BIG

NUMBERS

How can I determine the exact value of something like 200!

Which Structure to use

An array of integers.

See this as the number's base n representation.

If we store x as a[0] to a[k] to base n then:

$x = a[0] + a[1].n + a[2].n^2 + a[3].n^3 + \ldots + a[k].n^k.$

Then all we still need is a variable for the sign.

Bignumber = array [-2..max] of integer

- -2 is to store the max exponent in a bignumber.
- -1 is to store the sign of the bignumber.
- 0..max store the coefficients of $base^{place}$.
- How do I decide what base to use:
  - Normally we choose the base as a power of 10
  - This makes writing it down in the end easier
  - Choose the base so as to prevent overflow
  - Suppose you choose base n – hence 0 .. n-1 have to fit.
  - If you are only adding make sure 2*(n-1) will fit into your int type.
  - If you are multiplying as well make sure that $(n-1)^2$ will fit

Operations

- Comparison
- Addition
- Subtraction
- Multiplication
- Division

Comparison

- I like to use -1 for negative, 0 for 0 and 1 for positive.

```
If  signA > signB then return A > B else
if signB > signA then return B > A else (if signA=signB)
if signA = 0 then return A = B (because both are 0)
else
   ctr = max (sizeA,sizeB)
   while (A[ctr] = B[ctr]) and (ctr > 0) do
       ctr = ctr – 1
   if A[ctr] > B[ctr] then  (implying Abs(A) > Abs(B))
       if signA = 1 then return A > B
       else return A < B
   else
       If A[ctr] < B[ctr] then
           if signA = 1 then return A < B
           else  return A > B
       else (if Abs(A) = Abs(B)) then return A = B
```

Addition

- Firstly write a absolute_sum procedure
- Secondly write a absolute_difference one
- Use absolute_sum for equal sign
- Use absolute_difference for opposite sign

- Note : if it is known that the numbers are all positive you can leave out the absolute_difference procedure.

Absolute sum

```
carry = 0
for pos = 0 to max (sizeA,sizeB) do
   C(pos) = A(pos) + B(pos) + carry
   carry = C(pos) div base
   C(pos) = C(pos) mod base
If carry <> 0 then
   sizeC = max(sizeA,sizeB) + 1
   C(sizeC) = carry
else
   sizeC = max(sizeA,sizeB)
```

```
   1   1 1
  23  874
+15  487
---------
  39  361
=========
```

Absolute difference

```
borrow = 0
for pos = 0 to max (sizeA,sizeB) do
   C(pos) = A(pos) - B(pos) - borrow
   If C(pos) < 0
      C(pos) = C(pos) + base
      borrow = 1
   else
      borrow = 0
While (C(pos) = 0) and (pos > 0 ) do
   pos = pos - 1
sizeC = pos
```

sizeC = pos (this works for pos=0 as well)

```
  1     1 1
 23  874
-15  487
---------
  8  367
=========
```

Make sure that A > B for this or take care of it in procedure

Add

A + B = C

If A and B have the same sign do Absolute addition and signC = signA

If they have different sign do Absolute difference (remember large minus small abs value) and adjust sign

To find out which one has larger absolute value you might consider writing an absolute comparison.

Subtract

Negate the sign of B and Add A and (-B)

Multiplication by scalar

```
If s < 0 then
   signB = -signA
   s = -s                    (so that we multiply with a positive)
else
   signB = signA
carry = 0
for pos = 0 to sizeA do
   B[pos] = A[pos]*s + carry
   carry = B[pos] div base
   B[pos] = B[pos] mod base
pos = sizeA
While (carry <> 0) do        (taking care of the overflow problem)
   pos = pos + 1
   B[pos] = carry mod base
   carry = carry div base
sizeB = pos
```

Multiplication by bignumber

The idea behind this is to first write a procedure to take care of the offset. (call it multiply_and_add)

Difference from scalar multiplication:

1. Replace **B[pos]** with **C[pos+offset]** throughout (use C because in the main procedure we are multiplying A with B to get C)
2. Do not assign **A[pos]*s + carry** directly to **C[pos+offset]** but add it to the existing value.

The main procedure will then look something like this:

for pos = 0 to sizeB do

multiply_and_add(A,B(pos)(the scalar),**pos**(the offset)**,C)**

sinC = signA * signB

Division by scalar

Like with the other cases we will first write a division by scalar:

```
rem = 0
sizeC = 0
for pos = sizeA to 0 do
   rem = (rem*base) = A[pos]
   C[pos] = rem div s
   if (C[pos] > 0) and (pos > sizeC) then
      rem = rem mod s (this will in the end give the remainder)
```

Division by bignumber

Division by multiple subtraction:

Note that this is much too slow for most large cases

This time declare **rem** as a bignumber as well

```
rem = 0
For pos = sizeA to 0 do
  rem = rem*base(scalar) + A[pos]   (use procedures)
  C[pos] = 0
  While (rem > B) do (use compare procedure)
    C[pos] = C[pos] + 1
    rem = rem – B       (use subtract or add procedure)
  if (C[pos] > 0)  and (pos > sizeC) then
    sizeC = pos
```

Division by using binary search

Once again let **rem** also be a bignumber

```
rem = 0
For = sizeA to 0 do
  rem = rem*base + A[pos]   (use procedures as above)
  lower = 0
  upper = base - 1
  while upper > lower do
    mid = (upper + lower) div 2 + 1
    D = B * mid (a scalar)
    E = D – rem
    if signE >= 0
      lower = mid
    else
      upper = mid – 1
  C[pos] = lower
  rem = rem – B*lower and then control C's size like before
```