3. Implementing the Russian Peasant's algorithm in Java (using ints/ longs) and verify its correctness.

Take the pseudocode below and translate into a basic java algorithm that implements the Russian Peasant's algorithm:

1. Try your best to write this code yourself (no cutting and pasting or searching the web for solutions) - this is the best way to learn. If you get stuck as a demonstrator for assistance.
2. Your algorithm can take input either through hard-coded integers or through prompt input.
3. Test your algorithm on several input integers and verify the correctness of the output using a calculator (or by hand).

Unfortunately, due to the simplistic nature of the Russian Peasant's algorithm, it is almost impossible to measure the running time on a modern system. Though the algorithm does run in O(log n) time, a typically measurable result, there are some specific details relating to the algorithm which influence the actual maximum and minimum runtimes possible using this algorithm. The 'n', in the above O(log n) characterisation of the algorithm, represents the position of the most significant set bit (1) in the input integers. However, most modern systems support at most 64-bit integers, meaning that there is a hard ceiling on how large the input can be. In absolute terms, the loop which is the most computationally expensive part of the algorithm (see code highlighted in red below), is only iterated over between 0 and 8 times for a 64-bit integer. Given these tiny differences, it is almost impossible to measure the difference in running time, as the loop itself contains only three very basic arithmetic instructions. Therefore, the algorithm effectively runs in constant time given the limited constraints of the test environment.

```
public static long russianPeasantAlgorithm(long a, long b) throws Exception {
        long result = 0;                // this red section is iterated over between 0 and 8 times.
        while(b > 0){                   // on modern systems, this is effectively constant-time,
                if((b % 2) == 1){       // because the loop contains very simple instructions
                        result += a;
                }
                b /= 2;
                a *= 2;
        }
        return result;
}
```

To test the correctness of the algorithm, I used the BigInteger library. It supports accurate multiplication of arbitrarily large integers.