Informe Complejidades Computacionales

Brayan David Zuluaga

29 de mayo de 2023

Índice

1. Operaciones constructoras

1.1. Constructor Vacío

BigInteger::BigInteger()

La complejidad de la función es constante, es decir, O(1). Esto se debe a que la función simplemente realiza una operación de inserción `push_back()` en el vector interno `vec` con un único elemento (0), sin importar el tamaño actual del

vector. No se requiere iterar ni realizar operaciones que dependan del tamaño del vector, por lo que la complejidad es constante.

1.2. Constructor con parámetros

BigInteger::BigInteger(const string e)

La verificación del primer carácter de la cadena e tiene una complejidad constante, O(1). El bucle for itera desde e.size() - 1 hasta 1 o 0, dependiendo de si el primer carácter es un signo negativo o no. En el peor de los casos, cuando e no es negativo, el bucle se ejecutará n veces, por lo que la complejidad es O(n). Dentro del bucle, se realiza una operación de inserción `push_back()` en el vector `vec` para cada dígito de la cadena e. Dado que el bucle se ejecuta n veces, la complejidad total de las operaciones de inserción es O(n).

Entonces, la complejidad es O(n), donde n es el tamaño de la cadena e. Esto se debe a que se realiza un bucle que itera a través de la cadena y se realiza una operación de inserción para cada dígito.

1.3. Copia

BigInteger::BigInteger(const BigInteger other)

La asignación vec = other.vec realiza una copia del vector interno vec del objeto other al vector interno vec del objeto actual. La complejidad de esta operación de asignación es proporcional al tamaño del vector, es decir, O(n), donde n es el tamaño del vector interno vec de other. Por lo tanto la complejidad computacional es O(n)

2. Operaciones Modificadoras

2.1. Add

void BigInteger::add(BigInteger big)

Tendremos en cuenta que n es el tamaño del vector del BigIntger y m es el tamaño del vector de big La verificación inicial de si ambos objetos tienen el mismo signo (negativo o no) tiene una complejidad constante, O(1). Si los objetos tienen el mismo signo, se realiza un bucle que itera hasta que se procesen todos los dígitos en los vectores vec del objeto actual y big. En el peor caso, el bucle se ejecuta hasta el máximo entre el tamaño del vector interno vec del objeto actual, el tamaño del vector interno vec de big y hasta que aux sea diferente de cero. Esto implica que el bucle se ejecutará a lo sumo max(n, m) + 1 veces. La complejidad de este bucle es proporcional al máximo entre el tamaño de los vectores internos, es decir, O(max(n, m)). Dentro del bucle, se realizan operaciones de suma, asignación, inserción y división. Estas operaciones tienen una complejidad constante, O(1), ya que involucran operaciones aritméticas y asignación de enteros.

Si los objetos tienen signos opuestos, se crea un objeto temporal temp que es una copia del objeto big. Esto implica una copia del vector interno vec de

big, lo cual tiene una complejidad proporcional al tamaño del vector interno vec de big, es decir, O(n). Si el objeto actual es negativo y big no lo es, se realizan operaciones de cambio de signo (negativo = !negativo) y llamadas a la función substract(temp). La función substract() tiene su propia complejidad, que analizaremos por separado. En este caso, la complejidad de las operaciones adicionales es constante, O(1). Si ambos objetos tienen el mismo signo o el objeto actual es negativo y big es negativo, se llama a la función substract(temp) para realizar la resta. La función substract() tiene su propia complejidad, que analizaremos por separado la cual adelantando es O(max(n,m).

entonces la complejidad de la operacion Add es O(max(n,m))

2.2. Substract

void BigInteger::substract(BigInteger big)

Tendremos en cuenta que n es el tamaño del vector del BigIntger y m es el tamaño del vector de big La verificación inicial de si los objetos tienen signos opuestos tiene una complejidad constante, O(1). Si los objetos tienen signos opuestos, se crea un objeto temporal temp que es una copia del objeto big. Esto implica una copia del vector interno vec de big, lo cual tiene una complejidad proporcional al tamaño del vector interno vec de big, es decir, O(m).

Si el objeto actual no es negativo y big es negativo, se realiza una operación de cambio de signo (temp.negativo = false) y se llama a la función add(temp). La función add() tiene su propia complejidad, que hemos analizado anteriormente y suponemos como O(max(n, m)) en este caso. Si los objetos tienen el mismo signo o ambos son negativos, se crea un objeto temporal temp que es una copia del objeto big. Esto implica una copia del vector interno vec de big, lo cual tiene una complejidad proporcional al tamaño del vector interno vec de big, es decir, O(m).

Si el objeto actual es negativo y big es negativo, se realiza una operación de cambio de signo (temp.negativo = false) y se llama a la función add(temp). La función add() tiene su propia complejidad, que hemos analizado anteriormente y suponemos como O(max(n, m) en este caso. Si los objetos tienen el mismo signo o el objeto actual es mayor que big, se realiza un bucle que itera a través de los dígitos del objeto actual. El bucle tiene una complejidad proporcional al tamaño del vector interno vec del objeto actual, es decir, O(n). Dentro del bucle, se realizan operaciones de resta, asignación y comparación, que tienen una complejidad constante, O(1). Después del bucle, se realiza una eliminación de los ceros no significativos en el vector interno vec. La complejidad de esta operación es proporcional al tamaño del vector interno vec, es decir, O(n). Por lo tanto la complejidad de la operacion Substract es O(max(n,m)

2.3. Product

void BigInteger::product(BigInteger big)

Las variables n y m se inicializan con el tamaño de los vectores internos vec del objeto actual y big, respectivamente. Estas operaciones tienen una comple-

jidad constante, O(1). Se crea un vector ans de tamaño n + m e inicializado con ceros. La complejidad de esta operación es O(n + m), ya que se realiza una asignación para cada posición del vector. Se realiza un bucle anidado que itera n veces en el bucle exterior y m veces en el bucle interior. Por lo tanto, el bucle anidado se ejecutará en total n * m veces. Dentro del bucle, se realizan operaciones de multiplicación, suma, división y asignación. Estas operaciones tienen una complejidad constante, O(1). Después del bucle anidado, se realiza un bucle para eliminar los ceros no significativos en el vector ans. La complejidad de esta operación es proporcional al tamaño del vector ans, que tiene un tamaño máximo de n + m, es decir, O(n + m). Se asigna el vector ans al vector interno vec del objeto actual. La complejidad de esta operación es proporcional al tamaño del vector ans, es decir, O(n + m). Se realiza una comparación de signos y se actualiza el signo del objeto actual en base a la comparación. Esta operación tiene una complejidad constante, O(1). la complejidad de la función product depende del tamaño de los vectores internos vec del objeto actual y big, y se puede considerar como O(n * m), donde n es el tamaño del vector interno vec del objeto actual y m es el tamaño del vector interno vec de big. Esto se debe a que se realizan operaciones de multiplicación y suma en un bucle anidado que itera n * m veces.

2.4. Pow

void BigInteger::pow(BigInteger big)

La variable par se asigna con el resultado de verificar si e es par. Esta operación tiene una complejidad constante, O(1). Se verifica si e es igual a cero. En caso afirmativo, se realiza una operación de limpiar el vector interno vec y se inserta el número 1. Estas operaciones tienen una complejidad constante, O(1). En caso contrario, se crean dos objetos BigInteger ans y base, inicializados con una copia del objeto actual. Esto implica una copia del vector interno vec del objeto actual, lo cual tiene una complejidad proporcional al tamaño del vector interno vec del objeto actual, es decir, O(n). Donde es el tamaño del vector interno del objeto actual.

Se realiza un bucle que itera e-1 veces. Dentro del bucle, se llama a la función product(base) para multiplicar ans por base. La función product() tiene su propia complejidad, que hemos analizado anteriormente y suponemos como O(n * m), donde n es el tamaño del vector interno vec de ans y m es el tamaño del vector interno vec de base. Por lo tanto, la complejidad de este bucle es O((e-1) * n * m). Después del bucle, se asigna el vector interno vec de ans al vector interno vec del objeto actual. La complejidad de esta operación es proporcional al tamaño del vector interno vec de ans, es decir, O(z). Se realiza una comparación de paridad y se actualiza el signo del objeto actual en base a la comparación. Esta operación tiene una complejidad constante, O(1). la complejidad de la función pow depende del valor del exponente e, así como del tamaño del vector interno vec del objeto actual y los objetos ans y base. Si suponemos que la complejidad de la función product() es O(n * m), la complejidad total de pow() puede aproximarse como O((e-1) * n * m + z).

2.5. Quotient

void BigInteger::quotient(BigInteger big)

Se declaran las variables signo, cero, y uno. Estas operaciones tienen una complejidad constante, O(1). Se verifica si big es igual a cero. En caso afirmativo, no se realiza ninguna operación. Esta operación tiene una complejidad constante, O(1). En caso contrario, se verifica el signo de big y se asigna el valor a la variable signo. Esta operación tiene una complejidad constante, O(1). Se crean dos objetos BigInteger dividido y ans, inicializados con copias del objeto actual. Esto implica una copia del vector interno vec del objeto actual, lo cual tiene una complejidad proporcional al tamaño del vector interno vec del objeto actual, es decir, O(vec.size()). Se realiza una serie de operaciones para preparar big y dividido para la división. Estas operaciones tienen una complejidad constante, O(1). Se ejecuta un bucle que continúa mientras big sea menor o igual que dividido. Dentro del bucle, se realizan operaciones de resta y suma llamando a las funciones substract() y add() respectivamente. La complejidad de estas operaciones depende del tamaño de los vectores internos vec de los objetos involucrados, suponiendo que la complejidad de substract() es O(max(n, m) y la complejidad de add() es igual a la de substract. Después del bucle, se asigna el vector interno vec de ans al vector interno vec del objeto actual. La complejidad de esta operación es proporcional al tamaño del vector interno vec de ans, es decir, O(z). Se realiza una actualización del signo del objeto actual en base a la variable signo. Esta operación tiene una complejidad constante, O(1).

la complejidad total de quotient() puede aproximarse como O(max(n, m * x), donde x es el número de iteraciones del bucle, determinado por la relación entre big y dividido.

2.6. Remainder

void BigInteger::remainder(BigInteger big)

Se declaran las variables signo y cero. Estas operaciones tienen una complejidad constante, O(1). Se verifica si big es igual a cero. En caso afirmativo, no se realiza ninguna operación. Esta operación tiene una complejidad constante, O(1). En caso contrario, se verifica el signo de big y se asigna el valor a la variable signo. Esta operación tiene una complejidad constante, O(1). Se crea un objeto BigInteger dividido, inicializado con una copia del objeto actual. Esto implica una copia del vector interno vec del objeto actual, lo cual tiene una complejidad proporcional al tamaño del vector interno vec del objeto actual, es decir, O(n). Se realiza una operación para preparar big y dividido para la división. Esta operación tiene una complejidad constante, O(1). Se ejecuta un bucle que continúa mientras big sea menor o igual que dividido. Dentro del bucle, se realiza una operación de resta llamando a la función substract(). La complejidad de esta operación depende del tamaño de los vectores internos vec de los objetos involucrados, suponiendo que la complejidad de substract() es O(max(n, m)). Después del bucle, se asigna el vector interno vec de dividido al vector interno vec del objeto actual. La complejidad de esta operación es pro-

porcional al tamaño del vector interno vec de dividido, es decir, O(z). Se realiza una actualización del signo del objeto actual en base a la variable signo. Esta operación tiene una complejidad constante, O(1).

a complejidad total de remainder() puede aproximarse como O(max(n, m * x), donde x es el número de iteraciones del bucle, determinado por la relación entre big y dividido.

3. Analizadoras

3.1. ToString

void BigInteger::toString(BigInteger big)

Se declara la variable ans como una cadena de caracteres vacía. Esta operación tiene una complejidad constante, O(1). Se verifica si el objeto BigInteger es negativo. En caso afirmativo, se agrega el carácter '-' a la cadena ans. Esta operación tiene una complejidad constante, O(1). Se ejecuta un bucle que itera desde el índice más alto hasta el índice más bajo del vector interno vec del objeto BigInteger. Dentro del bucle, se convierte cada elemento del vector interno en una cadena de caracteres utilizando la función *to_string()* que se considera O(1) y se agrega a la cadena ans. La complejidad de este bucle depende del tamaño del vector interno vec, es decir, O(n). Se devuelve la cadena ans. Esta operación tiene una complejidad constante, O(1). la complejidad de la función toString es O(n), donde n es el tamaño del vector interno vec del objeto BigInteger.

3.2. Igual que

bool BigInteger::operator==(BigInteger big)

Se declara la variable ans como verdadera (true). Esta operación tiene una complejidad constante, O(1). Se verifica si el signo (negativo) del objeto actual es diferente del signo del objeto big. En caso afirmativo, se asigna false a la variable ans. Esta operación tiene una complejidad constante, O(1). Se verifica si el tamaño del vector interno vec del objeto actual es diferente del tamaño del vector interno vec del objeto big, siempre y cuando ans sea verdadero (true). En caso afirmativo, se asigna false a la variable ans. Esta operación tiene una complejidad constante, O(1).

Se ejecuta un bucle que itera desde el índice más alto hasta el índice más bajo del vector interno vec del objeto actual, siempre y cuando ans sea verdadero (true). Dentro del bucle, se compara cada elemento del vector interno vec del objeto actual con el elemento correspondiente del vector interno vec del objeto big. Si se encuentra una diferencia, se asigna false a la variable ans. La complejidad de este bucle depende del tamaño del vector interno vec, es decir, O(n). Se devuelve el valor de la variable ans. Esta operación tiene una complejidad constante, O(1). En resumen, la complejidad de la función sobrecarga del operador == es O(n), donde n es el tamaño del vector interno vec del objeto BigInteger.

3.3. Menor que

bool BigInteger::operator < BigInteger big)

Se declara la variable ans sin inicializar y la variable flag como verdadera (true). Estas operaciones tienen una complejidad constante, O(1). Se verifica una serie de condiciones para determinar el valor de la variable ans y la variable flag. Cada verificación tiene una complejidad constante, O(1).

En caso de que las condiciones anteriores no se cumplan, se ejecuta un bucle que itera desde el índice más alto hasta el índice más bajo del vector interno vec del objeto actual o del objeto big, dependiendo de si los números son negativos o no. Dentro del bucle, se compara cada elemento del vector interno vec del objeto actual con el elemento correspondiente del vector interno vec del objeto big. Se actualiza el valor de la variable ans según el resultado de la comparación. La complejidad de este bucle depende del tamaño del vector interno vec, es decir, O(vec.size()). Se devuelve el valor de la variable ans. Esta operación tiene una complejidad constante, O(1). En resumen, la complejidad de la función sobrecarga operador < es O(n), donde n es el tamaño del vector interno vec del objeto BigInteger.

3.4. Menor o igual que

bool BigInteger::operator <= BigInteger big)

Esta función utiliza los operadores operator ¡y operator== previamente definidos. El código simplemente combina las comparaciones utilizando los operadores lógicos —— (or) para verificar si uno de los operadores es verdadero.

El operador operator< y operator== tienen una complejidad de O(n), donde n es el tamaño del vector interno vec del objeto BigInteger.

Por lo tanto, la complejidad de la función sobrecarga de operador <=() es O(n), siendo n el tamaño del vector interno vec del objeto BigInteger.

4. Estáticas

4.1. sumarListaValores

BigInteger BigInteger::sumarListaValores(list<BigInteger> lista)

Se declara una variable suma de tipo BigInteger. Esta operación tiene una complejidad constante, O(1). Se ejecuta un bucle que itera a través de la lista utilizando un iterador. El bucle se repite hasta que el iterador alcance el final de la lista. Dentro del bucle, se llama al método add() que tiene una complejidad O(max(n,m)) del objeto suma pasando como argumento el objeto BigInteger actual del iterador. La complejidad de este bucle depende del tamaño de la lista, es decir, O(l). Se devuelve el objeto BigInteger suma. Esta operación tiene una complejidad constante, O(1). En resumen, la complejidad de la función sumarListaValores() es O(l*k), donde l es el tamaño de la lista de objetos BigInteger y k es un promedio constante del tamaño de los vectores internos vec.

4.2. multiplicarListaValores

BigInteger BigInteger::multiplicarListaValores(list<BigInteger> lista)

Se declara una variable producto de tipo BigInteger e inicializada con el valor "1". Esta operación tiene una complejidad constante, O(1). Se ejecuta un bucle que itera a través de la lista utilizando un iterador. El bucle se repite hasta que el iterador alcance el final de la lista. Dentro del bucle, se llama al método product() del objeto producto pasando como argumento el objeto BigInteger actual del iterador. La complejidad de este bucle depende del tamaño de la lista, es decir, O(n). Se devuelve el objeto BigInteger producto. Esta operación tiene una complejidad constante, O(1). a complejidad de la función multiplicarListaValores sería O(n * k**2), donde n es el tamaño de la lista de objetos BigInteger y k es un promedio constante del tamaño de los vectores internos vec.

5. Sobrecarga Operadores Aritméticos

Para todos los operadores aritméticos, su complejidad será la misma a la de su implementación asociadas