Programming Challenges

Week 3 - Arithmetic and Algebra

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Sciences

2015-04-27

Last updated April 27, 2015

Notes: Vito's Family (1)

- Some people had trouble, they tried to calculate the "minimal path"
- The trick for Vito's Family is that the "minimal path" is the median!
- Calculating the median, you could solve Vito's Family in 10 lines.

Think carefully about the problem before beginning to code.

Notes: Vito's Family (2)

```
a,r['~~'];main(b,c,d,e,f,s){
for(scanf("%d",&b);b--;a=!printf("%d\n",a))
{for(f=!scanf("%d",&s);f<s;)scanf("%d",r+f++);
for(a=1<<29;f--;a=a>d?d:a)for(d=c=0;c<s;d+=e>0?e:-e
e=r[c++]-r[f];}}
```

Notes: Vito's Family (2)

```
a,r['~~'];main(b,c,d,e,f,s){
for(scanf("%d",&b);b--;a=!printf("%d\n",a))
{for(f=!scanf("%d",&s);f<s;)scanf("%d",r+f++);
for(a=1<<29;f--;a=a>d?d:a)for(d=c=0;c<s;d+=e>0?e:-e
e=r[c++]-r[f];}}
```

http://www.ioccc.org/

Notes: Code Comments

Remember that adding comments to your code will increase your grade. Make sure to describe what is the idea for each program that you solve.

```
/* Vito's Family: The minimum distance can be
 * easily calculated from the Median of
 * the addresses
 */
```

Class Outline

Two things are quite hard when dealing with numbers on a computer: Very big numbers, and very small numbers.

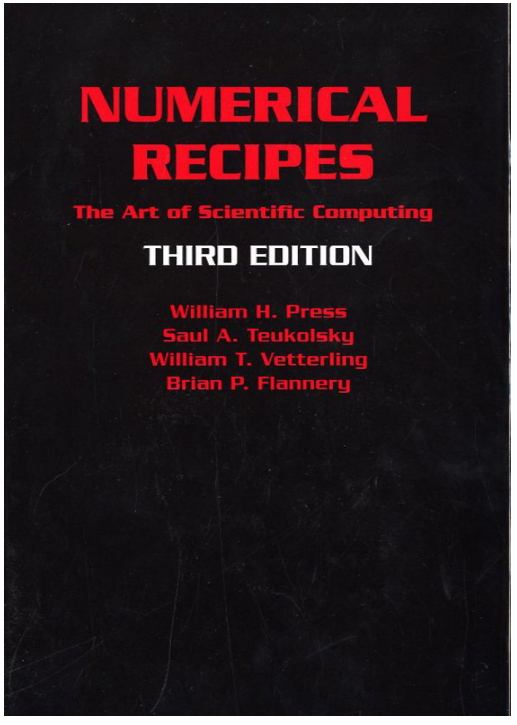
- Usually, numbers in computers are represented as variables with fixed sizes (2 bytes, 4 bytes, etc).
- This means that there is a limit to which numbers a *int* or a *float* can store.
- Mathematics does not care for these limits, which can result in *overflow* or *underflow* problems.
- Writing functions to deal with large (or small) numbers can be a hard but interesting problem for programming challenges.

The Pentium FDIV bug

In 1994, Intel's Pentium CPU had a bug in that returned wrong results in floating point divisions. The recall cost them 475 million dollars.

Book Suggestion

- The book Numerical Recipes has many algorithms and programming examples for arithmetic operations. Worth checking it out!



Big numbers in computers

When we talk about Big Numbers, we usually mean large integers. Programming languages usually have primitive variables with a fixed number of bits. What is the biggest number that we can represent using this amount of bits?

- 32 bits: 2.147.483.648
- 64 bits: 9.223.372.036.854.775.808
- 128 bits: $1.701331 * 10^{38}$

How big is big enough?

- 32 bits: 2.147.483.648
- 64 bits: 9.223.372.036.854.775.808
- 128 bits: $1.701331 * 10^{38}$

Where can we use these?

- Speed of Light: 300.000 m/s;

How big is big enough?

- 32 bits: 2.147.483.648
- 64 bits: 9.223.372.036.854.775.808
- 128 bits: $1.701331 * 10^{38}$

Where can we use these?

- Speed of Light: 300.000 m/s;
- Closest star system (5 light years): 47.304.000.000.000 km;

How big is big enough?

- 32 bits: 2.147.483.648
- 64 bits: 9.223.372.036.854.775.808
- 128 bits: $1.701331 * 10^{38}$

Where can we use these?

- Speed of Light: 300.000 m/s;
- Closest star system (5 light years): 47.304.000.000.000 km;
- Number of atoms in the universe: $1 * 10^{80} atoms$;

How big is big enough?

- 32 bits: 2.147.483.648
- 64 bits: 9.223.372.036.854.775.808
- 128 bits: $1.701331 * 10^{38}$

Where can we use these?

- Speed of Light: 300.000 m/s;
- Closest star system (5 light years): 47.304.000.000.000 km;
- Number of atoms in the universe: $1 * 10^{80} atoms$;
- Digits of pi: more than $1 * 10^{100.000.000}$; damn mathematicians!

Size and Precision

By using the correct representation, we can store numbers of any size in a fixed length integer:

- 47.304.000.000.000 km -> 4.7304+*e*13 km

However, our precision is limited:

- 47.304.000.000.000 km -> 4.7304+*e*13 km
- 47.304.000.013.200 km -> 4.7304+*e*13 km
- 47.304.005.037.001 km -> 4.7304+*e*13 km

Don’t forget your libraries!

- C++: *Integer* class
- Java: *java.math.BigInteger* class

Many programming languages provide support for big intergers “Out of the Box”. Sometimes it may be necessary to make your implementation of big integers, but keep these classes in mind!

How to represent big Integers (1)

Array of Digits

- Very simple to implement;
- First element of the array is the least significant digit;
- Keep the index of the last digit at hand;

Linked List of Digits

- Since each item is just a digit, you occupy double the space with pointers.
- If you have few big numbers, but many small ones, this saves some memory.

How to represent big Integers (2)

What base should we use?

- A decimal base sounds natural, but not required!
- A higher base means we need fewer digits (smaller arrays)
  - Example: Octal base (base 8), Hexadecimal base (base 16)
- On the other hand, you need extra functions to transform the numbers;

Pay attention to the sign!

- You will need to keep an explicit field for the number's sign;
- This creates a special case: -0 and +0 are different!
- Don't forget to transform -0 numbers into +0 in your operators;

BigNum Operations

Traditional arithmetic operations in big numbers are not different from what we know. However, there are a number of caveats that occur because:

- We are operating on one digit at a time;
- We are explicitly dealing with the sign;
- We want to be efficient;

Addition (a,b)

Start the addition digit by digit, starting from the least significant. Any overflow should be added as a carry to the next digit

Caveats

- If *a* and *b* have different signs, it becomes a subtraction. Switch the sign of the parameter and send it to the subtraction function!
- Don't forget that the carry may add an extra digit to the final result!

Subtracting (borrow method)

Subtract each digit, starting from the least significant. If the result is negative, add *base* and increment a *borrow* counter. This counter is subtracted from the next digit.

Caveats

- Order the parameters by their most significant digit, to avoid special cases when borrowing;
- Ordering the parameters also avoids having to decide what is the final sign;
- Don't forget to adjust zeroes, and deal with -0;

Subtracting (complement method)

We can use the complement of the base to make subtraction using only additions.
This makes more sense in the machine code level, because the complement of base 2 has many useful properties. But it can also be done at higher bases.

caveats

- You need to either order the numbers, to always subtract the smaller from the larger;
- Or you need to check for carry, to complement the result again in case it underflows;
- Don't forget to add 1 in the regular case!

Multiplication

A simple way to implement multiplication is to use the traditional method that we learn at school:

```
  1234
   312 x
  ======
  1234*2 +
 12340*1 +
123400*3
========
```

```
  1234 +
  1234 +
 12340 +
123400 +
123400 +
123400
========
```

Multiplication

Schoolhouse Multiplication

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

Fast Multiplication

Complexity

Traditional multiplication on n digits requires $O(n^2)$ elementary operations. There are special algorithms that produce faster results for large numbers.

Karatsuba Method

Breaks a and b into numbers with less digits $(a_0 * b^m + a_1, b_0 * b^m + a_2)$, and multiply these numbers. This is done recursively. Complexity is $O(n^{\log_2 3})$

Schonhage-Strassen Algorithm

Uses Fast-Fourier Transform to get a complexity $O(n \log n \log \log n)$. Only used in practice when the number of *digits* is 30.000 or more.

Division

Division can, in principle, be dealt in a similar way to multiplication. We repeatedly subtract the dividend, and then shift it to the right.

```
450203
    23 %
======
230000*1 -
 23000*9 -
  2300*5 -
   230*7 -
    23*4 -
======
     1    (19574)
```

Exponentiation

There is a neat trick to make fast exponentiation, do you remember it?

Exponentiation

There is a neat trick to make fast exponentiation, do you remember it?

Normally we approach exponentiation in a “divide and conquer” fashion. Cache the partial answers, because they will repeat many times.

- $A^n = A^{floor(n/2)} * A^{floor(n/2)} * A^{(n(mod)2)}$

Real Numbers

Pesky infinity

- In Mathematics, there is always a number between a and b: $\frac{a+b}{2}$:
- In Computers, there is a limit to our precision, so this is not always true;
- In the case of floats, this is often not true!

$$a = (x + y) + z$$
$$b = x + (y + z)$$

$a == b$ may return false, if a and b are floats! :-(

Types of Real Numbers

Rational Numbers

Can be represented by a/b, where a and b are integers. In general it is easier to do all representation and operations directly on these integers.
Note: Every integer c can be represented as $c/1$

Irrational Numbers

Some Real numbers can't be represented as a fraction of two integers. For example, π, e, $\sqrt{2}$, etc. Like with large integer, there is a limit to the precision of these numbers which depends on the number of digits that we can store.

Representing Real Numbers

Floating point numbers in *floats* and *doubles* are represented using the IEEE notation:

$$a * 2^c$$

Where a (the mantissa) and c (the exponent) have a limited number of bits. This implies that making operations on numbers with very different exponents is a recipe for numerical imprecisions.

Warning

Do not directly compare two floats using $==$: there is frequently precision errors. Instead, you want to test if their differences lie whithin a limit ϵ. Or just use integers.

Operations on Rational Numbers

Given two rational numbers, $c = x_1/y_1$ and $d = x_2/y_2$, the basic arithmetic operations are easy to program:

- Addition: $c + d = \frac{x_1 y_2 + x_2 y_1}{y_1 y_2}$
- Subtraction: $c - d = \frac{x_1 y_2 - x_2 y_1}{y_1 y_2}$
- Multiplication: $cd = \frac{x_1 x_2}{y_1 y_2}$
- Division: $c/d = \frac{x_1 y_2}{x_2 y_1}$

Warning!

Repeating all these multiplications leads to a serious danger of overflows! Reduce the Fractions from time to time to avoid this.

Polynomials

It is useful to think of univariate polynomials as a special case of Big Numbers. In this case, the *base* of the big number is the variable, and the digits are the coefficients. (Except that there is no carry)

```
Polynomial(x) = {c0,c1,c2,c3...cn}
```

Operation on Polynomials

Evaluation

Brute force evaluation costs $O(n^2)$ multiplications. We can reduce this to $O(n)$ by using Horner's rule:

$$(((c_n x + c_{n-1})x + c_{n-2})x + ...)x + c_0$$

Addition, Subtraction

These operate in the sabe way as bignum additions and subtractions, but without the carry/borrow.

Operations on Polynomials

Multiplication

The product of $P(x)$ and $Q(x)$ is the combination of all items:

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

This all against all operation is known as convolution. The brute force complexity is $O(n^2)$, but it can be done in $O(n \log n)$ with a FFT. Other convolutions: String matching, digit addition, etc.

Division

Division is not a closed operation for polynomials, and needs to be treated as a special case. For example:

$$P(x) = 2x; Q(x) = x^2 + 1; P(x)/Q(x)$$

Is not a polynomial, but rather a fractional function.

Logarithms and exponential roots

Remember the basic rules of logarithms and exponentiations:

$$\log xy = \log x + \log y$$
$$\log x^y = y \log x$$

We can use these to easily find fractional exponents:

$$a^b = \exp(\ln(a^b)) = \exp(b \ln(a))$$
$$\sqrt{x} = x^{1/2} = \exp(0.5 \ln(x))$$

Just beware of computational uncertainties!

Problems for this week

- Primary Arithmetic
- Reverse and Add
- Polynomial Coefficient
- Stern Brocot Number System