PROYECTO FINAL

Informe Complejidad

Estructura de Datos, Grupo B, 2023 -1

Departamento de Electrónica y Ciencias de la Computación

Pontificia Universidad Javeriana Cali

Profesores:

Carlos Alberto Ramírez Restrepo

carlosalbertoramirez@javerianacali.edu.co

Luis Gonzalo Noreña

luis.norena@javerianacali.edu.co

Autor: Jean Karlo Buitrago Orozco

Código: 8972318

TAD - BigIntegers

El tipo de dato abstracto BigInteger es implementado de una forma que permite representar números enteros que son demasiado grandes para ser representados de manera precisa utilizando los tipos de datos enteros estándar en computación. Este tipo de dato es especialmente útil cuando se necesita trabajar con números extremadamente grandes, como en criptografía, algoritmos de factorización de números, cálculos científicos o financieros, entre otros.

Para la implementación de los BigIntegers se implementó la clase que internamente contiene un vector para representar en cada posición de un nodo un digito del número ingresado como una cadena de caracteres y contiene un valor booleano que permite identificar el signo del entero correspondiente.

Análisis de Complejidad

- **Operaciones Auxiliares.**
  - **vectorToString():** La complejidad de esta operación tiene una complejidad O(n) donde n es la cantidad de elementos en el vector ya que se recorre el vector de inicio a fin añadiendo los números en forma de carácter en una cadena resultante.
  - **delZero():** La complejidad de esta operación es O(n) donde n es la cantidad de ceros añadidos al final de un vector de la clase BigInteger que son añadidos en otras operaciones.
  - **Minor():** La complejidad de esta operación en el mejor de los casos es O(1) ya que si los vectores tienen diferente tamaño se devuelve la respuesta true o false sin recorrer el vector, sin embargo, en el peor caso se por el ultimo else siendo el mejor caso O(1) cuando la primera posición del primer vector es mayor o menor al segundo generando que el ciclo pare y la respuesta sea false, en el peor caso del

else sería O(n) dando el resultado true al ser ya que ninguno de los elementos del primer vector son mayores al segundo.

- **Equal():** La complejidad de esta función en el mejor de los casos es O(1) ya que compara el tamaño de 2 vectores y si son diferentes ya retorna el valor false y termina la operación, sin embargo, en el mejor de los casos del else cuando tienen el mismo tamaño sería O(1) al no tener que recorrer los vectores ya que la variable se haría false y saldría del ciclo y en el peor de los casos del else recorre todo el vector y todos los datos sin iguales dando por resultado true y haciendo un recorrido total haciendo que sea O(n) donde n es el tamaño de cualquiera de los 2 vectores.
- **Sum():** La complejidad de esta operación es O(n) donde n es el tamaño del vector más grande o cualquiera de los vectores en el caso de que sean de igual tamaño, ya que se recorre todas las posiciones del vector sumando y en el caso de necesitar un número adicional el push_back en un vector es O(1) amortizado, que no afecta la complejidad anteriormente dicha.
- **subtraction():** La complejidad de esta operación es O(n) donde n es el tamaño del vector más grande, ya que se recorre todas las posiciones del vector restando posición a posición sumando y en el caso de necesitar un número adicional el push_back en un vector es O(1) amortizado, que no afecta la complejidad anteriormente dicha.
- **subtraction2():** Al igual que la operación anterior tiene la misma complejidad, pero se usa al cambiar de orden los números, teniendo el mayor siempre arriba, pero modificando siempre el primero, dando al igual una complejidad O(n) donde n es el tamaño del vector mayor.
- **multiplication():** La complejidad de esta operación es O(n) donde n es el tamaño del vector ya que multiplica cada posición del vector por un entero dado, y al igual que la suma si se necesita agregar un número es O(1) amortizado.

- **Operaciones del TAD**
  - **Add():** La complejidad de esta operación es O(n) ya que dependiendo de los signos de los BigInteger ingresados hace llama la función sum que es O(n) o subtraction/subtraction2 son O(n) igualmente.
  - **Product():** La complejidad de esta operación es O(n*m) donde n es el tamaño del vector del primer BigInteger y m el tamaño del vector del segundo ya que se hacen m veces la operación suma y multiplicación siendo ambas O(n).
  - **Substract():** La complejidad de esta operación es O(n) ya que dependiendo de los signos de los BigInteger ingresados hace llama la función sum que es O(n) o subtraction/subtraction2 son O(n) igualmente.
  - **Quotient():** La complejidad de esta operación O(n^2).
  - **Remainder():** La complejidad de esta operación O(n^2).
  - **Pow():** La complejidad de esta operación es O(n^2*x) ya que esta operación utiliza la operación product pero al ser el mismo número operado con el mismo un numero de veces pasa de ser O(n*m) a O(n*n) y esta operación se hace x veces donde x es el numero entero proporcionado.

- **sumarListasValores():** La complejidad de esta operación es O(n*m) donde ya que se hacen m veces la operación add que tiene complejidad O(n).
- **multiplicarListasValores:** La complejidad de esta operación O(n*m*x) donde se hacen x veces la operación product entre un BigInteger de tamaño m y otro de tamaño m.
- **toString():** La complejidad de esta operación es O(n) donde n es el tamaño del vector del BigInteger que se proporciona ya que se hace push_back(O(1)) en una cadena n veces.

- **Sobrecarga de operadores**
  - **operator+:** La complejidad de esta operación es O(n) ya que se hace uso de la operación add que tiene dicha complejidad.
  - **operator-:** La complejidad de esta operación es O(n) ya que se hace uso de la operación substract que tiene dicha complejidad.
  - **operator*:** La complejidad de esta operación es O(n*m) ya que se hace uso de la operación product que tiene dicha complejidad.
  - **operator/:** La complejidad de esta operación es O(n^2).
  - **operator%:** La complejidad de esta operación es O(n^2).
  - **operator==:** La complejidad de esta operación es en el mejor de los casos O(1) ya que el tamaño de los vectores dados anteriormente o los signos son diferentes saliendo de la función con false, en el peor de los casos es O(n) donde n es el tamaño de los vectores ya que al ser iguales se recorriendo por completo.
  - **operator<:** La complejidad de esta operación es en el mejor de los casos O(1) ya que al segundo BigInteger ser menor al primero o al ser iguales sale de la función con false y en el else en el mejor de los casos es O(1) ya que al ser menor el ultimo elemento del primer vector para el ciclo y da true o cuando es mayor ya que para y retorna false y en el peor de los casos es O(n) ya que ningún numero es mayor y el numero igual recorriendo los n elementos del vector.
  - **operator<=:** La complejidad de esta operación es en el mejor caso es O(1) ya que sería menor en la primera comparación y saldría del ciclo y en el peor de los casos es O(n) donde n es el tamaño del vector al ser iguales.