Code Jam 2019 - Qualification Round

Analysis: Cryptopangrams

Yeah, but what about *de*crypting?

The statement tells us how the plaintext is encrypted, but it says nothing about the decryption mechanism! Figuring that out is part of the problem. Since this is a Qualification Round problem, there is slightly less time pressure and competitive urgency, and you have some extra time to think about how this cryptosystem is supposed to be used.

Suppose that Cameron and Jamie are members of the Code Jam team who know the secret list of 26 primes, and suppose that Cameron has just sent the ciphertext to Jamie. Each value in the ciphertext is the product of two of those primes, so Jamie can try dividing each value by each known prime to find the value's factors. (Notice that a value in the ciphertext might be the square of a prime, if the plaintext has the same letter twice in a row.)

After getting the list of factored ciphertext values, how should Jamie recover the plaintext? We might be tempted to say that the second letter of the plaintext is the one whose prime appears as factors of both the first and second ciphertext values, the third letter is the one whose prime appears as factors of both the second and third ciphertext values, and so on. Then the remaining factors in the first and last ciphertext values give us the first and last letters of the plaintext.

This is almost correct, but we (and Jamie) would have to deal with one significant annoyance. If the plaintext starts with something like `ABABA`..., for example, then the first, second, third, and fourth ciphertext values will all be the same, being the product of the same two factors (the primes corresponding to `A` and `B`). In particular, the start of `BABAB`... looks just the same as the start of `ABABA`...! The good news is that we know that this kind of pattern must terminate somewhere in the message; eventually, either we will get the same letter twice in a row, or (since the plaintext uses more than two different letters) three consecutive different letters. As soon as either of these things happens, we have a "break-in point", and we know which factors of a particular ciphertext value go with which of the two letters of the plaintext that it encodes. Then, we can "unzip" the rest of the plaintext from there, working backwards and forwards.

For instance, if the plaintext starts with `ABABAABAB`, the first four ciphertext values will all be the same: the product of the prime corresponding to `A` and the prime corresponding to `B`. The fifth ciphertext value will represent the square of `A`, so we will know that the fifth and sixth plaintext letters are both `A`. We can then reason that the fourth plaintext letter must be the fourth ciphertext value divided by the prime corresponding to `A`, the third plaintext letter must be the third ciphertext value divided by the prime corresponding to `B`, and so on going backwards. We can also determine that the seventh plaintext letter is the sixth ciphertext value divided by the prime corresponding to `A`, and so on going forwards.

Similarly, if the plaintext starts with `ABABCBABA`, when we inspect the third and fourth ciphertext values, we will see that they are different, but both have the prime corresponding to `B` as a factor. Then we can unzip from there, as above.

However, we must remember that Jamie has an advantage that we do not have: we do not know the 26 secret primes! We need to find a way to get them.

Test set 1

In test set 1, the ciphertext values are products of small primes. Each prime is less than 10^4, so each ciphertext value is no larger than 10^8. It is straightforward to factor these values by testing every possible (prime) factor between 2 and 10^4. Once we have all of the factors, we will have our set of 26 primes, since each prime will be represented in at least one factor of one ciphertext value. We can assign them in increasing order to the letters `A` through `Z`.

Then, to recover the plaintext, we can use a bit of brute force to sidestep the need to unzip as described before. Consider the two factors that contribute to the first ciphertext value; arbitrarily choose one of them. Let us first assume that that factor corresponds to the first letter of the plaintext, and the other corresponds to the second. Then we can take the remaining factor and try to divide the second ciphertext value by that factor. If we cannot, we have a contradiction, and we should go back and make the other factor correspond to the first letter of the plaintext. Otherwise, the quotient is the factor corresponding to the third letter of the plaintext, and so on. For one of our choices, this method will work and will give us the correct plaintext; for the other choice, we will reach a contradiction, since (as described above) it is guaranteed that there is only one possible decryption.

Test set 2

In test set 2, the primes can be enormous (as large as a googol), and the product of two such primes can be even more enormous. We should realize that it is hopeless to try to factor such a product. If we could do that, we could also break modern cryptosystems that depend on the assumption that factoring large numbers is intractable! The Code Jam servers do not run on quantum computers (yet...), so there is no way for us to try to use a quantum algorithm, either.

To solve this seemingly impossible test set, we need to find a different vulnerability of this cryptosystem. The key insight is that any two consecutive values in the ciphertext have at least one factor in common. Factoring very large numbers may be intractable, but we *do* know how to efficiently find the greatest common divisor of two very large numbers! A method like Euclid's algorithm will be fast enough.

Notice that if we have a plaintext like `ABABC`..., it is possible that the prime corresponding to `A` will not appear in any of the pairwise GCD values. So, we should compute GCDs of consecutive ciphertext values until we get a value that is not 1; at least one such value must exist, as described in the introduction to the problem. At that point, we can unzip the rest of the ciphertext, as described previously, finding all of the primes as we go. Then we can proceed as above. And we do not even have to know a bevy of DP flux algorithms, whatever those are!

A note on language choice

An essential skill in Code Jam is picking the right tool (language) for the right job. For some problems, a fast language such as C++ may be needed to obtain a solution within the time limits. For this problem, it is probably better to choose a language like Python that can handle very large numbers without any extra hassle, even at the cost of some speed, or a language like Java that has a built-in "big integer" implementation.