PONTIFICIA UNIVERSIDAD JAVERIANA CALI

28 de Mayo del 2023

Estructura De Datos

Profesores: Carlos Alberto Ramírez – Luis Gonzalo Noreña

Implementación Big Integer en C++

Un Big Integer:

Un Big integer es una implementación para tipos de datos de números enteros que no se pueden representar por sí solo en algunos lenguajes, por ejemplo en C++ o C# el entero más grande que se podría representar es el "Int64" que es el entero de 64bits *(9223372036854775807),* entonces si deseamos utilizar o manipular números más grandes, no podríamos realizar estas operaciones por sí solas, ya que no existen suficientes bits para representar aquel numero, por lo que probablemente termine representando números negativos que no tienen para nada que ver con el resultado, por lo que para solucionar este problema podemos utilizar una estructura llamada *BigInteger,* en la que no realizamos las operaciones de manera binaria, sino que los valores son representados como *strings* ya que de esta manera no importa que tan grande sea el número, siempre se realizará el mismo procedimiento que realiza el ser humano para operar estos números.

¿Cuál es la importancia de esta estructura?

Puede que para realizar las actividades de nuestra vida diaria, no le encontremos ningún sentido de utilizar un BigInteger, pero existen muchas ramas en la actualidad en las que es sumamente importante la aplicación de números muy grandes, como lo son principios matemáticos, principios físicos, para movimientos mecánicos, o inclusive, para la astronomía dinámica en la que se realizan cálculos para detallar el movimiento de los planetas, rotaciones, o anomalías en el comportamiento, por lo que se debe ser demasiado preciso y se utilizan números muy extensos, anteriormente, los humanos debían realizar estos problemas matemáticos porque las computadoras no podían realizar estos cálculos.

Implementación De BigInteger

La elección entre vectores o listas enlazadas para implementar un BigInteger depende de varios factores y consideraciones. Ambas estructuras de datos tienen ventajas y desventajas que deben tenerse en cuenta al realizar la implementación.

Acceso a los dígitos: Los vectores permiten un acceso directo y eficiente a los elementos mediante índices, lo que los hace más adecuados cuando se requiere acceder a los dígitos de forma secuencial o realizar operaciones aritméticas. En cambio, en las listas enlazadas, el acceso a los elementos requiere recorrer la estructura desde el principio hasta el índice deseado, lo que puede ser más lento en comparación.

Inserción y eliminación de dígitos: Las listas enlazadas son más eficientes en términos de inserción y eliminación de elementos, ya que no requieren mover o redimensionar una estructura subyacente, como en el caso de los vectores. Esto puede ser beneficioso cuando se realizan operaciones como la suma o multiplicación de BigIntegers, donde es necesario agregar o quitar dígitos.

Consumo de memoria: Los vectores generalmente tienen un consumo de memoria más eficiente en comparación con las listas enlazadas. Esto se debe a que los vectores almacenan los elementos de manera contigua en la memoria, mientras que las listas enlazadas requieren almacenar referencias adicionales para enlazar los nodos. Si se necesita trabajar con BigIntegers extremadamente grandes, el consumo de memoria puede ser un factor importante a considerar.

En general, si se prioriza el acceso eficiente a los dígitos y se espera realizar muchas operaciones aritméticas, como suma, resta, multiplicación, división, etc., los vectores (arrays) suelen ser una opción más adecuada. Por otro lado, si se espera realizar muchas operaciones de inserción y eliminación de dígitos, o si el tamaño del BigInteger puede variar significativamente durante su manipulación, las listas enlazadas (linked lists) pueden ser una mejor opción.

- Constructores de un BigInteger

```
-           BigInteger(const string& str);
```

Esta función, necesita de un string, que es lo que tomaremos como el numero, teniendo este string procederemos a recorrerlo y asignarle cada digito en una posición de un vector, por lo que la complejidad de esta implementación seria lineal, en la que n seria la longitud del string a convertir.

$$Complejidad\ BigO: \qquad O(n) \rightarrow \ n\ Tamaño\ del\ String\ o\ del\ Vector$$

```
BigInteger::BigInteger(const BigInteger& n1)
```

Esta función necesita de otro BigInteger, que internamente tendría la implementación de un vector, y lo que hace es copiar el valor de cada una de las posiciones del vector para ingresarlos en el nuevo vector que hará parte del nuevo BigInteger, por lo que la complejidad de esta implementación es parecida a la anterior porque debe recorrer el vector, por lo que seria lineal, dependiendo de la longitud del vector asignado como parámetro.

$$Complejidad\ BigO: \qquad O(n) \rightarrow \ n\ Tamaño\ del\ String\ o\ del\ Vector$$

Operaciones sobre el Objeto Actual

- Operación ADD:

En las operaciones con enteros dependiendo del signo del los numeros con los que se desea operar estos pueden restarse o sumarse, por ende es necesario definir dos funciones auxiliares que cumplan esta funciones segun sea requerido.

Ambas operaciones reciben en Biginteger que se desea sumar o restar respectivamente al valor actual

- Operación Product:

```
void productVector(vector<int>& n1, vector<int>& n2)
```

Función productVector

Entrada: Dos vectores de enteros por referencia, num1 y num2.

Salida: Ninguna (void).

Descripción: Esta función realiza la multiplicación de dos vectores que representan números enteros grandes. El resultado se almacena en el vector num1.

El algoritmo utiliza el método de la multiplicación de columnas para realizar la multiplicación. Itera sobre cada dígito de num1 y num2, realiza la multiplicación correspondiente y actualiza los dígitos del vector ans. Luego, se ajustan los dígitos y se eliminan los ceros no significativos del resultado.

Es importante destacar que los vectores num1 y num2 se pasan por referencia, lo que significa que se modifican directamente dentro de la función.

Complejidad de la implementación:

$$Complejidad\ O(n * m) \rightarrow n\ es\ el\ tamaño\ del\ vector\ 1\ y\ m\ del\ vector\ 2$$

- Operación Substract:

El bucle while se ejecuta hasta que se cumpla la condición *currentIndex < num1.size()* o *resizeFlag* sea false. En el peor de los casos, el bucle se *ejecutará max(num1.size(), num2.size())* veces, es decir, la cantidad de veces linealmente a la longitud del vector que tenga mayor tamaño.

Dentro del bucle, las operaciones son principalmente operaciones aritméticas simples y asignaciones, que tienen una complejidad de tiempo constante, es decir, O(1).

Al final del bucle, hay una operación de redimensionamiento (num1.resize(firstNonZeroIndex)), que en el peor de los casos requiere copiar todos los elementos del vector num1 a una nueva ubicación de memoria. Esta operación tiene una complejidad de tiempo lineal en relación con el tamaño del vector num1, es decir, O(num1.size()).

$$Complejidad\ O(n) \rightarrow Donde\ N\ es\ la\ longitud\ del\ vector\ mas\ grande$$

- Operación POW:

En cada iteración, se realiza una multiplicación utilizando la función product, que tiene una complejidad dependiente del tamaño de los números involucrados. La complejidad de la multiplicación de dos números de tamaño n es O(n^2), ya que se deben realizar multiplicaciones de todos los dígitos de uno de los números por todos los dígitos del otro número. Sin embargo, en cada iteración, el tamaño de los números involucrados se reduce a la mitad debido a la división sucesiva por 2.

$$\underline{Complejidad\ es\ O(n^2) + O(Log\ n)}$$

- Operación toString:

La función realiza un bucle for que itera desde size() - 1 hasta 0. En cada iteración, se concatena el dígito convertido a un carácter y se agrega al final de la cadena ans. Esto implica una operación de concatenación de cadenas en cada iteración.

Dado que el número de iteraciones del bucle for depende directamente del tamaño del número representado por el objeto BigInteger, la complejidad de la función es proporcional a n, es decir, O(n).

$$Complejidad\ O(n) \rightarrow Donde\ N\ es\ la\ longitud\ del\ Vector$$

- Operación getDigit:

La complejidad de la función *BigInteger::getDigit(int pos)* es O(1), es decir, tiempo constante.

Dado que la función simplemente devuelve el valor almacenado en la posición pos del arreglo digito, no realiza ningún cálculo adicional ni iteración sobre otros elementos. Por lo tanto, la complejidad de esta función no depende del tamaño del arreglo digito y se considera constante

Operadores Binarios:

- Operadores Comparadores:

En el caso de los operadores aritmetricos lo que se realizo fue simplemente asignarles la

- Operadores Aritmetricos:

En el caso de los operadores binarios, simplemente lo que hice fue asignarles las operaciones para cada uno de ellos, para la suma asignamos la función add, para la resta la función substract, para la multiplicación producto y para la división quotient y remainder, se pidió que el objeto no sea modificado sino que sea creado como una copia.

Funciones Estáticas:

- Sumar Valores de una lista

```
-   BigInteger BigInteger::sumarListaValores(list<BigInteger>& l )
```

Para la realización de esta implementación utilice un iterador que apuntara al inicio de la lista, y una variable BigInteger que llevara la sumatoria a medida que esta recorra la lista, ans se inicializa con el valor que posee la primera posición del iterador, y el iterador iniciaría a recorrer desde la segunda posición para evitar que se sume 2 veces el primer valor. La complejidad de esta implementación es lineal, O(n), donde n es el número de elementos en la lista l. Esto se debe a que recorremos la lista una vez utilizando el iterador it en un ciclo while y realizamos una operación de suma (ans.add(*it)) en cada iteración.

$$Complejidad\ O(n) \rightarrow Donde\ n\ es\ el\ numero\ de\ elementos\ de\ la\ Lista\ L$$

- Multiplicar Valores de una Lista:

Para la realización de esta implementación fue demasiado parecida a la implementación anterior con la suma, a diferencia de que esta vez en vez de aplicar la función ".add" aplique la función ".product" que es la encargada de llevar una sumatoria de multiplicaciones de cada elemento de la lista, por lo que la complejidad de la

implementación seria multiplicar 2 elementos y este por la cantidad de elementos de la lista.

$$O(u \,.\, v) \,.\, O(\, l) \rightarrow Donde\ deberia\ recorrer\ cada\ elemento\ de\ la\ lista\ y\ realizar\ multiplicacion\ en\ cada\ elemento$$

Realización del UVA – Bigger Or Smaller:

Para la realización del UVA, copie solamente las implementaciones que consideraba necesarias para la realización, como el constructor BigInteger que se creaba a partir de un "string" y la sobrecarga del operador "<", la parte de ingreso de datos separo los 2 numeros en 4 Strings. Separo la parte entera de la parte decimal, para realizar esto creo un ciclo que recorra el string hasta que encuentre el carácter "." Y lo guardo en el primer string, y después la posición donde esta el "."+1 pasa a ser la parte del siguiente string, cuando tenemos los 4 strings, creamos los 4 BigIntegers por medio del constructor, y pasamos a la fase de comparación.

$$204578.4521 \rightarrow "204578" \ y \ "4521"$$

$$78978.4621 \rightarrow "789578" \ y \ "4621"$$

Después, en la parte de comparación, revisamos primero las 2 partes enteras de los números, que seria el "num1FirstPart" y "num2FirstPart" revisamos la posibilidad en el

que este sea mayor y en el caso que sea menor, en caso tal de que la parte entera sean iguales, tendríamos que pasar a comparar la parte decimal, que seria un procedimiento muy parecido a lo nombrado anteriormente.

En el código para ahorrarme trabajo utilizo "int dotIndex = num1.find('.');" esta línea busca el primer punto (.) en la cadena num1 y utilizando la función find('.'). Devuelve la posición del primer punto encontrado, y como siempre va a estar el carácter "." En el string de entrada, puedo facilitar la implementación aplicando esta función.

```
num2SecondPart = num2.substr(dotIndex + 1);
```

Al utilizar la función subcadena str, desde la posición dotIndex, que es la posición donde esta el "." Más 1hasta el final de la cadena, es decir, tomaremos solamente la parte decimal.