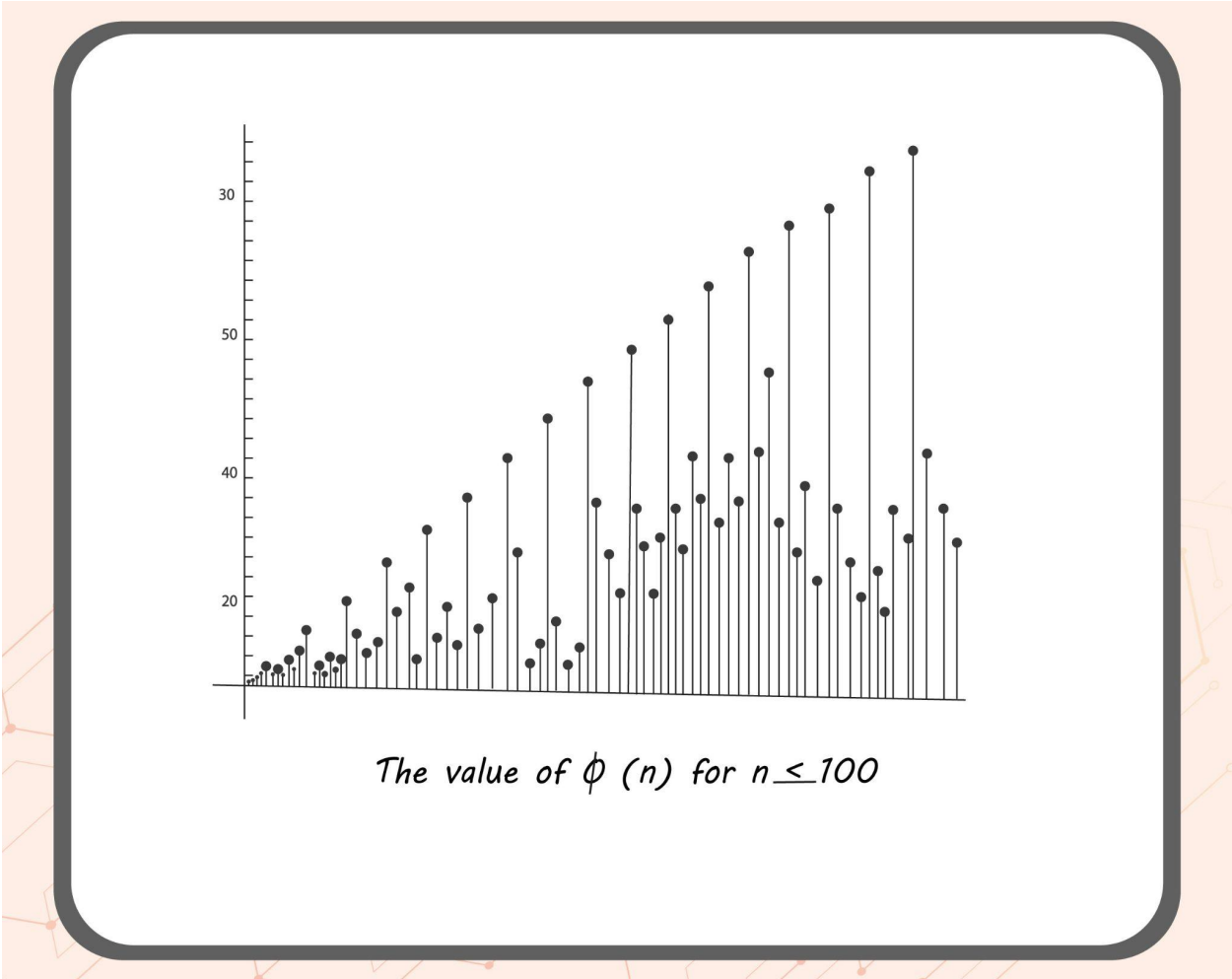
Number Theory 3

Euler's Totient Function

Euler's totient function gives us the count of positive integers less than n that are co prime to n. That is, **ϕ(n)** is the number of **m ∈ N** such that **1 ≤ m < n** and **gcd(m,n)=1**.



For example, Find **ϕ(15)**.

Among positive numbers less than 15, eliminate multiples of 3 or 5, which are {3, 5, 6, 9, 10, 12}. The remaining numbers are **{1,2,4,7,8,11,13,14}**, so **ϕ(15) = 8**.

According to a property of euler totient function :

ϕ(a * b) = ϕ(a) * ϕ(b), given that gcd(a, b) = 1

And we know that any number can be written using prime numbers as **$N \rightarrow 2^a 3^b 5^c$**

Therefore, **$\phi(n) = \phi(p_1^a * p_2^b * p_3^c \ldots\ldots\ldots\ldots * p_k^k)$**

$\rightarrow \quad \phi(p_1^a) * \phi(p_2^b) * \phi(p_3^c) \ldots\ldots\ldots\ldots * \phi(p_k^k)$

Now, **$\phi(p^a)$** means that count of numbers from 1 to p^a that are co-prime to p^a.

And those will be excluding the multiples of p, that is, 1, p, 2p, 3p and so on ..

The total count of these multiples will be **p^{a-1}**. Now,

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

$\phi(p^a) = p^a (1 - (1/p))$

$$\phi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots\left(1 - \frac{1}{p_k}\right)$$

Where k is the number of distinct prime factors.

Basic Approach :

The simple solution is to iterate through all numbers from 1 to N-1 and count numbers whose GCD(Greatest Common Divisor) with N is 1 and we can find the GCD of two numbers in an efficient manner using the Euclidean algorithm.

Approach 2:

The better solution is to use Euler's product formula which states that the total number of integers between 1 and N-1 inclusive which are coprime to N can be found using this formula Euler Product Formula
N*(1-(1/p1))*(1-(1/p2))*.....*(1-(1/pk)) where p1,p2...pk are the prime factors of N.

So now the problem is to find all the prime factors and use the above formula.

To find the prime factors and count the number of integers between 1 and N which are coprime to N below are the steps.

Steps:

1. Initially declare a variable total = N.
2. Then start a loop from p = 2 to sqrt(N) and do the following:
    1. if(N%p==0) then
        1. While p divides N:
            1. Divide N by p.
        2. total =total- total/p
3. If N is a prime number and is greater than 1, then:
    1. total = total-total/N
4. At last, return the total.

OR, we could also use a sieve instead of carrying out the above steps.

Code:

```
#include<iostream>
using namespace std;


void eulerPhi(int n){

    int phi[n+1];

    for(int i=1;i<=n;i++){
        phi[i] = i;
    }

    for(int i=2;i<=n;i++){
        if(phi[i] == i){
            phi[i] = i-1;
            for(int j=2*i;j<=n;j+=i){
                phi[j] = (phi[j]*(i-1))/i;
            }
        }
    }

    for(int i=1;i<=n;i++){
        cout << "Euler Totient Phi For " << i << "Is :" << phi[i]<<endl;
    }

}
int main(){

    eulerPhi(10);
    return 0;
}
```

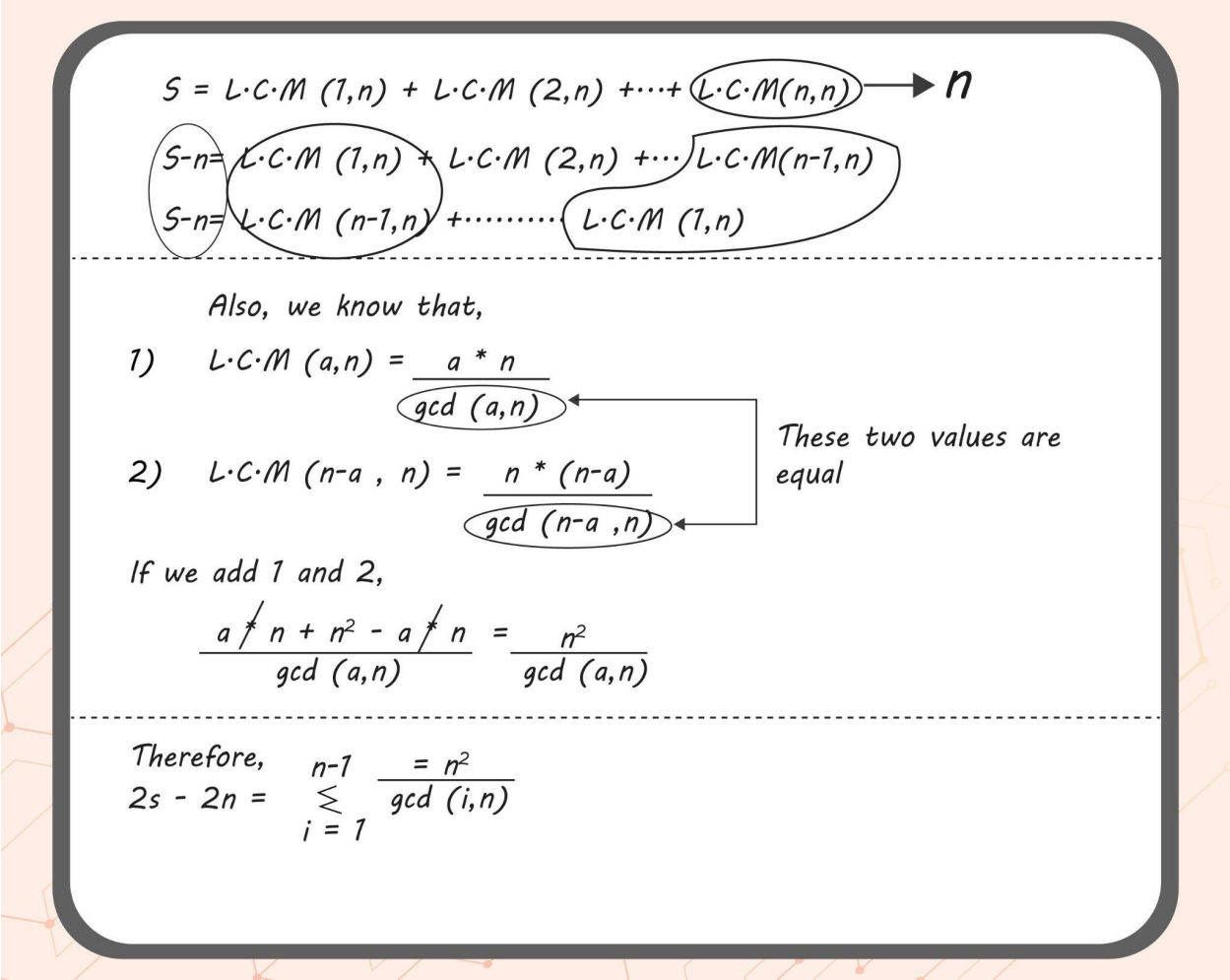
Sum of LCM

Problem Statement : Given n, calculate and print the sum :

LCM(1,n) + LCM(2,n) + .. + LCM(n,n)

where LCM(i,n) denotes the Least Common Multiple of the integers i and n.

Explanation:



Segmented Sieve

We have to calculate the number of primes in a particular range from L to R. Now earlier we solved this question using a sieve of size R, but if R is given to be greater than 10^8 we can't use that approach. But what we can do is make a sieve of size L - R which is well given in our acceptable range, and adjust the indexing accordingly.

Everything else will work according to the previous discussion of sieve of eratosthenes.

Code:

```
#include<bits/stdc++.h>
using namespace std;
#define MAX 100001
vector<int>* sieve(){

    bool isPrime[MAX];
    for(int i=0;i<MAX;i++){
        isPrime[i] = true;
    }
    for(int i=2;i*i<MAX;i++){
        if(isPrime[i]){
            for(int j=i*i;j<MAX;j+=i){
                isPrime[j] = false;
            }
        }
    }
    vector<int>* primes = new vector<int>();
    primes->push_back(2);
    for(int i=3;i<MAX;i+=2){
        if(isPrime[i]){
            primes->push_back(i);
        }
    }
    return primes;
}
void printPrimes(long long l,long long r,vector<int>* & primes){
```

```
    bool isPrime[r-l+1];

    for(int i=0;i<=r-l;i++){
        isPrime[i] = true;
    }

    for(int i=0;primes->at(i)*(long long)primes->at(i) <= r;i++){
        int currPrime = primes->at(i);
        // Just smaller or equal value to l
        long long base = (l/(currPrime))*(currPrime);
        if(base < l){
            base = base + currPrime;
        }

        // Mark all mutliples within L To R as false
        for(long long j = base;j<=r ;j+= currPrime){
            isPrime[j-l] = false;
        }

        // There may be a case where base is itself a prime number .
        if(base == currPrime){
            isPrime[base-l] = true;
        }
    }

    for(int i=0;i<=r-l;i++){
        if(isPrime[i] == true){
            cout << i + l << endl;
        }
    }
}
int main(){
    vector<int>* primes = sieve();
    int t;
    cin >> t;
    while(t--){
        long long l,r;
        cin>>l>>r;
        printPrimes(l,r,primes);
    }
```

```
    return 0;
}
```

Optimized Power Function

We have to find out the **pow (x, n)**.

Basic Approach : We can solve this problem recursively by **x * pow (x, n)**.

Optimized Approach :

If n is even :

Val = pow (x, n)

Pow (x, n) = Val * Val

Example: $\mathbf{x^8 = x^4 * x^4}$

If n is odd :

Val = pow (x, n/2)

pow (x, n) = Val * Val * x

Example: $\mathbf{x^9 = x^4 * x^4 * x}$

Time complexity of this approach will be O(log(n)).

Modular Exponentiation

If in the previous topic of optimized power function, we have to find $\mathbf{2^{1024}}$ then it will be out of the range of integers or long long.

We can prevent the answers to go out of range using modulo arithmetic

(a * b) % c = ((a % c) * (b % c)) % c