High Precision Arithmetic
Guest Speaker: Stefan Funke

Knuth [Knu81] is a great source on high-precision arithmetic.

1 Arbitrary Precision Integers

We want to do addition, subtraction, multiplication and division on arbitrarily large integers. In C/C++ you usually have access to a data type long int, which can store a 32 bit integer (or 64 for newer machines) and which implements arithmetic modulo $B = 2^{32}$. Thus native integer arithmetic can safely add/subtract 31bit integers and multiply 16bit integers.

How can we represent arbitrarily large integers? Well, we simply take a vector of digits, where a digit is an integer in $[0 \mathrel{..} B-1]$ and represent integers in base B. For example, $X = (x_3, x_2, x_1, x_0)$ with $0 \le x_i < B$, represents $x_3 \cdot 2^{96} + x_2 \cdot 2^{64} + x_1 \cdot 2^{32} + x_0$. With arbitrary length vectors, we can represent arbitrary large integers.

How do we realize the arithmetic operations $+$, $-$, $\times$ and $/$? In elementary school, we learned realizations for base 10 and they carry over to arbitrary base. We assume we have two numbers $X = \sum_i x_i B^i$ and $Y = \sum_i y_i B^i$ and want to compute their sum, difference, product, and integer quotient, respectively.

Addition: Take for example addition of two numbers

```
   12345
+  67890
--------
   80235
```

While doing this, what basic operations did we use?

— Well, we only have to know how to add two digits and possibly a carry of 1. We produce a digit and maybe another carry. For example, we have $4+9=13$, we write down the 3 and keep the 1 to be added to 8 and 3.

Let us first see how to add two digits x and y in $[0 \mathrel{..} B-1]$ and a carry $c \in \{0, 1\}$. The result is a digit $z \in [0 \mathrel{..} B-1]$ and a carry $d \in \{0, 1\}$. We want

$$x + y + c = d \cdot B + z \ .$$

The following C-program does the job. It returns z as the return value and d in c. Observe, that c is a reference-parameter.

```
digit AddDigits(digit x, digit y, int &c)
{ y = y + c;
  if ( y > 0 )
  {  x += y;
```

```
    c = ( x < y );
  }
  return x;
}
```

Let us argue that this is correct. In the argument we carefully have to distinguish between integer addition and addition modulo B. The latter is realized by the machine arithmetic. Assume first that $y + c \pmod B = 0$. This is the case if either $y = c = 0$ or $y = B - 1$ and $c = 1$. In either case, we have $z = x$ and $d = c$. So assume $y' = y + c \pmod B \neq 0$. Then $y' = y + c = y + c \pmod B$ and hence the first line adds y and c over the integers. It remains to add x and y'. If $x + y' < B$, then $x + y' > x$ and hence we set the outgoing carry to zero. If $x + y' \geq B$ and hence $x + y' \pmod B = x + y' - B < x$, we set the carry to one.

With our basic operation it is easy to add two n-digit numbers X and Y, producing an n- or $n+1$- digit result.

```
int c = 0;
for (int i = 0; i < n; i++) z[i] = AddDigits(x[i],y[i],c);
z[n] = c;
```

Subtraction: Let us again start with an example.

```
  54321
-  9876
-------
  44445
```

What basic operation do we need here ? We need to be able to subtract one digit from another taking into account "borrowed" 1s from the previous subtraction and possibly borrowing another 1. If, we use x and y for our digits and b and d for the old and new borrow, respectively, we want

$$x - (y + b) + dB = z \ ,$$

where again z is the result digit. The following procedure does the job.

```
digit SubDigits(digit x, digit y, int& borrow)
{
  y += borrow;
  if ( y ) Borrow = ( x < y );
  return x - y;
}
```

Let us again argue correctness. If $y+b = 0 \pmod B$, we have $z = x$ and $d = b$. So assume $y' = y+b \pmod B \neq 0$. Then $y' = y+b$ and $d = 1$ iff $x < y'$. Also $z = dB+x-y = x-y \pmod B$.

Multiplication: Consider again an example.

```
1234*5678
---------
     0
      4
       8
        2
----------
```

What basic operation do we need here? Since we know already how to add its suffices to be able to compute the product of two single digits. The result is a number which is represented with two digits, i.e.

$$x \cdot y = z_1 B + z_0$$

where x, y, z_0 and z_1 are digits.

How can we multiply two single digits. We exploit the fact that $xy = xy \pmod{B}$ as long as $xy < B$. This will be the case if $x, y \in [0 \mathinner{..} H-1]$ where $B = 2^k$ and $H = 2^{k/2}$. This assumes that k is even. An integer in $[0 \mathinner{..} H-1]$ is sometimes called a half-digit. Half-digits are multiplied correctly by the hardware. We split our digits into two half-digits and obtain:

$$x = x_1 2^{k/2} + x_0 \qquad y = y_1 2^{k/2} + y_0$$

$$x \cdot y = x_1 y_1 \cdot 2^k + (x_1 \cdot y_0 + x_0 \cdot y_1) \cdot 2^{k/2} + x_0 \cdot y_0$$

Native hardware is able to carry out the multiplications. The following code takes care of the additions. The procedure returns z_0 directly and z_1 in *High*.

We use the following notation. For a digit c, $LOW(c)$ produces the low half-digit and $HIGH(c)$ produces the high half-digit.

```
digit MulDigits(digit x, digit B, digit &High)
{
  digit c = x_0*y_0;
  digit L = LOW(c);             // have finished bits 0..15 in L
  c = HIGH(c)+y_0*x_1;
  High = HIGH(c);                 // parts of bits 32..47 in High
  c = LOW(c)+y_1*x_0;          // have finished bits 16..31 in LOW(c)
  L| = LOW(c) << (k/2);                // finish bits 0..31 in L
  High = High + HIGH(c) + y_1*x_1;
  return L;
}
```

Several remarks are in order. We first compute $c = x_0 \cdot y_0$ and write it in base H, say as $c_1 H + c_0$; c_0 is the lower half-digit of z_0. We then compute $d = c_1 + y_0 \cdot x_1 \pmod{B}$. It is crucial to observe that there is no overflow here and hence $c_1 + y_0 \cdot x_1 \pmod{B} = c_1 + y_0 \cdot x_1$.

This follows from $c_1, y_0, x_1 \le 2^{k/2}-1$ and hence $c_1+y_0x_1 \le 2^{k/2}-1+(2^{k/2}-1)(2^{k/2}-1) < 2^k$. We split d into its half-digits and move the higher half-digit into *High*. We add $y_1 \cdot x_0$ to the lower half-digit and store it again in c. The lower-half digit of c is the upper half-digit of z_0 and its upper half-digit is added into *High*. This completes the description of the basic operation.

The standard method for multiplication has running time $O(n^2)$. There are faster methods, e.g., Karatsuba's method (running time $O(n^{\log 3})$) and FFT-based methods (running time $O(n \log^2 n)$. LEDA currently uses Karazuba's method. Sören explores the use of FFT-based methods. You can find a presentation of Karatsuba's method (including experimental results) on KM's web-page (`http://www.mpi-sb.mpg.de/~mehlhorn/Informatik5.html`, section Introduction).

Division Division is the most complex arithmetic operation. We learn it last in elementary school and, in fact, what we learn does not generalize easily to an arbitrary base. What did we learn? We invite the reader to do a few examples at this point.

Assume that the divisor Y is an n-digit number. We slide a window of length $n+1$ over the dividend X. Let W be the number in this window. We then guess the result q of the division W/Y. We verify the guess by doing a multiplication and subtraction. More precisely, if $W < qY$ our guess was too large, and if $W - qY \ge Y$ our guess was too small. When we have guessed correctly, i.e., $0 \le W - qY < Y$, we add q as the next digit to the result and we move the window one to the right. The new content of the window is $(W - qY)B + x$, where x is the next digit of X. Observe that $(W - qY) + x < BY$ (since $W - qY \le Y - 1$ and $x \le B - 1$) and hence the division of the new window content by Y yields indeed a single digit.

In the school method for division the digits of the result is guessed. This is not a serious problem since there are only nine digits and since most of us have developed heuristics which produce fairly good guesses. A popular heuristic is to divide the number formed by the first two digits of W by the first digit of Y and to take the result as a guess for q. We have B distinct digits and therefore it is important to have a heuristic which provably comes close to the true result of the division. We show that the divide-two-digits-by-one-digit heuristics works provided the first digit of Y is large. There is an easy way to make the first digit of Y large. If the first digit of Y is in $[2^{l-1} \mathinner{..} 2^l - 1]$ for integer $l \ge 1$, we multiply X and Y by 2^{k-l}. This does not change the result of the division and moves the first digit of Y into $[B/2 \mathinner{..} B-1]$.

Lemma 1 *Let* $W = \sum_{0 \le i \le n} w_i B^i$, $Y = \sum_{0 \le i < n} y_i B^i$, $y_{n-1} \ge B/2$, $W/Y < B$, $q = W/Y$, *and* $\tilde{q} = (w_n B + w_{n-1})/y_{n-1}$. *Then*

$$\lfloor q \rfloor \le \lfloor \tilde{q} \rfloor < \lfloor q \rfloor + 3 \ .$$

Proof: We first show the lower bound. Let $\bar{q} = \lfloor \tilde{q} \rfloor$. Then

$$w_n \cdot B + w_{n-1} = \bar{q} \cdot y_{n-1} + r$$

where $0 \le r < y_{n-1}$. Now assume, that $\bar{q} < \lfloor q \rfloor$ or in other words $\lfloor q \rfloor \ge \bar{q} + 1 \le \lfloor q \rfloor$. We show that $(\bar{q}+1) \cdot Y > W$, hence giving a contradiction.

$$\begin{aligned}
(\bar{q}+1) \cdot Y &\ge (\bar{q}+1) \cdot (y_{n-1} \cdot B^{n-1}) \\
&= \bar{q} \cdot y_{n-1} \cdot B^{n-1} + y_{n-1} \cdot B^{n-1} \\
&= (w_n \cdot B + w_{n-1} - r) \cdot B^{n-1} + y_{n-1} \cdot B^{n-1} \\
&= w_n \cdot B^n + w_{n-1} \cdot B^{n-1} - r \cdot B^{n-1} + y_{n-1} \cdot B^{n-1} \\
&= w_n \cdot B^n + w_{n-1} \cdot B^{n-1} + B^{n-1} \cdot (y_{n-1} - r) \\
&\ge w_n \cdot B^n + (w_{n-1} + 1) \cdot B^{n-1} > W \ .
\end{aligned}$$

We turn to the upper bound. We have

$$\begin{aligned}
\frac{w_n B + w_{n-1}}{y_{n-1}} &= \frac{w_n B^n + w_{n-1} B^{n-1}}{y_{n-1} B^{n-1}} \\
&\le \frac{W}{Y - B^{n-1}} \\
&\le \frac{W}{Y(1 - 2/B)} \\
&\le \frac{WB}{Y(B-2)} = \frac{W(B-2+2)}{Y(B-2)} \\
&= q + q\frac{2}{B-2} \\
&< q + \frac{2B}{B-2} < q + 3 \ ,
\end{aligned}$$

∎

The computation of $\tilde{q}$ can be varied in several ways. We can get a better estimate by considering the division of three digits by two digits. We get a simpler computation if we simply compute $w_n B / y_{n-1}$. Observe that $(w_n B + w_{n-1})/y_{n-1} = w_n B/y_{n-1} + w_{n-1}/y_{n-1} = w_n B/y_{n-1} + [0 \mathinner{..} 2]$ since $y_{n-1} \ge B/2$.

The algorithm is now as follows. We compute $Z = \lfloor X/Y \rfloor$.

1. If the first digit of Y is less than $B/2$ multiply X and Y by an appropriate power of two. This will not change the number of digits of Y and may increase the number of digits of X by one. We use m for the new number of digits of X and n for the number of digits of Y. If $m < n$, we are done and return zero.

2. Let W be the number formed by the first n digits of X. If $W \ge Y$, set[1] $W = W - Y$ and $Z = 1$. Otherwise, leave W unchanged and set $Z = 0$. Also set $i = m - n$.

[1]Observe that $W < 2Y$ since the first digit of Y is at least $B/2$.

3. (LOOP). We have

$$W < Y \quad \text{and} \quad \sum_{j=i}^{j=n} x_j B^{j-i} = Z \cdot Y + W \ .$$

   (a) If $i = 0$, stop and return Z.

   (b) Our invariant implies

$$\begin{aligned} \sum_{j=i-1}^{j=n} x_j B^{j-i} &= (\sum_{j=i}^{j=n} x_j B^{j-i})B + x_{i-1} \\ &= (Z \cdot Y + W)B + x_{i-1} \\ &= ZB \cdot Y + WB + x_{i-1} \\ &= (ZB + q) \cdot Y + WB + x_{i-1} - qY \ , \end{aligned}$$

   where $q = \lfloor (WB + x_{i-1})/Y \rfloor$. We hence have the following algorithm. Decrement i and set $W = WB + x_i$. Then $W < BY$. Let $q = \lfloor W/Y \rfloor$. In order to find q compute $\tilde{q} = \lfloor (w_n B + w_{n-1})/y_{n-1}) \rfloor$ and try the values in $[\lfloor \tilde{q} \rfloor \, .. \, \lfloor \tilde{q} \rfloor + 3]$. Set $W = W - qY$ and $Z = ZB + q$.

It remains to implement our basic operation, the division of a two-digit integer $T = \mathit{high}B + \mathit{low}$ by a one-digit integer D with $D \geq B/2$. We proceed as follows. We store $\mathit{high}B$ and low in doubles and sum them in S. Then $T - 2^{12} \leq S \leq T$ since the mantissa of a double is only 52 bits long. We compute $Q = \lfloor S/\mathit{double}(D) \rfloor$. Then $T/D - 1 \leq Q \leq T/D$. We use a multiplication to check, whether Q is to be increased by one, i.e., we compute $S - QD$ and if this difference is D or more, we increase Q by one. We want to compute $S - QD$ with 52-bit arithmetic. This should be possible, because the result is at most $2D$. We split Q into its half-digits, i.e., $Q = Q_h 2^{16} + Q_l$, and compute

$$\mathit{diff} = S - QD = H + L - Q_h 2^{16} D - Q_l D = (H - Q_h 2^{16} D) - Q_l D + L \ .$$

The final expression stays within 52-bit arithmetic.

```
digit DivTwoByOne(digit high, digit low, digit D, digit &Q)
{
  double H = high*B;
  double L = low;
  double S = H + L;
  digit Q = floor(S/double(D));
  double Q_high = (double) (HIGH(Q));            // upper half-digit of Q
  double Q_low =  (double) (LOW(Q));             // lower half-digit of Q
  double diff = ((H - Q_high*2^{16}*D) - D*Q_low) + L;
  if (diff >= D)
  {
     diff- = D;
```

```
      Q++;
    }
    return diff;
  }
```

2 Arbitrary Precision Floating Point Numbers

(s, m, e) represents number $(-1)^s * m * 2^e$ m,e are integers of arbitrary length.

How to compute with such numbers ?

Addition/Subtraction

Adjust numbers such that they have the same exponent and do integer addition/subtraction and round afterwards.

Tricks: IF ranges do not intersect, and precision small enough, only use other operand to bias rounding

Multiplication Multiply mantissa and exponent and round afterwards.

Tricks: If required precision about the same as precision of operands, do not compute the full product.

Division

We shift dividend to left as far as necessary for the required precision and then do an integer division of the significants.

Squareroot ...are approximated using Newtonian iteration, so to solve $f(x) = x^2 - a$, we subsequently compute $x_n + 1 = x_n - (x^2 - a)/2 * x_n$, until the required precision is reached.

3 LEDA Reals

With LEDA reals you can compute the sign of expressions involving roots and divisions. The idea is as mentioned at the beginning, you compute approximations of increasing precision until the sign can be determined (either because the approximation interval does not contain zero or it is contained in the separation bound). But to be able to do this, one needs to be able to recalculate the approximation of an expression, so one needs to keep track of the history of an arithmetic expression.

LEDA reals represent an expression as a directed acyclic graph. The leaves are integer constants, and the internal nodes are the operators. For example $E = \sqrt{(5 + 6 * 4)} - 4$

References

[Knu81] D.E. Knuth. *The Art of Computer Programming (Volume II): Seminumerical Algorithms*. Addison-Wesley, 1981.