# COME ON CODE ON

A blog about programming and more programming.

## Combination

with 16 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)!)**

```cpp
#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.

```cpp
#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)) % MOD)) % 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>

/*
    Time Complexity: O(n*r)
    Space Complexity: O(r)
*/
long long C(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];
}

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

**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
```

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

long long C(int n, int r)
{
    if (r==0) return 1;
    else return C(n-1,r-1) * n / r;
}

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

**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.

```cpp
#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 >
Matrix< T >::Matrix( const Matrix< T > &A )
{
    this->m = A.m;
    this->n = A.n;
    data = new T[m*n];
    for( int i = 0; i < m * n; i++ )
        data[i] = A.data[i];
}

template< class T >
Matrix< T >::~Matrix()
```

```
{
    delete [] data;
}

template< class T >
const Matrix< T > &Matrix< T >::operator=( const Matrix< T > &A )
{
    if( &A != this )
    {
        delete [] data;
        m = A.m;
        n = A.n;
        data = new T[m*n];
        for( int i = 0; i < m * n; i++ )
            data[i] = A.data[i];
    }
    return *this;
}

template< class T >
const Matrix< T > Matrix< T >::operator*( const Matrix< T > &A )
{
    Matrix C( m, A.n );
    for( int i = 0; i < m; ++i )
        for( int j = 0; j < A.n; ++j )
        {
            C.data[i*C.n+j]=0;
            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*A.n+j])%MOD)%MOD;
        }
    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;
}

long long C(int n, int r)
```

```
{
    Matrix<long long> M(r+1,r+1);
    for (int i=0;i<(r+1)*(r+1);i++)
        M.data[i]=0;
    M.data[0]=1;
    for (int i=1;i<r+1;i++)
    {
        M.data[i*(r+1)+i-1]=1;
        M.data[i*(r+1)+i]=1;
    }
    return (M^n).data[r*(r+1)];
}

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

**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.

```
#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>0)
    {
        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;
}

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*.

```
int factMOD(int n, int MOD)
{
    long long res = 1;
    while (n > 1)
    {
        long long cur = 1;
        for (int i=2; i<MOD; ++i)
            cur = (cur * i) % MOD;
        res = (res * powmod (cur, n/MOD, MOD)) % MOD;
        for (int i=2; i<=n%MOD; ++i)
            res = (res * i) % MOD;
        n /= MOD;
    }
    return int (res % MOD);
}
```

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:

```
long long factMOD(int n, int MOD)
{
    long long res = 1;
    while (n > 1)
    {
        res = (res * pow(MOD - 1, n/MOD, MOD)) % MOD;
        for (int i=2, j=n%MOD; i<=j; i++)
            res = (res * i) % MOD;
        n/=MOD;
    }
    return res;
}
```

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:

```
long long factMOD(int n, int MOD)
{
    long long res = 1;
    while (n > 0)
    {
        for (int i=2, m=n%MOD; i<=m; i++)
            res = (res * i) % MOD;
        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;
    while (n > 0)
    {
        for (int i=2, m=n%MOD; i<=m; i++)
            res = (res * i) % MOD;
        if ((n/=MOD)%2 > 0)
            res = MOD - res;
    }
    return res;
```

```
}

long long C(int n, int r, int MOD)
{
    if (countFact(n, MOD) > countFact(r, MOD) + countFact(n-r, MOD))
        return 0;

    return (factMOD(n, MOD) *
            ((InverseEuler(factMOD(r, MOD), MOD) *
            InverseEuler(factMOD(n-r, MOD), MOD)) % MOD)) % MOD;
}

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

-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

# 16 Responses

Subscribe to comments with RSS.

are you calculating the time and space complexity on the basis of code you are writing
or by using the calculus proofs or simply through wikipedia

**gauravalgo**

August 3, 2011 at 12:22 AM

Reply

On the basis of code.

**fR0DDY**

August 22, 2011 at 3:00 PM

Reply

i think i must congratulate fr0ddy for this beautiful technical blog but then
there are beginners who do not understand this article at first site so i recommend them to
go through this writeup of euclid algorithm and modular inverse

http://algosolver.blogspot.com/2011/09/euclid-algorithm-and-its-applications.html

**gauravalgo**

September 7, 2011 at 4:55 PM

Reply

Awesome post. I use this post as a cheat book for all my programming contests 😉

**Suraj Chandran**

February 5, 2012 at 11:20 PM

Reply
Just to make this blog better – you use header and printf/scanf functions for IO.. Thanks.

**valker**

June 29, 2012 at 12:57 AM

Reply
Hi
Can you provide algorithm to calculate sum of digits in 2^1000??
Thanks.

**minoz**

August 20, 2012 at 2:34 AM

Reply
Hi Froddy,

This is a very nice post.

I just found a small bug in one of your code.
In point 1. Using factorials .
The code written below this
"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."

Here in function "long long C(int n, int r, int MOD)"
at line 34 it should be
vector f(n+1,1); instead of vector f(n,1);

**Gaurav**

February 3, 2013 at 3:06 AM

Reply

Thanks for pointing that out. Edited.

**fR0DDY**

February 3, 2013 at 9:38 AM

Reply

Can you tell me which method would you prefer for a general question with a n as high as 10^9 ?

**mayanknatani**

May 31, 2013 at 4:37 PM

Reply

how we calculate a^(-1)%mod , if mod is not a prime number ?

**Avneet**

June 5, 2013 at 12:49 AM

Reply

See this post: https://comeoncodeon.wordpress.com/2011/10/09/modular-multiplicative-inverse/

**fR0DDY**

June 5, 2013 at 8:26 AM

Reply

Use Chinese Remainder Theorem

**fR0DDY**

June 17, 2013 at 8:29 PM

Reply

**How do I find nCr % m in the most efficient way, where n and r are very large integers?**

I think you can go through the following links, they contain efficient methods to calculate [math]\binom{n}{r} \% m[/math], most of them make use of the Lucas Theorem. [1]: Best known algos for calculating nCr % M [2]: This one is a question which fe…

**Quora**

April 26, 2015 at 7:03 PM

Reply

**What is an efficient algorithm to find the factorial of huge numbers which lie in the range 100000-1000000?**

A2A In JAVA, you can use BigInteger. There are certain BigInt libraries in C++. Without the libraries, the basic procedure is to create an array of 200-300 numbers and perform elementary multiplication. You can use the tutorial here: here:http://discus…

**Quora**

June 17, 2015 at 11:24 AM

Reply

Hey Man great article,I have simple doubt how we will calculate c(n,r)%(p^a) ,and n,r,and (p^a) are very big numbers,thanks in advance

**Arun Kumar**

August 29, 2016 at 3:25 AM

Reply

[…] In many problems we need to calculate nCr%m where n, r and m are three positive integers. If the mod value m is a prime number then we can calculate nCr%m in different ways like using loops, using pascal’s triangle, using modular multiplicative inverse, using dp technique etc. This ways are described with source codes in this post. […]

**nCr%m when m is not prime and n and r is sufficiently large. | Life Coding**

May 22, 2017 at 11:17 AM

Reply