Lecture 5
Arithmetic and Algebra

Euiseong Seo
(euiseong@skku.edu)
Numbers

- **Value ranges of a number data type**
  - Limited by the size of the data type
  - Usually 32bit or 64bit
  - Extremely large numbers

- **Arbitrary precision or arbitrarily big numbers**
  - Rarely we need
  - Data structures and arithmetic operations
Representation of Arbitrary Precision Integers

- **Arrays**

- **Linked Lists**
Bignum Data Type

- Sign and magnitude
- Reverse order

```c
#define MAXDIGITS 100 /* maximum length bignum */
#define PLUS 1 /* positive sign bit */
#define MINUS -1 /* negative sign bit */

typedef struct {
    char digits[MAXDIGITS]; /* represent the number */
    int signbit; /* PLUS or MINUS */
    int lastdigit; /* index of high-order digit */
} bignum;
```
print_bignum(bignum *n)
{
    int i;

    if (n->signbit == MINUS) printf("- ");
    for (i=n->lastdigit; i>=0; i--)
        printf("%c",’0’+ n->digits[i]);

    printf("\n");
}
## Addition

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Addition</th>
<th>Subtraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) + (B)</td>
<td>(A+B)</td>
<td>A&gt;B</td>
</tr>
<tr>
<td>(A) + (-B)</td>
<td>(A-B)</td>
<td>-(B-A)</td>
</tr>
<tr>
<td>(-A) + (B)</td>
<td>(-A-B)</td>
<td>+B-A</td>
</tr>
<tr>
<td>(-A) + (-B)</td>
<td>-B</td>
<td>(A-B)</td>
</tr>
<tr>
<td>(A) - (B)</td>
<td>(A-B)</td>
<td>-(B-A)</td>
</tr>
<tr>
<td>(A) - (-B)</td>
<td>(A+B)</td>
<td>(B-A)</td>
</tr>
<tr>
<td>(-A) - (B)</td>
<td>(-A-B)</td>
<td>(A-B)</td>
</tr>
<tr>
<td>(-A) - (-B)</td>
<td>-B</td>
<td>(A-B)</td>
</tr>
</tbody>
</table>
Addition

```c
add_bignum(bignum *a, bignum *b, bignum *c)
{
    int carry;  /* carry digit */
    int i;     /* counter */

    initialize_bignum(c);

    if (a->signbit == b->signbit) c->signbit = a->signbit;
    else {
        if (a->signbit == MINUS) {
            a->signbit = PLUS;
            subtract_bignum(b, a, c);
            a->signbit = MINUS;
        } else {
            b->signbit = PLUS;
            subtract_bignum(a, b, c);
            b->signbit = MINUS;
        }
        return;
    }

    c->lastdigit = max(a->lastdigit, b->lastdigit) + 1;
    carry = 0;

    for (i=0; i<=(c->lastdigit); i++) {
        c->digits[i] = (char)
            (carry+a->digits[i]+b->digits[i]) % 10;
        carry = (carry + a->digits[i] + b->digits[i]) / 10;
    }

    zero_justify(c);
}
```
Zero Justification

Zero Justification operates as follows:

- **Adjust lastdigit to avoid leading zeros**
- **Corrects -0**
- **Call after every arithmetic**

```c
zero_justify(bignum *n)
{
    while ((n->lastdigit > 0) && (n->digits[n->lastdigit] == 0))
        n->lastdigit --;

    if ((n->lastdigit == 0) && (n->digits[0] == 0))
        n->signbit = PLUS; /* hack to avoid -0 */
}
```
Subtraction

Subtract_bignum(bignum *a, bignum *b, bignum *c) {
    int borrow;  /* anything borrowed? */
    int v;      /* placeholder digit */
    int i;      /* counter */

    if ((a->signbit == MINUS) || (b->signbit == MINUS)) {
        b->signbit = -1 * b->signbit;
        add_bignum(a,b,c);
        b->signbit = -1 * b->signbit;
        return;
    }
    if (compare_bignum(a,b) == PLUS) {
        subtract_bignum(b,a,c);
        c->signbit = MINUS;
        return;
    }
    c->lastdigit = max(a->lastdigit,b->lastdigit);
    borrow = 0;
    for (i=0; i<=(c->lastdigit); i++) {
        v = (a->digits[i] - borrow - b->digits[i]);
        if (a->digits[i] > 0)
            borrow = 0;
        if (v < 0) {
            v = v + 10;
            borrow = 1;
        }
        c->digits[i] = (char) v % 10;
    }
    zero_justify(c);
}
Comparison

```c
compare_bignum(bignum *a, bignum *b)
{
    int i; /* counter */

    if ((a->signbit==MINUS) && (b->signbit==PLUS)) return(PPLUS);
    if ((a->signbit==PLUS) && (b->signbit==MINUS)) return(MINUS);
    if (b->lastdigit > a->lastdigit) return (PLUS * a->signbit);
    if (a->lastdigit > b->lastdigit) return (MINUS * a->signbit);

    for (i = a->lastdigit; i>=0; i--) {
        if (a->digits[i] > b->digits[i])
            return(MINUS * a->signbit);
        if (b->digits[i] > a->digits[i])
            return(PLUS * a->signbit);
    }
    return(0);
}
```
Multiplication

```c
multiply_bignum(bignum *a, bignum *b, bignum *c)
{
    bignum row;    /* represent shifted row */
    bignum tmp;    /* placeholder bignum */
    int i,j;       /* counters */

    initialize_bignum(c);

    row = *a;

    for (i=0; i<=b->lastdigit; i++) {
        for (j=1; j<=b->digits[i]; j++) {
            add_bignum(c,&row,&tmp);
            *c = tmp;
        }
        digit_shift(&row,1);
    }

    c->signbit = a->signbit * b->signbit;
    zero_justify(c);
}
```
Digit Shift

```c
digit_shift(bignum *n, int d) /* multiply n by 10^d */
{
    int i; /* counter */

    if ((n->lastdigit == 0) && (n->digits[0] == 0)) return;

    for (i=n->lastdigit; i>=0; i--)
        n->digits[i+d] = n->digits[i];

    for (i=0; i<d; i++) n->digits[i] = 0;

    n->lastdigit = n->lastdigit + d;
}
```
Exponentiation

- Repeated multiplication
- Reduce the number of multiplications

\[ a^n = a^{n/2} \times a^{n/2} \times (a^{n\%2}) \]
Division

By repeated subtractions

divide_bignum(bignum *a, bignum *b, bignum *c)
{
    bignum row;       /* represent shifted row */
    bignum tmp;       /* placeholder bignum */
    int asign, bsign; /* temporary signs */
    int i,j;          /* counters */

    initialize_bignum(c);

    c->signbit = a->signbit * b->signbit;

    asign = a->signbit;
    bsign = b->signbit;

    a->signbit = PLUS;
    b->signbit = PLUS;

    initialize_bignum(&row);
    initialize_bignum(&tmp);

    c->lastdigit = a->lastdigit;
Division

for (i=a->lastdigit; i>=0; i--) {
    digit_shift(&row,1);
    row.digits[0] = a->digits[i];
    c->digits[i] = 0;
    while (compare_bignum(&row,b) != PLUS) {
        c->digits[i] ++;
        subtract_bignum(&row,b,&tmp);
        row = tmp;
    }
}

zero_justify(c);

a->signbit = asign;
b->signbit = bsign;

- Remainder of \(a/b\) is \(a - b(a/b)\)
Base Conversion

- Converting base a number x to base b number y
- Two approaches
  - Left to right
    - MSD of y is $d_i$ such that
      \[(d_i + 1)b^k > x \geq d_i b^k\]
    - $1 \leq d_i \leq b-1$
  - Right to left
    - LSD of y is the remainder of x divided by b
Polynomials

- **Data structures**
  - Array of coefficients
  - Linked list of coefficients

- **Evaluation**
  - Naïve approach
  - Horner’s Rule
    
    \[ (((a_n x + a_{n-1}) x + \ldots) x + a_0 \]
Reverse and Add

The reverse and add function starts with a number, reverses its digits, and adds the reverse to the original. If the sum is not a palindrome (meaning it does not give the same number read from left to right and right to left), we repeat this procedure until it does.

For example, if we start with 195 as the initial number, we get 9,339 as the resulting palindrome after the fourth addition:

\[
\begin{array}{cccc}
195 & 786 & 1,473 & 5,214 \\
591 & 687 & 3,741 & 4,125 \\
+ & + & + & + \\
786 & 1,473 & 5,214 & 9,339 \\
\end{array}
\]

This method leads to palindromes in a few steps for almost all of the integers. But there are interesting exceptions. 196 is the first number for which no palindrome has been found. It has never been proven, however, that no such palindrome exists.

You must write a program that takes a given number and gives the resulting palindrome (if one exists) and the number of iterations/additions it took to find it.

You may assume that all the numbers used as test data will terminate in an answer with less than 1,000 iterations (additions), and yield a palindrome that is not greater than 4,294,967,295.
Reverse and Add

Input
The first line will contain an integer $N$ ($0 < N \leq 100$), giving the number of test cases, while the next $N$ lines each contain a single integer $P$ whose palindrome you are to compute.

Output
For each of the $N$ integers, print a line giving the minimum number of iterations to find the palindrome, a single space, and then the resulting palindrome itself.

Sample Input
3
195
265
750

Sample Output
4  9339
5  45254
3  6666
Given any integer $0 \leq n \leq 10,000$ not divisible by 2 or 5, some multiple of $n$ is a number which in decimal notation is a sequence of 1’s. How many digits are in the smallest such multiple of $n$?

**Input**

A file of integers at one integer per line.

**Output**

Each output line gives the smallest integer $x > 0$ such that $p = \sum_{i=0}^{x-1} 1 \times 10^i$, where $a$ is the corresponding input integer, $p = a \times b$, and $b$ is an integer greater than zero.
<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>9901</td>
<td>12</td>
</tr>
</tbody>
</table>
A Multiplication Game

Stan and Ollie play the game of multiplication by multiplying an integer \( p \) by one of the numbers 2 to 9. Stan always starts with \( p = 1 \), does his multiplication, then Ollie multiplies the number, then Stan, and so on. Before a game starts, they draw an integer \( 1 < n < 4, 294, 967, 295 \) and the winner is whoever reaches \( p \geq n \) first.

**Input**

Each input line contains a single integer \( n \).

**Output**

For each line of input, output one line – either

Stan wins.

or

Ollie wins.

assuming that both of them play perfectly.
A Multiplication Game

*Sample Input*

162
17
34012226

*Sample Output*

Stan wins.
Ollie wins.
Stan wins.