The algorithm goes a little something like this:

1. First a function “generate_prime_number” generates a random number of the given amount of bits (set later in the code under the “RSA setup”) and then it tests if the number is prime with using the buil-in sympy.isprime and repeats the process until it finds a suitable number that is prime
2. Next, the function “power_mod(base, exponent, modulus)” efficiently computes the base, modulus and exponent needed for RSA
3. Next, “euler_totient” computes the φ of n (due to time complexity being too slow with bits that are more than 32, I’m going with setting the amount of bits to 32)
4. Next the function “choose_e” selects the RSA public exponent e with e being initialised at the standard value of 65537 which is typically used in RSA encryption since the $\gcd(e, \varphi(n)) = 1$ is neded to get the private key d
5. Next the function, “string_to_int(s)” converts a text string into an integer by converting the strings into bytes using UTF-8
6. The following function “int_to_string” converts an integer string into a text string
7. Next up is the “RSA setup” code block where we have p and q which are random large prime numbers used in the computation of $n = p \cdot q$ which gives the modulus to be used in RSA; $\varphi(n)$ is the euler’s totient while e and d are the public and private exponents respectively
8. Next is the part where the user inputs a small text which then gets encrypted (“cipher”) and then decrypted (decrypted_int) and the integer string gets converted back into the text string (decrypted)

Time complexities of the functions:

- generate_large_prime → $O(\log n \cdot \log^3 n)$
- power_mod(base, exponent, modulus) → $O(\log n \cdot \log^2 n)$
- euler_totient → $O(\sqrt{n})$
- choose_e → $O(\log n)$
- string_to_int and int_to_string → $O(k)$
- encryption and decryption → $O(\log n \cdot \log^2 n)$

The overall time complexity of the RSA code is $O(\sqrt{n})$

I used Microsoft Copilot to help me correct my code and make it more cohesive due to my previous set up running into errors and not working properly. I have attached the Code Documentation (31st January 2026) and today’s (7th February 2026) latest model to compare the codes and how they changed. I also used it to figure out the time complexity of each function and the total code.

This algorithm is good for a short text since the number of bits is quite small. To encrypt larger and longer messages, it would be better to stop using the Euler's totient since larger bit size would crash and freeze the algorithm since the totient factoring time is nearly impossible. Another good way of doing this would be by implementing a hybrid of RSA and AES encryption, one asymmetric and the other symmetric which are used in real world encryption processes.

Sources I used to learn about the topic:

- https://www.geeksforgeeks.org/python/secrets-python-module-generate-secure-random-numbers/
- https://github.com/freezing/learn-cryptography/blob/main/rsa.py
- https://github.com/breezy-codes/cryptography-guide/blob/main/RSA-Algorithm/RSA-example.ipynb
- https://stackoverflow.com/questions/69801359/python-how-to-convert-a-string-to-an-integer-for-rsa-encryption