The University of Melbourne
School of Computing and Information Systems
COMP10002 Foundations of Algorithms
Semester 1, 2018
Assignment 1
Due: 4pm Tuesday 24th April 2018

1 Learning Outcomes

In this assignment, you will demonstrate your understanding of arrays, pointers, input processing, and functions. You will also extend your skills in terms of code reading, program design, testing, and debugging.

2 The Story...

On 2 December 2014, a post appeared on YouTube's Google+ account[1]: "'Gangnam Style' has been viewed so many times we have to upgrade!" It turned out that the Korean rapper PSY's viral music video had been viewed for over 2,147,483,647 ($2^{31} - 1$) times which surpassed the limit of a signed 32-bit integer counter. YouTube had since upgraded to a 64-bit integer counter. This upgraded counter would not be broken again in a foreseeable future. However, in many other scenarios such as popular cryptography algorithms (e.g., RSA), even larger integers are still needed.

In C, an `int` variable can only store values up to $2^{31} - 1$. A `long` variable can handle up to $2^{63} - 1 = 9,223,372,036,854,775,807$ (machine dependant). For even larger numbers, you will have to use `double`, which is inaccurate. In this assignment, you will enhance the C language by supporting "*huge*" integers.

3 Your Task

Your task is to develop a huge integer calculator that works with integers of **up to 100 digits**. The calculator has a simple user interface, and 10 "variables" (`n0, n1, ..., n9`) into which integers can be stored. For example, a session with your calculator might look like:

```
mac: ./assmt1
> n0=2147483648
> n0+3
> n0?
2147483651
> n1=1000000000000000000
> n1+n0
> n1?
1000000002147483651
> n0?
2147483651
> exit
mac:
```

Note: "`mac: `" is the terminal prompt and "`> `" is the prompt of the calculator.

[1]https://plus.google.com/+YouTube/posts/BUXfdWqu86Q

The calculator to be implemented has a very limited syntax: constants or other variables can be assigned to variables using an '=' operator; variable values can be altered using a single operator and another variable or constant; variable values can be printed using '?'. Each line of input command always starts with a variable followed by a single operator, which is then followed by (optional depending on the operator) another variable or a constant. The only exception is the "exit" command, which has no additional parameters.

To allow storage of huge integers, your calculator needs to use arrays of integers, with one digit stored per element in the array. The number of digits in a huge integer needs to be stored as well. We assume a maximum of **100 digits** in a huge integer, and use a "reverse order" representation. For example, 123 will be represented by:

```
#define INT_SIZE 100
typedef int huge_t[INT_SIZE];

huge_t var_n0 = {3, 2, 1};
int var_len0 = 3;
```

Here, array var_n0 stores the digits of 123 in a reversed way. The reverse order representation will make calculations such as addition and multiplication easier. However, if you wish, you are free to change how the digits are stored in the array. By using struct we can further simplify the representation, but you are *not* required to do so as it has not been covered yet.

The expected input format for all stages is identical - a sequence of simple commands as shown above. **You may assume that the input is always valid, that is, you do *not* need to consider input errors. All input integers are unsigned.** You do *not* need to consider negative numbers or subtraction. There will *not* be "n0=+0", "n0=+123", "n0=-123", "n0-123", or "n0++123" in the input. It will just be in the forms of, e.g., "n0=0", "n0=123", "n0+123", or "n0*123". **There will *not* be leading 0's in the input numbers, e.g.,** 001.

You will be given a skeleton code file named "assmt1.c" for this assignment in LMS. The skeleton code file contains a main function where a loop is used to continuously read in user commands, and calls relevant functions to process the commands. The exit function has been implemented already, which can handle the exit command if you compile and run the skeleton code. The echo function for the '?' operator to print out a huge integer has been given to you for reference as well, but you need to implement the init function to initialise the variables first before echo can work properly. All the other function bodies are empty. Your task is to add code into them to process the other calculator commands. **Note that you should *not* change the main function, but you are free to modify any other parts of the skeleton code (including adding more functions)**. You can change echo as well if you wish to change how a huge integer is stored in an array.

3.1 Stage 1 - Getting Started (Up to 3 Marks)

Your first task is to understand the skeleton code. Note the use of the type huge_t in the skeleton code. Each of the huge_t variables in the array vars stores a huge integer, with 10 variables available in total, named "n0", "n1", ..., "n9", respectively. In this stage, you will start with implementing the init function to initialise the variable values to be 0. A sample execution session of this stage is shown below.

```
> n0?
0
> n8?
0
```

3.2 Stage 2 - Reading in a Huge Integer (Up to 8 Marks)

Next, you will implement the assign function to enable assigning a constant value to a variable, or the value of a variable to another variable (both could be the same variable, e.g., "n0=n0"). A sample execution session of this stage is shown below.

```
> n0=2147483648
> n0?
2147483648
```

```
> n1=n0
> n1?
2147483648
```

You should plan carefully, rather than just leaping in and starting to edit the skeleton code. Then, before moving through the rest of the stages, you should test your program thoroughly to make sure its correctness.

3.3 Stage 3 - Adding up Two Huge Integers (Up to 13 Marks)

Now add code to the `add` function to enable adding up two huge integers. Note that the '`+`' operator always starts with a variable, e.g., "n0", followed by '`+`' itself. After that there are two cases: an integer constant or another variable. Your code should be able to handle both cases: adding up a variable with a constant value, and adding up two variables (both could be the same variable). *The sum should always be stored in the starting variable of the command.* A sample execution session of this stage is shown below.

```
> n0=2147483648
> n0+1
> n0?
2147483649
> n0+2
> n0?
2147483651
> n1=10000000000000000000000000000000000000000
> n1+n0
> n1?
10000000000000000000000000000002147483651
> n0?
2147483651
```

Note that we assume integers with up to 100 digits. They can still overflow if we are adding up even larger numbers. In this case, we will simply ignore the overflowed digits and keep the remaining part in the sum. **You should also remove any leading `0`'s in the sum, i.e., your program should not produce numbers like `0001` (which should be `1`).**

3.4 Stage 4 - Multiplying Two Huge Integers (Up to 15 Marks)

The next operator to add is '`*`'. You will need to modify the `multiply` function for this stage. When this stage is done, your program should be able to process the following sample input.

```
> n0=123
> n1=4567
> n0*n1
> n0?
561741
> n1?
4567
> n2=100000000000000000000000000000000000000000000000000
> n2*100000000000000000000000000000000000000000000000000
> n2?
0
```

When there is an overflow, we follow the same procedure as in addition. That is, to keep the non-overflowed part only. **You should also remove any leading `0`'s in the product.**

You need to think carefully about the *algorithm* to use in this task. You may implement the process to do long multiplication that you learned at school. Some supporting functions may be needed. For example, a function `multiply_digit` that takes a `huge_t` variable and multiplies it by a single-digit integer may be useful; as might a function `multiply_base` that multiplies a `huge_t` variable by 10.

There are also other (more efficient) ways to do multiplication, and you can be creative if you wish. *Just do not try and do it by repeated addition (penalties apply).*

3.5 Stage 5 - Supporting the "Power of" Operator (Up to 15 Marks)

This stage is for a challenge. Please do not start on this stage until your program through to Stage 4 is (in your opinion) perfect, and is submitted, and is verified, and is backed up.

For a challenge, implement the `power` function for the "power of" operator '^' (no need to consider 0^0):

```
> n0=2
> n0^4
> n0?
16
> n1=2
> n1^n0
> n1?
65536
```

Hint: You may use repeated multiplications for this stage.

If you are successful in this stage, you will be able to earn back 1 mark you lost in the earlier stages (assuming that you lost some). Your total mark will not exceed 15.

4 Submission and Assessment

This assignment is worth 15% of the final mark. A detailed marking scheme will be provided on the LMS.

You need to submit all your code in one file named `assmt1.c` for assessment; detailed instructions on how to submit will be posted on the LMS once submissions are opened. Submission will NOT be done via the LMS; instead you will need to log in to a Unix server and submit your files to a software system known as `submit`. You can (and should) use submit both **early and often** - to get used to the way it works, and also to check that your program compiles correctly on our test system, which has some different characteristics to the lab machines. Only the last submission made before the deadline will be marked.

You will be given a sample test file `test0.txt` and the sample output `test0-output.txt`. You can test your code on your own machine with the following command and compare the output with `test0-output.txt`:

```
mac: ./assmt1 < test0.txt    /* Here '<' feeds the data from test0.txt into assmt1 */
```

You may discuss your work with others, but what gets typed into your program must be individual work, **not** copied from anyone else. Do **not** give hard copy or soft copy of your work to anyone else; do **not** "lend" your memory stick to others; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "no" when they ask for a copy of, or to see, your program, pointing out that your "no", and their acceptance of that decision, is the only thing that will preserve your friendship. *A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode.* See https://academichonesty.unimelb.edu.au for more information.

Deadline: Programs not submitted by **4pm Tuesday 24th April 2018** will lose penalty marks at the rate of two marks per day or part day late. Late submissions after 4pm Friday 27th April 2018 will **not** be accepted. Students seeking extensions for medical or other "outside my control" reasons should email the lecturer at jianzhong.qi@unimelb.edu.au. If you attend a GP or other health care professional as a result of illness, be sure to take a Health Professional Report form with you (get it from the Special Consideration section of the Student Portal), you will need this form to be filled out if your illness develops into something that later requires a Special Consideration application to be lodged. You should scan the HPR form and send it in connection with any non-Special Consideration assignment extension requests.

And remember, *Algorithms are fun!*