Pontificia Universidad
JAVERIANA
Cali

Ingenieria en sistemas y ciencias de la computación

Reporte de complejidades proyecto final

Estructuras de datos 2023-1

Profesores:

Carlos Ramírez

Gonzalo Noreña

Estudiante:

Juan David Bernal Maldonado

Santiago de Cali, mayo del 2023

1. Reporte de complejidades.

Constructor predeterminado.

La complejidad de este constructor es constante, siendo siempre O(1).

Constructor que hace una copia.

La complejidad de este constructor es constante (O(1)). Esto es porque el constructor únicamente hace copias de “num.number” y de “num.sign” y las asigna a los atributos del objeto actual. Por lo tanto, no hay ninguna operación que varie su tiempo de ejecución.

Constructor.

La complejidad de este constructor es lineal. A pesar de que las primeras líneas previas al ciclo for se ejecutan de forma constante, el ciclo hace que la complejidad dependa del tamaño que tenga el string que se le paso como parámetro al constructor, esto sucede porque el ciclo recorre la cadena y la empieza a convertir en números enteros para posteriormente agregarlos a un vector de enteros. Por lo tanto, la complejidad de esta operación es O(n).

Constructor valor absoluto.

La complejidad de este constructor es constante, ya que lo que hace es tomar copiar un vector y un entero y asignar estas copias a dos atributos de un objeto de la clase BigInteger. Por lo tanto, la complejidad de esta operación es O(1).

Función removeLeadingZeros.

La complejidad de esta función es lineal, donde n depende del tamaño del vector. El ciclo se itera mientras que el vector tenga como mínimo un numero entero o se encuentre un valor diferente de 0. En el peor de todos los casos, se debe recorrer todo el vector eliminando cada cero, hasta que únicamente quede un elemento en el vector.

Sobrecarga operator+.

La complejidad de esta función es lineal, O(n), donde n es el tamaño del objeto BigInteger con menor longitud entre el objeto actual y el pasado como parámetro. El primer ciclo recorre los números comunes en términos de tamaño, y realiza la adición de dígitos.

Después de esto, hay dos ciclos while que completan la adición con los números restantes del numero mas largo. Cada uno tiene una complejidad lineal de O(n), donde n representa la cantidad de dígitos restantes en el número.

En el caso de que los signos sean distintos, la complejidad de esta operación será equivalente a la complejidad de la sobrecarga de la resta.

Por estas razones la complejidad de esta operación es lineal, O(n).

Sobrecarga operator-.

La complejidad de esta función también es lineal, O(n), donde n es el tamaño del objeto BigInteger con menor longitud entre el objeto actual y el objeto pasado como parámetro. El primer ciclo while recorre las posiciones comunes de ambos números y realiza la resta de cada par de dígitos.

Después hay un ciclo que recorre el numero mas largo y termina de realizar las restas. Finalmente se ejecuta también una operación que podría cambiar la complejidad que es removeLeadingZeros, pero su complejidad también es lineal, por lo tanto, la complejidad de esta sobrecarga es O(n).

Sobrecarga operator *.

La complejidad de esta función es cuadrática, O(nˆ2), donde n es en el primer ciclo la cantidad de dígitos que hay en el objeto BigInteger actual lo cual tiene como complejidad O(n) porque n depende del tamaño del vector del objeto actual.

En el segundo ciclo la complejidad también es de O(n), donde n representa el tamaño del vector de “num” (Objeto que se pasó como parámetro), en este ciclo se realizan las operaciones aritméticas simples como la multiplicación, suma, división y modulo, aunque todo aplicado sobre datos de tipo entero, por esta razón la complejidad no se ve afectada.

Ahora, teniendo en cuenta que el segundo ciclo for, está anidado al primero, se debe multiplicar las complejidades de ambos ciclos para saber cual será la complejidad final. Como el primer ciclo tiene complejidad O(n) y el segundo también es O(n), la complejidad será O(n*n) = O(nˆ2).

Sobrecarga operator /.

Esta sobrecarga tiene una complejidad cuadrática, esto se puede evidenciar siguiendo un proceso de observación por el cual se puede notar que antes del for, todas las líneas se ejecutan en tiempo constante, en cambio cuando se llega al primer for la complejidad empieza a ser O(n), donde n representa la longitud del vector que se almaceno en la variable “dividend”. Dentro de este ciclo se definen variables y se asignan valores que se usaran en el próximo ciclo.

El segundo ciclo, que en este caso es un while que realiza la resta y el incremento, tiene una complejidad que depende de la implementación del TAD, pero en general, las operaciones de resta y comparación tienen una complejidad lineal en función a la cantidad de dígitos.

La inserción de elementos tiene una complejidad de O(n) ya que se inserta un elemento al principio del vector, haciendo que todos los demás elementos se tengan que mover una posición, entonces n, es la cantidad de dígitos que tiene el vector al cual se está insertando.

Finalmente, esta la operación de removeLeadingZeros que también es O(n) por lo tanto no afectara el cálculo final.

Se debe multiplicar la complejidad del primer ciclo, con la del segundo, ya que este esta anidado al primero, sus complejidades al ser O(n) ambas hacen que la multiplicación de como resultado O(nˆ2).

Sobrecarga operator%.

La complejidad de esta sobrecarga es O(nˆ2) ya que es la misma que maneja la división, en este caso, al principio la complejidad es constante, no obstante, cuando se llega al primer ciclo for, la complejidad de este ciclo es O(n), donde n es depende del tamaño del vector del "dividend". Luego en el segundo while la complejidad depende de la implementación del TAD porque se utiliza la sobrecarga de los operadores menor que y la resta, los cuales, como ya se menciono anteriormente tienen una complejidad lineal.

Entonces al final multiplicamos la complejidad de los ciclos anidados y obtenemos nuevamente una complejidad cuadrática, O(nˆ2).

Sobrecarga operator==.
La complejidad de esta sobrecarga O(1), es constante porque lo único que hace es retornar verdadero (true) si el numero del objeto actual es equivalente al objeto que se pasa como parámetro ("*this" == "num.number"), lo que hace esto es comparar dos vectores. Por otro lado, también es necesario que los signos de estos dos objetos que están siendo comparados sean iguales, en el caso de que sean iguales y la anterior condición también se cumpla, se retornara true. Estas operaciones no tienen un costo computacional alto por lo que al final la complejidad de esta operación es O(1).

Sobrecarga operator <.

La complejidad de esta sobrecarga comienza siendo lineal, las primeras condiciones hacen que este costo computacional sea tan bajo, hasta que se llega al ciclo while, el cual se itera la cantidad de números enteros que contienen los objetos que están siendo comparados, lo que hace que la complejidad de esta operación sea lineal, donde n depende directamente del tamaño de los objetos los cuales están siendo comparados. En el mejor caso el ciclo while nunca se iniciará y la complejidad de esta operación será constante, sin embargo, en el peor de los casos, la complejidad es O(n). Por lo tanto, la complejidad final es lineal O(n).

Sobrecarga operator <=.

La complejidad de esta sobrecarga al principio parece ser constante, no obstante, cuando observamos que, dentro de esta sobrecarga, se está utilizando la implementación de dos operadores sobrecargados, los cuales son el igual que y menor que, nos damos cuenta de que la complejidad dependerá de cual es el costo computacional de estos operadores.

En el caso del menor que, sabemos que la complejidad en el mejor de los casos es constante y en el peor es lineal, por lo tanto, la complejidad es O(n).

En el caso del igual, la complejidad es constante, O(1), lo cual hace que el mayor costo computacional se de en el caso de que el menor que tenga que entrar en el ciclo while.

Por lo tanto, la complejidad de esta operación es lineal, O(n), donde n depende del tamaño de los objetos BigInteger que están siendo comparados.

Función add.

La complejidad de esta función es la misma complejidad de la suma. Lo único que cambia es que en este caso la función si modifica el objeto actual *this.

Función producto.

La complejidad de esta función es la misma complejidad de lamultiplicación. Lo único que cambia es que en este caso la función si modifica el objeto actual *this.

Función substract.

La complejidad de esta función es la misma complejidad de la resta. Lo único que cambia es que en este caso la función si modifica el objeto actual *this.

Función quotient.

La complejidad de esta función es la misma complejidad de la división. Lo único que cambia es que en este caso la función si modifica el objeto actual *this.

Función remainder.

La complejidad de esta función es la misma complejidad del módulo. Lo único que cambia es que en este caso la función si modifica el objeto actual *this.

Función pow.

La complejidad de esta operación empieza siendo O(n) por el primer ciclo while, donde n es el numero entero que se pasa como parámetro a la función. No obstante, dentro del while hay una multiplicación, la cual tiene una complejidad cuadrática, lo que hace que al final la complejidad del pow sea O(nˆ2).

Función toString.

La complejidad de esta función es lineal, O(n), donde n es la cantidad de números enteros que tiene el vector del objeto BigInteger actual. Luego en cada iteración se convierte el numero entero que esta en el vector en la posición i a un char, y se concatena al string ans que es el que retornara al final, por lo tanto, la complejidad no aumenta por otras razones a parte del ciclo for, la complejidad es O(n).

Función sumarListasValores.

La complejidad de esta función es lineal, n depende de la cantidad de objetos BigInteger que tenga la lista pasada como parámetro, dentro del ciclo se hace una suma, por lo que la complejidad podría cambiar, pero como la complejidad de la suma es O(n), entonces la complejidad no cambia de la anteriormente mencionada en el primer for. Por lo tanto la complejidad es O(n * m), donde m es la longitud de la lista.

Función multiplicarListaValores.

La complejidad de esta función comienza siendo lineal por el ciclo while, donde n también depende de la cantidad de elementos que tenga la lista. Dentro de este ciclo se realiza la multiplicación la cual tiene una complejidad cuadrática, lo cual hace que la complejidad ascienda a O(nˆ2). Aunque también se puede tomar en cuenta de que la función será de complejidad O(nˆ2 * l), donde l es el tamaño de la lista.