INFO1056
Aula 07/08
Aritmética e Álgebra

Prof. João Comba

Baseado no Livro Programming Challenges

High Precision Integers

$$2^{63} = 9{,}223{,}372{,}036{,}854{,}775{,}808$$

- *Arrays of Digits* — The easiest representation for long integers is as an array of digits, where the initial element of the array represents the least significant digit. Maintaining a counter with the length of the number in digits can aid efficiency by minimizing operations which don't affect the outcome.
- *Linked Lists of Digits* — Dynamic structures are necessary if we are *really* going to do arbitrary precision arithmetic, i.e., if there is no upper bound on the length

Big Numbers

```
#define MAXDIGITS 100   /* maximum length bignum */
#define PLUS    1   /* positive sign bit */
#define MINUS   -1    /* negative sign bit */
typedef struct {
  char digits[MAXDIGITS];/* represent the number */
  int signbit;     /* 1 if positive, -1 if negative */
  int lastdigit;     /* index of high-order digit */
} bignum;

print_bignum(bignum *n) {
  int i;

  if (n->signbit == MINUS) printf("- ");
  for (i=n->lastdigit; i>=0; i--)
    printf("%c",'0'+ n->digits[i]);

  printf("\n");
}
```

Big Numbers

```
compare_bignum(bignum *a, bignum *b) {
  int i;        /* counter */

  if ((a->signbit == MINUS) && (b->signbit == PLUS)) return(PLUS);
  if ((a->signbit == PLUS) && (b->signbit == MINUS)) return(MINUS);

  if (b->lastdigit > a->lastdigit) return (PLUS * a->signbit);
  if (a->lastdigit > b->lastdigit) return (MINUS * a->signbit);

  for (i = a->lastdigit; i>=0; i--) {
    if (a->digits[i] > b->digits[i]) return(MINUS * a->signbit);
    if (b->digits[i] > a->digits[i]) return(PLUS * a->signbit);
  }

  return(0);
}
```

Big Numbers

```
zero_justify(bignum *n) {
  while ((n->lastdigit > 0) && (n->digits[ n->lastdigit ] == 0))
    n->lastdigit --;

  if ((n->lastdigit == 0) && (n->digits[0] == 0))
     n->signbit = PLUS; /* hack to avoid -0 */
}

digit_shift(bignum *n, int d)    /* multiply n by 10^d */ {
  int i;        /* counter */

  if ((n->lastdigit == 0) && (n->digits[0] == 0)) return;
  for (i=n->lastdigit; i>=0; i--)
    n->digits[i+d] = n->digits[i];
  for (i=0; i<d; i++) n->digits[i] = 0;
   n->lastdigit = n->lastdigit + d;
}
```

Big Numbers

```
add_bignum(bignum *a, bignum *b, bignum *c) {
   int carry;       /* carry digit */
   int i;           /* counter */

   initialize_bignum(c);

   if (a->signbit == b->signbit) c->signbit = a->signbit;
   else {
      if (a->signbit == MINUS) {
         a->signbit = PLUS;
         subtract_bignum(b,a,c);
         a->signbit = MINUS;
      } else {
         b->signbit = PLUS;
         subtract_bignum(a,b,c);
         b->signbit = MINUS;
      }
      return;
   }

   c->lastdigit = max(a->lastdigit,b->lastdigit)+1;
   carry = 0;

   for (i=0; i<=(c->lastdigit); i++) {
      c->digits[i] = (char) (carry+a->digits[i]+b->digits[i]) % 10;
      carry = (carry + a->digits[i] + b->digits[i]) / 10;
   }
   zero_justify(c);
}
```

Big Numbers

```
subtract_bignum(bignum *a, bignum *b, bignum *c) {
   int borrow;         /* has anything been borrowed? */
   int v;           /* placeholder digit */
   int i;           /* counter */

   if ((a->signbit == MINUS) || (b->signbit == MINUS)) {
              b->signbit = -1 * b->signbit;
              add_bignum(a,b,c);
              b->signbit = -1 * b->signbit;
      return;
    }

   if (compare_bignum(a,b) == PLUS) {
      subtract_bignum(b,a,c);
      c->signbit = MINUS;
      return;
   }

   c->lastdigit = max(a->lastdigit,b->lastdigit);
   borrow = 0;

   for (i=0; i<=(c->lastdigit); i++) {
      v = (a->digits[i] - borrow - b->digits[i]);
      if (a->digits[i] > 0)
        borrow = 0;
      if (v < 0) {
         v = v + 10;
         borrow = 1;
      }

      c->digits[i] = (char) v % 10;
   }
   zero_justify(c);
}
```

Big Numbers

```
multiply_bignum(bignum *a, bignum *b, bignum *c) {
  bignum row;     /* represent shifted row */
  bignum tmp;     /* placeholder bignum */
  int i,j;      /* counters */

  initialize_bignum(c);

  row = *a;

  for (i=0; i<=b->lastdigit; i++) {
     for (j=1; j<=b->digits[i]; j++) {
        add_bignum(c,&row,&tmp);
        *c = tmp;
      }
     digit_shift(&row,1);
  }

  c->signbit = a->signbit * b->signbit;
  zero_justify(c);
}
```

Big Numbers

```
divide_bignum(bignum *a, bignum *b, bignum *c) {
    bignum row;            /* represent shifted row */
    bignum tmp;            /* placeholder bignum */
    int asign, bsign;      /* temporary signs */
    int i,j;               /* counters */

   initialize_bignum(c);

    c->signbit = a->signbit * b->signbit;
    asign = a->signbit;
    bsign = b->signbit;
    a->signbit = PLUS;
    b->signbit = PLUS;
    initialize_bignum(&row);
    initialize_bignum(&tmp);

    c->lastdigit = a->lastdigit;

    for (i=a->lastdigit; i>=0; i--) {
        digit_shift(&row,1);
        row.digits[0] = a->digits[i];
        c->digits[i] = 0;
        while (compare_bignum(&row,b) != PLUS) {
            c->digits[i] ++;
            subtract_bignum(&row,b,&tmp);
            row = tmp;
        }
    }
    zero_justify(c);
    a->signbit = asign;
    b->signbit = bsign;
}
```

Bases Numéricas

- *Binary* — Base-2 numbers are made up of the digits 0 and 1. They provide the integer representation used within computers, because these digits map naturally to on/off or high/low states.
- *Octal* — Base-8 numbers are useful as a shorthand to make it easier to read binary numbers, since the bits can be read off from the right in groups of three. Thus $10111001_2 = 371_8 = 249_{10}$. They also play a role in the only base-conversion joke ever written. Why do programmers think Christmas is Halloween? Because 31 Oct = 25 Dec!
- *Decimal* — We use base-10 numbers because we learned to count on our ten fingers. The ancient Mayan people used a base-20 number system, presumably because they counted on both fingers and toes.
- *Hexadecimal* — Base-16 numbers are an even easier shorthand to represent binary numbers, once you get over the fact that the digits representing 10 through 15 are "A" to ''F."
- *Alphanumeric* — Occasionally, one sees even higher numerical bases. Base-36 numbers are the highest you can represent using the 10 numerical digits with the 26 letters of the alphabet. Any integer can be represented in base-X provided you can display X different symbols.

Conversões

Convert x_a to y_b

- *Left to Right* — Here, we find the most-significant digit of y first. It is the integer d_l such that

$$(d_l + 1)b^k > x \geq d_l b^k$$

where $1 \leq d_l \leq b - 1$. In principle, this can be found by trial and error, although you must to be able to compare the magnitude of numbers in different bases. This is analogous to the long-division algorithm described above.

- *Right to Left* — Here, we find the least-significant digit of y first. This is the remainder of x divided by b. Remainders are exactly what is computed when doing modular arithmetic in Section 7.3. The cute thing is that we can compute the remainder of x on a digit-by-digit basis, making it easy to work with large integers.

Big Numbers

```
int_to_bignum(int s, bignum *n) {
  int i;        /* counter */
  int t;        /* int to work with */

  if (s >= 0) n->signbit = PLUS;
  else n->signbit = MINUS;

  for (i=0; i<MAXDIGITS; i++) n->digits[i] = (char) 0;

  n->lastdigit = -1;
  t = abs(s);

  while (t > 0) {
    n->lastdigit ++;
    n->digits[ n->lastdigit ] = (t % 10);
    t = t / 10;
  }

  if (s == 0) n->lastdigit = 0;
}

initialize_bignum(bignum *n) {
  int_to_bignum(0,n);
}
```

Números

- *Integers* — These are the counting numbers, $-\infty, \ldots, -2, -1, 0, 1, 2, \ldots, \infty$. Subsets of the integers include the *natural* numbers (integers starting from 0) and the *positive* integers (those starting from 1), although the notation is not universal.
  A limiting aspect of integers is that there are gaps between them. An April Fool's Day edition of a newspaper once had a headline announcing, "Scientists Discover New Number Between 6 and 7." This is funny because while there is *always* a rational number between any two rationals x and y ($(x+y)/2$ is a good example), it would indeed be newsworthy if they found an *integer* between 6 and 7.

- *Rational Numbers* — These are the numbers which can be expressed as the ratio of two integers, i.e. c is rational if $c = a/b$ for integers a and b. Every integer can be represented by a rational, namely, $c/1$. The rational numbers are synonymous with fractions, provided we include *improper* fractions a/b where $a > b$.

- *Irrational Numbers* — There are many interesting numbers which are not rational numbers. Examples include $\pi = 3.1415926\ldots$, $\sqrt{2} = 1.41421\ldots$, and $e = 2.71828\ldots$. It can be proven that there does not exist any pair of integers x and y such that x/y equals any of these numbers.

  So how can you represent them on a computer? If you *really* need the values to arbitrary precision, they can be computed using Taylor series expansions. But for all practical purposes it suffices to approximate them using the ten digits or so.

Números

Floating-Point Numbers

There is an IEEE standard for floating point arithmetic which an increasing number of vendors adhere to, but you must always expect trouble on computations which require very high precision. Floating point numbers are represented in scientific notation, i.e., $a \times 2^c$, with a limited number of bits assigned to both the *mantissa* a and *exponent* c. Operating on two numbers with vastly different exponents often results in overflow or underflow errors, since the mantissa does not have enough bits to accommodate the answer.

Round x Trunc

Many problems will ask you to display an answer to a given number of digits of precision to the right of the decimal point. Here we must distinguish between *rounding* and *truncating*. Truncation is exemplified by the `floor` function, which converts a real number of an integer by chopping off the fractional part. Rounding is used to get a more accurate value for the least significant digit. To round a number X to k decimal digits, use the formula

$$\text{round}(X, k) = \text{floor}(10^k X + (1/2))/10^k$$

Números

Rational Numbers

Exact rational numbers x/y are best represented by pairs of integers x, y, where x is the *numerator* and y is the *denominator* of the fraction.

The basic arithmetic operations on rationals $c = x_1/y_1$ and $d = x_2/y_2$ are easy to program:

- *Addition* — We must find a common denominator before adding fractions, so

$$c + d = \frac{x_1 y_2 + x_2 y_1}{y_1 y_2}$$

- *Subtraction* — Same as addition, since $c - d = c + -1 \times d$, so

$$c - d = \frac{x_1 y_2 - x_2 y_1}{y_1 y_2}$$

- *Multiplication* — Since multiplication is repeated addition, it is easily shown that

$$c \times d = \frac{x_1 x_2}{y_1 y_2}$$

- *Division* — To divide fractions you multiply by the *reciprocal* of the denominator, so

$$c/d = \frac{x_1}{y_1} \times \frac{y_2}{x_2} = \frac{x_1 y_2}{x_2 y_1}$$

Álgebra

Manipulating Polynomial

- *Evaluation* — Computing $P(x)$ for some given x can easily be done by brute force, namely, computing each term $c_i x^n$ independently and adding them together. The trouble is that this will cost $O(n^2)$ multiplications where $O(n)$ suffice. The secret is to note that $x^i = x^{i-1}x$, so if we compute the terms from smallest degree to highest degree we can keep track of the current power of x, and get away with two multiplications per term ($x^{i-1} \times x$, and then $c_i \times x^i$).

  Alternately, one can employ *Horner's rule*, an even slicker way to do the same job:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0 = ((a_n x + a_{n-1})x + \ldots)x + a_0$$

- *Addition/Subtraction* — Adding and subtracting polynomials is even easier than the same operations on long integers, since there is no borrowing or carrying. Simply add or subtract the coefficients of the ith terms for all i from zero to the maximum degree.

Álgebra

Manipulating Polynomial

- *Multiplication* — The product of polynomials $P(x)$ and $Q(x)$ is the sum of the product of every pair of terms, where each term comes from a different polynomial:

$$P(x) \times Q(x) = \sum_{i=0}^{degree(P)} \sum_{j=0}^{degree(Q)} (c_i c_j) x^{i+j}$$

  Such an all-against-all operation is called a *convolution*. Other convolutions in this book include integer multiplication (all digits against all digits) and string matching (all possible positions of the pattern string against all possible text positions). There is an amazing algorithm (the fast Fourier transform, or FFT) which computes convolutions in $O(n \log n)$ time instead of $O(n^2)$, but it is well beyond the scope of this book. Still, it is nice to recognize when you are doing a convolution so you know that such tools exist.

- *Division* — Dividing polynomials is a tricky business, since the polynomials are not closed under division. Note that $1/x$ may or may not be thought of as a polynomial, since it is x^{-1}, but $2x/(x^2+1)$ certainly isn't. It is a *rational function*.

Álgebra

Root Finding

Given a polynomial $P(x)$ and a target number t, the problem of *root finding* is identifying any or all x such that $P(x) = t$.

If $P(x)$ is a first-degree polynomial, the root is simply $x = (t - a_0)/a_1$, where a_i is the coefficient of x_i in $P(x)$. If $P(x)$ is a second-degree polynomial, then the *quadratic equation* applies:

$$x = \frac{-a_1 \pm \sqrt{{a_1}^2 - 4a_2(a_0 - t)}}{2a_2}$$

Álgebra

Root Finding

Given a polynomial $P(x)$ and a target number t, the problem of *root finding* is identifying any or all x such that $P(x) = t$.

If $P(x)$ is a first-degree polynomial, the root is simply $x = (t - a_0)/a_1$, where a_i is the coefficient of x_i in $P(x)$. If $P(x)$ is a second-degree polynomial, then the *quadratic equation* applies:

$$x = \frac{-a_1 \pm \sqrt{{a_1}^2 - 4a_2(a_0 - t)}}{2a_2}$$

Beyond quadratic equations, numerical methods are typically used. Any text on numerical analysis will describe a variety of root-finding algorithms, including Newton's method and Newton-Raphson, as well as many potential traps such as numerical stability. But the basic idea is that of binary search. Suppose a function $f(x)$ is *monotonically increasing* between l and u, meaning that $f(i) \leq f(j)$ for all $l \leq i \leq j \leq u$. Now suppose we want to find the x such that $f(x) = t$. We can compare $f((l + u)/2)$ with t. If $t < f((l + u)/2)$, then the root lies between l and $(l + u)/2$; if not, it lies between $(l+u)/2$ and u. We can keep recurring until the window is narrow enough for our taste.

This method can be used to compute square roots because this is equivalent to solving $x^2 = t$ between 1 and t for all $t \geq 1$. However, a simpler method to find the ith root of t uses exponential functions and logarithms to compute $t^{1/i}$.

Álgebra

Logarithms

$$b^x = y \qquad x = \log_b y.$$

$$\ln x \qquad e = 2.71828\ldots \qquad \exp(\ln x) = x$$

$$\log_a n^b = b \cdot \log_a n$$

$$a^b = \exp(\ln(a^b)) = \exp(b \ln a)$$

$$\log_a b = \frac{\log_c b}{\log_c a}$$

Exemplo

The *Stern-Brocot tree* is a beautiful way for constructing the set of all non-negative fractions $\frac{m}{n}$ where m and n are relatively prime. The idea is to start with two fractions $\left(\frac{0}{1}, \frac{1}{0}\right)$ and then repeat the following operation as many times as desired:

Insert $\frac{m+m'}{n+n'}$ between two adjacent fractions $\frac{m}{n}$ and $\frac{m'}{n'}$.

For example, the first step gives us one new entry between $\frac{0}{1}$ and $\frac{1}{0}$,

$$\frac{0}{1}, \frac{1}{1}, \frac{1}{0}$$

and the next gives two more:

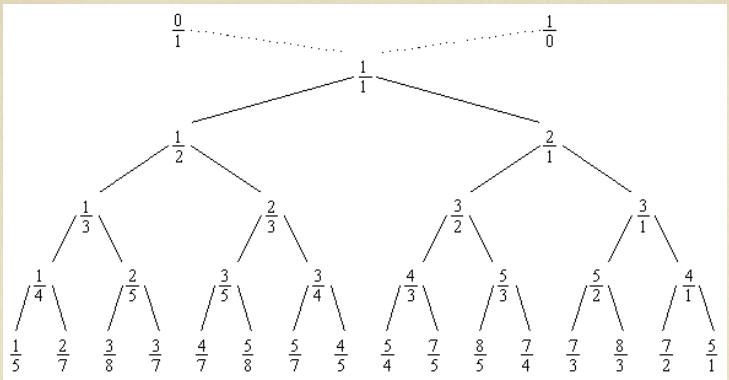
$$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{1}{0}$$

The next gives four more:

$$\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}, \frac{3}{2}, \frac{2}{1}, \frac{3}{1}, \frac{1}{0}$$

The entire array can be regarded as an infinite binary tree structure whose top levels look like this–

Exemplo



This construction preserves order, and thus we cannot possibly get the same fraction in two different places.

We can, in fact, regard the *Stern-Brocot tree* as a *number system* for representing rational numbers, because each positive, reduced fraction occurs exactly once. Let us use the letters "`L`" and "`R`" to stand for going down the left or right branch as we proceed from the root of the tree to a particular fraction; then a string of `L`'s and `R`'s uniquely identifies a place in the tree. For example, `LRRL` means that we go left from $\frac{1}{1}$ down to $\frac{1}{2}$, then right to $\frac{2}{3}$, then right to $\frac{3}{4}$, then left to $\frac{5}{7}$. We can consider LRRL to be a representation of $\frac{5}{7}$. Every positive fraction gets represented in this way as a unique string of `L`'s and `R`'s.

Exemplo

Well, almost every fraction. The fraction $\frac{1}{1}$ corresponds to the empty string. We will denote it by I, since that looks something like 1 and stands for "identity."

In this problem, given a positive rational fraction, represent it in the *Stern-Brocot number system*.

Input

The input file contains multiple test cases. Each test case consists of a line containing two positive integers m and n, where m and n are relatively prime. The input terminates with a test case containing two 1's for m and n, and this case must not be processed.

Output

For each test case in the input file, output a line containing the representation of the given fraction in the *Stern-Brocot number system*.

Sample Input

```
5 7
878 323
1 1
```

Sample Output

```
LRRL
RRLRRLRLLLLRLRRR
```

```cpp
#include <cstdlib>
#include <iostream>
using namespace std;
void calcula(int n, int m){
    int ln=0;
    int lm=1;
    int mn=1;
    int mm=1;
    int rn=1;
    int rm=0;
    double f, mf;
    f=float(n)/float(m);

    while(mn!=n || mm!=m){
        mf=float(mn)/float(mm);
        if(f<mf){//pega esquerda (L)
            rn=mn;
            rm=mm;
            mn+=ln;
            mm+=lm;
            cout << "L";
        }else{//pega direita (R)
            ln=mn;
            lm=mm;
            mn+=rn;
            mm+=rm;
            cout << "R";
        }
    }
    cout << endl;
}

int main(int argc, char *argv[]) {
    int n, m;
    while(cin >> n >> m){
        if(n==1 && m==1) break;
        calcula(n,m);
    }
    return 0;
}
```