Joseph A. Boyle

This program can perform math operations on arbitrarily large numbers. Given four arguments: **operand** (**+**/**-**/*****/**^**), **number1**, **number2**, and **returnFormat**, the program will output the resulting number, in returnFormat base, of **number1 operand number2**. For example:

Command: ./calc * d9 b101011 h
h9 * h2B : h183

Where the first line signifies the mathematical representation of the command, and the second line signifies the result. *Note that this program can handle **arbitrarily long** numbers. The following three commands will return correct results:*

Command	Interpretation	Time to execute
./calc \^ h9 hFFFF d	9^{65535}	50.52 seconds
./calc \+ hFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF hFFFFFFFFFFFFFFFFFFF d	3923188584616675477397368389504791510063972152790021570 5 + 7555786372591432341913 5	0.02 seconds
./calc * hFFFF hFFFFF d	65535 * 1048575	0.04 seconds

Expected Input Formats

The program accepts inputs in the following format:

opsign(+, -,*, or ^) **number1**(-?(b\|o\|d\|x)d_n...d_0) **number2**(-?(b\|o\|d\|x)d_n...d_0) **outputBase**(b, o , d, or x)

It is expected that all numbers, regardless of base, are positive, unless prefaced by a -sign. That is, we interpret hFFFFFFFF = b11111111111111111111111111111111, which is positive 2^{32}_{10}, while -hFFFFFFFF = b11111111111111111111111111111111 , which equals -2^{32}_{10}.

Here are several inputs and what the program interprets them to mean:

Input	Interpretation	Result
+ hFF -hFF	hFF + (-hFF)	h0
+ b1101 b1	b1101 + b1	b1110
- -b1101 -b1	-b1101 + b1	-b1100

Internal Number Representation

The program defines a struct **Number**, which holds an internal unsigned char array, *rep*. The length of this array is stored in an integer, *capacity*. We represent whether this number is negative with an integer, *negative*, an integer *base* to signify the base, and store an integer,*digits*, to signify the number of digits in the number. Since individual digits of hex numbers, the largest base handled by the program, can be represented by just 4 bits, we hold that a single unsigned char in a uchar array can hold two digits. That is, we store two digits, *n* and *m* (left and right, respectively), in one element, *ch*, of the unsigned char array, via bitshifting and other logical expressions.

Since we restrict program inputs to **INTEGERS** (thus no decimal numbers), it makes logical sense to store numbers backwards (that is, 123_{10} is stored as 321), as integers will never change past the ones place via mathematical operations. The advantage of this design is the ease of adding leading zeros and dynamically resizing our internal array, which require O(1) and O(N) time, respectively (though resizing the internal array is handled by realloc, an optimized compiler construct). Thus, the number 12345_{10} in memory looks like (just the first 3 elements of rep):

Decimal	**5**	**4**	**3**	**2**	**1**	**0**
Binary Rep	0101	0100	0011	0010	0001	0000
Array Index	0		1		2	
Array[Index]	01010100		00110010		00010000	

Setting digits of a Number is done via the **setDigit** method, which works as follows:
Input: A Number struct, *number*, the integer to insert (<= 15), *num*, and the digit position, *digit*

1. If *digit* greater than or equal to capacity * 2, resize the number.
2. Set **key** to *digit*/2, and **ch** to rep[key].
3. If *digit* modulo 2 is 0, set the left 4 bits of **ch** to *num*:
    a. **ch** = (**ch** & 15) | (*num* << 4)
4. Otherwise, set the right 4 bits of **ch** to *num*:
    a. **ch** = (**ch** & 240) | (*num*)
5. If *digit* is greater than number->digits, set number->digits to *digit* + 1

Fetching a digit, via **getDigit**, works similarly, but returns the value as follows:
Input: A Number struct, *number*, and the digit position, *digit*

1. If *digit* greater than or equal to *number->digits*, return -1.
2. Set **key** to *digit* / 2, and **ch** to rep[key].

3. If *digit* modulo 2 is 0, return the left 4 bits of **ch**:
    a. return **ch** & 240
6. Otherwise, return the right 4 bits of **ch**:
    a. return **ch** & 15

Operations

All operations, save for base conversion, assumes that the inputs are all of the same base. Assuming the two input numbers are the same base, math is performed in the given base. That is, the same function that handles the addition of two binary numbers handles the addition of two hexadecimal numbers. Since base conversion is the most expensive step of the program, we convert the input Numbers to the desired output base, thus producing an output of the same base.

Converting Base

Since addition and multiplication, as defined following this section, assume that the numbers they are working on are in the same base, we are able to use them to convert arbitrarily long numbers between bases. The algorithm essentially looks as follows:
Input: A Number, *number*, and the base we wish to convert to, *base*

1. Set **result** to a Number equal to zero, and **power** to 0.
2. Set **i** to 0 and iterate through each digit in *number*
    a. Set **intermediate** to the value of the i^{th} digit of *number* in *base*
    b. Set **power** to the i^{th} power of *base*
    c. Set **intermediate** to (**intermediate** * **power**)
    d. Set **result** to (**intermediate** + **result**)
3. Return **result**

Addition

Addition runs in O(N) time. The algorithm to do so is as follows:
Input: Two numbers to add, *number1* and *number2*

1. Create a number, **result**, which is equal to whichever is bigger - *number1* or *number2*
2. Create a number, **smaller**, which is equal to whichever is smaller - *number1* or *number2*
3. If **smaller** and **result** both have the same sign, set **answerNegative** to their sign.
4. Otherwise, set **answerNegative** to **result**'s sign.
5. Set **carry** to 0
6. Set *i* to 0, iterate through all digits in **result**
    a. Set **sum** to the i^{th} digit of **result** + the i^{th} digit of **smaller** + **carry**
    b. If **sum** is greater than the base, subtract base from sum and set carry to base.
    c. Otherwise, If **sum** is negative:
        i. If *i* is the last digit of **result**, the answer is negative. Return the answer.
        ii. Otherwise, subtract one from the $(i+1)^{th}$ digit and add base to **sum**.
    d. Otherwise, set **carry** to 0.
    e. Set the i^{th} digit of **result** to **sum**.

7. Return **result**

Subtract

Subtract uses the predefined addition function, but sets *number2*->negative to the opposite of its current value (*ie*: 1 if currently 0, or 0 if currently 1), and then calls addition. This effectively looks like:

number1 - number2 = result

number1 + (-number2) = result

Thus, subtract runs in O(N) time, as addition does.

Multiply

Multiplying two numbers, *number1* and *number2*, runs in O(N^2) time, as it uses the traditional multiplication algorithm we are used to: Starting the first digit of *number1*, multiply by every digit in *number2*, and then add resulting number to the running result. We then move to the next digit of *number2*, and repeat the multiplication of each digit. Shift this value over *(i - 1)* times (as we do when multiplying by hand), and then add it to the running result. Repeat this for every digit in *number1*.

This algorithm runs trivially fast for most numbers with less than a couple hundred digits. For each n digits multiplied, there are n addition calls, which is inherently the slowest part of the algorithm.

Power

Power takes two numbers, *number1* and *number2*, and will return $number^{number2}$, which effectively involves multiplying *number1 * number1*, *number2* times. Thus, this algorithm runs in O(*number2* * N^2). In effect, for *number2* which is equal to *number1*, this requires O(N^3) computations. This runs trivially fast for powers less than 1000. As previously described, $9^{FFFF}{}_{16}$ takes approximately 50 seconds to compute, as it is 51,936 digits long.

Other Efficiencies

To compute $9^{FF}{}_{16}$ requires just over 4,000 allocations (237,703 bytes), which is fairly expensive. The original design of this system required 30,000 allocations (7,817,747 bytes), as it used integers to represent digits and stored numbers forwards instead of backwards. For every Number that is created, at least 70 bytes of data is allocated, as the internal char array defaults to 64 bytes. If we need a longer internal array, the array is expanded by an additional 64 bytes. In effect, every reallocation gives us another 128 digits (2 digits per character * 64 characters). Thus, many small numbers tend to have much larger arrays than needed. As often as possible, small numbers are stored in memory and reused throughout program functionality, so as to save on that used space. We elect to add a constant amount of space to a realloced array so as to save on malloc calls, which have significant overhead. The memory usage is roughly O(N^2) for all operations, where N is the sum of the number of digits in *number1* and *number2.*

The previous architecture required over six minutes to compute $9^{FFFF}{}_{16}$, whereas the current architecture requires only 50 seconds to compute the 51,936 digit number. For small numbers (ie: those with fewer than 300 digits), computations are arbitrarily fast. To demonstrate this, consider the addition of FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF$_{16}$ and FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF$_{16}$ in binary (309 digits). The computation requires less than 0.05 seconds. Computing the number squared takes only 0.07 seconds (624 digits).

To further enhance the system would require more efficient multiplication algorithms. There exist several algorithms which can bring the multiplication runtime down to approximate $O(N^{1.6})$, though are too complicated to implement in the scope of this project, as they would require a division function.