



Big Numbers

Sample Problem: Factorial

Given N ($1 \leq N \leq 200$), calculate $N!$ exactly.

The Problem

All built-in integers have a limited value range. Sometimes, a problem will ask for the calculation of a number which is outside that range for all the available numbers. For example, $200!$ has 375 digits and would require a 1246 bit number (if one chose to store the number in binary) to hold it, which is unlikely to be available as a native datatype in contest environments. Thus, what is needed is a way to store and manipulate large numbers.

The Structure

One method to store these numbers is actually fairly straight-forward: a list of numbers and a sign. This list can be either an array, if an upper bound is known on the length of the numbers, or a linked list, if the numbers have no upper bound.

One way to think of this storage method is keeping the 'digits' of a number in a huge base.

Let b be the base in which the number is stored, and a_0, a_1, \dots, a_n be the digits stored. Then, the number represented is $(-1)^{\text{sign}} \text{ times } (a_0 + b^1 a_1 + b^2 a_2 + \dots + b^n a_n)$. Note that a_0, a_1, \dots, a_n must be in the range $0..b-1$.

Generally, the base b is selected to be a power of 10, as it makes displaying the number very easy (but don't forget the leading zeroes).

Note that this representation stores the numbers in the order presented: a_0 , then a_1 , then a_2 , etc. This is the reverse of the obvious ordering, and, for linked lists, it may be worthwhile to keep a deque, as some of the algorithms will want to walk through the list in the opposite order.

Operations

For this data structure, figuring out how to do the various operations requires recalling how to do the operations by hand. The main problem that one has to be aware of is overflow. Always make sure that every addition and multiplication will not result in an overflow, or the entire operation will be incorrect.

For simplicity, the algorithms presented here will assume arrays, so if a number is a_0, a_1, \dots, a_k , then for all $i > k$, $a_i = 0$. For linked lists, the algorithms become a bit more difficult. In addition, the result bignums are assumed to be initialized to be 0, which will

not be true in most cases.

Comparison

To compare two numbers a_0, a_1, \dots, a_n and b_0, b_1, \dots, b_k , with signs `signA` and `signB` goes as follows:

```
# note that sign
# sizeA = number of digits of A
# signA = sign of A
# (0 => positive, 1 => negative)
1  if (signA < signB)
2    return A is smaller
3  else if (signA > signB)
4    return A is larger
5  else
6    for i = max(sizeA,sizeB) to 0
7      if (a(i) > b(i))
8        if (signA = 0)
9          return A is larger
10     else
11       return A is smaller
12   return A = B
```

Addition

Given two numbers a_0, a_1, \dots, a_n and b_0, b_1, \dots, b_k , calculate the sum and store it in `c`. In order for addition to be possible, $2 \times b$ must be smaller than the largest representable number.

If the numbers have opposite signs, then: calculate which is larger in absolute value, subtract the smaller from the larger, and set the sign to be the same as the larger number. Otherwise, simply add the numbers from 0 to $\max(n, k)$, maintaining a carry bit.

Note that if it is known that both numbers have positive sign, the operation becomes much simpler and doesn't even require the writing of `absolute_subtract`.

Here is the pseudocode for addition:

```
1 absolute_subtract(bignum A, bignum B,
                    bignum C)
2   borrow = 0
3   for pos = 0 to max(sizeA, sizeB)
4     C(pos) = A(pos)-B(pos)-borrow
5     if (C(pos) < 0)
6       C(pos) = C(pos) + base
7       borrow = 1
8     else
9       borrow = 0
10    # it has to be done this way,
11    # to handle the case of
12    # subtracting two very close nums
13    # (e.g., 7658493 - 7658492)
14    if C(pos) != 0
15      sizeC = pos
16  assert (borrow == 0,
17         "|B| > |A|; can't handle this")
18
19 absolute_add(bignum A,
              bignum B, bignum C)
20   carry = 0
21   for pos = 0 to max(sizeA,sizeB)
22     C(pos) = A(pos)+B(pos)+carry
23     carry = C(pos) / base
24     C(pos) = C(pos) mod base
25   if carry != 0
26     CHECK FOR OVERFLOW
27     C(max(sizeA,sizeB)+1) = carry
28     sizeC = max(sizeA,sizeB)+1
29   else
30     sizeC = max(sizeA, sizeB)
```

```

25 add (bignum A, bignum B, bignum C)
26   if signA == signB
27     absolute_add(A,B)
28     signC = signA
29   else
30     if (Compare(A,B) = A is larger)
31       then
32         absolute_subtract(A,B)
33         signC = signA
34       else
35         absolute_subtract(B,A)
36         signC = signB

```

Subtraction

Subtraction is simple, given the addition operation above. To calculate $A - B$, negate the sign of B and add A and $(-B)$.

Multiplication by scalar

To multiply a bignum A by the scalar s , if $|s \times b|$ is less than the maximum number representable, can be done in a similar manner to how it is done on paper.

```

1  if (s < 0)
2    signB = 1 - signA
3    s = -s
4  else
5    signB = signA
6  carry = 0
7  for pos = 0 to sizeA
8    B(pos) = A(pos) * s + carry
9    carry = B(pos) / base
10   B(pos) = B(pos) mod base
11  pos = sizeA+1
12  while carry != 0
13    CHECK OVERFLOW
14    B(pos) = carry mod base
15    carry = carry / base
16    pos = pos + 1
17  sizeB = pos - 1

```

Multiplication of two bignums

Multiplying two numbers, if b^2 is below the largest representable number is a combination of scalar multiplication and in-place addition.

Basically, multiply one of the numbers by each digit of the other, and add it (with the appropriate offset) to a running total, the exact same way one does long multiplication on paper.

```

1  multiply_and_add(bignum A, int s,
2                  int offset, bignum C)
3  carry = 0
4  for pos = 0 to sizeA
5    C(pos+offset) = C(pos+offset) +
6      A(pos) * s + carry
7    carry = C(pos+offset) / base
8    C(pos+offset) =
9      C(pos+offset) mod base
10  pos = sizeA + offset + 1
11  while carry != 0
12    CHECK OVERFLOW
13    C(pos) = C(pos) + carry
14    carry = C(pos) / base
15    C(pos) = C(pos) mod base
16    pos = pos + 1
17  if (sizeC < pos - 1)
18    sizeC = pos - 1

```

```

16 multiply(bignum A,
17          bignum B, bignum C)
18 for pos = 0 to sizeB

```

```

18     multiply_and_add(A,
        B(pos), pos, C)
19     signC = (signA + signB) mod 2

```

Division by scalar

In order to divide the bignum b by a scalar s , $s \times b$ must be representable. Division is done in a very similar manner to long division.

```

20 divide_by_scalar (bignum A,
        int s, bignum C)
21     rem = 0
22     sizeC = 0
23     for pos = sizeA to 0
24         rem = (rem * b) + A(pos)
25         C(pos) = rem / s
26         if C(pos) > 0 and
            pos > sizeC then
27             sizeC = pos
28         rem = rem mod s
# remainder has the remainder
# of the division

```

Division by bignum

This is similar to division by scalar, except the division is done by multiple subtractions. Note that if b is large, this particular formulation takes too long.

```

1 divide_by_bignum (bignum A,
        bignum B, bignum C)
2     bignum rem = 0
3     for pos = sizeA to 0
4         mult_by_scalar_in_place(rem, b)
5         add_scalar_in_place(rem,
            A(pos))
6         C(pos) = 0
7         while (greater(rem, B))
8             C(pos) = C(pos) + 1
9             subtract_in_place(rem, B)
10        if C(pos) > 0 and pos > sizeC
            then
11            sizeC = pos

```

Binary Search

Binary search is a very helpful thing in general, but in particular for bignums, as operations are expensive. Given an upper and lower bound, check the mean of those two to see if it is above or below those bounds, and cut the range by (almost) a factor of 2.

For example, if b is large, then division by a bignum is slow to do by the method above, but the following works well:

```

1 divide_by_bignum2 (bignum A,
        bignum B, bignum C)
2     bignum rem = 0
3     for pos = sizeA to 0
4         mult_by_scalar_in_place(rem, b)
5         add_scalar_in_place(rem,
            A(pos))
6         lower = 0
7         upper = s-1
8         while upper > lower
9             mid = (upper + lower)/2 + 1
10            mult_by_scalar(B, mid, D)
11            subtract(rem, D, E)
12            if signE = 0
13                lower = mid
14            else
15                upper = mid - 1
16        C(pos) = lower
17        mult_by_scalar(B, lower, D)

```

```
18     subtract_in_place(rem, D)
19     if C(pos) > 0 and
20         sizeC = pos
```

[Back to USACO Training Gateway](#) | [Comment or Question](#)