Encryption Protocol

For both my Client and Server programs, I have implemented RSA 3072 bit encryption to ensure the secure sending of data between both programs.
Both of these programs generate their own set of public and private keys that are different from each other upon starting up each program. These are both randomised each time the program is started.
When the two programs connect, they securely exchange their public keys with each other to allow for decryption of corresponding messages.
Both programs encrypt the data using their own private key before sending it to each other.

Password Protocol

My client program implements a password that is required for users to be able to access the program. This password can be generated using the createPass file. The password that is created is both hashed and salted before it is stored in the ‘pass.txt’ file using SHA256 and a randomly generated 32 bit salt. For password validation, the entered password is checked against the stored one using the same salt and hash which is found by reading the first and last 32 characters. If both passwords match, the user is granted access. The program also implements a feature which will close the client program if 5 incorrect passwords are entered in a row, to help mitigate against brute force attacks.

Optimization Protocol

For my optimisation rules, I chose to follow the 3rd-cg rule included in the berthub article. From my understanding, codons with either a G or a C character on the end are considered the most ‘optimised’. My code first checks if the last letter is either a G or a C. If so, no change is made. If not, the program first tries to change the last letter to a G, and if it matches the change is kept. The program will then try to change the last letter to a C and again if it matches it is kept, otherwise the change is discarded.
I chose this implementation as it looked like it had a high Nucleotide match rate, only being 5% below the highest, and its codon match rate, while a bit lower than the

highest, was still quite good. It was relatively easy to implement compared to some of the other solutions.

How to use: Server

You can run the server either by entering ./startServer.sh or running the python file by entering ./server.py. It will prompt you to enter a port between 1024 and 65535. Enter a number between these ranges and the server should launch.

How to use: Client

Next, you can launch the client by entering ./startClient.sh or by running the python file by entering ./client.py. You must enter the correctly stored password to gain access to the program. By default this is ‘0n3c00lC4T’. Next it will prompt you to enter the server IP. In a local use case you must enter ‘127.0.0.1’. After this you will be prompted to enter the port that the server is being hosted on. If these are entered correctly the client will connect to the server. After connection the client must type ‘START RNA’ if they want to begin optimizing. After the user enters this they are free to input their codon sequences into the command line. If a valid sequence has been entered, the server will return the optimized sequence if there is one available. Once it has been returned the user can either input more sequences or type ‘Disconnect’ to close the client program.

Changelog

- Increased port range to 1024-65535
- Capitalisation no longer matters for client input
- User must now enter ‘start rna’ each time they wish to send a new codon sequence
- Added password verification
- Added RSA encryption for both programs

Notes

I have chosen to include the client and server printing their encrypted sequences purely for a visual demonstration that it is actually being encrypted, but this can be removed if needed by commenting out line 60 in client.py and line 144 in server.py