Last update: October 22, 2024 Translated From: e-maxx.ru

Binomial Coefficients

Binomial coefficients $\binom{n}{k}$ are the number of ways to select a set of k elements from n different elements without taking into account the order of arrangement of these elements (i.e., the number of unordered sets).

Binomial coefficients are also the coefficients in the expansion of $(a+b)^n$ (so-called binomial theorem):

$$(a+b)^n = \binom{n}{0}a^n + \binom{n}{1}a^{n-1}b + \binom{n}{2}a^{n-2}b^2 + \cdots + \binom{n}{k}a^{n-k}b^k + \cdots + \binom{n}{n}b^n$$

It is believed that this formula, as well as the triangle which allows efficient calculation of the coefficients, was discovered by Blaise Pascal in the 17th century. Nevertheless, it was known to the Chinese mathematician Yang Hui, who lived in the 13th century. Perhaps it was discovered by a Persian scholar Omar Khayyam. Moreover, Indian mathematician Pingala, who lived earlier in the 3rd. BC, got similar results. The merit of the Newton is that he generalized this formula for exponents that are not natural.

Calculation

Analytic formula for the calculation:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

This formula can be easily deduced from the problem of ordered arrangement (number of ways to select k different elements from n different elements). First, let's count the number of ordered selections of k elements. There are n ways to select the first element, $n-1$ ways to select the second element, $n-2$ ways to select the third element, and so on. As a result, we get the formula of the number of ordered arrangements: $n(n-1)(n-2)\cdots(n-k+1) = \frac{n!}{(n-k)!}$. We can easily move to unordered arrangements, noting that each unordered arrangement corresponds to exactly $k!$ ordered arrangements ($k!$ is the number of possible permutations of k elements). We get the final formula by dividing $\frac{n!}{(n-k)!}$ by $k!$.

Recurrence formula (which is associated with the famous "Pascal's Triangle"):

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

It is easy to deduce this using the analytic formula.

Note that for $n < k$ the value of $\binom{n}{k}$ is assumed to be zero.

Properties

Binomial coefficients have many different properties. Here are the simplest of them:

- Symmetry rule:

$$\binom{n}{k} = \binom{n}{n-k}$$

- Factoring in:

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$$

- Sum over k:

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

- Sum over n:

$$\sum_{m=0}^{n} \binom{m}{k} = \binom{n+1}{k+1}$$

- Sum over n and k:

$$\sum_{k=0}^{m} \binom{n+k}{k} = \binom{n+m+1}{m}$$

- Sum of the squares:

$$\binom{n}{0}^2 + \binom{n}{1}^2 + \cdots + \binom{n}{n}^2 = \binom{2n}{n}$$

- Weighted sum:

$$1\binom{n}{1} + 2\binom{n}{2} + \cdots + n\binom{n}{n} = n2^{n-1}$$

- Connection with the Fibonacci numbers:

$$\binom{n}{0} + \binom{n-1}{1} + \cdots + \binom{n-k}{k} + \cdots + \binom{0}{n} = F_{n+1}$$

Calculation

Straightforward calculation using analytical formula

The first, straightforward formula is very easy to code, but this method is likely to overflow even for relatively small values of n and k (even if the answer completely fit into some datatype, the calculation of the intermediate factorials can lead to overflow). Therefore, this method often can only be used with long arithmetic:

```
int C(int n, int k) {
    int res = 1;
    for (int i = n - k + 1; i <= n; ++i)
        res *= i;
    for (int i = 2; i <= k; ++i)
        res /= i;
    return res;
}
```

Improved implementation

Note that in the above implementation numerator and denominator have the same number of factors (k), each of which is greater than or equal to 1. Therefore, we can replace our fraction with a product k fractions, each of which is real-valued. However, on each step after multiplying current answer by each of the next fractions the answer will still be integer (this follows from the property of factoring in).

C++ implementation:

```
int C(int n, int k) {
    double res = 1;
    for (int i = 1; i <= k; ++i)
        res = res * (n - k + i) / i;
    return (int)(res + 0.01);
}
```

Here we carefully cast the floating point number to an integer, taking into account that due to the accumulated errors, it may be slightly less than the true value (for example, 2.99999 instead of 3).

Pascal's Triangle

By using the recurrence relation we can construct a table of binomial coefficients (Pascal's triangle) and take the result from it. The advantage of this method is that intermediate results never exceed the answer and calculating each new table element requires only one addition. The flaw is slow execution for large n and k if you just need a single value and not the whole table (because in order to calculate $\binom{n}{k}$ you will need to build a table of all $\binom{i}{j}, 1 \le i \le n, 1 \le j \le n$, or at least to $1 \le j \le \min(i, 2k)$). The time complexity can be considered to be $\mathcal{O}(n^2)$.

C++ implementation:

```
const int maxn = ...;
int C[maxn + 1][maxn + 1];
C[0][0] = 1;
for (int n = 1; n <= maxn; ++n) {
    C[n][0] = C[n][n] = 1;
    for (int k = 1; k < n; ++k)
        C[n][k] = C[n - 1][k - 1] + C[n - 1][k];
}
```

If the entire table of values is not necessary, storing only two last rows of it is sufficient (current n-th row and the previous $n-1$-th).

Calculation in $O(1)$

Finally, in some situations it is beneficial to precompute all the factorials in order to produce any necessary binomial coefficient with only two divisions later. This can be advantageous when using long arithmetic, when the memory does not allow precomputation of the whole Pascal's triangle.

Computing binomial coefficients modulo m

Quite often you come across the problem of computing binomial coefficients modulo some m.

Binomial coefficient for small n

The previously discussed approach of Pascal's triangle can be used to calculate all values of $\binom{n}{k} \bmod m$ for reasonably small n, since it requires time complexity $\mathcal{O}(n^2)$. This approach can handle any modulo, since only addition operations are used.

Binomial coefficient modulo large prime

The formula for the binomial coefficients is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

so if we want to compute it modulo some prime $m > n$ we get

$$\binom{n}{k} \equiv n! \cdot (k!)^{-1} \cdot ((n-k)!)^{-1} \mod m.$$

First we precompute all factorials modulo m up to $\mathrm{MAXN!}$ in $O(\mathrm{MAXN})$ time.

```
factorial[0] = 1;
for (int i = 1; i <= MAXN; i++) {
    factorial[i] = factorial[i - 1] * i % m;
}
```

And afterwards we can compute the binomial coefficient in $O(\log m)$ time.

```
long long binomial_coefficient(int n, int k) {
    return factorial[n] * inverse(factorial[k] * factorial[n - k] % m) % m;
}
```

We even can compute the binomial coefficient in $O(1)$ time if we precompute the inverses of all factorials in $O(\mathrm{MAXN} \log m)$ using the regular method for computing the inverse, or even in $O(\mathrm{MAXN})$ time using the congruence $(x!)^{-1} \equiv ((x-1)!)^{-1} \cdot x^{-1}$ and the method for computing all inverses in $O(n)$.

```
long long binomial_coefficient(int n, int k) {
    return factorial[n] * inverse_factorial[k] % m * inverse_factorial[n - k] % m;
}
```

Binomial coefficient modulo prime power

Here we want to compute the binomial coefficient modulo some prime power, i.e. $m = p^b$ for some prime p. If $p > \max(k, n-k)$, then we can use the same method as described in the previous section. But if $p \leq \max(k, n-k)$, then at least one of $k!$ and $(n-k)!$ are not coprime with m, and therefore we cannot compute the inverses - they don't exist. Nevertheless we can compute the binomial coefficient.

The idea is the following: We compute for each $x!$ the biggest exponent c such that p^c divides $x!$, i.e. $p^c \mid x!$. Let $c(x)$ be that number. And let $g(x) := \frac{x!}{p^{c(x)}}$. Then we can write the binomial coefficient as:

$$\binom{n}{k} = \frac{g(n)p^{c(n)}}{g(k)p^{c(k)}g(n-k)p^{c(n-k)}} = \frac{g(n)}{g(k)g(n-k)}p^{c(n)-c(k)-c(n-k)}$$

The interesting thing is, that $g(x)$ is now free from the prime divisor p. Therefore $g(x)$ is coprime to m, and we can compute the modular inverses of $g(k)$ and $g(n-k)$.

After precomputing all values for g and c, which can be done efficiently using dynamic programming in $\mathcal{O}(n)$, we can compute the binomial coefficient in $O(\log m)$ time. Or precompute all inverses and all powers of p, and then compute the binomial coefficient in $O(1)$.

Notice, if $c(n) - c(k) - c(n-k) \geq b$, than $p^b \mid p^{c(n)-c(k)-c(n-k)}$, and the binomial coefficient is 0.

Binomial coefficient modulo an arbitrary number

Now we compute the binomial coefficient modulo some arbitrary modulus m.

Let the prime factorization of m be $m = p_1^{e_1} p_2^{e_2} \cdots p_h^{e_h}$. We can compute the binomial coefficient modulo $p_i^{e_i}$ for every i. This gives us h different congruences. Since all moduli $p_i^{e_i}$ are coprime, we can apply the Chinese Remainder Theorem to compute the binomial coefficient modulo the product of the moduli, which is the desired binomial coefficient modulo m.

Binomial coefficient for large n and small modulo

When n is too large, the $\mathcal{O}(n)$ algorithms discussed above become impractical. However, if the modulo m is small there are still ways to calculate $\binom{n}{k} \bmod m$.

When the modulo m is prime, there are 2 options:

- Lucas's theorem can be applied which breaks the problem of computing $\binom{n}{k} \bmod m$ into $\log_m n$ problems of the form $\binom{x_i}{y_i} \bmod m$ where $x_i, y_i < m$. If each reduced coefficient is calculated using precomputed factorials and inverse factorials, the complexity is $\mathcal{O}(m + \log_m n)$.
- The method of computing factorial modulo P can be used to get the required g and c values and use them as described in the section of modulo prime power. This takes $\mathcal{O}(m \log_m n)$.

When m is not prime but square-free, the prime factors of m can be obtained and the coefficient modulo each prime factor can be calculated using either of the above methods, and the overall answer can be obtained by the Chinese Remainder Theorem.

When m is not square-free, a generalization of Lucas's theorem for prime powers can be applied instead of Lucas's theorem.

Practice Problems

- Codechef - Number of ways
- Codeforces - Curious Array
- LightOj - Necklaces
- HACKEREARTH: Binomial Coefficient
- SPOJ - Ada and Teams
- SPOJ - Greedy Walking
- UVa 13214 - The Robot's Grid
- SPOJ - Good Predictions
- SPOJ - Card Game
- SPOJ - Topper Rama Rao
- UVa 13184 - Counting Edges and Graphs
- Codeforces - Anton and School 2
- Codeforces - Bacterial Melee
- Codeforces - Points, Lines and Ready-made Titles
- SPOJ - The Ultimate Riddle
- CodeChef - Long Sandwich
- Codeforces - Placing Jinas

References

- Blog fishi.devtail.io
- Question on Mathematics StackExchange
- Question on CodeChef Discuss