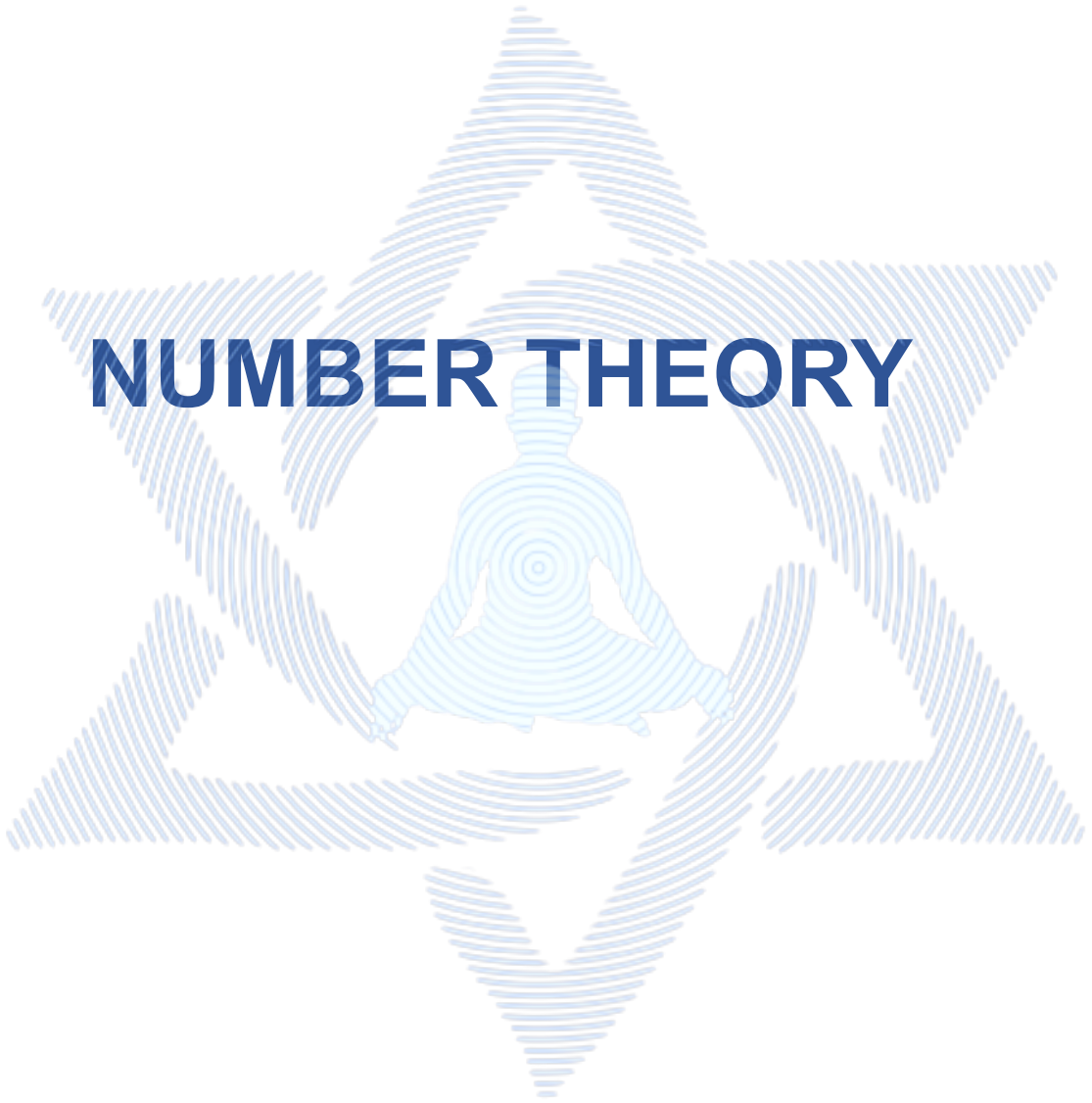


NUMBER THEORY



Lecture - 4

Topic to be covered :

1. Intro to modular arithmetic
2. Modular Multiplicative Inverse
3. Modular exponentiation 3 methods
4. Fermet's little theorem
5. Wilson's theorem
6. Euler totient

Introduction to Modular Arithmetic

Modular functions come into use when answers to the question go beyond a limit . So we need to use modulus operation(%) which returns the remainder of the two numbers.

$a \% b$, it will return the value with the range $[0, b-1]$.

Now when we divide a negative number and find out the mod, the answer will be negative, but it will not be in the range which is required.

So, what are we going to do in that case???

We will add the mod to the answer

$$A \% B = (-r) + B$$

The basic modulus function are:

$$(A + B) \% m = (A \% m + B \% m) \% m$$

$$(A - B) \% m = (A \% m - B \% m + m) \% m$$

$$A * B \% m = ((A \% m) * (B \% m)) \% m$$

But it didn't work for division for all cases:

$$(A / B) \% m = ((A \% m) / (B \% m)) \% m$$

Why does modular division not work in all cases?

$$(A/B) \% m = ((A\%m) / (B\%m)) \% m$$

Let's take B to be the multiple of m, in that case B%m will be equal to 0, and we can't divide any number with 0. So it won't work always.

Modulus division will be applied only if modulo multiplicative inverse exists for the number.

Modular Multiplicative Inverse

Given two integers 'a' and 'm', find modular multiplicative inverse of 'a' under modulo 'm'.

The modular multiplicative inverse is an integer 'x' such that.

$$A * x \equiv 1 \pmod{m}$$

The value of x should be in $\{1, 2, \dots, m-1\}$, i.e., in the range of integer modulo m. (Note that x cannot be 0 as $a*0 \pmod{m}$ will never be 1)

The multiplicative inverse of "a modulo m" exists if and only if a and m are relatively prime (i.e., if $\gcd(a, m) = 1$).

Input: a = 3, m = 11

Output: 4

Input: a = 10, m = 17

Output: 12

Method 1 (Naive)

A Naive method is to try all numbers from 1 to m. For every number x, check if $(a*x)\%m$ is 1.

Code:

```
1. #include <iostream>
```

```

2. using namespace std;
3. int modInverse(int a, int m)
4. {
5.     for (int x = 1; x < m; x++)
6.         if (((a%m) * (x%m)) % m == 1)
7.             return x;
8. }
9. int main()
10. {
11.     int a = 3, m = 11;
12.
13.     cout << modInverse(a, m);
14.     return 0;
15. }
16.

```

Method 2:

The idea is to use Extended Euclidean theorem that takes two integers 'a' and 'b', finds their gcd and also find 'x' and 'y' such that

$$ax + by = \text{gcd}(a, b)$$

To find the multiplicative inverse of 'a' under 'm', we put $b = m$ in above formula. Since we know that a and m are relatively prime, we can put the value of gcd as 1.

$$ax + my = 1$$

If we take modulo m on both sides, we get

$$ax + my \equiv 1 \pmod{m}$$

We can remove the second term on the left side as ' $my \pmod m$ ' would always be 0 for an integer y .

$$ax \equiv 1 \pmod m$$

So the ' x ' that we can find using the Extended Euclid Algorithm is the multiplicative inverse of ' a '.

Code:

```
1. #include <iostream>
2. using namespace std;
3. int gcdExtended(int a, int b, int* x, int* y);
4. void modInverse(int a, int m)
5. {
6.     int x, y;
7.     int g = gcdExtended(a, m, &x, &y);
8.     if (g != 1)
9.         cout << "Inverse doesn't exist";
10.    else
11.    {
12.        int res = (x % m + m) % m;
13.        cout << "Modular multiplicative inverse is " << res;
14.    }
15.}
16.int gcdExtended(int a, int b, int* x, int* y)
17.{
18.    if (a == 0)
19.    {
20.        *x = 0, *y = 1;
21.        return b;
22.    }
23.    int x1, y1;
```



```

24. int gcd = gcdExtended(b % a, a, &x1, &y1);
25. *x = y1 - (b / a) * x1;
26. *y = x1;
27. return gcd;
28.}
29.int main()
30.{
31. int a = 3, m = 11;
32. modInverse(a, m);
33. return 0;
34.}
35.

```

Code for modular division using the above modular inverse function:

```

1. void modDivide(int a, int b, int m)
2. {
3.     a = a % m;
4.     int inv = modInverse(b, m);
5.     if (inv == -1)
6.         printf("Division not defined");
7.     else
8.     {
9.         int c = (inv * a) % m ;
10.        printf("Result of division is %d", c) ;
11.    }
12. }
13.

```

Modular Exponentiation (Power in Modular Arithmetic)

Given three numbers x , y and p , compute $(x^y) \% p$.

Input: $x = 2, y = 3, p = 5$

Output: 3

Input: $x = 2, y = 5, p = 13$

Output: 6

Method 1:

We can run a loop from 1 to the y and keep on multiplying the result with a but its time complexity will be $O(n)$.

Method 2:(recursive app)

Code:

```
1. int Pow(int a,int b,int m){
2. if(b==0) return 1;
3. if(b%2==0)
4.     Return Pow((a*a)%m,b/2,m)%m;
5. return a*(Pow((a*a)%m,(b-1)/2,m)%m)%m;
6. }
7.
```

Method 3:

Code:

```
1. int power(long long x, unsigned int y, int p)
2. {
3.     int res = 1;
4.     x = x % p;
5.     if (x == 0) return 0;
6.     while (y > 0)
7.     {
8.         if (y & 1)
```

```

9.      res = (res*x) % p;
10.     y = y>>1; // y = y/2
11.     x = (x*x) % p;
12.     }
13.     return res;
14.     }
15.

```

Fermat's little theorem

If n is a prime number, then for every a , $1 < a < n-1$,

$$a^{n-1} \equiv 1 \pmod{n}$$

OR

$$a^{n-1} \% n = 1$$

Example: Since 5 is prime, $2^4 \equiv 1 \pmod{5}$ [or $2^4 \% 5 = 1$],
 $3^4 \equiv 1 \pmod{5}$ and $4^4 \equiv 1 \pmod{5}$

Since 7 is prime, $2^6 \equiv 1 \pmod{7}$,
 $3^6 \equiv 1 \pmod{7}$, $4^6 \equiv 1 \pmod{7}$
 $5^6 \equiv 1 \pmod{7}$ and $6^6 \equiv 1 \pmod{7}$

Fermat's theorem is useful for finding modular multiple inverse.

Let P be a prime number. For any integer A ,

$$(A^{P-1}) \% P = 1 \quad \dots (1)$$

Multiply A inverse on both side (A^{-1}):

$$(A^{-1})(A^{(P-1)}) \bmod P = (A^{-1}) \bmod P$$

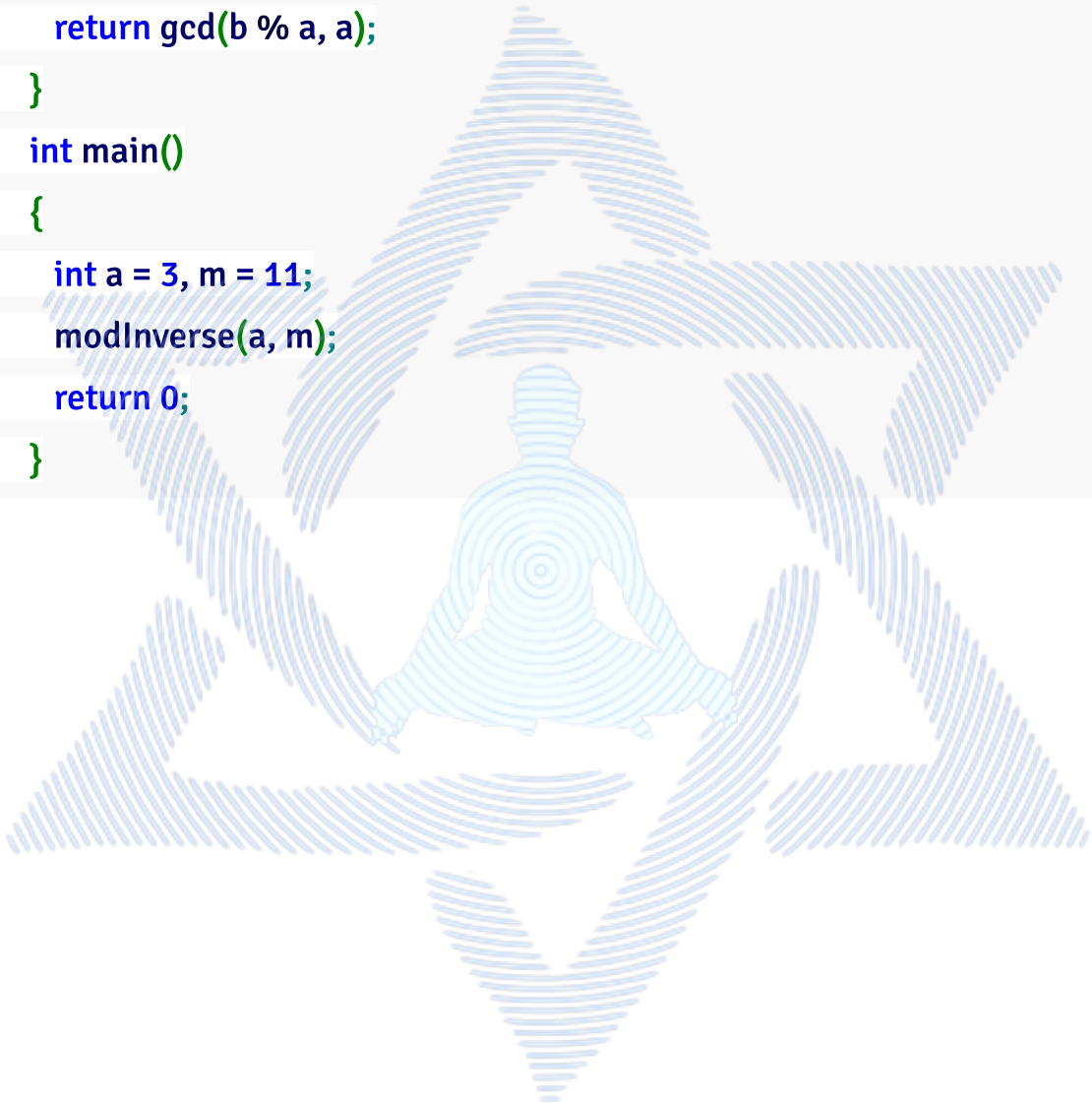
$$(A^{(P-2)}) \bmod P = (A^{-1}) \bmod P$$

Which means if we find out A to the power P-2 modulo P, then the result will be equal to A inverse modulo P.

Code:

```
1. #include <iostream>
2. using namespace std;
3. int gcd(int a, int b);
4. int power(int x, unsigned int y, unsigned int m);
5. void modInverse(int a, int m)
6. {
7.     int g = gcd(a, m);
8.     if (g != 1)
9.         cout << "Inverse doesn't exist";
10.    else
11.        { cout << "Modular multiplicative inverse is "
12.          << power(a, m - 2, m);
13.        }
14.    }
15. int power(int x, unsigned int y, unsigned int m)
16. {
17.     if (y == 0)
18.         return 1;
19.     int p = power(x, y / 2, m) % m;
20.     p = (p * p) % m;
21.     return (y % 2 == 0) ? p : (x * p) % m;
```

```
22. }
23. int gcd(int a, int b)
24. {
25.     if (a == 0)
26.         return b;
27.     return gcd(b % a, a);
28. }
29. int main()
30. {
31.     int a = 3, m = 11;
32.     modInverse(a, m);
33.     return 0;
34. }
```



Wilson's Theorem

Wilson's theorem states that a natural number $p > 1$ is a prime number if and only if

$$(p - 1)! \equiv -1 \pmod{p}$$

$$\text{OR } (p - 1)! \equiv (p-1) \pmod{p}$$

Iff p is a prime, then $(p - 1)! + 1$ is a multiple of p , that is

$$(p - 1)! \equiv -1 \pmod{p}.$$

$$p = 5$$

$$(p-1)! = 24$$

$$24 \% 5 = 4$$

$$p = 7$$

$$(p-1)! = 6! = 720$$

$$720 \% 7 = 6$$

How does it work?

1) We can quickly check the result for $p = 2$ or $p = 3$.

2) For $p > 3$: If p is composite, then its positive divisors are among the integers $1, 2, 3, 4, \dots, p-1$ and it is clear that $\gcd((p-1)!, p) > 1$, so we can not have $(p-1)! \equiv -1 \pmod{p}$.

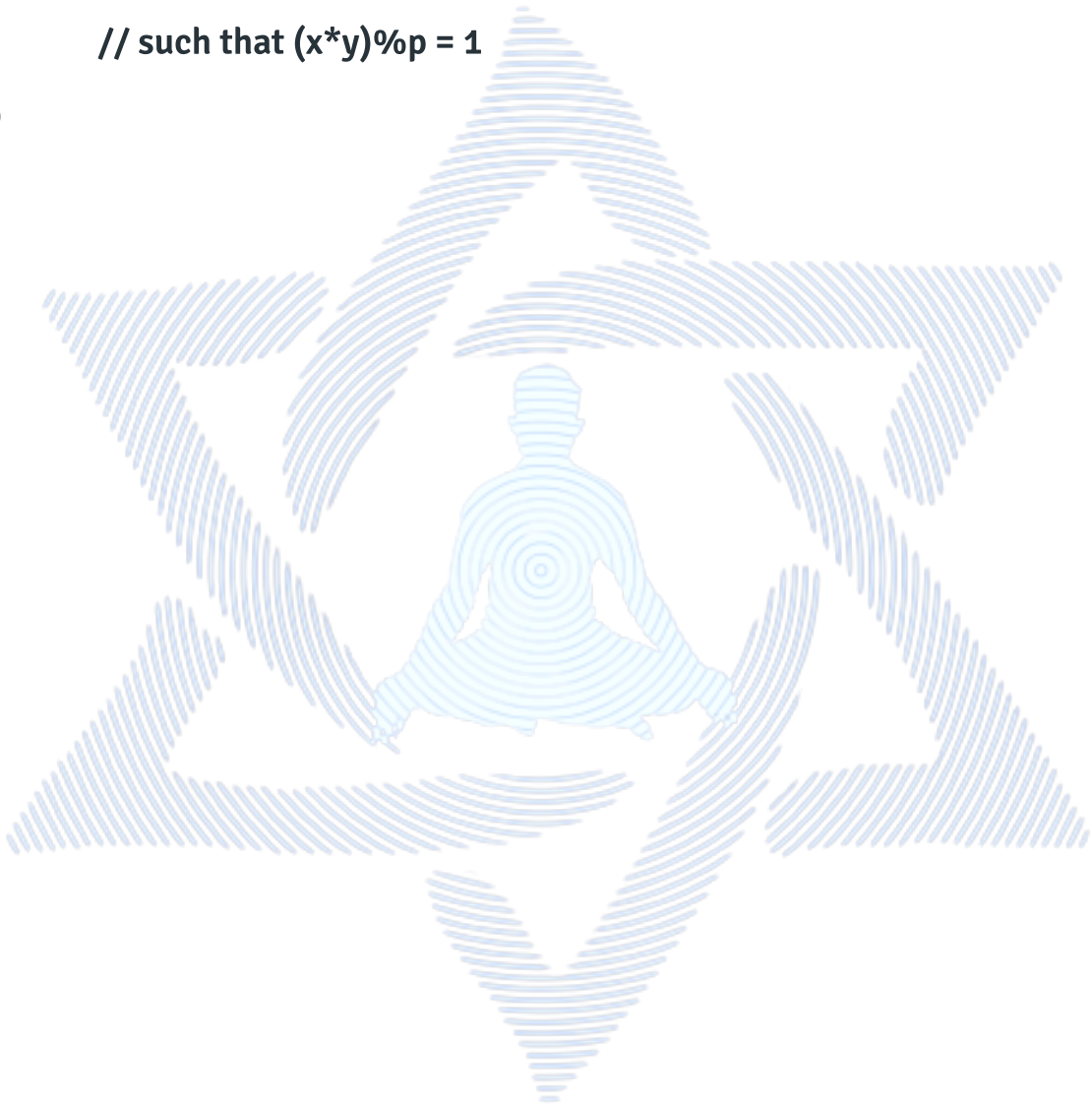
3) Now let us see how it is exactly -1 when p is a prime. If p is a prime, then all numbers in $[1, p-1]$ are relatively prime to p. And for every number x in range $[2, p-2]$, there must exist a pair y such that $(x*y) \% p = 1$. So

$$[1 * 2 * 3 * \dots * (p-1)] \% p$$

$$= [1 * 1 * 1 \dots (p-1)] \text{ // Group all x and y in } [2..p-2]$$

$$\text{// such that } (x*y) \% p = 1$$

$$= (p-1)$$



Euler Totient Function

The totient function $\phi(n)$, also called Euler's totient function, is defined as the number of positive integers $\leq n$ that are relatively prime to (i.e., do not contain any factor in common with) n , where 1 is counted as being relatively prime to all numbers. Since a number less than or equal to and relatively prime to a given number is called a totative, the totient function $\phi(n)$ can be simply defined as the number of totatives of n . For example, there are eight totatives of 24 (1, 5, 7, 11, 13, 17, 19, and 23), so $\phi(24)=8$.

Property of Euler Function:

$$\phi(a,b) = \phi(a)*\phi(b), \text{ given } \gcd(a,b)=1$$

Euler's product formula:

It states

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right),$$

If Number is prime Number then,

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

If p is prime and $k \geq 1$, then

$$\varphi(p^k) = p^k - p^{k-1} = p^{k-1}(p - 1) = p^k \left(1 - \frac{1}{p}\right).$$

Proof: Since p is a prime number, the only possible values of $\gcd(p^k, m)$ are $1, p, p^2, \dots, p^k$, and the only way to have $\gcd(p^k, m) > 1$ is if m is a multiple of p , i.e. $m = p, 2p, 3p, \dots, p^{k-1}p = p^k$, and there are p^{k-1} such multiples less than p^k . Therefore, the other $p^k - p^{k-1}$ numbers are all relatively prime to p^k .

$$\begin{aligned}
 \varphi(n) &= \varphi(p_1^{k_1}) \varphi(p_2^{k_2}) \cdots \varphi(p_r^{k_r}) \\
 &= p_1^{k_1-1} (p_1 - 1) p_2^{k_2-1} (p_2 - 1) \cdots p_r^{k_r-1} (p_r - 1) \\
 &= p_1^{k_1} \left(1 - \frac{1}{p_1}\right) p_2^{k_2} \left(1 - \frac{1}{p_2}\right) \cdots p_r^{k_r} \left(1 - \frac{1}{p_r}\right) \\
 &= p_1^{k_1} p_2^{k_2} \cdots p_r^{k_r} \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_r}\right) \\
 &= n \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_r}\right).
 \end{aligned}$$

Code:-

Simple solution:

```

1.  #include <iostream>
2.  using namespace std;
3.
4.  // Function to return gcd of a and b
5.  int gcd(int a, int b)
6.  {
7.      if (a == 0)
8.          return b;
9.      return gcd(b % a, a);
10. }

```

```
11.
12. // A simple method to evaluate Euler Totient
    Function
13. int phi(unsigned int n)
14. {
15.     unsigned int result = 1;
16.     for (int i = 2; i < n; i++)
17.         if (gcd(i, n) == 1)
18.             result++;
19.     return result;
20. }
21.
22.
23. int main()
24. {
25.     int n;
26.     for (n = 1; n <= 10; n++)
27.         cout << "phi("<<n<<" ) = " << phi(n) << endl;
28.     return 0;
29. }
```

Efficient Approach :-

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int phi(int n)
5. {
6.     // Initialize result as n
7.     float result = n;
8.
9.     // Consider all prime factors of n
10.    // and for every prime factor p,
11.    // multiply result with (1 - 1/p)
12.    for(int p = 2; p * p <= n; ++p)
13.    {
14.        // Check if p is a prime factor.
15.        if (n % p == 0)
16.        {
17.            // If yes, then update n and result
18.            while (n % p == 0)
19.                n /= p;
20.            result *= (1.0 - (1.0 / (float)p));
21.        }
22.    }
23.    // If n has a prime factor greater than sqrt(n)
24.    // (There can be at-most one such prime factor)
25.    if (n > 1)
26.        result *= (1.0 - (1.0 / (float)n));
27.
28.    return (int)result;
```

```

29. }
30.
31. // Driver code
32. int main()
33. {
34.     int n;
35.     for(n = 1; n <= 10; n++)
36.     {
37.         cout << "Phi" << "("
38.             << n << ")" << " = "
39.             << phi(n) << endl;
40.     }
41.     return 0;
42. }

```

Some Interesting Properties of Euler's Totient Function

- 1) For a prime number p , $\Phi(p)$ is $p-1$. For example $\Phi(5)$ is 4, $\Phi(7)$ is 6 and $\Phi(13)$ is 12. This is obvious, gcd of all numbers from 1 to $p-1$ will be 1 because p is a prime.
- 2) For two numbers a and b , if $\text{gcd}(a, b)$ is 1, then $\Phi(ab) = \Phi(a) * \Phi(b)$. For example $\Phi(5)$ is 4 and $\Phi(6)$ is 2, so $\Phi(30)$ must be 8 as 5 and 6 are relatively prime.
- 3) For any two prime numbers p and q , $\Phi(pq) = (p-1)*(q-1)$. This property is used in the RSA algorithm.
- 4) If p is a prime number, then $\Phi(pk) = pk - pk-1$. This can be proved using Euler's product formula.

5) Sum of values of totient functions of all divisors of n is equal to n . For example, $n = 6$, the divisors of n are 1, 2, 3 and 6. According to Gauss, the sum of $\Phi(1) + \Phi(2) + \Phi(3) + \Phi(6)$ should be 6. We can verify the same by putting values, we get $(1 + 1 + 2 + 2) = 6$.

