# COMP250 Assignment 1

Adam Hooper <adam.hooper@mail.mcgill.ca>

## Table of Contents



# Question 1

This question was relatively simple: instead of trying to modify the given recursive algorithm, it's much easier to go back to the definition of the multiplication, as it was given in class. Keep a running sum; divide `b` by two every time `a` is multiplied by two, and add the remainder to the sum all the time. Once `b` is `0`, the sum is the answer.

Keeping in mind how binary numbers are stored, there's some farther simplification: Instead of just using integer division and multiplication, use bitshifting instead. Not only is it much faster way of dividing or multiplying by two, but the algorithm works perfectly with both positive and negative numbers.

Below is the program listing.

```
public static long mulalarusse (long a, long b)
{
        long result = 0;

        while (b != 0)
        {
                if ((b & 1) == 1)
                {
                        result += a;
                }

                a <<= 1;
                b >>>= 1;
        }

        return result;
}
```

# Question 2

First of all, two `mulmod` methods must be created: one based on the algorithm given, and another done the most natural way. Call them `mulmod_java` and `mulmod`, respectively. The latter is an obvious

one-liner, and the former is simply a transcription of the given pseudo-code to Java (though bit-shifting was used instead of division and subtraction/division, for simplicity's sake).

I've included a method called `addmod_java`, which simply returns `(a + b) % n`. This method will be edited in the section called “Question 3”.

```
public static long addmod_java (long a, long b, long n)
{
        // Return something impossible if Java can't go that high
        if (Long.MAX_VALUE - a - b < 0) return -1;

        return (a + b) % n;
}

public static long mulmod_java (long a, long b, long n)
{
        return (a * b) % n;
}

public static long mulmod (long a, long b, long n)
{
        long a_doubled;

        if (b == 0) return 0; // a * 0 = 0 %n

        a_doubled = addmod_java (a, a, n);

        if (a_doubled < 0) return -1;

        if ((b & 1) == 0)
        {
                return mulmod (a_doubled, b >>> 1, n);
        }
        else
        {
                return addmod_java (a, mulmod (a_doubled, b >>> 1, n), n);
        }
}
```

The question next asks for the largest value of n for which `mulmod (a, b, n)` is correct, given `0 # a, b # n - 1`. Obviously, the most challenging situation is when the largest product possible is reached: in other words, when `a = b = n - 1`. The problem is that normally, a product which reaches or exceeds $2^{63}$ would overflow the `long` data type, losing data. The `mulmod` algorithm circumvents this to a great extent, but does not remove the possibility altogether.

Perhaps the easiest way to find the largest possible value of `n` for which the `mulmod` method works is brute force. When a negative number is returned, there was obviously an overflow; this is how we know whether a given calculation is valid or not. Unfortunately, we can't just write the entire thing in a `for` loop iterating from `1` to `9223372036854775807` because even on modern computers the program would take longer to run than the age of the universe.

The goal is therefore to run `mulmod` the fewest times possible. My algorithm is simple: take an iterator of $2^{62}$ (approximately halfway up the number line) and try the `mulmod` method. If it returns a negative, backtrack and divide the iterator by two, then try again. Therefore the `mulmod` method will only be called around a hundred times.

The code follows. It accepts a `func` argument, and it works with methods which are only explained in future questions.

```
public static long find_max_n (String func)
{
        // func accepts paramaters a, b, n
```

```
        long a, b, n, jump;
        boolean is_valid;

        jump = (long) 1 << 62;

        for (n = jump; ; n += jump)
        {
                a = b = n - 1;

                // {{{ is_valid = (n > 0 && func (a, b, n) >= 0);
                is_valid = n > 0; // check if n itself exceeds Java's capacity

                if (is_valid)
                {
                        if (func.equals ("mulmod"))
                        {
                                is_valid = mulmod (a, b, n) >= 0;
                        }
                        else if (func.equals ("mulmod_java"))
                        {
                                is_valid = mulmod_java (a, b, n) >= 0;
                        }
                        else if (func.equals ("mulmod_with_addmod"))
                        {
                                is_valid = mulmod_with_addmod (a, b, n) >= 0;
                        }
                        else if (func.equals ("addmod"))
                        {
                                is_valid = addmod (a, b, n) >= 0;
                        }
                        else if (func.equals ("addmod_java"))
                        {
                                is_valid = addmod_java (a, b, n) >= 0;
                        }
                        else if (func.equals ("expmod"))
                        {
                                is_valid = expmod (a, b, n) >= 0;
                        }
                }

                // }}} is_valid

                if (is_valid == false)
                {
                        n -= jump; // backtrack to last known good value

                        jump >>>= 1;

                        if (jump == 0) break; // The previous n was indeed the max
                        // So loop -- add (old jump)/2 to n and try that.
                }
        }

        return n;
}
```

And so, the answers:

- `mulmod_java` accepts a maximum `n` of `3037000500` (`a` and `b` are less than the square root of $2^{63}$).
- `mulmod` accepts a maximum `n` of `4611686018427387904` ($2^{62}$).

# Question 3

The question asks for an improvement to the `addmod_java` shown in the section called "Question 2".

The problem with the existing algorithm is that it `a + b` can exceed the maximum value of a `long`. Since `(a + b) % n` can only be equal to either `a + b - n` or `a + b`, it is easy to write a special-case algorithm to optimize `mulmod`.

```
public static long addmod (long a, long b, long n)
{
        // assume a < n,  b < n, and all three are positive

        long result = a - n + b;

        if (result >= 0)
        {
                return result;
        }
        else
        {
                return result + n;
        }
}
```

Then, edit `mulmod` to use `addmod` in the place of `addmod_java`.

```
public static long mulmod_with_addmod (long a, long b, long n)
{
        long a_doubled;

        if (b == 0) return 0; // a * 0 = 0 %n

        a_doubled = addmod (a, a, n);

        if ((b & 1) == 0)
        {
                // a * b = (2 * a) * (b / 2) % n
                return mulmod_with_addmod (a_doubled,
                                           b >>> 1,
                                           n);
        }
        else
        {
                // a * b = a + (2 * a) * ((b - 1) / 2) % n
                return addmod (a,
                               mulmod_with_addmod (a_doubled,
                                                   b >>> 1,
                                                   n),
                               n);
        }
}
```

This simple algorithm also speeds up the `mulmod` somewhat, since it replaces a costly mod operation with a comparison and two or three additions.

The algorithm now extends to the maximum positive value of a `long` integer: that is, `n = 9223372036854775807`.

# Question 4

The `expmod` method is even *easier* to write than the `mulmod` method, since it's basically all written already. It just needs a few replacements.

```
public static long expmod (long a, long b, long n)
{
        long a_squared;

        if (b == 0) return 1; // a ^ n = 1 % n

        a_squared = mulmod_with_addmod (a, a, n);

        if ((b & 1) == 0)
        {
                // a ^ b = (a ^ 2) ^ (b / 2)
                return expmod (a_squared, b >>> 1, n);
        }
        else
        {
                // a ^ b = a * (a ^ 2) ^ ((b - 1) / 2)
                return mulmod_with_addmod (a,
                                           expmod (a_squared,
                                                   b >>> 1,
                                                   n),
                                           n);
        }
}
```

Given `a` = `1274434408442`, `b` = `589394265617`, and `n` = `1606818609763`, the resultant (`c`) is `308250308250` (notice the pattern in the digits: it's our course number repeated twice). Calling the method again with given parameters `308250308250` (`c`), `433371342353` (`d`), and `1606818609763` (`n`) gives the answer `1274434408442` (this number is actually `a` from above).

# Code

## basis.java

All program listings from the preceding sections were placed in the file ../basis.java. Each of the following classes extends the `basis` class, adding a front-end.

## q1.java

```
public class q1 extends basis
{
        public static void main(String[] argv)
        {
                long a, b;

                if (argv.length != 2)
                {
                        System.err.println ("usage: q1 [multiplier]"+
                                            " [multiplicand]");
                        return;
                }

                a = Long.parseLong (argv[0]);
                b = Long.parseLong (argv[1]);

                System.out.println (a + " * " + b + " = " +
                                    mulalarusse (a, b));
```

```
        }
}
```

The program in ../q1.java is a simple program to multiply two numbers. It requires two arguments, representing the two (`long`) numbers to be multiplied. For example:

```
$ java q1 35 -143
35 * -143 = -5005
```

# q2.java

```
public class q2 extends basis
{
        public static void main(String[] argv)
        {
                long n;

                if (argv.length != 1)
                {
                        System.err.println ("usage: q2 [-m] [-j] [-a]");
                        return;
                }

                if (argv[0].equals ("-m"))
                {
                        n = find_max_n ("mulmod");
                        System.out.println ("Largest value of n with"
                                        + " mulmod: " + n);
                        return;
                }

                if (argv[0].equals ("-j"))
                {
                        n = find_max_n ("mulmod_java");
                        System.out.println ("Largest value of n with"
                                        + " mulmod_java: " + n);
                        return;
                }

                if (argv[0].equals ("-a"))
                {
                        n = find_max_n ("mulmod_with_addmod");
                        System.out.println ("Largest value of n with"
                                        + " mulmod_with_addmod: " + n);
                        return;
                }

                System.err.println ("usage: q2 [-m] [-j] [-a]");
        }
}
```

The program in ../q2.java finds the maximum valid `n`, as described in the section called “Question 2”. If called with the `-m` argument, it will test the `mulmod` method. If called with the `-j` argument, it will test the `mulmod_java` method. If called with the `-a` method, it will test the `mulmod_with_addmod` method from the section called “Question 3”. For example:

```
$ java q2 -j
Largest value of n with mulmod_java: 3037000500
$ java q2 -m
```

```
Largest value of n with mulmod: 4611686018427387904
$ java q2 -a
Largest value of n with mulmod_with_addmod: 9223372036854775807
```

Note that this program covers both the section called “Question 2” and the section called “Question 3”.

# q4.java

```
public class q4 extends basis
{
        public static void main (String[] argv)
        {
                long a, b, c, d, e, n;

                a = 1274434408442L;
                b = 589394265617L;
                n = 1606818609763L;

                System.out.println ("a: " + a);
                System.out.println ("b: " + b);
                System.out.println ("n: " + n);

                System.out.println ("c = (a ^ b) % n");
                c = expmod (a, b, n);
                System.out.println ("c: " + c);

                d = 433371342353L;

                System.out.println ("d: " + d);

                System.out.println ("e = (c ^ d) % n");
                e = expmod (c, d, n);
                System.out.println ("e: " + e);
        }
}
```

The program in ../q4.java goes through the question and prints its progress.

```
$ java q4
a: 1274434408442
b: 589394265617
n: 1606818609763
c = (a ^ b) % n
c: 308250308250
d: 433371342353
e = (c ^ d) % n
e: 1274434408442
```