TAD Big Integer

Harold David Guerrero Caicedo

Estructura de Datos

Carlos Ramírez – Gonzalo Noreña

Facultad de Ingeniería de Sistemas y Computación

Pontificia Universidad Javeriana Cali

27 de mayo de 2023

Tabla de contenido

1. ¿Qué es un TAD?

Un TAD (Tipo Abstracto de Datos) es una abstracción que define un conjunto de operaciones o funciones que se realizan sobre un tipo de datos en particular.

Considera una forma de encapsular datos y operaciones relacionadas.

Es considerada una interfaz clara y define como se pueden manipular los datos y oculta los detalles de las implementaciones de cada función subyacentes. Para implementar un TAD en C ++ se implementa utilizando una clase y esta define los miembros de datos necesarios y las funciones miembro que operan en esos datos. Los objetos de la clase representan instancias del tipo de datos definido por el TAD.

2. Análisis de complejidad de *BigInteger.h*

El archivo *BigInteger.h* es un archivo que normalmente contiene las declaraciones de funciones, estructuras, clases y variables que utilizaremos para la solución del problema de Online Judge.

3. Análisis de complejidad de *BigInteger.cpp*

Para la solución del proyecto se usó un vector donde se añaden los valores que el problema proporciona, estos se reciben y voltean, pues esto facilitará el poder operar de una forma más fácil, dado que las operaciones en el papel se suelen hacer desde las unidades hacia arriba y no de la otra manera. También se usa un booleano que permitirá saber el signo de cada número.

- **Operación add**: primero lo que se hace es distinguir los signos de los números. Si los signos son iguales, los suma recorriendo desde la cifra menos significativa a la más significativa, sumando una por una en el objeto, y si un digito del resultado se pasa de 10 entonces se le pasa 1 al siguiente digito. Si los signos son diferentes hace uso de la función **substract** con los valores absolutos, restando el mayor con el menor y al final el resultado quedando con el signo del número con mayor valor absoluto.

  La complejidad de esta función es O(n), siendo n el tamaño de los números, los cuales se recorren al mismo tiempo si son de igual signo y si son de diferente singo se hace uso de la función **substract** que también es O(n).

- **Operación substract**: función que resta al objeto actual el valor que se pasa por argumento. Lo primero que se hace es verificar si el segundo número es negativo pues en este caso estamos hablando de una suma, con lo que se hace llamado a la operación **add**. En caso contrario, si el primero es negativo, entonces el resultado es la suma de los valores absolutos en negativo. Sino pasa ninguna de las anteriores, ambos son positivos, entonces restamos el mayor con el menor en valor absoluto y se mantiene el signo de la parte más grande. Para hacer esta resta recorremos desde la cifra menos significativa a la más significativa restando el minuendo con el sustraendo, y si el sustraendo es menor que el minuendo se le pide “prestado” a la siguiente cifra.

  La complejidad de esta función es O(n) pues solo recorremos a los números una vez, o llamamos a la función **add** que es O(n).

- **Función product:** primero se revisa si alguno de los factore es 0 en cuyo caso la respuesta es 0. Caso contrario se multiplica cada cifra de un factor con cada cifra del otro factor con el mismo algoritmo que usamos los humanos para la multiplicación.
  Al principio creamos un vector con suficientes ceros para guardar la respuesta. Y al final eliminamos los ceros sobrantes.
  El signo del resultado depende de los signos. Si los signos son diferentes da negativo.
  La complejidad es O(n^2).

- **Función quotient:** primero se guarda el signo del resultado el cual es negativo si los signos son diferentes y después se dividen los valores absolutos de ambos números.
  Si el dividendo es mejor que el divisor el resultado es 0, también si el divisor es 0 se imprime un mensaje de error y el resultado es 0. En el caso contrario se divide con el algoritmo común de la división.
  Primero se bajan todas las cifras del dividendo hasta que el número resultante sea mayor que el divisor, este número resultante lo vamos a guardar en t que es un BigInterger.
  Luego se sigue con el siguiente algoritmo: se prueba con todos los números del 9 hasta el 0 multiplicando por el divisor hasta que el producto sea menor que t. Este número se guarda en la respuesta, el producto se le resta t, se guarda la siguiente cifra del dividendo en t.
  Esto se repite hasta que no queden cifras en el dividendo.
  El resultado será el mismo que la división en c++, es decir que se aproxima hacia abajo.

Tiene una complejidad de O(n^2). Pues hace uso de la multiplicación varias veces por cada cifra del dividendo, pero la multiplicación solo se hace con un número de una cifra.

- **Función remainder:** hace el mismo procedimiento que la función quotient, pero el resultado es lo que queda en t al final, que equivale al residuo. La complejidad O(n^2) es la misma complejidad que la función de quotient, pues se hace prácticamente el mismo procedimiento.

- **Función pow:** para la función pow se hizo uso de una función auxiliar expo que calcula el resultado de una base a una potencia específica de forma recursiva. Dicha función para encontrar el resultado de a^b se llama a si misma para calcular a^(b/2) y luego encuentra el cuadrado de este resultado para encontrar a^b.

  Si b es impar, después de hallar el cuadrado este lo multiplica por a para contrarrestar el hecho de que b/2 se aproxime hasta abajo. A su vez de la función pow, si el exponente es impar se mantiene el signo de la base, caso contrario el resultado es positivo.
  Esta función solo fue programada para exponentes mayores que 0.
  La complejidad total es O(log2(n)*n^2).

- **Operadores binarios aritméticos (+, -, /, *, %):** la complejidad de estos operadores van de la mano con las funciones que se explicaron anteriormente, por ende, tienen la complejidad correspondiente.

- **Operadores binarios (==, < y <=):**
    - **Operador ==:** la complejidad del operador == es O(n), la función de esta es analizar si el número y el valor tienen diferente cantidad de números y diferentes datos, para ello se verifica primero si tienen diferente tamaño o si tienen diferente símbolo (+/-), donde se cumpla esto automáticamente se sabe que los dos números no son iguales. En caso contrario, se verifica que todas las cifras sean iguales, para retornar adecuadamente le igualdad.

    - **Operador <:** la complejidad del operador < es O(n), la función de esta es analizar cuál de los dos (número o valor) es menor que el otro, se inicializa ans en false y lo primero que se hace es verificar el tamaño tanto de número como de valor, si el tamaño de valor es mayor que el de número ans pasa a ser true, pero si valor es un número negativo, pues quiere decir que número es mayor que valor y ans pasa a ser false.
      Lo mismo se hace, pero al revés. En el caso donde el tamaño para los dos sean iguales pasan al primer for, este se recorre n veces dependiendo del tamaño que tenga número, lo recorremos de atrás hacia delante, pues desde un inicio se voltearon los números para poder operar de una manera más fácil, después de recorrer se compara si número es menor que valor, si eso pasa ans es true. Pero si valor es negativo, esto quiere decir que número es mayor por lo tanto ans es false, luego se hace al revés y por último retorna ans. Teniendo una complejidad total de O(n) pues puede que se recorran todas las cifras en el peor de los casos.

- **Operador <=:** la complejidad del operador <= es O(n), la función de esta es comparar entre número y valor, lo primero que se hace es verificar si número es menor a valor o si número es igual a valor, haciendo uso de los operadores anteriormente mencionados con complejidad O(n). Si una de estas dos se cumple, entonces ans es true, sino ans es false y al final se retorna ans por lo tanto la complejidad total del siguiente código es O(n).

- **Bool esNegativo:** es un booleano que retorna true si el número es negativo. Tiene una complejidad de O (1).

- **Void invertir:** nos permite hallar el inverso multiplicativo del número, es decir, cambiar el signo. Tiene una complejidad de O(1).

- **BigInteger absoluteValue:** tiene una complejidad de O(n), la función de esta es hallar el valor absoluto del BigInteger. Se verifica si el número es negativo y si es así pues llamamos a la función invertir.

- **Int info:** la función de esta es tomar el valor según la posición donde se encuentre el iterador al recorrer el vector, el cual guarda los dígitos del BigInteger a su debido valor. Tiene una complejidad total de O(1).

- **Int tamanoN:** tiene una complejidad total de O(1), la función de esta es retornar el tamaño total del vector número, que representa el número de dígitos del BigInteger.

- **Operaciones constructoras:**

    - **BigInteger (long long valor):** el constructor toma un valor entero largo y lo almacena en forma de dígitos individuales dentro del objeto BigInteger. Los dígitos se almacenan en orden inverso, comenzando desde el dígito menos significativo hasta el más significativo. Complejidad: O(n).

    - **BigInteger (cons string &cadena):** el constructor toma una cadena de caracteres y la interpreta como un número para almacenarlo en forma de dígitos individuales dentro del objeto BigInteger. Los dígitos se almacenan en orden inverso, comenzando desde el dígito menos significativo hasta el más significativo. Complejidad: O(n).

    - **BigInteger (const BigInteger &argumento):** el constructor toma un objeto BigInteger como argumento y crea un nuevo objeto BigInteger que es una copia exacta del objeto de entrada. Complejidad: O(n).

    - **BigInteger toString():** esta función convierte el objeto BigInteger en una cadena de caracteres que representa su valor. Los dígitos del objeto BigInteger se concatenan en orden inverso, desde el dígito menos significativo hasta el más significativo. Complejidad: O(n).

    - **BigInteger expo (BigInteger base, BigInteger):** esta función realiza la operación de exponente para calcular la potencia de la base elevada al valor (exponente) proporcionado. La complejidad de esta función es log2(e) * n^2, donde e es el valor (exponente) y n es el tamaño de los números que se multiplican. Se utiliza un caso base cuando el exponente es igual a 1, donde el resultado es simplemente la base. La complejidad total es de O(log2(n)*n^2).

- **Operaciones estáticas:**
    - **sumarListaValores:** esta función calcula la suma de todos los valores en la lista de objetos BigInteger proporcionada. Tiene una complejidad total de O(n^2), ya que depende del tamaño de nuestro vector, y del tamaño de cada BigInteger para realizar las sumas.

    - **multiplicarListaValores:** esta función calcula el producto de todos los valores en la lista de objetos BigInteger proporcionada. Tiene una complejidad total de O(n^3), ya que depende del tamaño de nuestro vector, y el tamaño de cada BigInteger para realizar las multiplicaciones uno por uno.

4. Análisis de complejidad de main.cpp

La complejidad total es O(t * n). Los números se dividen en parte entera y decimal, y se almacenan en variables de tipo string. Se utilizan variables auxiliares y bucles para procesar los números y realizar las comparaciones. La función realiza la impresión de los resultados en la salida estándar según el resultado de las comparaciones. Los ciclos que son utilizados tiene una complejidad de O(n), pero como están dentro del while que recibe los casos pues tiene una complejidad total O(t * n).