Análisis de Complejidad Proyecto

Jhon Deiby Mejias Cuevas

29/05/2023

Contents

1 Introducción

El presente documento tiene como objetivo analizar la complejidad de las operaciones implementadas en la clase "BigInteger". El "BigInteger" es una estructura de datos que permite manejar números enteros de tamaño arbitrariamente grande. A medida que los números crecen en tamaño, las operaciones matemáticas se vuelven más complejas y requieren un análisis cuidadoso de su eficiencia.

En este análisis, nos enfocaremos en evaluar la complejidad de los constructores, funciones y sobrecargas de operadores implementados en la clase "BigInteger". Además, exploraremos la complejidad de las operaciones de comparación y algunas operaciones estáticas.

Es importante comprender la complejidad de estas operaciones, ya que nos brinda información sobre el rendimiento y eficiencia del uso de la clase "BigInteger". La complejidad temporal nos indica cuánto tiempo tomará ejecutar una operación a medida que crece el tamaño de los números involucrados.

A lo largo del documento, se presentarán las complejidades de las operaciones en notación Big O, que nos permite clasificar la eficiencia de las operaciones en función del tamaño de los datos. Esta información será de utilidad para seleccionar la mejor estrategia al realizar cálculos con números grandes y optimizar el rendimiento de los algoritmos que utilizan la clase "BigInteger".

2 Desarrollo

2.1 Complejidad de Constructores

- **BigInteger(const string cadena)**: La función tiene una complejidad de tiempo de O(n), donde n es el tamaño del string de entrada. Esto se debe a que hay un bucle 'for' que itera desde el inicio hasta el final de la cadena, y en cada iteración se realiza la operación 'pushback'. En general, la operación 'pushback' tiene una complejidad amortizada de O(1), lo que significa que agregar un elemento al final del vector es eficiente en la mayoría de los casos. Sin embargo, en el peor caso, la complejidad de 'pushback' podría ser O(n). Esto ocurre cuando la capacidad actual del vector se agota y se necesita realizar una realocación completa de memoria para acomodar el nuevo elemento. En este escenario, si ocurre en cada iteración del bucle 'for', la complejidad general de la función sería O(n**2), ya que se realizaría una realocación y copia de elementos en cada iteración.

- **BigInteger(const BigInteger bignumber)**: La complejidad de la función es O(n), donde n es el tamaño del vector dentro de la clase 'BigInteger'. En la mayoría de los casos y en promedio, la complejidad es O(n), mientras que en el peor caso puede llegar a ser O(n**2) debido a posibles realocaciones y copias completas del vector en cada iteración.

2.2 Complejidad de Funciones

- **add**:la función add tiene una complejidad de O(n), donde n es el tamaño del vector más grande entre vec1 y vec2.La función utiliza un bucle while que itera mientras max1 o max2 sean mayores que cero, lo cual implica que el bucle se ejecutará n veces.En cada iteración, se realizan operaciones de suma y asignación en tiempo constante. Además, al final del bucle, se realiza una operación de eliminación de elementos en el vector vec3 si es necesario. En general, la función realiza una suma de números representados por vectores de enteros en un algoritmo de suma "columna a columna" similar al que se realiza a mano.

- **product**: La función realiza la multiplicación de dos números grandes representados por listas de dígitos. En términos de dificultad, la función es algo compleja. Si el número de dígitos en los números es 'n', entonces llevaría un tiempo aproximado de 'n al cuadrado' realizar la multiplicación. Esto se debe a que hay dos bucles que se ejecutan uno dentro del otro, y para cada dígito de uno de los números, se realiza una multiplicación con todos los dígitos del otro número. También hay una operación para eliminar ceros innecesarios al final, pero no afecta mucho la dificultad general.

- **substract**:la función substract tiene una complejidad de O(n), donde n es el tamaño del vector más grande entre vec1 y vec2. La función utiliza

un bucle for que itera sobre el vector vec1, lo cual implica que el bucle se ejecutará n veces. En cada iteración, se realizan operaciones de resta y asignación en tiempo constante. También se realiza una operación de eliminación de elementos en el vector vec3 al final del bucle, si es necesario. Similar a la función add, substract realiza una resta de números representados por vectores de enteros en un algoritmo "columna a columna" para restar dígitos.

- **quotient**: La complejidad en el peor caso de la función quotient es de O(n**2), donde n es el tamaño del vector dividendo. Esto se debe a que el bucle while se ejecuta aproximadamente n veces, y en cada iteración se realizan operaciones de resta y suma que tienen una complejidad de O(n). En el peor caso, asumiendo una reducción de tamaño de 1 en cada iteración, el número total de operaciones se aproxima a la suma de los primeros n números naturales, lo que da como resultado una complejidad cuadrática.

- **remainder**:La complejidad de la función remainder en el peor caso es de O(n**2), donde n es el tamaño del vector dividendo. Esto se debe a que el bucle while se ejecuta aproximadamente n veces, y en cada iteración se realiza una operación de resta que tiene una complejidad de O(n). Utilizo el mismo codigo de la divison para remainder solo que no hago la suma al conciente que no es necesaria ya que solo quiero el residuo.

- **pow**:La complejidad de la función pow en el peor caso es de O(n**3), donde n es el exponente recibido como entrada. Esto se debe a que hay un bucle for que se ejecuta n veces, y dentro de cada iteración se realiza una operación de multiplicación que tiene una complejidad de O(n**2). En el peor caso, el número total de operaciones se aproxima a n veces la complejidad de la operación de multiplicación, lo que resulta en una complejidad cubica.

- **toString**:La función toString tiene una complejidad de tiempo de O(n), donde n es el tamaño del vector vec1. Esto significa que el tiempo de ejecución aumenta proporcionalmente al tamaño del vector.

  La función recorre el vector vec1 en un bucle for, comenzando desde el último elemento hasta el primero. En cada iteración, se realiza una concatenación de caracteres, convirtiendo el valor entero en un carácter y agregándolo a la cadena cadena. Esta operación de concatenación tiene una complejidad constante, ya que no depende del tamaño del vector.

2.3 Complejidad de Sobrecargas

- **operator==**:En el mejor de los casos la complejidad es constante cuando los vectores tienen diferente tamaño o diferentes signos, pero en el peor de los casos La complejidad es 0(n) donde n es el tamaño del vector, ya que la función luego, en un bucle while, compara cada elemento de los

vectores vec1. Si encuentra alguna diferencia, establece ans en falso y sale del bucle, en el pero de los casos recorre hasta el ultimo. Si los vectores son vacíos, establece ans en verdadero. Finalmente, devuelve el valor de ans. x

- **operator¡**: La complejidad de esta función es constantes en el mejor de los casos cuando los vectores tienen diferente tamaño o tienen diferentes signos, sin embargo en el peor de los casos es O(n), donde n es el tamaño del vector 'BigInteger' comparados. Comienza comprobando los signos de los objetos y establece result y encontrar en función de las condiciones. Luego, verifica si los tamaños de los vectores vec1 son diferentes y realiza comparaciones adicionales para determinar si el primer objeto es menor que el segundo. Si no se cumple ninguna de las condiciones anteriores, utiliza un bucle while para comparar elemento por elemento de los vectores. En función de los valores encontrados, establece result en verdadero o falso. Finalmente, devuelve el valor de result.

- **operator¡=**: la función operator¡= utiliza el operador ¡ y el operador == para verificar si un objeto BigInteger es menor o igual que otro. Llama a los operadores ¡ y == en el objeto actual y bignumber para realizar las comparaciones correspondientes. Si alguna de las comparaciones es verdadera, establece ans en verdadero. Finalmente, devuelve el valor de ans. Por lo tanto como utiliza ambas operaciones, la complejidad es O(n) tambien donde n es el tamaño del vector comparado.

- La complejidad de los siguientes operadores depende en gran medida de la implementación de sus funciónes anteriormente explicadas, ya que su principal tarea es realizar llamar a dicha función. También utiliza el constructor, pero como hemos observado, su complejidad es lineal y no afecta significativamente. Por lo tanto, la complejidad total del operador será determinada principalmente por la complejidad de la función a continuación recordamos la complejidad:

- **operator+** O(n)

- **operator-** o(n)

- **operator*** o(n**2)

- **operator/**: o(n**2)

- **operator modulo"**: o(n**2)

2.4 Complejidad de las Operaciones Estáticas

- **static BigInteger sumarListaValores(vector¡BigInteger¿ list)**:La complejidad de la función sumarListaValores es O(m * n), donde m es la cantidad de elementos en la lista y n es el tamaño del vector más grande entre los BigInteger de la lista. El bucle for recorre cada elemento de la

lista y llama a la función add, que tiene una complejidad de O(n), para sumar los BigInteger.

- **static BigInteger multiplicarListaValores(vector¡BigInteger¿ list)**:La complejidad de la función multiplicarListaValores es O(m * n**2), donde m es la cantidad de elementos en la lista y n es el tamaño del vector más grande entre los BigInteger de la lista. El bucle for recorre cada elemento de la lista y llama a la función product, que tiene una complejidad de O(n**2), para multiplicar los BigInteger.