个词及相应的频率存入文件，这样又得到了5000个文件。最后就是把这5000个文件进行归并（类似于归并排序）的过程了。

4、海量数据分布在100台电脑中，想个办法高效统计出这批数据的TOP10

解法一：

如果同一个数据元素只出现在某一台机器中，那么可以采取以下步骤统计出现次数TOP10的数据元素：

- 1.堆排序
 - 在每台电脑上求出TOP 10，可以采用包含10个元素的堆完成（TOP 10小，用最大堆，TOP 10大，用最小堆，比如求TOP10大，我们首先取前10个元素调整成最小堆，如果发现，然后扫描后面的数据，并与堆顶元素比较，如果比堆顶元素大，那么用该元素替换堆顶，然后再调整为最小堆。最后堆中的元素就是TOP 10大）。
- 2.组合归并
 - 求出每台电脑上的TOP 10后，然后把这100台电脑上的TOP 10组合起来，共1000个数据，再利用上面类似的方法求出TOP 10就可以了。

解法二：

但如果同一个元素重复出现在不同的电脑中呢，比如拿两台机器求top 2的情况来说：

- 第一台的数据分布及各自出现频率为：a(50)，b(50)，c(49)，d(49)，e(0)，f(0)
 - 其中，括号里的数字代表某个数据出现的频率，如a(50)表示a出现了50次。
- 第二台的数据分布及各自出现频率为：a(0)，b(0)，c(49)，d(49)，e(50)，f(50)

这个时候，你可以有两种方法：

- 遍历一遍所有数据，重新hash取摸，如此使得同一个元素只出现在单独的一台电脑中，然后采用上面所说的方法，统计每台电脑中各个元素的出现次数找出TOP 10，继而组合100台电脑上的TOP 10，找出最终的TOP 10。
- 或者，暴力求解：直接统计统计每台电脑中各个元素的出现次数，然后把同一个元素在不同机器中的出现次数相加，最终从所有数据中找出TOP 10。

5、有**10**个文件，每个文件**1G**，每个文件的每一行存放的都是用户的**query**，每个文件的**query**都可能重复。要求你按照**query**的频度排序

解法一：

- 1.hash映射
 - 顺序读取10个文件，按照 $\text{hash}(\text{query})\%10$ 的结果将query写入到另外10个文件（记为a0,a1,..a9）中。这样新生成的文件每个的大小大约也1G（假设hash函数是随机的）。
- 2.hash_map统计
 - 找一台内存在2G左右的机器，依次对用hash_map(query, query_count)来统计每个query出现的次数。注：hash_map(query, query_count)是用来统计每个query的出现次数，不是存储他们的值，出现一次，则count+1。
- 3.堆/快速/归并排序
 - 利用快速/堆/归并排序按照出现次数进行排序，将排序好的query和对应的query_cout输出到文件中，这样得到了10个排好序的文件（记为□）。最后，对这10个文件进行归并排序（内排序与外排序相结合）。

解法二：

一般query的总量是有限的，只是重复的次数比较多而已，可能对于所有的query，一次性就可以加入到内存了。这样，我们就可以采用trie树/hash_map等直接来统计每个query出现的次数，然后按出现次数做快速/堆/归并排序就可以了。

解法三：

与解法1类似，但在做完hash，分成多个文件后，可以交给多个文件来

处理，采用分布式的架构来处理（比如MapReduce），最后再进行合并。

6、给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url？

解法：

可以估计每个文件安的大小为 $5G \times 64 = 320G$ ，远远大于内存限制的4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

- 1.分而治之/hash映射
 - 遍历文件a，对每个url求取 \square ，然后根据所取得的值将url分别存储到1000个小文件（记为 \square ，这里漏写个了a1）中。这样每个小文件的大约为300M。遍历文件b，采取和a相同的方式将url分别存储到1000小文件中（记为 \square ）。这样处理后，所有可能相同的url都在对应的小文件（ \square ）中，不对应的小文件不可能有相同的url。然后我们只要求出1000对小文件中相同的url即可。
- 2.hash_set统计
 - 求每对小文件中相同的url时，可以把其中一个小文件的url存储到hash_set中。然后遍历另一个小文件的每个url，看其是否在刚才构建的hash_set中，如果是，那么就是共同的url，存到文件里面就可以了。

7、100万个数中找出最大的100个数

解法一：采用局部淘汰法。选取前100个元素，并排序，记为序列L。然后一次扫描剩余的元素x，与排好序的100个元素中最小的元素比，如果比这个最小的要大，那么把这个最小的元素删除，并把x利用插入排序的思想，插入到序列L中。依次循环，知道扫描了所有的元素。复杂度为 $O(100万 \times 100)$ 。

解法二：采用快速排序的思想，每次分割之后只考虑比主元大的一部分，直到比主元大的一部分比100多的时候，采用传统排序算法排序，取前100个。复杂度为 $O(100万 \times 100)$ 。

解法三：在前面的题中，我们已经提到了，用一个含100个元素的最小

堆完成。复杂度为 $O(100万 * \lg 100)$ 。

举一反三

1、怎么在海量数据中找出重复次数最多的一个？

提示：先做hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。然后找出上一步求出的数据中重复次数最多的一个就是所求（具体参考前面的题）。

2、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。

提示：上千万或上亿的数据，现在的机器的内存应该能存下。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，可以用第2题提到的堆机制完成。

3、一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前10个词，请给出思想，给出时间复杂度分析。

提示：这题是考虑时间效率。用trie树统计每个词出现的次数，时间复杂度是 $O(n * le)$ （le表示单词的平均长度）。然后是找出出现最频繁的前10个词，可以用堆来实现，前面的题中已经讲到了，时间复杂度是 $O(n * \lg 10)$ 。所以总的时间复杂度，是 $O(n * le)$ 与 $O(n * \lg 10)$ 中较大的哪一个。

4、1000万字符串，其中有些是重复的，需要把重复的全部去掉，保留没有重复的字符串。请怎么设计和实现？

提示：这题用trie树比较合适，hash_map也行。当然，也可以先hash成小文件分开处理再综合。

5、一个文本文件，找出前10个经常出现的词，但这次文件比较长，说是上亿行或十亿行，总之无法一次读入内存，问最优解。

提示：首先根据用hash并求模，将文件分解为多个小文件，对于单个文件利用上题的方法求出每个文件中10个最常出现的词。然后再进行归并处理，找出最终的10个最常出现的词。

simhash算法

方法介绍

背景

如果某一天，面试官问你如何设计一个比较两篇文章相似度的算法？可能你会回答几个比较传统点的思路：

- 一种方案是先将两篇文章分别进行分词，得到一系列特征向量，然后计算特征向量之间的距离（可以计算它们之间的欧氏距离、海明距离或者夹角余弦等等），从而通过距离的大小来判断两篇文章的相似度。
- 另外一种方案是传统hash，我们考虑为每一个web文档通过hash的方式生成一个指纹（finger print）。

下面，我们来分析下这两种方法。

- 采取第一种方法，若是只比较两篇文章的相似性还好，但如果是海量数据呢，有着数以百万甚至亿万网页，要求你计算这些网页的相似度。你还会去计算任意两个网页之间的距离或夹角余弦么？想必你不会了。
- 而第二种方案中所说的传统加密方式md5，其设计的目的是为了让整个分布尽可能地均匀，但如果输入内容一旦出现哪怕轻微的变化，hash值就会发生很大的变化。

举个例子，我们假设有以下三段文本：

- the cat sat on the mat
- the cat sat on a mat
- we all scream for ice cream

使用传统hash可能会得到如下的结果：

- `irb(main):006:0> p1 = 'the cat sat on the mat'`

- irb(main):007:0> p1.hash => 415542861
- irb(main):005:0> p2 = 'the cat sat on a mat'
 - irb(main):007:0> p2.hash => 668720516
- irb(main):007:0> p3 = 'we all scream for ice cream'
 - irb(main):007:0> p3.hash => 767429688 "

可理想当中的hash函数，需要对几乎相同的输入内容，产生相同或者相近的hash值，换言之，hash值的相似程度要能直接反映输入内容的相似程度，故md5等传统hash方法也无法满足我们的需求。

出世

车到山前必有路，来自于Google Moses Charikar发表的一篇论文“detecting near-duplicates for web crawling”中提出了simhash算法，专门用来解决亿万级别的网页的去重任务。

simhash作为locality sensitive hash（局部敏感哈希）的一种：

- 其主要思想是降维，将高维的特征向量映射成低维的特征向量，通过两个向量的Hamming Distance来确定文章是否重复或者高度近似。
 - 其中，Hamming Distance，又称汉明距离，在信息论中，两个等长字符串之间的汉明距离是两个字符串对应位置的不同字符的个数。也就是说，它就是将一个字符串变换成另外一个字符串所需要替换的字符个数。例如：1011101 与 1001001 之间的汉明距离是 2。至于我们常说的字符串编辑距离则是一般形式的汉明距离。

如此，通过比较多个文档的simHash值的海明距离，可以获取它们的相似度。

流程

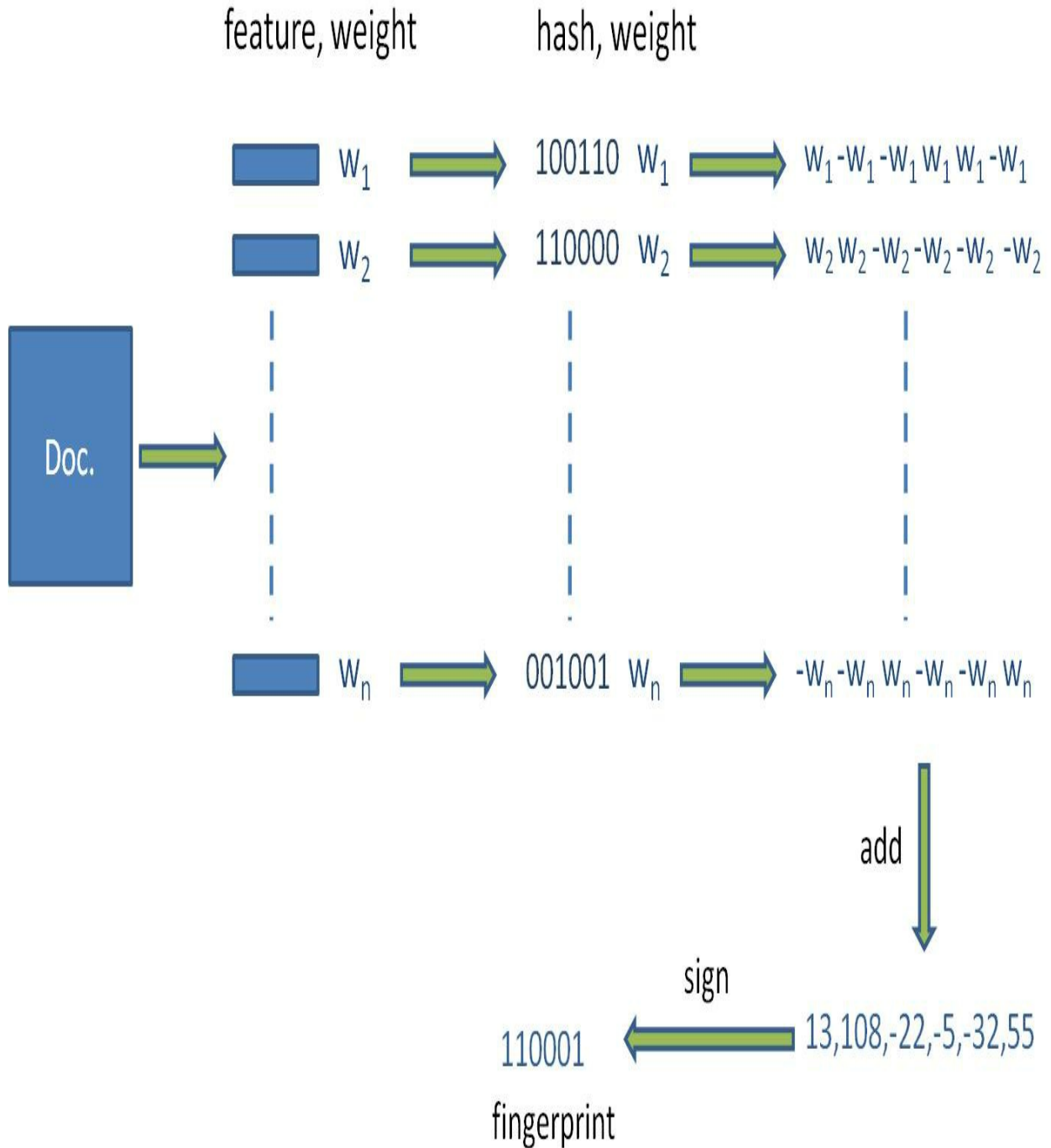
simhash算法分为5个步骤：分词、hash、加权、合并、降维，具体过程如下所述：

- 分词

- 给定一段语句，进行分词，得到有效的特征向量，然后为每一个特征向量设置1-5等5个级别的权重（如果是给定一个文本，那么特征向量可以是文本中的词，其权重可以是这个词出现的次数）。例如给定一段语句：“CSDN博客结构之法算法之道的作者July”，分词后为：“CSDN 博客 结构 之法 算法 之 道的 作者 July”，然后为每个特征向量赋予权值：CSDN(4) 博客(5) 结构(3) 之(1) 法(2) 算法(3) 之(1) 道(2) 的(1) 作者(5) July(5)，其中括号里的数字代表这个单词在整条语句中的重要程度，数字越大代表越重要。
- hash
 - 通过hash函数计算各个特征向量的hash值，hash值为二进制数01组成的n-bit签名。比如“CSDN”的hash值Hash(CSDN)为100101，“博客”的hash值Hash(博客)为“101011”。就这样，字符串就变成了一系列数字。
- 加权
 - 在hash值的基础上，给所有特征向量进行加权，即 $W = Hash \times weight$ ，且遇到1则hash值和权值正相乘，遇到0则hash值和权值负相乘。例如给“CSDN”的hash值“100101”加权得到： $W(CSDN) = 100101 \times 4 = 4 -4 -4 4 -4 4$ ，给“博客”的hash值“101011”加权得到： $W(博客) = 101011 \times 5 = 5 -5 5 -5 5 5$ ，其余特征向量类似此般操作。
- 合并
 - 将上述各个特征向量的加权结果累加，变成只有一个序列串。拿前两个特征向量举例，例如“CSDN”的“4 -4 -4 4 -4 4”和“博客”的“5 -5 5 -5 5 5”进行累加，得到“4+5 -4+-5 -4+5 4+-5 -4+5 4+5”，得到“9 -9 1 -1 1 1”。
- 降维
 - 对于n-bit签名的累加结果，如果大于0则置1，否则置0，从而得到该语句的simhash值，最后我们便可以根据不同语句simhash的海明距离来判断它们的相似度。例如把上面计算出来的“9 -9 1 -1 1 1”降维（某位大于0记为1，小于0记为0），得到的01串为：“1 0 1 0 1 1”，从而形成它们的simhash签名。

其流程如下图所示：

Simhash



应用

- 每篇文档得到SimHash签名值后，接着计算两个签名的海明距离即可。根据经验值，对64位的 SimHash值，海明距离在3以内的可认为相似度比较高。
 - 海明距离的求法：异或时，只有在两个比较的位不同时其结果是1，否则结果为0，两个二进制“异或”后得到1的个数即为海明距离的大小。

举个例子，上面我们计算到的“CSDN博客”的simhash签名值为“1 0 1 0 1 1”，假定我们计算出另外一个短语的签名值为“1 0 1 0 0 0”，那么根据异或规则，我们可以计算出这两个签名的海明距离为2，从而判定这两个短语的相似度是比较高的。

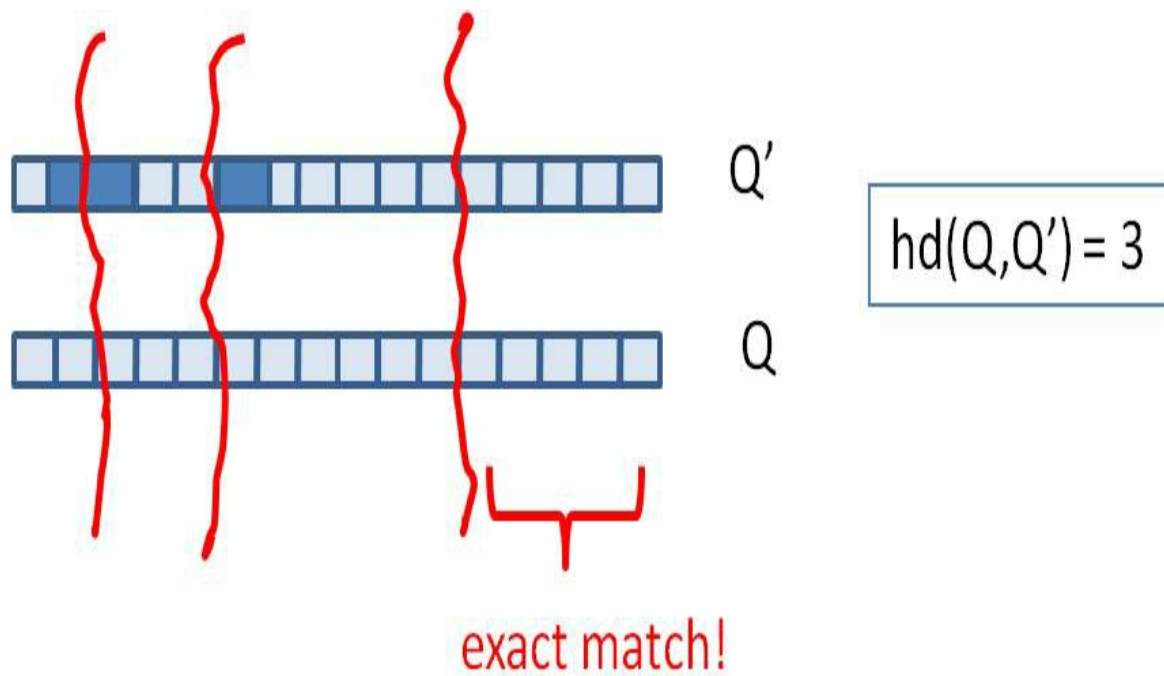
换言之，现在问题转换为：对于64位的SimHash值，我们只要找到海明距离在3以内的所有签名，即可找出所有相似的短语。

但关键是，如何将其扩展到海量数据呢？譬如如何在海量的样本库中查询与其海明距离在3以内的记录呢？

- 一种方案是查找待查询文本的64位simhash code的所有3位以内变化的组合
 - 大约需要四万多次的查询。
- 另一种方案是预生成库中所有样本simhash code的3位变化以内的组合
 - 大约需要占据4万多倍的原始空间。

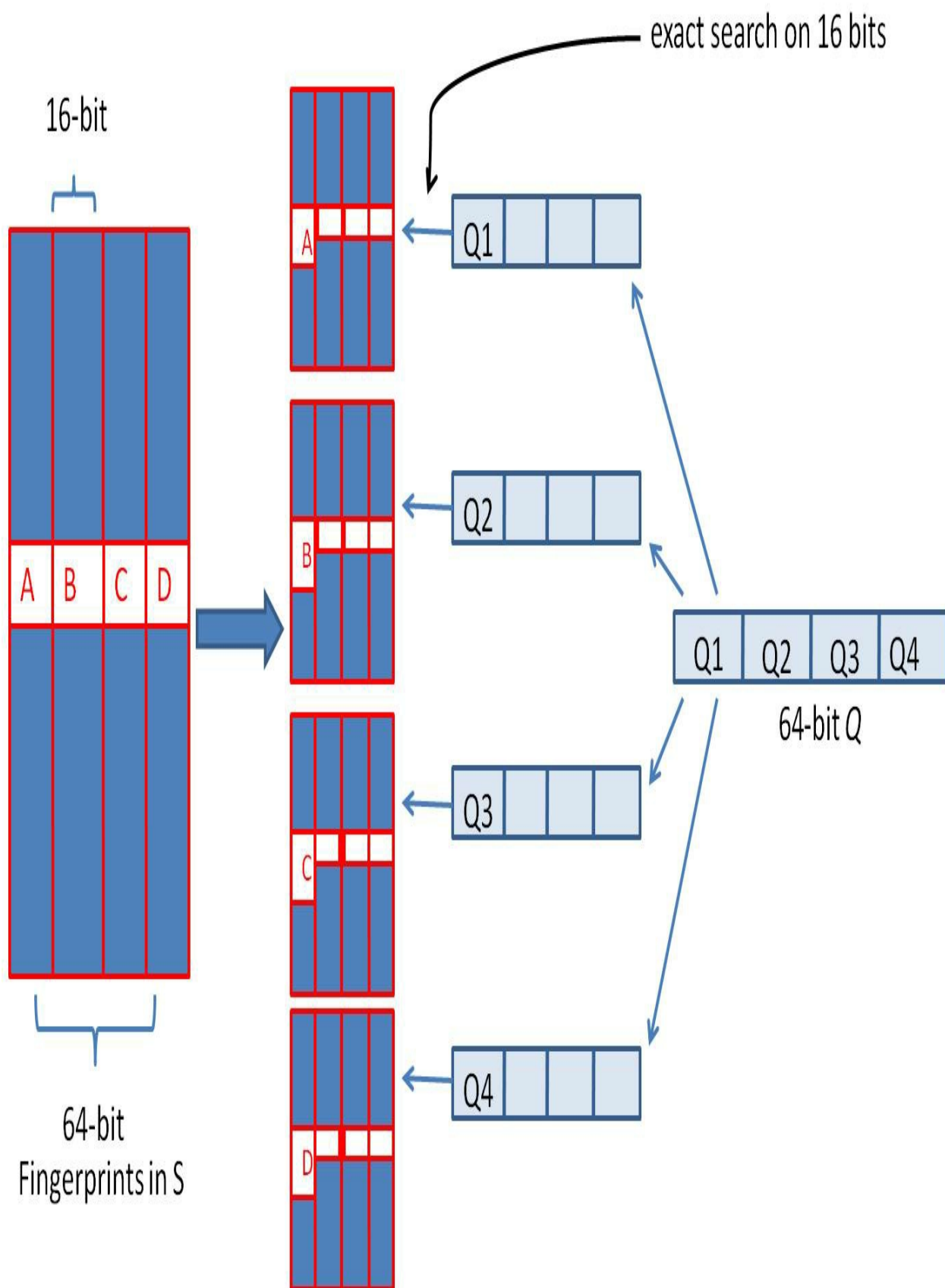
这两种方案，要么时间复杂度高，要么空间复杂度复杂，能否有一种方案可以达到时空复杂度的绝佳平衡呢？答案是肯定的：

- 我们可以把 64 位的二进制simhash签名均分成4块，每块16位。根据鸽巢原理（也称抽屉原理），如果两个签名的海明距离在 3 以内，它们必有一块完全相同。如下图所示：



- 然后把分成的4块中的每一个块分别作为前16位来进行查找，建倒排索引。

具体如下图所示：



如此，如果样本库中存有 2^{34} （差不多10亿）的simhash签名，则每个table返回 $2^{(34-16)}=262144$ 个候选结果，大大减少了海明距离的计算成本。

- 假设数据是均匀分布，16位的数据，产生的像限为 2^{16} 个，则平均每个像限分布的文档数则为 $2^{34}/2^{16} = 2^{(34-16)}$ ，四个块返回的总结果数为 $4 * 262144$ （大概 100 万）。
 - 这样，原本需要比较10亿次，经过索引后，大概只需要处理100万次。

（部分内容及图片参考自：<http://grunt1223.iteye.com/blog/964564>，后续图片会全部重画）

问题实例

待续。

@复旦李斌: simhash不是google发明的，是princeton的人早在stoc02上发表的。google在www07上的那篇论文只是在网页查重上应用了下。事实上www07中的算法是stoc02中随机超平面的一个极其巧妙的实现，bit差异的期望正好等于原始向量的余弦。

外排序

方法介绍

所谓外排序，顾名思义，即是在内存外面的排序，因为当要处理的数据量很大，而不能一次装入内存时，此时只能放在读写较慢的外存储器（通常是硬盘）上。

外排序通常采用的是一种“排序-归并”的策略。

- 在排序阶段，先读入能放在内存中的数据量，将其排序输出到一个临时文件，依此进行，将待排序数据组织为多个有序的临时文件；
- 尔后在归并阶段将这些临时文件组合为一个大的有序文件，也即排序结果。

假定现在有20个数据的文件A：{5 11 0 18 4 14 9 7 6 8 12 17 16 13 19 10 2 1 3 15}，但一次只能使用仅装4个数据的内容，所以，我们可以每趟对4个数据进行排序，即5路归并，具体方法如下述步骤：

- 我们先把“大”文件A，分割为a1，a2，a3，a4，a5等5个小文件，每个小文件4个数据
 - a1文件为：5 11 0 18
 - a2文件为：4 14 9 7
 - a3文件为：6 8 12 17
 - a4文件为：16 13 19 10
 - a5文件为：2 1 3 15
- 然后依次对5个小文件分别进行排序
 - a1文件完成排序后：0 5 11 18
 - a2文件完成排序后：4 7 9 14
 - a3文件完成排序后：6 8 12 17
 - a4文件完成排序后：10 13 16 19
 - a5文件完成排序后：1 2 3 15

- 最终多路归并，完成整个排序
 - 整个大文件A文件完成排序后：0 1 2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19

问题实例

1、给 10^7 个数据量的磁盘文件排序

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。

解法一：位图方案

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。

熟悉位图的朋友可能会想到用位图来表示这个文件集合。例如正如编程珠玑一书上所述，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 $5000000/8=625000\text{Byte}$ ，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，也只需要0.625M（可以使用第一次处理申请的内存）。因此，总共也只需要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。

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 $10000000/(1024 \times 8) \approx 1.2M$ ）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？

解法二：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 $1.25/2M$ ）， $1\sim4999999$ ，和 $5000000\sim9999999$ 。先遍历一趟，首先排序处理 $1\sim4999999$ 之间的整数（用 $5000000/8=625000$ 个字的存储空间来排序 $0\sim4999999$ 之间的整数），然后再第二趟，对 $5000001\sim10000000$ 之间的整数进行排序处理。

解法总结

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：

	时间复杂度	空间复杂度
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 $O(k \times n/k \times \log n/k)$ ），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

举一反三

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。

分布式处理之MapReduce

方法介绍

MapReduce是一种计算模型，简单的说就是将大批量的工作（数据）分解（MAP）执行，然后再将结果合并成最终结果（REDUCE）。这样做的好处是可以在任务被分解后，可以通过大量机器进行并行计算，减少整个操作的时间。但如果你要我再通俗点介绍，那么，说白了，Mapreduce的原理就是一个归并排序。

适用范围：数据量大，但是数据种类小可以放入内存

基本原理及要点：将数据交给不同的机器去处理，数据划分，结果归约。

基础架构

想读懂此文，读者必须先要明确以下几点，以作为阅读后续内容的基础知识储备：

1. MapReduce是一种模式。
2. Hadoop是一种框架。
3. Hadoop是一个实现了MapReduce模式的开源的分布式并行编程框架。

所以，你现在，知道了什么是MapReduce，什么是hadoop，以及这两者之间最简单的联系，而本文的主旨即是，一句话概括：在**hadoop**的框架上采取**MapReduce**的模式处理海量数据。下面，咱们可以依次深入学习和了解MapReduce和hadoop这两个东西了。

MapReduce模式

前面说了，MapReduce是一种模式，一种什么模式呢？一种云计算的核心计算模式，一种分布式运算技术，也是简化的分布式编程模式，它主要用于解决问题的程序开发模型，也是开发人员拆解问题的方法。

Ok, 光说不上图, 没用。如下图所示, MapReduce模式的主要思想是将自动分割要执行的问题(例如程序)拆解成Map(映射)和Reduce(化简)的方式, 流程图如下图1所示:



在数据被分割后通过Map函数的程序将数据映射成不同的区块, 分配给计算机机群处理达到分布式运算的效果, 在通过Reduce函数的程序将结果汇整, 从而输出开发者需要的结果。

MapReduce借鉴了函数式程序设计语言的设计思想, 其软件实现是指定一个Map函数, 把键值对(key/value)映射成新的键值对(key/value), 形成一系列中间结果形式的key/value对, 然后把它们传给Reduce(规约)函数, 把具有相同中间形式key的value合并在一起。Map和Reduce函数具有一定的关联性。函数描述如表1所示:



MapReduce致力于解决大规模数据处理的问题, 因此在设计之初就考虑了数据的局部性原理, 利用局部性原理将整个问题分而治之。

MapReduce集群由普通PC机构成, 为无共享式架构。在处理之前, 将数据集分布至各个节点。处理时, 每个节点就近读取本地存储的数据处理(map), 将处理后的数据进行合并(combine)、排序(shuffle and sort)后再分发(至reduce节点), 避免了大量数据的传输, 提高了处理效率。无共享式架构的另一个好处是配合复制(replication)策略, 集群可以具有良好的容错性, 一部分节点的down机对集群的正常工作不会造成影响。

ok, 你可以再简单看看下副图, 整幅图是有关hadoop的作业调优参数及原理, 图的左边是MapTask运行示意图, 右边是ReduceTask运行示意图:



如上图所示, 其中map阶段, 当map task开始运算, 并产生中间数据后并非直接而简单的写入磁盘, 它首先利用内存buffer来对已经产生的buffer进行缓存, 并在内存buffer中进行一些预排序来优化整个map的性能。而上图右边的reduce阶段则经历了三个阶段, 分别Copy->Sort-

>reduce。我们能明显的看出，其中的Sort是采用的归并排序，即merge sort。

问题实例

1. The canonical example application of MapReduce is a process to count the appearances of each different word in a set of documents:
2. 海量数据分布在100台电脑中，想个办法高效统计出这批数据的TOP10。
3. 一共有N个机器，每个机器上有N个数。每个机器最多存 $O(N)$ 个数并对它们操作。如何找到 N^2 个数的中数(median)?

多层划分

方法介绍

多层划分法，本质上还是分而治之的思想，因为元素范围很大，不能利用直接寻址表，所以通过多次划分，逐步确定范围，然后最后在一个可以接受的范围内进行。

问题实例

1、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

分析：有点像鸽巢原理，整数个数为 2^{32} ，也就是，我们可以将这 2^{32} 个数，划分为 2^8 个区域(比如用单个文件代表一个区域)，然后将数据分离到不同的区域，然后不同的区域在利用bitmap就可以直接解决了。也就是说只要有足够的磁盘空间，就可以很方便的解决。

2、5亿个int找它们的中位数

分析：首先我们将int划分为 2^{16} 个区域，然后读取数据统计落到各个区域里的数的个数，之后我们根据统计结果就可以判断中位数落到那个区域，同时知道这个区域中的第几大数刚好是中位数。然后第二次扫描我们只统计落在这个区域中的那些数就可以了。

实际上，如果不是int是int64，我们可以经过3次这样的划分即可降低到可以接受的程度。即可以先将int64分成 2^{24} 个区域，然后确定区域的第几大数，在将该区域分成 2^{20} 个子区域，然后确定是子区域的第几大数，然后子区域里的数的个数只有 2^{20} ，就可以直接利用direct addr table进行统计了。

Bitmap

方法介绍

什么是Bit-map

所谓的Bit-map就是用一个bit位来标记某个元素对应的Value，而Key即是该元素。由于采用了Bit为单位来存储数据，因此在存储空间方面，可以大大节省。

来看一个具体的例子，假设我们要对0-7内的5个元素(4,7,2,5,3)排序（这里假设这些元素没有重复）。那么我们就可以采用Bit-map的方法来达到排序的目的。要表示8个数，我们就只需要8个Bit（1Bytes），首先我们开辟1Byte的空间，将这些空间的所有Bit位都置为0(如下图：)



然后遍历这5个元素，首先第一个元素是4，那么就把4对应的位置为1（可以这样操作 $p+(i/8)|(0 \times 01 \ll (i \% 8))$ 当然了这里的操作涉及到Big-ending和Little-ending的情况，这里默认为Big-ending），因为是从零开始的，所以要把第五位置为一（如下图）：



然后再处理第二个元素7，将第八位置为1，接着再处理第三个元素，一直到最后处理完所有的元素，将相应的位置为1，这时候的内存的Bit位的状态如下：



然后我们现在遍历一遍Bit区域，将该位是一的位的编号输出（2，3，4，5，7），这样就达到了排序的目的。下面的代码给出了一个Bitmap的用法：排序。

```
//定义每个Byte中有8个Bit位  
# include <memory.h>
```

```

#define BYTESIZE 8

void SetBit ( char *p, int posi)
{
    for ( int i= 0 ; i < (posi/BYTESIZE); i++)
    {
        p++;
    }

    *p = *p|( 0x01 <<(posi%BYTESIZE)); //将该Bit位赋值1

    return ;
}

void BitMapSortDemo ()
{
    //为了简单起见，我们不考虑负数

    int num[] = { 3 , 5 , 2 , 10 , 6 , 12 , 8 , 14 , 9 };

    //BufferLen这个值是根据待排序的数据中最大值确定的
    //待排序中的最大值是14，因此只需要2个Bytes(16个Bit)
    //就可以了。

    const int BufferLen = 2 ;

    char *pBuffer = new char [BufferLen];

    //要将所有的Bit位置为0，否则结果不可预知。

    memset (pBuffer, 0 ,BufferLen);

    for ( int i= 0 ;i< 9 ;i++)
    {
        //首先将相应Bit位上置为1

        SetBit(pBuffer,num[i]);
    }
}

```



```

//输出排序结果

for ( int i= 0 ;i<BufferLen;i++) //每次处理一个字节(Byte)
{
    for ( int j= 0 ;j<BYTESIZE;j++) //处理该字节中的每个Bit位
    {
        //判断该位上是否是1，进行输出，这里的判断比较笨。
        //首先得到该第j位的掩码（0x01<<j），将内存区中的
        //位和此掩码作与操作。最后判断掩码是否和处理后的
        //结果相同
        if ((*pBuffer&( 0x01 <<j)) == ( 0x01 <<j))
        {
            printf ( "%d " ,i*BYTESIZE + j);
        }
    }
    pBuffer++;
}

int _tmain( int argc, _TCHAR* argv[])
{
    BitMapSortDemo();

    return 0 ;
}

```

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.

问题实例

1、在2.5亿个整数中找出不重复的整数，注，内存不足以容纳这2.5亿个整数

解法一：采用2-Bitmap（每个数分配2bit，00表示不存在，01表示出现一次，10表示多次，11无意义）进行，共需内存 $2^{32} * 2 \text{ bit} = 1 \text{ GB}$ 内存，还可以接受。然后扫描这2.5亿个整数，查看Bitmap中相对应位，如果是00变01，01变10，10保持不变。扫描完后，查看bitmap，把对应位是01的整数输出即可。

解法二：也可采用与第1题类似的方法，进行划分小文件的方法。然后在小文件中找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。”

2、给40亿个不重复的unsigned int的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那40亿个数当中？

解法一：可以用位图/Bitmap的方法，申请512M的内存，一个bit位代表一个unsigned int值。读入40亿个数，设置相应的bit位，读入要查询的数，查看相应bit位是否为1，为1表示存在，为0表示不存在。

Bloom Filter

方法介绍

一、什么是Bloom Filter

Bloom Filter，被译作称布隆过滤器，是一种空间效率很高的随机数据结构，Bloom filter可以看做是对bit-map的扩展,它的原理是：

- 当一个元素被加入集合时，通过K个Hash函数将这个元素映射成一个位阵列（Bit array）中的K个点，把它们置为1**。检索时，我们只要看看这些点是不是都是1就（大约）知道集合中有没有它了：
 - 如果这些点有任何一个0，则被检索元素一定不在；
 - 如果都是1，则被检索元素很可能在。

其可以用来实现数据字典，进行数据的判重，或者集合求交集。

但Bloom Filter的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter通过极少的错误换取了存储空间的极大节省。

1.1、集合表示和元素查询

下面我们具体来看Bloom Filter是如何用位数组表示集合的。初始状态时，Bloom Filter是一个包含m位的位数组，每一位都置为0。



为了表达 $S=\{x_1, x_2, \dots, x_n\}$ 这样一个n个元素的集合，Bloom Filter使用k个相互独立的哈希函数（Hash Function），它们分别将集合中的每个元素映射到 $\{1, \dots, m\}$ 的范围中。对任意一个元素x，第i个哈希函数映射的位置 $h_i(x)$ 就会被置为1（ $1 \leq i \leq k$ ）。注意，如果一个位置多次被置为1，那么只有第一次会起作用，后面几次将没有任何效果。在下图中，

$k=3$ ，且有两个哈希函数选中同一个位置（从左边数第五位，即第二个“1”处）。



在判断 y 是否属于这个集合时，我们对 y 应用 k 次哈希函数，如果所有 $h_i(y)$ 的位置都是1（ $1 \leq i \leq k$ ），那么我们就认为 y 是集合中的元素，否则就认为 y 不是集合中的元素。下图中 y_1 就不是集合中的元素（因为 y_1 有一处指向了“0”位）。 y_2 或者属于这个集合，或者刚好是一个false positive。



1.2、错误率估计

前面我们已经提到了，Bloom Filter在判断一个元素是否属于它表示的集合时会有一定的错误率（false positive rate），下面我们就来估计错误率的大小。在估计之前为了简化模型，我们假设 x_1, x_2, \dots, x_n 的所有元素都被 k 个哈希函数映射到 m 位的位数组中时，这个位数组中某一位还是0的概率是：

$$p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

其中 $1/m$ 表示任意一个哈希函数选中这一位的概率（前提是哈希函数是完全随机的）， $(1-1/m)$ 表示哈希一次没有选中这一位的概率。要把 S 完全映射到位数组中，需要做 kn 次哈希。某一位还是0意味着 kn 次哈希都没有选中它，因此这个概率就是 $(1-1/m)$ 的 kn 次方。令 $p = e^{-kn/m}$ 是为了简化运算，这里用到了计算 e 时常用的近似：

$$\lim_{x \rightarrow \infty} \left(1 - \frac{1}{x}\right)^x = e^{-1}$$

令 p 为位数组中0的比例，则 p 的数学期望 $E(p) = p'$ 。在 p 已知的情况下，要求的错误率（false positive rate）为：

$$(1 - p')^k \approx (1 - p)^k$$

$(1 - p)$ 为位数组中1的比例， $(1 - p)^k$ 就表示 k 次哈希都刚好选中1的区域，即false positive rate。上式中第二步近似在前面已经提到了，现在来看第一步近似。 p' 只是 p 的数学期望，在实际中 p 的值有可能偏离它的数学期望值。M. Mitzenmacher已经证明^[2]，位数组中0的比例非常集中地分布在它的数学期望值的附近。因此，第一步的近似得以成立。分别将 p 和 p' 代入上式中，得：

$$f' = \left(1 - \frac{1}{m}\right)^{kn} \approx (1 - p')^k$$

$$f = \left(1 - e^{-kn/m}\right)^k \approx (1 - p)^k$$

相比 p' 和 f' ，使用 p 和 f 通常在分析中更为方便。

1.3、最优的哈希函数个数

既然Bloom Filter要靠多个哈希函数将集合映射到位数组中，那么应该选择几个哈希函数才能使元素查询时的错误率降到最低呢？这里有两个互斥的理由：如果哈希函数的个数多，那么在对一个不属于集合的元素进行查询时得到0的概率就大；但另一方面，如果哈希函数的个数少，那么位数组中的0就多。为了得到最优的哈希函数个数，我们需要根据上一小节中的错误率公式进行计算。

先用 p 和 f 进行计算。注意到 $f = \exp(k \ln(1 - e^{-kn/m}))$ ，我们令 $g = k \ln(1 - e^{-kn/m})$ ，只要让 g 取到最小， f 自然也取到最小。由于 $p = e^{-kn/m}$ ，我们可以将 g 写成

$$g = -\frac{mn}{\ln(p) \ln(1-p)}$$

根据对称性法则可以很容易看出当 $p = 1/2$ ，也就是 $k = \ln 2 \cdot (m/n)$ 时， g 取得最小值。在这种情况下，最小错误率 f 等于 $(1/2)^k \approx (0.6185)^{m/n}$ 。另外，注意到 p 是位数组中某一位仍是0的概率，所以 $p = 1/2$ 对应着位数组中0和1各一半。换句话说，要想保持错误率低，最好让位数组有一半还空着。

需要强调的一点是， $p = 1/2$ 时错误率最小这个结果并不依赖于近似值 p 和 f 。同样对于 $f' = \exp(k \ln(1 - (1 - 1/m)^{kn}))$ ， $g' = k \ln(1 - (1 - 1/m)^{kn})$ ， $p' = (1 - 1/m)^{kn}$ ，我们可以将 g' 写成

$$g' = \frac{1}{\ln(1-1/m)} \frac{n}{\ln(p') \ln(1-p')}$$

同样根据对称性法则可以得到当 $p' = 1/2$ 时， g' 取得最小值。

1.4、位数组的大小

下面我们来看看，在不超过一定错误率的情况下，Bloom Filter至少需要多少位才能表示全集中任意 n 个元素的集合。假设全集中共有 u 个元素，允许的最大错误率为 ϵ ，下面我们来求位数组的位数 m 。

假设 X 为全集中任取 n 个元素的集合， $F(X)$ 是表示 X 的位数组。那么对于集合 X 中任意一个元素 x ，在 $s = F(X)$ 中查询 x 都能得到肯定的结果，即 s 能够接受 x 。显然，由于Bloom Filter引入了错误， s 能够接受的不仅仅是 X 中的元素，它还能够接受 $\epsilon(u - n)$ 个false positive。因此，对于一个确定的位数组来说，它能够接受总共 $n + \epsilon(u - n)$ 个元素。在 $n + \epsilon(u - n)$ 个元素中， s 真正表示的只有其中 n 个，所以一个确定的位数组可以表示

$$C_{n + \epsilon(u - n)}^n$$

个集合。 m 位的位数组共有 2^m 个不同的组合，进而可以推出， m 位的位数组可以表示

$$2^m C_{\epsilon(u-n)}^{n+}$$

个集合。全集中n个元素的集合总共有

$$C_u^n$$

个，因此要让m位的位数组能够表示所有n个元素的集合，必须有

$$2^m C_{\epsilon(u-n)}^{n+} \geq C_u^n$$

即：

$$m \geq \log_2 \frac{C_u^n C_{\epsilon(u-n)}^{n+}}{C_u^n} \approx \log_2 \frac{C_u^n}{C_{\epsilon(u-n)}^{n+}} \geq \log_2 \epsilon^{-n} = n \log_2(1/\epsilon)$$

上式中的近似前提是n和eu相比很小，这也是实际情况中常常发生的。根据上式，我们得出结论：在错误率不大于ε的情况下，m至少要等于n log₂(1/ε)才能表示任意n个元素的集合。

上一小节中我们曾算出当k = ln2 · (m/n)时错误率f最小，这时f = (1/2)^k = (1/2)^{mln2 / n}。现在令f ≤ ε，可以推出

$$m \geq n \frac{\log_2(1/\epsilon)}{\ln 2} = n \log_2 \log_2(1/\epsilon)$$

这个结果比前面我们算得的下界n log₂(1/ε)大了log₂ e ≈ 1.44倍。这说明在哈希函数的个数取到最优时，要让错误率不超过ε，m至少需要取到最小值的1.44倍。

问题实例

1、给你A,B两个文件，各存放50亿条URL，每条URL占用64字节，内存限制是4G，让你找出A,B文件共同的URL。如果是三个乃至n个文件呢？

分析：如果允许有一定的错误率，可以使用Bloom filter，4G内存大概可以表示340亿bit。将其中一个文件中的url使用Bloom filter映射为这340亿bit，然后挨个读取另外一个文件的url，检查是否与Bloom filter，如果是，那么该url应该是共同的url（注意会有一定的错误率）。”

Trie树（字典树）

方法介绍

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。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，查询效率比较高。

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

咱们先来看一个问题：假如现在给你10万个长度不超过10的单词，对于每一个单词，我们要判断它出没出现过，如果出现了，求第一次出现在第几个位置。对于这个问题，我们该怎么解决呢？

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10万的范围难以接受。

换个思路想：

- 假设我要查询的单词是abcd，那么在它前面的单词中，以b，c，d，f之类开头的显然不必考虑，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。

即如果现在有b, abc, abd, bcd, abcd, efg, hii 这6个单词，我们可以构建一棵如下图所示的树：



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，只要顺着他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。

这样一来我们查询和插入可以一起完成，所用时间仅仅为单词长度（在这个例子中，便是10）。这就是一棵trie树。

我们可以看到，trie树每一层的节点数是 26^i 级别的。所以为了节省空间，我们还可以用动态链表，或者用数组来模拟动态。而空间的花费，不会超过单词数×单词长度。

1.3、查询

Trie树是简单但实用的数据结构，通常用于实现字典查询。我们做即时响应用户输入的AJAX搜索框时，就是Trie开始。本质上，Trie是一颗存储多个字符串的树。相邻节点间的边代表一个字符，这样树的每条分支代表一则子串，而树的叶节点则代表完整的字符串。和普通树不同的地方是，相同的字符串前缀共享同一条分支。

下面，再举一个例子。给出一组单词，inn, int, at, age, adv, ant, 我们可以得到下面的Trie：



可以看出：

- 每条边对应一个字母。
- 每个节点对应一项前缀。叶节点对应最长前缀，即单词本身。
- 单词inn与单词int有共同的前缀“in”，因此他们共享左边的一条分支，root->i->in。同理，ate, age, adv, 和ant共享前缀"a"，所以他们共享从根节点到节点"a"的边。

查询操纵非常简单。比如要查找int，顺着路径i -> in -> int就找到了。

搭建Trie的基本算法也很简单，无非是逐一把每则单词的每个字母插入Trie。插入前先看前缀是否存在。如果存在，就共享，否则创建对应的节点和边。比如要插入单词add，就有下面几步：

1. 考察前缀"a"，发现边a已经存在。于是顺着边a走到节点a。
2. 考察剩下的字符串"dd"的前缀"d"，发现从节点a出发，已经有边d存在。于是顺着边d走到节点ad
3. 考察最后一个字符"d"，这下从节点ad出发没有边d了，于是创建节点ad的子节点add，并把边ad->add标记为d。

问题实例

1、一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前**10**个词，请给出思想，给出时间复杂度分析

提示：用trie树统计每个词出现的次数，时间复杂度是 $O(n*le)$ （ le 表示单词的平均长度），然后是找出出现最频繁的前10个词。当然，也可以用堆来实现，时间复杂度是 $O(n*\lg 10)$ 。所以总的时间复杂度，是 $O(n*le)$ 与 $O(n*\lg 10)$ 中较大的哪一个。

2、寻找热门查询

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，但是如果去除重复和，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。

提示：利用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。

数据库

方法介绍

当遇到大数据量的增删改查时，一般把数据装进数据库中，从而利用数据的设计实现方法，对海量数据的增删改查进行处理。

倒排索引(Inverted index)

方法介绍

倒排索引是一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射，常被应用于搜索引擎和关键字查询的问题中。

以英文为例，下面是要被索引的文本：

```
T0 = "it is what it is"
```

```
T1 = "what is it"
```

```
T2 = "it is a banana"
```

我们就能得到下面的反向文件索引：

```
"a": {2}
```

```
"banana": {2}
```

```
"is": {0, 1, 2}
```

```
"it": {0, 1, 2}
```

```
"what": {0, 1}
```

检索的条件"what","is"和"it"将对应集合的交集。

正向索引开发出来用来存储每个文档的单词的列表。正向索引的查询往往满足每个文档有序频繁的全文查询和每个单词在校验文档中的验证这样的查询。在正向索引中，文档占据了中心的位置，每个文档指向了一个它所包含的索引项的序列。也就是说文档指向了它包含的那些单词，而反向索引则是单词指向了包含它的文档，很容易看到这个反向的关系。

问题实例

1、文档检索系统，查询那些文件包含了某单词，比如常见的学术论文的关键字搜索

提示：建倒排索引。

本章海量数据的习题

1 有100W个关键字，长度小于等于50字节。用高效的算法找出top10的热词，并对内存的占用不超过1MB。

提示：老题，与caopengcs讨论后，得出具体思路为：

- 先把100W个关键字hash映射到小文件，根据题意， $100W \ 50B = 50 \ 10^6B = 50M$ ，而内存只有1M，故干脆搞一个hash函数 $\% 50$ ，分解成50个小文件；
- 针对每个小文件依次运用hashmap(key, value)完成每个key的value次数统计，后用堆找出每个小文件中value次数最大的top 10；
-最后依次对每两小文件的top 10归并，得到最终的top 10。

此外，很多细节需要注意下，举个例子，如若hash映射后导致分布不均的话，有的小文件可能会超过1M，故为保险起见，你可能会说根据数据范围分解成50~500或更多的小文件，但到底是多少呢？我觉得这不重要，勿纠结答案，虽准备在平时，但关键还是看临场发挥，保持思路清晰关注细节即可。

2

单机5G内存，磁盘200T的数据，分别为字符串，然后给定一个字符串，判断这200T数据里面有没有这个字符串，怎么做？如果查询次数会非常的多，怎么预处理？

提示：如果数据是200g且允许少许误差的话，可以考虑用布隆过滤器 Bloom Filter。但本题是200T，得另寻良策，具体解法请读者继续思考。

3

现在有一个大文件，文件里面的每一行都有一个group标识（group很多，但是每个group的数据量很小），现在要求把这个大文件分成十个小文件，要求：

- 1、同一个group的必须在一个文件里面；

- 2、切分之后，要求十个小文件的数据量尽可能均衡。

7

服务器内存1G，有一个2G的文件，里面每行存着一个QQ号（5-10位数），怎么最快找出出现过最多次的QQ号。

8

尽量高效的统计一片英文文章（总单词数目）里出现的所有英文单词，按照在文章中首次出现的顺序打印输出该单词和它的出现次数。

9

在人人好友里，A和B是好友，B和C是好友，如果A 和C不是好友，那么C是A的二度好友，在一个有10万人的数据库里，如何在时间 $O(n)$ 里，找到某个人的十度好友。

12

海量记录，记录形式如下：TERMINOURLNOCOUNT urlno1 urlno2 ..., urlnon, 请问怎么考虑资源和时间这两个因素，实现快速查询任意两个记录的交集，并集等，设计相关的数据结构和算法。

14

有一亿个整数，请找出最大的1000个，要求时间越短越好，空间占用越少越好。

18

10亿个int型整数，如何找出重复出现的数字。

19

有2G的一个文本文档，文件每行存储的是一个句子，每个单词是用空格隔开的。问：输入一个句子，如何找到和它最相似的前10个句子。

提示：可用倒排文档。

20

某家视频网站，每天有上亿的视频被观看，现在公司要请研发人员找出最热门的视频。该问题的输入可以简化为一个字符串文件，每一行都表示一个视频id，然后要找出出现次数最多的前100个视频id，将其输出，同时输出该视频的出现次数。

- 1.假设每天的视频播放次数为3亿次，被观看的视频数量为一百万个，每个视频ID的长度为20字节，限定使用的内存为1G。请简述做法，再写代码。
- 2.假设每个月的视频播放次数为100亿次，被观看的视频数量为1亿，每个视频ID的长度为20字节，一台机器被限定使用的内存为1G。

提示：万变不离其宗，分而治之/Hash映射 + Hash统计 + 堆/快速/归并排序。

21

有一个log文件，里面记录的格式为：

QQ号	时间	flag
-----	----	------

123456	14: 00: 00	0
--------	------------	---

123457	14: 00: 01	1
--------	------------	---

其中flag=0表示登录 flag=1表示退出

问：统计一天平均在线的QQ数。

22

一个文本，一万行，每行一个词，统计出现频率最高的前10个词（词的平均长度为Len）。并分析时间复杂度。

23

在一个文件中有 10G 个整数，乱序排列，要求找出中位数。内存限制为 2G。只写出思路即可。

24

一个url指向的页面里面有另一个url,最终有一个url指向之前出现过的url或空，这两种情形都定义为null。这样构成一个单链表。给两条这样单链表，判断里面是否存在同样的url。url以亿级计，资源不足以hash。

25

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字节，内存限制大小是1M。返回频数最高的100个词。

26

1000万字符串，其中有些是重复的，需要把重复的全部去掉，保留没有重复的字符串。请怎么设计和实现？

27 有10个文件，每个文件1G，每个文件的每一行都存放的是用户的query，每个文件的query都可能重复。要你按照query的频度排序。

28

现有一200M的文本文件，里面记录着IP地址和对应地域信息，如

202.100.83.56 北京 北京大学

202.100.83.120 北京 人民大学

202.100.83.134 北京 中国青年政治学院

211.93.120.45 长春市 长春大学

211.93.120.129 吉林市 吉林大学

211.93.120.200 长春 长春KTV

现有6亿个IP地址，请编写程序，读取IP地址便转换成IP地址相对应的城市，要求有较好的时间复杂度和空间复杂度。

第七章 机器学习

K近邻算法

1.1、什么是K近邻算法

何谓K近邻算法，即K-Nearest Neighbor algorithm，简称KNN算法，单从名字来猜想，可以简单粗暴的认为是：K个最近的邻居，当K=1时，算法便成了最近邻算法，即寻找最近的那个邻居。为何要找邻居？打个比方来说，假设你来到一个陌生的村庄，现在你要找到与你有着相似特征的人群融入他们，所谓入伙。

用官方的话来说，所谓K近邻算法，即是给定一个训练数据集，对新的输入实例，在训练数据集中找到与该实例最邻近的K个实例（也就是上面所说的K个邻居），这K个实例的多数属于某个类，就把该输入实例分类到这个类中。根据这个说法，咱们来看下引自维基百科上的一幅图：



如上图所示，有两类不同的样本数据，分别用蓝色的小正方形和红色的小三角形表示，而图正中间的那个绿色的圆所标示的数据则是待分类的数据。也就是说，现在，我们不知道中间那个绿色的数据是从属于哪一类（蓝色小正方形or红色小三角形），下面，我们就要解决这个问题：给这个绿色的圆分类。

- 我们常说，物以类聚，人以群分，判别一个人是一个什么样品质特征的人，常常可以从他/她身边的朋友入手，所谓观其友，而识其人。我们不是要判别上图中那个绿色的圆是属于哪一类数据么，好说，从它的邻居下手。但一次性看多少个邻居呢？从上图中，你还能看到：
- 如果K=3，绿色圆点的最近的3个邻居是2个红色小三角形和1个蓝色小正方形，少数从属于多数，基于统计的方法，判定绿色的这个待分类点属于红色的三角形一类。如果K=5，绿色圆点的最近的5个邻居是2个红色三角形和3个蓝色的正方形，还是少数从属于多数，基于统计的方法，判定绿色的这个待分类点属于蓝色的正方形一类。

于此我们看到，当无法判定当前待分类点是从属于已知分类中的哪一类时，我们可以依据统计学的理论看它所处的位置特征，衡量它周围邻居的权重，而把它归为(或分配)到权重更大的那一类。这就是K近邻算法的核心思想。

1.2、近邻的距离度量表示法

上文第一节，我们看到，K近邻算法的核心在于找到实例点的邻居，这个时候，问题就接踵而至了，如何找到邻居，邻居的判定标准是什么，用什么来度量。这一系列问题便是下面要讲的距离度量表示法。但有的读者可能就有疑问了，我是要找邻居，找相似性，怎么又跟距离扯上关系了？

这是因为特征空间中两个实例点的距离和反应出两个实例点之间的相似性程度。K近邻模型的特征空间一般是n维实数向量空间，使用的距离可以使欧式距离，也可以是其它距离，既然扯到了距离，下面就来具体阐述下都有哪些距离度量的表示法，权当扩展。

1. 欧氏距离，最常见的两点之间或多点之间的距离表示法，又称之为欧几里得度量，它定义于欧几里得空间中，如点 $x = (x_1, \dots, x_n)$ 和 $y = (y_1, \dots, y_n)$ 之间的距离为：



(1)二维平面上两点 $a(x_1, y_1)$ 与 $b(x_2, y_2)$ 间的欧氏距离：



(2)三维空间两点 $a(x_1, y_1, z_1)$ 与 $b(x_2, y_2, z_2)$ 间的欧氏距离：



(3)两个n维向量 $a(x_{11}, x_{12}, \dots, x_{1n})$ 与 $b(x_{21}, x_{22}, \dots, x_{2n})$ 间的欧氏距离：



也可以用表示成向量运算的形式：



其上，二维平面上两点欧式距离，代码可以如下编写：


```
//unixfy: 计算欧氏距离

double euclideanDistance ( const vector< double >& v1, const vector< double
>& v2)
{
    assert(v1.size() == v2.size());

    double ret = 0.0 ;

    for ( vector < double >::size_type i = 0 ; i != v1.size(); ++i)
    {
        ret += (v1[i] - v2[i]) * (v1[i] - v2[i]);
    }

    return sqrt (ret);
}
```

2. 曼哈顿距离，我们可以定义曼哈顿距离的正式意义为L1-距离或城市区块距离，也就是在欧几里得空间的固定直角坐标系上两点所形成的线段对轴产生的投影的距离总和。例如在平面上，坐标（x1, y1）的点P1与坐标（x2, y2）的点P2的曼哈顿距离为：，要注意的是，曼哈顿距离依赖座标系统的转度，而非系统在座标轴上的平移或映射。

通俗来讲，想象你在曼哈顿要从一个十字路口开车到另外一个十字路口，驾驶距离是两点间的直线距离吗？显然不是，除非你能穿越大楼。而实际驾驶距离就是这个“曼哈顿距离”，此即曼哈顿距离名称的来源，同时，曼哈顿距离也称为城市街区距离(City Block distance)。

(1)二维平面两点a(x1,y1)与b(x2,y2)间的曼哈顿距离



(2)两个n维向量a(x11,x12,...,x1n)与 b(x21,x22,...,x2n)间的曼哈顿距离



3. 切比雪夫距离，若二个向量或二个点 p 、and q ，其座标分别为 (x_1, y_1) 及 (x_2, y_2) ，则两者之间的切比雪夫距离定义如下：
$$d_{\infty}(p, q) = \max(|x_2 - x_1|, |y_2 - y_1|)$$

这也等于以下 L_p 度量的极值：
$$d_{\infty}(p, q) = \lim_{p \rightarrow \infty} d_p(p, q)$$
，因此切比雪夫距离也称为 L_{∞} 度量。

以数学的观点来看，切比雪夫距离是由一致范数（uniform norm）（或称为上确界范数）所衍生的度量，也是超凸度量（injective metric space）的一种。

在平面几何中，若二点 p 及 q 的直角坐标系坐标为 (x_1, y_1) 及 (x_2, y_2) ，则切比雪夫距离为：
$$d_{\infty}(p, q) = \max(|x_2 - x_1|, |y_2 - y_1|)$$
。

玩过国际象棋的朋友或许知道，国王走一步能够移动到相邻的8个方格中的任意一个。那么国王从格子 (x_1, y_1) 走到格子 (x_2, y_2) 最少需要多少步？。你会发现最少步数总是 $\max(|x_2 - x_1|, |y_2 - y_1|)$ 步。有一种类似的一种距离度量方法叫切比雪夫距离。

(1) 二维平面两点 $a(x_1, y_1)$ 与 $b(x_2, y_2)$ 间的切比雪夫距离

$$d_{\infty}(a, b) = \max(|x_2 - x_1|, |y_2 - y_1|)$$

(2) 两个 n 维向量 $a(x_{11}, x_{12}, \dots, x_{1n})$ 与 $b(x_{21}, x_{22}, \dots, x_{2n})$ 间的切比雪夫距离

$$d_{\infty}(a, b) = \max_{1 \leq i \leq n} |x_{i1} - x_{i2}|$$

这个公式的另一种等价形式是
$$d_{\infty}(a, b) = \max_{1 \leq i \leq n} |x_{i1} - x_{i2}|$$

4. 闵可夫斯基距离(Minkowski Distance)，闵氏距离不是一种距离，而是一组距离的定义。

(1) 闵氏距离的定义

两个 n 维变量 $a(x_{11}, x_{12}, \dots, x_{1n})$ 与 $b(x_{21}, x_{22}, \dots, x_{2n})$ 间的闵可夫斯基距离定义为：

$$d_p(a, b) = \left(\sum_{i=1}^n |x_{i1} - x_{i2}|^p \right)^{1/p}$$

其中 p 是一个变参数。

当 $p=1$ 时，就是曼哈顿距离

当 $p=2$ 时，就是欧氏距离

当 $p \rightarrow \infty$ 时，就是切比雪夫距离

根据变参数的不同，闵氏距离可以表示一类的距离。

5. 标准化欧氏距离 (**Standardized Euclidean distance**)，标准化欧氏距离是针对简单欧氏距离的缺点而作的一种改进方案。标准欧氏距离的思路：既然数据各维分量的分布不一样，那先将各个分量都“标准化”到均值、方差相等。至于均值和方差标准化到多少，先复习点统计学知识。

假设样本集 X 的数学期望或均值(mean)为 m ，标准差(standard deviation, 方差开根)为 s ，那么 X 的“标准化变量” X^* 表示为： $(X-m)/s$ ，而且标准化变量的数学期望为0，方差为1。即，样本集的标准化过程(standardization)用公式描述就是：



标准化后的值 = (标准化前的值 - 分量的均值) / 分量的标准差

经过简单的推导就可以得到两个 n 维向量 $a(x_{11}, x_{12}, \dots, x_{1n})$ 与 $b(x_{21}, x_{22}, \dots, x_{2n})$ 间的标准化欧氏距离的公式：



如果将方差的倒数看成是一个权重，这个公式可以看成是一种加权欧氏距离(Weighted Euclidean distance)。

6. 马氏距离(Mahalanobis Distance)

(1) 马氏距离定义

有 M 个样本向量 $X_1 \sim X_M$ ，协方差矩阵记为 S ，均值记为向量 μ ，则其中样本向量 X 到 μ 的马氏距离表示为：



(协方差矩阵中每个元素是各个矢量元素之间的协方差 $Cov(X, Y)$ ，

$\text{Cov}(X,Y) = E\{ [X-E(X)] [Y-E(Y)] \}$, 其中E为数学期望)

而其中向量 X_i 与 X_j 之间的马氏距离定义为:



若协方差矩阵是单位矩阵(各个样本向量之间独立同分布),则公式就成了:



也就是欧氏距离了。

若协方差矩阵是对角矩阵,公式变成了标准化欧氏距离。

(2)马氏距离的优缺点:量纲无关,排除变量之间的相关性的干扰。

「微博上的seafood高清版点评道:原来马氏距离是根据协方差矩阵演变,一直被老师误导了,怪不得看Killian在05年NIPS发表的LMNN论文时候老是看到协方差矩阵和半正定,原来是这回事」

7. 巴氏距离 (**Bhattacharyya Distance**), 在统计中, Bhattacharyya距离测量两个离散或连续概率分布的相似性。它与衡量两个统计样品或种群之间的重叠量的Bhattacharyya系数密切相关。Bhattacharyya距离和Bhattacharyya系数以20世纪30年代曾在印度统计研究所工作的一个统计学家A. Bhattacharya命名。同时, Bhattacharyya系数可以被用来确定两个样本被认为相对接近的,它是用来测量中的类分类的可分离性。

(1) 巴氏距离的定义

对于离散概率分布 p 和 q 在同一域 X , 它被定义为:



其中:



是Bhattacharyya系数。

对于连续概率分布，Bhattacharyya系数被定义为：



在 \square 这两种情况下，巴氏距离 \square 并没有服从三角不等式.（值得一提的是，Hellinger距离不服从三角不等式 \square ）。

对于多变量的高斯分布 \square ， \square ，和是手段和协方差的分布 \square 。

需要注意的是，在这种情况下，第一项中的Bhattacharyya距离与马氏距离有关联。

（2）Bhattacharyya系数

Bhattacharyya系数是两个统计样本之间的重叠量的近似测量，可以被用于确定被考虑的两个样本的相对接近。

计算Bhattacharyya系数涉及集成的基本形式的两个样本的重叠的时间间隔的值的两个样本被分裂成一个选定的分区数，并且在每个分区中的每个样品的成员的数量，在下面的公式中使用



考虑样品a和b，n是分区数，并且 \square ， \square 被一个和b i的分区中的样本数量的成员。更多介绍请参看：

http://en.wikipedia.org/wiki/Bhattacharyya_coefficient。

8. 汉明距离(Hamming distance)，两个等长字符串s1与s2之间的汉明距离定义为将其中一个变为另外一个所需要作的最小替换次数。例如字符串“1111”与“1001”之间的汉明距离为2。应用：信息编码（为了增强容错性，应使得编码间的最小汉明距离尽可能大）。或许，你还没明白我再说什么，不急，看下上篇blog中第78题的第3小题整理的一道面试题目，便一目了然了。如下图所示：



//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）。

与此同时，面试官还可以继续问下去：那么，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>，及这里关于simhash算法的介绍：

<http://www.cnblogs.com/linecong/archive/2010/08/28/simhash.html>），接下来，便引出了下文关于夹角余弦的讨论。（[上篇blog](#)中第78题的第3小题给出了多种方法，读者可以参看之。同时，程序员编程艺术系列第二十八章将详细阐述这个问题）

9. 夹角余弦(Cosine)，几何中夹角余弦可用来衡量两个向量方向的差异，机器学习中借用这一概念来衡量样本向量之间的差异。

(1)在二维空间中向量A(x1,y1)与向量B(x2,y2)的夹角余弦公式：



(2) 两个n维样本点a(x11,x12,...,x1n)和b(x21,x22,...,x2n)的夹角余弦



类似的，对于两个n维样本点a(x11,x12,...,x1n)和b(x21,x22,...,x2n)，可以使用类似于夹角余弦的概念来衡量它们间的相似程度，即：



夹角余弦取值范围为[-1,1]。夹角余弦越大表示两个向量的夹角越小，夹角余弦越小表示两向量的夹角越大。当两个向量的方向重合时夹角余弦取最大值1，当两个向量的方向完全相反夹角余弦取最小值-1。

10. 杰卡德相似系数(Jaccard similarity coefficient)

(1) 杰卡德相似系数

两个集合A和B的交集元素在A，B的并集中所占的比例，称为两个集合的杰卡德相似系数，用符号 $J(A,B)$ 表示。



杰卡德相似系数是衡量两个集合的相似度一种指标。

(2) 杰卡德距离

与杰卡德相似系数相反的概念是杰卡德距离(Jaccard distance)。

杰卡德距离可用如下公式表示：



杰卡德距离用两个集合中不同元素占有所有元素的比例来衡量两个集合的区分度。

(3) 杰卡德相似系数与杰卡德距离的应用

可将杰卡德相似系数用在衡量样本的相似度上。

举例：样本A与样本B是两个n维向量，而且所有维度的取值都是0或1，例如：A(0111)和B(1011)。我们将样本看成是一个集合，1表示集合包含该元素，0表示集合不包含该元素。

M11：样本A与B都是1的维度的个数

M01：样本A是0，样本B是1的维度的个数

M10：样本A是1，样本B是0 的维度的个数

M00：样本A与B都是0的维度的个数

依据上文给的杰卡德相似系数及杰卡德距离的相关定义，样本A与B的杰卡德相似系数J可以表示为：



这里 $M_{11}+M_{01}+M_{10}$ 可理解为A与B的并集的元素个数，而 M_{11} 是A与B的交集的元素个数。而样本A与B的杰卡德距离表示为J'：



11. 皮尔逊系数(Pearson Correlation Coefficient)

在具体阐述皮尔逊相关系数之前，有必要解释下什么是相关系数 (Correlation coefficient)与相关距离(Correlation distance)。

相关系数 (Correlation coefficient)的定义是：



(其中，E为数学期望或均值，D为方差，D开根号为标准差， $E\{ [X-E(X)] [Y-E(Y)] \}$ 称为随机变量X与Y的协方差，记为 $Cov(X,Y)$ ，即 $Cov(X,Y) = E\{ [X-E(X)] [Y-E(Y)] \}$ ，而两个变量之间的协方差和标准差的商则称为随机变量X与Y的相关系数，记为 ☐)

相关系数衡量随机变量X与Y相关程度的一种方法，相关系数的取值范围是 $[-1,1]$ 。相关系数的绝对值越大，则表明X与Y相关度越高。当X与Y线性相关时，相关系数取值为1（正线性相关）或-1（负线性相关）。

具体的，如果有两个变量：X、Y，最终计算出的相关系数的含义可以有如下理解：

当相关系数为0时，X和Y两变量无关系。

当X的值增大（减小），Y值增大（减小），两个变量为正相关，相关系数在0.00与1.00之间。

当X的值增大（减小），Y值减小（增大），两个变量为负相关，相关系数在-1.00与0.00之间。

相关距离的定义是：



OK，接下来，咱们来重点了解下皮尔逊相关系数。

在统计学中，皮尔逊积矩相关系数（英语：Pearson product-moment correlation coefficient，又称作 PPMCC或PCCs, 用 r 表示）用于度量两个变量 X 和 Y 之间的相关（线性相关），其值介于-1与1之间。

通常情况下通过以下取值范围判断变量的相关强度：

相关系数

0.8-1.0 极强相关

0.6-0.8 强相关

0.4-0.6 中等程度相关

0.2-0.4 弱相关

0.0-0.2 极弱相关或无相关

在自然科学领域中，该系数广泛用于度量两个变量之间的相关程度。它是由卡尔·皮尔逊从弗朗西斯·高尔顿在19世纪80年代提出的一个相似却又稍有不同的想法演变而来的。这个相关系数也称作“皮尔森相关系数 r ”。

(1)皮尔逊系数的定义：

两个变量之间的皮尔逊相关系数定义为两个变量之间的协方差和标准差的商：



以上方程定义了总体相关系数, 一般表示成希腊字母 $\rho(\text{rho})$ 。基于样本对协方差和方差进行估计, 可以得到样本标准差, 一般表示成 r :



一种等价表达式的是表示成标准分的均值。基于 (X_i, Y_i) 的样本点, 样本皮尔逊系数是

□

其中□、□及□，分别是标准分、样本平均值和样本标准差。

或许上面的讲解令你头脑混乱不堪，没关系，我换一种方式讲解，如下：

假设有两个变量X、Y，那么两变量间的皮尔逊相关系数可通过以下公式计算：

- 公式一：□

注：勿忘了上面说过，“皮尔逊相关系数定义为两个变量之间的协方差和标准差的商”，其中标准差的计算公式为：□

- 公式二：□
- 公式三：□
- 公式四：□

以上列出的四个公式等价，其中E是数学期望，cov表示协方差，N表示变量取值的个数。

(2)皮尔逊相关系数的适用范围

当两个变量的标准差都不为零时，相关系数才有定义，皮尔逊相关系数适用于：

1. 两个变量之间是线性关系，都是连续数据。
2. 两个变量的总体是正态分布，或接近正态的单峰分布。
3. 两个变量的观测值是成对的，每对观测值之间相互独立。

(3)如何理解皮尔逊相关系数

rubyist：皮尔逊相关系数理解有两个角度

其一，按照高中数学水平来理解，它很简单，可以看做将两组数据首先做Z分数处理之后，然后两组数据的乘积和除以样本数，Z分数一般代表正态

分布中, 数据偏离中心点的距离.等于变量减掉平均数再除以标准差.(就是高考的标准分类似的处理)

样本标准差则等于变量减掉平均数的平方和, 再除以样本数, 最后再开方, 也就是说, 方差开方即为标准差, 样本标准差计算公式为:



所以, 根据这个最朴素的理解,我们可以将公式依次精简为:



其二, 按照大学的线性数学水平来理解, 它比较复杂一点,可以看做是两组数据的向量夹角的余弦。下面是关于此皮尔逊系数的几何学的解释, 先来看一幅图, 如下所示:



回归直线: $y=gx(x)$ [红色] 和 $x=gy(y)$ [蓝色]

如上图, 对于没有中心化的数据, 相关系数与两条可能的回归线 $y=gx(x)$ 和 $x=gy(y)$ 夹角的余弦值一致。对于没有中心化的数据 (也就是说, 数据移动一个样本平均值以使其均值为0), 相关系数也可以被视作由两个随机变量 向量 夹角的 余弦值 (见下方)。

举个例子, 例如, 有5个国家的国民生产总值分别为 10, 20, 30, 50 和 80 亿美元。假设这5个国家 (顺序相同) 的贫困百分比分别为 11%, 12%, 13%, 15%, and 18% 。令 x 和 y 分别为包含上述5个数据的向量: $x = (1, 2, 3, 5, 8)$ 和 $y = (0.11, 0.12, 0.13, 0.15, 0.18)$ 。

利用通常的方法计算两个向量之间的夹角 (参见 数量积), 未中心化的相关系数是:



我们发现以上的数据特意选定为完全相关: $y = 0.10 + 0.01 x$ 。于是, 皮尔逊相关系数应该等于1。将数据中心化 (通过 $E(x) = 3.8$ 移动 x 和通过 $E(y) = 0.138$ 移动 y) 得到 $x = (-2.8, -1.8, -0.8, 1.2, 4.2)$ 和 $y = (-0.028, -0.018, -0.008, 0.012, 0.042)$, 从中



(4)皮尔逊相关的约束条件

从以上解释,也可以理解皮尔逊相关的约束条件:

1. 两个变量间有线性关系
2. 变量是连续变量
3. 变量均符合正态分布,且二元分布也符合正态分布
4. 两变量独立

在实践统计中,一般只输出两个系数,一个是相关系数,也就是计算出来的相关系数大小,在-1到1之间;另一个是独立样本检验系数,用来检验样本一致性。

简单说来,各种“距离”的应用场景简单概括为,空间:欧氏距离,路径:曼哈顿距离,国际象棋国王:切比雪夫距离,以上三种的统一形式:闵可夫斯基距离,加权:标准化欧氏距离,排除量纲和依存:马氏距离,向量差距:夹角余弦,编码差别:汉明距离,集合近似度:杰卡德类似系数与距离,相关:相关系数与相关距离。

1.3、K值的选择

除了上述1.2节如何定义邻居的问题之外,还有一个选择多少个邻居,即K值定义为多大的问题。不要小看了这个K值选择问题,因为它对K近邻算法的结果会产生重大影响。如李航博士的一书「统计学习方法」上所说:

1. 如果选择较小的K值,就相当于用较小的领域中的训练实例进行预测,“学习”近似误差会减小,只有与输入实例较近或相似的训练实例才会对预测结果起作用,与此同时带来的问题是“学习”的估计误差会增大,换句话说,K值的减小就意味着整体模型变得复杂,容易发生`过拟合`;
2. 如果选择较大的K值,就相当于用较大领域中的训练实例进行预测,其优点是可以减少学习的估计误差,但缺点是学习的近似误差会增大。这时候,与输入实例较远(不相似的)训练实例也会对预测器作用,使预测发生错误,且K值的增大就意味着整体的模型变

得简单。

3. $K=N$ ，则完全不足取，因为此时无论输入实例是什么，都只是简单的预测它属于在训练实例中最多的类，模型过于简单，忽略了训练实例中大量有用信息。

在实际应用中， K 值一般取一个比较小的数值，例如采用 交叉验证法（简单来说，就是一部分样本做训练集，一部分做测试集）来选择最优的 K 值。

支持向量机

第一层、了解SVM

支持向量机，因其英文名为support vector machine，故一般简称SVM，通俗来讲，它是一种二类分类模型，其基本模型定义为特征空间上的间隔最大的线性分类器，其学习策略便是间隔最大化，最终可转化为一个凸二次规划问题的求解。

1.1、线性分类

理解SVM，咱们必须先弄清楚一个概念：线性分类器。

1.1.1、分类标准

考虑一个二类的分类问题，数据点用 x 来表示，类别用 y 来表示，可以取1或者-1，分别代表两个不同的类，且是一个 n 维向量， w^T 中的 T 代表转置。一个线性分类器的学习目标就是要在 n 维的数据空间中找到一个分类 超平面，其方程可以表示为：



可能有读者对类别取1或-1有疑问，事实上，这个1或-1的分类标准起源于logistic回归，为了过渡的自然性，咱们就再来看看这个logistic回归。

1.1.2、1或-1分类标准的起源：logistic回归

Logistic回归目的是从特征学习出一个0/1分类模型，而这个模型是将特性的线性组合作为自变量，由于自变量的取值范围是负无穷到正无穷。因此，使用logistic函数（或称作sigmoid函数）将自变量映射到(0,1)上，映射后的值被认为是属于 $y=1$ 的概率。

假设函数



其中 x 是 n 维特征向量，函数 g 就是logistic函数。

而 σ 的图像是



可以看到，通过logistic函数将自变量从无穷映射到了 $(0,1)$ ，而假设函数就是特征属于 $y=1$ 的概率。



从而，当我们要判别一个新来的特征属于哪个类时，只需求 σ ，若大于0.5就是 $y=1$ 的类，反之属于 $y=0$ 类。

再审视一下 σ ，发现 σ 只和 z 有关， $z > 0$ ，那么 $\sigma > 0.5$ ， $g(z)$ 只不用来映射，真实的类别决定权还在 z 。还有当 $z \gg 0$ 时， $\sigma = 1$ ，反之 $\sigma = 0$ 。如果我们只从 z 出发，希望模型达到的目标无非就是让训练数据中 $y=1$ 的特征 $z \gg 0$ ，而是 $y=0$ 的特征 $z \ll 0$ 。Logistic回归就是要学习得到 w ，使得正例的特征远大于0，负例的特征远小于0，强调在全部训练实例上达到这个目标。

1.1.3、形式化标示

- 我们这次使用的结果标签是 $y=-1, y=1$ ，替换在logistic回归中使用的 $y=0$ 和 $y=1$ 。
- 同时将 σ 替换成 w 和 b 。
 - 以前的 σ ，其中认为 z ，现在我们替换为 b ；
 - 后面的 σ 替换为 w （即 z ）。

这样，我们让 y ，进一步 z 。也就是说除了 y 由 $y=0$ 变为 $y=-1$ ，只是标不同外，与logistic回归的形式化表示没区别。

再明确下假设函数



上面提到过我们只需考虑的 z 正负问题，而不用关心 $g(z)$ ，因此我们这里将 $g(z)$ 做一个简化，将其简单映射到 $y=-1$ 和 $y=1$ 上。映射关系如下：



于此，想必已经解释明白了为何线性分类的标准一般用1 或者-1 来标示。

1.2、线性分类的一个例子

假定现在有一个二维平面，如下图所示，平面上有两种不同的点，分别用两种不同的颜色表示，一种为红颜色的点，另一种为蓝颜色的点，如果我们要在这个二维平面上找到一个可行的超平面的话，那么这个超平面可以是下图中那根红颜色的线(在二维空间中，超平面就是一条直线)。



从上图中我们可以看出，这条红颜色的线作为一个超平面，把红颜色的点和蓝颜色的点分开来了，在超平面一边的数据点所对应的y全是 -1，而在另一边全是1。

接着，我们可以令分类函数：



显然，如果 $f(x)=0$ ，那么x是位于超平面上的点。我们不妨要求对于所有满足 $f(x)<0$ 的点，其对应的 $y=-1$ ，而 $f(x)>0$ 则对应 $y=1$ 的数据点。



注：上图中，定义特征到结果的输出函数 $yf(x)$ ，与我们之前定义的 $yf(x)$ 是一样的。为什么？因为无论是 $yf(x)$ 还是 $yf(x)$ ，不影响最终优化结果。下文你将看到，当我们转化到优化 $yf(x)$ 的时候，为了求解方便，会把 $yf(x)$ 令为 1，即 $yf(x)$ 是 $y(w^T x + b)$ ，还是 $y(w^T x - b)$ ，对我们要优化的式子 $\max 1/\|w\|$ 已无影响。

从而在我们进行分类的时候，将数据点 x 代入 $f(x)$ 中，如果得到的结果小于 0，则赋予其类别 -1，如果大于 0 则赋予类别 1。如果 $f(x)=0$ ，则很难办了，分到哪一类都不是。

此外，有些时候，或者说大部分时候数据并不是线性可分的，这时满足这样条件的超平面可能就根本不存在，这里咱们先从最简单的情形开始推导，就假设数据都是线性可分的，亦即这样的超平面是存在的。

1.3、函数间隔 *Functional margin* 与几何间隔 *Geometrical margin*

一般而言，一个点距离超平面的远近可以表示为分类预测的确信或准确程度。

- 在超平面 $w \cdot x + b = 0$ 确定的情况下， $|w \cdot x + b|$ 能够相对地表示点 x 到距离超平面的远近，而 $w \cdot x + b$ 的符号与类标记 y 的符号是否一致表示分类是否正确，所以，可以用量 $y \cdot (w \cdot x + b)$ 的正负性来判定或表示分类的正确性和确信度。

于此，我们便引出了定义样本到分类间隔距离的函数间隔 *functional margin* 的概念。

1.3.1、函数间隔 *Functional margin*

我们定义函数间隔 *functional margin* 为：



接着，我们定义超平面 (w, b) 关于训练数据集 T 的函数间隔为超平面 (w, b) 关于 T 中所有样本点 (x_i, y_i) 的函数间隔最小值，其中， x 是特征， y 是结果标签， i 表示第 i 个样本，有：

$$\gamma = \min_i y_i (w \cdot x_i + b) \quad (i=1, \dots, n)$$

然与此同时，问题就出来了：上述定义的函数间隔虽然可以表示分类预测的正确性和确信度，但在选择分类超平面时，只有函数间隔还远远不够，因为如果成比例的改变 w 和 b ，如将他们改变为 $2w$ 和 $2b$ ，虽然此时超平面没有改变，但函数间隔的值 $f(x)$ 却变成了原来的2倍。

事实上，我们可以对法向量 w 加些约束条件，使其表面上看起来规范化，如此，我们很快又将引出真正定义点到超平面的距离--几何间隔 *geometrical margin* 的概念（很快你将看到，几何间隔就是函数间隔除以

个 $\|w\|$ ，即 $yf(x) / \|w\|$ ）。

1.3.2、点到超平面的距离定义：几何间隔 **Geometrical margin**

对于一个点 x ，令其垂直投影到超平面上的对应的为 x_0 ， w 是垂直于超平面的一个向量， \square 为样本 x 到分类间隔的距离，

\square

我们有

\square

其中， $\|w\|$ 表示的是范数。

又由于 x_0 是超平面上的点，满足 $f(x_0)=0$ ，代入超平面的方程即可算出：

\square

不过这里的 \square 是带符号的，我们需要的只是它的绝对值，因此类似地，也乘上对应的类别 y 即可，因此实际上我们定义 几何间隔 **geometrical margin** 为(注：别忘了，上面 \square 的定义， $\square = y(w^T x + b) = yf(x)$)：

\square

代入相关式子可以得出： $y_i(w/\|w\| + b/\|w\|)$ 。

综上，函数间隔 $y(w^T x + b) = y f(x)$ 实际上就是 $|f(x)|$ ，只是人为定义的一个间隔度量；而几何间隔 $|f(x)|/\|w\|$ 才是直观上的点到超平面距离。

1.4、最大间隔分类器 **Maximum Margin Classifier** 的定义

由上，我们已经知道，函数间隔 **functional margin** 和 几何间隔 **geometrical margin** 相差一个的缩放因子。按照我们前面的分析，对一个数据点进行分类，当它的 **margin** 越大的时候，分类的 **confidence** 越大。对于一个包含 n 个点的数据集，我们可以很自然地定义它的 **margin**

为所有这 n 个点的 margin 值中最小的那个。于是，为了使得分类的 confidence 高，我们希望所选择的超平面 hyper plane 能够最大化这个 margin 值。

\square

且

1. functional margin 明显是不太适合用来最大化一个量，因为在 hyper plane 固定以后，我们可以等比例地缩放 w 的长度和 b 的值，这样可以使得 \square 的值任意大，亦即 functional margin 可以在 hyper plane 保持不变的情况下被取得任意大，
2. 而 $\text{geometrical margin}$ 则没有这个问题，因为除上了 \square 这个分母，所以缩放 w 和 b 的时候 \square 的值是不会改变的，它只随着 hyper plane 的变动而变动，因此，这是更加合适的一个 margin 。

这样一来，我们的 $\text{maximum margin classifier}$ 的目标函数可以定义为：

\square

当然，还需要满足一定的约束条件：

\square

其中 \square (等价于 $\square = \square / \|w\|$ ，故有稍后的 $\square = 1$ 时， $\square = 1 / \|w\|$)，处于推导和优化的目的，我们可以令 $\square = 1$ (对目标函数的优化没有影响)，此时，上述的目标函数 \square 转化为：

\square

其中， s.t. ，即 subject to 的意思，它导出的是约束条件。

通过求解这个问题，我们就可以找到一个 margin 最大的 classifier ，通过最大化 margin ，我们使得该分类器对数据进行分类时具有了最大的 confidence ，从而设计决策最优分类超平面。

如下图所示，中间的红色线条是 Optimal Hyper Plane，另外两条线到红线的距离都是等于 $\frac{1}{\|w\|}$ 的($\frac{1}{\|w\|}$ 便是上文所定义的geometrical margin，当令 $\|w\|=1$ 时， $\frac{1}{\|w\|}$ 便为 $1/\|w\|$ ，而我们上面得到的目标函数便是在相应的约束条件下，要最大化这个 $1/\|w\|$ 值)：



1.5、到底什么是 *Support Vector*

通过上节1.4节最后一张图：



我们可以看到两个支撑着中间的 gap 的超平面，到中间的纯红线 separating hyper plane 的距离相等，即我们所能得到的最大的 geometrical margin，而“支撑”这两个超平面的必定会有一些点，而这些“支撑”的点便叫做支持向量Support Vector。

换言之，Support Vector便是下图中那蓝色虚线和粉红色虚线上的点：



很显然，由于这些 supporting vector 刚好在边界上，所以它们满足 $\|w \cdot x + b\| = 1$ ，而对于所有不是支持向量的点，也就是在“阵地后方”的点，则显然有 $\|w \cdot x + b\| < 1$ 。

第二层、深入SVM

2.1、从线性可分到线性不可分

2.1.1、从原始问题到对偶问题的求解

根据我们之前得到的目标函数（subject to导出的则是约束条件）：



由于求 \square 的最大值相当于求 \square 的最小值，所以上述目标函数等价于：



- 这样，我们的问题成为了一个凸优化问题，因为现在的目标函数是二次的，约束条件是线性的，所以它是一个凸二次规划问题。这个问题可以用任何现成的 [QP \(Quadratic Programming\)](#) 的优化包进行求解，一言以蔽之：在一定的约束条件下，目标最优，损失最小。
- 进一步，虽然这个问题确实是一个标准的 QP 问题，但由于它的特殊结构，我们可以通过 [Lagrange Duality](#) 变换到对偶变量 (dual variable) 的优化问题，这样便可以找到一种更加有效的方法来进行求解，而且通常情况下这种方法比直接使用通用的 QP 优化包进行优化要高效得多。

换言之，除了用解决QP问题的常规方法之外，还可以通过求解对偶问题得到最优解，这就是线性可分条件下支持向量机的对偶算法，这样做的优点在于：一者对偶问题往往更容易求解；二者可以自然的引入核函数，进而推广到非线性分类问题。

那什么是Lagrange duality？简单地来说，通过给每一个约束条件加上一个 Lagrange multiplier(拉格朗日乘值)，即引入拉格朗日乘子 \square ，如此我们便可以通过拉格朗日函数将约束条件融和到目标函数里去(也就是说把条件融合到一个函数里头，现在只用一个函数表达式便能清楚的表达出我们的问题)：



然后我们令

λ_i

容易验证：

- 当某个约束条件不满足时，例如 $\lambda_i < 0$ ，那么我们显然有 $\lambda_i x_i < 0$ （只要令 $x_i = 0$ 即可）。
- 而当所有约束条件都满足时，则有 $\lambda_i \geq 0$ ，亦即我们最初要最小化的量 L 。

因此，在要求约束条件得到满足的情况下最小化 L ，实际上等价于直接最小化 L （当然，这里也有约束条件，就是 $\lambda_i \geq 0, i=1, \dots, n$ ），因为如果约束条件没有得到满足， L 会等于无穷大，自然不会是我们所要求的最小值。

具体写出来，我们现在的目标函数变成了：

$$\min_{\lambda} L(\lambda)$$

这里用 L^* 表示这个问题的最优值，这个问题和我们最初的问题是等价的。不过，现在我们来把最小和最大的位置交换一下：

$$\max_{\lambda} L(\lambda)$$

当然，交换以后的问题不再等价于原问题，这个新问题的最优值用 L^* 来表示。并且，我们有 $L^* \leq L$ ，这在直观上也不难理解，最大值中最小的一个总也比最小值中最大的一个要大。总之，第二个问题的最优值 L^* 在这里提供了一个第一个问题的最优值 L 的一个下界，在满足某些条件的情况下，这两者相等，这个时候我们就可以通过求解第二个问题来间接地求解第一个问题。

也就是说，下面我们可以先求 L 对 w, b 的极小，再求 L 对 λ 的极大。而且，之所以从 minmax 的原始问题 L ，转化为 maxmin 的对偶问题 L^* ，一方面因为 L^* 是 L 的近似解，二者，转化为对偶问题后，更容易求解。

2.1.2、KKT条件

与此同时，上段说“在满足某些条件的情况下”，这所谓的“满足某些条件”就是要满足KKT条件。那KKT条件的表现形式是什么呢？

据维基百科：[KKT 条件](#)的介绍，一般地，一个最优化数学模型能够表示成下列标准形式：



其中， $f(x)$ 是需要最小化的函数， $h(x)$ 是等式约束， $g(x)$ 是不等式约束， p 和 q 分别为等式约束和不等式约束的数量。同时，我们得明白以下两个定理：

- 凸优化的概念： C 为一凸集， f 为一凸函数。凸优化就是要找出点 x^* ，使得每一 $x \in C$ 满足 $f(x) \geq f(x^*)$ 。
- KKT条件的意义：它是一个非线性规划（Nonlinear Programming）问题能有最优化解法的必要和充分条件。

而KKT条件就是指上面最优化数学模型的标准形式中的最小点 x^* 必须满足下面的条件：



经过论证，我们这里的问题是满足 KKT 条件的（首先已经满足Slater condition，再者 f 和 g_i 也都是可微的，即 L 对 w 和 b 都可导），因此现在我们便转化为求解第二个问题。

也就是说，现在，咱们的原问题通过满足一定的条件，已经转化成了对偶问题。而求解这个对偶学习问题，分为3个步骤，首先要让 $L(w, b, \alpha)$ 关于 w 和 b 最小化，然后求对 α 的极大，最后利用SMO算法求解对偶因子。

2.1.3、对偶问题求解的3个步骤

1）、首先固定 α ，要让 L 关于 w 和 b 最小化，我们分别对 w ， b 求偏导数，即令 $\partial L / \partial w$ 和 $\partial L / \partial b$ 等于零（对 w 求导结果的解释请看本文评论下第45楼回复）：



以上结果代回上述的 L:



得到:



提醒: 有读者可能会问上述推导过程如何而来? 说实话, 其具体推导过程是比较复杂的, 如下图所示:



最后, 得到:



如 jerrylead 所说: “倒数第4步”推导到“倒数第3步”使用了线性代数的转置运算, 由于 a_i 和 y_i 都是实数, 因此转置后与自身一样。“倒数第3步”推导到“倒数第2步”使用了 $(a+b+c+...)(a+b+c+...)=aa+ab+ac+ba+bb+bc+...$ 的乘法运算法则。最后一步是上一步的顺序调整。

L(从上面的最后一个式子, 我们可以看出, 此时的拉格朗日函数只包含了一个变量, 那就是 α , 然后下文的第2步, 求出了 α 便能求出 w , 和 b , 由此可见, 上文第1.2节提出来的核心问题: 分类函数 α 也就可以轻而易举的求出来了。

2)、求对 α 的极大, 即是关于对偶问题的最优化问题, 从上面的式子得到:

(不得不提醒下读者: 经过上面第一个步骤的求 w 和 b , 得到的拉格朗日函数式子已经没有了变量 w , b , 只有 α , 而反过来, 求得的将能导出 w , b 的解, 最终得出分离超平面和分类决策函数。为何呢? 因为如果求出了 α , 根据 α , 即可求出 w 。然后通过 α , 即可求出 b)



3)、如前面所说, 这个问题有更加高效的优化算法, 即我们常说的 SMO 算法。

2.1.4、序列最小最优化SMO算法

细心的读者读至上节末尾处，怎么拉格朗日乘子 α 的值可能依然心存疑惑。实际上，关于 α 的求解可以用一种快速学习算法即SMO算法，这里先简要介绍下。

OK，当：

$$\alpha$$

要解决的是在参数 α 上求最大值 W 的问题，至于 α 和 α 都是已知数（中是一个参数，用于控制目标函数中两项（“寻找 margin 最大的超平面”和“保证数据点偏差量最小”）之间的权重。和上文最后的式子对比一下，可以看到唯一的区别就是现在 dual variable α 多了一个上限 C ，关于 C 的具体由来请查看下文第2.3节）。

要了解这个SMO算法是如何推导的，请跳到下文第3.5节、SMO算法。

到目前为止，我们的 SVM 还比较弱，只能处理线性的情况，下面我们将引入核函数，进而推广到非线性分类问题。

2.2、核函数Kernel

2.2.1、特征空间的隐式映射：核函数

在线性不可分的情况下，支持向量机通过某种事先选择的非线性映射(核函数)将输入变量映射到一个高维特征空间，在这个空间中构造最优分类超平面。我们使用SVM进行数据集分类工作的过程首先是同预先选定的一些非线性映射将输入空间映射到高维特征空间(下图很清晰的表达了通过映射到高维特征空间，而把平面上本身不好分的非线性数据分开了来)：

$$\alpha$$

使得在高维属性空间中有可能最训练数据实现超平面的分割，避免了在原输入空间中进行非线性曲面分割计算，且在处理高维输入空间的分类时，这种方法尤其有效，其工作原理如下图所示：



而在我们遇到核函数之前，如果用原始的方法，那么在用线性学习器学习一个非线性关系，需要选择一个非线性特征集，并且将数据写成新的表达形式，这等价于应用一个固定的非线性映射，将数据映射到特征空间，在特征空间中使用线性学习器，因此，考虑的假设集是这种类型的函数：



这里 $\phi: X \rightarrow F$ 是从输入空间到某个特征空间的映射，这意味着建立非线性学习器分为两步：

1. 首先使用一个非线性映射将数据变换到一个特征空间F，
2. 然后在特征空间使用线性学习器分类。

这意味着假设可以表达为训练点的线性组合，因此决策规则可以用测试点和训练点的内积来表示：



如果有一种方式可以在特征空间中直接计算内积 $\langle \phi(x_i) \cdot \phi(x) \rangle$ ，就像在原始输入点的函数中一样，就有可能将两个步骤融合到一起建立一个非线性的学习器，这样直接计算的方法称为核函数方法，于是，核函数便横空出世了。

定义：核是一个函数K，对所有 $x, z \in X$ ，满足 $K(x, z) = \langle \phi(x) \cdot \phi(z) \rangle$ ，这里 ϕ 是从X到内积特征空间F的映射。

2.2.2、核函数：如何处理非线性数据

我们已经知道，如果是线性方法，所以对非线性的数据就没有办法处理。举个例子来说，则是如下图所示的两类数据，分别分布为两个圆圈的形状，这样的数据本身就是线性不可分的，此时咱们该如何把这两类数据分开呢？



此时，一个理想的分界应该是一个“圆圈”而不是一条线（超平面）。如

果用 X_1 和 X_2 来表示这个二维平面的两个坐标的话，我们知道一条二次曲线（圆圈是二次曲线的一种特殊情况）的方程可以写作这样的形式：



如果我们构造另外一个五维的空间，其中五个坐标的值分别为 $Z_1=X_1$, $Z_2=X_1^2$, $Z_3=X_2$, $Z_4=X_2^2$, $Z_5=X_1X_2$ ，那么显然，上面的方程在新的坐标系下可以写作：



关于新的坐标 Z ，这正是一个 hyper plane 的方程！也就是说，如果我们做一个映射 $\phi: \mathbb{R}^2 \rightarrow \mathbb{R}^5$ ，将 X 按照上面的规则映射为 Z ，那么在新的空间中原来的数据将变成线性可分的，从而使用之前我们推导的线性分类算法就可以进行处理了。这正是 Kernel 方法处理非线性问题的基本思想。

再进一步描述 Kernel 的细节之前，不妨再来看看这个例子映射过后的直观例子。具体来说，我这里的超平面实际的方程是这个样子（圆心在 X_2 轴上的一个正圆）：



因此我只需要把它映射到 $Z_1=X_1$, $Z_2=X_1^2$, $Z_3=X_2$ 这样一个三维空间中即可，下图即是映射之后的结果，将坐标轴经过适当的旋转，就可以很明显地看出，数据是可以通过一个平面来分开的：



回忆一下，我们上一次2.1节中得到的最终的分类函数是这样的：



映射过后的空间是：



而其中的 α 也是通过求解如下 dual 问题而得到的：



这样一来问题就解决了吗？其实稍想一下就会发现有问题：在最初的例子里，我们对一个二维空间做映射，选择的新空间是原始空间的所有一阶和二阶的组合，得到了五个维度；如果原始空间是三维，那么我们会得到 19 维的新空间，这个数目是呈爆炸性增长的，这给 $\phi(\cdot)$ 的计算带来了非常大的困难，而且如果遇到无穷维的情况，就根本无从计算了。所以需要 Kernel 出马了。

还是从最开始的简单例子出发，设两个向量 \square 和 \square ，而 $\phi(\cdot)$ 即是到前面 2.2.1 节说的五维空间的映射，因此映射过后的内积为：



（公式说明：上面的这两个推导过程中，所说的前面的五维空间的映射，这里说的前面便是文中 2.2.1 节的所述的映射方式，仔细看下 2.2.1 节的映射规则，再看那第一个推导，其实就是计算 x_1, x_2 各自的内积，然后相乘相加即可，第二个推导则是直接平方，去掉括号，也很容易推出来）

另外，我们又注意到：



二者有很多相似的地方，实际上，我们只要把某几个维度线性缩放一下，然后再加上一个常数维度，具体来说，上面这个式子的计算结果实际上和映射



之后的内积 \square 的结果是相等的，那么区别在于什么地方呢？

一个是映射到高维空间中，然后再根据内积的公式进行计算；而另一个则直接在原来的低维空间中进行计算，而不需要显式地写出映射后的结果。（公式说明：上面之中，最后的两个式子，第一个算式，是带内积的完全平方式，可以拆开，然后，通过凑一个得到，第二个算式，也是根据第一个算式凑出来的）

回忆刚才提到的映射的维度爆炸，在前一种方法已经无法计算的情况

下，后一种方法却依旧能从容处理，甚至是无穷维度的情况也没有问题。

我们把这里的计算两个向量在隐式映射过后的空间中的内积的函数叫做核函数 (Kernel Function)，例如，在刚才的例子中，我们的核函数为：



核函数能简化映射空间中的内积运算——刚好“碰巧”的是，在我们的 SVM 里需要计算的地方数据向量总是以内积的形式出现的。对比刚才我们上面写出来的式子，现在我们的分类函数为：



其中由如下 dual 问题计算而得：



这样一来计算的问题就算解决了，避开了直接在高维空间中进行计算，而结果却是等价的。

2.3、使用松弛变量处理 outliers 方法

在本文第一节最开始讨论支持向量机的时候，我们就假定，数据是线性可分的，亦即我们可以找到一个可行的超平面将数据完全分开。后来为了处理非线性数据，在上文2.2节使用 Kernel 方法对原来的线性 SVM 进行了推广，使得非线性的情况也能处理。虽然通过映射 $\phi(\cdot)$ 将原始数据映射到高维空间之后，能够线性分隔的概率大大增加，但是对于某些情况还是很难处理。

例如可能并不是因为数据本身是非线性结构的，而只是因为数据有噪音。对于这种偏离正常位置很远的数据点，我们称之为 outlier，在我们原来的 SVM 模型里，outlier 的存在有可能造成很大的影响，因为超平面本身就是只有少数几个 support vector 组成的，如果这些 support vector 里又存在 outlier 的话，其影响就很大了。例如下图：



用黑圈圈起来的那个蓝点是一个 outlier，它偏离了自己原本所应该在的

那个半空间，如果直接忽略掉它的话，原来的分隔超平面还是挺好的，但是由于这个 outlier 的出现，导致分隔超平面不得被挤歪了，变成途中黑色虚线所示（这只是一个示意图，并没有严格计算精确坐标），同时 margin 也相应变小了。当然，更严重的情况是，如果这个 outlier 再往右上移动一些距离的话，我们将无法构造出能将数据分开的超平面来。

为了处理这种情况，SVM 允许数据点在一定程度上偏离一下超平面。例如上图中，黑色实线所对应的距离，就是该 outlier 偏离的距离，如果把它移动回来，就刚好落在原来的超平面上，而不会使得超平面发生变形了。

我们原来的约束条件为：



现在考虑到 outlier 问题，约束条件变成了：



其中 ξ_i 称为松弛变量 (slack variable)，对应数据点 x_i 允许偏离的 functional margin 的量。当然，如果我们运行 ξ_i 任意大的话，那任意的超平面都是符合条件的了。所以，我们在原来的目标函数后面加上一项，使得这些 ξ_i 的总和也要最小：



其中 C 是一个参数，用于控制目标函数中两项（“寻找 margin 最大的超平面”和“保证数据点偏差量最小”）之间的权重。注意，其中 ξ_i 是需要优化的变量（之一），而 C 是一个事先确定好的常量。完整地写出来是这个样子：



用之前的方法将限制或约束条件加入到目标函数中，得到新的拉格朗日函数，如下所示：



分析方法和前面一样，转换为另一个问题之后，我们先让 α 针对 w 、 b 和 α 最小化：

$$\min_{\alpha} \sum_{i=1}^n \alpha_i$$

将 w 带回 α 并化简，得到和原来一样的目标函数：

$$\min_{\alpha} \sum_{i=1}^n \alpha_i$$

不过，由于我们得到 α 而又有 $\alpha_i \geq 0$ （作为 Lagrange multiplier 的条件），因此有 $\alpha_i \leq C$ ，所以整个 dual 问题现在写作：

$$\min_{\alpha} \sum_{i=1}^n \alpha_i$$

把前后的结果对比一下（错误修正：图中的 Dual formulation 中的 Minimize 应为 maximize）：

$$\max_{\alpha} \sum_{i=1}^n \alpha_i$$

可以看到唯一的区别就是现在 dual variable α 多了一个上限 C 。而 Kernel 化的非线性形式也是一样的，只要把 α 换成 α 即可。这样一来，一个完整的，可以处理线性和非线性并能容忍噪音和 outliers 的支持向量机才终于介绍完毕了。

行文至此，可以做个小结，不准确的说，SVM 它本质上即是一个分类方法，用 $w^T x + b$ 定义分类函数，于是求 w 、 b ，为寻最大间隔，引出 $1/2 \|w\|^2$ ，继而引入拉格朗日因子，化为对拉格朗日乘子 α 的求解（求解过程中会涉及到一系列最优化或凸二次规划等问题），如此，求 w 、 b 与求 α 等价，而 α 的求解可以用一种快速学习算法 SMO，至于核函数，是为处理非线性情况，若直接映射到高维计算恐维度爆炸，故在低维计算，等效高维表现。

第三层、扩展SVM

3.1、损失函数

在本文1.0节有这么一句话“支持向量机(SVM)是90年代中期发展起来的基于统计学习理论的一种机器学习方法，通过寻求结构化风险最小来提高学习机泛化能力，实现经验风险和置信范围的最小化，从而达到在统计样本量较少的情况下，亦能获得良好统计规律的目的。”但初次看到的读者可能并不了解什么是结构化风险，什么又是经验风险。要了解这两个所谓的“风险”，还得又从监督学习说起。

监督学习实际上就是一个经验风险或者结构风险函数的最优化问题。风险函数度量平均意义下模型预测的好坏，模型每一次预测的好坏用损失函数来度量。它从假设空间 F 中选择模型 f 作为决策函数，对于给定的输入 X ，由 $f(X)$ 给出相应的输出 Y ，这个输出的预测值 $f(X)$ 与真实值 Y 可能一致也可能不一致，用一个损失函数来度量预测错误的程度。损失函数记为 $L(Y, f(X))$ 。

常用的损失函数有以下几种（基本引用自《统计学习方法》）：



如此，SVM有第二种理解，即最优化+损失最小，或如@夏粉_百度所说“可从损失函数和优化算法角度看SVM，boosting，LR等算法，可能会有不同收获”。

关于损失函数，还可以看看张潼的这篇《Statistical behavior and consistency of classification methods based on convex risk minimization》。各种算法中常用的损失函数基本都具有fisher一致性，优化这些损失函数得到的分类器可以看作是后验概率的“代理”。

此外，他还有另外一篇论文《Statistical analysis of some multi-category large margin classification methods》，在多分类情况下margin loss的分析，这两篇对Boosting和SVM使用的损失函数分析的很透彻。

3.2、SMO算法

在上文2.1.2节中，我们提到了求解对偶问题的序列最小最优化SMO算法，但并未提到其具体解法。

事实上，SMO算法是由Microsoft Research的John C. Platt在1998年发表的一篇[论文](#)《Sequential Minimal Optimization A Fast Algorithm for Training Support Vector Machines》中提出，它很快成为最快的二次规划优化算法，特别针对线性SVM和数据稀疏时性能更优。

接下来，咱们便参考John C. Platt的[这篇](#)文章来看看SMO的解法是怎样的。

3.2.1、SMO算法的解法

咱们首先来定义特征到结果的输出函数为

$$f(x) = \sum_i y_i \langle x, x_i \rangle + b$$

再三强调，这个 u 与我们之前定义的 u 实质是一样的。

接着，咱们重新定义咱们原始的优化问题，权当重新回顾，如下：

$$\min_w \frac{1}{2} \|w\|^2$$

求导得到：

$$w = 0$$

代入 $f(x)$ 中，可得 $f(x) = b$ 。

引入对偶因子后，得：

$$\min_{\alpha} \frac{1}{2} \sum_i \alpha_i^2$$

s.t: $\alpha_i \geq 0$ 且 $\sum_i \alpha_i = 0$

注：这里得到的 \min 函数与我们之前的 \max 函数实质也是一样，因为把符号变下，即有 \min 转化为 \max 的问题，且 y_i 也与之前的 y_i 等价， y_j 亦如

此。

经过加入松弛变量后，模型修改为：

α_i

β_i

从而最终我们的问题变为：

α_i

继而，根据KKT条件可以得出其中取值的意义为：

α_i

这里的还是拉格朗日乘子(问题通过拉格朗日乘法数来求解)

1. 对于第1种情况，表明 α_i 是正常分类，在边界内部（我们知道正确分类的点 $y_i \cdot f(x_i) \geq 0$ ）；
2. 对于第2种情况，表明了 α_i 是支持向量，在边界上；
3. 对于第3种情况，表明了 α_i 是在两条边界之间；

而最优解需要满足KKT条件，即上述3个条件都得满足，以下几种情况出现将会出现不满足：

$\alpha_i \leq 1$ 但是 $\alpha_i < C$ 则是不满足的,而原本 $\alpha_i = C$

$\alpha_i \geq 1$ 但是 $\alpha_i > 0$ 则是不满足的而原本 $\alpha_i = 0$

$\alpha_i = 1$ 但是 $\alpha_i = 0$ 或者 $\alpha_i = C$ 则表明不满足的，而原本应该是 $0 < \alpha_i < C$

所以要找出不满足KKT条件的这些 α_i ，并更新这些 α_i ，但这些 α_i 又受到另外一个约束，即

α_i

注：别忘了2.1.1节中，L对a、b求偏导，得到：

☐

因此，我们通过另一个方法，即同时更新 a_i 和 a_j ，要求满足以下等式：

☐

就能保证和为0的约束。

利用 $y_i a_i + y_j a_j = \text{常数}$ ，消去 a_i ，可得到一个关于单变量 a_j 的一个凸二次规划问题，不考虑其约束 $0 \leq a_j \leq C$ ，可以得其解为：

☐

这里 \square ， \square ，表示旧值。

然后考虑约束 $0 \leq a_j \leq C$ 可得到 a 的解析解为：

☐

把SMO中对于两个参数求解过程看成线性规划来理解来理解的话，那么下图所表达的便是约束条件：☐

☐

根据 y_i 和 y_j 同号或异号，可得出两个拉格朗日乘子的上下界分别为：

☐

对于 \square 。

那么如何求得 a_i 和 a_j 呢？

- 对于 a_i ，即第一个乘子，可以通过刚刚说的那3种不满足KKT的条件来找；
- 而对于第二个乘子 a_j 可以找满足条件：☐求得。

而 b 的更新则是：

☐

在满足下述条件：



下更新 b ，且每次更新完两个乘子的优化后，都需要再重新计算 b ，及对应的 E_i 值。最后更新所有 a_i ， y 和 b ，这样模型就出来了，从而即可求出咱们开头提出的分类函数



此外，这里也有一篇类似的文章，大家可以参考下。

3.2.2、SMO算法的步骤

这样，SMO的主要步骤如下：



意思是，

第一步选取一对 a_i 和 a_j ，选取方法使用启发式方法；

第二步，固定除 a_i 和 a_j 之外的其他参数，确定 W 极值条件下的 a_i ，由 a_j 表示 a_i 。

假定在某一次迭代中，需要更新，对应的拉格朗日乘子，，那么这个小规模的二次规划问题写为：



那么在每次迭代中，如何更新乘子呢？引用 [这里](#) 的两张PPT说明下：



知道了如何更新乘子，那么选取哪些乘子进行更新呢？具体选择方法有以下两个步骤：

1. 步骤1：先“扫描”所有乘子，把第一个违反KKT条件的作

为更新对象，令为 a_2 ；

2. 步骤2：在所有不违反KKT条件的乘子中，选择使 $|E_1 - E_2|$ 最大的 a_1 （注：别忘了，其中 α_i ，而 α_j ，求出来的 E 代表函数 u_i 对输入 x_i 的预测值与真实输出类标记 y_i 之差）。

值得一提的是，每次更新完两个乘子的优化后，都需要再重新计算 b ，及对应的 E_i 值。

与此同时，乘子的选择务必遵循两个原则：

- 使乘子能满足KKT条件
- 对一个满足KKT条件的乘子进行更新，应能最大限度增大目标函数的值（类似于 [梯度下降](#)）

综上，SMO算法的基本思想是将Vapnik在1982年提出的Chunking方法推到极致，SMO算法每次迭代只选出两个分量 a_i 和 a_j 进行调整，其它分量则保持固定不变，在得到解 a_i 和 a_j 之后，再用 a_i 和 a_j 改进其它分量。与通常的分解算法比较，尽管它可能需要更多的迭代次数，但每次迭代的计算量比较小，所以该算法表现出整理的快速收敛性，且不需要存储核矩阵，也没有矩阵运算。

3.5.3、SMO算法的实现

行文至此，我相信，SVM理解到了一定程度后，是的确能在脑海里从头至尾推导出相关公式的，最初分类函数，最大化分类间隔， $\max 1/\|w\|$ ， $\min 1/2\|w\|^2$ ，凸二次规划，拉格朗日函数，转化为对偶问题，SMO算法，都为寻找一个最优解，一个最优分类平面。一步步梳理下来，为什么这样那样，太多东西可以追究，最后实现。如下图所示：



至于下文中将阐述的核函数则是为了更好的处理非线性可分的情况，而松弛变量则是为了纠正或约束少量“不安分”或脱离集体不好归类的因子。

台湾的林智仁教授写了一个封装SVM算法的 [libsvm](#) 库，大家可以看看，此外 [这里](#) 还有一份libsvm的注释文档。

除了在这篇论文《fast training of support vector machines using sequential minimal optimization》中platt给出了SMO算法的逻辑代码之外，[这里](#)也有一份SMO的实现代码，大家可以看下。

读者评论

本文发表后，[微博](#)上的很多朋友给了不少意见，以下是节选的一些精彩评论：

1. “压力”陡增的评论 → //@藏了个锋：我是看着July大神的博文长大的啊//@zlkysl：就是看了最后那一篇才决定自己的研究方向为SVM的。--
<http://weibo.com/1580904460/zraWk0u6u?mod=weibotime>
。
2. @张金辉：“SVM的三重境界，不得不转的一篇。其实Coursera的课堂上Andrew Ng讲过支持向量机，但显然他没有把这作为重点，加上Ng讲支持向量机的方法我一时半会难以完全消化，所以听的也是一知半解。真正开始了解支持向量机就是看的这篇“三重境界”，之后才对这个算法有了大概的概念，以至如何去使用，再到其中的原理为何，再到支持向量机的证明等。总之，这篇文章开启了我长达数月的研究支持向量机阶段，直到今日。”--
<http://zhan.renren.com/profile/249335584?from=template#!/tag/三重境界>。
3. @孤独之守望者：“最后，推出svm的cost function 是hinge loss，然后对比其他的方法的cost function，说明其实他们的目标函数很像，那么问题是svm为什么这么popular呢？您可以再加些VC dimension跟一些error bound的数学，点一下，提供一个思路和方向”。--
<http://weibo.com/1580904460/AiohoyDwq?mod=weibotime>
。
4. @夏粉_百度：“在面试时，考察SVM可考察机器学习各方面能力：目标函数,优化过程,并行方法，算法收敛性,样本复杂度，适用场景,调参经验，不过个人认为考察boosting和LR也还不错啊。此外，随着统计机器学习不断进步，SVM只被当成使用了一个替代01损失hinge研究，更通用的方法被提出，损失函数研究替代损失与贝叶斯损失关

系，算法稳定性研究替代损失与推广性能关系,凸优化研究如何求解凸目标函数，SVM,boosting等算法只是这些通用方法的一个具体组建而已。”

5. @居里猴姐：关于SVM损失函数的问题，可以看看张潼老师的这篇《Statistical behavior and consistency of classification methods based on convex risk minimization》。各种算法中常用的损失函数基本都具有fisher一致性，优化这些损失函数得到的分类器可以看作是后验概率的“代理”。此外，张潼老师还有另外一篇论文《Statistical analysis of some multi-category large margin classification methods》，在多分类情况下margin loss的分析，这两篇对Boosting和SVM使用的损失函数分析的很透彻。
6. @夏粉_百度：SVM用了hinge损失，hinge损失不可导，不如其它替代损失方便优化并且转换概率麻烦。核函数也不太用，现在是大数据时代，样本非常大，无法想象一个 n^2 的核矩阵如何存储和计算。而且，现在现在非线性一般靠深度学习了。//@Copper_PKU:请教svm在工业界的应用典型的有哪些？工业界如何选取核函数，经验的方法？svm的训练过程如何优化？
7. @Copper_PKU：July的svm tutorial 我个人觉得还可以加入和修改如下部分：(1) 对于支持向量解释，可以结合图和拉格朗日参数来表达，松弛中sv没有写出来。(2) SMO算法部分，加入Joachims论文中提到的算法，以及SMO算法选取workset的方法，包括SMO算法的收敛判断，还有之前共轭梯度求解方法，虽然是较早的算法，但是对于理解SMO算法有很好的效果。模型的优化和求解都是迭代的过程，加入历史算法增强立体感。--
http://weibo.com/1580904460/Akw6dl3Yk#_rnd1385474436177。
8. //@廖临川：之所以sgd对大训练集的效果更好，1.因为SGD优化每次迭代使用样本子集，比使用训练全集（尤其是百万数量级）要快得多；2.如果目标函数是凸的或者伪凸的，SGD几乎必然可以收敛到全局最优；否则，则收敛到局部最优；3.SGD一般不需要收敛到全局最优，只要得到足够好的解，就可以立即停止。//@Copper_PKU：sgd的

核心思想：是迭代训练，每拿到一个样本就算出基于当前 $w(t)$ 的 loss function， t 代表训练第 t 次，然后进行下一 $w(t+1)$ 的更新， $w(t+1) = w(t) - (\text{learning rate}) * \text{loss function}$ 的梯度，这个类比神经网络中 bp 中的参数训练方法。sample by sample 就是每次仅处理一个样本 而不是一个 batch。

9. // @Copper_PKU：从损失函数角度说：primal 问题可以理解为正则化项 + loss function，求解目标是在两个中间取平衡 如果强调 loss function 最小则会 overfitting，所以有 C 参数。// @研究者 July：SVM 还真就是在一定限定条件下，即约束条件下求目标函数的最优值问题，同时，为减少误判率，尽量让损失最小。

10. ...

参考文献及推荐阅读

1. 《支持向量机导论》，[美] Nello Cristianini / John Shawe-Taylor 著；
2. 支持向量机导论一书的支持网站：<http://www.support-vector.net/>；
3. 《数据挖掘导论》，[美] Pang-Ning Tan / Michael Steinbach / Vipin Kumar 著；
4. 《数据挖掘：概念与技术》，(加) Jiawei Han; Micheline Kamber 著；
5. 《数据挖掘中的新方法：支持向量机》，邓乃扬 田英杰 著；
6. 《支持向量机--理论、算法和扩展》，邓乃扬 田英杰 著；
7. 支持向量机系列，pluskid：http://blog.pluskid.org/?page_id=683；
8. http://www.360doc.com/content/07/0716/23/11966_615252.shtml；
9. 数据挖掘十大经典算法初探；
10. 《模式识别支持向量机指南》，C.J.C Burges 著；
11. 《统计学习方法》，李航著(第7章有不少内容参考自支持向量机导论一书，不过，可以翻翻看看)；
12. 《统计自然语言处理》，宗成庆编著，第十二章、文本分类；
13. SVM 入门系列，Jasper：
<http://www.blogjava.net/zhenandaci/category/31868.html>；
14. 最近邻决策和 SVM 数字识别的实现和比较，作者不详；
15. 斯坦福大学机器学习课程原始讲义：

- <http://www.cnblogs.com/jerrylead/archive/2012/05/08/2489725.html> ;
16. 斯坦福机器学习课程笔记:
<http://www.cnblogs.com/jerrylead/tag/Machine%20Learning/> ;
17. <http://www.cnblogs.com/jerrylead/archive/2011/03/13/1982639.html> ;
18. SMO算法的数学推导:
<http://www.cnblogs.com/jerrylead/archive/2011/03/18/1988419.html> ;
19. 数据挖掘掘中所需的概率论与数理统计知识、上;
20. 关于机器学习方面的文章，可以读读：
<http://www.cnblogs.com/vivounicorn/category/289453.html> ;
21. 数学系教材推荐：
http://blog.sina.com.cn/s/blog_5e638d950100dswh.html ;
22. 《神经网络与机器学习(原书第三版)》，[加] Simon Haykin 著；
23. 正态分布的前世今生： <http://t.cn/zlH3Ygc> ；
24. 《数理统计学简史》，陈希孺院士著；
25. 《最优化理论与算法(第2版)》，陈宝林编著；
26. A Gentle Introduction to Support Vector Machines in Biomedicine:
http://www.nyuinformatics.org/downloads/supplements/SVM_Tutorial_2 ,
此PPT很赞，除了对引入拉格朗日对偶变量后的凸二次规划问题的深入度不够之外，其它都挺好，配图很精彩，本文有几张图便引自此PPT中；
27. 来自卡内基梅隆大学carnegie mellon university(CMU)的讲解SVM的PPT: <http://www.autonlab.org/tutorials/svm15.pdf> ;
28. 发明libsvm的台湾林智仁教授06年的机器学习讲义SVM:
http://wenku.baidu.com/link?url=PWTGMYNb4HGUrUQUZwTH2B4r8pIMgLMiWIK1ymVORrdsJWab7IALDiors64JW_6mD93dtuWHwFWxsAk6p0rzchR8Qh5_4jWHC ;
29. <http://staff.ustc.edu.cn/~ketang/PPT/PRLec5.pdf> ;
30. Introduction to Support Vector Machines (SVM), By Debpakash Patnai M.E (SSA), <https://www.google.com.hk/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CCwQFjAA&url=http://Support%2520Vector%2520Machine-1%2eppt&ei=JRR6UqT5C-iyiQfWyIDgCg&usg=AFQjCNGw1fTbpH4ltQjjmx1d25ZqbCN9nA> ;
31. 多人推荐过的libsvm: <http://www.csie.ntu.edu.tw/~cjlin/libsvm/> ;
32. 《machine learning in action》，中文版为《机器学习实战》；
33. SMO算法的提出: Sequential Minimal Optimization A Fast Algorithm for Training Support Vector Machines:

- <http://research.microsoft.com/en-us/um/people/jplatt/smoTR.pdf> ;
34. 《统计学习理论的本质》，[美] Vladimir N. Vapnik著，非常晦涩，不做过多推荐；
 35. 张兆翔，机器学习第五讲之支持向量机
<http://irip.buaa.edu.cn/~zxzhang/courses/MachineLearning/5.pdf> ;
 36. VC维的理论解释：<http://www.svms.org/vc-dimension/>，中文VC维解释 <http://xiaoxia001.iteye.com/blog/1163338> ;
 37. 来自NEC Labs America的Jason Weston关于SVM的讲义
http://www.cs.columbia.edu/~kathy/cs4701/documents/jason_svm_tutorial ;
 38. 来自MIT的SVM讲义：
<http://www.mit.edu/~9.520/spring11/slides/class06-svm.pdf> ;
 39. PAC问题：<http://www.cs.huji.ac.il/~shashua/papers/class11-PAC2.pdf> ;
 40. 百度张潼老师的两篇论文：《Statistical behavior and consistency of classification methods based on convex risk minimization》
http://home.olemiss.edu/~xdang/676/Consistency_of_Classification_Con
《Statistical analysis of some multi-category large margin classification methods》 ;
 41. <http://jacoxu.com/?p=39> ;
 42. 《矩阵分析与应用》，清华张贤达著；
 43. SMO算法的实现：<http://blog.csdn.net/techq/article/details/6171688> ;
 44. 常见面试之机器学习算法思想简单梳理：
<http://www.cnblogs.com/tornadomeet/p/3395593.html> ;
 45. 矩阵的wikipedia页面：
<http://zh.wikipedia.org/wiki/%E7%9F%A9%E9%98%B5> ;
 46. 最小二乘法及其实现：
<http://blog.csdn.net/qll125596718/article/details/8248249> ;
 47. 统计学习方法概论：
<http://blog.csdn.net/qll125596718/article/details/8351337> ;
 48. <http://www.csdn.net/article/2012-12-28/2813275-Support-Vector-Machine> ;
 49. A Tutorial on Support Vector Regression:
<http://alex.smola.org/papers/2003/SmoSch03b.pdf> ; SVR简明版：
<http://www.cmlab.csie.ntu.edu.tw/~cyy/learning/tutorials/SVR.pdf> 。
 50. SVM Org: <http://www.support-vector-machines.org/> ;
 51. R. Collobert. Large Scale Machine Learning. Université Paris VI phd

thesis. 2004:

http://ronan.collobert.com/pub/matos/2004_phdthesis_lip6.pdf ;

52. Making Large-Scale SVM Learning Practical:

http://www.cs.cornell.edu/people/tj/publications/joachims_99a.pdf ;

53. 文本分类与SVM:

<http://blog.csdn.net/zhzhl202/article/details/8197109> ;

54. Working Set Selection Using Second Order Information for Training Support Vector Machines:

<http://www.csie.ntu.edu.tw/~cjlin/papers/quadworkset.pdf> ;

55. SVM Optimization: Inverse Dependence on Training Set Size:

<http://icml2008.cs.helsinki.fi/papers/266.pdf> ;

56. Large-Scale Support Vector Machines: Algorithms and Theory:

<http://cseweb.ucsd.edu/~akmenon/ResearchExam.pdf> ;

57. 凸优化的概念: <http://cs229.stanford.edu/section/cs229-cvxopt.pdf> ;

58. 《凸优化》，作者: Stephen Boyd / Lieven Vandenberghe, 原作名: Convex Optimization;

59. Large-scale Non-linear Classification: Algorithms and Evaluations, Zhuang Wang, 讲了很多SVM算法的新进展:

http://ijcai13.org/files/tutorial_slides/te2.pdf ;

附录 更多题型

语言基础

- 1、C++中虚拟函数的实现机制。
- 2、指针数组和数组指针的区别。
- 3、malloc-free和new-delete的区别。
- 4、sizeof和strlen的区别。
- 5、描述函数调用的整个过程。
- 6、C++ STL里面的vector的实现机制，
 - 当调用push_back成员函数时，怎么实现？
 - 内存足则直接 placement new构造对象，否则扩充内存，转移对象，新对象placement new上去。
 - 当调用clear成员函数时，做什么操作，如果要释放内存该怎么做。
 - 调用析构函数，内存不释放。clear没有释放内存，只是将数组中的元素置为空了，释放内存需要delete。

概率统计

1

已知有个rand7()的函数，返回1到7随机自然数，让利用这个rand7()构造rand10() 随机1~10。

分析：这题主要考的是对概率的理解。程序关键是要算出rand10，1到10，十个数字出现的考虑都为10%。根据排列组合，连续算两次rand7出现的组合数是 $7*7=49$ ，这49种组合每一种出现考虑是相同的。怎么从49平均概率的转换为1到10呢？方法是：

- 1.rand7执行两次，出来的数为 $a1=rand7()-1$ ， $a2=rand7()-1$ 。
- 2.如果 $a1*7+a2<40$, $b=(a1*7+a2)/4+1$ ；如果 $a1*7+a2\geq 40$,重复第一步。参考代码如下所示： ``c int rand7() { return rand() % 7 + 1; }

```
int rand10() { int a71, a72, a10; do { a71 = rand7() - 1; a72 = rand7() - 1; a10 = a71 * 7 + a72; } while (a10 >= 40); return (a71 * 7 + a72) / 4 + 1; } ``
```

2

给你5个球，每个球被抽到的可能性为30、50、20、40、10，设计一个随机算法，该算法的输出结果为本次执行的结果。输出A，B，C，D，E即可。

3

2D平面上有一个三角形ABC，如何从这个三角形内部随机取一个点，且使得在三角形内部任何点被选取的概率相同。

4

英雄升级，

- 从0级升到1级，概率100%。
- 从1级升到2级，有1/3的可能成功；1/3的可能停留原级；1/3的可能下降到0级；

- 从2级升到3级，有 $1/9$ 的可能成功； $4/9$ 的可能停留原级； $4/9$ 的可能下降到1级。

每次升级要花费一个宝石，不管成功还是停留还是降级。求英雄从0级升到3级平均花费的宝石数目。

提示：从第 n 级升级到第 $n+1$ 级成功的概率是 $(1/3)^n$ （指数），停留原级和降级的概率一样，都为 $[1-(1/3)^n]/2$ 。

5

甲包8个红球 2个蓝球，乙包2个红球 8个蓝球。抛硬币决定从哪个包取球，取了11次，7红4蓝。注，每次取后还放进去，只抛一次硬币。问选的是甲包的概率？

提示：贝叶斯公式 + 全概率公式作答。

6

一个桶里面有白球、黑球各100个，现在按下述规则取球：

- i、每次从桶里面拿出来两个球；
- ii、如果取出的是两个同色的球，就再放入一个黑球；
- ii、如果取出的是两个异色的球，就再放入一个白球。问：最后桶里面只剩下一个黑球的概率是多少？

7

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。

提示：5个人抽5个签，只有一个签意味着“中签”，轮流抽签，5个人中签的概率一样大，皆为 $1/5$ ，也就是说，抽签先后顺序不影响公平性。

智力逻辑

1

五个海盗抢到了100颗宝石，每一颗都一样大小和价值连城。他们决定这么分：抽签决定自己的号码（1、2、3、4、5）

首先，由1号提出分配方案，然后大家表决，当且仅当超过半数的人同意时，按照他的方案进行分配，否则将被扔进大海喂鲨鱼

如果1号死后，再由2号提出分配方案，然后剩下的4人进行表决，当且仅当超过半数的人同意时，按照他的方案进行分配，否则将被扔入大海喂鲨鱼，依此类推。

条件：每个海盗都是很聪明的人，都能很理智地做出判断，从而做出选择。

问题：第一个海盗提出怎样的分配方案才能使自己的收益最大化？

2

用天平（只能比较，不能称重）从一堆小球中找出其中唯一一个较轻的，使用 x 次天平，最多可以从 y 个小球中找出较轻的那个，求 y 与 x 的关系式。

3

有12个小球,外形相同,其中一个小球的质量与其他11个不同，给一个天平,问如何用3次把这个小球找出来，并且求出这个小球是比其他的轻还是重。

4

13个球一个天平，现知道只有一个和它的重量不同，问怎样称才能用三次就找到那个球？

5

有一根27厘米的细木杆，在第3厘米、7厘米、11厘米、17厘米、23厘米这五个位置上各有一只蚂蚁。木杆很细，不能同时通过一只蚂蚁。开始时，蚂蚁的头朝左还是朝右是任意的，它们只会朝前走或调头，但不会后退。当任意两只蚂蚁碰头时，两只蚂蚁会同时调头朝反方向走。假设蚂蚁们每秒钟可以走一厘米的距离。

6

有8瓶水，其中有一瓶有毒，最少尝试几次可以找出来。

7

五只猴子分桃。半夜，第一只猴子先起来，它把桃分成了相等的五堆，多出一只。于是，它吃掉了一个，拿走了一堆；第二只猴子起来一看，只有四堆桃。于是把四堆合在一起，分成相等的五堆，又多出一个。于是，它也吃掉了一个，拿走了一堆；.....其他几只猴子也都是这样分的。问：这堆桃至少有多少个？

分析：先给这堆桃子加上4个，设此时共有X个桃子，最后剩下a个桃子：

- 第一只猴子分完后还剩： $(1-1/5)X=(4/5)X$;
- 第二只猴子分完后还剩： $(1-1/5)2X$;
- 第三只猴子分完后还剩： $(1-1/5)3X$;
- 第四只猴子分完后还剩： $(1-1/5)4X$;
- 第五只猴子分完后还剩： $(1-1/5)5X=(1024/3125)X$;

得： $a=(1024/3125)X$ ；要使a为整数，X最小取3125，减去加上的4个，所以，这堆桃子最少有3121个。

8

我们有很多瓶无色的液体，其中有一瓶是毒药，其它都是蒸馏水，实验的小白鼠喝了以后会在5分钟后死亡，而喝到蒸馏水的小白鼠则一切正常。现在有5只小白鼠，请问一下，我们用这五只小白鼠，5分钟的时间，能够检测多少瓶液体的成分？

9

25匹赛马，5个跑道，也就是说每次有5匹马可以同时比赛。问最少比赛多少次可以知道跑得最快的5匹马。

10

宿舍内5个同学一起玩对战游戏。每场比赛有一些人作为红方，另一些人作为蓝方。请问至少需要多少场比赛，才能使任意两个人之间有一场红方对蓝方和蓝方对红方的比赛？

提示：答案为4场。

11、单词博弈

甲乙两个人用一个英语单词玩游戏。两个人轮流进行，每个人每次从中删掉任意一个字母，如果剩余的字母序列是严格单调递增的（按字典序 $a < b < c < \dots < z$ ），则这个人胜利。两个人都足够聪明（即如果有赢的方案，都不会选输的方案），甲先开始，问他能赢么？

例如：输入 bad，则甲可以删掉b或者a,剩余的是ad或者bd，他就赢了，输出1。又如：输入 aaa，则甲只能删掉1个a，乙删掉一个a,剩余1个a，乙获胜，输出0。

系统设计

1、搜索关键词智能提示suggestion

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。



提示：此题比较开放，简单直接的方法是：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。然后用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，我们把这个统计热词的方法称为TOP K算法。

当然，在实际中，还有很多细节需要考虑，有兴趣的读者可以继续参阅相关资料。

2

某服务器流量统计器，每天有1000亿的访问记录数据，包括时间、URL、IP。设计系统实现记录数据的保存、管理、查询。要求能实现以下功能：

- 计算在某一时间段（精确到分）时间内的，某URL的所有访问量。
- 计算在某一时间段（精确到分）时间内的，某IP的所有访问量。

3

假设某个网站每天有超过10亿次的页面访问量，出于安全考虑，网站会记录访问客户端访问的IP地址和对应的时间，如果现在已经记录了1000亿条数据，想统计一个指定时间段内的区域IP地址访问量，那么这些数据应该按照何种方式来组织，才能尽快满足上面的统计需求呢？设计完方案后，并指出该方案的优缺点，比如在什么情况下，可能会非常慢？提示：用B+树来组织，非叶子节点存储（某个时间点，页面访问量），叶子节点是访问的IP地址。这个方案的优点是查询某个时间段内

的IP访问量很快，但是要统计某个IP的访问次数或是上次访问时间就不得不遍历整个树的叶子节点。或者可以建立二级索引，分别是时间和地点来建立索引。

4

给你10台机器，每个机器2个CPU和2GB内存，现在已知在10亿条记录的数据库里执行一次查询需要5秒，问用什么方法能让90%的查询能在100毫秒以内返回结果。

提示：将10亿条记录排序，然后分到10个机器中，分的时候是一个记录一个记录的轮流分，确保每个机器记录大小分布差不多，每一次查询时，同时提交给10台机器，同时查询，因为记录已排序，可以采用二分法查询。

如果无排序，只能顺序查询，那就要看记录本身的概率分布，否则不可能实现。毕竟一个机器2个CPU未必能起到作用，要看这两个CPU能否并行存取内存，取决于系统架构。

5

有1000万条URL，每条URL长度为50字节，只包含主机前缀，要求实现URL提示系统，并满足以下条件：

- 实时更新匹配用户输入的地址，每输出一个字符，输出最新匹配URL
- 每次只匹配主机前缀，例如对www.abaidu.com和www.baidu.com，用户输入www.b时只提示www.baidu.com
- 每次提供10条匹配的URL

6

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：

- 一台服务器死掉后，不会造成大面积的访问错误，
- 原有的访问基本还是停留在同一台服务器上；
- 尽量考虑负载均衡。

提示：一致性hash算法。

7

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

有一大批数据，百万级别的。数据项内容是：用户ID、科目ABC各自的成绩。其中用户ID为0~1000万之间，且是连续的，可以唯一标识一条记录。科目ABC成绩均在0~100之间。有两块磁盘，空间大小均为512MB，内存空间64MB。

- 为实现快速查询某用户ID对应的各科成绩，问磁盘文件及内存该如何组织；
- 改变题目条件，ID为0~10亿之间，且不连续。问磁盘文件及内存该如何组织；
- 在问题2的基础上，增加一个需求。在查询各科成绩的同时，获取该用户的排名，问磁盘文件及内存该如何组织。

8

有几百亿的整数，分布的存储到几百台通过网络连接的计算机上，你能否开发出一个算法和系统，找出这几百亿数据的中值？就是在一组排序好的数据中居于中间的数。显然，一台机器是装不下所有的数据。也尽量少用网络带宽。

9

类似做一个手机键盘，上面有1到9个数字，每个数字都代表几个字母

（比如1代表abc三个字母，z代表wxyz等等），现在要求设计当输入某几个数字的组合时，查找出通讯录中的人名及电话号码。

10

这是一种用户登录验证手段，例如银行登录系统，这个设备显示6位数字，每60秒变一次，再经过服务器认证，通过则允许登录。问How to design this system?

- 系统设计思路？服务器端为何能有效认证动态密码的正确性？
- 如果是千万量级永固，给出系统设计图示或说明，要求子功能模块划分清晰，给出关键的数据结构或数据库表结构。考虑用户量级的影响和扩展性，用户密码的随机性等，如果设计系统以支持这几个因素.
- 系统算法升级时，服务器端和设备端可能都要有所修改，如何设计系统，能够使得升级过程（包括可能的设备替换或重设）尽量平滑？

11

http服务器会在用户访问某一个文件的时候，记录下该文件被访问的日志，网站管理员都会去统计每天每文件被访问的次数。写一个小程序，来遍历整个日志 文件，计算出每个文件被访问的访问次数。

- 请问这个管理员设计这个算法
- 该网站管理员后来加入腾讯从事运维工作，在腾讯，单台http服务器不够用的，同样的内容，会分布在全国各地上百台服务器上。每台服务器上的日志数量，都是之前的10倍之多，每天服务器的性能更好，之前他用的是单核cpu，现在用的是8核的，管理员发现在这种的海量的分布式服务器，基本没法使用了，请重新设计一个算法。

12

一在线推送服务，同时为10万个用户提供服务，对于每个用户服务从10万首歌的曲库中为他们随机选择一首，同一用户不能推送重复的，设计方案，内存尽可能小，写出数据结构与算法。

13

每个城市的IP段是固定的，新来一个IP，找出它是哪个城市的，设计一个后台系统。

14

请设计一个排队系统，能够让每个进入队伍的用户都能看到自己在队列中所处的位置 and 变化，队伍可能随时有人加入和退出；当有人退出影响到用户的位置排名时需要及时反馈到用户。

15

设计相应的数据结构和算法，尽量高效的统计一片英文文章（总单词数目）里出现的所有英文单词，按照在文章中首次出现的顺序打印输出该单词和它的出现次数。

16

有几百亿的整数，分布的存储到几百台通过网络连接的计算机上，你能否开发出一个算法和系统，找出这几百亿数据的中值？就是在一组排序好的数据中居于中间的数。显然，一台机器是装不下所有的数据。也尽量少用网络带宽。

17

假设已有10万个敏感词，现给你50个单词，查询这50个单词中是否有敏感词。

分析：换句话说，题目要你判断这50个单词是否存在那10万个敏感词库里，明显是字符串匹配，由于是判断多个单词不是一个，故是多模式字符串匹配问题，既是多模式字符串匹配问题，那么便有一类称之为多模式字符串匹配算法，而这类算法无非是KMP、hash、trie、AC自动机、wm等等。

那到底用哪种算法呢？这得根据题目的应用场景而定。

10万 + 50，如果允许误差的话，你可能会考虑用布隆过滤器；否则，只查一次的话，可能hash最快，但hash消耗空间大，故若考虑tire的话，可

以针对这10万个敏感词建立trie树，然后对那50个单词搜索这棵10万敏感词构建的tire树，但用tire树同样耗费空间，有什么更好的办法呢？Double Array Trie么？请读者继续思考。

操作系统

1

请问死锁的条件是什么？以及如何处理死锁问题？

解答：互斥条件（Mutual exclusion）：

- 1、资源不能被共享，只能由一个进程使用。
- 2、请求与保持条件（Hold and wait）：已经得到资源的进程可以再次申请新的资源。
- 3、非剥夺条件（No pre-emption）：已经分配的资源不能从相应的进程中被强制地剥夺。
- 4、循环等待条件（Circular wait）：系统中若干进程组成环路，该环路中每个进程都在等待相邻进程正占用的资源。

如何处理死锁问题：

- 1、忽略该问题。例如鸵鸟算法，该算法可以应用在极少发生死锁的情况下。为什么叫鸵鸟算法呢，因为传说中鸵鸟看到危险就把头埋在地底下，可能鸵鸟觉得看不到危险也就没危险了吧。跟掩耳盗铃有点像。
- 2、检测死锁并且恢复。
- 3、仔细地对资源进行动态分配，以避免死锁。
- 4、通过破除死锁四个必要条件之一，来防止死锁产生。

2

请阐述动态链接库与静态链接库的区别。

解答：静态链接库是.lib格式的文件，一般在工程的设置界面加入工程中，程序编译时会把lib文件的代码加入你的程序中因此会增加代码大小，你的程序一运行lib代码强制被装入你程序的运行空间，不能手动移除lib代码。

动态链接库是程序运行时动态装入内存的模块，格式*.dll，在程序运行时可以随意加载和移除，节省内存空间。

在大型的软件项目中一般要实现很多功能，如果把所有单独的功能写成一个一个lib文件的话，程序运行的时候要占用很大的内存空间，导致运行缓慢；但是如果将功能写成dll文件，就可以在用到该功能的时候调用功能对应的dll文件，不用这个功能时将dll文件移除内存，这样可以节省内存空间。

3

请阐述进程与线程的区别。

解答：

- ①从概念上：
 - 进程：一个程序对一个数据集的动态执行过程，是分配资源的基本单位。
 - 线程：一个进程内的基本调度单位。线程的划分尺度小于进程，一个进程包含一个或者更多的线程。
- ②从执行过程中来看：
 - 进程：拥有独立的内存单元，而多个线程共享内存，从而提高了应用程序的运行效率。
 - 线程：每一个独立的线程，都有一个程序运行的入口、顺序执行序列、和程序的出口。但是线程不能够独立的执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。
- ③从逻辑角度来看（重要区别）：
 - 多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但是，操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理及资源分配。

4

用户进程间通信主要哪几种方式？

解答：主要有以下6种：

- 1、管道：管道是单向的、先进先出的、无结构的、固定大小的字节流，它把一个进程的标准输出和另一个进程的标准输入连接在一起。写进程在管道的尾端写入数据，读进程在管道的道端读出数据。数据读出后将从管道中移走，其它读进程都不能再读到这些数

据。管道提供了简单的流控制机制。进程试图读空管道时，在有数据写入管道前，进程将一直阻塞。同样地，管道已经满时，进程再试图写管道，在其它进程从管道中移走数据之前，写进程将一直阻塞。

- 无名管道：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系（通常是指父子进程关系）的进程间使用。
- 命名管道：命名管道也是半双工的通信方式，在文件系统中作为一个特殊的设备文件而存在，但是它允许无亲缘关系进程间的通信。当共享管道的进程执行完所有的I/O操作以后，命名管道将继续保存在文件系统中以便以后使用。
- 2、信号量：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其它进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
- 3、消息队列：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
- 4、信号：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。
- 5、共享内存：共享内存就是映射一段能被其它进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的IPC方式，它是针对其它进程间通信方式运行效率低而专门设计的。它往往与其它通信机制（如信号量）配合使用，来实现进程间的同步和通信。
- 6、套接字：套接字也是一种进程间通信机制，与其它通信机制不同的是，它可用于不同机器间的进程通信。

网络协议

1

请简单阐述TCP连接的三次握手。

sift算法

sift算法的编译与实现

代码：Rob Hess维护的sift 库。

环境：windows xp+vc6.0。

条件：opencv1.0、gsl-1.8.exe

昨日，下载了Rob Hess的sift库，将其源码粗略的看了看，想要编译时，遇到了不少问题，先修改了下代码，然后下载opencv、gsl。最后，几经周折，才最终编译成功。

以下便是sift源码库编译后的效果图：



为了给有兴趣实现sift算法的朋友提供个参考，特整理此文如下。要了解什么是sift算法，请参考：[九、图像特征提取与匹配之SIFT算法](#)。
ok，咱们下面，就来利用Rob Hess维护的sift 库来实现sift算法：

首先，请下载Rob Hess维护的sift 库：

<http://blogs.oregonstate.edu/hess/code/sift>

下载Rob Hess的这个压缩包后，如果直接解压缩，直接编译，那么会出现下面的错误提示：

编译提

示:error C1083: Cannot open include file: 'cxcore.h': No such file or directory,
找不到这个头文件。

这个错误，是因为你还没有安装opencv，因为：cxcore.h和cv.h是开源的OPEN CV头文件,不是VC++的默认安装文件,所以你还得下载OpenCV并进行安装。然后，可以在OpenCV文件夹下找到你所需要的头文件了。

据网友称，截止2010年4月4日，还没有在VC6.0下成功使用opencv2.0的案例。所以，如果你是VC6.0的用户请下载opencv1.0版本。vs的话，opencv2.0,1.0任意下载。

以下，咱们就以 **vc6.0**为平台举例，下载并安装**opencv1.0**版本、**gsl** 等。当然，你也可以用vs编译，同样下载opencv（具体版本不受限制）、gsl等。

请按以下步骤操作：

一、下载**opencv1.0**

http://sourceforge.net/projects/opencvlibrary/files/opencv-win/1.0/OpenCV_1.0.exe/download

二、安装**opencv1.0**，配置**Windows**环境变量

1. 安装注意：假如你是将OpenCV安装到 **C :/Program Files/OpenCV**（如果你安装的时候选择不是安装在**C**盘，则下面所有对应的**C**盘都改为你所安装的那个“**X**盘”，即可），在安装时选择"将/OpenCV/bin加入系统变量"，打上“勾”。（Add/OpenCV/bin to the system PATH。这一步确认选上了之后，下面的检查环境变量的步骤，便可免去）



2. 检查环境变量。为了确保上述步骤中，加入了系统变量，在安装opencv1.0成功后，还得检查**C:/Program Files/OpenCV/bin**是否已经被加入到环境变量PATH，如果没有，请加入。
3. 最后是配置**Visual C++ 6.0**。
4. 全局设置

菜单Tools->Options->Directories：先设置lib路径，选择Library files，在下方填入路径：

C:/Program Files/OpenCV/lib

然后选择include files，在下方填入路径(参考下图):

C:/Program Files/OpenCV/cxcore/include

C:/Program Files/OpenCV/cv/include

C:/Program Files/OpenCV/cv_aux/include

C:/Program Files/OpenCV/ml/include

C:/Program Files/OpenCV/otherlibs/highgui

C:/Program Files/OpenCV/otherlibs/cvcam/include



最后选择source files，在下方填入路径:

C:/Program Files/OpenCV/cv/src

C:/Program Files/OpenCV/cxcore/src

C:/Program Files/OpenCV/cv_aux/src

C:/Program Files/OpenCV/otherlibs/highgui

C:/Program Files/OpenCV/otherlibs/cvcam/src/windows

5. 项目设置 每创建一个将要使用OpenCV的VC Project，都需要给它指定需要的lib。菜单：Project->Settings，然后将Setting for选为All Configurations，然后选择右边的link标签，在Object/library modules附上： cxcore.lib cv.lib ml.lib cvaux.lib highgui.lib cvcam.lib 当然，你不需要这么多lib，你可以只添加你需要的lib(见下图)



三、下载gsl，gsl也是一个库，也需要下载：

<http://sourceforge.net/projects/gnuwin32/files/gsl/1.8/gsl-1.8.exe/download>
。在编译时候GSL也是和OpenCV一样要把头文件和lib的路径指定好。

四、配置gsl

将C:/WinGsl/bin中的WinGsl.dll和WinGslD.dll复制到C:/VC6.0/Bin；将整个Gsl目录复制到C:/VC6.0/Bin下；lib目录下的所有.lib文件全部复制到C:/VC6.0/Lib下。

然后，在tools—options—directories中，将C:/WinGsl下的lib，gsl分别加入到库文件和头文件的搜索路径中。

以下是可能会出现的错误情况处理：

1. OpenCV安装后“没有找到cxcore100.dll”的错误处理 在安装时选择“将/OpenCV/bin加入系统变量”（Add/OpenCV/bin to the system PATH）。但该选项并不一定能成功添加到系统变量，如果编写的程序在运行时出现“没有找到cxcore100.dll，因为这个应用程序未能启动。重新安装应用程序可能会修复此问题。”的错误。手动在我的电脑->属性->高级->环境变量->系统变量->path添加c:/program files/opencv/bin;添加完成后需要重启计算机。
2. vc6.0下配置了一下，可是编译程序时遇到如下一个错误：Linking... LINK : fatal error LNK1104: cannot open file"odbccp32.libcxcore.lib" 可能是：在工程设置的时候添加连接库时没加空格或.来把两个文件名（odbccp32.lib cxcore.lib）分开。注意每一次操作后，记得保存。

若经过以上所有的步骤之后，如果还不能正常编译，那就是还要稍微修改下你下载的Rob Hess代码。ok，日后，若有空，再好好详细剖析下此sift的源码。最后，祝你编译顺利。

updated

今天下午试了下sift + KD + BBF，然后用两幅不同的图片做了下匹配（当然，运行结果显示是不匹配的），效果还不错：

[http://weibo.com/1580904460/yDmzAEwcV#1348475194313!](http://weibo.com/1580904460/yDmzAEwcV#1348475194313)



同时，编译的过程中，直接用的VS2010 + opencv（并没下gsl）。
2012.09.24。完。

教你一步一步用c语言实现sift算法、上

参考：Rob Hess维护的sift 库

环境：windows xp+vc6.0

条件：c语言实现。

引言：

在我写的关于sift算法的前俩篇文章里头，已经对sift算法有了初步的介绍：[九、图像特征提取与匹配之SIFT算法](#)，而后在：[九（续）、sift算法的编译与实现](#)里，我也简单记录下了如何利用opencv，gsl等库编译运行sift程序。

但据一朋友表示，是否能用c语言实现sift算法，同时，尽量不用到opencv，gsl等第三方库之类的东西。而且，Rob Hess维护的sift 库，也不好懂，有的人根本搞不懂是怎么一回事。

那么本文，就教你如何利用c语言一步一步实现sift算法，同时，你也能真正明白sift算法到底是怎么一回事了。

ok，先看一下，本程序最终运行的效果图，sift 算法分为五个步骤（下文详述），对应以下 第二~第六幅图：



sift算法的步骤

要实现一个算法，首先要完全理解这个算法的原理或思想。咱们先来简单了解下，什么叫sift算法：

sift，尺度不变特征转换，是一种电脑视觉的算法用来侦测与描述影像中的局部性特征，它在空间尺度中寻找极值点，并提取出其位置、尺度、旋转不变量，此算法由 David Lowe 在1999年所发表，2004年完善总结。

所谓，Sift算法就是用不同尺度（标准差）的高斯函数对图像进行平滑，然后比较平滑后图像的差别，

差别大的像素就是特征明显的点。

以下是sift算法的五个步骤：

一、建立图像尺度空间(或高斯金字塔)，并检测极值点

首先建立尺度空间，要使得图像具有尺度空间不变形，就要建立尺度空间，sift算法采用了高斯函数来建立尺度空间，高斯函数公式为：



上述公式 $G(x,y,e)$ ，即为尺度可变高斯函数。

而，一个图像的尺度空间 $L(x,y,e)$ ，定义为原始图像 $I(x,y)$ 与上述的一个可变尺度的2维高斯函数 $G(x,y,e)$ 卷积运算。

即，原始影像 $I(x,y)$ 在不同的尺度 e 下，与高斯函数 $G(x,y,e)$ 进行卷积，得到 $L(x,y,e)$ ，如下：



以上的 (x,y) 是空间坐标， e ，是尺度坐标，或尺度空间因子， e 的大小决定平滑程度，大尺度对应图像的概貌特征，小尺度对应图像的细节特征。大的 e 值对应粗糙尺度(低分辨率)，反之，对应精细尺度(高分辨率)。

尺度，受 e 这个参数控制的表示。而不同的 $L(x,y,e)$ 就构成了尺度空间，具体计算的时候，即使连续的高斯函数，都被离散为（一般为奇数大小） $(2*k+1)*(2*k+1)$ 矩阵，来和数字图像进行卷积运算。

随着 e 的变化，建立起不同的尺度空间，或称之为建立起图像的高斯金字塔。

但，像上述 $L(x,y,e) = G(x,y,e)*I(x,y)$ 的操作，在进行高斯卷积时，整个图像就要遍历所有的像素进行卷积(边界点除外)，于此，就造成了时间和空间上的很大浪费。

为了更有效的在尺度空间检测到稳定的关键点，也为了缩小时间和空间复杂度，对上述的操作作了一个改建：即，提出了高斯差分尺度空间（DOG scale-space）。利用不同尺度的高斯差分与原始图像 $I(x,y)$ 相乘，卷积生成。



DOG算子计算简单，是尺度归一化的LOG算子的近似。

ok，耐心点，咱们再来总结一下上述内容：

1. 高斯卷积

在组建一组尺度空间后，再组建下一组尺度空间，对上一组尺度空间的最后一幅图像进行二分之一采样，得到下一组尺度空间的第一幅图像，然后进行像建立第一组尺度空间那样的操作，得到第二组尺度空间，公式定义为

$$L(x,y,e) = G(x,y,e)*I(x,y)$$

图像金字塔的构建：图像金字塔共 O 组，每组有 S 层，下一组的图像由上一组图像降采样得到，效果图，图A如下（左为上一组，右为下一组）：



2. 高斯差分

在尺度空间建立完毕后，为了能够找到稳定的关键点，采用高斯差分的方法来检测那些在局部位置的极值点，即采用俩个相邻的尺度中的图像相减，即公式定义为：


$$D(x,y,e) = ((G(x,y,ke) - G(x,y,e)) * I(x,y) = L(x,y,ke) - L(x,y,e)$$

效果图，图B：



SIFT的精妙之处在于采用图像金字塔的方法解决这一问题，我们可以把两幅图像想象成是连续的，分别以它们作为底面作四棱锥，就像金字塔，那么每一个截面与原图像相似，那么两个金字塔中必然会有包含大小一致的物体的无穷个截面，但应用只能是离散的，所以我们只能构造有限层，层数越多当然越好，但处理时间会相应增加，层数太少不行，因为向下采样的截面中可能找不到尺寸大小一致的两个物体的图像。

咱们再来具体阐述下构造 **$D(x,y,e)$** 的详细步骤：

1. 首先采用不同尺度因子的高斯核对图像进行卷积以得到图像的不同尺度空间，将这一组图像作为金字塔图像的第一层。
2. 接着对第一层图像中的2倍尺度图像（相对于该层第一幅图像的2倍尺度）以2倍像素距离进行下采样来得到金字塔图像的第二层中的第一幅图像，对该图像采用不同尺度因子的高斯核进行卷积，以获得金字塔图像中第二层的一组图像。
3. 再以金字塔图像中第二层中的2倍尺度图像（相对于该层第一幅图像的2倍尺度）以2倍像素距离进行下采样来得到金字塔图像的第三层中的第一幅图像，对该图像采用不同尺度因子的高斯核进行卷积，以获得金字塔图像中第三层的一组图像。这样依次类推，从而获得了金字塔图像的每一层中的一组图像，如下图所示： 
4. 对上图得到的每一层相邻的高斯图像相减，就得到了高斯差分图像，如下述第一幅图所示。下述第二幅图中的右列显示了将每组中相邻图像相减所生成的高斯差分图像的结果，限于篇幅，图中只给出了第一层和第二层高斯差分图像的计算（下述俩幅图统称为图2）：



图像金字塔的建立：对于一幅图像 I ，建立其在不同尺度($scale$)的图像，也成为子八度($octave$)，这是为了 $scale$ - $invariant$ ，也就是在任何尺度都能够有对应的特征点，第一个子八度的 $scale$ 为原图大小，后面每个 $octave$ 为上一个 $octave$ 降采样的结果，即原图的 $1/4$ （长宽分别减半），构成下一个子八度（高一层金字塔）。

5. 因为高斯差分函数是归一化的高斯拉普拉斯函数的近似，所以可以从高斯差分金字塔分层结构提取出图像中的极值点作为候选的特征点。对DOG 尺度空间每个点与相邻尺度和相邻位置的点逐个进行比较，得到的局部极值位置即为特征点所处的位置和对应的尺度。

二、检测关键点

为了寻找尺度空间的极值点，每一个采样点要和它所有的相邻点比较，看其是否比它的图像域和尺度域的相邻点大或者小。如下图，图3所示，中间的检测点和它同尺度的8个相邻点和上下相邻尺度对应的 9×2 个点共26个点比较，以确保在尺度空间和二维图像空间都检测到极值点。



因为需要同相邻尺度进行比较，所以在一组高斯差分图像中只能检测到两个尺度的极值点（如上述第二幅图中右图的五角星标识），而其它尺度的极值点检测则需要在图像金字塔的上一层高斯差分图像中进行。依次类推，最终在图像金字塔中不同层的高斯差分图像中完成不同尺度极值的检测。

当然这样产生的极值点并不都是稳定的特征点，因为某些极值点响应较弱，而且DOG算子会产生较强的边缘响应。

三、关键点方向的分配

为了使描述符具有旋转不变性，需要利用图像的局部特征为给每一个关

键点分配一个方向。利用关键点邻域像素的梯度及方向分布的特性，可以得到梯度模值和方向如下：



其中，尺度为每个关键点各自所在的尺度。

在以关键点为中心的邻域窗口内采样，并用直方图统计邻域像素的梯度方向。梯度直方图的范围是0~360度，其中每10度一个方向，总共36个方向。

直方图的峰值则代表了该关键点处邻域梯度的主方向，即作为该关键点的方向。

在计算方向直方图时，需要用一个参数等于关键点所在尺度1.5倍的高斯权重窗对方向直方图进行加权，上图中用蓝色的圆形表示，中心处的蓝色较重，表示权值最大，边缘处颜色潜，表示权值小。如下图所示，该示例中为了简化给出了8方向的方向直方图计算结果，实际sift创始人David Lowe的原论文中采用36方向的直方图。



方向直方图的峰值则代表了该特征点处邻域梯度的方向，以直方图中最大值作为该关键点的主方向。为了增强匹配的鲁棒性，只保留峰值大于主方向峰值80%的方向作为该关键点的辅方向。因此，对于同一梯度值的多个峰值的关键点位置，在相同位置和尺度将会有多个关键点被创建但方向不同。仅有15%的关键点被赋予多个方向，但可以明显的提高关键点匹配的稳定性。

至此，图像的关键点已检测完毕，每个关键点有三个信息：位置、所处尺度、方向。由此可以确定一个SIFT特征区域。

四、特征点描述符

通过以上步骤，对于每一个关键点，拥有三个信息：位置、尺度以及方向。接下来就是为每个关键点建立一个描述符，使其不随各种变化而改变，比如光照变化、视角变化等等。并且描述符应该有较高的独特性，以便于提高特征点正确匹配的概率。

首先将坐标轴旋转为关键点方向，以确保旋转不变性。



接下来以关键点为中心取 8×8 的窗口。

上图，图5中左部分的中央黑点为当前关键点的位置，每个小格代表关键点邻域所在尺度空间的一个像素，箭头方向代表该像素的梯度方向，箭头长度代表梯度模值，图中蓝色的圈代表高斯加权的范围（越靠近关键点的像素梯度方向信息贡献越大）。

然后在每 4×4 的小块上计算8个方向的梯度方向直方图，绘制每个梯度方向的累加值，即可形成一个种子点，如图5右部分所示。此图中一个关键点由 2×2 共4个种子点组成，每个种子点有8个方向向量信息。这种邻域方向性信息联合的思想增强了算法抗噪声的能力，同时对于含有定位误差的特征匹配也提供了较好的容错性。

实际计算过程中，为了增强匹配的稳健性，Lowe建议对每个关键点使用 4×4 共16个种子点来描述，这样对于一个关键点就可以产生128个数据，即最终形成128维的SIFT特征向量。此时SIFT特征向量已经去除了尺度变化、旋转等几何变形因素的影响，再继续将特征向量的长度归一化，则可以进一步去除光照变化的影响。

五、最后一步

当两幅图像的SIFT特征向量生成后，下一步我们采用关键点特征向量的欧式距离来作为两幅图像中关键点的相似性判定度量。取上图中，图像A中的某个关键点，并找出其与图像B中欧式距离最近的前两个关键点，在这两个关键点中，如果最近的距离除以次近的距离少于某个比例阈值，则接受这一对匹配点。降低这个比例阈值，SIFT匹配点数目会减少，但更加稳定。关于sift算法的更多理论介绍请参看此文：

<http://blog.csdn.net/abcjennifer/article/details/7639681>。

sift算法的逐步c实现

ok，上文搅了那么多的理论，如果你没有看懂它，咋办咧？没关系，下面，咱们来一步一步实现此sift算法，即使你没有看到上述的理论，慢慢的，你也会明白sift算法到底是怎么回事，sift算法到底是怎么实现

的...。

yeah, 请看:

前期工作:

在具体编写核心函数之前, 得先做几个前期的准备工作:

1. 头文件:

```
# ifdef  _CH_

# pragma  package <opencv>

# endif


# ifndef  _EiC

# include  <stdio.h>


# include  "stdlib.h"
# include  "string.h"
# include  "malloc.h"
# include  "math.h"
# include  <assert.h>
# include  <ctype.h>
# include  <time.h>
# include  <cv.h>
# include  <cxcv.h>
# include  <highgui.h>
# include  <vector>

# endif


# ifdef  _EiC

# define  WIN32

# endif
```

2. 定义几个宏，及变量，以免下文函数中，突然冒出一个变量，而您却不知道怎么回事：

```
# define  NUMSIZE 2

# define  GAUSSKERN 3.5

# define  PI 3.14159265358979323846


//Sigma of base image -- See D.L.'s paper.

# define  INITSIGMA 0.5

//Sigma of each octave -- See D.L.'s paper.

# define  SIGMA sqrt(3) //1.6//


//Number of scales per octave.  See D.L.'s paper.

# define  SCALESPEROCTAVE 2

# define  MAXOCTAVES 4

int      numoctaves;


# define  CONTRAST_THRESHOLD    0.02

# define  CURVATURE_THRESHOLD  10.0

# define  DOUBLE_BASE_IMAGE_SIZE 1

# define  peakRelThresh 0.8

# define  LEN 128


// temporary storage

CvMemStorage* storage = 0 ;
```

3. 然后，咱们还得，声明几个变量，以及建几个数据结构（数据结构是一切程序事物的基础嘛，:D。）：

```
//Data structure for a float image.

typedef  struct  ImageSt {          /*金字塔每一层*/
```

```

    float levelsigma;

    int levelsigmalength;

    float absolute_sigma;

    CvMat *Level; //CvMat是OPENCV的矩阵类，其元素可以是图像的象素
    值

} ImageLevels;


typedef struct ImageSt1 { /*金字塔每一阶梯*/

    int row, col; //Dimensions of image.

    float subsample;

    ImageLevels *Octave;

} ImageOctaves;


ImageOctaves *DOGOctaves;

//DOG pyr, DOG算子计算简单，是尺度归一化的LoG算子的近似。


ImageOctaves *mag_thresh ;

ImageOctaves *mag_pyr ;

ImageOctaves *grad_pyr ;


//keypoint数据结构，
Lists of keypoints are linked by the "next" field.

typedef struct KeypointSt

{

    float row, col; /* 反馈回原图像大小，特征点的位置 */

    float sx, sy; /* 金字塔中特征点的位置*/

    int octave, level; /*金字塔中，特征点所在的阶梯、层次*/


    float scale, ori, mag; /*所在层的尺度sigma,主方向
orientation (range [-PI,PI]), 以及幅值*/

    float *descrip; /*特征描述字指针：128维或32维等*/

    struct KeypointSt *next;
/* Pointer to next keypoint in list. */

```

```
} *keypoint;
```

```
//定义特征点具体变量
```

```
Keypoint keypoints= NULL ; //用于临时存储特征点的位置等
```

```
Keypoint keyDescriptors= NULL ; //用于最后的确定特征点以及特征描述字
```

4. 声明几个图像的基本处理函数:

```
CvMat * halfSizeImage (CvMat * im) ; //缩小图像: 下采样
```

```
CvMat * doubleSizeImage (CvMat * im) ; //扩大图像: 最近临方法
```

```
CvMat * doubleSizeImage2 (CvMat * im) ; //扩大图像: 线性插值
```

```
float getPixelBI (CvMat * im, float col, float row) ; //双线性插值函数
```

```
void normalizeVec ( float * vec, int dim) ; //向量归一化
```

```
CvMat* GaussianKernel2D ( float sigma) ; //得到2维高斯核
```

```
void normalizeMat (CvMat* mat) ; //矩阵归一化
```

```
float * GaussianKernel1D ( float sigma, int dim) ; //得到1维高斯核
```

```
//在具体像素处宽度方向进行高斯卷积
```

```
float ConvolveLocWidth ( float * kernel, int dim, CvMat * src, int x, int y) ;
```

```
//在整个图像宽度方向进行1D高斯卷积
```

```
void Convolve1DWidth ( float * kern, int dim, CvMat * src, CvMat * dst) ;
```

```
//在具体像素处高度方向进行高斯卷积
```

```
float ConvolveLocHeight ( float * kernel, int dim, CvMat * src, int x, int y) ;
```

```
//在整个图像高度方向进行1D高斯卷积
```

```
void Convolve1DHeight ( float * kern, int dim, CvMat * src, CvMat * dst) ;
```

```
//用高斯函数模糊图像
```

```
int BlurImage (CvMat * src, CvMat * dst, float sigma) ;
```

算法核心

本程序中，sift算法被分为以下五个步骤及其相对应的函数（可能表述与上，或与前俩篇文章有所偏差，但都一个意思）：

//SIFT算法第一步：图像预处理

```
CvMat * ScaleInitImage (CvMat * im) ; //金字塔初始化
```

//SIFT算法第二步：建立高斯金字塔函数

```
ImageOctaves* BuildGaussianOctaves (CvMat * image) ; //建立高斯金字塔
```

//SIFT算法第三步：特征点位置检测，最后确定特征点的位置

```
int DetectKeypoint ( int numoctaves, ImageOctaves *GaussianPyr) ;
```

```
void DisplayKeypointLocation (IplImage* image, ImageOctaves *GaussianPyr) ;
```

//SIFT算法第四步：计算高斯图像的梯度方向和幅值，计算各个特征点的主方向

```
void ComputeGrad_DirecandMag ( int numoctaves, ImageOctaves *GaussianPyr) ;
```

```
int FindClosestRotationBin ( int binCount, float angle) ; //进行方向直方图统计
```

```
void AverageWeakBins ( double * bins, int binCount) ; //对方向直方图滤波
```

//确定真正的主方向

```
bool InterpolateOrientation ( double left, double middle, double right, double *degreeCorrection, double *peakValue) ;
```

//确定各个特征点处的主方向函数

```
void AssignTheMainOrientation ( int numoctaves, ImageOctaves *GaussianPyr, ImageOctaves *mag_pyr, ImageOctaves *grad_pyr) ;
```

//显示主方向

```
void DisplayOrientation (IplImage* image, ImageOctaves *GaussianPyr) ;
```

//SIFT算法第五步：抽取各个特征点处的特征描述字

```
void ExtractFeatureDescriptors ( int numoctaves, ImageOctaves *GaussianPyr) ;
```

```
//为了显示图象金字塔，而作的图像水平、垂直拼接
```

```
CvMat* MosaicHorizen ( CvMat* im1, CvMat* im2 ) ;
```

```
CvMat* MosaicVertical ( CvMat* im1, CvMat* im2 ) ;
```

```
//特征描述点，网格
```

```
# define GridSpacing 4
```

主体实现

ok，以上所有的工作都就绪以后，那么接下来，咱们就先来编写main函数，因为你一看主函数之后，你就立马能发现sift算法的工作流程及其原理了。

（主函数中涉及到的函数，下一篇文章：[一、教你一步一步用c语言实现sift算法、下](#)，咱们自会一个一个编写）：

```
int main ( void )
```

```
{
```

```
//声明当前帧IplImage指针
```

```
IplImage* src = NULL ;
```

```
IplImage* image1 = NULL ;
```

```
IplImage* grey_im1 = NULL ;
```

```
IplImage* DoubleSizeImage = NULL ;
```

```
IplImage* mosaic1 = NULL ;
```

```
IplImage* mosaic2 = NULL ;
```

```
CvMat* mosaicHorizen1 = NULL ;
```

```
CvMat* mosaicHorizen2 = NULL ;
```

```
CvMat* mosaicVertical1 = NULL ;
```

```

CvMat* image1Mat = NULL ;

CvMat* tempMat= NULL ;


ImageOctaves *Gaussianpyr;

int rows,cols;


# define Im1Mat(ROW,COL) ((float*)(image1Mat->data.fl + image1Mat->step/sizeof(float) *(ROW)))[(COL)]


//灰度图像像素的数据结构

# define Im1B(ROW,COL) ((uchar*)(image1->imageData + image1->widthStep*(ROW)))[(COL)*3]

# define Im1G(ROW,COL) ((uchar*)(image1->imageData + image1->widthStep*(ROW)))[(COL)*3+1]

# define Im1R(ROW,COL) ((uchar*)(image1->imageData + image1->widthStep*(ROW)))[(COL)*3+2]


storage = cvCreateMemStorage( 0 );


//读取图片

if ( (src = cvLoadImage( "street1.jpg" , 1 )) == 0 )
// test1.jpg einstein.pgm back1.bmp

return - 1 ;


//为图像分配内存

image1 = cvCreateImage(cvSize(src->width, src->height), IPL_DEPTH_8U, 3 );
grey_im1 = cvCreateImage(cvSize(src->width, src->height), IPL_DEPTH_8U, 1 );
DoubleSizeImage = cvCreateImage(cvSize( 2 *(src->width), 2 *(src->height)), IPL_DEPTH_8U, 3 );


//为图像阵列分配内存，假设两幅图像的大小相同，tempMat跟随image1的大小

image1Mat = cvCreateMat(src->height, src->width, CV_32FC1);

//转化成单通道图像再处理

cvCvtColor(src, grey_im1, CV_BGR2GRAY);

```



```

//转换进入Mat数据结构, 图像操作使用的是浮点型操作

cvConvert(grey_im1, image1Mat);

double t = ( double )cvGetTickCount();

//图像归一化

cvConvertScale( image1Mat, image1Mat, 1.0 / 255 , 0 );

int dim = min(image1Mat->rows, image1Mat->cols);

numoctaves = ( int ) ( log (( double ) dim) / log ( 2.0 )) - 2 ; //金字塔阶数

numoctaves = min(numoctaves, MAXOCTAVES);

//SIFT算法第一步, 预滤波除噪声, 建立金字塔底层

tempMat = ScaleInitImage(image1Mat) ;

//SIFT算法第二步, 建立Guassian金字塔和DOG金字塔

Gaussianpyr = BuildGaussianOctaves(tempMat) ;

t = ( double )cvGetTickCount() - t;

printf ( "the time of build Gaussian pyramid and DOG pyramid is %.1f/n"
, t/(cvGetTickFrequency()* 1000. ) );

# define ImLevels(OCTAVE, LEVEL, ROW, COL) ((float *)
(Gaussianpyr[(OCTAVE)].Octave[(LEVEL)].Level-
>data.fl + Gaussianpyr[(OCTAVE)].Octave[(LEVEL)].Level->step/sizeof(float) *
(ROW))[(COL)]

//显示高斯金字塔

for ( int i= 0 ; i<numoctaves;i++)
{
    if (i== 0 )
    {
        mosaicHorizen1=MosaicHorizen( (Gaussianpyr[ 0 ].Octave)[ 0 ].Level, (Gaussianpyr[ 0 ].Octave)[ 1 ].Level );

        for ( int j= 2 ;j<SCALESPEROCTAVE+ 3 ;j++)

```

```

        mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (Gaussianpyr[ 0 ].Octave)
[j].Level );

        for ( j= 0 ;j<NUMSIZE;j++)

            mosaicHorizen1=halfSizeImage(mosaicHorizen1);

    }

    else if (i== 1 )

    {

        mosaicHorizen2=MosaicHorizen( (Gaussianpyr[ 1 ].Octave)[ 0
].Level, (Gaussianpyr[ 1 ].Octave)[ 1 ].Level );

        for ( int j= 2 ;j<SCALESPEROCTAVE+ 3 ;j++)

            mosaicHorizen2=MosaicHorizen( mosaicHorizen2, (Gaussianpyr[ 1 ].Octave)
[j].Level );

        for ( j= 0 ;j<NUMSIZE;j++)

            mosaicHorizen2=halfSizeImage(mosaicHorizen2);

        mosaicVertical1=MosaicVertical( mosaicHorizen1, mosaicHorizen2 );

    }

    else

    {

        mosaicHorizen1=MosaicHorizen( (Gaussianpyr[i].Octave)[ 0
].Level, (Gaussianpyr[i].Octave)[ 1 ].Level );

        for ( int j= 2 ;j<SCALESPEROCTAVE+ 3 ;j++)

            mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (Gaussianpyr[i].Octave)
[j].Level );

        for ( j= 0 ;j<NUMSIZE;j++)

            mosaicHorizen1=halfSizeImage(mosaicHorizen1);

        mosaicVertical1=MosaicVertical( mosaicVertical1, mosaicHorizen1 );

    }

}

mosaic1 = cvCreateImage(cvSize(mosaicVertical1->width, mosaicVertical1-
>height), IPL_DEPTH_8U, 1 );

cvConvertScale( mosaicVertical1, mosaicVertical1, 255.0 , 0 );

cvConvertScaleAbs( mosaicVertical1, mosaic1, 1 , 0 );

// cvSaveImage("GaussianPyramid of me.jpg",mosaic1);

```

```

cvNamedWindow( "mosaic1" , 1 );
cvShowImage( "mosaic1" , mosaic1);
cvWaitKey( 0 );
cvDestroyWindow( "mosaic1" );

//显示DOG金字塔

for ( i= 0 ; i<numoctaves;i++)
{
    if (i== 0 )
    {
        mosaicHorizen1=MosaicHorizen( (DOGoctaves[ 0 ].Octave)[ 0
].Level, (DOGoctaves[ 0 ].Octave)[ 1 ].Level );

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

            mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (DOGoctaves[ 0 ].Octave)
[j].Level );

        for ( j= 0 ;j<NUMSIZE;j++)

            mosaicHorizen1=halfSizeImage(mosaicHorizen1);

    }

    else if (i== 1 )
    {
        mosaicHorizen2=MosaicHorizen( (DOGoctaves[ 1 ].Octave)[ 0
].Level, (DOGoctaves[ 1 ].Octave)[ 1 ].Level );

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

            mosaicHorizen2=MosaicHorizen( mosaicHorizen2, (DOGoctaves[ 1 ].Octave)
[j].Level );

        for ( j= 0 ;j<NUMSIZE;j++)

            mosaicHorizen2=halfSizeImage(mosaicHorizen2);

        mosaicVertical1=MosaicVertical( mosaicHorizen1, mosaicHorizen2 );

    }

    else
    {
        mosaicHorizen1=MosaicHorizen( (DOGoctaves[i].Octave)[ 0
].Level, (DOGoctaves[i].Octave)[ 1 ].Level );

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

```

```

        mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (DOGoctaves[i].Octave)
[j].Level );

    for ( j= 0 ;j<NUMSIZE;j++)

        mosaicHorizen1=halfSizeImage(mosaicHorizen1);

        mosaicVertical1=MosaicVertical( mosaicVertical1, mosaicHorizen1 );

    }

}

//考虑到DOG金字塔各层图像都会有正负，所以，必须寻找最负的，以将所有图像抬高一个台阶去显示

double min_val= 0 ;

double max_val= 0 ;

cvMinMaxLoc( mosaicVertical1, &min_val, &max_val, NULL , NULL , NULL );

if ( min_val< 0.0 )

    cvAddS( mosaicVertical1, cvScalarAll( (- 1.0 )*min_val ), mosaicVertical1,
NULL );

    mosaic2 = cvCreateImage(cvSize(mosaicVertical1->width, mosaicVertical1-
>height), IPL_DEPTH_8U, 1 );

    cvConvertScale( mosaicVertical1, mosaicVertical1, 255.0 /(max_val-min_val), 0
);

    cvConvertScaleAbs( mosaicVertical1, mosaic2, 1 , 0 );


// cvSaveImage("DOGPyramid of me.jpg",mosaic2);

cvNamedWindow( "mosaic1" , 1 );

cvShowImage( "mosaic1" , mosaic2);

cvWaitKey( 0 );


//SIFT算法第三步：特征点位置检测，最后确定特征点的位置

int keycount=DetectKeypoint(numoctaves, Gaussianpyr);

printf ( "the keypoints number are %d ;/n" , keycount);

cvCopy(src,image1, NULL );

DisplayKeypointLocation( image1 ,Gaussianpyr);


cvPyrUp( image1, DoubleSizeImage, CV_GAUSSIAN_5x5 );

```

```

cvNamedWindow( "image1" , 1 );
cvShowImage( "image1" , DoubleSizeImage);
cvWaitKey( 0 );
cvDestroyWindow( "image1" );

//SIFT算法第四步：计算高斯图像的梯度方向和幅值，计算各个特征点的主方向
ComputeGrad_DirecandMag(numoctaves, Gaussianpyr);
AssignTheMainOrientation( numoctaves, Gaussianpyr,mag_pyr,grad_pyr);
cvCopy(src,image1, NULL );
DisplayOrientation ( image1, Gaussianpyr);

// cvPyrUp( image1, DoubleSizeImage, CV_GAUSSIAN_5x5 );
cvNamedWindow( "image1" , 1 );
// cvResizeWindow("image1", 2*(image1->width), 2*(image1->height) );
cvShowImage( "image1" , image1);
cvWaitKey( 0 );

//SIFT算法第五步：抽取各个特征点处的特征描述字
ExtractFeatureDescriptors( numoctaves, Gaussianpyr);
cvWaitKey( 0 );

//销毁窗口
cvDestroyWindow( "image1" );
cvDestroyWindow( "mosaic1" );

//释放图像
cvReleaseImage(&image1);
cvReleaseImage(&grey_im1);
cvReleaseImage(&mosaic1);
cvReleaseImage(&mosaic2);

return 0 ;
}

```

更多见下文：[一、教你一步一步用c语言实现sift算法、下](#)。本文完。

教你一步一步用c语言实现sift算法、下

本文接上， [教你一步一步用c语言实现sift算法、上](#) 而来：

函数编写

ok，接上文，咱们一个一个的来编写main函数中所涉及到所有函数，这也是本文的关键部分：

```
//下采样原来的图像，返回缩小2倍尺寸的图像

CvMat * halfSizeImage(CvMat * im)
{
    unsigned int i,j;

    int w = im->cols/2;

    int h = im->rows/2;

    CvMat *imnew = cvCreateMat(h, w, CV_32FC1);

#define Im(ROW,COL) ((float *)(im->data.fl + im->step/sizeof(float) *(ROW)))
    [(COL)]

#define Imnew(ROW,COL) ((float *)(imnew->data.fl + imnew->step/sizeof(float) *
    (ROW)))[(COL)]

    for ( j = 0; j < h; j++)

        for ( i = 0; i < w; i++)

            Imnew(j,i)=Im(j*2, i*2);

    return imnew;
}

//上采样原来的图像，返回放大2倍尺寸的图像

CvMat * doubleSizeImage(CvMat * im)
{
    unsigned int i,j;
```

```

    int w = im->cols*2;

    int h = im->rows*2;

    CvMat *imnew = cvCreateMat(h, w, CV_32FC1);

#define Im(ROW,COL) ((float *)(im->data.fl + im->step/sizeof(float) *(ROW)))
[(COL)]

#define Imnew(ROW,COL) ((float *)(imnew->data.fl + imnew->step/sizeof(float) *
(ROW)))[(COL)]

    for ( j = 0; j < h; j++)

        for ( i = 0; i < w; i++)

            Imnew(j,i)=Im(j/2, i/2);

    return imnew;
}

//上采样原来的图像，返回放大2倍尺寸的线性插值图像
CvMat * doubleSizeImage2(CvMat * im)
{
    unsigned int i,j;

    int w = im->cols*2;

    int h = im->rows*2;

    CvMat *imnew = cvCreateMat(h, w, CV_32FC1);

#define Im(ROW,COL) ((float *)(im->data.fl + im->step/sizeof(float) *(ROW)))
[(COL)]

#define Imnew(ROW,COL) ((float *)(imnew->data.fl + imnew->step/sizeof(float) *
(ROW)))[(COL)]

    // fill every pixel so we don't have to worry about skipping pixels later
    for ( j = 0; j < h; j++)

    {

```



```

        for ( i = 0; i < w; i++)
        {
            Imnew(j,i)=Im(j/2, i/2);
        }
    }

    /*
    A B C
    E F G
    H I J

    pixels A C H J are pixels from original image
    pixels B E G I F are interpolated pixels
    */

    // interpolate pixels B and I
    for ( j = 0; j < h; j += 2)
        for ( i = 1; i < w - 1; i += 2)
            Imnew(j,i)=0.5*(Im(j/2, i/2)+Im(j/2, i/2+1));

    // interpolate pixels E and G
    for ( j = 1; j < h - 1; j += 2)
        for ( i = 0; i < w; i += 2)
            Imnew(j,i)=0.5*(Im(j/2, i/2)+Im(j/2+1, i/2));

    // interpolate pixel F
    for ( j = 1; j < h - 1; j += 2)
        for ( i = 1; i < w - 1; i += 2)
            Imnew(j,i)=0.25*
            (Im(j/2, i/2)+Im(j/2+1, i/2)+Im(j/2, i/2+1)+Im(j/2+1, i/2+1));

    return imnew;
}

```

//双线性插值，返回像素间的灰度值

```

float getPixelBI(CvMat * im, float col, float row)
{

```

```

    int irow, icol;

    float rfrac, cfrac;

    float row1 = 0, row2 = 0;

    int width=im->cols;

    int height=im->rows;

#define ImMat(ROW,COL) ((float *) (im->data.fl + im->step/sizeof(float) *(ROW)))
[(COL)]

    irow = (int) row;

    icol = (int) col;

    if (irow < 0 || irow >= height
        || icol < 0 || icol >= width)

        return 0;

    if (row > height - 1)

        row = height - 1;

    if (col > width - 1)

        col = width - 1;

    rfrac = 1.0 - (row - (float) irow);

    cfrac = 1.0 - (col - (float) icol);

    if (cfrac < 1)

    {

        row1 = cfrac * ImMat(irow,icol) + (1.0 - cfrac) * ImMat(irow,icol+1);

    }

    else

    {

        row1 = ImMat(irow,icol);

    }

    if (rfrac < 1)

    {

        if (cfrac < 1)

```

```

    {
        row2 = cfrac * ImMat(irow+1,icol) + (1.0 - cfrac) * ImMat(irow+1,icol+1)
    } else
    {
        row2 = ImMat(irow+1,icol);
    }
}

return rfrac * row1 + (1.0 - rfrac) * row2;
}

```

//矩阵归一化

```

void normalizeMat(CvMat* mat)
{
#define Mat(ROW,COL) ((float *) (mat->data.fl + mat->step/sizeof(float) *(ROW)))
[(COL)]

    float sum = 0;

    for (unsigned int j = 0; j < mat->rows; j++)
        for (unsigned int i = 0; i < mat->cols; i++)
            sum += Mat(j,i);

    for ( j = 0; j < mat->rows; j++)
        for (unsigned int i = 0; i < mat->cols; i++)
            Mat(j,i) /= sum;
}

```

//向量归一化

```

void normalizeVec(float* vec, int dim)
{
    unsigned int i;
    float sum = 0;

    for ( i = 0; i < dim; i++)

```

```

        sum += vec[i];
    for ( i = 0; i < dim; i++)
        vec[i] /= sum;
}

```

//得到向量的欧式长度，2-范数

```

float GetVecNorm( float* vec, int dim )
{
    float sum=0.0;
    for (unsigned int i=0;i<dim;i++)
        sum+=vec[i]*vec[i];
    return sqrt(sum);
}

```

//产生1D高斯核

```

float* GaussianKernel1D(float sigma, int dim)
{
    unsigned int i;

    //printf("GaussianKernel1D(): Creating 1x%d vector for sigma=%.3f gaussian kerne

    float *kern=(float*)malloc( dim*sizeof(float) );
    float s2 = sigma * sigma;
    int c = dim / 2;
    float m= 1.0/(sqrt(2.0 * CV_PI) * sigma);
    double v;
    for ( i = 0; i < (dim + 1) / 2; i++)
    {
        v = m * exp(-(1.0*i*i)/(2.0 * s2)) ;
        kern[c+i] = v;
        kern[c-i] = v;
    }
}

```

```

    }

    // normalizeVec(kern, dim);

    // for ( i = 0; i < dim; i++)

    // printf("%f ", kern[i]);

    // printf("/n");

    return kern;

}

```

//产生2D高斯核矩阵

```

CvMat* GaussianKernel2D(float sigma)
{
    // int dim = (int) max(3.0f, GAUSSKERN * sigma);

    int dim = (int) max(3.0f, 2.0 * GAUSSKERN * sigma + 1.0f);

    // make dim odd

    if (dim % 2 == 0)

        dim++;

    //printf("GaussianKernel(): Creating %dx%d matrix for sigma=%.3f gaussian/n", di

    CvMat* mat=cvCreateMat(dim, dim, CV_32FC1);

#define Mat(ROW,COL) ((float *) (mat->data.fl + mat->step/sizeof(float) *(ROW)))
[(COL)]

    float s2 = sigma * sigma;

    int c = dim / 2;

    //printf("%d %d/n", mat.size(), mat[0].size());

    float m= 1.0/(sqrt(2.0 * CV_PI) * sigma);

    for (int i = 0; i < (dim + 1) / 2; i++)

    {

        for (int j = 0; j < (dim + 1) / 2; j++)

        {

            //printf("%d %d %d/n", c, i, j);

            float v = m * exp(-(1.0*i*i + 1.0*j*j) / (2.0 * s2));

            Mat(c+i,c+j) =v;

```

```

        Mat(c-i,c+j) =v;
        Mat(c+i,c-j) =v;
        Mat(c-i,c-j) =v;

    }

}

// normalizeMat(mat);

return mat;

}

```

//x方向像素处作卷积

```

float ConvolveLocWidth(float* kernel, int dim, CvMat * src, int x, int y)
{
#define Src(ROW,COL) ((float *) (src->data.fl + src->step/sizeof(float) *(ROW)))
[(COL)]

    unsigned int i;
    float pixel = 0;
    int col;
    int cen = dim / 2;

    //printf("ConvolveLoc(): Applying convolution at location (%d, %d)/n", x, y);

    for ( i = 0; i < dim; i++)
    {
        col = x + (i - cen);

        if (col < 0)

            col = 0;

        if (col >= src->cols)

            col = src->cols - 1;

        pixel += kernel[i] * Src(y,col);

    }

    if (pixel > 1)

        pixel = 1;

    return pixel;
}

```

```
}
```

```
//x方向作卷积
```

```
void Convolve1DWidth(float* kern, int dim, CvMat * src, CvMat * dst)
```

```
{
```

```
#define DST(ROW,COL) ((float *) (dst->data.fl + dst->step/sizeof(float) *(ROW)))  
[(COL)]
```

```
    unsigned int i,j;
```

```
    for ( j = 0; j < src->rows; j++)
```

```
    {
```

```
        for ( i = 0; i < src->cols; i++)
```

```
        {
```

```
            //printf("%d, %d/n", i, j);
```

```
            DST(j,i) = ConvolveLocWidth(kern, dim, src, i, j);
```

```
        }
```

```
    }
```

```
}
```

```
//y方向像素处作卷积
```

```
float ConvolveLocHeight(float* kernel, int dim, CvMat * src, int x, int y)
```

```
{
```

```
#define Src(ROW,COL) ((float *) (src->data.fl + src->step/sizeof(float) *(ROW)))  
[(COL)]
```

```
    unsigned int j;
```

```
    float pixel = 0;
```

```
    int cen = dim / 2;
```

```
    //printf("ConvolveLoc(): Applying convolution at location (%d, %d)/n", x, y);
```

```
    for ( j = 0; j < dim; j++)
```

```
    {
```

```
        int row = y + (j - cen);
```

```

        if (row < 0)
            row = 0;

        if (row >= src->rows)
            row = src->rows - 1;

        pixel += kernel[j] * Src(row,x);
    }

    if (pixel > 1)
        pixel = 1;

    return pixel;
}

```

//y方向作卷积

```

void Convolve1DHeight(float* kern, int dim, CvMat * src, CvMat * dst)
{
#define Dst(ROW,COL) ((float *) (dst->data.fl + dst->step/sizeof(float) *(ROW)))
    [[COL]]

    unsigned int i,j;

    for ( j = 0; j < src->rows; j++)
    {
        for ( i = 0; i < src->cols; i++)
        {
            //printf("%d, %d/n", i, j);

            Dst(j,i) = ConvolveLocHeight(kern, dim, src, i, j);
        }
    }
}

```

//卷积模糊图像

```

int BlurImage(CvMat * src, CvMat * dst, float sigma)
{
    float* convkernel;

```



```

    int dim = (int) max(3.0f, 2.0 * GAUSSKERN * sigma + 1.0f);

    CvMat *tempMat;

    // make dim odd
    if (dim % 2 == 0)
        dim++;

    tempMat = cvCreateMat(src->rows, src->cols, CV_32FC1);

    convkernel = GaussianKernel1D(sigma, dim);

    Convolve1DWidth(convkernel, dim, src, tempMat);

    Convolve1DHeight(convkernel, dim, tempMat, dst);

    cvReleaseMat(&tempMat);

    return dim;
}

```

五个步骤

ok，接下来，进入重点部分，咱们依据上文介绍的sift算法的几个步骤，来一一实现这些函数。

为了版述清晰，再贴一下，主函数，顺便再加强下对sift 算法的五个步骤的认识：

1、 SIFT算法第一步：图像预处理

```
CvMat *ScaleInitImage(CvMat * im); //金字塔初始化
```

2、 SIFT算法第二步：建立高斯金字塔函数

```
ImageOctaves* BuildGaussianOctaves(CvMat * image); //建立高斯金字塔
```

3、 SIFT算法第三步：特征点位置检测，最后确定特征点的位置

```
int DetectKeypoint(int numoctaves, ImageOctaves *GaussianPyr);
```

4、 SIFT算法第四步：计算高斯图像的梯度方向和幅值，计算各个特征

点的主方向

```
void ComputeGrad_DirecandMag(int numoctaves, ImageOctaves
*GaussianPyr);
```

5、SIFT算法第五步：抽取各个特征点处的特征描述字

```
void ExtractFeatureDescriptors(int numoctaves, ImageOctaves
*GaussianPyr);
```

ok，接下来一一具体实现这几个函数：

SIFT算法第一步

SIFT算法第一步：扩大图像，预滤波剔除噪声，得到金字塔的最底层-第一阶的第一层：

```
CvMat * ScaleInitImage (CvMat * im)
{
    double sigma,preblur_sigma;

    CvMat *imMat;
    CvMat * dst;
    CvMat *tempMat;

    //首先对图像进行平滑滤波，抑制噪声

    imMat = cvCreateMat(im->rows, im->cols, CV_32FC1);
    BlurImage(im, imMat, INITSIGMA);

    //针对两种情况分别进行处理：初始化放大原始图像或者在原图像基础上进行后续操作

    //建立金字塔的最底层

    if (DOUBLE_BASE_IMAGE_SIZE)
    {
        tempMat = doubleSizeImage2(imMat); //对扩大两倍的图像进行二次采样，采样率为
        0.5，采用线性插值

        # define TEMPMAT(ROW,COL) ((float *) (tempMat->data.fl + tempMat-
        >step/sizeof(float) * (ROW)))[(COL)]

        dst = cvCreateMat(tempMat->rows, tempMat->cols, CV_32FC1);
```

```

    preblur_sigma = 1.0 ; //sqrt(2 - 4*INITSIGMA*INITSIGMA);

    BlurImage(tempMat, dst, preblur_sigma);

// The initial blurring for the first image of the first octave of the pyramid.

    sigma = sqrt ( ( 4
*INITSIGMA*INITSIGMA) + preblur_sigma * preblur_sigma );

    // sigma = sqrt(SIGMA * SIGMA - INITSIGMA * INITSIGMA * 4);

    //printf("Init Sigma: %f/n", sigma);

    BlurImage(dst, tempMat, sigma); //得到金字塔的最底层-放大2倍的图像

    cvReleaseMat( &dst );

    return tempMat;

}

else

{

    dst = cvCreateMat(im->rows, im->cols, CV_32FC1);

    //sigma = sqrt(SIGMA * SIGMA - INITSIGMA * INITSIGMA);

    preblur_sigma = 1.0 ; //sqrt(2 - 4*INITSIGMA*INITSIGMA);

    sigma = sqrt ( ( 4
*INITSIGMA*INITSIGMA) + preblur_sigma * preblur_sigma );

    //printf("Init Sigma: %f/n", sigma);

    BlurImage(imMat, dst, sigma); //得到金字塔的最底层：原始图像大小

    return dst;

}

}

```

SIFT算法第二步

SIFT第二步，建立Gaussian金字塔，给定金字塔第一阶第一层图像后，计算高斯金字塔其他尺度图像，每一阶的数目由变量 `SCALESPEROCTAVE` 决定，给定一个基本图像，计算它的高斯金字塔图像，返回外部向量是阶梯指针，内部向量是每一个阶梯内部的不同尺度图像。

//SIFT算法第二步

ImageOctaves* BuildGaussianOctaves(CvMat * image)

{

ImageOctaves *octaves;

CvMat *tempMat;

CvMat *dst;

CvMat *temp;

int i,j;

double k = pow(2, 1.0/((float)SCALESPEROCTAVE)); //方差倍数

float preblur_sigma, initial_sigma , sigma1,sigma2,sigma,absolute_sigma,sigma_f;

//计算金字塔的阶梯数目

int dim = min(image->rows, image->cols);

int numoctaves = (int) (log((double) dim) / log(2.0)) - 2; //金字塔阶数

//限定金字塔的阶梯数

numoctaves = min(numboctaves, MAXOCTAVES);

//为高斯金字塔和DOG金字塔分配内存

octaves=(ImageOctaves*) malloc(numoctaves * sizeof(ImageOctaves));

DOGoctaves=(ImageOctaves*) malloc(numoctaves * sizeof(ImageOctaves));

printf("BuildGaussianOctaves(): Base image dimension is %dx%d/n", (int)(0.5*(image->cols)), (int)(0.5*(image->rows)));

printf("BuildGaussianOctaves(): Building %d octaves/n", numoctaves);

// start with initial source image

tempMat=cvCloneMat(image);

// preblur_sigma = 1.0;//sqrt(2 - 4*INITSIGMA*INITSIGMA);

initial_sigma = sqrt(2);//sqrt((4*INITSIGMA*INITSIGMA) + preblur_sigma * preblu

// initial_sigma = sqrt(SIGMA * SIGMA - INITSIGMA * INITSIGMA * 4);

//在每一阶金字塔图像中建立不同的尺度图像

```

    for ( i = 0; i < numoctaves; i++)
    {
        //首先建立金字塔每一阶梯的最底层，其中0阶梯的最底层已经建立好

        printf("Building octave %d of dimesion (%d, %d)/n", i, tempMat-
>cols,tempMat->rows);

        //为各个阶梯分配内存

        octaves[i].Octave= (ImageLevels*) malloc( (SCALESPEROCTAVE + 3) * sizeof(Ima
        DOGoctaves[i].Octave= (ImageLevels*) malloc( (SCALESPEROCTAVE + 2) * sizeof(
        //存储各个阶梯的最底层

        (octaves[i].Octave)[0].Level=tempMat;

        octaves[i].col=tempMat->cols;
        octaves[i].row=tempMat->rows;
        DOGoctaves[i].col=tempMat->cols;
        DOGoctaves[i].row=tempMat->rows;

        if (DOUBLE_BASE_IMAGE_SIZE)
            octaves[i].subsample=pow(2,i)*0.5;
        else
            octaves[i].subsample=pow(2,i);

        if(i==0)
        {
            (octaves[0].Octave)[0].levelsigma = initial_sigma;
            (octaves[0].Octave)[0].absolute_sigma = initial_sigma;

            printf("0 scale and blur sigma : %f /n", (octaves[0].subsample) * ((octa
[0].absolute_sigma));

        }
        else
        {
            (octaves[i].Octave)[0].levelsigma = (octaves[i-1].Octave)
[SCALESPEROCTAVE].levelsigma;

            (octaves[i].Octave)[0].absolute_sigma = (octaves[i-1].Octave)
[SCALESPEROCTAVE].absolute_sigma;

```

```

        printf( "0 scale and blur sigma : %f /n", ((octaves[i].Octave)
[0].absolute_sigma) );

    }

    sigma = initial_sigma;

    //建立本阶梯其他层的图像

    for ( j = 1; j < SCALESPEROCTAVE + 3; j++)

    {

        dst = cvCreateMat(tempMat->rows, tempMat->cols, CV_32FC1);//用于存储高
        斯层

        temp = cvCreateMat(tempMat->rows, tempMat->cols, CV_32FC1);//用于存储
        DOG层

        // 2 passes of 1D on original

        // if(i!=0)

        // {

        //     sigma1 = pow(k, j - 1) * ((octaves[i-1].Octave)[j-
        1].levelsigma);

        //     sigma2 = pow(k, j) * ((octaves[i].Octave)[j-
        1].levelsigma);

        //     sigma = sqrt(sigma2*sigma2 - sigma1*sigma1);

        sigma_f= sqrt(k*k-1)*sigma;

        // }

        // else

        // {

        //     sigma = sqrt(SIGMA * SIGMA - INITSIGMA * INITSIGMA * 4)*pow(k,j

        // }

        sigma = k*sigma;

        absolute_sigma = sigma * (octaves[i].subsample);

        printf("%d scale and Blur sigma: %f /n", j, absolute_sigma);

        (octaves[i].Octave)[j].levelsigma = sigma;

        (octaves[i].Octave)[j].absolute_sigma = absolute_sigma;

        //产生高斯层

        int length=BlurImage((octaves[i].Octave)[j-1].Level, dst, sigma_f);

```

相应尺度

```
(octaves[i].Octave)[j].levelsigmalength = length;

(octaves[i].Octave)[j].Level=dst;

//产生DOG层

cvSub( ((octaves[i].Octave)[j]).Level, ((octaves[i].Octave)[j-1]).Level, temp, 0 );

//      cvAbsDiff( ((octaves[i].Octave)
[j]).Level, ((octaves[i].Octave)[j-1]).Level, temp );

((DOGoctaves[i].Octave)[j-1]).Level=temp;

}

// halve the image size for next iteration

tempMat = halfSizeImage( ( (octaves[i].Octave)
[SCALESPEROCTAVE].Level ) );

}

return octaves;

}
```

SIFT算法第三步

SIFT算法第三步，特征点位置检测，最后确定特征点的位置检测DOG金字塔中的局部最大值，找到之后，还要经过两个检验才能确认为特征点：一是它必须有明显的差异，二是他不应该是边缘点，（也就是说，在极值点处的主曲率比应该小于某一个阈值）。

```
//SIFT算法第三步，特征点位置检测，

int DetectKeypoint(int numoctaves, ImageOctaves *GaussianPyr)
{

//计算用于DOG极值点检测的主曲率比的阈值

double curvature_threshold;

curvature_threshold= ((CURVATURE_THRESHOLD + 1)*
(CURVATURE_THRESHOLD + 1))/CURVATURE_THRESHOLD;

#define ImLevels(OCTAVE, LEVEL, ROW, COL) ((float *)
(DOGoctaves[(OCTAVE)].Octave[(LEVEL)].Level-
>data.fl + DOGoctaves[(OCTAVE)].Octave[(LEVEL)].Level->step/sizeof(float) *
(ROW)))[(COL)]
```

```

int    keypoint_count = 0;

for (int i=0; i<numoctaves; i++)
{
    for(int j=1;j<SCALESPEROCTAVE+1;j++)//取中间的scaleperoctave个层
    {
        //在图像的有效区域内寻找具有显著性特征的局部最大值

        //float sigma=(GaussianPyr[i].Octave)[j].levelsigma;

        //int dim = (int) (max(3.0f, 2.0*GAUSSKERN *sigma + 1.0f)*0.5);

        int dim = (int)(0.5*((GaussianPyr[i].Octave)
[j].levelsiglength)+0.5);

        for (int m=dim;m<((DOGoctaves[i].row)-dim);m++)

            for(int n=dim;n<((DOGoctaves[i].col)-dim);n++)

                {

                    if ( fabs(ImLevels(i,j,m,n))>= CONTRAST_THRESHOLD )

                        {

                            if ( ImLevels(i,j,m,n)!=0.0 ) //1、首先是非零

                                {

                                    float inf_val=ImLevels(i,j,m,n);

                                    if(( (inf_val <= ImLevels(i,j-1,m-1,n-1))&&

                                        (inf_val <= ImLevels(i,j-1,m ,n-1))&&

                                        (inf_val <= ImLevels(i,j-1,m+1,n-1))&&

                                        (inf_val <= ImLevels(i,j-1,m-1,n ))&&

                                        (inf_val <= ImLevels(i,j-1,m ,n ))&&

                                        (inf_val <= ImLevels(i,j-1,m+1,n ))&&

                                        (inf_val <= ImLevels(i,j-1,m-1,n+1))&&

                                        (inf_val <= ImLevels(i,j-1,m ,n+1))&&

                                        (inf_val <= ImLevels(i,j-1,m+1,n+1))&& //底层的

小尺度9

                                        (inf_val <= ImLevels(i,j,m-1,n-1))&&

```



```

        (inf_val <= ImLevels(i,j,m ,n-1))&&
        (inf_val <= ImLevels(i,j,m+1,n-1))&&
        (inf_val <= ImLevels(i,j,m-1,n ))&&
        (inf_val <= ImLevels(i,j,m+1,n ))&&
        (inf_val <= ImLevels(i,j,m-1,n+1))&&
        (inf_val <= ImLevels(i,j,m ,n+1))&&
        (inf_val <= ImLevels(i,j,m+1,n+1))&& //当前层

```

8

```

        (inf_val <= ImLevels(i,j+1,m-1,n-1))&&
        (inf_val <= ImLevels(i,j+1,m ,n-1))&&
        (inf_val <= ImLevels(i,j+1,m+1,n-1))&&
        (inf_val <= ImLevels(i,j+1,m-1,n ))&&
        (inf_val <= ImLevels(i,j+1,m ,n ))&&
        (inf_val <= ImLevels(i,j+1,m+1,n ))&&
        (inf_val <= ImLevels(i,j+1,m-1,n+1))&&
        (inf_val <= ImLevels(i,j+1,m ,n+1))&&
        (inf_val <= ImLevels(i,j+1,m+1,n+1)) //下一层

```

大尺度9

```

    ) ||
    ( (inf_val >= ImLevels(i,j-1,m-1,n-1))&&
      (inf_val >= ImLevels(i,j-1,m ,n-1))&&
      (inf_val >= ImLevels(i,j-1,m+1,n-1))&&
      (inf_val >= ImLevels(i,j-1,m-1,n ))&&
      (inf_val >= ImLevels(i,j-1,m ,n ))&&
      (inf_val >= ImLevels(i,j-1,m+1,n ))&&
      (inf_val >= ImLevels(i,j-1,m-1,n+1))&&
      (inf_val >= ImLevels(i,j-1,m ,n+1))&&
      (inf_val >= ImLevels(i,j-1,m+1,n+1))&&

      (inf_val >= ImLevels(i,j,m-1,n-1))&&
      (inf_val >= ImLevels(i,j,m ,n-1))&&

```

```

        (inf_val >= ImLevels(i,j,m+1,n-1))&&
        (inf_val >= ImLevels(i,j,m-1,n ))&&
        (inf_val >= ImLevels(i,j,m+1,n ))&&
        (inf_val >= ImLevels(i,j,m-1,n+1))&&
        (inf_val >= ImLevels(i,j,m ,n+1))&&
        (inf_val >= ImLevels(i,j,m+1,n+1))&&

        (inf_val >= ImLevels(i,j+1,m-1,n-1))&&
        (inf_val >= ImLevels(i,j+1,m ,n-1))&&
        (inf_val >= ImLevels(i,j+1,m+1,n-1))&&
        (inf_val >= ImLevels(i,j+1,m-1,n ))&&
        (inf_val >= ImLevels(i,j+1,m ,n ))&&
        (inf_val >= ImLevels(i,j+1,m+1,n ))&&
        (inf_val >= ImLevels(i,j+1,m-1,n+1))&&
        (inf_val >= ImLevels(i,j+1,m ,n+1))&&
        (inf_val >= ImLevels(i,j+1,m+1,n+1))
    ) ) //2、满足26个中极值点
    {
        //此处可存储

        //然后必须具有明显的显著性，即必须大于
        CONTRAST_THRESHOLD=0.02

        if ( fabs(ImLevels(i,j,m,n))>= CONTRAST_THRESHOLD )
        {
            //最后显著处的特征点必须具有足够的曲率比，
            CURVATURE_THRESHOLD=10.0，首先计算Hessian矩阵

            // Compute the entries of the Hessian matrix at

            /*
            1    0    -1
            0    0    0
            -1   0    1          *0.25
            */

```

```

// Compute the trace and the determinant of the
//Tr_H = Dxx + Dyy;
//Det_H = Dxx*Dyy - Dxy^2;
float Dxx,Dyy,Dxy,Tr_H,Det_H,curvature_ratio;
Dxx = ImLevels(i,j,m,n-1) + ImLevels(i,j,m,n+1)-2.0*ImLevels(i,j,m,n);
Dyy = ImLevels(i,j,m-1,n) + ImLevels(i,j,m+1,n)-2.0*ImLevels(i,j,m,n);
Dxy = ImLevels(i,j,m-1,n-1) + ImLevels(i,j,m+1,n+1) - ImLevels(i,j,m+1,n-1) - ImLevels(i,j,m-1,n+1);
Tr_H = Dxx + Dyy;
Det_H = Dxx*Dyy - Dxy*Dxy;
// Compute the ratio of the principal curvatures
curvature_ratio = (1.0*Tr_H*Tr_H)/Det_H;
if ( (Det_H>=0.0) && (curvature_ratio <= curvature_ratio_thres) )
最后得到最具有显著性特征的特征点
{
//将其存储起来，以计算后面的特征描述字
keypoint_count++;
Keypoint k;
/* Allocate memory for the keypoint. */
k = (Keypoint) malloc(sizeof(struct Keypoint));
k->next = keypoints;
keypoints = k;
k->row = m*(GaussianPyr[i].subsample);
k->col =n*(GaussianPyr[i].subsample);
k->sy = m; //行
k->sx = n; //列
k->octave=i;
k->level=j;
k->scale = (GaussianPyr[i].Octave)
[j].absolute_sigma;
} //if >curvature_thresh

```

```

        }//if >contrast
    }//if inf value
} //if non zero
} //if >contrast
} //for concrete image level col
} //for levels
} //for octaves
return keypoint_count;
}

```

//在图像中，显示SIFT特征点的位置

```

void DisplayKeypointLocation(IplImage* image, ImageOctaves *GaussianPyr)
{
    Keypoint p = keypoints; // p指向第一个结点
    while(p) // 没到表尾
    {
        cvLine( image, cvPoint((int)((p->col)-3),(int)(p->row)),
                cvPoint((int)((p->col)+3),(int)(p->row)), CV_RGB(255,255,0),
                1, 8, 0 );
        cvLine( image, cvPoint((int)(p->col),(int)((p->row)-3)),
                cvPoint((int)(p->col),(int)((p->row)+3)), CV_RGB(255,255,0),
                1, 8, 0 );
        // cvCircle(image,cvPoint((uchar)(p->col),(uchar)(p->row)),
        // (int)((GaussianPyr[p->octave].Octave)[p->level].absolute_sigma),
        // CV_RGB(255,0,0),1,8,0);
        p=p->next;
    }
}

```

// Compute the gradient direction and magnitude of the gaussian pyramid images

```

void ComputeGrad_DirecandMag(int numoctaves, ImageOctaves *GaussianPyr)
{
    // ImageOctaves *mag_thresh ;

    mag_pyr=(ImageOctaves*) malloc( numoctaves * sizeof(ImageOctaves) );

    grad_pyr=(ImageOctaves*) malloc( numoctaves * sizeof(ImageOctaves) );

    // float sigma=( GaussianPyr[0].Octave
[SCALESPEROCTAVE+2].absolute_sigma ) / GaussianPyr[0].subsample;

    // int dim = (int) (max(3.0f, 2 * GAUSSKERN *sigma + 1.0f)*0.5+0.5);

#define ImLevels(OCTAVE,LEVEL,ROW,COL) ((float *)
(GaussianPyr[(OCTAVE)].Octave[(LEVEL)].Level-
>data.fl + GaussianPyr[(OCTAVE)].Octave[(LEVEL)].Level->step/sizeof(float) *
(ROW)))[(COL)]

    for (int i=0; i<numoctaves; i++)
    {
        mag_pyr[i].Octave= (ImageLevels*) malloc( (SCALESPEROCTAVE) * sizeof(ImageLe
        grad_pyr[i].Octave= (ImageLevels*) malloc( (SCALESPEROCTAVE) * sizeof(ImageL
        for(int j=1;j<SCALESPEROCTAVE+1;j++)//取中间的scaleperoctave个层
        {
            CvMat *Mag = cvCreateMat(GaussianPyr[i].row, GaussianPyr[i].col, CV_32FC
            CvMat *Ori = cvCreateMat(GaussianPyr[i].row, GaussianPyr[i].col, CV_32FC
            CvMat *tempMat1 = cvCreateMat(GaussianPyr[i].row, GaussianPyr[i].col, CV
            CvMat *tempMat2 = cvCreateMat(GaussianPyr[i].row, GaussianPyr[i].col, CV
            cvZero(Mag);
            cvZero(Ori);
            cvZero(tempMat1);
            cvZero(tempMat2);

#define MAG(ROW,COL) ((float *) (Mag->data.fl + Mag->step/sizeof(float) *(ROW)))
[(COL)]

#define ORI(ROW,COL) ((float *) (Ori->data.fl + Ori->step/sizeof(float) *(ROW)))
[(COL)]

#define TEMPMAT1(ROW,COL) ((float *) (tempMat1->data.fl + tempMat1-
>step/sizeof(float) *(ROW)))[(COL)]

#define TEMPMAT2(ROW,COL) ((float *) (tempMat2->data.fl + tempMat2-
>step/sizeof(float) *(ROW)))[(COL)]

```

```

        for (int m=1;m<(GaussianPyr[i].row-1);m++)
            for(int n=1;n<(GaussianPyr[i].col-1);n++)
            {
                //计算幅值
                TEMPMAT1(m,n) = 0.5*( ImLevels(i,j,m,n+1)-ImLevels(i,j,m,n-
1) ); //dx
                TEMPMAT2(m,n) = 0.5*( ImLevels(i,j,m+1,n)-ImLevels(i,j,m-
1,n) ); //dy
                MAG(m,n) = sqrt(TEMPMAT1(m,n)*TEMPMAT1(m,n)+TEMPMAT2(m,n)*TEMPMA
                //计算方向
                ORI(m,n) =atan( TEMPMAT2(m,n)/TEMPMAT1(m,n) );
                if (ORI(m,n)==CV_PI)
                    ORI(m,n)=-CV_PI;
            }
            ((mag_pyr[i].Octave)[j-1]).Level=Mag;
            ((grad_pyr[i].Octave)[j-1]).Level=Ori;
            cvReleaseMat(&tempMat1);
            cvReleaseMat(&tempMat2);
        }//for levels
    }//for octaves
}

```

SIFT算法第四步

```

//SIFT算法第四步：计算各个特征点的主方向，确定主方向

void AssignTheMainOrientation(int numoctaves, ImageOctaves *GaussianPyr,ImageOctaves
{
    // Set up the histogram bin centers for a 36 bin histogram.
    int num_bins = 36;
    float hist_step = 2.0*PI/num_bins;
    float hist_orient[36];
    for (int i=0;i<36;i++)

```

```

        hist_orient[i]=-PI+i*hist_step;

        float sigma1=( ((GaussianPyr[0].Octave)
[SCALESPEROCTAVE].absolute_sigma) ) / (GaussianPyr[0].subsample); //SCALESPEROCTAVE+2

        int zero_pad = (int) (max(3.0f, 2 * GAUSSKERN *sigma1 + 1.0f)*0.5+0.5);

        //Assign orientations to the keypoints.

#define ImLevels(OCTAVES,LEVELS,ROW,COL) ((float *)
((GaussianPyr[(OCTAVES)].Octave[(LEVELS)].Level)-
>data.fl + (GaussianPyr[(OCTAVES)].Octave[(LEVELS)].Level)->step/sizeof(float) *
(ROW)))[(COL)]

        int keypoint_count = 0;

        Keypoint p = keypoints; // p指向第一个结点

        while(p) // 没到表尾
        {
            int i=p->octave;

            int j=p->level;

            int m=p->sy;    //行

            int n=p->sx;    //列

            if ((m>=zero_pad)&&(m<GaussianPyr[i].row-zero_pad)&&
                (n>=zero_pad)&&(n<GaussianPyr[i].col-zero_pad) )
            {
                float sigma=( ((GaussianPyr[i].Octave)
[j].absolute_sigma) ) / (GaussianPyr[i].subsample);

                //产生二维高斯模板

                CvMat* mat = GaussianKernel2D( sigma );

                int dim=(int)(0.5 * (mat->rows));

                //分配用于存储Patch幅值和方向的空间

#define MAT(ROW,COL) ((float *) (mat->data.fl + mat->step/sizeof(float) *(ROW)))
[(COL)]

                //声明方向直方图变量

                double* orienthist = (double *) malloc(36 * sizeof(double));

                for ( int sw = 0 ; sw < 36 ; ++sw)

```

```

    {
        orienthist[sw]=0.0;
    }

    //在特征点的周围统计梯度方向

    for (int x=m-dim,mm=0;x<=(m+dim);x++,mm++)
        for(int y=n-dim,nn=0;y<=(n+dim);y++,nn++)
        {
            //计算特征点处的幅值

            double dx = 0.5*(ImLevels(i,j,x,y+1)-ImLevels(i,j,x,y-1)); //dx

            double dy = 0.5*(ImLevels(i,j,x+1,y)-ImLevels(i,j,x-1,y)); //dy

            double mag = sqrt(dx*dx+dy*dy); //mag

            //计算方向

            double Ori =atan( 1.0*dy/dx );

            int binIdx = FindClosestRotationBin(36, Ori);
            得到离现有方向最近的直方块

            orienthist[binIdx] = orienthist[binIdx] + 1.0* mag * MAT(mm,nn);
            利用高斯加权累加进直方图相应的块

        }

        // Find peaks in the orientation histogram using nonmax suppression.

        AverageWeakBins (orienthist, 36);

        // find the maximum peak in gradient orientation

        double maxGrad = 0.0;

        int maxBin = 0;

        for (int b = 0 ; b < 36 ; ++b)
        {
            if (orienthist[b] > maxGrad)
            {
                maxGrad = orienthist[b];

                maxBin = b;
            }
        }
    }

```



```

    }

    // First determine the real interpolated peak high at the maximum bi
    // position, which is guaranteed to be an absolute peak.

    double maxPeakValue=0.0;

    double maxDegreeCorrection=0.0;

    if ( (InterpolateOrientation ( orienthist[maxBin == 0 ? (36 - 1) : (
        orienthist[maxBin], orienthist[(maxBin + 1) % 36],
        &maxDegreeCorrection, &maxPeakValue)) == false)

        printf("BUG: Parabola fitting broken");

    // Now that we know the maximum peak value, we can find other keypoi
    // orientations, which have to fulfill two criterias:

    //

    // 1. They must be a local peak themselves. Else we might add a ver
    //     similar keypoint orientation twice (imagine for example the
    //     values: 0.4 1.0 0.8, if 1.0 is maximum peak, 0.8 is still add
    //     with the default threshold, but the maximum peak orientation
    //     was already added).

    // 2. They must have at least peakRelThresh times the maximum peak
    //     value.

    bool binIsKeypoint[36];

    for ( b = 0 ; b < 36 ; ++b)

    {

        binIsKeypoint[b] = false;

        // The maximum peak of course is

        if (b == maxBin)

        {

            binIsKeypoint[b] = true;

            continue;

        }

        // Local peaks are, too, in case they fulfill the threshold

```

```

        if (orienthist[b] < (peakRelThresh * maxPeakValue))
            continue;

        int leftI = (b == 0) ? (36 - 1) : (b - 1);

        int rightI = (b + 1) % 36;

        if (orienthist[b] <= orienthist[leftI] || orienthist[b] <= orienthist[rightI])
            continue; // no local peak

        binIsKeypoint[b] = true;
    }

    // find other possible locations

    double oneBinRad = (2.0 * PI) / 36;

    for ( b = 0 ; b < 36 ; ++b)
    {
        if (binIsKeypoint[b] == false)
            continue;

        int bLeft = (b == 0) ? (36 - 1) : (b - 1);

        int bRight = (b + 1) % 36;

        // Get an interpolated peak direction and value guess.

        double peakValue;

        double degreeCorrection;

        double maxPeakValue, maxDegreeCorrection;

        if (InterpolateOrientation ( orienthist[maxBin == 0 ? (36 - 1) : maxBin],
                                     orienthist[(maxBin + 1) % 36],
                                     &degreeCorrection, &peakValue) == false)
        {
            printf("BUG: Parabola fitting broken");
        }

        double degree = (b + degreeCorrection) * oneBinRad - PI;

        if (degree < -PI)

```

```

        degree += 2.0 * PI;

    else if (degree > PI)

        degree -= 2.0 * PI;

    //存储方向，可以直接利用检测到的链表进行该步主方向的指定;

    //分配内存重新存储特征点

    Keypoint k;

    /* Allocate memory for the keypoint Descriptor. */

    k = (Keypoint) malloc(sizeof(struct KeypointSt));

    k->next = keyDescriptors;

    keyDescriptors = k;

    k->descrip = (float*)malloc(LEN * sizeof(float));

    k->row = p->row;

    k->col = p->col;

    k->sy = p->sy;    //行

    k->sx = p->sx;    //列

    k->octave = p->octave;

    k->level = p->level;

    k->scale = p->scale;

    k->ori = degree;

    k->mag = peakValue;

    }//for

    free(orienthist);

}

p=p->next;

}

}

```

//寻找与方向直方图最近的柱，确定其index

```

int FindClosestRotationBin (int binCount, float angle)

{

    angle += CV_PI;

```

```

    angle /= 2.0 * CV_PI;

    // calculate the aligned bin
    angle *= binCount;

    int idx = (int) angle;

    if (idx == binCount)
        idx = 0;

    return (idx);
}

// Average the content of the direction bins.
void AverageWeakBins (double* hist, int binCount)
{
    // TODO: make some tests what number of passes is the best. (its clear
    // one is not enough, as we may have something like
    // ( 0.4, 0.4, 0.3, 0.4, 0.4 ))
    for (int sn = 0 ; sn < 2 ; ++sn)
    {
        double firstE = hist[0];
        double last = hist[binCount-1];

        for (int sw = 0 ; sw < binCount ; ++sw)
        {
            double cur = hist[sw];

            double next = (sw == (binCount - 1)) ? firstE : hist[(sw + 1) % binCount];

            hist[sw] = (last + cur + next) / 3.0;

            last = cur;
        }
    }
}

// Fit a parabol to the three points (-1.0 ; left), (0.0 ; middle) and

```

```

// (1.0 ; right).
// Formulas:
//  $f(x) = a(x - c)^2 + b$ 
// c is the peak offset (where  $f'(x)$  is zero), b is the peak value.
// In case there is an error false is returned, otherwise a correction
// value between [-1 ; 1] is returned in 'degreeCorrection', where -1
// means the peak is located completely at the left vector, and -0.5 just
// in the middle between left and middle and > 0 to the right side. In
// 'peakValue' the maximum estimated peak value is stored.
bool InterpolateOrientation (double left, double middle, double right, double *degree
{
    double a = ((left + right) - 2.0 * middle) / 2.0;    //抛物线捏合系数a
    // degreeCorrection = peakValue = Double.NaN;

    // Not a parabol
    if (a == 0.0)
        return false;

    double c = (((left - middle) / a) - 1.0) / 2.0;
    double b = middle - c * c * a;

    if (c < -0.5 || c > 0.5)
        return false;

    *degreeCorrection = c;
    *peakValue = b;
    return true;
}

//显示特征点处的主方向
void DisplayOrientation (IplImage* image, ImageOctaves *GaussianPyr)
{
    Keypoint p = keyDescriptors; // p指向第一个结点
    while(p) // 没到表尾

```

```

{
    float scale=(GaussianPyr[p->octave].Octave)[p->level].absolute_sigma;
    float autoscale = 3.0;
    float uu=autoscale*scale*cos(p->ori);
    float vv=autoscale*scale*sin(p->ori);
    float x=(p->col)+uu;
    float y=(p->row)+vv;
    cvLine( image, cvPoint((int)(p->col),(int)(p->row)),
            cvPoint((int)x,(int)y), CV_RGB(255,255,0),
            1, 8, 0 );
    // Arrow head parameters
    float alpha = 0.33; // Size of arrow head relative to the length of the vect
    float beta = 0.33; // Width of the base of the arrow head relative to the l

    float xx0= (p->col)+uu-alpha*(uu+beta*vv);
    float yy0= (p->row)+vv-alpha*(vv-beta*uu);
    float xx1= (p->col)+uu-alpha*(uu-beta*vv);
    float yy1= (p->row)+vv-alpha*(vv+beta*uu);
    cvLine( image, cvPoint((int)xx0,(int)yy0),
            cvPoint((int)x,(int)y), CV_RGB(255,255,0),
            1, 8, 0 );
    cvLine( image, cvPoint((int)xx1,(int)yy1),
            cvPoint((int)x,(int)y), CV_RGB(255,255,0),
            1, 8, 0 );
    p=p->next;
}
}

```

SIFT算法第五步

SIFT算法第五步：抽取各个特征点处的特征描述字，确定特征点的描述

字。描述字是Patch网格内梯度方向的描述，旋转网格到主方向，插值得到网格处梯度值。

一个特征点可以用 $2 \times 2 \times 8 = 32$ 维的向量，也可以用 $4 \times 4 \times 8 = 128$ 维的向量更精确的进行描述。

```
void ExtractFeatureDescriptors(int numoctaves, ImageOctaves *GaussianPyr)
{
    // The orientation histograms have 8 bins

    float orient_bin_spacing = PI/4;

    float orient_angles[8]={-PI, -PI+orient_bin_spacing, -PI*0.5, -
orient_bin_spacing,
        0.0, orient_bin_spacing, PI*0.5,  PI+orient_bin_spacing};

    //产生描述字中心各点坐标

    float *feat_grid=(float *) malloc( 2*16 * sizeof(float));

    for (int i=0;i<GridSpacing;i++)
    {
        for (int j=0;j<2*GridSpacing;++j,++j)
        {
            feat_grid[i*2*GridSpacing+j]=-6.0+i*GridSpacing;
            feat_grid[i*2*GridSpacing+j+1]=-6.0+0.5*j*GridSpacing;
        }
    }

    //产生网格

    float *feat_samples=(float *) malloc( 2*256 * sizeof(float));

    for ( i=0;i<4*GridSpacing;i++)
    {
        for (int j=0;j<8*GridSpacing;j+=2)
        {
            feat_samples[i*8*GridSpacing+j]=-(2*GridSpacing-0.5)+i;
            feat_samples[i*8*GridSpacing+j+1]=-(2*GridSpacing-0.5)+0.5*j;
        }
    }
}
```

```

float feat_window = 2*GridSpacing;

Keypoint p = keyDescriptors; // p指向第一个结点

while(p) // 没到表尾
{
    float scale=(GaussianPyr[p->octave].Octave)[p->level].absolute_sigma;

    float sine = sin(p->ori);
    float cosine = cos(p->ori);
    //计算中心点坐标旋转之后的位置
    float *featcenter=(float *) malloc( 2*16 * sizeof(float));
    for (int i=0;i<GridSpacing;i++)
    {
        for (int j=0;j<2*GridSpacing;j+=2)
        {
            float x=feat_grid[i*2*GridSpacing+j];
            float y=feat_grid[i*2*GridSpacing+j+1];
            featcenter[i*2*GridSpacing+j]=((cosine * x + sine * y) + p->sx);
            featcenter[i*2*GridSpacing+j+1]=((-sine * x + cosine * y) + p->sy);
        }
    }

    // calculate sample window coordinates (rotated along keypoint)
    float *feat=(float *) malloc( 2*256 * sizeof(float));
    for ( i=0;i<64*GridSpacing;i++,i++)
    {
        float x=feat_samples[i];
        float y=feat_samples[i+1];
        feat[i]=((cosine * x + sine * y) + p->sx);
        feat[i+1]=((-sine * x + cosine * y) + p->sy);
    }
}

```



```

//Initialize the feature descriptor.

float *feat_desc = (float *) malloc( 128 * sizeof(float));

for (i=0;i<128;i++)
{
    feat_desc[i]=0.0;

    // printf("%f ",feat_desc[i]);

}

//printf("\n");

for ( i=0;i<512;++i,++i)
{
    float x_sample = feat[i];
    float y_sample = feat[i+1];

    // Interpolate the gradient at the sample position

    /*
    0   1   0
    1   *   1
    0   1   0   具体插值策略如图示

    */

    float sample12=getPixelBI(((GaussianPyr[p->octave].Octave)[p->level]).Level, x_sample, y_sample-1);

    float sample21=getPixelBI(((GaussianPyr[p->octave].Octave)[p->level]).Level, x_sample-1, y_sample);

    float sample22=getPixelBI(((GaussianPyr[p->octave].Octave)[p->level]).Level, x_sample, y_sample);

    float sample23=getPixelBI(((GaussianPyr[p->octave].Octave)[p->level]).Level, x_sample+1, y_sample);

    float sample32=getPixelBI(((GaussianPyr[p->octave].Octave)[p->level]).Level, x_sample, y_sample+1);

    //float diff_x = 0.5*(sample23 - sample21);

    //float diff_y = 0.5*(sample32 - sample12);

    float diff_x = sample23 - sample21;

    float diff_y = sample32 - sample12;

    float mag_sample = sqrt( diff_x*diff_x + diff_y*diff_y );

```

```

        float grad_sample = atan( diff_y / diff_x );

        if(grad_sample == CV_PI)

            grad_sample = -CV_PI;

        // Compute the weighting for the x and y dimensions.

        float *x_wght=
(float *) malloc( GridSpacing * GridSpacing * sizeof(float));

        float *y_wght=
(float *) malloc( GridSpacing * GridSpacing * sizeof(float));

        float *pos_wght=
(float *) malloc( 8*GridSpacing * GridSpacing * sizeof(float));;

        for (int m=0;m<32;++m,++m)

        {

            float x=featcenter[m];

            float y=featcenter[m+1];

            x_wght[m/2] = max(1 - (fabs(x - x_sample)*1.0/GridSpacing), 0);

            y_wght[m/2] = max(1 - (fabs(y - y_sample)*1.0/GridSpacing), 0);


        }

        for ( m=0;m<16;++m)

            for (int n=0;n<8;++n)

                pos_wght[m*8+n]=x_wght[m]*y_wght[m];

        free(x_wght);

        free(y_wght);

        //计算方向的加权，首先旋转梯度场到主方向，然后计算差异

        float diff[8],orient_wght[128];

        for ( m=0;m<8;++m)

        {

            float angle = grad_sample-(p->ori)-orient_angles[m]+CV_PI;

            float temp = angle / (2.0 * CV_PI);

            angle -= (int)(temp) * (2.0 * CV_PI);

            diff[m]= angle - CV_PI;

        }

```

```

        // Compute the gaussian weighting.

        float x=p->sx;

        float y=p->sy;

        float g = exp(-((x_sample-x)*(x_sample-x)+(y_sample-y)*(y_sample-
y))/(2*feat_window*feat_window))/(2*CV_PI*feat_window*feat_window);

        for ( m=0;m<128;++m)
        {
            orient_wght[m] = max((1.0 - 1.0*fabs(diff[m%8])/orient_bin_spacing),
            feat_desc[m] = feat_desc[m] + orient_wght[m]*pos_wght[m]*g*mag_sampl
        }
        free(pos_wght);
    }
    free(feat);
    free(featcenter);
    float norm=GetVecNorm( feat_desc, 128);
    for (int m=0;m<128;m++)
    {
        feat_desc[m]/=norm;
        if (feat_desc[m]>0.2)
            feat_desc[m]=0.2;
    }
    norm=GetVecNorm( feat_desc, 128);
    for ( m=0;m<128;m++)
    {
        feat_desc[m]/=norm;
        printf("%f ",feat_desc[m]);
    }
    printf("/n");
    p->descrip = feat_desc;
    p=p->next;

```

```

    }

    free(feat_grid);

    free(feat_samples);

}

```

//为了显示图象金字塔，而作的图像水平拼接

```

CvMat* MosaicHorizen( CvMat* im1, CvMat* im2 )
{
    int row,col;

    CvMat *mosaic = cvCreateMat( max(im1->rows,im2->rows),(im1->cols+im2->cols),CV_32FC1);

#define Mosaic(ROW,COL) ((float*)(mosaic->data.fl + mosaic->step/sizeof(float)* (ROW)))[(COL)]

#define Im11Mat(ROW,COL) ((float*)(im1->data.fl + im1->step/sizeof(float)* (ROW)))[(COL)]

#define Im22Mat(ROW,COL) ((float*)(im2->data.fl + im2->step/sizeof(float)* (ROW)))[(COL)]

    cvZero(mosaic);

    /* Copy images into mosaic1. */

    for ( row = 0; row < im1->rows; row++)

        for ( col = 0; col < im1->cols; col++)

            Mosaic(row,col)=Im11Mat(row,col) ;

    for ( row = 0; row < im2->rows; row++)

        for ( col = 0; col < im2->cols; col++)

            Mosaic(row, (col+im1->cols) )= Im22Mat(row,col) ;

    return mosaic;
}

```

//为了显示图象金字塔，而作的图像垂直拼接

```

CvMat* MosaicVertical( CvMat* im1, CvMat* im2 )
{
    int row,col;

    CvMat *mosaic = cvCreateMat(im1->rows+im2->rows,max(im1->cols,im2->cols),CV_32FC1);
}

```

```

>cols), CV_32FC1);

#define Mosaic(ROW,COL) ((float*)(mosaic->data.fl + mosaic->step/sizeof(float)*
(ROW)))[(COL)]

#define Im11Mat(ROW,COL) ((float*)(im1->data.fl + im1->step/sizeof(float)*
(ROW)))[(COL)]

#define Im22Mat(ROW,COL) ((float*)(im2->data.fl + im2->step/sizeof(float)*
(ROW)))[(COL)]

    cvZero(mosaic);

    /* Copy images into mosaic1. */
    for ( row = 0; row < im1->rows; row++)
        for ( col = 0; col < im1->cols; col++)
            Mosaic(row,col)= Im11Mat(row,col) ;

    for ( row = 0; row < im2->rows; row++)
        for ( col = 0; col < im2->cols; col++)
            Mosaic((row+im1->rows),col)=Im22Mat(row,col) ;

    return mosaic;
}

```

ok, 为了版述清晰, 再贴一下上文所述的主函数（注, 上文已贴出, 此是为了版述清晰, 重复造轮）：

```

int main ( void )
{
    //声明当前帧IplImage指针

    IplImage* src = NULL ;

    IplImage* image1 = NULL ;

    IplImage* grey_im1 = NULL ;

    IplImage* DoubleSizeImage = NULL ;

    IplImage* mosaic1 = NULL ;

    IplImage* mosaic2 = NULL ;

```

```

CvMat* mosaicHorizen1 = NULL ;

CvMat* mosaicHorizen2 = NULL ;

CvMat* mosaicVertical1 = NULL ;


CvMat* image1Mat = NULL ;

CvMat* tempMat= NULL ;


ImageOctaves *Gaussianpyr;

int rows,cols;


# define Im1Mat(ROW,COL) ((float *) (image1Mat->data.fl + image1Mat->step/sizeof(float) *(ROW)))[(COL)]


//灰度图像像素的数据结构

# define Im1B(ROW,COL) ((uchar*)(image1->imageData + image1->widthStep*(ROW)))[(COL)*3]

# define Im1G(ROW,COL) ((uchar*)(image1->imageData + image1->widthStep*(ROW)))[(COL)*3+1]

# define Im1R(ROW,COL) ((uchar*)(image1->imageData + image1->widthStep*(ROW)))[(COL)*3+2]


storage = cvCreateMemStorage( 0 );


//读取图片

if ( (src = cvLoadImage( "street1.jpg" , 1 )) == 0 )
// test1.jpg einstein.pgm back1.bmp

return - 1 ;


//为图像分配内存

image1 = cvCreateImage(cvSize(src->width, src->height), IPL_DEPTH_8U, 3 );
grey_im1 = cvCreateImage(cvSize(src->width, src->height), IPL_DEPTH_8U, 1
);

DoubleSizeImage = cvCreateImage(cvSize( 2 *(src->width), 2 *(src->

```

```
>height)), IPL_DEPTH_8U, 3 );
```

```
//为图像阵列分配内存, 假设两幅图像的大小相同, tempMat跟随image1的大小
```

```
image1Mat = cvCreateMat(src->height, src->width, CV_32FC1);
```

```
//转化成单通道图像再处理
```

```
cvCvtColor(src, grey_im1, CV_BGR2GRAY);
```

```
//转换进入Mat数据结构, 图像操作使用的是浮点型操作
```

```
cvConvert(grey_im1, image1Mat);
```

```
double t = ( double )cvGetTickCount();
```

```
//图像归一化
```

```
cvConvertScale( image1Mat, image1Mat, 1.0 / 255 , 0 );
```

```
int dim = min(image1Mat->rows, image1Mat->cols);
```

```
numoctaves = ( int ) ( log ( ( double ) dim) / log ( 2.0 )) - 2 ; //金字塔阶数
```

```
numoctaves = min(numoctaves, MAXOCTAVES);
```

```
//SIFT算法第一步, 预滤波除噪声, 建立金字塔底层
```

```
tempMat = ScaleInitImage(image1Mat) ;
```

```
//SIFT算法第二步, 建立Guassain金字塔和DOG金字塔
```

```
Gaussianpyr = BuildGaussianOctaves(tempMat) ;
```

```
t = ( double )cvGetTickCount() - t;
```

```
printf ( "the time of build Gaussian pyramid and DOG pyramid is %.1f/n" , t/(cvGetTickFrequency()* 1000. ) );
```

```
# define ImLevels(OCTAVE, LEVEL, ROW, COL) ((float *)  
(Gaussianpyr[(OCTAVE)].Octave[(LEVEL)].Level->data.fl + Gaussianpyr[(OCTAVE)].Octave[(LEVEL)].Level->step/sizeof(float) *  
(ROW)))[(COL)]
```

```
//显示高斯金字塔
```

```
for ( int i= 0 ; i<numoctaves;i++)
```

```

{
    if (i== 0 )
    {
        mosaicHorizen1=MosaicHorizen( (Gaussianpyr[ 0 ].Octave)[ 0
].Level, (Gaussianpyr[ 0 ].Octave)[ 1 ].Level );

        for ( int j= 2 ;j<SCALESPEROCTAVE+ 3 ;j++)

            mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (Gaussianpyr[ 0
].Octave)[j].Level );

        for ( j= 0 ;j<NUMSIZE;j++)

            mosaicHorizen1=halfSizeImage(mosaicHorizen1);

    }

    else if (i== 1 )
    {
        mosaicHorizen2=MosaicHorizen( (Gaussianpyr[ 1 ].Octave)[ 0
].Level, (Gaussianpyr[ 1 ].Octave)[ 1 ].Level );

        for ( int j= 2 ;j<SCALESPEROCTAVE+ 3 ;j++)

            mosaicHorizen2=MosaicHorizen( mosaicHorizen2, (Gaussianpyr[ 1
].Octave)[j].Level );

        for ( j= 0 ;j<NUMSIZE;j++)

            mosaicHorizen2=halfSizeImage(mosaicHorizen2);

        mosaicVertical1=MosaicVertical( mosaicHorizen1, mosaicHorizen2 );

    }

    else
    {
        mosaicHorizen1=MosaicHorizen( (Gaussianpyr[i].Octave)[ 0
].Level, (Gaussianpyr[i].Octave)[ 1 ].Level );

        for ( int j= 2 ;j<SCALESPEROCTAVE+ 3 ;j++)

            mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (Gaussianpyr[i].Octave
[j].Level );

        for ( j= 0 ;j<NUMSIZE;j++)

            mosaicHorizen1=halfSizeImage(mosaicHorizen1);

        mosaicVertical1=MosaicVertical( mosaicVertical1, mosaicHorizen1 );

    }

}

```



```

    mosaic1 = cvCreateImage(cvSize(mosaicVertical1->width, mosaicVertical1->height), IPL_DEPTH_8U, 1 );

    cvConvertScale( mosaicVertical1, mosaicVertical1, 255.0 , 0 );

    cvConvertScaleAbs( mosaicVertical1, mosaic1, 1 , 0 );


    // cvSaveImage("GaussianPyramid of me.jpg",mosaic1);

    cvNamedWindow( "mosaic1" , 1 );
    cvShowImage( "mosaic1" , mosaic1);
    cvWaitKey( 0 );
    cvDestroyWindow( "mosaic1" );

    //显示DOG金字塔

    for ( i= 0 ; i<numoctaves;i++)
    {
        if (i== 0 )
        {
            mosaicHorizen1=MosaicHorizen( (DOGoctaves[ 0 ].Octave)[ 0 ].Level, (DOGoctaves[ 0 ].Octave)[ 1 ].Level );

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

                mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (DOGoctaves[ 0 ].Octave)[j].Level );

            for ( j= 0 ;j<NUMSIZE;j++)

                mosaicHorizen1=halfSizeImage(mosaicHorizen1);

        }

        else if (i== 1 )
        {
            mosaicHorizen2=MosaicHorizen( (DOGoctaves[ 1 ].Octave)[ 0 ].Level, (DOGoctaves[ 1 ].Octave)[ 1 ].Level );

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

                mosaicHorizen2=MosaicHorizen( mosaicHorizen2, (DOGoctaves[ 1 ].Octave)[j].Level );

            for ( j= 0 ;j<NUMSIZE;j++)

                mosaicHorizen2=halfSizeImage(mosaicHorizen2);

            mosaicVertical1=MosaicVertical( mosaicHorizen1, mosaicHorizen2 );

```

```

    }

    else

    {

        mosaicHorizen1=MosaicHorizen( (DOGoctaves[i].Octave)[ 0
].Level, (DOGoctaves[i].Octave)[ 1 ].Level );

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

            mosaicHorizen1=MosaicHorizen( mosaicHorizen1, (DOGoctaves[i].Octave)
[j].Level );

        for ( j= 0 ;j<NUMSIZE;j++)

            mosaicHorizen1=halfSizeImage(mosaicHorizen1);

        mosaicVertical1=MosaicVertical( mosaicVertical1, mosaicHorizen1 );

    }

}

```

//考虑到DOG金字塔各层图像都会有正负，所以，必须寻找最负的，以将所有图像抬高一个台阶去显示

```

double min_val= 0 ;

double max_val= 0 ;

cvMinMaxLoc( mosaicVertical1, &min_val, &max_val, NULL , NULL , NULL );

if ( min_val< 0.0 )

    cvAddS( mosaicVertical1, cvScalarAll( (- 1.0
)*min_val ), mosaicVertical1, NULL );

    mosaic2 = cvCreateImage(cvSize(mosaicVertical1->width, mosaicVertical1-
>height), IPL_DEPTH_8U, 1 );

    cvConvertScale( mosaicVertical1, mosaicVertical1, 255.0 /(max_val-min_val),
0 );

    cvConvertScaleAbs( mosaicVertical1, mosaic2, 1 , 0 );

    // cvSaveImage("DOGPyramid of me.jpg",mosaic2);

    cvNamedWindow( "mosaic1" , 1 );

    cvShowImage( "mosaic1" , mosaic2);

    cvWaitKey( 0 );

```

//SIFT算法第三步：特征点位置检测，最后确定特征点的位置

```
int keycount=DetectKeypoint(numoctaves, Gaussianpyr);
```

```
printf ( "the keypoints number are %d ;/n" , keycount);
```

```
cvCopy(src,image1, NULL );
```

```
DisplayKeypointLocation( image1 ,Gaussianpyr);
```

```
cvPyrUp( image1, DoubleSizeImage, CV_GAUSSIAN_5x5 );
```

```
cvNamedWindow( "image1" , 1 );
```

```
cvShowImage( "image1" , DoubleSizeImage);
```

```
cvWaitKey( 0 );
```

```
cvDestroyWindow( "image1" );
```

```
//SIFT算法第四步：计算高斯图像的梯度方向和幅值，计算各个特征点的主方向
```

```
ComputeGrad_DirecandMag(numoctaves, Gaussianpyr);
```

```
AssignTheMainOrientation( numoctaves, Gaussianpyr,mag_pyr,grad_pyr);
```

```
cvCopy(src,image1, NULL );
```

```
DisplayOrientation ( image1, Gaussianpyr);
```

```
// cvPyrUp( image1, DoubleSizeImage, CV_GAUSSIAN_5x5 );
```

```
cvNamedWindow( "image1" , 1 );
```

```
// cvResizeWindow("image1", 2*(image1->width), 2*(image1->height) );
```

```
cvShowImage( "image1" , image1);
```

```
cvWaitKey( 0 );
```

```
//SIFT算法第五步：抽取各个特征点处的特征描述字
```

```
ExtractFeatureDescriptors( numoctaves, Gaussianpyr);
```

```
cvWaitKey( 0 );
```

```
//销毁窗口
```

```
cvDestroyWindow( "image1" );
```

```
cvDestroyWindow( "mosaic1" );
```

```
//释放图像
```

```
    cvReleaseImage(&image1);  
    cvReleaseImage(&grey_im1);  
    cvReleaseImage(&mosaic1);  
    cvReleaseImage(&mosaic2);  
    return 0 ;  
}
```

最后，再看一下，运行效果（图中美女为老乡+朋友，何姐08年照）：



完。

updated

有很多朋友都在本文评论下要求要本程序的完整源码包（注：本文代码未贴全，复制粘贴编译肯定诸多错误），但由于时隔太久，这份代码我自己也找不到了，不过，我可以提供一份sift + KD + BBF，且可以编译正确的代码供大家参考学习，有pudn帐号的朋友可以前去下载：

http://www.pudn.com/downloads340/sourcecode/graph/texture_mapping/deta

（没有pudn帐号的同学请加群：169056165，验证信息：sift，至群共享下载），然后用两幅不同的图片做了下匹配（当然，运行结果显示是不匹配的），效果还不错：

<http://weibo.com/1580904460/yDmzAEwcV#1348475194313> ! July、二零一二年十月十一日。

其它

40亿个数中快速查找

题目描述

给40亿个不重复的unsigned int的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那40亿个数当中？

分析与解法

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。

解法一

咱们回到眼前要解决的这个问题，1个unsigned int占用4字节，40亿大约是4G个数，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0-255三个数存在，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 $n*32$ 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。

解法二

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CPU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遍历n个元素取出等概率载个一段时间吧。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ABCCAABCDDAACCCDB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：ABCCAABCDDAACCCDB，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！

问题扩展

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

```
STRING = get input character
```

```
WHILE there are still input characters DO
```

```
    CHARACTER = get input character
```

```
    IF STRING+CHARACTER is in the string table then
```



```
    STRING = STRING+character
ELSE
    output the code for STRING
    add STRING+CHARACTER to the string table
    STRING = CHARACTER
END of IF
END of WHILE
output the code for STRING
```

看过上面的适用范围再联想本题，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 。

hash表算法

第一部分：Top K 算法详解

问题描述

百度面试题：

搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。假设目前有一千万个记录（这些查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。），请你统计最热门的10个查询串，要求使用的内存不能超过1G。

必备知识

什么是哈希表？

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

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

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

问题解析：

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

即，此问题的解决分为以下 俩个步骤：

第一步：Query统计

Query统计有以下俩个方法，可供选择：

1、直接排序法

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

但是题目中有明确要求，那就是内存不能超过1G，一千万条记录，每条记录是255Byte，很显然要占据2.375G内存，这个条件就不满足要求了。

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

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

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

2、Hash Table法

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

题目中说明了，虽然有一千万个Query，但是由于重复度比较高，因此事实上只有300万的Query，每个Query255Byte，因此我们可以考虑把他

们都放进内存中去，而现在只是需要一个合适的数据结构，在这里，Hash Table绝对是我们优先的选择，因为Hash Table的查询速度非常的快，几乎是 $O(1)$ 的时间复杂度。

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

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

第二步：找出Top 10

算法一：普通排序

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

算法二：部分排序

题目要求是求出Top 10，因此我们没有必要对所有的Query都进行排序，我们只需要维护一个10个大小的数组，初始化放入10个Query，按照每个Query的统计次数由大到小排序，然后遍历这300万条记录，每读一条记录就和数组最后一个Query对比，如果小于这个Query，那么继续遍历，否则，将数组中最后一条数据淘汰，加入当前的Query。最后当所有的数据都遍历完毕之后，那么这个数组中的10个Query便是我们要找的Top10了。

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

算法三：堆

在算法二中，我们已经将时间复杂度由 $N\log N$ 优化到 NK ，不得不说这是

一个比较大的改进了，可是有没有更好的办法呢？

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

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

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

具体过程是，堆顶存放的是整个堆中最小的数，现在遍历N个数，把最先遍历到的k个数存放到最小堆中，并假设它们就是我们要找的最大的k个数， $X_1 > X_2 \dots X_{\min}$ (堆顶)，而后遍历后续的N-K个数，一一与堆顶元素进行比较，如果遍历到的 X_i 大于堆顶元素 X_{\min} ，则把 X_i 放入堆中，而后更新整个堆，更新的时间复杂度为 $\log K$ ，如果 $X_i < X_{\min}$ ，则不更新堆，整个过程的复杂度为 $O(K) + O((N-K) \log K) = O(N \log K)$ 。

(堆排序的3D动画演示可以参看此链接：

<http://www.benfrederickson.com/2013/10/10/heap-visualization.html>)

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

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

总结：

至此，算法就完全结束了，经过上述第一步、先用Hash表统计每个Query出现的次数， $O(N)$ ；然后第二步、采用堆数据结构找出Top 10， $N * O(\log K)$ 。所以，我们最终的时间复杂度是： $O(N) +$

$N * O(\log K)$ 。（N为1000万，N'为300万）。如果各位有什么更好的算法，欢迎留言评论。

此外，还可以看下此文第二部分的第二题：[教你如何迅速秒杀掉：99%的海量数据处理面试题](#)

第二部分、Hash表 算法的详细解析

什么是Hash

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

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

数组的特点是：寻址容易，插入和删除困难；而链表的特点是：寻址困难，插入和删除容易。那么我们能不能综合两者的特性，做出一种寻址容易，插入删除也容易的数据结构？答案是肯定的，这就是我们要提起的哈希表，哈希表有多种不同的实现方法，我接下来解释的是最常用的一种方法——拉链法，我们可以理解为“链表的数组”，如图：



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

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

1，除法散列法

最直观的一种，上图使用的就是这种散列法，公式：

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

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

2，平方散列法

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

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

如果数值分配比较均匀的话这种方法能得到不错的结果，但我上面画的那个图的各个元素的值算出来的index都是0——非常失败。也许你还有个问题，value如果很大， $\text{value} * \text{value}$ 不会溢出吗？答案是会的，但我们这个乘法不关心溢出，因为我们根本不是为了获取相乘结果，而是为了获取index。

3，斐波那契（Fibonacci）散列法

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

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

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

对我们常见的32位整数而言，公式：

$\text{index} = (\text{value} * 2654435769) \gg 28$

如果用这种斐波那契散列法的话，那上面的图就变成这样了：



注：用斐波那契散列法调整之后会比原来的取摸散列法好很多。

适用范围

快速查找，删除的基本数据结构，通常需要总数据量可以放入内存。

基本原理及要点

hash函数选择，针对字符串，整数，排列，具体相应的hash方法。

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

扩展

d-left hashing中的d是多个的意思，我们先简化这个问题，看一看2-left hashing。2-left hashing指的是将一个哈希表分成长度相等的两半，分别叫做T1和T2，给T1和T2分别配备一个哈希函数，h1和h2。在存储一个新的key时，同时用两个哈希函数进行计算，得出两个地址h1[key]和h2[key]。这时需要检查T1中的h1[key]位置和T2中的h2[key]位置，哪一个位置已经存储的（有碰撞的）key比较多，然后将新key存储在负载少的位置。如果两边一样多，比如两个位置都为空或者都存储了一个key，就把新key存储在左边的T1子表中，2-left也由此而来。在查找一个key时，必须进行两次hash，同时查找两个位置。

问题实例（海量数据处理）

我们知道hash表在海量数据处理中有着广泛的应用，下面，请看另一道百度面试题：

题目：海量日志数据，提取出某日访问百度次数最多的那个IP。

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

第三部分、最快的Hash表算法

接下来，咱们来具体分析一下一个最快的Hash表算法。

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

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

http://sfsrealm.hopto.org/inside_mopaq/chapter2.htm）：

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

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

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

            seed = (seed * 125 + 3) % 0x2AAAAAB ;

            temp1 = (seed & 0xFFFF) << 0x10 ;
```

```

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

        temp2 = ( seed & 0xFFFF );

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

```

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

```

unsigned long HashString ( char *lpszFileName, unsigned long
dwHashType )
{
    unsigned char *key = ( unsigned char *)lpszFileName;

    unsigned long seed1 = 0x7FED7FED ;
    unsigned long seed2 = 0xEEEEEEEE ;
    int ch;

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

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

    return seed1;
}

```

Blizzard的这个算法是非常高效的，被称为"One-Way Hash"(A one-way

hash is a an algorithm that is constructed in such a way that deriving the original string (set of strings, actually) is virtually impossible)。举个例子，字符串"unitneutralacritter.grp"通过这个算法得到的结果是0xA26067F3。

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

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

一种可能的结构体定义？

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

```
int GetHashTablePos ( har *lpzString, SOMESTRUCTURE *lpTable )
//lpzString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(lpzString); //调用上述函数二，返回要查找字符串lpzString的Hash值。
    int nHashPos = nHash % nTableSize;

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

```

    }

    else
    {
        return - 1 ;
    }
}

```

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

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

```

int  GetHashTablePos (  char  *lpzString, 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

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

补充1、一个简单的hash函数：

```
/*key为一个字符串，nTableLength为哈希表的长度
*该函数得到的hash值分布比较均匀*/

unsigned long  getHashIndex ( const char *key, int nTableLength )
{
    unsigned long nHash = 0 ;

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

    return ( nHash % nTableLength );
}
```

补充2、一个完整测试程序：

哈希表的数组是定长的，如果太大，则浪费，如果太小，体现不出效

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

下面是哈希表尺寸大小的可能取值：

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

以下为该程序的完整源码，已在linux下测试通过：

```
# include <stdio.h>

# include <ctype.h>          //多谢citylove指正。

//cryptTable[]里面保存的是HashString函数里面将会用到的一些数据，在prepareCryptTable
//函数里面初始化

unsigned long cryptTable[ 0x500 ];

//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 );
    }
}

```

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

//在下面GetHashTablePos函数里面调用本函数，其可以取的值为0、1、2；该函数

//返回lpszFileName 字符串的hash值；

```

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

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

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

```

```

    }

    return seed1;
}

//在main中测试argv[1]的三个hash值:
//./hash "arr/units.dat"
//./hash "unit/neutral/acritter.grp"
int main ( int argc, char **argv )
{
    unsigned long ulHashValue;

    int i = 0 ;

    if ( argc != 2 )
    {
        printf ( "please input two arguments/n" );
        return - 1 ;
    }

    /*初始化数组: cryptTable[0x500]*/
    prepareCryptTable();

    /*打印数组cryptTable[0x500]里面的值*/
    for ( ; i < 0x500 ; i++ )
    {
        if ( i % 10 == 0 )
        {
            printf ( "/n" );
        }

        printf ( "%-12X" , cryptTable[i] );
    }
}

```

```
    }

    ulHashValue = HashString( argv[ 1 ], 0 );
    printf ( "/n----%X ----/n" , ulHashValue );

    ulHashValue = HashString( argv[ 1 ], 1 );
    printf ( "----%X ----/n" , ulHashValue );

    ulHashValue = HashString( argv[ 1 ], 2 );
    printf ( "----%X ----/n" , ulHashValue );

    return 0 ;
}
```

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 ($\text{hash()} \bmod N$)：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，这时候会有 $(N-1)/N$ 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；

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；

$\text{hash}(\text{object}) \% N$

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

$\text{hash}(\text{object})\%(N-1)$;

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 $\text{hash}(\text{object})\%(N+1)$;

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。

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。

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。

环形 hash 空间

考虑通常的 hash 算法都是将 value 映射到一个 32 为的 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

把对象映射到**hash** 空间

接下来考虑 4 个对象 object1~object4，通过 hash 函数计算出的 hash 值 key 在环上的分布如图 2 所示。

$\text{hash}(\text{object1}) = \text{key1};$

... ..

$\text{hash}(\text{object4}) = \text{key4};$



图 2 4 个对象的 key 值分布

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。

$\text{hash}(\text{cache A}) = \text{key A};$

... ..

$\text{hash}(\text{cache C}) = \text{key C};$



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。

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；

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。

移除 cache

考虑假设 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 逆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

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

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：

object1->cache A2 ; object2->cache A1 ; object3->cache C1 ; object4->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

倒排索引关键词不重复Hash编码

作者：July、yansha。编程艺术室出品。

出处：结构之法算法之道

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

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

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

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，若有新的文档假如，直接为该文档建立一个新的索引块，挂接在原来索引文件的后面。若有文档删除，则直接找到该文档号文档对应的索引信息，将其直接删除。但是在查询的时候需对所有的文档进行扫描以确保没有遗漏，这样就使得检索时间大大延长，检索效率低下。

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



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：



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，而是会在它预先生成的倒排索引文件中查找和匹配包含这个关键词“结构之法”的所有网页。找到了之后，再按相关性度排序，最终把排序后的结果显示给用户。



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）



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) % 0x2AAAAAB ;
    temp1 = (seed & 0xFFFF) << 0x10 ;

    seed = (seed * 125 + 3) % 0x2AAAAAB ;
    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 = 0xEEEEEEEE ;
    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 an algorithm that is constructed in such a way that deriving the original string (set of strings, actually) is virtually impossible)。举个例子，字符串"unitneutralacritter.grp"通过这个算法得到的结果是0xA26067F3。

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

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

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

//一种可能的结构体定义？
```

函数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 ( har *lpzString, SOMESTRUCTURE *lpTable )

//lpzString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。下面，咱们来看看这个网上流传甚广的暴雪hash算法：

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

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

```
int GetHashTablePos ( char *lpzString, MPQHASHTABLE *lpTable, int nTableSize )
```

```
{
```

```
    const int HASH_OFFSET = 0 , HASH_A = 1 , HASH_B = 2 ;
```

```
    int nHash = HashString( lpzString, HASH_OFFSET );
```

```
    int nHashA = HashString( lpzString, HASH_A );
```

```
    int nHashB = HashString( lpzString, 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] ==
        //          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： 

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' ;

        keysize++;

        itoa(keysize, 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。

从头到尾彻底理解傅里叶变换算法、上

I、本文中阐述离散傅里叶变换方法，是根据此书：The Scientist and Engineer's Guide to Digital Signal Processing, By Steven W. Smith, Ph.D. 而翻译而成的，此书地址：<http://www.dspguide.com/pdfbook.htm>。

II、同时，有相当一部分内容编辑整理自dznlong的博客，也贴出其博客地址，向原创的作者表示致敬：<http://blog.csdn.net/dznlong>。这年头，真正静下心来写来原创文章的人，很少了。

从头到尾彻底理解傅里叶变换算法、上

前言

第一章、傅立叶变换的由来

第二章、实数形式离散傅立叶变换（Real DFT）

从头到尾彻底理解傅里叶变换算法、下

第三章、复数

第四章、复数形式离散傅立叶变换

前言：

“关于傅立叶变换，无论是书本还是在网上可以很容易找到关于傅立叶变换的描述，但是大都是些故弄玄虚的文章，太过抽象，尽是一些让人看了就望而生畏的公式的罗列，让人很难能够从感性上得到理解”---
dznlong

那么，到底什么是傅里叶变换算法列？傅里叶变换所涉及到的公式具体有多复杂列？

傅里叶变换（Fourier transform）是一种线性的积分变换。因其基本思想首先由法国学者傅里叶系统地提出，所以以其名字来命名以示纪念。

哦，傅里叶变换原来就是一种变换而已，只是这种变换是从时间转换为频率的变化。这下，你就知道了，傅里叶就是一种变换，一种什么变换？就是一种从时间到频率的变化或其相互转化。

ok，咱们再来总体了解下傅里叶变换，让各位对其有个总体大概的印象，也顺便看看傅里叶变换所涉及到的公式，究竟有多复杂， 以下就是傅里叶变换的4种变体（摘自，维基百科）：

连续傅里叶变换

一般情况下，若“傅里叶变换”一词不加任何限定语，则指的是“连续傅里叶变换”。连续傅里叶变换将平方可积的函数 $f(t)$ 表示成复指数函数的积分或级数形式。



这是将频率域的函数 $F(\omega)$ 表示为时间域的函数 $f(t)$ 的积分形式。

连续傅里叶变换的逆变换 (inverse Fourier transform)为：



即将时间域的函数 $f(t)$ 表示为频率域的函数 $F(\omega)$ 的积分。

一般可称函数 $f(t)$ 为原函数，而称函数 $F(\omega)$ 为傅里叶变换的像函数，原函数和像函数构成一个傅里叶变换对（transform pair）。除此之外，还有其它型式的变换对，以下两种型式亦常被使用。在通信或是信号处理方面，常以 \square 来代换，而形成新的变换对：



或者是因系数重分配而得到新的变换对：



一种对连续傅里叶变换的推广称为分数傅里叶变换（Fractional Fourier Transform）。分数傅里叶变换(fractional Fourier transform,FRFT)指的就是傅里叶变换(Fourier transform, FT)的广义化。

分数傅里叶变换的物理意义即做傅里叶变换 a 次，其中 a 不一定要为整数；而做了分数傅里叶变换之后，信号或输入函数便会出现于介于时域 (time domain) 与频域 (frequency domain) 之间的分数域 (fractional domain)。

当 $f(t)$ 为偶函数（或奇函数）时，其正弦（或余弦）分量将消亡，而可以称这时的变换为余弦变换（cosine transform）或正弦变换（sine transform）。

另一个值得注意的性质是，当 $f(t)$ 为纯实函数时， $F(-\omega) = F^*(\omega)$ 成立。

傅里叶级数

连续形式的傅里叶变换其实是傅里叶级数 (Fourier series) 的推广，因为积分其实是一种极限形式的求和算子而已。对于周期函数，其傅里叶级数是存在的：



其中 F_n 为复幅度。对于实值函数，函数的傅里叶级数可以写成：



其中 a_n 和 b_n 是实频率分量的幅度。

离散时域傅里叶变换

离散傅里叶变换是离散时间傅里叶变换 (DTFT) 的特例（有时作为后者的近似）。DTFT 在时域上离散，在频域上则是周期的。DTFT 可以被看作是傅里叶级数的逆变换。

离散傅里叶变换

离散傅里叶变换 (DFT)，是连续傅里叶变换在时域和频域上都离散的形式，将时域信号的采样变换为在离散时间傅里叶变换 (DTFT) 频域的采样。在形式上，变换两端（时域和频域上）的序列是有限长的，而实际上这两组序列都应当被认为是离散周期信号的主值序列。即使对有限长的离散信号作 DFT，也应当将其看作经过周期延拓成为周期信号再作变换。在实际应用中通常采用快速傅里叶变换以高效计算 DFT。

为了在科学计算和数字信号处理等领域使用计算机进行傅里叶变换，必须将函数 x_n 定义在离散点而非连续域内，且须满足有限性或周期性条件。这种情况下，使用离散傅里叶变换（*DFT*），将函数 x_n 表示为下面的求和形式：



其中 X_k 是傅里叶幅度。直接使用这个公式计算的计算复杂度为 $O(n^2)$ ，而快速傅里叶变换（*FFT*）可以将复杂度改进为 $O(n \lg n)$ 。（后面会具体阐述*FFT*是如何将复杂度降为 $O(n \lg n)$ 的。）计算复杂度的降低以及数字电路计算能力的发展使得*DFT*成为在信号处理领域十分实用且重要的方法。

下面，比较下上述傅立叶变换的4种变体，

变换	时间	频率
连续傅里叶变换	连续，非周期性	连续，非周期性
傅里叶级数	连续，周期性	离散，非周期性
离散时间傅里叶变换	离散，非周期性	连续，周期性
离散傅里叶变换	离散，周期性	离散，周期性

如上，容易发现：函数在时（频）域的离散对应于其像函数在频（时）域的周期性。反之连续则意味着在对应域的信号的非周期性。也就是说，时间上的离散性对应着频率上的周期性。同时，注意，离散时间傅里叶变换，时间离散，频率不离散，它在频域依然是连续的。

如果，读到此，你不甚明白，大没关系，不必纠结于以上4种变体，继续往下看，你自会豁然开朗。（有什么问题，也恳请提出，或者批评指正）

ok，本文，接下来，由傅里叶变换入手，后重点阐述离散傅里叶变换、快速傅里叶算法，到最后彻底实现**FFT**算法，全篇力求通俗易懂、阅读顺畅，教你从头到尾彻底理解傅里叶变换算法。由于傅里叶变换，也称傅立叶变换，下文所称为傅立叶变换，同一个变换，不同叫法，读者不必感到奇怪。

第一章、傅立叶变换的由来

要理解傅立叶变换，先得知道傅立叶变换是怎么变换的，当然，也需要一定的高等数学基础，最基本的是级数变换，其中傅立叶级数变换是傅立叶变换的基础公式。

一、傅立叶变换的提出

傅立叶是一位法国数学家和物理学家，原名是Jean Baptiste Joseph Fourier(1768-1830)。Fourier于1807年在法国科学学会上发表了一篇论文，论文里描述运用正弦曲线来描述温度分布，论文里有个在当时具有争议性的决断：任何连续周期信号都可以由一组适当的正弦曲线组合而成。

当时审查这个论文拉格朗日坚决反对此论文的发表，而后在近50年的时间里，拉格朗日坚持认为傅立叶的方法无法表示带有棱角的信号，如在方波中出现非连续变化斜率。直到拉格朗日死后15年这个论文才被发表出来。

谁是对的呢？拉格朗日是对的：正弦曲线无法组合成一个带有棱角的信号。但是，我们可以用正弦曲线来非常逼近地表示它，逼近到两种表示方法不存在能量差别，基于此，傅立叶是对的。

为什么我们要用正弦曲线来代替原来的曲线呢？如我们也还可以用方波或三角波来代替呀，分解信号的方法是无穷多的，但分解信号的目的是为了更加简单地处理原来的信号。

用正余弦来表示原信号会更加简单，因为正余弦拥有原信号所不具有的性质：正弦曲线保真度。一个正余弦曲线信号输入后，输出的仍是正余弦曲线，只有幅度和相位可能发生变化，但是频率和波的形状仍是一样的。且只有正余弦曲线才拥有这样的性质，正因如此我们才不用方波或三角波来表示。

二、傅立叶变换分类

根据原信号的不同类型，我们可以把傅立叶变换分为四种类别：

1. 非周期性连续信号 傅立叶变换（Fourier Transform）

2. 周期性连续信号 傅立叶级数 (Fourier Series)
3. 非周期性离散信号 离散 时域 傅立叶变换 (Discrete Time Fourier Transform)
4. 周期性离散信号 离散傅立叶变换 (Discrete Fourier Transform)

下图是四种原信号图例 (从上到下, 依次是FT, FS, DTFT, DFT) :



这四种傅立叶变换都是针对正无穷大和负无穷大的信号, 即信号的的长度是无穷大的, 我们知道这对于计算机处理来说是不可能的, 那么有没有针对长度有限的傅立叶变换呢? 没有。因为正余弦波被定义成从负无穷小到正无穷大, 我们无法把一个长度无限的信号组合成长度有限的信号。

面对这种困难, 方法是: 把长度有限的信号表示成长度无限的信号。如, 可以把信号无限地从左右进行延伸, 延伸的部分用零来表示, 这样, 这个信号就可以被看成是 非周期性 离散信号, 我们可以用到 离散时域傅立叶变换 (*DTFT*) 的方法。也可以把信号用复制的方法进行延伸, 这样信号就变成了 周期性 离散信号, 这时我们就可以用 离散傅立叶变换方法 (*DFT*) 进行变换。本章我们要讲的是离散信号, 对于连续信号我们不作讨论, 因为计算机只能处理离散的数值信号, 我们的最终目的是运用计算机来处理信号的。

但是对于非周期性的信号, 我们需要用无穷多不同频率的正弦曲线来表示, 这对于计算机来说是不可能实现的。所以对于离散信号的变换只有离散傅立叶变换 (**DFT**) 才能被适用, 对于计算机来说只有离散的和有限长度的数据才能被处理, 对于其它的变换类型只有在数学演算中才能用到, 在计算机面前我们只能用DFT方法, 后面我们要理解的也正是DFT方法。

这里要理解的是我们使用周期性的信号目的是为了能够用数学方法来解决, 至于考虑周期性信号是从哪里得到或怎样得到是无意义的。

每种傅立叶变换都分成实数和复数两种方法, 对于实数方法是最好理解的, 但是复数方法就相对复杂许多了, 需要懂得有关复数的理论知识, 不过, 如果理解了实数离散傅立叶变换(real DFT), 再去理解复数傅立叶变换就更容易了, 所以我们先把复数的傅立叶变换放到一边去, 先来

理解实数傅立叶变换，在后面我们会先讲讲关于复数的基本理论，然后在理解了实数傅立叶变换的基础上再来理解复数傅立叶变换。

还有，这里我们所要说的变换(transform)虽然是数学意义上的变换，但跟函数变换是不同的，函数变换是符合一一映射准则的，对于离散数字信号处理（DSP），有许多的变换：傅立叶变换、拉普拉斯变换、Z变换、希尔伯特变换、离散余弦变换等，这些都扩展了函数变换的定义，允许输入和输出有多种的值，简单地说变换就是把一堆的数据变成另一堆的数据的方法。

三、一个关于实数离散傅立叶变换(Real DFT)的例子

先来看一个变换实例，下图是一个原始信号图像：



这个信号的长度是16，于是可以把这个信号分解9个余弦波和9个正弦波（一个长度为N的信号可以分解成 $N/2+1$ 个正余弦信号，这是为什么呢？结合下面的18个正余弦图,我想从计算机处理精度上就不难理解，一个长度为N的信号，最多只能有 $N/2+1$ 个不同频率，再多的频率就超过了计算机所能所处理的精度范围），如下图：

9个余弦信号：



9个正弦信号：



把以上所有信号相加即可得到原始信号，至于是怎么分别变换出9种不同频率信号的，我们先不急，先看看对于以上的变换结果，在程序中又是该怎么表示的，我们可以看看下面这个示例图：



上图中左边表示时域中的信号，右边是频域信号表示方法，

从左向右， $-->$ ，表示正向转换 (Forward DFT)，从右向左， $<--$ ，表示

逆向转换 (Inverse DFT), 用小写 $x[]$ 表示信号在每个时间点上的幅度值数组, 用大写 $X[]$ 表示每种频率的副度值数组 (即时间 $x \rightarrow$ 频率 X),

因为有 $N/2+1$ 种频率, 所以该数组长度为 $N/2+1$,

$X[]$ 数组又分两种, 一种是表示余弦波的不同频率幅度值: $\text{Re } X[]$,

另一种是表示正弦波的不同频率幅度值: $\text{Im } X[]$,

Re 是实数(Real)的意思, Im 是虚数(Imagine)的意思, 采用复数的表示方法把正余弦波组合起来进行表示, 但这里我们不考虑复数的其它作用, 只记住是一种组合方法而已, 目的是为了便于表达 (在后面我们会知道, 复数形式的傅立叶变换长度是 N , 而不是 $N/2+1$)。如此, 再回过头去, 看上面的正余弦各9种频率的变化, 相信, 问题不大了。

第二章、实数形式离散傅立叶变换 (**Real DFT**)

上一章, 我们看到了一个实数形式离散傅立叶变换的例子, 通过这个例子能够让我们先对傅立叶变换有一个较为形象的感性认识, 现在就让我们来看看实数形式离散傅立叶变换的正向和逆向是怎么进行变换的。在此, 我们先来看一下频率的多种表示方法。

一、频域中关于频率的四种表示方法

1. 序号表示方法, 根据时域中信号的样本数取 $0 \sim N/2$, 用这种方法在程序中使用起来可以更直接地取得每种频率的幅度值, 因为频率值跟数组的序号是一一对应的: $X[k]$, 取值范围是 $0 \sim N/2$;
2. 分数表示方法, 根据时域中信号的样本数的比例值取 $0 \sim 0.5$: $X[f]$, $f = k/N$, 取值范围是 $0 \sim 1/2$;
3. 用弧度值来表示, 把 f 乘以一个 2π 得到一个弧度值, 这种表示方法叫做自然频率(natural frequency): $X[\omega]$, $\omega = 2\pi f = 2\pi k/N$, 取值范围是 $0 \sim \pi$;
4. 以赫兹(Hz)为单位来表示, 这个一般是应用于一些特殊应用, 如取样率为10 kHz表示每秒有10,000个样本数: 取值范围是0到取样率的一半。

二、**DFT**基本函数

$$ck[i] = \cos(2\pi ki/N) \quad sk[i] = \sin(2\pi ki/N)$$

其中k表示每个正余弦波的频率，如为2表示在0到N长度中存在两个完整的周期，10即有10个周期，如下图：



上图中至于每个波的振幅(amplitude)值($\text{Re } X[k]$, $\text{Im } X[k]$)是怎么算出来的,这个是DFT的核心，也是最难理解的部分，我们先来看看如何把分解出来的正余弦波合成原始信号(Inverse DFT)。

三、合成运算方法(Real Inverse DFT)


DFT合成等式（合成原始 时间 信号，频率-->时间，逆向变换）：



如果有学过傅立叶级数，对这个等式就会有似曾相识的感觉，不错！这个等式跟傅立叶级数是非常相似的：



当然，差别是肯定存在的，因为这两个等式是在两个不同条件下运用的，至于怎么证明DFT合成公式，这个我想需要非常强的高等数学理论知识了，这是研究数学的人的工作，对于普通应用者就不需要如此的追根究底了，但是傅立叶级数是好理解的，我们起码可以从傅立叶级数公式中看出DFT合成公式的合理性。

DFT合成等式中的  跟之前提到的 $\text{Im } X[k]$ 和 $\text{Re } X[k]$ 是不一样的，下面是转换方法（关于此公式的解释，见下文）：





但k等于0和N/2时,实数部分的计算要用下面的等式:




上面四个式中的N是时域中点的总数，k是从0到N/2的序号。

为什么要这样进行转换呢？这个可以从频谱密度(spectral density)得到理解，如下图就是个频谱图：




这是一个频谱图，横坐标表示频率大小，纵坐标表示振幅大小，原始信号长度为N（这里是32），经DFT转换后得到的 17 个频率的频谱，频谱密度表示每单位带宽中为多大的振幅，那么带宽是怎么计算出来的呢？看上图，除了头尾两个，其余点的所占的宽度是 $2/N$ ，这个宽度便是每个点的带宽，头尾两个点的带宽是 $1/N$ ，而 $\text{Im } X[k]$ 和 $\text{Re } X[k]$ 表示的是频谱密度，即每一个单位带宽的振幅大小，但  表示 $2/N$ （或 $1/N$ ）带宽的振幅大小，所以  分别应当是 $\text{Im } X[k]$ 和 $\text{Re } X[k]$ 的 $2/N$ （或 $1/N$ ）。

频谱密度就象物理中物质密度，原始信号中的每一个点就象是一个混合物，这个混合物是由不同密度的物质组成的，混合物中含有的每种物质的质量是一样的，除了最大和最小两个密度的物质外，这样我们只要把每种物质的密度加起来就可以得到该混合物的密度了，又该混合物的质量是单位质量，所以得到的密度值跟该混合物的质量值是一样的。

至于为什么虚数部分是负数，这是为了跟复数DFT保持一致，这个我们将在后面会知道这是数学计算上的需要（ $\text{Im } X[k]$ 在计算时就已经加上了一个负号（稍后，由下文，便可知），再加上负号，结果便是正的，等于没有变化）。

如果已经得到了DFT结果，这时要进行 逆转换，即合成原始信号，则可按如下步骤进行转换：

1. 先根据上面四个式子计算得出  的值；
2. 再根据DFT合成等式得到原始信号数据。

下面是用BASIC语言来实现的转换源代码：

```
'DFT逆转换方法
'/XX[]数组存储计算结果（时域中的原始信号）
'/REX[]数组存储频域中的实数分量，IMX[]为虚分量
'
DIM XX[511]
```

```

DIM REX[256]

DIM IMX[256]

/

PI = 3.14159265

N% = 512

/

GOSUB XXXX '转到子函数去获取REX[]和IMX[]数据

/

/

/

FOR K% = 0 TO 256

    REX[K%] = REX[K%] / (N%/2)

    IMX[K%] = -IMX[K%] / (N%/2)

NEXT k%

/

REX[0] = REX[0] / N

REX[256] = REX[256] / N

/

' 初始化XX[]数组

FOR I% = 0 TO 511

    XX[I%] = 0

NEXT I%

/

/

/

/

/

/

FOR K% =0 TO 256

    FOR I%=0 TO 511

/

```

```

        XX[I%] = XX[I%] + REX[K%] * COS(2 * PI * K% * I% / N%)
        XX[I%] = XX[I%] + IMX[K%] * SIN(2 * PI * K% * I% / N%)
    /
NEXT I%
NEXT K%
/
END

```

上面代码中420至490换成如下形式也许更好理解，但结果都是一样的：

```

FOR I% =0 TO 511
    FOR K%=0 TO 256
        /
        XX[I%] = XX[I%] + REX[K%] * COS(2 * PI * K% * I% / N%)
        XX[I%] = XX[I%] + IMX[K%] * SIN(2 * PI * K% * I% / N%)
        /
    NEXT I%
NEXT K%

```

四、分解运算方法（DFT）

有三种完全不同的方法进行DFT：一种方法是通过联立方程进行求解，从代数的角度看，要从N个已知值求N个未知值，需要N个联立方程，且N个联立方程必须是线性独立的，但这是这种方法计算量非常的大且极其复杂，所以很少被采用；第二种方法是利用信号的相关性

（*correlation*）进行计算，这个是我们后面将要介绍的方法；第三种方法是快速傅立叶变换（*FFT*），这是一个非常具有创造性和革命性的方法，因为它大大提高了运算速度，使得傅立叶变换能够在计算机中被广泛应用，但这种算法是根据复数形式的傅立叶变换来实现的，它把N个点的信号分解成长度为N的频域，这个跟我们现在所进行的实域DFT变换不一样，而且这种方法也较难理解，这里我们先不去理解，等先理解了复数DFT后，再来看一下FFT。有一点很重要，那就是这三种方法所得的变换结果是一样的，经过实践证明，当频域长度为32时，利用相

相关性方法进行计算效率最好，否则FFT算法效率较高。现在就让我们来看一下相关性算法。

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



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

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



第二个式子中加了个负号，是为了保持复数形式的一致，前面我们知道在计算 $\int_{-\infty}^{\infty} f(t) \cos(\omega t) dt$ 时又加了个负号，所以这只是个形式的问题，并没有实际意义，你也可以把负号去掉，并在计算 $\int_{-\infty}^{\infty} f(t) \sin(\omega t) dt$ 时也不加负号。

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

下面是实域傅立叶变换的BASIC语言代码:



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

从头到尾彻底理解傅里叶变换算法、下

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

前期回顾，在上一篇里，我们讲了傅立叶变换的由来、和实数形式离散傅立叶变换（Real DFT）俩个问题，

本文接上文，着重讲下复数、和复数形式离散傅立叶变换等俩个问题。

第三章、复数

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

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

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

一、复数的提出

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



其中 h 表示高度， g 表示重力加速度(9.8m/s^2)， v 表示初速度， t 表示时间。现在反过来，假如知道了高度，要求计算到这个高度所需要的时间。

间，这时我们又可以通过下式来计算：



（多谢JERRY_PRI提出：

1. 根据公式 $h=-(gt^2/2)+Vt$ （ gt 后面的2表示 t 的平方），我们可以讨论最终情况，也就是说小球运动到最高点时， $v=gt$ ，所以，可以得到 $t=\sqrt{2h/g}$ ，且在您给的公式中，根号下为 $1-(2h)/g$ ，化成分数形式为 $(g-2h)/g$ ， g 和 h 不能直接做加减运算。
2. g 是重力加速度，单位是 m/s^2 ， h 的单位是 m ，他们两个相减的话在物理上没有意义，而且使用您给的那个公式反向回去的话推出的是 $h=-(gt^2/2)+gt$ 啊（ gt 后面的2表示 t 的平方）。
3. 直接推到可以得出 $t=v/g \pm \sqrt{(v^2-2hg)/g^2}$ （ v 和 g 后面的2都表示平方），那么也就是说当 $v^2 < 2hg$ 时会产生复数，但是如果从实际的 v^2 是不可能小于 $2hg$ 的，所以我感觉复数不能从实际出发去推到，只能从抽象的角度说明一下。

)

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

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

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



上图中的ABC三个向量可以表示成如下的式子：

$$A = 2 + 6j$$

$$B = -4 - 1.5j$$

$$C = 3 - 7j$$

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

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

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

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

复数之间也可以进行加减乘除运算：



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

复数也符合代数运算中的交换律、结合律、分配律：

$$A B = B A$$

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

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

二、复数的极坐标表示形式

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



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



我们还可以通过下面的式子进行极坐标到直角坐标的转换：

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

上面这个等式中左边是直角坐标表达式，右边是极坐标表达式。

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

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

这个等式可以从下面的级数变换中得到证明：



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

这样子我们又可以把复数的表达式表示成指数的形式了：

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

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



三、复数是数学分析中的一个工具

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

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

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

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

四、用复数来表示正余弦函数表达式

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



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

对于极坐标形式可以按如下形式进行转换：



上式中， $M \Leftrightarrow M$ ， $\theta \Leftrightarrow \varphi$ 。

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

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

则采用复数形式来相乘结果是不一样的。幸运的是，我们已严格定义了正余弦复数形式的运算操作条件：

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

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



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

第四章、复数形式离散傅立叶变换

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

一、把正余弦函数表示成复数的形式

通过欧拉等式可以把正余弦函数表示成复数的形式：

$$\cos(x) = \frac{1}{2} e^{-jx} + \frac{1}{2} e^{jx}$$

$$\sin(x) = \frac{j}{2} (e^{-jx} - e^{jx})$$

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

可以通过复数的形式来表示。

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

二、把变换前后的变量都看成复数的形式

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

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

三、对复数进行相关性算法（正向傅立叶变换）

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

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

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

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



这个式子很容易可以得到欧拉变换式子：



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

对于上面的等式，我们要清楚如下几个方面（也是区别于实数DFT的地方）：

1. $X[k]$ 、 $x[n]$ 都是复数，但 $x[n]$ 的虚数部分都是由0组成的，实数部分表示原始信号；
2. k 的取值范围是 $0 \sim N-1$ (也可以表达成 $0 \sim 2\pi$)，其中 $0 \sim N/2$ （或 $0 \sim \pi$ ）是正频部分， $N/2 \sim N-1$ （ $\pi \sim 2\pi$ ）是负频部分，由于正余弦函数的对称性，所以我们把 $-\pi \sim 0$ 表示成 $\pi \sim 2\pi$ ，这是出于计算上方便的考虑。
3. 其中的 j 是一个不可分离的组成部分，就象一个等式中的变量一样，不能随便去掉，去掉之后意义就完全不一样了，但我们知道在实数DFT中， j 只是个符号而已，把 j 去掉，整个等式的意义不变；
4. 下图是个连续信号的频谱，但离散频谱也是与此类似的，所以不影响我们对问题的分析：



上面的频谱图把负频率放到了左边，是为了迎合我们的思维习惯，但在实际实

现中我们一般是把它移到正的频谱后面的。

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

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

谱)。

四、逆向傅立叶变换

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

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

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

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

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

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

这样我们就可以对 $x[n]$ 再次进行变换，如：

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

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

$$= (\text{Re } X[k] (\cos(2\pi kn/N) + j \sin(2\pi kn/N)) + \text{-----}(1)$$

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

这时我们就把原来的等式分成了两个部分，第一个部分是跟实域中的频谱相乘，第二个部分是跟虚域中的频谱相乘，根据频谱图我们可以知

道， $\text{Re } X[k]$ 是个偶对称的变量， $\text{Im } X[k]$ 是个奇对称的变量，即

$$\text{Re } X[k] = \text{Re } X[-k]$$

$$\text{Im } X[k] = -\text{Im } X[-k]$$

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

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

根据偶对称性和三角函数的性质，把上式化简得到：

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

这个式子最后的结果是：

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

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

对于(2)式，用同样的方法，我们也可以得到这样的结果：

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

注意上式前面多了个负符号，这是由于虚数变换的特殊性造成的，当然我们肯定不能把负符号的正弦函数跟余弦来相加，还好，我们前面是用 $\cos(2\pi kn/N) - j \sin(2\pi kn/N)$ 进行相关性计算，得到的 $\text{Im } X[k]$ 中有一个负的符号，这样最后的结果中正弦函数就没有负的符号了，这就是为什么在进行相关性计算时虚数部分要用到负符号的原因（我觉得这也许是复数形式DFT美中不足的地方，让人有一种拼凑的感觉）。

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

后缀树

1.1、后缀树的定义

后缀树（Suffix tree）是一种数据结构，能快速解决很多关于字符串的问题。后缀树的概念最早由Weiner于1973年提出，既而由McCreight在1976年和Ukkonen在1992年和1995年加以改进完善。

后缀，顾名思义，就是后面尾巴的意思。比如说给定一长度为 n 的字符串 $S=S_1S_2\ldots S_i\ldots S_n$ ，和整数 i ， $1 \leq i \leq n$ ，子串 $S_iS_{i+1}\ldots S_n$ 便都是字符串 S 的后缀。

以字符串 $S=XMADAMYX$ 为例，它的长度为8，所以 $S[1..8]$, $S[2..8]$, ..., $S[8..8]$ 都算 S 的后缀，我们一般还把空字符串也算成后缀。这样，我们一共有如下后缀。对于后缀 $S[i..n]$ ，我们说这项后缀起始于 i 。

$S[1..8]$, $XMADAMYX$ ，也就是字符串本身，起始位置为1

$S[2..8]$, $MADAMYX$ ，起始位置为2

$S[3..8]$, $ADAMYX$ ，起始位置为3

$S[4..8]$, $DAMYX$ ，起始位置为4

$S[5..8]$, $AMYX$ ，起始位置为5

$S[6..8]$, MYX ，起始位置为6

$S[7..8]$, YX ，起始位置为7

$S[8..8]$, X ，起始位置为8

空字符串，记为 $\$$ 。

而后缀树，就是包含一则字符串所有后缀的压缩Trie。把上面的后缀加入Trie后，我们得到下面的结构：



仔细观察上图，我们可以看到不少值得压缩的地方。比如蓝框标注的分支都是独苗，没有必要用单独的节点同边表示。如果我们允许任意一条边里包含多个字母，就可以把这种没有分叉的路径压缩到一条边。而另外每条边已经包含了足够的后缀信息，我们就不用再给节点标注字符串信息，只需要在叶节点上标注上每项后缀的起始位置。

于是我们得到下图：



这样的结构丢失了某些后缀。比如后缀**X**在上图中消失了，因为它正好是字符串**XMADAMYX**的前缀。为了避免这种情况，我们也规定每项后缀不能是其它后缀的前缀。要解决这个问题其实挺简单，在待处理的子串后加一个空字符串就行了。例如我们处理**XMADAMYX**前，先把**XMADAMYX**变为**XMADAMYX\$**，于是就得到suffix tree--后缀树了，如下图所示：



1.2、后缀树的应用

后缀树可以解决最长回文问题，那它和最长回文有什么关系呢？在此之前，我们得先知道两个简单概念：

- 最低共有祖先，**LCA**（Lowest Common Ancestor），也就是任意两节点（多个也行）最长的共有前缀。比如下图中，节点7同节点1的共同祖先是节点5与节点10，但最低共同祖先是5。查找LCA的算法是 $O(1)$ 的复杂度，当然，代价是需要对后缀树做复杂度为 $O(n)$ 的预处理。



- 广义后缀树(Generalized Suffix Tree)。传统的后缀树处理一坨单词的所有后缀。广义后缀树存储任意多个单词的所有后缀。例如下图是单词**XMADAMYX**与**XYMADAMX**的广义后缀树。注意我们需要区分不同单词的后缀，所以叶节点用不同的特殊符号与后缀位置

配对。



有了上面的概念，本文引言中提出的查找最长回文问题就相对简单了。咱们来回顾下引言中提出的回文问题的具体描述：找出给定字符串里的最长回文。例如输入XMADAMYX，则输出MADAM。

思维的突破点在于考察回文的半径，而不是回文本身。所谓半径，就是回文对折后的字串。比如回文MADAM的半径为MAD，半径长度为3，半径的中心是字母D。显然，最长回文必有最长半径，且两条半径相等。

还是以MADAM为例，以D为中心往左，我们得到半径DAM；以D为中心向右，我们得到半径DAM。二者肯定相等。因为MADAM已经是单词XMADAMYX里的最长回文，我们可以肯定从D往左数的字串DAMX与从D往右数的子串DAMYX共享最长前缀DAM。而这，正是解决回文问题的关键。现在我们有后缀树，怎么把从D向左数的字串DAMX变成后缀呢？

到这个地步，答案应该明显：把单词XMADAMYX翻转

（XMADAMYX=>XYMADAMX，DAMX就变成后缀了）就行了。于是我们把寻找回文的问题转换成了寻找两坨后缀的LCA的问题。当然，我们还需要知道到底查询哪些后缀间的LCA。很简单，给定字符串S，如果最长回文的中心在i，那从位置i向右数的后缀刚好是S(i)，而向左数的字符串刚好是翻转S后得到的字符串S'的后缀S'(n-i+1)。这里的n是字符串S的长度。

拿单词XMADAMYX来说，回文中心为D，那么D向右的后缀DAMYX假设是S(i)（当N=8，i从1开始计数，i=4时，便是S(4..8)）；而对于翻转后的单词XYMADAMX而言，回文中心D向右对应的后缀为DAMX，也就是S'(N-i+1)（N=8，i=4，便是S'（5..8））。此刻已经可以得出，它们共享最长前缀，即LCA(DAMYX, DAMX)=DAM。有了这套直观解释，算法自然呼之欲出：

1. 预处理后缀树，使得查询LCA的复杂度为O(1)。这步的开销是O(N)，N是单词S的长度；

2. 对单词的每一位置 i (也就是从0到 $N-1$), 获取 $LCA(S(i), S'(N-i+1))$ 以及 $LCA(S(i+1), S'(n-i+1))$ 。查找两次的原因是我们需要考虑奇数回文和偶数回文的情况。这步要考察每坨 i , 所以复杂度是 $O(N)$;
3. 找到最大的LCA, 我们也就得到了回文的中心 i 以及回文的半径长度, 自然也就得到了最长回文。总的复杂度 $O(n)$ 。

i 为4时, $LCA(4\$, 5\#)$ 为DAM, 正好是最长半径。此外, 创建后缀树为 $O(n)$ 的时间复杂度。

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：

- 将文档分割成几个大小相等的部分；
- 对词项ID-文档ID进行排序；
- 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；
- 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 INDEXCONSTRUCTION()
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。

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 $O(N \cdot \log N)$ ，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：

- 从头开始扫描每一个词项-文档ID（信息）对，遇一词，构建索引；
- 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
- 在内存内基于分块完成排序，后合并分块；
- 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
      /*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*/
      then postings_list = AddToDictionary(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*/
      else postings_list = GetPostingList(dictionary,term(token))
      if full(postings_list)
        then postings_list =DoublePostingList(dictionary,term(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*/
      AddToPostingsList (postings_list,docID(token))
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：

由于不需要排序操作，因此处理的速度更快，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，算是抵销了）。

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，

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。按照倒排索引的生成算法，生成 k 组倒排索引。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。而且针对大数量的 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。□

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：

- 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；
- 分词，找出所有单词；
- 过滤不相干的信息（如广告等信息）；
- 构建倒排索引，关键词=>（文章ID 出现次数 出现的位置）生成词典文件 频率文件 位置文件；
- 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 ) % 0x2AAAAAB ;

            temp1 = (seed & 0xFFFF )<< 0x10 ;

            seed = (seed * 125 + 3 ) % 0x2AAAAAB ;

            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;
}

```

```
}
```

//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 (HashATable[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）：

```
//处理空白字符和空白行
```

```
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strcmp字符串比较函数
```

```
int strcmp ( 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-
>classOne,

                infolist->classTwo, infolist->classThree, infolist-
>time, infolist->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

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： ☐

也就是说，上面建索引的过程本该是如下的： ☐

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 *pbuf, *move;

    int  filenum = GetFileName(filename);

    FILE *frp;

    pbuf = ( char *) malloc (BUF_MAX_LEN);

    memset (pbuf, 0 , BUF_MAX_LEN);


    FILE *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);
    rownum++;
}
}

```

```
    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： ☐

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： ☐

第五节、算法的二次改进

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);

```

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）：☐

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。



分析与解法

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](#) 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，查询效率往往比哈希表高。

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 $O(n^2)$ 。

（字符串比较的复杂度是否以strcmp次数计算更妥当，当此处为了讨论简化了细节）。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，我只要顺着他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：



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](#) 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（这些查询串的重度度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个。一个查询串的重度度越高，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，可先将Ip地址使用Ip2long函数（各种语言或库皆提供本函数）转为长整型IpId后，再按IpId%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。（小提示：常用的Ip hash还有一种基于bit的方法，聪明的

你一定想到了吧^_^)

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Query255Byte，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 $3M \times 1K / 4 = 0.75G$ （实际占用会略大一点，因具体实现而异）。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。

所以对数据量较少（可以全部放入内存的时候）我们放弃分而治之+hash映射的步骤，直接上hash统计，然后排序。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字符串，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，我们可以在对数量级的时间内查找和调整。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对应的计数，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：



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。”

当然，在实际的工程中，排序的依据还有很多，例如对于突发的热点问题，虽然之前用户没有输入过，但是却能排在最前面，是因为在系统内部有一些相应的模块进行了所谓的“调权”。此外，上面我们谈到trie的时候，都采取了简化的模型，即trie通常只是针对英语单词而言的，如果是中文，会有一个严重的问题：），你想到了吗？（小提示：英语和中文的基本字符，差别是不是很大？）当然，如果考虑到我们喜欢用拼音来表述中文，这个问题不是没有解决方法的^_^。

此外，在工程实现的时候，前端往往采用了ajax技术来保障速度上的优势，不然等用户输入完毕了咱的推荐词还在慢悠悠的传输中，那也就没有多少意义了。由于前端讨论和咱们这里的讨论关系不大，感兴趣的朋友可以参考相关资料。

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：[点击此处](#)。此外，StackOverflow上也有两个讨论帖子，大家可以看看：[帖子1](#)，[帖子2](#)。

最小操作数

题目描述

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"。

分析与解法

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](#)，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 $O(\text{length})$ ，所以这个建图的总复杂度是 $O(n^2 * \text{length})$ 。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：我们把字典里地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地每一个单词 $O(n)$ ，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，原因是我们经常忽略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 (intx,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 )
        { //已经达到终点了
            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：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;

        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,

            }

            else
            {

                search_next_reach(reach_end, reach_start, meet, kids_from_end, d

            }

        }


        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, set<string>& meet, map<string, vector<string>>& path, set

{
    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]);
        }
    }
}

};
```

最短摘要的生成

题目描述

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？

分析与解法

这个问题比较完整正规的说明是：

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）。

简单分析如下：

@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。

更进一步，我们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],..., 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],..., 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大经典原创吸了集锦与总结 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？

解法一

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。

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”（用户输入的关键词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”程“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）。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”序“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： $O(N^2 * M)$ 。

解法二

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。

解法三

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。搜索从上一轮的搜索结束处开始，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？

最长公共子序列

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。

分析与解法

解法一

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（ $2^m * 2^n$ ）。

解法二

事实上，最长公共子序列问题也有最优子结构性质。

记：

$X_i = \langle x_1, \dots, x_i \rangle$ 即X序列的前i个字符 ($1 \leq i \leq m$)（前缀）

$Y_j = \langle y_1, \dots, y_j \rangle$ 即Y序列的前j个字符 ($1 \leq j \leq n$)（前缀）

假定 $Z = \langle z_1, \dots, z_k \rangle \in \text{LCS}(X, Y)$ 。

- 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 $Z_{k-1} \in \text{LCS}(X_{m-1}, Y_{n-1})$ 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（ $\text{LCS}(X, Y)$ 的长度等于 $\text{LCS}(X_{m-1}, Y_{n-1})$ 的长度加1）。
- 若 $x_m \neq y_n$ ，则亦不难用反证法证明：要么 $Z \in \text{LCS}(X_{m-1}, Y)$ ，要么 $Z \in \text{LCS}(X, Y_{n-1})$ 。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 $Z \in \text{LCS}(X_{m-1}, Y)$ ，类似的，若 $z_k \neq y_n$ 则有 $Z \in \text{LCS}(X, Y_{n-1})$ 。此时，问题化归成求 X_{m-1} 与Y的LCS及X与 Y_{n-1} 的LCS。 $\text{LCS}(X, Y)$ 的长度

为： $\max\{\text{LCS}(X_{m-1}, Y)\text{的长度}, \text{LCS}(X, Y_{n-1})\text{的长度}\}$ 。

由于上述当 $x_m \neq y_n$ 的情况中，求 $\text{LCS}(X_{m-1}, Y)$ 的长度与 $\text{LCS}(X, Y_{n-1})$ 的长度，这两个问题不是相互独立的：两者都需要求 $\text{LCS}(X_{m-1}, Y_{n-1})$ 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、 $\text{LCS}(X_{m-1}, Y_{n-1})+1$ ；2、 $\text{LCS}(X_{m-1}, Y)$ ， $\text{LCS}(X, Y_{n-1})$ ；3、 $\max\{\text{LCS}(X_{m-1}, Y), \text{LCS}(X, Y_{n-1})\}$ 。

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 $X_{m-1} = \langle x_1, x_2, \dots, x_{m-1} \rangle$ ， $Y_{n-1} = \langle y_1, y_2, \dots, y_{n-1} \rangle$ ， $Z_{k-1} = \langle z_1, z_2, \dots, z_{k-1} \rangle$ 。

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：



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 $\theta(m*n)$ 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
   $m := \text{length}[X]$ ;
   $n := \text{length}[Y]$ ;
  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
           $c[i,j] := c[i-1,j-1] + 1$ ;
           $b[i,j] := "\backslash"$ ;
```



```

    end
    else if c[i-1,j]≥c[i,j-1] then
    begin
        c[i,j]:=c[i-1,j];
        b[i,j]:="↑";
    end
    else
    begin
        c[i,j]:=c[i,j-1];
        b[i,j]:="←"
    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"↖"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"↑"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"←"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 $O(mn)$ 。

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,length[X],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]="\" then
    begin
      LCS(b,X,i-1,j-1);
      print(x[i]); {打印x[i]}
    end
  else if b[i,j]="|" 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 $X=\langle A, B, C, B, D, A, B \rangle$ 和 $Y=\langle B, D, C, A, B, A \rangle$ 。由算法LCS_LENGTH和LCS计算出的结果如下图所示：



- 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 $\langle B, C, B, A \rangle$ 的长度。对于 $i, j > 0$ ，项 $c[i, j]$ 仅依赖于是否有 $x_i=y_i$ ，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 $x_i=y_i$ 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”。

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 $\min(m, n)$ 。

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：

$$f[1][1] = \text{same}(1,1)$$

$$f[i][j] = \max\{f[i-1][j-1] + \text{same}(i,j), f[i-1][j], f[i][j-1]\}$$

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 $O(n\log n)$ 。

举一反三

1、最长递增子序列LIS (Longest Increasing Subsequence)

给定一个长度为N的数组，找出一个最长的单调自增子序列（不一定连续，但是顺序不能乱）。例如：给定一个长度为6的数组A{5, 6, 7, 1, 2, 8}，则其最长的单调递增子序列为{5, 6, 7, 8}，长度为4。

分析：其实此LIS问题可以转换成最长公共子序列问题，为什么呢？

- 原数组为A {5, 6, 7, 1, 2, 8}
- 排序后：A' {1, 2, 5, 6, 7, 8}

因为，原数组A的子序列顺序保持不变，而且排序后A'本身就是递增的，这样，就保证了两序列的最长公共子序列的递增特性。如此，若求数组A的最长递增子序列，其实就是求数组A与它的排序数组A'的最长公共子序列。

此外，本题也可以使用动态规划来求解，读者可以继续思考。

木块砌墙

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 1×2 的多米诺骨牌填满 $M \times N$ 的矩形有多少种方案， $M \leq 5$, $N < 2^{31}$ ，输出答案 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)：



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：



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 $= 2 \times 2 = 4$

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 $= 1 \times 1 = 2$ ；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，最终都是分解成一系列求解。
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 $0 \sim \maxState$, $\maxState = 2^k - 1$ （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 $0 \sim \maxState$ 。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）：



即：



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leftstate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 $n=4$ ， $k=3$ ，leftstate=0，rightstate=0，但分解过程中leftstate和rightstate皆从0变化到了maxstate，故才让函数的第3和第4个参数采用leftstate和rightstate这两个变量的形式，公式也就理所当然的写成了 $f(n,k,leftstate,rightstate)$ 。

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：

$$f(2,3,2,5) = f(1,3,2,0) * f(1,3,0,5)$$

$$+ f(1,3,2,1) * f(1,3,1,5)$$

$$+ f(1,3,2,2) * f(1,3,2,5)$$

$$+ f(1,3,2,3) * f(1,3,3,5)$$

$$+ f(1,3,2,4) * f(1,3,4,5)$$

$$+ f(1,3,2,5) * f(1,3,5,5)$$

$$+ f(1,3,2,6) * f(1,3,6,5)$$

$$+ f(1,3,2,7) * f(1,3,7,5)$$

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，在分开处不断的变化，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) //
合并状态。因为只有一列，所以lState和rState相同。
            * Heap(n - 1 , state | rState, state | rState);
        }
        else
        {
            result += Heap(n - 1 , lState, state) * Heap(n - 1
, state, rState);
        }

        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
中

    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系列的情况进行处理，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Heap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，如果没有从缓存中读取结果再进行计算。

分解到一列时，不再分解，直接计算结果

```

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 $state=2$ 需要排除。同时在还需要合并状态，如 $state=1$ 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， $state$ 对应 $2^{(i-1)}$ ，加上所有行的贡献就得到 $state$ （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state，还记得上文中的图2-7么？



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state值别为：

- 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](#)。

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 1, 1 2, 2 * 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 2和2 1的两种木块。

假设我们从上到下，从左到右考虑每个1*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

00X00X0X0

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 1的覆盖掉，也可以用1 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 1的覆盖或者用2 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。那么我们要做多少次方阵乘法呢？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为一个整数。某一行列的状态我们可以用位运算来表示。例如，状态 x 第 i 列是否被覆盖，我们只需要判断 $x \& (1 \ll i)$ 是否非0即可，或者判断 $(x \gg i) \& 1$ ，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

# ifdef 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, [木块砌墙](#)
2. 红色标记, [木块砌墙](#)
3. LoveHarvy, [木块砌墙](#)
4. [在线编译测试木块砌墙问题](#)
5. hero上 [木块砌墙一题](#)

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。

分析与解法

此题是去年微软的三面题，类似于朋友@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，画一个圆，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。

即不是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](#)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：



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”是可以扩展到多维空间的）。
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： $(I, \text{tuple-identifier})$ 。

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。 $I=(I_0, I_1, \dots, I_{n-1})$ 。其结构如下图所示：



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 $a \leq I_0 \leq b$ ， $c \leq I_1 \leq d$ 。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多维空间。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： $(I, \text{child-pointer})$ 。

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。

但R树有些什么问题呢？如@宋梟_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，故才了解下GeoHash算法，[此文](#)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）：



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，则地图上同一个分块内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：

```
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 [这篇文章](#)，而geohash的进一步用法在 [这里](#) 可以了解到。

本章完。

随机取出其中之一元素

题目描述

一个文件中含有 n 个元素（ n 未知），要求在只能遍历一遍这 n 个元素的情况下，等概率随机的取出其中之一元素。

分析与解法

假设5个人轮流抽签，只有其中某一个人能中签，那么，这5个人每个人中签的概率是相等的。不信的话，咱们可以具体计算下。

首先，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，所以第二个人中签的概率是 $4/5 \times 1/4 = 1/5$ （ $4/5$ 表示第一个人未中， $1/4$ 表示第二个人在剩下的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 第二个人未中的概率 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 $1/5$ ，也就是说先后抽签顺序不影响每个人中签概率的大小。

回到咱们的问题，在明确了先后抽签顺序不影响不公平的原则之后，下面，给出选取策略：

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，面试官可能会问你这样做是等概率吗？那我们来证明一下。

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 $1/2$ ，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。

同理，当遍历到第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 $1/L * (1 - 1/(L+1)) = 1/(L+1)$ 。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。

下面给出一个此选取策略的伪代码：

```
Element RandomPick(file):  
    Int length = 1;  
    While (length <= file.size)  
        If (rand() % length == 0)  
            Picked = File[length];  
        Length++;  
    Return picked
```

举一反三

一个文件含有 n 个元素, n 未知的情况下, 顺序遍历一遍, 要求等概率随机取 r 个, 其中 $r < n$ 。

Table of Contents

[介紹](#)

[程序员如何准备面试中的算法](#)

[第一部分 数据结构](#)

[第一章 字符串](#)

[1.0 本章导读](#)

[1.1 旋转字符串](#)

[1.2 字符串包含](#)

[1.3 字符串转换成整数](#)

[1.4 回文判断](#)

[1.5 最长回文子串](#)

[1.6 字符串的全排列](#)

[1.10 本章习题](#)

[第二章 数组](#)

[2.0 本章导读](#)

[2.1 寻找最小的 k 个数](#)

[2.2 寻找和为定值的两个数](#)

[2.3 寻找和为定值的多个数](#)

[2.4 最大连续子数组和](#)

[2.5 跳台阶](#)

[2.6 奇偶排序](#)

[2.7 荷兰国旗](#)

[2.8 矩阵相乘](#)

[2.9 完美洗牌](#)

[2.15 本章习题](#)

[第三章 树](#)

[3.0 本章导读](#)

[3.1 红黑树](#)

[3.2 B树](#)

[3.3 最近公共祖先LCA](#)

[3.10 本章习题](#)

[第二部分 算法心得](#)

[第四章 查找匹配](#)

[4.1 有序数组的查找](#)

[4.2 行列递增矩阵的查找](#)

[4.3 出现次数超过一半的数字](#)

[第五章 动态规划](#)

[5.0 本章导读](#)

[5.1 最大连续乘积子串](#)

[5.2 字符串编辑距离](#)

[5.3 格子取数](#)

[5.4 交替字符串](#)

[5.10 本章习题](#)

[第三部分 综合演练](#)

[第六章 海量数据处理](#)

[6.0 本章导读](#)

[6.1 关联式容器](#)

[6.2 分而治之](#)

[6.3 simhash算法](#)

[6.4 外排序](#)

[6.5 MapReduce](#)

[6.6 多层划分](#)

[6.7 Bitmap](#)

[6.8 Bloom filter](#)

[6.9 Trie树](#)

[6.10 数据库](#)

[6.11 倒排索引](#)

[6.15 本章习题](#)

[第七章 机器学习](#)

[7.1 K 近邻算法](#)

[7.2 支持向量机](#)

[附录 更多题型](#)

[附录A 语言基础](#)

[附录B 概率统计](#)

[附录C 智力逻辑](#)

[附录D 系统设计](#)

[附录E 操作系统](#)

[附录F 网络协议](#)

[sift算法](#)

[sift算法的编译与实现](#)

[教你一步一步用c语言实现sift算法、上](#)

[教你一步一步用c语言实现sift算法、下](#)

[其它](#)

[40亿个数中快速查找](#)

[hash表算法](#)

[一致性哈希算法](#)

[倒排索引关键词不重复Hash编码](#)

[傅里叶变换算法、上](#)

[傅里叶变换算法、下](#)

[后缀树](#)

[基于给定的文档生成倒排索引的编码与实践](#)

[搜索关键词智能提示suggestion](#)

[最小操作数](#)

[最短摘要的生成](#)

[最长公共子序列](#)

[木块砌墙原稿](#)

[附近地点搜索](#)

[随机取出其中之一元素](#)