Reporte de funciones y complejidades

Julián Andrés Galvis Tocora

Estructuras de datos

CONSTRUCTORES:

BigInteger(const string& num): Este constructor permite hacer un bigInteger a partir de un Sting y su complejidad seria O(n) donde n sería el tamaño del Sting.

BigInteger(const **BigInteger& num):** Este constructor permite hacer un bigInteger a partir de otro Sting y su complejidad seria O(n) ya que se recorre el vector que esta por debajo y añade los valores a el bigInteger.

OPERADORES:

add(BigInteger& num): Esta operación permite sumar dos bigInteger cambiando el valor del primer bigInteger por la suma de los dos, esta operación tendría complejidad O(n) donde n sería el tamaño del bigInteger con más elementos.

product(BigInteger& num): Esta operación permite multiplicar dos bigInteger cambiando el valor del primero por la multiplicación de los dos, esta operación tendría complejidad O(n*m) donde n sería el tamaño de un bigInteger y m sería el tamaño del otro, este en el peor caso sería el mismo y se tendría que recorrer dos veces.

substract(BigInteger& num): Esta operación permite restar dos bigInteger cambiando el valor del primero por la resta de los dos, esta operación tendría complejidad O(n) donde n sería el tamaño del bigInteger con más elementos.

quotient(BigInteger& num): Esta operación permite dividir dos bigInteger cambiando el valor del primero por el cociente de la división del primer por el segundo, esta operación tiene la complejidad de O(n^2).

remainder(BigInteger& num): Esta operación permite obtener el residuo de dos bigInteger cambiando el valor del primero por el cociente de la división del primer por el segundo, esta operación tiene la complejidad de O(n^2).

pow(const int& num): Esta operación permite elevar un bigInteger a un entero cambiando el valor del bigInteger por la potencia a la que se solicite, esta operación tendría una complejidad

O(n^2 k) donde n^2 sería la cantidad de elementos del bigInteger y k sería el entero al cual se eleve.

string toString(): Esta operación permite convertir un bigInteger a un string para poder visualizarlo, esta complejidad seria O(n) donde n sería el tamaño del bigInteger.

BigInteger operator+(BigInteger&): Esta sobrecarga permite sumar dos bigInteger sobre cargando el operador + y permitiéndolo guardar en otro bigInteger, esta operación tendría la misma que la suma ya que esta adentro llama a la operación add().

BigInteger operator-(BigInteger&): Esta sobrecarga permite restar dos bigInteger sobre cargando el operador - y permitiéndolo guardar en otro bigInteger, esta operación tendría la misma que la resta ya que esta adentro llama a la operación substract().

BigInteger operator*(BigInteger&): Esta sobrecarga permite multiplicar dos bigInteger sobre cargando el operador * y permitiéndolo guardar en otro bigInteger, esta operación tendría la misma que la multiplicación ya que esta adentro llama a la operación product().

BigInteger operator/(BigInteger&): Esta sobrecarga permite dividir dos bigInteger sobre cargando el operador / y permitiéndolo guardar en otro bigInteger, esta operación tendría la misma que la división ya que esta adentro llama a la operación quotient().

BigInteger operator%(BigInteger&): Esta sobrecarga permite obtener el módulo de dos bigInteger sobre cargando el operador % y permitiéndolo guardar en otro bigInteger, esta operación tendría la misma que la modulo ya que esta adentro llama a la operación remainder ().

bool operator==(const BigInteger&): Esta sobrecarga permite comparar dos bigInteger sobrecargando el operador = = lo cual retornaría true si el primer bigInteger es exactamente igual al otro bigInteger, esta operación tendría mejor y peor caso, ya que en el mejor caso simplemente verificaría ciertas condiciones que permiten definir si es menor o no, y en el peor caso tendría que recorrerse el bigInteger lo cual tendría una complejidad O(n) siendo n el tamaño del bigInteger.

bool operator<(const BigInteger&): Esta sobrecarga permite comparar dos bigInteger sobre cargando el operados < lo cual retornaría true si el primer bigInteger es menor al otro bigInteger, esta operación tendría mejor y peor caso, siendo el mejor donde verificaría con ciertas condiciones y el peor caso donde tendría que recorrer el bigInteger para verificar si es menor o no teniendo complejidad O(n) siendo n el tamaño del bigInteger.

bool operator<=(const BigInteger&): Esta sobrecarga permite compara dos bigInteger sobre cargando el operador <= lo cual retornaría true si el bigInteger es menor o igual al otro bigInteger, esta operación llamaría a la operación < y a la operación = = lo cual tendría los mejores y peores casos de cada uno, donde el peor caso seria O(n) donde n sería el tamaño del bigInteger.

OPERACIONES ESTATICAS

BigInteger sumarListaValores(vector<BigInteger>& l): Esta operación toma una lista de bigInteger y retorna otro bigInteger el cual es la suma de todos los bigInteger, esta función tiene una complejidad de O(n*m) donde m sería los elementos de la lista y n seria las iteraciones de sumar los valores de la lista.

BigInteger MultiplicarListaValores(vector<BigInteger>& l): Esta operación toma una lista de bigInteger y retorna otro bigInteger el cual es la multiplicación de cada bigInteger en la lista, esta función tendría la complejidad de O(n^2m) donde n^2 seria las iteraciones de multiplicar cada bigInteger de la lista y m seria la cantidad de elementos de la lista.

AUXILIARES

Bool menorSinSigno(vector<int>& num , vector<int>& num1): Esta operación permite saber si un vector es menor a otro sin tomar en cuenta su signo, retornando true si es menor al otro vector, su complejidad seria O(n) siendo n el tamaño de un vector ya que la función solo funciona para vectores del mismo tamaño.

void elimCeros(vector<int>& vec): Esta operación elimina 0 que se encuentren en la parte final de un vector, su complejidad seria O(n) siendo n el numero de elementos del vector siendo el peor caso tener que recorrer el vector por completo.