GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: COEFICIENTES BINOMIALES

1. Introducción

Los coeficientes binomiales $\binom{n}{k}$ son la cantidad de formas de seleccionar un conjunto de elementos de K de n elementos diferentes sin tener en cuenta el orden de acuerdo de estos elementos (es decir, el número de conjuntos desordenados). Por ejemplo, $\binom{5}{3} = 10$, porque el conjunto $\{1, 2, 3, 4, 5\}$ tiene 10 subconjuntos de 3 elementos:

$$\{1,2,3\}, \{1,2,4\}, \{1,2,5\}, \{1,3,4\}, \{1,3,5\}, \{1,4,5\}, \{2,3,4\}, \{2,3,5\}, \{2,4,5\}, \{3,4,5\}$$

De como calcularlos y su utilizacion en la solución de problemas de programación competitiva abordará la siguiente guía.

2. Conocimientos previos

2.1. Triángulo de Pascal

En las matemáticas, el triángulo de Pascal es una representación de los coeficientes binomiales ordenados en forma de triángulo. Es llamado así en honor al filósofo y matemático francés Blaise Pascal, quien introdujo esta notación en 1654, en su Tratado del triángulo aritmético. Si bien las propiedades y aplicaciones del triángulo las conocieron matemáticos indios, chinos, persas, alemanes e italianos antes del triángulo de Pascal, fue Pascal quien desarrolló muchas de sus aplicaciones y el primero en organizar la información de manera conjunta.

$$\begin{array}{lccccccccc}
n = 0\text{:} & & & & & 1 & & & & \\
n = 1\text{:} & & & & 1 & & 1 & & & \\
n = 2\text{:} & & & 1 & & 2 & & 1 & & \\
n = 3\text{:} & & 1 & & 3 & & 3 & & 1 & \\
n = 4\text{:} & 1 & & 4 & & 6 & & 4 & & 1
\end{array}$$

2.2. Aritmética larga

La aritmética de precisión arbitraria, también conocida como *bignum* o simplemente *aritmética larga*, es un conjunto de estructuras de datos y algoritmos que permiten procesar números mucho mayores de los que pueden caber en tipos de datos estándar. A continuación se presentan varios tipos de aritmética de precisión arbitraria.

2.3. Inverso multiplicativo

Un inverso multiplicativo modular de un número entero a es un numero entero x tal que $a \cdot x$ es congruente con 1 modular algún módulo m. Para escribirlo de forma formal: queremos encontrar un número entero x de modo que:

$$a \cdot x \equiv 1 \mod m.$$

3. Desarrollo

Los coeficientes binomiales también son los coeficientes en la expansión de $(a+b)^n$ (el llamado teorema binomial):

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

Se cree que esta fórmula, así como el triángulo que permite un cálculo eficiente de los coeficientes, fue descubierta por Blaise Pascal en el siglo XVII. Sin embargo, era conocido por el matemático chino Yang Hui, que vivía en el siglo XIII. Quizás fue descubierto por un erudito persa Omar Khayyam. Además, el matemático indio Pingala, que vivió antes en el 3er. AC, obtuvo resultados similares. El mérito del Newton es que generalizó esta fórmula para los exponentes que no son naturales.

Hay dos fórmulas para los números catalanes: **recursiva** y **analítica**. Dado que creemos que todos los problemas donde se utilizan son equivalentes (tienen la misma solución), para la prueba de las fórmulas siguientes elegiremos la tarea que sea más fácil de realizar.

3.1. Fórmula analítica

La fórmula analítica para el cálculo es:

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

Esta fórmula se puede deducir fácilmente del problema del acuerdo ordenado (número de formas de seleccionar k diferentes elementos de n diferentes elementos). Primero, cuentemos el número de selecciones ordenadas de k elementos. Hay n formas de seleccionar el primer elemento, $N-1$ formas de seleccionar el segundo elemento, $n-2$ formas de seleccionar el tercer elemento, y así sucesivamente. Como resultado, obtenemos la fórmula del número de arreglos ordenados: $n\times(n-1)\times(n-2)\times\ldots\times(n-k+1) = \frac{n!}{(n-k)!}$. Podemos pasar fácilmente a los arreglos desordenados, señalando que cada acuerdo desordenado corresponde a los arreglos ordenados exactamente por $k!$ ($k!$ Es el número de permutaciones posibles de los elementos de k). Obtenemos la fórmula final dividiendo $\frac{n!}{(n-k)!}$ por $k!$.

Hay $n!$ Permutaciones de n elementos. Revisamos todas las permutaciones y siempre incluimos los primeros k elementos de la permutación en el subconjunto. Dado que el orden de los elementos en el subconjunto y fuera del subconjunto no importa, el resultado se divide por $k!$ y $(n-k)!$

3.2. Fórmula recursiva

Los coeficientes binomiales se pueden calcular de manera recursiva de la siguiente manera:

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

que está asociado con el famoso Triángulo de Pascal. Es fácil deducir esto utilizando la fórmula analítica. La idea es agregar un elemento x en el conjunto. Si se incluye x en el subconjunto, tenemos que elegir elementos $k-1$ de los elementos $n-1$, y si x no está incluido en el subconjunto, tenemos que elegir k elementos de los elementos $n-1$.

Tenga en cuenta que para $n < k$ se supone que el valor de $\binom{n}{k}$ es cero.

La idea es fijar un elemento x en el conjunto. Si x está incluido en el subconjunto, tenemos que elegir $k-1$ elementos de $n-1$ elementos, y si x no está incluido en el subconjunto, tenemos que elegir k elementos de $n-1$ elementos.

Los casos base para la recursividad son:

$$\binom{n}{0} = \binom{n}{n} = 1$$

porque siempre hay exactamente una manera de construir un subconjunto vacío y un subconjunto que contenga todos los elementos.

3.3. Propiedades

Los coeficientes binomiales tienen muchas propiedades diferentes. Aquí están las más simples:

1. Regla de simetría: $\binom{n}{k} = \binom{n}{n-k}$
2. Teniendo en cuenta: $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$
3. Suma sobre k : $\sum_{k=0}^{n} \binom{n}{k} = 2^n$
4. Suma sobre n:$\sum_{m=0}^{n} \binom{m}{k} = \binom{n+1}{k+1}$
5. Suma de n y k: $\sum_{k=0}^{m} \binom{n+k}{k} = \binom{n+m+1}{m}$
6. Suma de los cuadrados: $\binom{n}{0}^2 + \binom{n}{1}^2 + \cdots + \binom{n}{n}^2 = \binom{2n}{n}$
7. Suma ponderada: $1\binom{n}{1} + 2\binom{n}{2} + \cdots + n\binom{n}{n} = n2^{n-1}$
8. Conexión con los números de Fibonacci: $\binom{n}{0} + \binom{n-1}{1} + \cdots + \binom{n-k}{k} + \cdots + \binom{0}{n} = F_{n+1}$

3.4. Cálculo

3.4.1. Cálculo sencillo mediante fórmula analítica

La primera fórmula sencilla es muy fácil de codificar, pero es probable que este método se desborde incluso para valores relativamente pequeños de n y k (incluso si la respuesta encaja completamente en algún tipo de datos, el cálculo de los factoriales intermedios puede provocar un desbordamiento). Por lo tanto, este método a menudo sólo se puede utilizar con aritmética larga.

3.4.2. Implementación mejorada

Tenga en cuenta que en la idea anterior el numerador y el denominador tienen el mismo número de factores (k), cada uno de los cuales es mayor o igual a 1. Por lo tanto, podemos reemplazar nuestra fracción con un producto k fracciones, cada una de las cuales tiene un valor real. Sin embargo, en cada paso después de multiplicar la respuesta actual por cada una de las siguientes fracciones, la respuesta seguirá siendo un número entero (esto se desprende de la propiedad de factorizar).

Aquí convertimos cuidadosamente el número de coma flotante a un número entero, teniendo en cuenta que debido a los errores acumulados, puede ser ligeramente menor que el valor real (por ejemplo, $2{,}99999$ en lugar de 3).

3.4.3. Triángulo de Pascal

Utilizando la relación de recurrencia podemos construir una tabla de coeficientes binomiales (triángulo de Pascal) y obtener el resultado de ella. La ventaja de este método es que los resultados intermedios nunca exceden la respuesta y el cálculo de cada nuevo elemento de la tabla requiere solo una suma. El defecto es la ejecución lenta para grandes n y k si solo necesita un valor único y no toda la tabla (porque para calcular $\binom{n}{k}$ necesitarás construir una tabla de todos $\binom{i}{j}, 1 \le i \le n, 1 \le j \le n$, o al menos a $1 \le j \le \min(i, 2k)$).

3.4.4. Cálculo en O(1)

Finalmente, en algunas situaciones es beneficioso precalcular todos los factoriales para producir cualquier coeficiente binomial necesario con sólo dos divisiones después. Esto puede resultar ventajoso cuando se utiliza aritmética larga , cuando la memoria no permite el cálculo previo de todo el triángulo de Pascal.

3.5. Calcular coeficientes binomiales con módulo m

Muy a menudo te encuentras con el problema de calcular coeficientes binomiales módulo algunos m.

3.5.1. Coeficiente de binomial para pequeños n

El enfoque previamente discutido del triángulo de Pascal se puede utilizar para calcular todos los valores de $\binom{n}{k} \bmod m$ por un tamaño razonablemente pequeño n, ya que requiere complejidad de tiempo O(n^2). Este enfoque puede manejar cualquier módulo, ya que solo se utilizan operaciones de suma.

3.5.2. Coeficiente binomial módulo primo grande

La fórmula para los coeficientes binomiales es:

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

entonces si queremos calcularlo módulo algún primo $m > n$ obtenemos:

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

Primero calculamos previamente todos los módulos factoriales m hasta MAXN! en O(MAXN) tiempo. Y luego podemos calcular el coeficiente binomial en O($\log m$) tiempo. Incluso podemos calcular el coeficiente binomial en $O(1)$ tiempo si precalculamos los inversos multiplicativos de todos los factoriales en O(MAXN $\log m$) utilizando el método habitual para calcular el inverso multiplicativo, o incluso en O(MAXN) tiempo usando la congruencia $(x!)^{-1} \equiv ((x-1)!)^{-1} \cdot x^{-1}$ y el método para calcular todas todos los inversos en O(n).

3.5.3. Coeficiente binomial módulo potencia prima

Aquí queremos calcular el coeficiente binomial módulo de alguna potencia prima, es decir $m = p^b$ por alguna prima p. Si $p > \max(k, nk)$, entonces podemos usar el mismo método descrito en la sección anterior. Pero si $p \leq \max(k, nk)$, entonces al menos uno de $k!$ y $(nk)!$ no son coprimos con m, y por lo tanto no podemos calcular las inversas: no existen. Sin embargo, podemos calcular el coeficiente binomial.

La idea es la siguiente: Calculamos para cada $x!$ el mayor exponente c tal que p^c divide $x!$, es decir $p^c \mid x!$. Dejar $c(x)$ sea es número. Y deja $g(x) := \frac{x!}{p^{c(x)}}$. Entonces podemos escribir el coeficiente binomial como:

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

Lo interesante es que $g(x)$ ahora está libre del divisor primo p. Por lo tanto $g(x)$ es coprimo de m, y podemos calcular los inversos modulares de $g(k)$ y $g(nk)$.

Después de precalcular todos los valores para g y c, que se puede hacer de manera eficiente usando programación dinámica en O(n), podemos calcular el coeficiente binomial en $O(\log m)$ tiempo. O precalcular todas las inversas y todas las potencias de p y luego calcular el coeficiente binomial en $O(1)$.

Aviso, si $c(n) - c(k) - c(nk) \geq b$, que $p^b \mid p^{c(n)-c(k)-c(nk)}$, y el coeficiente binomial es 0

3.5.4. Módulo de coeficiente binomial un número arbitrario

Ahora calculamos el módulo del coeficiente binomial algún módulo arbitrario m.

Sea la factorización prima de m ser $m = p_1^{e_1} p_2^{e_2} \cdots p_h^{e_h}$. Podemos calcular el módulo del coeficiente binomial $p_i^{e_i}$ para cada i. esto nos da h diferentes congruencias. Dado que todos los módulos $p_i^{e_i}$ son coprimos, podemos aplicar el teorema chino del resto para calcular el módulo del coeficiente binomial, el producto de los módulos, que es el módulo del coeficiente binomial deseado m.

3.5.5. Coeficiente binomial para grandes n y módulo pequeño

Cuando n es demasiado grande, el O(n) los algoritmos discutidos anteriormente se vuelven poco prácticos. Sin embargo, si el módulo m es pequeño todavía hay maneras de calcular $\binom{n}{k} \bmod m$ cuando el módulo m es primo, hay 2 opciones:

- Se puede aplicar el teorema de Lucas, lo que resuelve el problema de la informática $\binom{n}{k} \bmod m$ en $\log_m n$ problemas de la forma $\binom{x_i}{y_i} \bmod m$ dónde $x_i, y_i < m$. Si cada coeficiente reducido se calcula utilizando factoriales precalculados y factoriales inversos, la complejidad es O($m+\log_m n$).
- El método de calcular el módulo factorial P se puede utilizar para obtener el valor requerido g y c valores y utilícelos como se describe en la sección de módulo de potencia primaria esto toma O($m \log_m n$).

Cuando m no está libre de cuadrados, se puede aplicar una generalización del teorema de Lucas para potencias primas en lugar del teorema de Lucas.

Cuando m no es primo sino libre de cuadrados, los factores primos de m se puede obtener y el módulo de coeficiente de cada factor primo se puede calcular utilizando cualquiera de los métodos anteriores, y la respuesta general se puede obtener mediante el teorema del resto chino.

4. Implementación

4.1. C++

4.1.1. Fórmula analítica

```
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;
}

int Cv2(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);
}
```

4.1.2. Triangulo de Pascal

```
const int maxn = 1000;
int pt[maxn + 1][maxn + 1];

void pascalTriangle(){
    pt[0][0] = 1;
    for (int n = 1; n <= maxn; ++n) {
        pt[n][0] = pt[n][n] = 1;
        for (int k = 1; k < n; ++k)
            pt[n][k] = pt[n - 1][k - 1] + pt[n - 1][k];
    }
}

int C(int n, int k) {
    return pt[n][k];
}
```

4.1.3. Coeficiente binomial módulo primo grande

```
#define int long long
int factorial[MAXN+10],m;

//precalculamos todos los factoriales
void calculateFactorial(){
    factorial[0] = 1;
    for (int i = 1; i <= MAXN; i++) {
        factorial[i] = factorial[i - 1] * i  % m;
    }
}

int inverse(int a) {
    return a <= 1 ? a : m - (int)(m/a) * inverse(m  % a)  % m;
}

//Y luego podemos calcular el coeficiente binomial.
```

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

Incluso podemos calcular el coeficiente binomial en de forma más eficiente tiempo si precalculamos los inversos de todos los factoriales usando el método regular para calcular el inverso, usando la congruencia $(x!)^{-1} \equiv ((x-1)!)^{-1} \cdot x^{-1}$.

```
#define int long long
int m;
vector<int> factorial, inverse_factorial;

int gcd(int a, int b, int& x, int& y) {
   x = 1, y = 0;
   int x1 = 0, y1 = 1, a1 = a, b1 = b;
   while (b1) {
      int q = a1 / b1;
      tie(x, x1) = make_tuple(x1, x - q * x1); tie(y, y1) = make_tuple(y1, y -
          q * y1);
      tie(a1, b1) = make_tuple(b1, a1 - q * b1);
   }
   return a1;
}

vector<int> invs(const std::vector<int> &a, int m) {
   int n = a.size();
   if (n == 0) return {};
   vector<int> b(n);
   int v = 1;
   for (int i = 0; i != n; ++i) {
      b[i] = v; v = static_cast<int>(v) * a[i] % m;
   }
   int x, y;
   gcd(v, m, x, y);
   x = (x % m + m) % m;
   for (int i = n - 1; i >= 0; --i) {
      b[i] = static_cast<int>(x) * b[i] % m;
      x = static_cast<int>(x) * a[i] % m;
   }
   return b;
}

void calculateFactorial(){
   factorial.resize(MAX_N+5); factorial[0] = 1;
   for (int i = 1; i <= MAX_N; i++) {
      factorial[i] = factorial[i - 1] * i % m;
   }
   inverse_factorial = invs(factorial,m);
}
```

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

4.2. Java

4.2.1. Fórmula analítica

```
public static 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;
}

public static int Cv2(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);
}
```

4.2.2. Triangulo de Pascal

```
public class Main {
   private final int MAX_N = 1000;
   private long [][] tp;

   public void pascalTriangle(){
      tp = new long [MAX_N+3][MAX_N+3];
      tp[0][0] = 1;
      for (int n = 1; n <= MAX_N; ++n) {
          tp[n][0] = tp[n][n] = 1;
          for(int k=1;k<n;++k) tp[n][k] = tp[n-1][k-1] + tp[n-1][k];
      }
   }

   public long C(int n, int k) { return tp[n][k]; }
}
```

4.2.3. Coeficiente binomial módulo primo grande

```
public class Main {
   private final int MAX_N = 1000010;
```

```
    private long [] factorial;
    private long m;

    //precalculamos todos los factoriales
    public void calculateFactorial(){
        factorial = new long [MAX_N+5];
        factorial[0] = 1;
        for (int i = 1; i <= MAX_N; i++) {
            factorial[i] = factorial[i - 1] * i % m;
        }
    }

    public long inverse(long a) {
        return a <= 1 ? a : m - (int)(m/a) * inverse(m % a) % m;
    }

    //Y luego podemos calcular el coeficiente binomial.
    public long binomial_coefficient(int n, int k) {
        return factorial[n] * inverse(factorial[k] * factorial[n - k] % m) % m;
    }
}
```

5. Aplicaciones

Dentro de las aplicaciones de los coeficientes binomiales podemos citar:

1. **Probabilidad:** Los coeficientes binomiales se utilizan en la teoría de la probabilidad para calcular la probabilidad de eventos en experimentos binomiales, como el lanzamiento de una moneda o el lanzamiento de un dado.
2. **Teorema del binomio:** Los coeficientes binomiales son fundamentales en el desarrollo del teorema del binomio, que establece cómo se expande una potencia de un binomio en una serie de términos.
3. **Combinatoria:** Los coeficientes binomiales se utilizan en problemas de combinatoria, como la cantidad de formas en que se pueden elegir k elementos de un conjunto de n elementos.
4. **Álgebra lineal:** Los coeficientes binomiales aparecen en la expansión de potencias de matrices y en la solución de sistemas lineales.
5. **Fórmulas de recurrencia:** Los coeficientes binomiales se utilizan en la formulación de fórmulas de recurrencia en matemáticas y ciencias de la computación.
6. **Estadística:** Los coeficientes binomiales se utilizan en la distribución binomial, que es fundamental en estadística para modelar experimentos con dos resultados posibles, como éxito o fracaso.

Además de las aplicaciones mencionadas anteriormente, los coeficientes binomiales también tienen aplicaciones en programación competitiva. En este contexto, se utilizan en la formulación

y optimización de algoritmos para resolver problemas matemáticos y de combinatoria de manera eficiente.

Por ejemplo, en programación competitiva, los coeficientes binomiales se pueden utilizar para calcular el número de formas en que se pueden colocar k elementos en n posiciones, lo cual es útil en la resolución de problemas de permutaciones y combinaciones.

También se utilizan en la optimización de algoritmos para resolver problemas de conteo y probabilidad, donde se requiere calcular el número de formas en que ciertos eventos pueden ocurrir.

6. Complejidad

La implementación del triángulo Pascal se puede considerar que la complejidad del tiempo es O (n^2), así que usar esta variante es mejor hacer un precalculo hasta mayor posible valor del $\binom{n}{k}$ para que la respuestas sean después en O(1)

La idea de precalcular todos los inversos de todos los factoriales hace posible calcular el coeficiente binomial en $O(1)$ tiempo si precalculamos los inversos de todos los factoriales en O(MAXN $\log m$) usando el método regular para calcular el inverso, o incluso en O(MAXN) tiempo usando la congruencia anteriormente citada y el método para calcular todas las inversas en $O(n)$.

7. Ejercicios

A continuacion una lista de problemas que su solución se puede hallar a partir de los elementos abordados en la presente guía:

- CSES - Binomial Coefficients
- CodeChef- Number of Ways
- CodeForces - C. Curious Array
- LightOJ - Necklace
- SPOJ - ADATEAMS - Ada and Teams
- SPOJ - UCV2013E - Greedy Walking
- UVA - 13214 - The Robot's Grid
- SPOJ - HC12 - Card Game
- UVA - 13184 - Counting Edges and Graphs
- CodeForces - E. Placing Jinas
- CodeChef - Long Sandwich
- SPOJ - DCEPC13D - The Ultimate Riddle
- CodeForces - E. Points, Lines and Ready-made Titles

- CodeForces - F. Bacterial Melee
- CodeForces - D. Anton and School - 2