Arithmetic in a base

The Central Processing Unit (CPU) of a typical computer has a *finite* set of "registers" which store data as "words" (binary strings of a fixed length), a *finite* set of "flags" (True/False values), and a *finite* set of possible operations that can be performed by the CPU. This is roughly how the mathematical design of a Turing machine is implemented in practice though the precise details need to be spelt out.

Nowadays, most CPU's have words of 64-bits. Thus, the registers can store numbers between 0 and $2^{64} - 1$. As a result, the CPU can perform arithmetic operations only on numbers of this size. Thus, we can imagine that the following operations are "built-in" to the CPI.

Addition: Addition of two numbers a and b which lie in $[0, 2^{64} - 1]$ to produce c and d in the same range such that $a + b = c + d \cdot 2^{64}$.

Multiplication: Multiplication of two numbers a and b which lie in $[0, 2^{64} - 1]$ to produce c and d in the same range such that $a \cdot b = c + d \cdot 2^{64}$.

Thinking of $B = 2^{64}$ as the *base* of arithmetic, it is as-if the CPU has "memorized" the "addition and multiplication tables" of "digits" in base B. (Note that even memory "lookup" takes time! So we will need to count each such basic operation.)

To understand how the computer carries out arithmetic of numbers larger than B, we need to recall how we do arithmetic in a given base. (We will focus on Addition and Multiplication but interested students can look up resources like "*Semi-Numerical Algorithms*" by D. Knuth for details on Subtraction and Division.)

Representation of numbers in a given base B

An integer $\underline{a}$ in base B is typically written as $\underline{a} = a_0 + a_1 \cdot B + a_2 \cdot B^2 + \cdots + a_n \cdot B^n$ where a_i are integers in the range $[0, B - 1]$ which we think of as "digits".

In a programming language like Python, we can represent $\underline{a}$ as a *list* $[a_0, \ldots, a_n]$.

In a programming language like C, we can use some library that implements lists. However, for simplicity, we can avoid lists by choosing an arbitrary size (say 1024) as the largest size we will ever use. We can then represent such "pseudo-lists" by using arrays along with their length.

```c
struct bignum {
    int dat[1024];
    int len;
};
```

We can then use `struct bignum alpha` to declare that `alpha` is a variable of type `bignum`.

Whether in Python or in C, the length of the relevant data is its *size* when examine how the complexity of arithmetic operations grows with data size.

Arithmetic with such numbers is called multi-precision arithmetic.

Basic operations

The above basic operations of our CPU will be *simulated* by us using the following two program snippets in Python. (Here, we have taken the conventional base 10 for our simulation.)

```python
base=10
def add_mac(a,b):
    c = a + b
    return c%base, c//base
def mul_mac(a,b):
    c = a * b
    return c%base, c//base
```

Equivalently, in C, we can implement them as the following snippet. Note that we cannot have an array variable as the output of a C function, so we use it as an additional argument with mutable contents.

```c
int base=10;

typedef int result_mac[2];

void add_mac(int a, int b, result_mac res) {
    int c;

    c = a + b;
    result_mac res;
    res[0] = c%base;
    res[1] = c/base;
}

void mul_mac(int a, int b, result_mac res) {
    int c;

    c = a * b;
    result_mac res;
    res[0] = c%base;
    res[1] = c/base;
}
```

The reason for defining these as functions is so that we can count the number of times they are called. In turn, this will allow us to count the number of operations carried out by the machine while computing the addition and multiplication of multi-precision numbers.

Addition

The procedure for adding two multi-precision numbers is by induction on the length. If $\underline{a} = (a_0, \ldots, a_n)$ and $\underline{b} = (b_0, \ldots, b_n)$ in base B, then

$$\underline{a} + \underline{b} = (a_0 + b_0) + (\underline{a'} + \underline{b'}) \cdot B$$

where $\underline{a'} = (a_1, \ldots, a_n)$ and $\underline{b'} = (b_1, \ldots, b_n)$. Now, $a_0 + b_0$ can be computed as $c_0 + c_1 \cdot B$ where $(c_0, c_1) = \texttt{add\_mac}(a, b)$. So the formula becomes

$$\underline{a} + \underline{b} = c_0 + (\underline{a'} + \underline{b'} + c_1) \cdot B$$

Hence, we need to define the helper function $\texttt{add1}(\underline{a}, \underline{b}, c)$ that adds two multi-digit numbers to a digit c.

Our final addition algorithm *pads* the given numbers so the two numbers $\underline{a}$ and $\underline{b}$ are of the same length. So we define our *real* addition function as follows.

```
define add($\underline{a}$, $\underline{b}$):
  Find the maximum $m$ of lengths of $\underline{a}$ and $\underline{b}$.
  Pad $\underline{a}$ and $\underline{b}$ by 0's on the right to make them of length $m$.
  return add1($\underline{a}$, $\underline{b}$, 0)

define add1($\underline{a}$, $\underline{b}$, $c$):
  $\underline{r}$ = add_mac($a_0$, $b_0$)
  $\underline{s}$ = add_mac($r_0$, $c$)
  $\underline{t}$ = add_mac($r_1$, $s_1$)
  if $\underline{a}$ and $\underline{b}$ are of size 1:
      return ($s_0$, $t_0$)
  return list with first entry $s_0$ and remaining entries from add1($\underline{a'}$, $\underline{b'}$, $t_0$)
```

(See the appendix for detailed implementations in Python and *C*.)

Multiplication

The method to multiply two multi-digit numbers $\underline{a} = (a_0, \ldots, a_p)$ and $\underline{b} = (b_0, \ldots, b_q)$ as we did in school uses a helper function that multiplies $\underline{b}$ with each digit a_i of $\underline{a}$. This uses the formula:

$$\underline{a} \cdot \underline{b} = a_0 \cdot \underline{b} + (a_1 \cdot \underline{b}) \cdot B + \cdots + (a_p \cdot \underline{b}) \cdot B^p = a_0 \cdot \underline{b} + (\underline{a'} \cdot \underline{b}) \cdot B$$

where $\underline{a'} = (a_1, \ldots, a_p)$ as before. Further, we have

$$a_0 \cdot \underline{b} = a_0 \cdot b_0 + (a_0 \cdot \underline{b'}) \cdot B$$

where $\underline{b'} = (b_1, \ldots, b_q)$ as before. Since $a_0 \cdot b_0 = c_0 + c_1 \cdot B$ where $(c_0, c_1) = \texttt{mul\_mac}(a_0, b_0)$ we have the formula

$$a_0 \cdot \underline{b} = c_0 + (a_0 \cdot \underline{b'} + c_1) \cdot B$$

Thus, we actually need the helper function `mul1` which takes a digit a, a multi-digit number $\underline{b}$ and a digit c and produces $a \cdot \underline{b} + c$.

```
**define** mul1($a, \underline{b}, c$):
    $\underline{r} = $ mul_mac($a, \underline{b}_0$)
    $\underline{s} = $ add_mac($r_0, c$)
    $\underline{t} = $ add_mac($r_1, s_1$)
    if $\underline{b}$ is of size 1:
        return $(s_0, t_0)$
    return list with first entry $s_0$ and remaining entries from mul1($a, \underline{b}', t_0$)
```

We now use this to complete the multiplication algorithm as follows:

```
define mul($\underline{a}, \underline{b}$):
    $\underline{r} = $ mul1($a_0, \underline{b}, 0$)
    if $\underline{a}$ is of size 1:
        return $\underline{r}$
    $\underline{s}' = $ mul($\underline{a}', \underline{b}$)
    Extend $\underline{s}'$ by a 0 on the left to get $\underline{s}$.
    return add($\underline{r}, \underline{s}, 0$)
```

Python code

Declare the base for our arithmetic:

```
base=10
```

Code for addition and multiplication of “digits” in the given base:

```
def add_mac(a,b):
    c = a + b
    return c%base, c//base

def mul_mac(a,b):
    c = a * b
    return c%base, c//base
```

The addition algorithm:

```
def add(alpha,beta):
    la,lb = len(alpha),len(beta)
    l = max(la,lb)
    alpha = alpha + (l-la)*[0]
    beta = beta + (l-lb)*[0]
    return add1(alpha,beta,0)

def add1(alpha,beta,c):
    r = add_mac(alpha[0],beta[0])
    s = add_mac(r[0],c)
    t = add_mac(r[1],s[1])
    if len(beta)==1:
        return [s[0], t[0]]
    return [s[0]]+add1(alpha[1:],beta[1:],t[0])
```

The multiplication algorithm, which uses `add`!

```
def mul(alpha,beta):
    r = mul1(alpha[0],beta,0)
    if len(alpha) == 1:
        return r
    s = [0] + mul(alpha[1:],beta)
    return add(r,s)

def mul1(a,beta,c):
    r = mul_mac(a,beta[0])
    s = add_mac(r[0],c)
    t = add_mac(r[1],s[1])
    if len(beta)==1:
        return [s[0], t[0]]
    return [s[0]]+mul1(a,beta[1:],t[0])
```

Check it!

```
mul([2,8],[9,1]))
```

Outputs `[8, 5, 5, 1, 0]` where $82 \cdot 19$ is 1558. So that seems OK!

C Code

Something necessary for the program to print anything!

```
#include <stdio.h>
```

C does not allow us to return an array from a function due to the way memory is managed. So we need to define the result `result_mac` as part of the argument of the function. Since arrays are pointers, we can modify the contents. Otherwise, the logic of these machine simulations is obvious. (Note that `/` returns an integer if the variable being assigned to is an integer; this is called “casting”).

```
int base;
typedef int result_mac[2];

void add_mac(int a, int b, result_mac res) {
    int c;

    c = a + b;
    res[0] = c%base;
    res[1] = c/base;
}
```

```
void mul_mac(int a, int b, result_mac res) {
    int c = a * b;
    res[0] = c%base;
    res[1] = c/base;
}
```

We define a multi-precision number as an array `dat` of fixed (large) size of which only the first `len` entries are relevant. We also define a function that prints such a number in the “usual” order.

```
struct bignum {
    int dat[1024];
    int len;
};

void print_bignum(struct bignum alpha) {
    int i;

    for(i=alpha.len; i>0; ) {
        printf("%d", alpha.dat[--i]);
        }
    printf("\n");
}
```

Now, we do addition. Note that we need to pass a *pointer* to the location where we want the result `ans` to be stored in each case. To avoid copying `alpha` and `beta` we can use some pointer arithmetic, but it can be tricky since we have arrays of fixed sizes and we want to avoid buffer overflows!

```
void add1(struct bignum alpha, struct bignum beta, int c, struct bignum *ans) {
    result_mac r, s, t;
    struct bignum ansp, alphap, betap;
    int i;

    add_mac(alpha.dat[0], beta.dat[0], r);
    add_mac(r[0], c, s);
    add_mac(r[1], s[1], t);

    if (beta.len == 1) {
        ans->dat[0] = s[0];
        ans->dat[1] = t[0];
        ans->len = 2;
    } else {
        for(i=0;i<alpha.len-1;++i) {
            alphap.dat[i] = alpha.dat[i+1];
        }
        alphap.len = alpha.len-1;
        for(i=0;i<beta.len-1;++i) {
            betap.dat[i] = beta.dat[i+1];
        }
        betap.len = beta.len-1;

        add1(alphap, betap, t[0], &ansp);

        ans->dat[0] = s[0];
        for(i=0; i<ansp.len; ++i) {
            ans->dat[i+1] = ansp.dat[i];
        }
        ans->len = ansp.len+1;
    }
}

void add(struct bignum alpha, struct bignum beta, struct bignum *ans) {
    int l = alpha.len;
    if (l < beta.len) {
        l = beta.len;
    }

    int i;
    for (i=alpha.len;i<l;i++) {
        alpha.dat[i] = 0;
    }
    alpha.len = l;
    for (i=beta.len;i<l;i++) {
        beta.dat[i] = 0;
    }
    beta.len = l;
    add1(alpha, beta, 0, ans);
}
```

Similarly for multiplication. Since we are also calculating addition, we need even more local storage for copies. Again, some pointer management can reduce the copying.

```
void mul1(int a, struct bignum beta, int c, struct bignum *ans) {
    result_mac r, s, t;
    struct bignum ansp, betap;
    int i;

    mul_mac(a, beta.dat[0], r);
    add_mac(r[0], c, s);
    add_mac(r[1], s[1], t);
    if (beta.len == 1) {
        ans->dat[0] = s[0];
        ans->dat[1] = t[0];
        ans->len = 2;
    } else {
        for(i=0;i<beta.len-1;++i) {
            betap.dat[i] = beta.dat[i+1];
        }
        betap.len = beta.len-1;

        mul1(a, betap, t[0], &ansp);

        ans->dat[0] = s[0];
        for(i=0; i<ansp.len; ++i) {
            ans->dat[i+1] = ansp.dat[i];
        }
        ans->len = ansp.len+1;
    }
}

void mul(struct bignum alpha, struct bignum beta, struct bignum *ans) {
    struct bignum ansp, ansq, ansr, alphap;
    int i;

    mul1(alpha.dat[0], beta, 0, &ansp);

    if (alpha.len == 1){
        for(i=0;i<ansp.len;++i) {
            ans->dat[i]=ansp.dat[i];
        }
        ans->len=ansp.len;
    } else {
        for(i=0;i<alpha.len-1;++i) {
            alphap.dat[i] = alpha.dat[i+1];
        }
        alphap.len = alpha.len-1;

        mul(alphap,beta,&ansq);

        ansr.dat[0] = 0;
        for(i=0;i<ansq.len;++i) {
            ansr.dat[i+1] = ansq.dat[i];
        }
        ansr.len = ansq.len + 1;

        add(ansp, ansr, ans);
    }
}
```

Finally, we define the `main` which assigns values for `base`, `alpha` and `beta`.

```
int main(){
    struct bignum alpha, beta, result;

    base = 10;

    alpha.dat[0] = 6;
    alpha.dat[1] = 8;
    alpha.dat[2] = 1;
    alpha.len = 3;

    beta.dat[0] = 9;
    beta.dat[1] = 3;
    beta.dat[2] = 7;
    beta.len = 3;

    print_bignum(alpha);
    print_bignum(beta);

    add(alpha,beta,&result);

    print_bignum(result);

    mul(alpha,beta,&result);

    print_bignum(result);
}
```