RSA Simulation and Visualization Tool

Introduction

With the growth of the digital world, protecting online information is more crucial than ever. RSA encryption, built on prime numbers and modular arithmetic, is essential for data security, though its implementation can appear complex.

The main goal of this project was to build a working simulation of RSA in Python that doesn't just encrypt and decrypt messages, but also helps visualize what's going on behind the scenes. The project aims to make RSA less of a black box and more of an accessible concept, especially for people learning cryptography or number theory for the first time.

Core Concepts Used

Two major concepts were implemented in this project: directed graphs and dynamic programming. These concepts helped visualize relationships within the RSA system and made key computations more efficient and manageable.

The **directed graph** concept was especially useful for understanding how different parts of RSA relate to each other—particularly the connection between the public exponent e, the modulus n, and the private exponent d. In this graph, each node can represent a value or operation, and the edges represent how values lead to others through functions like modular exponentiation or the extended Euclidean algorithm. Although the actual encryption process doesn't require building a graph structure, using one helped simulate and visually explore the interactions between components, especially for educational and debugging purposes.

Dynamic programming was used in modular exponentiation calculations and caching intermediate results to optimize repeated computations. For example, exponentiation by squaring was implemented with memoization to avoid redundant calculations, improving efficiency. This approach helped keep the RSA operations fast even for larger numbers.

Operations and Time Efficiencies

The following operations are used repeatedly throughout the code:

- **GCD (Greatest Common Divisor):**
  Used to check coprimality and compute the totient φ(n). This uses the classic Euclidean Algorithm and runs in *`O(log n)`* time.

- **Modular Inverse:**
  The Extended Euclidean Algorithm is used here, which also runs in `O(log n)` time. This is key for finding valid public and private key pairs.

- **Modular Exponentiation:**
  This is where RSA actually encrypts or decrypts data. Exponentiation by squaring, running in `O(log exp)` time, is used to keep this fast and caching makes it even more efficient if the same powers come up multiple times.

- **Graph Operations:**
  Adding nodes and edges is basically `O(1)` by leveraging dictionaries. Traversing the graph takes `O(V + E)` time, but that's only used for visualization, not core encryption or decryption.

- **Encryption and Decryption:**
  Since each character in a message is processed independently using modular exponentiation, the time complexity is linear in message length times the exponentiation time: `O(m·log exp)`.

Challenges Faced

One of the trickier parts of this project was making sure everything worked mathematically, especially when it came to modular inverses. If *e* and φ(n) aren't coprime, you can't generate a valid private key, so it was important to filter out bad choices for *e*.

Another challenge was the graph itself. As *n* increases, the number of valid (e, d) pairs grows rapidly, which slows down the graph-building process significantly. It also took considerable time and effort to ensure the graphs were drawn clearly and accurately. To address these issues, the visualization was limited to smaller, valid pairs, striking a balance between being educational and maintaining responsiveness.

Finally, the caching system for modular exponentiation worked great but had to be implemented carefully to avoid wasting memory.

Impact Statement

If taken further, this project could be turned into a really useful learning tool for students, educators, or anyone interested in how RSA works under the hood. There are already online RSA calculators out there, but few of them let users *see* the math in action—especially the relationship between the public and private keys as a directed graph.

This tool makes it easier for people to get started with understanding asymmetric cryptography. It also encourages experimentation, which is important for deep learning. The UI restricts users to selecting prime numbers, so primality is implicitly ensured. However, it’s worth noting that this is not secure for actual use. While it theoretically supports very large primes (as large as Python can handle), the tool doesn’t implement cryptographic-level key sizes or security measures. It’s not built to withstand real-world attacks, but that’s acceptable for an educational tool focused on learning, not defense.

Conclusion

Overall, this RSA simulation project met its goals: it works, it’s clear, and it helps users explore the foundations of a widely-used cryptographic algorithm. By combining group theory, modular arithmetic, and graph-based visualization, it builds a bridge between abstract math and practical security concepts.

There’s still room for improvement, such as adding support for automatic prime generation, customizable key sizes, and optimizing graph construction for better performance with larger inputs, but it works, as a proof of concept. I also plan to expand this program to support different types of algebraic groups.