假设你们班级100个同学每个人的学号是由院系-年级-班级和编号组成，例如学号为01100168表示是1系，10级1班的68号。为了快速查找到68号的成绩信息，可以建立一张表，但是不能用学号作为下标，学号的数值实在太大。因此将学号除以1100100取余，即得到编号作为该表的下标，那么，要查找学号为01100168的成绩的时候，只要直接访问表下标为68的数据即可。这就能够在O(1)时间复杂度内完成成绩查找。实际上这里就用到了散列的思想。

# 原理

理想散列表（哈希表）是一个包含关键字的具有**固定大小**的数组，它能够以**常数时间执行插入，删除和查找操作**。

·这块连续的存储空间称为散列表或哈希表(hash table)

·每个关键字被映射到0到数组大小N-1范围，并且放到合适的位置，这个映射规则就叫散列函数/哈希(hash)函数

·理想情况下，两个不同的关键字映射到不同的单元，然而由于数组单元有限，关键字范围可能远超数组单元，因此就会出现两个关键字散列到同一个值得时候，这就是散列冲突

注：当存储或查找记录时，通过散列函数计算出记录的散列地址

# 实例演示

通过前面的描述，我们已经了解了一些基本概念，现在来看一个实例。

假设有一个大小为7的表，现在，要将13，18，19，50，20散列到表中。

·选择散列函数，例如使用hash(x)=x%7作为散列函数

·计算数据散列值，并放到合适的位置

计算13 % 7得到6，因此将13放到下标为6的位置：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  |  |  |  |  |  | 13 |

计算18 % 7得到4，因此将18放到下标为4的位置：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  |  |  |  | 18 |  | 13 |

计算19 % 7得到5，因此将19放到下标为5的位置：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  |  |  |  | 18 | 19 | 13 |

计算50 % 7得到1，因此将50放到下标为1的位置：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | 50 |  |  |  |  | 13 |

计算20 % 7得到6，因此将20放到下标为6的位置，但是此时6的位置已经被占用了，因此就产生了散列冲突。

将数据散列之后，如何从表中查找呢？例如，查找数值为50的数据位置，只需要计算50 % 7，得到下标1，访问下标1的位置即可。但是如果考虑散列冲突，就没有那么简单了。

通过这个实例，了解了以下几个概念：

·散列函数，散列函数的选择非常重要

·散列冲突，涉及散列表时，因尽量避免散列冲突，对于冲突也要有好的解决方案

·快速从散列表中查找数据

# 特点

## 优点

通过关键字实现一对一查找速度非常快。

## 缺点

存在冲突。

# 散列函数

构造散列函数的两个基本原则：计算简单+均匀分布

## 直接定址法

我们可以取关键字的某个线性函数值为散列地址，即f(key)=a\*key+b（通过关键字的加减乘除等运算计算地址）。

这种方法简单均匀，但是我们需要事先知道关键字的分布情况，适合查找表比较小且连续的情况，比如按照年月日进行计算。

## 数字分析法

数字分析法通常适合处理关键字位数比较大的情况，例如我们现在要存储某家公司员工登记表，如用手机号作为关键字，那么我们发现抽样后面的四位数字作为散列地址是不错的选择。

## 平方取中法

平方取中法是将关键字平方之后取中间若干位数字作为散列地址。

比较适合于不知道关键字的分布且位数不是很大的情况。

## 折叠法

折叠法是将关键字从左到右分割成位数相等的几部分，然后将这几部分叠加求和，并按散列表表长后几位作为散列地址。

适合于不知道关键字分布且位数较多的情况。

## 除留余数法

除留余数法为最常用的构造散列函数方法，对于散列表长为m的散列函数计算公式为：

f(key) = key mod p(p<=m)

注：p的选择是关键！

事实上，这个方法不仅可以对关键字直接取模，也可以通过折叠、平方取中后再取模。

## 随机数法

选择一个随机数，取关键字的随机函数值为它的散列地址。即：

f(key) = random(key)

这里的random是随机函数，当关键字的长度不等时，采用这个方法构造散列函数是比较合适的。

## 散列函数选择

在实际应用中，我们应该视不同的情况采用不同的散列函数，主要考虑：

1. 计算散列地址所需的时间；
2. 关键字的长度；
3. 散列表的大小；
4. 关键字的分布情况；
5. 记录查找的频率。

# 冲突解决

解决散列冲突通常有以下几种方法：

拉链法

开放定址法

再散列

…

## 拉链法

分离链接法/链地址法的做法是将同一个值的关键字保存在同一个表中。例如，对于前面：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | 50 |  |  | 18 | 19 | 13 |

如果再要插入元素20，则在下标为6的位置存储表头，而表的内容是13和20。

这种方法的特点是需要另外分配新的单元来存储散列到同一个位置的数据。

查找的时候，除了根据计算出来的散列值找到对应位置外，还需要在链表上进行搜索。而在单链表上的查找速度是很慢的。另外散列函数如果设计得好，冲突的概率其实也会很小。

## 开放定址法

开放定址法的思想是，如果冲突发生，就选择另外一个可用/空的位置。只要散列表足够大，空的/可用的散列地址总能找到，并将记录存入。

开放定址法中也有常见的几种策略。

### 线性探测法

公式：

fi(key) = (s(key)+di) MOD m (di=1,2,…,m-1)

还是以前面的为例：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|  | 50 |  |  | 18 | 19 | 13 |

如果此时再要插入20，则20 % 7 = 6，但是6的位置已有元素，因此探测下一个位置（6+1）%7，在这里就是下标为0的位置。因此20的存储位置如下：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 20 | 50 |  |  | 18 | 19 | 13 |

但这种方式的一个问题是，可能造成一次聚集，因为一旦冲突发生，为了处理冲突就会占用下一个位置，而如果冲突较多时，就会出现数据都聚集在一块区域。这样就会导致任何关键字都需要多次尝试才可能解决冲突。

### 平方探测法

顾名思义，如果说前面的探测函数是f(i)= i % 7，那么平方探测法就是f(i)= (i^2 )% 7。

可以修改di的取值方式，例如使用平方运算来尽量解决堆积问题：

fi(key) = (f(key)+di) MOD m (di=12,-12,22,-22,…q2,-q2,q<=m/2)

但是平方探测法同样会产生二次聚集问题。

### 双散列

为了避免聚集，在探测时选择跳跃式的探测，即再使用一个散列函数，用来计算探测的位置。

在冲突时，对于位移量di采用随机函数计算得到，我们称之为随机探测法，公式：

fi(key) = (f(key)+di) MOD m (di是由一个随机函数获得的数列)

假设前面的散列函数为hash1(X)，用于探测的散列函数为hash2(X)，那么一种流行的选择是f(i) = i \* hash2(X)，即第一次冲突时探测hash1(X)+hash2(X)的位置，第二次探测hash1(X)+2hash2(X)的位置。

可以看到，无论是哪种开放定址法，它都要求表足够大。

## 再散列函数法

散列表可以认为是具有固定大小的数组，那么如果插入新的数据时散列表已满，或者散列表所剩容量不多该怎么办？这个时候就需要再散列，常见做法是，建立一个是原来两倍大小的散列表，将原来表中的关键字重新散列到新表中。

公式：

fi(key) = RHi(key) (1,2,3,…k)

注：RH为各种散列函数。

## 公共溢出区法

# 实现

## 初始化

#define HASHSIZE 12

typedef struct

{

int \*elem; //数据元素的基址,动态分配数组

int count; //当前数据元素个数

}HashTable;

int InitHashTable(HashTable \*H)

{

H->count = HASHSIZE;

H->elem = (int \*)malloc(HASHSIZE \* sizeof()int);

if(!H->elem)

{

return -1;

}

for(i=0;i<HASHSIZE;i++)

{

H->elem[i] = NULLKEY;

}

return 0;

}

//使用除留余数法

int Hash(int key)

{

return key % HASHSIZE;

}

## 插入

//插入关键字到散列表

void InsertHash(HashTable \*H, int key)

{

int addr;

addr = Hash(key);

while(H->elem[addr]!=NULLKEY) //如果不为空,则冲突出现

{

addr = (addr+1)%HASHSIZE; //开放定址法的线性探测

}

H->elem[addr] = key;

}

## 查找

//散列表查找关键字

int SearchHash(HashTable H, int key, int \*addr)

{

\*addr = Hash(key);

while(H.elem[\*addr]!=key)

{

\*addr = (\*addr+1)%HASHSIZE;

if(H.elem[\*addr] == NULLKEY || \*addr == Hash(key))

{

return -1;

}

}

}

# 应用

散列表应用很广泛。例如做**文件校验或数字签名**。当然还有快速查询功能的实现。例如，redis中的字典结构就使用了散列表，使用MurmurHash算法来计算字符串的hash值，并采用拉链法处理冲突，，当散列表的装载因子（关键字个数与散列表大小的比）接近某个大小时，进行再散列。

下面主要介绍哈希表在数据库中的应用：

## 哈希查找

哈希查找，由于哈希算法能够很好的把数据做比较均匀的分布，所以哈希查找的速度要比B+ Tree查找的速度要快。哈希查找的时间复杂度是O(1)，而B+ Tree的时间复杂度是O(h)。当然，B+ Tree的优势是在范围查找，不然也不会成为关系型数据库的基本算法了。

SQL Server在做联接(JOIN)的时候，如果两个表都不是很小，而且没有在关联列上排序，就很可能使用哈希联接。哈希联接使用的就是哈希查找，它的性能并不差，但是要注意的一点是，哈希联接需要先构造一个哈希表，而哈希表需要消耗不小的内存空间，如果数据库服务器的内存不足的话，SQL Server就只好使用“优雅的哈希联接”(Grace HASH JOIN)或者递归哈希联接(Recursive Hash Join)，这样性能就会受到影响了。在设计数据库的时候，我们应该注意建立/更新适当的索引和统计信息(STATISTICS)，以便SQL Server可以准确的估计联接的输入大小，以便选择正确的算法。

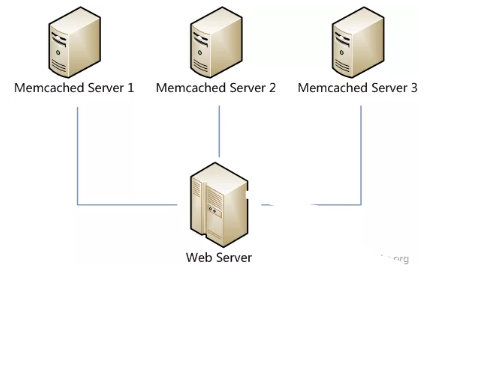
在使用哈希联接的时候，SQL Server的查询优化器会选择算法，通常我们并不需要做任何的指令，编写任何代码。但是在其它一些需要用到哈希算法的时候，就需要认真的选择算法，甚至可能需要自己写哈希算法的代码。

在业务系统中，可能生成的业务键（Business key）会比较长，例如某电商网站的订单号会类似这样：202210782169AC7G。这样的业务键如果用作主键的话，会占据16字节，显得有点浪费。如果只是单一系统使用，可以考虑用自动增量的数字作为主键。但是如果在多个系统中使用，比如在一个数据仓库系统中，使用自动增量作为代理键(Surrogate key)，就必须在处理事实表的ETL过程中，用新的自动增量替换掉订单表和详单表中的订单号，这会带来两个大表的JOIN，是一个相当耗时的操作。在这种情况下，我们就可以考虑使用哈希算法来生成代理键，只需要在订单表和详单表都使用同样的哈希算法，就可以保证得到的代理键是可以正确联接的。

## 一致性哈希

### 分布式缓存

假设我们有一个网站，最近发现随着流量增加，服务器压力越来越大，之前直接读写数据库的方式不太给力了，于是我们想引入Memcached作为缓存机制。现在我们一共有三台机器可以作为Memcached服务器，如下图所示：

很显然，最简单的策略是将每一次Memcached请求随机发送到一台Memcached服务器，但是这种策略可能会带来两个如下问题：

（1）一是同一份数据可能被存在不同的机器上而造成数据冗余；

（2）二是有可能某数据已经被缓存但是访问却没有命中，因为无法保证对相同key的所有访问都被发送到相同的服务器。因此，随机策略无论是时间效率还是空间效率都非常不好。

要解决上述问题只需做到如下一点：保证对相同key的访问会被发送到相同的服务器。很多方法可以实现这一点，最常用的方法是计算哈希。

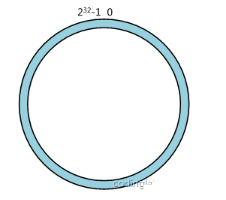
假设对于上图，对于每次访问，可以按如下算法计算其哈希值：h = Hash(key) % 3 其中Hash是一个从字符串到正整数的哈希映射函数。这样，如果我们将Memcached Server分别编号为0、1、2，那么就可以根据上式和key计算出服务器编号h，然后去访问。这个方法虽然解决了上面提到的两个问题，但是存在一些其它的问题。如果将上述方法抽象，可以认为通过：h = Hash(key) % N。这个算式计算每个key的请求应该被发送到哪台服务器，其中N为服务器的台数，并且服务器按照0–(N-1)编号。这个算法的问题在于容错性和扩展性不好。所谓容错性是指当系统中某一个或几个服务器变得不可用时，整个系统是否可以正确高效运行；而扩展性是指当加入新的服务器后，整个系统是否可以正确高效运行。

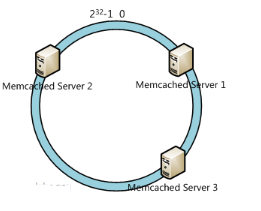
现假设有一台服务器宕机了，那么为了填补空缺，要将宕机的服务器从编号列表中移除，后面的服务器按顺序前移一位并将其编号值减一，此时每个key就要按h = Hash(key) % (N-1)重新计算。

同样，如果新增了一台服务器，虽然原有服务器编号不用改变，但是要按h = Hash(key) % (N+1)重新计算哈希值。因此系统中一旦有服务器变更，大量的key会被重定位到不同的服务器从而造成大量的缓存不命中。而这种情况在分布式系统中是非常糟糕的。

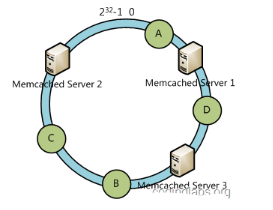
一个设计良好的分布式哈希方案应该具有良好的单调性，即服务节点的增减不会造成大量哈希重定位。一致性哈希算法就是这样一种哈希方案。

### 一致性哈希算法

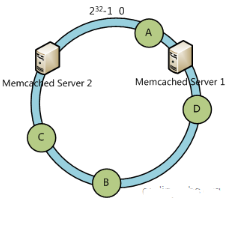
一致性哈希算法（Consistent Hashing）最早在论文《Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web》中被提出。简单来说，一致性哈希将整个哈希值空间组织成一个虚拟的圆环，如假设某哈希函数H的值空间为0-232-1（即哈希值是一个32位无符号整形），整个哈希空间环如下：

整个空间按顺时针方向组织。0和232-1在零点中方向重合。下一步将各个服务器使用H进行一个哈希，具体可以选择服务器的ip或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置，这里假设将上文中三台服务器使用ip地址哈希后在环空间的位置如下：

接下来使用如下算法定位数据访问到相应服务器：将数据key使用相同的函数H计算出哈希值h，通根据h确定此数据在环上的位置，从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器。 例如我们有A、B、C、D四个数据对象，经过哈希计算后，在环空间上的位置如下：

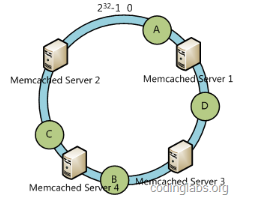
根据一致性哈希算法，数据A会被定为到Server 1上，D被定为到Server 3上，而B、C分别被定为到Server 2上。

### 容错性和扩展性分析

下面分析一致性哈希算法的容错性和可扩展性。现假设Server 3宕机了：

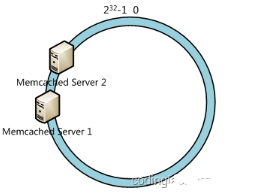
可以看到此时A、C、B不会受到影响，只有D节点被重定位到Server 2。一般的，在一致性哈希算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即顺着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

下面考虑另外一种情况，如果我们在系统中增加一台服务器Memcached Server 4：

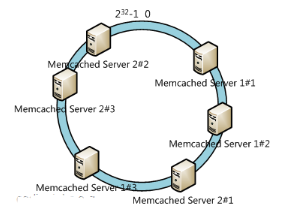
此时A、D、C不受影响，只有B需要重定位到新的Server 4。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即顺着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

综上所述，一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

### 虚拟节点

一致性哈希算法在服务节点太少时，容易因为节点分部不均匀而造成数据倾斜问题。 例如我们的系统中有两台服务器，其环分布如下：

此时必然造成大量数据集中到Server 1上，而只有极少量会定位到Server 2上。为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。

具体做法可以在服务器ip或主机名的后面增加编号来实现。例如上面的情况，我们决定为每台服务器计算三个虚拟节点，于是可以分别计算“Memcached Server 1#1”、“Memcached Server 1#2”、“Memcached Server 1#3”、“Memcached Server 2#1”、“Memcached Server 2#2”、“Memcached Server 2#3”的哈希值，于是形成六个虚拟节点：

同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Memcached Server 1#1”、“Memcached Server 1#2”、“Memcached Server 1#3”三个虚拟节点的数据均定位到Server 1上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为32甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

### 总结

目前一致性哈希基本成为了分布式系统组件的标准配置，例如Memcached的各种客户端都提供内置的一致性哈希支持。更深入的内容可以参看论文《Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web》。

# 总结

一个设计良好的散列表能够几乎在O(1)时间复杂度内完成插入，删除和查找，但前提是散列函数设计得足够优雅，以及有着合适散列冲突解决方案。常见冲突解决方案有：

拉链法

开放地址检测法

其中拉链法在实际中是很常见的一种解决方案。

参考：

《数据结构与算法分析》

《redis设计与实现》

https://en.wikipedia.org/wiki/Hash\_table

# 试题

## 最小包含字符窗口

题目要求：

给定一个字符串S（源字符串）和一个字符串T（目标字符串），使用O(n)的时间复杂度在S中找到一个最小的窗口大小，使得该窗口包含T中的所有字符。

代码：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

/\*

题目：一个字符串能包含另一个字符串中所有字幕的子串的最小长度

思路：使用两个指针，一个hash表，两个指针为了记录宽带，hash表为了记录是否存在目标子串在两个指针之间出现了多少次，直到所有字符都出现在这段子串内，移动前面的指针，直到某一个子串出现在子串中一次那么这个空间的长度就是当前最短的，这样遍历直到把所有字符串遍历结束

\*/

string MinLength(string& src,string& dest)

{

int i=0,j=0;

int flag =0;

int len = src.size();

int pos =0;

vector<int> hash(26,-1);

for(i=0;i<dest.length();i++)

hash[dest[i]-'A'] =0;

//开始处理

for(i=0;i<src.size();i++)

{

if(hash[src[i]-'A'] >= 0)

{

hash[src[i]-'A']++;

if(hash[src[i]-'A'] ==1)

flag++;

}

if(flag == dest.length())

{

for(;j<i;j++)

{

if(hash[src[j]-'A'] ==1)

break;

else

hash[src[j]-'A']--;

}

if(len>i-j+1)

{

len =i-j+1;

pos = j;

}

hash[src[j]-'A'] =0;

j++;

flag--;

}

}

cout<<string(src,pos,pos+len)<<" pos is "<<pos<<" len is "<<len<<endl;

return string(src,pos,pos+len);

}

int main()

{

string src("ADOBECODEBANCAAABC");

string dest("ABC");

cout<<MinLength(src,dest)<<endl;

return 0;

}

## 最长无重复字符字串

题目要求：给定一个字符串S，在该字符串中找到一个最长的没有重复字符的子串。

代码：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

/\*

题目：最长没有重复字符的子串

思路：假设所有的字符都是26个小写英文字符。以此建立hash表，hash表内记录着字符出现的位置，如果发现当前字符在以前遍历过的字符串中已经存在那么从上次出现这个字符的位置的下一个字符重新遍历那么把hash表清空，继续从当前位置开始遍历

\*/

int Longest\_substring(string& str)

{

vector<int> hash(26,-1);

int i;

int cur =0;

int maxlen =0;

int tmp;

for(i=0;i<str.size();i++)

{

if(hash[str[i]-'a'] ==-1)

{

hash[str[i]-'a'] =i;

++cur;

maxlen = maxlen > cur?maxlen:cur;

}

else

{

tmp = hash[str[i]-'a'];//此时记录的位置

i = tmp;

cur =0;

memset(&hash[0],-1,sizeof(int)\*hash.size());

}

}

return maxlen;

}

/\*

同样的思想 简洁的方法

\*/

int lengthOfLongeststring(string s)

{

vector<int> charIndex(256,-1);//这里扩展到了所有的字符

int longest =0;

int m=0;

for(int i=0;i<s.size();i++)

{

m = max(charIndex[s[i]]+1,m);

charIndex[s[i]] =i;

longest = max(longest,i-m+1);

}

return longest;

}

int main()

{

string str("abcabferafjlkcbb");

cout<<lengthOfLongeststring(str)<<endl;

return 0;

}

## 直线上最多点的个数

题目要求：有一个二维坐标点的集合，查找共线最多的点的个数。

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

共线最多的点的个数

思路：可以看所有两点能够构造的直线的斜率，根据斜率来判断共线的点的个数 这样斜率和该斜率上点的个数就构成了一个键值对，可以使用hash表来存储

\*/

int maxPoints(vector<pair<int,int> >& points )

{

if(points.size() == 0)

return 0;

int max =1;

double ratio=0.0;

//开始遍历点，构造键值对

for(int i=0;i<points.size()-1;i++)

{

hash\_map<double,int> map;

int numofSame=0;

int localMax=1;

//从当前点和其之后的点构成的斜率

for(int j=i+1;;j<points.size();j++)

{

if(points[j].first == points[j].first && points[j].second == points.second)

{

numofSame++;

continue;

}

else if(points[j].first == points[i].first)

{

ratio = numeric\_limits<double>::max();

}

else if(points[j].second == points[i].second)

{

ratio =0.0;

}

else

{

ratio = (double)(points[j].second-

points[i].second)/(double)(points[j].first-points[i].first);

}

int num;

if(map.find(ratio) != map.end())

map[ratio]++;

else

map[ratio]=2;

}

//开始查找某一斜率下最多的点的个数

hash\_map<double,int>::iterator it = map.begin();

fo(;it!=map.end();it++)

localMax = max(localMax,it->second);

localMax += numofSame;

max = max(max,localMax);

}

return max;

}