Proyecto Estructura de datos

Oscar Vargas Pabon

8975574

Aquí se encuentra la complejidad de todas las operaciones definidas en la interfaz del TAD BigInteger.

Funciones abstractas:

Estas funciones van a ser usadas para expresar la complejidad de algunas operaciones.

$$\max(a,b)\begin{cases}a, si\ a \geq b\\ b, si\ a < b\end{cases}$$

$$\min(a,b)\begin{cases}a, si\ b \geq a\\ b, si\ b < a\end{cases}$$

Definamos max para arbitraria cantidad de elementos:

$$\max(a_1, a_2, \ldots, a_k) = a_j \mid \forall_{i\in[1,k]}\ a_j \geq a_i$$

Constructores:

La clase BigInteger cuenta con tres constructores:

- Un constructor por defecto que inicia el BigInteger como 0
- Un constructor que recibe un std::string compuesto de dígitos y construye el BigInteger a partir de estos.
- Un constructor que recibe otra instancia de BigInteger y la copia.

Complejidad:

Constructor por defecto:

Este tiene una complejidad constante, pues lo único que hace es iniciar el signo del BigInteger.

$$T(n) = O(1)$$

Constructor que recibe un std::string, ‘s’:

> Este tiene una complejidad lineal en base al tamaño del std::string, pues lo que hace la operación es recorrer el std::string convirtiendo los dígitos en Ascii a los valores que representan, guardándolos en el BigInteger en el proceso.

$$n := s.size(), T(n) = O(n)$$

Constructor que recibe otra instancia de BigInteger ‘b’:

> Este tiene una complejidad lineal en base a la cantidad de dígitos que tiene el BigInteger que recibe, pues lo único que hace es copiar estos dígitos.

$$n := b.getDigitSize(), T(n) = O(n)$$

Operaciones aritméticas:

Nota: las versiones que sobrecargan operadores y las que modifican la instancia BigInteger con las que son invocadas, solo tienen la diferencia de que las que crean una nueva instancia realizan una invocación extra al constructor que recibe otra instancia de BigInteger.

add – substract:

Estas dos operaciones las considero en el mismo apartado, pues la única diferencia entre ambas es que ‘substract’ cambia el signo del segundo BigInteger. De hecho, internamente, ambas operaciones invocan a la misma función auxiliar.

Este apartado, a su vez, tiene incorporados un algoritmo para sumar y otro para restar que se usan dependiendo de los signos de los BigInteger. Estos van a ser considerados aparte para su complejidad, sin embargo, su complejidad en la notación Big-O es la misma.

Vamos a hacer referencia las instancias que realizan la suma – resta como: BigInteger ‘a’ y BigInteger ‘b’.

Complejidad add - substract:

Cuando ‘a’ y ‘b’ tienen el mismo signo (se realiza el algoritmo de la suma):

Este algoritmo recorre ambas instancias BigInteger desde el menor digito hasta el mayor. Mientras los recorre, va sumando ambos dígitos y guardando el resultado de esta suma en el BigInteger que devolverá la respuesta. Sin embargo, no siempre ambos BigInteger van a tener la misma cantidad de dígitos; en estos casos, terminaremos iterando tantas veces como la instancia de BigInteger con más dígitos (pues debemos seguir considerando cosas como el acarreo). Es por esto que la complejidad es la mayor entre el tamaño (en dígitos) de ambos BigInteger.

$$n \coloneqq \max(\, a.getDigitSize(), b.getDigitSize()\,), T(n) = O(n)$$

Cuando 'a' y 'b' tienen signos contrarios (se realiza el algoritmo de la resta):

Este resulta muy parecido a lo que hacemos cuando 'a' y 'b' se suman; las únicas diferencias resultan en que, en vez de realizar una suma realizamos una resta y que debemos determinar si $|a| > |b|$ o $|b| > |a|$. La primera diferencia no tiene ningún impacto en la complejidad. La segunda diferencia resulta en una invocación extra a una operación de comparación. Esta invocación extra no afecta la complejidad, pues las comparaciones funcionan en $O(\min(\, a.getDigitSize(),\ b.getDigitSize()))$, lo que es lineal en el menor tamaño entre 'a' y 'b'.

Si $l \coloneqq \max(\, a.getDigitSize(), b.getDigitSize())$ y $h \coloneqq \min(\, a.getDigitSize(),\ b.getDigitSize())$, entonces esta operación funciona en tiempo $O(\, l + h\,)$. Esto es en el peor de los casos, cuando $a.getDigitSize() == b.getDigitSize()$, $O(2 * a.getDigitSize()) = O(a.getDigitSize()) = O(l)$. Por esto la resta tiene complejidad lineal entre el mayor tamaño (en dígitos) de los dos BigInteger.

$$n \coloneqq \max(\, a.getDigitSize(), b.getDigitSize()\,), T(n) = O(n)$$

Por esto, tanto 'add', '+', 'substract' y '-' tienen complejidad:

$$n \coloneqq \max(\, a.getDigitSize(), b.getDigitSize()\,), T(n) = O(n)$$

producto:

Esta operación esta implementada con el algoritmo que se usa comúnmente para multiplicar números de varios dígitos.

Vamos a hacer referencia las instancias que realizan la multiplicación como: BigInteger 'a' y BigInteger 'b'.

Complejidad product:

Esta operación está regida por dos ciclos for. El primero recorre los dígitos de 'a', mientras que el segundo recorre los dígitos de 'b'. El segundo for esta anidado en el primero, por lo que este va a iterar $a.getDigitSize() * b.getDigitSize()$ veces.

Si $n \coloneqq \max(a.getDigitSize(), b.getDigitSize())$, nótese que $n^2 \geq a.getDigitSize() * b.getDigitSize()$, por lo que se pude decir que este algoritmo es cuadrático en n.

$$n \coloneqq \max(a.getDigitSize(), b.getDigitSize()), T(n) = O(n^2)$$

Remainder – quotient:

Estas dos operaciones son implementadas usando la misma operación auxiliar. Esta retorna tanto el residuo como el cociente, pues esta realiza la división euclidiana: $a = quotient * b + remainder$. Por este motivo, sus complejidades son las mismas y serán analizadas en el mismo apartado.

Vamos a hacer referencia la instancia que actúa como dividendo como BigInteger 'a' y la instancia que actúa como divisor BigInteger 'b'. Diremos también que 'Base' hace referencia a la base con la que se están guardando los datos dentro de BigInteger, la cual es en este caso base 10.

Complejidad remainder – quotient:

El algoritmo para la división euclidiana intenta imitar lo que se hace cuando se intentan dividir dos números a mano. Esto se hace en la operación agregándole ceros a la izquierda a 'b' hasta que $a.getDigitSize() == b.getDigitSize()$ para de esta manera restar tantas veces como este 'b' con ceros añadidos cabe en 'a' y guardándolo en el cociente con la misma cantidad de ceros que le agregamos. Esto se realiza mediante un for que recorre la cantidad de ceros que añadimos a 'b', los cuales serían $a.getDigitSize() - b.getDigitSize()$, con un while anidado que realiza invocaciones a una operación de comparación y a 'substract'. Como la complejidad de 'substract' es mayor o igual que la complejidad de una operación de comparación, entonces consideraremos su complejidad. Antes de seguir, nótese que el while solo puede iterar 'Base'-1 veces, pues de lo contrario el digito terminaría con un numero mayor o igual a 'Base'. Además, como

$$l \coloneqq a.getDigitSIze(), \qquad h \coloneqq b.getDigitSize()$$

$$T(n) = (l - h) * Base * O(\max(\mathrm{l}, \mathrm{h}))$$

Sin embargo, necesariamente $l > h$ porque de lo contrario el for iteraria 1 vez, siendo este el mejor caso. En caso de que $l == h$, la operación funciona en $O(l * Base)$. Si $l < h$, entonces funciona en $O(l)$.

$$T(n) = l * Base * O(l)$$

Entonces,

$$n := a.getDigitSize(), T(n) = O(Base * n^2)$$

Si tomamos la base como una constante, entonces tendríamos:

$$n := a.getDigitSize(), T(n) = O(n^2)$$

Podemos concluir que tanto 'quotient', '/', 'remainder' y '%' funcionan en tiempo cuadrático en el peor caso.

$$n := a.getDigitSize(), T(n) = O(n^2)$$

Pow:

Esta operación funciona en base a 'product'.

El exponente será referenciado como 'exponente', el BigInteger como 'a'.

Complejidad pow:

Esta operación tiene dos whiles, uno que llena un std::stack y otra que utiliza el std::stack para determinar un condicional. En el primer while, cada iteración añade un elemento al std::stack, mientras que en el segundo cada iteración elimina un elemento del std::stack (Esto quiere decir que ambos whiles iteran la misma cantidad de veces). El primer while realiza la división entera sobre 'exponente' hasta que 'exponente' == 0, lo que quiere decir que va a iterar $log_2(exponente)$ veces. El segundo while hace entre 1 y 2 invocaciones a 'product', el cual funciona en $T(n) = O(n^2)$. Tomaremos el caso en donde realiza las 2 invocaciones (cuando el std::stack solo tiene apilado el valor 'true').

$$donde\ e := exponente, n := \mathrm{a.getDigitSize}()$$

Se realiza un trabajo de $log_2(e) * (2 * O(n^2))$ en el segundo while. Esto resulta en:

$$T(n) = O(\log e * n^2)$$

Las operaciones ‘pow’ y ‘powNew’ funcionan en:

$$e := exponente, n := \text{a.getDigitSize}(), \qquad T(n) = O(\log e * n^2)$$

Operaciones de comparación:

Todas las funciones ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’ y ‘!=’ terminan invocando a la misma función auxiliar. Es por esto que todas las operaciones de comparación tienen la misma complejidad.

Las dos instancias de BigInteger a comparar serán referidas como ‘a’ y ‘b’.

Complejidad:

Si los signos o el tamaño (en dígitos) de ‘a’ y ‘b’, son distintos, se puede determinar fácilmente el mayor del menor en tiempo $O(1)$. Sin embargo, si $a.getDigitsSize() == b.getDigitSize() \wedge a.getSign() == b.getSign()$ entonces tendremos que recorrer desde el digito con mayor ‘peso’ (el digito asociado a un mayor valor por la base) hasta el digito con menor ‘peso’ de ‘a’ y ‘b’ mientras hacemos comparaciones. Si son distintos podremos saber cual es el mayor y cual es el menor, de lo contrario seguiremos iterando. El peor caso se da cuando $a == b$, pues su signo, tamaño y todos sus dígitos serán iguales. En este caso funcionaria en: $O(a.getDigitSize())$. Se puede decir entonces que:

$$n := \min(a.getDigitSize(), b.getDigitSize()) \ , T(n) = O(n)$$

Otras Funciones:

Aquí dejaremos la complejidad de operaciones como:

- toString
- sumarListaValores
- multiplicarListaValores
- getSign
- getDigitSize

- getDigit

toString:

Esta operación lo que hace es transformar el BigInteger a un std::String, de manera que:

$$Para\ todo\ BigInteger\ b: b == BigInteger(\ b.toString()\ )$$

Para hacerlo, debe recorrer todos los dígitos del BigInteger e irlos agregando al std::string. Es por esto que esta operación se realiza en $O(b.getDigitSize())$.

$$n := b.getDigitSize(), T(n) = O(n)$$

sumarListaValores:

Esta operación toma una std::list<BigInteger> 'l' y retorna un BigInteger. En cierta manera es como calcular:

$$k := l.size(), \{b_i\}\ para\ cualquier\ i\ \in [1,k]\ es\ BigInteger\ de\ l$$

$$sumarListaValores(\ l\ )\ = \sum_{i=1}^{k} b_i$$

Vamos a sumar todos los elementos de 'l'. Hay que considerar que si el primer elemento tiene el mayor tamaño, entonces vamos a realizar 'k' sumas donde el mayor tamaño entre ambos BigInteger va a ser el del elemento de mayor tamaño de 'l'. Este es entonces el peor caso.

La complejidad es entonces:

$$k := l.size(), n := \max(b_1, .., b_k)\ , T(n) = O(k * n)$$

multiplicarListaValores:

Esta operación hace lo mismo que 'sumarListaValores', con la única diferencia de que en vez de usar 'add', se usa 'product'.

$$k := l.size(), n := \max(b_1, .., b_k)\ , T(n) = O(k * n^2)$$

getSign:

Esta operación solo consulta el atributo privado ‘sign’ y lo retorna.

Su complejidad es:

$$T(n) = O(1)$$

getDigitSize:

Esta operación solo invoca a la función ‘size’ de std::list y retorna este valor. Como esta función de std::list funciona en $O(1)$, entonces la complejidad de ‘getDigitSize’ es:

$$T(n) = O(1)$$

getDigit:

llamaremos ‘a’ la instancia BigInteger sobre la que es invocada esta función.

Esta operación debe recorrer la lista desde algún extremo hasta llegar al digito ubicado en la posición que se le pasa por parámetro. El peor caso se da cuando la posición solicitada se encuentra es un digito en la mitad $posicion = a.getDigitSize()/2$. En este caso, debe recorrer $a.getDigitSize()/2$ digitos. En cualquier otro caso, buscará el extremo más cercano a la posición solicitada y recorrerá menos que $a.getDigitSize()/2$ digitos. Es por esto que esta funcion opera en:

$$n \coloneqq a.getDigitSize(), T(n) = O(n)$$