Informe de análisis de complejidad TAD BigInteger

Samuel Francisco Moncayo Moncayo

Facultad Ingeniería y Ciencias, Pontificia Universidad Javeriana

Estructuras de Datos

Carlos Alberto Ramírez Restrepo, Luis Gonzalo Noreña

30 de mayo de 2023

Análisis de complejidad de operaciones constructoras del TAD BigInteger

- `BigInteger::BigInteger(const string &num)`
  - La primera parte del código tiene una complejidad constante O(1). La asignación del primer dígito de num a bigInteger se realiza en tiempo constante, al igual que la asignación del signo.
  - El bucle for itera a través de la cadena num desde el índice 1 hasta size-1, lo que implica una complejidad lineal O(n-1) o simplemente O(n) en el peor de los casos. En cada iteración del bucle, se realiza una asignación del dígito de num a bigInteger y se agrega a la lista bigInteger. Dado que la operación de agregar un elemento a una lista enlazada tiene una complejidad O(1), la complejidad total del bucle es O(n).

- `BigInteger::BigInteger(const BigInteger &num)`
  - Se está utilizando una lista enlazada, por ello la copia de la lista tendrá una complejidad lineal en función del tamaño de la lista, es decir, O(n), donde "n" es el número de elementos en la lista.

Análisis de complejidad de operaciones modificadoras del TAD BigInteger

- `void BigInteger::add(BigInteger &num) void BigInteger::auxAdd(BigInteger &num)`
  - La primera parte del código (las dos secciones if que ajustan el tamaño de las listas) en el peor de los casos tendra que recorrer una lista casi por completo por lo que representaremos como k las iteraciones (k es el tamaño de alguna de las dos listas).
  - Luego, los iteradores it1 e it2 recorren las listas bigInteger y num.bigInteger, respectivamente, en orden inverso. Ambas listas tienen el mismo tamaño por lo que el ciclo iteraria el tamaño de la primera lista j(j es el tamaño de la primera lista), ya que se recorren ambas listas una vez.
  - En resumen, la complejidad total del código es O(n), donde n es la suma de ambas iteraciones de los ciclos
- `void BigInteger::substract(BigInteger &num) void BigInteger::auxSubstract (BigInteger &num)`
  - La sección de código que ajusta el tamaño de las listas tiene una complejidad O(n), donde n es la diferencia en tamaño entre las listas bigInteger y num.bigInteger. Esto se debe a los bucles for que agregan ceros al principio de la lista correspondiente. La sección de código que realiza la substracción tiene una complejidad O(n), donde n es el tamaño de las listas bigInteger y num.bigInteger. Esto se debe a los bucles while y for que recorren las listas y realiza las operaciones de la resta.
  - La funcion eraseZero() tienen una complejidad O(n), donde n es el tamaño de la lista bigInteger o num.bigInteger. Esta funcion eliminan los ceros a la izquierda del bigInteger.

- En general, la complejidad total del código es O(n), donde n es el tamaño de las listas bigInteger y num.bigInteger.

- ```
  void BigInteger::product(BigInteger &num) void BigInteger::auxProduct(BigInteger &num)
  ```
  - La complejidad del código es O(n * m), donde n es el tamaño de la lista bigInteger y m es el tamaño de la lista num.bigInteger. Esto se debe a que hay dos bucles anidados en el código, uno que itera sobre num.bigInteger y otro que itera sobre bigInteger. Dentro de estos bucles, se hacen operaciones que tienen complejidad constante y la función eraseZero practicamente no afecta mucho la complejidad.

- ```
  void BigInteger::quotient(BigInteger &num) void BigInteger::remainder(BigInteger &num)
  ```
  - Copiar una lista (saver.bigInteger = bigInteger) utilizando la operación del tipo de dato list de c++ tiene una complejidad lineal en función del tamaño de la lista bigInteger n(siendo el tamaño de la lista).
  - Comparar dos objetos BigInteger (num > saver y num > tempo) utiliza la sobre carga del operador > que recorre las listas para revisar si es mayor o no implica comparar en el peor de los casos toda la lista. La complejidad depende del tamaño de las listas m, donde m es el tamaño de la lista más grande entre num.bigInteger y saver.bigInteger.
  - Realizar una multiplicación (productNum.auxProduct(num)) y una resta
  - (productNum.auxSubstract(num)) La complejidad de ambas ya esta expresada como O(n) n siendo el tamaño de la lista involucrada. Si consideramos n como el número de dígitos de num y m como el número de dígitos de productNum ya que es un while que en su condición recorre una lista internamente, la complejidad hasta este punto es O(n * m).
  - Considerando que albergamos un while dentro de otro y la condicion del interno tambien contiene un ciclo, la complejidad total del código es aproximadamente O(n * m * k), donde n es el número de dígitos de num, m es el número de dígitos de productNum y k es el número de dígitos de bigInteger
- ```
  void BigInteger::pow(int num)
  ```
  - Tenemos un ciclo que opera un BigInteger las veces a la que este elevado, por lo tanto, la función auxProduct sabemos que tiene complejidad O(n), n siendo el tamaño de la lista y esta función esta anidada en un ciclo que itera k veces que seria el numero que recibe la función, teniendo una complejidad O(n*k) n siendo el tamaño de los Big Integers y k el numero al que se eleva.
- ```
  string BigInteger::toString()
  ```
  - La función toString() itera sobre los elementos de la lista bigInteger y los convierte en caracteres para construir la representación en forma de cadena del objeto BigInteger. El número de iteraciones es igual al tamaño de la lista bigInteger, por lo que la complejidad es lineal en relación con el tamaño de la lista.
  - En el peor de los casos, se deben recorrer todos los elementos de la lista una vez para construir la cadena resultante. Por lo tanto, la complejidad es O(n), donde n es el tamaño de la lista bigInteger.

Análisis de complejidad de sobrecargas de operadores del TAD BigInteger

- `BigInteger BigInteger::operator +(BigInteger &num)`
  - Igual complejidad que la operación Add, sin embargo, es necesario sumar las declaraciones y las asignaciones de Big Integers que tendría una complejidad O(n), que no afectaría la complejidad total de las operaciones, quedándose únicamente con O(n) (n tamaño-lista).
- `BigInteger BigInteger::operator -(BigInteger &num)`
  - Igual complejidad que la operación Substract, sin embargo, es necesario sumar las declaraciones y las asignaciones de Big Integers que tendría una complejidad O(n), que no afectaría la complejidad total de las operaciones, quedándose únicamente con O(n) (n tamaño-lista).
- `BigInteger BigInteger::operator *(BigInteger &num)`
  - Igual complejidad que la operación Product, sin embargo, es necesario sumar las declaraciones y las asignaciones de Big Integers que tendría una complejidad O(n), que no afectaría la complejidad total de las operaciones, quedándose únicamente con O(n) (n tamaño-lista).
- `BigInteger BigInteger::operator /(BigInteger &num) BigInteger BigInteger::operator %(BigInteger &num)`
  - Igual complejidad que la operación Quotient y Remainder, sin embargo, es necesario sumar las declaraciones y las asignaciones de Big Integers que tendría una complejidad O(n), que no afectaría la complejidad total de las operaciones, quedándose únicamente con O(n * m * k), donde n es el número de dígitos de num, m es el número de dígitos de productNum y k es el número de dígitos de bigInteger.
- `bool BigInteger::operator ==(BigInteger &num)`
  - La complejidad es O(n), donde n es el tamaño de la lista bigInteger.
  - En el peor de los casos, donde ambas listas tienen el mismo tamaño y todos los elementos son diferentes, el bucle while iterará n veces. Esto se debe a que se compara cada elemento de bigInteger con el correspondiente elemento de num.bigInteger.
- `bool BigInteger::operator<(BigInteger &num)`
  - En general la complejidad de los operadores comparativos es parecida, la complejidad del código se puede expresar como O(k), donde k es la suma de n y m que son los tamaños de las listas bigInteger y num.bigInteger, respectivamente. Esto se debe a que la comparación se detiene en el bucle while cuando se encuentra un elemento que no es igual en ambas listas o cuando se han comparado todos los elementos en la lista más corta. que en el peor de los casosambas listas son iguales.
- `bool BigInteger::operator <=(BigInteger &num)`
  - La sobre carga tiene una complejidad O(n) en el peor de los casos, ya que itera sobre los elementos de las listas bigInteger y num.bigInteger utilizando los iteradores t1 y t2. La condición del bucle while(t1 != bigInteger.end() && flag && flag1) se evalúa en cada iteración, y en cada iteración se incrementan los iteradores t1 y t2 con ++t1 y ++t2, llegando a recorrer al mismo tiempo las listas para encontrar si es menor o no.

Análisis de complejidad de las funciones estáticas del TAD BigInteger

- `BigInteger BigInteger::sumarListaValores(list<BigInteger> &bigList)`
  - La complejidad de esta operación depende de la complejidad ya anteriormente nombrada de la función Add o Substract ambas con una complejidad lineal O(n), sin embargo, también depende del tamaño de la lista que entre puesto que las funciones se iteran dependiendo el tamaño de la lista k.
  - En conclusión, al tener dos ciclos anidados tendríamos una complejidad O(n*k), n siendo el tamaño del Big Integer y k el tamaño de la lista que alberga Big Integers.
- `BigInteger BigInteger::multiplicarListaValores(list<BigInteger> &bigList)`
  - La complejidad de esta operación depende de la complejidad ya anteriormente nombrada de la función Product, con una complejidad lineal O(n), sin embargo, también depende del tamaño de la lista que entre puesto que las funciones se iteran dependiendo el tamaño de la lista k.
  - En conclusión, al tener dos ciclos anidados tendríamos una complejidad O(n*k), n siendo el tamaño del Big Integer y k el tamaño de la lista que alberga Big Integers.