Written problems

1. Textbook exercise 4.10 (DSA verification examples)
2. Textbook exercise 3.22 (Pollard's algorithm examples)
3. Consider the following implementation of one trial of Pollard's algorithm.

```
def pollardTrial(N):
        a = random.randint(1,N-1)
        # Check first whether a is a unit. If not, you have a factor.
        if math.gcd(a,N) != 1:
                return math.gcd(a,N)
        j = 2
        while math.gcd(a-1,N) == 1:
                a = pow(a,j,N)
                j += 1
        return math.gcd(a-1,N)
```

   (a) Suppose that this function is called on an input $N = pq$, a product of two distinct primes. Prove that in principle (i.e. given an unbounded amount of time), this function will always return some factor of N other than 1. Under what circumstances will it return N, rather than a proper factor?

   (b) Suppose that this function is called on $N = pq$, where both $p-1$ and $q-1$ have at least one prime factor greater than 2^{256}. Estimate how large you expect j to grow before this function will return an answer, and explain why. The result will depend on the random value of a this is chosen; try to justify why the estimate you give will be correct with very high probability. This can be a fairly informal argument; a rigorous proof would be a bit technical.

4. This problem explores the reasons why the primes p and q is DSA can be chosen to have somewhat different sizes.

   Suppose that p, q, g are DSA public parameters (i.e. p, q are primes, and g has order q modulo p), and $A \equiv g^a \pmod{p}$ is Samantha's public (verification) key, while a is her private (signing) key. As we discussed in class, there are two main sorts of algorithms that Eve might use to extract a from A: collision algorithms (whose runtime depends on q), and the number field sieve (whose runtime depends on p). For simplicity, assume that Eve has a collision algorithm that can extract a in $\sqrt{q}$ steps, and an implementation of the number field sieve (a state of the art DLP algorithm; you do not need to know any details about it, but the textbook has a good overview) that can extract a in $e^{2(\ln p)^{1/3}(\ln\ln p)^{2/3}}$ steps (the true runtimes would involve a constant factor that would depend on implementation, and various other factors depending on the cost of arithmetic modulo p and of finding collisions).

   (a) Suppose that Samantha is confident that her private key will be safe as long as Eve does not have time to perform more than 2^{64} steps in either algorithm. How many bits long should she choose p to be? How many bits long should q be?

   (b) What if she instead wants to be safe as long as Eve doesn't have time for 2^{128} steps?

(c) The NSA's recommendation for "Top Secret" government communications is to use 3072 bit values of p, and 384 bit values of q. How does this compare to your answers above? If the difference is significant, what might explain the discrepancy?

For parts (a) and (b), it is sufficient to write a short script to find the minimum safe numbers of bits by trial and error (there are more efficient ways, of course).

Programming problems

1. Write a function `verifyDSA(p,q,g,A,d,s1,s2)` that verifies DSA signatures. Here, p, q, g are public parameters, A is the public (verification) key, d is the document, and (s_1, s_2) is the signature. The function should return `True` or `False`.

2. Suppose that Samantha and Victor are using a variant of Elgamal signatures, in which the verification congruence that Victor will use is $s_1^{s_1} \cdot g^{s_2} \equiv A^d \pmod{p}$. Write a function `signElGamalVariation(p,g,a,d)`, which produces a valid signature in this system, given the public parameters p, g, Samantha's secret signing key a, and a document d.

3. Devise a method to create "blind forgeries" for a given DSA public key. That is, write a function `dsaBlind(p,q,g,A)` given p, g and A as in DSA, generate integers (d, s_1, s_2) such that (s_1, s_2) is a valid signature for d for the verification key A. You will likely want to adapt the strategy from one of last weeek's problems from Elgamal to DSA. Your method should be non-deterministic; the grading script will give the same test case multiple times to check that the same answer is not returned each time.

4. Implement Pollard's factoring algorithm, and use it to attempt to break 96-bit RSA encryption. More specifically: you should write a function `pollardRSA(N,e,c)` that takes an RSA public key (N, e) and ciphertext c, and attempts to extract the plaintext. For the grading test-cases, I have generated 96-bit RSA moduli *without making any effort to choose strong primes*; as a result, some of these moduli are more susceptible to Pollard's algorithm than others. However, many cases will not be unually susceptible to Pollard, so *for full points on this problem your code only needs to solve* 10 *of the* 50 *test cases.* But you may enjoy trying variations on Pollard's algorithm (or another factoring algorithm altogether) to see how many test-cases you are able to solve. I will enable an "leaderboard" on Gradescope to show the five highest scores obtained (your real name will not be used if you don't wish to use it; Gradescope will allow you to choose a pseudonym for the leaderboard).

   The sample cases (from the testing notebook) will include a few smaller cases to help debug your program, then ten 96-bit cases that I have specifically chosen to be susceptible to Pollard. So the sample cases will give you a sense of whether Pollard's algorithm is implemented well, but don't necessarily indicate that it will pass all of the cases on Gradescope.