

A competitive programmer’s Handbook



Welcome to the ultimate journey into the world of Competitive Programming and Data Structures. Whether you're a beginner or looking to sharpen your skills, this course is designed to equip you with the tools, techniques, and mindset to solve complex problems efficiently.

Let's dive into logic, algorithms, and beyond!

# 1 Number Theory

Number theory is perhaps the most interesting and beautiful area of mathematics. Computers have long been used in number theoretic research. Performing interesting number-theoretic computations on large integers requires great efficiency. Fortunately, there are many clever algorithms to help us out.

* + 1. Prime Numbers

A prime number is an integer p > 1 which is only divisible by 1 and itself. Said another way, if p is a prime number, then p = a · b for integers a ≤ b implies that a = 1 and b = p. The first ten prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, and 27.

Every integer can be expressed in only one way as the product of primes.

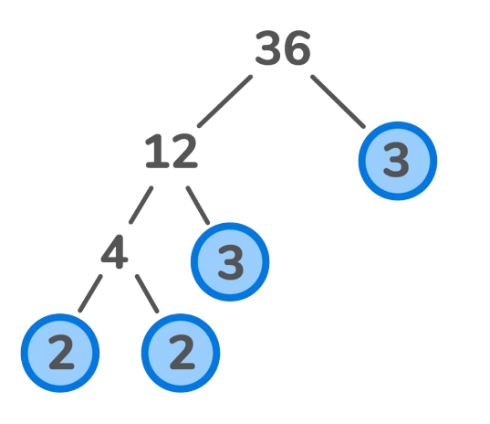
eg : 105 is uniquely expressed as 3 × 5 × 7,

while 32 is uniquely expressed as 2×2×2×2×2 . This unique set of numbers multiplying to N is called the prime factorization of N .

1.1.2 How to factorize into primes (optimized)

Every number has a unique prime factorization: away of decomposing it into a product of primes, as follows:

N =

 36 =

Algorithm :

Function factor(n)

Input: n (the number to be factorized)

Output: v (a list of all the prime factors of n)

V ← empty list

For i ←2 to

While n is divisible by i do :

n ← n /i

add i to the list v

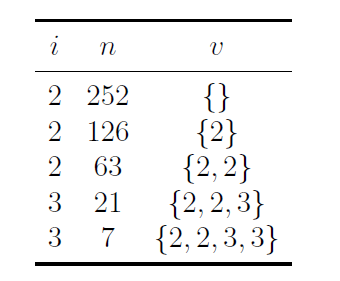
end

end

Return v

This algorithm runs in O() time, because the for loop checks divisibility for at most values. Even though there is a while loop inside the for loop, dividing n by i quickly reduces the value of n, which means that the outer for loop runs less iterations, which actually speeds up the code.

Example : N = 252



At this point, the for loop terminates, because i is already 3 which is greater than . In the last step, we add 7 to the list of factors v, because it otherwise won’t be added, for a final prime factorization of {2, 2, 3, 3, 7}.

PRACTICE PROBLEMS:

1. [You are given a positive integer *n*. Output its prime factorization](https://codeforces.com/problemset/problem/162/C).
2. [Representing a number as sum of primes](https://codeforces.com/problemset/problem/749/A)
3. [Given an array of positive integers nums, return *the number of****distinct prime factors****in the product of the elements of* nums](https://leetcode.com/problems/distinct-prime-factors-of-product-of-array/description/).
   * 1. Sieve Of Eratosthenes:

This Technique can be used on problems that requires list of primes (or) primality testing. It begins by listing all numbers from **2 to** . Starting with the smallest number, **2**, it marks all its proper multiples (i.e., numbers greater than 2 that are divisible by 2) as not prime. It then moves to the next unmarked number, **3**, marking all its multiples as not prime (3 is a prime, but 3\*k,where k->2,3…. is not a prime). This process continues with the next unmarked number, which is always prime, marking all its multiples until reaching ​, since larger composite numbers will already be marked

Here's a dry run of the algorithm:

Let’s generate all prime numbers from **1 to 20** using the **Sieve of Eratosthenes**.  
The algorithm runs for i=2 through = 4.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

1. We create an array of size **21** (indices from 0 to 20) and mark all numbers **as prime (true)**, except 0 and 1 (since they are not prime).
2. Begin from **i = 2** and mark all its multiples (except itself) up to 20.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

1. Increment i to 3 and check whether array[i] is marked**.**

* **If it is not marked** (meaning it’s a prime), then mark all multiples of 3 (except itself) up to 20.
* **If it is marked** (meaning it’s not a prime), increment i to 4.

Since array[i] is unmarked, we proceed with the first step.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

1. Increment **i to 4** (last iteration). Check if array[i] is marked… Yes, array[i] is marked. Since 4 is not a prime, it must be a multiple of some smaller prime number. This means its multiples would have already been marked when processing that smaller prime. Therefore, marking them again would be redundant. Thus, we skip this iteration.
2. End

Just iterate through the array and display all the unmarked elements.

Pseudo code:

Function sieve(N):

is\_prime[N+1] = true

is\_prime[0] = is\_prime[1] = False

for (i =2 to N ):

if is\_prime[i] == true: # If i is prime

for (j = 2 to N + 1):

is\_prime[j] = False

j += i

**Optimizing the Sieve of Eratosthenes**

The naive Sieve of Eratosthenes marks all multiples of a prime i starting from 2i. However, this leads to **redundant operations**, making it inefficient for large N.

**Redundant Marking:**

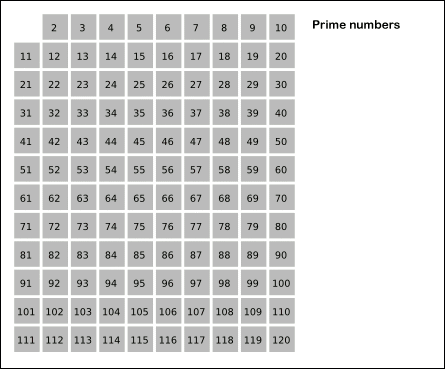
**Example:**

* For i=2 Marks 4**, 6**, 8, 10, **12**, ...
* For i=3 Marks **6**, 9, **12**, 15**, ...**

Here, 6 and 12 were already marked when processing i=2.

**Optimization Fix:** Start marking from Instead of , since smaller multiples were already handled by smaller primes.

Here’s an animation of the Sieve of Eratosthenes for a larger N.



Code Template :

C++:

int n;

vector<bool> is\_prime(n+1, true);

is\_prime[0] = is\_prime[1] = false;

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

if (is\_prime[i]) {

for (int j = i \* i; j <= n; j += i)

is\_prime[j] = false;

}

}

[Cite your source here.]

Python:

n = int(input())

is\_prime = [True] \* (n + 1)

is\_prime[0] = is\_prime[1] = False

for i in range(2, int(n\*\*0.5) + 1):

if is\_prime[i]:

for j in range(i \* i, n + 1, i):

is\_prime[j] = False

Pratice Problems:

1. [Straightforward implementation of Sieve of Eratosthenes.](https://www.geeksforgeeks.org/problems/sieve-of-eratosthenes5242/1)
2. [A modification on sieve.](https://codeforces.com/problemset/problem/230/B)
3. [Four divisors.](https://leetcode.com/problems/four-divisors/description/)
   * 1. Counting divisors of a number

**How many divisors does a number have?** Suppose you wish to find the number of divisors of 48. Starting with 1 we can work through the set of natural numbers and test divisibility in each case, noting that divisors can be listed in factor pairs.

6 \* 8

1 \* 48

2 \* 24

4 \* 12

3 \* 16

So, there are 10 divisors. But using this method we need to work from 1 to . It is impractical for larger numbers! Fortunately, there is an efficient way to do that! Denote as product of primes factors:

Then the number of divisors =

Eg : Prime factorization of 48 = , therefore, the no.of divisors is (4+1) . (1+1) = 10.

|  |  |  |
| --- | --- | --- |
|  |  | Divisior = |
|  |  | 1 |
|  |  | 6 |
|  |  | 4 |
|  |  | 24 |
|  |  | 16 |
|  |  | 3 |
|  |  | 2 |
|  |  | 12 |
|  |  | 8 |
|  |  | 48 |

As you can see, the table has 10 rows. Each divisor is formed by choosing an exponent for each prime factor:

* , where can be **0, 1, 2, 3, or 4** (**5 choices**).
* , where can be **0 or 1** (**2 choices**).

The total number of ways to combine these choices is:

(unordered pair)

which matches the number of divisors.

Pseudo code:

N = input()

primes = array containing primes till 10^6

ans = 1

for each prime p in primes:

if (p \* p \* p > N): // If p³ > N, no more small prime factors exist

break

count = 1

while (N % p == 0): // Count exponent of p in factorization

N = N / p

count = count + 1

ans = ans \* count // Apply divisor formula (e+1)

if (N is prime): // If N is still prime, it contributes (1+1) to divisor count

ans = ans \* 2

else if (N is a perfect square of a prime): // If N = p², it contributes (2+1)

ans = ans \* 3

else if (N != 1): // If N is still a product of two primes (pq), it contributes (1+1) \* (1+1) = 4

ans = ans \* 4

return ans

So, in this code we have already generated list of primes (generated from sieve of Eratosthenes) . Given a number we split into 2 parts:

GCD(

Since , we can compute the total divisors as:

Example:

Prime factorization of

-> for loop calculates this

-> outer if condition calculates this

So has 24 divisors

Time Complexity:

For generation of prime list : )

For prime factorization :

Overall :

Code Template:

int countDivisors(int n) {

if (n == 1) return 1;

/\* Assumes that the list of prime numbers and their squares are already available \*/ bool prime[n + 1], primesquare[n \* n + 1];

int a[n];

SieveOfEratosthenes(n, prime, primesquare, a);

int ans = 1;

for (int i = 0;; i++) {

if (a[i] \* a[i] \* a[i] > n) break;

int cnt = 1;

while (n % a[i] == 0) {

n = n / a[i];

cnt++;

}

ans \*= cnt;

}

if (prime[n]) ans \*= 2;

else if (primesquare[n]) ans \*= 3;

else if (n != 1) ans \*= 4;

return ans;

}

Python :

def count\_divisors(N, primes, prime\_squares):

"""

Assumes that the list of prime numbers and their squares are already available.

"""

ans = 1

for p in primes:

if p \* p \* p > N: # If p³ > N, no more small prime factors exist

break

count = 1

while N % p == 0: # Count exponent of p in factorization

N //= p

count += 1

ans \*= count # Apply divisor formula (e+1)

if N in primes: # If N is still prime, it contributes (1+1)

ans \*= 2

elif N in prime\_squares: # If N is a perfect square of a prime, it contributes (2+1)

ans \*= 3

elif N != 1: # If N is still a product of two primes (pq), it contributes (1+1) \* (1+1) = 4

ans \*= 4

return ans

Practice Problems:

1. [Strightforward implementation of counting divisors of a number N.](https://cses.fi/alon/task/1713)
2. [Count of square-free divisors of a number N](https://www.geeksforgeeks.org/count-of-square-free-divisors-of-a-given-number/?ref=ml_lbp)
3. [Counting divisors of a numbers within range [a ,b]](https://www.geeksforgeeks.org/find-the-number-of-divisors-of-all-numbers-in-the-range-1-n/?ref=ml_lbp)
4. [Find the sum of count of divisors](https://www.geeksforgeeks.org/find-the-sum-of-the-number-of-divisors/?ref=ml_lbp)
5. [Find i, such that the sum of divisors of i equals N](https://www.spoj.com/problems/INVDIV/)