**Euler’s Totient Function**

Euler’s Totient function ?(n) for an input n is count of numbers in {1, 2, 3, …, n} that are relatively prime to n, i.e., the numbers whose GCD (Greatest Common Divisor) with n is 1.

**Examples :**

?(1) = 1

gcd(1, 1) is 1

?(2) = 1

gcd(1, 2) is 1, but gcd(2, 2) is 2.

?(3) = 2

gcd(1, 3) is 1 and gcd(2, 3) is 1

?(4) = 2

gcd(1, 4) is 1 and gcd(3, 4) is 1

?(5) = 4

gcd(1, 5) is 1, gcd(2, 5) is 1,

gcd(3, 5) is 1 and gcd(4, 5) is 1

?(6) = 2

gcd(1, 6) is 1 and gcd(5, 6) is 1,

**How to compute ?(n) for an input n?**

A **simple solution** is to iterate through all numbers from 1 to n-1 and count numbers with gcd with n as 1. Below is the implementation of the simple method to compute Euler’s Totient function for an input integer n.

#include <iostream>

using namespace std;

int gcd(int a, int b) {

    if (a == 0)

        return b;

    return gcd(b % a, a);

}

// A simple method to evaluate Euler Totient Function

int phi(unsigned int n) {

    unsigned int result = 1;

    for (int i = 2; i < n; i++)

        if (gcd(i, n) == 1)

            result++;

    return result;

}

int main()  {

    int n;

    for (n = 1; n <= 10; n++)

        cout << "phi("<<n<<") = " << phi(n) << endl;

    return 0;

}

**Output :**

phi(1) = 1

phi(2) = 1

phi(3) = 2

phi(4) = 2

phi(5) = 4

phi(6) = 2

phi(7) = 6

phi(8) = 4

phi(9) = 6

phi(10) = 4

The above code calls gcd function O(n) times. Time complexity of the gcd function is O(h) where h is number of digits in smaller number of given two numbers. Therefore, an upper bound on time complexity of above solution is O(nLogn) [How? there can be at most Log10n digits in all numbers from 1 to n]

Below is a **Better Solution**. The idea is based on Euler’s product formula which states that value of totient functions is below product over all prime factors p of n.  
[eulersproduct](https://media.geeksforgeeks.org/wp-content/cdn-uploads/Eulers_Totient_function.png)  
The formula basically says that the value of ?(n) is equal to n multiplied by product of (1 – 1/p) for all prime factors p of n. For example value of ?(6) = 6 \* (1-1/2) \* (1 – 1/3) = 2.

We can find all prime factors using the idea used in this approach:

# Efficient program to print all prime factors of a given number

Following are the steps to find all prime factors.

**1)** While n is divisible by 2, print 2 and divide n by 2.  
**2)** After step 1, n must be odd. Now start a loop from i = 3 to square root of n. While i divides n, print i and divide n by i. After i fails to divide n, increment i by 2 and continue.  
**3)** If n is a prime number and is greater than 2, then n will not become 1 by above two steps. So print n if it is greater than 2.

#include <bits/stdc++.h>

using namespace std;

void primeFactors(int n)

{

    // Print the number of 2s that divide n

    while (n % 2 == 0)

    {

        cout << 2 << " ";

        n = n/2;

    }

    // n must be odd at this point. So we can skip

    // one element (Note i = i +2)

for (int i = 3; i <= sqrt(n); i = i + 2)

    {

        // While i divides n, print i and divide n

        while (n % i == 0)

        {

            cout << i << " ";

            n = n/i;

        }

    }

    // This condition is to handle the case when n

    // is a prime number greater than 2

    if (n > 2)

        cout << n << " ";

}

int main()

{

    int n = 315;

    primeFactors(n);

    return 0;

}

Output:

3 3 5 7

**How does this work?**  
The steps 1 and 2 take care of composite numbers and step 3 takes care of prime numbers. To prove that the complete algorithm works, we need to prove that steps 1 and 2 actually take care of composite numbers. This is clear that step 1 takes care of even numbers. And after step 1, all remaining prime factor must be odd (difference of two prime factors must be at least 2), this explains why i is incremented by 2.  
Now the main part is, the loop runs till square root of n not till n. To prove that this optimization works, let us consider the following property of composite numbers.  
Every composite number has at least one prime factor less than or equal to square root of itself.  
This property can be proved using counter statement. Let a and b be two factors of n such that a\*b = n. If both are greater than √n, then a.b > √n, \* √n, which contradicts the expression “a \* b = n”.

In step 2 of the above algorithm, we run a loop and do following in loop  
a) Find the least prime factor i (must be less than √n,)  
b) Remove all occurrences i from n by repeatedly dividing n by i.  
c) Repeat steps a and b for divided n and i = i + 2. The steps a and b are repeated till n becomes either 1 or a prime number.

Now back to our Euler’s product formula:

**1) Initialize : result = n**

**2) Run a loop from 'p' = 2 to sqrt(n), do following for every 'p'.**

**a) If p divides n, then**

**Set: result = result \* (1.0 - (1.0 / (float) p));**

**Divide all occurrences of p in n.**

**3) Return result**

#include <stdio.h>

int phi(int n)

{

    float result = n; // Initialize result as n

    // Consider all prime factors of n and for every prime

    // factor p, multiply result with (1 - 1/p)

    for (int p = 2; p \* p <= n; ++p) {

        // Check if p is a prime factor.

        if (n % p == 0) {

            // If yes, then update n and result

            while (n % p == 0)

                n /= p;

            result \*= (1.0 - (1.0 / (float)p));

        }

    }

    // If n has a prime factor greater than sqrt(n)

    // (There can be at-most one such prime factor)

    if (n > 1)

        result \*= (1.0 - (1.0 / (float)n));

    return (int)result;

}

// Driver program to test above function

int main()

{

    int n;

    for (n = 1; n <= 10; n++)

        printf("phi(%d) = %d\n", n, phi(n));

    return 0;

}

**Output :**

phi(1) = 1

phi(2) = 1

phi(3) = 2

phi(4) = 2

phi(5) = 4

phi(6) = 2

phi(7) = 6

phi(8) = 4

phi(9) = 6

phi(10) = 4

We can avoid floating point calculations in above method. The idea is to count all prime factors and their multiples and subtract this count from n to get the totient function value (Prime factors and multiples of prime factors won’t have gcd as 1)

1) Initialize result as n

2) Consider every number 'p' (where 'p' varies from 2 to ?n).

If p divides n, then do following

a) Subtract all multiples of p from 1 to n [all multiples of p

will have gcd more than 1 (at least p) with n]

b) Update n by repeatedly dividing it by p.

3) If the reduced n is more than 1, then remove all multiples

of n from result.

Below is the implementation of above algorithm.

#include <stdio.h>

int phi(int n) {

    int result = n; // Initialize result as n

    // Consider all prime factors of n and subtract their

    // multiples from result

    for (int p = 2; p \* p <= n; ++p) {

        // Check if p is a prime factor.

        if (n % p == 0) {

            // If yes, then update n and result

            while (n % p == 0)

                n /= p;

            result -= result / p;

        }

    }

    // If n has a prime factor greater than sqrt(n)

    // (There can be at-most one such prime factor)

    if (n > 1)

        result -= result / n;

    return result;

}

int main() {

    int n;

    for (n = 1; n <= 10; n++)

        printf("phi(%d) = %d\n", n, phi(n));

    return 0;

}

**Output :**

phi(1) = 1

phi(2) = 1

phi(3) = 2

phi(4) = 2

phi(5) = 4

phi(6) = 2

phi(7) = 6

phi(8) = 4

phi(9) = 6

phi(10) = 4

**Some Interesting Properties of Euler’s Totient Function**  
**1)** For a prime number p, ?(p) is p-1. For example ?(5) is 4, ?(7) is 6 and ?(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 gcd(a, b) is 1, then ?(ab) = ?(a) \* ?(b). For example ?(5) is 4 and ?(6) is 2, so ?(30) must be 8 as 5 and 6 are relatively prime.

**3)** For any two prime numbers p and q, ?(pq) = (p-1)\*(q-1). This property is used in RSA algorithm.

**4)** If p is a prime number, then ?(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.  
[gausss](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gausss.png)  
For example, n = 6, the divisors of n are 1, 2, 3 and 6. According to Gauss, sum of ?(1) + ?(2) + ?(3) + ?(6) should be 6. We can verify the same by putting values, we get (1 + 1 + 2 + 2) = 6.

**6)** The most famous and important feature is expressed in [***Euler’s theorem***](http://en.wikipedia.org/wiki/Euler%27s_theorem) :

The theorem states that if n and a are coprime

(or relatively prime) positive integers, then

a?(n) ? 1 (mod n)

The [RSA cryptosystem](http://en.wikipedia.org/wiki/RSA_%28algorithm%29) is based on this theorem:

In the particular case when m is prime say p, Euler’s theorem turns into the so-called [***Fermat’s little theorem***](http://en.wikipedia.org/wiki/Fermat%27s_little_theorem) :

ap-1 ? 1 (mod p)

**7)** [Number of generators of a finite cyclic group under modulo n addition is ?(n)](https://www.geeksforgeeks.org/generators-finite-cyclic-group-addition/).

---------------------------------------------------------------------------

***Now coming to our original LCM sum question :***

**∑LCM(i, n) = ((∑(d \* ETF(d)) + 1) \* n) / 2**  
where **ETF(d)** is Euler totient function of **d** and **d** belongs to the **set of divisors of n**.

**Example:**

Let n be 5 then LCM(1, 5) + LCM(2, 5) + LCM(3, 5) + LCM(4, 5) + LCM(5, 5)  
= 5 + 10 + 15 + 20 + 5  
= 55

With Euler Totient Function:  
All divisors of 5 are {1, 5}  
Hence, ((1\*ETF(1) + 5\*ETF(4) + 1) \* 5) / 2 = 55