关于

看过结构之法算法之道blog的朋友可能知道，从2010年10月起，July 开始整理一个微软面试100题的系列，他在整理这个系列的过程当中，越来越强烈的感觉到，可以从那100题中精选一些更为典型的题，每一题详细阐述成章，不断优化，于此，便成了程序员编程艺术系列。

原编程艺术系列从2011年4月至今，写了42个编程问题，在创作的过程当中，得到了很多朋友的支持，特别是博客上随时都会有朋友不断留言，或提出改进建议，或show出自己的思路、代码，或指正bug。

为了方便大家更好的改进、优化、增补编程艺术系列，特把博客上的这个**程序员编程艺术系列和博客内其它部分经典文章**同步到此，成立本项目「Csdn 600万博客结构之法算法之道部分经典博文优化版：《编程之法 — 面试和算法心得》」，邀请各位一起修正和优化。

若发现任何问题、错误、bug，或可以优化的每一段代码，欢迎随时pull request或发issue反馈，thanks。

Code Style

本项目暂约定以下代码风格(不断逐条添加中)：

- 关于空格
    - 所有代码使用4个空格缩进
    - 运算符后使用一个空格
    - "," 和for循环语句中的";" 后面跟上一个空格
    - 条件、分支保留字，如 if for while else switch 后留出一个空格
    - "[]", "."和"->" 前后不留空格
- 用空行把大块代码分成逻辑上的“段落
- 关于括号
    - 大括号另起一行
    - 即便只有一行代码也加大括号

- C 指针中的指针符靠近类型名，如写成int* p，而不写成int *p
- 关于标点
    - 中文表述，使用中文全角的标点符号，如：（ ）、。 ，？
    - 数学公式（包括文中混排的公式）和英文代码，使用英文半角的标点符号，如：(),.?...
- 关于注释
    - 注释统一用中文
    - 尽量统一用"//"，一般不用"/.../"
- 关于命名
    - 类名为大写字母开头的单词组合
    - 函数名比较长，由多个单词组成的，每个单词的首字母大写，如int MaxSubArray()；函数名很短，由一个单词组成，首字母小写，比如int swap()
    - 变量名比较长，由多个单词组成的，首个单词的首字母小写，后面紧跟单词的首字母大写，如maxEnd；变量名很短，由一个单词组成，首字母小写，如left
    - 变量尽量使用全名，能够描述所要实现的功能，如 highestTemprature；对于已经公认了的写法才使用缩写，如 tmp mid prev next
    - 变量名能“望文生义”，如v1, v2不如area, height
    - 常量的命名都是大写字母的单词，之间用下划线隔开，比如MY_CONSTANT
    - il < 4384 和 inputLength < MAX_INPUT_LENGTH，后一种写法更好
- 一个函数只专注做一件事
- 时间复杂度小写表示，如O(nlogn)，而不写成O(N*logN)
- 正文中绝大部分采用C实现，少量C++代码，即以C为主，但不去刻意排斥回避C++；
- 关于的地得

    - 形容词（代词）+ 的 + 名词，例如：我的小苹果
    - 副词 + 地 + 动词，例如：慢慢地走
    - 动词 + 得 + 副词，例如：走得很快

- 关于参考文献
    - 格式：主要责任者.书名〔文献类型标识〕.其他责任者.版本.出版地：出版者，出版年.文献数量.丛编项.附注项.文献标准编号。例子：1 刘少奇.论共产党员的修养.修订 2 版.北京：人民出版社，1962.76 页.

- 专业术语
    - 统一一律用“树结点”，而不是“树节点”。
    - 用左子树、右子树表示树的左右子树没问题，但是否用左孩子、右孩子表示树或子树的左右结点？

- ..
- 此外，更多C++ 部分可参考Google C++ Style Guide，中文版见：http://zh-google-styleguide.readthedocs.org/en/latest/contents/ ；

有何问题或补充意见，咱们可以随时到这里讨论：https://github.com/julycoding/The-Art-Of-Programming-By-July/issues/81 。

Ver Note

- 2010年10月11日，在CSDN上正式开博，感谢博客上所有读者的访问、浏览、关注、支持、留言、评论、批评、指正；
- 2011年1月，在学校的时候，第一家出版社联系出书，因“时机未到，尚需积累”的原因婉拒，随后第二家、第三家出版社陆续联系，因总感觉写书的时机还没到，一律婉拒；
- 2011年10月， 当时在图灵教育的杨海玲老师（现在人民邮电信息技术分社）再度联系出书，再度认为“时机未到”；
- 2014年1月18日，想通了一件事：如果什么都不去尝试，那么将年年一事无成，所以元旦一过，便正式确认今2014年之内要把拖了近3年之久的书出版出来；
- 2013年12月-2014年3月，本github的Contributors 转移结构之法算法之道blog的部分经典文章到本github上，感谢这近100位Contributors，包括但不限于：

- Boshen（除我之外，贡献本github的次数最多）
- sallen450
- marchtea（专门为本github书稿弄了一个HTML网页）
- nateriver520（劝我把书稿放在github上，才有了本github）

- 2014年3月，通读全部文章，修正明显错误，并邀请部分朋友review本github上的全部文章，包括cherry、王威扬、邬勇、高增琪、武博文、杨忠宝等；
- 2014年4月
  - 整个4月，精简篇幅，调整目录，Contributors 贡献其它语言代码，并翻译部分文章；
  - 4月25日，跟人民邮电出版社信息技术分社签订合同，书名暂定《程序员编程艺术：面试和算法心得》，有更好的名字再替换。

- 2014年5月，逐章逐节逐行逐字优化文字描述，测试重写优化每一段每一行每一个代码，确定代码基本风格；
- 2014年6月
  - 第一周，压缩篇幅，宁愿量少，但求质精；
  - 第二周，全面 review；
  - 第三周，本github的部分Contributors 把全部文章从github转到word上，这部分contributors包括包括：zhou1989、qiwsir、DogK、x140yu、ericxk、zhanglin0129、idouba.net、gaohua、kelvinkuo等；
  - 第四周，继续在Word 上做出最后彻底的改进，若未发现bug或pull request，本github将暂不再改动；
  - 6月30日，与出版社约定的交稿日期延期，理由：目前版本不是所能做到的最好的版本。

- 2014年7月，邀请部分好友进行第一轮审稿，包括曹鹏、邹伟、林奔、王婷、何欢，其中，曹鹏重写优化了部分代码。此外，葛立娜对书稿中的语言描述做了不少改进；

- 2014年8月
    - 8月上旬，新增KMP一节内容；
    - 8月下旬，重点修改SVM一节内容；
- 2014年9月
    - 9月上旬，和一些朋友一起重绘稿件中的部分图和公式，这部分朋友包括顾运（@陈笙）、mastermay、在山东大学读研二的丰俊丙、厦门大学电子工程系陈友和等等；
    - 9月下旬，再度邀请另一部分好友进行第二轮审稿，包括许利杰、王亮、陈赢、李祥老师、litaoye等，并在微博上公开征集部分读者审稿，包括李元超、刘琪等等；
- 2014年10月
    - 10月8日起，开始一章一章陆续交Word 稿给出版社初审
    - 10月9日，第一章、字符串完成修改；
    - 10月10日，第二章、数组完成修改；
    - 10月22日，第三章、树完成修改；
- 2014年11月
    - 11月5日，第三章、树完成第二版修改，主要修正部分图片、公式、语言描述的错误；
- 2014年12月
    - 12月1日，第四章、查找完成修改。至此，前4 章的修改稿交付出版社。
    - 12月8日，第五章、动态规划完成修改，等出版社反馈中。一个人坚持有点枯燥。
    - 12月31日，第六章仍未修改完。

- 2015年1月
    - 1月12日凌晨，第六章、海量数据处理完成修改，交付出版社。
- 2015年4月
    - 4月27日凌晨，交完第七章初稿，接下来编辑老师反馈，我修改审阅反馈稿。且书名由原来的《程序员编程艺术：面试和算法心得》暂时改为《编程之法：面试和算法心得》。
- 2015年5月
    - 5月2日，开始写书的前言，大致是：为何要写这本书，写的过程是怎样的；这是本什么书，有何特色，内容是什么，为什么这么写；写给谁看，怎么看更好。当然我还会加一些自己觉得比较个性化的内容。
    - 5月5日，审阅完编辑老师的第一章反馈，并合并。
    - 5月6日，审阅完第二章的一半。海玲姐两位老师给出了大量细致、详尽的修改建议，包括文字表述、语言表达、标点符号、字体格式、出版规范，尤其是正斜体、大小写、上下角。
    - 5月15日，和海玲姐审完第一、二章，标点、术语、表述、逻辑、图片、代码等一切细节。书稿进入一审阶段。
- 2015年6月
    - 6月28日，经过反复修改、确认，书稿第一、二、三章基本定稿，还剩4章。
- 2015年7月下旬，出版社重绘全部图片和公式，编辑加工，复审，三审；
- 2015年8月，发稿审批，排版校对；
- 2015年9月，出胶片，印刷，装订成书；
- ..

Contributors

感谢所有贡献的朋友：https://github.com/julycoding/The-Art-Of-Programming-by-July/graphs/contributors，并非常期待你的加入，thanks。

同时，任何人都可以加入编程艺术讨论QQ群：74631723，需要写验证信息。

此外，欢迎所有已经贡献过本github的99位朋友加入程序员编程艺术室QQ群：149638123，验证信息为你贡献本项目时用的github昵称，thanks。

孤军奋战的时代早已远去，我们只有团结起来，才能帮助到更多人。@研究者July，始于二零一三年十二月十四日。

Copyright

July' PDF

- 《支持向量机通俗导论（理解SVM的三层境界）》Latex排版精细版：http://vdisk.weibo.com/s/zrFL6OXKgnlcp ；Latex版本②：https://raw.githubusercontent.com/liuzheng712/Intro2SVM/master/Intro2SVM.pdf 。
- 原《程序员编程艺术第一~三十七章PDF》：http://download.csdn.net/detail/v_july_v/6694053 ，本github上的文章已经对此PDF进行了极大的优化和改进。
- 《微软面试100题系列之PDF》：http://download.csdn.net/detail/v_july_v/4583815
- 《十五个经典算法研究与总结之PDF》：http://download.csdn.net/detail/v_july_v/4478027
- 编程艺术HTML网页版：http://taop.marchtea.com/
- 2014年4月29日《武汉华科大第5次面试&算法讲座PPT》：http://pan.baidu.com/s/1hqh1E9e ；
- 新书初稿的4个PDF
    - B树的PDF：http://yun.baidu.com/s/1jGwup5k ；
    - 海量数据处理的PDF：http://yun.baidu.com/s/1dDreICL ；
    - 支持向量机的PDF：http://yun.baidu.com/s/1ntwof7j ；
    - KMP的PDF：http://yun.baidu.com/s/1eQel3PK ；
- 2014年9月3日西电第8次面试&算法讲座视频：http://v.youku.com/v_show/id_XNzc2MDYzNDg4.html ；PPT：http://pan.baidu.com/s/1pJ9HFqb ；
- 北京10月机器学习班的所有上课PPT：http://yun.baidu.com/share/home?uk=4214456744&view=share#category/type=0 ；
- 截止到2014年12月9日，结构之法算法之道blog所有155篇博文集锦CHM文件下载地

址：http://pan.baidu.com/s/1gdrJndp ；

- 持续更新..

第一部分 数据结构

第一章 字符串

1.0 本章导读

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

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

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

1.1 旋转字符串

题目描述

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

分析与解法

解法一：暴力移位法

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

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

因此，若要把字符串开头的m个字符移动到字符串的尾部，则可以如下操作：

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

下面，我们来分析一下这种方法的时间复杂度和空间复杂度。

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

解法二：三步反转法

对于这个问题，换一个角度思考一下。

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

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

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

如下图所示：

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->cb
a
    ReverseString(s, m, n - 1); //反转[m..n - 1]，例如Y->Y^T，即 def->fed
    ReverseString(s, 0, n - 1); //反转[0..n - 1]，即如整个反转，(X^TY^T)^T=YX，即 cbafed->defa
bc。
}
```

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

举一反三

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

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

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

1.3 字符串转换成整数

题目描述

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

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

分析与解法

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

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

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

思路有了，你可能不假思索，写下如下代码：

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

显然，上述代码忽略了以下细节：

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

上述其它问题比较好处理，但溢出问题比较麻烦，所以咱们来重点看下溢出问题。

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

我们先设置一些变量：

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

而后，你可能会编写如下代码段处理溢出问题：

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

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

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

为什么呢？因为当给定字符串" 10522545459"时，而MAX_INT是2147483647，即MAX_INT(2147483647) < n_10(1052254545_10)，所以当扫描到最后一个字符‘9’的时候，执行上面的这行代码：

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

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

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

- 一个取巧的方式是把转换后返回的值n定义成long long，即long long n；
- 另外一种则是只比较n和MAX_INT / 10的大小，即：

- 若n > MAX_INT / 10，那么说明最后一步转换时，n*10必定大于MAX_INT，所以在得知n > MAX_INT / 10时，当即返回MAX_INT。

- 若n == MAX_INT / 10时，那么比较最后一个数字c跟MAX_INT % 10的大小，即如果n == MAX_INT / 10且c > MAX_INT % 10，则照样返回MAX_INT。

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

对于上面第二种方式，先举两个例子说明下：

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

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

如此我们可以写出正确的处理溢出的代码：

```
c = *str - '0';
if (sign > 0 && (n > MAX_INT / 10 || (n == MAX_INT / 10 && c > MAX_INT % 10)))
{
    n = MAX_INT;
    break;
}
else if (sign < 0 && (n > (unsigned)MIN_INT / 10 || (n == (unsigned)MIN_INT / 10 && c > (un
signed)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;
```

```
    unsigned int n = 0;

    //判断是否输入为空
    if (str == 0)
    {
        return 0;
    }

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

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

    //确定是数字后才执行循环
    while (isdigit(*str))
    {
        //处理溢出
        int c = *str - '0';
        if (sign > 0 && (n > MAX_INT / 10 || (n == MAX_INT / 10 && c > MAX_INT % 10)))
        {
            n = MAX_INT;
            break;
        }
        else if (sign < 0 && (n >(unsigned)MIN_INT / 10 || (n == (unsigned)MIN_INT / 10 && c >
 (unsigned)MIN_INT % 10)))
        {
            n = MIN_INT;
            break;
        }

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

举一反三

1. 实现string到double的转换

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

1.4 回文判断

题目描述

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

那么，我们的第一个问题就是：判断一个字串是否是回文？

分析与解法

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

解法一

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

代码如下：：

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

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

    while (front < back)
    {
        if (*front != *back)
            return false; // 不是回文，立即返回
        ++front;
        --back;
    }
    return true; // 是回文
}
```

这是一个直白且效率不错的实现，时间复杂度：O(n)，空间复杂度：O(1)。

解法二

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

```
bool IsPalindrome2(const char *s, int n)
{
    if (s == NULL || n < 1)
        return false; // 非法输入
    char *first, *second;

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

    while (first >= s)
    if (s[first--] != s[second++])
        return false; // not equal, so it's not apalindrome
    return true; // check over, it's a palindrome
}
```

时间复杂度：O(n)，空间复杂度：O(1)。

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

举一反三

1、判断一条单向链表是不是“回文”

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

2、判断一个栈是不是“回文”

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

1.6 字符串的全排列

题目描述

输入一个字符串，打印出该字符串中字符的所有排列。

例如输入字符串abc，则输出由字符a、b、c 所能排列出来的所有字符串

abc、acb、bac、bca、cab 和 cba。

分析与解法

解法一、递归实现

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

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

代码可如下编写所示：

~~~
~~~

```
void CalcAllPermutation(char* perm, int from, int to)
{
if (to
```

1.10 本章习题

本章字符串和链表的习题

1、第一个只出现一次的字符

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

2、对称子字符串的最大长度

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

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

3、编程判断俩个链表是否相交

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

问题扩展：

- 如果链表可能有环列?
- 如果需要求出俩个链表相交的第一个节点列?

4、逆序输出链表

输入一个链表的头结点，从尾到头反过来输出每个结点的值。

5、在O(1)时间内删除单链表结点

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

6、找出链表的第一个公共结点

两个单向链表，找出它们的第一个公共结点。

7、在字符串中删除特定的字符

输入两个字符串，从第一字符串中删除第二个字符串中所有的字符。

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

8、字符串的匹配

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

9、字符个数的统计

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

10、最小子串

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

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

提示：文章也可以理解为一个大的字符串数组，单词之前只有空格，没有标点符号。

11、字符串的集合

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

提示：并查集。

12、五笔编码

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

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

13、最长重复子串

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

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

14、字符串的压缩

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

15、最大重复出现子串

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

给定一个字符串，求出其最长的重复子串。

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

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

故最终总的时间复杂度是 O(N^2*logN)

16、字符串的删除

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

17、字符串的移动

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

18、字符串的包含

输入：

L:“hello” “july”

S:“hellomehellojuly”

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

19、倒数第n个元素

链表倒数第n个元素。

提示：设置一前一后两个指针，一个指针步长为1，另一个指针步长为n，当一个指针走到链表尾端时，另

一指针指向的元素即为链表倒数第n个元素。

20、回文字符串

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

例如：

habbafgh

输出h,abba,f,g,h。

提示：一般的人会想到用后缀数组来解决这个问题。

21、最长连续字符

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

22、字符串反转

实现字符串反转函数。

22、字符串压缩

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

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

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

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

注意：只需要完成该函数功能算法，中间不需要有任何IO的输入输出

示例

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

23、集合的差集

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

24、最长公共子串

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

25、均分01

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

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

考虑到最差情况，则是把字符串切分(n - 1)次形成n个长度为1的串。

26、合法字符串

用n个不同的字符（编号1 - n），组成一个字符串，有如下2点要求：

- 1、对于编号为i 的字符，如果2 * i > n，则该字符可以作为最后一个字符，但如果该字符不是作为最后一个字符的话，则该字符后面可以接任意字符；
- 2、对于编号为i的字符，如果2 * i = 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

第二章 数组

2.0 本章导读

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

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

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

最后，注意细节处理，不要忽略边界条件，如字符串转换成整数。

2.1 寻找最小的 k 个数

题目描述

输入n个整数，输出其中最小的k个。

分析与解法

解法一

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

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

解法二

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

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

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

解法三

更好的办法是维护容量为k的最大堆，原理跟解法二的方法相似：

- 1、用容量为k的最大堆存储最先遍历到的k个数，同样假设它们即是最小的k个数；
- 2、堆中元素是有序的，令k1

2.2 寻找和为定值的两个数

题目描述

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

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

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

分析与解法

咱们试着一步一步解决这个问题（注意阐述中数列有序无序的区别）：

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

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

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

可以继续优化做到时间O(N)么？

解法一

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

- 1、 2、 4、 7、11、15

用输入数字15减一下各个数，得到对应的序列为：

- 14、13、11、8、4、 0

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

解法二

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

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

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

解法三

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

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

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

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

下面，咱们先来实现此思路（这里假定数组已经是有序的），代码可以如下编写：

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

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

    //俩头夹逼，或称两个指针两端扫描法，很经典的方法，O(N)
    while (begin < end)
    {
        long currSum = data[begin] + data[end];

        if (currSum == sum)
        {
            //题目只要求输出满足条件的任意一对即可
            printf("%d %d\n", data[begin], data[end]);

            //如果需要所有满足条件的数组对，则需要加上下面两条语句：
            //begin++
            //end--
            break;
        }
        else{
            if (currSum < sum)
                begin++;
            else
                end--;
        }
    }
}
```

解法总结

不论原序列是有序还是无序，解决这类题有以下三种办法：

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

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

综上，若是数组有序的情况下，优先考虑两个指针两端扫描法，以达到最佳的时O(N)，空O(1)效应。否

则，如果要排序的话，时间复杂度最快当然是只能达到O(N log N)，空间O(1)则不在话下。

问题扩展

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

举一反三

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

```
10
```

/ \
5 12
/ \
4 7

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

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

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

3、3-sum问题

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

4、4-sum问题

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

2.3 寻找和为定值的多个数

题目描述

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

分析与解法

解法一

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

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

参考代码如下：

~~~
listlist1;
void SumOfkNumber(int sum, int n)
{
// 递归出口
if (n
~~~

2.4 最大连续子数组和

题目描述

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

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

分析与解法

解法一

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

2.5 跳台阶

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  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
```

2.6 奇偶排序

题目描述

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

分析与解法

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

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

而partition过程有以下两种实现：

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

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

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

解法一

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

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

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

思路有了，接下来，写代码实现：

```
//判断是否为奇数
bool IsOddNumber(int data)
{
    return data & 1 == 1;
}

//交换两个元素
void swap(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

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

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

    while (pBegin < pEnd)
    {
        //如果pBegin指针指向的是奇数，正常，向右移
        if (IsOddNumber(*pBegin))
        {
            pBegin++;
        }
        //如果pEnd指针指向的是偶数，正常，向左移
        else if (!IsOddNumber(*pEnd))
        {
            pEnd--;
        }
        else
        {
            //否则都不正常，交换
            swap(*pBegin, *pEnd);
        }
    }
}
```

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

解法二

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

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

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p - 1
3  for j ← p to r - 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i]  A[j]
7  exchange A[i + 1]  A[r]
8  return i + 1
```

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

① j 指向元素2时，i 也指向元素2，2与2互换不变

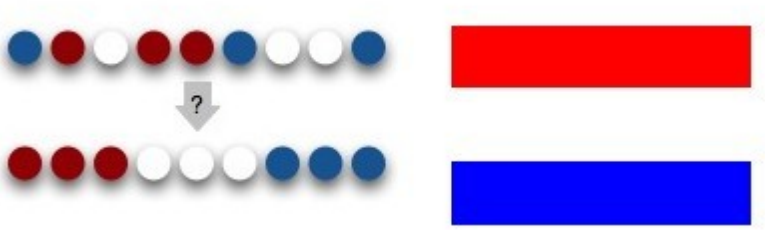
```
i  p/j

   2   8   7   1   3   5   6   4(主元)
```

② 于是j 继续后移，直到指向了1，1

2.7 荷兰国旗

题目描述

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



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：

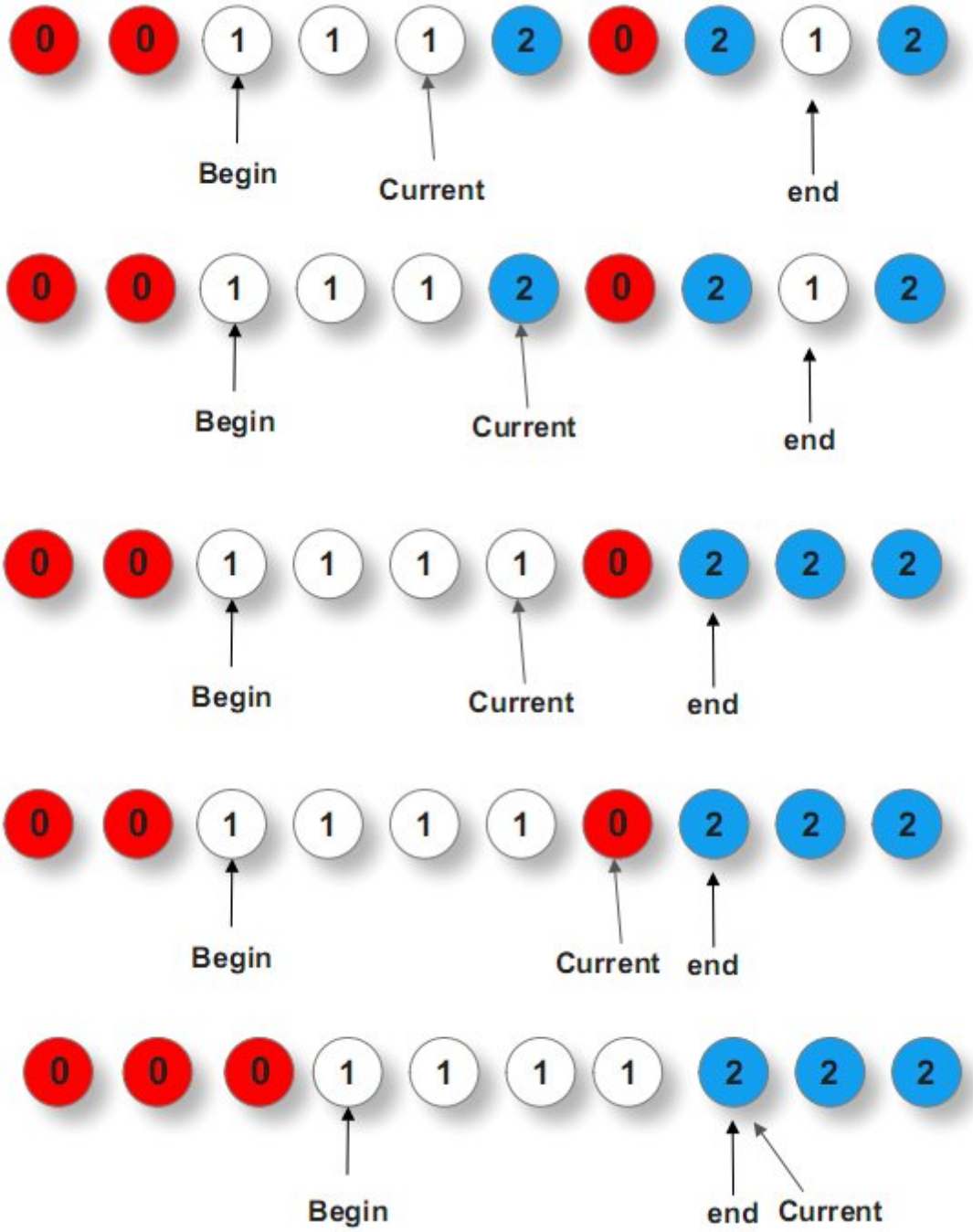

0 1 2 1 1 2 0 2 1 0
Begin Current
end


0 1 2 1 1 2 0 2 1 0
Current Begin
end


0 1 2 1 1 2 0 2 1 0
Begin Current
end


0 1 0 1 1 2 0 2 1 2
Begin Current
end


0 0 1 1 1 2 0 2 1 2
Begin Current
end



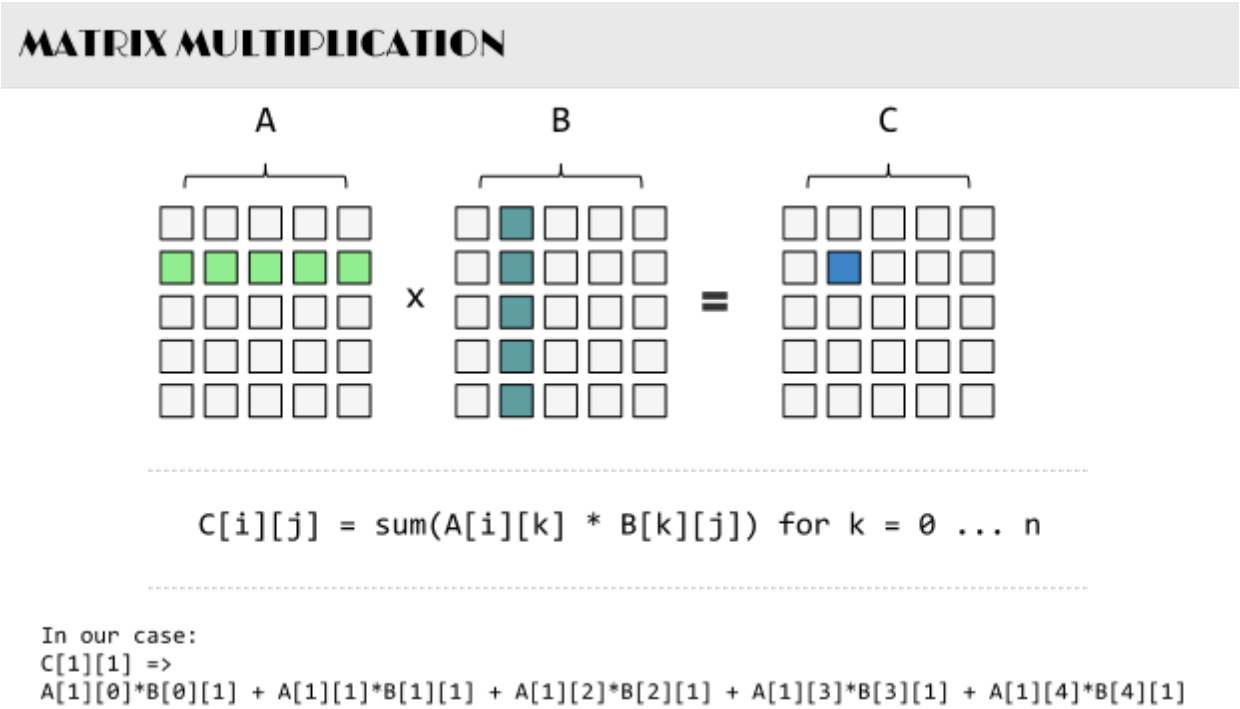
参考代码如下：

~~~
//引用自gnuhpc
while( current

# 2.8 矩阵相乘

## 题目描述
~~~

请编程实现矩阵乘法，并考虑当矩阵规模较大时的优化方法。

分析与解法

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



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：

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}\begin{bmatrix}0 & 1\\0 & 0\end{bmatrix}=\begin{bmatrix}0 & 1\\0 & 3\end{bmatrix},\qquad \begin{bmatrix}0 & 1\\0 & 0\end{bmatrix}\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}=\begin{bmatrix}3 & 4\\0 & 0\end{bmatrix}.$$

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：

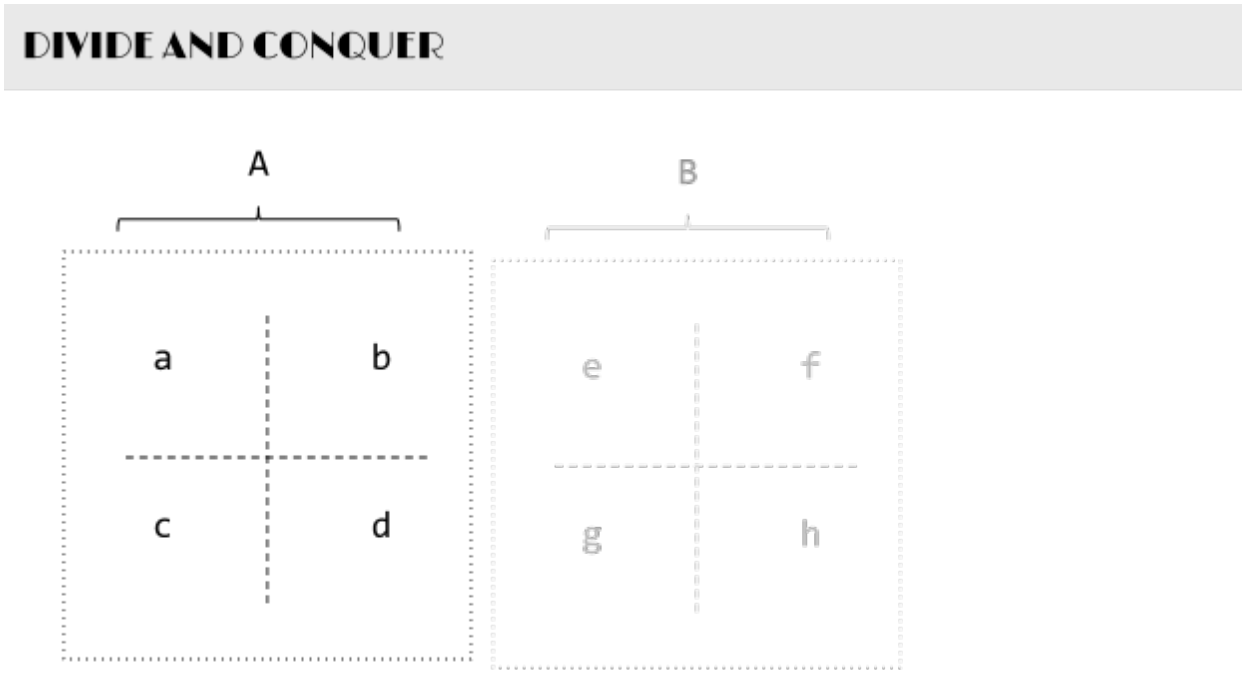
```
//矩阵乘法，3个for循环搞定
void MulMatrix(int** matrixA, int** matrixB, int** matrixC)
{
    for(int i = 0; i < 2; ++i)
    {
        for(int j = 0; j < 2; ++j)
        {
            matrixC[i][j] = 0;
            for(int k = 0; k < 2; ++k)
            {
                matrixC[i][j] += matrixA[i][k] * matrixB[k][j];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

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。

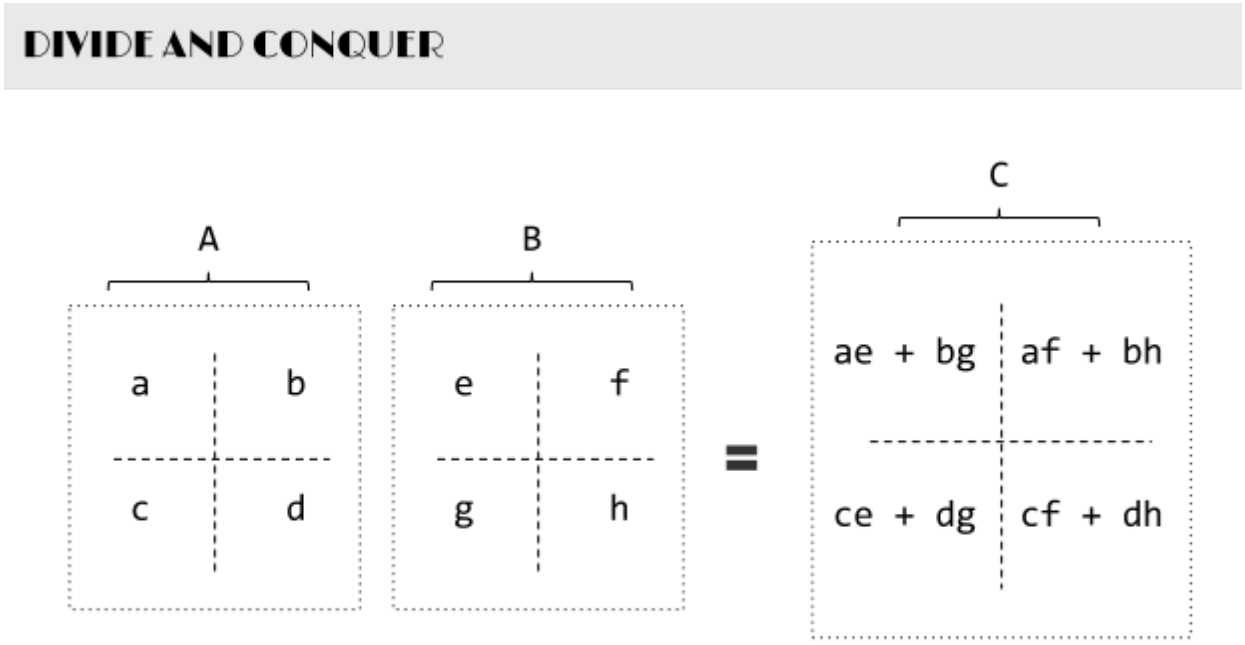
一般说来，当数据量一大时，我们往往会把大的数据分割成小的数据，各个分别处理。遵此思路，如果丢给我们一个很大的两个矩阵呢，是否可以考虑分治的方法循序渐进处理各个小矩阵的相乘，因为我们知道一个矩阵是可以分成更多小的矩阵的。

如下图，当给定一个两个二维矩阵A B时：



We can divide the matrix A in four square parts (in case n is a degree of 2).

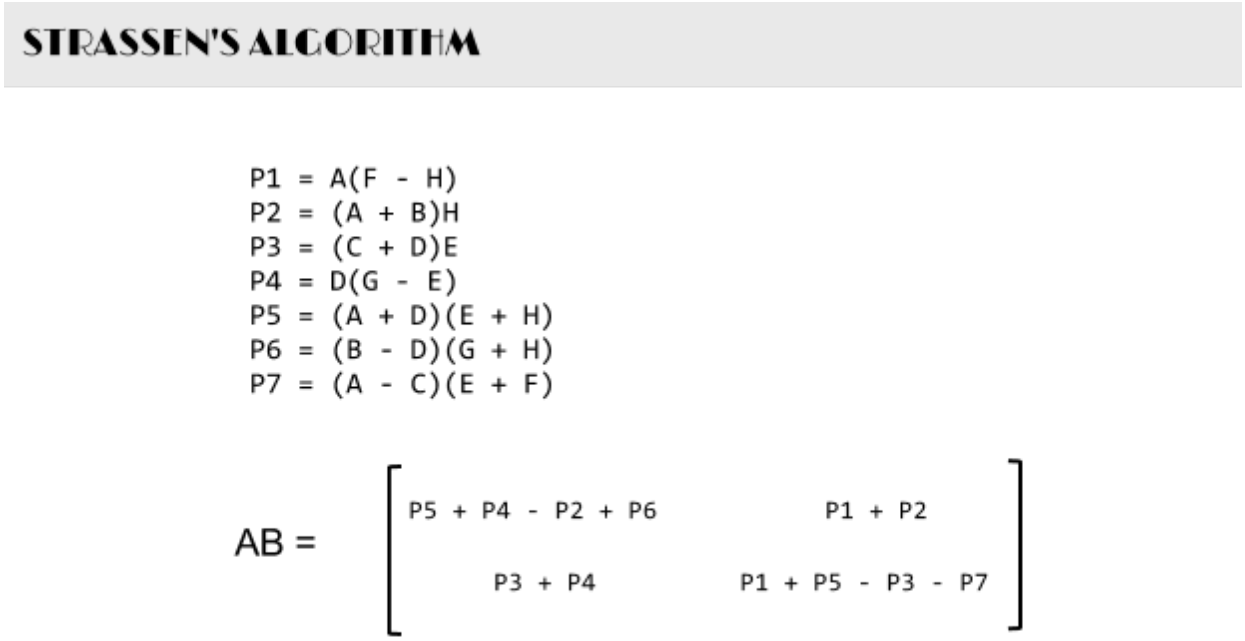
这两个矩阵A B相乘时，我们发现在相乘的过程中，有8次乘法运算，4次加法运算：



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：

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

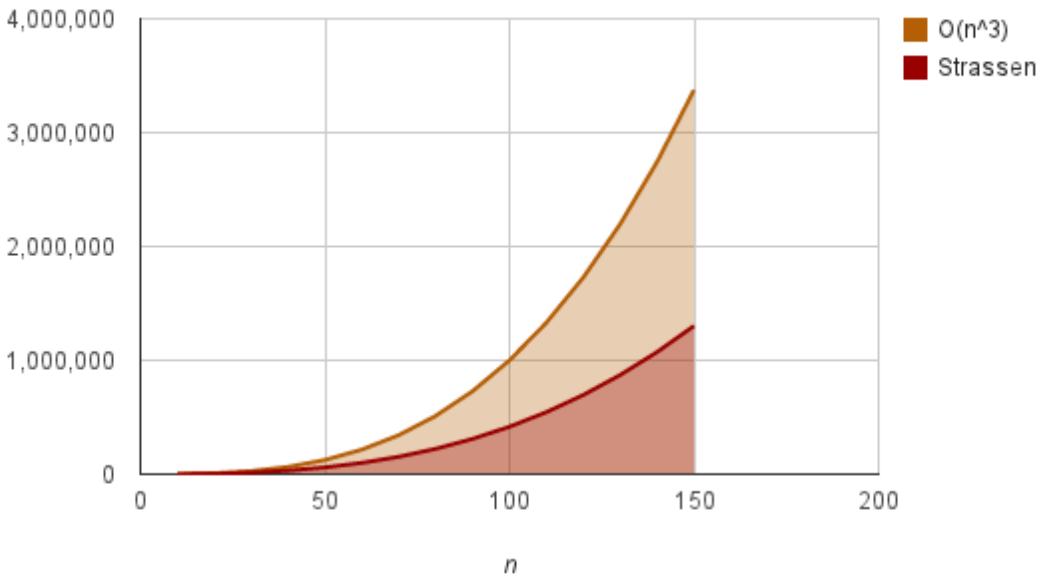
- 可以看出C是这么得来的：

- 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$n^3 = n^{log_2 8}$，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 >> 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* N大小的矩阵乘法的时间复杂度降低到了：$O(n^{2.375477})$，而2010年，Andrew Stothers再度把复杂度降低到了$O(n^{2.3736})$，一年后的2011年，Virginia Williams把复杂度最终定格为：$O(n^{2.3727})$。

2.9 完美洗牌

题目详情

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

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

分析与解法

解法一、蛮力变换

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

a1，a2，a3，a4，**b1，b2，b3，b4**

变成

a1，b1，a2，b2，a3，b3，a4，b4

1.1、步步前移

仔细观察变换前后两个序列的特点，我们可做如下一系列操作：

第①步、确定b1的位置，即让b1跟它前面的a2，a3，a4交换：

a1，b1，a2，a3，a4，**b2，b3，b4**

第②步、接着确定b2的位置，即让b2跟它前面的a3，a4交换：

a1，b1，a2，b2，a3，a4，**b3，b4**

第③步、b3跟它前面的a4交换位置：

a1，b1，a2，b2，a3，b3，a4，b4

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

1.2、中间交换

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

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

a1，a2，a3，b1，a4，**b2，b3，b4**

第②步，让最中间的两对元素各自交换：

a1，a2，b1，a3，b2，a4，**b3，b4**

第③步，交换最中间的三对元素，序列变成：

a1，b1，a2，b2，a3，b3，a4，b4

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

解法二、完美洗牌算法

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

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

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

这个算法解决一个什么问题呢？跟本题有什么联系呢？

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

即：

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

通过完美洗牌问题，得到：

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

再让上面相邻的元素两两swap，即可达到本题的要求：

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

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

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

2.1、位置置换pefect_shuffle1算法

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

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

从上面的例子我们能看到，前n个元素中，

> 第1个元素a1到了原第2个元素a2的位置，即1->2；
>
> 第2个元素a2到了原第4个元素a4的位置，即2->4；
>
> 第3个元素a3到了原第6个元素b2的位置，即3->6；
>
> 第4个元素a4到了原第8个元素b4的位置，即4->8；

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

上面是针对前n个元素，那么针对后n个元素，可以看出：

> 第5个元素b1到了原第1个元素a1的位置，即5->1；
>
> 第6个元素b2到了原第3个元素a3的位置，即6->3；
>
> 第7个元素b3到了原第5个元素b1的位置，即7->5；
>
> 第8个元素b4到了原第7个元素b3的位置，即8->7；

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

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

> 当0 < i < n时， 原式= (2i) % (2 * n + 1) = 2i；
>
> 当i > n时，原式(2 * i) % (2 * n + 1)保持不变。

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

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

#### 
```

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.09.md#22完美洗牌算法perfect_shuffle2)2.2、完美洗牌算法perfect_shuffle2

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.09.md#221走圈算法cycle_leader)2.2.1、走圈算法cycle_leader

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

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

当n=4的情况：

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

即通过置换，我们得到如下结论：

“于此同时，我也提醒下读者，根据上面变换的节奏，我们可以看出有两个圈，

> 一个是1 -> 2 -> 4 -> 8 -> 7 -> 5 -> 1；
>
> 一个是3 -> 6 -> 3。”

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

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

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

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

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

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

上面沿着圈走的算法我们给它取名为cycle_leader，这部分代码如下：

```
//数组下标从1开始，from是圈的头部，mod是要取模的数 mod 应该为 2 * n + 1，时间复杂度O(圈长）
void CycleLeader(int *a, int from, int mod)
{
int t,i;
```

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

}

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.09.md#222神级结论若2n3k---1则可确定圈的个数及各自头部的起始位置)2.2.2、神级结论：若2*n=（3^k - 1），则可确定圈的个数及各自头部的起始位置

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

论文原文部分为：

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

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

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

剩下的n-m部分单独计算；

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

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

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

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

这个翻转的代码如下：

```
//翻转字符串时间复杂度O(to - from)
void reverse(int *a, int from, int to)
{
int t;
for (; from < to; ++from, --to)
{
t = a[from];
a[from] = a[to];
a[to] = t;
}
}

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

翻转后，得到的目标数组的下标为：

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

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

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

既然m=4，即让上述数组中有下划线的两个部分交换，得到：

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

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

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/02.09.md#223完美洗牌算法perfect_shuffle3)2.2.3、完美洗牌算法perfect_shuffle3

从上文的分析过程中也就得出了我们的完美洗牌算法，其算法流程为：

> 输入数组　A[1..2 * n]
>
> step 1 找到 2 * m = 3^k - 1 使得 3^k
> step 2 把a[m + 1..n + m]那部分循环移m位
>
> step 3 对每个i = 0,1,2..k - 1，3^i是个圈的头部，做cycle_leader算法，数组长度为m，所以对2 * m + 1取模。
>
> step 4 对数组的后面部分A[2 * m + 1.. 2 * n]继续使用本算法, 这相当于n减小了m。

上述算法流程对应的论文原文为：

以上各个步骤对应的时间复杂度分析如下：

> 因为循环不断乘3的，所以时间复杂度O(logn)
>
> 循环移位O(n)
>
> 每个圈，每个元素只走了一次，一共2*m个元素，所以复杂度omega(m), 而m < n，所以 也在O(n)内。 T(n - m)
>
> 因此总的时间复杂度为 T(n) = T(n - m) + O(n) ，m = omega(n) ，解得：T(n) = O(n)。

此完美洗牌算法实现的参考代码如下：

```
//copyright@caopengcs 8/24/2013
//时间O(n)，空间O(1)
void PerfectShuffle2(int a, int n)
{
int n2, m, i, k, t;
for (; n > 1;)
{
// step 1
n2 = n 2;
for (k = 0, m = 1; n2 / m >= 3; ++k, m *= 3)
```

;

m /= 2;

// 2m = 3^k - 1 , 3^k

> 定义1 欧拉函数ϕ(m) 表示为不超过m（即小于等于m）的数中，与m互素的正整数个数
>
> 定义2 若ϕ(m)=Ordm(a) 则称a为m的原根，其中Ordm(a)定义为：a ^d （ mod m），其中d=0,1,2,3...，但取让等式成立的最小的那个d。

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

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

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

> 定义3 对模指数，a对模m的原根定义为 $Ord_m(a) = d$,st: $a^d \equiv 1(mod m)$ 中最小的正整数d

再比如，2是9的原根，因为$2^d \equiv 1(mod 9)$，为了让2^d除以9的余数恒等于1，可知最小的正整数d=6，而ϕ(m)=6，满足原根的定义。

> 定理1 同余定理：两个整数a，b，若它们除以正整数m所得的余数相等，则称a，b对于模m同余，记作 $a \equiv b \pmod{m}$，读做a与b关于模m同余。
>
> 定理2 当p为奇素数且a是p^2的原根时⇒ a也是p^k的原根
>
> 定理3 费马小定理：如果a和m互质，那么a^ϕ(m) mod m = 1
>
> 定理4 若(a,m)=1 且a为m的原根，那么a是(Z/mZ)*的生成元。

取a = 2, m = 3。

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

所以：

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

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

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

同理，任意S(k - t)中的数x，都满足gcd(x , 3^k) = 1,于是gcd(3^k , x* 3^t) = 3 ^ t, 并且x_3^t < 3^k。可见S(k - t)中的数x_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(mod 27)可以把（ /27)*取遍，故可得这些数分别在以下3个圈内：

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

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

具体点则是：如果3×2^k除27的余数能够被9整除，则有一个n使得3_2^k=9n(mod 27)，即3_2^k - 9n能够被27整除，从而3*2^k - 9n = 27m，其中n，m为整数，这样一来，式子约掉一个3，我们便能得到2^k = 9m + 3n，也就是说，2^k是3的倍数，这与2^k与27互素是矛盾的，所以，3×2^k除27的余数不可能被9整除。

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

式子两边同时乘以3得到：3_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，=> 3_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 = 37m，=> 3_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$i \in \{1,2,\ldots,2n\}$，故：

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

由于n=13，2n+1 = 27，据此$i \to 2i(mod(2n+1))$公式可知，上面第 i 位置的数将分别变成下述位置的：

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

根据i 和 i ‘ 前后位置的变动，我们将得到3个圈：

```
1->2->4->8->16->5->10->20->13->26->25->23->19->11->22->17->7->14->1；
3->6->12->24->21->15->3
9->18->9
```

没错，这3个圈的数字与咱们之前得到的3个圈一致吻合，验证完毕。

举一反三

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

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

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

2.15 本章习题

本章数组和队列的习题

1、不用除法运算

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

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

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

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

参考代码如下所示：

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

3、找出数组中唯一的重复元素

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

4、找出唯一出现的数

一个数组里，数都是两两出现的，但是有三个数是唯一出现的，找出这三个数。

5、找出反序的个数

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

6、

有两个序列A和B,A=(a1,a2,...,ak),B=(b1,b2,...,bk),A和B都按升序排列，对于1

第三章 树

3.0 本章导读

想要更好地理解红黑树，可以先理解二叉查找树和2-3树。为何呢？首先，二叉查找树中的结点是2-结点（一个键两条链），引入3-结点（两个键三条链），即成2-3树；然后将2-3树中3-结点分解，即成红黑

树，故结合二叉查找树易查找和2-3树易插入的特点，便成了红黑二叉查找树，简称红黑树。

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

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

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

3.2 B树

1.前言：

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

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

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

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

2.外存储器—磁盘

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

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

2.1 磁盘的构造

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

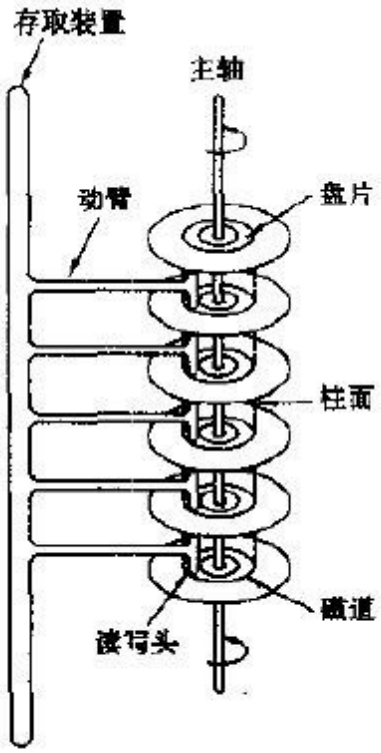


图 11.3 活动头盘示意图

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

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

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

2.2 磁盘的读/写原理和效率

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

读/写磁盘上某一指定数据需要下面3个步骤：

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

经过上面三个步骤，指定数据的存储位置就被找到。这时就可以开始读/写操作了。

访问某一具体信息，由3部分时间组成：

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

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

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

3.B- 树

3.1 什么是B-树

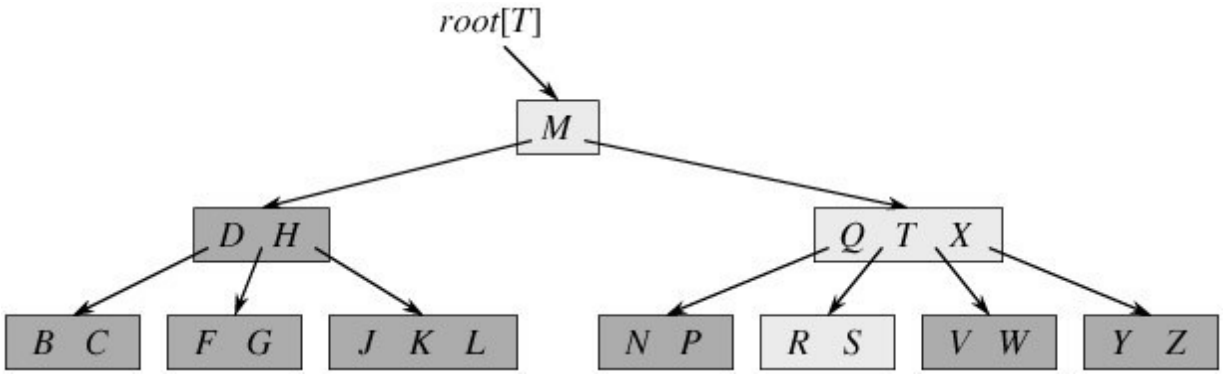
B-树，即为B树。顺便说句，因为B树的原英文名称为B-tree，而国内很多人喜欢把B-tree译作B-树，其实，这是个非常不好的直译，很容易让人产生误解。如人们可能会以为B-树是一种树，而B树又是另外一种树。而事实上是，**B-tree就是指的B树**。

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

B树与红黑树最大的不同在于，B树的结点可以有许多子女，从几个到几千个。不过B树与红黑树一样，一棵含n个结点的B树的高度也为 O(lgn) ，但可能比一棵红黑树的高度小许多，因为它的分支因子比较大。

所以，B树可以在O（logn）时间内，实现各种如插入（insert），删除（delete）等动态集合操作。

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



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

B树的定义

B 树又叫平衡多路查找树。**一棵m阶的B 树** (注：切勿简单的认为一棵m阶的B树是m叉树，虽然存在四叉树，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>=2）；
2. 除根结点和叶子结点外，其它每个结点至少有[ceil(m / 2)]个孩子（其中ceil(x)是一个取上限的函数）；
3. 根结点至少有2个孩子（除非B树只包含一个结点：根结点）；
4. 所有叶子结点都出现在同一层，叶子结点不包含任何关键字信息(可以看做是外部结点或查询失败的结点，指向这些结点的指针都为null)；（注：叶子节点只是没有孩子和指向孩子的指针，这些节点也存在，也有元素。类似红黑树中，每一个NULL指针即当做叶子结点，只是没画出来而已）。
5. 每个非终端结点中包含有n个关键字信息：(n，P0，K1，P1，K2，P2，……，Kn，Pn)。其中：
   a) Ki (i=1...n)为关键字，且关键字按顺序升序排序K(i-1)< Ki。
   b) Pi为指向子树根的结点，且指针P(i-1)指向子树种所有结点的关键字均小于Ki，但都大于K(i-1)。
   c) 关键字的个数n必须满足：[ceil(m / 2)-1]

3.3 最近公共祖先LCA

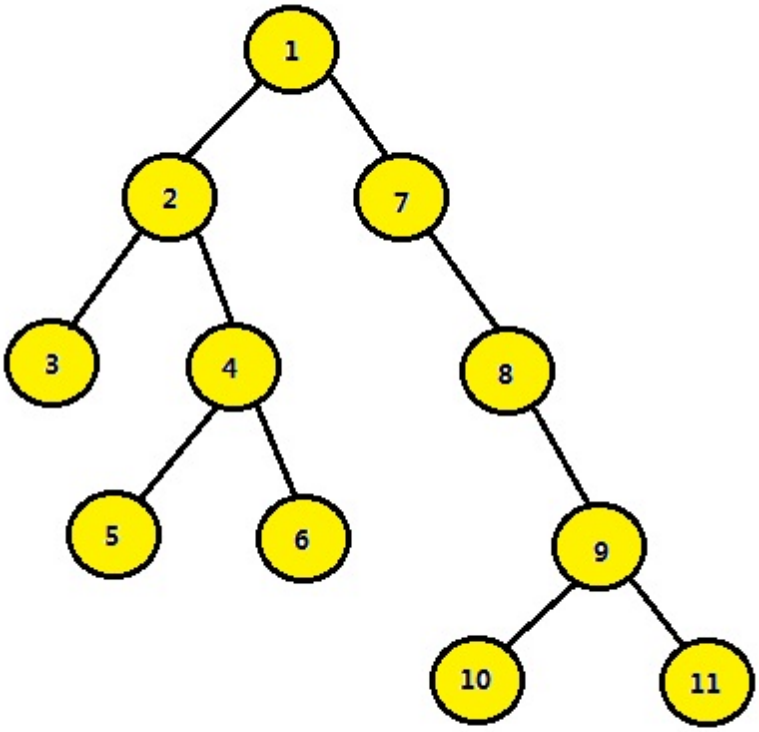
问题描述

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：



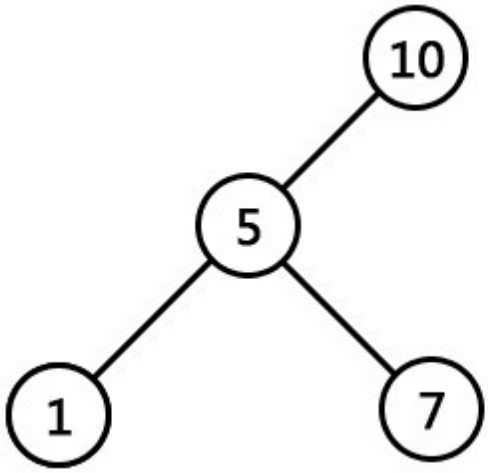
结点3和结点4的最近公共祖先是结点2，即LCA（3 4）=2 。在此，需要注意到当两个结点在同一棵子树上的情况，如结点3和结点2的最近公共祖先为2，即 LCA（3，2）=2。同理：
LCA（5，6）=4，LCA（6，10）=1。

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 right) {
  int temp = left;
  left = right;
  right = temp;
  }

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

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

```
    }
    }
```

[](https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.03.md#12不是二叉查找树)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。

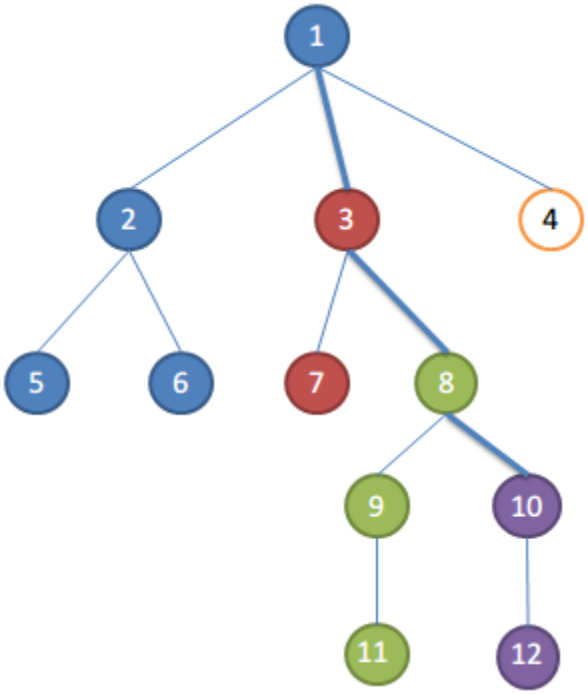
涉及到批量查询的时候，咱们可以借鉴离线处理的方式，这就引出了解决此LCA问题的Tarjan离线算法。

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：

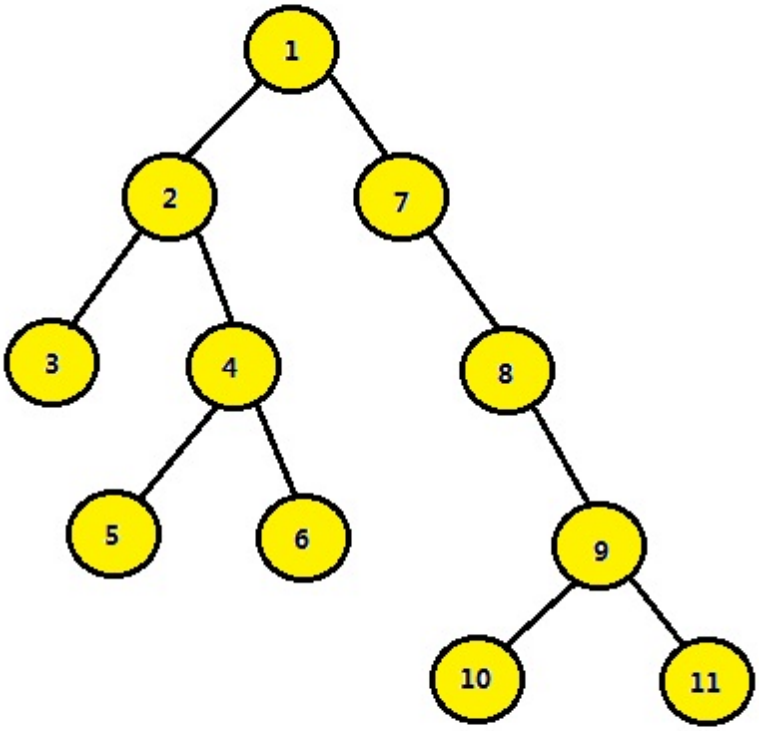
- 1，2，5，6为一个集合,祖先为1，集合中点和10的LCA为1

- 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。

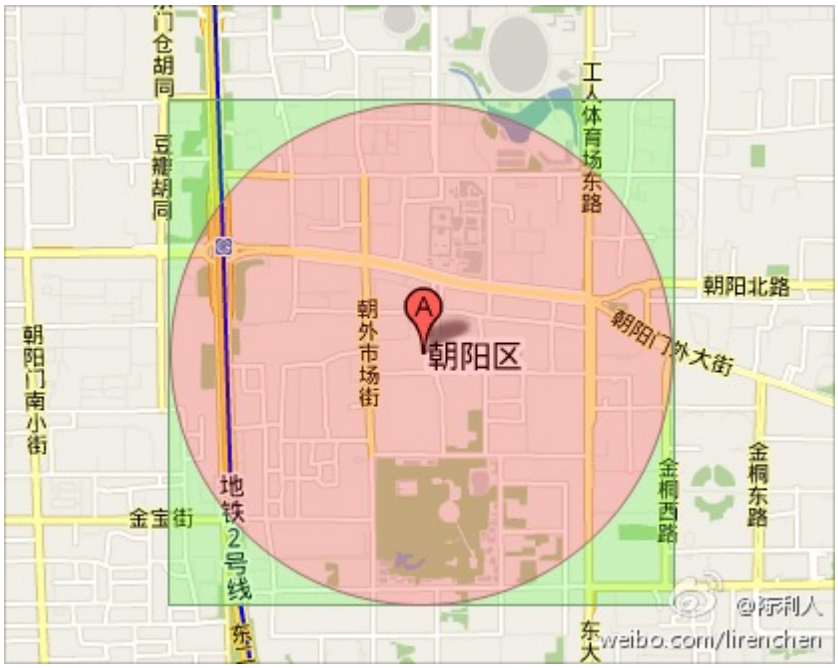
一如上文1.1节我们得到的结论： “如果当前结点t 满足 u

3.10 本章习题

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

请用递归和非递归俩种方法实现二叉树的前序遍历。

10、求树的深度

输入一棵二元树的根结点，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

例如：输入二元树： 10

```
                    /    /

                  6       14

                /       /   /

              4       12     16
```

输出该树的深度3。

实现简单的一个查找二叉树的深度的函数。

11、用俩个栈实现队列

某队列的声明如下：

```
template class CQueue
{
public:
      CQueue() {}
      ~CQueue() {}
      void appendTail(const T& node);  // append a element to tail
      void deleteHead();               // remove a element from head
private:
     T> m_stack1;
     T> m_stack2;
};
```

提示：这道题实质上是要求我们用两个栈来实现一个队列。栈是一种后入先出的数据容器，因此对队列进行的插入和删除操作都是在栈顶上进行；队列是一种先入先出的数据容器，我们总是把新元素插入到队列的尾部，而从队列的头部删除元素。

12

假设有一颗二叉树，已知这棵树的节点上不均匀的分布了若干石头，石头数跟这棵二叉树的节点数相同，石头只可以在子节点和父节点之间进行搬运，每次只能搬运一颗石头。请问如何以最少的步骤将石头搬运均匀，使得每个节点上的石头上刚好为1。

13

对于一颗完全二叉树，要求给所有节点加上一个pNext指针，指向同一层的相邻节点；如果当前节点已经是该层的最后一个节点，则将pNext指针指向NULL；给出程序实现，并分析时间复杂度和空间复杂度。

14

两个用户之间可能互相认识，也可能是单向的认识，用什么数据结构来表示？如果一个用户不认识别人，而且别人也不认识他，那么他就是无效节点，如何找出这些无效节点？自定义数据接口并实现之，要求尽可能节约内存和空间复杂度。

15

有一个一亿节点的树，现在已知两个点，找这两个点的共同的祖先。

16

给一个二叉树，每个节点都是正或负整数，如何找到一个子树，它所有节点的和最大？

提示：后序遍历，每一个节点保存左右子树的和加上自己的值。额外一个空间存放最大值。

写完后序遍历，面试官可能接着与你讨论，

- a). 如果要求找出只含正数的最大子树，程序该如何修改来实现？
- b). 假设我们将子树定义为它和它的部分后代，那该如何解决？
- c). 对于b，加上正数的限制，方案又该如何？

总之，一道看似简单的面试题，可能能变换成各种花样。

比如，面试管可能还会再提两个要求：第一，不能用全局变量；第二，有个参数控制是否要只含正数的子树。

17

有一个排序二叉树，数据类型是int型，如何找出中间大的元素。

18

中序遍历二叉树，结果为ABCDEFGH，后序遍历结果为ABEDCHGF，那么前序遍历结果为?

19

写程序输出8皇后问题的所有排列，要求使用非递归的深度优先遍历。

20

在8X8的棋盘上分布着n个骑士，他们想约在某一个格中聚会。骑士每天可以像国际象棋中的马那样移动一次，可以从中间像8个方向移动（当然不能走出棋盘），请计算n个骑士的最早聚会地点和要走多少天。要求尽早聚会，且n个人走的总步数最少，先到聚会地点的骑士可以不再移动等待其他的骑士。

从键盘输入n（0

第二部分 算法心得

第四章 查找匹配

4.1 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%的程序员写的程序中有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 = middle-1;
    - right = n => while(left 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 > 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

4.2 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。

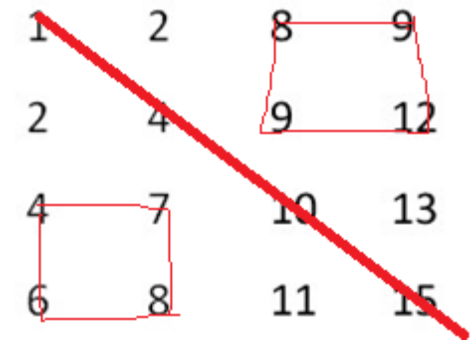
1	2	8	9
2	4	9	12
4	7	10	13
6	8	11	15

分析与解法

解法一、分治法

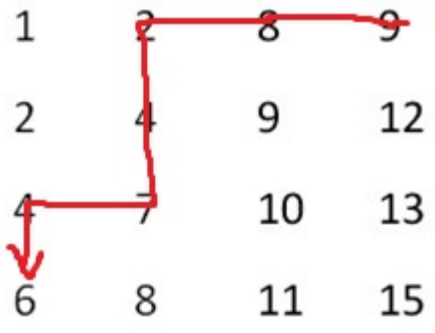
这种行和列分别递增的矩阵，有一个专有名词叫做杨氏矩阵，由剑桥大学数学家杨表在1900年推提出，在这个矩阵中的查找，俗称杨氏矩阵查找。

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×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。

4.3 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(nlogn + 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(nlogn)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*logn)。

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。

但是数组如果是{5, 5, 5, 5, 1}，还能运用上述思路么？很明显不能，咱们得另寻良策。

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

5.0 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）。

5.1 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_0.5_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 CommonSubstring）和最长公共子序列（LongestCommon Subsequence，LCS）是：

- 子串（Substring）是串的一个连续的部分，
- 子序列（Subsequence）则是从不改变序列的顺序，而从序列中去掉任意的元素而获得的新序列；

更简略地说，前者（子串）的字符的位置必须连续，后者（子序列LCS）则不必。比如字符串“acdfg”同“akdfc”的最长公共子串为“df”，而它们的最长公共子序列LCS是“adf”，LCS可以使用动态规划法解决。

解法一

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maxend = 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（N2），显然这不是最好的解法。

5.2 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：

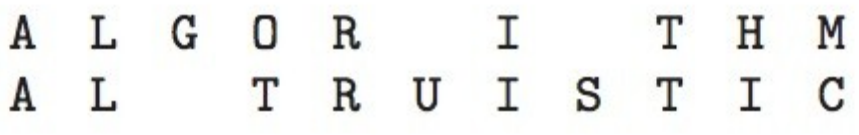
```
dp[i][j] =min{
   *   dp[i-1][j] + 1 , S[i]不在T[0...j]中
   *   dp[i-1][j-1] + 1/0 , S[i]在T[j]
   *   dp[i][j-1] + 1 , S[i]在T[0...j-1]中
}
```

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] ≠ 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)=... =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：



把图中上面的源串S[0...i] = “ALGORITHM”编辑成下面的目标串T[0...j] = “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 + 字符X，T + 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
```

5.3 格子取数

题目描述

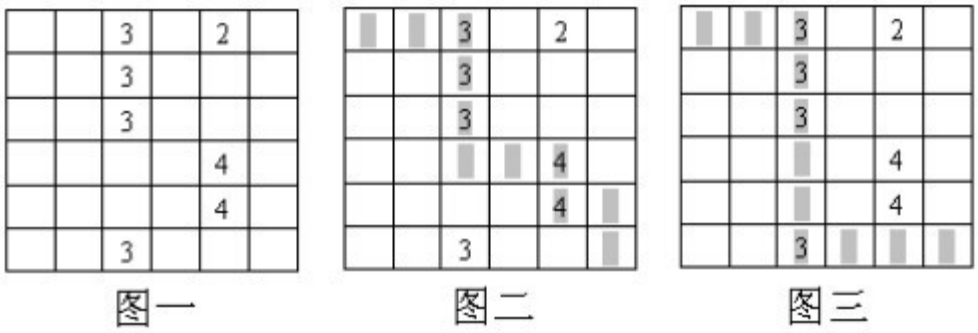
有n*n个格子，每个格子里有正数或者0，从最左上角往最右下角走，只能向下和向右，一共走两次（即从左上角走到右下角走两趟），把所有经过的格子的数加起来，求最大值SUM，且两次如果经过同一个格子，则最后总和SUM中该格子的计数只加一次。

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	13	0	0	6	0	0
3	0	0	0	0	7	0	0	0
4	0	0	0	14	0	0	0	0
5	0	21	0	0	0	4	0	0
6	0	0	15	0	0	0	0	0
7	0	14	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

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。

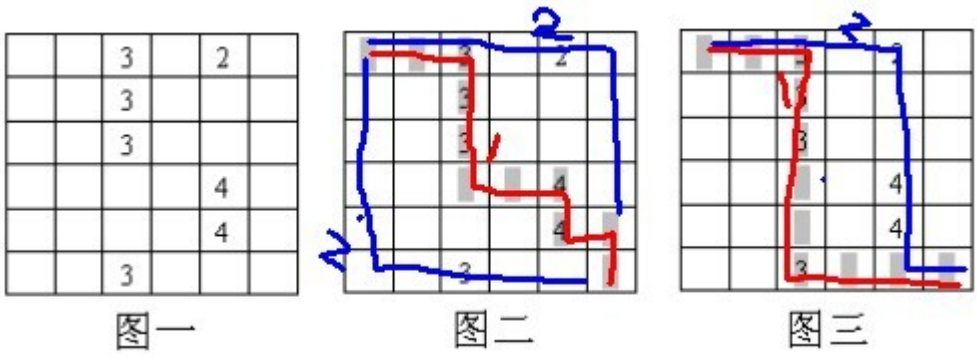
但问题马上就来了，虽然这一算法保证了连续的两次走法都是最优的，但却不能保证总体最优，相应的反例也不难给出，请看下图：



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。

再来分析一下经过其他s步后所处的位置，根据上面的讨论，可以知道：

- 经过8步后，一定处于右下角(8)；
- 那么经过5步后(s = 5)，肯定会处于编号为5的位置；
- 3步后肯定处于编号为3的位置；
- s = 4的时候，处于编号为4的位置，此时对于方格中，共有5（相当于n）个不同的位置，也是所有编号中最多的。

故推广来说，对于n*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 = 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
```

5.4 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 = “aadbbcbcac”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 2, String 3){
    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。

5.10 本章习题

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],.....a[n]，则非空序列a'=a[p1],a[p2].....a[pm]为a的一个子序列其中1

第三部分 综合演练

第六章 海量数据处理

6.0 本章导读

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。

6.1 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。

6.2 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*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 ‘logK。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(logk），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(logk))后，有k1>k2>...kmin（kmin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>kmin，则更新堆（x入堆，用时logk），否则不更新堆。这样下来，总费时O（k_logk+（n-k）_logk）=O（n*logk）。此方法得益于在堆中，查找等各项操作时间复杂度均为logk。

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个元素的最小推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hash(x)%5000，然后把该值存到5000个小文件（记为x0,x1,...x4999）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hash(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$b_0, b_1, \cdots, b_{10}$）。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×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$\text{hash}(url)\%1000$，然后根据所取得的值将url分别存储到1000个小文件（记为$a_0,a_2,\cdots,a_{999}$，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$b_0,b_1,\cdots,b_{999}$）。这样处理后，所有可能相同的url都在对应的小文件（$a_0 \text{ vs } b_0,a_1 \text{ vs } b_1,\ldots,a_0 \text{ vs } b_{999}$）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*100)。

解法二：采用快速排序的思想，每次分割之后只考虑比轴大的一部分，知道比轴大的一部分在比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*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*lg100)。

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表示单词的平准长度）。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*lg10)。所以总的时间复杂度，是O(n*le)与O(*n*lg10)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。

6.3 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的网页，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GoogleMoses 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。

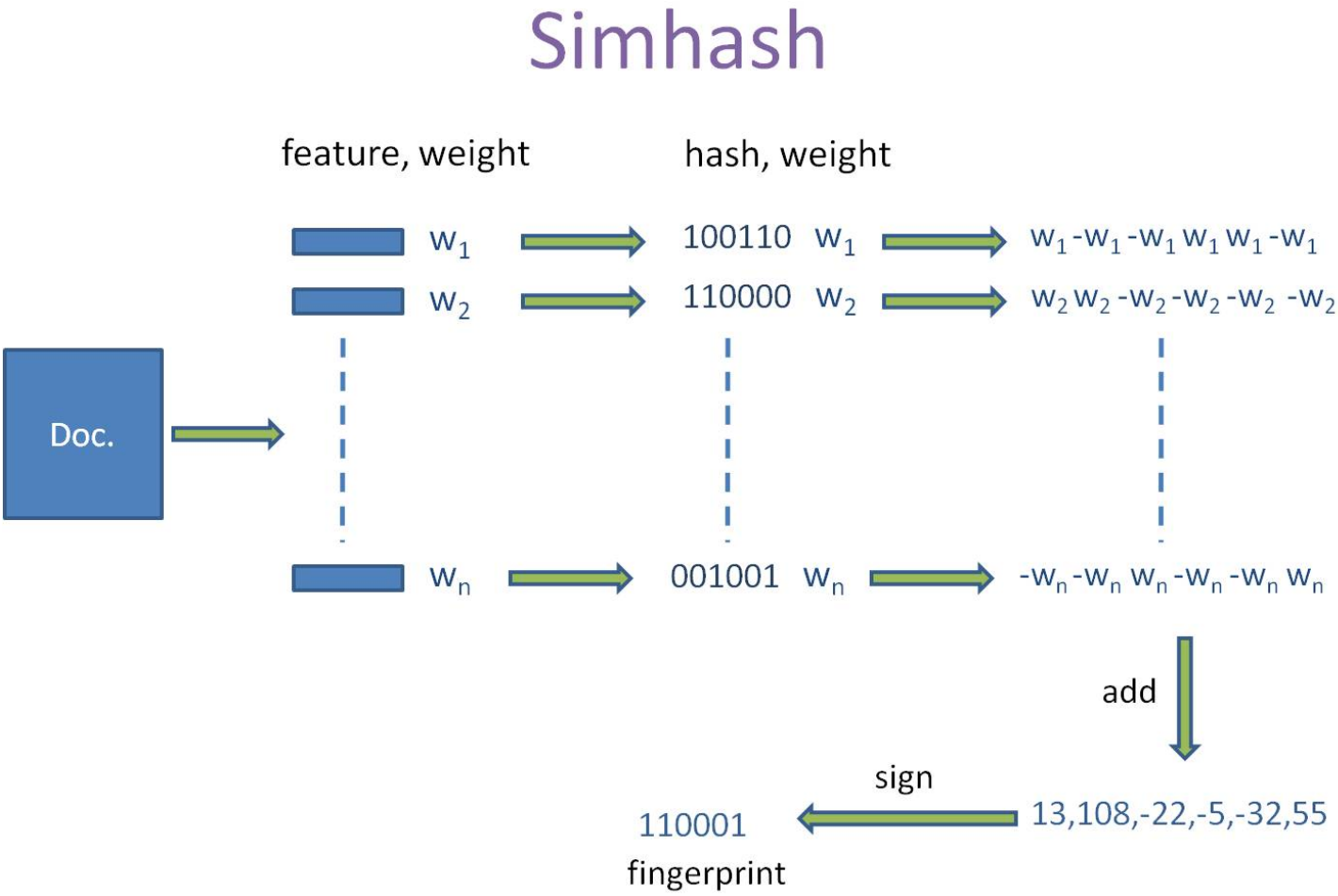
- hash
    - 通过hash函数计算各个特征向量的hash值，hash值为二进制数01组成的n-bit签名。比如“CSDN”的hash值Hash(CSDN)为100101，“博客”的hash值Hash(博客)为“101011”。就这样，字符串就变成了一系列数字。

- 加权
    - 在hash值的基础上，给所有特征向量进行加权，即W = Hash * 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_4 = 4 -4 -4 4 -4 4，给“博客”的hash值“101011”加权得到：W(博客)=101011_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”。

- 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 9”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：



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。

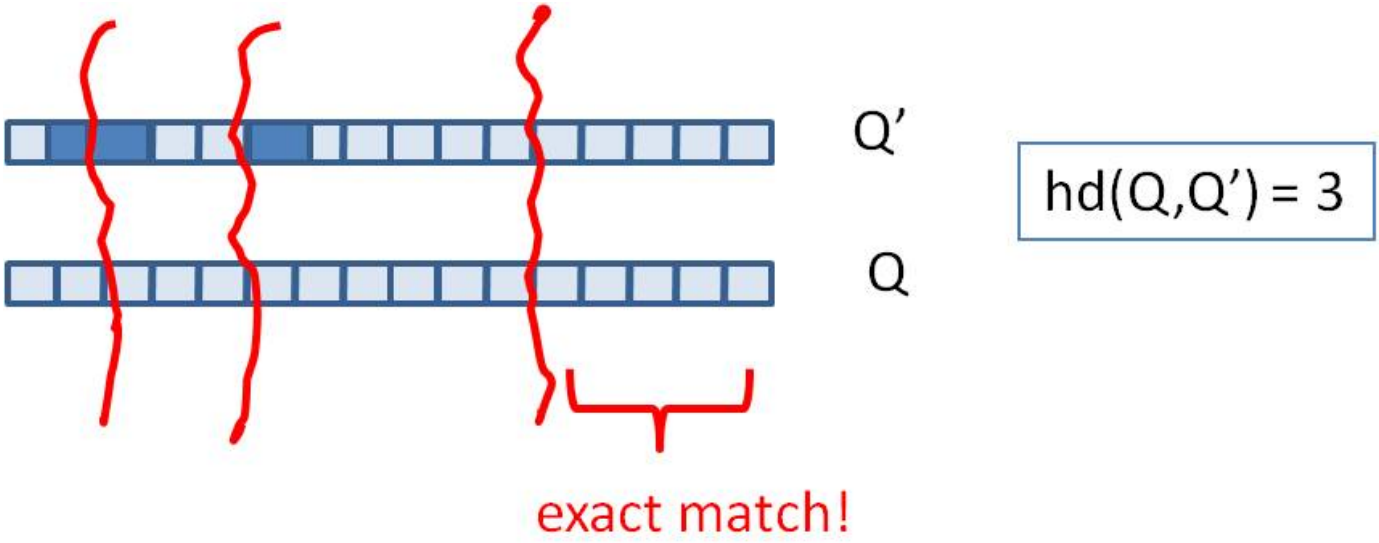
换言之，现在问题转换为：对于64位的SimHash值，我们只要找到海明距离在3以内的所有签名，即可找出所有相似的短语。

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。

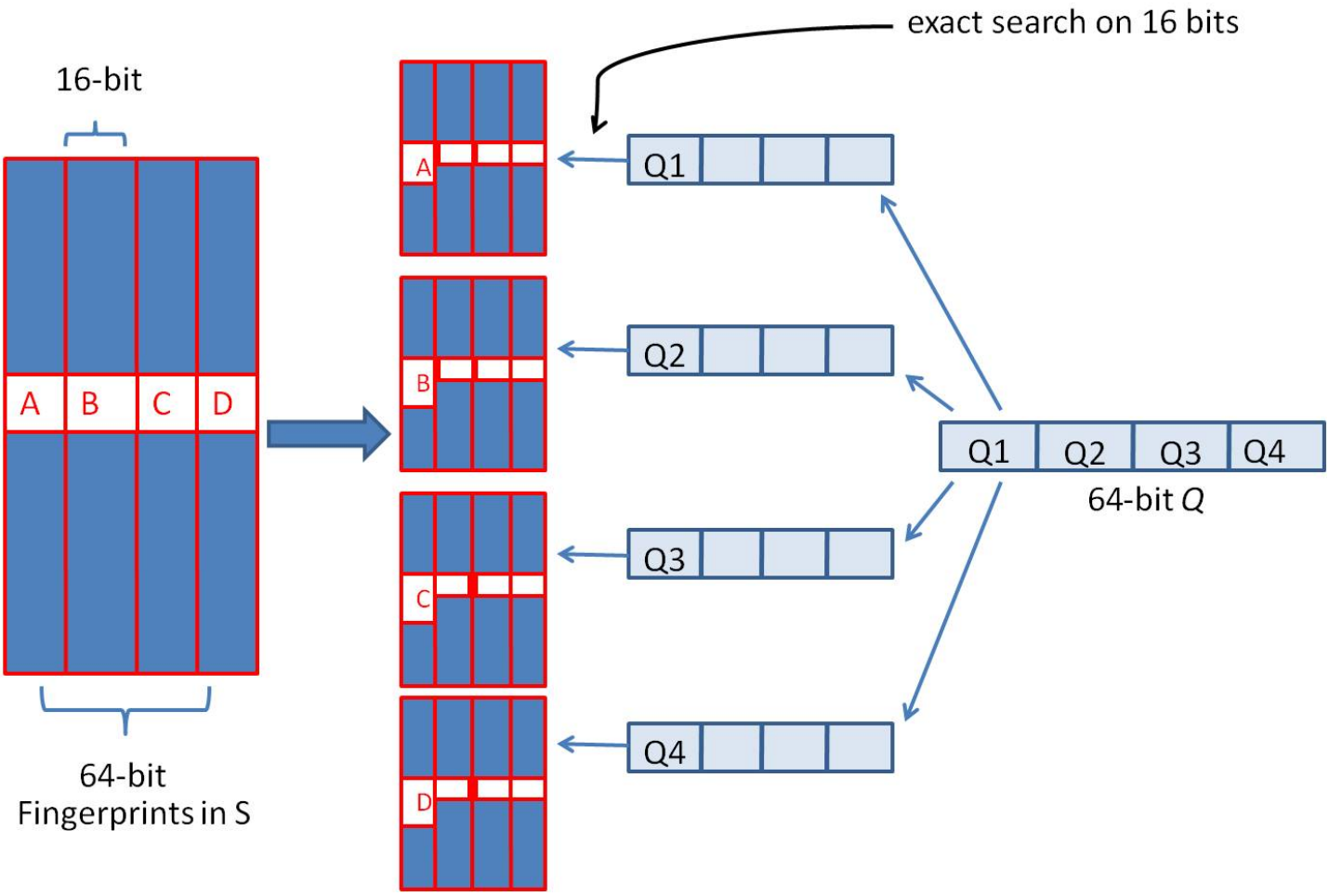
这两种方案，要么时间复杂度高，要么空间复杂度复杂，能否有一种方案可以达到时空复杂度的绝佳平衡呢？答案是肯定的：

- 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。

6.4 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=625000Byte，也就是0.625M，排序后输出。 第二次，扫描输入文件时，只处理4999999-10000000的数据项，也只需要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_1024_8) ~= 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），1~4999999，和5000000~9999999。先遍历一趟，首先排序处理1~4999999之间的整数（用5000000/8=625000个字的存储空间来排序0~4999999之间的整数），然后再第二趟，对5000001~1000000之间的整数进行排序处理。

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(Nlogn)	1M

（多路归并，时间复杂度为O（k_n/k_logn/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。

6.5 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：

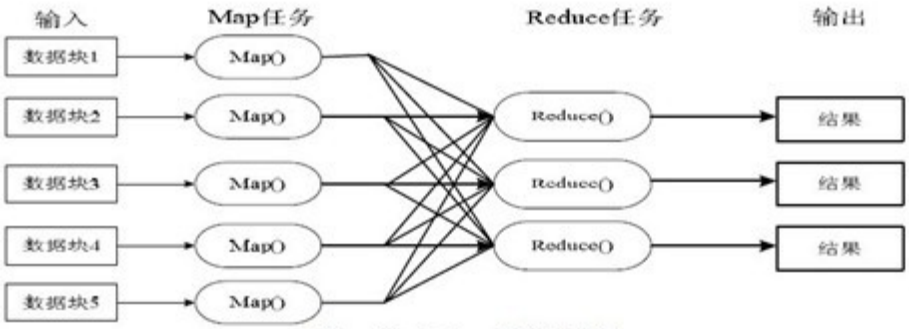


图 1　MapReduce 的处理流程

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：

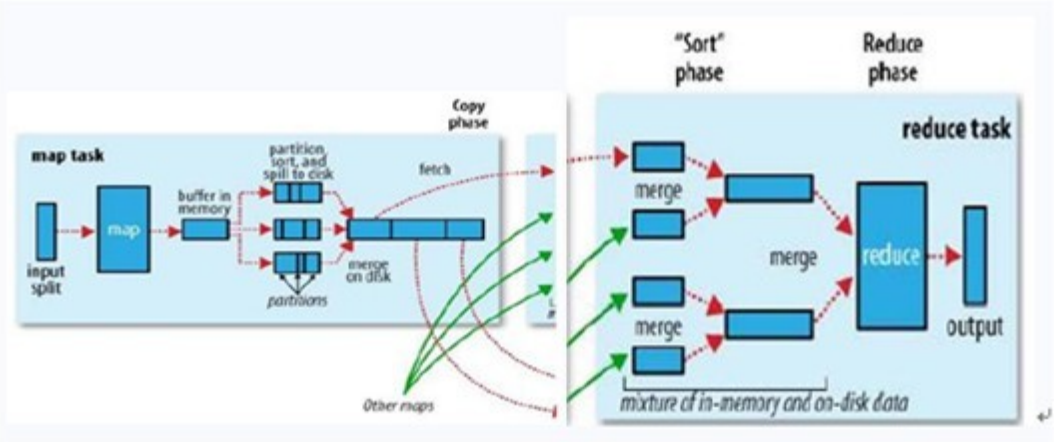
表 1　Map 函数和 Reduce 函数的描述

函数	输入	输出	说明
Map	<k1,v1>	List(<k2,v2>)	1.将小数据集进一步解析成一批<key,value>对，输入Map函数中进行处理。 2.每一个输入的<k1,v1>会输出一批<k2,v2>。<k2,v2>是计算的中间结果。
Reduce	<k2,List(v2)>	<k3,v3>	输入的中间结果<k2,List(v2)>中的List(v2)表示是一批属于同一个k2的value

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)？

6.6 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。

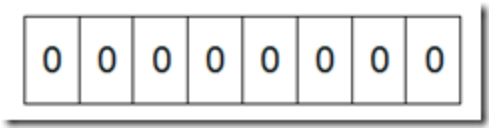
6.7 Bitmap

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)|(0×01

6.8 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：

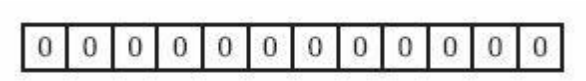
- 当一个元素被加入集合时，通过K个Hash函数将这个元素映射成一个位阵列（Bit array）中的K个点，把它们置为1**。检索时，我们只要看看这些点是不是都是1就（大约）知道集合中有没有它了：
    - 如果这些点有任何一个0，则被检索元素一定不在；
    - 如果都是1，则被检索元素很可能在。

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。

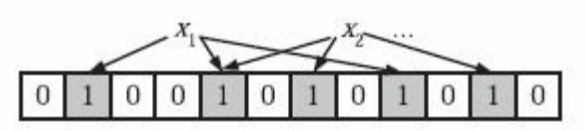
但Bloom Filter的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter通过极少的错误换取了存储空间的极大节省。

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={x1, x2,...,xn}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,...,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hi(x)就会被置为1（1≤i≤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hi(y)的位置都是1（1≤i≤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y1就不是集合中的元素（因为y1有一处指向了“0”位）。y2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kn1, x2,...,xn}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：

$$\lim_{x \to \infty} \left(1 - \frac{1}{x}\right)^{-x} = e$$

令ρ为位数组中0的比例，则ρ的数学期望E(ρ)= p’ 。在ρ已知的情况下，要求的错误率（false positive rate）为：

$$(1-\rho)^k \approx (1-p')^k \approx (1-p)^k$$

(1-ρ)为位数组中1的比例，(1-ρ)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ρ的数学期望，在实际中ρ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：

$$f' = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k = (1-p')^k$$

$$f = (1 - e^{-kn/m})^k = (1-p)^k$$

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

$$g = -\frac{m}{n} \ln(p) \ln(1-p)$$

根据对称性法则可以很容易看出当p = 1/2，也就是k = ln2· (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≈ (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

$$g' = \frac{1}{n\ln(1-1/m)}\ln(p')\ln(1-p')$$

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ϵ (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 + ϵ (u - n)个元素。在n + ϵ (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2m个不同的组合，进而可以推出，m位的位数组可以表示

$$2^m C_{n+\epsilon(u-n)}^{n}$$

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

$$2^m C_{n+\epsilon(u-n)}^{n} \geq C_u^n$$

即：

$$m \geq \log_2 \frac{C_u^n}{C_{n+\epsilon(u-n)}^{n}} \approx \log_2 \frac{C_u^n}{C_{\epsilon u}^{n}} \geq \log_2 \epsilon^{-n} = n\log_2(1/\epsilon)$$

上式中的近似前提是n和ϵu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log2e≈ 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）。”

6.9 Trie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？

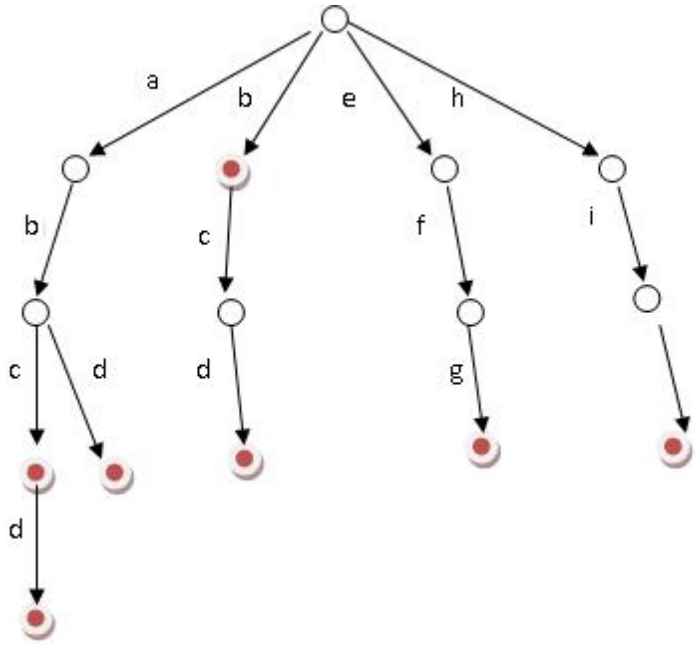
如果我们用最傻的方法，对于每一个单词，我们都要去查找它前面的单词中是否有它。那么这个算法的复杂度就是O(n^2)。显然对于10万的范围难以接受。

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。

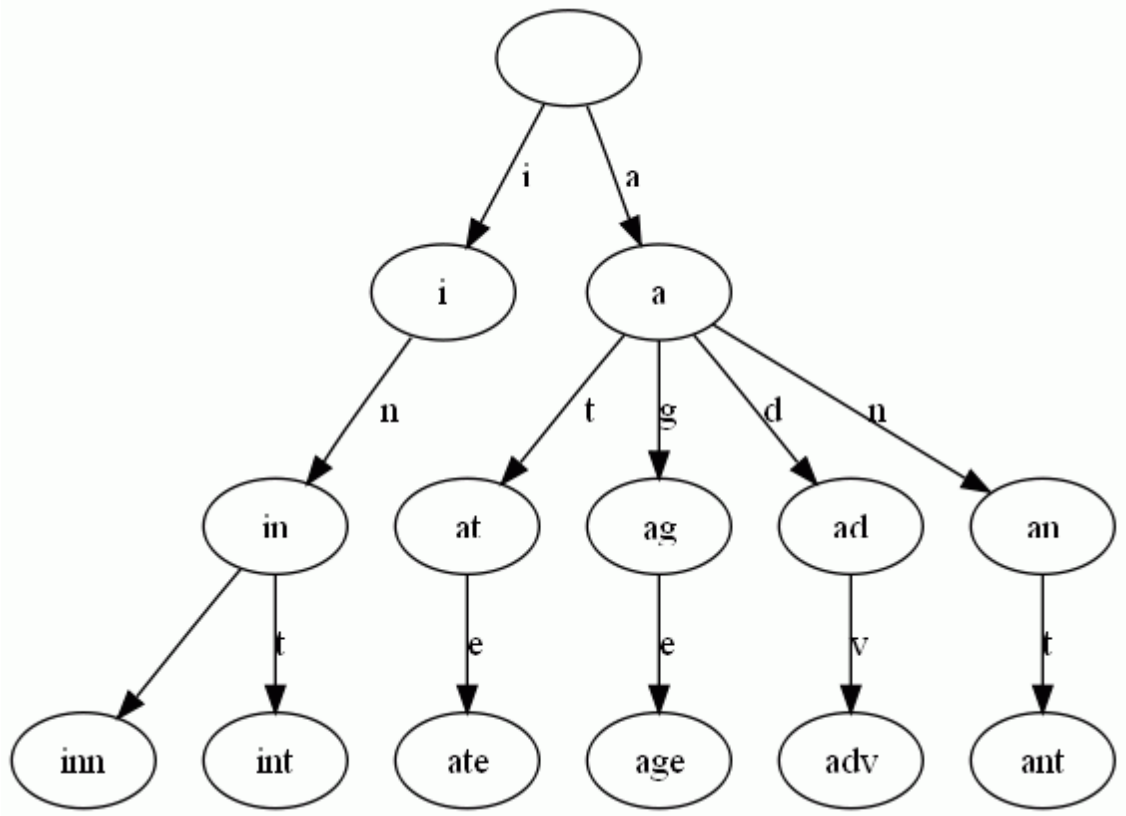
这样一来我们查询和插入可以一起完成，所用时间仅仅为单词长度（在这个例子中，便是10）。这就是一棵trie树。

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*lg10)。所以总的时间复杂度，是O(n*le)与O(n*lg10)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，这些查询串的重复读比较高，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个元素的最小推来对出现频率进行排序。

6.10 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。

6.11 倒排索引

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。

6.15 本章习题

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0（ n ）里，找到某个人的十度好友。

12

海量记录，记录形式如下： TERMID URLNOCOUNT 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。

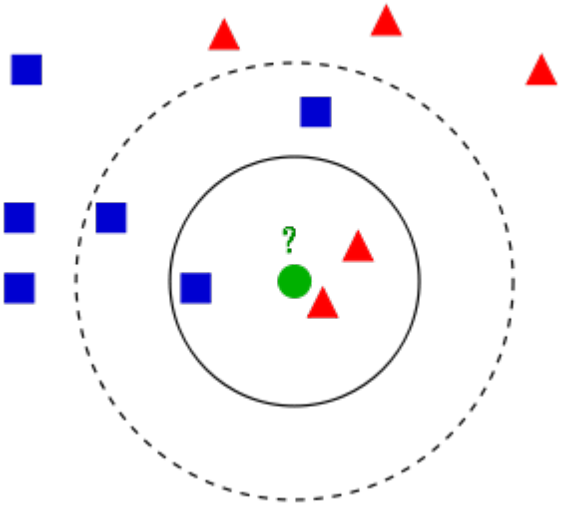
第七章 机器学习

7.1 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，也是可以是其它距离，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 = (x1,...,xn) 和 y = (y1,...,yn) 之间的距离为：

$$d(x,y) := \sqrt{(x_1-y_1)^2+(x_2-y_2)^2+\cdots+(x_n-y_n)^2} = \sqrt{\sum_{i=1}^{n}(x_i-y_i)^2}$$

(1)二维平面上两点a(x1,y1)与b(x2,y2)间的欧氏距离：

$$d_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

(2)三维空间两点a(x1,y1,z1)与b(x2,y2,z2)间的欧氏距离：

$$d_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

(3)两个n维向量a(x11,x12,...,x1n)与 b(x21,x22,...,x2n)间的欧氏距离：

$$d_{12} = \sqrt{\sum_{k=1}^{n} (x_{1k} - x_{2k})^2}$$

也可以用表示成向量运算的形式：

$$d_{12} = \sqrt{(a - b)(a - b)^T}$$

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& v1, const vector& v2)
{
    assert(v1.size() == v2.size());
    double ret = 0.0;
    for (vector::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：$|x_1 - x_2| + |y_1 - y_2|$，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

$$d_{12} = |x_1 - x_2| + |y_1 - y_2|$$

(2)两个n维向量a(x11,x12,...,x1n)与 b(x21,x22,...,x2n)间的曼哈顿距离

$$d_{12} = \sum_{k=1}^{n} |x_{1k} - x_{2k}|$$

3.**切比雪夫距离**，若二个向量或二个点p 、and q，其座标分别为及，则两者之间的切比雪夫距离定义如下：$D_{\text{Chebyshev}}(p,q) := \max_i(|p_i - q_i|)$.，

这也等于以下Lp度量的极值：$\lim_{k\to\infty}\left(\sum_{i=1}^{n}|p_i-q_i|^k\right)^{1/k}$，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：$D_{\text{Chess}} = \max\left(|x_2-x_1|, |y_2-y_1|\right)$。

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(x1,y1)走到格子(x2,y2)最少需要多少步？。你会发现最少步数总是max(|x2-x1|,|y2-y1|)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(x1,y1)与b(x2,y2)间的切比雪夫距离
$d_{12} = \max(|x_1-x_2|, |y_1-y_2|)$

(2)两个n维向量a(x11,x12,...,x1n)与 b(x21,x22,...,x2n)间的切比雪夫距离 $d_{12} = \max_i(|x_{1i}-x_{2i}|)$

这个公式的另一种等价形式是 $d_{12} = \lim_{k\to\infty}\left(\sum_{i=1}^{n}|x_{1i}-x_{2i}|^k\right)^{1/k}$

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(x11,x12,...,x1n)与 b(x21,x22,...,x2n)间的闵可夫斯基距离定义为：

$$d_{12} = \sqrt[p]{\sum_{k=1}^{n}|x_{1k}-x_{2k}|^p}$$

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→∞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：

$$X^* = \frac{X-m}{s}$$

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(x11,x12,…,x1n)与 b(x21,x22,…,x2n)间的标准化欧氏距离的公式：

$$d_{12} = \sqrt{\sum_{k=1}^{n}\left(\frac{x_{1k}-x_{2k}}{s_k}\right)^2}$$

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X1~Xm，协方差矩阵记为S，均值记为向量μ，则其中样本向量X到u的马氏距离表示为：

$$D(X) = \sqrt{(X-\mu)^T S^{-1}(X-\mu)}$$

（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)，Cov(X,Y) = E{ [X-E(X)] [Y-E(Y)]}，其中E为数学期望）

而其中向量Xi与Xj之间的马氏距离定义为：

$$D(X_i,X_j) = \sqrt{(X_i-X_j)^T S^{-1}(X_i-X_j)}$$

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：

$$D(X_i,X_j) = \sqrt{(X_i-X_j)^T (X_i-X_j)}$$

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：

$$D_B(p,q) = -\ln\left(BC(p,q)\right)$$

其中：

$$BC(p,q) = \sum_{x \in X} \sqrt{p(x)q(x)}$$

是Bhattacharyya系数。

对于连续概率分布，Bhattacharyya系数被定义为：

$$BC(p,q) = \int \sqrt{p(x)q(x)}\,dx$$

在 $0 \le BC \le 1$ and $0 \le D_B \le \infty$这两种情况下，巴氏距离D_B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. $0 \le D_B \le \infty D_B \sqrt{1 - BC}$ ）。

对于多变量的高斯分布$p_i = N(m_i, P_i)$，

$D_B = \frac{1}{8}(m_1 - m_2)^T P^{-1}(m_1 - m_2) + \frac{1}{2}\ln\left(\frac{\det P}{\sqrt{\det P_1 \det P_2}}\right)$，和是手段和协方差的分布

$P = \frac{P_1 + P_2}{2}$。

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

$$\mathrm{Bhattacharyya} = \sum_{i=1}^{n} \sqrt{(\Sigma \mathbf{a}_i \cdot \Sigma \mathbf{b}_i)}$$

考虑样品a 和 b ，n是的分区数，并且$\Sigma \mathbf{a}_i$，$\Sigma \mathbf{b}_i$被一个 和 b i的日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：

3.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。
点评：
1、此题反复出现，如上文第38题第4小题9月26日百度一二面试题，10月9日腾讯面试题第1小题，及上面第69题10月13日百度2013校招北京站笔试题第二大道题第3小题，同时，还可以看下这个链

```
//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：

$$\cos\theta = \frac{x_1x_2 + y_1y_2}{\sqrt{x_1^2 + y_1^2}\sqrt{x_2^2 + y_2^2}}$$

(2) 两个n维样本点a(x11,x12,...,x1n)和b(x21,x22,...,x2n)的夹角余弦

$$\cos(\theta) = \frac{a \cdot b}{|a|\,|b|}$$

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：

$$\cos(\theta) = \frac{\sum_{k=1}^{n} x_{1k}\, x_{2k}}{\sqrt{\sum_{k=1}^{n} x_{1k}^{2}}\,\sqrt{\sum_{k=1}^{n} x_{2k}^{2}}}$$

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。

$$\mathrm{J(A,B)} = \frac{|A \cap B|}{|A \cup B|}$$

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：

$$\mathrm{J_\delta(A,B)} = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

杰卡德距离用两个集合中不同元素占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：

$$J = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}.$$

这里M11+M01+M10可理解为A与B的并集的元素个数，而M11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'：

$$J' = \frac{M_{01} + M_{10}}{M_{01} + M_{10} + M_{11}}.$$

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：

$$\rho_{XY} = \frac{Cov(X,Y)}{\sqrt{D(X)}\sqrt{D(Y)}} = \frac{E((X-EX)(Y-EY))}{\sqrt{D(X)}\sqrt{D(Y)}}$$

(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ρ_{XY}）

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：

$$D_{xy} = 1 - \rho_{XY}$$

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：

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y},$$

以上方程定义了总体相关系数, 一般表示成希腊字母ρ(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：

$$r = \frac{\sum_{i=1}^{n}(X_i-\bar{X})(Y_i-\bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i-\bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i-\bar{Y})^2}}.$$

一种等价表达式的是表示成标准分的均值。基于(Xi, Yi)的样本点，样本皮尔逊系数是

$$r = \frac{1}{n-1}\sum_{i=1}^{n}\left(\frac{X_i-\bar{X}}{s_X}\right)\left(\frac{Y_i-\bar{Y}}{s_Y}\right)$$

其中$\frac{X_i-\bar{X}}{s_X}$、$\bar{X}$及s_X，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：$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{E((X-\mu_X)(Y-\mu_Y))}{\sigma_X \sigma_Y} = \frac{E(XY)-E(X)E(Y)}{\sqrt{E(X^2)-E^2(X)}\sqrt{E(Y^2)-E^2(Y)}}$$

  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：

  随机变量的标准差计算公式

  一随机变量 X 的标准差定义为：

  $$\sigma = \sqrt{\mathrm{E}((X-\mathrm{E}(X))^2)} = \sqrt{\mathrm{E}(X^2)-(\mathrm{E}(X))^2}$$

- 公式二：$$\rho_{X,Y} = \frac{N\sum XY - \sum X \sum Y}{\sqrt{N\sum X^2-(\sum X)^2}\sqrt{N\sum Y^2-(\sum Y)^2}}$$

- 公式三：$$\rho_{X,Y} = \frac{\sum (X-\overline{X})(Y-\overline{Y})}{\sqrt{\sum (X-\overline{X})^2 \sum (Y-\overline{Y})^2}}$$

- 公式四：

$$\rho_{X,Y} = \frac{\sum XY - \frac{\sum X \sum Y}{N}}{\sqrt{(\sum X^2 - \frac{(\sum X)^2}{N})(\sum Y^2 - \frac{(\sum Y)^2}{N})}}$$

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：

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

所以, 根据这个最朴素的理解,我们可以将公式依次精简为:

$$r_{xy}=\frac{\sum ZxZy}{N}$$

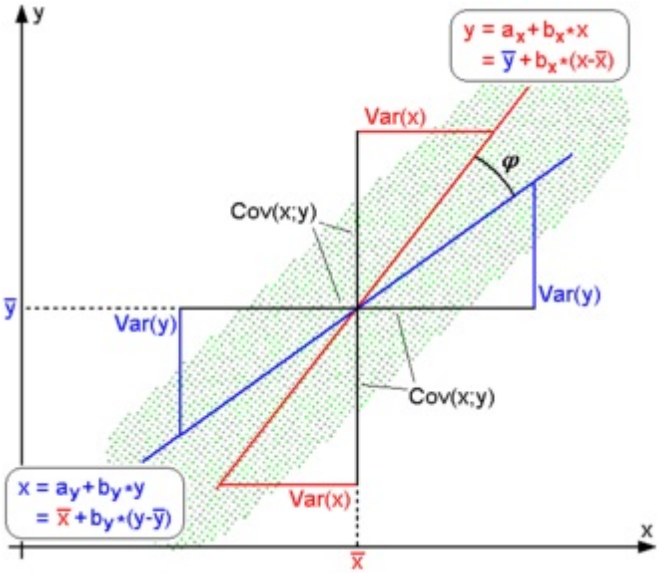
$$=\frac{\sum(\frac{X-\bar{X}}{Sx})(\frac{Y-\bar{Y}}{Sy})}{N}$$

$$=\frac{\sum(X-\bar{X})(Y-\bar{Y})}{N\bullet SxSy}$$

$$=\frac{\sum(X-\bar{X})(Y-\bar{Y})}{N\bullet(\sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_i-\bar{X})^2})(\sqrt{\frac{1}{n}\sum_{i=1}^{n}(Y_i-\bar{Y})^2})}$$

$$=\frac{\sum(X-\bar{X})(Y-\bar{Y})}{(\sqrt{\sum_{i=1}^{n}(X_i-\bar{X})^2})(\sqrt{\sum_{i=1}^{n}(Y_i-\bar{Y})^2})}$$

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:

$$\cos\theta = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \, \|\mathbf{y}\|} = \frac{2.93}{\sqrt{103}\sqrt{0.0983}} = 0.920814711.$$

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

$$\cos\theta = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \, \|\mathbf{y}\|} = \frac{0.308}{\sqrt{30.8}\sqrt{0.00308}} = 1 = \rho_{xy},$$

(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累，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。

7.2 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：

$$w^T x + b = 0$$

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

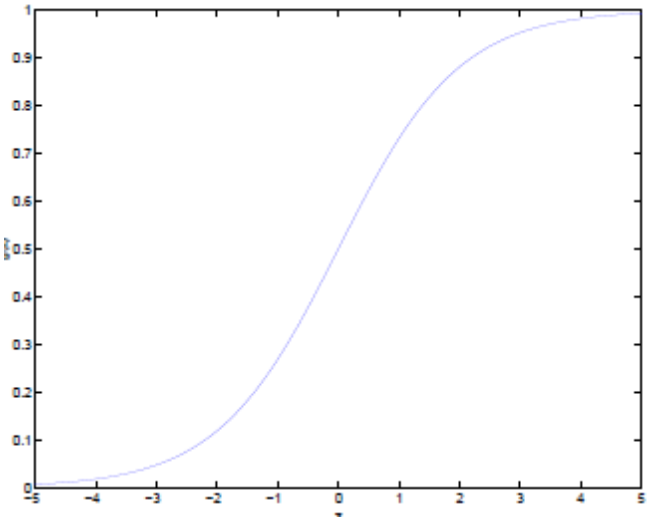
Logistic回归目的是从特征学习出一个0/1分类模型，而这个模型是将特性的线性组合作为自变量，由于自变量的取值范围是负无穷到正无穷。因此，使用logistic函数（或称作sigmoid函数）将自变量映射到(0,1)上，映射后的值被认为是属于y=1的概率。

假设函数

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}},$$

其中x是n维特征向量，函数g就是logistic函数。

而 $g(z) = \frac{1}{1+e^{-z}}$ 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。

$$\begin{aligned} P(y=1 \mid x;\theta) &= h_\theta(x) \\ P(y=0 \mid x;\theta) &= 1 - h_\theta(x) \end{aligned}$$

从而，当我们要判别一个新来的特征属于哪个类时，只需求$h_\theta(x)$，若大于0.5就是y=1的类，反之属于y=0类。

再审视一下$h_\theta(x)$，发现$h_\theta(x)$只和$\theta^T x$有关，$\theta^T x>0$，那么$h_\theta(x)>0.5$，g(z)只不过是用来映射，真实的类别决定权还在$\theta^T x$。还有当$\theta^T x>>0$时，$h_\theta(x)=1$，反之$h_\theta(x)=0$。如果我们只从$\theta^T x$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$\theta^T x>>0$，而是y=0的特征$\theta^T x<<0$。Logistic回归就是要学习得到θ，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θ替换成w和b。
    - 以前的$\theta^T x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$，其中认为$x_0 = 1$，现在我们替换为b；
    - 后面的$\theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$ 替换为$w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$（即$w^T x$）。

这样，我们让$\theta^T x = w^T x + b$，进一步$h_\theta(x) = g(\theta^T x) = g(w^T x + b)$。也就是说除了y由y=0变为y=-1，只是标记不同外，与logistic回归的形式化表示没区别。

再明确下假设函数

$$h_{w,b}(x) = g(w^T x + b)$$

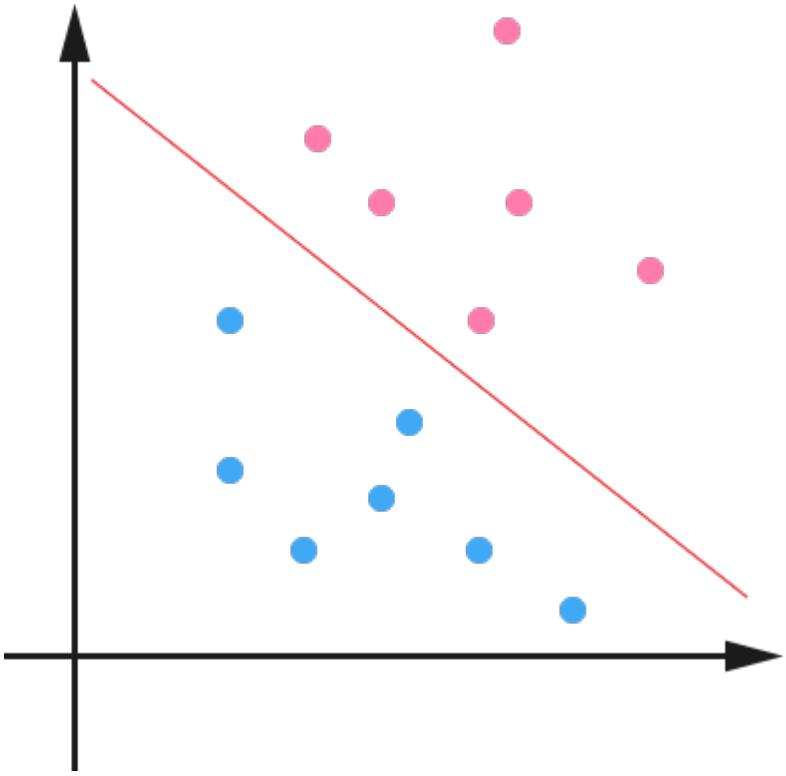
上面提到过我们只需考虑的$\theta^T x$正负问题，而不用关心g(z)，因此我们这里将g(z)做一个简化，将其简单映射到y=-1和y=1上。映射关系如下：

$$g(z) = \begin{cases} 1, & z \geq 0 \\ -1, & z < 0 \end{cases}$$

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)。

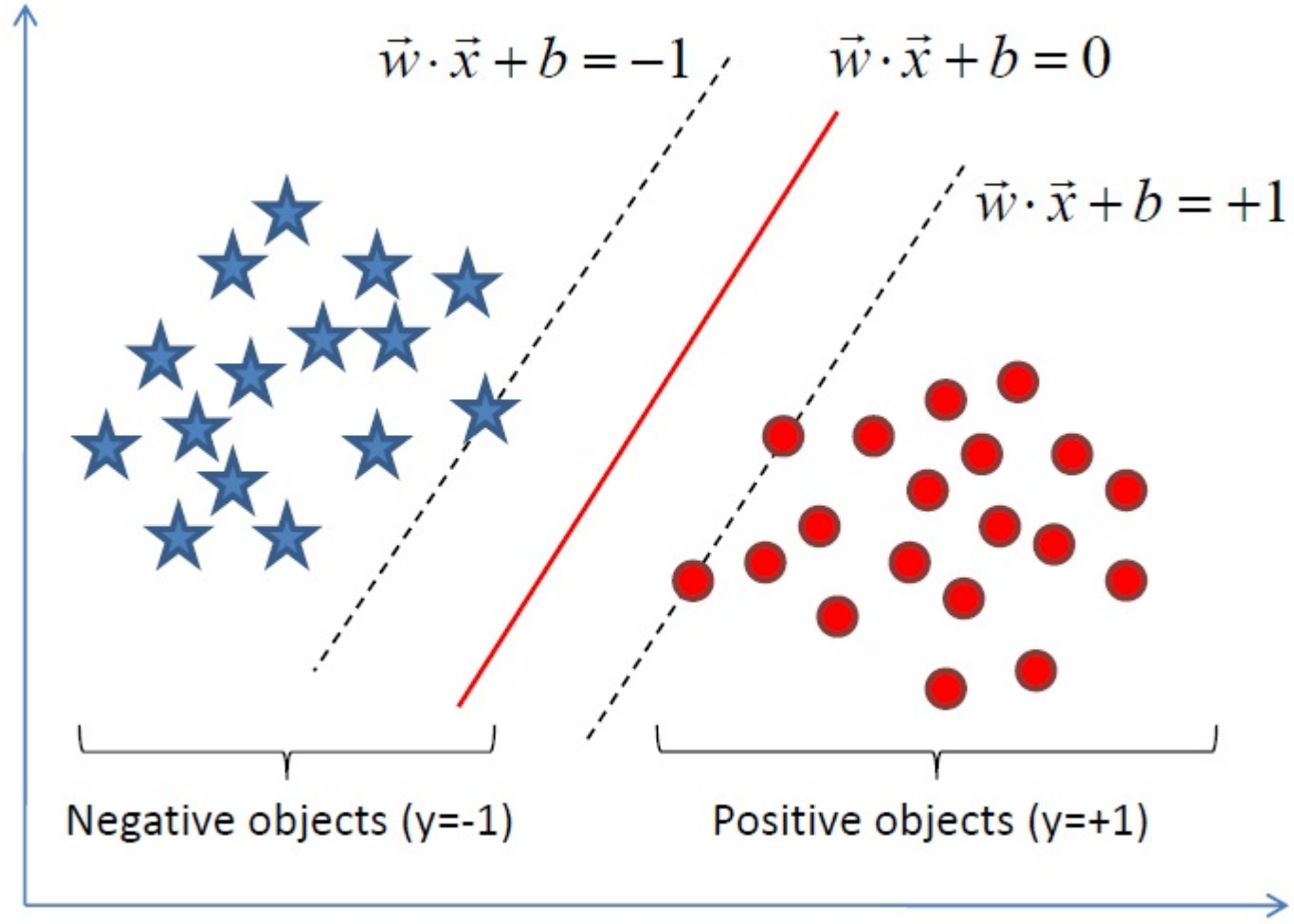


从上图中我们可以看出，这条红颜色的线作为一个超平面，把红颜色的点和蓝颜色的点分开来了，在超平面一边的数据点所对应的y全是 -1 ，而在另一边全是1。

接着，我们可以令分类函数：

$$f(x) = w^T x + b$$

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 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$u=\vec{w}\cdot\vec{x}-b$，与我们之前定义的$f(x)=w^Tx+b$实质是一样的。为什么？因为无论是，还是，不影响最终优化结果。下文你将看到，当我们转化到优化

$$\max \frac{1}{\|w\|}, \quad s.t., y_i(w^Tx_i+b)\geq 1, i=1,\ldots,n$$

的时候，为了求解方便，会把yf(x)令为1，即yf(x)是y(w^x + b)，还是y(w^x - b)，对我们要优化的式子max1/||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_x+b=0确定的情况下，|w_x+b|能够相对的表示点x到距离超平面的远近，而w_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_(w*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：

$$\hat{\gamma} = y(w^T x + b) = yf(x)$$

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(xi，yi)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：

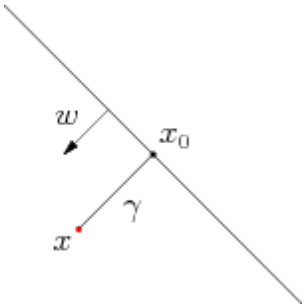
$\hat{\gamma}$ = min $\hat{\gamma}$ i (i=1，...n)

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 x0 ，w 是垂直于超平面的一个向量，$\hat{\gamma}$ 为样本x到分类间隔的距离，



我们有

$$x = x_0 + \gamma \frac{w}{\|w\|}$$

其中，||w||表示的是范数。

又由于 x0 是超平面上的点，满足 f(x0)=0 ，代入超平面的方程即可算出：

$$\gamma = \frac{w^T x + b}{\|w\|} = \frac{f(x)}{\|w\|}$$

不过这里的 $\hat{\gamma}$ 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 $\hat{\gamma}$ 的定义，$\hat{\gamma}$ =y(wTx+b)=yf(x))：

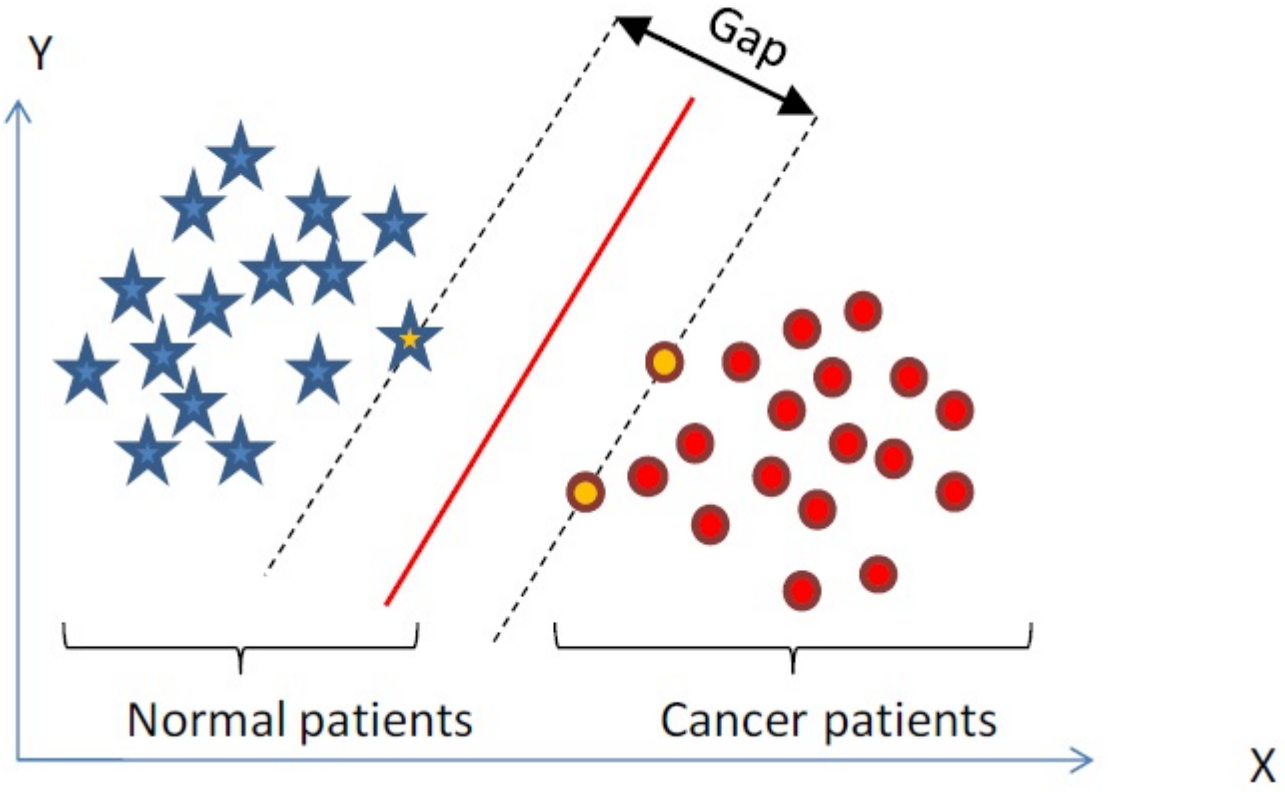
$$\tilde{\gamma} = y\gamma = \frac{\hat{\gamma}}{\|w\|}$$

代人相关式子可以得出：yi*(w/||w|| + b/||w||)。

综上，函数间隔y_(wx+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。



且

1. functional margin 明显是不太适合用来最大化的一个量，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 $f(x) = w^T x + b$ 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 geometrical margin 则没有这个问题，因为除上了 $\|w\|$ 这个分母，所以缩放 w 和 b 的时候 $\hat{\gamma}$ 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 maximum margin classifier 的目标函数可以定义为：

$$\max \tilde{\gamma}$$

当然，还需要满足一定的约束条件：

$$y_i(w^T x_i + b) = \hat{\gamma}_i \geq \hat{\gamma}, \quad i = 1, \ldots, n$$

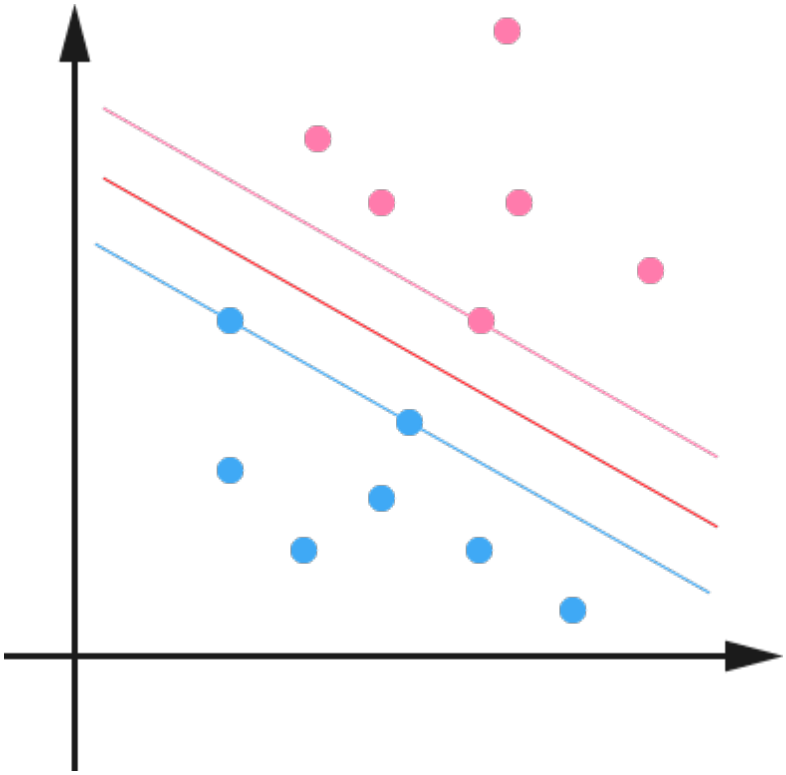
其中 $\hat{\gamma} = \tilde{\gamma}\|w\|$ (等价于 $\hat{\gamma}$ = $\hat{\gamma}$ /||w||，故有稍后的 $\hat{\gamma}$ =1 时， $\hat{\gamma}$ = 1 / ||w||)，处于方便推导和优化的目的，我们可以令 $\hat{\gamma}$ =1(对目标函数的优化没有影响)，此时，上述的目标函数 $\hat{\gamma}$ 转化为：

$$\max \frac{1}{\|w\|}, \quad s.t., y_i(w^T x_i + b) \geq 1, i = 1, \ldots, n$$

其中，s.t.，即subject to的意思，它导出的是约束条件。

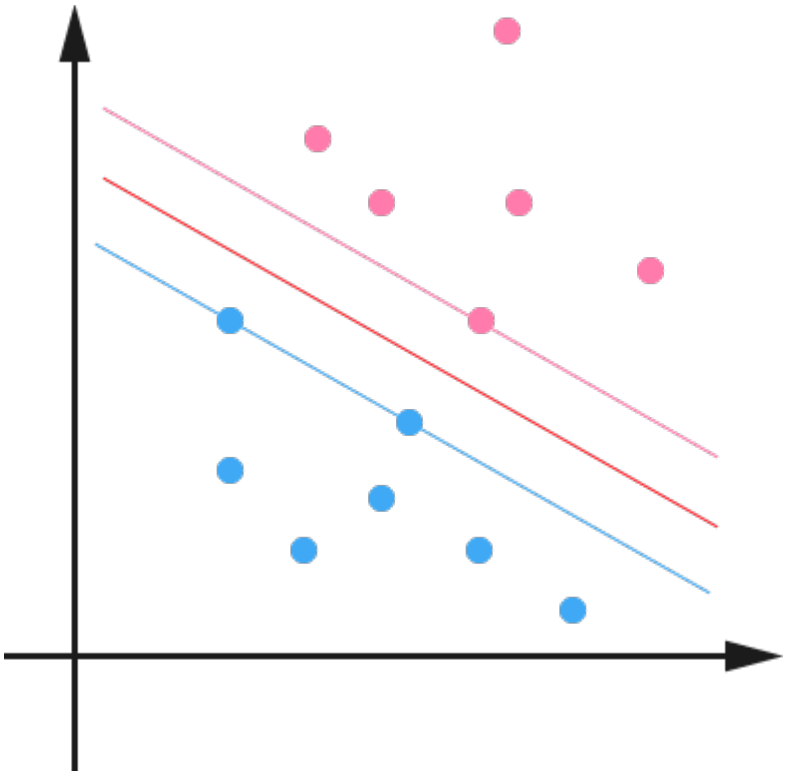
通过求解这个问题，我们就可以找到一个 margin 最大的 classifier ，通过最大化 margin ，我们使得该分类器对数据进行分类时具有了最大的 confidence，从而设计决策最优分类超平面。

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 $\hat{\gamma}$ 的($\hat{\gamma}$ 便是上文所定义的geometrical margin，当令 $\hat{\gamma}$ =1时， $\hat{\gamma}$ 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)：



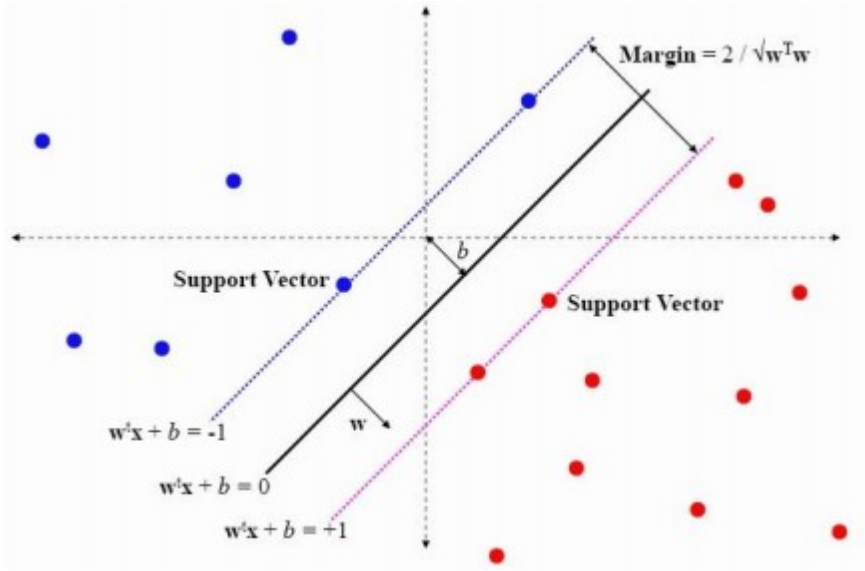
1.5、到底什么是Support Vector

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 $y(w^Tx+b)=1$，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 $y(w^Tx+b)>1$。

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）：

$$\max \frac{1}{\|w\|}, \quad s.t., y_i(w^T x_i + b) \geq 1, i = 1, \ldots, n$$

由于求$\frac{1}{\|w\|}$的最大值相当于求$\frac{1}{2}\|w\|^2$的最小值，所以上述目标函数等价于：

$$\min \frac{1}{2}\|w\|^2 \quad s.t., y_i(w^T x_i + b) \geq 1, i = 1, \ldots, n$$

- 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α，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)：

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{n} \alpha_i \left(y_i(w^T x_i + b) - 1 \right)$$

然后我们令

$$\theta(w) = \max_{\alpha_i \geq 0} \mathcal{L}(w, b, \alpha)$$

容易验证：

- 当某个约束条件不满足时，例如$y_i(w^T x_i + b) < 1$，那么我们显然有$\theta(w) = \infty$（只要令$\alpha_i = \infty$即可）。
- 而当所有约束条件都满足时，则有$\theta(w) = \frac{1}{2}\|w\|^2$，亦即我们最初要最小化的量$\frac{1}{2}\|w\|^2$。

因此，在要求约束条件得到满足的情况下最小化$\theta(w)$，实际上等价于直接最小化α_i（当然，这里也有约束条件，就是≥0,i=1,...,n），因为如果约束条件没有得到满足，$\theta(w)$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：

$$\min_{w,b}\theta(w)=\min_{w,b}\max_{\alpha_i\geq 0}\mathcal{L}(w,b,\alpha)=p^*$$

这里用p^*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：

$$\max_{\alpha_i\geq 0}\min_{w,b}\mathcal{L}(w,b,\alpha)=d^*$$

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d^*来表示。并且，我们有$d^* \leq p^*$，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d^*在这里提供了一个第一个问题的最优值p^*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α的极大。而且，之所以从minmax的原始问题p^*，转化为maxmin的对偶问题d^*，一者因为d^*是p^*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：

$$\begin{aligned}&\min. f(\mathbf{x})\\&\text{s.t.}\quad h_j(\mathbf{x})=0, j=1,\ldots,p,\\&\qquad g_k(\mathbf{x})\leq 0, k=1,\ldots,q,\\&\qquad \mathbf{x}\in\mathbf{X}\subset\Re^n\end{aligned}$$

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：$\mathcal{X}\subset\mathbb{R}^n$为一凸集，$f:\mathcal{X}\to\mathbb{R}$为一凸函数。凸优化就是要找出一点$x^*\in\mathcal{X}$，使得每一$x\in\mathcal{X}$满足$f(x^*)\leq f(x)$。
- 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：

1. $h_j(\mathbf{x}_*)=0,\ j=1,\ldots,p,\ g_k(\mathbf{x}_*)\le 0,\ k=1,\ldots,q,$

2. $\nabla f(\mathbf{x}_*)+\sum_{j=1}^{p}\lambda_j\nabla h_j(\mathbf{x}_*)+\sum_{k=1}^{q}\mu_k\nabla g_k(\mathbf{x}_*)=\mathbf{0},$

$\lambda_j\neq 0,\ \mu_k\ge 0,\ \mu_k g_k(\mathbf{x}_*)=0.$

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gi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，a) 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 ∂L/∂w 和 ∂L/∂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）：

$$\frac{\partial \mathcal{L}}{\partial w}=0\Rightarrow w=\sum_{i=1}^{n}\alpha_i y_i x_i$$
$$\frac{\partial \mathcal{L}}{\partial b}=0\Rightarrow \sum_{i=1}^{n}\alpha_i y_i=0$$

以上结果代回上述的 L：

$$\mathcal{L}(w,b,\alpha)=\frac{1}{2}\|w\|^2-\sum_{i=1}^{n}\alpha_i\left(y_i(w^T x_i+b)-1\right)$$

得到：

$$\begin{aligned}\mathcal{L}(w,b,\alpha)&=\frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^T x_j-\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^T x_j-b\sum_{i=1}^{n}\alpha_i y_i+\sum_{i=1}^{n}\alpha_i\\&=\sum_{i=1}^{n}\alpha_i-\frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^T x_j\end{aligned}$$

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：

$$\mathcal{L}(\mathrm{w},\mathrm{b},\alpha)=\frac{1}{2}\|w\|^2-\sum_{i=1}^{m}\alpha_i\left[y^{(i)}\left(w^Tx^{(i)}+b\right)-1\right]$$

$$=\frac{1}{2}\mathrm{w}^T\mathrm{w}-\sum_{i=1}^{m}\alpha_i y^{(i)}w^Tx^{(i)}-\sum_{i=1}^{m}\alpha_i y^{(i)}b+\sum_{i=1}^{m}\alpha_i$$

$$=\frac{1}{2}\mathrm{w}^T\sum_{i=1}^{m}\alpha_i y^{(i)}x^{(i)}-\sum_{i=1}^{m}\alpha_i y^{(i)}w^Tx^{(i)}-\sum_{i=1}^{m}\alpha_i y^{(i)}b+\sum_{i=1}^{m}\alpha_i$$

$$=\frac{1}{2}\mathrm{w}^T\sum_{i=1}^{m}\alpha_i y^{(i)}x^{(i)}-w^T\sum_{i=1}^{m}\alpha_i y^{(i)}x^{(i)}-\sum_{i=1}^{m}\alpha_i y^{(i)}b+\sum_{i=1}^{m}\alpha_i$$

$$=-\frac{1}{2}\mathrm{w}^T\sum_{i=1}^{m}\alpha_i y^{(i)}x^{(i)}-\sum_{i=1}^{m}\alpha_i y^{(i)}b+\sum_{i=1}^{m}\alpha_i$$

$$=-\frac{1}{2}\mathrm{w}^T\sum_{i=1}^{m}\alpha_i y^{(i)}x^{(i)}-b\sum_{i=1}^{m}\alpha_i y^{(i)}+\sum_{i=1}^{m}\alpha_i$$

$$=-\frac{1}{2}\left(\sum_{i=1}^{m}\alpha_i y^{(i)}x^{(i)}\right)^T\sum_{i=1}^{m}\alpha_i y^{(i)}x^{(i)}-b\sum_{i=1}^{m}\alpha_i y^{(i)}+\sum_{i=1}^{m}\alpha_i$$

$$=-\frac{1}{2}\sum_{i=1}^{m}\alpha_i y^{(i)}\left(x^{(i)}\right)^T\sum_{i=1}^{m}\alpha_i y^{(i)}x^{(i)}-b\sum_{i=1}^{m}\alpha_i y^{(i)}+\sum_{i=1}^{m}\alpha_i$$

$$=-\frac{1}{2}\sum_{i=1,j=1}^{m}\alpha_i y^{(i)}\left(x^{(i)}\right)^T\alpha_j y^{(j)}x^{(j)}-b\sum_{i=1}^{m}\alpha_i y^{(i)}+\sum_{i=1}^{m}\alpha_i$$

最后，得到：

$$\mathcal{L}(w,b,\alpha)=\frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^Tx_j-\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^Tx_j-b\sum_{i=1}^{n}\alpha_i y_i+\sum_{i=1}^{n}\alpha_i$$
$$=\sum_{i=1}^{n}\alpha_i-\frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^Tx_j$$

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ai和yi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_i，然后下文的第2步，求出了α_i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$f(x)=w^Tx+b$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_i，根据 $w=\sum_{i=1}^{m}\alpha_i y^{(i)}x^{(i)}$，即可求出w。然后通过 $b^*=-\frac{\max_{i:y^{(i)}=-1}w^{*T}x^{(i)}+\min_{i:y^{(i)}=1}w^{*T}x^{(i)}}{2}$，即可求出b)

$$\begin{aligned}&\max_{\alpha}\sum_{i=1}^{n}\alpha_i-\frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^T x_j\\&s.t.,\alpha_i\geq 0,i=1,\ldots,n\\&\sum_{i=1}^{n}\alpha_i y_i=0\end{aligned}$$

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：

$$\begin{aligned}\max_{\alpha}\quad & W(\alpha)=\sum_{i=1}^{m}\alpha_i-\frac{1}{2}\sum_{i,j=1}^{m}y^{(i)}y^{(j)}\alpha_i\alpha_j\langle x^{(i)},x^{(j)}\rangle\\\text{s.t.}\quad & 0\leq\alpha_i\leq C,\quad i=1,\ldots,m\\&\sum_{i=1}^{m}\alpha_i y^{(i)}=0,\end{aligned}$$

要解决的是在参数$\{\alpha_1,\alpha_2,\ldots,\alpha_n\}$上求最大值W的问题，至于$x^{(i)}$和$y^{(i)}$都是已知数（其中 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 ）。

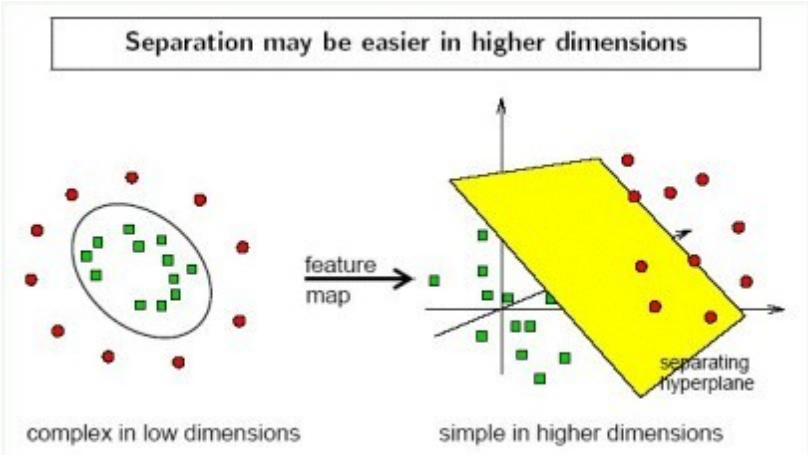
要了解这个SMO算法是如何推导的，请跳到下文第3.5节、SMO算法。

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。

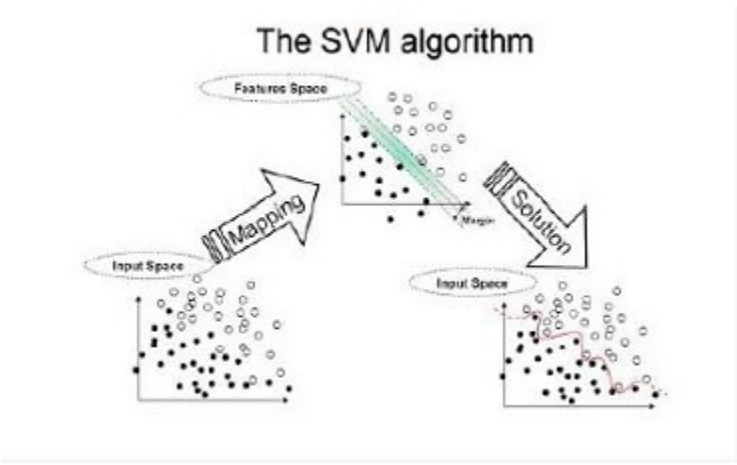
2.2、核函数Kernel

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了开来)：



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：

$$f(\mathbf{x}) = \sum_{i=1}^{N} w_i \phi_i(\mathbf{x}) + b,$$

这里ϕ：X->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：

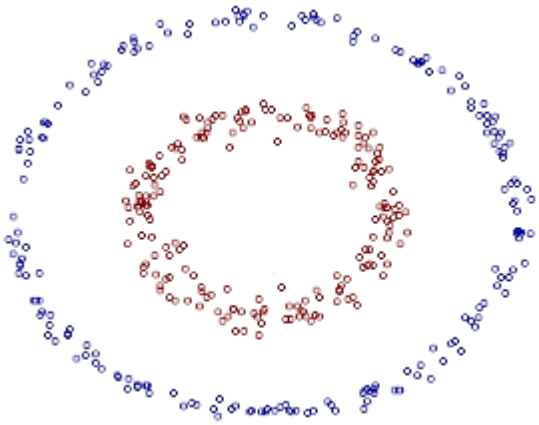
$$f(\mathbf{x}) = \sum_{i=1}^{\ell} \alpha_i y_i \langle \boldsymbol{\phi}(\mathbf{x}_i) \cdot \boldsymbol{\phi}(\mathbf{x}) \rangle + b.$$

如果有一种方式可以在特征空间中直接计算内积〈φ(xi · φ(x)〉，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，这样直接计算法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(-X，满足 $K(\mathbf{x}, \mathbf{z}) = \langle \boldsymbol{\phi}(\mathbf{x}) \cdot \boldsymbol{\phi}(\mathbf{z}) \rangle,$ ，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 X1 和 X2 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：

$$a_1 X_1 + a_2 X_1^2 + a_3 X_2 + a_4 X_2^2 + a_5 X_1 X_2 + a_6 = 0$$

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 Z1=X1, Z2=X21, Z3=X2, Z4=X22, Z5=X1X2，那么显然，上面的方程在新的坐标系下可以写作：

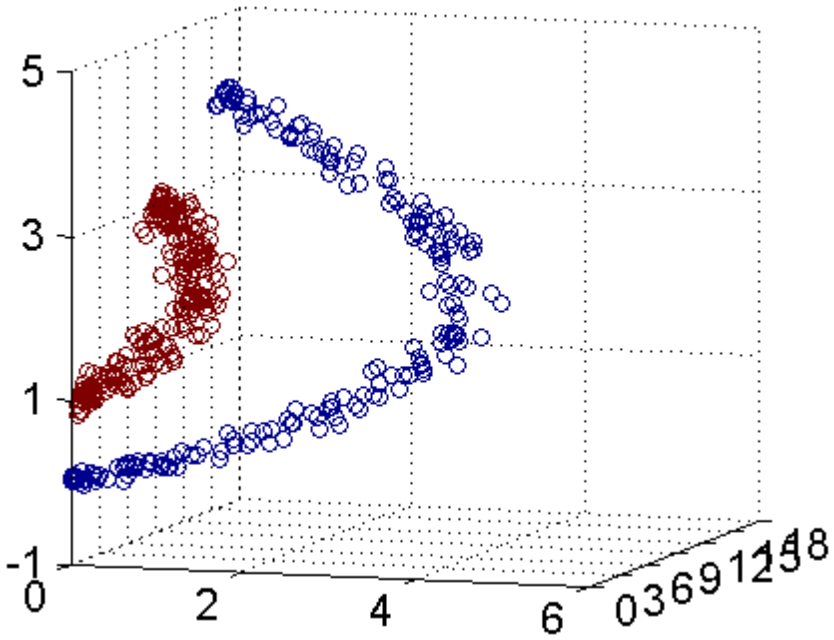
$$\sum_{i=1}^{5} a_i Z_i + a_6 = 0$$

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 ϕ:R2→R5 ，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 X2 轴上的一个正圆）：

$$\sum_{i=1}^{5} a_i Z_i + a_6 = 0$$

因此我只需要把它映射到 Z1=X21, Z2=X22, Z3=X2 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：

$$f(x) = \sum_{i=1}^{n} \alpha_i y_i \langle x_i, x \rangle + b$$

映射过后的空间是：

$$f(x) = \sum_{i=1}^{n} \alpha_i y_i \langle \phi(x_i), \phi(x) \rangle + b$$

而其中的 α 也是通过求解如下 dual 问题而得到的：

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \langle \phi(x_i), \phi(x_j) \rangle$$
$$s.t., \alpha_i \geq 0, i = 1, \ldots, n$$
$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

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 ϕ(·) 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。所以就需要 Kernel 出马了。

还是从最开始的简单例子出发，设两个向量$x_1 = (\eta_1, \eta_2)^T$和$x_2 = (\xi_1, \xi_2)^T$，而ϕ(·)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：

$$\langle \phi(x_1), \phi(x_2) \rangle = \eta_1\xi_1 + \eta_1^2\xi_1^2 + \eta_2\xi_2 + \eta_2^2\xi_2^2 + \eta_1\eta_2\xi_1\xi_2$$

（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x1，x2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：

$$(\langle x_1, x_2 \rangle + 1)^2 = 2\eta_1\xi_1 + \eta_1^2\xi_1^2 + 2\eta_2\xi_2 + \eta_2^2\xi_2^2 + 2\eta_1\eta_2\xi_1\xi_2 + 1$$

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

$$\varphi(X_1, X_2) = (\sqrt{2}X_1, X_1^2, \sqrt{2}X_2, X_2^2, \sqrt{2}X_1X_2, 1)^T$$

之后的内积$\langle \varphi(x_1), \varphi(x_2) \rangle$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：

$$\kappa(x_1, x_2) = (\langle x_1, x_2 \rangle + 1)^2$$

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：

$$\sum_{i=1}^{n} \alpha_i y_i \kappa(x_i, x) + b$$

其中 由如下 dual 问题计算而得：

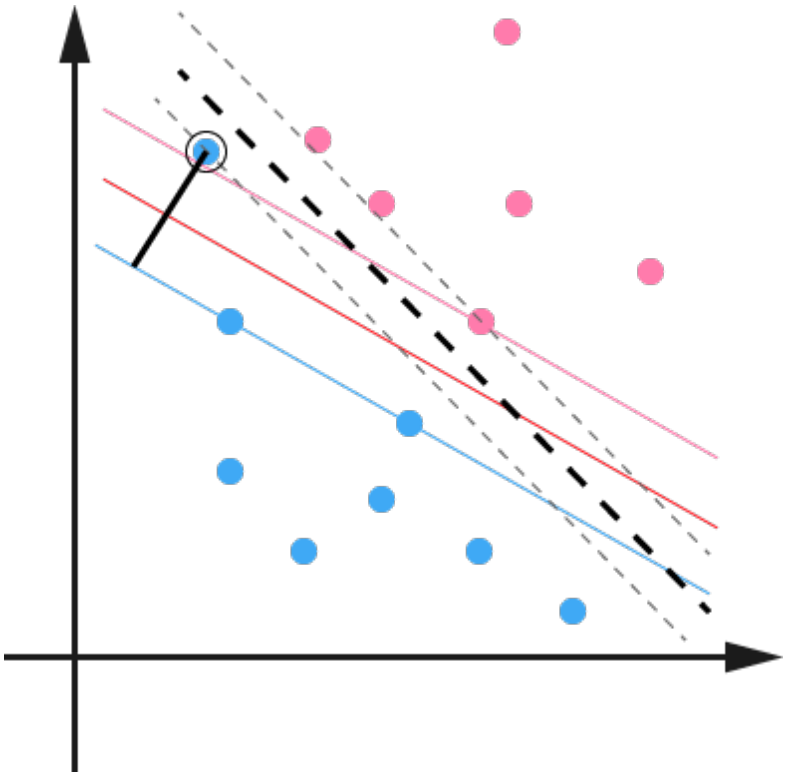
$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \kappa(x_i, x_j)$$
$$s.t., \alpha_i \geq 0, i = 1, \ldots, n$$
$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

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，使得非线性的的情况也能处理。虽然通过映射 ϕ(·) 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不被挤歪了，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：

$$y_i(w^T x_i + b) \geq 1, \quad i = 1, \ldots, n$$

现在考虑到outlier问题，约束条件变成了：

$$y_i(w^T x_i + b) \geq 1 - \xi_i, \quad i = 1, \ldots, n$$

其中 $\xi_i \geq 0$ 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：

$$\min \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{n} \xi_i$$

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 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：

$$\begin{aligned} &\min \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{n} \xi_i \\ &s.t., y_i(w^T x_i + b) \geq 1 - \xi_i, i = 1, \ldots, n \\ &\quad \xi_i \geq 0, i = 1, \ldots, n \end{aligned}$$

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：

$$\mathcal{L}(w, b, \xi, \alpha, r) = \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i \left(y_i(w^T x_i + b) - 1 + \xi_i \right) - \sum_{i=1}^{n} r_i \xi_i$$

分析方法和前面一样，转换为另一个问题之后，我们先让 $\mathcal{L}$ 针对w、b和 ξ_i 最小化：

$$\begin{aligned} &\frac{\partial \mathcal{L}}{\partial w} = 0 \Rightarrow w = \sum_{i=1}^{n} \alpha_i y_i x_i \\ &\frac{\partial \mathcal{L}}{\partial b} = 0 \Rightarrow \sum_{i=1}^{n} \alpha_i y_i = 0 \\ &\frac{\partial \mathcal{L}}{\partial \xi_i} = 0 \Rightarrow C - \alpha_i - r_i = 0, \quad i = 1, \ldots, n \end{aligned}$$

将 w 带回 $\mathcal{L}$ 并化简，得到和原来一样的目标函数：

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle$$

不过，由于我们得到$C - \alpha_i - r_i = 0$而又有ri >= 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：

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle$$
$$s.t., 0 \leq \alpha_i \leq C, i = 1, \ldots, n$$
$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

把前后的结果对比一下（错误修正：图中的Dual formulation中的Minimize应为maxmize）：

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$\langle x_i, x_j \rangle$换成$\kappa(x_i, x_j)$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+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a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a等价，而a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》）：

（1）0-1 损失函数

$$L(Y, f(X)) = \begin{cases} 1, Y \neq f(X) \\ 0, Y = f(X) \end{cases}$$

（2）平方损失函数

$$L(Y, f(X)) = (Y - f(X))^2$$

（3）绝对损失函数

$$L(Y, f(X)) = |Y - f(X)|$$

（4）对数损失函数

$$L(Y, P(Y|X)) = -\log P(Y|X)$$

给定一个训练数据集

$$T = \{(x_1, y_1), (x_2, y_2), \ldots (x_i, y_i) \ldots, (x_N, y_N)\}$$

模型 f(X)关于训练数据集的平均损失称为经验风险，如下：

$$R_{emp}(f) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i))$$

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

$$u = \vec{w} \cdot \vec{x} - b$$

再三强调，这个u与我们之前定义的 $f(x) = w^T x + b$ 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：

$$\min_{w,b} \frac{1}{2} \| \vec{w} \|^2 \text{ subject to } y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1, \forall i$$

求导得到：

$$\vec{w} = \sum_{i=1}^{N} y_i \alpha_i \vec{x}_i, \quad b = \vec{w} \cdot \vec{x}_k - y_k \text{ for some } \alpha_k > 0$$

代入 $u = \vec{w} \cdot \vec{x} - b$ 中，可得 $u = \sum_{j=1}^{N} y_j \alpha_j K(\vec{x}_j, \vec{x}) - b$。

引入对偶因子后，得：

$$\min_{\alpha} \Psi(\vec{\alpha}) = \min_{\alpha} \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j (\vec{x}_i \cdot \vec{x}_j) \alpha_i \alpha_j - \sum_{i=1}^{N} \alpha_i$$

s.t： $\alpha_i \geq 0, \forall i,$ 且 $\sum_{i=1}^{N} y_i \alpha_i = 0.$

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yi也与之前的 $y^{(i)}$ 等价，yj亦如此。

经过加入松弛变量后，模型修改为：

$$\min_{w,b,\xi} \frac{1}{2} \| \vec{w} \|^2 + C \sum_{i=1}^{N} \xi_i \text{ subject to } y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1 - \xi_i, \forall i$$

$$0 \leq \alpha_i \leq C, \forall i$$

从而最终我们的问题变为：

$$\min_{\alpha} \Psi(\vec{\alpha}) = \min_{\alpha} \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} y_i y_j K(\vec{x}_i, \vec{x}_j) \alpha_i \alpha_j - \sum_{i=1}^{N} \alpha_i$$

$$0 \leq \alpha_i \leq C, \forall i,$$

$$\sum_{i=1}^{N} y_i \alpha_i = 0.$$

继而，根据KKT条件可以得出其中取值的意义为：

$$\begin{aligned} \alpha_i = 0 &\Leftrightarrow y_i u_i \geq 1, \\ 0 < \alpha_i < C &\Leftrightarrow y_i u_i = 1, \\ \alpha_i = C &\Leftrightarrow y_i u_i \leq 1. \end{aligned}$$

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yi*f(xi)>=0）；
2. 对于第2种情况，表明了[![](https://github.com/julycoding/The-Art-Of-Pr

附录 更多题型

附录A 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。

附录B 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,重复第一步。参考代码如下所示：

```
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 、每次从通里面拿出来两个球；
- ii、如果取出的是两个同色的求，就再放入一个黑球；
- ii、如果取出的是两个异色的求，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。

附录C 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，现知道只有一个和其它的重量不同，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

附录D 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ServerIPIndex[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ServerIPIndex[QQNUM%n]与ServerIPIndex[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和变化，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。

附录E 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。

附录F 网络协议

1

请简单阐述TCP连接的三次握手。