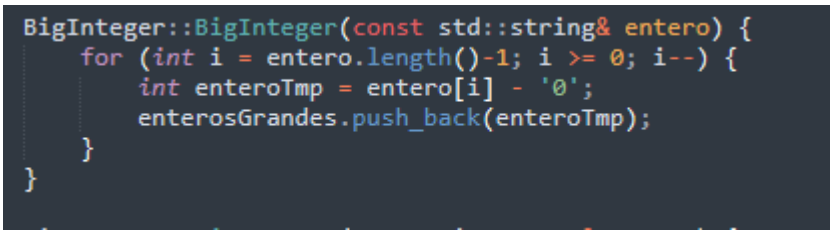
REPORTE COMPLEJIDADES

SANTIAGO ARANGO HENAO

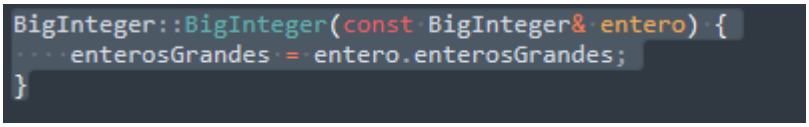
Primer Constructor:



```
BigInteger::BigInteger(const std::string& entero) {
    for (int i = entero.length()-1; i >= 0; i--) {
        int enteroTmp = entero[i] - '0';
        enterosGrandes.push_back(enteroTmp);
    }
}
```

La complejidad de esta operación es O(n), n es la longitud de la cadena que representa el BigInteger. En primer lugar tenemos un for que itera desde n-1 por el final de la cadena hasta 0 que representa el inicio de la cadena. En cada iteración se realiza la operación para convertir el carácter a entero, lo cual es constante y posteriormente se realiza un push_back el cuál tiene complejidad O(1) amortizado, es amortizado por que hay ocasiones donde no hay espacio al final del vector y toca asignar un nuevo bloque de memoria con capacidad mayor, pero se mantiene constante porque entre la reubicación de memoria y el numero total de inserciones, es mas bajo las inserciones de reubicación que el número total de inserciones.

Segundo Constructor:



```
BigInteger::BigInteger(const BigInteger& entero) {
    enterosGrandes = entero.enterosGrandes;
}
```

Como es una copia del primer constructor, la complejidad sigue siendo la misma, lo que cambia es que el O(n), la n vendría siendo el tamaño del vector enterosGrandes del objeto entero.

Operación Add:

```
void BigInteger::add(BigInteger& entero) {
    int lleva = 0;
    int maxEnterosGrandes = std::max(enterosGrandes.size(), entero.enterosGrandes.size());

    for (int i = 0; i < maxEnterosGrandes || lleva > 0; i++) {
        int sum = lleva;

        if (i < enterosGrandes.size())
            sum += enterosGrandes[i];

        if (i < entero.enterosGrandes.size())
            sum += entero.enterosGrandes[i];

        lleva = sum / 10;
        sum %= 10;

        if (i < enterosGrandes.size())
            enterosGrandes[i] = sum;
        else
            enterosGrandes.push_back(sum);
    }
}
```

La complejidad de esta operación es O(n) donde n es el valor máximo entre el tamaño del vector enteroGrandes y el tamaño del vector entero.enterosGrandes. Esto se debe a que el for itera hasta el máximo entre los tamaños de los vectores para asegurar que se recorran todos los dígitos. En las siguientes líneas solo se realizan operaciones aritméticas y asignaciones que tienen complejidad O(1).

Operación producto:

```
void BigInteger::product(BigInteger& entero) {
    BigInteger resultado;
    int maxDigitos = enterosGrandes.size() + entero.enterosGrandes.size();

    for (int i = 0; i < maxDigitos; i++) {
        resultado.enterosGrandes.push_back(0);
    }

    for (int i = 0; i < enterosGrandes.size(); i++) {
        int lleva = 0;

        for (int j = 0; j < entero.enterosGrandes.size() || lleva > 0; j++) {
            int prod = lleva;

            if (j < entero.enterosGrandes.size())
                prod += enterosGrandes[i] * entero.enterosGrandes[j];

            lleva = prod / 10;
            prod %= 10;

            resultado.enterosGrandes[i + j] += prod;
            if (resultado.enterosGrandes[i + j] >= 10) {
                resultado.enterosGrandes[i + j] -= 10;
                lleva++;
            }
        }
    }

    enterosGrandes = resultado.enterosGrandes;

}
```

La complejidad de esta operación es O(n*m) donde n es el tamaño del vector enterosGrandes y m el tamaño del vector entero. En el primer for se realiza un for que itera hasta la suma total del tamaño de los vectores (esto se hace para saber con anticipación cuantos números tendrá la multiplicación) y por lo tanto tiene complejidad O(n+m). En los siguientes for anidados, el primer for recorre el vector enterosGrandes y el

segundo el vector entero para hacer la respectiva multiplicación digito a digito, en el interior de los for solo se realizan operaciones aritméticas y de asignación que tienen un costo de O(1) lo cuál no influye mucho en la complejidad del programa. Siendo así la complejidad del algoritmo de O(n*m).

Operación substract:

```
void BigInteger::substract(BigInteger& entero) {
    int prestado = 0;
    int maxEnterosGrandes = std::max(enterosGrandes.size(), entero.enterosGrandes.size());

    for (int i = 0; i < maxEnterosGrandes || prestado > 0; i++) {
        int diferencia = prestado;

        if (i < enterosGrandes.size())
            diferencia += enterosGrandes[i];

        if (i < entero.enterosGrandes.size())
            diferencia -= entero.enterosGrandes[i];

        if (diferencia < 0) {
            diferencia += 10;
            prestado = -1;
        } else {
            prestado = 0;
        }

        if (i < enterosGrandes.size())
            enterosGrandes[i] = diferencia;
        else
            enterosGrandes.push_back(diferencia);
    }

    //Eliminar ceros restantes del frente del vector
     while (enterosGrandes.size() > 1 && enterosGrandes.back() == 0) {
        enterosGrandes.pop_back();
    }
}
```

La complejidad de este algoritmo, en primer lugar, depende del tamaño de los BigInteger, es decir, del tamaño de ambos vectores. De forma similar a la operación add, se realiza un ciclo for que itera hasta la cantidad máxima de uno de los dos vectores, se realizan algunas operaciones, las cuales son aritméticas y tienen un costo constante. La diferencia de esta operación con la suma es que al realizarse la resta quedan ceros en las primeras posiciones del vector actual y para eliminarlas se usa un ciclo while que itera hasta encontrar en la espalda del vector un numero distinto de 0, en caso de que el BigInteger sea solo ceros, itera hasta que el tamaño del vector sea 1, teniendo esto en cuenta, en el peor de los casos se puede tener un BigInteger de N elementos, entonces tocaría hacer ambos ciclos que dependen del tamaño del vector, así que la complejidad es de O(n).