# 6、数字求和&整数处理

**程序员面试题精选**(08)－求1+2+...+n

题目：求1+2+…+n，要求不能使用乘除法、for、while、if、else、switch、case等关键字以及条件判断语句（A?B:C）。

分析：这道题没有多少实际意义，因为在软件开发中不会有这么变态的限制。但这道题却能有效地考查发散思维能力，而发散思维能力能反映出对编程相关技术理解的深刻程度。

通常求1+2+…+n除了用公式n(n+1)/2之外，无外乎循环和递归两种思路。由于已经明确限制for和while的使用，循环已经不能再用了。同样，递归函数也需要用if语句或者条件判断语句来判断是继续递归下去还是终止递归，但现在题目已经不允许使用这两种语句了。

我们仍然围绕循环做文章。循环只是让相同的代码执行n遍而已，我们完全可以不用for和while达到这个效果。比如定义一个类，我们new一含有n个这种类型元素的数组，那么该类的构造函数将确定会被调用n次。我们可以将需要执行的代码放到构造函数里。如下代码正是基于这个思路：

class Temp

{

public:

Temp() { ++ N; Sum += N; }

static void Reset() { N = 0; Sum = 0; }

static int GetSum() { return Sum; }

private:

static int N;

static int Sum;

};

int Temp::N = 0;

int Temp::Sum = 0;

int solution1\_Sum(int n)

{

Temp::Reset();

Temp \*a = new Temp[n];

delete []a;

a = 0;

return Temp::GetSum();

}

我们同样也可以围绕递归做文章。既然不能判断是不是应该终止递归，我们不妨定义两个函数。一个函数充当递归函数的角色，另一个函数处理终止递归的情况，我们需要做的就是在两个函数里二选一。从二选一我们很自然的想到布尔变量，比如ture（1）的时候调用第一个函数，false（0）的时候调用第二个函数。那现在的问题是如和把数值变量n转换成布尔值。如果对n连续做两次反运算，即!!n，那么非零的n转换为true，0转换为false。有了上述分析，我们再来看下面的代码：

class A;

A\* Array[2];

class A

{

public:

virtual int Sum (int n) { return 0; }

};

class B: public A

{

public:

virtual int Sum (int n) { return Array[!!n]->Sum(n-1)+n; }

};

int solution2\_Sum(int n)

{

A a;

B b;

Array[0] = &a;

Array[1] = &b;

int value = Array[1]->Sum(n);

return value;

}

这种方法是用虚函数来实现函数的选择。当n不为零时，执行函数B::Sum；当n为0时，执行A::Sum。我们也可以直接用函数指针数组，这样可能还更直接一些：

typedef int (\*fun)(int);

int solution3\_f1(int i)

{

return 0;

}

int solution3\_f2(int i)

{

fun f[2]={solution3\_f1, solution3\_f2};

return i+f[!!i](i-1);

}

另外我们还可以让编译器帮我们来完成类似于递归的运算，比如如下代码：

template <int n> struct solution4\_Sum

{

enum Value { N = solution4\_Sum<n - 1>::N + n};

};

template <> struct solution4\_Sum<1>

{

enum Value { N = 1};

};

solution4\_Sum<100>::N就是1+2+...+100的结果。当编译器看到solution4\_Sum<100>时，就是为模板类solution4\_Sum以参数100生成该类型的代码。但以100为参数的类型需要得到以99为参数的类型，因为solution4\_Sum<100>::N=solution4\_Sum<99>::N+100。这个过程会递归一直到参数为1的类型，由于该类型已经显式定义，编译器无需生成，递归编译到此结束。由于这个过程是在编译过程中完成的，因此要求输入n必须是在编译期间就能确定，不能动态输入。这是该方法最大的缺点。而且编译器对递归编译代码的递归深度是有限制的，也就是要求n不能太大。

大家还有更多、更巧妙的思路吗？欢迎讨论^\_^

**程序员面试题精选**(14)－圆圈中最后剩下的数字

题目：n个数字（0,1,…,n-1）形成一个圆圈，从数字0开始，每次从这个圆圈中删除第m个数字（第一个为当前数字本身，第二个为当前数字的下一个数字）。当一个数字删除后，从被删除数字的下一个继续删除第m个数字。求出在这个圆圈中剩下的最后一个数字。

分析：既然题目有一个数字圆圈，很自然的想法是我们用一个数据结构来模拟这个圆圈。在常用的数据结构中，我们很容易想到用环形列表。我们可以创建一个总共有m个数字的环形列表，然后每次从这个列表中删除第m个元素。

在参考代码中，我们用STL中std::list来模拟这个环形列表。由于list并不是一个环形的结构，因此每次跌代器扫描到列表末尾的时候，要记得把跌代器移到列表的头部。这样就是按照一个圆圈的顺序来遍历这个列表了。

这种思路需要一个有n个结点的环形列表来模拟这个删除的过程，因此内存开销为O(n)。而且这种方法每删除一个数字需要m步运算，总共有n个数字，因此总的时间复杂度是O(mn)。当m和n都很大的时候，这种方法是很慢的。

接下来我们试着从数学上分析出一些规律。首先定义最初的n个数字（0,1,…,n-1）中最后剩下的数字是关于n和m的方程为f(n,m)。

在这n个数字中，第一个被删除的数字是m%n-1，为简单起见记为k。那么删除k之后的剩下n-1的数字为0,1,…,k-1,k+1,…,n-1，并且下一个开始计数的数字是k+1。相当于在剩下的序列中，k+1排到最前面，从而形成序列k+1,…,n-1,0,…k-1。该序列最后剩下的数字也应该是关于n和m的函数。由于这个序列的规律和前面最初的序列不一样（最初的序列是从0开始的连续序列），因此该函数不同于前面函数，记为f’(n-1,m)。最初序列最后剩下的数字f(n,m)一定是剩下序列的最后剩下数字f’(n-1,m)，所以f(n,m)=f’(n-1,m)。

接下来我们把剩下的的这n-1个数字的序列k+1,…,n-1,0,…k-1作一个映射，映射的结果是形成一个从0到n-2的序列：

k+1 -> 0

k+2 -> 1

…

n-1 -> n-k-2

0 -> n-k-1

…

k-1 -> n-2

把映射定义为p，则p(x)= (x-k-1)%n，即如果映射前的数字是x，则映射后的数字是(x-k-1)%n。对应的逆映射是p-1(x)=(x+k+1)%n。

由于映射之后的序列和最初的序列有同样的形式，都是从0开始的连续序列，因此仍然可以用函数f来表示，记为f(n-1,m)。根据我们的映射规则，映射之前的序列最后剩下的数字f’(n-1,m)= p-1 [f(n-1,m)]=[f(n-1,m)+k+1]%n。把k=m%n-1代入得到f(n,m)=f’(n-1,m)=[f(n-1,m)+m]%n。

经过上面复杂的分析，我们终于找到一个递归的公式。要得到n个数字的序列的最后剩下的数字，只需要得到n-1个数字的序列的最后剩下的数字，并可以依此类推。当n=1时，也就是序列中开始只有一个数字0，那么很显然最后剩下的数字就是0。我们把这种关系表示为：

0 n=1

f(n,m)={

[f(n-1,m)+m]%n n>1

尽管得到这个公式的分析过程非常复杂，但它用递归或者循环都很容易实现。最重要的是，这是一种时间复杂度为O(n)，空间复杂度为O(1)的方法，因此无论在时间上还是空间上都优于前面的思路。

思路一的参考代码：

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

// n integers (0, 1, ... n - 1) form a circle. Remove the mth from

// the circle at every time. Find the last number remaining

// Input: n - the number of integers in the circle initially

// m - remove the mth number at every time

// Output: the last number remaining when the input is valid,

// otherwise -1

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

int LastRemaining\_Solution1(unsigned int n, unsigned int m)

{

// invalid input

if(n < 1 || m < 1)

return -1;

unsigned int i = 0;

// initiate a list with n integers (0, 1, ... n - 1)

list<int> integers;

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

integers.push\_back(i);

list<int>::iterator curinteger = integers.begin();

while(integers.size() > 1)

{

// find the mth integer. Note that std::list is not a circle

// so we should handle it manually

for(int i = 1; i < m; ++ i)

{

curinteger ++;

if(curinteger == integers.end())

curinteger = integers.begin();

}

// remove the mth integer. Note that std::list is not a circle

// so we should handle it manually

list<int>::iterator nextinteger = ++ curinteger;

if(nextinteger == integers.end())

nextinteger = integers.begin();

-- curinteger;

integers.erase(curinteger);

curinteger = nextinteger;

}

return \*(curinteger);

}

思路二的参考代码：

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

// n integers (0, 1, ... n - 1) form a circle. Remove the mth from

// the circle at every time. Find the last number remaining

// Input: n - the number of integers in the circle initially

// m - remove the mth number at every time

// Output: the last number remaining when the input is valid,

// otherwise -1

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

int LastRemaining\_Solution2(int n, unsigned int m)

{

// invalid input

if(n <= 0 || m < 0)

return -1;

// if there are only one integer in the circle initially,

// of course the last remaining one is 0

int lastinteger = 0;

// find the last remaining one in the circle with n integers

for (int i = 2; i <= n; i ++)

lastinteger = (lastinteger + m) % i;

return lastinteger;

}

如果对两种思路的时间复杂度感兴趣的读者可以把n和m的值设的稍微大一点，比如十万这个数量级的数字，运行的时候就能明显感觉出这两种思路写出来的代码时间效率大不一样。

**程序员面试题精选**(16)－O(logn)求Fibonacci数列

题目：定义Fibonacci数列如下： / 0 n=0

f(n)= 1 n=1

\ f(n-1)+f(n-2) n=2

输入n，用最快的方法求该数列的第n项。

分析：在很多C语言教科书中讲到递归函数的时候，都会用Fibonacci作为例子。因此很多程序员对这道题的递归解法非常熟悉，看到题目就能写出如下的递归求解的代码。

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

// Calculate the nth item of Fibonacci Series recursively

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

long long Fibonacci\_Solution1(unsigned int n)

{

int result[2] = {0, 1};

if(n < 2)

return result[n];

return Fibonacci\_Solution1(n - 1) + Fibonacci\_Solution1(n - 2);

}

但是，教科书上反复用这个题目来讲解递归函数，并不能说明递归解法最适合这道题目。我们以求解f(10)作为例子来分析递归求解的过程。要求得f(10)，需要求得f(9)和f(8)。同样，要求得f(9)，要先求得f(8)和f(7)……我们用树形结构来表示这种依赖关系

f(10)

/ \

f(9) f(8)

/ \ / \

f(8) f(7) f(6) f(5)

/ \ / \

f(7) f(6) f(6) f(5)

我们不难发现在这棵树中有很多结点会重复的，而且重复的结点数会随着n的增大而急剧增加。这意味这计算量会随着n的增大而急剧增大。事实上，用递归方法计算的时间复杂度是以n的指数的方式递增的。大家可以求Fibonacci的第100项试试，感受一下这样递归会慢到什么程度。在我的机器上，连续运行了一个多小时也没有出来结果。

其实改进的方法并不复杂。上述方法之所以慢是因为重复的计算太多，只要避免重复计算就行了。比如我们可以把已经得到的数列中间项保存起来，如果下次需要计算的时候我们先查找一下，如果前面已经计算过了就不用再次计算了。

更简单的办法是从下往上计算，首先根据f(0)和f(1)算出f(2)，在根据f(1)和f(2)算出f(3)……依此类推就可以算出第n项了。很容易理解，这种思路的时间复杂度是O(n)。

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

// Calculate the nth item of Fibonacci Series iteratively

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

long long Fibonacci\_Solution2(unsigned n)

{

int result[2] = {0, 1};

if(n < 2)

return result[n];

long long fibNMinusOne = 1;

long long fibNMinusTwo = 0;

long long fibN = 0;

for(unsigned int i = 2; i <= n; ++ i)

{

fibN = fibNMinusOne + fibNMinusTwo;

fibNMinusTwo = fibNMinusOne;

fibNMinusOne = fibN;

}

return fibN;

}

这还不是最快的方法。下面介绍一种时间复杂度是O(logn)的方法。在介绍这种方法之前，先介绍一个数学公式：

{f(n), f(n-1), f(n-1), f(n-2)} ={1, 1, 1,0}n-1

(注：{f(n+1), f(n), f(n), f(n-1)}表示一个矩阵。在矩阵中第一行第一列是f(n+1)，第一行第二列是f(n)，第二行第一列是f(n)，第二行第二列是f(n-1)。)

有了这个公式，要求得f(n)，我们只需要求得矩阵{1, 1, 1,0}的n-1次方，因为矩阵{1, 1, 1,0}的n-1次方的结果的第一行第一列就是f(n)。这个数学公式用数学归纳法不难证明。感兴趣的朋友不妨自己证明一下。

现在的问题转换为求矩阵{1, 1, 1, 0}的乘方。如果简单第从0开始循环，n次方将需要n次运算，并不比前面的方法要快。但我们可以考虑乘方的如下性质：

/ an/2\*an/2 n为偶数时

an=

\ a(n-1)/2\*a(n-1)/2 n为奇数时

要求得n次方，我们先求得n/2次方，再把n/2的结果平方一下。如果把求n次方的问题看成一个大问题，把求n/2看成一个较小的问题。这种把大问题分解成一个或多个小问题的思路我们称之为分治法。这样求n次方就只需要logn次运算了。

实现这种方式时，首先需要定义一个2×2的矩阵，并且定义好矩阵的乘法以及乘方运算。当这些运算定义好了之后，剩下的事情就变得非常简单。完整的实现代码如下所示。

#include <cassert>

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

// A 2 by 2 matrix

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

struct Matrix2By2

{

Matrix2By2

(

long long m00 = 0,

long long m01 = 0,

long long m10 = 0,

long long m11 = 0

)

:m\_00(m00), m\_01(m01), m\_10(m10), m\_11(m11)

{

}

long long m\_00;

long long m\_01;

long long m\_10;

long long m\_11;

};

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

// Multiply two matrices

// Input: matrix1 - the first matrix

// matrix2 - the second matrix

//Output: the production of two matrices

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

Matrix2By2 MatrixMultiply

(

const Matrix2By2& matrix1,

const Matrix2By2& matrix2

)

{

return Matrix2By2(

matrix1.m\_00 \* matrix2.m\_00 + matrix1.m\_01 \* matrix2.m\_10,

matrix1.m\_00 \* matrix2.m\_01 + matrix1.m\_01 \* matrix2.m\_11,

matrix1.m\_10 \* matrix2.m\_00 + matrix1.m\_11 \* matrix2.m\_10,

matrix1.m\_10 \* matrix2.m\_01 + matrix1.m\_11 \* matrix2.m\_11);

}

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

// The nth power of matrix

// 1 1

// 1 0

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

Matrix2By2 MatrixPower(unsigned int n)

{

assert(n > 0);

Matrix2By2 matrix;

if(n == 1)

{

matrix = Matrix2By2(1, 1, 1, 0);

}

else if(n % 2 == 0)

{

matrix = MatrixPower(n / 2);

matrix = MatrixMultiply(matrix, matrix);

}

else if(n % 2 == 1)

{

matrix = MatrixPower((n - 1) / 2);

matrix = MatrixMultiply(matrix, matrix);

matrix = MatrixMultiply(matrix, Matrix2By2(1, 1, 1, 0));

}

return matrix;

}

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

// Calculate the nth item of Fibonacci Series using devide and conquer

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

long long Fibonacci\_Solution3(unsigned int n)

{

int result[2] = {0, 1};

if(n < 2)

return result[n];

Matrix2By2 PowerNMinus2 = MatrixPower(n - 1);

return PowerNMinus2.m\_00;

}

**程序员面试题精选**(22)－整数的二进制表示中1的个数

题目：输入一个整数，求该整数的二进制表达中有多少个1。例如输入10，由于其二进制表示为1010，有两个1，因此输出2。

分析：这是一道很基本的考查位运算的面试题。包括微软在内的很多公司都曾采用过这道题。

一个很基本的想法是，我们先判断整数的最右边一位是不是1。接着把整数右移一位，原来处于右边第二位的数字现在被移到第一位了，再判断是不是1。这样每次移动一位，直到这个整数变成0为止。现在的问题变成怎样判断一个整数的最右边一位是不是1了。很简单，如果它和整数1作与运算。由于1除了最右边一位以外，其他所有位都为0。因此如果与运算的结果为1，表示整数的最右边一位是1，否则是0。

得到的代码如下：

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

// Get how many 1s in an integer's binary expression

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

int NumberOf1\_Solution1(int i)

{

int count = 0;

while(i)

{

if(i & 1)

count ++;

i = i >> 1;

}

return count;

}

可能有读者会问，整数右移一位在数学上是和除以2是等价的。那可不可以把上面的代码中的右移运算符换成除以2呢？答案是最好不要换成除法。因为除法的效率比移位运算要低的多，在实际编程中如果可以应尽可能地用移位运算符代替乘除法。

这个思路当输入i是正数时没有问题，但当输入的i是一个负数时，不但不能得到正确的1的个数，还将导致死循环。以负数0x80000000为例，右移一位的时候，并不是简单地把最高位的1移到第二位变成0x40000000，而是0xC0000000。这是因为移位前是个负数，仍然要保证移位后是个负数，因此移位后的最高位会设为1。如果一直做右移运算，最终这个数字就会变成0xFFFFFFFF而陷入死循环。

为了避免死循环，我们可以不右移输入的数字i。首先i和1做与运算，判断i的最低位是不是为1。接着把1左移一位得到2，再和i做与运算，就能判断i的次高位是不是1……这样反复左移，每次都能判断i的其中一位是不是1。基于此，我们得到如下代码：

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

// Get how many 1s in an integer's binary expression

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

int NumberOf1\_Solution2(int i)

{

int count = 0;

unsigned int flag = 1;

while(flag)

{

if(i & flag)

count ++;

flag = flag << 1;

}

return count;

}

另外一种思路是如果一个整数不为0，那么这个整数至少有一位是1。如果我们把这个整数减去1，那么原来处在整数最右边的1就会变成0，原来在1后面的所有的0都会变成1。其余的所有位将不受到影响。举个例子：一个二进制数1100，从右边数起的第三位是处于最右边的一个1。减去1后，第三位变成0，它后面的两位0变成1，而前面的1保持不变，因此得到结果是1011。

我们发现减1的结果是把从最右边一个1开始的所有位都取反了。这个时候如果我们再把原来的整数和减去1之后的结果做与运算，从原来整数最右边一个1那一位开始所有位都会变成0。如1100&1011=1000。也就是说，把一个整数减去1，再和原整数做与运算，会把该整数最右边一个1变成0。那么一个整数的二进制有多少个1，就可以进行多少次这样的操作。

这种思路对应的代码如下：

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

// Get how many 1s in an integer's binary expression

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

int NumberOf1\_Solution3(int i)

{

int count = 0;

while (i)

{

++ count;

i = (i - 1) & i;

}

return count;

}

扩展：如何用一个语句判断一个整数是不是二的整数次幂？

**程序员面试题精选**(23)－跳台阶问题

题目：一个台阶总共有n级，如果一次可以跳1级，也可以跳2级。求总共有多少总跳法，并分析算法的时间复杂度。

分析：这道题最近经常出现，包括MicroStrategy等比较重视算法的公司都曾先后选用过个这道题作为面试题或者笔试题。

首先我们考虑最简单的情况。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-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-2) n>2

分析到这里，相信很多人都能看出这就是我们熟悉的Fibonacci序列。至于怎么求这个序列的第n项，请参考本面试题系列

题目：输入两个整数序列。其中一个序列表示栈的push顺序，判断另一个序列有没有可能是对应的pop顺序。为了简单起见，我们假设push序列的任意两个整数都是不相等的。

比如输入的push序列是1、2、3、4、5，那么4、5、3、2、1就有可能是一个pop系列。因为可以有如下的push和pop序列：push 1，push 2，push 3，push 4，pop，push 5，pop，pop，pop，pop，这样得到的pop序列就是4、5、3、2、1。但序列4、3、5、1、2就不可能是push序列1、2、3、4、5的pop序列。

分析：这到题除了考查对栈这一基本数据结构的理解，还能考查我们的分析能力。

这道题的一个很直观的想法就是建立一个辅助栈，每次push的时候就把一个整数push进入这个辅助栈，同样需要pop的时候就把该栈的栈顶整数pop出来。

我们以前面的序列4、5、3、2、1为例。第一个希望被pop出来的数字是4，因此4需要先push到栈里面。由于push的顺序已经由push序列确定了，也就是在把4 push进栈之前，数字1，2，3都需要push到栈里面。此时栈里的包含4个数字，分别是1，2，3，4，其中4位于栈顶。把4 pop出栈后，剩下三个数字1，2，3。接下来希望被pop的是5，由于仍然不是栈顶数字，我们接着在push序列中4以后的数字中寻找。找到数字5后再一次push进栈，这个时候5就是位于栈顶，可以被pop出来。接下来希望被pop的三个数字是3，2，1。每次操作前都位于栈顶，直接pop即可。

再来看序列4、3、5、1、2。pop数字4的情况和前面一样。把4 pop出来之后，3位于栈顶，直接pop。接下来希望pop的数字是5，由于5不是栈顶数字，我们到push序列中没有被push进栈的数字中去搜索该数字，幸运的时候能够找到5，于是把5 push进入栈。此时pop 5之后，栈内包含两个数字1、2，其中2位于栈顶。这个时候希望pop的数字是1，由于不是栈顶数字，我们需要到push序列中还没有被push进栈的数字中去搜索该数字。但此时push序列中所有数字都已被push进入栈，因此该序列不可能是一个pop序列。

也就是说，如果我们希望pop的数字正好是栈顶数字，直接pop出栈即可；如果希望pop的数字目前不在栈顶，我们就到push序列中还没有被push到栈里的数字中去搜索这个数字，并把在它之前的所有数字都push进栈。如果所有的数字都被push进栈仍然没有找到这个数字，表明该序列不可能是一个pop序列。

基于前面的分析，我们可以写出如下的参考代码：

#include <stack>

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

// Given a push order of a stack, determine whether an array is possible to

// be its corresponding pop order

// Input: pPush - an array of integers, the push order

// pPop - an array of integers, the pop order

// nLength - the length of pPush and pPop

// Output: If pPop is possible to be the pop order of pPush, return true.

// Otherwise return false

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

bool IsPossiblePopOrder(const int\* pPush, const int\* pPop, int nLength)

{

bool bPossible = false;

if(pPush && pPop && nLength > 0)

{

const int \*pNextPush = pPush;

const int \*pNextPop = pPop;

// ancillary stack

std::stack<int>stackData;

// check every integers in pPop

while(pNextPop - pPop < nLength)

{

// while the top of the ancillary stack is not the integer

// to be poped, try to push some integers into the stack

while(stackData.empty() || stackData.top() != \*pNextPop)

{

// pNextPush == NULL means all integers have been

// pushed into the stack, can't push any longer

if(!pNextPush)

break;

stackData.push(\*pNextPush);

// if there are integers left in pPush, move

// pNextPush forward, otherwise set it to be NULL

if(pNextPush - pPush < nLength - 1)

pNextPush ++;

else

pNextPush = NULL;

}

// After pushing, the top of stack is still not same as

// pPextPop, pPextPop is not in a pop sequence

// corresponding to pPush

if(stackData.top() != \*pNextPop)

break;

// Check the next integer in pPop

stackData.pop();

pNextPop ++;

}

// if all integers in pPop have been check successfully,

// pPop is a pop sequence corresponding to pPush

if(stackData.empty() && pNextPop - pPop == nLength)

bPossible = true;

}

return bPossible;

}

? 依次检查pop序列，当前栈顶不对就压栈，直到满足为止。如果push序列空了，就返回false

不知道我写的对不对。。。。

int good\_order(int push[], int pop[], int size)

{

int \*tmp = (int \*)malloc(size \* sizeof(int));

int top = 0, cur\_push = 0, cur\_pop = 0;

tmp[top] = push[cur\_push++];

for(; cur\_pop < size; cur\_pop++) {

while(cur\_push < size && tmp[top] != pop[cur\_pop])

tmp[++top] = push[cur\_push++];

if(tmp[top] == pop[cur\_pop])

top--;

else{

free(tmp);

tmp = NULL;

return 0;

}

}

free(tmp);

return 1;

}

**程序员面试题精选**100题(25)-在从1到n的正数中1出现的次数

题目：输入一个整数n，求从1到n这n个整数的十进制表示中1出现的次数。

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

分析：这是一道广为流传的google面试题。用最直观的方法求解并不是很难，但遗憾的是效率不是很高；而要得出一个效率较高的算法，需要比较强的分析能力，并不是件很容易的事情。当然，google的面试题中简单的也没有几道。

首先我们来看最直观的方法，分别求得1到n中每个整数中1出现的次数。而求一个整数的十进制表示中1出现的次数，就和本面试题系列的第22题很相像了。我们每次判断整数的个位数字是不是1。如果这个数字大于10，除以10之后再判断个位数字是不是1。基于这个思路，不难写出如下的代码：

int NumberOf1(unsigned int n);

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

// Find the number of 1 in the integers between 1 and n

// Input: n - an integer

// Output: the number of 1 in the integers between 1 and n

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

int NumberOf1BeforeBetween1AndN\_Solution1(unsigned int n)

{

int number = 0;

// Find the number of 1 in each integer between 1 and n

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

number += NumberOf1(i);

return number;

}

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

// Find the number of 1 in an integer with radix 10

// Input: n - an integer

// Output: the number of 1 in n with radix

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

int NumberOf1(unsigned int n)

{

int number = 0;

while(n)

{

if(n % 10 == 1)

number ++;

n = n / 10;

}

return number;

}

这个思路有一个非常明显的缺点就是每个数字都要计算1在该数字中出现的次数，因此时间复杂度是O(n)。当输入的n非常大的时候，需要大量的计算，运算效率很低。我们试着找出一些规律，来避免不必要的计算。

我们用一个稍微大一点的数字21345作为例子来分析。我们把从1到21345的所有数字分成两段，即1-1235和1346-21345。

先来看1346-21345中1出现的次数。1的出现分为两种情况：一种情况是1出现在最高位（万位）。从1到21345的数字中，1出现在10000-19999这10000个数字的万位中，一共出现了10000（104）次；另外一种情况是1出现在除了最高位之外的其他位中。例子中1346-21345，这20000个数字中后面四位中1出现的次数是2000次（2\*103，其中2的第一位的数值，103是因为数字的后四位数字其中一位为1，其余的三位数字可以在0到9这10个数字任意选择，由排列组合可以得出总次数是2\*103）。

至于从1到1345的所有数字中1出现的次数，我们就可以用递归地求得了。这也是我们为什么要把1-21345分为1-1235和1346-21345两段的原因。因为把21345的最高位去掉就得到1345，便于我们采用递归的思路。

分析到这里还有一种特殊情况需要注意：前面我们举例子是最高位是一个比1大的数字，此时最高位1出现的次数104（对五位数而言）。但如果最高位是1呢？比如输入12345，从10000到12345这些数字中，1在万位出现的次数就不是104次，而是2346次了，也就是除去最高位数字之后剩下的数字再加上1。

基于前面的分析，我们可以写出以下的代码。在参考代码中，为了编程方便，我把数字转换成字符串了。

#include "string.h"

#include "stdlib.h"

int NumberOf1(const char\* strN);

int PowerBase10(unsigned int n);

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

// Find the number of 1 in an integer with radix 10

// Input: n - an integer

// Output: the number of 1 in n with radix

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

int NumberOf1BeforeBetween1AndN\_Solution2(int n)

{

if(n <= 0)

return 0;

// convert the integer into a string

char strN[50];

sprintf(strN, "%d", n);

return NumberOf1(strN);

}

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

// Find the number of 1 in an integer with radix 10

// Input: strN - a string, which represents an integer

// Output: the number of 1 in n with radix

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

int NumberOf1(const char\* strN)

{

if(!strN || \*strN < '0' || \*strN > '9' || \*strN == '\0')

return 0;

int firstDigit = \*strN - '0';

unsigned int length = static\_cast<unsigned int>(strlen(strN));

// the integer contains only one digit

if(length == 1 && firstDigit == 0)

return 0;

if(length == 1 && firstDigit > 0)

return 1;

// suppose the integer is 21345

// numFirstDigit is the number of 1 of 10000-19999 due to the first digit

int numFirstDigit = 0;

// numOtherDigits is the number of 1 01346-21345 due to all digits

// except the first one

int numOtherDigits = firstDigit \* (length - 1) \* PowerBase10(length - 2);

// numRecursive is the number of 1 of integer 1345

int numRecursive = NumberOf1(strN + 1);

// if the first digit is greater than 1, suppose in integer 21345

// number of 1 due to the first digit is 10^4. It's 10000-19999

if(firstDigit > 1)

numFirstDigit = PowerBase10(length - 1);

// if the first digit equals to 1, suppose in integer 12345

// number of 1 due to the first digit is 2346. It's 10000-12345

else if(firstDigit == 1)

numFirstDigit = atoi(strN + 1) + 1;

return numFirstDigit + numOtherDigits + numRecursive;

}

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

// Calculate 10^n

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

int PowerBase10(unsigned int n)

{

int result = 1;

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

result \*= 10;

return result;

}

**程序员面试题精选**100题(26)-和为n连续正数序列

题目：输入一个正数n，输出所有和为n连续正数序列。

例如输入15，由于1+2+3+4+5=4+5+6=7+8=15，所以输出3个连续序列1-5、4-6和7-8。

分析：这是网易的一道面试题。

这道题和本面试题系列的第10题有些类似。我们用两个数small和big分别表示序列的最小值和最大值。首先把small初始化为1，big初始化为2。如果从small到big的序列的和大于n的话，我们向右移动small，相当于从序列中去掉较小的数字。如果从small到big的序列的和小于n的话，我们向右移动big，相当于向序列中添加big的下一个数字。一直到small等于(1+n)/2，因为序列至少要有两个数字。

基于这个思路，我们可以写出如下代码：

void PrintContinuousSequence(int small, int big);

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

// Find continuous sequence, whose sum is n

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

void FindContinuousSequence(int n)

{

if(n < 3)

return;

int small = 1;

int big = 2;

int middle = (1 + n) / 2;

int sum = small + big;

while(small < middle)

{

// we are lucky and find the sequence

if(sum == n)

PrintContinuousSequence(small, big);

// if the current sum is greater than n,

// move small forward

while(sum > n)

{

sum -= small;

small ++;

// we are lucky and find the sequence

if(sum == n)

PrintContinuousSequence(small, big);

}

// move big forward

big ++;

sum += big;

}

}

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

// Print continuous sequence between small and big

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

void PrintContinuousSequence(int small, int big)

{

for(int i = small; i <= big; ++ i)

printf("%d ", i);

printf("\n");

}

**程序员面试题精选**（38）：2008百度校园招聘的一道笔试题

题目大意如下：

一排N（最大１Ｍ）个正整数+1递增，乱序排列，第一个不是最小的，把它换成-1，最小数为ａ且未知求第一个被

-1替换掉的数原来的值，并分析算法复杂度。

解题思路：

一般稍微有点算法知识的人想想就会很容易给出以下解法：

设　Sn = a + (a+1) + (a+2) + .........+ (a+n-1) = na +n(n-1)/2

扫一次数组即可找到最小值ａ，时间复杂度O(n)

设　S = 修改第一项后所有数组项之和,　　求和复杂度为O(n)

则被替换掉的第一项为　　a1=Sn-S-１

总的时间复杂度为　Ｏ(1)+O(n)+O(n) = O(n)

根据该算法写出程序很简单，就不写了

主要是解题过程中没有太考虑题目中给的１Ｍ这个数字，一面的时候被问到求和溢出怎么办？

当时我一想，如果要考虑溢出，必然是要处理大数问题，以前没有看到大数就头疼……所以立马想了个绕过大数加法的方法，如下：

设定另外一个数组b[N]

用　a, a+1,a+2....a+n-1依次分别减去原数组，得到的差放在该数组里，此求差过程复杂度为O(n)

对该数组各项求和即可得到Sn-S

面试官让证明一下我的设想，当时还没有给我纸和笔，用手在桌子上比划了一下没想出来，回来躺在床上想了一会就想出来了，也没什么难度：

相减求和后的数组，最差情况下应该是连续n/2个负数或者正数相加，如果不溢出，后面正负混合相加的话肯定不会溢出；这种情况下的最差特殊情况就 是，原数列按照降序排列（除了第一项被替换掉了），而我们减时所用数列是增序排列。所得结果将是１个正数，n/2-1个负数，n/2个正数；而且我们相当 于用最大的n/2个数减去最小的n/2个数，差值之和最大，取到了最差情况，我们只考虑后面一半求和的情况即可（前面有个-1不方便处理）：

S(n/2) = (n-1) + (n-3) + (n-5)+ .....+ 1 (n为奇数时最后一项是0，不影响我们讨论数量级计算溢出）

　　　=　[(n-1)+1] \* n/4 = n^2/4

题目中给定n最大为1M = 1024\*1024

那么S(n/2)的最大量级为1024^4 = 2^40

而long long类型为64位，可以存放下该和，成功避免大数问题。

直接求和办法，一是和可能溢出，二是面试官要求把原始数组改称long long的话(即a可以也可能很大，求和时稍微加一下就会溢出)就得考虑大数求解了；而这种差值办法可以直接消掉a，求和只和n相关，和a无关。

**程序员面试题精选**（42）：约瑟夫问题的数学方法

问题描述：N个人围成圆圈，从1开始报数，到第M个人令其出列，然后下一个人继续从1开

始报数，到第M个人令其出列，如此下去，直到只剩一个人为止。显示最后一个人为剩者。

无论是用链表实现还是用数组实现都有一个共同点：要模拟整个游戏过程，不仅程序写起

来比较烦，而且时间复杂

度高达O(nm)，当n，m非常大(例如上百万，上千万)的时候，几乎是没有办法在短时间内出

结果的。

为了讨论方便，先把问题稍微改变一下，并不影响原意：

问题描述：n个人（编号0~(n-1))，从0开始报数，报到(m-1)的退出，剩下的人继续从0开

始报数。求胜利者的编号。

我们知道第一个人(编号一定是m%n-1) 出列之后，剩下的n-1个人组成了一个新的约瑟夫环

（以编号为k=m%n的人开

始）:

k k+1 k+2 ... n-2, n-1, 0, 1, 2, ... k-2

并且从k开始报0。

现在我们把他们的编号做一下转换：

k --> 0

k+1 --> 1

k+2 --> 2

...

...

k-2 --> n-2

k-1 --> n-1

变换后就完完全全成为了(n-1)个人报数的子问题，假如我们知道这个子问题的解：例如x

是最终的胜利者，那么根

据上面这个表把这个x变回去不刚好就是n个人情况的解吗？！！变回去的公式很简单，相

信大家都可以推出来：x'

=(x+k)%n

如何知道(n-1)个人报数的问题的解？对，只要知道(n-2)个人的解就行了。(n-2)个人的解

呢？当然是先求(n-3)的

情况 ---- 这显然就是一个倒推问题！好了，思路出来了，下面写递推公式：

令f[i]表示i个人玩游戏报m退出最后胜利者的编号，最后的结果自然是f[n]

递推公式

f[1]=0;

f[i]=(f[i-1]+m)%i; (i>1)

有了这个公式，我们要做的就是从1-n顺序算出f[i]的数值，最后结果是f[n]。因为实际生

活中编号总是从1开始，

我们输出f[n]+1

由于是逐级递推，不需要保存每个f[i]，程序也是异常简单：

＃include <stdio.h>

main()

{

int n, m, i, s=0;

printf ("N M = "); scanf("%d%d", &n, &m);

for (i=2; i<=n; i++) s=(s+m)%i;

printf ("The winner is %d\n", s+1);

}

这个算法的时间复杂度为O(n)，相对于模拟算法已经有了很大的提高。算n，m等于一百万

，一千万的情况不是问题

了。

**程序员面试题精选**（43）：数组中连续元素相加和最小的元素序列

题目描述： 有一个集合{14,56,53,4,-9,34,...n}里面共n个数

里面可以有负数也可以没有

用一个时间复杂度为o(n)的算法找出其中的一个连续串象(53,4,-9) 这样(串里的数字个数任意)

使得这个连续串为所有这样连续串里各个数字相加和最小的一个

代码实现如下（程序没有考虑有多组解的情况）

#include <iostream>

using namespace std;

template<typename T>

int getMinSum(T\* a,int n,T\* pbegin,T\* pend)

{

T min = a[0];

T sum = a[0];

T tempbegin = 0;

\*pbegin = 0;

\*pend = 0;

for (int i = 1; i < n; i++)

{

if(sum < 0)

sum = sum + a[i];

else

{

tempbegin = i;

sum = a[i];

}

if (sum < min)

{

min = sum;

\*pbegin = tempbegin;

\*pend = i;

}

}

return min;

}

int main()

{

int a[] = {8, 9, -3 ,-10 ,7 ,0 ,8 ,-12, 9, 8 ,-1 ,-2 ,9};

int begin;

int end;

int sum;

int k = sizeof(a) / sizeof(int);

sum=getMinSum(a,k,&begin,&end);

cout<<"The min sum is "<<sum<<endl;

cout<<"And the begin is "<<begin<<",and the end is "<<end<<endl;

return 0;

}

**程序员面试题精选**（44）：整数分割（即求一个数N由小于等于N的数相加所得的所有组合）

题目描述：比如给定一整数4，其有如下情况：4=4；

4=3+1；

4=2+2；

4=2+1+1；

4=1+1+1+1；

下面便是两种版本的分割实现代码。

#include "stdio.h"

int Compute( int number, int maximum)

{

if (number == 1 || maximum == 1 )

return 1 ;

else if (number < maximum)

return Compute(number, number);

else if (number == maximum)

return 1 + Compute(number, maximum - 1 );

else

return Compute(number, maximum - 1 ) + Compute(number - maximum, maximum);

}

int IntPartionNo( int n)///////////////////求组合总数版本；

{

return Compute(n, n);

}

int IntegerPartition( int n)///////////////求组合总数并打印出所有情况版本；

{

int \* partition = new int [n]();

int \* repeat = new int [n]();

partition[ 1 ] = n;

repeat[ 1 ] = 1 ;

int count = 1 ;

int part = 1 ;

int last, smaller, remainder;

printf( " %3d " , partition[ 1 ]);

do

{

last = (partition[part] == 1 ) ? (repeat[part -- ] + 1 ) : 1 ;

smaller = partition[part] - 1 ;

if (repeat[part] != 1 )

-- repeat[part ++ ];

partition[part] = smaller;

repeat[part] = 1 + last / smaller;

if ((remainder = last % smaller) != 0 )

{

partition[ ++ part] = remainder;

repeat[part] = 1 ;

}

++ count;

printf( " \n " );

for ( int i = 1 ; i <= part; ++ i)

for ( int j = 1 ; j <= repeat[i]; ++ j)

printf( " %3d " , partition[i]);

} while (repeat[part] != n);

if (partition)

{

delete [] partition;

partition = 0 ;

}

if (repeat)

{

delete [] repeat;

repeat = 0 ;

}

return count;

}

int main()

{

printf("%d\n",IntPartionNo(4));

IntegerPartition(4);

getchar();

}

**程序员面试题精选**（45）：求给定整数其二进制形式含1的个数

题目描述：求给定整数其二进制形式含1的个数，比如255含8个1，因其二进制表示为11111111；

下面给出了两种求解代码实现。

#include <iostream>

using namespace std;

int CountOne( int n)

{

int count = 0 ;

while (n)

{

++ count;

n &= n - 1 ;

}

return count;

}

int CountOnesUsingTable(unsigned int i)

{

static int BIT\_TABLES[ 256 ] =

{

0 , 1 , 1 , 2 , 1 , 2 , 2 , 3 , 1 , 2 , 2 , 3 , 2 , 3 , 3 , 4

, 1 , 2 , 2 , 3 , 2 , 3 , 3 , 4 , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5

, 1 , 2 , 2 , 3 , 2 , 3 , 3 , 4 , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5

, 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5 , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6

, 1 , 2 , 2 , 3 , 2 , 3 , 3 , 4 , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5

, 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5 , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6

, 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5 , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6

, 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6 , 4 , 5 , 5 , 6 , 5 , 6 , 6 , 7

, 1 , 2 , 2 , 3 , 2 , 3 , 3 , 4 , 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5

, 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5 , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6

, 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5 , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6

, 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6 , 4 , 5 , 5 , 6 , 5 , 6 , 6 , 7

, 2 , 3 , 3 , 4 , 3 , 4 , 4 , 5 , 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6

, 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6 , 4 , 5 , 5 , 6 , 5 , 6 , 6 , 7

, 3 , 4 , 4 , 5 , 4 , 5 , 5 , 6 , 4 , 5 , 5 , 6 , 5 , 6 , 6 , 7

, 4 , 5 , 5 , 6 , 5 , 6 , 6 , 7 , 5 , 6 , 6 , 7 , 6 , 7 , 7 , 8

} ;

return BIT\_TABLES[i & 255 ] + BIT\_TABLES[i >> 8 & 255 ] +

BIT\_TABLES[i >> 16 & 255 ] + BIT\_TABLES[i >> 24 & 255 ];

}

int main()

{

cout<<"158 has : "<<CountOne(158)<<endl;

cout<<"158 has : "<<CountOnesUsingTable(158)<<endl;

getchar();

}

**程序员面试题（60）：不要被阶乘吓倒收藏**

新一篇: 用fstream对二进制文件的读写 | 旧一篇: static\_cast、dynamic\_cast、reinterpret\_cast、和const\_cast

阶乘（Factorial）是个很有意思的函数，但是不少人都比较怕它，我们来看看两个与阶乘相关的问题：

1. 给定一个整数N，那么N的阶乘N！末尾有多少个0呢？例如：N＝10，N！＝3 628 800，N！的末尾有两个0。

2. 求N！的二进制表示中最低位1的位置。

请点击"我要发言"，提交您的解法或者问题。

我要看答案

有些人碰到这样的题目会想：是不是要完整计算出N！的值？如果溢出怎么办？事实上，如果我们从"哪些数相乘能得到10"这个角度来考虑，问题就变得简单了。

首 先考虑，如果N！= K×10M，且K不能被10整除，那么N！末尾有M个0。再考虑对N！进行质因数分解，N！=（2x）×（3y）×（5z）…，由于10 = 2×5，所以M只跟X和Z相关，每一对2和5相乘可以得到一个10，于是M = min（X, Z）。不难看出X大于等于Z，因为能被2整除的数出现的频率比能被5整除的数高得多，所以把公式简化为M = Z。

根据上面的分析，只要计算出Z的值，就可以得到N！末尾0的个数。

【问题1的解法一】

要计算Z，最直接的方法，就是计算i（i =1, 2, …, N）的因式分解中5的指数，然后求和：

代码清单2-6

--------------------------------------------------------------------------------

ret = 0;

for(i = 1; i <= N; i++)

{

j = i;

while(j % 5 ==0)

{

ret++;

j /= 5;

}

}

--------------------------------------------------------------------------------

【问题1的解法二】

公式：Z = [N/5] +[N/52] +[N/53] + …（不用担心这会是一个无穷的运算，因为总存在一个K，使得5K > N，[N/5K]=0。）

公式中，[N/5]表示不大于N的数中5的倍数贡献一个5，[N/52]表示不大于N的数中52的倍数再贡献一个5，……代码如下：

ret = 0;

while(N)

{

ret += N / 5;

N /= 5;

}

问题2要求的是N！的二进制表示中最低位1的位置。给定一个整数N，求N！二进制表示的最低位1在第几位？例如：给定N = 3，N！= 6，那么N！的二进制表示（1 010）的最低位1在第二位。

为了得到更好的解法，首先要对题目进行一下转化。

首先来看一下一个二进制数除以2的计算过程和结果是怎样的。

把一个二进制数除以2，实际过程如下：

判断最后一个二进制位是否为0，若为0，则将此二进制数右移一位，即为商值（为什么）；反之，若为1，则说明这个二进制数是奇数，无法被2整除（这又是为什么）。

所以，这个问题实际上等同于求N！含有质因数2的个数。即答案等于N！含有质因数2的个数加1。

【问题2的解法一】

由于N! 中含有质因数2的个数，等于 N/2 + N/4 + N/8 + N/16 + …[1]，

根据上述分析，得到具体算法，如下所示：

代码清单2-7

--------------------------------------------------------------------------------

int lowestOne(int N)

{

int Ret = 0;

while(N)

{

N >>= 1;

Ret += N;

}

return Ret;

}

--------------------------------------------------------------------------------

【问题2的解法二】

N！含有质因数2的个数，还等于N减去N的二进制表示中1的数目。我们还可以通过这个规律来求解。

下面对这个规律进行举例说明，假设 N = 11011，那么N!中含有质因数2的个数为 N/2 + N/4 + N/8 + N/16 + …

即： 1101 + 110 + 11 + 1

=（1000 + 100 + 1）

+（100 + 10）

+（10 + 1）

+ 1

=（1000 + 100+ 10 + 1）+（100 + 10 + 1）+ 1

= 1111 + 111 + 1

=（10000 -1）+（1000 - 1）+（10-1）+（1-1）

= 11011-N二进制表示中1的个数

小结

任 意一个长度为m的二进制数N可以表示为N = b[1] + b[2] \* 2 + b[3] \* 22 + … + b[m] \* 2(m-1)，其中b [ i ]表示此二进制数第i位上的数字（1或0）。所以，若最低位b[1]为1，则说明N为奇数；反之为偶数，将其除以2，即等于将整个二进制数向低位移一位。

相关题目

给定整数n，判断它是否为2的方幂（解答提示：n>0&&（（n&（n-1））==0））。

--------------------------------------------------------------------------------

[1] 这个规律请读者自己证明（提示N/k，等于1, 2, 3, …, N中能被k整除的数的个数）。

一道经典的面试题：如何从N个数中选出最大（小）的n个数？收藏

新一篇: c/c++基本文件读写 | 旧一篇: 变态比赛规则

**程序员面试题精选** **1到n之间1的个数**收藏

Consider a function which, for a given whole number n, returns the number of ones required when writing out all numbers between 0 and n.

For example, f(13)=6. Notice that f(1)=1. What is the next largest n such that f(n)=n?

算法思想：

循环求出每个数中1的个数，累计之，若满足f(n)=n，则退出，否则继续。

代码如下：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 0~n之间1的个数,如f(13)=6

\* 1,2,3,4,5,6,7,8,9,10,11,12,13.1的个数为6

\* 要求:输入一个正整数n,求出f(n),及求解f(n)=n

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <string.h>

#include <Windows.h>

class CalculationTimes

{

public:

CalculationTimes(){}

~CalculationTimes(){}

int GetTimes(int n);

};

//计算正整数n中1的个数

int CalculationTimes::GetTimes(int n)

{

int count=0;

while(n)

{

if(n%10==1)

count++;

n/=10;

}

return count;

}

//显示菜单

void show\_menu()

{

printf("--------------------------------------------- ");

printf("input command to test the program ");

printf(" i or I : input n to test ");

printf(" g or G : get n to enable f(n)=n ");

printf(" q or Q : quit ");

printf("--------------------------------------------- ");

printf("$ input command >");

}

void main()

{

char sinput[10];

int n;

show\_menu();

scanf("%s",sinput);

while(stricmp(sinput,"q")!=0)

{

int t=0,count=0;

if(stricmp(sinput,"i")==0)

{

printf(" please input an integer:");

scanf("%d",&n);

count=0;

CalculationTimes obj;

t=GetTickCount();

for(int i=1;i<=n;i++)

count+=obj.GetTimes(i);

t=GetTickCount()-t;

printf(" count=%d time=%d ",count,t);

}

else if(stricmp(sinput,"g")==0)

{

CalculationTimes obj;

n=2;

count=1;

t=GetTickCount();

while(1)

{

count+=obj.GetTimes(n);

if(count==n)

break;

n++;

}

t=GetTickCount()-t;

printf(" f(n)=n=%d time=%d ",n,t);

}

//输入命令

printf("$ input command >");

scanf("%s",sinput);

}

}

**程序员面试题精选100题(37)-寻找丑数[算法]**

题目：我们把只包含因子

2、3和5的数称作丑数（Ugly Number）。例如6、8都是丑数，但14不是，因为它包含因子7。习惯上我们把1当做是第一个丑数。求按从小到大的顺序的第1500个丑数。

分析：这是一道在网络上广为流传的面试题，据说google曾经采用过这道题。

所谓一个数m是另一个数n的因子，是指n能被m整除，也就是n % m == 0。根据丑数的定义，丑数只能被2、3和5整除。也就是说如果一个数如果它能被2整除，我们把它连续除以2；如果能被3整除，就连续除以3；如果能被5整除，就除以连续5。如果最后我们得到的是1，那么这个数就是丑数，否则不是。

基于前面的分析，我们可以写出如下的函数来判断一个数是不是丑数：

bool IsUgly(int number)

{

    while(number % 2 == 0)

        number /= 2;

    while(number % 3 == 0)

        number /= 3;

    while(number % 5 == 0)

        number /= 5;

    return (number == 1) ? true : false;

}

接下来，我们只需要按顺序判断每一个整数是不是丑数，即：

int GetUglyNumber\_Solution1(int index)

{

    if(index <= 0)

        return 0;

    int number = 0;

    int uglyFound = 0;

    while(uglyFound < index)

    {

        ++number;

        if(IsUgly(number))

        {

            ++uglyFound;

        }

    }

    return number;

}

我们只需要在函数GetUglyNumber\_Solution1中传入参数1500，就能得到第1500个丑数。该算法非常直观，代码也非常简洁，但最大的问题我们每个整数都需要计算。即使一个数字不是丑数，我们还是需要对它做求余数和除法操作。因此该算法的时间效率不是很高。

接下来我们换一种思路来分析这个问题，试图只计算丑数，而不在非丑数的整数上花费时间。根据丑数的定义，丑数应该是另一个丑数乘以2、3或者5的结果（1除外）。因此我们可以创建一个数组，里面的数字是排好序的丑数。里面的每一个丑数是前面的丑数乘以2、3或者5得到的。

这种思路的关键在于怎样确保数组里面的丑数是排好序的。我们假设数组中已经有若干个丑数，排好序后存在数组中。我们把现有的最大丑数记做M。现在我们来生成下一个丑数，该丑数肯定是前面某一个丑数乘以2、3或者5的结果。我们首先考虑把已有的每个丑数乘以2。在乘以2的时候，能得到若干个结果小于或等于M的。由于我们是按照顺序生成的，小于或者等于M肯定已经在数组中了，我们不需再次考虑；我们还会得到若干个大于M的结果，但我们只需要第一个大于M的结果，因为我们希望丑数是按从小到大顺序生成的，其他更大的结果我们以后再说。我们把得到的第一个乘以2后大于M的结果，记为M2。同样我们把已有的每一个丑数乘以3和5，能得到第一个大于M的结果M3和M5。那么下一个丑数应该是M2、M3和M5三个数的最小者。

前面我们分析的时候，提到把已有的每个丑数分别都乘以2、3和5，事实上是不需要的，因为已有的丑数是按顺序存在数组中的。对乘以2而言，肯定存在某一个丑数T2，排在它之前的每一个丑数乘以2得到的结果都会小于已有最大的丑数，在它之后的每一个丑数乘以2得到的结果都会太大。我们只需要记下这个丑数的位置，同时每次生成新的丑数的时候，去更新这个T2。对乘以3和5而言，存在着同样的T3和T5。

有了这些分析，我们不难写出如下的代码：

int GetUglyNumber\_Solution2(int index)

{

    if(index <= 0)

        return 0;

    int \*pUglyNumbers = new int[index];

    pUglyNumbers[0] = 1;

    int nextUglyIndex = 1;

    int \*pMultiply2 = pUglyNumbers;

    int \*pMultiply3 = pUglyNumbers;

    int \*pMultiply5 = pUglyNumbers;

    while(nextUglyIndex < index)

    {

        int min = Min(\*pMultiply2 \* 2, \*pMultiply3 \* 3, \*pMultiply5 \* 5);

        pUglyNumbers[nextUglyIndex] = min;

        while(\*pMultiply2 \* 2 <= pUglyNumbers[nextUglyIndex])

            ++pMultiply2;

        while(\*pMultiply3 \* 3 <= pUglyNumbers[nextUglyIndex])

            ++pMultiply3;

        while(\*pMultiply5 \* 5 <= pUglyNumbers[nextUglyIndex])

            ++pMultiply5;

        ++nextUglyIndex;

    }

    int ugly = pUglyNumbers[nextUglyIndex - 1];

    delete[] pUglyNumbers;

    return ugly;

}

int Min(int number1, int number2, int number3)

{

    int min = (number1 < number2) ? number1 : number2;

    min = (min < number3) ? min : number3;

    return min;

}

和第一种思路相比，这种算法不需要在非丑数的整数上做任何计算，因此时间复杂度要低很多。感兴趣的读者可以分别统计两个函数GetUglyNumber\_Solution1(1500)和GetUglyNumber\_Solution2(1500)的运行时间。当然我们也要指出，第二种算法由于要保存已经生成的丑数，因此需要一个数组，从而需要额外的内存。第一种算法是没有这样的内存开销的。

**程序员面试题精选100题(38)-输出1到最大的N位数[算法]**

题目：输入数字n，按顺序输出从1最大的n位10进制数。比如输入3，则输出1、2、3一直到最大的3位数即999。

分析：这是一道很有意思的题目。看起来很简单，其实里面却有不少的玄机。

应聘者在解决这个问题的时候，最容易想到的方法是先求出最大的n位数是什么，然后用一个循环从1开始逐个输出。很快，我们就能写出如下代码：

// Print numbers from 1 to the maximum number with n digits, in order

void Print1ToMaxOfNDigits\_1(int n)

{

    // calculate 10^n

    int number = 1;

    int i = 0;

    while(i++ < n)

        number \*= 10;

    // print from 1 to (10^n - 1)

    for(i = 1; i < number; ++i)

        printf("%d\t", i);

}

初看之下，好像没有问题。但如果我们仔细分析这个问题，就能注意到这里没有规定n的范围，当我们求最大的n位数的时候，是不是有可能用整型甚至长整型都会溢出？

分析到这里，我们很自然的就想到我们需要表达一个大数。最常用的也是最容易实现的表达大数的方法是用字符串或者整型数组（当然不一定是最有效的）。

用字符串表达数字的时候，最直观的方法就是字符串里每个字符都是’0’到’9’之间的某一个字符，表示数字中的某一位。因为数字最大是n位的，因此我们需要一个n+1位字符串（最后一位为结束符号’\0’）。当实际数字不够n位的时候，在字符串的前半部分补零。这样，数字的个位永远都在字符串的末尾（除去结尾符号）。

首先我们把字符串中每一位数字都初始化为’0’。然后每一次对字符串表达的数字加1，再输出。因此我们只需要做两件事：一是在字符串表达的数字上模拟加法。另外我们要把字符串表达的数字输出。值得注意的是，当数字不够n位的时候，我们在数字的前面补零。输出的时候这些补位的0不应该输出。比如输入3的时候，那么数字98以098的形式输出，就不符合我们的习惯了。

基于上述分析，我们可以写出如下代码：

// Print numbers from 1 to the maximum number with n digits, in order

void Print1ToMaxOfNDigits\_2(int n)

{

    // 0 or minus numbers are invalid input

    if(n <= 0)

        return;

    // number is initialized as 0

    char \*number = new char[n + 1];

    memset(number, '0', n);

    number[n] = '\0';

    while(!Increment(number))

    {

        PrintNumber(number);

    }

    delete []number;

}

// Increment a number. When overflow, return true; otherwise return false

bool Increment(char\* number)

{

    bool isOverflow = false;

    int nTakeOver = 0;

    int nLength = strlen(number);

    // Increment (Add 1) operation begins from the end of number

    for(int i = nLength - 1; i >= 0; i --)

    {

        int nSum = number[i] - '0' + nTakeOver;

        if(i == nLength - 1)

            nSum ++;

        if(nSum >= 10)

        {

            if(i == 0)

                isOverflow = true;

            else

            {

                nSum -= 10;

                nTakeOver = 1;

                number[i] = '0' + nSum;

            }

        }

        else

        {

            number[i] = '0' + nSum;

            break;

        }

    }

    return isOverflow;

}

// Print number stored in string, ignore 0 at its beginning

// For example, print "0098" as "98"

void PrintNumber(char\* number)

{

    bool isBeginning0 = true;

    int nLength = strlen(number);

    for(int i = 0; i < nLength; ++ i)

    {

        if(isBeginning0 && number[i] != '0')

            isBeginning0 = false;

        if(!isBeginning0)

        {

            printf("%c", number[i]);

        }

    }

    printf("\t");

}

    第二种思路基本上和第一种思路相对应，只是把一个整型数值换成了字符串的表示形式。同时，值得提出的是，判断打印是否应该结束时，我没有调用函数strcmp比较字符串number和”99…999”（n个9）。这是因为strcmp的时间复杂度是O(n)，而判断是否溢出的平均时间复杂度是O(1)。

         第二种思路虽然比较直观，但由于模拟了整数的加法，代码有点长。要在面试短短几十分钟时间里完整正确写出这么长代码，不是件容易的事情。接下来我们换一种思路来考虑这个问题。如果我们在数字前面补0的话，就会发现n位所有10进制数其实就是n个从0到9的全排列。也就是说，我们把数字的每一位都从0到9排列一遍，就得到了所有的10进制数。只是我们在输出的时候，数字排在前面的0我们不输出罢了。

         全排列用递归很容易表达，数字的每一位都可能是0到9中的一个数，然后设置下一位。递归结束的条件是我们已经设置了数字的最后一位。

// Print numbers from 1 to the maximum number with n digits, in order

void Print1ToMaxOfNDigits\_3(int n)

{

    // 0 or minus numbers are invalid input

    if(n <= 0)

        return;

    char\* number = new char[n + 1];

    number[n] = '\0';

    for(int i = 0; i < 10; ++i)

    {

        // first digit can be 0 to 9

        number[0] = i + '0';

        Print1ToMaxOfNDigitsRecursively(number, n, 0);

    }

    delete[] number;

}

// length: length of number

// index: current index of digit in number for this round

void Print1ToMaxOfNDigitsRecursively(char\* number, int length, int index)

{

    // we have reached the end of number, print it

    if(index == length - 1)

    {

        PrintNumber(number);

        return;

    }

    for(int i = 0; i < 10; ++i)

    {

        // next digit can be 0 to 9

        number[index + 1] = i + '0';

        // go to the next digit

        Print1ToMaxOfNDigitsRecursively(number, length, index + 1);

    }

}

函数PrintNumber和前面第二种思路中的一样，这里就不重复了。对比这两种思路，我们可以发现，递归能够用很简洁的代码来解决问题。

**程序员面试题精选100题(40)-扑克牌的顺子[算法]**

题目：从扑克牌中随机抽5张牌，判断是不是一个顺子，即这5张牌是不是连续的。2-10为数字本身，A为1，J为11，Q为12，K为13，而大小王可以看成任意数字。

         分析：这题目很有意思，是一个典型的寓教于乐的题目。

         我们需要把扑克牌的背景抽象成计算机语言。不难想象，我们可以把5张牌看成由5个数字组成的数组。大小王是特殊的数字，我们不妨把它们都当成0，这样和其他扑克牌代表的数字就不重复了。

         接下来我们来分析怎样判断5个数字是不是连续的。最直观的是，我们把数组排序。但值得注意的是，由于0可以当成任意数字，我们可以用0去补满数组中的空缺。也就是排序之后的数组不是连续的，即相邻的两个数字相隔若干个数字，但如果我们有足够的0可以补满这两个数字的空缺，这个数组实际上还是连续的。举个例子，数组排序之后为{0，1，3，4，5}。在1和3之间空缺了一个2，刚好我们有一个0，也就是我们可以它当成2去填补这个空缺。

         于是我们需要做三件事情：把数组排序，统计数组中0的个数，统计排序之后的数组相邻数字之间的空缺总数。如果空缺的总数小于或者等于0的个数，那么这个数组就是连续的；反之则不连续。最后，我们还需要注意的是，如果数组中的非0数字重复出现，则该数组不是连续的。换成扑克牌的描述方式，就是如果一副牌里含有对子，则不可能是顺子。

         基于这个思路，我们可以写出如下的代码：

// Determine whether numbers in an array are continuous

// Parameters: numbers: an array, each number in the array is between

//             0 and maxNumber. 0 can be treeted as any number between

//             1 and maxNumber

//             maxNumber: the maximum number in the array numbers

bool IsContinuous(std::vector<int> numbers, int maxNumber)

{

    if(numbers.size() == 0 || maxNumber <=0)

        return false;

    // Sort the array numbers.

    std::sort(numbers.begin(), numbers.end());

    int numberOfZero = 0;

    int numberOfGap = 0;

    // how many 0s in the array?

    std::vector<int>::iterator smallerNumber = numbers.begin();

    while(smallerNumber != numbers.end() && \*smallerNumber == 0)

    {

        numberOfZero++;

        ++smallerNumber;

    }

    // get the total gaps between all adjacent two numbers

    std::vector<int>::iterator biggerNumber = smallerNumber + 1;

    while(biggerNumber < numbers.end())

    {

        // if any non-zero number appears more than once in the array,

        // the array can't be continuous

        if(\*biggerNumber == \*smallerNumber)

            return false;

        numberOfGap += \*biggerNumber - \*smallerNumber - 1;

        smallerNumber = biggerNumber;

        ++biggerNumber;

    }

    return (numberOfGap > numberOfZero) ? false : true;

}

         本文为了让代码显得比较简洁，上述代码用C++的标准模板库中的vector来表达数组，同时用函数sort排序。当然我们可以自己写排序算法。为了有更好的通用性，上述代码没有限定数组的长度和允许出现的最大数字。要解答原题，我们只需要确保传入的数组的长度是5，并且maxNumber为13即可。

**程序员面试题精选100题(41)-把数组排成最小的数[算法]**

题目：输入一个正整数数组，将它们连接起来排成一个数，输出能排出的所有数字中最小的一个。例如输入数组{32, 321}，则输出这两个能排成的最小数字32132。请给出解决问题的算法，并证明该算法。

分析：这是09年6月份百度新鲜出炉的一道面试题，从这道题我们可以看出百度对应聘者在算法方面有很高的要求。

这道题其实是希望我们能找到一个排序规则，根据这个规则排出来的数组能排成一个最小的数字。要确定排序规则，就得比较两个数字，也就是给出两个数字m和n，我们需要确定一个规则m和n哪个更大，而不是仅仅只是比较这两个数字的数值哪个更大。

根据题目的要求，两个数字m和n排成的数字mn和nm，如果mn<nm，那么我们应该输出mn，也就是m应该排在n的前面，也就是m小于n；反之，如果nm<mn，n小于m。如果mn==mn，m等于n。

接下来我们考虑怎么去拼接数字，即给出数字m和n，怎么得到数字mn和nm并比较它们的大小。直接用数值去计算不难办到，但需要考虑到的一个潜在问题是m和n都在int能表达的范围内，但把它们拼起来的数字mn和nm就不一定能用int表示了。所以我们需要解决大数问题。一个非常直观的方法就是把数字转换成字符串。

另外，由于把数字m和n拼接起来得到的mn和nm，它们所含有的数字的个数肯定是相同的。因此比较它们的大小只需要按照字符串大小的比较规则就可以了。

基于这个思路，我们可以写出下面的代码：

// Maxinum int number has 10 digits in decimal system

const int g\_MaxNumberLength = 10;

// String buffers to combine two numbers

char\* g\_StrCombine1 = new char[g\_MaxNumberLength \* 2 + 1];

char\* g\_StrCombine2 = new char[g\_MaxNumberLength \* 2 + 1];

// Given an array, print  the minimum number

// by combining all numbers in the array

void PrintMinNumber(int\* numbers, int length)

{

    if(numbers == NULL || length <= 0)

        return;

    // Convert all numbers as strings

    char\*\* strNumbers = (char\*\*)(new int[length]);

    for(int i = 0; i < length; ++i)

    {

        strNumbers[i] = new char[g\_MaxNumberLength + 1];

        sprintf(strNumbers[i], "%d", numbers[i]);

    }

    // Sort all strings according to algorithm in function compare

    qsort(strNumbers, length, sizeof(char\*), compare);

    for(int i = 0; i < length; ++i)

        printf("%s", strNumbers[i]);

    printf("\n");

    for(int i = 0; i < length; ++i)

        delete[] strNumbers[i];

    delete[] strNumbers;

}

// Compare two numbers in strNumber1 and strNumber2

// if [strNumber1][strNumber2] > [strNumber2][strNumber1],

// return value > 0

// if [strNumber1][strNumber2] = [strNumber2][strNumber1],

// return value = 0

// if [strNumber1][strNumber2] < [strNumber2][strNumber1],

// return value < 0

int compare(const void\* strNumber1, const void\* strNumber2)

{

    // [strNumber1][strNumber2]

    strcpy(g\_StrCombine1, \*(const char\*\*)strNumber1);

    strcat(g\_StrCombine1, \*(const char\*\*)strNumber2);

    // [strNumber2][strNumber1]

    strcpy(g\_StrCombine2, \*(const char\*\*)strNumber2);

    strcat(g\_StrCombine2, \*(const char\*\*)strNumber1);

    return strcmp(g\_StrCombine1, g\_StrCombine2);

}

上述代码中，我们在函数compare中定义比较规则，并根据该规则用库函数qsort排序。最后把排好序的数组输出，就得到了根据数组排成的最小的数字。

找到一个算法解决这个问题，不是一件容易的事情。但更困难的是我们需要证明这个算法是正确的。接下来我们来试着证明。

首先我们需要证明之前定义的比较两个数字大小的规则是有效的。一个有效的比较需要三个条件：1.自反性，即a等于a；2.对称性，即如果a大于b，则b小于a；3.传递性，即如果a小于b，b小于c，则a小于c。现在分别予以证明。

1.

自反性。显然有aa=aa，所以a=a。

2.       对称性。如果a小于b，则ab<ba，所以ba>ab。因此b大于a。

3.       传递性。如果a小于b，则ab<ba。当a和b用十进制表示的时候分别为l位和m位时，ab=a×10m+b，ba=b×10l+a。所以a×10m+b<b×10l+a。于是有a×10m-a< b×10l –b，即a(10m -1)<b(10l -1)。所以a/(10l -1)<b/(10m -1)。

如果b小于c，则bc<cb。当c表示成十进制时为m位。和前面证明过程一样，可以得到b/(10m -1)<c/(10n -1)。

所以a/(10l -1)< c/(10n -1)。于是a(10n -1)<c(10l -1)，所以a×10n +c<c×10l +a，即ac<ca。

所以a小于c。

在证明了我们排序规则的有效性之后，我们接着证明算法的正确性。我们用反证法来证明。

我们把n个数按照前面的排序规则排好顺序之后，表示为A1A2A3…An。我们假设这样排出来的两个数并不是最小的。即至少存在两个x和y（0<x<y<n），交换第x个数和地y个数后，A1A2…Ay…Ax…An<A1A2…Ax…Ay…An。

由于A1A2…Ax…Ay…An是按照前面的规则排好的序列，所以有Ax<Ax+1<Ax+2<…<Ay-2<Ay-1<Ay。

由于Ay-1小于Ay，所以Ay-1Ay<AyAy-1。我们在序列A1A2…Ax…Ay-1Ay…An交换Ay-1和Ay，有A1A2…Ax…Ay-1Ay…An<A1A2…Ax…AyAy-1…An（这个实际上也需要证明。感兴趣的读者可以自己试着证明）。我们就这样一直把Ay和前面的数字交换，直到和Ax交换为止。于是就有A1A2…Ax…Ay-1Ay…An<A1A2…Ax…AyAy-1…An< A1A2…Ax…AyAy-2Ay-1…An<…< A1A2…AyAx…Ay-2Ay-1…An。

同理由于Ax小于Ax+1，所以AxAx+1<Ax+1Ax。我们在序列A1A2…AyAxAx+1…Ay-2Ay-1…An仅仅只交换Ax和Ax+1，有A1A2…AyAxAx+1…Ay-2Ay-1…An<A1A2…AyAx+1Ax…Ay-2Ay-1…An。我们接下来一直拿Ax和它后面的数字交换，直到和Ay-1交换为止。于是就有A1A2…AyAxAx+1…Ay-2Ay-1…An<A1A2…AyAx+1Ax…Ay-2Ay-1…An<…< A1A2…AyAx+1Ax+2…Ay-2Ay-1Ax…An。

所以A1A2…Ax…Ay…An< A1A2…Ay…Ax…An。这和我们的假设的A1A2…Ay…Ax…An <A1A2…Ax…Ay…An相矛盾。

所以假设不成立，我们的算法是正确的。

**程序员面试题精选100题(43)-n个骰子的点数[算法]**

题目：把n个骰子扔在地上，所有骰子朝上一面的点数之和为S。输入n，打印出S的所有可能的值出现的概率。

分析：玩过麻将的都知道，骰子一共6个面，每个面上都有一个点数，对应的数字是1到 6之间的一个数字。所以，n个骰子的点数和的最小值为n，最大值为6n。因此，一个直观的思路就是定义一个长度为6n-n的数组，和为S的点数出现的次数保存到数组第S-n个元素里。另外，我们还知道n个骰子的所有点数的排列数6^n。一旦我们统计出每一点数出现的次数之后，因此只要把每一点数出现的次数除以n^6，就得到了对应的概率。

该思路的关键就是统计每一点数出现的次数。要求出n个骰子的点数和，我们可以先把n个骰子分为两堆：第一堆只有一个，另一个有n-1个。单独的那一个有可能出现从1到6的点数。我们需要计算从1到6的每一种点数和剩下的n-1个骰子来计算点数和。接下来把剩下的n-1个骰子还是分成两堆，第一堆只有一个，第二堆有n-2个。我们把上一轮那个单独骰子的点数和这一轮单独骰子的点数相加，再和剩下的n-2个骰子来计算点数和。分析到这里，我们不难发现，这是一种递归的思路。递归结束的条件就是最后只剩下一个骰子了。

基于这种思路，我们可以写出如下代码：

int g\_maxValue = 6;

void PrintSumProbabilityOfDices\_1(int number)

{

    if(number < 1)

        return;

    int maxSum = number \* g\_maxValue;

    int\* pProbabilities = new int[maxSum - number + 1];

    for(int i = number; i <= maxSum; ++i)

        pProbabilities[i - number] = 0;

    SumProbabilityOfDices(number, pProbabilities);

    int total = pow((float)g\_maxValue, number);

    for(int i = number; i <= maxSum; ++i)

    {

        float ratio = (float)pProbabilities[i - number] / total;

        printf("%d: %f\n", i, ratio);

    }

    delete[] pProbabilities;

}

void SumProbabilityOfDices(int number, int\* pProbabilities)

{

    for(int i = 1; i <= g\_maxValue; ++i)

        SumProbabilityOfDices(number, number, i, 0, pProbabilities);

}

void SumProbabilityOfDices(int original, int current, int value, int tempSum, int\* pProbabilities)

{

    if(current == 1)

    {

        int sum = value + tempSum;

        pProbabilities[sum - original]++;

    }

    else

    {

        for(int i = 1; i <= g\_maxValue; ++i)

        {

            int sum = value + tempSum;

            SumProbabilityOfDices(original, current - 1, i, sum, pProbabilities);

        }

    }

}

上述算法当number比较小的时候表现很优异。但由于该算法基于递归，它有很多计算是重复的，从而导致当number变大时性能让人不能接受。关于递归算法的性能讨论，详见[本博客系列的第16题](http://zhedahht.blog.163.com/blog/static/25411174200722991933440/)。

我们可以考虑换一种思路来解决这个问题。我们可以考虑用两个数组来存储骰子点数每一总数出现的次数。在一次循环中，第一个数组中的第n个数字表示骰子和为n出现的次数。那么在下一循环中，我们加上一个新的骰子。那么此时和为n的骰子出现的次数，应该等于上一次循环中骰子点数和为n-1、n-2、n-3、n-4、n-5与n-6的总和。所以我们把另一个数组的第n个数字设为前一个数组对应的第n-1、n-2、n-3、n-4、n-5与n-6之和。基于这个思路，我们可以写出如下代码：

void PrintSumProbabilityOfDices\_2(int number)

{

    double\* pProbabilities[2];

    pProbabilities[0] = new double[g\_maxValue \* number + 1];

    pProbabilities[1] = new double[g\_maxValue \* number + 1];

    for(int i = 0; i < g\_maxValue \* number + 1; ++i)

    {

        pProbabilities[0][i] = 0;

        pProbabilities[1][i] = 0;

    }

    int flag = 0;

    for (int i = 1; i <= g\_maxValue; ++i)

        pProbabilities[flag][i] = 1;

    for (int k = 2; k <= number; ++k)

    {

        for (int i = k; i <= g\_maxValue \* k; ++i)

        {

            pProbabilities[1 - flag][i] = 0;

            for(int j = 1; j <= i && j <= g\_maxValue; ++j)

                pProbabilities[1 - flag][i] += pProbabilities[flag][i - j];

        }

        flag = 1 - flag;

    }

    double total = pow((double)g\_maxValue, number);

    for(int i = number; i <= g\_maxValue \* number; ++i)

    {

        double ratio = pProbabilities[flag][i] / total;

        printf("%d: %f\n", i, ratio);

    }

    delete[] pProbabilities[0];

    delete[] pProbabilities[1];

}

    值得提出来的是，上述代码没有在函数里把一个骰子的最大点数硬编码(hard code)为6，而是用一个变量g\_maxValue来表示。这样做的好处时，如果某个厂家生产了最大点数为4或者8的骰子，我们只需要在代码中修改一个地方，扩展起来很方便。如果在面试的时候我们能对面试官提起对程序扩展性的考虑，一定能给面试官留下一个很好的印象。

**程序员面试题精选100题(44)－数值的整数次方[算法]**

题目：实现函数double Power(double base, int exponent)，求base的exponent次方。不需要考虑溢出。

分析：这是一道看起来很简单的问题。可能有不少的人在看到题目后30秒写出如下的代码：

double Power(double base, int exponent)

{

      double result = 1.0;

      for(int i = 1; i <= exponent; ++i)

            result \*= base;

      return result;

}

上述代码至少有一个问题：由于输入的exponent是个int型的数值，因此可能为正数，也可能是负数。上述代码只考虑了exponent为正数的情况。

接下来，我们把代码改成：

bool g\_InvalidInput = false;

double Power(double base, int exponent)

{

    g\_InvalidInput = false;

    if(IsZero(base) && exponent < 0)

    {

        g\_InvalidInput = true;

        return 0.0;

    }

    unsigned int unsignedExponent = static\_cast<unsigned int>(exponent);

    if(exponent < 0)

        unsignedExponent = static\_cast<unsigned int>(-exponent);

    double result = PowerWithUnsignedExponent(base, unsignedExponent);

    if(exponent < 0)

        result = 1.0 / result;

    return result;

}

double PowerWithUnsignedExponent(double base, unsigned int exponent)

{

      double result = 1.0;

      for(int i = 1; i <= exponent; ++i)

            result \*= base;

      return result;

}

上述代码较之前的代码有了明显的改进，主要体现在：首先它考虑了exponent为负数的情况。其次它还特殊处理了当底数base为0而指数exponent为负数的情况。如果没有特殊处理，就有可能出现除数为0的情况。这里是用一个全局变量来表示无效输入。关于不同方法来表示输入无效的讨论，[详见本系列第17题](http://zhedahht.blog.163.com/blog/static/25411174200731139971)。

最后需要指出的是：由于0^0次方在数学上是没有意义的，因此无论是输入0还是1都是可以接受的，但需要在文档中说明清楚。

这次的代码在逻辑上看起来已经是很严密了，那是不是意味了就没有进一步改进的空间了呢？接下来我们来讨论一下它的性能：

如果我们输入的指数exponent为32，按照前面的算法，我们在函数PowerWithUnsignedExponent中的循环中至少需要做乘法31次。但我们可以换一种思路考虑：我们要求出一个数字的32次方，如果我们已经知道了它的16次方，那么只要在16次方的基础上再平方一次就可以了。而16次方是8次方的平方。这样以此类推，我们求32次方只需要做5次乘法：求平方，在平方的基础上求4次方，在4次方的基础上平方求8次方，在8次方的基础上求16次方，最后在16次方的基础上求32次方。

32刚好是2的整数次方。如果不巧输入的指数exponent不是2的整数次方，我们又该怎么办呢？我们换个数字6来分析，6就不是2的整数次方。但我们注意到6是等于2+4，因此我们可以把一个数的6次方表示为该数的平方乘以它的4次方。于是，求一个数的6次方需要3次乘法：第一次求平方，第二次在平方的基础上求4次方，最后一次把平方的结果和4次方的结果相乘。

现在把上面的思路总结一下：把指数分解了一个或若干个2的整数次方。我们可以用连续平方的方法得到以2的整数次方为指数的值，接下来再把每个前面得到的值相乘就得到了最后的结果。

到目前为止，我们还剩下一个问题：如何将指数分解为一个或若干个2的整数次方。我们把指数表示为二进制数再来分析。比如6的二进制表示为110，在它的第2位和第3位为1，因此6=2^(2-1)+2^(3-1) 。也就是说只要它的第n位为1，我们就加上2的n-1次方。

最后，我们根据上面的思路，重写函数PowerWithUnsignedExponent：

double PowerWithUnsignedExponent(double base, unsigned int exponent)

{

    std::bitset<32> bits(exponent);

    if(bits.none())

        return 1.0;

    int numberOf1 = bits.count();

    double multiplication[32];

    for(int i = 0; i < 32; ++i)

    {

        multiplication[i] = 1.0;

    }

    // if the i-th bit in exponent is 1,

    // the i-th number in array multiplication is base ^ (2 ^ n)

    int count = 0;

    double power = 1.0;

    for(int i = 0; i < 32 && count < numberOf1; ++i)

    {

        if(i == 0)

            power = base;

        else

            power = power \* power;

        if(bits.at(i))

        {

            multiplication[i] = power;

            ++count;

        }

    }

    power = 1.0;

    for(int i = 0; i < 32; ++i)

    {

        if(bits.at(i))

            power \*= multiplication[i];

    }

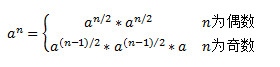
    return power;

}

       在上述代码中，我们用C++的标准函数库中bitset把整数表示为它的二进制，增大代码的可读性。如果exponent的第i位为1，那么在数组multiplication的第i个数字中保存以base为底数，以2的i次方为指数的值。最后，我们再把所以位为1在数组中的对应的值相乘得到最后的结果。

上面的代码需要我们根据base的二进制表达的每一位来确定是不是需要做乘法。对二进制的操作很多人都不是很熟悉，因此编码可能觉得有些难度。我们可以换一种思路考虑：我们要求出一个数字的32次方，如果我们已经知道了它的16次方，那么只要在16次方的基础上再平方一次就可以了。而16次方是8次方的平方。这样以此类推，我们求32次方只需要做5次乘法：先求平方，在平方的基础上求4次方，在4次方的基础上平方求8次方，在8次方的基础上求16次方，最后在16次方的基础上求32次方。

也就是说，我们可以用如下公式求a的n次方：



这个公式很容易就能用递归来实现。新的PowerWithUnsignedExponent代码如下：

double PowerWithUnsignedExponent(double base, unsigned int exponent)

{

    if(exponent == 0)

        return 1;

    if(exponent == 1)

        return base;

    double result = PowerWithUnsignedExponent(base, exponent >> 1);

    result \*= result;

    if(exponent & 0x1 == 1)

        result \*= base;

    return result;

}

**程序员面试题精选100题(55)-不用＋、－、×、÷做加法[算法]**

**题目：写一个函数，求两个整数的之和，要求在函数体内不得使用＋、－、×、÷。**

分析：这又是一道考察发散思维的很有意思的题目。当我们习以为常的东西被限制使用的时候，如何突破常规去思考，就是解决这个问题的关键所在。

看到的这个题目，我的第一反应是傻眼了，四则运算都不能用，那还能用什么啊？可是问题总是要解决的，只能打开思路去思考各种可能性。首先我们可以分析人们是如何做十进制的加法的，比如是如何得出5+17=22这个结果的。实际上，我们可以分成三步的：第一步只做各位相加不进位，此时相加的结果是12（个位数5和7相加不要进位是2，十位数0和1相加结果是1）；第二步做进位，5+7中有进位，进位的值是10；第三步把前面两个结果加起来，12+10的结果是22，刚好5+17=22。

前面我们就在想，求两数之和四则运算都不能用，那还能用什么啊？对呀，还能用什么呢？对数字做运算，除了四则运算之外，也就只剩下位运算了。位运算是针对二进制的，我们也就以二进制再来分析一下前面的三步走策略对二进制是不是也管用。

5的二进制是101，17的二进制10001。还是试着把计算分成三步：第一步各位相加但不计进位，得到的结果是10100（最后一位两个数都是1，相加的结果是二进制的10。这一步不计进位，因此结果仍然是0）；第二步记下进位。在这个例子中只在最后一位相加时产生一个进位，结果是二进制的10；第三步把前两步的结果相加，得到的结果是10110，正好是22。由此可见三步走的策略对二进制也是管用的。

接下来我们试着把二进制上的加法用位运算来替代。第一步不考虑进位，对每一位相加。0加0与 1加1的结果都0，0加1与1加0的结果都是1。我们可以注意到，这和异或的结果是一样的。对异或而言，0和0、1和1异或的结果是0，而0和1、1和0的异或结果是1。接着考虑第二步进位，对0加0、0加1、1加0而言，都不会产生进位，只有1加1时，会向前产生一个进位。此时我们可以想象成是两个数先做位与运算，然后再向左移动一位。只有两个数都是1的时候，位与得到的结果是1，其余都是0。第三步把前两个步骤的结果相加。如果我们定义一个函数AddWithoutArithmetic，第三步就相当于输入前两步骤的结果来递归调用自己。

有了这些分析之后，就不难写出如下的代码了：

int AddWithoutArithmetic(int num1, int num2)

{

        if(num2 == 0)

                return num1;

        int sum = num1 ^ num2;

        int carry = (num1 & num2) << 1;

        return AddWithoutArithmetic(sum, carry);

}

                之前我的系列博客中有这么一道题，求1+2+…+n，要求不能使用乘除法、for、while、if、else、switch、case等关键字以及条件判断语句（A?B:C）。刚兴趣的读者，可以到<http://zhedahht.blog.163.com/blog/static/2541117420072915131422/>看看。