BigInteger (autore: Vittorio Albertoni)

Premessa

I tipi di dato numerico che manipoliamo con i linguaggi di programmazione sono due: intero e decimale, in informatica meglio denominato a virgola mobile. Ci sarebbe anche il tipo complesso ma in questa sede non ci interessa.

Perché questi dati numerici possano essere manipolati è necessario siano inseriti nella memoria del computer in formato binario e tradizionalmente, al fine di equilibrare validità dei risultati della manipolazione, tempo necessario per ottenerli e impiego di risorse, si dedica alla memorizzazione del numero uno spazio di memoria prefissato e limitato, il cui valore massimo in passato era di 4 byte (32 bit), oggi, grazie al progresso tecnologico, di 8 byte (64 bit).

L'inserimento di numeri a virgola mobile in questi spazi di memoria limitati avviene in modo tale che del valore numerico si salvano solo i bit che ci stanno ed interviene pertanto il concetto di precisione. Se lo spazio di memoria è di 32 bit si parla di precisione singola (in pratica abbiamo solo sette cifre buone), se lo spazio di memoria è di 64 bit si parla di precisione doppia (in pratica abbiamo 15 o 16 cifre buone).

In ogni caso i numeri a virgola mobile, male che vada, vengono salvati in memoria in modo approssimato e la loro manipolazione genera risultati spesso approssimati ma mai sbagliati.

Utilizzando certi linguaggi di programmazione abbiamo a disposizione strumenti, così detti di precisione estesa, attraverso i quali è possibile aumentare lo spazio di memoria in modo da aumentare il numero di cifre buone da memorizzare e migliorare così l'approssimazione ma, per quanto riguarda l'aritmetica dei numeri a virgola mobile direi che basta ciò che troviamo normalmente.

L'inserimento di numeri interi in spazi di memoria limitati avviene invece in maniera secca: se il numero che intendiamo memorizzare ci sta, bene, se non ci sta può succedere di tutto in maniera assolutamente imprevedibile.

Se lavoriamo a 32 bit senza riguardo al segno il più grande intero che possiamo memorizzare è pari a $2^{32} - 1$, cioè 4.294.967.295, se lavoriamo a 64 bit senza riguardo al segno il più grande intero che possiamo memorizzare è $2^{64} - 1$, cioè 18.446.744.073.709.551.615.

Sicuramente numeri di tutto rispetto ma non sempre adeguati per certi tipi di calcolo.

Anche a 64 bit non riusciremmo, per esempio, a calcolare il fattoriale di un numero superiore a 20, un po' poco per fare del calcolo combinatorio.

Per non parlare di cosa succederebbe se affrontassimo con queste limitazioni calcoli per la crittografia.

E' per ovviare a questi inconvenienti che esiste la possibilità di trattare i numeri interi, come si suol dire, a precisione arbitraria, ovvero riservando al loro trattamento tante cifre utili quante disponibili in tutta la memoria del sistema ospite.

In questo modo possiamo rappresentare e trattare con esattezza numeri interi anche di svariate migliaia di cifre.

La velocità di elaborazione ne soffre ma, con le potenze delle CPU di oggi, nemmeno ce ne accorgiamo.

Utilizzando certi linguaggi di programmazione abbiamo la precisione arbitraria a disposizione per default per il trattamento dei numeri interi, altri linguaggi di programmazione ci offrono la possibilità di disporne con una semplice scelta, per altri ancora possiamo disporne con qualche fatica in più, per altri non abbiamo rimedi ragionevolmente praticabili e non potremo utilizzarli per fare certe cose.

In questo manualetto mi propongo di fare chiarezza su questo scenario, ovviamente limitandomi alla citazione dei linguaggi di programmazione più noti ed usati.

Indice

1 Linguaggi con interi a precisione arbitraria

Sono quelli che non ci danno problemi nel trattamento di numeri interi anche di migliaia di cifre, quanto può stare nella memoria dell'hardware su cui lavoriamo.

In ordine di anzianità sono LISP, Perl, ora rigenerato in Raku, Tcl/Tk, Python e Ruby.

Quanto meno, con questi linguaggi, se i numeri che tentiamo di manipolare sono troppo grandi per poter essere elaborati, ma devono essere proprio enormi, composti da molte migliaia di cifre, non otteniamo risultati e, pertanto, non corriamo il rischio di ottenere risultati errati senza saperlo.

2 Linguaggi con tipi di intero a precisione arbitraria

In questi linguaggi abbiamo a disposizione più tipi di intero tra cui un tipo che genera un trattamento in precisione arbitraria.

Si tratta dei linguaggi NewLISP (dotato, tra gli altri tipi base, del tipo `bigint`) e Julia (dotato, tra gli altri tipi base, del tipo `BigInt` o semplicemente `big`).

In questi linguaggi possiamo scegliere per alcuni numeri il tipo a precisione arbitraria con la stessa semplicità con la quale scegliamo altri tipi, e utilizzare tipi con spazio di memoria limitato per altri numeri dove siamo certi che lo spazio basta, allo scopo di evitare sprechi di risorse e velocizzare l'elaborazione. Se si utilizzano questi altri tipi occorre però evitare la memorizzazione di numeri di dimensione superiore a quella prevista dal tipo di dato in quanto, purtroppo, non sempre la conseguenza di ciò è il crash del programma ma potrebbe essere la fornitura di risultati completamente sballati senza che lo sappiamo.

3 Linguaggi con interi a memoria limitata

In questi linguaggi non c'è alcun modo di ottenere il trattamento di numeri interi per una dimensione superiore a quella della memoria richiesta dal loro tipo dichiarato.

Si tratta dei linguaggi BASIC e Pascal nelle loro varie edizioni[1].

Nel linguaggio Pascal possiamo molto parzialmente ovviare all'inconveniente incanalando l'intero di dimensione superiore a quella massima ammissibile come tipo intero verso il tipo `extended`, che è un tipo a virgola mobile che incamera fino a 20 cifre significative in uno spazio di memoria di 10 bytes.

Un esempio per capirci.

Un classico programmino che crea gli sballi di cui stiamo parlando è quello che calcola il fattoriale di un numero.

Se lo scriviamo, in linguaggio Pascal, in questo modo

```
program fattoriale;
var n:  integer;
function f(n:  integer):  qword;
    begin
        if n =1 then f := 1
        else f := n * f(n-1);
    end;
begin
    write('Numero di cui calcolare il fattoriale:  ');
    readln(n);
    writeln(f(n));
end.
```

[1] Per quanto riguarda il linguaggio Pascal esiste, per la verità, una libreria che ci potrebbe consentire di trattare grandi numeri anche con questo linguaggio, si chiama Numerix e chi voglia approfondire può farlo all'indirizzo *http://pauillac.inria.fr/~quercia/*. Si tratta comunque di una cosa inaccessibile a comuni mortali come noi.

stando attenti ad inserire un numero inferiore a 65.535 (il tipo `integer` che abbiamo assegnato al numero `n` di cui dobbiamo calcolare il fattoriale occupa 2 bytes, cioè 16 bit, di memoria e il numero che gli inseriamo non può pertanto essere superiore a $2^{16} - 1$) otteniamo come risultato un numero di tipo `qword` per il quale, nel linguaggio Pascal, viene riservato uno spazio di memoria di 8 bytes, cioè 64 bit e questo risultato non può pertanto essere superiore a 18.446.744.073.709.551.615 (cioè $2^{64} - 1$).

Se come numero n di cui calcolare il fattoriale inseriamo 20, otteniamo come risultato il numero 2.432.902.008.176.640.000, risultato esatto in quanto è stato possibile memorizzarlo nella spazio disponibile (il suo valore è inferiore al limite massimo).

Se come numero n di cui calcolare il fattoriale inseriamo 21, otteniamo come risultato il numero 14.197.454.024.290.336.768 ma il vero valore del fattoriale di 21 è 51.090.942.171.709.440.000. Dal momento che si tratta di un numero superiore al massimo memorizzabile nello spazio a disposizione si è creato un casino. Tra l'altro aggravato dal fatto che nessuno ci ha avvisati che qualche cosa non andava e, per come sono andate le cose, potremmo essere convinti che il fattoriale di 21 sia 14.197.454.024.290.336.768.

Come si vede è un bell'inconveniente.

Se scrivessimo il nostro programma così

```
program fattoriale;
var n:  integer;
function f(n:  integer):  extended;
    begin
        if n =1 then f := 1
        else f := n * f(n-1);
    end;
begin
    write('Numero di cui calcolare il fattoriale:  ');
    readln(n);
    writeln(f(n));
end.
```

incanalando il risultato dell'operazione sul tipo `extended`, otterremmo come risultato il numero espresso in virgola mobile con notazione scientifica 5.1090942171709440000E+0019 che, con la sua precisione a 20 cifre significative, rappresenta l'esatto valore di 21!, che, caso vuole, è proprio un numero di 20 cifre.

Se calcolassimo con questo programma il fattoriale di 30 otterremmo come risultato il numero 2.652528598121910586 47E+0032, cioè 265.252.859.812.191.058.647.000.000.000.000 quando il valore esatto è 265.252.859.812.191.058.636.308.480.000.000. Come si vede abbiamo una rappresentazione del risultato esatta per le prime 19 cifre con la ventesima arrotondata. Abbiamo, cioè, un valore approssimato ma non errato.

E' chiaro che, più che di un rimedio, si tratta di un palliativo. Se ci serve conoscere esattamente il numero delle permutazioni di un numero di elementi superiori alla ventina che ce ne facciamo di un numero approssimato? Quanto meno, tuttavia, con questo trucco sappiamo di incappare in approssimazioni ma evitiamo di ottenere risultati sbagliati senza che nemmeno lo sappiamo.

Nel linguaggio di programmazione Javascript, dove tutti i valori numerici sono rappresentati in virgola mobile, questo trucco fa parte del linguaggio.

4 Altri casi

Alla categoria dei linguaggi con interi a memoria limitata di cui ho parlato nel precedente paragrafo appartengono anche due linguaggi cardine della programmazione: C/C++ e Java con le relative evoluzioni Go e Kotlin.

Si tratta di linguaggi talmente importanti che non sarebbe possibile confinare tra quelli non adatti ad elaborazioni con grandi numeri interi e si sono cercati rimedi, più o meno completamente risolutivi, comunque quasi sempre di non semplicissima applicazione.

Vediamoli uno per uno.

4.1 Linguaggio C/C++

Per il linguaggio C e C++ la situazione è esattamente la stessa che ho descritto per il linguaggio Pascal nel paragrafo precedente, senza nemmeno la possibilità di applicare il trucco del tipo extended, che in questi casi non c'è, per cui ci dovremmo semmai accontentare del tipo double.

Esiste tuttavia un rimedio risolutivo, nato nel mondo del software libero, che, pur richiedendoci un bel po' di impegno, arriva a parificare il linguaggio C al linguaggio Python nel trattamento di grandi numeri interi: il pacchetto GMP, GNU Multiprecision Library[2].

La prima fatica è quella di installare il pacchetto, che è contenuto nel file `gmp-6.2.1.tar.xz` che ci possiamo procurare all'indirizzo *https://gmplib.org*, nella pagina DOWNLOAD.

Il file che scarichiamo contiene il sorgente da compilare.

Chi lavora su sistema operativo Linux, almeno su Ubuntu e derivate, molto probabilmente si trova già installato tutto ciò che serve senza bisogno di procurarsi il pacchetto e compilarlo. Bene comunque verificare, con un gestore di pacchetti tipo Synaptic, che siano installati anche i file `libgmpdev` e `libgmpxx4ldbl`. Sempre su Linux abbiamo per default tutta l'attrezzatura necessaria per eventualmente compilare il pacchetto con la solita serie di comandi `configure`, `make`, `make check` e `make install` impartiti da root.

Chi lavora su Mac dovrà compilare il sorgente come se fosse su Linux.

Chi lavora su Windows deve innanzi tutto accertarsi di avere installato la versione del compilatore GNU GCC attraverso la distribuzione MinGW e il suo sotto-pacchetto `msys`. Dalla directory `MinGW\msys\1.0` si lancia il file `msys.bat` che apre una shell. Agendo in questa shell ci posizioniamo nella directory che contiene il sorgente da compilare e lo compiliamo, sempre da questa shell, con i soliti comandi `configure`, `make`, `make check` e `make install`.

Una volta riusciti nell'impresa dell'installazione del pacchetto GMP abbiamo a disposizione gli header `gmp.h` per il linguaggio C e `gmp.h` con `gmpxx.h` per il linguaggio C++, attraverso i quali possiamo lavorare in precisione arbitraria utilizzando una serie di funzioni che hanno il prefisso `mpz_`. Il tutto secondo una non semplice sintassi che troviamo illustrata nel manuale GMP che possiamo scaricare in formato PDF da *https://gmplib.org*.

La compilazione dei sorgenti in cui utilizziamo queste cose dovrà avvenire con i seguenti comandi

. in C: `gcc` <nome_source.c> `-o` <nome_eseguibile> `-lgmp`

. in C++: `g++` <nome_source.cpp> `-o` <nome_eseguibile> `- lgmpxx - lgmp`

Così, per disporre di un programmino che calcoli il fattoriale di un numero a precisione arbitraria, con il linguaggio C++ possiamo scrivere

```
#include <iostream>
#include <gmp.h>
#include <gmpxx.h>
using namespace std;
mpz_class fatt(int n)
{
    mpz_class f(1);
    if(n < 0) throw "n < 0";
    if(n == 0 || n == 1) return f;
    for(int i = n; i > 1; i--)
        f *= i;
    return f;
}
```

[2] Anche per il linguaggio C vale quanto ho detto nella nota 1 a pagina 3

```
int main()
{
    int n;
    cout << "Numero di cui calcolare il fattoriale:  ";
    cin >> n;
    cout << fatt(n) << endl;
}
```

e in linguaggio C possiamo scrivere

```
#include <stdio.h>
#include <gmp.h>
void fatt(mpz_t f, int n)
{
    if(n == 0 || n == 1) mpz_set_ui(f, 1);
    if(n < -1) mpz_set_ui(f, 0);
    mpz_set_ui(f, 1);
    for(int i = n; i > 1; i--)
    mpz_mul_ui(f, f, i);
}
int main()
{
    int n;
    mpz_t f;
    mpz_init(f);
    printf("Numero di cui calcolare il fattoriale:  ");
    scanf("%d", &n);
    fatt(f, n);
    gmp_printf("%Zd\n", f);
    mpz_clear(f);
}
```

Si nota come il programma scritto in C++ sia più in sintonia con la normale scrittura in C++ grazie al fatto che si può sfruttare un interfacciamento per oggetti, previsto dalla sintassi del pacchetto GMP, creando una classe per il calcolo del fattoriale all'interno della quale ci si esprime in modo normale.

Nella versione in linguaggio C, dove ci si esprime secondo una logica funzionale, siamo costretti ad utilizzare la sintassi delle funzioni previste dal pacchetto GMP (`mpz_t` per dichiarare il tipo della variabile `f` a precisione arbitraria destinata a contenere il risultato del calcolo, `mpz_set_ui()` come funzione di assegnamento, `mpz_init()` come funzione di inizializzazione, `mpz_mul_ui` come funzione per eseguire il prodotto) e ciò rende l'impresa alquanto ardua.

4.2 Linguaggio Go

Se abbiamo a che fare con il linguaggio Go, nato per fare meglio e più facilmente ciò che si può fare con i linguaggi C e C++, abbiamo innanzi tutto la semplificazione di non dover ricorrere ad un pacchetto esterno per risolvere il nostro problema della precisione arbitraria in quanto è il linguaggio stesso che è dotato di un package che ci consente di lavorare in precisione arbitraria.

Si tratta del package `math/big` e, per poterlo utilizzare, lo dobbiamo semplicemente importare scrivendo all'inizio del nostro programma il comando

```
import "math/big"
```

senza dover preventivamente installare nulla in aggiunta al software Go che abbiamo già installato.

Questo pacchetto prevede tre tipi numerici a precisione arbitraria: `Int` per i numeri interi con segno, `Rat` per i numeri razionali e `Float` per i numeri a virgola mobile (notare le iniziali maiuscole, fondamentali per distinguerli). In questa sede ci interessa il primo.

Purtroppo l'utilizzo di questo package è tutt'altro che semplice e la documentazione che troviamo all'indirizzo *https://pkg.go.dev/math/big* è di difficilissima comprensione per un dilettante alle prime armi.

Alla base di tutto sta il concetto che, una volta importato il package, possiamo creare una variabile di tipo intero a precisione arbitraria con
`var` <nome_variabile> `big.Int`
che possiede una lunga serie di metodi per autoalimentarsi fino a raggiungere valori a precisione arbitraria.

Esiste inoltre la funzione `big.NewInt()` per inizializzare un valore espresso come int64 alla precisione arbitraria.

Sicché, se vogliamo, per esempio, calcolare una potenza ed esprimere il risultato in precisione arbitraria agiamo in questo modo:

```
package main
import "fmt"
import "math/big"
func main() {
    var p big.Int
    p.Exp(big.NewInt(670), big.NewInt(87), nil)
    fmt.Println(&p)
}
```

ed otterremo il risultato di 670^{87} espresso esattamente con un bel numero di 246 cifre.

Tra le funzioni ne troviamo una, `MulRange(x, y)`, che moltiplica tra loro i numeri compresi nel range x...y espressi come int64 e che ci torna molto comoda per calcolare il fattoriale di un numero esprimendo il risultato in precisione arbitraria.

Basta fare così

```
package main
import "fmt"
import "math/big"
func main() {
    var n int64
    fmt.Println("Numero di cui calcolare il fattoriale:")
    fmt.Scan(&n)
    var f big.Int
    f.MulRange(1, n)
    fmt.Println(&f)
}
```

4.3 Linguaggio Java

Anche il linguaggio Java, se ci basiamo sui suoi tipi primitivi, ci consente di elaborare numeri interi non superiori al fatidico limite di $2^{64} - 1$ ma è meno infido dei linguaggi Pascal e C/C++ e quando non ce la fa a gestire un numero troppo grande o interrompe l'esecuzione del programma senza fornire risultato o fornisce risultato 0.

Per poter lavorare con numeri interi più grandi abbiamo a disposizione la classe `BigInteger` che si trova nel pacchetto `java.math`.

Tutto sommato lavorare con questa classe è più agevole rispetto ai complicati procedimenti che abbiamo visto per C/C++ e Go.

Una volta importato il pacchetto con
import java.math.*;
abbiamo praticamente a disposizione la classe BigInteger come fosse un tipo di dato ma che, in realtà, è un oggetto dotato di metodi propri che va costruito e non semplicemente dichiarato.

La documentazione completa della classe BigInteger del pacchetto java.math si trova nella documentazione Java.

Qui richiamo le cose essenziali.

Il costruttore di un oggetto BigInteger è `BigInteger("<valore>")`. Sicché con

```
BigInteger x = new BigInteger("78367362789949372763684900049487225 6");
```

creiamo l'oggetto x contenente il valore intero a precisione arbitraria passato come stringa.

Possiamo utilizzare questo valore in operazioni aritmetiche con altri valori avvalendoci dei metodi dell'oggetto che lo contiene.

Per compiere le operazioni aritmetiche più comuni, al posto degli operatori numerici (`+ - * /`) che utilizziamo con i tipi primitivi dobbiamo utilizzare i metodi chiamati `add`, `subtract`, `multiply` e `divide` di uno degli oggetti coinvolti nell'operazione ed operare con altri oggetti BigInteger come argomenti da passare a questi metodi.

Per esempio, se vogliamo dividere per due il valore dell'oggetto BigInteger x creato prima e scrivere il risultato, possiamo fare così:

```
System.out.println(x.divide(new BigInteger("2")));
```

Per l'elevamento a potenza possiamo creare un oggetto BigInteger per la base e utilizzare il suo metodo `pow()` che, questa volta, accetta come argomento l'esponente espresso come normale intero.

Così la potenza che abbiamo calcolato nel precedente paragrafo con il linguaggio Go, la calcoliamo in linguaggio Java con

```
import java.math.*;
public class Potenza
{
    public static void main(String[] args)
    {
        BigInteger base = new BigInteger("670");
        int esponente = 87;
        BigInteger risultato = base.pow(esponente);
        System.out.println(risultato);
        System.exit(0);
    }
}
```

Il fattoriale di un numero possiamo calcolarlo così

```
import java.io.*;
import java.math.*;
public class Fattoriale {
    public static BigInteger fatt(int n) {
        BigInteger f;
        if (n == 0) {
f = new BigInteger("" + 1);
            }
        else {
            f = (new BigInteger("" + n)).multiply(fatt(n-1));
            }
        return f;
    }
public static void main(String [] args) throws Exception {
    String valore;
    int val;
    BufferedReader br = new BufferedReader( new InputStreamReader(System.in) );
    System.out.println("Introdurre il valore di cui calcolare il fattoriale:  ");
    valore = br.readLine();
    val = Integer.parseInt(valore);
    System.out.println("Il fattoriale vale:  " + fatt(val));
    System.exit(0);
    }
}
```

Mi pare che con Java le cose siano leggermente più semplici rispetto a quanto avveniva nei due casi precedenti.

Forse a causa del relativamente maggiore impegno di memoria virtuale di Java i grandi numeri non sono altrettanto grandi quanto lo sono con i linguaggi con interi a precisione arbitraria e con i correttivi utilizzabili per C/C++ e Go, ma possiamo sempre tranquillamente arrivare a trattare interi fino a 12.000 cifre (tipo fattoriale di 3700).

4.4 Linguaggio Kotlin

Con il linguaggio Kotlin possiamo utilizzare la classe `BigInteger` che si trova nel pacchetto `java.math` importandola con
`import java.math.BigInteger`
ed utilizzarla in un ambiente molto semplificato rispetto all'ambiente Java.

Costruiamo l'oggetto BigInter assegnandolo ad una variabile con
`var x = BigInteger("<valore>")`
Sicché con
`var x = BigInteger("78367362789949372763684900049487225 6")`
creiamo la variabile x contenente il valore intero a precisione arbitraria passato come stringa.

Utilizzando le variabili così costruite possiamo effettuare le operazioni aritmetiche di base utilizzando i soliti normali operatori numerici (`+ - * /`), purché gli operandi siano dello stesso tipo BigInteger.
Per esempio, se vogliamo dividere per due il valore dell'oggetto BigInteger x creato prima e scrivere il risultato, possiamo fare così:
`println(x/BigInteger("2"))`
Per l'elevamento a potenza possiamo creare un oggetto BigInteger per la base e utilizzare il suo metodo `pow()` che accetta come argomento l'esponente espresso come normale intero.
In questo modo

```
import java.math.BigInteger
fun main()
{
    var base = BigInteger("670")
    var esponente = 87
    var risultato = base.pow(esponente)
    println(risultato)
}
```

Con queste semplificazioni diventa anche più agevole scrivere il programma per il calcolo del fattoriale

```
import java.math.BigInteger
fun fattoriale(n:  BigInteger):  BigInteger {
    if (n == BigInteger("0")) {
        return BigInteger("1")
    }
    else {
        return n * fattoriale(n-BigInteger("1"))
    }
}
fun main() {
    println("Numero di cui calcolare il fattoriale:  ")
    var n = BigInteger(readLine())
    println(fattoriale(n))
}
```

* * *

Con quanto abbiamo visto in questo capitolo riusciamo a trattare grandi numeri interi utilizzando i linguaggi C/C++ e Java con le rispettive evoluzioni Go e Kotlin senza avere brutte sorprese.

La fatica richiesta non è tuttavia di poco conto, anche considerando che le esemplificazioni portate sono relative a problemi semplici.

Ovviamente tutto si complica quando le espressioni dei calcoli da effettuare diventano più complesse di quelle utilizzate negli esempi: per esempio quando il calcolo del fattoriale di un numero non serve soltanto per essere stampato sullo schermo o per ottenere il numero delle permutazioni, che coincide con esso, ma serve per essere inserito nelle formule per calcolare disposizioni e combinazioni.

Si vengono così a creare situazioni in cui sono necessarie conoscenze e abilità informatiche oltre la normalità.

Ecco perché scienziati e ricercatori con normali abilità informatiche preferiscono sempre più linguaggi come Python.