Implement Elgamal signatures as described in the course readings. Since you will need to manipulate large integers you will need to use a programming language that supports them. If you wish to use C you may self-study the BIGNUM library in OpenSSL and use that, or you can use the built in support for big integers in Python or Java.

Signature Generation

Write a command-line program that takes arguments p, g, g^d, d, and x and outputs r and s (all expressed as decimal integers). Here's an example that follows the example in the reading.

```
sign 101 2 14 10 5
59
35
```

Part of signature generation is random. To pick your random 0 < e < p you should first figure out how many bits are in p and then repeatedly generate that many random bits and until the result satisfies all criteria. You will have to investigate how to do these things in your chosen language. (Note: In cryptographically secure code, you would need to use high-quality cryptographically secure random bits. In this program you may use your programming language's non-secure random bit generator if you wish.)

Signature Verification

Write a command-line program that takes arguments p, g, g^d, x, r, and s and outputs 1 if (r,s) is a valid signature for x and 0 otherwise (all expressed as decimal integers). Here's an example that follows the example in the reading.

```
verify 101 2 14 5 59 35
1
```

Testing and Submission

Test your programs thoroughly. You can manually create your own keys following the procedure in the reading and then check that your verifier outputs 1 for good signatures and 0 for ones that are bad in various ways. You should do this for small p (so that you can verify computations by hand) and then scale up to a larger p to make sure everything works when the integers are large.

When manually creating your own test keys, you know that p is prime if Wolfram Alpha responds positively to something like `13 prime?` You know g generates Z^*_p when Wolfram Alpha responds with p-1 to the query like `order of 3 mod 13`. Wolfram Alpha can also compute gcd for you `gcd(2,12)`.

Submit via our Fileinbox procedure two files: `sign.py` and `verify.py`, or `sign.java` and `verify.java`, or `sign.c` and `verify.c`.

Your code should behave exactly as shown above. It should compile and run without warnings or errors, and it should not print anything except what is specified. You may assume that the correct number of inputs are given and that p, g, g^d, d form a correct Elgamal key. You may also assume that x, r, and s are elements of Z^*_p.

You are welcome to share your signature output with one another so that you can test interoperability.