Facoltà di Ingegneria

Corso di laurea in Ingegneria Informatica

“BigInt”

Professore del corso di POO:

Prof. Libero Nigro

Studente:

Carmelo Gugliotta

ANNO ACCADEMICO 2020-2021

Progetto: BigInt

Il progetto consente di effettuare operazioni aritmetiche di numeri interi (senza segno), consentendo anche di superare le capacità di calcolo degli interi long (64 bit). In particolare, si ha la possibilità di manipolare Interi Immutabili a precisione arbitraria.

La gerarchia di classi utilizzata per lo sviluppo del progetto è la seguente:

1. BigInt (**Interfaccia**)
2. AbstractBigInt (**Classe Astratta**)
3. BigIntLL (**Classe Concreta**)

1. BigInt (Interfaccia)

L'interfaccia ci permette di osservare i dettagli più importanti della classe implementata. Quest'ultima presenta 13 metodi, di cui 3 **astratti** e 10 **definiti**.

I 3 metodi **astratti** sono i seguenti:

- Add ()
- Sub ()
- Factory ()

I metodi elencati non sono stati **definiti** all'interno dell'interfaccia poiché la loro risoluzione varia, al variare della **Collection** utilizzata per manipolare gli interi, per tanto verranno definiti nelle classi concrete.

I 10 metodi **definiti** sono i seguenti:

- Value ()
  - Restituisce il valore del BigInt istanziato(this) come oggetto Stringa.
- Zero ()
  - Restituisce un BigInt che esprime il valore intero 0.
- Uno ()
  - Restituisce un BigInt che esprime il valore intero 1.
- Lenght ()
  - Restituisce il numero dei numeri utilizzati per esprimere il BigInt.
- Decr ()
  - Decrementa Il BigInt istanziato (this) di 1.
- Incr ()
  - Incrementa il BigInt istanziato (this) di 1.
- Mul (BigInt m)
  - Riceve un BigInt M. Restituisce un nuovo BigInt che è il risultato dell'operazione [this*m].

Tale operazione all'interno dell'interfaccia è implementata come somma ripetuta, processo molto dispendioso se vengono trattati Numeri>>0. All'interno della classe concreta tale metodo potrà essere ridefinito per attuare determinate semplificazioni.

- Pow (int exp)
  - Riceve un Int exp. Restituisce un nuovo BigInt che è il risultato dell'operazione [this^exp].
    Tale operazione all'interno dell'interfaccia è implementata come moltiplicazione ripetuta.

- Rem (BigInt d)
  - Riceve un BigInt d. Restituisce un nuovo BigInt che è il resto della divisione [this/d].
    Tale operazione all'interno dell'interfaccia è implementata secondo la definizione stessa di resto qui riportata.
    Resto: Il resto è quel numero positivo che bisogna addizionare al prodotto del divisore per il quoziente per ottenere il dividendo da cui segue che
    $$\boldsymbol{rem = Dividendo - (Quoziente * Divisore)}$$

- Div (BigInt d)
  - Riceve un BigInt d. Restituisce un nuovo BigInt che è il Quoziente della divisione [this/d]. Tale operazione all'interno dell'interfaccia è implementata come sottrazione ripetuta.
    Si basa sul concetto di contenimento della quantità espressa dal divisore nella quantità espressa dal dividendo. Intuitivamente, infatti, se il divisore sta, ad esempio, circa tre volte all'interno del dividendo, allora il loro rapporto (e quindi il risultato della divisione) sarà circa tre.

2. AbstractBigInt (Classe Astratta)

La classe astratta implementa l'interfaccia discussa precedentemente, e contiene al suo interno 4 metodi fondamentali per la manipolazione dei dati:

- toString ()
  - Ritorna la rappresentazione in Stringa dell'oggetto (tramite StringBuilder).
- Equals (Object x)
  - Riceve un oggetto x e ritorna un booleano True/False che indica il risultato della comparazione tra i due oggetti. (In questo caso BigInt)
- hashCode ()
  - Ritorna un intero come valore che indica l'hashCode del BigInt istanziato.
    Numero Primo utilizzato per effettuare l'hashing, 43.
- CompareTo (BigInt x)
  - Riceve un BigInt e ritorna come valore un intero appartenente al seguente insieme {-1,0,1} che indicano se il BigInt istanziato è
    minore|uguale|maggiore del BigInt passato come parametro.
    Il ragionamento utilizzato è il seguente:
    - Inizialmente si controllano le lunghezze dei due numeri da confrontare:

1. Se il primo contiene più cifre del secondo allora esprimerà sicuramente un valore numerico maggiore.
2. Se il secondo contiene più cifre del primo allora a priori possiamo affermare che sicuramente esprime un valore più grande.

- Successivamente ai controlli, se il metodo è ancora in esecuzione allora la lunghezza dei due BigInt è uguale, quindi:
  1. Se la cifra più significativa del primo è maggiore della cifra più significativa del secondo allora, il valore espresso dal primo è sicuramente più grande.
  2. Se la cifra più significativa del primo è minore della cifra più significativa del secondo allora, il valore espresso dal secondo è sicuramente più grande.
  3. A questo punto siamo nel caso in cui le cifre sono uguali e passiamo alla prossima, ovvero la seconda più significativa ripetendo i ragionamenti svolti nel primo e secondo punto.
- Se successivamente al controllo delle cifre, il metodo è ancora in esecuzione allora le cifre contenute nel primo e nel secondo BigInt, sono identiche, e allora possiamo affermare che i due oggetti esprimono lo stesso valore.

3. BigIntLL (Classe Concreta)

La classe concreta estende la classe astratta AbstractBigInt, e permette di manipolare, grazie al supporto di una LinkedList di Integer, gli interi a precisione arbitraria.

Presenta 12 metodi che permettono le operazioni, in maniera ottimizzata, che erano state proposte come obbiettivo del progetto.

Metodi:

- BigIntLL (int x)
  - Costruttore che riceve un intero x positivo, che tramite la conversione in un oggetto stringa di quest'ultimo, si preoccupa dell'istanziamento corretto dell'oggetto BigIntLL.
- BigIntLL ()
  - Costruttore privato, che si preoccupa dell'istanziamento di un oggetto BigIntLL caratterizzato da una LinkedList vuota. (Metodi di appoggio).
- BigIntLL (String x)

- Costruttore che riceve una stringa x, e prelevando volta per volta le cifre di cui è caratterizzata costruisce l'oggetto BigIntLL. Lancia **IllegalArgumentException** se la stringa esprime valore con segno.
  Se la stringa passata come parametro non rispetta le condizioni di conversione, viene lanciata l'eccezione **NumberFormatException** che viene catturata dal blocco ***try-catch*** e gestita in modo tale da avvisare l'utente della stringa malformata inserita.

- BigInt div (BigInt d)
  - Riceve un BigInt d (**Divisore**). Restituisce un nuovo BigInt che è il Quoziente della divisione [this/d]. Lancia **IllegalArgumentException** se il **divisore** è minore o uguale a 0 e se il **divisore** è maggiore del **dividendo**.
    Il seguente metodo ridefinisce la divisione implementata all'interno dell'interfaccia, diminuendo il costo temporale grazie al metodo privato div2, che attua la divisione per 2.
    Inizialmente vengo inizializzati gli operatori principali della nostra divisione (**Dividendo**, **Divisore**, Quoziente(**Q**)), successivamente finché il Dividendo>=Divisore, dividiamo il **Dividendo** per 2, conserviamo il risultato in **tempA,** conserviamo il Divisore in tempB e inizializziamo il counter a uno.

    Qui avviene l'ottimizzazione che ci consente di diminuire il costo temporale dell'algoritmo di partenza.

```
  while (tempA.compareTo(tempB)>=0) {
1
2         tempB=(BigIntLL)tempB.mul(factory (2));
3
4         counter=(BigIntLL)counter.mul(factory(2));
  }
```

    Infatti, noi non effettuiamo la differenza tra il Dividendo e il Divisore passato come parametro, poiché all'aumentare della differenza di valore tra quest'ultimi aumentano anche le operazioni di differenza necessarie per restituire il valore esatto della divisione.
    Per evitare ciò, moltiplichiamo il nostro Divisore per 2 finché esso non diventa maggiore del Dividendo/2, tenendo conto anche del quoziente (**counter**).
    Dopodiché si effettua la differenza tra **Dividendo** e il nuovo **Divisore** (tempB), conserviamo il risultato in Dividendo e aggiorniamo il **Quoziente** (Q).
    Ripetiamo il processo fintantoché il **Dividendo** non è minore del **Divisore**.

- Add (BigInt a)
  - Riceve un BigInt a (Addendo) e restituisce un BigInt ris che è il risultato dell'operazione [this+a].
    L'algoritmo implementa l'operazione simulando l'addizione in colonna. Innanzitutto, si inizializzano due iteratori che puntano tra la cifra meno significativa e la size (così da richiamare la prima cifra meno significativa all'esecuzione di previous), dei numeri da addizionare. Successivamente, se possibile, si effettua su tutte e due le liste una call di previous, e si effettua la somma delle cifre prelevate, tenendo conto di un probabile resto (proveniente da una somma precedente ovviamente).

La somma è sempre un valore appartenente al seguente intervallo [0, 19], in particolare si può notare che se si effettua il **modulo**, tra la **somma ottenuta** e 10, otteniamo sempre la cifra meno significativa che deve essere aggiunta al nostro risultato. Un'altra importante osservazione è che se la **somma ottenuta** è un valore **maggiore o uguale** a **10** ovviamente abbiamo un resto che influenzerà la somma successiva altrimenti no (resto=0).
Il procedimento continua finché tutte e due i numeri, **contemporaneamente,** sono costituiti da una successiva cifra più significativa.
Dopodiché, se l'iterazione precedente è cessata e uno dei due numeri è ancora costituito da cifre più significative, allora aggiungiamo le successive cifre tenendo sempre conto di un probabile resto precedente.
(l'if è ridondante, aggiunto solo per marcare il perché dei due cicli while).
Finite le cifre, restituiamo il risultato. (Ricordandoci ovviamente anche del resto).

- Sub (BigInt b)
    - Riceve un BigInt b (Sottraendo) e restituisce un BigInt Differenza che è il risultato dell'operazione [this-a]. Lancia **IllegalArgumentException** se il BigInt passato come parametro b è maggiore di this (Minuendo).
      L'algoritmo implementa l'operazione simulando la sottrazione in colonna. Innanzitutto, si inizializzano due iteratori che puntano tra la cifra meno significativa e la size (così da richiamare la prima cifra meno significativa all'esecuzione di previous), dei numeri da sottrarre. Successivamente, se possibile, si effettua su tutte e due le liste una call di previous, e si effettua la differenza delle cifre prelevate, tenendo conto di un probabile prestito dell'unità (proveniente da una differenza precedente) e della richiesta di un prestito dell'unità (per la corretta riuscita della differenza corrente).

      Prestito dell'unità: Attenzione quando preleviamo la successiva cifra più significativa, dobbiamo tenere conto se è stato effettuato un prestito dell'unità e se tale prestito si ripercuote anche sulle cifre più significative. (ES: Una cifra meno significativa richiede il prestito dell'unità a una cifra più significativa che è pari a 0).
      Il procedimento continua finché il sottraendo**,** è costituito da una successiva cifra più significativa. (Il sottraendo è costituito da un numero di cifre, minori del minuendo o pari a quest'ultimo, ma non può possedere in alcun modo una cifra in più.) (**MINUNDO>=SOTTRAENDO**).
      Dopodiché, se l'iterazione precedente è cessata e il minuendo è costituito da successive cifre più significative, allora aggiungiamo le successive cifre tenendo sempre conto di un probabile prestito dell'unità precedente.
      Finite le cifre, restituiamo il risultato. (Ricordandoci di eventuali 0 superflui che si possono essere generati, durante l'operazione di sottrazione cifra a cifra).

- Mul (BigInt B)
    - Riceve un BigInt b e restituisce un BigInt Prod che è il prodotto della moltiplicazione [this X b].

L'algoritmo implementa l'operazione simulando la moltiplicazione per colonna.
Innanzitutto, al fine di effettuare meno operazioni possibili, si sceglie come secondo fattore, il numero costituito da meno cifre, il più piccolo tra i due.
Successivamente, si preleva la cifra meno significativa disponibile e si effettua la moltiplicazione con le cifre del primo fattore, tenendo conto anche degli eventuali riporti.
Riporti: Per determinare il riporto che influenzerà i successivi passaggi, si divide per dieci il prodotto corrente, ottenendo così la seconda cifra**. I riporti possono assumere valori tra [0,90].**
Finite le cifre del **primo fattore**, aggiungiamo il risultato ottenuto al risultato corrente e passiamo alla successiva cifra del **secondo fattore** (sino all'ultima).

- factory (int x)
  Riceve un intero x e restituisce un BigIntLL che esprime il valore del parametro passato.
- Iterator ()
  Restituisce l'iteratore della collezione di dati utilizzata (LinkedList in questo caso).
- Div2()
  Restituisce il quoziente dell'operazione [this/2].
  Un caso semplice di divisione in colonna è la divisione per 2.
  La divisione per due, infatti è molto semplice da implementare poiché il resto della divisione tra due cifre ([0,9] / 2) può essere o 0 se pari, o 1 se dispari.
  Il metodo è definito private, poiché è finalizzato all'implementazione della divisione per N.

- Rimuovi_0(LinkedList<Integer> x)
  Riceve una lista di cifre (Integer) ed effettua delle modifiche su quest'ultima per una corretta espressione del valore contenuto.
  In particolare, se tramite l'iteratore (dalla cifra più significativa alla cifra meno significativa) si incontrano degli 0 e sono classificati come superflui, vengono rimossi, altrimenti no.
  La classificazione di superfluo viene dettata da un **booleano**:
  Flag==true→ è stato incontrato precedentemente almeno un numero tra [1,9]→non superfluo
  Flag==false→ non è stato incontrato precedentemente almeno un numero tra [1,9]→superfluo

- listIterator (int indice)
  Riceve un intero e restituisce un listIterator (Iteratore in grado di muoversi sia verso posizioni successive che precedenti) che viene inizializzato nella posizione passata come parametro.