```
/*
Función BigInteger
Entrada: Una string de numeros
Salida: ninguna
Descripción: se convierten los elementos de la string en ints y se agregan al vector digits que representan el BigInteger

*/

BigInteger::BigInteger(string digits){

    if (!digits.empty() && digits[0] == '-') {
        isNegative = true;
        digits = digits.substr(1);
    }

    for(char c : digits){
        if (isdigit(c)) {
            int digit = c - '0';
            this->digits.push_back(digit);
        }
    }

}
```

La complejidad de esta función es O(n^2) ya que se entra a un ciclo for que depende del tamaño del string “digits”.

```
/*
Función BigInteger
Entrada: Un objeto BigInteger
Salida: ninguna
Descripción: copia otro BigInteger

*/

BigInteger::BigInteger(const BigInteger& other) {
    digits = other.digits;
    isNegative = other.isNegative;
}
```

La complejidad de esta operación es O(1) ya que siempre se recorrerá la misma cantidad de líneas.

```
Función equalOP
Entrada: Un objeto BigInteger
Salida: Un boolean representando la igualdad entre 2 BigInteger
Descripción: Se comparan cosas como negativos y numero de digitos primeros y si estos no dan la respuesta deseada se pasa a un forloop
donde se comparan los digitos desde el primero (mas grande) al ultimo (mas pequeño) de ambos objetos, si no retorna falso hasta entonces
retorna true

*/

bool BigInteger::equalOP(const BigInteger& other) {
    if (isNegative != other.isNegative) {
        return false;
    }

    if (digits.size() != other.digits.size()) {
        return false;
    }

    for (int i = digits.size() - 1; i >= 0; i--) {
        if (digits[i] != other.digits[i]) {
            return false;
        }
    }

    return true;
}
```

La complejidad de esta operación es O(1) en el mejor caso donde cualquiera de los BigInteger sea de signo diferente o de diferente tamaño, en el peor donde ambos sean iguales y tenga que recorrer todo el loop la complejidad es O(n).

```
/*
Función compareOP
Entrada: Un objeto BigInteger ...
Salida: Un boleano que representa "menor que"
Descripción: primero se chequea si los negativos son distintos, si ninguno o
ambos son negativos pasa a un forloop recorriendo todos
los digitos desde el primero (mas grande) al ultimo (mas pequeño) y se comparan
los de ambos, si los numeros terminan siendo iguales
regresa verdadero, pero ya que esta operacion se reutiliza bastante no se puede
arreglar este problema, asi que despues se usa el
equalOP para confirmar que no sean iguales

*/

bool BigInteger::compareOP(const BigInteger& other) {
    // true = less than, false = more than

    if (isNegative && !other.isNegative) {
        return true;
    } else if (!isNegative && other.isNegative) {
        return false;
    }

    if (isNegative && other.isNegative) {
        // Both numbers are negative
        cout<<"works"<<endl;
        if (digits.size() == other.digits.size()) {
            for (int i = 0; i<digits.size(); i++) {
                if (digits[i] > other.digits[i]) {
                    return false; // Negative number with larger digits is
smaller
                } else if (digits[i] < other.digits[i]) {
                    return true; // Negative number with smaller digits is larger
                }
            }
        } else {
            cout<<"works"<<endl;
            return digits.size() > other.digits.size(); // Negative number with
fewer digits is larger
        }
```

```
    } else {
        // Both numbers are positive

        if (digits.size() == other.digits.size()) {
            for (int i = 0; i<digits.size(); i++) {
                if (digits[i] > other.digits[i]) {
                    return false; // Positive number with larger digits is
smaller
                } else if (digits[i] < other.digits[i]) {
                    return true; // Positive number with smaller digits is larger
                }
            }
        } else {
            //return digits.size() > other.digits.size(); // Positive number with
more digits is larger

            if(digits.size() > other.digits.size()){
                return false;
            }else if(digits.size() < other.digits.size()){
                return true;
            }
        }
    }
    return true; // Numbers are equal
}
```

La complejidad de esta operación es O(1) en el mejor caso donde cualquiera de los BigInteger sea de signo diferente o de diferente tamaño, en el peor donde ambos sean iguales y tenga que recorrer todo un for la complejidad es O(n).

```
/*
Función operator==
Entrada: Un objeto BigInteger
Salida: Un boolean que representa la igualdad de 2 BigInteger
Descripción: retorna la operacion equalOP

*/

bool BigInteger::operator==(const BigInteger& other){
    return equalOP(other);
}
```

La complejidad de esta función es igual que la de “equalOP”: O(1) mejor caso, O(n) peor caso.

```
/*
Función operator!=
Entrada: Un objeto BigInteger
Salida: Un boolean que representa la desigualdad de 2 BigInteger
Descripción: retorna lo contrario de equalOP

*/

bool BigInteger::operator!=(const BigInteger& other){
    return !equalOP(other);
}
```

La complejidad de esta función es igual que la de “equalOP”: O(1) mejor caso, O(n) peor caso.

```
/*
Función operator<
Entrada: Un objeto BigInteger
Salida: Un boolean que represeta "menor que"
Descripción: devuelve comparOP mientras no se cumpla equalOP

*/

bool BigInteger::operator<(const BigInteger& other){
    bool result = compareOP(other);
    if(equalOP(other)){
        result=false;
    }
    return result;
}
```

La complejidad de esta función es igual que la de “compareOP” o “equalOP”: O(1) mejor caso, O(n) peor caso

```
/*
Función operator<=
Entrada: Un objeto BigInteger
Salida: Un boolean que represeta "menor que"
Descripción: devuelve true cuando comparOP o equalOP se cumplen

*/

bool BigInteger::operator<=(const BigInteger& other){
    bool result=false;
    if(compareOP(other) || equalOP(other)){
        result= true;
    }
    return result;
}
```

La complejidad de esta función es igual que la de “compareOP” o “equalOP”: O(1) mejor caso, O(n) peor caso.

```
/*
Función operator>
Entrada: Un objeto BigInteger
Salida: Un boolean que represeta "menor que"
Descripción: devuelve lo contrario a comparOP mientras no se cumpla equalOP

*/

bool BigInteger::operator>(const BigInteger& other){
    bool result=false;
    if(!(compareOP(other))){
        result= true;
    }

    if(equalOP(other)){
        result=false;
    }
    return result;
}
```

La complejidad de esta función es igual que la de “compareOP” o “equalOP”: O(1) mejor caso, O(n) peor caso.

```
/*
Función operator<=
Entrada: Un objeto BigInteger
Salida: Un boolean que represeta "menor que"
Descripción: devuelve true cuando lo contrario a comparOP o equalOP se cumplen

*/

bool BigInteger::operator>=(const BigInteger& other){
    bool result=false;
    if(!(compareOP(other)) || equalOP(other)){
        result= true;
    }
    return result;
}
```

La complejidad de esta función es igual que la de “compareOP” o “equalOP”: O(1) mejor caso, O(n) peor caso.

```
/*
Función sumOP
Entrada: Un objeto BigInteger
```

```
Salida: Un objeto BigInteger que representa la suma de 2 BigInteger
Descripción: recorre los elementos de BigInteger desde el ultimo numero hasta el
primero y los suma, si un BigInteger es mas grande
que el otro, considera los valores nulos como 0, cuando la respuesta es mayor que
10, le suma 1 al siguiente digito
*/
string BigInteger::SumOp(const BigInteger& other) {
    string result = "";
    int share = 0;

    for (int i = 0; i < this->digits.size() || i < other.digits.size(); i++) {
        int digit1;
        int digit2;

        if (i < this->digits.size()) {
            digit1 = this->digits[(this->digits.size()) - (i + 1)];
        } else {
            digit1 = 0;
        }

        if (i < other.digits.size()) {
            digit2 = other.digits[(other.digits.size()) - (i + 1)];
        } else {
            digit2 = 0;
        }

        int sum = digit1 + digit2 + share;

        if (sum > 9) {
            sum -= 10;
            share = 1;
        } else {
            share = 0;
        }

        result = to_string(sum) + result;
    }

    if (share > 0) {
        result = to_string(share) + result;
    }

    return result;
}
```

La complejidad de esta función es O(max(n, m)) ya que se entra en un loop for el cual siempre se recorre todo, el ciclo se repite despendiendo del BigInteger con mas dígitos.

```
/*
Función: restOP
Entrada: Un objeto BigInteger
Salida: Un objeto BigInteger que representa la diferencia entere 2 BigInteger
Descripción: Sigue una logica similar a la suma, pero con la resta en vez de
sumarle 1 al siguiente digito, cuando se resta un numero
menor con un numero mayor, se le suma 10 al menor y se le resta 1 al siguiente
digito, si los BigInteger tienen el mismo problema, se
cambia el orden de la resta y se cambia el resultado a negativo.

*/

string BigInteger::RestOp(const BigInteger& other){
    string result="";
    int borrow=0;

    for(int i=0; i<this->digits.size() || i<other.digits.size(); i++){

        int digit1;
        int digit2;


        if (i < this->digits.size()) {

            digit1 = this->digits[(this->digits.size())-(i+1)];
        }else{
            digit1 = 0;
        }

        if (i < other.digits.size()) {

            digit2 = other.digits[(other.digits.size())-(i+1)];
        }else{
            digit2 = 0;
        }

        int digit3;

        if (digits.size()>other.digits.size()){

            if (borrow) {
                digit1--;
```

```
                borrow = 0;
            }

            if (digit1 < digit2) {
                digit1 += 10;
                borrow = 1;
            }

            digit3 = digit1 - digit2;
        }else{

            if (borrow) {
                digit2--;
                borrow = 0;
            }

            if (digit2 < digit1) {
                digit2 += 10;
                borrow = 1;
            }

            digit3 = digit2 - digit1;
        }

        result= to_string(digit3) + result;
    }

    while (result.size() > 1 && result[0] == '0') {
        result.erase(0, 1);
    }

    if (compareOP(other)) {
        result = "-" + result;
    }
    return result;
}
```

La complejidad de esta función es O(max(n, m)) ya que se entra en un loop for el cual siempre se recorre todo, el ciclo se repite despendiendo del BigInteger con mas dígitos.

```
/*
Función operator+
Entrada: Un objeto BigInteger
Salida: Un objeto BigInteger que representa la suma de 2 BigIntegers
```

```
Descripción: Segun si los operadores son negativos o positivos llama sumOP o
restOP segun las leyes de signos negativos.

*/

BigInteger BigInteger::operator+(const BigInteger& other) {
    BigInteger result("");

    // Case 1: Both operands are positive
    if (!isNegative && !other.isNegative) {
        result = BigInteger(SumOp(other));
    }
    // Case 2: Both operands are negative
    else if (isNegative && other.isNegative) {
        result = BigInteger(SumOp(other));
        result.isNegative = true;
    }
    // Case 3: One operand is positive, the other is negative
    else {
        // If this operand is negative, swap the operands
        if (isNegative) {
            BigInteger copy = *this;
            result = copy.RestOp(other);
            if(other.digits.size()<copy.digits.size()){
                result.isNegative=true;
            }
        }
        else {
            BigInteger copy = other;
            copy.isNegative=false;
            result = BigInteger(RestOp(copy));
        }
    }

    return result;
}
```

La complejidad de esta función es la misma que “restOP” o “sumOP”: O(max(n, m)) en todos los casos

```
/*
Función operator-
Entrada: Un objeto BigInteger
Salida: Un objeto BigInteger que representa la diferencia de 2 BigIntegers
Descripción: Segun si los operadores son negativos o positivos llama sumOP o
restOP segun las leyes de signos negativos.
```

```
*/

BigInteger BigInteger::operator-(const BigInteger& other){

    BigInteger result("");

    // Case 1: Both operands are positive
    if (!isNegative && !other.isNegative) {
        result = BigInteger(RestOp(other));
    }
    // Case 2: Both operands are negative
    else if (isNegative && other.isNegative) {
        BigInteger copy = other;
        copy.isNegative=false;
        result = BigInteger(RestOp(copy));
        if(digits.size()>other.digits.size()){
            result.isNegative = true;
        }else{
            result.isNegative=false;
        }
    }
    // Case 3: One operand is positive, the other is negative
    else {
        // If this operand is negative, swap the operands
        if (isNegative) {
            BigInteger copy = *this; //non const copy
            result = copy.SumOp(other);
            result.isNegative = true;
        }
        else {
            result = BigInteger(SumOp(other));
        }
    }

    return result;
}
```

La complejidad de esta función es la misma que “restOP” o “sumOP”: O(max(n, m)) en todos los casos

```
/*
Función operator*
Entrada: Un objeto BigInteger
Salida: Un objeto BigInteger que representa la multiplicacion de 2 BigIntegers
Descripción: suma el BigInteger con sigo mismo el numero de veces de la entrada y
aplica signos negativos segun leyes
```

```
*/

BigInteger BigInteger::operator*(const BigInteger& other){
    BigInteger result("");
    BigInteger copy = *this;
    BigInteger copy2 = other;
    copy2.isNegative=false;
    copy.isNegative=false;

    for(BigInteger i("0"); i.compareOP(copy2) && !i.equalOP(copy2); i=
i+BigInteger("1")){
        result= result.SumOp(copy);
    }

    if((isNegative && !other.isNegative) || (!isNegative && other.isNegative)){
        result.isNegative=true;
    }

    return result;
}
```

La complejidad de esta operación es O(n*max(n, m)) ya que se entra a un for loop que llama “SumOP” en cada ciclo

```
/*
Función operator/
Entrada: Un objeto BigInteger
Salida: Un objeto BigInteger que representa la divicion de 2 BigIntegers
Descripción: resta un BigInteger con otro repetidamente mientras que el primer
numero sea mayor a 0 y cuenta el numero de restas totales

*/

BigInteger BigInteger::operator/(const BigInteger& other){
    BigInteger result("0");
    BigInteger copy = *this;
    BigInteger copy2 = other;
    copy2.isNegative=false;
    copy.isNegative=false;

    while (copy>BigInteger("0")){
        copy= copy-copy2;
        if(copy>=BigInteger("0")){
            result = result + BigInteger("1");
        }
    }
```

```
    if((isNegative && !other.isNegative) || (!isNegative && other.isNegative)){
        result.isNegative=true;
    }

    return result;
}
```

La complejidad de esta operación es O(n*max(n, m)) ya que se entra a un for loop que llama “RestOP” en cada ciclo

```
/*
Función operator%
Entrada: Un objeto BigInteger
Salida: Un objeto BigInteger que representa el reciduo de la divicion entre 2
BigIntegers
Descripción: hace la division pero devuelve el numero final antes de que la resta
diera menos de 0

*/

BigInteger BigInteger::operator%(const BigInteger& other){
    BigInteger result("0");
    BigInteger copy = *this;
    BigInteger copy2 = other;
    copy2.isNegative=false;
    copy.isNegative=false;

    if(copy==copy2){
        return BigInteger("0");
    }

    while (copy>BigInteger("0")){
        if(copy-copy2<=BigInteger("0")){
            result = copy;
        }
        copy= copy-copy2;
    }


    if((isNegative && !other.isNegative) || (!isNegative && other.isNegative)){
        result.isNegative=true;
    }
```

```
    return result;
}
```

La complejidad es la misma que “operator/” ya que es una versión ligeramente modificada del anterior: O(n*max(n, m))

```
/*
Función operator^
Entrada: Un Int
Salida: Un objeto BigInteger que representa un BigInteger elevado por un Int
Descripción: parecido a la multiplicacion pero en vez de sumarse por si mismo se
mutiplica por si mismo la cantidad de veces que el
exponente indique, y se aplican signos segun leyes

*/

BigInteger BigInteger::operator^(const int& exponent){
    BigInteger result("");
    BigInteger copy = *this;
    int copy2 = exponent;
    copy.isNegative=false;

    for(int i=0; i<copy2; i++){
        result= result*result;
    }

    if(isNegative && copy2%2!=0 && copy%BigInteger("0")!=BigInteger("0")){
        result.isNegative=true;
    }

    return result;
}
```

La complejidad de esta función es O($n^2 * \max(n, m)$) ya que se repite la operación * en el for loop

```
/*
Función: sumarListaValores
Entrada: Un un vector de objetos BigInteger
Salida: Un objeto BigInteger que representa la suma de todos los valores de un
vector
Descripción: Se usa el operador suma en todos los valores

*/

BigInteger BigInteger::sumarListaValores(const vector<BigInteger>& values) {
    BigInteger sum("0");

    for ( BigInteger value : values) {
```

```
        sum = sum + value;
    }

    return sum;
}
```

La complejidad de esta función es de O(n*max(n, m)) ya que se llama una suma repetidamente como en la operación *.

```
/*
Función: sumarListaValores
Entrada: Una lista de objetos BigInteger
Salida: Un objeto BigInteger que representa la suma de todos los valores de una lista
Descripción: Se usa el operador suma en todos los valores

*/

BigInteger BigInteger::sumarListaValores(const list<BigInteger>& values) {
    BigInteger sum("0");

    for (BigInteger value : values) {
        sum = sum + value;
    }

    return sum;
}
```

Exactamente lo mismo que la anterior pero con listas O(n*max(n, m))

```
/*
Función: multiplicarListaValores
Entrada: Un un vector de objetos BigInteger
Salida: Un objeto BigInteger que representa el producto de todos los valores de un vector
Descripción: Se usa el operador * en todos los valores

*/

BigInteger BigInteger::multiplicarListaValores(const std::vector<BigInteger>& values) {
    BigInteger result("1");

    for (const BigInteger& value : values) {
        result = result * value;
    }

    return result;
```

```
}

/*
Función: multiplicarListaValores
Entrada: Una lista de objetos BigInteger
Salida: Un objeto BigInteger que representa el producto de todos los valores de
una lista
Descripción: Se usa el operador * en todos los valores

*/

BigInteger BigInteger::multiplicarListaValores(const std::list<BigInteger>&
values) {
    BigInteger result("1");

    for (const BigInteger& value : values) {
        result = result * value;
    }

    return result;
}
```

Ambas funciones tienen una complejidad de O(n^2*max(n,M)) al tener una operación * dentro de un ciclo for.

```
/*
Función: toString
Entrada: nada
Salida: Una string representando el valor del objeto
Descripción: recorre los elementos de digits, los convierte a strings, los agrega
al resultado y los devuelve.

*/

string BigInteger::toString(){
    string toString;

    for(int i : digits){
        toString+= to_string(i);
    }
    if(isNegative){
        toString="-" + toString;
    }
    return toString;
}
```

Esta función tiene complejidad O(n) al ser solo recorrer un vector.

```
/*
Función size
Entrada: Nada
Salida: Un objeto BigInteger que representa el numero de digitos
Descripción: devuelve el tamaño del vector digits

*/

BigInteger BigInteger::size(){
    return BigInteger(to_string(digits.size()));
}
```

Esta función tiene complejidad O(n) por que llama al constructor de BigInteger el cual tiene complejidad O(n).

```
string compareBigFloat(BigInteger& intP1,BigInteger& floatP1, BigInteger& intP2,
BigInteger& floatP2){

    string f1= floatP1.toString();
    string f2= floatP2.toString();

    if(intP1<intP2){
        return "Smaller";
    }
    if(intP1>intP2){
        return "Bigger";
    }else{
        BigInteger smaller("");
        if(floatP1.size()<floatP2.size()){
            smaller=floatP1.size();
        }else{
            smaller=floatP2.size();
        }

        for(int i=0; i< stoi(smaller.toString()); i++){
            if(f1[i]<f2[i]){
                return "Smaller";
            }
            if(f1[i]>f2[i]){
                return "Bigger";
            }
        }
    }

    return "equal";
```

```
}
```

Esta función tiene complejidad O(1) en el mejor caso donde intP1 e intP2 no son iguales, en el peor caso donde estos son iguales la complejidad es O(n)

```
int main(){

    string input;

    string input2;
    string input3;
    string intPart;
    string floatPart;

    bool point;

    int cases=0;

    while (getline(cin, input) && !cin.eof()) {
        input2="";
        input3="";

        point=false;
        for(char c: input){

            if(c!=' ' && !point){
                input2 += c;
            }else if(c!=' ' && point){
                input3 += c;
            }else if(c==' '){
                point=true;
            }

        }
        point=false;
        for(char c: input2){
            if(c!=' '){
                if(c!='.' && !point){
                    intPart += c;
                }else if(c!='.' && point){
                    floatPart += c;
                }else{
                    point=true;
                }
            }
```

```
        }

        BigInteger numInt1(intPart);
        BigInteger numFloat1(floatPart);


        intPart="";
        floatPart="";

        point=false;
        for(char c: input3){
            if(c!=' '){
                if(c!='.' && !point){
                    intPart += c;
                }else if(c!='.' && point){
                    floatPart += c;
                }else{
                    point=true;
                }
            }
        }

        BigInteger numInt2(intPart);


        BigInteger numFloat2(floatPart);

        intPart="";
        floatPart="";

        cout<<"Case "<<cases+1<< ": " <<compareBigFloat(numInt1, numFloat1,
numInt2, numFloat2)<<endl;

        cases++;

    }

    return 0;
}
```

Este código tiene complejidad O(1) en el mejor caso donde el archivo de entrada esta vacio por lo que no entra al while, en el peor caso donde entra un archivo con entradas la complejidad es O(n^2) al tener for loops dentro del while.