

Euler's Totient Function

A Comprehensive Guide with Visual Explanations

Mathematical Foundations Series

February 5, 2026

Abstract

This document provides a comprehensive exploration of Euler's Totient Function, denoted as $\phi(n)$. We examine its definition, properties, calculation methods, and practical applications in modern cryptography, particularly in RSA encryption. Through detailed examples, visual diagrams, and step-by-step explanations, we demonstrate how this fundamental number-theoretic function plays a crucial role in contemporary information security.

Contents

1	Introduction	3
2	Mathematical Definition	3
3	The Core Concept: A Detailed Example	3
3.1	Step-by-Step Calculation for $n = 6$	4
3.2	Visual Representation of Coprimality	5
4	Calculation Methods	5
4.1	Direct Method (Brute Force)	5
4.2	Formula-Based Method	6
5	Additional Examples	7
5.1	Example: $\phi(12)$	7
5.2	Example: Prime Numbers	7
6	Properties of Euler's Totient Function	8
7	Why Do We Use It? Applications	8
7.1	RSA Encryption - The Most Famous Application	8
7.1.1	Concrete RSA Example	9
7.2	Why Totient Function Makes RSA Secure	9
8	Step-by-Step Algorithm Summary	10

9 Practice Problems	10
9.1 Problem 1	10
9.2 Problem 2	10
10 Conclusion	11
11 Algorithm Implementation	11
11.1 Pseudocode Algorithms	11
11.1.1 Method 1: Brute Force Approach	11
11.1.2 Method 2: Optimized Using Prime Factorization	12
11.2 Algorithm Flowchart	12
11.3 Java Implementation	13
11.3.1 Method 1: Brute Force Implementation	13
11.3.2 Method 2: Optimized Implementation	14
11.3.3 Complete Program with Both Methods	16
11.4 Understanding the Code	17
11.4.1 Key Points	17
11.4.2 Time Complexity Comparison	17
12 References	17

1 Introduction

Euler's Totient Function, also known as Euler's phi function, is one of the most important functions in number theory. Named after the Swiss mathematician Leonhard Euler, this function counts the number of integers from 1 to n that are relatively prime to n (i.e., integers that share no common factor with n other than 1).

Key Point

Two numbers are **relatively prime** (or coprime) if their greatest common divisor (GCD) is 1.

2 Mathematical Definition

Definition 2.1: Euler's Totient Function

For a positive integer n , Euler's totient function $\phi(n)$ is defined as:

$$\phi(n) = |\{k \in \mathbb{Z} : 1 \leq k \leq n \text{ and } \gcd(k, n) = 1\}|$$

where $|\cdot|$ denotes the cardinality (count) of the set.

In simpler terms, $\phi(n)$ counts how many numbers from 1 to n are coprime with n .

3 The Core Concept: A Detailed Example

Let's explore the calculation of $\phi(6)$ in detail to understand the fundamental concept.

3.1 Step-by-Step Calculation for $n = 6$

Example 3.1: Computing $\phi(6)$

We need to check each integer from 1 to 6 and determine if it is relatively prime to 6.

Checking GCD for $n = 6$

$$\begin{array}{|c|} \hline k = 1 \\ \hline 1(1, 6) = 1 \\ \text{Yes} \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline k = 2 \\ \hline 2(2, 6) = 2 \\ \text{No} \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline k = 3 \\ \hline 3(3, 6) = 3 \\ \text{No} \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline k = 4 \\ \hline 2(4, 6) = 2 \\ \text{No} \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline k = 5 \\ \hline 1(5, 6) = 1 \\ \text{Yes} \\ \hline \end{array}$$

$$\begin{array}{|c|} \hline k = 6 \\ \hline 6(6, 6) = 6 \\ \text{No} \\ \hline \end{array}$$

 Relatively Prime (Coprime)

 Not Relatively Prime

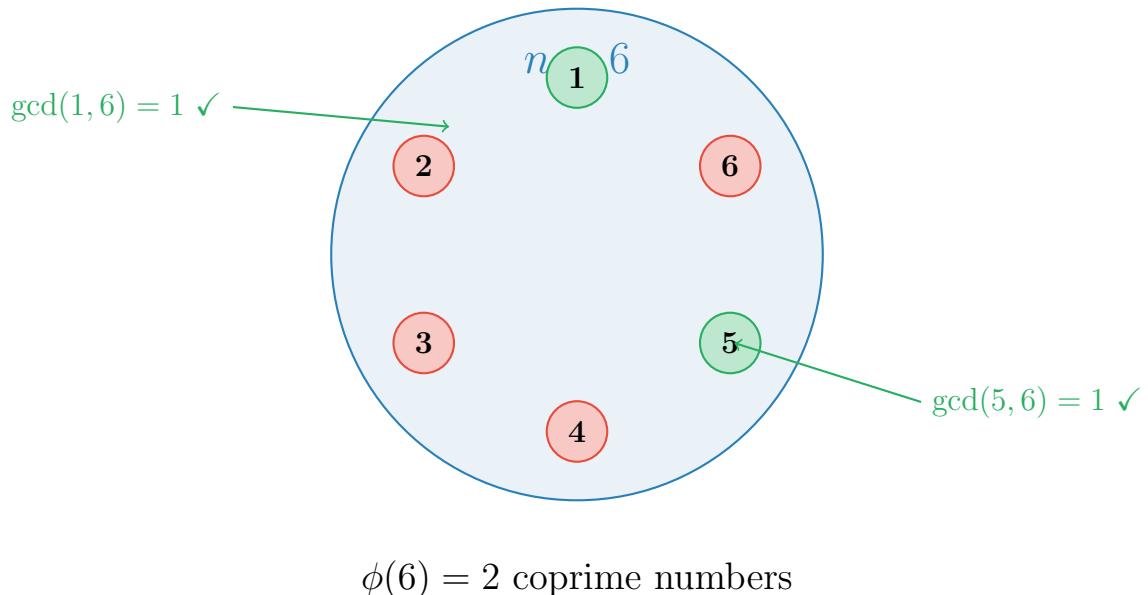
Detailed Analysis:

1. **Check** $k = 1$: $\gcd(1, 6) = 1 \Rightarrow$ **Coprime**
The number 1 is coprime to every integer.
2. **Check** $k = 2$: $\gcd(2, 6) = 2 \Rightarrow$ **Not Coprime**
Both 2 and 6 are even, sharing the common factor 2.
3. **Check** $k = 3$: $\gcd(3, 6) = 3 \Rightarrow$ **Not Coprime**
Both 3 and 6 are divisible by 3.
4. **Check** $k = 4$: $\gcd(4, 6) = 2 \Rightarrow$ **Not Coprime**
Both share the common factor 2.
5. **Check** $k = 5$: $\gcd(5, 6) = 1 \Rightarrow$ **Coprime**
5 is prime and does not divide 6.
6. **Check** $k = 6$: $\gcd(6, 6) = 6 \Rightarrow$ **Not Coprime**
A number always shares all its factors with itself.

Conclusion: There are exactly 2 numbers (1 and 5) that are relatively prime to 6.

Therefore, $\boxed{\phi(6) = 2}$

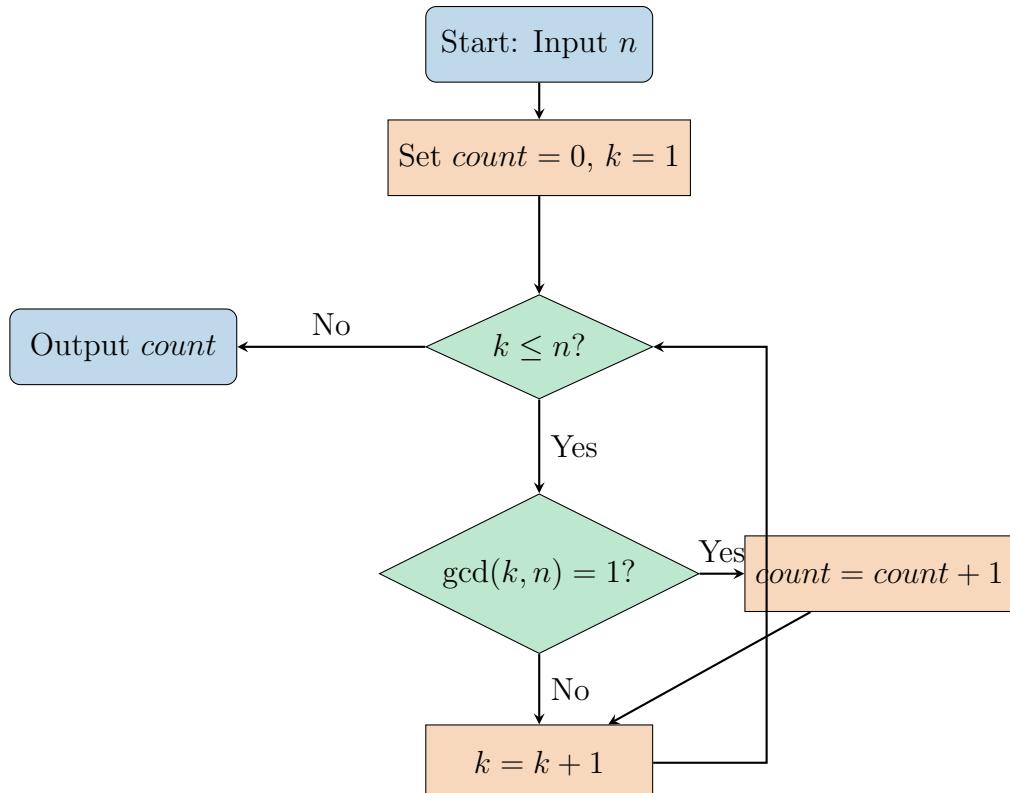
3.2 Visual Representation of Coprimality



4 Calculation Methods

4.1 Direct Method (Brute Force)

The most straightforward approach is to check each number from 1 to n :



Complexity: $O(n \log n)$ due to GCD calculations.

4.2 Formula-Based Method

For more efficient calculation, we use the multiplicative property of $\phi(n)$.

Theorem 4.1: Euler's Product Formula

If n has the prime factorization $n = p_1^{a_1} \cdot p_2^{a_2} \cdots p_k^{a_k}$, then:

$$\phi(n) = n \prod_{p|n, p \text{ prime}} \left(1 - \frac{1}{p}\right) = n \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_k}\right)$$

Example 4.1: Using the Formula for $n = 6$

First, find the prime factorization of 6:

$$6 = 2^1 \cdot 3^1$$

The prime factors are $p_1 = 2$ and $p_2 = 3$.

Apply the formula:

$$\begin{aligned} \phi(6) &= 6 \cdot \left(1 - \frac{1}{2}\right) \cdot \left(1 - \frac{1}{3}\right) \\ &= 6 \cdot \frac{1}{2} \cdot \frac{2}{3} \\ &= 6 \cdot \frac{2}{6} \\ &= \boxed{2} \end{aligned}$$

This matches our brute-force result!

5 Additional Examples

5.1 Example: $\phi(12)$

Example 5.1: Computing $\phi(12)$

Method 1: Direct Counting

Check each number from 1 to 12:

k	gcd(k,12)	Coprime?
1	1	Yes ✓
2	2	No
3	3	No
4	4	No
5	1	Yes ✓
6	6	No
7	1	Yes ✓
8	4	No
9	3	No
10	2	No
11	1	Yes ✓
12	12	No

Numbers coprime to 12: {1, 5, 7, 11}

Therefore, $\phi(12) = 4$

Method 2: Using the Formula

Prime factorization: $12 = 2^2 \cdot 3^1$

$$\begin{aligned}
 \phi(12) &= 12 \cdot \left(1 - \frac{1}{2}\right) \cdot \left(1 - \frac{1}{3}\right) \\
 &= 12 \cdot \frac{1}{2} \cdot \frac{2}{3} \\
 &= 12 \cdot \frac{1}{3} \\
 &= \boxed{4}
 \end{aligned}$$

5.2 Example: Prime Numbers

Theorem 5.1: Totient of a Prime

If p is prime, then:

$$\phi(p) = p - 1$$

Proof: Every number from 1 to $p - 1$ is coprime to a prime p , since p has no divisors other than 1 and itself.

Example 5.2: $\phi(11)$

Since 11 is prime:

$$\phi(11) = 11 - 1 = \boxed{10}$$

6 Properties of Euler's Totient Function

1. **Multiplicative Property:** If $\gcd(m, n) = 1$, then:

$$\phi(m \cdot n) = \phi(m) \cdot \phi(n)$$

2. **Power of Prime:** For a prime p and positive integer k :

$$\phi(p^k) = p^k - p^{k-1} = p^{k-1}(p - 1)$$

3. **Sum Property:** For any positive integer $n > 1$:

$$\sum_{d|n} \phi(d) = n$$

where the sum is over all divisors d of n .

7 Why Do We Use It? Applications

7.1 RSA Encryption - The Most Famous Application

RSA (Rivest–Shamir–Adleman) is the most widely used public-key cryptosystem, securing almost all modern internet communications (HTTPS, SSH, email encryption, etc.). Euler's totient function is **central** to its operation.

RSA Encryption System

Key Generation

Step 1: Choose two large prime numbers p and q

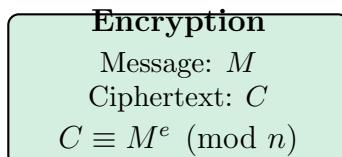
Step 2: Compute $n = p \times q$

Step 3: Compute $\phi(n) = (p - 1)(q - 1)$ ← Uses Totient!

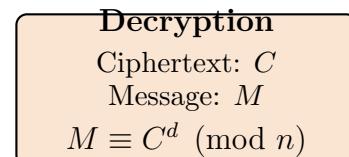
Step 4: Choose e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$

Step 5: Find d such that $d \cdot e \equiv 1 \pmod{\phi(n)}$

Public Key: (n, e) **Private Key:** (n, d)



Send C



7.1.1 Concrete RSA Example

Example 7.1: Simple RSA Encryption

Key Generation:

1. Choose primes: $p = 5, q = 11$
2. Compute: $n = 5 \times 11 = 55$
3. Compute totient: $\phi(55) = \phi(5) \times \phi(11) = 4 \times 10 = 40$
4. Choose: $e = 3$ (since $\gcd(3, 40) = 1$)
5. Find d : We need $3d \equiv 1 \pmod{40}$

Testing: $3 \times 27 = 81 = 2 \times 40 + 1 \equiv 1 \pmod{40}$

So $d = 27$

Public Key: $(n = 55, e = 3)$

Private Key: $(n = 55, d = 27)$

Encryption: Message $M = 13$

$$\begin{aligned} C &\equiv 13^3 \pmod{55} \\ &\equiv 2197 \pmod{55} \\ &\equiv 52 \pmod{55} \end{aligned}$$

Ciphertext: $C = 52$

Decryption:

$$M \equiv 52^{27} \pmod{55}$$

Using modular exponentiation techniques:

$$M \equiv 13 \pmod{55}$$

We successfully recovered the original message!

7.2 Why Totient Function Makes RSA Secure

Key Point

The security of RSA relies on the fact that:

- Computing $\phi(n)$ is **easy** if you know the prime factors p and q
- Computing $\phi(n)$ is **extremely hard** if you only know n but not its factorization

This asymmetry creates the "trapdoor" that makes public-key cryptography possible.

8 Step-by-Step Algorithm Summary

Step 1: Find the prime factorization of $n = p_1^{a_1} \cdot p_2^{a_2} \cdots p_k^{a_k}$



Step 2: Identify all unique prime factors: p_1, p_2, \dots, p_k



Step 3: Apply the formula: $\phi(n) = n \prod_{i=1}^k \left(1 - \frac{1}{p_i}\right)$



Step 4: Simplify to get the final value of $\phi(n)$

9 Practice Problems

9.1 Problem 1

Calculate $\phi(20)$.

Solution:

- Prime factorization: $20 = 2^2 \times 5$
- Prime factors: 2 and 5
- Apply formula: $\phi(20) = 20 \times \left(1 - \frac{1}{2}\right) \times \left(1 - \frac{1}{5}\right)$
- $= 20 \times \frac{1}{2} \times \frac{4}{5} = 20 \times \frac{4}{10} = 8$

Answer: $\phi(20) = 8$

9.2 Problem 2

Calculate $\phi(100)$.

Solution:

- Prime factorization: $100 = 2^2 \times 5^2$
- Prime factors: 2 and 5
- Apply formula: $\phi(100) = 100 \times \left(1 - \frac{1}{2}\right) \times \left(1 - \frac{1}{5}\right)$
- $= 100 \times \frac{1}{2} \times \frac{4}{5} = 100 \times \frac{4}{10} = 40$

Answer: $\phi(100) = 40$

10 Conclusion

Euler's Totient Function is a cornerstone of number theory with profound applications in modern cryptography. Its elegant mathematical properties enable secure communication across the internet through RSA encryption. Understanding $\phi(n)$ provides insight into:

- The distribution of coprime numbers
- Multiplicative number theory
- The foundations of public-key cryptography
- Computational complexity and security

The function's dual nature—easy to compute with factorization, hard without—exemplifies the beautiful connection between pure mathematics and practical applications in computer science.

11 Algorithm Implementation

This section provides practical implementations of Euler's Totient Function, including pseudocode, flowcharts, and Java code examples.

11.1 Pseudocode Algorithms

11.1.1 Method 1: Brute Force Approach

Algorithm 1: Brute Force Totient

```
Algorithm: EulerTotientBruteForce(n)
Input: A positive integer n
Output: phi(n) - count of numbers coprime to n

1. count ← 0
2. for i ← 1 to n do
3.     if gcd(i, n) = 1 then
4.         count ← count + 1
5.     end if
6. end for
7. return count
```

Time Complexity: $O(n \log n)$

Space Complexity: $O(1)$

11.1.2 Method 2: Optimized Using Prime Factorization

Algorithm 2: Optimized Totient (Formula-Based)

```
Algorithm: EulerTotientOptimized(n)
Input: A positive integer n
Output: phi(n) using Euler's product formula

1. result ← n
2. temp ← n
3. p ← 2
4. while p * p <= temp do
5.     if temp mod p = 0 then
6.         // Remove all occurrences of p
7.         while temp mod p = 0 do
8.             temp ← temp / p
9.         end while
10.        // Apply formula: result *= (1 - 1/p)
11.        result ← result - result / p
12.    end if
13.    p ← p + 1
14. end while
15. // If temp > 1, then it's a prime factor
16. if temp > 1 then
17.     result ← result - result / temp
18. end if
19. return result
```

Time Complexity: $O(\sqrt{n})$

Space Complexity: $O(1)$

11.2 Algorithm Flowchart

11.3 Java Implementation

11.3.1 Method 1: Brute Force Implementation

Java Code: Brute Force Method

```
public class EulerTotient {  
  
    /**  
     * Computes GCD using Euclidean Algorithm  
     * @param a First number  
     * @param b Second number  
     * @return GCD of a and b  
     */  
    public static int gcd(int a, int b) {  
        if (b == 0) {  
            return a;  
        }  
        return gcd(b, a % b);  
    }  
  
    /**
```

```

 * Computes Euler's Totient using brute force
 * Time Complexity: O(n log n)
 * @param n Input number
 * @return phi(n)
 */
public static int phiBruteForce(int n) {
    int count = 0;

    // Check each number from 1 to n
    for (int i = 1; i <= n; i++) {
        if (gcd(i, n) == 1) {
            count++;
        }
    }

    return count;
}

// Example usage
public static void main(String[] args) {
    int n = 6;
    System.out.println("phi(" + n + ") = "
                        + phiBruteForce(n));
    // Output: phi(6) = 2

    n = 12;
    System.out.println("phi(" + n + ") = "
                        + phiBruteForce(n));
    // Output: phi(12) = 4
}
}

```

11.3.2 Method 2: Optimized Implementation

Java Code: Optimized Method (Formula-Based)

```

public class EulerTotientOptimized {

    /**
     * Computes Euler's Totient using prime factorization
     * Time Complexity: O(sqrt(n))
     * @param n Input number
     * @return phi(n)
     */
    public static int phiOptimized(int n) {
        int result = n; // Initialize result as n

```

```
// Consider all prime factors of n
for (int p = 2; p * p <= n; p++) {

    // Check if p is a prime factor
    if (n % p == 0) {

        // Remove all occurrences of p
        while (n % p == 0) {
            n = n / p;
        }

        // Apply formula: phi(n) *= (1 - 1/p)
        // Equivalent to: result = result - result/p
        result = result - result / p;
    }
}

// If n is still greater than 1,
// then it's a prime factor
if (n > 1) {
    result = result - result / n;
}

return result;
}

// Example usage with multiple test cases
public static void main(String[] args) {
    int[] testCases = {6, 12, 20, 100, 11};

    System.out.println("Euler's Totient Function");
    System.out.println("====");

    for (int n : testCases) {
        System.out.printf("phi(%3d) = %3d\n",
                          n, phiOptimized(n));
    }

    /* Output:
     * Euler's Totient Function
     * =====
     * phi( 6) =   2
     * phi(12) =   4
     * phi( 20) =   8
     * phi(100) =  40
     * phi( 11) =  10
     */
}
```

```
    }  
}
```

11.3.3 Complete Program with Both Methods

Complete Java Program

```
public class TotientCalculator {  
  
    // GCD using Euclidean algorithm  
    public static int gcd(int a, int b) {  
        return (b == 0) ? a : gcd(b, a % b);  
    }  
  
    // Brute force method  
    public static int phiBruteForce(int n) {  
        int count = 0;  
        for (int i = 1; i <= n; i++) {  
            if (gcd(i, n) == 1) count++;  
        }  
        return count;  
    }  
  
    // Optimized method  
    public static int phiOptimized(int n) {  
        int result = n;  
        for (int p = 2; p * p <= n; p++) {  
            if (n % p == 0) {  
                while (n % p == 0) n /= p;  
                result -= result / p;  
            }  
        }  
        if (n > 1) result -= result / n;  
        return result;  
    }  
  
    // Main method with comparison  
    public static void main(String[] args) {  
        int n = 100;  
  
        long startTime, endTime;  
  
        // Test brute force  
        startTime = System.nanoTime();  
        int result1 = phiBruteForce(n);  
        endTime = System.nanoTime();  
        System.out.println("Brute Force:");
```

```

        System.out.println("  Result: phi(" + n
                           + ") = " + result1);
        System.out.println("  Time: "
                           + (endTime - startTime) + " ns");

    // Test optimized
    startTime = System.nanoTime();
    int result2 = phiOptimized(n);
    endTime = System.nanoTime();
    System.out.println("\nOptimized:");
    System.out.println("  Result: phi(" + n
                       + ") = " + result2);
    System.out.println("  Time: "
                       + (endTime - startTime) + " ns");
}
}

```

11.4 Understanding the Code

11.4.1 Key Points

- GCD Function:** Uses Euclidean algorithm with recursion. The base case is when $b = 0$.
- Brute Force:** Simple and intuitive. Checks every number from 1 to n . Good for understanding but slow for large n .
- Optimized Method:** Uses Euler's product formula. Only checks up to \sqrt{n} , making it much faster.
- Formula Application:** Instead of $(1 - \frac{1}{p})$, we use `result - result/p` to avoid floating-point arithmetic.

11.4.2 Time Complexity Comparison

Method	Time Complexity	Best For
Brute Force	$O(n \log n)$	Small values, learning
Optimized	$O(\sqrt{n})$	Large values, production

12 References

- Euler, L. (1763). "Theoremata arithmetica nova methodo demonstrata"
- Rivest, R., Shamir, A., & Adleman, L. (1978). "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems"
- Hardy, G. H., & Wright, E. M. (2008). "An Introduction to the Theory of Numbers"
- Apostol, T. M. (1976). "Introduction to Analytic Number Theory"