Student Name: Juan Diego Collazos
Student ID: 8977837

Estructura de Datos - BigInteger

Introducción

Un BigInteger es un tipo de dato utilizado en programación para representar números enteros de tamaño arbitrariamente grande. A diferencia de los tipos de datos numéricos convencionales, como int o long, que tienen un límite fijo en el rango de valores que pueden almacenar, un BigInteger puede manejar números enteros sin restricciones en cuanto a su tamaño.

Un BigInteger se construye utilizando una estructura de datos interna que almacena de manera dinámica los dígitos del número. Esto significa que puede adaptarse y ajustarse automáticamente según la cantidad de dígitos necesarios para representar el número.

Debido a su flexibilidad, un BigInteger puede representar números enteros extremadamente grandes, incluso más allá de los límites que podrían ser manejados por los tipos de datos numéricos convencionales. Esto lo hace útil en aplicaciones donde se requiere trabajar con números de gran magnitud, como criptografía, cálculos científicos o financieros, y otras áreas donde la precisión y el tamaño de los números son críticos.

Los BigInteger también proporcionan una amplia gama de operaciones matemáticas y métodos para realizar cálculos y manipulaciones con estos números, como suma, resta, multiplicación, división, entre otros. Esto permite realizar operaciones aritméticas complejas con números enteros de gran tamaño y mantener una alta precisión en los resultados.

En resumen, un BigInteger es un tipo de dato utilizado en programación para representar números enteros de tamaño arbitrariamente grande, sin restricciones en cuanto a su magnitud, y proporciona operaciones matemáticas para trabajar con ellos.

Consideraciones importantes de esta implementación de Biginteger

Esta implementación un Biginteger se compone esencialmente de un dato de tipo bool el que determina si el número representado en positivo (flase) o negativo (true) y de un vector de enteros de la Librería STL de C++ donde el valor de cada posición del vector corresponde un dígito del número representado, estos se encuentran ordenados secuencialmente y de manera ascendente donde la cifra más significativa se encuentra en la última posición del vector, ejemplificando el número 7896541359 se encontraría representado de la siguiente manera en el vector:

9	5	3	1	4	5	6	9	8	7
0	1	2	3	4	5	6	7	8	9

Dato Importante: La complejidad del cambio de tamaño del vector para almacenar un número más grande o más pequeño está dada por la complejidad de la operación resize en vectores, se

elijo la operación resize en lugar de push-back porque mejora un poco la complejidad, si se hacen constantemente varios push-back podria tener que trasladar todos los elementos de vector el lugares distintos de memoria varias veces. Por otro lado con el resize en el caso que ya no exista el espacio disponible solo trasladara todos los elementos una sola vez.

La complejidad de la operación resize con vectores en C++

Depende del tamaño final al que se desea redimensionar el vector. Hay dos escenarios principales a considerar:

Aumentar el tamaño del vector: Si se utiliza resize para aumentar el tamaño del vector, la complejidad de tiempo será lineal en función de la diferencia entre el tamaño actual del vector y el tamaño final deseado. Esto se debe a que puede requerir asignar y copiar los elementos existentes a una nueva ubicación de memoria con el tamaño adecuado. En términos de notación Big O, esto sería O(n), donde n es la diferencia en tamaño.

Reducir el tamaño del vector: Si se utiliza resize para reducir el tamaño del vector, la complejidad de tiempo es constante. Esto se debe a que la operación simplemente ajusta el tamaño del vector y no implica copiar o mover elementos. En términos de notación Big O, esto sería O(1).

1. Constructores

1.1. BigInteger(string)

Esta operación toma como parámetro un string y posteriormente recorriéndolo carácter a carácter para asignarle ese valor a una posición respectiva en el vector.

Complejidad computacional (Big O):

$$O(n)$$

Siendo n el tamaño del string que se pasa al constructor.

1.2. BigInteger(BigInteger)

Esta operación toma como parámetro otro Biginteger copiando su respectivo vector para asignarlo al Biginteger que se está construyendo.

Complejidad computacional (Big O):

$$O(n)$$

Siendo n el tamaño del vector del Biginteger se pasa al constructor.

2. Operaciones modificadoras

2.1. add y substract

En las operaciones con enteros dependiendo del signo del los números con los que se desea operar estos pueden restarse o sumarse, por ende es necesario definir dos funciones auxiliares que cumplan esta funciones según sea requerido.

Ambas operaciones reciben en Biginteger que se desea sumar o restar respectivamente al valor actual.

2.1.1. sum

Esta operacion recibe el vector del objeto actual y el Biginteger con que se desea operar y añade dicho valor al vector del objeto actual. Para ello emplea el algoritmo de suma que se conoce comúnmente como "suma en columna", "suma vertical.º "suma con acarreo". Es el método utilizado para sumar números de varias cifras alineando las columnas de los dígitos y realizando la suma de derecha a izquierda, llevando las decenas o unidades superiores en caso de que la suma en una columna sea mayor a 9.

Como se alinean los dígitos y suman de acuerdo a su posición en el peor de los casos se tendrá que recorrer por completo el número con más dígitos, por tanto la complejidad computacional (Big O) seria:

$$O(n)$$

Siendo n el tamaño del Biginteger con más dígitos.

2.1.2. rest

Esta operacion recibe el vector del objeto actual y el Biginteger con que se desea operar y sustrae dicho valor al vector del objeto actual. Para ello emplea el algoritmo de resta que se conoce como - esta en columna", resta vertical.º resta con préstamo". Este algoritmo se utiliza para restar números de varias cifras alineando los dígitos y realizando la resta de derecha a izquierda, llevando prestado a la columna correspondiente cuando es necesario.

Como se alinean los dígitos y restan de acuerdo a su posición en el peor de los casos se tendrá que recorrer por completo el número con más dígitos, por tanto la complejidad computacional (Big O) seria:

$$O(n)$$

Siendo n el tamaño del Biginteger con más dígitos.

Dato Importante: la precondición de esta función es que se pase la representación del número mayor como primer parámetro por tanto el segundo parámetro debe ser menor o igual al primero. Para ello se sobrecargó el operador <= (para más información consulte la sección 4.2) que compara las representaciones de Biginteger, así se garantiza el orden correcto de los elementos al pasárselos a la operación rest.

2.2. product

Esta operación recibe otro Biginteger y multiplica el objeto actual por dicho valor. Para ello emplea el algoritmo de multiplicación que conoce como "multiplicación por descomposición.º "multiplicación por distribución". Este algoritmo se utiliza para multiplicar números de varias cifras alineando los dígitos de uno de los números debajo del otro y realizando las multiplicaciones parciales y sumas correspondientes para obtener el producto final.

Como en todos lo casos se multiplica cada dígito del primer número por cada dígito del segundo se puede afirmar que la complejidad computacional (Big O) de dicha operación es:

$$O(n * m)$$

Siendo n el número de dígitos de primer número y m el número dígitos del segundo número, Sin embargo como en el peor de los casos el número de dígitos de ambos números es igual, se acercaría

a una complejidad computacional (Big O):

$$O(n^2)$$

Siendo n el número dígitos de ambos Bigintegers.

2.3. quotient y remainder

En esta implementación las operaciones de división y residuo con enteros se lleva a cabo esencialmente el mismo algoritmo por ello se definió una operación auxiliar que llevara cabo ambos procesos.

divisionAux

Esta función recibe como primer parámetro el vector del objeto actual y el Biginteger con que se desea operar como segundo parámetro, y realizar el proceso de división tomando al primer parámetro como dividendo y el segundo paramentara como divisor.

El algoritmo que se lleva a cabo es restarle consecutivamente al dividendo el máximo número que pueda componer de la multiplicación del divisor y 10 elevado a una potencia s (donde n es mayor igual 0) que sea menor igual al dividendo (a este valor lo denominaremos w), el algoritmo termina cuando el dividendo sea menor que el divisor. Por cada resta que se pueda efectuar se le suman uno al vector resultado en la posición s. Por ello al terminar el algoritmo el vector resultado contendrá la razón y el vector pasado por primer parámetro el módulo.

En el peor los casos se tendrá que realizar restas pasando por todos los valores w, y como ya se mencionó antes en la sección 2.1.2 la complejidad de restar dígito a dígito es O(n) siendo n el número dígitos del número mayor. Por ende la complejidad de esta operación es:

$$O(n^2)$$

Siendo n la numero de dígitos de divisor.

2.4. pow

Esta operación recibe un dato de tipo entero y calcula la potencia del objeto actual elevado al entero recibido, para ello lleva a cabo un algoritmo que se conoce como .exponenciación rápida.o .exponenciación binaria". Este algoritmo permite calcular una potencia en log(m) multiplicaciones (siendo m el exponente) aprovechando las propiedades matemáticas de las potencias para reducir la cantidad de multiplicaciones necesarias y mejorar la eficiencia del cálculo. Además como ya se mencionó en la sección 2.2 la complejidad de multiplicar un número por sí mismo se $O(n^2)$ siendo n el número de dígitos de número, entonces en primera instancia la complejidad sería aproximadamente:

$$O(n^2 * log(m))$$

Siendo n el número de dígitos de la base y m el exponente, Sin embargo se debe tener en cuenta que cada vez multiplicamos la base por sí misma aumenta su número de dígitos como máximo 2 veces su cantidad de dígitos anterior, por ello se plantea la siguiente expresión, conservando los mismos n y m:

$$f(m,n) = m^2 + \sum_{i=1}^{log(n)-1} (m * 2 * i)^2$$

$$f(m,n) = m^2 + 4m^2 \sum_{i=1}^{log(n)-1} (i)^2$$

$$f(m,n) = m^2 + 4m^2 * \frac{(log(n)-1) * (log(n)) * (2log(n)-1)}{6}$$

$$f(m,n) = m^2 + 4m^2 * \frac{2log^3(n) - 3log^2(n) + log(n)}{6}$$

$$f(m,n) = m^2 + \frac{2}{3}(2m^2log^3(n) - 3m^2log^2(n) + m^2log(n))$$

$$f(m,n) = m^2 + \frac{4}{3}m^2log^3(n) - 2m^2log^2(n) + \frac{2}{3}m^2log(n)$$

Aislando el término más significante obtendremos que la complejidad computacional (Big O) es:

$$O(m^2log^3(n))$$

3. Operaciones Analizadoras

3.1. toSring

Esta función no recibe ningún parámetro y retorna una string con los dígitos del valor del objeto actual, para ello recorre todo el vector interior del Biginteger obtenido cada uno de los dígitos, por ello su complejidad computacional (Big O) es:

$$O(n)$$

Siendo n el número de dígitos del Biginteger.

4. Operadores

4.1. Operadores aritméticos($+, -, *, /, \%$)

Los operadores aritméticos tiene las mismas complejidades que sus respectivas operaciones equivalentes descritas en la sección 2, Adicionando en el caso de los operadores aritméticos la complejidad de hacer una copia de objeto actual la cual es $O(n)$ siendo n el número de dígitos del objeto actual.

4.2. Operadores analizadores($==, <, <=$)

Estos operadores comparar dos Biginteger determinado una relación de magnitud entre ellos, Para ello recorren dígito a dígito los Bigintegers, por tanto este proceso tiene complejidad en el peor de los casos (el peor de los casos es cuando ambos Bigintegers son iguales) (Big O) O(n) siendo n el número de dígitos de ambos Bigintegers. Todos los operadores tiene la misma complejidad en el peor de los casos, diferenciándose solo en las condiciones para parar anticipadamente el algoritmo.

5. Operaciones estáticas

5.1. sumarListaValores

Esta operación recibe una lista de Bigintegers y calcula el resultado de sumar todos los Bigintegers dentro de la lista. Para ello se suman parcialmente con la operación add (De esta forma $((n1 + n2) + n3) + n4...$). Como anterior mente se mencionó en la sección 2.1 la complejidad de sumar dos Bigintegers es O(n) siendo n el número de dígitos del Biginteger mayor, por ende la complejidad de esta operación es:

$$O(n * m)$$

Siendo m el número de elementos en la lista y n el número de dígitos de mayor número respectivo de cada suma.

5.2. multiplicarListaValores

Esta operación recibe una lista de Bigintegers y calcula el resultado de multiplicar todos los Bigintegers dentro de la lista. Para ello se multiplican parcialmente con la operación product (De esta forma $((n1 * n2) * n3) * n4...$). Como anterior mente se mencionó en la sección 2.2 la complejidad de multiplicar dos Bigintegers es O(n * m) siendo n el número de dígitos del primer Biginteger y m del segundo, por ende la complejidad de esta operación es:

$$O(n * m * t)$$

Siendo t el número de elementos en la lista, n el número de dígitos del primer Biginteger y m del segundo, de las respectivas multiplicaciones requeridas.