# 4、数组&排序

**程序员面试题精选**(03)－求子数组的最大和

题目：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：本题最初为2005年浙江大学计算机系的考研题的最后一道程序设计题，在2006年里包括google在内的很多知名公司都把本题当作面试题。由于本题在网络中广为流传，本题也顺利成为2006年程序员面试题中经典中的经典。

如果不考虑时间复杂度，我们可以枚举出所有子数组并求出他们的和。不过非常遗憾的是，由于长度为n的数组有O(n2)个子数组；而且求一个长度为n的数组的和的时间复杂度为O(n)。因此这种思路的时间是O(n3)。

很容易理解，当我们加上一个正数时，和会增加；当我们加上一个负数时，和会减少。如果当前得到的和是个负数，那么这个和在接下来的累加中应该抛弃并重新清零，不然的话这个负数将会减少接下来的和。基于这样的思路，我们可以写出如下代码。

参考代码：

/////////////////////////////////////////////////////////////////////////////

// Find the greatest sum of all sub-arrays

// Return value: if the input is valid, return true, otherwise return false

/////////////////////////////////////////////////////////////////////////////

bool FindGreatestSumOfSubArray

(

int \*pData, // an array

unsigned int nLength, // the length of array

int &nGreatestSum // the greatest sum of all sub-arrays

)

{

// if the input is invalid, return false

if((pData == NULL) || (nLength == 0))

return false;

int nCurSum = nGreatestSum = 0;

for(unsigned int i = 0; i < nLength; ++i)

{

nCurSum += pData;

// if the current sum is negative, discard it

if(nCurSum < 0)

nCurSum = 0;

// if a greater sum is found, update the greatest sum

if(nCurSum > nGreatestSum)

nGreatestSum = nCurSum;

}

// if all data are negative, find the greatest element in the array

if(nGreatestSum == 0)

{

nGreatestSum = pData[0];

for(unsigned int i = 1; i < nLength; ++i)

{

if(pData > nGreatestSum)

nGreatestSum = pData;

}

}

return true;

}

讨论：上述代码中有两点值得和大家讨论一下：

??????????函数的返回值不是子数组和的最大值，而是一个判断输入是否有效的标志。如果函数返回值的是子数组和的最大值，那么当输入一个空指针是应该返回什么呢？返回0？那这个函数的用户怎么区分输入无效和子数组和的最大值刚好是0这两中情况呢？基于这个考虑，本人认为把子数组和的最大值以引用的方式放到参数列表中，同时让函数返回一个函数是否正常执行的标志。

??????????输入有一类特殊情况需要特殊处理。当输入数组中所有整数都是负数时，子数组和的最大值就是数组中的最大元素。

**程序员面试题精选**(05)－查找最小的k个元素

题目：输入n个整数，输出其中最小的k个。

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

分析：这道题最简单的思路莫过于把输入的n个整数排序，这样排在最前面的k个数就是最小的k个数。只是这种思路的时间复杂度为O(nlogn)。我们试着寻找更快的解决思路。

我们可以开辟一个长度为k的数组。每次从输入的n个整数中读入一个数。如果数组中已经插入的元素少于k个，则将读入的整数直接放到数组中。否则长度为k的数组已经满了，不能再往数组里插入元素，只能替换了。如果读入的这个整数比数组中已有k个整数的最大值要小，则用读入的这个整数替换这个最大值；如果读入的整数比数组中已有k个整数的最大值还要大，则读入的这个整数不可能是最小的k个整数之一，抛弃这个整数。这种思路相当于只要排序k个整数，因此时间复杂可以降到O(n+nlogk)。通常情况下k要远小于n，所以这种办法要优于前面的思路。

这是我能够想出来的最快的解决方案。不过从给面试官留下更好印象的角度出发，我们可以进一步把代码写得更漂亮一些。从上面的分析，当长度为k的数组已经满了之后，如果需要替换，每次替换的都是数组中的最大值。在常用的数据结构中，能够在O(1)时间里得到最大值的数据结构为最大堆。因此我们可以用堆（heap）来代替数组。

另外，自己重头开始写一个最大堆需要一定量的代码。我们现在不需要重新去发明车轮，因为前人早就发明出来了。同样，STL中的set和multiset为我们做了很好的堆的实现，我们可以拿过来用。既偷了懒，又给面试官留下熟悉STL的好印象，何乐而不为之？

参考代码：

#include <set>

#include <vector>

#include <iostream>

using namespace std;

typedef multiset<int, greater<int>> IntHeap;

///////////////////////////////////////////////////////////////////////

// find k least numbers in a vector

///////////////////////////////////////////////////////////////////////

void FindKLeastNumbers

(

const vector<int>& data, // a vector of data

IntHeap& leastNumbers, // k least numbers, output

unsigned int k

)

{

leastNumbers.clear();

if(k == 0 || data.size() < k)

return;

vector<int>::const\_iterator iter = data.begin();

for(; iter != data.end(); ++ iter)

{

// if less than k numbers was inserted into leastNumbers

if((leastNumbers.size()) < k)

leastNumbers.insert(\*iter);

// leastNumbers contains k numbers and it's full now

else

{

// first number in leastNumbers is the greatest one

IntHeap::iterator iterFirst = leastNumbers.begin();

// if is less than the previous greatest number

if(\*iter < \*(leastNumbers.begin()))

{

// replace the previous greatest number

leastNumbers.erase(iterFirst);

leastNumbers.insert(\*iter);

}

}

}

}

**程序员面试题精选**(10)－在排序数组中查找和为给定值的两个数字

题目：输入一个已经按升序排序过的数组和一个数字，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：如果我们不考虑时间复杂度，最简单想法的莫过去先在数组中固定一个数字，再依次判断数组中剩下的n-1个数字与它的和是不是等于输入的数字。可惜这种思路需要的时间复杂度是O(n2)。

我们假设现在随便在数组中找到两个数。如果它们的和等于输入的数字，那太好了，我们找到了要找的两个数字；如果小于输入的数字呢？我们希望两个数字的和再大一点。由于数组已经排好序了，我们是不是可以把较小的数字的往后面移动一个数字？因为排在后面的数字要大一些，那么两个数字的和也要大一些，就有可能等于输入的数字了；同样，当两个数字的和大于输入的数字的时候，我们把较大的数字往前移动，因为排在数组前面的数字要小一些，它们的和就有可能等于输入的数字了。

我们把前面的思路整理一下：最初我们找到数组的第一个数字和最后一个数字。当两个数字的和大于输入的数字时，把较大的数字往前移动；当两个数字的和小于数字时，把较小的数字往后移动；当相等时，打完收工。这样扫描的顺序是从数组的两端向数组的中间扫描。

问题是这样的思路是不是正确的呢？这需要严格的数学证明。感兴趣的读者可以自行证明一下。

参考代码：

///////////////////////////////////////////////////////////////////////

// Find two numbers with a sum in a sorted array

// Output: ture is found such two numbers, otherwise false

///////////////////////////////////////////////////////////////////////

bool FindTwoNumbersWithSum

(

int data[], // a sorted array

unsigned int length, // the length of the sorted array

int sum, // the sum

int& num1, // the first number, output

int& num2 // the second number, output

)

{

bool found = false;

if(length < 1)

return found;

int ahead = length - 1;

int behind = 0;

while(ahead > behind)

{

long long curSum = data[ahead] + data[behind];

// if the sum of two numbers is equal to the input

// we have found them

if(curSum == sum)

{

num1 = data[behind];

num2 = data[ahead];

found = true;

break;

}

// if the sum of two numbers is greater than the input

// decrease the greater number

else if(curSum > sum)

ahead --;

// if the sum of two numbers is less than the input

// increase the less number

else

behind ++;

}

return found;

}

扩展：如果输入的数组是没有排序的，但知道里面数字的范围，其他条件不变，如和在O(n)时间里找到这两个数字？

**程序员面试题精选**100题(29)-调整数组顺序使奇数位于偶数前面

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

分析：如果不考虑时间复杂度，最简单的思路应该是从头扫描这个数组，每碰到一个偶数时，拿出这个数字，并把位于这个数字后面的所有数字往前挪动一位。挪完之后在数组的末尾有一个空位，这时把该偶数放入这个空位。由于碰到一个偶数，需要移动O(n)个数字，因此总的时间复杂度是O(n２)。

要求的是把奇数放在数组的前半部分，偶数放在数组的后半部分，因此所有的奇数应该位于偶数的前面。也就是说我们在扫描这个数组的时候，如果发现有偶数出现在奇数的前面，我们可以交换他们的顺序，交换之后就符合要求了。

因此我们可以维护两个指针，第一个指针初始化为数组的第一个数字，它只向后移动；第二个指针初始化为数组的最后一个数字，它只向前移动。在两个指针相遇之前，第一个指针总是位于第二个指针的前面。如果第一个指针指向的数字是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。

基于这个思路，我们可以写出如下的代码：

void Reorder(int \*pData, unsigned int length, bool (\*func)(int));

bool isEven(int n);

/////////////////////////////////////////////////////////////////////////

// Devide an array of integers into two parts, odd in the first part,

// and even in the second part

// Input: pData - an array of integers

// length - the length of array

/////////////////////////////////////////////////////////////////////////

void ReorderOddEven(int \*pData, unsigned int length)

{

if(pData == NULL || length == 0)

return;

Reorder(pData, length, isEven);

}

/////////////////////////////////////////////////////////////////////////

// Devide an array of integers into two parts, the intergers which

// satisfy func in the first part, otherwise in the second part

// Input: pData - an array of integers

// length - the length of array

// func - a function

/////////////////////////////////////////////////////////////////////////

void Reorder(int \*pData, unsigned int length, bool (\*func)(int))

{

if(pData == NULL || length == 0)

return;

int \*pBegin = pData;

int \*pEnd = pData + length - 1;

while(pBegin < pEnd)

{

// if \*pBegin does not satisfy func, move forward

if(!func(\*pBegin))

{

pBegin ++;

continue;

}

// if \*pEnd does not satisfy func, move backward

if(func(\*pEnd))

{

pEnd --;

continue;

}

// if \*pBegin satisfy func while \*pEnd does not,

// swap these integers

int temp = \*pBegin;

\*pBegin = \*pEnd;

\*pEnd = temp;

}

}

/////////////////////////////////////////////////////////////////////////

// Determine whether an integer is even or not

// Input: an integer

// otherwise return false

/////////////////////////////////////////////////////////////////////////

bool isEven(int n)

{

return (n & 1) == 0;

}

讨论：

上面的代码有三点值得提出来和大家讨论：

１．函数isEven判断一个数字是不是偶数并没有用%运算符而是用&。理由是通常情况下位运算符比%要快一些；

２．这道题有很多变种。这里要求是把奇数放在偶数的前面，如果把要求改成：把负数放在非负数的前面等，思路都是都一样的。

３．在函数Reorder中，用函数指针func指向的函数来判断一个数字是不是符合给定的条件，而不是用在代码直接判断（hard code）。这样的好处是把调整顺序的算法和调整的标准分开了（即解耦，decouple）。当调整的标准改变时，Reorder的代码不需要修改，只需要提供一个新的确定调整标准的函数即可，提高了代码的可维护性。例如要求把负数放在非负数的前面，我们不需要修改Reorder的代码，只需添加一个函数来判断整数是不是非负数。这样的思路在很多库中都有广泛的应用，比如在ＳＴＬ的很多算法函数中都有一个仿函数（functor）的参数（当然仿函数不是函数指针，但其思想是一样的）。如果在面试中能够想到这一层，无疑能给面试官留下很好的印象。

**程序员面试题精选**（36）：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？

将1001个元素相加减去1,2,3,……1000数列的和，得到的差即为重复的元素。

int Find(int \* a)

{

int i;//变量

for (i = 0 ;i<=1000;i++)

{

a[1000] += a[i];

}

a[1000] -= (i\*(i-1))/2 //i的值为1001

return a[1000];

}

利用下标与单元中所存储的内容之间的特殊关系，进行遍历访问单元，一旦访问过的单

元赋予一个标记，利用标记作为发现重复数字的关键。代码如下：

void FindRepeat(int array[], int length)

{

int index=array[length-1]-1;

while ( true )

{

if ( array[index]<0 )

break;

array[index]\*=-1;

index=array[index]\*(-1)-1;

}

cout<<"The repeat number is "<<index+1<<endl;

}

此种方法不非常的不错，而且它具有可扩展性。在坛子上有人提出：

对于一个既定的自然数 N ，有一个 N + M 个元素的数组，其中存放了小于等于 N 的所有

自然数，求重复出现的自然数序列{X} 。

对于这个扩展需要，自己在A\_B\_C\_ABC(黄瓜儿才起蒂蒂)的算法的基础上得到了自己的算法

代码：

按照A\_B\_C\_ABC(黄瓜儿才起蒂蒂)的算法，易经标记过的单元在后面一定不会再访问到，除非它是重复的数字，也就是说只要每次将重复数字中的一个改为靠近N+M的自然数，让遍历能访问到数组后面的单元，就能将整个数组遍历完。

代码：

\*/

void FindRepeat(int array[], int length, int num)

{

int index=array[length-1]-1;

cout<<"The repeat number is ";

while ( true )

{

if ( array[index]<0 )

{

num--;

array[index]=length-num;

cout<<index+1<<'t';

}

if ( num==0 )

{

cout<<endl;

return;

}

array[index]\*=-1;

index=array[index]\*(-1)-1;

}

}

**程序员面试题精选**（54）：找出数组中两个只出现一次的数字收藏

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

分析：这是一道很新颖的关于位运算的面试题。

首先我们考虑这个问题的一个简单版本：一个数组里除了一个数字之外，其他的数字都出现了两次。请写程序找出这个只出现一次的数字。

这个题目的突破口在哪里？题目为什么要强调有一个数字出现一次，其他的出现两次？我们想到了异或运算的性质：任何一个数字异或它自己都等于0。也就是说，如果我们从头到尾依次异或数组中的每一个数字，那么最终的结果刚好是那个只出现依次的数字，因为那些出现两次的数字全部在异或中抵消掉了。

有了上面简单问题的解决方案之后，我们回到原始的问题。如果能够把原数组分为两个子数组。在每个子数组中，包含一个只出现一次的数字，而其他数字都出现两次。如果能够这样拆分原数组，按照前面的办法就是分别求出这两个只出现一次的数字了。

我们还是从头到尾依次异或数组中的每一个数字，那么最终得到的结果就是两个只出现一次的数字的异或结果。因为其他数字都出现了两次，在异或中全部抵消掉了。由于这两个数字肯定不一样，那么这个异或结果肯定不为0，也就是说在这个结果数字的二进制表示中至少就有一位为1。我们在结果数字中找到第一个为1的位的位置，记为第N位。现在我们以第N位是不是1为标准把原数组中的数字分成两个子数组，第一个子数组中每个数字的第N位都为1，而第二个子数组的每个数字的第N位都为0。

现在我们已经把原数组分成了两个子数组，每个子数组都包含一个只出现一次的数字，而其他数字都出现了两次。因此到此为止，所有的问题我们都已经解决。

代码如下：

#include <iostream>

using namespace std;

///////////////////////////////////////////////////////////////////////

// Find the index of first bit which is 1 in num (assuming not 0)

///////////////////////////////////////////////////////////////////////

unsigned int FindFirstBitIs1(int num)

{

int indexBit = 0;

while (((num & 1) == 0) && (indexBit < 32))

{

num = num >> 1;

++ indexBit;

}

return indexBit;

}

///////////////////////////////////////////////////////////////////////

// Is the indexBit bit of num 1?

///////////////////////////////////////////////////////////////////////

bool IsBit1(int num, unsigned int indexBit)

{

num = num >> indexBit;

return (num & 1);

}

///////////////////////////////////////////////////////////////////////

// Find two numbers which only appear once in an array

// Input: data - an array contains two number appearing exactly once,

// while others appearing exactly twice

// length - the length of data

// Output: num1 - the first number appearing once in data

// num2 - the second number appearing once in data

///////////////////////////////////////////////////////////////////////

void FindNumsAppearOnce(int data[], int length, int &num1, int &num2)

{

if (length < 2)

return;

// get num1 ^ num2

int resultExclusiveOR = 0;

for (int i = 0; i < length; ++ i)

resultExclusiveOR ^= data[i];

// get index of the first bit, which is 1 in resultExclusiveOR

unsigned int indexOf1 = FindFirstBitIs1(resultExclusiveOR);

num1 = num2 = 0;

for (int j = 0; j < length; ++ j)

{

// divide the numbers in data into two groups,

// the indexOf1 bit of numbers in the first group is 1,

// while in the second group is 0

if(IsBit1(data[j], indexOf1))

num1 ^= data[j];

else

num2 ^= data[j];

}

}

int main()

{

int a[8]={2,3,6,8,3,2,7,7};

int x,y;

FindNumsAppearOnce(a,8,x,y);

cout<<x<<"\t"<<y;

getchar();

}

**程序员面试题精选 中位数**腾讯的一道面试题:(与百度相似,可惜昨天百度死在这方面了)////

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。

答案:

1， 把整数分成256M段，每段可以用64位整数保存该段数据个数，256M\*8 = 2G内存，先清0

2，读10G整数，把整数映射到256M段中，增加相应段的记数

3，扫描256M段的记数，找到中位数的段和中位数的段前面所有段的记数，可以把其他段的内存释放

4，因中位数段的可能整数取值已经比较小（如果是32bit整数，当然如果是64bit整数的话，可以再次分段），对每个整数做一个记数，再读一次10G整数，只读取中位数段对应的整数，并设置记数。

5，对新的记数扫描一次，即可找到中位数。

如果是32bit整数，读10G整数2次，扫描256M记数一次，后一次记数因数量很小，可以忽略不记

(设是32bit整数，按无符号整数处理

整数分成256M段？ 整数范围是0 - 2^32 - 1 一共有4G种取值，4G/256M = 16，每16个数算一段 0-15是1段，16-31是一段，...

整数映射到256M段中？ 如果整数是0-15，则增加第一段记数，如果整数是16-31，则增加第二段记数，...

其实可以不用分256M段，可以分的段数少一写，这样在扫描记数段时会快一些，还能节省一些内存)

腾讯题二:

**程序员面试题精选 整数查询？**一个文件中有40亿个整数，每个整数为四个字节，内存为1GB，写出一个算法：求出这个文件里的整数里不包含的一个整数

答:

方法一: 4个字节表示的整数，总共只有2^32约等于4G个可能。

为了简单起见，可以假设都是无符号整数。

分配500MB内存，每一bit代表一个整数，刚好可以表示完4个字节的整数，初始值为0。基本思想每读入一个数，就把它对应的bit位置为１，处理完40G个数后，对500M的内存遍历，找出一个bit为0的位，输出对应的整数就是未出现的。

算法流程：

１）分配５００ＭＢ内存buf，初始化为０

２）unsigned int x=0x1;

for each int j in file

buf=buf |x <<j;

end

(3) for(unsigned int i=0; i <= 0xffffffff; i++)

if (!(buf & x <<i))

{

output(i);

break;

}

以上只是针对无符号的，有符号的整数可以依此类推。

方法二:

文件可以分段读啊，这个是O（2n）算法，应该是很快的了，而且空间也允许的。

不过还可以构造更快的方法的，更快的方法主要是针对定位输出的整数优化算法。

思路大概是这样的，把值空间等分成若干个值段，比如值为无符号数，则

00000000H-00000FFFH

00001000H-00001FFFH

......

0000F000H-0000FFFFH

.....

FFFFF000H-FFFFFFFFH

这样可以订立一个规则，在一个值段范围内的数第一次出现时，对应值段指示值Xn=Xn+1，如果该值段的所有整数都出现过，则Xn=1000H，这样后面输出定位时就可以直接跳过这个值段了，因为题目仅仅要求输出一个，这样可以大大减少后面对标志数值的遍历步骤。

理论上值段的划分有一定的算法可以快速的实现，比如利用位运算直接定位值段对应值进行计算。

腾讯面试题:

有1到10w这10w个数，去除2个并打乱次序，如何找出那两个数。（不准用位图！！）

位图解决:

位图的方法如下

假设待处理数组为A[10w-2]

定义一个数组B[10w],这里假设B中每个元素占用1比特,并初始化为全0

for(i=0;i <10w-2;i++)

{

B[ A[i] ]=1

}

那么B中不为零的元素即为缺少的数据

这种方法的效率非常高，是计算机中最常用的算法之一

其它方法:

求和以及平方和可以得到结果，不过可能求平方和运算量比较大（用64位int不会溢出）

腾讯面试题:

腾讯服务器每秒有2w个QQ号同时上线，找出5min内重新登入的qq号并打印出来。

解答: 第二题如果空间足够大,可以定义一个大的数组

a[qq号],初始为零,然后这个qq号登陆了就a[qq号]++

最后统计大于等于2的QQ号

这个用空间来代替时间

第二个题目，有不成熟的想法。

2w x 300s

所以用 6,000,000 个桶。删除超时的算法后面说，所以平均桶的大小是 1 。

假设 qq 号码一共有 10^10 个，所以每个桶装的 q 号码是 10^10 / (6 \* 10^6) 个，这个是插入时候的最坏效率（插入同一个桶的时候是顺序查找插入位置的）。

qq的节点结构和上面大家讨论的基本一样，增加一个指针指向输出列表，后面说。

struct QQstruct {

num\_type qqnum;

timestamp last\_logon\_time;

QQstruct \*pre;

QQstruct \*next;

OutPutList \*out; // 用于 free 节点的时候，顺便更新一下输出列表。

}

另外增加两个指针列表。

第一个大小 300 的循环链表，自带一个指向 QQStruct 的域，循环存 300 秒内的qq指针。时间一过

就 free 掉， 所以保证所有桶占用的空间在 2w X 300 以内。

第二个是 输出列表， 就是存放题目需要输出的节点。

如果登陆的用户，5分钟内完全没有重复的话，每秒 free 掉 2w 个节点。

不过在 free 的时候，要判断一下时间是不是真的超时，因为把节点入桶的时候，遇到重复的，会更

新一下最后登陆的时间。当然啦，这个时候，要把这个 qq 号码放到需要输出的列表里面。

求大数据量数组中不重复元素的个数收藏

新一篇: 几道面试题的求解 | 旧一篇: 求n!末尾0的个数

有2.5亿个整数(这2.5亿个整数存储在一个数组里面，至于数组是放在外存还是内存，没有进一步具体说明)；

要求找出这2.5亿个数字里面，不重复的数字的个数（那些只出现一次的数字的数目）；

另外，可用的内存限定为600M；

要求算法尽量高效，最优；

1. caoxic的算法

BYTE marks[2^29];//512M // BYTE marks[2^32/8]; //用这个就更清楚了,标志所有整数(2^32)出现的可能

BYTE repmarks[2^25];//32M 32M\*8>2.5亿 //标志2.5亿个数字数组里面的数字是否重复过

const BYTE bitmarks[8]={1,2,4,8,16,32,64,128};

DWORD CalcDifNum(DWORD \*pBuf,DWORD bufcount)

{

DWORD dw ;

DWORD count = 0 ;// 不重复的数字（包括出现多次的数字，只算一个）的个数,例子：1 2 2 3 5 3 4 算5个

DWORD count2 = 0 ;//重复出现的数字的个数，例子：1 2 2 3 5 3 4 算2个

memset(marks,0,sizeof(marks));

memset(repmarks,0,sizeof(repmarks));

ASSERT(sizeof(repmarks)\*8>=bufcount);//断言repmarks数组够用

for(dw=0;dw<bufcount;dw++)

{

if(marks[pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])

{

repmarks[dw>>3] |= bitmarks[dw&7];//标志pBuf[dw]这个数字出现重复

}

else

{

count ++;

marks[pBuf[dw]>>3] |= bitmarks[pBuf[dw]&7];//标志pBuf[dw]这个数字出现

}

}

memset(marks,0,sizeof(marks));

for(dw=0;dw<bufcount;dw++)

{

if(repmarks[dw>>3] & bitmarks[dw&7])//

{

if(marks[pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])

{

}

else

{

count2 ++; //重复的数字的数量

marks[pBuf[dw]>>3] |= bitmarks[pBuf[dw]&7];

}

}

}

return count-count2;

}

2. 改一下，应该也可以：

BYTE marks[2^29];//512M // BYTE marks[2^32/8]; //用这个就更清楚了,标志所有整数(2^32)出现的可能

BYTE repmarks[2^25];//32M 32M\*8>2.5亿 //标志2.5亿个数字数组里面的数字是否重复过

const BYTE bitmarks[8]={1,2,4,8,16,32,64,128};

DWORD CalcDifNum(DWORD \*pBuf,DWORD bufcount)

{

DWORD dw ;

DWORD count = 0 ;// 不重复的数字（包括出现多次的数字，只算一个）的个数,例子：1 2 2 3 5 3 4 算5个

DWORD count2 = 0 ;//只出现一次数字的个数，例子：1 2 2 3 5 3 4 算3个

memset(marks,0,sizeof(marks));

memset(repmarks,0,sizeof(repmarks));

ASSERT(sizeof(repmarks)\*8>=bufcount);//断言repmarks数组够用

for(dw=0;dw<bufcount;dw++)

{

if(marks[pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])

{

repmarks[dw>>3] |= bitmarks[dw&7];//标志pBuf[dw]这个数字出现重复

}

else

{

count ++;

marks[pBuf[dw]>>3] |= bitmarks[pBuf[dw]&7];//标志pBuf[dw]这个数字出现

}

}

for(dw=0;dw<bufcount;dw++)

{

if(!(repmarks[dw>>3] & bitmarks[dw&7]))//非重复的位置

{

count2 ++; //只出现一次的数字的数量

}

}

return count2;

}

3. 把数组里面的数字分两类，正数负数，marks数组分成两部分标志,1.该数已经标志[0 --> 2^28-1] 2.标识该数已经重复[2^28 -->2^29-1]。

BYTE marks[2^29];//512M // BYTE marks[2^32/8]; //用这个就更清楚了,标志所有整数(2^32)出现的可能

const BYTE bitmarks[8]={1,2,4,8,16,32,64,128};

DWORD CalcDifNum(DWORD \*pBuf,DWORD bufcount)

{

DWORD dw ;

DWORD count = 0 ;// 不重复的数字（包括出现多次的数字，只算一个）的个数,例子：1 2 2 3 5 3 4 算5个

DWORD count2 = 0 ;//重复出现的数字的个数，例子：1 2 2 3 5 3 4 算2个

memset(marks,0,sizeof(marks));

for(dw=0;dw<bufcount;dw++)

{

if(pBuf[dw]>=0)

{

if(marks[pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])

{

if(marks[2^28+pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])

{

}else

{

marks[2^28+pBuf[dw]>>3] |= bitmarks[dw&7];//标志pBuf[dw]这个数字出现重复

count2 ++;

}

}

else

{

count ++;

marks[pBuf[dw]>>3] |= bitmarks[pBuf[dw]&7];//标志pBuf[dw]这个数字出现

}

}

}

memset(marks,0,sizeof(marks));

for(dw=0;dw<bufcount;dw++)

{

if(pBuf[dw]<0)

{

pBuf[dw] = -pBuf[dw];

if(marks[pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])

{

if(marks[2^28+pBuf[dw]>>3]&bitmarks[pBuf[dw]&7])

{

}else

{

marks[2^28+pBuf[dw]>>3] |= bitmarks[dw&7];//标志pBuf[dw]这个数字出现重复

count2 ++;

}

}

else

{

count ++;

marks[pBuf[dw]>>3] |= bitmarks[pBuf[dw]&7];//标志pBuf[dw]这个数字出现

}

}

}

return count-count2;

}

**一道经典的面试题：如何从N个数中选出最大（小）的n个数？**

北京交大LuoBin

这 个问题我前前后后考虑了有快一年了，也和不少人讨论过。据我得到的消息，Google和微软都面过这道题。这道题可能很多人都听说过，或者知道答案（所谓 的“堆”），不过我想把我的答案写出来。我的分析也许存有漏洞，以交流为目的。但这是一个满复杂的问题，蛮有趣的。看完本文，也许会启发你一些没有想过的 解决方案（我一直认为堆也许不是最高效的算法）。在本文中，将会一直以寻找n个最“大”的数为分析例子，以便统一。注：本文写得会比较细节一些，以便于绝 大多数人都能看懂，别嫌我罗嗦:) 我很不确定多少人有耐心看完本文！

Naive 方法：

首先，我们假设n和N都是内存可容纳的，也就是说N个数可以一次load到内存里存放在数组里（如果非要存在链表估计又是另一个challenging的 问题了）。从最简单的情况开始，如果n=1，那么没有任何疑惑，必须要进行N-1次的比较才能得到最大的那个数，直接遍历N个数就可以了。如果n=2呢？ 当然，可以直接遍历2遍N数组，第一遍得到最大数max1，但是在遍历第二遍求第二大数max2的时候，每次都要判断从N所取的元素的下标不等于max1 的下标，这样会大大增加比较次数。对此有一个解决办法，可以以max1为分割点将N数组分成前后两部分，然后分别遍历这两部分得到两个“最大数”，然后二 者取一得到max2。

也可以遍历一遍就解决此问题，首先维护两个元素max1，max2（max1>=max2），取到N中的一个数以后，先和max1比，如果比 max1大（则肯定比max2大），直接替换max1，否则再和max2比较确定是否替换max2。采用类似的方法，对于n=2，3，4……一样可以处 理。这样的算法时间复杂度为O（nN）。当n越来越大的时候（不可能超过N/2，否则可以变成是找N-n个最小的数的对偶问题），这个算法的效率会越来越 差。但是在n比较小的时候（具体多小不好说），这个算法由于简单，不存在递归调用等系统损耗，实际效率应该很不错.

堆：

当n较大的时候采用什么算法呢？首先我们分析上面的算法，当从N中取出一个新的数m的时候，它需要依次和max1，max2，max3……max n比较，一直找到一个比m小的max x，就用m来替换max x，平均比较次数是n/2。可不可以用更少的比较次数来实现替换呢？最直观的方法是，也就是网上文章比较推崇的堆。堆有这么一些好处：1.它是一个完全二 叉树，树的深度是相同节点的二叉树中最少的，维护效率较高；2.它可以通过数组来实现，而且父节点p与左右子节l，r点的数组下标的关系是s[l] = 2\*s[p]+1和s[r] = 2\*s[p]+2。在计算机中2\*s[p]这样的运算可以用一个左移1位操作来实现，十分高效。再加上数组可以随机存取，效率也很高。3.堆的 Extract操作，也就是将堆顶拿走并重新维护堆的时间复杂度是O（logn），这里n是堆的大小。

具体到我们的问题，如何具体实现呢？首先开辟一个大小为n的数组区A，从N中读入n个数填入到A中，然后将A维护成一个小顶堆（即堆顶A[0]中存放的是 A中最小的数）。然后从N中取出下一个数，即第n+1个数m，将m与堆顶A[0]比较，如果m<=A[0]，直接丢弃m。否则应该用m替换 A[0]。但此时A的堆特性可能已被破坏，应该重新维护堆：从A[0]开始，将A[0]与左右子节点分别比较（特别注意，这里需要比较“两次”才能确定最 大数，在后面我会根据这个来和“败者树”比较），如果A[0]比左右子节点都小，则堆特性能够保证，勿需继续，否则如左（右）节点最大，则将A[0]与左 （右）节点交换，并继续维护左（右）子树。依次执行，直到遍历完N，堆中保留的n个数就是N中最大的n个数。这都是堆排序的基本知识， 唯一的trick就是维护一个小顶堆，而不是大顶堆。不明白的稍微想一下。维护一次堆的时间复杂度为O（logn），总体的复杂度是O（Nlogn）这样 一来，比起上面的O（nN），当n足够大时，堆的效率肯定是要高一些的。当然，直接对N数组建堆，然后提取n次堆顶就能得到结果，而且其复杂度是 O（nlogN）,当n不是特别小的时候这样会快很多。但是对于online数据就没办法了，比如N不能一次load进内存，甚至是一个流，根本不知道N 是多少。

败者树：

有没有别的算法呢？我先来说一说败者树（loser tree）。也许有些人对loser tree不是很了解，其实它是一个比较经典的外部排序方法，也就是有x个已经排序好的文件，将其归并为一个有序序列。败者树的思想咋一看有些绕，其实是为 了减小比较次数。首先简单介绍一下败者树：败者树的叶子节点是数据节点，然后两两分组（如果节点总数不是2的幂，可以用类似完全树的结构构成树），内部节 点用来记录左右子树的优胜者中的“败者”（注意记录的是输的那一方），而优胜者则往上传递继续比较，一直到根节点。如果我们的优胜者是两个数中较小的数， 则根节点记录的是最后一次比较中的“败者”，也就是所有叶子节点中第二小的那个数，而最小的那个数记录在一个独立的变量中。这里要注意，内部节点不但要记 录败者的数值，还要记录对应的叶子节点。如果是用链表构成的树，则内部节点需要有指针指向叶子节点。这里可以有一个trick，就是内部节点只记录“败者 ”对应的叶子节点，具体的数值可以在需要的时候间接访问（这一方法在用数组来实现败者树时十分有用，后面我会讲到）。关键的来了，当把最小值输出后，最小 值所对应的叶子节点需要变成一个新的数（或者改为无穷大，在文件归并的时候表示文件已读完）。接下来维护败者树，从更新的叶子节点网上，依次与内部节点比 较，将“败者”更新，胜者往上继续比较。由于更新节点占用的是之前的最小值的叶子节点，它往上一直到根节点的路径与之前的最小值的路径是完全相同的。内部 节点记录的“败者”虽然称为“败者”，但却是其所在子树中最小的数。也就是说，只要与“败者”比较得到的胜者，就是该子树中最小的那个数（这里讲得有点绕 了，看不明白的还是找本书看吧，对照着图比较容易理解）。

注：也可以直接对N构建败者树，但是败者树用数组实现时不能像堆一样进行增量维护，当叶子节点的个数变动时需要完全重新构建整棵树。为了方便比较堆和败者树的性能，后面的分析都是对n个数构建的堆和败者树来分析的。

总而言之，败者树在进行维护的时候，比较次数是logn+1。与堆不同的是，败者树是从下往上维护，每上一层，只需要和败者节点比较“一次”即可。而堆在 维护的时候是从上往下，每下一层，需要和左右子节点都比较，需要比较两次。从这个角度，败者树比堆更优一些。但是，请注意但是，败者树每一次维护必定需要 从叶子节点一直走到根节点，不可能中间停止；而堆维护时，“有可能”会在中间的某个层停止，不需要继续往下。这样一来，虽然每一层败者树需要的比较次数比 堆少一倍，但是走的层数堆会比败者树少。具体少多少，从平均意义上到底哪一个的效率会更好一些？那我就不知道了，这个分析起来有点麻烦。感兴趣的人可以尝 试一下，讨论讨论。但是至少说明了，也许堆并非是最优的。

具体到我们的问题。类似的方法，先构建一棵有n个叶子节点的败者树，胜出者w是n个中最小的那一个。从N中读入一个新的数m后，和w比较，如果比w小，直 接丢弃，否则用m替换w所在的叶子节点的值，然后维护该败者树。依次执行，直到遍历完N，败者树中保留的n个数就是N中最大的n个数。时间复杂度也是 O（Nlogn）

类快速排序方法：

快速排序大家大家都不陌生了。主要思想是找一个“轴”节点，将数列交换变成两部分，一部分全都小于等于“轴”，另一部分全都大于等于“轴”，然后对两部分 递归处理。其平均时间复杂度是O（NlogN）。从中可以受到启发，如果我们选择的轴使得交换完的“较大”那一部分的数的个数j正好是n，不也就完成了在 N个数中寻找n个最大的数的任务吗？当然，轴也许不能选得这么恰好。可以这么分析，如果j>n，则最大的n个数肯定在这j个数中，则问题变成在这j 个数中找出n个最大的数；否则如果j<n，则这j个数肯定是n个最大的数的一部分，而剩下的j-n个数在小于等于轴的那一部分中，同样可递归处理。

令人愉悦的是，这个算法的平均复杂度是O（N）的。怎么样？比堆的O（Nlogn）可能会好一些吧？！（n如果比较大肯定会好）

需要注意的是，这里的时间复杂度是平均意义上的，在最坏情况下，每次分割都分割成1：N-2，这种情况下的时间复杂度为O（n）。但是我们还有杀手锏，可 以有一个在最坏情况下时间复杂度为O（N）的算法，这个算法是在分割数列的时候保证会按照比较均匀的比例分割，at least 3n/10-6。具体细节我就不再说了，感兴趣的人参考算法导论（Introduction to Algorithms 第二版第九章 “Medians and Orders Statistics”）。

还是那个结论，堆不见得会是最优的。

本文快要 结束了，但是还有一个问题：如果N非常大，存放在磁盘上，不能一次装载进内存呢？怎么办？对于介绍的Naive方法，堆，败者树等等，依然适用，需要注意 的就是每次从磁盘上尽量多读一些数到内存区，然后处理完之后再读入一批。减少IO次数，自然能够提高效率。而对于类快速排序方法，稍微要麻烦一些：分批读 入，假设是M个数，然后从这M个数中选出n个最大的数缓存起来，直到所有的N个数都分批处理完之后，再将各批次缓存的n个数合并起来再进行一次类快速排序 得到最终的n个最大的数就可以了。在运行过程中，如果缓存数太多，可以不断地将多个缓存合并，保留这些缓存中最大的n个数即可。由于类快速排序的时间复杂 度是O（N），这样分批处理再合并的办法，依然有极大的可能会比堆和败者树更优。当然，在空间上会占用较多的内存。

总结：对于这个问题，我 想了很多，但是觉得还有一些地方可以继续深挖：1. 堆和败者树到底哪一个更优？可以通过理论分析，也可以通过实验来比较。也许会有人觉得这个很无聊；2. 有没有近似的算法或者概率算法来解决这个问题？我对这方面实在不熟悉，如果有人有想法的话可以一块交流。如果有分析错误或遗漏的地方，请告知，我不怕丢 人，呵呵！最后请时刻谨记，时间复杂度不等于实际的运行时间，一个常数因子很大的O（logN）算法也许会比常数因子小的O（N）算法慢很多。所以说，n 和N的具体值，以及编程实现的质量，都会影响到实际效率。我看过一篇论文，给出的算法在进行字符串查找时，比hash还要快，是不是难以想象？

**程序员面试题精选**(13)－第一个只出现一次的字符

题目：在一个字符串中找到第一个只出现一次的字符。如输入abaccdeff，则输出b。 分析：这道题是2006年google的一道笔试题。

看到这道题时，最直观的想法是从头开始扫描这个字符串中的每个字符。当访问到某字符时拿这个字符和后面的每个字符相比较，如果在后面没有发现重复的字符，则该字符就是只出现一次的字符。如果字符串有n个字符，每个字符可能与后面的O(n)个字符相比较，因此这种思路时间复杂度是O(n2)。我们试着去找一个更快的方法。

由于题目与字符出现的次数相关，我们是不是可以统计每个字符在该字符串中出现的次数？要达到这个目的，我们需要一个数据容器来存放每个字符的出现次数。在这个数据容器中可以根据字符来查找它出现的次数，也就是说这个容器的作用是把一个字符映射成一个数字。在常用的数据容器中，哈希表正是这个用途。

哈希表是一种比较复杂的数据结构。由于比较复杂，STL中没有实现哈希表，因此需要我们自己实现一个。但由于本题的特殊性，我们只需要一个非常简单的哈希表就能满足要求。由于字符（char）是一个长度为8的数据类型，因此总共有可能256 种可能。于是我们创建一个长度为256的数组，每个字母根据其ASCII码值作为数组的下标对应数组的对应项，而数组中存储的是每个字符对应的次数。这样我们就创建了一个大小为256，以字符ASCII码为键值的哈希表。

我们第一遍扫描这个数组时，每碰到一个字符，在哈希表中找到对应的项并把出现的次数增加一次。这样在进行第二次扫描时，就能直接从哈希表中得到每个字符出现的次数了。

参考代码如下：

///////////////////////////////////////////////////////////////////////

// Find the first char which appears only once in a string

// Input: pString - the string

// Output: the first not repeating char if the string has, otherwise 0

///////////////////////////////////////////////////////////////////////

char FirstNotRepeatingChar(char\* pString)

{

// invalid input

if(!pString)

return 0;

// get a hash table, and initialize it

const int tableSize = 256;

unsigned int hashTable[tableSize];

for(unsigned int i = 0; i < tableSize; ++ i)

hashTable = 0;

// get the how many times each char appears in the string

char\* pHashKey = pString;

while(\*(pHashKey) != '\0')

hashTable[\*(pHashKey++)] ++;

// find the first char which appears only once in a string

pHashKey = pString;

while(\*pHashKey != '\0')

{

if(hashTable[\*pHashKey] == 1)

return \*pHashKey;

pHashKey++;

}

// if the string is empty

// or every char in the string appears at least twice

return 0;

}

**程序员面试题精选100题(42)-旋转数组的最小元素[算法]**

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

         分析：这道题最直观的解法并不难。从头到尾遍历数组一次，就能找出最小的元素，时间复杂度显然是O(*N*)。但这个思路没有利用输入数组的特性，我们应该能找到更好的解法。

         我们注意到旋转之后的数组实际上可以划分为两个排序的子数组，而且前面的子数组的元素都大于或者等于后面子数组的元素。我们还可以注意到最小的元素刚好是这两个子数组的分界线。我们试着用二元查找法的思路在寻找这个最小的元素。

         首先我们用两个指针，分别指向数组的第一个元素和最后一个元素。按照题目旋转的规则，第一个元素应该是大于或者等于最后一个元素的（这其实不完全对，还有特例。后面再讨论特例）。

接着我们得到处在数组中间的元素。如果该中间元素位于前面的递增子数组，那么它应该大于或者等于第一个指针指向的元素。此时数组中最小的元素应该位于该中间元素的后面。我们可以把第一指针指向该中间元素，这样可以缩小寻找的范围。同样，如果中间元素位于后面的递增子数组，那么它应该小于或者等于第二个指针指向的元素。此时该数组中最小的元素应该位于该中间元素的前面。我们可以把第二个指针指向该中间元素，这样同样可以缩小寻找的范围。我们接着再用更新之后的两个指针，去得到和比较新的中间元素，循环下去。

按照上述的思路，我们的第一个指针总是指向前面递增数组的元素，而第二个指针总是指向后面递增数组的元素。最后第一个指针将指向前面子数组的最后一个元素，而第二个指针会指向后面子数组的第一个元素。也就是它们最终会指向两个相邻的元素，而第二个指针指向的刚好是最小的元素。这就是循环结束的条件。

基于这个思路，我们可以写出如下代码：

// Get the minimum number of a roatation of a sorted array

int Min(int \*numbers, int length)

{

    if(numbers == 0 || length <= 0)

        throw new std::exception("Invalid parameters");

    int index1 = 0;

    int index2 = length - 1;

    int indexMid = index1;

    while(numbers[index1] >= numbers[index2])

    {

        if(index2 - index1 == 1)

        {

            indexMid = index2;

            break;

        }

        indexMid = (index1 + index2) / 2;

        if(numbers[indexMid] >= numbers[index1])

            index1 = indexMid;

        else if(numbers[indexMid] <= numbers[index2])

            index2 = indexMid;

    }

    return numbers[indexMid];

}

由于我们每次都把寻找的范围缩小一半，该算法的时间复杂度是O(log*N*)

值得注意的是，如果在面试现场写代码，通常我们需要用一些测试用例来验证代码是不是正确的，我们在验证的时候尽量能考虑全面些。像这道题，我们出来最简单测试用例之外，我们至少还需要考虑如下的情况：

1.              把数组前面的0个元素从最前面搬到最后面，也就是原数组不做改动，根据题目的规则这也是一个旋转，此时数组的第一个元素是大于小于或者等于最后一个元素的；

2.              排好序的数组中有可能有相等的元素，我们特别需要注意两种情况。一是旋转之后的数组中，第一个元素和最后一个元素是相等的；另外一种情况是最小的元素在数组中重复出现

3.              在前面的代码中，如果输入的数组不是一个排序数组的旋转，那将陷入死循环。因此我们需要跟面试官讨论是不是需要判断数组的有效性。在面试的时候，面试官讨论如何验证输入的有效性，能显示我们思维的严密性。本文假设在调用函数Min之前，已经验证过输入的有效性了。

最后需要指出的是，如果输入的数组指针是非法指针，我们是用异常来做错误处理。这是因为在这种情况下，如果我们用return来结束该函数，返回任何数字都不是正确的。关于无效输入时的函数如何返回错误信息并结束，[本博客第17题](http://zhedahht.blog.163.com/blog/static/25411174200731139971)有更详细的讨论，可以参考。

**程序员面试题精选100题(57)-O(n)时间的排序[算法]**

**题目：某公司有几万名员工，请完成一个时间复杂度为O(n)的算法对该公司员工的年龄作排序，可使用O(1)的辅助空间。**

分析：排序是面试时经常被提及的一类题目，我们也熟悉其中很多种算法，诸如插入排序、归并排序、冒泡排序，快速排序等等。这些排序的算法，要么是O(n2)的，要么是O(n*log*n)的。可是这道题竟然要求是O(n)的，这里面到底有什么玄机呢？

                题目特别强调是对一个公司的员工的年龄作排序。员工的数目虽然有几万人，但这几万员工的年龄却只有几十种可能。上班早的人一般也要等到将近二十岁才上班，一般人再晚到了六七十岁也不得不退休。

                由于年龄总共只有几十种可能，我们可以很方便地统计出每一个年龄里有多少名员工。举个简单的例子，假设总共有5个员工，他们的年龄分别是25、24、26、24、25。我们统计出他们的年龄，24岁的有两个，25岁的也有两个，26岁的一个。那么我们根据年龄排序的结果就是：24、24、25、25、26，即在表示年龄的数组里写出两个24、两个25和一个26。

                想明白了这种思路，我们就可以写出如下代码：

void SortAges(int ages[], int length)

{

    if(ages == NULL || length <= 0)

        return;

    const int oldestAge = 99;

    int timesOfAge[oldestAge + 1];

    for(int i = 0; i <= oldestAge; ++ i)

        timesOfAge[i] = 0;

    for(int i = 0; i < length; ++ i)

    {

        int age = ages[i];

        if(age < 0 || age > oldestAge)

            throw new std::exception("age out of range.");

        ++ timesOfAge[age];

    }

    int index = 0;

    for(int i = 0; i <= oldestAge; ++ i)

    {

        for(int j = 0; j < timesOfAge[i]; ++ j)

        {

            ages[index] = i;

            ++ index;

        }

    }

}

                在上面的代码中，允许的范围是0到99岁。数组timesOfAge用来统计每个年龄出现的次数。某个年龄出现了多少次，就在数组ages里设置几次该年龄。这样就相当于给数组ages排序了。该方法用长度100的整数数组辅助空间换来了O(n)的时间效率。由于不管对多少人的年龄作排序，辅助数组的长度是固定的100个整数，因此它的空间复杂度是个常数，即O(1)。

**程序员面试题精选100题(61)-数对之差的最大值[算法]**

**题目：在数组中，数字减去它右边的数字得到一个数对之差。求所有数对之差的最大值。例如在数组{2, 4, 1, 16, 7, 5, 11, 9}中，数对之差的最大值是11，是16减去5的结果。**

分析：看到这个题目，很多人的第一反应是找到这个数组的最大值和最小值，然后觉得最大值减去最小值就是最终的结果。这种思路忽略了题目中很重要的一点：数对之差是一个数字减去它右边的数字。由于我们无法保证最大值一定位于数组的左边，因此这个思路不管用。

于是我们接下来可以想到让每一个数字逐个减去它右边的所有数字，并通过比较得到数对之差的最大值。由于每个数字需要和它后面的O(n)个数字作减法，因此总的时间复杂度是O(n2)。

**解法一：分治法**

通常蛮力法不会是最好的解法，我们想办法减少减法的次数。假设我们把数组分成两个子数组，我们其实没有必要拿左边的子数组中较小的数字去和右边的子数组中较大的数字作减法。我们可以想象，数对之差的最大值只有可能是下面三种情况之一：（1）被减数和减数都在第一个子数组中，即第一个子数组中的数对之差的最大值；（2）被减数和减数都在第二个子数组中，即第二个子数组中数对之差的最大值；（3）被减数在第一个子数组中，是第一个子数组的最大值。减数在第二个子数组中，是第二个子数组的最小值。这三个差值的最大者就是整个数组中数对之差的最大值。

在前面提到的三种情况中，得到第一个子数组的最大值和第二子数组的最小值不是一件难事，但如何得到两个子数组中的数对之差的最大值？这其实是原始问题的子问题，我们可以递归地解决。下面是这种思路的参考代码：

int MaxDiff\_Solution1(int numbers[], unsigned length)

{

    if(numbers == NULL || length < 2)

        return 0;

    int max, min;

    return MaxDiffCore(numbers, numbers + length - 1, &max, &min);

}

int MaxDiffCore(int\* start, int\* end, int\* max, int\* min)

{

    if(end == start)

    {

        \*max = \*min = \*start;

        return 0x80000000;

    }

    int\* middle = start + (end - start) / 2;

    int maxLeft, minLeft;

    int leftDiff = MaxDiffCore(start, middle, &maxLeft, &minLeft);

    int maxRight, minRight;

    int rightDiff = MaxDiffCore(middle + 1, end, &maxRight, &minRight);

    int crossDiff = maxLeft - minRight;

    \*max = (maxLeft > maxRight) ? maxLeft : maxRight;

    \*min = (minLeft < minRight) ? minLeft : minRight;

    int maxDiff = (leftDiff > rightDiff) ? leftDiff : rightDiff;

    maxDiff = (maxDiff > crossDiff) ? maxDiff : crossDiff;

    return maxDiff;

}

在函数MaxDiffCore中，我们先得到第一个子数组中的最大的数对之差leftDiff，再得到第二个子数组中的最大数对之差rightDiff。接下来用第一个子数组的最大值减去第二个子数组的最小值得到crossDiff。这三者的最大值就是整个数组的最大数对之差。

**解法二：转化成求解子数组的最大和问题**

接下来再介绍一种比较巧妙的解法。如果输入一个长度为n的数组numbers，我们先构建一个长度为n-1的辅助数组diff，并且diff[i]等于numbers[i]-numbers[i+1]（0<=i<n-1）。如果我们从数组diff中的第i个数字一直累加到第j个数字（j > i），也就是diff[i] + diff[i+1] + … + diff[j] = (numbers[i]-numbers[i+1]) + (numbers[i + 1]-numbers[i+2]) + ... + (numbers[j] – numbers[j + 1]) = numbers[i] – numbers[j + 1]。

分析到这里，我们发现原始数组中最大的数对之差（即numbers[i] – numbers[j + 1]）其实是辅助数组diff中最大的连续子数组之和。我们在本系列的博客的第3篇[《求子数组的最大和》](http://zhedahht.blog.163.com/blog/static/254111742007219147591/)中已经详细讨论过这个问题的解决方法。基于这个思路，我们可以写出如下代码：

int MaxDiff\_Solution2(int numbers[], unsigned length)

{

    if(numbers == NULL || length < 2)

        return 0;

    int\* diff = new int[length - 1];

    for(int i = 1; i < length; ++i)

        diff[i - 1] = numbers[i - 1] - numbers[i];

    int currentSum = 0;

    int greatestSum = 0x80000000;

    for(int i = 0; i < length - 1; ++i)

    {

        if(currentSum <= 0)

            currentSum = diff[i];

        else

            currentSum += diff[i];

        if(currentSum > greatestSum)

            greatestSum = currentSum;

    }

    delete[] diff;

    return greatestSum;

}

**解法三：动态规划法**

既然我们可以把求最大的数对之差转换成求子数组的最大和，而子数组的最大和可以通过动态规划求解，那我们是不是可以通过动态规划直接求解呢？下面我们试着用动态规划法直接求数对之差的最大值。

我们定义diff[i]是以数组中第i个数字为减数的所有数对之差的最大值。也就是说对于任意h（h < i），diff[i]≥number[h]-number[i]。diff[i]（0≤i<n）的最大值就是整个数组最大的数对之差。

假设我们已经求得了diff[i]，我们该怎么求得diff[i+1]呢？对于diff[i]，肯定存在一个h（h < i），满足number[h]减去number[i]之差是最大的，也就是number[h]应该是number[i]之前的所有数字的最大值。当我们求diff[i+1]的时候，我们需要找到第i+1个数字之前的最大值。第i+1个数字之前的最大值有两种可能：这个最大值可能是第i个数字之前的最大值，也有可能这个最大值就是第i个数字。第i+1个数字之前的最大值肯定是这两者的较大者。我们只要拿第i+1个数字之前的最大值减去number[i+1]，就得到了diff[i+1]。

int MaxDiff\_Solution3(int numbers[], unsigned length)

{

    if(numbers == NULL || length < 2)

        return 0;

    int max = numbers[0];

    int maxDiff =  max - numbers[1];

    for(int i = 2; i < length; ++i)

    {

        if(numbers[i - 1] > max)

            max = numbers[i - 1];

        int currentDiff = max - numbers[i];

        if(currentDiff > maxDiff)

            maxDiff = currentDiff;

    }

    return maxDiff;

}

在上述代码中，max表示第i个数字之前的最大值，而currentDiff表示diff[i] （0≤i<n），diff[i]的最大值就是代码中maxDiff。

**解法小结**

上述三种代码，虽然思路各不相同，但时间复杂度都是O(n)（第一种解法的时间复杂度可以用递归公式表示为T(n)=2(n/2)+O(1)，所以总体时间复杂度是O(n)）。我们也可以注意到第一种方法是基于递归实现，而递归调用是有额外的时间、空间消耗的（比如在调用栈上分配空间保存参数、临时变量等）。第二种方法需要一个长度为n-1的辅助数组，因此其空间复杂度是O(n)。第三种方法则没有额外的时间、空间开销，并且它的代码是最简洁的，因此这是最值得推荐的一种解法。