

# Level 1 Training

---

Session 5

## Number Theory Part 1

Divisors, prime factorization,  
and the sieve

Find the factors. Explain the exponents.  
Precompute only the information you need.

# How to use this reference

This reference explains the mathematics behind the code as well as the implementation. Study it after the session in small stages. A formula is useful only when you can explain what it counts, why it is valid, and which inputs make its implementation safe.

## A study routine for number theory

1. **Start with a small number.** List the divisors of 12 or factor 72 on paper. Use the list to check the meaning of a new formula.
2. **Name the stored information.** Distinguish the original number from its remaining unfactored part. State whether a table stores a flag, a factor, an exponent, or a count.
3. **Explain the stopping condition.** Prove why searching beyond a boundary is unnecessary. Include a perfect square in every square-root-loop trace.
4. **Test and rebuild.** Include 1, a prime, a prime square, repeated factors, the table limit, and an empty batch when allowed. Use hints before solutions; later rewrite the method without looking.

## How the material is labelled

A **complete example** or **complete solution** has its own headers and `main`. Each listing is loaded from a separate included `.cpp` file. Compile one program at a time. A fragment illustrates a smaller idea; deliberately wrong examples are not working templates.

**Pause and predict** questions have an explained answer nearby. Cover it first. The four optional extensions and Practice 18 to 20 can wait until a second reading. Main topics and coding solutions begin on fresh pages.

### Three objects that must not be confused

The divisors of 12 are 1, 2, 3, 4, 6, 12. Its prime factors with multiplicity are 2, 2, 3. Its distinct prime factors are 2, 3. Their counts are 6, 3, and 2 respectively. A program may compile correctly while answering the wrong one of these questions.

## Language and prerequisites

The complete programs use **standard C++23**, without GNU extension headers. You should know loops, functions, vectors, pairs, basic complexity, and Level 0 frequency arrays and prefix sums. Session 4's multiplication rule explains divisor-count formulas. Session 1's binary search is reused in one practice solution.

GCD, LCM, modular inverses, and binary exponentiation belong to **Session 6**. This chapter does not require those algorithms. Exact results are computed under explicit bounds rather than assuming `long long` can hold every possible product.

# The Level 1 learning path

| Session | Topics                                                                                                                                        |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| 1       | <b>Two Pointers and Binary Search.</b> Part A: two pointers and sliding windows. Part B: sorted lookup, boundaries, and search on the answer. |
| 2       | <b>Recursion and Backtracking.</b> Calls, decision trees, enumeration, pruning, and state restoration.                                        |
| 3       | <b>Bit Manipulation and Bitmasking.</b> Binary values, bitwise operations, XOR, compact sets, and subsets.                                    |
| 4       | <b>Counting Principles.</b> Addition, multiplication, $nCr$ , $nPr$ , and inclusion-exclusion.                                                |
| 5       | <b>Number Theory, Part 1.</b> Divisors, prime factorization, and the sieve.                                                                   |
| 6       | <b>Number Theory, Part 2.</b> Modular arithmetic, binary power (exponentiation), modular inverse, GCD, and LCM.                               |

## What this edition contains

This handout contains **Session 5: Number Theory, Part 1**. Section numbers begin with 5 and match the collected book. The collected edition now contains **Sessions 1 to 5**; Session 6 is planned but is not included yet.

## What you should be able to do afterwards

Find positive divisors without scanning every candidate up to the number. Test primality, factor a positive integer, preserve repeated prime factors, derive divisor counts and sums, and select an appropriate precomputation for many questions. Explain why a leftover factor is prime and why a sieve can begin crossing off at a prime's square.

## Conventions used throughout

Unless a task says otherwise, a divisor is **positive** and the number being factored is at least 1. Primality tests may accept 0, which is not prime. All input obeys the stated bounds; programs print no prompts. **EMPTY** means a requested factor or prime list has no entries. The divisor list of 1 is *not* empty: it contains 1.

We write  $n$  for one input number,  $N$  or  $M$  for a precomputation limit, and  $q$  for a query count.  $D$  denotes the number of positive divisors of the current input. Complexity counts fixed-width arithmetic operations, includes preprocessing and output where relevant, and states stored-table memory separately from small loop variables.

# Contents

|                                                                                   |          |
|-----------------------------------------------------------------------------------|----------|
| How to use this reference                                                         | i        |
| The Level 1 learning path                                                         | ii       |
| <b>5 Number Theory, Part 1</b>                                                    | <b>1</b> |
| 5.1 Divisibility: an exact integer relationship . . . . .                         | 2        |
| 5.2 Positive divisors, proper divisors, and the special input 1 . . . . .         | 3        |
| 5.3 From a direct scan to paired divisors . . . . .                               | 4        |
| 5.4 Square-root boundaries without floating-point arithmetic . . . . .            | 5        |
| 5.5 Print paired divisors in increasing order . . . . .                           | 6        |
| 5.6 Count or sum divisors without storing the list . . . . .                      | 8        |
| 5.7 Prime numbers and the smallest-factor argument . . . . .                      | 10       |
| 5.8 Single-number methods: time, input size, and limitations . . . . .            | 11       |
| 5.9 Prime factorization: one number, a unique prime structure . . . . .           | 12       |
| 5.10 Trial factorization: divide out every copy before advancing . . . . .        | 13       |
| 5.11 The shrinking bound and the final leftover factor . . . . .                  | 15       |
| 5.12 Distinct prime factors, repeated factors, and divisors . . . . .             | 17       |
| 5.13 Count divisors by choosing prime exponents . . . . .                         | 19       |
| 5.14 Sum divisors by multiplying prime-power sums . . . . .                       | 21       |
| 5.15 Proper-divisor sums and square conditions . . . . .                          | 23       |
| 5.16 Generate all divisors from the prime factorization . . . . .                 | 24       |
| 5.17 Choose a precomputation from the query pattern . . . . .                     | 26       |
| 5.18 The sieve of Eratosthenes: cross off composites together . . . . .           | 27       |
| 5.19 Implement the prime-flag sieve safely . . . . .                              | 28       |
| 5.20 Sieve complexity: count visits, not just nested loops . . . . .              | 30       |
| 5.21 Prime-count intervals: combine a sieve with prefix sums . . . . .            | 31       |
| 5.22 Smallest prime factors: store a useful witness, not only a flag . . . . .    | 33       |
| 5.23 Factor a bounded value by following SPF entries . . . . .                    | 35       |
| 5.24 Use precomputed primes to factor larger numbers . . . . .                    | 37       |
| 5.25 Combine prime exponents without constructing a huge product . . . . .        | 39       |
| 5.26 Divisor tables: reverse the direction and visit multiples . . . . .          | 41       |
| 5.27 Reusable tables, fresh queries, and honest total complexity . . . . .        | 43       |
| 5.28 Debug the mathematical meaning before changing the syntax . . . . .          | 44       |
| 5.29 Optional extension: a linear smallest-prime-factor sieve . . . . .           | 45       |
| 5.30 Optional extension: a segmented sieve for a short distant interval . . . . . | 47       |
| 5.31 Optional extension: prime exponents in factorials . . . . .                  | 49       |
| 5.32 Optional extension: the smallest multiplier that makes a square . . . . .    | 50       |
| 5.33 Practice problems: choose the information before the loop . . . . .          | 52       |
| 5.34 Trace, correctness, and complexity drills . . . . .                          | 59       |
| 5.35 Hints before solutions . . . . .                                             | 62       |
| 5.36 Explained practice solutions . . . . .                                       | 65       |
| 5.37 Explained trace and reasoning drills . . . . .                               | 86       |
| 5.38 Quick reference, glossary, and readiness checklist . . . . .                 | 90       |
| 5.39 Technical notes, verification, and references . . . . .                      | 95       |

## SESSION 5

# Number Theory, Part 1

**Find the factors. Explain the exponents. Precompute only what you need.**

Many integer problems ask about exact structure: which values divide a number, which primes build it, or how often a property occurs in a bounded range. The useful shortcut is not a new container. It is an argument that tells you which divisions can be skipped or which answers can be prepared together.

We begin with the meaning of divisibility, then derive increasingly reusable methods. Every method has a domain: trial division is not a general fast factorizer for arbitrary 64-bit inputs, and a sieve cannot answer a table lookup beyond the limit that was built.

## Three connected viewpoints

| Viewpoint          | Information                       | Typical question                                |
|--------------------|-----------------------------------|-------------------------------------------------|
| Divisor pairs      | $d$ and $n/d$ multiply to $n$ .   | Which rectangle side lengths are possible?      |
| Prime exponents    | $n$ is a product of prime powers. | How many different divisors does $n$ have?      |
| Precomputed tables | Many related values share work.   | How many primes are in each requested interval? |

## Your route through the chapter

**Stage A: one number.** Understand divisors, paired enumeration, primality, and safe square-root boundaries.

**Stage B: prime structure.** Factor a number completely, group repeated primes, and use independent exponent choices to count, sum, or generate divisors.

**Stage C: many questions.** Build prime flags, prime prefixes, smallest-prime-factor tables, and divisor tables. Decide what to reuse and what to reset.

**Stage D: independent work.** Solve the practice, justify each method, and test its boundaries. Optional topics introduce linear and segmented sieves, factorial exponents, and completing a square.

### The question to answer before choosing an algorithm

Am I processing one number, a few large numbers, or many bounded numbers? The largest input and the number of queries matter together. A fast single-query method can still be too slow when repeated many times.

## 5.1 Divisibility: an exact integer relationship

### What does “d divides n” mean?

For a positive integer  $d$ , we write  $d \mid n$  when there is an integer  $k$  such that  $n = d \cdot k$ . Read the vertical bar as “divides”. For example,  $7 \mid 84$  because  $84 = 7 \cdot 12$ , but  $5 \nmid 84$  because 84 is not an integer multiple of 5.

The divisor is the number doing the dividing. A multiple is the number obtained after multiplication. In  $84 = 7 \cdot 12$ , 7 and 12 are divisors of 84, and 84 is a multiple of both. The statement  $84 \mid 7$  is a different, false statement.

### Translate the definition into C++

For integers,  $n / d$  computes the quotient with the fractional part discarded, and  $n \% d$  computes the remainder. Therefore, with positive  $d$ ,  $n \% d == 0$  tests exact divisibility. Division or remainder by zero is undefined behavior; the task must exclude zero divisors or the program must guard them.<sup>[5-5]</sup>

**Contract.** Read  $0 \leq n \leq 10^{12}$  and  $1 \leq d \leq 10^{12}$ . Print YES if  $d$  divides  $n$ , otherwise NO.

**Complete example | C++23**

```
#include <iostream>

int main() {
    long long n = 0, d = 0;
    std::cin >> n >> d;
    std::cout << (n % d == 0 ? "YES" : "NO") << '\n';
}
```

#### Sample input

84 7

#### Sample output

YES

This test takes  $O(1)$  time and extra space in our fixed-width model. It does not list factors; it answers one yes/no question about one candidate.

#### The mathematical bar is not the C++ operator

In C++,  $a \mid b$  is bitwise OR, which you studied in Session 3. It is not a divisibility test. Similarly,  $n / d == 0$  asks whether the truncated quotient is zero, not whether the division has no remainder.

#### Pause and predict: 1

Does 7 divide 0? Is  $0 \% 7$  valid? Is  $7 \% 0$  valid?

**Check your prediction.** Yes,  $0 = 7 \cdot 0$ , and  $0 \% 7$  is 0.  $7 \% 0$  is invalid. The two operands have different roles.

## 5.2 Positive divisors, proper divisors, and the special input 1

### Agree on the objects before counting them

The positive divisors of 12 are

1, 2, 3, 4, 6, 12.

All six divide 12 exactly. The **proper positive divisors** exclude 12 itself, leaving 1, 2, 3, 4, 6. Unless explicitly stated otherwise, “divisor” in this chapter means positive divisor, including 1 and the number itself.

Negative divisors exist mathematically:  $-3$  divides 12 because  $12 = (-3)(-4)$ . They are not part of our lists. A task requesting all signed divisors would require a different output convention.

| $n$ | Positive divisors | Proper divisors | Prime factors |
|-----|-------------------|-----------------|---------------|
| 1   | 1                 | None            | None          |
| 2   | 1, 2              | 1               | 2             |
| 9   | 1, 3, 9           | 1, 3            | 3, 3          |
| 12  | 1, 2, 3, 4, 6, 12 | 1, 2, 3, 4, 6   | 2, 2, 3       |

### Why 1 must not be treated like 0

The only positive divisor of 1 is 1. Its divisor count and divisor sum are both 1. It has no prime factors: the product of no factors is 1, called the **empty product**. Its proper-divisor sum is 0.

Every positive integer divides 0. Consequently, 0 does not have a finite list of all positive divisors. The paired-divisor and factorization tasks require  $n \geq 1$ ; a primality test can separately accept 0 and return false.

### Read the output requirement precisely

A problem may ask for a count, a sum, a list, a prime factor, or a factor pair. These are not interchangeable. For  $n = 12$ , the largest positive divisor is 12, the largest proper divisor is 6, and the largest prime divisor is 3.

#### Pause and predict: 2

A routine begins with an empty result vector. Should the divisor list for 1 remain empty?

**Check your prediction.** No. It must contain 1. An empty prime-factor list for 1 is correct, but that is a different kind of list.

## 5.3 From a direct scan to paired divisors

### The slow method is easy to justify

For positive  $n$ , every positive divisor lies between 1 and  $n$ . Check all candidates and print those that pass the remainder test. This is a good first solution and a useful checker for small inputs.

**Contract.** Read  $1 \leq n \leq 100000$ . Print its positive divisors in increasing order.

**Complete example | C++23**

```
#include <iostream>

int main() {
    int n = 0;
    std::cin >> n;
    bool first = true;
    for (int d = 1; d <= n; ++d) {
        if (n % d != 0) continue;
        if (!first) std::cout << ' ';
        std::cout << d;
        first = false;
    }
    std::cout << '\n';
}
```

#### Sample input

12

#### Sample output

1 2 3 4 6 12

The loop performs  $n$  tests:  $O(n)$  time and  $O(1)$  extra space if the results are printed immediately. The output is already sorted because candidates are tested in increasing order.

### A divisor gives you its partner for free

If  $d \mid n$ , then  $n/d$  is also an integer divisor. For 36, the pairs are (1, 36), (2, 18), (3, 12), (4, 9), and (6, 6). Checking 18 again discovers the same pair as checking 2.

In every pair, at least one side is at most  $\sqrt{n}$ . Otherwise both sides would exceed  $\sqrt{n}$ , so their product would exceed  $n$ . Thus we can discover every pair by testing only  $1 \leq d \leq \lfloor \sqrt{n} \rfloor$ .

#### Pause and predict: 3

For  $n = 35$ , is the pair (5, 7) discovered before the square-root boundary?

**Check your prediction.** Yes.  $5 \leq \sqrt{35} < 6$ , so the test at  $d = 5$  discovers both 5 and 7. There is no need to test  $d = 7$  again.

## 5.4 Square-root boundaries without floating-point arithmetic

### The endpoint must include a perfect square

For  $n = 49$ , the pair  $(7, 7)$  sits exactly on the boundary. Using  $d^2 < n$  instead of  $d^2 \leq n$  misses this divisor. A primality test with the same strict inequality can incorrectly declare 49 prime.

In integer C++, even `d * d <= n` needs care. The multiplication happens before the comparison. An overflowing intermediate expression is not rescued by the comparison's result or by assigning that result to a wider variable. Signed overflow is undefined behavior.<sup>[5-6]</sup>

For positive integers, an exact alternative is

$$d^2 \leq n \iff d \leq \left\lfloor \frac{n}{d} \right\rfloor.$$

Use a strictly positive loop variable:

```
// Fragment inside a function; n >= 1.
for (long long d = 1; d <= n / d; ++d) {
    // Check whether d divides n.
}
```

No square is formed in the condition. Since  $d$  starts at 1, the division is defined. This is a boundary comparison; a separate `n % d == 0` still decides divisibility.

### Trace the complete paired scan for 36

| $d$ | $36/d$ | $36 \bmod d$ | Divides? | New divisors |
|-----|--------|--------------|----------|--------------|
| 1   | 36     | 0            | Yes      | 1 and 36     |
| 2   | 18     | 0            | Yes      | 2 and 18     |
| 3   | 12     | 0            | Yes      | 3 and 12     |
| 4   | 9      | 0            | Yes      | 4 and 9      |
| 5   | 7      | 1            | No       | None         |
| 6   | 6      | 0            | Yes      | 6, once      |
| 7   | 5      |              | Stop     | $7 > 36/7$   |

#### A useful alternative under a smaller contract

With `int p` and a bounded sieve limit, `1LL * p * p` performs the multiplication in `long long`. It is safe only after the actual bounds are checked. Adding `1LL` does not make an arbitrarily large 64-bit product safe.

#### Pause and predict: 4

Why does the condition use `d <= n / d`, but the duplicate check uses `d == n / d` only after the remainder test?

**Check your prediction.** The condition locates the square-root boundary even for nondivisors. After exact divisibility is known, equality identifies the one square-root divisor whose partner is itself.

## 5.5 Print paired divisors in increasing order

### Pair discovery order is not sorted output order

Printing both partners immediately for 36 gives 1, 36, 2, 18, 3, 12, 4, 9, 6. That is a complete list, but not an increasing list. Sorting the collected divisors would work. We can also exploit the order already present.

As  $d$  grows, small divisors grow. Their partners  $n/d$  decrease. Put the small divisors in one vector and the distinct large partners in another. Append the second vector backwards after the first.

### Why the helper works

Each divisor pair is discovered once, using its smaller member. The equality guard stores a square root only once. All entries in the first vector are no greater than  $\sqrt{n}$ , and all entries in the second are greater. Reversing the second vector therefore produces one increasing, duplicate-free list.

If  $D$  is the number of divisors, the cost is  $O(\sqrt{n} + D)$  time and  $O(D)$  storage. The two vectors and the returned list are part of that storage bound. No  $O(D \log D)$  sorting step is used here.

#### Pause and predict: 5

For  $n = 16$ , what are the two vectors before the append?

**Check your prediction.** The small vector is  $[1, 2, 4]$  and the large vector is  $[16, 8]$ . The final list is  $[1, 2, 4, 8, 16]$ . The divisor 4 must not appear in both vectors.

**Implementation: sorted paired divisors**

**Contract.** Read  $1 \leq n \leq 10^{12}$ . Print all positive divisors in increasing order.

**Complete example | C++23**

```
#include <iostream>
#include <vector>

std::vector<long long> divisors(long long n) {
    std::vector<long long> small, large;
    for (long long d = 1; d <= n / d; ++d) {
        if (n % d != 0) continue;
        small.push_back(d);
        if (d != n / d) large.push_back(n / d);
    }
    for (int i = static_cast<int>(large.size()) - 1; i >= 0; --i) {
        small.push_back(large[i]);
    }
    return small;
}

int main() {
    long long n = 0;
    std::cin >> n;
    const auto result = divisors(n);

    bool first = true;
    for (long long value : result) {
        if (!first) std::cout << ' ';
        std::cout << value;
        first = false;
    }
    if (first) std::cout << "EMPTY";
    std::cout << '\n';
}
```

---

**Sample input**

---

**Sample output**

---

36

---

1 2 3 4 6 9 12 18 36

---

## 5.6 Count or sum divisors without storing the list

### Each ordinary pair contributes two, but a square contributes one

A task asking only for a count does not need a vector. Count two for a pair with distinct members and one for a pair with equal members. The same idea adds the values instead of counting them when a sum is requested.

**Contract.** Read  $1 \leq n \leq 10^{12}$ . Print the number of positive divisors.

**Complete example | C++23**

```
#include <iostream>

int main() {
    long long n = 0, count = 0;
    std::cin >> n;
    for (long long d = 1; d <= n / d; ++d) {
        if (n % d != 0) continue;
        count += (d == n / d ? 1 : 2);
    }
    std::cout << count << '\n';
}
```

#### Sample input

36

#### Sample output

9

Both count-only and sum-only paired scans use  $O(\sqrt{n})$  time and  $O(1)$  extra space. Do not add a vector merely because another divisor problem used one.

### A square has an odd divisor count

Every nonsquare divisor is matched with a different partner. A perfect square has exactly one unmatched middle divisor,  $\sqrt{n}$ . Thus a positive integer has an odd number of divisors exactly when it is a perfect square. This includes 1, whose divisor count is 1.

#### Pause and predict: 6

A program adds 2 for every successful test. What does it print for 36, and why is that wrong?

**Check your prediction.** It prints 10 instead of 9. The five discovered pairs include (6,6), which represents only one divisor, not two.

**Implementation: add the values of each pair**

**Contract.** Read  $1 \leq n \leq 10^{12}$ . Print the sum of its positive divisors.

**Complete example | C++23**

```
#include <iostream>

int main() {
    long long n = 0, sum = 0;
    std::cin >> n;
    for (long long d = 1; d <= n / d; ++d) {
        if (n % d != 0) continue;
        sum += d;
        if (d != n / d) sum += n / d;
    }
    std::cout << sum << '\n';
}
```

**Sample input****Sample output**

36

91

For 36, the sum is  $1 + 2 + 3 + 4 + 6 + 9 + 12 + 18 + 36 = 91$ . The conditional addition prevents adding 6 twice.

**The accumulator needs its own bound**

The sum can exceed  $n$ . For example, the divisor sum of 12 is 28. An input fitting in `int` does not guarantee its divisor sum fits in `int`.

A deliberately loose bound is enough here: there are at most  $2\lfloor\sqrt{n}\rfloor$  positive divisors and each is at most  $n$ , so the sum is at most  $2n\lfloor\sqrt{n}\rfloor \leq 2 \cdot 10^{18}$  when  $n \leq 10^{12}$ . This fits a signed 64-bit `long long`. We use the bound as a proof; the program does not multiply those large terms to compute the answer.

**Avoid unnecessary stored state**

If the task needs a list, store or print the list. If it needs only a sum or count, keep only that aggregate. This reduces memory and usually makes the invariant easier to explain.

## 5.7 Prime numbers and the smallest-factor argument

### What makes a positive integer prime?

A **prime** is an integer greater than 1 with exactly two positive divisors: 1 and itself. A **composite** is an integer greater than 1 that is not prime. The numbers 0 and 1 are neither prime nor composite under these definitions. The only even prime is 2: every larger even number has 2 as a proper divisor.<sup>[5-3]</sup>

To prove a number composite, one proper divisor is enough. To prove it prime, a search must justify why no untested candidate could be a proper divisor.

### Every composite number has a small prime divisor

If  $n = ab$  with  $1 < a, b < n$ , at least one factor is at most  $\sqrt{n}$ . If that factor is composite, keep splitting it until a prime divisor is obtained. That prime is still at most  $\sqrt{n}$ . Thus every composite  $n$  has a prime divisor at most  $\sqrt{n}$ .

Checking all integers from 2 through  $\lfloor \sqrt{n} \rfloor$  is therefore sufficient. Checking only prime candidates would also be sufficient, but requires knowing those primes. We first remove the obvious even candidates without building a table.

**Contract.** Read  $0 \leq n \leq 10^{12}$ . Print PRIME or NOT PRIME.

**Complete example | C++23**

```
#include <iostream>

bool isPrime(long long n) {
    if (n < 2) return false;
    if (n % 2 == 0) return n == 2;
    for (long long d = 3; d <= n / d; d += 2) {
        if (n % d == 0) return false;
    }
    return true;
}

int main() {
    long long n = 0;
    std::cin >> n;
    std::cout << (isPrime(n) ? "PRIME" : "NOT PRIME") << '\n';
}
```

#### Sample input

49

#### Sample output

NOT PRIME

$n < 2$  handles 0 and 1. The even-number branch handles 2 correctly and rejects larger evens. Each later candidate is odd. A discovered divisor permits an immediate return; otherwise the square-root argument justifies returning true.

#### Pause and predict: 7

Why is “n is odd” not a primality test?

**Check your prediction.** Odd composites such as 9, 15, and 49 exist. Oddness eliminates the factor 2, not every possible factor.

## 5.8 Single-number methods: time, input size, and limitations

### Worst-case work is controlled by the boundary, not by easy samples

A prime input forces the trial-division test to reach its square-root boundary. Some composite inputs also have no very small factor. Trying only odd candidates roughly halves the number of trial divisions, but the asymptotic worst-case bound remains  $O(\sqrt{n})$ . Extra space is  $O(1)$ .

For  $n \leq 10^{12}$ , the boundary is at most  $10^6$ . For  $n$  near  $10^{18}$ , it is around  $10^9$ . The latter is not a reasonable general-purpose contest strategy just because the input fits in `long long`. A numeric type answers “Can I store it?”; a complexity estimate answers “Can I process it this way?”

### Count the batch as well as one call

For queries  $n_1, \dots, n_q$ , separate trial tests cost

$$O\left(q + \sum_{i=1}^q \sqrt{n_i}\right)$$

in the worst case. The simpler upper bound  $O(q\sqrt{M})$  applies when all  $n_i \leq M$  and  $M \geq 1$ . A million small-factor inputs may run quickly, but that does not establish a bound for a million prime inputs.

### Value size and written input length are different

A number  $n$  needs about  $\log_2 n$  binary digits. An  $O(\sqrt{n})$  method is not polynomial in that bit length. You do not need this distinction for the implementation, but it explains why enlarging the allowed number of digits can change feasibility dramatically.

#### No silent generalization to arbitrary 64-bit factorization

The programs are exact within their stated bounds. This session does not teach probabilistic primality testing, Pollard’s rho, or cryptographic factorization. Those are separate algorithms, not automatically supplied by changing `int` to `long long`.

#### Pause and predict: 8

You need primality for 200000 values, all at most 1000000. Should you automatically run trial division separately for each?

**Check your prediction.** No. A shared sieve up to 1000000 prepares the whole bounded domain once, then each primality lookup is constant-time. The precomputation topics develop that alternative.

## 5.9 Prime factorization: one number, a unique prime structure

### Break composite factors until only primes remain

Consider two ways to split 72:

$$72 = 8 \cdot 9 = (2 \cdot 2 \cdot 2)(3 \cdot 3), \quad 72 = 6 \cdot 12 = (2 \cdot 3)(2 \cdot 2 \cdot 3).$$

The order of discovery differs, but the prime occurrences are the same. Grouping equal primes gives the compact form  $72 = 2^3 \cdot 3^2$ .

The **fundamental theorem of arithmetic** states that every integer greater than 1 has a prime factorization unique up to the order of its factors. Thus, once prime values are ordered increasingly, their exponents are determined by the number.<sup>[5-1]</sup>

### What the notation means

Write

$$n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k},$$

where the  $p_i$  are different primes and  $e_i \geq 1$ . For 360, the primes are 2, 3, 5 and the exponents are 3, 2, 1:

$$360 = 2^3 \cdot 3^2 \cdot 5.$$

An exponent tells you how many copies of that prime are multiplied, not how many distinct primes there are.  $2^3$  means  $2 \cdot 2 \cdot 2$ , not  $2 \cdot 3$ .

For 1, the factor list is empty. We do not add 1 as a prime factor, and we do not print a factor pair (1, 1). A divisor representation and a prime-factor representation answer different questions.

### Why uniqueness is useful in algorithms

Instead of repeatedly testing all possible divisors, we can reason about how many copies of each prime a divisor may use. Equality of products, divisor counts, and perfect-square properties can be expressed through exponents. The next topics derive these consequences one at a time.

#### Pause and predict: 9

Does  $12 = 2 \cdot 6$  count as a prime factorization? What must happen next?

**Check your prediction.** No, because 6 is composite. Continue splitting it into  $2 \cdot 3$ , giving  $12 = 2^2 \cdot 3$ . A factorization is complete only when every listed factor is prime.

## 5.10 Trial factorization: divide out every copy before advancing

### The remaining value records unfinished work

Start with `remaining = n`. Try candidate divisors in increasing order, beginning at 2. Whenever a candidate divides the remaining value, record it and divide the remaining value by it. Keep trying the *same* candidate until it no longer divides.

The invariant is

$$\text{original } n = (\text{product of recorded prime factors}) \cdot \text{remaining}.$$

Every successful division moves one exact factor from the remaining value into the output list. Nothing is lost, and no approximate arithmetic is involved.

### Why a successful candidate is prime

Suppose a composite candidate  $p$  divided the remaining value. Some smaller prime would divide  $p$ , and would therefore divide the remaining value too. But every copy of every smaller prime has already been removed. This contradiction explains why trying all integers still records only primes.

#### Pause and predict: 10

Why is the repeated-division statement a `while`, not an `if`?

**Check your prediction.** A prime may occur several times. For  $n = 12$ , removing only one 2 leaves 6. Advancing immediately can leave a composite remainder and lose the correct exponents. Every copy of the current prime must be removed before advancing.

**Implementation: remove all copies of each prime**

**Contract.** Read  $1 \leq n \leq 10^{12}$ . Print its prime factors in increasing order, repeating each occurrence. Print EMPTY for 1.

**Complete example | C++23**

```
#include <iostream>
#include <vector>

int main() {
    long long remaining = 0;
    std::cin >> remaining;
    std::vector<long long> result;
    for (long long p = 2; p <= remaining / p; ++p) {
        while (remaining % p == 0) {
            result.push_back(p);
            remaining /= p;
        }
    }
    if (remaining > 1) result.push_back(remaining);

    bool first = true;
    for (long long value : result) {
        if (!first) std::cout << ' ';
        std::cout << value;
        first = false;
    }
    if (first) std::cout << "EMPTY";
    std::cout << '\n';
}
```

---

**Sample input****Sample output**

---

360

2 2 2 3 3 5

---

## 5.11 The shrinking bound and the final leftover factor

### The bound belongs to the remaining number

When factors have been removed, we need to factor only what remains. It is safe to stop trying candidates once  $p > \text{remaining}/p$ . Keeping the original  $n$  in the bound is correct but can perform unnecessary trials. Updating the bound to the remainder captures the progress already made.

For 360, the process is:

| Candidate | Before | After | What was recorded                                                 |
|-----------|--------|-------|-------------------------------------------------------------------|
| 2         | 360    | 45    | Three copies: $360 \rightarrow 180 \rightarrow 90 \rightarrow 45$ |
| 3         | 45     | 5     | Two copies: $45 \rightarrow 15 \rightarrow 5$                     |
| 4         | 5      | 5     | Stop: $4 > 5/4$                                                   |
| Leftover  | 5      |       | One prime factor 5                                                |

### Prove the leftover is prime

When the loop ends, every smaller candidate has been considered and all of its prime factors have been removed. If the remaining value greater than 1 were composite, it would have a prime divisor at most its square root. That divisor would be smaller than the first untested candidate, so it would already have been removed. Contradiction.

Therefore a remaining value greater than 1 is one final prime factor. If the remainder is 1, do not append anything. The same reasoning handles an originally prime input: no trial succeeds, and the original number becomes the leftover.

### Grouped exponents make the structure explicit

**Contract.** Read  $1 \leq n \leq 10^{12}$ . Print one line **prime exponent** for every distinct prime, in increasing order; print **EMPTY** for 1.

## Complete example | C++23

```

#include <iostream>
#include <vector>
#include <utility>

std::vector<std::pair<long long, int>> factorize(long long remaining) {
    std::vector<std::pair<long long, int>> factors;
    for (long long p = 2; p <= remaining / p; ++p) {
        if (remaining % p != 0) continue;
        int exponent = 0;
        while (remaining % p == 0) {
            remaining /= p;
            ++exponent;
        }
        factors.push_back({p, exponent});
    }
    if (remaining > 1) factors.push_back({remaining, 1});
    return factors;
}

int main() {
    long long n = 0;
    std::cin >> n;
    const auto factors = factorize(n);

    if (factors.empty()) std::cout << "EMPTY\n";
    for (const auto& [p, exponent] : factors) {
        std::cout << p << ' ' << exponent << '\n';
    }
}

```

## Sample input

360

## Sample output

2 3

3 2

5 1

## Pause and predict: 11

A prime list contains only 2, 3, 5. After trying it on 77, can the program automatically declare the leftover 77 prime?

**Check your prediction.** No. The list ended before reaching the necessary square-root coverage:  $7 \leq \sqrt{77}$ . The leftover proof requires exhausting all possible smaller prime divisors, not merely exhausting an arbitrary list.

## 5.12 Distinct prime factors, repeated factors, and divisors

### Three counts with different meanings

For  $72 = 2^3 \cdot 3^2$ , there are two distinct prime values, five prime occurrences, and twelve positive divisors. It helps to name these quantities separately:

$$\omega(n) = k, \quad \Omega(n) = e_1 + \cdots + e_k, \quad \tau(n) = \text{number of positive divisors}.$$

The Greek notation is optional vocabulary. The program names `distinct`, `total`, and `divisorCount` are often easier to read.

### Account for both kinds of factorization work

The candidate loop has at most  $\lfloor \sqrt{n} \rfloor$  iterations. Each successful division removes a factor at least 2, so there are at most  $\lfloor \log_2 n \rfloor$  successful divisions. The nested loops do *not* create  $O(\sqrt{n} \log n)$  work: successful divisions across all candidates share that one total budget.

The total worst-case time is  $O(\sqrt{n})$  for  $n \geq 2$ , plus output work. The repeated-factor list uses  $O(\Omega(n))$  entries; the grouped list uses  $O(\omega(n))$ . For 1, the work is constant and both lists are empty.

#### Keep the original value when later calculations need it

A helper that takes `remaining` by value may safely destroy its local copy. A program that mutates its only copy of `n` and later prints it, subtracts it from a sum, or checks an invariant will instead be using the final remainder. Preserve the original explicitly when needed.

**Implementation: two different prime-factor counts**

**Contract.** Read  $1 \leq n \leq 10^{12}$ . Print the number of distinct prime factors and the total number of prime factors with multiplicity.

**Complete example | C++23**

```
#include <iostream>
#include <vector>
#include <utility>

std::vector<std::pair<long long, int>> factorize(long long remaining) {
    std::vector<std::pair<long long, int>> factors;
    for (long long p = 2; p <= remaining / p; ++p) {
        if (remaining % p != 0) continue;
        int exponent = 0;
        while (remaining % p == 0) {
            remaining /= p;
            ++exponent;
        }
        factors.push_back({p, exponent});
    }
    if (remaining > 1) factors.push_back({remaining, 1});
    return factors;
}

int main() {
    long long n = 0;
    std::cin >> n;
    const auto factors = factorize(n);
    int total = 0;
    for (const auto& factor : factors) total += factor.second;
    std::cout << factors.size() << ' ' << total << '\n';
}
```

**Sample input****Sample output**

72

2 5

## 5.13 Count divisors by choosing prime exponents

### A divisor is a tuple of independent choices

Let  $n = p_1^{e_1} \cdots p_k^{e_k}$ . Every positive divisor has the form

$$d = p_1^{f_1} \cdots p_k^{f_k}, \quad 0 \leq f_i \leq e_i.$$

It cannot contain a new prime or more copies of a prime than  $n$  contains. Conversely, any such choices produce a divisor, because the remaining exponents  $e_i - f_i$  form the complementary divisor  $n/d$ .

There are  $e_i + 1$  choices for exponent  $f_i$ :  $0, 1, \dots, e_i$ . Uniqueness of prime factorization ensures that different exponent tuples give different divisors. Session 4's multiplication rule therefore gives

$$\tau(n) = (e_1 + 1)(e_2 + 1) \cdots (e_k + 1).$$

This standard divisor-count identity also follows from the prime-power rules in the number-theory reference.<sup>[5-2]</sup>

For  $72 = 2^3 \cdot 3^2$ , choose a power of 2 from 1, 2, 4, 8 and a power of 3 from 1, 3, 9. There are  $4 \cdot 3 = 12$  products. For 360, the count is  $4 \cdot 3 \cdot 2 = 24$ .

Factorization dominates the time:  $O(\sqrt{n})$ . The returned factor vector occupies  $O(\omega(n))$  storage. Initializing the answer to 1 gives the correct empty-product result for  $n = 1$ .

#### Pause and predict: 12

Why does  $2^4$  contribute 5 choices rather than 4 when counting divisors?

**Check your prediction.** A divisor may use zero copies of 2. The allowed exponents are 0, 1, 2, 3, 4, so there are five choices. Omitting exponent 0 would lose every divisor not divisible by 2, including 1.

**Implementation: count the exponent choices**

**Contract.** Read  $1 \leq n \leq 10^{12}$ . Print  $\tau(n)$ .

**Complete example | C++23**

```
#include <iostream>
#include <vector>
#include <utility>

std::vector<std::pair<long long, int>> factorize(long long remaining) {
    std::vector<std::pair<long long, int>> factors;
    for (long long p = 2; p <= remaining / p; ++p) {
        if (remaining % p != 0) continue;
        int exponent = 0;
        while (remaining % p == 0) {
            remaining /= p;
            ++exponent;
        }
        factors.push_back({p, exponent});
    }
    if (remaining > 1) factors.push_back({remaining, 1});
    return factors;
}

int main() {
    long long n = 0, answer = 1;
    std::cin >> n;
    for (const auto& factor : factorize(n)) {
        answer *= factor.second + 1;
    }
    std::cout << answer << '\n';
}
```

---

**Sample input**

---

**Sample output**

---

360

---

24

---

## 5.14 Sum divisors by multiplying prime-power sums

### Expand a product to see every divisor once

For  $n = p_1^{e_1} \cdots p_k^{e_k}$ , consider

$$\sigma(n) = \prod_{i=1}^k (1 + p_i + p_i^2 + \cdots + p_i^{e_i}).$$

Choosing one term from each factor produces exactly one divisor. Distributing the product therefore adds all divisors once. This is the sum analogue of the exponent-choice proof.<sup>[5-2]</sup>

For  $36 = 2^2 \cdot 3^2$ , the product is

$$(1 + 2 + 4)(1 + 3 + 9) = 7 \cdot 13 = 91.$$

The familiar geometric-series form  $(p^{e+1} - 1)/(p - 1)$  is mathematically exact, but computing  $p^{e+1}$  creates a larger intermediate than the required  $p^e$ . We avoid that unnecessary expression.

### Why the integer arithmetic is safe here

The variable `power` grows only through powers that actually divide  $n$ , so it never exceeds  $n$ . Each block is a sum of divisors. The product of the completed blocks is the divisor sum of a factor of  $n$ , so it is at most  $\sigma(n)$ . The earlier bound  $\sigma(n) \leq 2 \cdot 10^{18}$  proves all these positive accumulations fit under the stated input contract.

The calculation uses  $O(\sqrt{n} + \Omega(n)) = O(\sqrt{n})$  time for  $n \geq 2$  and  $O(\omega(n))$  stored factors. No floating-point `pow` is needed.

#### Pause and predict: 13

Why is the empty factorization of 1 consistent with a divisor sum of 1?

**Check your prediction.** The product of no prime-power blocks is initialized to 1. This equals the sum of the only positive divisor of 1, which is 1.

**Implementation: multiply prime-power sums**

**Contract.** Read  $1 \leq n \leq 10^{12}$ . Print  $\sigma(n)$ .

**Complete example | C++23**

```
#include <iostream>
#include <vector>
#include <utility>

std::vector<std::pair<long long, int>> factorize(long long remaining) {
    std::vector<std::pair<long long, int>> factors;
    for (long long p = 2; p <= remaining / p; ++p) {
        if (remaining % p != 0) continue;
        int exponent = 0;
        while (remaining % p == 0) {
            remaining /= p;
            ++exponent;
        }
        factors.push_back({p, exponent});
    }
    if (remaining > 1) factors.push_back({remaining, 1});
    return factors;
}

int main() {
    long long n = 0, answer = 1;
    std::cin >> n;
    for (const auto& [p, exponent] : factorize(n)) {
        long long power = 1, block = 1;
        for (int i = 0; i < exponent; ++i) {
            power *= p;
            block += power;
        }
        answer *= block;
    }
    std::cout << answer << '\n';
}
```

---

**Sample input**

---

**Sample output**

---

360

---

1170

---

## 5.15 Proper-divisor sums and square conditions

### Proper divisors exclude the number itself

Once the total divisor sum is known, the proper-divisor sum is  $\sigma(n) - n$ . A positive integer is **perfect** if this equals  $n$ , **deficient** if it is smaller, and **abundant** if it is larger. These names describe a comparison; do not confuse “perfect number” with “perfect square”.

For 28, the proper divisors are 1, 2, 4, 7, 14, whose sum is 28. For 12, the proper-divisor sum is 16, so 12 is abundant. For 1, it is 0, so 1 is deficient.

**Contract.** Read  $1 \leq n \leq 10^{12}$ . Print PERFECT, DEFICIENT, or ABUNDANT according to its proper-divisor sum.

**Complete example | C++23**

```
#include <iostream>

int main() {
    long long n = 0, sum = 0;
    std::cin >> n;
    for (long long d = 1; d <= n / d; ++d) {
        if (n % d != 0) continue;
        if (d < n) sum += d;
        long long other = n / d;
        if (other != d && other < n) sum += other;
    }
    if (sum == n) std::cout << "PERFECT\n";
    else if (sum < n) std::cout << "DEFICIENT\n";
    else std::cout << "ABUNDANT\n";
}
```

#### Sample input

28

#### Sample output

PERFECT

### Recognize squares through the factorization

A positive integer is a square exactly when every prime exponent is even. Squaring  $p_1^{a_1} \cdots p_k^{a_k}$  doubles each exponent; conversely, even exponents can all be halved to construct an integer square root. This includes 1 through the empty factorization.

The formula for  $\tau(n)$  gives a second explanation of the odd-divisor-count rule: a product of  $e_i + 1$  is odd exactly when every  $e_i$  is even.

#### Pause and predict: 14

Can a perfect number fail to be a perfect square? Give one example from this topic.

**Check your prediction.** Yes. The number 28 is perfect because its proper divisors sum to 28, but it is not a square. The words describe different properties.

## 5.16 Generate all divisors from the prime factorization

### Build the exponent combinations in stages

Start with the divisor list  $[1]$ . When introducing a new prime power  $p^e$ , retain every existing divisor and append that divisor multiplied by  $p, p^2, \dots, p^e$ .

For  $18 = 2 \cdot 3^2$ , the stages are

$$[1] \longrightarrow [1, 2] \longrightarrow [1, 2, 3, 6, 9, 18].$$

Before processing the power of 3, the old list has two entries. Use exactly those two old entries for both the multiplication by 3 and the multiplication by 9. Do not let newly appended entries become new starting points during that stage.

### The invariant that prevents duplicate construction

After processing the first  $j$  prime groups, the list contains exactly the divisors using those  $j$  primes with the allowed exponents. The next stage appends all positive choices for the new exponent and keeps the zero choice. Uniqueness of factorization makes the groups disjoint and complete.

Every created number divides the original  $n$ , so each multiplication in the construction is at most  $n$ . The loop saves `oldSize` before appending. This is about the mathematical stage boundary, not merely avoiding a C++ iterator issue.

#### Pause and predict: 15

Why is looping up to the vector's changing `size()` wrong while appending multiples of the new prime?

**Check your prediction.** It can process newly added divisors again, repeatedly adding more powers than the factorization permits. The stage must combine each allowed new power only with the divisors that existed before that prime was introduced.

The generation itself appends  $D - 1$  new values, so it uses  $O(D)$  amortized vector work. Sorting costs  $O(D \log(D + 1))$ . Including trial factorization gives  $O(\sqrt{n} + D \log(D + 1))$  time and  $O(D)$  stored output. The paired-list method is simpler when the only goal is a sorted divisor list; this method is useful when the factorization is already available or exponent choices are part of the problem.

**Contract.** Read  $1 \leq n \leq 10^{12}$ . Print all positive divisors in increasing order.

#### Sample input

18

#### Sample output

1 2 3 6 9 18

## Implementation: prime powers to a sorted divisor list

### Complete example | C++23

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>

std::vector<std::pair<long long, int>> factorize(long long remaining) {
    std::vector<std::pair<long long, int>> factors;
    for (long long p = 2; p <= remaining / p; ++p) {
        if (remaining % p != 0) continue;
        int exponent = 0;
        while (remaining % p == 0) {
            remaining /= p;
            ++exponent;
        }
        factors.push_back({p, exponent});
    }
    if (remaining > 1) factors.push_back({remaining, 1});
    return factors;
}

int main() {
    long long n = 0;
    std::cin >> n;
    std::vector<long long> result{1};
    for (const auto& [p, exponent] : factorize(n)) {
        int oldSize = static_cast<int>(result.size());
        long long power = 1;
        for (int e = 1; e <= exponent; ++e) {
            power *= p;
            for (int i = 0; i < oldSize; ++i) {
                result.push_back(result[i] * power);
            }
        }
    }
    std::sort(result.begin(), result.end());

    bool first = true;
    for (long long value : result) {
        if (!first) std::cout << ' ';
        std::cout << value;
        first = false;
    }
    if (first) std::cout << "EMPTY";
    std::cout << '\n';
}
```

## 5.17 Choose a precomputation from the query pattern

### The largest value is not the only constraint

A trial method is attractive for one number because it uses little memory and needs no preparation. With many bounded values, however, repeated trial divisions revisit the same candidates. A precomputed table trades memory and an initial build for cheaper queries.

| Need                                       | Useful preparation                | What a query reads               |
|--------------------------------------------|-----------------------------------|----------------------------------|
| One or a few divisor lists                 | Usually none.                     | A paired scan of that number.    |
| Many primality tests for $x \leq N$        | Prime-flag sieve up to $N$ .      | One flag at index $x$ .          |
| Many factorizations for $x \leq N$         | Smallest-prime-factor table.      | A succession of stored factors.  |
| A few much larger factorizations           | A prime list through $\sqrt{M}$ . | Trial divisions by those primes. |
| Many prime counts in value intervals       | Sieve and prime prefix counts.    | A difference of two totals.      |
| Divisor counts for every $1 \leq x \leq N$ | Divisor-multiple table.           | One stored count.                |

### A table index must fit the domain you actually allocated

A vector with entries  $0, 1, \dots, N$  needs  $N + 1$  elements. The last valid index is  $N$ , not  $N + 1$ . A table built through  $10^6$  cannot answer a direct lookup for  $10^9 + 7$ , even if that number fits in the variable holding the query.

For one input near  $10^{12}$ , a full table up to that input is not practical. A prime list up to  $10^6$  is a different object: it is used to *divide* the large input, not to index a flag at that input.

### Read all queries first only when it helps

If the input supplies a global limit, build to that limit. Otherwise, you may store a batch, find its maximum, and build once. That stores  $O(q)$  query values in addition to the table. If queries arrive online without a known limit, that same offline plan does not apply unchanged.

#### Pause and predict: 16

Does a sieve through 1000000 provide direct primality lookup for 1000000007?

**Check your prediction.** No. The index is outside the allocated table. The sieve's prime list can instead be used as trial divisors, with the square-root coverage argument, to test that larger number.

## 5.18 The sieve of Eratosthenes: cross off composites together

### One prime explains many composite numbers

Begin by treating every integer from 2 through  $N$  as a candidate prime. When a prime  $p$  is identified, its multiples  $2p, 3p, 4p, \dots$  are composite. Marking many multiples together avoids testing each number separately. This is the sieve of Eratosthenes.<sup>[5-3]</sup>

We choose the representation `prime[x] == 1` for a remaining candidate and 0 for a known nonprime. Other references may use the opposite flag or store a factor instead. An array name does not establish its meaning; the initialization and updates do.

### Why marking may begin at $p^2$

The multiples  $2p, 3p, \dots, (p-1)p$  already have a prime factor smaller than  $p$ . They were marked during earlier prime stages. Thus the first multiple that may still need work is  $p^2$ .

For  $N = 30$ , the significant stages are:

| $p$ | Start | Multiples visited from that start                   |
|-----|-------|-----------------------------------------------------|
| 2   | 4     | 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30 |
| 3   | 9     | 9, 12, 15, 18, 21, 24, 27, 30                       |
| 4   |       | Skip: already marked composite                      |
| 5   | 25    | 25, 30                                              |

Marking 12 more than once is harmless: setting an already-zero flag to zero does not change the answer. The optimization removes unnecessary early visits; it does not promise that each composite is visited only once.

### Why stopping after the square-root stage is enough

Every composite  $x \leq N$  has a prime divisor  $p \leq \sqrt{x} \leq \sqrt{N}$ . When that prime's stage runs,  $x$  is marked. No prime is marked because every marked value is a multiple  $p \cdot k$  with  $k \geq p \geq 2$ . Both directions are needed for correctness.

#### Pause and predict: 17

Why must we not start the marking loop at `multiple = p`?

**Check your prediction.** That would mark the prime itself composite. Starting at  $p^2$  keeps the prime and removes composite multiples. Starting at  $2p$  is correct but performs some redundant work.

## 5.19 Implement the prime-flag sieve safely

### Allocation and initialization are part of the algorithm

We allocate  $N + 1$  flags, initially 1. Index 0 exists even when  $N = 0$ , so clearing it is safe. Clear index 1 only when  $N \geq 1$ . The vector fill constructor initializes every flag to the supplied value.<sup>[5-7]</sup>

### Read the two loop roles separately

The outer loop decides which candidate primes need a marking stage. The inner loop walks multiples with step  $p$ , not step 1. Its starting expression `1LL * p * p` is evaluated in `long long`; the explicit input limit keeps subsequent additions and indexing safe.

The final scan goes all the way through  $N$ . Stopping the *marking stages* at  $\sqrt{N}$  does not mean there are no primes above  $\sqrt{N}$ . For example, 29 survives in the sieve through 30 and must still be printed.

#### Pause and predict: 18

What should a sieve built with  $N = 0$  or  $N = 1$  print?

**Check your prediction.** EMPTY in both cases. It must also avoid writing index 1 when the allocated vector has only index 0.

**Implementation: prime flags through a bounded limit**

**Contract.** Read  $0 \leq N \leq 2000000$ . Print all primes at most  $N$  in increasing order, or EMPTY if none exist.

**Complete example | C++23**

```
#include <iostream>
#include <vector>

std::vector<char> buildSieve(int limit) {
    std::vector<char> prime(limit + 1, 1);
    prime[0] = 0;
    if (limit >= 1) prime[1] = 0;
    for (int p = 2; p <= limit / p; ++p) {
        if (!prime[p]) continue;
        for (long long multiple = 1LL * p * p;
             multiple <= limit; multiple += p) {
            prime[multiple] = 0;
        }
    }
    return prime;
}

int main() {
    int limit = 0;
    std::cin >> limit;
    const auto prime = buildSieve(limit);
    bool first = true;
    for (int x = 2; x <= limit; ++x) {
        if (!prime[x]) continue;
        if (!first) std::cout << ' ';
        std::cout << x;
        first = false;
    }
    if (first) std::cout << "EMPTY";
    std::cout << '\n';
}
```

**Sample input**

30

**Sample output**

2 3 5 7 11 13 17 19 23 29

## 5.20 Sieve complexity: count visits, not just nested loops

### Not every outer value starts a full inner scan

A prime  $p$  visits at most about  $N/p$  multiples. Only primes start marking loops, so a useful upper expression is

$$O(N) + O\left(N \sum_{\substack{p \leq \sqrt{N} \\ p \text{ prime}}} \frac{1}{p}\right).$$

The standard resulting bound is  $O(N \log \log N)$  time for  $N \geq 3$  and  $O(N)$  flag storage. The prime-reciprocal estimate behind this bound is a number-theoretic result; it is not necessary to prove that estimate before implementing the sieve.<sup>[5-4]</sup>

A simpler bound that you *can* derive immediately includes all possible outer divisors:

$$\sum_{d=1}^N \left\lfloor \frac{N}{d} \right\rfloor \leq N \left(1 + \frac{1}{2} + \cdots + \frac{1}{N}\right) = O(N \log N).$$

This is a valid but looser upper bound for Eratosthenes. It becomes the relevant bound for a later algorithm that intentionally visits multiples of *every* divisor, including composites.

### What a primality query costs after the build

Once the table exists and  $0 \leq x \leq N$ , reading `prime[x]` costs  $O(1)$ . Answering  $q$  such questions costs  $O(N \log \log N + q)$  in total, not  $O(q)$  for the entire program. Small limits  $N < 3$  are handled in constant initialization time.

### Memory includes the complete stored representation

The implementation uses `vector<char>` for explicit 0/1 flags. In the tested environment, two million `char` entries occupy roughly two million bytes of element storage, before small container and allocator overhead. An SPF table uses `int` entries and therefore consumes more memory in that environment. Neither should be allocated from an unchecked arbitrary input.

#### Pause and predict: 19

Is it correct to describe the sieve as  $O(N^2)$  only because it contains two loops?

**Check your prediction.** No. The inner loop advances by  $p$ , and it runs only for surviving primes. Sum the number of visits over those primes; the product of two written loop bounds is not the actual work.

## 5.21 Prime-count intervals: combine a sieve with prefix sums

### This prefix is over values, not input positions

After building `prime`, define `prefix[x]` as the number of primes in the integer interval  $[0, x]$ . Since 0 is not prime, `prefix[0] = 0`. For  $x \geq 1$ ,

$$\text{prefix}[x] = \text{prefix}[x - 1] + \text{prime}[x].$$

The count in an inclusive value interval  $[L, R]$  is `prefix[R] - prefix[L-1]` when  $L > 0$ , and simply `prefix[R]` when  $L = 0$ .

### Why and how much work?

The prefix recurrence adds one exactly at each prime value. Subtracting the earlier prefix removes all values before  $L$  and retains every prime from  $L$  through  $R$ . Building both tables costs  $O(N \log \log N)$  for  $N \geq 3$ ; each query costs  $O(1)$ . Total storage is  $O(N)$ , including both flags and prefix entries.

#### Pause and predict: 20

Why is a query  $[0, 1]$  a useful boundary test?

**Check your prediction.** It should return 0 and must not access `prefix[-1]`. It tests both nonprime small values and the left-endpoint special case.

**Implementation: answer prime-count intervals**

**Contract.** Read  $0 \leq N \leq 2000000$  and  $0 \leq q \leq 200000$ , followed by  $q$  intervals with  $0 \leq L \leq R \leq N$ . Print the prime count for each interval.

**Complete example | C++23**

```
#include <iostream>
#include <vector>

std::vector<char> buildSieve(int limit) {
    std::vector<char> prime(limit + 1, 1);
    prime[0] = 0;
    if (limit >= 1) prime[1] = 0;
    for (int p = 2; p <= limit / p; ++p) {
        if (!prime[p]) continue;
        for (long long multiple = 1LL * p * p;
             multiple <= limit; multiple += p) {
            prime[multiple] = 0;
        }
    }
    return prime;
}

int main() {
    int limit = 0, q = 0;
    std::cin >> limit >> q;
    const auto prime = buildSieve(limit);
    std::vector<int> prefix(limit + 1, 0);
    for (int x = 1; x <= limit; ++x) {
        prefix[x] = prefix[x - 1] + prime[x];
    }
    while (q-- > 0) {
        int left = 0, right = 0;
        std::cin >> left >> right;
        int before = (left == 0 ? 0 : prefix[left - 1]);
        std::cout << prefix[right] - before << '\n';
    }
}
```

**Sample input****Sample output**

|       |   |
|-------|---|
| 20 3  | 0 |
| 0 1   | 4 |
| 2 10  | 4 |
| 11 20 |   |

## 5.22 Smallest prime factors: store a useful witness, not only a flag

### What one SPF entry means

For each  $x \geq 2$ , `spf[x]` will store the **smallest prime factor** of  $x$ . For a prime  $p$ , `spf[p] == p`. We reserve `spf[0] = spf[1] = 0` as sentinels; they are not valid prime factors to divide by.

Process candidates  $p$  in increasing order. If `spf[p]` is still 0, then  $p$  is prime; set `spf[p] = p`. Visit multiples from  $p^2$  and fill only entries still equal to 0. A later, larger prime must not overwrite a smaller factor already stored.

### Why the first assigned prime is the smallest

Let  $p$  be the smallest prime divisor of a composite  $x$ . Then  $x \geq p^2$ , so the marking stage for  $p$  reaches  $x$ . No earlier prime divides  $x$ , hence that entry has not yet been filled. The first assignment stores precisely  $p$ , and the guard preserves it afterwards.

This Eratosthenes-style SPF construction costs  $O(N \log \log N)$  time for  $N \geq 3$  and  $O(N)$  integer storage. A different linear construction is optional later; an SPF table by itself does not imply a particular construction complexity.

#### Pause and predict: 21

What happens if the condition `if (spf[multiple] == 0)` is removed?

**Check your prediction.** Later primes can overwrite earlier factors. The entry may remain a prime divisor, but it is no longer guaranteed to be the smallest. Any algorithm or explanation relying on smallest-first order would lose its contract.

**Implementation: construct smallest-prime-factor entries**

**Contract.** Read  $0 \leq N \leq 100000$ . Print `spf[1]` through `spf[N]`, using 0 for entry 1; print `EMPTY` when  $N = 0$ .

**Complete example | C++23**

```
#include <iostream>
#include <vector>

std::vector<int> buildSpf(int limit) {
    std::vector<int> spf(limit + 1, 0);
    for (int p = 2; p <= limit; ++p) {
        if (spf[p] != 0) continue;
        spf[p] = p;
        if (p > limit / p) continue;
        for (long long multiple = 1LL * p * p;
             multiple <= limit; multiple += p) {
            if (spf[multiple] == 0) spf[multiple] = p;
        }
    }
    return spf;
}

int main() {
    int limit = 0;
    std::cin >> limit;
    const auto spf = buildSpf(limit);
    for (int x = 1; x <= limit; ++x) {
        if (x > 1) std::cout << ' ';
        std::cout << spf[x];
    }
    if (limit == 0) std::cout << "EMPTY";
    std::cout << '\n';
}
```

**Sample input**

12

**Sample output**

0 2 3 2 5 2 7 2 3 2 11 2

## 5.23 Factor a bounded value by following SPF entries

### Each lookup removes one known prime occurrence

Given a valid SPF table and  $x \geq 2$ , read  $p = \text{spf}[x]$ , record  $p$ , then set  $x \leftarrow x/p$ . Repeat until  $x = 1$ . No new divisibility search is needed.

For  $x = 72$ , the chain is:

|             |    |    |    |   |   |
|-------------|----|----|----|---|---|
| Current $x$ | 72 | 36 | 18 | 9 | 3 |
| Stored SPF  | 2  | 2  | 2  | 3 | 3 |
| Next $x$    | 36 | 18 | 9  | 3 | 1 |

### Correctness, progress, and cost

Each stored factor is prime and divides the current value. The product-of-recorded-factors invariant is preserved. Dividing by at least 2 guarantees progress, so a query uses  $O(1 + \log x)$  time including the empty-factor case. The actual number of division steps is  $\Omega(x)$ , the number of prime occurrences.

Total time is  $O(N \log \log N + q + \sum \log x_i)$  for ordinary nontrivial limits. The table occupies  $O(N)$  storage. This implementation streams the factors of one query without retaining the entire batch or an extra result vector.

#### Pause and predict: 22

Why must the query loop stop at  $x > 1$  instead of  $x > 0$ ?

**Check your prediction.** The sentinel  $\text{spf}[1]$  is 0. Attempting to divide 1 by that entry would divide by zero. Reaching 1 means all prime occurrences have already been removed.

## Implementation: factor many bounded inputs

**Contract.** Read  $1 \leq N \leq 2000000$  and  $0 \leq q \leq 200000$ , then  $q$  numbers in  $[1, N]$ . Print each number's prime factors with multiplicity on its own line; print EMPTY for 1.

**Complete example | C++23**

```
#include <iostream>
#include <vector>

std::vector<int> buildSpf(int limit) {
    std::vector<int> spf(limit + 1, 0);
    for (int p = 2; p <= limit; ++p) {
        if (spf[p] != 0) continue;
        spf[p] = p;
        if (p > limit / p) continue;
        for (long long multiple = 1LL * p * p;
             multiple <= limit; multiple += p) {
            if (spf[multiple] == 0) spf[multiple] = p;
        }
    }
    return spf;
}

int main() {
    int limit = 0, q = 0;
    std::cin >> limit >> q;
    const auto spf = buildSpf(limit);
    while (q-- > 0) {
        int x = 0;
        std::cin >> x;
        bool first = true;
        while (x > 1) {
            int p = spf[x];
            if (!first) std::cout << ' ';
            std::cout << p;
            first = false;
            x /= p;
        }
        if (first) std::cout << "EMPTY";
        std::cout << '\n';
    }
}
```

### Sample input

30 3

1

12

29

### Sample output

EMPTY

2 2 3

29

## 5.24 Use precomputed primes to factor larger numbers

### A smaller table can support larger arithmetic inputs

Suppose a few inputs are at most  $M = 10^{12}$ . Any composite remainder has a prime factor at most  $\sqrt{M} = 10^6$ . Build a sieve through  $10^6$ , extract its primes, and try only those primes as divisors. The large input is never used as an index into that small table.

The remainder can shrink during the search. Stop when  $p > \text{remaining}/p$ , just as in ordinary trial factorization. The full prime-list coverage ensures that a final remainder greater than 1 is prime, even when no smaller factor divides the original input.

### What this improvement does and does not guarantee

Let  $B = 10^6$  and let  $\pi(B)$  mean the number of primes at most  $B$ . Preprocessing costs  $O(B \log \log B)$  time and  $O(B)$  storage, including the sieve flags. A conservative bound for the batch is

$$O\left(B \log \log B + q \pi(B) + \sum_i \log n_i\right).$$

The actual number of candidates often decreases with the shrinking remainder. This method avoids composite trial candidates, but it is still a trial-division algorithm. A huge batch of large prime inputs can remain expensive.

#### Pause and predict: 23

Why is the input promise  $n \leq 10^{12}$  important for a prime list ending at  $10^6$ ?

**Check your prediction.** It guarantees that the list covers every potentially necessary small prime divisor. A larger input could be a product of two primes above  $10^6$ , leaving a composite remainder after the list is exhausted.

**Contract.** Read  $0 \leq q \leq 50$ , then  $q$  values  $1 \leq n \leq 10^{12}$ . Print each factor list in increasing order with multiplicity, or EMPTY for 1.

| Sample input | Sample output |
|--------------|---------------|
| 3            | EMPTY         |
| 1            | 2 2 3 7       |
| 84           | 1000000007    |
| 1000000007   |               |

## Implementation: one shared prime list, several large inputs

### Complete example | C++23

```
#include <iostream>
#include <vector>

std::vector<char> buildSieve(int limit) {
    std::vector<char> prime(limit + 1, 1);
    prime[0] = 0;
    if (limit >= 1) prime[1] = 0;
    for (int p = 2; p <= limit / p; ++p) {
        if (!prime[p]) continue;
        for (long long multiple = 1LL * p * p;
             multiple <= limit; multiple += p) {
            prime[multiple] = 0;
        }
    }
    return prime;
}

int main() {
    const int limit = 1000000;
    const auto prime = buildSieve(limit);
    std::vector<int> primes;
    for (int x = 2; x <= limit; ++x) {
        if (prime[x]) primes.push_back(x);
    }
    int q = 0;
    std::cin >> q;
    while (q-- > 0) {
        long long remaining = 0;
        std::cin >> remaining;
        std::vector<long long> result;
        for (int p : primes) {
            if (p > remaining / p) break;
            while (remaining % p == 0) {
                result.push_back(p);
                remaining /= p;
            }
        }
        if (remaining > 1) result.push_back(remaining);
        bool first = true;
        for (long long value : result) {
            if (!first) std::cout << ' ';
            std::cout << value;
            first = false;
        }
        if (first) std::cout << "EMPTY";
        std::cout << '\n';
    }
}
```

## 5.25 Combine prime exponents without constructing a huge product

### Multiplication becomes addition of exponents

Consider  $12 \cdot 5 \cdot 18$ . The separate factorizations are  $2^2 \cdot 3$ ,  $5$ , and  $2 \cdot 3^2$ . Add matching exponents to obtain

$$12 \cdot 5 \cdot 18 = 2^3 \cdot 3^3 \cdot 5.$$

If a list contains thousands of numbers, its product may exceed every ordinary integer type. The prime-exponent representation can still remain small. The task can ask for the representation without ever asking you to evaluate the product.

### What the accumulator stores

After reading the first  $i$  values, `exponent[p]` equals the total number of copies of prime  $p$  in their product. Factor each new value through the SPF table and increment the entry for every removed prime occurrence. Equal input values contribute repeatedly; this is not a set of distinct inputs.

**Contract.** Read  $1 \leq N \leq 1000000$  and  $0 \leq m \leq 200000$ , followed by  $m$  values in  $[1, N]$ . Print the product's prime-exponent report in increasing prime order, or `EMPTY` when the product is 1.

## Complete example | C++23

```

#include <iostream>
#include <vector>

std::vector<int> buildSpf(int limit) {
    std::vector<int> spf(limit + 1, 0);
    for (int p = 2; p <= limit; ++p) {
        if (spf[p] != 0) continue;
        spf[p] = p;
        if (p > limit / p) continue;
        for (long long multiple = 1LL * p * p;
             multiple <= limit; multiple += p) {
            if (spf[multiple] == 0) spf[multiple] = p;
        }
    }
    return spf;
}

int main() {
    int limit = 0, count = 0;
    std::cin >> limit >> count;
    const auto spf = buildSpf(limit);
    std::vector<long long> exponent(limit + 1, 0);
    for (int i = 0; i < count; ++i) {
        int x = 0;
        std::cin >> x;
        while (x > 1) {
            ++exponent[spf[x]];
            x /= spf[x];
        }
    }
    bool any = false;
    for (int p = 2; p <= limit; ++p) {
        if (exponent[p] == 0) continue;
        std::cout << p << ' ' << exponent[p] << '\n';
        any = true;
    }
    if (!any) std::cout << "EMPTY\n";
}

```

## Sample input

## Sample output

|         |     |
|---------|-----|
| 20 3    | 2 3 |
| 12 5 18 | 3 3 |
|         | 5 1 |

The invariant follows by adding the exponents of each next factorization. Time is  $O(N \log \log N + m + \sum \log x_i)$  for  $N \geq 3$ , including the final domain scan. Storage is  $O(N)$ . The empty input product and a product of only ones both have an empty prime-factor report. The arithmetic that would reconstruct a huge product is deliberately not performed.

## 5.26 Divisor tables: reverse the direction and visit multiples

### One divisor contributes to every number it divides

To compute divisor counts for all values up to  $N$ , avoid factoring each value separately. Let each possible divisor  $d$  visit its positive multiples  $d, 2d, 3d, \dots$ . Increment the count at every visited number. To compute divisor sums too, add  $d$  at that same position.

### Why every divisor contributes once

An update from outer value  $d$  reaches  $x$  exactly when  $x$  is a positive multiple of  $d$ . Thus it contributes exactly once for each  $d \mid x$ , and never for a nondivisor. The table entry at index 0 is unused: its initialized zero does not represent the number of divisors of 0.

The number of updates is  $\sum_{d=1}^N \lfloor N/d \rfloor = O(N \log N)$ , with  $O(N)$  storage. Both prime and composite values of  $d$  are essential here, so the prime-only  $O(N \log \log N)$  sieve bound does not apply.

#### Pause and predict: 24

Why would skipping composite outer values of  $d$  make the divisor-count table wrong?

**Check your prediction.** Composite divisors also count. For  $x = 12$ , the divisors 4, 6, and 12 are composite. A prime-only outer loop would not count all six positive divisors.

**Implementation: all divisor counts and sums**

**Contract.** Read  $0 \leq N \leq 200000$ . For every  $x$  from 1 through  $N$ , print  $x$  `divisor_count` `divisor_sum`, one line per value. When  $N = 0$ , print no lines.

**Complete example | C++23**

```
#include <iostream>
#include <vector>

int main() {
    int limit = 0;
    std::cin >> limit;
    std::vector<int> count(limit + 1, 0);
    std::vector<long long> sum(limit + 1, 0);
    for (int d = 1; d <= limit; ++d) {
        for (int multiple = d; multiple <= limit; multiple += d) {
            ++count[multiple];
            sum[multiple] += d;
        }
    }
    for (int x = 1; x <= limit; ++x) {
        std::cout << x << ' ' << count[x] << ' ' << sum[x] << '\n';
    }
}
```

**Sample input**

6

**Sample output**

```
1 1 1
2 2 3
3 2 4
4 3 7
5 2 6
6 4 12
```

## 5.27 Reusable tables, fresh queries, and honest total complexity

### Separate immutable preparation from changing state

A sieve or SPF table depends on its limit, not on the current query. Build it once and reuse it across all queries within that limit. In contrast, a query's remaining value, factor list, exponent count, and answer accumulator must start fresh.

```
// Fragment after a valid shared SPF table has been built.
while (q-- > 0) {
    int x = 0;
    std::cin >> x;
    int total = 0;           // Fresh for this query.
    while (x > 1) {
        x /= spf[x];
        ++total;
    }
    std::cout << total << '\n';
}
```

Leaving `total` outside the query loop would accumulate across unrelated answers. Rebuilding the entire SPF table inside that loop would repeat the expensive preparation. Both mistakes ignore which information is shared and which is local.

### Four quantities to include in a complexity statement

Count the build, the queries, the output, and the largest simultaneously stored data. A method that prints every factor must pay for all those printed occurrences. A method that stores every query must include the  $O(q)$  query array. A constant-time lookup does not erase the earlier build cost.

If different test cases require different maximum values, rebuilding for each case costs the sum of their build costs. Building once to a known global maximum can be better, but only when the memory limit allows it and the table meaning is unchanged.

### Avoid copying the preparation into each helper

Pass a large read-only table by `const` reference. Passing it by value can copy  $O(N)$  entries for each call, turning a cheap query into a linear-time operation. This is the same function-parameter principle introduced in Level 0.

#### A reliable small regression batch

Run queries for 1, a prime, a prime square, and the same composite twice. For example: 1, 13, 49, 72, 72. The last two answers must agree, and the answer for 1 must not inherit factors from an earlier query.

## 5.28 Debug the mathematical meaning before changing the syntax

### Use counterexamples that target one assumption

| Mistake                                        | Test     | What the test reveals                           |
|------------------------------------------------|----------|-------------------------------------------------|
| Accept every positive number as prime.         | 1        | Prime numbers must exceed 1.                    |
| Use a strict square-root bound-ary.            | 49       | The only small divisor may be exactly the root. |
| Always count both divisor part-ners.           | 36       | The middle divisor must appear once.            |
| Remove one copy of each prime.                 | 12       | Repeated factors cannot be discarded.           |
| Forget the final leftover.                     | 13       | An original prime may be the entire leftover.   |
| Mark a prime starting at itself.               | 2        | The prime is incorrectly erased.                |
| Divide using <code>spf[1]</code> .             | 1        | The sentinel is not a factor.                   |
| Use a prefix at $L - 1$ without checking $L$ . | $[0, 1]$ | The expression requests a negative index.       |
| Assume a short prime list always suffices.     | 77       | The missing candidate 7 is necessary.           |

### Cross-check two correct but different methods

For small positive inputs, compare a paired divisor list with a direct scan. Compare the paired count and sum with the exponent formulas. Reconstruct a factorization's product and check that every recorded factor is prime. Compare prime flags with trial primality for a small complete domain.

These checks are stronger than repeating one sample, but they are not a proof for every possible input. Pair them with invariants: exact factor removal, square-root coverage, first-assigned smallest factors, and the meaning of each table entry.

### Keep the allowed input contract visible

Changing  $n \leq 10^{12}$  to  $n \leq 10^{18}$  affects runtime and arithmetic. Changing  $x \geq 1$  to allow 0 affects factorization itself. Extending a table query beyond  $N$  affects memory safety. A solution is a method *plus* its assumptions, not just a code block.

## 5.29 Optional extension: a linear smallest-prime-factor sieve

### Arrange for each composite to be assigned once

The ordinary SPF sieve may visit one composite from several primes. A **linear sieve** uses a prime list and a rule restricting which products are generated. It is an optional alternative, not a prerequisite for the earlier query methods.

Process  $x = 2, 3, \dots, N$ . If its SPF entry is 0, mark  $x$  prime by setting  $\text{spf}[x] = x$  and append it to the prime list. Then combine  $x$  only with primes  $p \leq \text{spf}[x]$ , stopping when  $px > N$ . Store  $\text{spf}[p \cdot x] = p$ .

### Why the product rule identifies the smallest prime

If  $p \leq \text{spf}[x]$ , no prime smaller than  $p$  divides  $x$ . Hence  $p$  is the smallest prime factor of  $px$ . Conversely, for any composite  $m$ , let  $p$  be its smallest prime factor and  $x = m/p$ . Every prime factor of  $x$  is at least  $p$ , so  $p \leq \text{spf}[x]$ . Also  $p \leq x < m$ , so  $p$  is available when  $x$  is processed. The pair  $(x, p)$  produces  $m$ .

Any permitted pair that produces  $m$  must use its smallest prime as the multiplier. Therefore the pair is unique. Every composite is assigned once, and each outer iteration has only constant additional stopping overhead. This gives  $O(N)$  time and  $O(N)$  storage.

#### Pause and predict: 25

Why does the inner loop stop after primes become greater than  $\text{spf}[x]$ ?

**Check your prediction.** Larger primes would no longer be the smallest prime factor of their product with  $x$ . Allowing those products would lose the unique-generation argument and introduce repeated assignments.

## Implementation: the optional linear construction

**Contract.** Read  $0 \leq N \leq 100000$ . Print SPF entries from 1 through  $N$ , with 0 at entry 1, or **EMPTY** when  $N = 0$ .

**Complete example | C++23**

```
#include <iostream>
#include <vector>

int main() {
    int limit = 0;
    std::cin >> limit;
    std::vector<int> spf(limit + 1, 0), primes;
    for (int x = 2; x <= limit; ++x) {
        if (spf[x] == 0) {
            spf[x] = x;
            primes.push_back(x);
        }
        for (int p : primes) {
            if (p > spf[x] || p > limit / x) break;
            spf[p * x] = p;
        }
    }
    for (int x = 1; x <= limit; ++x) {
        if (x > 1) std::cout << ' ';
        std::cout << spf[x];
    }
    if (limit == 0) std::cout << "EMPTY";
    std::cout << '\n';
}
```

### Sample input

12

### Sample output

0 2 3 2 5 2 7 2 3 2 11 2

The guard `p > limit / x` checks the product bound before evaluating `p * x`. Since  $x \geq 2$ , its division is valid. After the guard passes, the product is at most the stated limit and fits in `int` in the training environment. A linear asymptotic bound does not by itself promise the smallest wall-clock time on every input; the simpler Eratosthenes sieve remains a good default for prime flags.

## 5.30 Optional extension: a segmented sieve for a short distant interval

### Store only the interval you were asked about

Suppose the task asks for primes in  $[L, R]$ , where  $R$  is large but  $W = R - L + 1$  is small. A full array through  $R$  wastes memory on values outside the requested interval. Instead, build base primes through  $B = \lfloor \sqrt{R} \rfloor$  and keep only  $W$  interval flags.

The flag for value  $x$  is at index  $x - L$ . For example, when  $L = 90$ , entry 0 means 90, entry 7 means 97, and entry 20 means 110. The stored index is not the original value.

### Derive the first multiple to cross off

For each base prime  $p$ , the first multiple at least  $L$  is  $\lceil L/p \rceil p$ . But starting there might cross off the prime  $p$  itself when it lies in the interval. Start instead at

$$\max\left(p^2, \left\lceil \frac{L}{p} \right\rceil p\right).$$

Then mark that value and every  $p$ th value until  $R$ . Clear 0 and 1 separately when they lie in the segment.

For  $[90, 110]$ , prime 2 starts at 90, prime 3 starts at 90, prime 5 starts at 90, and prime 7 starts at 91. The survivors are 97, 101, 103, 107, and 109.

### Correctness and storage

Every composite  $x \leq R$  has a prime factor at most  $\sqrt{R}$ . Its base-prime marking stage reaches  $x$ , while a prime is never crossed off as a proper multiple. The interval flags therefore retain exactly the primes after clearing 0 and 1.

Memory is  $O(B+W)$ , not  $O(R)$ . For  $B \geq 3$ , a safe time bound is  $O(B \log \log B + W \log \log B + \pi(B))$ : build the base sieve, scan its domain, and sum the interval marking visits. The root-finding loop in this implementation costs  $O(B)$  and is included. Very small bounds require only constant setup plus the interval scan.

#### Pause and predict: 26

For the segment  $[2, 7]$ , why is  $\lceil L/2 \rceil \cdot 2$  alone the wrong marking start?

**Check your prediction.** It equals 2 and would erase the prime 2. Taking the maximum with  $2^2$  starts at 4, preserving 2 while crossing off its composite multiples.

The ceiling expression  $(\text{left} + p - 1) / p$  is safe here because  $L \leq 10^{12}$  and  $p \leq 10^6$ . It is not an overflow-proof ceiling formula for arbitrary 64-bit inputs. The expression  $\text{right} - \text{left} + 1$  is at most 100000 because of the separate width constraint.

**Contract.** Read  $0 \leq L \leq R \leq 10^{12}$  with  $R - L + 1 \leq 100000$ . Print all primes in  $[L, R]$  in increasing order, or EMPTY.

#### Sample input

90 110

#### Sample output

97 101 103 107 109

**Implementation: shifted indices and safe bounded arithmetic****Complete example | C++23**

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> buildSieve(int limit) {
    std::vector<char> prime(limit + 1, 1);
    prime[0] = 0;
    if (limit >= 1) prime[1] = 0;
    for (int p = 2; p <= limit / p; ++p) {
        if (!prime[p]) continue;
        for (long long multiple = 1LL * p * p;
             multiple <= limit; multiple += p) {
            prime[multiple] = 0;
        }
    }
    return prime;
}

std::vector<char> segmentPrimes(long long left, long long right) {
    int root = 0;
    while (root + 1LL <= right / (root + 1LL)) ++root;
    const auto base = buildSieve(root);
    std::vector<char> prime(right - left + 1, 1);
    for (int p = 2; p <= root; ++p) {
        if (!base[p]) continue;
        long long start = std::max(1LL * p * p,
                                   ((left + p - 1) / p) * p);
        for (long long x = start; x <= right; x += p) {
            prime[x - left] = 0;
        }
    }
    for (long long x = left; x <= right && x < 2; ++x) {
        prime[x - left] = 0;
    }
    return prime;
}

int main() {
    long long left = 0, right = 0;
    std::cin >> left >> right;
    const auto prime = segmentPrimes(left, right);
    bool first = true;
    for (long long x = left; x <= right; ++x) {
        if (!prime[x - left]) continue;
        if (!first) std::cout << ' ';
        std::cout << x;
        first = false;
    }
    if (first) std::cout << "EMPTY";
    std::cout << '\n';
}
```

## 5.31 Optional extension: prime exponents in factorials

### Count contributions without constructing the factorial

For prime  $p$ , let  $v_p(n!)$  be the number of copies of  $p$  in  $1 \cdot 2 \cdots n$ . Every multiple of  $p$  contributes at least one copy. Every multiple of  $p^2$  contributes a second, every multiple of  $p^3$  a third, and so on. Therefore

$$v_p(n!) = \left\lfloor \frac{n}{p} \right\rfloor + \left\lfloor \frac{n}{p^2} \right\rfloor + \left\lfloor \frac{n}{p^3} \right\rfloor + \cdots$$

The terms eventually become zero. For  $10!$  and  $p = 2$ , the layers contribute  $5 + 2 + 1 = 8$ : the factor 8 contributes to all three layers, while a factor such as 6 contributes only to the first.

Instead of constructing successive powers, divide the current  $n$  by  $p$  repeatedly. Each quotient is the next floor term. This avoids overflow from computing an unused large power.

**Contract.** Read  $0 \leq n \leq 10^{18}$  and a *promised prime*  $2 \leq p \leq 10^9$ . Print  $v_p(n!)$ .

**Complete example | C++23**

```
#include <iostream>

long long exponentInFactorial(long long n, long long p) {
    long long exponent = 0;
    while (n > 0) {
        n /= p;
        exponent += n;
    }
    return exponent;
}

int main() {
    long long n = 0, p = 0;
    std::cin >> n >> p;
    std::cout << exponentInFactorial(n, p) << '\n';
}
```

#### Sample input

10 2

#### Sample output

8

The loop takes  $O(1 + \log_p(n + 1))$  time and  $O(1)$  space. For  $p \geq 2$ , the sum is at most  $n$ , so it fits under the stated bound. The prime condition is supplied by the task, not checked by this helper. The empty factorial  $0! = 1$  has exponent 0.

#### Pause and predict: 27

Can you replace the promised prime  $p$  by composite 4 and interpret the same floor sum as the exponent of 4 in  $10!$ ?

**Check your prediction.** No. The floor sum would give 2, but  $v_2(10!) = 8$ , so  $4^4$  divides  $10!$ . The prime-contribution proof does not apply unchanged to a composite base.

## 5.32 Optional extension: the smallest multiplier that makes a square

### Repair odd exponents, leave even exponents alone

Find the smallest positive  $m$  such that  $nm$  is a perfect square. In the factorization of a square, every prime exponent is even. If  $n$  contains  $p^e$  with odd  $e$ ,  $m$  must contain an odd number of copies of  $p$ ; one copy is the smallest choice. If  $e$  is even, no additional copy is needed.

Thus the smallest multiplier is the product of the distinct primes whose exponents in  $n$  are odd. For  $72 = 2^3 \cdot 3^2$ , only the exponent of 2 is odd, so  $m = 2$  and  $72 \cdot 2 = 144$ .

### Why this is the minimum, not just one possible answer

Every valid multiplier must supply at least one copy of each odd-exponent prime. Their product therefore divides every valid multiplier. That product itself repairs all exponents, so it is the smallest positive valid choice.

The returned multiplier divides  $n$  and is at most  $n$ , so it fits in `long long`. However,  $nm$  can be as large as  $n^2$ ; the code deliberately does not compute it. Time is  $O(\sqrt{n})$  and stored factors use  $O(\omega(n))$  space. For an existing square, including 1, the answer is 1.

#### Pause and predict: 28

For square-free  $n$ , where every prime exponent is 1, what multiplier is needed? Must the program evaluate the resulting square?

**Check your prediction.** The multiplier is  $n$  itself. The result would be  $n^2$ , but only the multiplier is requested, so evaluating that potentially overflowing product is unnecessary.

**Implementation: complete odd prime exponents**

**Contract.** Read  $1 \leq n \leq 10^{12}$ . Print that smallest positive multiplier.

**Complete example | C++23**

```
#include <iostream>
#include <vector>
#include <utility>

std::vector<std::pair<long long, int>> factorize(long long remaining) {
    std::vector<std::pair<long long, int>> factors;
    for (long long p = 2; p <= remaining / p; ++p) {
        if (remaining % p != 0) continue;
        int exponent = 0;
        while (remaining % p == 0) {
            remaining /= p;
            ++exponent;
        }
        factors.push_back({p, exponent});
    }
    if (remaining > 1) factors.push_back({remaining, 1});
    return factors;
}

int main() {
    long long n = 0, multiplier = 1;
    std::cin >> n;
    for (const auto& [p, exponent] : factorize(n)) {
        if (exponent % 2 == 1) multiplier *= p;
    }
    std::cout << multiplier << '\n';
}
```

---

**Sample input**

---

**Sample output**

---

72

---

2

---

### 5.33 Practice problems: choose the information before the loop

Try the problems without copying the example programs. For each solution, write its input assumptions, explain why every requested object is counted or found exactly once, and give the total time and storage costs.

Problems 1 to 17 use the core material. Problems 18 to 20 are optional challenges. All input satisfies the printed bounds. Samples are illustrations, not a complete test set. A query batch of size 0 produces no query-answer lines unless a task explicitly requests an aggregate result.

#### 1. Which candidate values divide the number?

You are given a positive integer  $n$  and a list of  $k$  candidate divisor values. Count how many *positions* in the list contain a divisor of  $n$ . Equal candidate values in different positions count separately.

**Input and output.** Read  $n$  and  $k$ , then  $k$  values  $d_i$ . Bounds:  $1 \leq n, d_i \leq 10^{12}$  and  $0 \leq k \leq 200000$ . Print one integer.

| Sample input         | Sample output |
|----------------------|---------------|
| 36 6<br>1 4 5 6 36 6 | 5             |

Explained solution; hints appear before the solutions.

#### 2. Print the positive divisors

List every positive divisor of  $n$  exactly once, in increasing order. Include 1 and  $n$ .

**Input and output.** Read  $1 \leq n \leq 10^{12}$ . Print the list on one line.

| Sample input | Sample output |
|--------------|---------------|
| 45           | 1 3 5 9 15 45 |

Explained solution; hints appear before the solutions.

#### 3. The sum of proper divisors

Find the sum of the positive divisors of  $n$  that are strictly smaller than  $n$ .

**Input and output.** Read  $1 \leq n \leq 10^{12}$ . Print the exact sum, with 0 for  $n = 1$ .

| Sample input | Sample output |
|--------------|---------------|
| 12           | 16            |

Explained solution; hints appear before the solutions.

#### 4. Select the $k$ th positive divisor

Sort the positive divisors of  $n$  increasingly. Return the divisor in one-based position  $k$ , or print  $-1$  if the list has fewer than  $k$  entries.

**Input and output.** Read  $1 \leq n \leq 10^{12}$  and  $1 \leq k \leq 2000001$ . Print one integer.

| Sample input | Sample output |
|--------------|---------------|
| 36 5         | 6             |

Explained solution; hints appear before the solutions.

#### 5. The closest rectangle sides

A rectangle has positive integer side lengths  $a \leq b$  and area  $n$ . Choose the side lengths with the smallest difference  $b - a$ . Print the chosen pair.

**Input and output.** Read  $1 \leq n \leq 10^{12}$ . Print  $a$  and  $b$  on one line.

| Sample input | Sample output |
|--------------|---------------|
| 48           | 6 8           |

Explained solution; hints appear before the solutions.

#### 6. A small batch of primality questions

For each input number, decide whether it is prime. The values can be large; use a method that does not allocate a table indexed by every possible input value.

**Input and output.** Read  $0 \leq q \leq 200$ , then  $q$  integers  $0 \leq n_i \leq 10^9$ . Print YES or NO for each.

| Sample input | Sample output |
|--------------|---------------|
| 5            | NO            |
| 0            | NO            |
| 1            | YES           |
| 2            | NO            |
| 49           | YES           |
| 97           |               |

Explained solution; hints appear before the solutions.

## 7. Find the smallest prime divisor

Find the smallest prime divisor of  $n$ . The input is guaranteed to have at least one prime factor.

**Input and output.** Read  $2 \leq n \leq 10^{12}$ . Print the smallest prime divisor.

| Sample input | Sample output |
|--------------|---------------|
| 91           | 7             |

Explained solution; hints appear before the solutions.

## 8. Find the largest prime divisor

Find the largest prime divisor of  $n$ . If  $n$  has no prime divisors, print  $-1$ .

**Input and output.** Read  $1 \leq n \leq 10^{12}$ . Print one integer.

| Sample input | Sample output |
|--------------|---------------|
| 84           | 7             |

Explained solution; hints appear before the solutions.

## 9. A compact prime-factor report

Represent  $n$  as prime powers. Print one line **prime exponent** per distinct prime, in increasing prime order. Print **EMPTY** for 1.

**Input and output.** Read  $1 \leq n \leq 10^{12}$ . Exponents count all repeated occurrences.

| Sample input | Sample output     |
|--------------|-------------------|
| 756          | 2 2<br>3 3<br>7 1 |

Explained solution; hints appear before the solutions.

## 10. Divisor counts for several numbers

Answer several independent queries asking for the number of positive divisors of the given number.

**Input and output.** Read  $0 \leq q \leq 200$ , then  $q$  integers  $1 \leq n_i \leq 10^9$ . Print one divisor count per line.

| Sample input | Sample output |
|--------------|---------------|
| 4            | 1             |
| 1            | 2             |
| 13           | 9             |
| 36           | 24            |
| 360          |               |

Explained solution; hints appear before the solutions.

## 11. How many divisors are perfect squares?

Count the positive divisors of  $n$  that are perfect squares. Include the divisor 1.

**Input and output.** Read  $1 \leq n \leq 10^{12}$ . Print the exact count.

| Sample input | Sample output |
|--------------|---------------|
| 72           | 4             |

Explained solution; hints appear before the solutions.

## 12. Exactly three positive divisors

For each positive input, decide whether it has exactly three positive divisors.

**Input and output.** Read  $0 \leq q \leq 200000$ , then  $q$  values  $1 \leq n_i \leq 10^{12}$ . Print YES or NO per query.

| Sample input | Sample output |
|--------------|---------------|
| 5            | NO            |
| 1            | YES           |
| 4            | YES           |
| 9            | NO            |
| 16           | YES           |
| 49           |               |

Explained solution; hints appear before the solutions.

### 13. Many prime-count intervals

Count the primes in each inclusive interval of integer values. The domain does not change between queries.

**Input and output.** Read  $0 \leq N \leq 2000000$  and  $0 \leq q \leq 200000$ , then  $q$  intervals  $0 \leq L \leq R \leq N$ . Print one count per interval.

| Sample input | Sample output |
|--------------|---------------|
| 30 3         | 10            |
| 0 30         | 0             |
| 14 16        | 1             |
| 17 17        |               |

Explained solution; hints appear before the solutions.

### 14. Select a prime inside a bounded domain

Consider only primes at most  $N$ , in increasing order. For each one-based rank  $k$ , return the  $k$ th such prime or  $-1$  if it does not exist.

**Input and output.** Read  $0 \leq N \leq 2000000$  and  $0 \leq q \leq 200000$ , then  $q$  ranks  $1 \leq k \leq N + 1$ .

| Sample input | Sample output |
|--------------|---------------|
| 20 3         | 2             |
| 1            | 19            |
| 8            | -1            |
| 9            |               |

Explained solution; hints appear before the solutions.

### 15. Fast prime-factor statistics

For every query value, print the number of distinct prime factors and the total number of prime factors with multiplicity.

**Input and output.** Read  $1 \leq N \leq 2000000$  and  $0 \leq q \leq 200000$ , followed by  $q$  values  $1 \leq x \leq N$ . Print both counts on each line.

| Sample input | Sample output |
|--------------|---------------|
| 100 3        | 0 0           |
| 1            | 2 5           |
| 72           | 1 1           |
| 97           |               |

Explained solution; hints appear before the solutions.

## 16. Reuse all divisor counts

Prepare divisor counts for a bounded domain, then answer many queries from that same domain.

**Input and output.** Read  $1 \leq N \leq 200000$  and  $0 \leq q \leq 200000$ , followed by  $q$  values  $1 \leq x \leq N$ . Print the positive-divisor count of each.

| Sample input | Sample output |
|--------------|---------------|
| 12 3         | 1             |
| 1            | 4             |
| 8            | 6             |
| 12           |               |

Explained solution; hints appear before the solutions.

## 17. How many input values are divisible by d?

You have a list of positive integers. For each query divisor  $d$ , count how many positions contain a value divisible by  $d$ . Duplicates count separately.

**Input and output.** Read  $1 \leq M \leq 200000$ ,  $0 \leq n, q \leq 200000$ , then  $n$  values in  $[1, M]$ , then  $q$  divisors in  $[1, M]$ . Print one count per query.

| Sample input | Sample output |
|--------------|---------------|
| 12 5 3       | 4             |
| 2 6 6 9 12   | 4             |
| 2            | 0             |
| 3            |               |
| 5            |               |

Explained solution; hints appear before the solutions.

## 18. Complete the missing square factors

**Optional challenge.** Find the smallest positive integer  $m$  such that  $nm$  is a perfect square. Print  $m$ , not the square itself.

**Input and output.** Read  $1 \leq n \leq 10^{12}$ . Print the exact multiplier.

| Sample input | Sample output |
|--------------|---------------|
| 180          | 5             |

Explained solution; hints appear before the solutions.

## 19. Count and sum primes in a distant interval

**Optional challenge.** Count the primes in a possibly distant interval and compute their exact sum.

**Input and output.** Read  $0 \leq L \leq R \leq 10^{12}$  with  $R - L + 1 \leq 100000$ . Print the count and the sum on one line. If there are no primes, print 0 0.

| Sample input | Sample output |
|--------------|---------------|
| 90 110       | 5 517         |

Explained solution; hints appear before the solutions.

## 20. A prime exponent inside a binomial coefficient

**Optional challenge.** Given a prime  $p$ , find the largest integer  $e \geq 0$  such that  $p^e$  divides  $\binom{n}{r}$ . The binomial coefficient itself may be far too large to store.

**Input and output.** Read  $0 \leq r \leq n \leq 10^{18}$  and a promised prime  $2 \leq p \leq 10^9$ . Print  $e$ .

| Sample input | Sample output |
|--------------|---------------|
| 10 3 2       | 3             |

Explained solution; hints appear before the solutions.

## 5.34 Trace, correctness, and complexity drills

For each drill, explain the reasoning as well as the final result. Use paper first. A code fragment is not a complete program, and fragments explicitly marked wrong are intentionally incorrect.

### Drill 1: One number, several different answers

For  $n = 36$ , write the positive-divisor list and determine its divisor count, divisor sum, proper-divisor sum, number of distinct prime factors, and total prime-factor multiplicity. Explain why these are different questions.

### Drill 2: A boundary error hidden inside a short loop

The following *wrong* fragment tries to count divisors of positive  $n$ :

```
long long count = 0;
for (long long d = 1; d * d < n; ++d) {
    if (n % d == 0) count += 2;
}
```

What does it do for  $n = 1$  and  $n = 4$ ? Identify the endpoint mistake, the duplicate-pair issue that a partial fix could introduce, and the arithmetic risk for much larger bounds. Write a corrected fragment.

### Drill 3: Trace the remaining value, not the original boundary

Trace grouped trial factorization of 84. For each successful candidate, record the before value, exponent removed, and after value. What is the final leftover? State the product invariant after removing the copies of 2.

### Drill 4: Two different incomplete factorizations

In one version, the repeated-division `while` is replaced by a single `if`; in a second version, the final `if (remaining > 1)` is deleted. Keep the shrinking square-root condition in both versions.

Trace the first version on 12. Trace the second on 13. Explain why the leftover-prime proof no longer applies to the first version, and what the second version forgets to output.

**Drill 5: Exponent choices instead of a long divisor list**

You are given  $1080 = 2^3 \cdot 3^3 \cdot 5$ . Determine its divisor count, divisor sum, and the number of its divisors that are perfect squares. For each calculation, describe what one choice represents before substituting numbers.

**Drill 6: Which sieve entries can change?**

A sieve through 30 has finished the marking stage for 2. What starting value should it use for prime 3? Why can 6 be skipped as a new starting point? Is marking 12 again an error? Explain what goes wrong if every prime starts marking at itself.

**Drill 7: Follow an SPF chain**

Write the successive values and removed factors when an SPF query factors 60. Give the distinct-factor count and total multiplicity. Why is `spf[1]` not a value that can be used as the next divisor? What property would be lost by overwriting assigned entries with later primes during the build?

**Drill 8: Choose a representation for the workload**

Choose a method for each task and state the preparation it needs: one sorted divisor list for  $n \leq 10^{12}$ ; 200000 primality queries with  $x \leq 10^6$ ; 200000 full factorizations with  $x \leq 10^6$ ; and many prime counts in intervals  $[L, R] \subseteq [0, 10^6]$ .

Explain why “all of them are number theory” is not enough to justify using the same table in all four tasks.

**Drill 9: The harmonic double loop**

Consider the following fragment, where  $1 \leq N \leq 200000$  and `visits` is wide enough:

```
for (int d = 1; d <= N; ++d) {
    for (int multiple = d; multiple <= N; multiple += d) {
        ++visits;
    }
}
```

How many inner-body executions occur for  $N = 6$ ? Give the general sum and asymptotic bound. Why is this not the prime-only sieve bound? Relate the total visits to divisor counts for the entire domain.

**Drill 10: An exhausted list is not a proof**

A helper divides by the prime list  $[2, 3, 5]$ , then appends the remaining value if it is greater than 1. Apply it to 77. Which input-bound guarantee would make such a leftover rule valid? Separately, why can a flag table through 1000000 not be indexed at 1000000007?

**Drill 11: Optional: shifted indices and prime-preserving starts**

For a segmented sieve on  $[0, 10]$ , give the marking starts for base primes 2 and 3, the values that must be cleared separately, and the surviving primes. Then analyze the singleton interval  $[49, 49]$ . Explain both the offset index and the role of the  $p^2$  lower bound.

**Drill 12: Optional: read the exponent without storing the number**

Compute  $v_2(10!)$ ,  $v_2(3!)$ , and  $v_2(7!)$ , then obtain the exponent of 2 in  $\binom{10}{3}$ . Independently check the answer using the small exact binomial coefficient. Why does the method still work when the coefficient itself is too large to store? What happens when  $r = 0$ ?

## 5.35 Hints before solutions

A hint should help you take the next step, not replace the entire attempt. Read only the hint for the problem you are currently working on. The complete solutions begin later.

### Hints for coding problems

- 1. Which candidate values divide the number?** Each candidate already tells you which divisor to test. Do not enumerate all divisors of  $n$ .
- 2. Print the positive divisors.** Discover a small divisor and its large partner together. The large partners arrive in decreasing order.
- 3. The sum of proper divisors.** A paired scan can compute the sum of all divisors. Which one contribution should then be removed?
- 4. Select the  $k$ th positive divisor.** Build the increasing divisor list, then translate the requested one-based position to an index only after checking that it exists.
- 5. The closest rectangle sides.** Only the smaller side needs to be tested, and it cannot exceed  $\sqrt{n}$ . Which valid smaller side makes the gap smallest?
- 6. A small batch of primality questions.** Reuse the trial-primality helper, including its separate cases for numbers below 2 and for even values.
- 7. Find the smallest prime divisor.** The first divisor at least 2 discovered in an increasing scan must itself be prime.
- 8. Find the largest prime divisor.** A complete grouped factorization is already ordered by prime. Check the empty case before reading its last entry.
- 9. A compact prime-factor report.** Reset an exponent counter when a new candidate succeeds, then repeatedly divide by that same candidate.
- 10. Divisor counts for several numbers.** Factor each input, multiply one more than each exponent, and initialize the product again for every query.

**Coding hints, continued**

- 11. How many divisors are perfect squares?** For a prime exponent  $e$ , which allowed divisor exponents in  $0, 1, \dots, e$  are even?
- 12. Exactly three positive divisors.** A product of positive integers  $e_i + 1$  equals 3 only in one prime-exponent pattern. Test the resulting shape using an exact integer square root.
- 13. Many prime-count intervals.** Build one prime-flag sieve and one prefix of those flags. Handle  $L = 0$  before reading an earlier prefix.
- 14. Select a prime inside a bounded domain.** Extract the surviving primes from the sieve into a sorted vector. The question is about that vector, not about the  $k$ th table index.
- 15. Fast prime-factor statistics.** An SPF table turns factorization into successive lookups. Group consecutive divisions by the same prime.
- 16. Reuse all divisor counts.** Let every positive divisor visit all of its multiples once. Include composite divisors in the outer loop.
- 17. How many input values are divisible by  $d$ ?** Build a frequency array of the data. For each  $d$ , add frequencies at  $d, 2d, 3d, \dots$
- 18. Complete the missing square factors.** Every prime exponent in the final product must be even. Supply the smallest correction for each odd exponent.
- 19. Count and sum primes in a distant interval.** Use base primes through  $\sqrt{R}$ , interval indices  $x - L$ , and a marking start that does not erase a base prime itself.
- 20. A prime exponent inside a binomial coefficient.** Use  $\binom{n}{r} = n! / (r!(n - r)!)$ . Multiplying adds prime exponents, so an exact quotient subtracts them.

## Hints for the trace and reasoning drills

1. Factor 36 as  $2^2 \cdot 3^2$ , but list all of its positive divisors separately before comparing the six requested quantities.
2. A strict comparison loses the square-root divisor. Including the boundary then requires distinguishing equal from unequal partners. Check arithmetic before the comparison as well.
3. Remove every copy of 2, then every copy of 3. Check whether the next candidate is still at most the square root of the current remainder.
4. After removing one 2 from 12, the remainder is 6. For the original prime 13, no successful trial division records any factor at all.
5. Ordinary divisor exponents range from 0 through the available exponent. Square-divisor exponents must also be even. For a sum, distribute prime-power blocks.
6. The prime 3 should remain prime. Earlier smaller-prime stages have already handled its early composite multiples.
7. The sequence of removed smallest factors is ordered, with repeated occurrences next to one another. The sentinel marks completion, not a factorization step.
8. Decide whether the query needs a list, a flag, a prime factor, or an interval total. Then compare the domain size with the one-time preparation cost.
9. Outer value  $d$  has  $\lfloor N/d \rfloor$  multiples. For  $N = 6$ , add  $6 + 3 + 2 + 1 + 1 + 1$ .
10. A composite remainder can remain when a necessary small prime was never tried. Compare the list's coverage with the square root of the largest allowed input.
11. Value  $x$  belongs to index  $x - L$ . Starting at the first interval multiple alone may erase the base prime itself.
12. Multiplication adds prime exponents; the exact factorial quotient subtracts them. A coefficient equal to 1 has no prime factors.

## 5.36 Explained practice solutions

Each solution is one complete C++23 program. Read the idea and its justification, then close the book and rebuild the solution yourself. The code targets the problem's stated bounds; changing those bounds may change the required method or arithmetic.

### Solution 1: Which candidate values divide the number?

**Idea.** Read each candidate and increment the answer when its remainder test is zero. No sieve or factorization is needed.

**Complete solution | C++23**

```
#include <iostream>

int main() {
    long long n = 0;
    int k = 0, answer = 0;
    std::cin >> n >> k;
    for (int i = 0; i < k; ++i) {
        long long d = 0;
        std::cin >> d;
        if (n % d == 0) ++answer;
    }
    std::cout << answer << '\n';
}
```

**Why it works.** The condition  $n \bmod d_i = 0$  is true exactly for valid candidates. Each input position is visited once, so repeated values are counted with the requested multiplicity.

**Complexity.**  $O(k + 1)$  time and  $O(1)$  extra space. Candidates are processed as they are read.

**Boundary checks.** An empty list gives 0. Candidate 1 always divides positive  $n$ , and a candidate greater than  $n$  cannot divide it.

## Solution 2: Print the positive divisors

**Idea.** Use the paired-list helper: collect small divisors in increasing order and distinct large partners in decreasing order, then append the large list backwards.

**Complete solution | C++23**

```
#include <iostream>
#include <vector>

std::vector<long long> divisors(long long n) {
    std::vector<long long> small, large;
    for (long long d = 1; d <= n / d; ++d) {
        if (n % d != 0) continue;
        small.push_back(d);
        if (d != n / d) large.push_back(n / d);
    }
    for (int i = static_cast<int>(large.size()) - 1; i >= 0; --i) {
        small.push_back(large[i]);
    }
    return small;
}

int main() {
    long long n = 0;
    std::cin >> n;
    const auto result = divisors(n);

    bool first = true;
    for (long long value : result) {
        if (!first) std::cout << ' ';
        std::cout << value;
        first = false;
    }
    if (first) std::cout << "EMPTY";
    std::cout << '\n';
}
```

**Why it works.** Every divisor pair has a smaller member at most  $\sqrt{n}$ , so the loop discovers every pair. The equality guard prevents a duplicate square root. The ordering of the two lists yields increasing output.

**Complexity.**  $O(\sqrt{n} + D)$  time and  $O(D)$  storage for  $D$  positive divisors.

**Boundary checks.** For 1, print just 1. A prime gives two entries. A square's root appears exactly once.

### Solution 3: The sum of proper divisors

**Idea.** Sum every divisor pair, counting an equal pair once, then subtract the original  $n$ .

**Complete solution | C++23**

```
#include <iostream>

int main() {
    long long n = 0, total = 0;
    std::cin >> n;
    for (long long d = 1; d <= n / d; ++d) {
        if (n % d != 0) continue;
        total += d;
        if (d != n / d) total += n / d;
    }
    std::cout << total - n << '\n';
}
```

**Why it works.** The paired scan sums each positive divisor once. The only divisor that is not proper is  $n$ , so removing it leaves exactly the requested sum. The original value is not changed during the scan.

**Complexity.**  $O(\sqrt{n})$  time and  $O(1)$  extra space. The chapter's  $2 \cdot 10^{18}$  bound protects the total.

**Boundary checks.** The answer for 1 is 0, not 1. Do not subtract a destroyed factorization remainder instead of the original number.

## Solution 4: Select the $k$ th positive divisor

**Idea.** Generate the sorted list with paired vectors. If  $k$  exceeds its size, report failure; otherwise read entry  $k - 1$ .

**Complete solution | C++23**

```
#include <iostream>
#include <vector>

std::vector<long long> divisors(long long n) {
    std::vector<long long> small, large;
    for (long long d = 1; d <= n / d; ++d) {
        if (n % d != 0) continue;
        small.push_back(d);
        if (d != n / d) large.push_back(n / d);
    }
    for (int i = static_cast<int>(large.size()) - 1; i >= 0; --i) {
        small.push_back(large[i]);
    }
    return small;
}

int main() {
    long long n = 0;
    int k = 0;
    std::cin >> n >> k;
    const auto all = divisors(n);
    if (k > static_cast<int>(all.size())) std::cout << -1 << '\n';
    else std::cout << all[k - 1] << '\n';
}
```

**Why it works.** The helper returns every divisor in increasing order. Position  $k$  therefore corresponds exactly to zero-based index  $k - 1$ . The size check excludes every invalid access.

**Complexity.**  $O(\sqrt{n} + D)$  time and  $O(D)$  storage.

**Boundary checks.** Test the first divisor, last divisor, and one past the end. For  $n = 1$ , only  $k = 1$  succeeds.

## Solution 5: The closest rectangle sides

**Idea.** Scan all possible smaller divisors up to the square root and retain the greatest one. Its partner is  $n/a$ .

**Complete solution | C++23**

```
#include <iostream>

int main() {
    long long area = 0, best = 1;
    std::cin >> area;
    for (long long side = 1; side <= area / side; ++side) {
        if (area % side == 0) best = side;
    }
    std::cout << best << ' ' << area / best << '\n';
}
```

**Why it works.** Every eligible rectangle corresponds to exactly one divisor  $a \leq \sqrt{n}$ . As positive  $a$  increases,  $n/a$  decreases and  $a$  increases, so the gap  $n/a - a$  decreases. The greatest eligible  $a$  therefore minimizes the gap.

**Complexity.**  $O(\sqrt{n})$  time and  $O(1)$  space.

**Boundary checks.** A square returns equal sides. A prime returns  $1, n$ . The area 1 returns  $1, 1$ .

## Solution 6: A small batch of primality questions

**Idea.** Call the odd-candidate primality test for each query. The batch bound makes a separate square-root search a reasonable method here.

**Complete solution | C++23**

```
#include <iostream>

bool isPrime(long long n) {
    if (n < 2) return false;
    if (n % 2 == 0) return n == 2;
    for (long long d = 3; d <= n / d; d += 2) {
        if (n % d == 0) return false;
    }
    return true;
}

int main() {
    int q = 0;
    std::cin >> q;
    while (q-- > 0) {
        long long n = 0;
        std::cin >> n;
        std::cout << (isPrime(n) ? "YES" : "NO") << '\n';
    }
}
```

**Why it works.** The helper rejects nonprimes below 2 and all even composites. Any remaining composite has an odd divisor at most its square root, which the loop tests. If no such divisor exists, the number is prime.

**Complexity.**  $O(q + \sum_i \sqrt{n_i})$  worst-case time and  $O(1)$  extra space. Empty batches print no lines.

**Boundary checks.** Check 0, 1, 2, 3, odd squares, and a large prime. An odd input is not automatically prime.

## Solution 7: Find the smallest prime divisor

**Idea.** Return the first divisor between 2 and the square-root boundary. If no candidate divides, return  $n$ .

**Complete solution | C++23**

```
#include <iostream>

int main() {
    long long n = 0;
    std::cin >> n;
    for (long long d = 2; d <= n / d; ++d) {
        if (n % d == 0) {
            std::cout << d << '\n';
            return 0;
        }
    }
    std::cout << n << '\n';
}
```

**Why it works.** If the first divisor were composite, one of its smaller prime factors would also divide  $n$ , contradicting that it was first. If no divisor through the square root exists,  $n$  is prime and is its own smallest prime divisor.

**Complexity.**  $O(\sqrt{n})$  worst-case time and  $O(1)$  extra space.

**Boundary checks.** The contract excludes 1. For 49, the boundary divisor 7 must be included. For a prime, the fallback returns the input.

## Solution 8: Find the largest prime divisor

**Idea.** Factor the number with repeated division and the final-leftover rule. Read the prime in the final group, or report  $-1$  for an empty list.

**Complete solution | C++23**

```
#include <iostream>
#include <vector>
#include <utility>

std::vector<std::pair<long long, int>> factorize(long long remaining) {
    std::vector<std::pair<long long, int>> factors;
    for (long long p = 2; p <= remaining / p; ++p) {
        if (remaining % p != 0) continue;
        int exponent = 0;
        while (remaining % p == 0) {
            remaining /= p;
            ++exponent;
        }
        factors.push_back({p, exponent});
    }
    if (remaining > 1) factors.push_back({remaining, 1});
    return factors;
}

int main() {
    long long n = 0;
    std::cin >> n;
    const auto factors = factorize(n);
    if (factors.empty()) std::cout << -1 << '\n';
    else std::cout << factors.back().first << '\n';
}
```

**Why it works.** All groups are prime, every prime divisor appears, and the groups are in increasing prime order. The final group's prime is therefore the largest. Only 1 has an empty factorization in this input domain.

**Complexity.**  $O(\sqrt{n})$  time and  $O(\omega(n))$  stored factor groups.

**Boundary checks.** Do not confuse the largest divisor  $n$  with the largest prime divisor. Test 1, a prime power, and a number with a large leftover prime.

## Solution 9: A compact prime-factor report

**Idea.** Use a grouped trial factorization. Each successful candidate receives the number of exact divisions by it; a final remainder greater than 1 receives exponent 1.

**Complete solution | C++23**

```
#include <iostream>
#include <vector>
#include <utility>

std::vector<std::pair<long long, int>> factorize(long long remaining) {
    std::vector<std::pair<long long, int>> factors;
    for (long long p = 2; p <= remaining / p; ++p) {
        if (remaining % p != 0) continue;
        int exponent = 0;
        while (remaining % p == 0) {
            remaining /= p;
            ++exponent;
        }
        factors.push_back({p, exponent});
    }
    if (remaining > 1) factors.push_back({remaining, 1});
    return factors;
}

int main() {
    long long n = 0;
    std::cin >> n;
    const auto factors = factorize(n);

    if (factors.empty()) std::cout << "EMPTY\n";
    for (const auto& [p, exponent] : factors) {
        std::cout << p << ' ' << exponent << '\n';
    }
}
```

**Why it works.** Repeated division removes exactly all copies of the current prime. The remaining-factor invariant preserves the product. The square-root stopping argument proves the final leftover prime, so the report is complete and canonical.

**Complexity.**  $O(\sqrt{n} + \omega(n))$  time including the report and  $O(\omega(n))$  storage; constant work for 1.

**Boundary checks.** A prime has exponent 1. Equal prime occurrences belong in one group. Neither 0 nor 1 is a prime group.

**Solution 10: Divisor counts for several numbers**

**Idea.** Compute the grouped factorization and multiply the factors  $e + 1$ . Keep the running answer inside the query loop.

**Complete solution | C++23**

```
#include <iostream>
#include <vector>
#include <utility>

std::vector<std::pair<long long, int>> factorize(long long remaining) {
    std::vector<std::pair<long long, int>> factors;
    for (long long p = 2; p <= remaining / p; ++p) {
        if (remaining % p != 0) continue;
        int exponent = 0;
        while (remaining % p == 0) {
            remaining /= p;
            ++exponent;
        }
        factors.push_back({p, exponent});
    }
    if (remaining > 1) factors.push_back({remaining, 1});
    return factors;
}

int main() {
    int q = 0;
    std::cin >> q;
    while (q-- > 0) {
        long long n = 0, answer = 1;
        std::cin >> n;
        for (const auto& factor : factorize(n)) {
            answer *= factor.second + 1;
        }
        std::cout << answer << '\n';
    }
}
```

**Why it works.** Each positive divisor chooses an exponent from 0 through  $e$  for every prime. The multiplication rule counts all tuples exactly once. Starting from 1 also counts the single divisor of 1.

**Complexity.**  $O(q + \sum_i \sqrt{n_i})$  time. The largest temporary factor list uses  $O(1 + \log M)$  entries as a loose bound when all  $n_i \leq M$ .

**Boundary checks.** Repeated inputs must give repeated identical outputs, not a cumulative product. Test 1, 13, 36, 360.

**Solution 11: How many divisors are perfect squares?**

**Idea.** Factor  $n$ . For each exponent  $e$ , multiply the answer by  $\lfloor e/2 \rfloor + 1$ .

**Complete solution | C++23**

```
#include <iostream>
#include <vector>
#include <utility>

std::vector<std::pair<long long, int>> factorize(long long remaining) {
    std::vector<std::pair<long long, int>> factors;
    for (long long p = 2; p <= remaining / p; ++p) {
        if (remaining % p != 0) continue;
        int exponent = 0;
        while (remaining % p == 0) {
            remaining /= p;
            ++exponent;
        }
        factors.push_back({p, exponent});
    }
    if (remaining > 1) factors.push_back({remaining, 1});
    return factors;
}

int main() {
    long long n = 0, answer = 1;
    std::cin >> n;
    for (const auto& factor : factorize(n)) {
        answer *= factor.second / 2 + 1;
    }
    std::cout << answer << '\n';
}
```

**Why it works.** A divisor is square exactly when each selected prime exponent is even. There are  $0, 2, 4, \dots, 2\lfloor e/2 \rfloor$ , giving  $\lfloor e/2 \rfloor + 1$  independent choices for that prime. Unique exponent tuples describe unique square divisors.

**Complexity.**  $O(\sqrt{n})$  time and  $O(\omega(n))$  stored factors.

**Boundary checks.** For 72, the square divisors are 1, 4, 9, 36. For a prime, only 1 qualifies; for  $n = 1$ , the answer is also 1.

## Solution 12: Exactly three positive divisors

**Idea.** Exactly three divisors means  $n = p^2$  for a prime  $p$ . Sieve through  $10^6$  once. For each query, use integer binary search to find its floor square root; check exact squareness and the root's prime flag.

**Complete solution | C++23**

```
#include <iostream>
#include <vector>

std::vector<char> buildSieve(int limit) {
    std::vector<char> prime(limit + 1, 1);
    prime[0] = 0;
    if (limit >= 1) prime[1] = 0;
    for (int p = 2; p <= limit / p; ++p) {
        if (!prime[p]) continue;
        for (long long multiple = 1LL * p * p;
             multiple <= limit; multiple += p) {
            prime[multiple] = 0;
        }
    }
    return prime;
}

long long integerRoot(long long n) {
    long long low = 0, high = 1000000;
    while (low < high) {
        long long mid = low + (high - low + 1) / 2;
        if (mid <= n / mid) low = mid;
        else high = mid - 1;
    }
    return low;
}

int main() {
    const auto prime = buildSieve(1000000);
    int q = 0;
    std::cin >> q;
    while (q-- > 0) {
        long long n = 0;
        std::cin >> n;
        long long root = integerRoot(n);
        bool answer = root * root == n && prime[root];
        std::cout << (answer ? "YES" : "NO") << '\n';
    }
}
```

**Why it works.** By the divisor-count formula, a count of 3 requires one prime group with exponent 2. Conversely,  $p^2$  has divisors  $1, p, p^2$ . The binary search returns the exact floor root, so the two checks accept precisely those inputs.

**Complexity.** With  $B = 10^6$ , preprocessing is  $O(B \log \log B)$ , queries take  $O(q \log B)$ , and stored flags use  $O(B)$  space.

**Boundary checks.** The upper-midpoint search never divides by 0 because its midpoint is positive whenever the loop runs. The root is at most  $10^6$ , so its square fits. Squares of composite roots, such as 16, must be rejected.

### Solution 13: Many prime-count intervals

**Idea.** Sieve the domain, form cumulative prime counts, and answer each interval by subtracting the count before its left endpoint.

**Complete solution | C++23**

```
#include <iostream>
#include <vector>

std::vector<char> buildSieve(int limit) {
    std::vector<char> prime(limit + 1, 1);
    prime[0] = 0;
    if (limit >= 1) prime[1] = 0;
    for (int p = 2; p <= limit / p; ++p) {
        if (!prime[p]) continue;
        for (long long multiple = 1LL * p * p;
             multiple <= limit; multiple += p) {
            prime[multiple] = 0;
        }
    }
    return prime;
}

int main() {
    int limit = 0, q = 0;
    std::cin >> limit >> q;
    const auto prime = buildSieve(limit);
    std::vector<int> prefix(limit + 1, 0);
    for (int x = 1; x <= limit; ++x) {
        prefix[x] = prefix[x - 1] + prime[x];
    }
    while (q-- > 0) {
        int left = 0, right = 0;
        std::cin >> left >> right;
        std::cout << prefix[right] - (left > 0 ? prefix[left - 1] : 0)
                  << '\n';
    }
}
```

**Why it works.** The sieve identifies primes exactly. Each prefix contains one contribution per prime at most its index. Removing the prefix before  $L$  leaves exactly the contributions in  $[L, R]$ .

**Complexity.**  $O(N \log \log N + q)$  time for  $N \geq 3$ , with constant-size initialization for smaller  $N$ ;  $O(N + 1)$  storage.

**Boundary checks.** An interval containing only 0 and 1 has count 0. A singleton prime has count 1. Include queries touching both domain ends.

**Solution 14: Select a prime inside a bounded domain**

**Idea.** Build the sieve once, scan values increasingly into a prime list, and answer each valid rank at index  $k - 1$ .

**Complete solution | C++23**

```
#include <iostream>
#include <vector>

std::vector<char> buildSieve(int limit) {
    std::vector<char> prime(limit + 1, 1);
    prime[0] = 0;
    if (limit >= 1) prime[1] = 0;
    for (int p = 2; p <= limit / p; ++p) {
        if (!prime[p]) continue;
        for (long long multiple = 1LL * p * p;
             multiple <= limit; multiple += p) {
            prime[multiple] = 0;
        }
    }
    return prime;
}

int main() {
    int limit = 0, q = 0;
    std::cin >> limit >> q;
    const auto prime = buildSieve(limit);
    std::vector<int> primes;
    for (int x = 2; x <= limit; ++x) {
        if (prime[x]) primes.push_back(x);
    }
    while (q-- > 0) {
        int k = 0;
        std::cin >> k;
        if (k > static_cast<int>(primes.size())) std::cout << -1 << '\n';
        else std::cout << primes[k - 1] << '\n';
    }
}
```

**Why it works.** The scan appends exactly the prime values and appends them in increasing order. A valid rank therefore has the required prime at index  $k - 1$ . A rank exceeding the list size has no answer in the bounded domain.

**Complexity.**  $O(N \log \log N + q)$  time for  $N \geq 3$  and  $O(N + 1)$  storage, including flags and the prime list.

**Boundary checks.** For  $N < 2$ , every rank fails. Do not enlarge the domain or invent the next prime when the requested rank is absent.

## Solution 15: Fast prime-factor statistics

**Idea.** Build SPF once. Each new prime group increments **distinct**; each exact division increments **total**. Continue until the current query value reaches 1.

**Complete solution | C++23**

```
#include <iostream>
#include <vector>

std::vector<int> buildSpf(int limit) {
    std::vector<int> spf(limit + 1, 0);
    for (int p = 2; p <= limit; ++p) {
        if (spf[p] != 0) continue;
        spf[p] = p;
        if (p > limit / p) continue;
        for (long long multiple = 1LL * p * p;
             multiple <= limit; multiple += p) {
            if (spf[multiple] == 0) spf[multiple] = p;
        }
    }
    return spf;
}

int main() {
    int limit = 0, q = 0;
    std::cin >> limit >> q;
    const auto spf = buildSpf(limit);
    while (q-- > 0) {
        int x = 0, distinct = 0, total = 0;
        std::cin >> x;
        while (x > 1) {
            int p = spf[x];
            ++distinct;
            while (x > 1 && spf[x] == p) {
                x /= p;
                ++total;
            }
        }
        std::cout << distinct << ' ' << total << '\n';
    }
}
```

**Why it works.** SPF supplies the smallest remaining prime. Removing all copies forms exactly one group for that prime, so groups count distinct primes and divisions count occurrences. Every factor is removed once.

**Complexity.**  $O(N \log \log N + q + \sum \log x_i)$  time for  $N \geq 3$  and  $O(N)$  stored SPF entries, with  $O(1)$  extra query state.

**Boundary checks.** The input 1 returns 0 0. For 72, the two answers are 2 and 5. Guard  $x > 1$  before reading an SPF entry as a divisor.

## Solution 16: Reuse all divisor counts

**Idea.** Build the count table with an outer loop over divisors and an inner loop over their positive multiples. A query then reads one table entry.

**Complete solution | C++23**

```
#include <iostream>
#include <vector>

int main() {
    int limit = 0, q = 0;
    std::cin >> limit >> q;
    std::vector<int> count(limit + 1, 0);
    for (int d = 1; d <= limit; ++d) {
        for (int multiple = d; multiple <= limit; multiple += d) {
            ++count[multiple];
        }
    }
    while (q-- > 0) {
        int x = 0;
        std::cin >> x;
        std::cout << count[x] << '\n';
    }
}
```

**Why it works.** For any  $x$ , an update from  $d$  reaches it exactly when  $d \mid x$ , and that outer divisor contributes only once. Thus the final entry counts exactly all positive divisors of  $x$ .

**Complexity.**  $O(N \log(N + 1) + q)$  time and  $O(N)$  storage. The logarithmic factor comes from visits by all divisors, not only primes.

**Boundary checks.** Entry 1 receives one contribution. Index 0 is not queried. A square's root needs no special duplicate guard because each divisor value is an outer iteration only once.

**Solution 17: How many input values are divisible by  $d$ ?**

**Idea.** Store input multiplicities in `frequency`. Precompute `count[d]` as the sum of frequencies at all positive multiples of  $d$ .

**Complete solution | C++23**

```
#include <iostream>
#include <vector>

int main() {
    int limit = 0, n = 0, q = 0;
    std::cin >> limit >> n >> q;
    std::vector<int> frequency(limit + 1, 0), count(limit + 1, 0);
    for (int i = 0; i < n; ++i) {
        int x = 0;
        std::cin >> x;
        ++frequency[x];
    }
    for (int d = 1; d <= limit; ++d) {
        for (int multiple = d; multiple <= limit; multiple += d) {
            count[d] += frequency[multiple];
        }
    }
    while (q-- > 0) {
        int d = 0;
        std::cin >> d;
        std::cout << count[d] << '\n';
    }
}
```

**Why it works.** An input position contributes exactly once to `count[d]` when its value is a multiple of  $d$ . Frequencies preserve duplicate positions. No nonmultiple is visited, so the sum matches the requested count.

**Complexity.**  $O(n + M \log(M + 1) + q)$  time and  $O(M)$  space. The input list itself need not be stored.

**Boundary checks.** An empty input list makes every answer 0. The query  $d = 1$  returns  $n$ . No zero-valued inputs are permitted by this frequency-domain contract.

**Solution 18: Complete the missing square factors**

**Idea.** Factor  $n$  and multiply one copy of each prime whose exponent is odd.

**Complete solution | C++23**

```
#include <iostream>
#include <vector>
#include <utility>

std::vector<std::pair<long long, int>> factorize(long long remaining) {
    std::vector<std::pair<long long, int>> factors;
    for (long long p = 2; p <= remaining / p; ++p) {
        if (remaining % p != 0) continue;
        int exponent = 0;
        while (remaining % p == 0) {
            remaining /= p;
            ++exponent;
        }
        factors.push_back({p, exponent});
    }
    if (remaining > 1) factors.push_back({remaining, 1});
    return factors;
}

int main() {
    long long n = 0, answer = 1;
    std::cin >> n;
    for (const auto& [p, exponent] : factorize(n)) {
        if (exponent % 2 != 0) answer *= p;
    }
    std::cout << answer << '\n';
}
```

**Why it works.** Every valid multiplier must contain each odd-exponent prime to an odd power, hence at least once. The product of one copy of each such prime already makes all exponents even and divides every valid multiplier. It is therefore the smallest.

**Complexity.**  $O(\sqrt{n})$  time and  $O(\omega(n))$  stored factors.

**Boundary checks.** The multiplier is at most  $n$ , but  $nm$  may exceed `long long`. Do not compute an unrequested product merely to print the multiplier. A square returns 1.

**Solution 19: Count and sum primes in a distant interval**

**Idea.** Build a segmented prime table for the interval, then scan its surviving entries to accumulate the count and sum.

**Why it works.** Every composite in the segment has a base-prime divisor whose marking loop reaches it. Starting at the maximum of  $p^2$  and the first interval multiple preserves primes. Clearing 0 and 1 leaves exactly the primes, and the final scan includes each survivor once.

**Complexity.** For  $B = \lfloor \sqrt{R} \rfloor \geq 3$  and  $W = R - L + 1$ , use the safe bound  $O(B \log \log B + W \log \log B + \pi(B))$  time and  $O(B + W)$  memory. Tiny  $B$  cases use constant setup plus  $O(W)$  work.

**Boundary checks.** The count is at most  $10^5$  and the sum at most  $10^{17}$ , so the chosen types fit. Test segments containing 0 or 1, a singleton prime, and a prime square.

## Implementation: Count and sum primes in a distant interval

### Complete solution | C++23

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> buildSieve(int limit) {
    std::vector<char> prime(limit + 1, 1);
    prime[0] = 0;
    if (limit >= 1) prime[1] = 0;
    for (int p = 2; p <= limit / p; ++p) {
        if (!prime[p]) continue;
        for (long long multiple = 1LL * p * p;
             multiple <= limit; multiple += p) {
            prime[multiple] = 0;
        }
    }
    return prime;
}

std::vector<char> segmentPrimes(long long left, long long right) {
    int root = 0;
    while (root + 1LL <= right / (root + 1LL)) ++root;
    const auto base = buildSieve(root);
    std::vector<char> prime(right - left + 1, 1);
    for (int p = 2; p <= root; ++p) {
        if (!base[p]) continue;
        long long start = std::max(1LL * p * p,
                                   ((left + p - 1) / p) * p);
        for (long long x = start; x <= right; x += p) {
            prime[x - left] = 0;
        }
    }
    for (long long x = left; x <= right && x < 2; ++x) {
        prime[x - left] = 0;
    }
    return prime;
}

int main() {
    long long left = 0, right = 0;
    std::cin >> left >> right;
    const auto prime = segmentPrimes(left, right);
    int count = 0;
    long long sum = 0;
    for (long long x = left; x <= right; ++x) {
        if (!prime[x - left]) continue;
        ++count;
        sum += x;
    }
    std::cout << count << ' ' << sum << '\n';
}
```

**Solution 20: A prime exponent inside a binomial coefficient**

**Idea.** Compute  $v_p(n!)$ ,  $v_p(r!)$ , and  $v_p((n-r)!)$  with repeated integer division. Subtract the latter two from the first.

**Complete solution | C++23**

```
#include <iostream>

long long exponentInFactorial(long long n, long long p) {
    long long exponent = 0;
    while (n > 0) {
        n /= p;
        exponent += n;
    }
    return exponent;
}

int main() {
    long long n = 0, r = 0, p = 0;
    std::cin >> n >> r >> p;
    long long answer = exponentInFactorial(n, p)
        - exponentInFactorial(r, p)
        - exponentInFactorial(n - r, p);
    std::cout << answer << '\n';
}
```

**Why it works.** The identity  $n! = \binom{n}{r} r! (n-r)!$  equates prime exponents on both sides. Subtracting the denominator exponents therefore gives exactly the exponent of  $p$  in the integer binomial coefficient.

**Complexity.**  $O(1 + \log_p(n+1))$  time and  $O(1)$  space. Each factorial exponent is at most  $n$ , and the final result is nonnegative.

**Boundary checks.** For  $r = 0$  or  $r = n$ , the coefficient is 1 and the answer is 0. The prime promise matters. No factorial, full binomial coefficient, or modular inverse is evaluated.

## 5.37 Explained trace and reasoning drills

Use these answers to check the meaning of your reasoning, not only your arithmetic. A different correct implementation is acceptable when its invariant, bounds, and total cost are justified.

### Drill 1: One number, several different answers

The positive divisors of 36 are 1, 2, 3, 4, 6, 9, 12, 18, 36. The divisor count is 9, the divisor sum is 91, and the proper-divisor sum is  $91 - 36 = 55$ .

Its prime factorization is  $2^2 \cdot 3^2$ . There are two distinct prime factors and four prime occurrences. The divisor count is also  $(2 + 1)(2 + 1) = 9$ , and the divisor sum is  $(1 + 2 + 4)(1 + 3 + 9) = 91$ . These cross-checks agree with the list but count different underlying objects from the prime-factor counts.

### Drill 2: A boundary error hidden inside a short loop

For 1, the loop runs zero times and prints 0 instead of 1. For 4, it visits only  $d = 1$  and gives 2 instead of 3. Replacing  $<$  by  $<=$  alone would then double-count the equal pair  $(2, 2)$ . A safer fragment is:

```
long long count = 0;
for (long long d = 1; d <= n / d; ++d) {
    if (n % d != 0) continue;
    count += (d == n / d ? 1 : 2);
}
```

The division-based condition avoids the potentially overflowing square and is defined because  $d \geq 1$ . The guard handles the unique middle divisor of a square.

### Drill 3: Trace the remaining value, not the original boundary

Removing two copies of 2 gives  $84 \rightarrow 42 \rightarrow 21$ , so the invariant is  $84 = 2^2 \cdot 21$ . Removing one 3 gives  $21 \rightarrow 7$ . The next candidate is 4, but  $4 > 7/4$ , so the loop stops. Append the leftover prime 7.

The complete factorization is  $84 = 2^2 \cdot 3 \cdot 7$ . The bound belongs to 7 at the end, not to 84. If 7 were composite, it would need a prime factor no larger than its square root; all such smaller candidates have already been exhausted.

**Drill 4: Two different incomplete factorizations**

With a single division at candidate 2, the first version changes 12 to 6 and records just one 2. Its next test is  $3 \leq 6/3$ , which is false. It then appends 6 as the leftover, producing factors 2 and 6 rather than 2, 2, 3.

The leftover proof fails because not every copy of the smaller prime 2 was removed; 6 can still contain that earlier prime. In the second version, 13 has no successful trial divisor, so no factor is recorded. Deleting the leftover step loses the entire factorization of an original prime and can also lose a final prime after successful divisions.

**Drill 5: Exponent choices instead of a long divisor list**

An ordinary divisor chooses exponents  $0 \dots 3$ ,  $0 \dots 3$ , and  $0 \dots 1$ . Therefore

$$\tau(1080) = 4 \cdot 4 \cdot 2 = 32.$$

Adding the values of those choices through a distributed product gives

$$\sigma(1080) = (1 + 2 + 4 + 8)(1 + 3 + 9 + 27)(1 + 5) = 15 \cdot 40 \cdot 6 = 3600.$$

A square divisor may use exponent 0 or 2 for each of 2 and 3, and only exponent 0 for 5. The count is  $2 \cdot 2 \cdot 1 = 4$ , corresponding to 1, 4, 9, 36.

**Drill 6: Which sieve entries can change?**

The marking stage for 3 starts at 9. Its earlier multiple 6 was already removed by the stage for 2. Marking 12 again is harmless because setting a composite flag to zero is idempotent: doing it twice has the same result as doing it once.

Starting at the prime itself would cross off an actual prime, violating the “no prime is removed” half of the proof. The same problem occurs for every prime whose stage runs. Prime values above the outer-loop boundary could still survive, which makes this error more subtle than simply producing an empty output in every implementation.

**Drill 7: Follow an SPF chain**

The values are  $60 \rightarrow 30 \rightarrow 15 \rightarrow 5 \rightarrow 1$ , removing 2, 2, 3, 5. There are three prime groups and four occurrences. The two copies of 2 belong to one distinct group.

At 1, all factors have been removed. The table sentinel `spf[1] = 0` is not a prime divisor, and using it in division is invalid. Overwriting entries during construction may store a later prime factor instead of the smallest one. It can retain some factorization information, but it no longer satisfies the declared SPF contract or the smallest-first ordering argument.

**Drill 8: Choose a representation for the workload**

For one large divisor list, use paired trial enumeration with small and large lists. A full table through  $10^{12}$  is unnecessary and too large for the intended setting.

For many bounded primality questions, build prime flags once. For many bounded full factorizations, build SPF entries so each query can remove known prime factors. For many prime interval counts, add a prefix-count table to the prime flags. A Boolean flag cannot provide all prime factors, and an SPF entry alone does not answer a whole interval count in constant time. The requested information determines the representation.

**Drill 9: The harmonic double loop**

For  $N = 6$ , the six outer stages make 6, 3, 2, 1, 1, 1 visits, for a total of 14. In general, the count is

$$\sum_{d=1}^N \left\lfloor \frac{N}{d} \right\rfloor = O(N \log N).$$

The prime-only sieve avoids stages for composite  $d$ ; this loop deliberately includes them. We can also count the same visits by their destinations: each value  $x$  is reached once for every positive divisor, so the total equals  $\sum_{x=1}^N \tau(x)$ . For  $N = 6$ , the divisor counts 1, 2, 2, 3, 2, 4 also sum to 14.

**Drill 10: An exhausted list is not a proof**

None of 2, 3, and 5 divides 77, so the helper appends 77. That is composite: the necessary trial prime 7 was missing. A sufficient guarantee is that the prime list includes every prime through  $\lfloor \sqrt{M} \rfloor$ , where all queried positive inputs are at most  $M$ .

A direct flag lookup has a different requirement: its index must lie in the allocated domain. A sieve through one million provides no flag at 1000000007. Its list of small primes can still assist a properly bounded trial test, but that is a different operation from indexing the table.

**Drill 11: Optional: shifted indices and prime-preserving starts**

For  $[0, 10]$ , the starts are  $\max(4, 0) = 4$  for 2 and  $\max(9, 0) = 9$  for 3. Clear 0 and 1 separately; the surviving values are 2, 3, 5, 7. Their count is 4 and their sum is 17.

For  $[49, 49]$ , value 49 maps to index 0. The base prime 7 starts at 49 and crosses it off, so there are no primes in the interval. The  $p^2$  lower bound includes composite squares while protecting a base prime in a segment such as  $[2, 7]$ . The offset  $x - L$  ensures every accessed interval index lies between 0 and  $R - L$ .

**Drill 12: Optional: read the exponent without storing the number**

The exponents are

$$v_2(10!) = 5 + 2 + 1 = 8, \quad v_2(3!) = 1, \quad v_2(7!) = 3 + 1 = 4.$$

Thus  $v_2\left(\binom{10}{3}\right) = 8 - 1 - 4 = 3$ . Indeed,  $\binom{10}{3} = 120 = 2^3 \cdot 3 \cdot 5$ .

The method uses the exact factorization identity for the factorial quotient, so it never needs to store the coefficient itself. For  $r = 0$ , the subtraction becomes  $v_p(n!) - v_p(0!) - v_p(n!) = 0$ , matching  $\binom{n}{0} = 1$ . The same holds for  $r = n$ .

## 5.38 Quick reference, glossary, and readiness checklist

### The formulas, with their meanings attached

Throughout this page,  $n \geq 1$  and

$$n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k},$$

where the  $p_i$  are different primes and every  $e_i \geq 1$ . For  $n = 1$ , the factor list is empty: an empty product has value 1, and an empty sum has value 0.

| Question                       | Answer and reason                                                                                      |
|--------------------------------|--------------------------------------------------------------------------------------------------------|
| Does $d$ divide $n$ ?          | For $d > 0$ , test <code>n % d == 0</code> . The division must be exact.                               |
| What is the paired divisor?    | $n/d$ . Test only $d \leq n/d$ , and do not duplicate a pair with equal members.                       |
| How many positive divisors?    | $\tau(n) = \prod_i (e_i + 1)$ . Choose each exponent from 0 through $e_i$ .                            |
| What is their sum?             | $\sigma(n) = \prod_i (1 + p_i + \cdots + p_i^{e_i})$ . Expand the product to obtain each divisor once. |
| Sum of proper divisors?        | $\sigma(n) - n$ . The number itself is the only excluded positive divisor.                             |
| How many prime factors?        | $\omega(n) = k$ counts distinct primes; $\Omega(n) = \sum_i e_i$ counts copies.                        |
| Is $n$ a perfect square?       | Every $e_i$ is even. This includes $n = 1$ .                                                           |
| How many square divisors?      | $\prod_i (\lfloor e_i/2 \rfloor + 1)$ . Choose only even exponents.                                    |
| Exactly three divisors?        | $n = p^2$ for a prime $p$ . Its divisors are $1, p, p^2$ .                                             |
| Smallest $m$ with $nm$ square? | Multiply the primes with odd exponents in $n$ once each.                                               |

#### A mathematical formula still needs an arithmetic bound

A value fitting in `long long` does not imply every rearranged intermediate expression fits. Build a prime-power sum one term at a time instead of calculating an unnecessary  $p^{e+1}$ . Never use floating-point `pow` to represent an exact integer factorization.

## Choose the preparation that matches the workload

Let  $N$  be a table limit,  $q$  the number of queries,  $M$  the largest number to factor, and  $D = \tau(n)$ . Bounds below count fixed-width operations. The logarithmic expressions describe nontrivial growing inputs; tiny limits take constant setup work.

| Task                           | Method and time                                                                                    | Stored information                                                       |
|--------------------------------|----------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------|
| One divisor list               | Paired scan: $O(\sqrt{n} + D)$ , already sorted with two lists.                                    | $O(D)$ output storage.                                                   |
| One prime test                 | Trial division: $O(\sqrt{n})$ worst case.                                                          | $O(1)$ extra state.                                                      |
| One factorization              | Repeated trial division: $O(\sqrt{n})$ worst case.                                                 | Prime/exponent output; $O(1)$ beyond it.                                 |
| Many prime tests               | Sieve once: $O(N \log \log N)$ ; $O(1)$ lookup.                                                    | $O(N)$ flags.                                                            |
| Prime counts in ranges         | Sieve plus $O(N)$ prefix; $O(1)$ query.                                                            | Flags and cumulative counts.                                             |
| Many bounded factorizations    | SPF preparation: $O(N \log \log N)$ here; $O(\log x)$ per factorization.                           | $O(N)$ smallest-prime-factor entries.                                    |
| Several larger factorizations  | Sieve primes through $B = \lfloor \sqrt{M} \rfloor$ , then try the required primes for each input. | Base primes and their sieve; complete coverage through $B$ is essential. |
| All divisor counts/sums        | Visit every multiple of every $d$ : $O(N \log N)$ total.                                           | $O(N)$ result entries.                                                   |
| Optional narrow prime interval | Segmented marking using base primes through $\sqrt{R}$ .                                           | Segment plus base-prime information, not a full table through $R$ .      |

## Count the whole program

For  $q$  independent trial divisions with largest input  $M$ , a safe worst-case bound is  $O(q\sqrt{M})$ , not  $O(\sqrt{M})$ . SPF factorization after one table is  $O(N \log \log N + \sum_i \log x_i)$  plus output work. Printing  $D$  divisors requires at least  $D$  output operations even after the factors are known.

A prime table has no information above its bound. A factorization routine using a short prime list cannot automatically declare the remaining number prime. The proof depends on having checked every possible smaller prime factor.

**Three core fragments to reconstruct, not memorize**

**Paired traversal inside a function.** Preconditions:  $1 \leq n \leq 10^{12}$ , and `n` and `d` use `long long`. Replace the comments with the required action.

```
for (long long d = 1; d <= n / d; ++d) {
    if (n % d != 0) {
        continue;
    }
    // Process divisor d.
    long long other = n / d;
    if (other != d) {
        // Process divisor other exactly once.
    }
}
```

**Factor extraction inside a function.** Preconditions:  $1 \leq n \leq 10^{12}$  and `n` uses `long long`. Preserve the original in another variable when it is still needed.

```
long long remaining = n;
for (long long p = 2; p <= remaining / p; ++p) {
    int exponent = 0;
    while (remaining % p == 0) {
        remaining /= p;
        ++exponent;
    }
    if (exponent > 0) {
        // Process the factor p with this exponent.
    }
}
if (remaining > 1) {
    // Process the final prime with exponent 1.
}
```

**SPF traversal inside a function.** Preconditions:  $1 \leq x \leq N$ , and `spf` was built through `N`. Each step divides the current value, not the original input.

```
while (x > 1) {
    int p = spf[x];
    // Process one occurrence of prime p.
    x /= p;
}
```

## A glossary for this session

| Term                       | Meaning                                                                                                       |
|----------------------------|---------------------------------------------------------------------------------------------------------------|
| Divisibility               | $d \mid n$ means that $n = dq$ for some integer $q$ .                                                         |
| Positive divisor           | A positive integer that divides the number exactly.                                                           |
| Proper divisor             | A positive divisor other than the number itself.                                                              |
| Prime                      | A positive integer greater than 1 with exactly two positive divisors.                                         |
| Composite                  | An integer greater than 1 that is not prime.                                                                  |
| Prime factorization        | Writing an integer greater than 1 as a product of primes, unique up to order.                                 |
| Exponent                   | The number of copies of a prime in the grouped factorization.                                                 |
| Residual / remaining value | The part of the original number not yet extracted as known factors.                                           |
| $\tau(n)$ / $\sigma(n)$    | The count / sum of all positive divisors of $n$ .                                                             |
| Sieve                      | A computation over a bounded integer domain that systematically marks or records information at multiples.    |
| SPF                        | Smallest prime factor. For $x \geq 2$ , <code>spf[x]</code> is a prime dividing $x$ .                         |
| Preprocessing              | Work done once to make later queries cheaper.                                                                 |
| Segmented sieve            | Marking only a requested interval, using smaller base primes.                                                 |
| Prime valuation $v_p(n)$   | The exponent of a promised prime $p$ in $n$ .                                                                 |
| Invariant                  | A statement preserved throughout the algorithm, such as extracted product times residual equals the original. |

## Words that must not be exchanged

A **divisor** need not be prime. A **distinct prime factor** is not an occurrence count. A **factorization** is not a divisor list. A **prime flag** is not an SPF entry. Being **unmarked so far** is not always the same as having completed a primality decision.

The input 1 is especially useful: its positive divisor list is 1, its prime-factor list is empty, its divisor count and sum are both 1, and its proper-divisor sum is 0. None of these statements makes 1 prime.

## Readiness checklist

Before moving on, explain the following without opening a complete solution:

- Why does every composite positive number have a prime factor at most its square root? Why must the square-root boundary be included?
- Why do repeated divisions use `while`? Why is a leftover value greater than 1 prime after the complete trial loop?
- What does one exponent choice represent in the divisor-count and divisor-sum formulas? Why are the resulting divisors unique?
- Why does Eratosthenes start marking at  $p^2$ ? Why are 0 and 1 not prime? What must be initialized when  $N = 0$ ?
- What does `spf[x]` mean, and why does repeatedly dividing by it finish in  $O(\log x)$  steps?
- Which parts should be prepared once for many queries? How much time, input storage, output storage, and table storage does the full solution use?

## A practice plan, not an extra lecture

**First pass: one number.** Read divisor pairs, primality, repeated factorization, and exponent formulas. Solve Practices 1 to 11. Use a hand-written divisor list as your first oracle.

**Second pass: shared work.** Read prime flags, prime prefixes, SPF, and divisor tables. Solve Practices 12 to 17. For each, write the preparation cost and per-query cost separately before coding.

**Third pass: optional extensions.** Return to linear and segmented sieves, factorial exponents, and square completion. Attempt Practices 18 to 20 only after the core contracts feel clear.

### Session practice details

**Practice sheet:** \_\_\_\_\_

**Start date:** \_\_\_\_\_

**End date:** \_\_\_\_\_

Fill these fields when the training team publishes the sheet and agreed timeline. During practice, record the failed assumption or boundary case, not only the final accepted code.

## What comes next

Session 6 will develop modular arithmetic, binary power, modular inverses, GCD, and LCM. This chapter uses exact divisibility and bounded integer arithmetic; it does not assume a modular inverse exists or replace exact division with modular division. The collected edition contains Sessions 1 to 5, not the unwritten final session.

## 5.39 Technical notes, verification, and references

### Language, numeric model, and scope

The complete programs use standard C++23 headers. The checking environment used GCC 14.2.0 with a 32-bit `int` and a 64-bit `long long`. C++ does not require `int` to be exactly 32 bits; this edition assumes an `int` wide enough for its stated limits, as on the tested environment. Exact output bounds are part of each problem contract.

All factorization and divisor-list tasks use positive integers. Prime classification and sieve-domain tasks explicitly include 0 where appropriate. Negative-factor conventions and factorization of zero are not silently folded into positive-number formulas. For integer remainder, the right operand must be nonzero. Signed overflow is not a valid way to approximate a large exact answer.<sup>[5-5][5-6]</sup>

Complexities count fixed-width operations. They are not bit-complexity bounds for arbitrary-size integers, and they are not runtime guarantees for a particular online judge. A trial loop with  $O(\sqrt{n})$  work is exponential in the order of the bit length of  $n$ ; it is not an appropriate general-purpose factorizer for arbitrary large cryptographic integers.

A table's **bound**, the number of **queries**, and the required **output size** must all be included in an implementation decision. Illustrative memory amounts elsewhere in this chapter refer to the tested implementation and omit allocator overhead unless stated otherwise.

### Citations and original teaching material

The number-theory references support prime factorization, divisor identities, and sieve analysis. The C++ references point to the fixed N4950 C++23 draft and official GCC documentation. These are technical references, not prerequisites for reading the beginner-facing explanations.

The chapter's worked traces, problems, examples, proofs of its specific algorithms, and test inputs were written for this training. A superscript such as <sup>[5-2]</sup> links to a numbered source entry below. The prefix 5 keeps the reference namespace separate from earlier sessions.

### Boundary conditions worth keeping beside the code

Use  $d \leq n / d$  only with positive  $d$  and nonnegative  $n$ . Use a widened product only after checking that the widened type itself is large enough. Reset the residual and per-query output for each new number. Construct a vector with actual elements before indexing; reserving capacity alone does not create them.<sup>[5-7]</sup>

## What was checked for this edition

The source package contains **24 complete examples and 20 complete practice solutions** for Session 5. All 44 compiled with GCC 14.2.0 in C++23 mode and passed **14204 automated input/output checks**, including all 44 printed samples. The suite uses independent small-input enumeration, generated inputs, targeted boundary cases, and larger workload checks.

Checks include 1, 0 where permitted, prime squares, repeated prime powers, leftover primes, absent ranks, empty query batches, full table endpoints, repeated input values, single-cell segments, high-valued narrow segments, and wide sums. One process execution counts as one input/output check even when its input contains many queries. A separate Python script also passed 12 grouped checks of numerical claims in the reasoning drills; those are not added to the input/output total.

The earlier five topic suites also completed successfully: Two Pointers, Binary Search, Recursion and Backtracking, Bit Manipulation, and Counting Principles. Together with the new suite, **264 complete programs passed 78123 input/output checks**. The earlier 220 complete C++ files were checked against their original SHA-256 hashes and remain unchanged.

The verification build used:

```
-std=c++23 -pedantic-errors -Wall -Wextra  
-Wconversion -Wshadow -Werror -O1 -g  
-D_GLIBCXX_ASSERTIONS -fsanitize=undefined  
-fno-sanitize-recover=all
```

The compiler warnings, undefined-behavior checks, and library assertions help detect executed mistakes; they are not a proof for all valid inputs. Deliberately wrong teaching fragments are labelled and are not distributed as working solution files.<sup>[5-8][5-9]</sup>

## Reproduce the code checks

From the source-package root, run:

```
python3 tests/verify_number_theory_1.py
```

The runner uses Python's standard library and a suitable GNU C++23 compiler. It records each program's result in `reports/number_theory_1_verification.json`. The manifest in `tests/number_theory_1_programs.json` links each program to its printed sample. Earlier suites use stable topic names, not obsolete session numbers.

## Compile one example or rebuild the PDFs

```
g++ -std=c++23 -O2 -Wall -Wextra \  
    examples/number_theory_1/e06_primality.cpp -o example  
python3 build.py --target session5  
python3 build.py --target book
```

Every complete C++ file has its own `main`; compile one file at a time. Both PDF targets include the same chapter source and the same external program files. The README describes the full project layout and the commands for every completed session.

## Number-theory sources

### [5-1] Wissam Raji: The Fundamental Theorem of Arithmetic

[https://math.libretexts.org/Bookshelves/Combinatorics\\_and\\_Discrete\\_Mathematics/Elementary\\_Number\\_Theory\\_\(Raji\)/02:\\_Prime\\_Numbers/2.03:\\_The\\_Fundamental\\_Theorem\\_of\\_Arithmetic](https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/Elementary_Number_Theory_(Raji)/02:_Prime_Numbers/2.03:_The_Fundamental_Theorem_of_Arithmetic)

### [5-2] Wissam Raji: Multiplicative Number Theoretic Functions

[https://math.libretexts.org/Bookshelves/Combinatorics\\_and\\_Discrete\\_Mathematics/Elementary\\_Number\\_Theory\\_\(Raji\)/04:\\_Multiplicative\\_Number\\_Theoretic\\_Functions/4.02:\\_Multiplicative\\_Number\\_Theoretic\\_Functions](https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/Elementary_Number_Theory_(Raji)/04:_Multiplicative_Number_Theoretic_Functions/4.02:_Multiplicative_Number_Theoretic_Functions)

### [5-3] Wissam Raji: The Sieve of Eratosthenes

[https://math.libretexts.org/Bookshelves/Combinatorics\\_and\\_Discrete\\_Mathematics/Elementary\\_Number\\_Theory\\_\(Raji\)/02:\\_Prime\\_Numbers/2.01:\\_The\\_Sieve\\_of\\_Eratosthenes](https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/Elementary_Number_Theory_(Raji)/02:_Prime_Numbers/2.01:_The_Sieve_of_Eratosthenes)

### [5-4] Antti Laaksonen: Competitive Programmer's Handbook, chapter 21.1

<https://cses.fi/book/book.pdf>

The handbook's chapter on basic number theory discusses factors, prime factorization, and prime sieving. Its sieve analysis distinguishes visiting multiples of primes from visiting multiples of every integer.

## Language and implementation sources

### [5-5] C++23 N4950: multiplicative operators, division, and remainder

<https://timsong-cpp.github.io/cppwp/n4950/expr.mul>

### [5-6] C++23 N4950: expressions and results outside representable ranges

<https://timsong-cpp.github.io/cppwp/n4950/expr.pre>

### [5-7] C++23 N4950: vector construction and element counts

<https://timsong-cpp.github.io/cppwp/n4950/vector.cons>

### [5-8] GCC: warning options and strict diagnostics

<https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

### [5-9] GCC: instrumentation and sanitizer options

<https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>