Reporte de complejidad de las operaciones del proyecto de Estructuras de Datos TAD BigInteger

María Camila Caicedo Florez

Observación: Todas las sobrecargas de los operadores aritméticos tienen la misma complejidad que sus operaciones correspondientes.

BigInteger(const deque<int> &, bool)

Está operación tiene una complejidad de O(n) donde n es la cantidad de elementos del deque que se pasa por parámetro, pues para crear una instancia BigInteger de esta manera, hay que copiar todos los elementos del deque en la lista del objeto que se va a crear.

BigInteger(const string &)

Esta operación también tiene una complejidad de O(n) pues hay que recorrer todo el string y copiar cada uno de los caracteres de este, a la lista del objeto a crear, convirtiéndolos en enteros.

BigInteger()

Esta operación se realiza en un tiempo constante ya que solo es necesario asignar un signo por defecto true, que tiene una complejidad O(1).

BigInteger (const BigInteger &)

Está operación tiene una complejidad de O(n) donde n es la cantidad de elementos del deque del BigInteger que se pasa por parámetro, pues para crear una instancia BigInteger de esta manera, hay que copiar todos los elementos del deque del otro big integer en la lista del objeto que se va a crear, como asignar el booleano de signo es constante, entonces su complejidad se puede omitir.

void add (BigInteger &)

Esta operación tiene una complejidad de O(n) donde n es la longitud del deque del BigInteger más largo. Es decir, si la longitud del deque del BigInteger que se pasa por referencia es mayor que la del BigInteger actual, entonces n va a ser la longitud de ese deque, en el caso contrario entonces n sería la longitud del deque actual. Cuando se refiere a longitud, se refiere a la cantidad de elementos que tiene cada uno de los deque.

void product (BigInteger &)

Esta operación tiene una complejidad de O(n*m) donde n es la cantidad de elementos del deque del BigInteger actual y m es la cantidad de elementos del deque que se pasa por referencia. Se llega a esta conclusión debido a que por cada elemento del BigInteger actual se recorre todo el otro BigInteger para ir multiplicando elemento a elemento. Cuando se termina el ciclo interno, se suma esa multiplicación a la multiplicación anterior. Ya se

mencionó anteriormente que el add tiene una complejidad de O(n), sin embargo, al estar anidado en un ciclo y tomando en cuenta que puede tener desde 0 hasta 3 dígitos más que el BigInteger actual, hace que también se deba tener en cuenta al calcular la complejidad. Así que en realidad m sería la longitud de elementos que tiene el BigInteger que guarda las sumas o la cantidad de elementos del segundo BigInteger dependiendo de cual sea mayor.

void substract (BigInteger &)

Esta operación tiene una complejidad de O(n) donde n es la cantidad de elementos del deque del BigInteger más largo, es decir, el BigInteger con la mayor cantidad de elementos. Esto, debido a que solo hay un ciclo en esta operación y el resto de las operaciones dentro de él son constantes.

void quotient (BigInteger &)

Esta operación en el mejor caso tiene una complejidad de O(n) donde n es la longitud del deque, pues en el mejor caso, el BigInteger que se pasa por referencia es mayor que el BigInteger actual, por lo que solo se debe limpiar el deque de todos los elementos que tenía antes (lo cual es una operación con complejidad O(n)) y se le debe insertar un 0, que tiene una complejidad constante. En el peor caso es O(n*m) donde n es la longitud del deque del BigInteger actual y m la cantidad de elementos que tenga x por cada iteración, donde x es la porción del elemento que se va a dividir, como se debe realizar una resta, y la resta es lineal, es lo que se tiene en cuenta.

void remainder (BigInteger &)

Esta operación, en esencia tiene la misma complejidad que el quotient ya que el proceso es exactamente el mismo, la única diferencia es que acá no se guarda el resultado de "cuantas veces cabe un número en otro" sino, solamente la resta que se realiza en cada ciclo, que termina siendo el resultado de esta operación. Por lo que la complejidad de la operación en el peor caso termina siendo O(n*m) donde n es la longitud del BigInteger actual y m la cantidad de elementos que tenga x por cada iteración. En el mejor caso, que es cuando el BigInteger pasado por referencia es mayor que el BigInteger actual, si cambia la complejidad pues no se le tiene que hacer ninguna modificación al deque del BigInteger actual, por lo que su complejidad es O(1).

void pow (int n)

La complejidad de esta operación es de O(n*l*m) donde n es la cantidad de elementos del BigInteger actual y m el valor del entero que se pasa por parámetro. Como el BigInteger se multiplica por si mismo, ignorando el ciclo del entero, la longitud del BigInteger aumenta cada vez que se multiplica por si mismo, así que l sería esto y n sería la longitud del BigInteger inicial.

string toString ()

La complejidad de esta operación es lineal amortizada debido a que se debe recorrer todo el deque para poder añadir elemento a elemento al string, sin embargo, como realizar la

operación push_back en un string es constante amortizado, por esta razón la complejidad es O(n+).

void setSign(bool sign)

Esta operación tiene una complejidad de O(1), ya que asignar un booleano es constante en cualquier caso.

int getIth(int)

Esta operación tiene una complejidad de O(1) ya que acceder a un elemento en una posición específica de un deque es constante, que es básicamente la función de esta operación.

int getSize ()

La complejidad de esta operación es de O(n) donde n es la cantidad de elementos del deque, pues para saber su longitud, se debe recorrer todo el deque de principio a fin.

bool getSign ()

Esta operación tiene una complejidad de O(1) ya que saber el valor de un booleano es constante, que es lo que hace esta operación.

vector<BigInteger> tablaMult()

Esta operación tiene una complejidad de O(10*n), pues, como todos los números por los cuales se multiplica el BigInteger actual son de una sola cifra, su resultado es O(n), pero como esto mismo hay que hacerlo 10 veces, por eso se añade el 10. Aunque al final, la complejidad estaría dada por O(n).

bool operator== (BigInteger &)

La complejidad de este operador en el peor caso es de O(n) (que es cuando ambos BigInteger, el actual y el que es pasado por referencia son iguales), pues se deben recorrer ambos de principio a fin para asegurarse de que son iguales. En el mejor caso, la complejidad sería de O(1), pues lo primero que se hace es comparar signos y acceder a los signos de cada uno de los BigInteger es constante, y si son diferentes, ya no hace más comparaciones.

bool operator< (BigInteger &)

La complejidad de este operador en el peor caso es de O(n) (que es cuando ambos BigInteger, el actual y el que es pasado por referencia son iguales hasta el último dígito),porque se deben recorrer ambos de principio a fin para asegurarse de que son iguales. En el mejor caso, la complejidad sería de O(1), pues lo primero que se hace es comparar signos y acceder a los signos de cada uno de los BigInteger es constante. Si son diferentes, ya no hace más comparaciones y saca true o false dependiendo que BigInteger tenga que signo.

bool operator<= (BigInteger &)

La complejidad de este operador en el peor caso es de O(n) (que es cuando ambos BigInteger, el actual y el que es pasado por referencia son iguales), pues se deben recorrer ambos de

principio a fin para asegurarse de que son iguales. En el mejor caso, la complejidad sería de O(1), pues lo primero que se hace es comparar signos, como acceder a los signos de cada uno de los BigInteger es constante, si son diferentes, ya no hace más comparaciones y retorna el resultado dependiendo de cuál es menor.

static BigInteger sumarListaValores (list<BigInteger> &)

Esta operación tiene una complejidad de O(n*m) donde n es la suma de cada uno de los BigInteger, que como se mencionó anteriormente es lineal. Y m es la longitud de la lista de BigInteger que entra por referencia.

static BigInteger multiplicarListaValores (list<BigInteger> &)

Esta Operación tiene una complejidad de O(n*m*l) donde n es la longitud del BigInteger en el que se van almacenando las multiplicaciones consecutivas, m es la longitud del siguiente número por el que se multiplica (pues es necesario recorrer los dos) y l es la longitud de la lista de BigInteger que entra por referencia.