# BRAC University Competitive Programming Workshop

Handouts for Day 6

**Ahnaf Shahriar Asif**

Department of CSE
BRAC University
30th April, 2025

# Contents

# 1 Introduction

*A **very** brief introduction to Number Theory and its applications in Competitive Programming*

Number Theory[1], often referred to as the "Queen of Mathematics," is a branch of mathematics that deals with the properties and relationships of numbers, particularly integers. It encompasses a wide range of topics, including prime numbers, divisibility, modular arithmetic, and Diophantine equations. Number Theory has significant applications in various fields, including cryptography, computer science, and algorithm design.

In Competitive Programming, Number Theory is a crucial area of study. Many problems in programming contests require a deep understanding of number-theoretic concepts and techniques. We will study Divisibility, Modular Arithmetic, Prime Numbers, and a few other topics. All the necessary theorems, their proofs, and relevant problems and their solutions will be provided. For Competitive Programming, it is not absolutely necessary to learn how to prove something super rigorously, so you can just casually go through the proofs. The most important part is to actually understand the theorems, and also build a good intuition. However, if you choose to understand and learn the proofs in a rigorous way, it will be super beneficial for you in future.

Number Theory, vaguely speaking, is the study of integers. Whenever you have to deal with integer numbers while solving a competitive programming problem, there is a chance that you have to use Number Theory there, especially if you are working with prime numbers, divisors, and sometimes counting!

Let's look at a few problems where we can apply Number Theory. We will discuss the problems in detail in the next few chapters, but for now, we will just look at the problems and think about them. The whole point of this is to find some underlying motivation behind learning number theory.

**Problem 1.1: Find Divisors**

Given an integer $N$, find the number of divisors of $N$.

**Solution 1.1: Divisibility? Primes?**

It is possible to find the number of divisors of a number $N$ by looping through all the numbers that are less than $N$. However, with some knowledge of divisibility and prime numbers, you can solve this problem way more efficiently!

**Problem 1.2: Find GCD and LCM**

Given two integers $A$ and $B$, find the GCD[2] and LCM[3] of $A$ and $B$.

[1] Wikipedia - Number Theory
[2] GCD - Greatest Common Divisor
[3] LCM - Least Common Multiple

**Solution 1.2: Similar Issue?**

It is possible to find the GCD or LCM of two numbers just by mindlessly looping through all possible candidates, but we have limited time, so we must find a solution that runs faster. Here you can efficiently solve this using some ideas from Modular Arithmetic and divisibility.

You can see an occuring pattern here. The problems we are trying to solve aren't necessarily that hard, but it is just time consuming. In competitive programming contests, the problems have time limites, so we have to find better solutions. You will often find yourself in situations where your number theory knowledge will help you optimize the time complexity. We will try to go through all the fundamental ideas of number theory that we use in competitive programming.

From the next chapter, we will study each of the topics in detail, see problems and their solutions. It is recommended that you go through the note sequentially.

# 2 Integer Numbers and Divisibility

In this chapter, we will study integers and divisibility. After that, we will see how we can use our theoretial knowledge to solve some problems.

## 2.1 Integers

**Definition 2.1: Integers**

The set of integers $\mathbb{Z}$ is the set of all whole numbers, both positive and negative, including zero. It can be represented as:

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$$

There are some interesting properties of integers. Let's look at them.

**Properties: Integers($\mathbb{Z}$)**

**Closure:** The sum, difference, and product of any two integers are also integers. For example, if $a$ and $b$ are integers, then $a + b$, $a - b$, and $a \cdot b$ are also integers.

**Associativity:** The sum and product of integers are associative. This means that for any integers $a$, $b$, and $c$, we have:

$$(a + b) + c = a + (b + c)$$
$$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$

**Commutativity:** The sum and product of integers are commutative. This means that for any integers $a$ and $b$, we have:

$$a + b = b + a$$
$$a \cdot b = b \cdot a$$

**Distributivity:** The product of an integer with a sum of integers is equal to the sum of the products. This means that for any integers $a$, $b$, and $c$, we have:

$$a \cdot (b + c) = a \cdot b + a \cdot c$$

**Identity Elements:** The identity element for addition is 0, and the identity element for multiplication is 1. This means that for any integer $a$, we have:

$$a + 0 = a$$
$$a \cdot 1 = a$$

**Inverse Elements:** For every integer $a$, there exists an integer $-a$ such that $a+(-a) = 0$. For multiplication, the inverse element is not always an integer. For example, the multiplicative inverse of 2 is $\frac{1}{2}$, which is not an integer.

In programming languages, integers are usually represented as a fixed-size data type. For example, in c++, there are multiple data types that can be used to represent an integer number. Here is a small list:

```
int a; // 4 bytes
long long b; // 8 bytes
short c; // 2 bytes
long d; // 4 bytes
unsigned int e; // 4 bytes
unsigned long long f; // 8 bytes
unsigned short g; // 2 bytes
unsigned long h; // 4 bytes
int8_t i; // 1 byte
int16_t j; // 2 bytes
int32_t k; // 4 bytes
```

In Python, you don't really have to think about data types. in C++, you will often find yourself working mostly with **int** and **long long**. Generally, if you need to work with numbers in the range of $[-2 \cdot 10^9, 2 \cdot 10^9]$, you should use **int**, and if you need bigger numbers, likely in the range of $[-2 \cdot 10^{18}, 2 \cdot 10^{18}]$, you should use **long long**. It is important to note that division by 0 is not defined, so if you are getting runtime errors, you might want to check if you have mistakenly divided something by 0.

**Problem 2.1: calculate $b^n$**

You are given two integers $b$ and $n$. You need to calculate $b^n$. The value of $b$ and $n$ can be very large, but don't worry about integer overflow. **You are not allowed to use any existing functions like pow()**

**Solution 2.1**

The easiest way to calculate this is to run a for loop from 1 to $n$ and multiply $b$ with the answer.

```
    int ans = 1;
    for(int i = 1; i <= n;i++){
      ans *= b;
    }
```

The complexity will be $\mathcal{O}(n)$. However, $n$ can be very large, so this is not the best way to calculate this by a normal linear loop. We can optimize our solution. Let's say $n = 100$, and instead of running the loop until 100, we can run the loop till 50, which will give us $b^{50}$. We need $b^{100}$, which we can find by doing $b^{50} * b^{50}$. Similarly, we don't need to run a loop from 1 to 50 either, because we can calculate $b^{25}$ and then do $b^{25} * b^{25}$.

Now, let's define a function $f$ by $f(b,n) = b^n$. Now, let's see if we can do this:

$$f\Big(b,n\Big) = f\Big(b, \Big\lfloor \frac{n}{2} \Big\rfloor\Big) \cdot f\Big(b, \Big\lfloor \frac{n}{2} \Big\rfloor\Big) \text{ if } n \text{ is even}$$
$$f\Big(b,n\Big) = b \cdot f\Big(b, \Big\lfloor \frac{n}{2} \Big\rfloor\Big) \cdot f\Big(b, \Big\lfloor \frac{n}{2} \Big\rfloor\Big) \text{ if } n \text{ is odd}$$

For example, if $n = 100$, $f(b,100) = f(b,50) \cdot f(b,50)$.
On the other hand, if $n = 99$, $f(b,99) = b \cdot f(b,49) \cdot f(b,49)$.

Now, what we have is a recursive function. Whenever we're dealing with a recursive function, we should think about its base cases. In this situation, the base case would be $f(b,0) = 1$ because $b^0 = 1$ for all $b \in \mathbb{N}$.
Solution:

```
int f(int b, int n){
  if(n == 0)return 1;
  int x = f(b, n/2);
  x = (x * x); // multiplying x by itself to do f(b, n/2) * f(b, n/2)
  if(n % 2 == 1)x = (x * b); // if n is odd, we need to multiply by b
  return x;
}
```

Since we are recursively calling the function and every time we're dividing $n$ by 2, the overall complexity will be $\mathcal{O}(\log_2(n))$, which is significantly faster than $\mathcal{O}(n)$. It is also possible to write this function using a single **while loop**. you will find the implementation along with detailed explanation and applications here.

## 2.2 Divisibility

**Definition 2.2: Divisibility**

Let $a$ and $b$ be integers and $b \neq 0$. We say that $a$ is divisible by $b$ if there exists an integer $k$ such that $a = b \cdot k$. In this case, we also say that $b$ divides $a$, and we write $b \mid a$. We can also say $a \equiv 0 \pmod{b}$ which basically means if we divide $a$ by $b$, the remainder will be 0.

Let's look at a few properties of divisibility.

**Properties: Divisibility**

Let $a$, $b$, and $c$ be integers. Then:

(i) If $b \mid a$ and $c \mid a$, then $bc \mid a$.

(ii) If $b \mid a$ and $c \mid b$, then $c \mid a$.

(iii) If $b \mid a$ and $c \mid b$, then $bc \mid ab$.

(iv) If $c \mid a$ and $c \mid b$, then $c \mid (a + b)$ and $c \mid (a - b)$.

The above properties are very easy to prove. Intuitively, we can see that they are true. The proof directly follows from Definition 2.2. We are often interested in the divisors of a number. In order to find all divisors, we can just run a for loop from 1 to $n$ and check if that number divides $n$. If yes, we push that to our vector.

```
vector<int> find_divisors(int n){
  vector<int> divisors;
  for(int i = 1; i <= n;i++){
    if(n % i == 0){
      divisors.push_back(i);
    }
  }
  return divisors;
}
```

Since we are looping from 1 to $n$, the time complexity of this code will be $\mathcal{O}(n)$.
However, it is possible to find all divisors in $\mathcal{O}(\sqrt{n})$ time.

**Theorem 2.1**

Let $n$ be a positive integer and $d \mid n$ and $d \geq \sqrt{n}$. Then $\frac{n}{d} \leq \sqrt{n}$.

*Proof.* Let $k = \frac{n}{d}$. Then, assume $k > \sqrt{n}$. Now,

$$\begin{aligned} d &\geq \sqrt{n} \\ d * k &> \sqrt{n} * \sqrt{n} \\ n &> n \end{aligned}$$

This is a contradiction. So, we can conclude that $k \leq \sqrt{n}$. □

From Theorem 2.1, we can conclude that if $d \mid n$ and $d \geq \sqrt{n}$, then $k = \frac{n}{d} \leq \sqrt{n}$. This means that if we find a divisor $d$ of $n$ such that $d \leq \sqrt{n}$, then we can also find another divisor $k$ such that $k > \sqrt{n}$. So, we can just loop from 1 to $\sqrt{n}$ and check if that number divides $n$. If yes, we push both the numbers to our vector.

```
vector<int> find_divisors(int n){
  vector<int> divisors;
  for(int i = 1; i * i <= n;i++){
    if(n % i == 0){
      divisors.push_back(i);
      if(i != n / i){
        // If n is a perfect square, i and n/i are same numbers.
        // So we need to check that before pushing n/i
        divisors.push_back(n / i);
      }
    }
  }
  return divisors;
}
```

Here, we are running the loop from 1 to $\sqrt{n}$, so the time complexity will be $\mathcal{O}(\sqrt{n})$.

Now, let's say we have to find divisors for all numbers upto $n$. We can loop through from 1 to $n$ and use our **find_divisors**.

```
vector<int> divisors[n + 1];
for(int i = 1; i <= n; i++){
  divisors[i] = find_divisors(i);
}
```

This will take $\mathcal{O}(n\sqrt{n})$ time. But we can do better. Look at the following code:

```
vector<int> divisors[n + 1];
for(int i = 1; i <= n; i++){
  for(int j = i; j <= n; j += i){
    divisors[j].push_back(i);
  }
}
```

Here, we are looping through from 1 to $n$ and for each number $i$, we are pushing $i$ to all multiples of $i$ that are less than or equal to $n$. The number of multiples of $i$ will be $\frac{n}{i}$. So, for each $i$, the inner-loop is running $\frac{n}{i}$ times. So, in total, it will run $n\left(1 + \frac{1}{2} + \cdots + \frac{1}{n}\right)$ times.

Now, let $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$. $H_n$ is also known as $n$-th harmonic number. It can be proven[4] that $H_n \leq \ln n + 1$. So, the complexity of our code becomes $\mathcal{O}(n \ln n)$.

We will discuss more about divisibility and solve related problems later. First let's study a few more things.

## 2.3 Prime numbers (Overview)

**Definition 2.3: Prime Numbers**

A prime number $p$ is a natural number greater than 1 that has exactly 2 divisors: 1 and $p$ itself.

The first few prime numbers are $2, 3, 5, 7, 11, 13, 17, 19, 23, 29, \ldots$. The number 1 is not considered a prime number because it has only one divisor: itself. All the numbers that are not primes are known as **composite numbers**. There are lots of interesting things we can do with primes. Let's try to solve the following problem:

**Problem 2.2: Find number of divisors of $n$**

Given an integer $n$, find the number of divisors of $n$. Seems pretty simple. We have already solved this problem in the previous section. You can already find all divisors from 1 to $n$ in $\mathcal{O}(n \log(n))$ complexity. However, $n \leq 10^9$.

[4] The partial sums of harmonic series

**Solution 2.2**

We cannot loop through all the numbers from 1 to $n$ as it will be too slow. So, the previous solution where we looped through all the numbers and updated their multiples - simply doesn't work. We can use some ideas from prime-factorization to potentially solve this problem in significantly faster than $\mathcal{O}(\sqrt{n})$.

**Theorem 2.2**

for any integer $n > 1$, there exists some $k \geq 1$ such that:

$$n = p_1^{a_1} \cdot p_2^{a_2} \cdots p_k^{a_k}$$

where $p_i$ are distinct prime numbers and $a_i$ are positive integers. We call it the prime power factorization of $n$. This prime power factorization is also unique, or in other words, there is only **one way** to prime factorize a number.

*Proof.* See Here. □

For example, let's say we have $n = 60$. The prime factorization of 60 is: $2^2 \cdot 3^1 \cdot 5^1$, and that's the only way to prime-factorize 60 (upto ordering).

In competitive programming, the proof is not that important, but please check it out if you want to. We will often apply Theorem 2.2 in many different situations.

Now, let's get back to Solution 2.2. Let's assume we can somehow prime-factorize $n$ roughly in $\mathcal{O}(\sqrt{n})$. In other words, we have figured out $n = p_1^{a_1} \cdot p_2^{a_2} \dots p_k^{a_k}$ for some $k > 0$. Now, how can we generate all divisors of $n$ from its prime factorization? Let's look at the following example:

$$\begin{aligned} 60 &= 2^2 \cdot 3^1 \cdot 5^1 \\ &= 1 * 60 \\ &= 2 * 30 \\ &= 3 * 20 \\ &= 4 * 15 \\ &= 5 * 12 \\ &= 6 * 10 \end{aligned}$$

Let's prime factorize all its divisors:

$$\begin{aligned} 1 &= 2^0 \cdot 3^0 \cdot 5^0 \\ 2 &= 2^1 \cdot 3^0 \cdot 5^0 \\ 3 &= 2^0 \cdot 3^1 \cdot 5^0 \\ 4 &= 2^2 \cdot 3^0 \cdot 5^0 \\ 5 &= 2^0 \cdot 3^0 \cdot 5^1 \\ 6 &= 2^1 \cdot 3^1 \cdot 5^0 \end{aligned}$$

$$\begin{aligned}
10 &= 2^1 \cdot 3^0 \cdot 5^1\\
12 &= 2^2 \cdot 3^1 \cdot 5^0\\
15 &= 2^0 \cdot 3^1 \cdot 5^1\\
20 &= 2^2 \cdot 3^0 \cdot 5^1\\
30 &= 2^1 \cdot 3^1 \cdot 5^1\\
60 &= 2^2 \cdot 3^1 \cdot 5^1
\end{aligned}$$

Look at their prime factorization. If you just take all possible combinations of the available primes, you will get all the divisors of $n$. In our example, $60 = 2^2 \cdot 3^1 \cdot 5^1$. Or in other words, we have $2, 2, 3, 5$, these 4 numbers (not necessarily unique), and we can either take it or not. Look at the following box:

$$\boxed{(2,2) \mid (3,1) \mid (5,1)}$$

From the first box, we can take 0 or 1 or 2 of the first number. From the second box, we can take 0 or 1 of the second number. From the third box, we can take 0 or 1 of the third number. So, we can take all possible combinations of these numbers. The total number of divisors will be: $(2+1)\cdot(1+1)\cdot(1+1) = 12$.

So, generally, if $n = p_1^{a_1} \cdot p_2^{a_2} \cdots p_k^{a_k}$, then the number of divisors of $n$ will be: $(a_1+1)\cdot(a_2+1)\dots(a_k+1) = \prod_{i=1}^{k}(a_i+1)$. Now, you can argue that we still have to loop through 1 to $k$, but the number of different prime factors of a number is pretty low. In fact, it is bounded by $\frac{\log n}{\log \log n}$. Realistically, a number that is less than $2 \cdot 10^9$ can have at most 9 different prime factors.

It was a very introductory section on prime numbers. We will study more about it in the next chapter.

## 2.4 GCD and LCM

**Definition 2.4: GCD**

Let $a$ and $b$ two integers. A non-negative integer $g$ is called a **greatest common divisor**(GCD) of $a$ and $b$ if:

- $g$ divides both $a$ and $b$.
- If $d$ is any other common divisor of $a$ and $b$, then $d$ divides $g$.

We denote the GCD of $a$ and $b$ by $\gcd(a,b)$ or just $(a,b)$.

For example, consider 24 and 36. The common divisors of 24 and 36 are 1, 2, 3, 4, 6, 12 and the GCD is $\gcd(24,36) = 12$. **Note that all of their common divisors divide 12**. When we are working with integer numbers in particular, GCD of two numbers is always unique. GCD can be defined in a way such that it might not be unique, but that is not our concern as we're working with integers only. You can read more about it Here[5].

[5] GENERALIZATIONS OF GREATEST COMMON DIVISORS OF GCD DOMAINS

**Definition 2.5: LCM**

Let $a$ and $b$ be two integers. A non-negative integer $l$ is called a **least common multiple**(LCM) if it is the smallest integer that is divisible by both $a$ and $b$.

For example, consider 24 and 36. The common multiples of 24 and 36 are 72, 108, 144, 180, 216, 252, etc and the LCM is $\text{lcm}(24, 36) = 72$. **Note that all of their common multiples are multiples of 72**. When we are working with integer numbers in particular, LCM of two numbers is always unique.

**Properties: GCD and LCM**

1. $\gcd(a, b) = \gcd(b, a)$
2. $\gcd(a, b) = \gcd(a, b - a)$
3. $\gcd(a, 0) = |a|$
4. $\gcd(a, b) = 1$ if and only if $a$ and $b$ are coprime.
5. $\text{lcm}(a, b) = \text{lcm}(b, a)$
6. $\text{lcm}(a, 0) = 0$
7. $\gcd(a, b) \cdot \text{lcm}(a, b) = |a| \cdot |b|$
8. $\gcd(a, b) \leq \sqrt{a \cdot b} \leq \text{lcm}(a, b)$
9. $\gcd(a, b) = \gcd(b, a \mod b)$

*Proof.* (1) Let $g = \gcd(a, b)$. Then, $g$ divides both $a$ and $b$. So, $g$ divides $b$ and $a$. Hence, $\gcd(b, a) = g$. □

*Proof.* (2) Let $g = \gcd(a, b)$. Then, $g$ divides both $a$ and $b$. So, $g$ divides $b - a$ and $a$. Hence, $\gcd(a, b - a) = g$. □

*Proof.* (3) Let $g = \gcd(a, 0)$. Then, $g$ divides both $a$ and 0. So, $g$ divides $a$ and 0. Now, $g$ has to be less than or equal to $|a|$, otherwise $g \nmid a$. Hence, $\gcd(a, 0) = |a|$. □

*Proof.* (4) Two integers $a$ and $b$ are coprime if they have no common divisors other than 1. So, if $\gcd(a, b) = 1$, $a$ and $b$ must be coprime. □

*Proof.* (5) Let $l = \text{lcm}(a, b)$. Then, $l$ is the smallest integer that is divisible by both $a$ and $b$. So, $l$ is divisible by both $b$ and $a$. Hence, $\text{lcm}(b, a) = l$. □

*Proof.* (6) Divisibility by 0 is not defined. But, we can consider 0 as a multiple of 0. So, $\text{lcm}(a, b) = 0$. □

*Proof.* (7)
This is one of the most interesting properties of gcd-lcm. In order to understand why it works, let's

first prime-factorize $a$ and $b$.
Let $a = p_1^{x_1} \cdot p_2^{x_2} \cdots p_k^{x_k}$ and $b = p_1^{y_1} \cdot p_2^{y_2} \cdots p_k^{y_k}$, where $p_i$ are the prime factors of $a$ and $b$. Then, let $g = \gcd(a, b)$ and $l = \operatorname{lcm}(a, b)$. The prime factorization of $g$ will be: $p_1^{z_1} \cdot p_2^{z_2} \cdots p_k^{x_k}$. Now, $g$ is the greatest common divisor, so we want the powers to be as large as possible, and at the same time, make sure it still divides both $a$ and $b$. Let's assume $x_1 \leq y_1$. In that case, if we take $p_1^{x_1}$ as a candidate, it divides both $a$ and $b$. However, if we take anything more than that, it won't divide $a$ anymore. so, $z_1 = \min(x_1, y_1)$, similarly, $z_i = \min(x_1, y_1)$ for all $1 \leq i \leq k$.

Similarly, for LCM, you have to take the maximum power, otherwise it won't be divisible by both $a$ and $b$. So,

$$\begin{aligned} a &= p_1^{x_1} \cdot p_2^{x_2} \cdots p_k^{x_k} \\ b &= p_1^{y_1} \cdot p_2^{y_2} \cdots p_k^{y_k} \\ g &= p_1^{\min(x_1,y_1)} \cdot p_2^{\min(x_2,y_2)} \cdots p_k^{\min(x_k,y_k)} \\ l &= p_1^{\max(x_1,y_1)} \cdot p_2^{\max(x_2,y_2)} \cdots p_k^{\max(x_k,y_k)} \end{aligned}$$

Now, multiply $a$ and $b$ and separately multiply $g$ and $l$, we get:

$$\begin{aligned} a \cdot b &= p_1^{x_1} \cdot p_2^{x_2} \cdots p_k^{x_k} \cdot p_1^{y_1} \cdot p_2^{y_2} \cdots p_k^{y_k} \\ &= p_1^{x_1+y_1} \cdot p_2^{x_2+y_2} \cdots p_k^{x_k+y_k} \\ g \cdot l &= p_1^{\min(x_1,y_1)} \cdot p_2^{\min(x_2,y_2)} \cdots p_k^{\min(x_k,y_k)} \cdot p_1^{\max(x_1,y_1)} \cdot p_2^{\max(x_2,y_2)} \cdots p_k^{\max(x_k,y_k)} \\ &= p_1^{\min(x_1,y_1)+\max(x_1,y_1)} \cdots p_k^{\min(x_k,y_k)+\max(x_k,y_k)} \\ &= p_1^{x_1+y_1} \cdots p_k^{x_k+y_k} \\ &= a \cdot b \end{aligned}$$

For example, let's say $a = 12$ and $b = 18$. $\gcd(a, b) = 6$ and $\operatorname{lcm}(a, b) = 36$. Now, $a \cdot b = 12 \cdot 18 = 216$ and $\gcd(a, b) \cdot \operatorname{lcm}(a, b) = 6 \cdot 36 = 216$. So, we can conclude that $\gcd(a, b) \cdot \operatorname{lcm}(a, b) = |a| \cdot |b|$. □

*Proof.* (8) Without loss of generality, let $a \leq b$. Then,

$$\begin{aligned} \gcd(a, b) &\leq a \\ \gcd(a, b)^2 &\leq a^2 \\ \gcd(a, b)^2 &\leq a \cdot b \\ \gcd(a, b) &\leq \sqrt{a \cdot b} \quad \cdots (1) \\ \operatorname{lcm}(a, b) &\geq b \\ \operatorname{lcm}(a, b)^2 &\geq b^2 \\ \operatorname{lcm}(a, b)^2 &\geq a \cdot b \\ \operatorname{lcm}(a, b) &\geq \sqrt{a \cdot b} \quad \cdots (2) \end{aligned}$$

Combining (1) and (2), we get:

$$\gcd(a, b) \leq \sqrt{a \cdot b} \leq \operatorname{lcm}(a, b)$$

□

*Proof.* (9) Let $g = \gcd(a, b)$. Then, $a = k_1 \cdot g$ and $b = k_2 \cdot g$. Let $m = a \mod b$. In that case, $a = k_3 \cdot b + m$. $g \mid a$, so, $g \mid (k_3 \cdot b + m)$. But $g \mid b$, so, $g \mid m$. Proven. □

You will find these properties very useful in competitive programming. Now, let's look at a problem.

**Problem 2.3: GCD and LCM**

Given two integers $a$ and $b$, find their GCD and LCM.

**Solution 2.3: GCD and LCM**

**GCD:**
The trivial solution is to loop through 1 to $\min(a, b)$ and check if that number divides both of them. Time complexith $\mathcal{O}(\min(a, b))$.

Now, we can prime-factorize both $a$ and $b$ and then take the minimum power for all primes, and then multiply them. As per our assumption, prime factorization takes $\mathcal{O}(\sqrt{a} + \sqrt{b})$ time, and that's our time complexity.
However, we can do better. We have already proven that $\gcd(a, b) = \gcd(b, a \mod b)$. Based on that, we can write the following recursive function to find GCD:

```
int gcd(int a, int b){
  if(b == 0){
    return a;
  }
  return gcd(b, a % b);
}
```

For example, let's simulate $\gcd(24, 36)$:

$$\begin{aligned}
\gcd(24, 36) &= \gcd(36, 24 \mod 36) \\
&= \gcd(36, 24) \\
&= \gcd(24, 36 \mod 24) \\
&= \gcd(24, 12) \\
&= \gcd(12, 24 \mod 12) \\
&= \gcd(12, 0) \\
&= 12
\end{aligned}$$

Now, let's think about the complexity of this code. For simplicity, let's assume $a \geq b$. Now, we start with $\gcd(a, b)$. From that, we go to $\gcd(b, a \mod b)$. The value of $a \bmod b$ is always less than $b$. What would be the value of $a \mod b$ in the worst case scenario? Well, maybe the bigger the better, right? So, let's assume $a \mod b = b - 1$. But in that case, our next call is $\gcd(b, b - 1)$, and from that we'll go to $\gcd(b - 1, 1)$, which will end very fast. On the other hand, if the mod value is too small, it will again end pretty fast. So, intuitively, you can figure out that the worst case scenario happens when the mod value is somewhere around $\frac{b}{2}$. So, from $\gcd(a, b)$, we are going to somewhere around $\gcd(b, \frac{b}{2})$. And from that, we go somewhere around $\gcd(\frac{b}{2}, \frac{b}{4})$. We are basically approaching 0 at a logarithmic rate, so the complexity becomes $\mathcal{O}(\log \min(a, b))$.

**LCM:**
Now that we can find $\gcd(a, b)$ in $\mathcal{O}(\log \min(a, b))$ time, we can use the property $\gcd(a, b) \cdot \operatorname{lcm}(a, b) = |a| \cdot |b|$ to find LCM. So, we can write:

```cpp
int lcm(int a, int b){
  // instead of doing (a * b) / gcd(a,b), we re doing the following:
  return (a / gcd(a, b)) * b;
  // the reason we are doing this is to avoid overflow.
  // sometimes LCM itself fits into long long, but a*b itself doesn't.
}
```

Time complexity of this code is $\mathcal{O}(\log \min(a, b))$ as well.

## 2.5 Modular Arithmetic

In modular arithmetic, instead of directly working with integers, we work with the remainder of integers when divided by some integer $m$. for example, if $m = 5$, and some other integer $a = 17$, then dividing $a$ by $m$ leaves us with a remainder of 2. we write this as $17 \equiv 2 \pmod 5$. This is same as saying:

$$a = m \cdot k + r$$

where $k$ is some integer and $r$ is the remainder. In this case, $r = 2$ and $k = 3$. Generally, we can say that $a \equiv r \pmod m$ where $r$ represents the remainder.

**Properties: Modular Arithmetic**

1. $(a + b) \pmod m \equiv (a \mod m + b \mod m) \mod m$
2. $(a - b) \pmod m \equiv (a \mod m - b \mod m) \mod m$
3. $(a \cdot b) \pmod m \equiv (a \mod m \cdot b \mod m) \mod m$
4. $a^b \pmod m \equiv (a \mod m)^b \mod m$

*Proof.* (1) Let $a \equiv a_1 \pmod m$ and $b \equiv b_1 \pmod m$. Now, we can write:

$$\begin{aligned}
a &= m \cdot k_1 + a_1 \\
b &= m \cdot k_2 + b_1 \\
a + b &= m \cdot k_1 + a_1 + m \cdot k_2 + b_1 \\
&= m \cdot (k_1 + k_2) + (a_1 + b_1) \\
\implies a + b &\equiv (a_1 + b_1) \mod m
\end{aligned}$$

□

*Proof.* (2) Directly follows from Proof (1) by replacing $b$ with $-b$. □

*Proof.* (3) Let $a \equiv a_1 \pmod{m}$ and $b \equiv b_1 \pmod{m}$. Now, we can write:

$$\begin{aligned} a &= m \cdot k_1 + a_1 \\ b &= m \cdot k_2 + b_1 \\ a \cdot b &= (m \cdot k_1 + a_1) \cdot (m \cdot k_2 + b_1) \\ &= m^2 \cdot (k_1 \cdot k_2) + m \cdot (k_1 \cdot b_1 + k_2 \cdot a_1) + (a_1 \cdot b_1) \\ \implies a \cdot b &\equiv (a_1 \cdot b_1) \mod m \end{aligned}$$

□

*Proof.* (4) Directly follows from Proof (3). Hint: Use the fact that $a^b = a \cdot a^{b-1}$. Then you can use Induction[6] to prove this. □

**Problem 2.4:** $a - b$

You are given two integers $a$ and $b$. You need to calculate $a - b$ modulo $10^9 + 7$.

**Solution 2.4**

The solution should be very simple. Just do:

```
  const int mod = 1e9 + 7;
  ans = (a - b) % mod;
```

However, you might run into some issues with negative numbers. In competitive programming scenario, when you are asked to find some expression modulo some $m$, you are basically asked to find out the smallest non-negative integer $r$ such that the expression is equivalent to $r$ modulo $m$. So, instead of just subtracting and taking the modulo, do this:

```
const int mod = 1e9 + 7;
ans = (a - b) % mod;
if(ans < 0) ans += mod;
```

**Problem 2.5: Big Mod**

You are given two integers $a$ and $b$, where $1 \leq a, b \leq 10^9$. You need to calculate $a^b$ modulo $10^9 + 7$.

**Solution 2.5**

We already know how to solve $a^b$ in $\mathcal{O}(\log b)$ time in Problem 2.1. Now that we know modulo distributes over addition and multiplication, we can use the exact same solution, but every step, we will keep the modulo value, and that will never exceed $10^9 + 7$, which means even if we multiply two numbers $x = x * x$, it will still fit within **long long**.

```
const int mod = 1e9 + 7;
```

[6] Wikipedia - Mathematical Induction

```
long long bigmod(long long b, long long e){
  if(e == 0) return 1;
  long long x = bigmod(b, e/2);
  x = (x * x) % mod;
  if(e % 2 == 1){
    x = (x * b) % mod;
  }
  return x;
}
```

**Problem 2.6: Factorial Division**

You are given $1 \le q \le 10^5$ queries. in each query, you are given two integers $a$ and $b$ where $1 \le b \le a \le 10^6$. You need to calculate $\frac{a!}{b!}$ modulo $10^9 + 7$.

**Solution 2.6**

It is easy to see that $b!$ always divides $a!$ as $a \ge b$. now, $\frac{a!}{b!} = a \cdot (a-1) \cdots (b+1)$. However, you cannot loop through it and keep multiplying the numbers and modding it by $10^9 + 7$ because there are too many queries. The time complexity will be $\mathcal{O}(q \cdot (a-b))$ which is too slow. we can precalculate the factorials modulo $10^9 + 7$ and store them in an array.

```
    const int mod = 1e9 + 7;
    const int maxn = 1e6 + 5;
    long long fact[maxn];
    fact[0] = fact[1] = 1;
    for(int i = 2; i < maxn; i++){
      fact[i] = (fact[i-1] * i) % mod;
    }
```

Then, for each query, we have **fact**[**a**] and **fact**[**b**] already calculated. However, they are calculated modulo some prime number $m$. But in the properties of modular arithmetic, we didn't add anything about division. Distribution over division doesn't always work. For example,

$$\frac{54}{18} \pmod 5 = 3$$
$$\frac{54 \pmod 5}{18 \pmod 5} = \frac{4}{3}$$

So, we need to do something else. We can use the fact that $m$ is prime. So, we can use Fermat's Little Theorem[a] to calculate the inverse of $b!$ modulo $m$. It states that if $p$ is a prime number and $a$ is an integer not divisible by $p$, then:

$$a^{p-1} \equiv 1 \pmod p$$
$$a^{p-2} \equiv a^{-1} \pmod p$$

So, $\frac{a!}{b!} \equiv a! \cdot (b!)^{p-2} \pmod p$. However, in this scenario, $p = 10^9 + 7$, so we cannot find the power using a loop. But we've already studied Bigmod 2.5. So, the final solution (code) will be:

```
    // fact[] is already calculated
    while(q--){
      long long a, b;
      cin >> a >> b;
      long long ans = (fact[a] * bigmod(fact[b], mod-2)) % mod;
    }
```

[a]Wikipedia - Fermat's Little Theorem

## 2.6 Problem Solving

We will use all the knowledge we have gained so far to solve some problems.

**Problem 2.7: 1370A - Maximum GCD [a]**

[a]https://codeforces.com/problemset/problem/1370/A

You are given an integer $2 \leq n \leq 10^6$. you have to find

$$\max_{1 \leq i < j \leq n} \left\{ \gcd(i, j) \right.$$

**Solution 2.7**

Looping through all possible $i$ and $j$ is $\mathcal{O}(n^2)$ which is not good enough. You can find one observation that if $i < j$, then $\gcd(i, j)$ will be at most $i$, and $j$ has to be at least $2i$. Also, if $i$ is the gcd, then $i \mid j$. Now, my claim is that the maximum gcd will be $\lfloor \frac{n}{2} \rfloor$. Let's assume that $i > \lfloor \frac{n}{2} \rfloor$. Then $j$ has to be at least $2i > n$, which is a contradiction. So, the maximum gcd will be $\lfloor \frac{n}{2} \rfloor$.

**Problem 2.8: 1968A - Maximize?[a]**

[a]https://codeforces.com/problemset/problem/1968/A

You are given an integer $x$. Your task is to find any integer $y$ such that $1 \leq y < x$ and $\gcd(x, y) + y$ is maximum possible. Note that if there is more than one $y$ which satisfies the statement, you are allowed to find any.

The problem has small constraints, only $1 \leq t \leq 1000$ test cases, and $1 \leq x \leq 1000$. However, let's try to solve it assuming $1 \leq t \leq 10^6$ and $1 \leq x \leq 10^{18}$.

**Solution 2.8**

The bruteforce solution is $\mathcal{O}(t \cdot x)$, which is not good enough.
From the properties of GCD, we know that $\gcd(x, y) = \gcd(x - y, y)$. From that,

$$\begin{aligned} \gcd(x, y) = \gcd(x - y, y) &\leq x - y \\ \gcd(x, y) + y &\leq x \end{aligned}$$

So, it is guaranteed, that the final value cannot be greater than $x$. Now we just have to find a $y$ such that $\gcd(x,y) + y = x$. Take $y = x - 1$. Then, $\gcd(x,y) = \gcd(x, x-1) = 1$. So, $\gcd(x,y) + y = 1 + (x-1) = x$. So, the answer is $x - 1$.

**Problem 2.9: 1617B - GCD Problem**[a]

[a]https://codeforces.com/problemset/problem/1617/B

Given a positive integer $n$. Find three distinct positive integers $a, b, c$ such that $a + b + c = n$ and $\gcd(a,b) = c$, where $\gcd(x,y)$ denotes the greatest common divisor (GCD) of integers $x$ and $y$.
There will be $1 \le t \le 10^5$ test cases, and $10 \le n \le 10^9$.

**Solution 2.9**

The bruteforce solution is $\mathcal{O}(t \cdot n^2)$, which is not good enough.
$\gcd(a,b) = c$ means that $c$ divides both $a$ and $b$. So, basically we have to find a $c$ such that for some $k_1, k_2$, $ak_1 + bk_2 + c = n$ and $\gcd(k_1, k_2) = 1$.

In these types of problems, it might seem a bit harder to approach mathematically. Look at the statement carefully. We have to find three distinct positive numbers that follow two conditions. It is enough to find one valid solution. So, what we can do is assume $c = 1$ and see if we can find suitable $a$ and $b$. In that case $\gcd(a,b) = 1$ and $a + b = n - 1$.

**Proposition**: If we choose a prime number $p$ as $a$ such that $a \nmid (n-1)$ and take $b = n - 1 - a$, all the conditions will satisfy.

*Proof.*

$$a + b + c = p + (n - 1 - p) + 1 = n \tag{1}$$

$$\begin{aligned} \gcd(a,b) &= \gcd(p, n-1-p) \\ &= \gcd(p, n-1) \\ &= 1 = c \end{aligned} \tag{2}$$

So, from (1) and (2), we can see that all the conditions are satisfied.

□

Now, another observation is, we don't have to look too far to find such $p$ that doesn't divide $n-1$. You can do some calculations: $2 \times 3 \times 5 \times \ldots \times 29 > 10^9$.So, among the first 10 primes, there must be at least 1 prime that doesn't divide $n - 1$. So, we can just loop through the first 10 primes and find the suitable one. Also note that $n \ge 10$, so $n - 1 \ge 9$. So, we can surely assume that we will find some candidates. Complexity: $\mathcal{O}(t)$.

**Note**: It might feel a bit weird to assume $c = 1$. If we didn't make that assumption, finding a suitable solution might be a bit harder. These kinds of assumptions comes with practice. You'll gradually recognize patterns that'll lead you to a correct solution faster. However, it is very important to prove your assumption. If your assumption is wrong, you can move to another logical assumption. Sometimes you won't have enough time to rigorously prove your assumptions. There will always be situations like that. In that case, you can just write the code, submit, and "Prove" it by AC. But most of the cases, spending a few minutes to prove your assumptions is worth it. Overtime, you will be good at intuitions, and you will be able to find relatively correct assumptions faster.

**Problem 2.10: 1312C - Adding Powers** [a]

[a] https://codeforces.com/problemset/problem/1312/C

Suppose you are performing the following algorithm. There is an array $v_1, v_2, \ldots, v_n$ filled with zeroes at the start. The following operation is applied to the array several times — at the $i$-th step (0-indexed) you can:

- either choose a position pos ($1 \leq \text{pos} \leq n$) and increase $v_{\text{pos}}$ by $k^i$;
- or not choose any position and skip this step.

You can choose how the algorithm would behave on each step and when to stop it. The question is: can you make the array $v$ equal to the given array $a$ (i.e., $v_j = a_j$ for each $j$) after some step? Here $1 \leq n \leq 30$, $2 \leq k \leq 100$ and $0 \leq a_i \leq 10^{16}$.

**Solution 2.10**

**Observation**: for all $i$, $a_i$ has to be in the following form:

$$a_i = k^{x_1} + k^{x_2} + \ldots + k^{x_m} \text{ for some } x_1, x_2, \ldots, x_m$$

Now, in $i$-th step, we can choose some $j$ and add $j^i$ to $a_j$. So, if there is a situation where we have to add $k^i$ to multiple elements, it is impossible because we can use $k^0, k^1, \ldots..$ etc only once. So, for all $a_i$, find all the $k^i$ we have to add. If there are multiple $k^i$ for some $i$, then it is impossible. And if $a_i$ is not in the above form, the answer is also impossible.

Time complexity $\mathcal{O}(n \cdot \log(\max_{1 \leq i \leq n} a_i))$.

```
long long a[105], p[105], inf = 1e16 + 10;
long long n, k, mxp;
map<long long, long long> M;
void calc_ai(long long x) {
  if (x == 0)
    return;
  vector<long long> v;
  for (int i = mxp; i >= 0; i--) {
    if (x >= p[i])
      v.push_back(i), x -= p[i];
  }
  if (x != 0)
    M[0] = 2;
  else
    for (auto it : v)
      M[it]++;
}
```

```cpp
int main() {
  // input and stuff
  p[0] = 1;
  for (int i = 1; i <= 60; i++) {
    p[i] = p[i - 1] * k;
    if (p[i] > inf) {
      mxp = i;
      break;
    }
  }

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

  int f = 1;
  for (auto it : M) {
    if (it.second > 1)
      f = 0;
  }

  if (f)
    cout << "YES" << endl;
  else
    cout << "NO" << endl;
}
```

# 3 Primes

In the previous chapter, we have seen some theorems and properties about prime numbers on section 2.3. We will now study deeper about prime numbers.

## 3.1 Primality Testing

**Problem 3.1: Primality Testing**

Given an integer $1 \leq n \leq 10^9$, determine whether $n$ is prime or not.

**Solution 3.1**

**naive solution**: Loop from 2 to $n-1$ and check if that number divides $n$. If yes, $n$ is not prime. 1 is not prime, so handle that separately. Complexity $\mathcal{O}(n)$.

**Slightly Better Solution**: From Theorem 2.1, we can say that if $n$ is not prime, there must be a divisor of $n$ that is less than or equal to $\sqrt{n}$. So, looping till $\sqrt{n}$ is enough. Time complexity will be $\mathcal{O}(\sqrt{n})$.

**Observation**: It is enough to just check the prime numbers till $\sqrt{n}$. The number of prime numbres till $x$ is $x/\log(x)$[a]. So, if we somehow had the list of all prime numbers, we could potentially loop through the primes upto $\sqrt{n}$ and check if any of them divides $n$. The complexity of checking would be $\mathcal{O}(\sqrt{n}/\log(\sqrt{n}))$. So, given that we magically have the list of all primes upto $\sqrt{10^9}$, we can check if $n$ is prime in $\mathcal{O}(\sqrt{n}/\log(\sqrt{n}))$, which is roughly 3052 operations.

**Even Better(?) Solution**: If you are only asked to check if a number is prime or not, you can use the Miller–Rabin [b]primality test. It is a probabilistic test that can determine if a number is a prime or not. The complexity is somewhere around $\mathcal{O}(\log^3 n)$. It is particularly useful for large numbers. Like if you have to check primality for $n$ closer to $10^{18}$, it is too slow to do it in $\mathcal{O}(\sqrt{n}/\log(\sqrt{n}))$. **note that** this is a probabilistic test, so it can give false positives, but it is rare.

[a] https://en.wikipedia.org/wiki/Prime_number_theorem
[b] https://cp--algorithms.com/algebra/primality_tests.html#miller--rabin--primality--test

**Problem 3.2: Listing all primes**

Given an integer $1 \leq n \leq 10^6$, list all prime numbers less than or equal to $n$.

**Solution 3.2**

**Naive Solution**: Loop through all numbers from 1 to $n$ and check if that number is prime or not in $\mathcal{O}(\sqrt{i})$. The full complexity will be $\mathcal{O}(n\sqrt{n})$.

```
bool is_prime(int x){
  if(x == 1)return false;
  for(int j = 2; j * j <= i;j++){
    if(x % j == 0)return false;
  }
  return true;
}
vector<int> primes;
for(int i = 1; i <= n;i++){
  if(is_prime(i)){
    primes.push_back(i);
  }
}
```

**Better Solution(?)**: We can loop through all numbers from 1 to $n$ and check if that number is prime using miller–rabin primality test. The complexity will be $\mathcal{O}(n \log^3 n)$. However, as miller–rabin is not deterministic, it is better to use deterministic algorithms if possible. Furthermore, it is possible to do it in $\mathcal{O}(n \log \log n)$ time using **Sieve of Eratosthenes**.

## 3.2 Sieve of Eratosthenes

The **Sieve of Eratosthenes** is an efficient algorithm for finding all primes less than or equal to a given number $n$. The basic idea is to eliminate the multiples of each prime number starting from 2.

We demonstrate the sieve for $n = 40$.

**Initial table:** All numbers from 2 to 40 are unmarked (white).

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | |

**Step 1: Pick 2 (first prime), mark all multiples of 2 (except 2 itself):**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 🟩 2 | 3 | 🟥 4 | 5 | 🟥 6 | 7 | 🟥 8 | 9 | 🟥 10 | 11 |
| 🟥 12 | 13 | 🟥 14 | 15 | 🟥 16 | 17 | 🟥 18 | 19 | 🟥 20 | 21 |
| 🟥 22 | 23 | 🟥 24 | 25 | 🟥 26 | 27 | 🟥 28 | 29 | 🟥 30 | 31 |
| 🟥 32 | 33 | 🟥 34 | 35 | 🟥 36 | 37 | 🟥 38 | 39 | 🟥 40 | |

**Step 2: Pick 3, mark all its multiples:**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 🟩 2 | 🟩 3 | 🟥 4 | 5 | 🟥 6 | 7 | 🟥 8 | 🟥 9 | 🟥 10 | 11 |
| 🟥 12 | 13 | 🟥 14 | 🟥 15 | 🟥 16 | 17 | 🟥 18 | 19 | 🟥 20 | 🟥 21 |
| 🟥 22 | 23 | 🟥 24 | 25 | 🟥 26 | 🟥 27 | 🟥 28 | 29 | 🟥 30 | 31 |
| 🟥 32 | 🟥 33 | 🟥 34 | 35 | 🟥 36 | 37 | 🟥 38 | 🟥 39 | 🟥 40 | |

**Step 3: Pick 5, mark its multiples:**

|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| 🟩 2 | 🟩 3 | 🟥 4 | 🟩 5 | 🟥 6 | 7 | 🟥 8 | 🟥 9 | 🟥 10 | 11 |
| 🟥 12 | 13 | 🟥 14 | 🟥 15 | 🟥 16 | 17 | 🟥 18 | 19 | 🟥 20 | 🟥 21 |
| 🟥 22 | 23 | 🟥 24 | 🟥 25 | 🟥 26 | 🟥 27 | 🟥 28 | 29 | 🟥 30 | 31 |
| 🟥 32 | 🟥 33 | 🟥 34 | 🟥 35 | 🟥 36 | 37 | 🟥 38 | 🟥 39 | 🟥 40 |  |

**Step 4: Pick 7, mark its multiples:**

|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| 🟩 2 | 🟩 3 | 🟥 4 | 🟩 5 | 🟥 6 | 🟩 7 | 🟥 8 | 🟥 9 | 🟥 10 | 11 |
| 🟥 12 | 13 | 🟥 14 | 🟥 15 | 🟥 16 | 17 | 🟥 18 | 19 | 🟥 20 | 🟥 21 |
| 🟥 22 | 23 | 🟥 24 | 🟥 25 | 🟥 26 | 🟥 27 | 🟥 28 | 29 | 🟥 30 | 31 |
| 🟥 32 | 🟥 33 | 🟥 34 | 🟥 35 | 🟥 36 | 🟩 37 | 🟥 38 | 🟥 39 | 🟥 40 |  |

After going through all primes $\leq \sqrt{40}$, we are done. All remaining unmarked numbers are primes.

**Final prime list up to 40:**

$$\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37\}$$

Now, look at the simulation carefully. We are taking a prime, and essentially cutting all the numbers that are multiples of that prime. Try to do it with $n = 100$. You'll see that for a prime $p$, all the numbers that are less than $p^2$ and also multiples of $p$ are already cut. It's because in that case, that number must have another prime divisor smaller than $p$. So, for each $p$, we can start from $p^2$ and cut all the multiples of $p$. Let's look at the code below:

```
const int N = 1e6 + 10;
bool is_composite[N];
vector<int> primes;

void sieve(int n){
  is_composite[1] = true;
  for(int i = 2; i <= n;i++){
    if(!is_composite[i]){
      primes.push_back(i);
      for(int j = i * i; j <= n;j+=i){
        is_composite[j] = true;
      }
    }
  }
}
```

The sieve function marks all composite numbers up to $n$ by iterating over each number $i$ from 2 to $n$. If $i$ is not marked as composite, it is a prime number, and all its multiples starting from $i^2$ are marked as composite.

We analyze the total number of operations performed in all inner loops. For a given prime $p$, the inner loop marks:

$$\left\lfloor \frac{n - p^2}{p} \right\rfloor + 1 \approx \frac{n}{p} - p + 1$$

Summing over all prime numbers $p \leq n$, the total number of operations is approximately:

$$\sum_{p \le n,\ p \text{ prime}} \frac{n}{p} = n \sum_{p \le n} \frac{1}{p}$$

The harmonic sum over primes is asymptotically bounded by:

$$\sum_{p \le n} \frac{1}{p} = \log \log n + c + o(1)$$

Therefore, the total time complexity becomes:

$$\mathcal{O}(n \log \log n)$$

This efficiency is achieved by marking each composite number only once using its smallest prime factor, and by starting the marking from $i^2$, which avoids redundant operations.

Learn more about Sieve Here[7].

It is also possible to do sieve in (almost) Linear Time[8] But most of the time, it is unnecessary.

## 3.3 Prime Power Factorization (PPF)

We already know how to find all the primes upto $n$ using sieve from the subsection 3.2.

We can now use this to prime factorize a number. Let's assume we have a list of primes upto some $N$. Then, we can factorize any $n \le N$ like the following:

```
vector<pair<int,int>> ppf(int n){
  vector<pair<int,int>> factors;
  for (int i = 0; primes[i] * primes[i] <= n; i++){
    int cnt = 0;
    while (n % primes[i] == 0){
      n /= primes[i];
      cnt++;
    }
    if (cnt > 0) factors.push_back({primes[i], cnt});
  }
  if (n > 1) factors.push_back({n, 1});
  return factors;
}
// if n = 12, it will return:
// {{2, 2}, {3, 1}} which means 12 = 2^2 * 3^1
```

Time complexity of this algorithm is $\mathcal{O}(\sqrt{n}/\log\sqrt{n})$. In practice, it is actually quite faster than the theoretical bound.

**Note**: The PPF function is pretty slow if $n$ is prime because it is unnecessarily looping through all the prime numbers till $\sqrt{n}$. So, in rare cases, it might be a good idea to first check its primality

[7] https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html
[8] https://cp-algorithms.com/algebra/prime-sieve-linear.html

using miller-rabin or such. If it is prime, just return $n$ as the only factor.

You can now go back to Section 2.3 and solve all the problems again as we had previously assumed we could prime factorize a number easily.

**Problem 3.3: HS08PAUL - A conjecture of Paul Erdős**[a]

[a] https://www.spoj.com/problems/HS08PAUL/

In number theory, there is a very deep unsolved conjecture posed by the Hungarian mathematician Paul Erdős (1913–1996): that there exist infinitely many primes of the form $x^2 + 1$, where $x$ is an integer.

However, a weaker form of this conjecture has been proven: there are infinitely many primes of the form $x^2 + y^4$, where $x$ and $y$ are integers.

You do not need to prove this. Your task is simply to compute the number of **positive primes** not larger than a given integer $n$, which can be expressed in the form $x^2 + y^4$ for some integers $x$ and $y$.

There will be $10^4$ test cases, and $1 \le n \le 10^7$.

**Solution 3.3**

First of all, we can find all the primes upto $10^7$ using sieve. Let's say we've the **is_composite** array. We haven't pushed anything in the **primes** vector yet. Now, what we can do is go through all possible $x^2 + y^4$ and mark all the primes that are of that form. It is easy to notice that $x^2 \le 10^7$ and $y^4 \le 10^7$. Or in other words, $x \le 10^4$ and $y \le 60$. So, we can definitely loop through all possible $x$ and $y$, and for each pair, we can check if $x^2 + y^4$ is prime. If it is, we can mark it in the **is_composite** array. Based on that array, we can generate a prefix sum array that can answer all the queries. And we can do all of these before the queries.

```
// run sieve and populate is_composite array
const int N = 1e7 + 10;
vector<bool> tmp(N);
for(long long x = 0; x <= 10000;x++){
   for(long long y = 0; y <= 60;y++){
      long long ep = x*x + y*y*y*y;
      if(ep >= N)break;
      if(!is_composite[ep]){
        tmp[ep] = true;
      }
   }
 }
// now we can generate a prefix sum array
vector<int> prefix(N);
for(int i = 1; i <= N-10;i++){
  prefix[i] = prefix[i-1] + (int)(tmp[i]);
}
// .... inputs and stuff
```

```
while(testcase--){
  int n;
  cin >> n;
  cout << prefix[n] << endl;
}
```

The time complexity of this algorithm is $\mathcal{O}(n \log\log n)$ for sieve, and $\mathcal{O}(n)$ for prefix sum. So, the total time complexity is $\mathcal{O}(n \log\log n)$. The space complexity is $\mathcal{O}(n)$ for the sieve and prefix sum arrays.

# 4 Number Theoretic Functions

In this chapter, we will study number theoretic functions. We will see how we can use our theoretical knowledge to solve some problems.

## 4.1 Multiplicative Functions

**Definition 4.1: Multiplicative functions**

A function $f : \mathbb{N} \to \mathbb{C}$ is called **multiplicative** if:

- $f(1) = 1$ (unless $f(x) = 0$ for all $x$)
- For all coprime integers $a$ and $b$, we have $f(ab) = f(a)f(b)$.

For example, The following functions are multiplicative.

$\phi(n)$ **Euler's Totient Function** – counts the number of positive integers $\leq n$ that are coprime to $n$.

$\mu(n)$ **Möbius Function** – returns 0 if $n$ has squared prime factors, $(-1)^k$ if $n$ is a product of $k$ distinct primes.

$\sigma(n)$ **Sum of Divisors Function** – returns the sum of all positive divisors of $n$.

$\tau(n)$ **Number of Divisors Function** – returns the total number of positive divisors of $n$.

$\omega(n)$ **Number of Distinct Prime Factors** – counts how many distinct prime numbers divide $n$.

$\lambda(n)$ **Carmichael Function** – gives the smallest positive integer $m$ such that $a^m \equiv 1 \pmod{n}$ for all $a$ coprime to $n$.

Please try to prove that the above functions are indeed multiplicative.

There are many other multiplicative functions, and they can be used to derive properties of integers. For example, the product of two multiplicative functions is also multiplicative. This property is useful in number theory and combinatorial problems.

Multiplicative functions are really inportant in competitive programming. We already know that a positive integer $n$ can be uniquely represented as a product of prime powers:

$$n = p_1^{a_1} \cdot p_2^{a_2} \cdots p_k^{a_k}$$

Now, let's say you have to calculate a multiplicative function $f$. Since it is multiplicative, the following holds:

$$f(n) = f(p_1^{a_1}) \cdot f(p_2^{a_2}) \cdots f(p_k^{a_k})$$

This property is extremely strong because it allows us to calculate $f(n)$ by calculating the function values for the prime powers only, which is often much easier.

## 4.2 Number of Divisors

We have already solved how to find number of divisors in Problem 2.2. Now that we know $\tau(n)$ is multiplicative, we can verify if our combinatorial solution was indeed correct.

$$\begin{aligned}
\tau(1) &= 1 \\
\tau(p^k) &= k + 1 \\
\tau(n) &= \tau(p_1^{a_1}) \cdot \tau(p_2^{a_2}) \cdots \tau(p_k^{a_k}) \\
&= (a_1 + 1)(a_2 + 1) \cdots (a_k + 1)
\end{aligned}$$

As you can see, the formula we've found earlier was indeed correct. And please notice that how easy it was to figure this out once we knew this function was multiplicative.

## 4.3 Sum of Divisors

Sum of divisors of $n$, $\sigma(n)$ is also a multiplicative function. Previously, we have seen that $\sigma(n)$ can be calculated by finding all divisors of $n$ and summing them up. However, if $n$ is too large, this is not appropriate. Moreover, we just need the sum of divisors, why would we find all divisors? We can do something better. We will be using the fact that $\sigma(n)$ is multiplicative.

$$\begin{aligned}
\sigma(1) &= 1 \\
\sigma(p^k) &= 1 + p + p^2 + \ldots + p^k = \frac{p^{k+1} - 1}{p - 1} \\
\sigma(n) &= \sigma(p_1^{a_1}) \cdot \sigma(p_2^{a_2}) \cdots \sigma(p_k^{a_k}) \\
&= \prod_{i=1}^{k} \frac{p_i^{a_i+1} - 1}{p_i - 1}
\end{aligned}$$

Now, when we do prime power factorization, we can easily find $\sigma(n)$.

```
long long SumOfDivisors(long long num) {
  long long total = 1;
  for (int i = 2; (long long)i * i <= num; i++) {
    if (num % i == 0) {
      int e = 0;
      do {
        e++;
        num /= i;
      } while (num % i == 0);

      long long sum = 0, pow = 1;
      do {
        sum += pow;
        pow *= i;
      } while (e-- > 0);
      total *= sum;
    }
  }
  if (num > 1) {
    total *= (1 + num);
  }
  return total;
}
```

Learn more about number of divisors and sum of divisors from CP Algorithms. You'll find relevant problems underneath, should be helpful.

## 4.4 Euler Totient Function

Eulet totient function, $\phi(n)$, counts the number of integers from 1 to $n$ that are coprime to $n$. For example, $\phi(6) = 2$ because the integers 1 and 5 are coprime to 6. As it is multiplicative, we can try to derive a nice formula to calculate this.

$$\begin{aligned}
\phi(1) &= 1 \\
\phi(p^k) &= p^k - p^{k-1} = p^k(1 - \frac{1}{p}) \\
\phi(n) &= \phi(p_1^{a_1}) \cdot \phi(p_2^{a_2}) \cdots \phi(p_k^{a_k}) \\
&= n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \cdots (1 - \frac{1}{p_k})
\end{aligned}$$

Now you can calculate $\phi(n)$ during PPF like sum of divisors. However, if you need to calculate $\phi(i)$ for all $i$ less than or equal to some $n$, you can do that in $\mathcal{O}(n \log \log n)$ time using a sieve-like approach. The idea is to iterate through all integers from 2 to $n$ and for each integer $i$, if it is prime, we can update all multiples of $i$ by multiplying them with $(1 - \frac{1}{i})$. This is similar to the Sieve of Eratosthenes, but instead of marking primes, we are updating the values of $\phi(i)$.

Here is the code:

```
void phi_1_to_n(int n) {
    vector<int> phi(n + 1);
    for (int i = 0; i <= n; i++)
        phi[i] = i;

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

The following property is also useful sometimes:

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

Learn more about Euler's totient function from CP Algorithms. You'll find relevant problems underneat, should be helpful.

## 5 Miscellaneous

In this section, I will add some codes that you can keep in your template so that you can use them directly whenever necessary.

### 5.1 Miller-Rabin Primality Test

Study more about it Here.

```cpp
using u64 = uint64_t;
using u128 = __uint128_t;

u64 binpower(u64 base, u64 e, u64 mod) {
    u64 result = 1;
    base %= mod;
    while (e) {
        if (e & 1)
            result = (u128)result * base % mod;
        base = (u128)base * base % mod;
        e >>= 1;
    }
    return result;
}
bool check_composite(u64 n, u64 a, u64 d, int s) {
    u64 x = binpower(a, d, n);
    if (x == 1 || x == n - 1)
        return false;
    for (int r = 1; r < s; r++) {
        x = (u128)x * x % n;
        if (x == n - 1)
            return false;
    }
    return true;
};
bool MillerRabin(u64 n, int iter=5) {
  // returns true if n is probably prime, else returns false.
    if (n < 4)
        return n == 2 || n == 3;

    int s = 0;
    u64 d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        s++;
    }
    for (int i = 0; i < iter; i++) {
        int a = 2 + rand() % (n - 3);
        if (check_composite(n, a, d, s))
            return false;
    }
    return true;
}
```

## 5.2 Pollard Rho

Study More about it Here. Basically, the following function is used to find a non-trivial divisor of $n$. And if not found, it returns 1.

```
long long pollards_p_minus_1(long long n) {
    int B = 10;
    long long g = 1;
    while (B <= 1000000 && g < n) {
        long long a = 2 + rand() %  (n - 3);
        g = gcd(a, n);
        if (g > 1)
            return g;

        // compute a^M
        for (int p : primes) {
            if (p >= B)
                continue;
            long long p_power = 1;
            while (p_power * p <= B)
                p_power *= p;
            a = power(a, p_power, n);

            g = gcd(a - 1, n);
            if (g > 1 && g < n)
                return g;
        }
        B *= 2;
    }
    return 1;
}
```

## 5.3 ModInt Class

Whenever we have to perform modular arithmetic over addition, multiplication (and sometimes we have to work with inverses), it is common to miss modding in a few places. This can lead to wrong answers, and might be extremely hard to debug. It is better to use a class that will handle all the modular arithmetic for us. I am giving you one below, modify it however you want.

```cpp
class ModInt {
public:
  long long value;
  static const long long MOD = 1000000007;
  ModInt(long long v = 0) {
    value = v % MOD;
    if (value < 0)
      value += MOD;
  }
  ModInt &operator=(long long v) {
    value = v % MOD;
    if (value < 0)
      value += MOD;
    return *this;
  }
  ModInt operator+(const ModInt &other) const {
    return ModInt(value + other.value);
  }
  ModInt operator-(const ModInt &other) const {
    return ModInt(value - other.value);
  }
  ModInt operator*(const ModInt &other) const {
    return ModInt(value * other.value);
  }
  ModInt operator/(const ModInt &other) const { return *this * other.inv(); }
  ModInt pow(long long exp) const {
    ModInt base = *this;
    ModInt result = 1;
    while (exp > 0) {
      if (exp % 2 == 1)
        result = result * base;
      base = base * base;
      exp /= 2;
    }
    return result;
  }
  ModInt inv() const { return this->pow(MOD - 2); }
  friend std::ostream &operator<<(std::ostream &os, const ModInt &m) {
    os << m.value;
    return os;
  }

  friend std::istream &operator>>(std::istream &is, ModInt &m) {
    is >> m.value;
    m.value = m.value % MOD;
    if (m.value < 0)
      m.value += MOD;
    return is;
  }

  bool operator==(const ModInt &other) const { return value == other.value; }

  bool operator!=(const ModInt &other) const { return !(*this == other); }
};
```

```cpp
int main(){
  ModInt a = 5;
  ModInt b = 3;
  cout << a + b << "\n"; //(a+b) % MOD
  cout << a - b << "\n"; // (a-b+MOD) % MOD
  cout << a * b << "\n"; // (a*b) % MOD
  cout << a / b << "\n"; // (a * b^(-1)) % MOD
  cout << a.pow(3) << "\n"; // (a^3) % MOD
  cout << a.inv() << "\n"; // a^(-1) % MOD
}
```