《计算概论A》课程 程序设计部分

习题讲解（1）

李　戈

北京大学 信息科学技术学院 软件研究所

lige@sei.pku.edu.cn

议题1：大整数很罗嗦

大整数乘法

- 问题描述
  - 请编写一个程序帮助统计局完成以下计算任务：

    从键盘输入两个正整数m和n（根据统计需要，m和n最多可以是200位十进制正整数），计算并m和n的积，并打印输出。

大整数乘法

- 基本运算过程

‘8’	‘3’	‘5’					

‘4’	‘9’						

					8	3	5

×

						4	9

					72	27	45
				32	12	20	
				32	84	47	45
			4	0	9	1	5

大整数乘法

- 为了写程序方便

‘8’	‘3’	‘5’					

‘4’	‘9’						

5	3	8					

×

9	4						

45	27	72					

	20	12	32				

45	47	84	32				

5	1	9	0	4			

大整数乘法

- 为了写程序方便

‘8’	‘3’	‘5’					

‘4’	‘9’						

5	3	8					

×

9	4						

45	27	72					
	20	12	32				
45	47	84	32				
5	1	9	0	4			

```
for(i = 0; i < nLen2; i++)  //对应49
   for(j = 0; j < nLen1; j++) //对应835
     anResult[i+j]
        = anResult[i+j] + an2[i] * an1[j];
```

北京大學

大整数乘法

- 为了写程序方便

'8'	'3'	'5'					

'4'	'9'						

⇨

5	3	8					

×

9	4						

45	27	72					

	20	12	32				

45	47	84	32				

5	1	9	0	4			

- 需要做的事情：
  - 定义两个字符数组用来输入大数；
  - 定义两个整数数组用来放转换后的整数；
  - 把字符数组中的值转换到整数数组；
  - 用一个嵌套循环实现逐位相乘；
  - 用一个循环处理进位；
  - 把结果输出出来；

（1）定义两个字符数组用来输入大数；

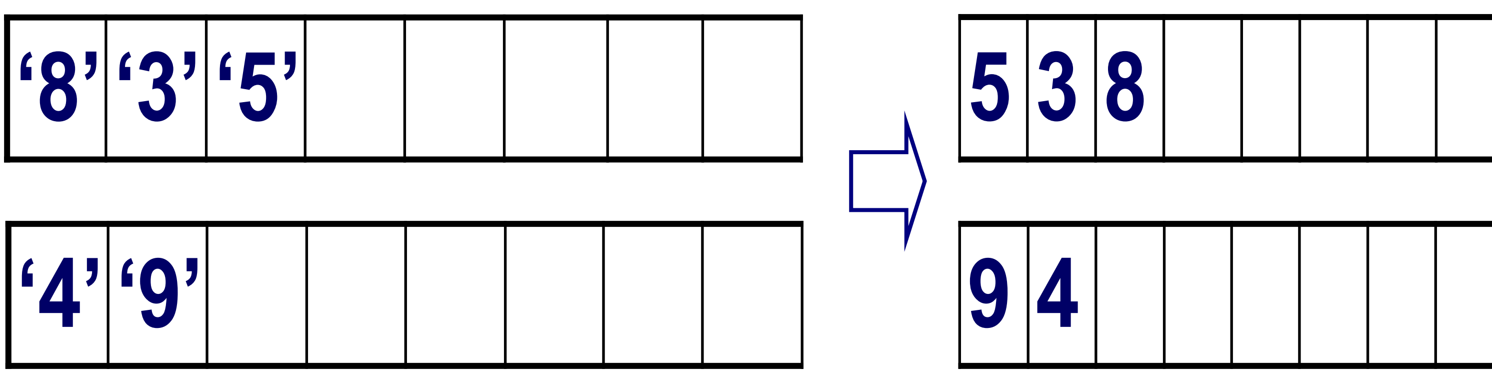
（2）定义两个整数数组用来放转换后的整数；

```
int main()
{  const int MAX_LEN = 200;
   char seLine1[MAX_LEN ], seLine2[MAX_LEN ];
   int an1[MAX_LEN], an2[MAX_LEN];
   int anResult[MAX_LEN * 2] ;
   memset(an1,0,sizeof(an1));
   memset(an2,0,sizeof(an2));
   memset(anResult,0,sizeof(anResult));
   cout << "please input two integers" << endl;
   cin.getline(seLine1, MAX_LEN + 1);
   cin.getline(seLine2, MAX_LEN + 1);
```

（3）把字符数组中的值转换到整数数组，并倒置；

```
int nLen1 = strlen(seLine1);
int nLen2 = strlen(seLine2);

int i, j=0;
for (i = nLen1-1; i>=0;i--)
    an1[j++] = seLine1[i] - '0';
j=0;
for (i = nLen2-1; i>=0;i--)
    an2[j++] = seLine2[i] - '0';
```



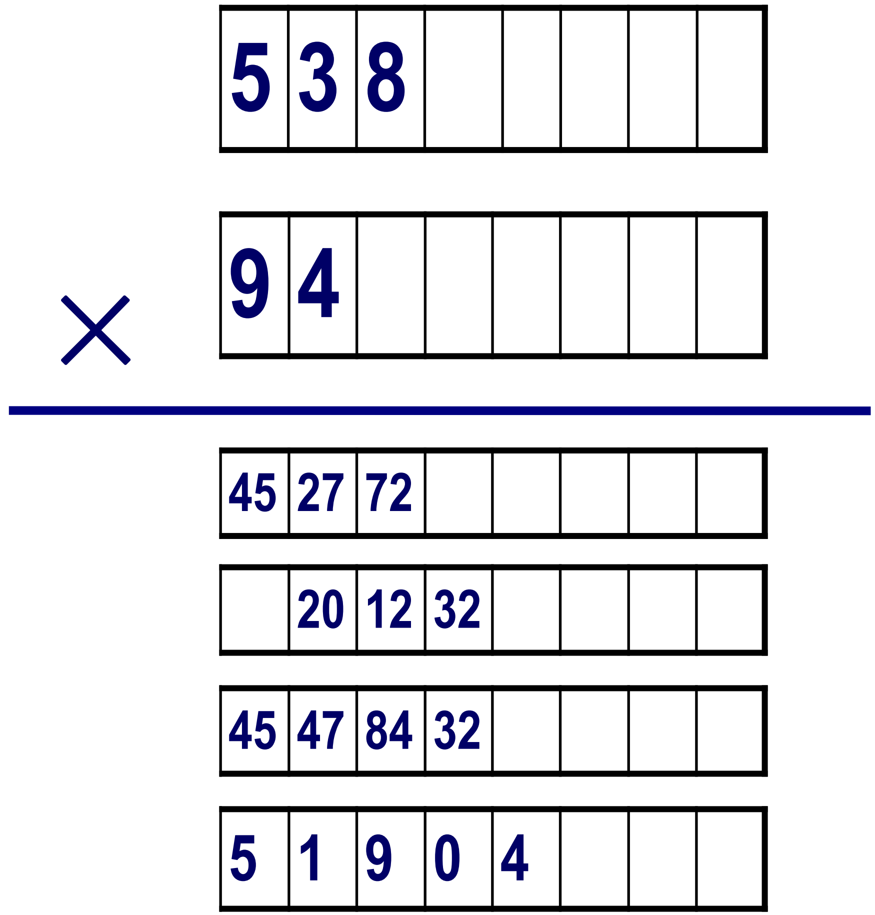
（3）用一个嵌套循环实现逐位相乘；

```
for(i = 0; i < nLen2; i++)
    for(j = 0; j < nLen1; j++)
        anResult[i+j]
            = anResult[i+j] + an2[i] * an1[j];
```

	5	3	8					
×	9	4						
	45	27	72					
		20	12	32				
	45	47	84	32				

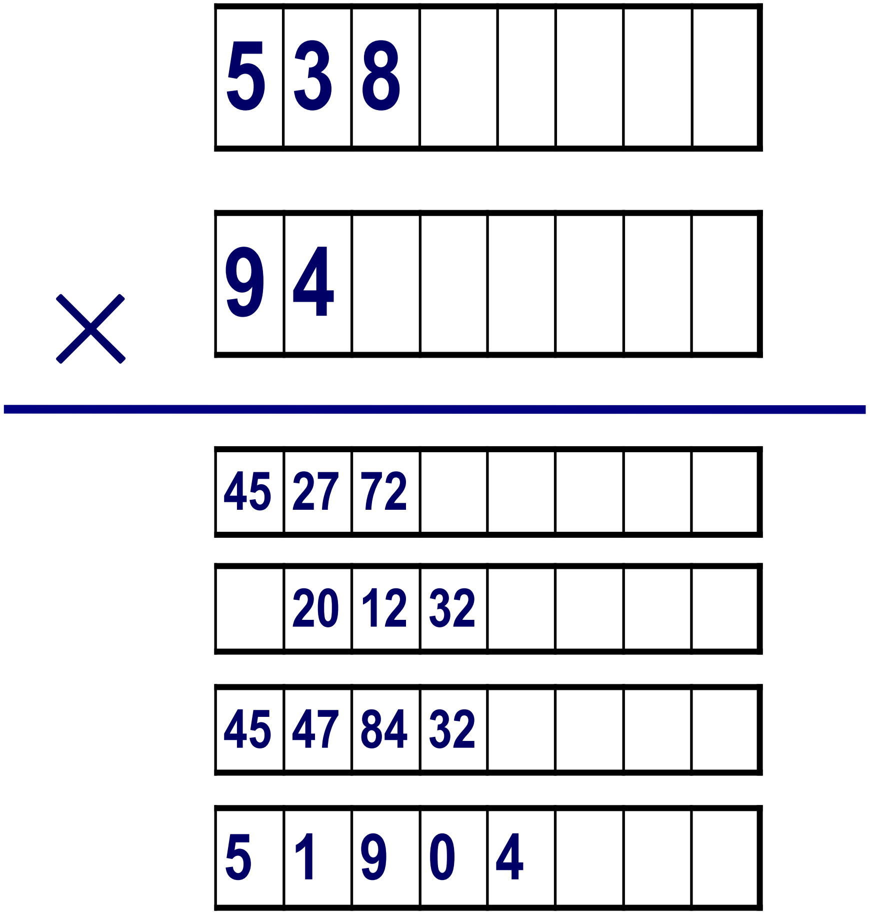
（4）用一个循环处理进位；

```
for (i = 0; i < MAX_LEN * 2; i++)
    if(anResult[i] >= 10)
    {
      anResult[i+1] = anResult[i+1] + anResult[i] / 10;
      anResult[i] %= 10;
    }
```



（5）输出结果；

```cpp
i = MAX_LEN * 2 - 1;
while(anResult[i]==0) i--;          //找到第一个不为0的位
for(;i >= 0; i--)
    cout << anResult[i];            //输出每一位数
cout << endl;
}
```



参考程序：

```
int main()
{  const int MAX_LEN = 200;
   unsigned an1[MAX_LEN];unsigned an2[MAX_LEN];
   char seLine1[MAX_LEN + 1];char seLine2[MAX_LEN + 1];
   unsigned anResult[MAX_LEN * 2] ;
   cout << "please input two integers" << endl;
   cin.getline(seLine1, MAX_LEN + 1);
   cin.getline(seLine2, MAX_LEN + 1);
   int nLen1 = strlen(seLine1);
   int nLen2 = strlen(seLine2);
   memset(an1,0,sizeof(an1));
   memset(an2,0,sizeof(an2));
   memset(anResult,0,sizeof(anResult));
```

```
int i,j=0;
for (i = nLen1-1; i>=0;i--)
    an1[j++] = seLine1[i] - '0'; //将字符数组变成整数数组，并倒置
j=0;
for (i = nLen2-1; i>=0;i--)
    an2[j++] = seLine2[i] - '0';
for(i = 0; i < nLen2; i++)
    for(j = 0; j < nLen1; j++)
      anResult[i+j] += an2[i] * an1[j]; //个位和十位相乘等于十位
for (i = 0; i < MAX_LEN * 2;i++)
    if(anResult[i] >= 10)
    {
      anResult[i+1] += anResult[i] / 10; //处理进位，每位只有
      anResult[i] %= 10; //一个数，两位数就要进位
    }
i = MAX_LEN * 2 - 1;
while(anResult[i]==0) i--; //找到第一个不为0的位
for(;i >= 0; i--)
    cout << anResult[i]; //输出每一位数
cout << endl;}
```

好罗嗦啊。。

大整数加法

- 分析：
  - 用字符数组表达长整数：

```
char num[30] =
"123456789012345678901234567890";
int num[30] =
{1,2,3,4,……,2,3,4,5,6,7,8,9,0};
```

- 有没有什么办法可减少计算次数？

北京大學

10进制 vs. 10000进制

- 10进制：
  - int num[30]={1,2,3,4,……,2,3,4,5,6,7,8,9,0};
- 10000进制：
  - int num[8]={12,3456,……,9012,3456,7890};
- 10000 vs. 10
  - 更少的空间
  - 更少的运算次数

高精度整数

- 解决方案 2：
  - 可否把：

int num[30]

= {1,2,3,4,……,2,3,4,5,6,7,8,9,0};

  - 转换成：

int num[8]

= {12,3456,……,9012,3456,7890};

大整数加法

- 计算高精度数a+b，结果放在c中

```
void Add(int a[], int b[], int c[])
{
   int i, j = 0;          // j表示进位
   // 从低位向高位计算
   for (i = MAXL - 1; i >= 0; i --)
   {
        c[i] = (a[i] + b[i] + j);
        j = c[i] / 10000;       // 超过10000的部分是进位
        c[i] = c[i] % 10000;
   }
   assert(0 == j);    // 断言，如果还有进位说明数组太小
}
```

大整数减法

```
void Sub(int a[], int b[], int c[])
{
    int i, j = 0;                      // j表示借位
    for (i = MAXL - 1; i >= 0; i --)
    {
        c[i] = (a[i] - b[i] - j);
        if (c[i] >= 0)                 // 不需要借位
            j = 0;
        else                           // 向高位借位
            j = 1, c[i] = c[i] + 10000;
    }
    assert(0 == j);                    // 若有借位，说明a<b
}
```

大整数乘法

```
void Mul(int a[], int b[], int c[])    //a×b放入c
{   int i, j, k;
    for (i = MAXL - 1; i >= 0; i --)
    {     for (j = MAXL - 1; j >= 0; j --)
          {          k = i + j - (MAXL - 1);
                                        // 确定相乘结果在c中的位置
                     c[k] = c[k] + a[i] * b[j];
          }
    }
    j = 0;
    for (i = MAXL - 1; i >= 0; i --)              // 处理进位
    {
          c[i] += j, j = c[i] / 10000, c[i] = c[i] % 10000;
    }
    assert(0 == j);
}
```

北京大学

大整数除法

- 计算c mod k，除法结果替换c中的原有内容，并返回余数

```
int Mod(int c[], int k)
{
    int i, j, tmp;
    j = 0;                          // j为余数
    for (i = 0; i < MAXL; i ++) //从最高位到最低位
    {
        tmp = j * 10000 + c[i];     //高位留下的余数×位值+本位
        c[i] = tmp / k;       //计算当前位的商
        j = tmp % k;          //计算当前位的余数
    }
    return j;
}
```

大整数除法

```
void Mod(int a[], int b[], int c[])  //a/b=c
{
    int i, j, k;
    int tmp[MAXL];
    for (i = 0; i < MAXL; i ++)
    {
        for (j = 1; j < 10000; j ++)
        {
            Mul(b, j, tmp); // 计算除数与所试商的积
            if (Cmp(tmp, a, i) > 0)
            // 若b[0..MAXL-1] * j > a[0..i]则商j太大
                break;
        }
        c[i] = j - 1;     //记录第i位的商
        Mul(b, c[i], tmp);
        Sub(a, i, tmp); // 从被除数中减去商与乘数的积
    }
}
```

大整数运算

- 更充分地利用空间：
  - int num[8] = {12,3456,……,9012,3456,7890};
  - int num[4] = {123,456789012,345678901,234567890}
  - int num[3] = {1234567890,1234567890,1234567890}
- 记住：
  - 32位无符号整数：
    - $2^{32} - 1 = 4294967293$【最多能表示10位数】
  - 64位无符号double型数
    - $2^{64} - 1 = 18446744073709551615$【最多能表示20位数】

议题2：大素数很麻烦

大素数判定

- 问题
  - 有如下公式：$f(m)=2^m+1$, $m=2^n$. 求n=0,1,…,6 时输出值为质数的f(m).

$f(1) = 3$

$f(2) = 5$

$f(4) = 17$

$f(8) = 257$

$f(16)= 65537$

$f(32)= 4294967297$

$f(64)= 2^{64}+1= 18446744073709551617$（20位）

大素数判定

- 素数判定基本方法：试除法

```
for (i = 2; i * i <= N; i ++)     // 枚举约数
{
    if (N % i == 0)               // 判断能否整除
    {
        return false;
    }
}
return true;
```

问题分析

```
int n, m; __int64 fm, x;      // __int64为64位整数
for (n = 0; n <= 6; n ++)
{
    bool isPrime = true;
    m = (__int64)1 << n;            // m = 2^n
    fm = ((__int64)1 << m) + 1;     // f(m) = 2^m + 1
    for ( x = 2; x <= sqrt(fm); x ++)
    {
        if (fm % x == 0)
        {
            isPrime = false;
            break;
        }
    }
    /* ... */
}
```

北京大学

问题分析

```
int n, m; __int64 fm, x;      // __int64为64位整数
for (n = 0; n <= 6; n ++)
{
    bool isPrime = true;
    m = (__int64)1 << n;              // m = 2^n
    fm = ((__int64)1 << m) + 1;       // f(m) = 2^m + 1
    for ( x = 2; x < ((__int64)1<<(m/2)); x ++)
    {
        if (fm % x == 0)      // 需要判断(2^64+1)%x是否为0
        {
            isPrime = false;
            break;
        }
    }
    /* ... */
}
```

北京大學

问题分析

- 由于：
  - $(2^{64} + 1) \bmod x$

    $= (2^{64} \bmod x + 1) \bmod x$
- 因此：
  - 只需解决：

    如何计算$2^{64} \bmod x$的值？

$$2^{i} \bmod x = (2 * 2^{i-1} \bmod x) \bmod x$$

方法1

- 设$g(i)=2^i \bmod x$ 则有$g(i) = (2*g(i-1)) \bmod x$

```
__int64 Mod(__int64 x)
{
   int i;
   __int64 g;
   g = 1;                    // 初始值g(0)=1
   for (i = 1; i <= 64; i ++)
   {
       g = (2 * g) % x;   // 由g(i-1)递推g(i)
   }
   return g;
}
```

$$2^i \bmod x = (2^{i/2} \bmod x)^2 \bmod x$$

方法2

- 折半：g(i)=(g(i/2)*g(i/2)) mod x

```
__int64 Mod (__int64 x)
{
   __int64 g = 2 % x;            // g(1) = 2 mod x
   for (i = 2; i <= 64; i = i * 2)
   {
        g = (g * g) % x;
   }
   return g;
}
```

回到原来的问题

```
int n, m; __int64 fm, x;        // __int64为64位整数
for (n = 0; n <= 6; n ++)
{
    bool isPrime = true;
    m = (__int64)1 << n;                    // m = 2n
  //fm = ((__int64)1 << m) + 1;       // f(m) = 2m + 1
    for ( x = 2; x < ((__int64)1<<32); x ++)
    {
            if ((Mod(x) + 1)%x==0)          // 判断(264+1)%x是否为0
            {
                        isPrime = false;
                        break;
            }
    }
    /* ... */
}
```

大素数判定结果

- f(1) = 3 素数
- f(2) = 5 素数
- f(4) = 17 素数
- f(8) = 257 素数
- f(16)= 65537 素数
- f(32)= 4294967297= 641 * 6700417
- f(64)= 18446744073709551617 = 274177 * 67280421310721

总结与提问

- 经验总结
  - 如果计算过程中，可能产生突破边界的大数值结果，可以想办法通过调整算法，避免出现中间结果中的大数！

- 提问：
  - 对于本题而言，如果就是需要判定某个大数是否为素数，怎么办？

再论素数的判定

- 当n>=6时，$f(2^n)$是否为合数?
- 有没有更快的办法?
  - Rabin-Miller素数测试
    - 基本思想：Fermat小定理
    - 基本方法：概率算法
  - 利用概率降低时间复杂度

素数判定

- Fermat小定理
  - 如果p是一个素数，则对任意a有
    - $a^p \equiv a \pmod p$
  - 特别的，如果p不能整除a，则还有
    - $a^{p-1} \equiv 1 \pmod p$
- 可知
  - 若p不满足上述条件，则p是个合数，否则p可能是一个素数；
- 说明：
  - 一般地，两个整数a和b，除以一个大于1的自然数m所得的余数相同，就称a和b对于模m同余或a和b在模m下同余，记为：$a \equiv b \pmod m$

素数判定

- **Rabin-Miller素数测试**
  - 取2, 3, 5, 7, 11, 13等前9个素数作为基a，若p都能够通过测试($a^{p-1}\equiv 1 \pmod p$)，则p是合数的概率非常小($p<=2^{64}$时,概率不超过$1/2^{18}$)，从概率角度，可以认为p是一个素数。
  - 测试所取的基越多，p为合数的概率越小
- **因此，可以采取以下方法进行测试：**
  - 对不同的基a，求$a^{p-1} \bmod p$的值，值都为1，则认为p是素数，否则p是合数

素数判定

- 如何求a^n mod p
  - 设 $f(n) = a^n \bmod p$
  - 若n为奇数，可以采用：
    - $f(n) = (a*f(n-1)) \bmod p$
  - 若n为偶数，可以采用：
    - $f(n)=(f(n/2)*f(n/2)) \bmod p$
  - 用递归求解

素数判定

```
__int64 PowerMod(__int64 a, __int64 n, __int64 p)
{
    if (n == 1)            // 递归边界：a1 mod p
        return a % p;
    if (n % 2 == 1)    // 奇数时的情况
        return a * PowerMod(a, n -1, p) % p;
                       // 偶数时的情况
    __int64  tmp = PowerMod(a, n/2, p);
    return   tmp * tmp % p;
}
```

素数判定

```
int i, prime[9] = {2, 3, 5, ……};
for (i = 0; i < 9; i ++)
{
   if (PowerMod(prime[i], p - 1) % p != 1)
   {
        return false;      // 测试失败，p是合数
   }
}
return true;               // p是素数
```

议题3：红与黑

红与黑

- 问题
  - 有一间长方形的房子，地上铺了红色、黑色两种颜色的正方形瓷砖。你站在其中一块黑色的瓷砖上，只能向相邻的黑色瓷砖移动。请写一个程序，计算你总共能够到达多少块黑色的瓷砖。

红与黑

- 输入
  - 包括多个数据集合。每个数据集合的第一行是两个整数W和H，分别表示x方向和y方向瓷砖的数量。W和H都不超过20。在接下来的H行中，每行包括W个字符。每个字符表示一块瓷砖的颜色，规则如下
    1）‘.’：黑色的瓷砖；
    2）‘#’：红色的瓷砖；
    3）‘@’：黑色的瓷砖，并且你站在这块瓷砖上。该字符在每个数据集合中唯一出现一次。
    当在一行中读入的是两个零时，表示输入结束。

红与黑

- 输出
  - 对每个数据集合，分别输出一行，显示你从初始位置出发能到达的瓷砖数(记数时包括初始位置的瓷砖)。

```
6 9
....#.
.....#
......
......
......
......
......
#@...#
.#..#.
11 9
.#.........
.#.#######.
.#.#.....#.
.#.#.###.#.
.#.#..@#.#.
.#.#####.#.
.#.......#.
.#########.
...........
0 0
```

输入的方法之一

```
#include <iostream>
using namespace std;
int main()
{
   while(1)
   {
        cin>>H>>W;
        if(W == 0 || H == 0) break;
        ……
   }
   return 0;
}
```

输入的方法之二

```
#include <iostream>
using namespace std;
int main()
{
    while(cin>>H && cin>>W &&W != 0 &&H != 0)
    {

    }
    return 0;
}
```

思路一

- “试走法”
  - 从指定位置开始走；
    - 向上
    - 向下
    - 向左
    - 向右
  - 每到一处，只要符合条件，标记为'$'
  - 不符合条件什么都不做
  - “走完”后数$

```
11 6
..#..#..#..
..#..#..#..
..#..#..###
..#..#..#@.
..#..#..#..
..#..#..#..
```

北京大学

```
void f(int x, int y)
{
if(x<0 || x>=W || y<0 || y>=H || z[x][y] == '#' || z[x][y]
        == '$') // 如果走出矩阵范围
        return;
 else
 {
        z[x][y] = '$'; // 将走过的瓷砖做标记
        f(x-1, y);
        f(x+1, y);
        f(x, y-1);
        f(x, y+1);
 }
}
```

```
#include <iostream>
using namespace std;
int W, H;
char z[21][21];
int main()
{
 int i, j, num, sum;
 while(cin>>H && cin>>W && W != 0 && H != 0)
 {
        num = 0;
        sum = 0;
        for(i = 0; i < W; i++) // 读入矩阵
              cin>>z[i];
        for(i = 0; i < W; i++)
              for(j = 0; j < H; j++)
                      if(z[i][j] == '@') f (i,j);
        for(i = 0; i < W; i++)
              for(j = 0; j < H; j++)
              {
                      if(z[i][j] == '$') sum++;
              }
        cout<<sum<<endl;

 }
 return 0;
}
```

思路二

- “直接法”
  - 求的是从某点能到达的瓷砖数；
  - 假设有个函数能直接返回这个数；
  - 那么这个函数该如何写呢？

    （1）这个函数的原型应该长成什么样？

    （2）求得结果的计算过程应该如何描述？

```
11 6
..#..#..#..
..#..#..#..
..#..#..###
..#..#..#@.
..#..#..#..
..#..#..#..
```

$$f(x, y) = 1 + f(x - 1, y) + f(x + 1, y) + f(x, y - 1) + f(x, y + 1)$$

北京大學

```
int f(int x, int y)
{
 if(x < 0 || x >= W || y < 0 || y >= H)
                                                  // 如果走出矩阵范围
       return 0;
 if(z[x][y] == '#')
       return 0;
 else
 {
       z[x][y] = '#';                             // 将走过的瓷砖做标记
       return 1 + f(x - 1, y) + f(x + 1, y) + f(x, y - 1) +
       f(x, y + 1);
 }
}
```

```
#include <iostream>
using namespace std;
int W, H;
char z[21][21];
int main()
{ int i, j, num;
 while(cin>>H && cin>>W && W != 0 && H != 0)
 {
       num = 0;
       for(i = 0; i < W; i++) // 读入矩阵
            cin>>z[i];
       for(i = 0; i < W; i++)
            for(j = 0; j < H; j++)
                     if(z[i][j] == '@') cout<<f (i , j)<<endl;
 }
 return 0;
}
```

数字三角形

- 问题描述
  - 有一个数字三角形。从三角形的顶部到底部有很多条不同的路径，路径上的每一步只能从一个数走到下一层上和它最近的左边的数或者右边的数。对于每条路径，把路径上面的数加起来可以得到一个和，和最大的路径称为最佳路径。请找出最佳路径上的数字之和。

```
    7
   3 8
  8 1 0
 2 7 4 4
4 5 2 6 5
```

例题1：数字三角形

- 是否可用贪心法?

- 贪心法
  - 每次选择下一层中较大的数字作为下一步；
  - 每次选择下两层中和较大的数字作为下一步；

- 能否获得最优解?

例题1：数字三角形

- 递归解法：
  - D(r, j)表示：第r 行第 j 个数字(r , j 都从1 开始算)；
  - MaxSum(r, j) 代表：从第 r 行的第 j 个数字到底边的最佳路径的数字之和，则本题是要求 MaxSum(1, 1) 。
  - 从某个D(r, j)出发，下一步只能走D(r+1, j)或者D(r+1, j+1)：
    - 如果走D(r+1, j)，那么得到的MaxSum(r, j)就是MaxSum(r+1, j) + D(r, j)；
    - 如果走D(r+1, j+1)，那么得到的MaxSum(r, j)就是MaxSum(r+1, j+1) + D(r, j)。
  - 选择往哪里走，就看MaxSum(r+1, j)和MaxSum(r+1, j+1)哪个更大了。

例题1：数字三角形

```cpp
#include <iostream>
using namespace std;
int D[100 + 10][100 + 10], N;
int main()
{
    int m;
    cin>>N;
    for( int i = 1; i <= N; i ++ )
        for( int j = 1; j <= i; j ++ )
            cin>>D[i][j];
    cout<<MaxSum(1, 1);
    return 0;
}
```

```cpp
int MaxSum( int r, int j)
{
    if( r == N )
            return D[r][j];
    int nSum1 = MaxSum(r+1, j);
    int nSum2 = MaxSum(r+1, j+1);
    if( nSum1 > nSum2 )
            return nSum1+D[r][j];
    return nSum2+D[r][j];
}
```

北京大学

例题1：数字三角形

- 递归的坏处
  - 每次计算MaxSum(r, j)的时候，都要计算一次MaxSum(r+1, j)；
  - 每次计算MaxSum(r, j+1)的时候，也要计算一次MaxSum(r+1, j)；
  - 重复计算因此产生！
  - 计算次数：

$$2^0 + 2^1 + 2^2 \ldots\ldots 2^{N-1} = 2^N$$

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

例题1：数字三角形

- 解决方案
  - 一个值一旦算出来，就要记住，不必重新计算！
  - 第一次算出MaxSum(r, j)的值时，就将该值存放起来，下次再需要计算MaxSum(r, j)时，直接取用存好的值即可，不必再次调用MaxSum 进行函数递归计算。
  - 这样，每个MaxSum(r, j)都只需要计算1 次即可，那么总的计算次数（即调用MaxSum 函数的次数）就是三角形中的数字总数，即：

$$1+2+3+\ldots\ldots N = N(N+1)/2$$

例题1：数字三角形

- 解法：
  - 用一个二维数组aMaxSum[N][N]来记录计算结果，aMaxSum[r][j]就存放MaxSum(r, j)的计算结果。
  - 下次再需要MaxSum(r, j)的值时，不必再调用MaxSum 函数，只需直接取aMaxSum[r][j]的值即可。

```
int MaxSum( int r, int j)
{         if( r == N )
                    return D[r][j];
          if( aMaxSum[r+1][j] == -1 ) //如果MaxSum(r+1, j)没有计算过
                    aMaxSum[r+1][j] = MaxSum(r+1, j);
          if( aMaxSum[r+1][j+1] == -1) //如果MaxSum(r+1, j+1)没有计算过
                    aMaxSum[r+1][j+1] = MaxSum(r+1, j+1);
          if( aMaxSum[r+1][j] > aMaxSum[r+1][j+1] )
                    return aMaxSum[r+1][j] +D[r][j];
          return  aMaxSum[r+1][j+1] + D[r][j];
}
```

例题1：数字三角形

- 动态规划
  - 这种将一个问题分解为子问题递归求解，并且将中间结果保存以避免重复计算的办法，就叫做“动态规划”。
  - 能用动态规划解决的求最优解问题，必须满足，最优解的每个局部解也都是最优的。
  - 以上题为例：最佳路径上面的每个数字到底部的那一段路径，都是从该数字出发到达到底部的最佳路径。

例题1：数字三角形

- 进一步的调整
  - 既然每一步都记录了计算结果，可否通过递推的方式改写程序：

$$
anMaxSum[r][j] = \begin{cases} D[r][j] & r = N \\ Max(anMaxSum[r+1][j], anMaxSum[r+1][j+1]) + D[r][j] & \text{其他情况} \end{cases}
$$

```
#include <stdio.h>
#include <memory.h>
#define MAX_NUM 100
int D[MAX_NUM + 10][MAX_NUM + 10];
int N;
int aMaxSum[MAX_NUM + 10][MAX_NUM + 10];
main()
{
    int i, j;
    cin>>N;
    for( i = 1; i <= N; i ++ )
        for( j = 1; j <= i; j ++ )
            cin>>D[i][j];
        for( j = 1; j <= N; j ++ )
            aMaxSum[N][j] = D[N][j];
        for( i = N ; i > 1 ; i -- )
            for( j = 1; j < i ; j ++ ) {
                if( aMaxSum[i][j] > aMaxSum[i][j+1] )
                    aMaxSum[i-1][j] = aMaxSum[i][j] + D[i-1][j];
                else
                    aMaxSum[i-1][j] = aMaxSum[i][j+1] + D[i-1][j];
            }
        cout<< aMaxSum[1][1];
}
```

再谈动态规划

- 动态规划
  - 是运筹学的一个重要分支，是解决多阶段决策过程最优化的一种方法。
- 多阶段决策过程
  - 将所研究的过程划分为若干个相互联系的阶段，在求解时，对每一个阶段都要做出决策，前一个决策确定以后，常常会影响下一个阶段的决策。

再谈动态规划

- 动态规划的指导思想
  - 在做每一步决策时，列出各种可能的局部解，之后依据某种判定条件，舍弃那些肯定不能得到最优解的局部解。
  - 每一步都经过筛选，通过确保每一步都是最优的来保证全局是最优的。
  - 筛选相当于最大限度地有效剪枝（从搜索角度看），效率会十分高。

再谈动态规划

- 贪心与动态规划
  - 贪心法产生一个按贪心策略形成的判定序列；
  - 贪心法只能做到局部最优，不能保证全局最优，因为有些问题不符合最优性原理。
  - 动态规划会产生许多判定序列，再按最优性原理对这些序列加以筛选，去除那些非局部最优的子序列。
  - 动态规划能够获得最优解。

好好想想，有没有问题?

谢　谢　！