Algorithms for Generating Prime Numbers and their Application in Cryptography

Johnny Ruan and Jacob Helhoski

Problem Statement

The project will explore various algorithms for generating prime numbers, with a focus on the Sieve of Eratosthenes, Sieve of Sundaram, and Sieve of Atkin. We aim to compare these algorithms in terms of efficiency, complexity, and practical performance implications. We will then demonstrate their use by implementing and discussing the RSA encryption algorithm. The objective is to understand how the algorithms operate and their relevance in today's computational applications.

Prime Generation with Sieve Algorithms

Sieve of Eratosthenes

One of the oldest methods for finding primes up to a certain number. Takes numbers and marks their multiples: starting at 2 and proceeding to the next unmarked number, repeat. An important optimization is that for each prime number 'p' found, you start marking multiples from p^2 instead of 2p. This is because smaller multiples of 'p' (like 2p, 3p, … up to (p-1)p) would already have been marked as composite by smaller prime numbers.

Sieve of Sundaram

This sieve uses the same operation of marking numbers and their multiples, but goes about it using a different formula: $i + j + 2ij < n$. This algorithm uses a boolean array “marked” with the size of nNew + 1, which is equal to $(n - 1)/2$. The core of this algorithm is in the loops where $i \leq j$, marking the index $i + j + 2ij$ as true in the marked array, eliminating the numbers that are not prime. With each marked number corresponding to an integer that isn’t a prime number when transformed back to the original range. The formula $i + j + 2ij$ generates all the numbers in $2i + 1$, since if $2i + 1$ is prime, then i cannot be the sum of two integers and double of their product.

Sieve of Atkin

The sieve of Atkin is a modern optimization algorithm for the sieve of Eratosthenes. The algorithm adds 2 and 3 to the list of primes if the limit is greater than 2 and 3, respectively, since these are the first two primes. It then uses several computations for each pair of x and y, with each number satisfying the specific conditions being marked in the sieve array.

The conditions are:

- Numbers of the form 4x^2 + y^2 are marked if they are less than the limit and their remainder when divided by 12 is 1 or 5.
- Numbers of the form 3x^2 + y^2 are marked if they are less than the limit and their remainder when divided by 12 is 7.
- Numbers of the form 3x^2 - y^2 (where x > y) are marked if they are less than the limit and their remainder when divided by 12 is 11.

The sieve array isn’t a mark of primality, but potential primality, numbers that satisfy the above conditions are flipped (true>false, vice versa). A loop runs for each number r starting from 5 to the square root of the limit. For each number r, if it is marked as true in the sieve array, it indicates r is a potential prime. Then, all multiples of r^2 are set to

false in the sieve array, as they cannot be primes.The final step is to iterate through the sieve array and collect numbers marked as true, starting from 5. These numbers are the primes found by the algorithm.

RSA Encryption

Encryption is the transformation of data into a form where its original meaning is unrecognizable to anyone attempting to intercept it. This data is then decrypted by the recipient of the data, producing the original message. In this way, if data used in decryption is kept as private knowledge, then an outside party is unable to decrypt a message that they intercept; the data is kept secure.

RSA, put forth in 1978 by Rivest, Shamir, and Adleman, uses private-public key pairs for encryption. A high level view of this system is as follows: Johnny wishes to send a message to Jacob which only Jacob can read. Jacob makes available, in a "public file"[1], his public key pair (e, n). Johnny then runs the encryption function on the message using the public key, producing ciphertext, c. Jacob, who is also in possession of a private key pair (d, n) can use this key to run the decryption function on c, which produces the original message. The details of encryption and key determination will be discussed later on, but first it is important to understand the principles on which RSA was designed.

The creators of the algorithm describe decryption and encryption as "one-way trap-door functions"[1]. This takes into account previous work that inspired the authors, which put forth the idea of one-way behavior being necessary for encryption[3]. The ciphertext, c, should be easily computable from a message string, allowing anyone to send a message to the recipient. However, computing the inverse of this function to produce the original message is near impossible.

This is where the notion of a "trap-door" is important; the recipient should have some data which makes computation of the inverse simple and time efficient. The recipient, and only the recipient, has the key to the "trap-door". The encryption/decryption function used in RSA has the properties described above, so a sender can be sure that only the recipient can read their message. But how can a recipient prove that a message originates from a certain sender?

Signatures

In the same way that one signs a letter to attach one's identity to it, a digital signature proves a sender's identity. The signing process makes use of the already existing encryption and decryption functions for opposite purposes. This is possible because, as the authors state, "each message is the ciphertext for some other message"[1]; any integer message of appropriate length is a valid input to both the encryption or decryption functions.

Before the signing process, the original message is used as the input to a hashing function. For each input, hashing functions produce a seemingly random string. The randomness, however, is artificial, since the same string will always be produced for the same input. The hashing function produces a string that is much smaller than the message. This is done because the original message can't be an input to the encryption and decryption functions without being separated into smaller pieces,

With the message hash created, the signing process is as follows (R-recipient, S-sender): S uses their own private key to encrypt the hash (now called a signature),

then it is included with the message and encrypted again with R's public key. When R receives the message, they decrypt it, as well as the signature, using their private key. The signature can now be decrypted with S's public key. R now has the original hash that S produced for the message. R uses the decrypted message, m, as the input to the same hashing function, then compares the resulting hash with the hash they received. If equal, the message they received was used to produce the hash, and, more importantly, S definitely sent the message.

Through this scheme, RSA ensures that only R can decrypt a message intended for R, and S can prove ownership of a message that was sent. Now, we present the details of encryption and key determination.

Details of the Algorithm

To determine the public and private keys:

1. choose two sufficiently large prime numbers, p and q
2. compute n = p * q
3. find e, a large value that is relatively prime to car = (p-1)(q-1) (Carmichael value)
4. find d by computing the modular multiplicative inverse of e mod car

To encrypt message m (assuming m has been properly split and encoded as an integer):

5. ciphertext $c = m^e \pmod n$

To decrypt ciphertext c:

6. result $r = c^d \pmod n$

An outside party, who we will refer to as an attacker, does, in fact, have all of the information needed to determine the private key, d, and decrypt messages intended for other people. Factoring n would give the attacker a simple way to determine d. However, the prime factorization problem is hugely inefficient for large inputs, and the current best methods to solve it complete their runtime over a period of days. Within that time, the prime factors originally used to generate the key pairs could be changed by the recipient, and this change reflected in the public key. There may be other methods to compute d, but, if they were computed efficiently, it would prove the existence of a polynomial factoring algorithm, which is highly unlikely to exist$_1$; therefore, alternative methods themselves are unlikely to exist. Because of these factors, RSA is secure against both known and unknown types of attacks.

Our Implementation

A simple RSA example, without signatures or hashing, is implemented in RSA.cpp. A difficulty we encountered in our implementation was the handling of large values that overflow C++ integer limits (unsigned long long int = 64 bits, RSA suggests at least 1024 bits). We overcame this through the use of the GMP (GNU Multiple Precision) library, which splits up large values in memory to let a program perform arithmetic operations on them. The program outputs the initial prime values, as well as the values computed throughout its execution.

Key generation from the initial primes runs in $O(n^2)$, due the multiplicative inverse and Carmichael value calculations. Encryption and decryption run in $O(n^3)$, due to the

modular exponentiation operation. To represent the value n in memory, more than twice the number of bits in the initial primes is needed; all other values are similar in size to the initial primes. This results in a space complexity of $O(n)_4$.

For our implementation of the sieve of Eratosthenes, we used a vector<bool>, initialized with n+1 elements, which takes O(n) time. The outer loop of the sieve process runs from 2 to n, using On iterations, with the inner loop marking all primes starting from its square, the number of operations for each prime, p, is approximately n/p. The total work is the sum of n/p for all prime numbers up to n, and using the two equations given:

$$\sqrt{n} \sum_{p<\sqrt{n}} \frac{\sqrt{n}}{p} = n \sum_{p<\sqrt{n}} \frac{1}{p}$$

$$\sum_{p<n} \frac{1}{p} = \log \log n + O(1)$$

(Karimi, 2023)

Therefore the time complexity for the sieve of Eratosthenes is O(nlog log n).

The space complexity for sieve of Eratosthenes is dominated by vector<bool>, which is used to keep track of whether each number up to n is prime or not. It holds n+1 elements, its space requirement is O(n).

For our implementation of the sieve of Sundaram, the outer loop runs from 1 to nNew, and the inner loop's condition is i+j+2ij nNew. The number of iterations of the inner loop depends on i, as it increases, the range of j that satisfies the condition decreases. The time complexity can be represented by the following equation:

$$\sqrt{n} \sum_{i<\sqrt{n}} \frac{\sqrt{n}}{i} = n \sum_{i<\sqrt{n}} \frac{1}{i}$$

(Karimi, 2023)

There is a difference between this equation and the one used in the sieve of Eratosthenes, while p could take only the prime numbers, i can take all the numbers between 1 and n, resulting in a larger sum. Therefore, it can be concluded using a harmonic series direct comparison test that:

$$\sum_{i=1}^{i=\sqrt{n}} \frac{1}{i} \geq 1 + \frac{\log n}{4}$$

(Karimi, 2023)

Therefore the time complexity of the sieve of Sundaram is O(nlogn).

The space complexity for the sieve of Sundaram is dominated by the marked array, using the data structure vector<bool>, which has a space requirement of n/2, so the space complexity is O(n).

The time complexity of the sieve of Atkin is O(n), despite the complexity of the loops seeming higher, as the conditions within the loops make it so not all combinations of x and y result in computations. The algorithm uses quadratic forms and modular checks to limit the operations significantly. The step of eliminating squares of primes is On, so the final loop of collecting the primes from the sieve is O(n), therefore the time complexity for the sieve of Atkin is O(n).
The space complexity of the sieve of Atkin is dominated by the vector<bool> data structure, and since this vector holds n+1 potential values, the space requirement for it is O(n). Therefore the space complexity of this algorithm is O(n).

Results

For our results, the sieve algorithms could only generate prime numbers up to 60,000 before running out of memory and a segmentation fault. Due to the relatively small amount of primes being generated, the sieve of Sundaram was the fastest, followed by the sieve of Eratosthenes, then the sieve of Atkin. This is likely due to the fact that even though the sieve of Atkin has the best theoretical time complexity, the complexity of the algorithm itself causes it to be the slowest for generating relatively small amounts of primes.

Limitations of Using Sieve Generated Primes

The Sieve algorithms require memory proportional to the range of numbers being sieved. For large ranges, such as those required for RSA, the memory needed becomes impractical. Generating primes of at least 1024 bits (the minimum required for use in RSA) would require an astronomical amount of memory. Segmentation faults due to programs trying to use more memory than available is a common issue when sieve algorithms are used to generate very large primes. RSA and similar cryptographic algorithms require very large primes, often 1024 bits or more. Sieve algorithms are not practical for generating primes of this magnitude because they need to process an enormous range of numbers. In RSA, the message (encoded as a number) must be smaller than the modulus (n), which is the product of two large primes. If the primes generated are not large enough, it limits the size of messages that can be securely encrypted. Breaking messages into smaller chunks is a workaround, but it adds complexity and potential security concerns. Due to the nature of sieve algorithms, which involve processing every number in a large range, makes them impractical for generating large primes needed in cryptography.

Possible Alternatives for Sieves

Segmented Sieve Implementation

Instead of sieving all numbers up to n at once, the segmented sieve works by dividing the range into smaller segments and sieving each segment one at a time. This reduces the memory requirement to only enough for the largest segment, rather than the entire range. First, generate all primes up to the square root of n using a standard sieve method. These primes are used to sieve each segment. The range from 2 to n is divided into segments of fixed size. For each segment, use the stored primes to mark multiples as non-prime within that segment. Once a segment is sieved, it can be processed or stored, and its memory can be reused for the next segment. While the

segmented sieve addresses the memory issue, it is still not practical for generating very large primes, such as those needed for RSA encryption, because the process remains time-consuming and computationally intensive for such large numbers. The segmented sieve is more useful for finding all primes within a large range, rather than finding individual large primes.

Miller-Rabin Primality Test

The Miller-Rabin Primality Test is a probabilistic test to determine whether a given number is prime. It's very efficient for testing large numbers and is widely used in cryptographic applications. The test repeatedly checks whether a number is likely to be prime using random 'witnesses'. If a number passes the Miller-Rabin test with a base (or witness), it is a probable prime. If it fails, it is definitely composite. The Miller-Rabin test is much faster than deterministic tests, especially for large numbers. By increasing the number of rounds, the probability of falsely identifying a composite number as prime can be made extremely low. In RSA key generation, large primes are needed. The Miller-Rabin test is used to quickly identify candidates for these primes. After a number passes enough rounds of Miller-Rabin tests, it's considered prime for cryptographic purposes.

References

1. Rivest, Shamir, Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.” *Communications of the ACM*, 1 Feb. 1978, dl.acm.org/doi/abs/10.1145/359340.359342.
2. Karimi, A. (2023, May 13). Fastest algorithm to find prime numbers. Baeldung on Computer Science. https://www.baeldung.com/cs/prime-number-algorithms
3. Diffie, W., et al. “New Directions in Cryptography.” *IEEE Transactions on Information Theory*, 1 Nov. 1976, dl.acm.org/doi/10.1109/TIT.1976.1055638.
4. Chaterjee, Sumanta. *Answer to post: “What is the complexity of RSA cryptographic algorithm?”. quora.com/What-is-the-complexity-of-RSA-cryptographic-algorithm*