Courseworks 2 and 3

Due: April 1, and May 9, 2011

These coursework assignments are intended to test knowledge of more complex parts of C. Coursework 2 is worth 25%, and is intended to test:

- Familiarity with the C build process, including use of multiple compilation units
- The use of C structures, and dynamic memory allocation (including cleanup)
- The handling of command-line arguments
- The ability to create abstract data types in C
- The ability to follow a set of coding conventions

Coursework 3 is worth 10% and is designed to test your ability to write fast C code. I will prepare a sample solution to coursework 3 and a set of test inputs. Any submission that works in a reasonable time will get at least 40%. Any submission that matches the speed of my solution will be awarded 70%. Submissions that run faster than my solution will be awarded more than 70%, depending on how much faster they are.

1 Coding Conventions

If you write code in any language as part of a larger project, you will be expected to conform to a set of coding conventions. 20% of the marks in Coursework 2 are awarded for code conforming to the set of rules described in this section.

1.1 Indenting

Each new depth of nested block should be indented by one tab. The braces around each block should be on their own line, at the same indent level as the preceding line. All conditionals and loops should be followed by a new block, even if they only contain one statement. For example:

```
void function(void)
{
    // Indent is one tab here
    if (someCondition)
    {
        // Indent is two tabs here
        doStuff();
    }
}
```

1.2 Spacing

Insert blank lines between related groups of statements, to visually group the set of statements. When calling a function, there should be no space between the function name and the call operator, but parenthetical expressions after a language keyword must have a space separating them. For example:

```
if(x) // wrong!
{
    if (y) // Correct
    {
        doStuff(); // Correct
        thenMoreStuf (); //Wrong!
    }
}
```

There should be no space before the semicolon, or other punctuation, but there should be space after each comma. Spacing within expressions should mirror the operator precedence.

When declaring pointers, the `*` should be adjacent to the variable. When using pointer types in other contexts, it should be adjacent to the type name. For example:

```
int* example(void)
{
    int *a;
    a = (int*)b;
```

1.3 Comments

Comments should describe why things are being done, not what. All functions and data structures should be declared with documentation comments (starting `/**`) explaining their purpose. For example:

```
/**
 * Structure for storing a point, using 2D Cartesian
    coordinates.
 */
struct Point
{
    /** X (horizontal) coordinate. */
    float x;
    /** Y (vertical) coordinatel. */
    float y;
};
/**
 * Determine whether two points intersect.  Returns 1 if
    they do.
 */
 int pointsIntersect(Point p1, Point p2);
```

1.4 Naming Conventions

Function names should start with a lowercase letter and describe the expected behaviour of the function. Global variables and structure names should begin with an uppercase letter. Type definitions that refer to pointer types should use the `_t` suffix. Preprocessor macros should be entirely in uppercase. For example:

```
typedef struct Point* point_t;
```

2 Coursework 2

The goal of this assignment is to write a simple arbitrary-length integer implementation. You must submit four files:

- A Makefile used to build the project (see the end of the hand-out notes for how to create a Makefile)
- A header (.h) file containing the interface to the integer manipulation functions
- A source (.c) file containing the implementation of the integer manipulation functions
- A source (.c) file containing a test program

Both of the source files will need to include the header file, allowing the test program to call the functions that you have written to manipulate your arbitrary-length integers.

The header file should contain preprocessor guards against multiple inclusion. It should also declare the following (or equivalent) types and functions (along with documentation comments):

```c
typedef struct BigInteger* big_integer_t;

big_integer_t bigIntFromString(const char *str);
void freeBigInt(big_integer_t a);

big_integer_t add(big_integer_t a, big_integer_t);
big_integer_t subtract(big_integer_t a, big_integer_t);
big_integer_t multiply(big_integer_t a, big_integer_t);
big_integer_t divide(big_integer_t a, big_integer_t);
```

The first two functions are used to construct large integers from strings and to free them. Because arbitrary-length integers can have any size, this will require some dynamic allocation.

Each of the four remaining functions will perform the associated arithmetic operation, returning a new big integer. You will need to use the long addition/subtraction/multiplication/division algorithms that you learned as a small child. If you have forgotten how these worked, then find a maths textbook aimed at 7-year-olds to refresh your memory.

The exact representation used for storing the integers is up to you. I would suggest splitting them into smaller values that C can do arithmetic on. For example, treating them as being comprised for 32-bit 'digits'.

If you are stuck, ask in the lectures.

The test program should take three command-line arguments and perform the associated operation, printing the result. For example:

```
$ bigintmaths 1234567890123456789 * 47
58024690835802469083
$ bigintmaths 58024690835802469083 / 47
1234567890123456789
```

2.1 Marking Scheme

Marks will be awarded as follows:

20% for conforming to the coding conventions

10% for submitting code with the correct structure

10% for correctly constructing and freeing big integers

10% for a working test program

10% for correctly implementing addition

10% for correctly implementing subtraction

15% for correctly implementing multiplication

15% for correctly implementing division

3 Coursework 3

Using your big integer code from Coursework 2, write a program that takes a single command-line argument and prints all of its prime factors, or the string 'PRIME' if it is prime.

This coursework is assessed entirely on the performance of your code. Performance tests will be run on one of the Linux lab machines, which has a 3GHz Core 2 Duo processor. You may use any of the techniques discussed in lectures, or anything you have read about, to make your code faster.