Finding the nth term of a single dimensional recurrence using matrix multiplication in O(logn * (m^3))

Author:	**Anna Fariha**
Revised by:	**Shariful Islam**

Content:

Introduction:

When finding the nth term of a recurrence relation, where the recurrence function has a single parameter, normally it takes linear time. Here we discuss an algorithm to find the nth term of a mentioned recurrence relation in a better complexity.

Idea:

We all know about Fibonacci series. In a Fibonacci series, the nth term is represented as

$$f_n = f_{n-1} + f_{n-2}$$

Given the first two terms, namely f_0 and f_1, we can find f_n in **O(n)** complexity.

Firstly, let's imagine that we have something like a **[]**. If we make any operation between **[]** and f_n we get f_{n+1}. Again it's clear that only f_n cannot generate f_{n+1}. So, we need both f_n and f_{n+1}.

So, if ¤ is the operation, logically we can say that

$$\left.\begin{array}{l} [] ¤ (f_0, f_1) = f_2 \\ [] ¤ (f_1, f_2) = f_3 \\ \ldots \\ [] ¤ (f_{n-1}, f_n) = f_{n+1} \end{array}\right\} \ldots (1)$$

So, now it's clear that

$$\left.\begin{array}{l} [] ¤ (f_0, f_1) = f_2 \\ []^2 ¤ (f_0, f_1) = f_3 \\ []^3 ¤ (f_0, f_1) = f_4 \\ \ldots \\ []^{n-1} ¤ (f_0, f_1) = f_n \end{array}\right\} \ldots (2)$$

Now, if **[] ¤ (f_0, f_1) = f_2**, that means we are keeping only one term, then we can't generate **f_3**, because we need **f_1** too. That means after the operation we should generate **f_n** and we should also keep **f_{n-1}** for future use. So, the operation can be redefined as

$$\left.\begin{array}{l} [] ¤ (f_0, f_1) = (f_1, f_2) \\ []^2 ¤ (f_0, f_1) = (f_2, f_3) \\ \ldots \\ []^{n-1} ¤ (f_0, f_1) = (f_{n-1}, f_n) \end{array}\right\} \ldots (3)$$

Now, we are actually close to our solution. If we think a while we can see that the input and output are actually nothing but a collection of values which can be formulated as a matrix. There are two values actually so, we can think of a **2 × 1** matrix. So, we can rewrite the functions as

$$\begin{bmatrix} & \\ & \end{bmatrix} ¤ \begin{bmatrix} \mathbf{f_n} \\ \mathbf{f_{n-1}} \end{bmatrix} = \begin{bmatrix} \mathbf{f_{n+1}} \\ \mathbf{f_n} \end{bmatrix}$$

So, how about defining ¤? If it is an addition operation, it's clear that it's impossible to make the output matrix from the input matrix, because $\mathbf{f_{n+1}} = \mathbf{f_n} + \mathbf{f_{n-1}}$. But if the operation is a multiplication operation, then it's easy. So,

$$\begin{bmatrix} & \\ & \end{bmatrix} \times \begin{bmatrix} \mathbf{f_n} \\ \mathbf{f_{n-1}} \end{bmatrix} = \begin{bmatrix} \mathbf{f_{n+1}} \\ \mathbf{f_n} \end{bmatrix}$$

Now, we have to define **[]**. Since we are multiplying it with a matrix so, it should be a matrix. Now let's assume that the dimension of this matrix is $\mathbf{m \times n}$. So, if we consider the dimensions only, we can see that

$$\mathbf{(m \times n) \times (2 \times 1) = (2 \times 1)}$$

So, according to the rule of matrix multiplication $\mathbf{n = 2}$, and again $\mathbf{m = 2}$. Thus the matrix is a $\mathbf{2 \times 2}$ matrix. And since m and n both are equal so, it's a square matrix and the power can be constructed.

$$\begin{bmatrix} \mathbf{a} & \mathbf{b} \\ \mathbf{c} & \mathbf{d} \end{bmatrix} \times \begin{bmatrix} \mathbf{f_n} \\ \mathbf{f_{n-1}} \end{bmatrix} = \begin{bmatrix} \mathbf{f_{n+1}} \\ \mathbf{f_n} \end{bmatrix}$$

Now let's try to build its contents. Again we observe carefully to note that, ***a*** and ***b*** both should be **1**. Otherwise $\mathbf{f_{n+1}}$ can't be constructed.

$$\begin{bmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{c} & \mathbf{d} \end{bmatrix} \times \begin{bmatrix} \mathbf{f_n} \\ \mathbf{f_{n-1}} \end{bmatrix} = \begin{bmatrix} \mathbf{f_{n+1}} \\ \mathbf{f_n} \end{bmatrix}$$

So, calculation of $\mathbf{f_{n+1}}$ is completed. Now for $\mathbf{f_n}$ we see that ***c*** should be **1** and ***d*** should be **0**. So, the matrix becomes

$$\begin{bmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} \end{bmatrix} \times \begin{bmatrix} \mathbf{f_n} \\ \mathbf{f_{n-1}} \end{bmatrix} = \begin{bmatrix} \mathbf{f_{n+1}} \\ \mathbf{f_n} \end{bmatrix}$$

So, we have constructed the desired matrix. Now from **(3)** we can say that

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n} \times \begin{bmatrix} f_1 \\ f_0 \end{bmatrix} = \begin{bmatrix} f_{n+1} \\ f_n \end{bmatrix}$$

Now to find the ***(n+1)th*** term we have to find the **nth** power of our constructed matrix. The brute-force implementation of this idea is to find the nth power of the matrix and then multiply it with the initial terms, here **[f_1 f_0]** represents the initial matrix. But observe that it is of the same complexity as we stated before.

Better implementation:

For an integer K, if we need to find the nth power of K we can do it by multiplying K n times. But we can observe that

if n =0, K^n = 1
else if n is even, K^n = (K^(n/2)) ×(K^(n/2))
else if n is odd, K^n = K × K^(n-1)

This takes **O(log(n))** time complexity.
The same technique works for raising a square matrix to an arbitrary power, simply replacing **1** with ***I***, the identity matrix. As multiplying two matrices is of **O(m^3)** complexity, the total complexity of finding the nth term is **O((m^3)*log(n)).**

The C implementation will be

```
matrix find_power( matrix mat, int k ){
      if( k == 0 ) return identity_matrix;
      if( k == 1 ) return mat;

      if( k&1 )                 // k is odd
            return multiply(mat, find_power( mat, k - 1 ) );
      matrix ret;
      ret = find_power( mat, k/2 );
      return multiply( ret, ret );
}
```

For the Fibonacci series, the matrix formed was very simple, consisting of only 1s and 0s.
The variation of this is used to solve many problems.

Example:

$F(n)=a_1*F(n-1)+a_2*F(n-2)+a_3*F(n-3)+.............+ a_k*F(n-k)$, where $a_1,a_2,a_3.....a_k$ are constants.

For this recurrence relation, we see only last k values are necessary to find the **nth** value. So we have to keep information of k values in each step. If we define the initial matrix as

$$\mathbf{Init} = \begin{pmatrix} \mathbf{F(k-1)} \\ \mathbf{F(k-2)} \\ \mathbf{F(k-3)} \\ . \\ . \\ . \\ \mathbf{F(1)} \\ \mathbf{F(0)} \end{pmatrix}$$

Suppose we have a matrix **M.** We want to produce the matrix

$$\mathbf{D} = \begin{pmatrix} \mathbf{F(k)} \\ \mathbf{F(k-1)} \\ \mathbf{F(k-2)} \\ . \\ . \\ . \\ \mathbf{F(2)} \\ \mathbf{F(1)} \end{pmatrix}$$

after multiplying the initial matrix with **M.**

Now we will try to construct a *matrix M* such that **Init×M= D**

By calculating the dimensions of Init and D, we get,

$$\mathbf{(1\times k)\times (p\times q) = (1\times k)}$$

So we can surely say that p = k and q =k

So M is a k×k matrix.

So if we define **M** as

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} & \cdots & x_{1(k-1)} & x_{1k} \\ x_{21} & x_{22} & x_{23} & \cdots & x_{2(k-1)} & x_{2k} \\ x_{31} & x_{32} & x_{33} & \cdots & x_{3(k-1)} & x_{3k} \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ x_{k1} & x_{k2} & x_{k3} & \cdots & x_{k(k-1)} & x_{kk} \end{pmatrix}$$

The equation must be **Init×M= D**

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1(k-1)} & x_{1k} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2(k-1)} & x_{2k} \\ x_{31} & x_{32} & x_{33} & \dots & x_{3(k-1)} & x_{3k} \\ . \\ . \\ . \\ x_{k1} & x_{k2} & x_{k3} & \dots & x_{k(k-1)} & x_{kk} \end{pmatrix} \times \begin{pmatrix} F(k-1) \\ F(k-2) \\ F(k-3) \\ . \\ . \\ . \\ F(0) \end{pmatrix} = \begin{pmatrix} F(k) \\ F(k-1) \\ F(k-2) \\ \\ \\ \\ F(1) \end{pmatrix}$$

We know that

$$\mathbf{F(k) = a_1*F(k-1)+a_2*F(k-2)+a_3*F(k-3)+.............+ a_k*F(k-k)}$$

So the value of x_{11} must be a_1, value of x_{12} must be a_2, value of x_{13} must be a_3 and so on.
Now , as we want to keep the value of *F(k-1)* in the second row of *D*, the value of x_{21} must be 1. Through a closer look, we can see that to keep the values of all F(k)s in D in the row one lower than that in the Init, we must make $x_{32}, x_{43}, x_{54}...., x_{k(k-1)} = 1$, and the other values should be = 0. so the matrix M finally becomes :

$$\mathbf{M} = \begin{pmatrix} \mathbf{a1} & \mathbf{a2} & \mathbf{a3} & & \mathbf{a(k-1)} & \mathbf{ak} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ . \\ . \\ . \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{1} & \mathbf{0} \end{pmatrix}$$

Then By multiplying initial matrix with **M,** We can get the desired matrix, **D**. From the previous example of Fibonacci numbers, it is clearly understood that **F(n+k)th** value will be found by multiplying the initial matrix with **M^$^{\mathbf{n}}$** and so on. And in finding the **nth** power of the matrix we will use the way of divide and conquer.

Best Implementation:

It is often seen that the problem requires to find the various nth powers of the same matrix repeatedly. So if possible, we can **store** some values while calculating various **nth** powers.

An implementation is useful when the matrix is large, that is finding the binary representation of the value **n** and with that value implementing a **bottom-up** approach for finding **nth** power of the matrix.

A common idiom in contests is to ask for an answer %Mulo some number *m*, since the actual answer is too large to easily represent. This technique works just as well for these situations, because the only operations that are performed are addition and multiplication. For each basic computation, take the result %Mulo *m*. You must, however, be more careful than usual about overflow, because even if *the matrix (M)* contains small entries, the fast exponentiation algorithm may multiply together two matrices with large entries. You should ensure that m^2 is not too large for the type you are using.

Finding the *nth* power of the matrix (*M*) can be even better if we pre-calculate some values for some **n**. If we **pre-calculate** the values for all powers of 2, that means for **1, 2, 4, 8, 16,….** Then while finding the **nth** power, from its binary representation, we'll just have to call the pre-calculated values and multiply them. No extra calculation is needed each time.

Recursion can also be used to do this, but it often produces stack overflow, because storing different intermediate values of the matrix in the memory is too much costly!

Sample:

Problem: A fantastic sequence $\mathbf{a_i}$ is defined in the following way: $a_0, \ldots, a_{k-1}$ are given integers, and the subsequent elements are defined by the linear recurrence relation

$$a_n = \left(\sum_{i=1}^{k} c_i a_{n-i}\right) + c_{k+1}. \quad (n \geq k)$$

Here $c_1, \ldots, c_{k+1}$ are known integers.

You have to find $\mathbf{a_n}$ %M **m** , where **n** and **m** are given.

[Ref: UVa-10754]

Solution: This can be simply implemented by finding a matrix M.

here, the matrix will be

$$\mathbf{M} = \begin{pmatrix} C_1 & C_2 & C_3 & \ldots & C_{k-1} & C_k & C_{k+1} \\ 1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ . & . & & & & & \\ 0 & 0 & 0 & \ldots & 1 & 0 & 0 \\ 0 & 0 & 0 & \ldots & 0 & 0 & 1 \end{pmatrix}$$

Here the 1 used in the rightmost bottom corner is fixed for the constant value, that is defined in the function definition.

The initial matrix

$$\textbf{Init} = \begin{pmatrix} a_{k-1} \\ a_{k-2} \\ . \\ . \\ . \\ a_0 \\ 1 \end{pmatrix}$$

When finding the nth term, we will just have to find **M^(n-k+1) (n>=k).**
And then by multiplying the result with the initial matrix, we get the result.

Here is a C++ implementation of the solution:

```
#include <iostream>

using namespace std;

#ifdef ONLINE_JUDGE
#define i64 long long
#else
#define i64 __int64
#endif

struct matrix{
      int size;
      int n[26][26];
};

int M, D;

matrix mult( matrix &m, matrix &q ){       // function for multiplying matrices p
                                           // and q
      matrix ret;
      ret.size = m.size;

      int i, j, k;
      for( i = 0; i < m.size; i++ ){
            for( j = 0; j < m.size; j++ ){
                  ret.n[i][j] = 0;
                  for( k = 0; k < q.size;  k++ )
       ret.n[i][j] = ( ret.n[i][j] + ( m.n[i][k] * ( i64 )q.n[k][j] ) % M ) % M;
            }
      }
      return ret;                                  // result of multipliacation
}

int gun( int a[], int b[] ){  // multiplying a row with a column of two different
                              //matrices
      int ret = 0, i;
      for( i = 0; i <= D; i++ )
            ret = ( ret + ( a[i] * ( i64 )b[i] ) % M ) % M;
      return ret;
}

int main(){
      int k, n, m, i, j;
      bool fl = false;
      int tc;
      cin >> tc;                           // test cases
      while( tc-- ){
            cin >> k >> m >> n;           // k+1 values of the constants - c( 1 ),
                                          // c( 2 ), ..... c( k+1 )
                                          // m is the value for finding the
                                          //result modulo m
                                          // nth term is required to find
            M = m;
            D = k;

            int sp;

            matrix mat;
            mat.size = k + 1;
            for( i = 0; i <= k; i++ ){     // constructing the matrix with
                                           // constants
                  cin>>mat.n[0][i];
                  if( mat.n[0][i] < 0 ){
                     mat.n[0][i] = ( ( mat.n[0][i] % M ) + M );   // for negative
                                                                  //numbers
                  }
                  mat.n[0][i] = mat.n[0][i] % M;    // for positive numbers
            }

            sp = mat.n[0][k];
            int initial[27] = {0};
            for( i = k - 1; i>= 0; i-- ){
                  cin>>initial[i];         // initial funtions a( 0 )... a( k-1 )
                  if( initial[i] < 0 )  initial[i] = ( initial[i] % M ) + M;

                  initial[i] = initial[i] % M;
            }

            initial[k] = 1;                      // for the last constant term
            for( i = 1; i <= k; i++ )            //  constructing  the  matrix  for
                                                 // storing the
                                                 //previous function values
                  for( j = 0; j <= k; j++ ){
                        if( j == i - 1 )  mat.n[i][j] = 1;
                        else  mat.n[i][j] = 0;
                  }
            mat.n[k][k] = 1;         // for skipping the value f( n-k ),
            mat.n[k][k - 1] = 0;           //that is not necessary and storing the
                                                   //constant term

            int r;
            if( n < k ) r = initial[k - n - 1]; // if a( n ) is in initial values
            else{
                  matrix res;
                  memset( res.n, 0, sizeof( res.n ) );
                  res = mat;
                  n = n - ( unsigned int )k + 1;
                  unsigned int bit;
                  for( bit = ( unsigned int )1 << 31; ; bit = bit >> 1 ){
                                          // finding the MSB position of n
                        if( bit & n )     break;
                  }



                  for( bit = bit >> 1; bit > 0; bit = bit >> 1 ){
                                                //finding mat^n in O( logn )
                        res = mult( res, res ); // multiplying takes O( m^3 )
                  if( bit & n )   res = mult( res, mat );
                  }
                  r = gun( res.n[0], initial );       // finding the nth term
            }
            if( !fl )   fl = 1;
            else    cout << endl;
            if( k == 0 )  r = sp;
            cout << r << endl;
      }
      return 0;
}
```

So, for best implementation, we can do some specific optimization:

- **Pre-calculate** the values of matrix powers for all powers of 2, that means for **1, 2, 4, 8, 16….**
- When performing matrix multiplication, send **reference** to the multiply function, do not call by value.
- Avoid recursion to perform finding powers of the matrix, use **bottom up** approach with the pre-calculated values of matrix powers.

Drawback:

- When the matrix becomes very large the factor m^3 is also large, so it’s difficult to decreast the time.

Benefit

- It is very easy to implement once you’ve find the matrix!
- It has minimal time complexity.
- It works quite well for very large n.

Betterments:

- Using a hash table for storing various powers of the matrix can be even better, because it decreases calculating the same power value over and over again.
- We can send the result matrix as reference also while multiplying two matrices.
  It prevents declaring a new local variable each time in the multiply function.
  Example:

```
void multiply( matrix &a, matrix &b, matrix &res )
{
      //matrix res ;    this is eliminated
      .
      .
}
```

A method to multiply matrices can be implemented in O(m^2.8).

Variations:

Variations of the problem stated before are found. The hardest part is to transform the problem that implements this algorithm. For a big clue, whenever we find that there is some kind of series of recurrence relations and the problem requires finding the nth term and n is such big that it is impossible to find the result in O(n) time complexity, it indicates that the problem might be solvable in this algorithm.

Variation 1: Given a grid having **7** columns. **4** players will start there journey in that grid. Initially all of them are in first row. A player can move diagonally to the next row. For example, a player is in row 3 and column 4, which can also be represented as **(3,4),** this player has only two valid moves, **(4,3)** and **(4,5).** But If the player is in **(3, 1),** he has only one valid move, which is **(4,2).** In there journey, any cell of the grid can not be visited more than one player.

In this problem you have to find, in how many ways all of the players can reach to row **N**.

[Ref: IUT contest, 2008]

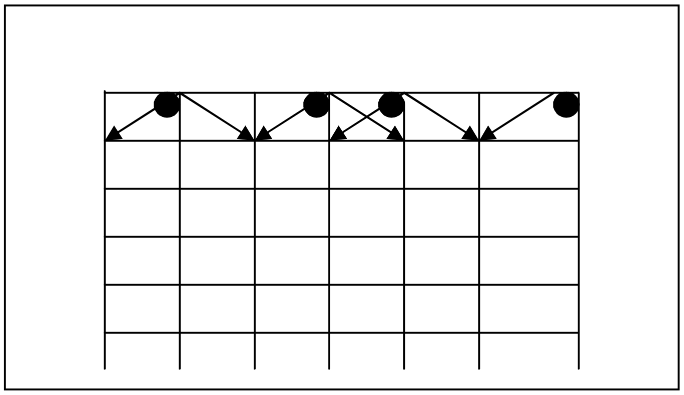

Fig (1)

Idea: In this problem, in the initial matrix, we can place **1** in the positions where a player stands and **0** elsewhere. A closer look through the problem will reveal that a state in any row depends only on the state of the previous row. Here available states are $^{7}C_{4}=$ **35**. So we can form a function that shows each state of the **nth** row depends on which states of the **(n-1)th** row. If a state **a**, in the **nth** row, depends on the state **b**, of previous row, we can place a **1** in the position **[a][b]** of the matrix (**M**). The matrix will consist of **35** rows and **35** columns, because, there are **35** possible state (in this example).

The **nth** term of the sequence gives the number of ways to reach the **nth** row. The **nth** term can be found by finding the **(n-1)th** power of the matrix and multiplying it with the initial matrix containing the initial positions of the players.

Variation 2: You are given a **6*n** chessboard. The number of columns in this chessboard is variable. In each of the columns you have to place exactly **2** knights. So you have to place total **2*n** knights. You have to count the number of valid placing of these **2*n** knight. A placing is invalid if any of the **2** knights attack each other. Those who are not familiar with knight moves "**A knight in cell (x,y) attacks the knights in the cell(x±2,y±1) and cell(x±1,y±2)**".

[Ref: UVa - 11091]

Idea: This problem is much similar to the previous one. But the difference is, here we have to keep track with the previous two columns. By examination, we can find all possible state for a **6*2** chess board. And then we can form the matrix by generating possible states. States can be formed using bit masking.

Here is a C++ implementation for this problem:

```
#include <iostream>
#include <map>
#include <vector>
using namespace std;
#define M 10007

bool chk( int a, int b ){    // verifies if state b is possible in column 2 while
                                    //column 1 is in state a

      int i;
      for( i = 1; i <= ( 1 << 5 ); i = i << 1 ){
            if( i&b ){
                  if( i >= 4 ){      if( ( i >> 2 ) & a )     return false; }
                  if( i < 16 ){      if( ( i << 2 ) & a )     return false; }

            }
      }
      return true;
}
```

```
bool check( int a, int c, int b ){  // verifies if state b is possible in
                                    //column 3 while column 2 is in state a
      int i;                        //and column 1 is in state c
      for( i = 1; i <= ( 1 << 5 ); i = i << 1 ){
            if( i&b ){
                  if( i >= 4 ){     if( ( i >> 2 ) & a )    return false; }
                  if( i < 16 ){     if( ( i << 2 ) & a )    return false; }
                  if( i >= 2 ){     if( ( i >> 1 ) & c )    return false; }
                  if( i < 32 ){     if( ( i << 1 ) & c )    return false; }

            }
      }
      return true;
}

int mp[65][65];

struct matrix{
      int n[69][69];
}mat, iden, sto[65];

void mult( matrix &m, matrix &q, matrix &ret ){ // matrix multiplication
      int i, j, k;
      for( i = 0; i < 69; i++ ){
            for( j = 0; j < 69; j++ ){
                  ret.n[i][j] = 0;
                  for( k = 0; k < 69; k++ )
            ret.n[i][j] = ( ret.n[i][j] + ( m.n[i][k] * q.n[k][j] ) % M ) % M;
            }
      }
}
struct keep{
      int one, two;
}pos[70];

int main(){
      int y = 0;
      int i, j, k, l, one, two;
      int h = 0;

      memset( iden.n, 0, sizeof( iden.n ) );
      for( i = 0; i < 69; i++ )     // Identity matrix
            iden.n[i][i] = 1;
      for( i = 1; i <= ( 1 << 5 ); i = i << 1 ){
            for( j = i; j <= ( 1 << 5 ); j = j << 1 ){
                  if( i!= j ){
                        one = i|j;        // state for column one
                        for( k = 1; k <= ( 1 << 5 ); k = k << 1 ){
                              for( l = k; l <= ( 1 << 5 ); l = l << 1 ){
                                    if( k!= l ){
                                          two = k|l; // state for column two
                                          if( chk( one, two ) ){
                                                mp[one][two] = y; //mapping state
                                                                  //for two
                                                            //consecutive columns
                                                pos[y].one = one;
                                                pos[y].two = two;
                                                y++;
```

```
                                        }
                                  }

                            }
                      }

                }
          }
    }
    int thr;
    memset( mat.n, 0, sizeof( mat.n ) );



    for( i = 0; i < 69; i++ ){
          for( k = 1; k <= ( 1 << 5 ); k = k << 1 )
                 for( l = k << 1; l <= ( 1 << 5 ); l = l << 1 ){
                      thr = k|l;  // state for column three
                      if( check( pos[i].two, pos[i].one, thr ) )
                            // validity for states of 3 consecutiove columnns
                mat.n[mp[pos[i].two][thr]][mp[pos[i].one][pos[i].two]] = 1;
                                        // constructing matrix
                 }
    }



    sto[0] = mat;
    for( i = 1; i < 32; i++ )
          mult( sto[i - 1], sto[i - 1], sto[i] );//storing powers for n( n is a
                                                       //power of two )
    int q;
    int tc;
    cin >>tc;
    while( tc-- )
    {
          cin >>q;
          int t = 0;
          if( q == 1 )                        // for first 2 columns
                cout << "15" << endl;
          else if( q == 2 ) cout << "69" << endl;
          else{
                q = q - 2;
                matrix res, ret;
                int qq;
                memset( res.n, 0, sizeof( res.n ) );
                for( qq = 0; qq < 69; qq++ )//initialize res with identity
                                                      //matrix
                      res.n[qq][qq] = 1;
                int g = 0;
                for( ; q > 0; q = q >> 1, g++ ){// finding power of the matrix
                      if( q&1 ){
                            mult( res, sto[g], ret );

                            res = ret;
                      }
                }
```

```
                    for( i = 0; i < 69; i++ ){          // finding total ways
                          for( j = 0; j < 69; j++ )
                                t = ( t + res.n[i][j] )%M;
                    }
                    cout << t << endl;
              }
        }
        return 0;
}
```

Variation 3: Like the Fibonacci sequence, many series can be calculated from this form. For example, let $a_i = 1+2c+3c^2+...+ic^{i-1}$, for some constant c.

We can solve this by forming a matrix. From the definition of the function, we can say that

$a_1 = 1$
$a_2 = 1+2c = a1+2c$
$a_3 = 1+2c+3c^2 = a_2 + 3c^2$
.
.

So, $a_{i+1} = a_i + (i+1)c^i \Big\}$ (1)

And we can see that,
$(i+2)c^{i+1} = (i+1)c^i \times c + c^i \times c \Big\}$ (2)
$c^{i+1} = c^i \times c \Big\}$ (3)

So, from (1), (2), (3) we can say that, if

$$x_i = \begin{pmatrix} a_i \\ (i+1)\,c^i \\ c^i \end{pmatrix} \quad \text{and} \quad M = \begin{pmatrix} 1 & 1 & 0 \\ 0 & c & c \\ 0 & 0 & c \end{pmatrix}$$

Then

$$x_{i+1} = \begin{pmatrix} a_{i+1} \\ (i+2)c^{i+1} \\ c^{i+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & c & c \\ 0 & 0 & c \end{pmatrix} \times \begin{pmatrix} a_i \\ (i+1)c^i \\ c_i \end{pmatrix}$$

$$= M \times x_i$$

Some related problems:

- UVa 10870
- UVa 10754
- UVa 11149
- UVa 11486
- UVa 11091

Reference:

1. Jane Alam Jan
2. http://icpcres.ecs.baylor.edu/onlinejudge
3. http://acm.uva.es/board
4. Linear recurrences By bmerry
   http://www.topcoder.com/tc?%Mule=Static&d1=features&d2=010408