Dirty Deeds (DirtyDeeds)

UNA LIBRERIA C PER I NUMERI RAZIONALI

6 April 2013

Oh, con questo articolo esco un po' da quelli che sono i miei soliti argomenti. Recentemente, infatti, mi è capitato di dover scrivere una libreria C che permettesse di fare calcoli ragionevolmente veloci con i numeri razionali (tenendo conto di alcune criticità di cui dirò al §8). Mi sembra sufficientemente completa da potervela proporre, nel caso qualcun altro avesse questo tipo di esigenza.

1 Requisiti e portabilità

La libreria è scritta in C standard, per cui non ci dovrebbero essere grossi problemi di portabilità. E' però richiesto un compilatore che implementi lo standard C99. Chi usasse il compilatore gcc può compilare con l'opzione -std=c99.

2 Tipi e inizializzazioni

Questa libreria definisce tre tipi di razionali, uno per ogni tipo intero supportato dal C:

```
typedef struct {
    int num, den;
} rational;

typedef struct {
    long num, den;
} lrational;

typedef struct {
    long long num, den;
} llrational;
```

Se vogliamo inizializzare una nuova frazione possiamo scrivere:

```
rational r = {2, 3};
```

In r.num viene memorizzato il numeratore e in r.den il denominatore.

In generale, però, è meglio lavorare mantenendo le frazioni normalizzate, con numeratore e denominatore ridotti ai minimi termini e denominatore positivo. Per inizializzare una frazione in modo che sia normalizzata si possono usare le funzioni:

```
rational rnew(int num, int den)
lrational rnewl(long num, long den)
llrational rnewll(long long num, long long den)
```

Per esempio,

```
rational r = rnew(4,6);
```

memorizzerà in r la frazione 2/3. Se vogliamo invece normalizzare una frazione dopo l'assegnazione possiamo usare le funzioni

```
void rnorm(rational *rp)
void rnorml(lrational *rp)
void rnormll(llrational *rp)
```

Per esempio,

```
rational r = {4,-6};
rnorm(&r);
```

normalizzerà r a -2/3.

Tutte le funzioni della libreria garantiscono un risultato normalizzato se le variabili di ingresso sono normalizzate.

3 Frazioni invalide

Se una frazione ha denominatore nullo, il comportamento delle funzioni della libreria è *indefinito*. Nessun tentativo viene fatto per intercettare una divisione per 0. Tale comportamento è voluto ed è coerente con lo standard del linguaggio C che, per gli interi, non prevede nessun meccanismo per la gestione della divisione per 0. Sta quindi al programmatore garantire che le frazioni in ingresso alle funzioni siano valide. Per controllare se una frazione è valida si possono usare le funzioni seguenti:

```
int risvalid(rational r)
int risvalidl(lrational r)
int risvalidll(llrational r)
```

Il valore di ritorno è diverso da zero se r è una frazione valida.

4 Costanti

Vengono definite le seguenti costanti:

```
const rational RATIONAL_ZERO = {0,1};
const lrational LRATIONAL_ZERO = {0L,1L};
const llrational LLRATIONAL_ZERO = {0LL,1LL};

const rational RATIONAL_ONE = {1,1};
const lrational LRATIONAL_ONE = {1L,1L};
const llrational LLRATIONAL_ONE = {1LL,1LL};

const rational RATIONAL_MAX = {INT_MAX,1};
const rational RATIONAL_MIN = {INT_MIN,1};

const lrational LRATIONAL_MAX = {LONG_MAX,1};
const lrational LRATIONAL_MIN = {LONG_MIN,1};

const llrational LLRATIONAL_MAX = {LLONG_MAX,1};
const llrational LLRATIONAL_MIN = {LLONG_MIN,1};
```

Insomma, lo 0, l'1 e i limiti delle variabili di ciascun tipo. Tali costanti si possono usare nelle assegnazioni e nelle chiamate di funzioni.

5 Classificazione delle frazioni

La libreria rende disponibili alcune funzioni che permettono di verificare se una frazione è intera, propria, unitaria, negativa o positiva.

```
int risinteger(rational r)
int risintegerl(lrational r)
int risintegerll(llrational r)
```

Queste funzioni ritornano un valore non nullo se r è equivalente a un numero intero.

```
int risproper(rational r)
int risproperl(lrational r)
int risproperll(llrational r)
```

Queste funzioni ritornano un valore non nullo se r è una frazione propria, cioè se il valore assoluto del numeratore è strettamente minore del valore assoluto del denominatore.

```
int risunitary(rational r)
int risunitaryl(lrational r)
int risunitaryll(llrational r)
```

Queste funzioni ritornano un valore non nullo se r è una frazione unitaria, cioè se il numeratore vale +1 o -1 (quando la frazione sia ridotta ai minimi termini).

```
int risnegative(rational r)
int risnegativel(lrational r)
int risnegativell(llrational r)
```

Queste funzioni ritornano un valore non nullo se r è una frazione negativa, cioè se numeratore e denominatore hanno segni opposti. L'implementazione si basa sul metodo descritto da S. E. Anderson in Bit Twiddling Hacks.

```
int rispositive(rational r)
int rispositivel(lrational r)
int rispositivell(llrational r)
```

Queste funzioni ritornano un valore non nullo se r è una frazione positiva, cioè se numeratore e denominatore hanno lo stesso segno.

Un esempio d'uso delle funzioni precedenti è dato dal seguente programma:

```
/* test-classification */

#include <stdio.h>
#include "rational.h"


int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++) {
        rational a;
        if (sscanf(argv[i], "%d/%d", &a.num, &a.den) < 2 || !risvalid(a)) {
            fprintf(stderr, "test-classification: invalid fraction: %s\n", argv[i]);
            return 1;
        }

        printf("isinteger(%d/%d): %c\n", a.num, a.den, (risinteger(a) ? 'T' : 'F'));
        printf("isproper(%d/%d): %c\n", a.num, a.den, (risproper(a) ? 'T' : 'F'));
        printf("isunitary(%d/%d): %c\n", a.num, a.den, (risunitary(a) ? 'T' : 'F'));
        printf("ispositive(%d/%d): %c\n", a.num, a.den, (rispositive(a) ? 'T' : 'F'));
        printf("isnegative(%d/%d): %c\n\n", a.num, a.den, (risnegative(a) ? 'T' : 'F'))
```

```
    }

    return 0;
}
```

Tale programma può essere eseguito con il comando:

```
test-classification [r1] [r2] ...
```

dove r1, r2 ecc. sono delle frazioni nella forma "numeratore/denominatore". Per ogni frazione immessa, il programma riporta quali proprietà sono vere (T) e quali sono false (F). Per esempio:

```
>test-classification 18/14 -6/12 15/3
isinteger(18/14): F
isproper(18/14): F
isunitary(18/14): F
ispositive(18/14): T
isnegative(18/14): F

isinteger(-6/12): F
isproper(-6/12): T
isunitary(-6/12): T
ispositive(-6/12): F
isnegative(-6/12): T

isinteger(15/3): T
isproper(15/3): F
isunitary(15/3): F
ispositive(15/3): T
isnegative(15/3): F
```

6 Operazioni

6.1 Opposto

```
rational ropposite(rational r)
lrational roppositel(lrational r)
llrational roppositell(llrational r)
```

Queste funzioni ritornano la frazione $-r$.

6.2 Reciproco

```
rational rreciprocal(rational r)
lrational rreciprocall(lrational r)
llrational rreciprocalll(llrational r)
```

Queste funzioni ritornano la frazione $1/r$.

6.3 Somma

```
rational radd(rational a, rational b)
lrational raddl(lrational , lrational b)
llrational raddll(llrational a, llrational b)
```

Queste funzioni ritornano la frazione $a + b$ (normalizzata).

Un esempio d'uso di tale funzione è dato dal seguente programma che esegue la somma di un certo numero di frazioni:

```
/* test-add.c */

#include <stdio.h>
#include "rational.h"

int main(int argc, char *argv[])
{
    llrational S = LLRATIONAL_ZERO;

    for (int i = 1; i < argc; i++) {
        llrational a;
        if (sscanf(argv[i], "%lld/%lld", &a.num, &a.den) < 2 || !risvalidll(a)) {
            fprintf(stderr, "test-add: invalid fraction: %s\n", argv[i]);
            return 1;
        }
        rnormll(&a);
        S = raddll(S,a);
    }

    printf("The sum is %lld/%lld.\n", S.num, S.den);

    return 0;
}
```

Il programma può essere eseguito con il comando

```
test-add [r1] [r2]... [rn]
```

e il risultato è la somma $r_1 + r_2 + \ldots + r_n$. Per esempio,

```
>test-add 3/7 -5/4 2/3
The sum is -13/84.
```

6.4 Differenza

```
rational rsub(rational a, rational b)
lrational rsubl(lrational a, lrational b)
llrational rsubll(llrational a, llrational b)
```

Queste funzioni ritornano la frazione $a - b$ (normalizzata).

6.5 Prodotto

```
rational rmul(rational a, rational b)
lrational rmull(lrational a, lrational b)
llrational rmulll(llrational a, llrational b)
```

Queste funzioni ritornano la frazione ab (normalizzata).

Un esempio d'uso di tale funzione è dato dal seguente programma che esegue il prodotto di un certo numero di frazioni:

```
/* test-mul.c */

#include <stdio.h>
#include "rational.h"

int main(int argc, char *argv[])
{
    llrational P = LLRATIONAL_ONE;

    for (int i = 1; i < argc; i++) {
        llrational a;
        if (sscanf(argv[i], "%lld/%lld", &a.num, &a.den) < 2 || !risvalidll(a)) {
            fprintf(stderr, "test-mul: invalid fraction: %s\n", argv[i]);
            return 1;
        }
        rnormll(&a);
        P = rmulll(P,a);
    }
```

```
    printf("The product is %lld/%lld.\n", P.num, P.den);

    return 0;
}
```

Il programma può essere eseguito con il comando

```
test-mul [r1] [r2]... [rn]
```

e il risultato è il prodotto $r_1 \cdots r_n$. Per esempio,

```
>test-mul 18/147 -63/12 15/727
The product is -135/10178.
```

6.6 Divisione

```
rational rdiv(rational a, rational b)
lrational rdivl(lrational a, lrational b)
llrational rdivll(llrational a, llrational b)
```

Queste funzioni ritornano la frazione a/b (normalizzata).

6.7 Mediante

```
rational rmediant(rational a, rational b)
lrational rmediantl(lrational a, lrational b)
llrational rmediantll(llrational a, llrational b)
```

Queste funzioni ritornano la mediante (normalizzata) di due frazioni a e b, cioè la frazione che ha per numeratore la somma dei numeratori e per denominatore la somma dei denominatori. Date due frazioni $a = p_1/q_1$ e $b = p_2/q_2$, la mediante è quindi

$$\frac{p_1 + p_2}{q_1 + q_2}$$

Per $p_1/q_1 < p_2/q_2$ si ha

$$\frac{p_1}{q_1} < \frac{p_1 + p_2}{q_1 + q_2} < \frac{p_2}{q_2}$$

6.8 Somma "armonica"

```
rational rharm(rational a, rational b)
lrational rharml(lrational a, lrational b)
llrational rharmll(llrational a, llrational b)
```

Queste funzioni ritornano la frazione

$$\frac{1}{1/a + 1/b}$$

6.9 Valore assoluto

```
rational rabs(rational r)
lrational rabsl(lrational r)
llrational rabsll(llrational r)
```

Queste funzioni ritornano la frazione

$$|r| = \begin{cases} r & r \geq 0 \\ -r & r < 0 \end{cases}$$

7 Sviluppo in frazione continua

Per mezzo dell'algoritmo di Euclide per il massimo comun divisore, qualunque frazione r può essere scritta come frazione continua

$$r = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}$$

dove gli a_i (gli *elementi* della frazione continua) sono coefficienti interi, positivi se r è positiva, negativi viceversa. Lo sviluppo è finito e l'indice n dell'ultimo elemento che compare nella frazione è detto *ordine* della frazione continua. La scrittura sopra può essere compattata in

$$r = [a_0; a_1, a_2, \ldots, a_n]$$

Le funzioni

```
rational rcfrac(rational r, int *a, size_t *order)
lrational rcfracl(lrational r, long *a, size_t *order)
llrational rcfracll(llrational r, long long *a, size_t *order)
```

sviluppano la frazione r in frazione continua. I coefficienti vengono memorizzati nel vettore a, la cui dimensione deve essere almeno (*order+1). La variabile order è un puntatore a una variabile di tipo size_t che al momento della chiamata deve contenere il massimo ordine a cui si vuole arrivare. Al ritorno, *order contiene l'effettivo ordine della frazione continua: a[*order] conterrà quindi il valore dell'ultimo elemento dello sviluppo.

Il valore di ritorno delle funzioni è il *resto* dello sviluppo ed è diverso da RATIONAL_ZERO (o LRATIONAL_ZERO o LLRATIONAL_ZERO, per le funzioni con suffisso l ed ll) solo se il valore specificato per *order al momento della chiamata non è sufficiente per contenere tutto lo sviluppo. In tal caso si ha

$$r = [a_0; a_1, \ldots, a_n, r_{n+1}] = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_n + r_{n+1}}}}} \qquad (*)$$

Nota: In matematica il resto è definito in modo che si abbia

$$r = [a_0; a_1, \ldots, a_n, r_{n+1}] = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_n + \cfrac{1}{r_{n+1}}}}}}$$

ma qui ho preferito fare sì che il valore di ritorno fosse in accordo con la (*), in modo da ottenere come resto sempre una frazione valida.

Le funzioni sopra possono essere utilizzate anche per rappresentare le frazioni come frazioni miste, cioè come la somma di un numero intero e di una frazione propria. Per esempio:

```
size_t order = 0;
int ip;
rational fp = rcfrac(r, &ip, &order);
```

Un programma d'esempio per la funzione rfrac è il seguente:

```
/* test-cfrac.c */

#include <stdio.h>
#include "rational.h"

#define MAX_ORDER 100

int main(int argc, char *argv[])
{
    int a[MAX_ORDER+1];
    size_t order;
    rational rem;

    for (int i = 1; i < argc; i++) {
        rational r;
        if (sscanf(argv[i], "%d/%d", &r.num, &r.den) < 2 || !risvalid(r)) {
            fprintf(stderr, "test-cfrac: invalid fraction: %s\n", argv[i]);
            return 1;
        }
        rnorm(&r);

        order = MAX_ORDER;
        rem = rcfrac(r, a, &order);
        printf("%d/%d = [%d", r.num, r.den, a[0]);
        for (int i = 1; i <= order; i++) {
            printf("%c%d", (i == 1) ? ';' : ',', a[i]);
        }
        if (rcomp(rem,RATIONAL_ZERO) != 0) /* If there is a remainder, print it! */
            printf(",%d/%d", rem.num, rem.den);
        printf("]\n");

    }
```

```
    return 0;
}
```

Si esegue con la linea di comando

```
test-cfrac [r1] [r2]...
```

e per ogni frazione inserita riporta la frazione continua associata. Per esempio,

```
>test-cfrac 5/17 -59/7
5/17 = [0;3,2,2]
-59/7 = [-8;-2,-3]
```

8 Comparare le frazioni

Devo ammettere che scrivere una funzione che permettesse di comparare due frazioni in modo affidabile è stato più "tricky" del previsto. Il requisito critico è che per ogni variabile r di tipo rational si deve avere

```
RATIONAL_MIN <= r <= RATIONAL_MAX
```

e analogamente per le variabili di tipo lrational e llrational.

In matematica, la relazione d'ordine < sui razionali è definita a partire da quella degli interi come

$$\frac{a}{b} < \frac{c}{d} \Leftrightarrow ad < bc \Leftrightarrow ad - bc < 0$$

dove si suppone che le frazioni siano normalizzate con $b > 0$ e $d > 0$. Con l'aritmetica finita, però, a causa dell'overflow, l'espressione $ad - bc$ non viene valutata correttamente per tutte le possibili coppie a/b e c/d e il requisito specificato all'inizio non può essere soddisfatto. Insomma, se si vuole comparare due frazioni in modo corretto con l'aritmetica finita non si può pensare di fare moltiplicazioni, ma solo divisioni. L'idea è allora quella di utilizzare lo sviluppo in frazioni continue. Date due frazioni *positive*

$$r_1 = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}$$

$$r_2 = b_0 + \cfrac{1}{b_1 + \cfrac{1}{b_2 + \cfrac{1}{\ddots + \cfrac{1}{b_m}}}}$$

si ha che:

1) Se $a_k < b_k$ per un $k \leq \min\{n, m\}$, allora $r_1 < r_2$, se k è pari (con zero considerato pari), o $r_2 < r_1$, se k è dispari.

2) Se $a_k = b_k$ per ogni $k \leq \min\{n, m\}$, allora: i) $r_1 = r_2$, se $m = m$; e ii) $r_1 < r_2$ se $n < m$ ed n è pari o se $n > m$ ed n è dispari.

Se $r_1 < 0$ e $r_2 < 0$, le disuguaglianze sopra vanno invertite.

Le funzioni

```
int rcomp(rational r1, rational r2);
int rcompl(lrational r1, lrational r2);
int rcompll(llrational r1, llrational r2);
```

implementano la comparazione secondo quanto detto sopra. In particolare, ritornano -1 se r1 < r2, 0 se r1 == r2 e 1 se r1 > r2.

Un programma d'esempio è il seguente:

```
/* test-comp.c */

#include <stdio.h>
#include <stdlib.h>
#include "rational.h"

int comp(const void *ap, const void *bp)
{
    return rcomp(*((rational *)ap), *((rational *)bp));
```

```
}

int main(int argc, char *argv[])
{
    if (argc == 1)
        return 0;

    rational r[argc-1];

    for (int i = 0; i < argc-1; i++) {
        if (sscanf(argv[i+1], "%d/%d", &r[i].num, &r[i].den) < 2 || !risvalid(r[i])) {
            fprintf(stderr, "test-comp: invalid fraction: %s\n", argv[i+1]);
            return 1;
        }
        rnorm(&r[i]);
    }

    /* Sorts the array of rationals with qsort */
    qsort(r, argc-1, sizeof(rational), comp);

    for (int i = 0; i < argc-1; i++)
        printf("%d/%d\n", r[i].num, r[i].den);

    return 0;
}
```

Eseguito con

```
test-comp r1 [r2] [r3]... [rn]
```

ordina le frazioni in ordine crescente. Per esempio

```
>test-comp 1/3 5/6 -5/12 27/8 -55/3
-55/3
-5/12
1/3
5/6
27/8
```

9 Altre funzioni

Le funzioni

```
int gcd(int m, int n)
long gcdl(long m, long n)
long long gcdll(long long m, long long n)
```

ritornano il massimo comun divisore di m ed n, determinato con l'algoritmo Euclideo.

Le funzioni

```
int lcm(int m, int n)
long lcml(long m, long n)
long long lcmll(long long m, long long n)
```

ritornano invece il minimo comune multiplo di m ed n.

11 Compilazione e conclusioni

Nel pacchetto rational.zip allegato all'articolo si trovano due makefile per generare automaticamente la libreria: uno per Windows (rational.mak.windows) e uno per linux (rational.mak.unix). Sono adatti per chi usa gcc, ma non è difficile modificarli per altri compilatori (ho provato solo quello per Windows). Sotto Windows, se si dispone di make.exe, si può scrivere

```
make --makefile rational.mak.windows
```

la libreria generata è librational.a.

In rational.zip si trovano anche i file di esempio dati sopra. Una volta generata la libreria, volendo compilarne uno, sempre con gcc, si può scrivere:

```
gcc -std=c99 -O2 test-add.c librational.a -otest-add.exe
```

Bene, fine del lungo articolo: qui sotto trovate il link dei file delle libreria, se vi dovesse capitare di usarla e doveste rilevare dei bug, fatemelo sapere!

Download rational.zip