GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ENTERO GRANDE (*BIG INTEGER*)

1. Introducción

La aritmética de precisión arbitraria, también conocida como *Big Num* o simplemente *aritmética larga*, es un conjunto de estructuras de datos y algoritmos que permite procesar números mucho mayores de los que caben en los tipos de datos estándar.

2. Conocimientos previos

2.1. Complemento a dos

El complemento a 2 es una forma de representar números negativos en el sistema binario. El complemento a dos de un número N , expresado en el sistema binario con n dígitos, se define como:

$C_2(N) = 2^n - N$

donde total de números positivos será $2^{n-1} - 1$ y el de negativos 2^{n-1}, siendo n el número de bits. El 0 contaría como positivo, ya que los positivos son los que empiezan por 0 y los negativos los que empiezan por 1.

2.2. Transformada rápida de Fourier

La transformada rápida de Fourier, conocida por la abreviatura FFT (del inglés Fast Fourier Transform) es un algoritmo eficiente que permite calcular la transformada de Fourier discreta (DFT) y su inversa. La FFT es de gran importancia en una amplia variedad de aplicaciones, desde el tratamiento digital de señales y filtrado digital en general a la resolución de ecuaciones en derivadas parciales o los algoritmos de multiplicación rápida de grandes enteros. Cuando se habla del tratamiento digital de señales, el algoritmo FFT impone algunas limitaciones en la señal y en el espectro resultante ya que la señal muestreada y que se va a transformar debe consistir de un número de muestras igual a una potencia de dos. La mayoría de los analizadores de FFT permiten la transformación de 512, 1024, 2048 o 4096 muestras. El rango de frecuencias cubierto por el análisis FFT depende de la cantidad de muestras recogidas y de la proporción de muestreo.

La transformada rápida de Fourier es de importancia fundamental en el análisis matemático y ha sido objeto de numerosos estudios. La aparición de un algoritmo eficaz para esta operación fue un hito en la historia de la informática.

2.3. Algoritmo de Karatsuba

El algoritmo de Karatsuba es un procedimiento para multiplicar números grandes eficientemente, que fue descubierto por Anatolii Alexeevitch Karatsuba en 1960 y publicado en 1962. El algoritmo consigue reducir la múltiplicación de dos números de n dígitos a como máximo $3n^{\log_2 3} \approx 3n^{1,585}$ multiplicaciones de un dígito. Es, por lo tanto, más rápido que el algoritmo clásico, que requiere n^2 productos de un dígito. Si $n = 2^10 = 1024$, en particular, el cómputo final exacto es $3^10 = 59{,}049 y (2^10)^2 = 1{,}048{,}576$, respectivamente.

El algoritmo de Karatsuba es un claro ejemplo del paradigma divide y vencerás, concretamente del algoritmo de partición binaria.

2.4. Teorema del resto chino

El teorema chino del resto es un resultado sobre congruencias en teoría de números y sus generalizaciones en álgebra abstracta. Fue publicado por primera vez en el siglo III por el matemático chino Sun Tzu.

3. Desarrollo

La idea principal es que el número se almacene como un arreglo o vector de sus dígitos en alguna base. Varias de las bases más utilizadas son decimales, potencias de decimales (10^4 o 10^9) y binarias. Para el caso de trabajar con enteros negativos es preferible usar la representación de enteros en complemento a dos.

Las operaciones con números en esta forma se realizan utilizando algoritmos escolares de suma, resta, multiplicación y división de columnas. También es posible utilizar algoritmos de multiplicación rápida: transformada rápida de Fourier y algoritmo de Karatsuba.

Aquí describimos la aritmética larga para cada una de las operaciones mencionadas anteriormente.

3.1. Estructura de datos

Guardaremos los números como un vector<int>, en el que cada elemento es un solo dígito del número. Para saber el signo del numero podemos definir un entero cuyo valor sea 1 o -1 dependiendo de que si es un numero positivo o negativo.

Para mejorar el rendimiento, usaremos 10^9 como base, de modo que cada dígito del número largo contenga 9 dígitos decimales a la vez.

Los dígitos se almacenarán en orden de menor a mayor. Todas las operaciones se implementarán de manera que después de cada una de ellas el resultado no lleve ceros a la izquierda, siempre que los operandos tampoco lleven ceros a la izquierda. Todas las operaciones que puedan dar como resultado un número con ceros a la izquierda deben ir seguidas de un código que los elimine. Tenga en cuenta que en esta representación hay dos notaciones válidas para el número cero: un vector vacío y un vector con un solo dígito cero.

3.2. Impresión

Imprimir el entero largo es la operación más fácil. Primero imprimimos el signo del entero luego imprimimos el último elemento del vector (o 0 si el vector está vacío), seguido del resto de los elementos rellenados con ceros a la izquierda si es necesario para que tengan exactamente n dígitos de acuerdo la enésima potencia de 10 que fue seleccionada, en nuestro caso $n = 9$.

3.3. Lectura

Para leer un entero largo, lea su notación en una cadena y luego conviértala en dígitos, teniendo en cuenta la base seleccionada y que el primer elemento de la cadena puede indicar el signo del valor. Una vez leido el valor en forma de cadena debemos ir almacenando los digitos de dicho valor convertidos en la base seleccionada este proceso se comienza por los digitos mas a la derecha del valor.

3.4. Adicción

Para esta operación basta con implementar un algoritmo que simule el proceso aprendido en la escuela para esta operación teniendo en cuenta los signos de los valores que intervienen en la operaciones.

3.5. Subtracción

Para esta operación basta con implementar un algoritmo que simule el proceso aprendido en la escuela para esta operación teniendo en cuenta los signos de los valores que intervienen en la operaciones.

3.6. Multiplicación

Para esta operación bien pudieramos implementar un algoritmo que simules el proceso de multiplicación que aprendimos en la escuela pero el mismo sería muy complejo en cuanto a cantidad de operaciones a realizar es por eso que vamos a apoyarnos en el algoritmo de Karatsuba que permite la multiplicación de enteros largos de manera más eficiente.

3.7. División

Similar a lo que ocurre con la multiplicación implementar un algoritmo que simule el proceso que aprendemos en la escuela sería demasiado costoso por tanto vamos apoyarnos en el algoritmo ideado por Donald Kunth para esta operación. Lo interesante de este algoritmo es que permite calcular el cociente y resto de la operación, como se está trabjando con enteros se descarta parte decimal del resultado.

4. Implementación

4.1. C++

```
#define fst first
#define snd second
#define all(c) ((c).begin()), ((c).end())

struct bigint : vector<long long> {
// Si B es chiquito ==> tratos mas grandes/operaciones mas lentas
```

```
// B = 100000000  (10^8) ==> se trata de 10^645636
// B = 1000000000 (10^9) ==> se trata de  10^7378
   static constexpr long long B = 1000000000, W = log10(B);
   char sign;
   bigint& trim() {
      while (!empty() && !back()) pop_back();
      if (empty()) sign = 1;
      return *this;
   }

   bigint(long long v = 0) : sign(1) {
      if (v < 0) { sign = -1; v = -v; }
      while (v > 0) { push_back(v % B); v /= B; }
   }

   bigint(string s) : sign(+1) {
      long long e = 0;
      for (; e < s.size() && s[e] <= '0' || s[e] >= '9'; ++e)
         if (s[e] == '-') sign = -sign;
      for (long long i = s.size()-1; i >= e; i -= W) {
         push_back(0);
         for (long long j = max(e, i - W + 1); j <= i; ++j)
            back() = 10 * back() + (s[j] - '0');
      }
      trim();
   }

   bigint operator-() const {
      bigint x = *this;
      if (!empty()) x.sign = -x.sign;
      return x;
   }

   operator vector<long long>(); // proteger de lanzamientos inesperados
   bigint &operator+=(bigint x);
};

// entrada y salida
ostream &operator<<(ostream &os, const bigint &x) {
   if (x.sign < 0) os << '-';
   os << (x.empty() ? 0 : x.back());
   for (int i = (int)x.size()-2; i >= 0; --i)
      os << setfill('0') << setw(bigint::W) << x[i];
   return os;
}
istream &operator>>(istream &is, bigint &x) {
   string s; is >> s; x = bigint(s); return is;
}

bigint operator-(bigint x, bigint y);
```

```
bigint operator+(bigint x, bigint y) {
   if (x.sign != y.sign) return x - (-y);
   if (x.size() < y.size()) swap(x, y);
   int c = 0;
   for (int i = 0; i < y.size(); ++i) {
      x[i] += y[i] + c;
      c = (x[i] >= bigint::B);
      if (c) x[i] -= bigint::B;
   }
   if (c) x.push_back(c);
   return x;
}

bigint operator-(bigint x, bigint y) {
   if (x.sign != y.sign) return x + (-y);
   int sign = x.sign; x.sign = y.sign = +1;
   if (x < y) {swap(x, y); sign = -sign;}
   int c = 0;
   for (int i = 0; i < x.size(); ++i) {
       x[i] -= (i < y.size() ? y[i] : 0) + c;
       c = (x[i] < 0);
       if (c) x[i] += bigint::B;
   }
   x.sign = sign; return x.trim();
}

bigint operator*(bigint x, int y) {
   if (y == 0) return bigint(0);
   if (y < 0) { x.sign = -x.sign; y = -y; }
   int c = 0;
   for (int i = 0; i < x.size() || c; ++i) {
      if (i == x.size()) x.push_back(0);
      x[i] = x[i] * y + c;
      c = x[i] / bigint::B;
      if (c) x[i] %= bigint::B;
   }
   return x;
}
bigint operator*(int x, bigint y) { return y * x; }

// multiplicacion utilizando el algoritmo de Karatsuba
bigint operator*(bigint x, bigint y) {
   if (x.empty()||y.empty()) return bigint(0);
   function<void(long long*, int, long long*, int, long long*)>
      rec = [&](long long *x, int xn, long long *y, int yn, long long *z) {
         if (xn < yn) { auto tmp =xn; xn=yn; yn=tmp; auto tmp1 =x;x=y;y=tmp1;}
         fill(z, z+xn+yn, 0);
         if (yn <= 2) {
            for (int i = 0; i < xn; ++i)
               for (int j = 0; j < yn; ++j)
```

```
                z[i+j] += x[i] * y[j];
        } else {
           int m = (xn+1)/2, n = min(m, yn);
           for (int i = 0; i+m < xn; ++i) x[i] += x[i+m];
           for (int j = 0; j+m < yn; ++j) y[j] += y[j+m];
           rec(x, m, y, n, z+m);
           for (int i = 0; i+m < xn; ++i) x[i] -= x[i+m];
           for (int j = 0; j+m < yn; ++j) y[j] -= y[j+m];
           long long p[2*m+2]; rec(x, m, y, n, p);
           for (int i = 0; i < m+n; ++i){ z[i] += p[i]; z[i+m] -= p[i]; }
           rec(x+m, xn-m, y+m, yn-n, p);
           for (int i = 0; i < xn+yn-m-n; ++i){
              z[i+m] -= p[i]; z[i+2*m] += p[i];
           }
        }
   };
   bigint z; z.resize(x.size()+y.size()+2);
   z.sign = x.sign * y.sign;
   rec(&x[0], x.size(), &y[0], y.size(), &z[0]);
   for (int i = 0; i+1 < z.size(); ++i) {
      z[i+1] += z[i] / bigint::B;
      z[i] %= bigint::B;
      if(z[i] < 0){ z[i+1] -= 1; z[i] += bigint::B;}
   }
   return z.trim();
}

bigint pow(bigint x, int b) {
   bigint y(1);
   for (; b > 0; b /= 2) {
      if (b % 2) y = y * x;
      x = x * x;
   }
   return y;
}

pair<bigint, int> divmod(bigint x, int y) {
   if (y < 0) { x.sign = -x.sign; y = -y; }
   int rem = 0;
   for (int i = x.size()-1; i >= 0; --i) {
      x[i] += rem * bigint::B; rem = x[i] % y;
      x[i] /= y;
   }
   return {x.trim(), rem};
}

bigint operator/(bigint x, int y) { return divmod(x, y).fst; }
bigint operator%(bigint x, int y) { return divmod(x, y).snd; }

pair<bigint, bigint> divmod(bigint x, bigint y) {
```

```
    auto norm = bigint::B / (y.back() + 1);
    int qsign = x.sign * y.sign, rsign = x.sign;
    x = x * norm; x.sign = +1; y = y * norm; y.sign = +1;
    bigint q, r;
    for (int i = (int)x.size()-1; i >= 0; --i) {
        r.insert(r.begin(), x[i]);
        long long d = ((r.size()   > y.size() ? bigint::B * r[y.size()] : 0)
                    + (r.size()+1 > y.size() ? r[y.size()-1] : 0)) / y.back();
        r = r - y * d;
        while (r.sign < 0) { r = r + y; d -= 1; }
        q.push_back(d);
    }
    reverse(all(q));
    q.sign = qsign; r.sign = rsign;
    return make_pair(q.trim(), divmod(r.trim(), norm).fst);
}
bigint operator/(bigint x, bigint y) { return divmod(x, y).fst; }
bigint operator%(bigint x, bigint y) { return divmod(x, y).snd; }
```

Aunque la implementación presentada aqui es bastante larga, demasiada larga es porque hemos incluidos algunas otras operaciones no analizadas en la guía.

4.2. Java

Java incorporó una clase destinada a operaciones aritméticas que requieran gran precisión con enteros: BigInteger . La forma de operar con objetos de estas clases difiere de las operaciones con variables primitivas. En este caso hay que realizar las operaciones utilizando métodos propios de estas clases.

- **abs():** Devuelve un BigInteger cuyo valor es el valor absoluto de este BigInteger.
- **add():** Este método devuelve un BigInteger simplemente calculando el valor 'this + val'.
- **and():** Este método devuelve un BigInteger calculando el valor 'this & val'.
- **andNot():** Este método devuelve un BigInteger calculando el valor 'this & $\sim$ val'.
- **bitCount():** Este método devuelve el número de bits en la representación de complemento a dos de este BigInteger que difiere de su bit de signo.
- **bitLength():** Este método devuelve el número de bits en la representación mínima en complemento a dos de este bit de signo, excluyendo el bit de signo.
- **clearBit():** Este método devuelve un BigInteger cuyo valor es igual a este BigInteger cuyo bit designado se borra.
- **compareTo():** Este método compara este BigInteger con el BigInteger especificado.
- **divide():** Este método devuelve un BigInteger al calcular el valor 'this / val'.

- **divideAndRemainder():** Este método devuelve un BigInteger calculando el valor 'this & ∼ val'seguido de 'this %value'.
- **doubleValue():** Este método convierte este BigInteger en double.
- **equals():** Este método compara este BigInteger con el Objeto dado para la igualdad.
- **flipBit():** Este método devuelve un BigInteger cuyo valor es igual a este BigInteger con el bit designado invertido.
- **floatValue():** Este método convierte este BigInteger en float.
- **gcd():** Este método devuelve un BigInteger cuyo valor es el máximo común divisor entre abs(this) y abs(val).
- **getLowestSetBit():** Este método devuelve el índice del bit más a la derecha (orden más bajo) en este BigInteger (el número de bits cero a la derecha del bit más a la derecha).
- **hashCode():** Este método devuelve el código hash para este BigInteger.
- **intValue():** Este método convierte este BigInteger en int.
- **isProbablePrime():** Este método devuelve un valor booleano 'verdadero'si y solo si este BigInteger es primo; de lo contrario, para valores compuestos devuelve falso.
- **longValue():** Este método convierte este BigInteger en long.
- **max():** Este método devuelve el máximo entre este BigInteger y val.
- **min():** Este método devuelve el mínimo entre BigInteger y val.
- **mod():** Este método devuelve un valor BigInteger para este mod m.
- **modInverse():** Este método devuelve un BigInteger cuyo valor es 'this módulo inverso m'.
- **modPow():** Este método devuelve un BigInteger cuyo valor es 'this exponente mod m'.
- **multiply():** Este método devuelve un BigInteger al calcular el valor 'this *val'.
- **negate():** Este método devuelve un BigInteger cuyo valor es '-this'.
- **nextProbablePrime():** Este método devuelve el siguiente número primo mayor que este BigInteger.
- **not():** Este método devuelve un BigInteger cuyo valor es '∼this'.
- **or():** Este método devuelve un BigInteger cuyo valor es 'this | val'
- **pow():** Este método devuelve un BigInteger cuyo valor es 'this exponente'.
- **probablePrime():** Este método devuelve un BigInteger primo positivo, con el bitLength especificado.
- **remainder():** Este método devuelve un BigInteger cuyo valor es 'this % val'.

- **setBit():** Este método devuelve un BigInteger cuyo valor es igual a este BigInteger con el conjunto de bits designado.
- **shiftLeft():** Este método devuelve un BigInteger cuyo valor es 'this ≪ val'.
- **shiftRight():** Este método devuelve un BigInteger cuyo valor es 'this ≫ val'.
- **signum():** Este método devuelve la función signum de este BigInteger.
- **subtract():** Este método devuelve un BigInteger cuyo valor es 'this - val'.
- **testbit():** Este método devuelve un valor booleano 'verdadero'si se establece el bit designado.
- **toByteArray():** Este método devuelve un arreglo de bytes que contiene la representación en complemento a dos de este BigInteger.
- **toString():** Este método devuelve la representación de cadena decimal de este BigInteger.
- **valueOf():** Este método devuelve un BigInteger cuyo valor es equivalente al del long especificado.
- **xor():** Este método devuelve un BigInteger al calcular el valor 'this ŷval'.

```
import java.math.*;

public class Main {

   public static void main(String[] args) throws Exception {
      // Inicializar resultado
      BigInteger bigInteger = new BigInteger("1");
      int n = 4;
      for (int i = 2; i <= n; i++) {
          // devuelve un BigInteger calculando ?este *val ? valor.
          bigInteger = bigInteger.multiply(BigInteger.valueOf(i));
      }
      System.out.println("Factorial de 4 : " + bigInteger);
      // devuelve un valor booleano ? verdadero? si y solo
      // si este BigInteger es primo
      BigInteger bigInteger2 = new BigInteger("197");
      System.out.println("IsProbablePrime method will return : "
      + bigInteger2.isProbablePrime(2));
      // devuelve el siguiente entero primo que es mayor que este BigInteger.
      BigInteger nextPrimeNumber = bigInteger2.nextProbablePrime();
      System.out.println("Prime Number next to 197 : " + nextPrimeNumber);
      // devuelve el minimo entre este BigInteger y val
      BigInteger min = bigInteger.min(bigInteger2);
      System.out.println("Minimo valor : " + min);
      // devuelve el maximo entre este BigInteger y val
      BigInteger max = bigInteger.max(bigInteger2);
      System.out.println("Maximo valor: " + max);

      // Inicializar resultado
```

```
        BigInteger bigInteger3 = new BigInteger("17");
        //devuelve la funcion signum de este BigInteger
        BigInteger bigInteger4 = new BigInteger("171");
        System.out.println("El signo del valor "+bigInteger4+" : "
        + bigInteger4.signum());
        BigInteger sub=bigInteger4.subtract(bigInteger3);
        System.out.println(bigInteger4+"-"+bigInteger3+" : "+sub);

        // devuelve el cociente despues de dividir dos valores bigInteger
        BigInteger quotient=bigInteger4.divide(bigInteger3);
        System.out.print(bigInteger4+" / "+
        bigInteger3+" :     Cociente : "+quotient);

        //devuelve el resto despues de dividir dos valores bigIntger
        BigInteger remainder=bigInteger4.remainder(bigInteger3);
        System.out.println("       Resto : "+remainder);

        //devuelve un BigInteger cuyo valor es ?este << val?
        BigInteger shiftLeft=bigInteger3.shiftLeft(4);
        System.out.println("Valor de desplazamiento a la izquierda: "+shiftLeft)
            ;
    }
}
```

5. Aplicaciones

La aritmética larga tiene varias aplicaciones mas allá de problemas o ejercicios de la programación competitiva aunque su utilizanción en este ámbito tiene como objetivo aumentar el grado de complejidad que ya tenga el problema o ejercicio en sí que en la gran mayoría es baja. Incluso en muchos casos no debemos hacer una implementación tan elaborada como la mostrada ya que por lo general no se necesita hacer uso de todas las operaciones o por determinadas condiciones del problema se pueden acotar las que se necesitan. Entre las aplicaciones podemos citar:

5.1. Aritmética de enteros largos para la representación de factorización

Aritmética de enteros largos para la representación de factorización

La idea es almacenar el entero como su factorización, es decir, las potencias de los números primos que lo dividen.

Este enfoque es muy fácil de implementar y permite hacer multiplicaciones y divisiones fácilmente (asintóticamente más rápido que el método clásico), pero no sumas ni restas. También es muy eficiente en memoria en comparación con el enfoque clásico.

Este método se usa a menudo para cálculos módulo número no primo M; en este caso, un número se almacena como potencias de divisores de M que dividen el número más el resto módulo M.

5.2. Aritmética de enteros largos en módulos primos (algoritmo de Garner)

La idea es elegir un conjunto de números primos (por lo general, son lo suficientemente pequeños como para caber en el tipo de datos de números enteros estándar) y almacenar un número entero como un vector de residuos de la división del número entero por cada uno de esos números primos.

El teorema del resto chino establece que esta representación es suficiente para restaurar de forma única cualquier número de 0 al producto de estos números primos menos uno. El algoritmo de Garner permite restaurar el número de dicha representación a un entero normal.

Este método permite ahorrar memoria en comparación con el enfoque clásico (aunque los ahorros no son tan drásticos como en la representación de factorización). Además, permite realizar sumas, restas y multiplicaciones rápidas en un tiempo proporcional al número de números primos utilizados como módulos.

La contrapartida es que volver a convertir el entero a su forma normal es bastante laborioso y requiere implementar la aritmética clásica de precisión arbitraria con la multiplicación. Además, este método no admite la división.

5.3. Aritmética fraccionaria de precisión arbitraria

Las fracciones ocurren en competencias de programación con menos frecuencia que los números enteros, y la aritmética larga es mucho más complicada de implementar para fracciones, por lo que las competencias de programación presentan solo un pequeño subconjunto de aritmética larga fraccionaria.

5.3.1. Aritmética en Fracciones Irreducibles

Un número se representa como una fracción irreducible $\frac{a}{b}$, donde a y b son números enteros. Todas las operaciones con fracciones se pueden representar como operaciones con numeradores y denominadores enteros de estas fracciones. Por lo general, esto requiere el uso de la aritmética clásica de precisión arbitraria para almacenar el numerador y el denominador, pero a veces es suficiente un tipo de datos integrado de enteros de 64 bits.

5.3.2. Almacenamiento de posición de punto flotante como tipo separado

A veces, un problema requiere el manejo de números muy pequeños o muy grandes sin permitir el desbordamiento o subdesbordamiento. El tipo de datos doble incorporado usa de 8 a 10 bytes y permite valores del exponente en $[-308; 308]$ rango, que a veces puede ser insuficiente.

El enfoque es muy simple: se utiliza una variable entera separada para almacenar el valor del exponente y, después de cada operación, el número de coma flotante se normaliza, es decir, se vuelve a $[0{,}1; 1)$ intervalo ajustando el exponente en consecuencia.

Cuando dos de estos números se multiplican o dividen, sus exponentes deben sumarse o restarse, respectivamente. Cuando se suman o se restan números, primero se tienen que llevar al

exponente común multiplicando uno de ellos por 10 elevado a la potencia igual a la diferencia de los valores de los exponentes.

Como nota final, la base del exponente no tiene que ser igual a 10. Según la representación interna de los números de punto flotante, tiene más sentido usar 2 como base del exponente.

6. Complejidad

La complejidad de las implementaciones de enteros grandes va depender fundamentalemente de las operaciones que se implementen y para algunas operaciones cual de las posible implementaciones para ella se selecionó. En el caso de las implementación en C++ podemos analizar que las operaciones de comparación, suma y subtración su complejidad es O(n) , para el caso de la multiplicación que utiliza el algoritmo de Karatsuba su complejidad es O(n^1,7) mientras la divisón basada en el algoritmo de Knuth su complejidad es O(n^2) siendo n todos los casos la mayor cantidad de dígitos de los números involucrados en las operaciones.

7. Ejercicios

A continuación una lista de ejercicios que se pueden resolver aplicando los visto en esta guía:

- DMOJ - Tarea del Aula
- DMOJ - Div 6
- UVA - 10106 - Product
- SPOJ - MUL - Fast Multiplication
- UVA - 10814 - Simplifying Fractions