COME ON CODE ON

A blog about programming and more programming.

Modular Multiplicative Inverse

with 22 comments

The modular multiplicative inverse of an integer a modulo m is an integer x such that $a^{-1} \equiv x \pmod{m}$.

That is, it is the multiplicative inverse in the ring of integers modulo m. This is equivalent to $ax \equiv aa^{-1} \equiv 1 \pmod{m}$.

The multiplicative inverse of a modulo m exists if and only if a and m are coprime (i.e., if gcd(a, m) = 1).

Let's see various ways to calculate Modular Multiplicative Inverse:

1. Brute Force
We can calculate the inverse using a brute force approach where we multiply *a* with all possible values *x* and find a *x* such that $ax \equiv 1 \pmod{m}$. Here's a sample C++ code:

```
1  int modInverse(int a, int m) {
2      a %= m;
3      for(int x = 1; x < m; x++) {
4          if((a*x) % m == 1) return x;
5      }
6  }
```

The time complexity of the above codes is O(m).

2. Using Extended Euclidean Algorithm
We have to find a number x such that a·x = 1 (mod m). This can be written as well as a·x = 1 + m·y, which rearranges into a·x – m·y = 1. Since x and y need not be positive, we can write it as well in the standard form, a·x + m·y = 1.

In number theory, Bézout's identity for two integers a, b is an expression ax + by = d, where x and y are integers (called Bézout coefficients for (a,b)), such that d is a common divisor of a and b. If d is the greatest common divisor of a and b then Bézout's identity ax + by = gcd(a,b) can be solved using Extended Euclidean Algorithm.

The Extended Euclidean Algorithm is an extension to the Euclidean algorithm. Besides finding the greatest common divisor of integers a and b, as the Euclidean algorithm does, it also finds integers x and y (one of which is typically negative) that satisfy Bézout's identity
ax + by = gcd(a,b). The Extended Euclidean Algorithm is particularly useful when a and b are coprime, since x is the multiplicative inverse of a modulo b, and y is the multiplicative inverse of b modulo a.

We will look at two ways to find the result of Extended Euclidean Algorithm.

Iterative Method
This method computes expressions of the form $r_i = ax_i + by_i$ for the remainder in each step i of the Euclidean algorithm. Each successive number r_i can be written as the remainder of the division of the previous two such numbers, which remainder can be expressed using the whole quotient q_i of that division as follows:
$r_i = r_{i-2} - q_i r_{i-1}.$
By substitution, this gives:
$r_i = (ax_{i-2} + by_{i-2}) - q_i(ax_{i-1} + by_{i-1})$, which can be written
$r_i = a(x_{i-2} - q_i x_{i-1}) + b(y_{i-2} - q_i y_{i-1}).$
The first two values are the initial arguments to the algorithm:
$r_1 = a = a \times 1 + b \times 0$
$r_2 = b = a \times 0 + b \times 1.$
So the coefficients start out as $x_1 = 1$, $y_1 = 0$, $x_2 = 0$, and $y_2 = 1$, and the others are given by
$x_i = x_{i-2} - q_i x_{i-1},$
$y_i = y_{i-2} - q_i y_{i-1}.$
The expression for the last non-zero remainder gives the desired results since this method computes every remainder in terms of a and b, as desired.

So the algorithm looks like,

1. Apply Euclidean algorithm, and let qn(n starts from 1) be a finite list of quotients in the division.
2. Initialize x_0, x_1 as 1, 0, and y_0, y_1 as 0,1 respectively.
    1. Then for each i so long as q_i is defined,
    2. Compute $x_{i+1} = x_{i-1} - q_i x_i$
    3. Compute $y_{i+1} = y_{i-1} - q_i y_i$
    4. Repeat the above after incrementing i by 1.
3. The answers are the second-to-last of x_n and y_n.

```
1  /* This function return the gcd of a and b followed by
2      the pair x and y of equation ax + by = gcd(a,b)*/
3  pair<int, pair<int, int> > extendedEuclid(int a, int b) {
4      int x = 1, y = 0;
5      int xLast = 0, yLast = 1;
6      int q, r, m, n;
7      while(a != 0) {
8          q = b / a;
9          r = b % a;
10         m = xLast - q * x;
```

```
            n = yLast - q * y;
            xLast = x, yLast = y;
            x = m, y = n;
            b = a, a = r;
        }
        return make_pair(b, make_pair(xLast, yLast));
    }

    int modInverse(int a, int m) {
        return (extendedEuclid(a,m).second.first + m) % m;
    }
```

Recursive Method

This method attempts to solve the original equation directly, by reducing the dividend and divisor gradually, from the first line to the last line, which can then be substituted with trivial value and work backward to obtain the solution.

Notice that the equation remains unchanged after decomposing the original dividend in terms of the divisor plus a remainder, and then regrouping terms. So the algorithm looks like this:

1. If b = 0, the algorithm ends, returning the solution x = 1, y = 0.
2. Otherwise:
   - Determine the quotient q and remainder r of dividing a by b using the integer division algorithm.
   - Then recursively find coefficients s, t such that bs + rt divides both b and r.
   - Finally the algorithm returns the solution x = t, and y = s − qt.

Here's a C++ implementation:

```
/* This function return the gcd of a and b followed by
    the pair x and y of equation ax + by = gcd(a,b)*/
pair<int, pair<int, int> > extendedEuclid(int a, int b) {
    if(a == 0) return make_pair(b, make_pair(0, 1));
    pair<int, pair<int, int> > p;
    p = extendedEuclid(b % a, a);
    return make_pair(p.first, make_pair(p.second.second - p.second.first
}

int modInverse(int a, int m) {
    return (extendedEuclid(a,m).second.first + m) % m;
}
```

The time complexity of the above codes is $O(log(m)^2)$.

3. Using Fermat's Little Theorem

Fermat's little theorem states that if m is a prime and a is an integer co-prime to m, then $a^p - 1$ will be evenly divisible by m. That is $a^{m-1} \equiv 1 \pmod{m}$. or $a^{m-2} \equiv a^{-1} \pmod{m}$. Here's a sample C++ code:

```
/* This function calculates (a^b)%MOD */
```

```
int pow(int a, int b, int MOD) {
int x = 1, y = a;
    while(b > 0) {
        if(b%2 == 1) {
            x=(x*y);
            if(x>MOD) x%=MOD;
        }
        y = (y*y);
        if(y>MOD) y%=MOD;
        b /= 2;
    }
    return x;
}

int modInverse(int a, int m) {
    return pow(a,m-2,m);
}
```

The time complexity of the above codes is O(log(m)).

4. Using Euler's Theorem
Fermat's Little theorem can only be used if m is a prime. If m is not a prime we can use Euler's Theorem, which is a generalization of Fermat's Little theorem. According to Euler's theorem, if a is coprime to m, that is, gcd(a, m) = 1, then $a^{\varphi(m)} \equiv 1 \pmod{m}$, where where ϕ(m) is Euler Totient Function. Therefore the modular multiplicative inverse can be found directly: $a^{\varphi(m)-1} \equiv a^{-1} \pmod{m}$. The problem here is finding ϕ(m). If we know ϕ(m), then it is very similar to above method.

Now lets take a little different question. Now suppose you have to calculate the inverse of first n numbers. From above the best we can do is O(n log(m)). Can we do any better? Yes.

We can use sieve to find a factor of composite numbers less than n. So for composite numbers inverse(i) = (inverse(i/factor(i)) * inverse(factor(i))) % m, and we can use either Extended Euclidean Algorithm or Fermat's Theorem to find inverse for prime numbers. But we can still do better.

a * (m / a) + m % a = m
(a * (m / a) + m % a) mod m = m mod m, or
(a * (m / a) + m % a) mod m = 0, or
(- (m % a)) mod m = (a * (m / a)) mod m.
Dividing both sides by (a * (m % a)), we get
– inverse(a) mod m = ((m/a) * inverse(m % a)) mod m
inverse(a) mod m = (- (m/a) * inverse(m % a)) mod m

Here's a sample C++ code:

```
vector<int> inverseArray(int n, int m) {
    vector<int> modInverse(n + 1,0);
    modInverse[1] = 1;
    for(int i = 2; i <= n; i++) {
```

```
        modInverse[i] = (-(m/i) * modInverse[m % i]) % m + m;
    }
    return modInverse;
}
```

The time complexity of the above code is O(n).

-fR0DDY

Written by fR0DDY

October 9, 2011 at 12:29 AM

Posted in Programming

Tagged with algorithm, C, code, euclidean, Euler, fermat, inverse, little, modular, multiplicative, theorem

Combination

with 13 comments

In mathematics a combination is a way of selecting several things out of a larger group, where (unlike permutations) order does not matter. More formally a k-combination of a set S is a subset of k distinct elements of S. If the set has n elements the number of k-combinations is equal to the binomial coefficient. In this post we will see different methods to calculate the binomial.

1. Using Factorials
We can calculate nCr directly using the factorials.
nCr = n! / (r! * (n-r)!)

```
#include<iostream>
using namespace std;

long long C(int n, int r)
{
    long long f[n + 1];
    f[0]=1;
    for (int i=1;i<=n;i++)
        f[i]=i*f[i-1];
    return f[n]/f[r]/f[n-r];
}

int main()
{
    int n,r,m;
```

```
    while (~scanf("%d%d",&n,&r))
    {
        printf("%lld\n",C(n, min(r,n-r)));
    }
}
```

But this will work for only factorial below 20 in C++. For larger factorials you can either write big factorial library or use a language like Python. The time complexity is O(n).
If we have to calcuate nCr mod p(where p is a prime), we can calculate factorial mod p and then use modular inverse to find nCr mod p. If we have to find nCr mod m(where m is not prime), we can factorize m into primes and then use Chinese Remainder Theorem(CRT) to find nCr mod m.

```
#include<iostream>
using namespace std;
#include<vector>

/* This function calculates (a^b)%MOD */
long long pow(int a, int b, int MOD)
{
    long long x=1,y=a;
    while(b > 0)
    {
        if(b%2 == 1)
        {
            x=(x*y);
            if(x>MOD) x%=MOD;
        }
        y = (y*y);
        if(y>MOD) y%=MOD;
        b /= 2;
    }
    return x;
}

/*  Modular Multiplicative Inverse
    Using Euler's Theorem
    a^(phi(m)) = 1 (mod m)
    a^(-1) = a^(m-2) (mod m) */
long long InverseEuler(int n, int MOD)
{
    return pow(n,MOD-2,MOD);
}

long long C(int n, int r, int MOD)
{
    vector<long long> f(n + 1,1);
    for (int i=2; i<=n;i++)
        f[i]= (f[i-1]*i) % MOD;
    return (f[n]*((InverseEuler(f[r], MOD) * InverseEuler(f[n-r], MOD)) %
}

int main()
{
```

```
    int n,r,p;
    while (~scanf("%d%d%d",&n,&r,&p))
    {
        printf("%lld\n",C(n,r,p));
    }
}
```

2. Using Recurrence Relation for nCr
The recurrence relation for nCr is **C(i,k) = C(i-1,k-1) + C(i-1,k)**. Thus we can calculate nCr in time complexity O(n*r) and space complexity O(n*r).

```
#include<iostream>
using namespace std;
#include<vector>

/*
    C(n,r) mod m
    Using recurrence:
    C(i,k) = C(i-1,k-1) + C(i-1,k)
    Time Complexity: O(n*r)
    Space Complexity: O(n*r)
*/

long long C(int n, int r, int MOD)
{
    vector< vector<long long> > C(n+1,vector<long long> (r+1,0));

    for (int i=0; i<=n; i++)
    {
        for (int k=0; k<=r && k<=i; k++)
            if (k==0 || k==i)
                C[i][k] = 1;
            else
                C[i][k] = (C[i-1][k-1] + C[i-1][k])%MOD;
    }
    return C[n][r];
}
int main()
{
    int n,r,m;
    while (~scanf("%d%d%d",&n,&r,&m))
    {
        printf("%lld\n",C(n, r, m));
    }
}
```

We can easily reduce the space complexity of the above solution by just keeping track of the previous row as we don't need the rest rows.

```
#include<iostream>
using namespace std;
#include<vector>
```

```
4
5   /*
6       Time Complexity: O(n*r)
7       Space Complexity: O(r)
8   */
9   long long C(int n, int r, int MOD)
10  {
11      vector< vector<long long> > C(2,vector<long long> (r+1,0));
12
13      for (int i=0; i<=n; i++)
14      {
15          for (int k=0; k<=r && k<=i; k++)
16              if (k==0 || k==i)
17                  C[i&1][k] = 1;
18              else
19                  C[i&1][k] = (C[(i-1)&1][k-1] + C[(i-1)&1][k])%MOD;
20      }
21      return C[n&1][r];
22  }
23
24  int main()
25  {
26      int n,r,m,i,k;
27      while (~scanf("%d%d%d",&n,&r,&m))
28      {
29          printf("%lld\n",C(n, r, m));
30      }
31  }
```

3. Using expansion of nCr
Since

```
C(n,k) = n!/((n-k)!k!)
         [n(n-1)...(n-k+1)][(n-k)...(1)]
       = -------------------------------
           [(n-k)...(1)][k(k-1)...(1)]
```

We can cancel the terms: [(n-k)…(1)] as they appear both on top and bottom, leaving:

```
    n (n-1)     (n-k+1)
    - ----- ... -------
    k (k-1)       (1)
```

which we might write as:

```
C(n,k) = 1,                  if k = 0
       = (n/k)*C(n-1, k-1), otherwise
```

```
1   #include<iostream>
2   using namespace std;
3
4   long long C(int n, int r)
```

```
5  {
6      if (r==0) return 1;
7      else return C(n-1,r-1) * n / r;
8  }
9
10 int main()
11 {
12     int n,r,m;
13     while (~scanf("%d%d",&n,&r))
14     {
15         printf("%lld\n",C(n, min(r,n-r)));
16     }
17 }
```

4. Using Matrix Multiplication
In the last post we learned how to use Fast Matrix Multiplication to calculate functions having linear equations in logarithmic time. Here we have the recurrence relation C(i,k) = C(i-1,k-1) + C(i-1,k).
If we take k=3 we can write,
C(i-1,1) + C(i-1,0) = C(i,1)
C(i-1,2) + C(i-1,1) = C(i,2)
C(i-1,3) + C(i-1,2) = C(i,3)
Now on the left side we have four variables C(i-1,0), C(i-1,1), C(i-1,2) and C(i-1,3).
On the right side we have three variables C(i,1), C(i,2) and C(i,3).
We need those two sets to be the same, except that the right side index numbers should be one higher than the left side index numbers. So we add C(i,0) on the right side. NOw let's get our all important Matrix.

```
(.   .   .   .)  ( C(i-1,0) )   ( C(i,0) )
(.   .   .   .)  ( C(i-1,1) ) = ( C(i,1) )
(.   .   .   .)  ( C(i-1,2) )   ( C(i,2) )
(.   .   .   .)  ( C(i-1,3) )   ( C(i,3) )
```

The last three rows are trivial and can be filled from the recurrence equations above.

```
(.   .   .   .)  ( C(i-1,0) )   ( C(i,0) )
(1   1   0   0)  ( C(i-1,1) ) = ( C(i,1) )
(0   1   1   0)  ( C(i-1,2) )   ( C(i,2) )
(0   0   1   1)  ( C(i-1,3) )   ( C(i,3) )
```

The first row, for C(i,0), depends on what is supposed to happen when k = 0. We know that C(i,0) = 1 for all i when k=0. So the matrix reduces to

```
(.   .   .   .)  ( C(i-1,0) )   ( C(i,0) )
(1   1   0   0)  ( C(i-1,1) ) = ( C(i,1) )
(0   1   1   0)  ( C(i-1,2) )   ( C(i,2) )
(0   0   1   1)  ( C(i-1,3) )   ( C(i,3) )
```

And this then leads to the general form:

```
                i
(.   .   .   .)  ( C(0,0) )   ( C(i,0) )
(1   1   0   0)  ( C(0,1) ) = ( C(i,1) )
(0   1   1   0)  ( C(0,2) )   ( C(i,2) )
(0   0   1   1)  ( C(0,3) )   ( C(i,3) )
```

For example if we wan't C(4,3) we just raise the above matrix to the 4th power.

```
               4
(1   0   0   0)  ( 1 )   ( 1 )
(1   1   0   0)  ( 0 ) = ( 4 )
(0   1   1   0)  ( 0 )   ( 6 )
(0   0   1   1)  ( 0 )   ( 4 )
```

Here's a C++ code.

```
#include<iostream>
using namespace std;

/*
    C(n,r) mod m
    Using Matrix Exponentiation
    Time Complexity: O((r^3)*log(n))
    Space Complexity: O(r*r)
*/

long long MOD;

template< class T >
class Matrix
{
    public:
        int m,n;
        T *data;

        Matrix( int m, int n );
        Matrix( const Matrix< T > &matrix );

        const Matrix< T > &operator=( const Matrix< T > &A );
        const Matrix< T > operator*( const Matrix< T > &A );
        const Matrix< T > operator^( int P );

        ~Matrix();
};

template< class T >
Matrix< T >::Matrix( int m, int n )
{
    this->m = m;
    this->n = n;
    data = new T[m*n];
}

template< class T >
```

```
39  Matrix< T >::Matrix( const Matrix< T > &A )
40  {
41      this->m = A.m;
42      this->n = A.n;
43      data = new T[m*n];
44      for( int i = 0; i < m * n; i++ )
45          data[i] = A.data[i];
46  }
47
48  template< class T >
49  Matrix< T >::~Matrix()
50  {
51      delete [] data;
52  }
53
54  template< class T >
55  const Matrix< T > &Matrix< T >::operator=( const Matrix< T > &A )
56  {
57      if( &A != this )
58      {
59          delete [] data;
60          m = A.m;
61          n = A.n;
62          data = new T[m*n];
63          for( int i = 0; i < m * n; i++ )
64              data[i] = A.data[i];
65      }
66      return *this;
67  }
68
69  template< class T >
70  const Matrix< T > Matrix< T >::operator*( const Matrix< T > &A )
71  {
72      Matrix C( m, A.n );
73      for( int i = 0; i < m; ++i )
74          for( int j = 0; j < A.n; ++j )
75          {
76              C.data[i*C.n+j]=0;
77              for( int k = 0; k < n; ++k )
78                  C.data[i*C.n+j] = (C.data[i*C.n+j] + (data[i*n+k]*A.dat
79          }
80      return C;
81  }
82
83  template< class T >
84  const Matrix< T > Matrix< T >::operator^( int P )
85  {
86      if( P == 1 ) return (*this);
87      if( P & 1 ) return (*this) * ((*this) ^ (P-1));
88      Matrix B = (*this) ^ (P/2);
89      return B*B;
90  }
91
92  long long C(int n, int r)
```

```
 93  {
 94      Matrix<long long> M(r+1,r+1);
 95      for (int i=0;i<(r+1)*(r+1);i++)
 96          M.data[i]=0;
 97      M.data[0]=1;
 98      for (int i=1;i<r+1;i++)
 99      {
100          M.data[i*(r+1)+i-1]=1;
101          M.data[i*(r+1)+i]=1;
102      }
103      return (M^n).data[r*(r+1)];
104  }
105
106  int main()
107  {
108      int n,r;
109      while (~scanf("%d%d%lld",&n,&r,&MOD))
110      {
111          printf("%lld\n",C(n, r));
112      }
113  }
```

5. Using the power of prime p in n factorial
The power of prime p in n factorial is given by

$\varepsilon_p = \lfloor n/p \rfloor + \lfloor n/p^2 \rfloor + \lfloor n/p^3 \rfloor \ldots$

If we call the power of p in n factorial, the power of p in nCr is given by
e = countFact(n,i) – countFact(r,i) – countFact(n-r,i)
To get the result we multiply p^e for all p less than n.

```
 1  #include<iostream>
 2  using namespace std;
 3  #include<vector>
 4
 5  /* This function calculates power of p in n! */
 6  int countFact(int n, int p)
 7  {
 8      int k=0;
 9      while (n>0)
10      {
11          k+=n/p;
12          n/=p;
13      }
14      return k;
15  }
16
17  /* This function calculates (a^b)%MOD */
18  long long pow(int a, int b, int MOD)
19  {
20      long long x=1,y=a;
21      while(b > 0)
22      {
```

```
        if(b%2 == 1)
        {
            x=(x*y);
            if(x>MOD) x%=MOD;
        }
        y = (y*y);
        if(y>MOD) y%=MOD;
        b /= 2;
    }
    return x;
}

long long C(int n, int r, int MOD)
{
    long long res = 1;
    vector<bool> isPrime(n+1,1);
    for (int i=2; i<=n; i++)
        if (isPrime[i])
        {
            for (int j=2*i; j<=n; j+=i)
                isPrime[j]=0;
            int k = countFact(n,i) - countFact(r,i) - countFact(n-r,i);
            res = (res * pow(i, k, MOD)) % MOD;
        }
    return res;
}

int main()
{
    int n,r,m;
    while (scanf("%d%d%d",&n,&r,&m))
    {
        printf("%lld\n",C(n,r,m));
    }
}
```

6. Using Lucas Theorem

For non-negative integers m and n and a prime p, the following congruence relation holds:

$$\binom{m}{n} \equiv \prod_{i=0}^{k} \binom{m_i}{n_i} \pmod{p},$$

where

$$m = m_k p^k + m_{k-1} p^{k-1} + \cdots + m_1 p + m_0,$$

and

$$n = n_k p^k + n_{k-1} p^{k-1} + \cdots + n_1 p + n_0$$

are the base p expansions of m and n respectively.

We only need to calculate nCr only for small numbers (less than equal to p) using any of the above methods.

```
#include<iostream>
using namespace std;
#include<vector>

```

```
long long SmallC(int n, int r, int MOD)
{
    vector< vector<long long> > C(2,vector<long long> (r+1,0));

    for (int i=0; i<=n; i++)
    {
        for (int k=0; k<=r && k<=i; k++)
            if (k==0 || k==i)
                C[i&1][k] = 1;
            else
                C[i&1][k] = (C[(i-1)&1][k-1] + C[(i-1)&1][k])%MOD;
    }
    return C[n&1][r];
}

long long Lucas(int n, int m, int p)
{
    if (n==0 && m==0) return 1;
    int ni = n % p;
    int mi = m % p;
    if (mi>ni) return 0;
    return Lucas(n/p, m/p, p) * SmallC(ni, mi, p);
}

long long C(int n, int r, int MOD)
{
    return Lucas(n, r, MOD);
}

int main()
{

    int n,r,p;
    while (~scanf("%d%d%d",&n,&r,&p))
    {
          printf("%lld\n",C(n,r,p));
    }
}
```

7. Using special n! mod p

We will calculate n factorial mod p and similarly inverse of r! mod p and (n-r)! mod p and multiply to find the result. But while calculating factorial mod p we remove all the multiples of p and write

n!* mod p = 1 * 2 * … * (p-1) * 1 * 2 * … * (p-1) * 2 * 1 * 2 * … * n.

We took the usual factorial, but excluded all factors of p (1 instead of p, 2 instead of 2p, and so on). Lets call this *strange factorial*.

So *strange factorial* is really several blocks of construction:

1 * 2 * 3 * … * (p-1) * i

where i is a 1-indexed index of block taken again without factors p.

The last block could be *not* full. More precisely, there will be floor(n/p) full blocks and some tail (its result we can compute easily, in O(P)).

The result in each block is multiplication 1 * 2 * … * (p-1), which is common to all blocks, and multiplication of all *strange indices* i from 1 to floor(n/p).
But multiplication of all *strange indices* is really a strange factorial again, so we can compute it recursively. Note, that in recursive calls n reduces exponentially, so this is rather fast algorithm.
Here's the algorithm to calculate *strange factorial*.

```
1  int factMOD(int n, int MOD)
2  {
3      long long res = 1;
4      while (n > 1)
5      {
6          long long cur = 1;
7          for (int i=2; i<MOD; ++i)
8              cur = (cur * i) % MOD;
9          res = (res * powmod (cur, n/MOD, MOD)) % MOD;
10         for (int i=2; i<=n%MOD; ++i)
11             res = (res * i) % MOD;
12         n /= MOD;
13     }
14     return int (res % MOD);
15 }
```

But we can still reduce our complexity.
By Wilson's Theorem, we know $(n-1)! \equiv -1 \pmod{n}$ for all primes n. SO our method reduces to:

```
1  long long factMOD(int n, int MOD)
2  {
3      long long res = 1;
4      while (n > 1)
5      {
6          res = (res * pow(MOD - 1, n/MOD, MOD)) % MOD;
7          for (int i=2, j=n%MOD; i<=j; i++)
8              res = (res * i) % MOD;
9          n/=MOD;
10     }
11     return res;
12 }
```

Now in the above code we are calculating (-1)^(n/p). If (n/p) is even what we are multiplying by 1, so we can skip that. We only need to consider the case when (n/p) is odd, in which case we are multiplying result by (-1)%MOD, which ultimately is equal to MOD-res. SO our method again reduces to:

```
1  long long factMOD(int n, int MOD)
2  {
3      long long res = 1;
4      while (n > 0)
5      {
6          for (int i=2, m=n%MOD; i<=m; i++)
7              res = (res * i) % MOD;
8          if ((n/=MOD)%2 > 0)
```

```
            res = MOD - res;
        }
        return res;
    }
```

Finally the complete code here:

```
#include<iostream>
using namespace std;
#include<vector>

/* This function calculates power of p in n! */
int countFact(int n, int p)
{
    int k=0;
    while (n>=p)
    {
        k+=n/p;
        n/=p;
    }
    return k;
}

/* This function calculates (a^b)%MOD */
long long pow(int a, int b, int MOD)
{
    long long x=1,y=a;
    while(b > 0)
    {
        if(b%2 == 1)
        {
            x=(x*y);
            if(x>MOD) x%=MOD;
        }
        y = (y*y);
        if(y>MOD) y%=MOD;
        b /= 2;
    }
    return x;
}

/*  Modular Multiplicative Inverse
    Using Euler's Theorem
    a^(phi(m)) = 1 (mod m)
    a^(-1) = a^(m-2) (mod m) */
long long InverseEuler(int n, int MOD)
{
    return pow(n,MOD-2,MOD);
}

long long factMOD(int n, int MOD)
{
    long long res = 1;
```

```
47      while (n > 0)
48      {
49          for (int i=2, m=n%MOD; i<=m; i++)
50              res = (res * i) % MOD;
51          if ((n/=MOD)%2 > 0)
52              res = MOD - res;
53      }
54      return res;
55  }
56
57  long long C(int n, int r, int MOD)
58  {
59      if (countFact(n, MOD) > countFact(r, MOD) + countFact(n-r, MOD))
60          return 0;
61
62      return (factMOD(n, MOD) *
63              ((InverseEuler(factMOD(r, MOD), MOD) *
64              InverseEuler(factMOD(n-r, MOD), MOD)) % MOD)) % MOD;
65  }
66
67  int main()
68  {
69      int n,r,p;
70      while (~scanf("%d%d%d",&n,&r,&p))
71      {
72          printf("%lld\n",C(n,r,p));
73      }
74  }
```

-fR0D

Written by fR0DDY

July 31, 2011 at 5:30 PM

Posted in Algorithm

Tagged with algorithm, binomial, code, combination, Euler, factorial, inverse, Lucas, matrix, multiplication, recurrence, theorem, wilson

Recurrence Relation and Matrix Exponentiation

with 14 comments

Recurrence relations appear many times in computer science. Using recurrence relation and dynamic programming we can calculate the n^{th} term in O(n) time. But many times we need to calculate the n^{th} in O(log n) time. This is where Matrix Exponentiation comes to rescue.

We will specifically look at linear recurrences. A linear recurrence is a sequence of vectors defined by the equation X_{i+1} = M X_i for some constant matrix M. So our aim is to find this constant matrix M, for a given recurrence relation.

Let's first start by looking at the common structure of our three matrices X_{i+1}, X_i and M. For a recurrence relation where the next term is dependent on last k terms, X_{i+1} and X_{i+1} are matrices of size 1 x k and M is a matrix of size k x k.

```
| f(n+1)   |       | f(n)   |
|  f(n)    |       | f(n-1) |
| f(n-1)   | = M x | f(n-2) |
| ......   |       | ...... |
| f(n-k+1) |       | f(n-k) |
```

Let's look at different type of recurrence relations and how to find M.

1. Let's start with the most common recurrence relation in computer science, The Fibonacci Sequence. $F_{i+1} = F_i + F_{i-1}$.

```
| f(n+1) | = M x | f(n)   |
| f(n)   |       | f(n-1) |
```

Now we know that M is a 2 x 2 matrix. Let it be

```
| f(n+1) | = | a b | x | f(n)   |
| f(n)   |   | c d |   | f(n-1) |
```

Now a*f(n) + b*f(n-1) = f(n+1) and c*f(n) + d*f(n-1) = f(n). Solving these two equations we get a=1,b=1,c=1 and d=0. So,

```
| f(n+1) | = | 1 1 | x | f(n)   |
| f(n)   |   | 1 0 |   | f(n-1) |
```

2. For recurrence relation f(n) = a*f(n-1) + b*f(n-2) + c*f(n-3), we get

```
| f(n+1) | = | a b c | x | f(n)   |
| f(n)   |   | 1 0 0 |   | f(n-1) |
| f(n-1) |   | 0 1 0 |   | f(n-2) |
```

3. What if the recurrence relation is f(n) = a*f(n-1) + b*f(n-2) + c, where c is a constant. We can also add it in the matrices as a state.

```
| f(n+1) | = | a b 1 | x | f(n)   |
| f(n)   |   | 1 0 0 |   | f(n-1) |
| c      |   | 0 0 1 |   | c      |
```

4. If a recurrence relation is given like this f(n) = f(n-1) if n is odd, f(n-2) otherwise, we can convert it to f(n) = (n&1) * f(n-1) + (!(n&1)) * f(n-2) and substitute the value accordingly in the matrix.

5.If there are more than one recurrence relation, g(n) = a*g(n-1) + b*g(n-2) + c*f(n), and, f(n) = d*f(n-1) + e*f(n-2). We can still define the matrix X in following way

```
| g(n+1) |   | a b c 0 |   | g(n)   |
| g(n)   | = | 1 0 0 0 | x | g(n-1) |
| f(n+2) |   | 0 0 d e |   | f(n+1) |
| f(n+1) |   | 0 0 1 0 |   | f(n)   |
```

Now that we have got our matrix M, how are we going to find the nth term.

X_{i+1} = M X_i

(Multiplying M both sides)

M * X_{i+1} = M * M X_i

X_{i+2} = M^2 X_i

..

X_{i+k} = M^k X_i

So all we need now is to find the matrix M^k to find the k-th term. For example in the case of Fibonacci Sequence,

```
M^2 = | 2 1 |
      | 1 1 |
```

Hence F(2) = 2.

Now we need to learn to find M^k in $O(n^3 \log b)$ time. The brute force approach to calculate a^b takes O(b) time, but using a recursive divide-and-conquer algorithm takes only O(log b) time:

- If b = 0, then the answer is 1.
- If b = 2k is even, then $a^b = (a^k)^2$.
- If b is odd, then $a^b = a * a^{b-1}$.

We take a similar approach for Matrix Exponentiation. The multiplication part takes the $O(n^3)$ time and hence the overall complexity is $O(n^3 \log b)$. Here's a sample code in C++ using template class:

```
#include<iostream>
using namespace std;

template< class T >
class Matrix
{
    public:
        int m,n;
        T *data;

        Matrix( int m, int n );
        Matrix( const Matrix< T > &matrix );

        const Matrix< T > &operator=( const Matrix< T > &A );
```

```
15          const Matrix< T > operator*( const Matrix< T > &A );
16          const Matrix< T > operator^( int P );
17
18          ~Matrix();
19  };
20
21  template< class T >
22  Matrix< T >::Matrix( int m, int n )
23  {
24      this->m = m;
25      this->n = n;
26      data = new T[m*n];
27  }
28
29  template< class T >
30  Matrix< T >::Matrix( const Matrix< T > &A )
31  {
32      this->m = A.m;
33      this->n = A.n;
34      data = new T[m*n];
35      for( int i = 0; i < m * n; i++ )
36          data[i] = A.data[i];
37  }
38
39  template< class T >
40  Matrix< T >::~Matrix()
41  {
42      delete [] data;
43  }
44
45  template< class T >
46  const Matrix< T > &Matrix< T >::operator=( const Matrix< T > &A )
47  {
48      if( &A != this )
49      {
50          delete [] data;
51          m = A.m;
52          n = A.n;
53          data = new T[m*n];
54          for( int i = 0; i < m * n; i++ )
55              data[i] = A.data[i];
56      }
57      return *this;
58  }
59
60  template< class T >
61  const Matrix< T > Matrix< T >::operator*( const Matrix< T > &A )
62  {
63      Matrix C( m, A.n );
64      for( int i = 0; i < m; ++i )
65          for( int j = 0; j < A.n; ++j )
66          {
67              C.data[i*C.n+j]=0;
68              for( int k = 0; k < n; ++k )
```

```
                C.data[i*C.n+j] = C.data[i*C.n+j] + data[i*n+k]*A.data[k
        }
    return C;
}

template< class T >
const Matrix< T > Matrix< T >::operator^( int P )
{
    if( P == 1 ) return (*this);
    if( P & 1 ) return (*this) * ((*this) ^ (P-1));
    Matrix B = (*this) ^ (P/2);
    return B*B;
}

int main()
{
    Matrix<int> M(2,2);
    M.data[0] = 1;M.data[1] = 1;
    M.data[2] = 1;M.data[3] = 0;

    int F[2]={0,1};
    int N;
    while (~scanf("%d",&N))
        if (N>1)
            printf("%lld\n",(M^N).data[0]);
        else
            printf("%d\n",F[N]);
}
```

Written by fR0DDY

May 8, 2011 at 5:26 PM

Posted in Programming

Tagged with algorithm, code, complexity, divide-and-conquer, equation, exponentiation, Fibonacci, matrix, recurrence, relation, time

Pollard Rho Brent Integer Factorization

with 10 comments

Pollard Rho is an integer factorization algorithm, which is quite fast for large numbers. It is based on Floyd's cycle-finding algorithm and on the observation that two numbers x and y are congruent modulo p with probability 0.5 after $1.177\sqrt{p}$ numbers have been randomly chosen.

Algorithm

```
Input : A number N to be factorized
Output : A divisor of N
If x mod 2 is 0
        return 2

Choose random x and c
y = x
g = 1
while g=1
        x = f(x)
        y = f(f(y))
        g = gcd(x-y,N)
return g
```

Note that this algorithm may not find the factors and will return failure for composite n. In that case, use a different f(x) and try again. Note, as well, that this algorithm does not work when n is a prime number, since, in this case, d will be always 1. We choose f(x) = x*x + c. Here's a python implementation :

```
1  def pollardRho(N):
2          if N%2==0:
3                  return 2
4          x = random.randint(1, N-1)
5          y = x
6          c = random.randint(1, N-1)
7          g = 1
8          while g==1:
9                  x = ((x*x)%N+c)%N
10                 y = ((y*y)%N+c)%N
11                 y = ((y*y)%N+c)%N
12                 g = gcd(abs(x-y),N)
13         return g
```

In 1980, Richard Brent published a faster variant of the rho algorithm. He used the same core ideas as Pollard but a different method of cycle detection, replacing Floyd's cycle-finding algorithm with the related Brent's cycle finding method. It is quite faster than pollard rho. Here's a python implementation :

```
1  def brent(N):
2          if N%2==0:
3                  return 2
4          y,c,m = random.randint(1, N-1),random.randint(1, N-1),random.ran
5          g,r,q = 1,1,1
6          while g==1:
7                  x = y
8                  for i in range(r):
9                          y = ((y*y)%N+c)%N
10                 k = 0
11                 while (k<r and g==1):
12                         ys = y
```

```
                        for i in range(min(m,r-k)):
                                y = ((y*y)%N+c)%N
                                q = q*(abs(x-y))%N
                        g = gcd(q,N)
                        k = k + m
                r = r*2
        if g==N:
                while True:
                        ys = ((ys*ys)%N+c)%N
                        g = gcd(abs(x-ys),N)
                        if g>1:
                                break

        return g
```

-fR0DDY

Written by fR0DDY

September 18, 2010 at 11:51 PM

Posted in Algorithm, Programming

Tagged with algorithm, brent, cycle, integer, pollard, rho

Miller Rabin Primality Test

with 15 comments

Miller Rabin Primality Test is a probabilistic test to check whether a number is a prime or not. It relies on an equality or set of equalities that hold true for prime values, then checks whether or not they hold for a number that we want to test for primality.

Theory

1> Fermat's little theorem states that if p is a prime and $1 \le a < p$ then $a^{p-1} \equiv 1 \pmod{p}$.
2> If p is a prime and $x^2 \equiv 1 \pmod{p}$ or $(x-1)(x+1) \equiv 0 \pmod{p}$ then $x \equiv 1 \pmod{p}$ or $x \equiv -1 \pmod{p}$.
3> If n is an odd prime then n-1 is an even number and can be written as $2^s.d$. By Fermat's Little Theorem either $a^d \equiv 1 \pmod{n}$ or $a^{2^r \cdot d} \equiv -1 \pmod{n}$ for some $0 \le r \le$ s-1.
4> The Miller–Rabin primality test is based on the contrapositive of the above claim. That is, if we can find an a such that $a^d \not\equiv 1 \pmod{n}$ and $a^{2^r \cdot d} \not\equiv -1 \pmod{n}$ for all $0 \le r \le$ s-1 then a is witness of compositeness of n and we can say n is not a prime. Otherwise, n may be a prime.

5> We test our number N for some random a and either declare that N is definitely a composite or probably a prime. The probably that a composite number is returned as prime after k itereations is 4^{-k}.

Algorithm

Input :A number N to be tested and a variable iteration-the number of 'a' for which algorithm will test N.
Output :0 if N is definitely a composite and 1 if N is probably a prime.

```
Write N as 2^s.d
For each iteration
        Pick a random a in [1,N-1]
        x = a^d mod n
        if x =1 or x = n-1
                Next iteration
        for r = 1 to s-1
                x  = x^2 mod n
                if x = 1
                        return false
                if x = N-1
                        Next iteration
        return false
return true
```

Here's a python implementation :

```
1  import random
2  def modulo(a,b,c):
3          x = 1
4          y = a
5          while b>0:
6                  if b%2==1:
7                          x = (x*y)%c
8                  y = (y*y)%c
9                  b = b/2
10         return x%c
11
12 def millerRabin(N,iteration):
13         if N<2:
14                 return False
15         if N!=2 and N%2==0:
16                 return False
17
18         d=N-1
19         while d%2==0:
20                 d = d/2
21
22         for i in range(iteration):
23                 a = random.randint(1, N-1)
24                 temp = d
```

```
                x = modulo(a,temp,N)
                while (temp!=N-1 and x!=1 and x!=N-1):
                        x = (x*x)%N
                        temp = temp*2

                if (x!=N-1 and temp%2==0):
                        return False

        return True
```

-fR0DDY

Written by fR0DDY

September 18, 2010 at 12:23 PM

Posted in Algorithm, Programming

Tagged with algorithm, code, fermat, little, miller, primality, prime, probability, Python, rabin, test, theorem

Knuth–Morris–Pratt Algorithm (KMP)

with one comment

Knuth–Morris–Pratt algorithm is the most popular linear time algorithm for string matching. It is little difficult to understand and debug in real time contests. So most programmer's have a precoded KMP in their kitty.

To understand the algorithm, you can either read it from Introduction to Algorithms (CLRS) or from the wikipedia page. Here's a sample C++ code.

```
void preKMP(string pattern, int f[])
{
        int m = pattern.length(),k;
        f[0] = -1;
        for (int i = 1; i<m; i++)
        {
                k = f[i-1];
                while (k>=0)
                {
                        if (pattern[k]==pattern[i-1])
                                break;
                        else
                                k = f[k];
                }
                f[i] = k + 1;
```

```
        }
}

bool KMP(string pattern, string target)
{
        int m = pattern.length();
        int n = target.length();
        int f[m];

        preKMP(pattern, f);

        int i = 0;
        int k = 0;

        while (i<n)
        {
                if (k==-1)
                {
                        i++;
                        k = 0;
                }
                else if (target[i]==pattern[k])
                {
                        i++;
                        k++;
                        if (k==m)
                                return 1;
                }
                else
                        k=f[k];
        }
        return 0;
}
```

NJOY!
-fR0DDY

Written by fR0DDY

August 29, 2010 at 12:20 PM

Posted in Algorithm, Programming

Tagged with C, clrs, code, Knuth, linear, matching, morris, pratt, string, time

The Z Algorithm

with one comment

In this post we will discuss an algorithm for linear time string matching. It is easy to understand and code and is usefull in contests where you cannot copy paste code.

Let our string be denoted by S.
z[i] denotes the length of the longest substring of S that starts at i and is a prefix of S.
α denotes the substring.
r denotes the index of the last character of α and ***l*** denotes the left end of α.

To find whether a pattern(P) of length n is present in a target string(T) of length m, we will create a new string S = P$T where $ is a character present neither in P nor in T. The space taken is n+m+1 or O(m). We will compute z[i] for all i such that 0 < i < n+m+1. If z[i] is equal to n then we have found a occurrence of P at position i – n – 1. So we can all the occurrence of P in T in O(m) time. To calculate z[i] we will use the z algorithm.

The Z Algorithm can be read from the section 1.3-1.5 of book Algorithms on strings, trees, and sequences by Gusfield. Here is a sample C++ code.

```
bool zAlgorithm(string pattern, string target)
{
    string s = pattern + '$' + target ;
    int n = s.length();
    vector<int> z(n,0);

    int goal = pattern.length();
    int r = 0, l = 0, i;
    for (int k = 1; k<n; k++)
    {
        if (k>r)
        {
            for (i = k; i<n && s[i]==s[i-k]; i++);
            if (i>k)
            {
                z[k] = i - k;
                l = k;
                r = i - 1;
            }
        }
        else
        {
            int kt = k - l, b = r - k + 1;
            if (z[kt]>b)
            {
                for (i = r + 1; i<n && s[i]==s[i-k]; i++);
                z[k] = i - k;
                l = k;
                r = i - 1;
            }
        }
        if (z[k]==goal)
```

```
33                  return true;
34          }
35          return false;
36  }
```

NJOY!
-fR0D

Written by fR0DDY

August 29, 2010 at 12:03 AM

Posted in Algorithm, Programming

Tagged with algorithm, C, code, complexity, linear, matching, string, time

COME ON CODE ON

Blog at WordPress.com. The Journalist v1.9 Theme.

- Follow

Follow “COME ON CODE ON”