Complejidades del Proyecto BigInteger

Fecha: 28/05/2023

Por: Kevin Yavari Yoshioka

1 Introducción

En esta presentación, se detallarán las complejidades asociadas al proyecto BigInteger, que consiste en la implementación de una biblioteca de números enteros de longitud arbitraria en el lenguaje de programación C++. La finalidad de este proyecto es proporcionar una herramienta versátil para realizar operaciones matemáticas con números enteros de gran magnitud, superando las limitaciones impuestas por los tipos de datos estándar.

En el desarrollo del proyecto BigInteger, se han empleado diversas funciones para realizar operaciones básicas como suma, resta, multiplicación y división, así como funciones avanzadas como exponenciación y cálculo de factorial. Cada una de estas funciones presenta una complejidad algorítmica que determina la eficiencia y el rendimiento de la biblioteca.

El objetivo principal de este informe es analizar en detalle las complejidades asociadas a cada una de las funciones implementadas en el proyecto BigInteger. Se examinarán los algoritmos utilizados, se calcularán las complejidades en términos de tiempo y espacio, y se discutirán las implicaciones prácticas de estas complejidades.

A través de este informe, se espera proporcionar una comprensión clara de las complejidades inherentes al proyecto BigInteger, permitiendo una evaluación precisa de su rendimiento y eficiencia en diferentes escenarios de uso. Además, se discutirán posibles mejoras y optimizaciones para minimizar las complejidades y maximizar el rendimiento en futuras iteraciones del proyecto.

A continuación, se presentarán los resultados obtenidos en el análisis de las complejidades de cada función implementada en el proyecto BigInteger, proporcionando una visión detallada de las fortalezas y limitaciones de esta biblioteca de números enteros de longitud arbitraria.

2 Lista De Funciones

- BigInteger()
- BigInteger(const string& numero)
- BigInteger(const BigInteger& other)
- void add(BigInteger& other)
- void product(BigInteger& other)
- void subtract(BigInteger& other)
- void quotient(BigInteger& other)
- void remainder(BigInteger& other)
- void pow(const int& other)
- string toString()

- BigInteger operator+(BigInteger& other)
- BigInteger operator-(BigInteger& other)
- BigInteger operator*(BigInteger& other)
- BigInteger operator/(BigInteger& other)
- BigInteger operator%(BigInteger& other)
- bool operator==(BigInteger& other)
- bool operator¡(BigInteger& other)
- bool operator¡=(BigInteger& other)
- static BigInteger sumarListaValores(list¡BigInteger¿& lis)
- static BigInteger multiplicarListaValores(list¡BigInteger¿& lis)
- string sumar(BigInteger& other)
- string restar(BigInteger& other)
- void borrarCeros(list¡int¿& other)
- bool menor(BigInteger other)

Funciones constructoras

BigInteger();

la complejidad de BigInteger() es constante, lo que significa que el tiempo de ejecución es siempre el mismo.

BigInteger(const string& numero)

La complejidad total de la función constructora BigInteger(const string numero) puede considerarse O(n), donde n es el tamaño de la cadena de entrada. Esto se debe a que el bucle principal que recorre la cadena tiene una complejidad lineal, y el resto de las operaciones son de complejidad constante.

BigInteger(const BigInteger& other)

La complejidad total de la función constructora BigInteger(const BigInteger other) puede considerarse O(m), donde m es el tamaño de la lista del objeto other. Esto se debe a que la operación principal de copiar la lista implica recorrer todos los elementos de la lista y realizar operaciones de copia, lo cual tiene una complejidad lineal en función del tamaño de la lista.

funciones auxiliares

Nota: antes de explicar las funciones que hacen las operaciones es mejor explicar las auxiliares ya que apartir de estan se crear las diversas operaciones.

string sumar(BigInteger& other)

La complejidad total de la función sumar(BigInteger other) puede considerarse O(max(tamlista, tamother)), donde tamlista y tamother son los tamaños de las listas del objeto actual y del objeto other, respectivamente. Esto se debe a que las operaciones principales de ajuste de listas y recorrido de las listas tienen una complejidad lineal en función del tamaño de la lista más larga

string restar(BigInteger& other)

esta tendria la misma complejidad de la funcion sumar ya que para esta funcion se realiza la misma operacion buscando al de mayor tamaño para iterar esa cantidad de veces

void borrarCeros(list¡int¿& other)

La función borrarCeros(list¡int¿ other) elimina los ceros innecesarios al principio de una lista de enteros. En pocas palabras, la complejidad de esta función puede considerarse lineal en función del número de ceros al principio de la lista.

operaciones

add(BigInteger other)

la complejidad de esta funcion seria igual a la funcion sumar ya que la funcion add(BigInteger other) realmete solo ejecuta esta funcion

substrac(BigInteger other)

esta funcion tendria la misma complejidad que la funcion add

quotient(BigInteger other)

la complejidad del operator% es O(n**2)

remainder(BigInteger other)

la complejidad del operator% es O(n**2)

toString()

la complejidad de toString() es O(n), donde n es el número de elementos en la lista lista.

void pow(const int& other)

la complejidad de pow() es O(1) cuando other es cero, y O((other - 1) * n**2) cuando other es diferente de cero, donde n es el número de elementos en la lista más grande.

void product()

la complejidad total de la función product() está dominada por la multiplicación de los dos números, lo que resulta en una complejidad cuadrática O(n**2), donde n es el tamaño de los números representados por los objetos *this y other.

Sobre carga de operadores

operator-(BigInteger other)

esta tendria la misma complejidad de substrac con la diferencia de que no se reemplaza ningun valor solo se retorna un BigInteger

operator+(BigInteger other)

esta tendria la misma complejidad de add con la diferencia de que no se reemplaza ningun valor solo se retorna un BigInteger

operator*(BigInteger other)

la complejidad de operator*() es O(n**2) en la implementación actual, donde n es el número de elementos en la lista más grande entre lista y other.lista.

operator%(BigInteger other)

la complejidad del operator% es O(n**2)

operator/(BigInteger other)

la complejidad del operator% es O(n**2)

bool operator==(BigInteger& other);

la complejidad total de la función operator==() es O(n), donde n es el tamaño de las listas de dígitos en los objetos BigInteger.

bool operator¡(BigInteger other)

la complejidad total de la función operator¡() es O(n), donde n es el tamaño de las listas de dígitos en los objetos BigInteger.

bool operator¡=(BigInteger other)

la complejidad de esta sobre carga seria la suma de las dos complejidad anteriores pero alfinal este seria igualmente O(n)

Otras funciones

string sumar(BigInteger& other)

la complejidad de la función sumarListaValores() es O(n * m), donde n es el número de elementos en la lista y m es el tamaño de la lista de dígitos en cada objeto BigInteger.

void borrarCeros(list¡int¿& other)

dado que se tendria que ejecutar n veces, la complejidad total se vuelve O((n**2) * k). k siendo la cantidad de elementos en la lista

menor(BigInteger other)

la complejidad de esta funcion en O(n) n representa la cantidad de elementos que hay en other ya que en el peor caso other y el objeto tendrias el mismo tamaño y se tendria que recorrer por completa