Giorgio Ubbriaco – matricola:209899 – 2° anno Ingegneria Informatica formativo

Aritmetica Intera Estesa

Progetto costituito da quattro file:

* BigInt
* AbstractBigInt
* BigIntLL
* Application

Essi si trovano tutti all’interno del medesimo package denominato poo.math.

Il primo file è un’interfaccia pubblica che estende due interfacce: Comparable e Iterable. Pertanto, gli oggetti saranno comparabili con altri oggetti di tipo BigInt e iterabili essendo essi stessi Integer. In essa sono presenti sia metodi implementati come default sia metodi dichiarati abstract così da essere implementati direttamente nella classe concreta.

Il metodo value(), implementato direttamente nell’iterfaccia come default, ritorna il valore del BigInt sottoforma di stringa di caratteri. Pertanto, ho dichiarato uno StringBuilder nel quale aggiungere gradualmente le cifre del numero considerato e un iterator che mi permette di “navigare” cifra per cifra sul numero. Infine, restituisco il valore in formato stringa.

Il metodo length(), implementato come default, ritorna il numero di cifre del BigInt considerato. Inizializzo una variabile intera, chiamata c, e itero sul BigInt (this) e infine ritorno la variabile c.

Il metodo incr() ritorna un BigInt incrementato di una cifra unitaria. Pertanto, invoca il metodo add(BigInt a) passando come parametro, quindi, la cifra “1”.

Il metodo decr() ritorna un BigInt decrementato di una cifra unitaria. Ovviamente, se il BigInt considerato (this) è pari a zero bisogna invocare un’eccezione unchecked.

Il metodo mul(BigInt m), implementato anch’esso come default, riceve come parametro un oggetto BigInt ed effettua la moltiplicazione tra il BigInt (this) e quello ricevuto come parametro. Vengono creati due BigInt denominati ret, il quale verrà inzializzato a this tramite un costruttore per copia così da non modificare il “this”, e c inizializzato a 0. Pertanto, si instanzia un while e finché c risulta essere minore del BigInt passato come parametro, si aggiunge a ret il BigInt this e si incrementa c. Infine, ritorna il BigInt ret che conterrà il risultato della moltiplicazione.

Il metodo div(BigInt d) riceve come parametro un oggetto BigInt ed effettua la divisione tra il BigInt (this) e quello ricevuto come parametro. Inizialmente vengono verificate determinate condizioni: se this risulta essere minore del parametro passato o se il parametro passato risulta essere pari a zero allora sollevo un’eccezione unchecked. Verifico, oltretutto, che this non sia pari a zero così da restituire (nell’ipotetico caso) subito zero e pertanto non effettuare ulteriori operazioni. Se tutti i “controlli” sono tali da permettere la “normale” divisione, allora inizializzo un BigInt ret alla cifra unitaria e finché this risulta essere maggiore o uguale della moltiplicazione tra ret e il parametro passato al metodo, incremento ret. Infine, essendo che ret sarà uguale al risultato della divisione però incrementato di una cifra unitaria, viene decrementato e ritornato.

Il metodo rem(BigInt d) ritorna il quoziente della divisione intera tra this e il parametro BigInt d passato al metodo. Se this risulta essere minore di d allora sollevo un’eccezione unchecked, altrimenti inizializzo due BigInt: ret (pari a zero) e sottraendo, che contiene il risultato della moltiplicazione tra d e il risultato della divisione ottenuto tra this e d stesso. Infine, ritorno ret che conterrà il risultato della sottrazione tra this e l’oggetto sottraendo.

Il metodo pow(int exponent) calcola il this elevato ad un esponente intero che gli viene passato come parametro. Controllo che l’esponente non sia pari a zero altrimenti per default viene ritornato 1. Quindi, inizializzo un BigInt ret, che dovrà infine essere ritornato, al valore di this tramite il costruttore per copia e assegno a ret il risultato della moltiplicazione tra ret stesso e this tante volte quanto è grande l’esponente passato al metodo (il tutto gestito da un for).

La classe AbstractBigInt è una classe astratta che implementa l’interfaccia BigInt. In essa sono presenti tre metodi di cui viene fatto l’override essendo metodi già presenti nella classe Object. Nel metodo toString() inizializzo uno StringBuilder e aggiungo tramite un iteratore le cifre del BigInt corrispondente e infine ritorno la stringa con il valore all’interno. Nel metodo equals(Object o) inizialmente verifico le condizioni base e successivamente inizializzo due iteratori che andranno a confrontare i due BigInt cifra per cifra. Nel metodo hashCode() effettuo le operazioni di base con la piccola aggiunta che l’intero che restituirò conterrà la sommatoria dell’intero stesso moltiplicato per una costante M e sommato all’hashCode della cifra i-esima che compone il this.

Nella classe BigIntLL ho instanziato una linked list, che conterrà appunto le cifre ordinate del BigInt considerato, e la regex per verificare se la stringa, che viene passata al costruttore, soddisfa i requisiti adatti per essere considerato appunto BigInt. La stringa, che verrà passata come parametro al primo costruttore, verrà controllata affinché essa non contenga tutti zeri (controllo effettuato con un metodo zero(String s)) così da aggiungere le varie cifre alla linked list. Il controllo degli “zeri” verrà gestito tramite una variabile booleana (inzializzata a true) che aiuterà ad eliminare gli zeri in eccesso all’inizio di un numero (nel caso in cui la stringa sia del tipo “000345” considerato quindi come “345”). Ovviamente gli zeri trovati dopo una cifra diversa da zero verranno considerate cifre vere e proprie (ad esempio “0030450” -> “30450”). Ho previsto anche un costruttore al quale viene passato soltanto un intero per valori pressoché “piccoli” e un costruttore per copia utile nelle inizializzazioni di BigInt in varie operazioni.

Il metodo factory implementato qui nella classe concreta come anche altri metodi quali add e sub, crea appunto un BigInt tramite il costruttore “per intero”.

Nel metodo add, inizialmente viene creato un BigInt ret (pari a zero) che sarà poi quello che verrà restituito alla fine (dopo le varie operazioni dell’add) così da non dover modificare gli operandi (come succede infatti in ogni operazione implementata). Inizializzo due list iterator (all’ultima cifra del BigInt) sia per this che per il BigInt x passato come parametro e due variabili intere (sommaCorr e riporto) che rispettivamente conterranno la cifra i-esima ottenuta dopo l’operazione add, che dovrà essere aggiunta alla linked list, e il riporto i-esimo-1 che dovrà essere aggiunto gradualmente alla cifra i-esima prima di essere aggiunta alla linked list. Ovviamente se this è piu grande di x effettuo this+x altrimenti richiamo l’add come x+this così da avere un caso generale. Fin quando il primo list iterator ha un precedente sommo le cifre i-esime del primo BigInt e del secondo e propago un eventuale riporto. Se il secondo list iterator non ha piu precedenti aggiungo le cifre rimanenti del primo ed un eventuale riporto finale (line 105).

Nel metodo sub il modus operandi è molto simile al metodo add tranne per il fatto che viene fatta la sottrazione delle cifre i-esime con un eventuale “prestito” se la prima cifra i-esima è minore della seconda cifra i-esima. Inizialmente viene verificato che il primo operando non sia minore del secondo altrimenti viene sollevata un’eccezione unchecked. Successivamente, durante le varie operazioni se il secondo list iterator non ha ulteriori precedenti si aggiungono le rimanenti cifre del primo BigInt considerando ovviamente precedenti “prestiti” di cifra. Infine, tramite l’ausilio di un iterator viene fatto un controllo sul risultato ottenuto per verificare che non siano presenti “zeri” in testa al risultato così eventualmente da essere rimossi (line 156-165 e line 185-194).

Nel metodo mul la metodologia delle operazioni è molto simile all’add (tanto è vero che possono essere considerate valide operazioni sia se this>x che x>this semplicemente richiamando x.mul(this)). Vengono effettuati preliminarmente alcuni controlli così da evitare eventualmente operazioni in “eccesso”. La gestione della moltiplicazione è ottenuta con l’ausilio di tanti zeri (da aggiungere) quanto è la posizione del moltiplicatore i-esimo (del secondo BigInt) considerato (line 223). Tanto è vero che il risultato ottenuto dalla moltiplicazione tra il this e il moltiplicatore i-esimo viene salvato in una linked list la quale verrà sommata al BigInt ret (che dovrà essere restituito alla fine come risultato della moltiplicazione generale) cifra per cifra (line 231) e successivamente incrementata la posizione i-esima del moltiplicatore da considerare per la prossima moltiplicazione (line 232). Pertanto, si avranno N moltiplicazioni per N cifre del moltiplicatore generale (il BigInt x passato come parametro alla mul) e quindi N somme algebriche.

Il metodo compareTo viene implementato per confrontare i BigInt tramite un confronto cifra per cifra ottenute grazie ai due list iterator inizializzati all’inizio del metodo.

Infine, mi sono permesso di aggiungere un’ulteriore classe (Application) che comprende una piccola “GUI” (anche se non era prevista nella traccia) che permette all’utente di effettuare operazioni algebriche inserendo la tipologia dell’operazione tramite parole chiavi quali “add”, “addizione”, “sub”, “sottrazione”, “pow”, “potenza”, ecc. all’interno di un input dialog. Ovviamente, la parola chiave inserita (la tipologia di operazione inserita come richiesto nella finestra) non è case-sensitive poiché di essa viene fatto in automatico l’upper case. Dopo aver inserito l’operazione, vengono visualizzate due ulteriori input dialog dove dovranno essere inseriti rispettivamente il primo e il secondo BigInt. Ovviamente per parole chiavi (tipologie di operazioni) o valori non validi vengono visualizzate finestre di errore tramite l’ausilio di finestre message dialog. Se i valori o le parole chiavi inserite non sono valide viene re-visualizzata la finestra di input per inserire nuovamente il valore o la parola chiave. Il risultato ottenuto dall’operazione, calcolato tramite un metodo relativo per ogni operazione, considerata viene visualizzato tramite una finestra message dialog.