Длинная арифметика

Неспирный В.Н.

7 августа 2010г.

Как известно, в большинстве языков программирования в переменных целочисленного типа могут храниться значения из довольно ограниченного диапазона. Так в 32-разрядной знаковой переменной могут быть представлены значения не превышающие по абсолютной величине $2^{31} - 1 \approx 2 \cdot 10^9$, в 64-разрядной – до $2^{63} - 1 \approx 9 \cdot 10^{18}$. В то же время в ряде олимпиадных задач и некоторых приложениях приходится работать с целыми числами, которые имеют большее количество знаков, или с вещественными заданными с довольно большой точностью.

Следует отметить, что в некоторых языках (Python, Java и др.) реализована поддержка больших чисел. Однако в тех же Pascal и C++ приходится самостоятельно реализовывать все необходимые операции над числами многократной точности. Работа с такими числами и называется длинной арифметикой.

1 Представление чисел с многократной точностью

Как правило, числа записываются в некоторой позиционной системе счисления. Пусть это будет система с основанием $Base$. Тогда удобно представлять это число в памяти компьютера, записывая его цифры в некоторый массив $A[MaxLen]$, где $MaxLen$ – это максимальное количество цифр в числе, с которым может потребоваться работа. В отличие от привычной записи, где на первой позиции стоит самый старший разряд, в машинном представлении удобнее использовать обратную запись, где $A[0]$ будет соответствовать младшему разряду. Тогда число A будет представляться в виде $A[0] + A[1] \cdot Base + A[2] \cdot Base^2 + \ldots$.

Разумеется, далеко не всегда все $MaxLen$ цифр числа будут необходимы. При представлении маленьких чисел большинство разрядов будут просто нулевыми и их обрабатывать при большинстве операций не имеет смысла. Поэтому добавим в нашему представлению переменную len, в которой будет храниться позиция старшего разряда числа (то есть $A[len] \neq 0$ и $A[i] = 0$ для всех $i > len$, при этом эти старшие нули не обязательно будут действительно записываться в массив A, но это будет неявно предполагаться). Возможно использование len как длины представляемого числа (тогда старшая цифра будет храниться в $A[len - 1]$, однако мы в данной лекции будем рассматривать первый вариант.

Еще один вопрос, который возникает при представлении чисел – это выбор основания системы. Разумеется для компьютера более удобной является двоичная система, в этом случае многие элементарные операции могут быть реализованы с помощью битовых операций. Однако, как правило, на вход подаются и на выходе требуются числа, представленные в десятичной системе. Поскольку перевод из одной системы в другую является достаточно трудоемкой операцией (сложность $O(len^2)$), то предпочтительнее использовать десятичную систему. О двоичной системе представления есть смысл задумываться лишь в том случае, когда количество операций, выполняемых над длинными числами, довольно велико, и среди них встречаются операции со сложностью не менее $O(len^2)$.

Следует также отметить, что чем короче длина представления нашего числа, тем меньше время выполнения операций. Поскольку $len = \lfloor \log_{Base} A \rfloor$, ясно что чем больше будет основание системы, тем быстрее будут выполняться арифметические операции. При этом следует учитывать, что 32-разрядный процессор наиболее эффективен при работе с числами разрядности не больше 32. Следовательно, основание должно быть таким, чтобы все промежуточные вычисления не выходили за пределы 32-разрядного целого числа. При этом, чтобы процесс преобразования десятичных чисел на входе в систему с основанием $Base$ и обратное преобразование при выводе, основание должно быть степенью десятки: $Base = 10^d$. В этом случае, d последовательных десятичных цифр компонуются в одну по основанию $Base$.

Если в процессе работы требуется лишь сложение и вычитание, то в качестве d можно брать 9, если же есть умножение и деление, то следует ограничиться значением 4. В дальнейшем мы будем рассматривать основание $Base = 10^4$, однако это будет играть существенную роль лишь при вводе-выводе данных.

Следует отметить, что число 0 мы будем отмечать значением $len = -1$.

2 Ввод длинных чисел.

Чтобы выполнить чтение длинного числа, сначала считаем его десятичное представление в какой-нибудь промежуточный буфер (это потребуется хотя бы для того, чтобы точно узнать длину числа). После этого мы будем собирать десятичные цифры с конца, объединяя их по d в одну цифру по основанию 10^d.

```
char buf[MaxLen*d],*s;
int deg=Base;
scanf('%d',buf);
len=-1;
for (s=buf+strlen(buf); --s!=buf;)
{
  if (deg==Base)
      A[++len]=0, deg=1;
  A[len] += (*s-'0') * deg;
  deg *= 10;
}
while (len>=0 && A[len]==0)
  len--;
```

Последний цикл while необходим лишь в том случае, когда число может быть задано с ведущими нулями. Этот цикл также нужен при чтении числа 0, хотя в этом случае его можно заменить на if. Нетрудно видеть, что сложность этого алгоритма пропорциональна длине ввода – $O(len \cdot d)$.

3 Вывод длинных чисел.

Поскольку у нас в одном элементе массива a скомпоновано 4 десятичных цифры числа A, то за исключением $A[len]$ все остальные элементы должны выводиться с ведущими нулями, дополняющими их до 4 знаков. В языке C для этого можно пользоваться строкой формата "%04d":

```
if (len==-1)
  putchar('0');
else
{
  printf('%d',A[len]);
  for (i=len;i--;)
    printf('%04d',A[i]);
}
```

В случае отсутствия в языке возможности дополнения нулями, придется выполнить это самостоятельно. Например, вывод значения t, дополненного нулями до $\log_{10} Base$ знаков, может быть выполнено следующим образом:

```
for (int deg=Base; deg/=10, deg>1 && t<deg;)
  putchar('0');
printf('%d',t);
```

Сложность алгоритма вывода составит $O(len)$.

4 Присваивание длинных чисел

Эта операция требует особого внимания в том случае, если реализовывать длинные числа как класс. В языке C++ операция присваивания объекту некоторого класса другого объекта этого класса определена по умолчанию, как полное копирование всех данных из области памяти, занимаемой первым объектом, в область памяти, занимаемой вторым объектом. То есть будут копироваться все $MaxLen$ элементов. Однако, учитывая особенности выбранного нами представления, достаточно скопировать лишь элементы с 0 по len:

```
B.len=A.len;
for (int i=0;i<=A.len;i++)
  B[i]=A[i];
```

Сложность – $O(len)$.

5 Присваивание короткого числа

Операция присваивания короткого числа (пусть это будет t), не превышающего $Base - 1$, довольно проста. Достаточно поместить это число в $A[0]$ и, если оно отлично от нуля, установить $len = 0$. Тем не менее можно присваивать и большие числа. Необходимо их только перевести в систему с основанием $Base$.

```
for (len=-1;t;t/=Base)
  A[++len]=t%Base;
```

6 Сравнение длинных чисел

Для того, чтобы определить какое из чисел A или B больше, достаточно сравнить их длины, а при их равенстве лексикографически сравнить последовательности цифр, начиная со старшей:

```
int t;
if (t=A.len-B.len)
  return t;
for (i=A.len;i>=0;i--)
  if (t=A[i]-B[i])
    return t;
return 0;
```

Следует заметить, что в условном операторе используется именно операция присваивания, а не сравнения. Это позволяет присвоить значение переменной t, и если она окажется отличной от 0, вернуть ее значение.

Данный фрагмент схож с функцией сравнения строк strcmp в том смысле, что возвращаемое значение может быть любым целым числом. При этом, если $A < B$, то будет возвращено некоторое отрицательное число, если $A > B$ – положительное, и если $A = B$ – значение 0.

Сложность в худшем случае – $O(len)$.

7 Сложение

Пусть заданы два длинных числа A и B, каждое из которых имеет длину $len + 1$. (Ясно, что если длины не равны, то можно более короткое число дополнить в старших разрядах нулями. В принципе этого можно не делать явно, но тогда нужно особо обрабатывать запросы к разрядам, находящимся за старшим.) Требуется найти их сумму C. Воспользуемся классическим алгоритмом сложения в столбик, который проходит по числам от младших разрядов к старшим, складывая их и сохраняя цифру переноса. При этом следует учесть, что если после сложения старших разрядов возникает перенос, то C будет содержать на одну цифру больше:

```
int t=0;
```

```
for (int i=0;i<=len;i++)
{
  t+=A[i]+B[i];
  C[i]=t%Base, t/=Base;
}
C.len=len;
if (t)
  C[++C.len]=t;
```

На машинах, где операция деления выполняется долго, фрагмент с делением может быть заменен условным оператором:

```
if (t>=Base)
  C[i]=t-Base, t=1;
else
  C[i]=t, t=0;
```

На современных компьютерах, как правило, такая замена не дает ускорения.

Сложность алгоритма – $O(len)$.

8 Вычитание

Пусть снова есть два длинных числа A и B, каждое из которых имеет длину $len + 1$. Требуется найти их разность C. Алгоритм вычитания в целом оказывается таким же, как и алгоритм сложения:

```
int t=0;
for (int i=0;i<=len;i++)
{
  t+=A[i]-B[i];
  C[i]=t%Base, t/=Base;
}
C.len=len;
```

Однако здесь возникает проблема "отрицательного деления". В математике считается, что знак остатка от деления совпадает со знаком делителя. В машинной арифметике знак остатка совпадает со знаком делимого. Поэтому, несмотря на то, что математически записан верный алгоритм, при выполнении на компьютере он будет давать неверный результат. Для преодоления этой проблемы мы можем либо снова воспользоваться условным оператором, либо добавлять каждый раз еще и $Base$. В таком случае, остаток от деления на $Base$ не изменится, а частное увеличится на 1 и при этом мы не будем иметь проблем с отрицательными числами:

```
t+=A[i]-B[i]+Base;
C[i]=t%Base, t=t/Base-1;
```

Более того, можно хранить цифру переноса (в данном случае заема) увеличенную на 1 (для наглядности обозначим ее $t1 = t + 1$). Тогда изначально $t1 = 1$, соответствующий фрагмент можно будет записать следующим образом:

```
t1+=A[i]-B[i]+(Base-1);
C[i]=t1%Base, t1/=Base;
```

Теперь следует разобраться, что делать после выполнения этого цикла. Если t оказалось равным 0, это означает, что уменьшаемое A было не меньше вычитаемого B, и все что нам остается сделать – это убрать лишние ведущие нули из разности C:

```
while (C.len>=0 && C[C.len]==0)
  C.len--;
```

Если же t получилось отрицательным (т.е. -1), это означает, что $A < B$, и в C будет записано значение разности $A - B$ в дополнительном коде. Дописав за старшим разрядом (то есть в позицию $len + 1$) значение -1, у нас окажется верным равенство $C = C[0] + C[1] \cdot Base + C[2] \cdot Base^2 + \ldots$. Однако при этом следует тоже убрать лишние разряды (значение $Base - 1$, находящееся непосредственно перед разрядом -1, может быть заменено на -1, а тот разряд -1 удален). Соответствующий фрагмент может быть записан так:

```
while (C.len>=0 && C[C.len]==Base-1)
  C.len--;
C[++C.len]=-1;
```

Таким образом, у нас появилось представление и для отрицательных чисел. Однако следует помнить, что в случае работы с такими числами у нас и в операции сложения может возникнуть ситуация с обнулением нескольких старших разрядов, а также с отрицательной суммой. В то же время в результате операции вычитания может появиться новый разряд, как это было при сложении. Тогда алгоритм сложения/вычитания примет следующий вид:

```
int t=0;
for (int i=0;i<=len;i++)
{
  t += A[i] +- B[i] + Base;
  C[i] = t%Base, t = t/Base - 1;
}
C.len=len;
switch (t)
{
  case 1:
    C[++C.len]=1;
    break;
  case 0:
    while (C.len>=0 && C[C.len]==0)
      C.len--;
    break;
  case -1:
    while (C.len>=0 && C[C.len]==Base-1)
      C.len--;
    C[++C.len]=-1;
}
```

Здесь знаком $+-$ обозначена та операция, которая должна выполняться (сложение или вычитание). Рекомендуется по возможности избегать работы с отрицательными числами, чтобы не возникало подобных нагромождений в стандартных операциях.

Сложность алгоритма вычитания – $O(len)$.

9 Умножение на короткое

Умножение является многократным применением операции прибавления числа A к 0, поэтому неудивительно, что алгоритм умножения будет очень похож на алгоритм сложения. Итак, пусть у нас есть число A длины $len + 1$, и короткое число b, являющееся одноцифровым в системе с основанием $Base$ (т.е. $0 \le b \le Base - 1$). Тогда произведение $C = Ab$ может быть вычислено так:

```
int t=0;
for (int i=0;i<=len;i++)
{
  t+=A[i]*b;
  C[i]=t%Base, t/=Base;
```

```
}
C.len=len;
if (t)
  C[++C.len]=t;
```

Заметим, что если в алгоритме сложения переменная t должна была иметь возможность хранить числа от 0 до $2Base-1$, при умножении могут возникать числа из диапазона от 0 до $(Base-1)Base-1$.

На самом деле мы можем умножать и на числа, превышающие $Base-1$, однако следует учесть, что при этом в конце может появиться не одна старшая цифра, а несколько. Поэтому последний условный оператор потребуется заменить циклом, аналогичным тому, который возникал при присваивании длинному числу короткого. Кроме того, теперь переменная t должна будет иметь возможность сохранять значения до $b \cdot Base - 1$ включительно.

Сложность алгоритма – $O(len)$.

10 Деление на короткое

Деление является обратной операцией к умножению и поэтому для его выполнения все действия, которые выполнялись при умножении, следует обратить. Теперь мы проходим от старших разрядов к младшим и выполняем умножение на $Base$, а деление на b. Пусть задано число A длины $len+1$, и короткое число b $(0 \le b \le Base-1)$. Тогда целочисленное частное $C = A/b$ определяется следующим алгоритмом:

```
int t=0;
for (int i=len;i>=0;i--)
{
  t=t*Base+A[i];
  C[i]=t/b, t%=b;
}
C.len=len;
if (C.len>=0 && C[C.len]==0)
  C.len--;
return t;
```

Заметим, что по окончании алгоритма в переменной t будет находиться остаток от деления A на b.

Как и в случае с умножением, можно делить не только на числа, меньшие основания, но при этом снова потребуется заменить if на соответствующий while (поскольку может обнулиться более одного старшего разряда), а также обеспечить для t такой тип, который позволит хранить числа до $b \cdot Base - 1$.

Сложность алгоритма – $O(len)$.

11 Умножение длинных чисел

Во всех приведенных выше алгоритмах было не принципиально, чтобы операнды операций (A, B) и результат (C) хранились в разных переменных. Связано это с тем, что в $C[i]$ записывается значение уже после того, как значения $A[i]$ и $B[i]$ уже будут использованы. Поэтому легко могут быть реализованы операции $+=$, $-=$, а также для коротких вторых операндов операции $*=$, $/=$. В случае же длинного умножения и деления некоторые цифры результата C будут изменяться раньше, чем произойдут обращения к соответствующим цифрам чисел A и B.

Пусть есть числа A и B (здесь мы не будем предполагать равенства их длин). Требуется получить их произведение $C = A * B$. Известный из начальных классов средней школы алгоритм умножения "в столбик" заключается в том, что мы множим A на каждую из цифр числа B и складываем полученные результаты с соответствующим сдвигом. В первом приближении этот алгоритм можно записать так:

```
C=0;
for (int j=0; j<=B.len; j++)
```

```
C += A*B[j] << j;
```

Заметим, что если бы мы изначально не обнуляли переменную C, то можно было бы получить алгоритм который добавляет к C произведение $A * B$. Естественно, что нет смысла в сохранении промежуточного результата умножения A на $B[j]$, можно сразу же при получении очередной цифры этого произведения добавлять ее в C с соответствующим сдвигом. Таким образом, код длинного умножения будет иметь вид:

```
int i,j,k;
for (k=A.len;k>=0;k--)
  C[k]=0;
for (j=0;j<=B.len;j++)
{
  int t=0, b=B[j];
  for (i=0, k=j; i<=A.len; i++, k++)
  {
      t+=A[i]*b+C[k];
      C[k]=t%Base, t/=Base;
  }
  C[k]=t;
}
C.len=k;
if (C[C.len]==0)
  C.len--;
```

Отдельного рассмотрения требует случай, когда хотя бы один из множителей равен 0, в этом случае следует $C.len$ присвоить -1 и выйти.

Общая сложность алгоритма – $O(A.len \cdot B.len)$.

Следует отметить, что внутренний цикл можно не выполнять, если очередная цифра множителя $B[j]$ равна нулю, однако добавлять соответствующий условный оператор, как правило, не имеет смысла, поскольку при большом основании вероятность того, что цифра числа окажется равной нулю, будет малой. Как показано у Кнута, при реализации данного алгоритма на машине MIX выигрыш при пропуске нулевой цифры имеет место лишь тогда, когда $Z/B.len > 3/(28A.len + 3)$, где Z количество нулевых цифр в множителе B.

12 Деление длинных чисел

Пусть заданы число A длины $A.len+1$ и B длины $B.len+1$. В результате работы алгоритма в C мы получим целое частное от их деления, а в A будет остаток (при желании можно ввести дополнительную переменную, в которую первоначально записать значение A, и работать уже затем с ней – это позволит не испортить значения делимого A).

Сразу следует убрать из рассмотрения случай когда $B.len = -1$ (т.е. $B = 0$). Поскольку частное от деления на 0 не определено, а остаток равен делимому, то в этом случае можно сразу выходить – во всех переменных будут математически корректные значения.

То же самое следует сделать и в том случае, когда $A.len < B.len$, только при этом еще частное C должно быть нулевым (то есть нужно $C.len$ присвоить -1).

В дальнейшем мы будем рассматривать лишь случай, когда $A.len \geq B.len$. Вообще говоря, при делении числа, старший разряд которого находится в позиции $A.len$, на число со старшей позицией $B.len$ получится число, в котором старший разряд будет либо в позиции $A.len - B.len$, либо в позиции $A.len - B.len - 1$. Будем считать, что всегда получается число со старшей позицией $A.len - B.len$, цифра в которой может оказаться равной 0. Поэтому мы добавим еще одну нулевую цифру в делимое и после этого сможем начать деление. В первом приближении алгоритм будет иметь вид:

```
int N=B.len+1;
C.len=A.len-B.len;
A[++A.len]=0;
```

```
for (i=C.len; i>=0; i--)
{
  int q = A[i:i+N] / B;
  A[i:i+N] -= q*B;
  C[i] = q;
}
if (C[C.len]==0)
  C.len--;
```

Здесь под $A[i:i+N]$ понимается число $(N+1)$-разрядное число $A[i]+A[i+1]\cdot Base+\ldots+A[i+N]\cdot Base^N$.

Как видим, задача сводится к тому, чтобы найти частное q от деления $N+1$-разрядного числа (пусть это будет $u=(u_N u_{N-1}\ldots u_1 u_0)_{Base}$) на N-разрядное ($v=(v_N v_{N-1}\ldots v_1 v_0)_{Base}$). При этом, по построению u и v будут такими, что $q=\lfloor u/v\rfloor$ всегда будет одноцифровым числом, т.е. $0\le q\le Base-1$.

Первый вариант нахождения q – это вычитание v из u до тех пор, пока это возможно. Количество произведенных вычитаний и будет определять q:

```
for (q=0; u>=v; q++)
  u-=v;
```

По окончанию этого фрагмента в переменной u будет содержаться значение $u-qv$, что позволяет нам выполнить сразу два действия из алгоритма деления. Однако такой подход требует $O(Base)$ операций вычитания, каждая из которых имеет сложность $O(N)$.

Второй вариант – использовать двоичный поиск:

```
for (ql=0, qr=Base; qr-ql>1; )
{
  q=(ql+qr)/2;
  if (u < q*v)
      qr=q;
  else
      ql=q;
}
q=ql;
```

Корректность алгоритма следует из того, что сохраняются инварианты $q_l v\le u<q_r v$. Поскольку по окончанию алгоритма $q_r=q+1$, а $q_l=q$, выполняется $0\le u-q\cdot v<v$, что и является определением целочисленного частного. Этот подход требует уже $O(\log_2 Base)$ операций умножения на короткое, каждая из которых имеет сложность $O(N)$. Возможны некоторые оптимизации, которые позволяют избавиться от умножений, однако они никак не влияют на асимптотическую оценку сложности.

И, наконец, третий способ вычисления q использует $O(1)$ операций умножения длинного на короткое. Для этого рассмотрим некоторую оценку частного q, которая получается при делении двух старших цифр u на старшую цифру v. (Для упрощения записи выкладок будем вместо $Base$ писать b). Итак, положим

$$\hat q=\min\left(\left\lfloor\frac{u_n b+u_{n-1}}{v_{n-1}}\right\rfloor, b-1\right).$$

Оказывается, что $\hat q$ почти всегда является достаточно хорошим приближение к искомому ответу q, если только v_{n-1} достаточно велико. Сначала докажем, что $\hat q$ не может быть слишком мало.

Теорема A. $\hat q\ge q$.

Доказательство. Так как $q\le b-1$, теорема, безусловно, верна при $\hat q=b-1$. Если же $\hat q=\lfloor(u_n b+u_{n-1}/v_{n-1}\rfloor$, то $\hat q v_{n-1}\ge u_n b+u_{n-1}-v_{n-1}+1$. Отсюда следует, что

$$\begin{aligned}u-\hat q v\le u-\hat q v_{n-1}b^{n-1}\le u_n b^n+\ldots+u_0-(u_n b+u_{n-1}-v_{n-1}+1)b^{n-1}\le\\ \le(u_{n-2}\ldots u_0)_b-b^{n-1}+v_{n-1}b^{n-1}<v_{n-1}b^{n-1}\le v.\end{aligned}$$

Поскольку $u - \hat{q}v < v$, должно быть $\hat{q} \geq q$, что и требовалось доказать.

Докажем теперь, что на практике $\hat{q}$ не может быть намного больше q. Предположим, что $\hat{q} \geq q+3$. Тогда

$$\hat{q} \leq \frac{u_n b + u_{n-1}}{v_{n-1}} = \frac{u_n b^n + u_{n-1} b^{n-1}}{v_{n-1} b^{n-1}} < \frac{u}{v - b^{n-1}}.$$

Последний переход будет некорректен, если $v = b^{n-1}$, однако в нашем случае это невозможно поскольку тогда $v = (10 \ldots 0)$ и $q = \hat{q}$. Поскольку $q > (u/v) - 1$, то

$$3 \leq \hat{q} - q < \frac{u}{v - b^{n-1}} - \frac{u}{v} + 1 = \frac{u}{v} \cdot \frac{b^{n-1}}{v - b^{n-1}} + 1.$$

Поэтому

$$\frac{u}{v} > 2\frac{v - b^{n-1}}{b^{n-1}} \geq 2(v_{n-1} - 1).$$

Таким образом, поскольку $b-4 \geq \hat{q}-3 \geq q = \lfloor u/v \rfloor \geq 2(v_{n-1}-1)$, то $v_{n-1} \leq (b-4)/2+1 = b/2-1$ и, значит, $v_{n-1} < \lfloor b/2 \rfloor$. Это доказывает следующую теорему.

Теорема B. Если $v_{n-1} \geq \lfloor b/2 \rfloor$, то $\hat{q} - 2 \leq q \leq \hat{q}$.

Таким образом, мы видим, что вне зависимости от того, насколько велико $Base$, пробное частное $\hat{q}$ никогда не отличается от истинного q более чем на 2.

Однако далеко не всегда делитель будет иметь старшую цифру не меньшую половины основания. Чтобы обеспечить это, будем домножать и делимое, и делитель на 2 до тех пор, пока условие теоремы B не станет верным. Либо можно сразу домножить их на $d = \lfloor b/(v_{n-1}+1) \rfloor$. Очевидно это никак не изменит частного, а остаток увеличится ровно в d раз.

Следует отметить, что если делитель состоит не менее чем из двух цифр, оценку $\hat{q}$ можно уточнить за счет рассмотрения отношения числа, состоящего из старших трех цифр делимого, и числа из двух старших цифр делителя. Тогда для истинного значения q останутся лишь два варианта – либо $\hat{q}$, либо $\hat{q}-1$. Причем как показано у Кнута, вероятность второго случая чрезвычайно мала (порядка $2/Base$).

С учетом всего вышесказанного алгоритм деления примет следующий вид:

```
C.len=A.len-B.len;
if (C.len<0 || B.len==-1)
  C.len=-1;
else if (B.len==0)
{
  ; используем короткое деление
  C = A/B[0], остаток помещаем в A[0], A.len=-1;
}
else
{
  int N=B.len;
  ; нормализация
  int d = Base / (b[B.len] + 1);
  B *= d;
  A *= d; ; при этом последний перенос мы записываем даже если он нулевой
  int bn = B[N], bn1=B[N-1];
  for (k=A.len, i=C.len;i>=0;i--,k--)
  {
    int t = A[k] * Base + A[k-1];
    int q = t / bn, r = t % bn;
    while (q == b || q * bn1 > Base * r + A[k-2])
      q--, r+=bn;

    A[i:k] -= q*B;
```

```
    if (A[i:k] - отрицательное, то есть цифра заема оказалась в конце равна -1)
      A[i:k] += B, q++;

    C[i] = q;
  }
  if (C[C.len]==0)
    C.len--;
  ; денормализация
  A /= d;
  B /= d;  ; это нужно делать, если требуется вернуть исходное значение делителя
}
```

Сложность алгоритма – $O(C.len \cdot B.len)$.

13 Извлечение квадратного корня

Пусть задано длинное число A состоящее из $A.len + 1$ цифр, причем $A.len + 1$ – четное число (если это не так, можно дописать еще один старший разряд равный 0). Требуется найти наибольшее целое число такое, что $C^2 \le A$. Очевидно, в нем будет $n = (A.len + 1)/2$ цифр.

Итак, нам известно $A = (a_{2n-1}a_{2n-2}\dots a_1a_0)_b$ и нужно найти $C = (c_{n-1}c_{n-2}\dots c_1c_0)_b$ (для сокращения записи мы вновь используем обозначение b вместо $Base$).

Пусть $p = \lfloor\sqrt{a_{2n-1}b + a_{2n-2}}\rfloor$. Покажем, что $c_{n-1} = p$. Из определения функции $\lfloor\cdot\rfloor$ следует, что $p \le \sqrt{a_{2n-1}b + a_{2n-2}} < p + 1$, Поэтому

$$p^2 \le a_{2n-1}b + a_{2n-2} \le (p + 1)^2 - 1.$$

$$(p \cdot b^{n-1})^2 \le a_{2n-1}b^{2n-1} + a_{2n-2}b^{2n-2} \le ((p + 1) \cdot b^{n-1})^2 - b^{2n-2}.$$

Из левого неравенства следует, что $((p00\dots0)_b)^2 \le a_{2n-1}b^{2n-1} + a_{2n-2}b^{2n-2} \le A$, откуда следует, что $C \le (p00\dots0)_b$. Из правого же неравенства получается, что $(((p + 1)00\dots0)_b)^2 \ge a_{2n-1}b^{2n-1} + a_{2n-2}b^{2n-2} + b^{2n-2} > A$, поэтому $C < ((p + 1)00\dots0)_b$. Таким образом, старшая цифра корня действительно равна p.

Пусть уже определены цифры C с $(n-1)$-ой до $(i+1)$-ой. Обозначим $X_{i+1} = c_{n-1}b^{n-(i+1)} + \dots + c_{i+2}b + c_{i+1}$, $R_i = \lfloor A/b^{2i}\rfloor - X^2b^2$. Тогда цифру c_i следует искать как наибольшую из тех, которые удовлетворяют условию $(X_{i+1} \cdot b + c_i)^2 \le \lfloor A/b^{2i}\rfloor$. Раскроем квадрат суммы:

$$X_{i+1}^2b^2 + 2X_{i+1}bc_i + c_i^2 \le \lfloor A/b^{2i}\rfloor,$$

откуда

$$(2X_{i+1}b + c_i)c_i \le R_i.$$

Следовательно, c_i следует подбирать таким образом, чтобы было c_i не превышало $\lfloor R_{i+1}/(2X_{i+1}b + c_i)\rfloor$. При этом число $2X_{i+1}$ будет иметь $n-i$ знаков (возможно на 1 больше за счет удвоения), а R_{i+1} будет состоять из $n - i + 1$ знаков. Поэтому c_i будет почти всегда определяться частным от деления $(n - i + 1)$-значного числа R_{i+1} на $(n - i)$-значное X_i. Фактически это же мы выполняли и в алгоритме деления, при этом, как было показано, для оценки частного достаточно две старшие цифры делимого разделить на старшую цифру частного (младший разряд, которым в данном случае будет c_i, особой роли не играет). Нетрудно видеть, что после определения c_i довольно легко выражаются X_i и R_i:

$$X_i = X_{i+1}b + c_i,$$
$$R_{i-1} = (R_i - (2X_{i+1}b + c_i)c_i)b^2 + a_{2i-1}b + a_{2i-2}.$$

Таким образом, алгоритм извлечения квадратного корня оказывается похожим на алгоритм деления, за исключением того, что на каждом шаге увеличивается размер делителя и рассматриваемой части делимого.

Перейдем теперь к особенностям реализации. Как и в алгоритме деления, на первом шаге следует произвести нормализацию, чтобы обеспечить, что старшая цифра результата будет не менее половины основания. Однако заметим, что нам следует делить каждый на число, которое вдвое больше, чем уже найденная часть корня. Это может привести, к тому что увеличится количество разрядов и в старшем будет находиться 1. Решается эта проблема довольно просто – не позволять увеличиваться разрядности за счет того, что в старшем разряде могут быть значения до $2Base - 1$ (разумеется, при этом нужно обеспечить возможность хранения в промежуточных переменных значений до $2Base^2 - 1$). С учетом того, что старшая цифра X_i не меньше $\lfloor Base/2 \rfloor$, старший разряд числа $2X_i$ будет не меньше $Base - 1$.

Можно показать, что при такой старшей цифре оценка $\hat{q}$ из предыдущего пункта дает приближение $\hat{q} - 1 \le q \le \hat{q}$. Действительно, предположим, что $\hat{q} - q \ge 2$. Тогда, повторяя начальные выкладки, получим:

$$2 \le \hat{q} - q < \frac{u}{v} \cdot \frac{b^{n-1}}{v - b^{n-1}} + 1.$$

Отсюда следует, что

$$\frac{u}{v} \cdot \frac{b^{n-1}}{v - b^{n-1}} > 1$$

и

$$\frac{u}{v} > \frac{v - b^{n-1}}{b^{n-1}} \ge v_{n-1} - 1.$$

Таким образом, поскольку $b - 3 \ge \hat{q} - 2 \ge q = \lfloor u/v \rfloor \ge v_{n-1} - 1$, то $v_{n-1} \le b - 2$. В нашем случае это не так, поэтому приходим к противоречию.

Кроме того, нет смысла хранить значение X_i домножая его каждый раз на 2, будем хранить сразу $2X_i$, а в конце алгоритма разделим его на 2 и (возможно) еще на нормирующий множитель. При этом значения X_i мы будем хранить в массиве C, а R_i – в массиве A.

Итак, алгоритм извлечения квадратного корня может быть записан так:

```
if (A.len==-1)
  C.len=-1;
else
{
  if (A.len%2==0)
      A[++A.len]=0;
  int i=A.len, j=C.len=i/2;

  ; нахождение старшей цифры корня
  int t;
  t=A[i]*Base+A[i-1];
  C[j]=int(sqrt(double(t)));

  ; нормализация
  int d=Base/(C[j]+1);
  A*=d; A*=d;  ; если есть тип позволяющий хранить значения до 1/4 * Base^3,
               ; то можно умножать сразу на d*d
  t=A[i]*Base+A[i-1];
  C[j]=int(sqrt(double(t)));
  t -= C[j]*C[j];
  A[i]=t/Base, A[i-1]=t%Base;
  C[j]*=2;
  while (j!=0)
  {
     j--, i--;
     A[i]+=A[i+1]*Base, A[i+1]=0;
```

```
      ; нахождение оценки
      q=(A[i]*Base+A[i-1])/C[C.len];
      if (q>=Base)
          q=Base-1;

      ; нахождение нового остатка R[j]
      C[j]=q;
      int ki,kj;
      for (t=0, kj=j, ki=2*j; kj<=C.len; kj++, ki++)
      {
          t+=A[ki]-q*C[kj];
          A[ki]=t%Base, t/=Base;
      }
      t+=A[i]; A[i]=t;

      ; если оценка оказалась неточной
      if (t<0)
      {
        A[j]=--q;
        t=q+1;
        for (kj=j,ki=2*j;kj<=len;kj++,ki++)
        {
          t+=A[ki]+C[kj];
          A[ki]=t%Base;
          t=t/Base;
        }
        A[i]+=t;
      }

      ; удвоение последней цифры
      t=C[j]*2;
      C[j]=t%Base, C[j+1]+=t/Base;
    }

    ; денормализация
    C /= 2*d;
  }
```

Отметим, что при нахождении остатка R_j у нас выполняется вычитание с возможностью получить отрицательное число. Учитывая машинную проблему "отрицательного деления" следует переписать этот фрагмент таким образом, чтобы отрицательных чисел не возникало:

```
  ; нахождение нового остатка R[j]
  C[j]=q;
  int ki,kj;
  for (t=0, kj=j, ki=2*j; kj<=C.len; kj++, ki++)
  {
    t+=A[ki]-q*C[kj]+2*Base^2;
    A[ki]=t%Base, t=t/Base - 2*Base;
  }
  t+=A[i]; A[i]=t;
```

Разумеется при этом константы $2Base$ и $2 \cdot Base^2$ следует вычислить заранее. Сложность полученного алгоритма – $O(C.len^2)$.