what are all those algorithm to do prime number computation
sieve of eratothenes
list them all and related concept to it how to implement it and all use cases in very detail

Here is a **complete, structured, crystal-clear guide** to *all major algorithms and related concepts* used in **prime number computation**, with **what they are**, **when to use them**, **complexity**, **concepts involved**, and **how to implement** each.

I'll cover everything from **basic to advanced CP-level prime algorithms**.

✅ I. BASIC PRIME CHECKING ALGORITHMS

1. Trial Division

The simplest method: try dividing n by all numbers from 2 to $\sqrt{n}$.

⏱️ Complexity

- Worst case: $O(\sqrt{n})$

🧠 Concepts

- Any composite number must have a factor ≤ √n.
- Checking only odd numbers (after 2) is enough.

✔️ Use-case

- Very good when you need to check **one number**.
- Works for numbers up to **1e12** comfortably in contests.

---

2. Optimized Trial Division

Further improvements:

1. Divide by **2 separately**.
2. Check only numbers of form `6k ± 1`.

Why 6k ± 1?

All primes > 3 are of form 6k ± 1.

Complexity

Still $O(\sqrt{n})$ but constant factor reduces ~3x.

✅ II. PRIME SIEVING ALGORITHMS

These algorithms generate ALL primes up to some number N.

---

1. SIEVE OF ERATOSTHENES

The most famous and widely used sieve.

🔍 How it works

- Mark 2 as prime.
- Start from 2 → mark all multiples as composite.
- Move to next unmarked number → that's a prime.
- Repeat until √N.

🧠 Concepts

- Multiples of prime p start from **p²**.
- Each composite is marked once per prime.

⏱️ Complexity

- Time: $O(N \log\log N)$
- Space: $O(N)$

✔️ Use Case

- Need primes up to 1e7 or 1e8.
- Need a precomputation for:
  - factorization
  - prefix sums
  - prime gaps
  - totient computation

---

2. SEGMENTED SIEVE

Used when you need primes in a **range [L, R]** where R can be as large as 10^{12} or more.

🔍 Concept

- First generate primes up to √R using normal sieve.
- Use them to mark composites in the segment [L, R].

⏱️ Complexity

- Time: $O((R - L + 1) \log \log R)$
- Space: $O(R - L + 1)$

✔️ Use-case

- Print primes in **[1e10, 1e10 + 1e6]**
- When R is too big for normal sieve.

---

3. LINEAR SIEVE / SIEVE OF ERATOSTHENES (O(N))

Also known as **Euler's sieve**.

🔍 Idea

Each composite number is marked *exactly once* using the smallest prime that divides it.

⏱️ Complexity

- Time: **O(N)**
- Space: **O(N)**

✔️ Use-case

- Need **prime factorization for all numbers up to N**.
- Need **smallest prime factor (SPF) array**.
- Useful for:
  - number of divisors
  - prime factorization in log n
  - Euler phi computation

✅ III. PRIMALITY TESTING (FAST CHECKING FOR VERY LARGE NUMBERS)

These test if a number is prime—fast and suitable for **64-bit integers**.

---

1. MILLER–RABIN

A *probabilistic* test but deterministic for 64 bits with fixed bases.

⏱️ Complexity

- $O(k \log^3 n)$, with k = number of iterations

✔️ Use-case

- Check primality of numbers up to **2^{64}**
- Used in CP whenever n > 1e12

🔍 Why it's great

- Much faster than √n
- Works for very large numbers
- Deterministic for C++ 64-bit using bases:
  `{2, 325, 9375, 28178, 450775, 9780504, 1795265022}`

---

2. FERMAT PRIMALITY TEST (Weak, rarely used)

Uses Fermat's little theorem:

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

× Problems

- Carmichael numbers fool it.

✔️ Use-case

- Almost never in CP; Miller–Rabin dominates.

✅ IV. PRIME FACTORIZATION ALGORITHMS

---

1. Smallest Prime Factor (SPF) Sieve

Using sieve to store smallest prime factor for each number.

🔍 Use-case

Factorize any number ≤ N in **O(logN)**.

✔️ Very useful for

- divisor computation
- prime exponent counting
- multiplicative functions
    - phi
    - mobius
    - divisor sum

---

2. Pollard’s Rho Algorithm

State-of-the-art for factorizing **64-bit integers**.

🔍 Idea

Randomized algorithm using Floyd cycle detection to find non-trivial factor.

⏱️ Complexity

~ $O(n^{1/4})$

✔️ Use-case

- Factorizing large numbers up to 10^{18}
- Used along with Miller–Rabin for primality test.

✅ V. ADVANCED NUMBER THEORY RELATED TO PRIME COMPUTATIONS

---

1. Euler Totient (φ(n)) Using Prime Factorization

If

$$n = p_1^{a_1} p_2^{a_2} ... p_k^{a_k}$$

Then

$$\phi(n) = n \prod (1 - 1/p_i)$$

Use-case

- modular inverse
- number theory DP
- counting reduced fractions

2. Sieve for Euler Phi

Compute φ for all numbers up to N in O(N log log N) or O(N).

Useful in:

- divisor-based DP
- mobius inversion
- multiplicative functions

---

3. Mobius Function (μ(n)) Sieve

Used in advanced number theory:

- multiplicative functions
- inclusion–exclusion
- counting coprime pairs

---

4. Prime Counting Function π(N)

Count of primes ≤ N.

Algorithms:

- Simple sieve → prefix count
- Meissel–Lehmer method (advanced)
- Lagarias–Miller–Odlyzko method

✅ VI. SPECIALIZED PRIMES AND CONCEPTS

1. Prefix prime sieve with cumulative sums

Useful for answering queries like:

- “how many primes in range [L, R]?”

2. Twin primes, prime gaps

Use sieve → check differences.

3. Goldbach-related problems

Need fast prime checking + sieve.

4. Circular primes, palindromic primes

Use primality + string manipulation.

SUMMARY TABLE (BEST ALGO FOR EACH USE-CASE)

Task	Best Algorithm
Check if *one number* ≤ 1e12 is prime	Trial division (optimized)
Check if one **64-bit number** is prime	Miller–Rabin
Generate all primes ≤ 1e7	Sieve of Eratosthenes
Generate primes ≤ 1e8	Bitset sieve
Generate primes and SPF array	Linear sieve
Factorize a number ≤ 1e7 repeatedly	SPF sieve
Factorize a number up to 1e18	Pollard Rho + Miller Rabin
Primes in range [L, R]	Segmented sieve
Compute φ, μ for all numbers	Linear sieve variants

If you want, I can provide:

✔️ Code for every algorithm (Python + C++)

✔️ Diagrams showing sieve operations

✔️ Comparison charts

✔️ Implementation templates for CP

Just tell me **“Give code”** or ask for a specific algorithm.

GIVE CODE IMPLEMENTATIONS FOR ALL OF THEM FRAME PROBLEMS FOR EACH CASE BY YOURSELF THEN WITH PRECISE CONSTRAINT
Code for every algorithm (Python + C++)
✔️ Diagrams showing sieve operations
✔️ Comparison charts
✔️ Implementation templates for CP
Just tell me “Give code” or ask for a specifi

Thought for 7s >

Prime-algorithms-implementations (python + C++) Copy Edit Download

Prime Algorithms — Implementations, CP Templates, Diagrams, Comparison

Comprehensive code implementations (Python + C++), framed CP-style problems with precise constraints, complexity, diagrams, and quick templates.

Table of contents

1. Trial Division (basic)
2. Optimized Trial Division (6k±1)
3. Sieve of Eratosthenes
4. Segmented Sieve
5. Linear Sieve (Euler) + SPF
6. SPF-based Factorization (using linear sieve)
7. Miller–Rabin (deterministic for 64-bit)
8. Pollard's Rho (factorization for large 64-bit)

9. Euler Totient sieve (phi for all up to N)
10. Mobius function sieve (μ up to N)
11. Comparison chart & when to use
12. Competitive programming templates (Python + C++)

> **Note:** Each algorithm section contains: short description, framed CP problem, **precise constraints**, complexity, Python implementation, C++ implementation.

---

1) Trial Division (basic)

Idea: Check divisibility by `2..sqrt(n)`.

CP frame: Given `n (1 ≤ n ≤ 10^12)`, determine whether `n` is prime.

Constraints: single `n`, up to `1e12`.

Complexity: O(√n)

Python

```
# trial_division.py
import sys
import math

def is_prime(n: int) -> bool:
    if n < 2:
        return False
    if n % 2 == 0:
        return n == 2
    r = int(math.isqrt(n))
    for i in range(3, r+1, 2):
        if n % i == 0:
            return False
    return True

if __name__ == '__main__':
    n = int(sys.stdin.readline())
```

C++

```cpp
// trial_division.cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

bool is_prime(long long n){
    if(n < 2) return false;
    if(n % 2 == 0) return n == 2;
    long long r = floor(sqrt((long double)n));
    for(long long i = 3; i <= r; i += 2)
        if(n % i == 0) return false;
    return true;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    long long n; if(!(cin >> n)) return 0;
    cout << (is_prime(n) ? "YES\n" : "NO\n");
}
```

---

2) Optimized Trial Division (6k ± 1)

Idea: After handling 2 and 3, check numbers of form `6k ± 1` up to sqrt(n).

CP frame: Same as above but for many queries `q ≤ 1e5`, each `n_i ≤ 1e12`.

Constraints: sum of queries large → reduce constant factors.

Complexity: ~ O(√n / 3)

Python

```python
# trial_6k1.py
import sys, math

def is_prime(n: int) -> bool:
    if n <= 1: return False
    if n <= 3: return True
    if n % 2 == 0 or n % 3 == 0: return False
    r = int(math.isqrt(n))
    i = 5
    while i <= r:
        if n % i == 0 or n % (i+2) == 0:
            return False
        i += 6
    return True

if __name__ == '__main__':
    input_data = sys.stdin.read().strip().split()
    it = iter(input_data)
    q = int(next(it))
    out = []
    for _ in range(q):
        out.append("YES" if is_prime(int(next(it))) else '
    print("\n".join(out))
```

C++

```
// trial_6k1.cpp
#include <bits/stdc++.h>
using namespace std; using ll = long long;

bool is_prime(ll n){
    if(n <= 1) return false;
    if(n <= 3) return true;
    if(n % 2 == 0 || n % 3 == 0) return false;
    for(ll i = 5; i * i <= n; i += 6)
        if(n % i == 0 || n % (i + 2) == 0) return false;
    return true;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int q; if(!(cin >> q)) return 0;
    while(q--){ long long n; cin >> n; cout << (is_prime(n)
}
```

---

3) Sieve of Eratosthenes

Idea: Mark multiples of primes; start crossing out from p*p.

CP frame: Precompute primes ≤ `N` then answer `q` queries: "is x prime?" or count primes in ranges.

Constraints: `N ≤ 1e8` (memory limit permitting; typical safe N ≤ 5e7–1e8 depending on memory). Use bitset/packed representation to reduce memory.

Complexity: O(N log log N), space O(N).

Python (simple)

```python
# sieve_basic.py
import sys

def sieve(n):
    is_prime = [True] * (n+1)
    is_prime[0:2] = [False, False]
    p = 2
    import math
    r = int(math.isqrt(n))
    while p <= r:
        if is_prime[p]:
            step = p
            start = p*p
            is_prime[start:n+1:step] = [False] * (((n - sta
        p += 1
    primes = [i for i, v in enumerate(is_prime) if v]
    return primes, is_prime

if __name__ == '__main__':
    n = int(sys.stdin.readline())
    primes, is_prime_arr = sieve(n)
    print(len(primes))
```

C++ (fast, using vector)

```cpp
// sieve_eratosthenes.cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

vector<int> sieve(int n){
    vector<char> is_prime(n+1, true);
    is_prime[0] = is_prime[1] = false;
    int r = floor(sqrt((long double)n));
    for(int p = 2; p <= r; ++p) if(is_prime[p]){
        for(long long j = 1LL*p*p; j <= n; j += p) is_prin
    }
    vector<int> primes;
    for(int i = 2; i <= n; ++i) if(is_prime[i]) primes.pus
    return primes;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n; if(!(cin >> n)) return 0;
    auto primes = sieve(n);
    cout << primes.size() << '\n';
```

Diagram (ASCII):

```
Index: 2 3 4 5 6 7 8 9 10 11 ...
Start: P P C P C P C C  C  P  ...
Mark multiples: when p=2 -> mark 4,6,8,10
when p=3 -> mark 9,12
... continue to sqrt(N)
```

---

4) Segmented Sieve

Idea: Sieve primes in [L,R] using primes up to √R.

CP frame: Print all primes in segment `[L, R]` where `1 ≤ L ≤ R ≤ 10^{12}`, `R-L ≤ 10^6`.

Complexity: O((R-L+1) log log R + sqrt(R) log log sqrt(R))

Python

```
# segmented_sieve.py
import sys, math

def simple_sieve(limit):
    is_prime = [True] * (limit+1)
    is_prime[0:2] = [False, False]
    for i in range(2, int(math.isqrt(limit))+1):
        if is_prime[i]:
            for j in range(i*i, limit+1, i):
                is_prime[j] = False
    return [i for i, v in enumerate(is_prime) if v]

if __name__ == '__main__':
    L, R = map(int, sys.stdin.readline().split())
    if L < 2: L = 2
    limit = int(math.isqrt(R)) + 1
    primes = simple_sieve(limit)
    segment = [True] * (R - L + 1)
    for p in primes:
        start = max(p*p, ((L + p - 1)//p) * p)
        for j in range(start, R+1, p):
            segment[j - L] = False
    for i, is_p in enumerate(segment):
        if is_p:
            print(L + i)
```

C++

```
// segmented_sieve.cpp
#include <bits/stdc++.h>
using namespace std; using ll = long long;

vector<int> simple_sieve(int limit){
    vector<char> mark(limit+1, true);
    mark[0]=mark[1]=false;
    for(int i=2;i*i<=limit;i++) if(mark[i])
        for(int j=i*i;j<=limit;j+=i) mark[j]=false;
    vector<int> primes;
    for(int i=2;i<=limit;i++) if(mark[i]) primes.push_back(
    return primes;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    long long L,R; if(!(cin>>L>>R)) return 0;
    if(L<2) L=2;
    int limit = floor(sqrt((long double)R)) + 1;
    auto primes = simple_sieve(limit);
    vector<char> is_prime(R-L+1, true);
    for(int p: primes){
        long long start = max(1LL*p*p, ((L + p - 1)/p) * 1L
        for(long long j = start; j <= R; j += p) is_prime[j
    }
    for(long long i = L; i <= R; ++i) if(is_prime[i-L]) cou
}
```

---

5) Linear Sieve (Euler) + SPF

Idea: Generate primes and smallest prime factor (spf) in O(N).

CP frame: Precompute primes and spf up to `N = 5e6`. Then answer `q ≤ 1e5` queries factorizing numbers `≤ N`.

Complexity: O(N)

Python

```python
# linear_sieve.py
import sys

def linear_sieve(n):
    spf = [0] * (n+1)
    primes = []
    for i in range(2, n+1):
        if spf[i] == 0:
            spf[i] = i
            primes.append(i)
        for p in primes:
            v = i * p
            if v > n: break
            spf[v] = p
            if p == spf[i]: break
    return primes, spf

if __name__ == '__main__':
    n = int(sys.stdin.readline())
    primes, spf = linear_sieve(n)
    print(len(primes))
```

C++

```cpp
// linear_sieve.cpp
#include <bits/stdc++.h>
```

```
using namespace std; using ll = long long;

pair<vector<int>, vector<int>> linear_sieve(int n){
    vector<int> spf(n+1, 0), primes;
    for(int i=2;i<=n;i++){
        if(spf[i] == 0){ spf[i] = i; primes.push_back(i);
        for(int p: primes){
            long long v = 1LL * p * i; if(v > n) break;
            spf[v] = p;
            if(p == spf[i]) break;
        }
    }
    return {primes, spf};
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n; if(!(cin>>n)) return 0;
    auto res = linear_sieve(n);
    cout << res.first.size() << '\n';
}
```

6) SPF-based Factorization (using linear sieve)

Idea: Using `spf[]`, factorize in `O(log n)` per number.

CP frame: Factorize `t ≤ 1e5` numbers each ≤ `N` (precomputed) and print prime factors and exponents.

Python

```
# factorize_with_spf.py
import sys

def factorize(x, spf):
    res = []
    while x > 1:
        p = spf[x]
        cnt = 0
        while x % p == 0:
            x //= p
            cnt += 1
        res.append((p,cnt))
    return res
```

C++

```
// factorize_with_spf.cpp
#include <bits/stdc++.h>
using namespace std; using ll = long long;

vector<pair<int,int>> factorize_with_spf(int x, const vecto
    vector<pair<int,int>> res;
    while(x > 1){
        int p = spf[x]; int cnt = 0;
        while(x % p == 0){ x /= p; ++cnt; }
        res.push_back({p,cnt});
    }
    return res;
}
```

---

7) Miller–Rabin (deterministic for 64-bit)

Idea: Strong probable prime test using modular exponentiation.

CP frame: Test primality for `n` up to `2^64-1` in multiple testcases.

Deterministic bases for 64-bit: `{2, 3, 5, 7, 11, 13}` or stronger set `{2, 325, 9375, 28178, 450775, 9780504, 1795265022}`.

Complexity: O(k * log^3 n)

Python

```
# miller_rabin.py
import sys
import random

def is_probable_prime(n: int) -> bool:
    if n < 2: return False
    small_primes = [2,3,5,7,11,13,17,19,23,29]
    for p in small_primes:
        if n % p == 0:
            return n == p
    # write n-1 as d*2^s
    d = n - 1
    s = 0
    while d % 2 == 0:
        d //= 2; s += 1
    # bases — deterministic for 64-bit
    bases = [2, 325, 9375, 28178, 450775, 9780504, 17952650
    for a in bases:
        if a % n == 0: continue
        x = pow(a, d, n)
        if x == 1 or x == n-1: continue
        skip_to_next_n = False
        for _ in range(s-1):
            x = (x * x) % n
            if x == n-1:
                skip_to_next_n = True
                break
        if skip_to_next_n: co
```

Done — I created a full **document** titled **"Prime-Algorithms-Implementations (Python + C++)"** on the canvas with:

- Python and C++ code for every algorithm you asked for (trial division, optimized trial, Eratosthenes, segmented sieve, linear sieve + SPF, SPF factorization, Miller–Rabin, Pollard's Rho, phi & mobius sieves).
- CP-style framed problems with **precise constraints**.
- Complexity notes, ASCII diagrams, comparison chart, and CP templates.

Open the canvas (left panel) to view the complete implementations and explanations. Want downloadable single-file archives for the Python and C++ code, or should I add runnable tests / sample inputs & outputs inside the document now?

NOW GO IN DETAIL IN Sieve of Eratosthenes give its code implementaion most commonly used and all things

Thought for a couple of seconds >

Sieve of Eratosthenes — deep dive (everything you'll actually need)

Nice — let's fully unpack the Sieve of Eratosthenes: how it works, why it's fast, practical optimizations, memory/time tradeoffs, common pitfalls, CP templates, and **multiple production-ready code implementations** (Python + C++): simple, odd-only, and memory / bit-packed. I'll also show sample problems, complexity, and testing tips.

---

1) What it is (short)

Sieve of Eratosthenes enumerates all primes ≤ `N` by iteratively marking multiples of each prime starting from its square. Each composite is marked by its smallest prime factor at or before the iteration that uses that prime.

Key facts:

- Start with boolean array `is_prime[0..N] = true` (except 0/1).
- For `p` from 2 to `√N`: if `is_prime[p]` then mark `p*p, p*p+p, p*p+2p, ...` as composite.
- After the loop, indices still `true` are primes.

---

2) Complexity & memory

- Time: `O(N log log N)` (practical constant is small).
- Space: `O(N)` bits/booleans.
- If you store booleans as bytes, memory ≈ `N` bytes. If bit-packed, memory ≈ `N/8` bytes.

Practical limits (on typical machines / contest env):

- Python: comfortable up to `N ~ 5e7` maybe too slow/mem-hungry; `N ≤ 1e7` safe.

- C++ (optimized): `N ≤ 1e8` with ~100 MB memory if using 1 byte per entry; bit-packed allows `N` up to several 1e8.

---

3) Correctness notes (important)

- Marking from `p*p` is correct because smaller multiples `< p*p` have a smaller prime factor already processed.
- Use integer type with enough range for `p*p` (if `p` is int and `p*p` can overflow, cast to 64-bit).
- When `N` small (<2) handle edge cases: no primes.

---

4) Practical optimizations (what to use in CP)

A. **Start from p*p** — always do this.

B. **Skip even numbers** (store only odds). Cuts memory/time ≈ 2x.

C. **Use a byte array (vector<char>) rather than Python list of booleans** in C++ and in Python use `bytearray` or `array('b')` / `memoryview` for speed.

D. **Bit-packed sieve**: pack 8 flags per byte or use `std::vector<uint64_t>` as bitset. Great memory savings; operations are more complex but fast if well implemented.

E. **Wheel / 6k±1**: further reduces constant factors by skipping numbers divisible by 2 and 3 (more complex). Usually skip only evens (B) is enough.

F. **Cache friendliness**: block sieving (segmented sieve) when `N` doesn't fit L1/L2 cache. For huge N, segmented sieve is better.

G. **Linear sieve** (Euler): different algorithm that runs in `O(N)` and can output SPF array. If you need SPF or factorization quickly for many numbers ≤ N, use linear sieve.

---

5) Implementations (Python + C++)

I give three common implementations for each language:

- Simple/clear (good for learning / small N)
- Odd-only optimized (practical CP)
- Bit-packed optimized C++ (memory-efficient, fast for large N)

Python — 1) Simple Sieve (clear)

python Copy code

```python
# sieve_simple.py
import math, sys

def sieve_simple(n):
    if n < 2:
        return []
    is_prime = bytearray(b'\x01') * (n + 1)  # small speed & mem win vs
    is_prime[0:2] = b'\x00\x00'
    r = int(math.isqrt(n))
    for p in range(2, r + 1):
        if is_prime[p]:
            step = p
            start = p * p
            is_prime[start:n+1:step] = b'\x00' * (((n - start)//step) +
    return [i for i in range(2, n+1) if is_prime[i]]

if __name__ == "__main__":
    n = int(sys.stdin.readline().strip())
    primes = sieve_simple(n)
    print(len(primes))
    # print(primes[:50])  # debug
```

Notes:

- `bytearray` is faster & smaller than Python `list` of bools.
- For marking slices we create a bytes object repeated; this is efficient.

Python — 2) Odd-only Sieve (practical)

Store only odd numbers: index `i` represents number `2*i + 1`. Saves ≈50% memory/time.

python Copy code

```python
# sieve_odds.py
import math, sys
```

```python
def sieve_odd(n):
    if n < 2:
        return []
    # Only odds stored: index 0 -> 1, index 1 -> 3, index 2 -> 5, ...
    size = (n + 1) // 2  # number of odd entries up to n
    is_odd_prime = bytearray(b'\x01') * size
    is_odd_prime[0] = 0  # 1 is not prime
    r = int(math.isqrt(n))
    for i in range(1, (r // 2) + 1):  # i corresponds to p = 2*i + 1
        if is_odd_prime[i]:
            p = 2 * i + 1
            # start marking from p*p: index = (p*p - 1)//2
            start = (p * p - 1) // 2
            step = p
            is_odd_prime[start::step] = b'\x00' * (((size - 1 - start)/
    primes = [2] + [2*i + 1 for i in range(1, size) if is_odd_prime[i]
    return primes

if __name__ == "__main__":
    n = int(sys.stdin.readline().strip())
    primes = sieve_odd(n)
    print(len(primes))
```

Why this is good:

- Simple mapping math.
- Reduces memory/time ~50% vs simple sieve.

C++ — 1) Simple Sieve (clear & safe)

cpp Copy code

```cpp
// sieve_simple.cpp
#include <bits/stdc++.h>
using namespace std;

vector<int> sieve_simple(int n){
    if(n < 2) return {};
    vector<char> is_prime(n+1, true);
    is_prime[0] = is_prime[1] = false;
    int r = (int)floor(sqrt((long double)n));
    for(int p = 2; p <= r; ++p){
        if(is_prime[p]){
            long long start = 1LL * p * p;
            for(long long j = start; j <= n; j += p)
                is_prime[(int)j] = false;
        }
    }
    vector<int> primes;
    for(int i = 2; i <= n; ++i) if(is_prime[i]) primes.push_back(i);
    return primes;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n; if(!(cin >> n)) return 0;
    auto primes = sieve_simple(n);
    cout << primes.size() << "\n";
}
```

Notes:

- Use `vector<char>` for memory & speed.
- Use `long long` for `p * p` to avoid overflow.

C++ — 2) Odd-only Sieve (fast & commonly used)

cpp Copy code

```cpp
// sieve_odds.cpp
#include <bits/stdc++.h>
using namespace std;

// index i (i>=1) -> number 2*i + 1
vector<int> sieve_odd(int n){
    if(n < 2) return {};
    int size = (n + 1) / 2; // includes index 0 for 1
    vector<char> is_odd_prime(size, true);
    is_odd_prime[0] = false; // 1 isn't prime
    int r = (int)floor(sqrt((double)n));
    for(int i = 1; 2*i + 1 <= r; ++i){
        if(is_odd_prime[i]){
            int p = 2*i + 1;
            long long start = 1LL*p*p;
            int start_idx = (int)((start - 1) / 2);
            for(int j = start_idx; j < size; j += p)
                is_odd_prime[j] = false;
        }
    }
    vector<int> primes; primes.push_back(2);
    for(int i = 1; i < size; ++i) if(is_odd_prime[i]) primes.push_back(
    return primes;
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n; if(!(cin >> n)) return 0;
    auto primes = sieve_odd(n);
    cout << primes.size() << "\n";
}
```

Why use this in CP:

- Good balance: easy to implement, ~2x saving vs full sieve.
- No bit-twiddling required.

C++ — 3) Bit-packed Sieve (memory efficient & fast)

Use 64-bit words to pack bits; operations rely on arithmetic to mark multiples. This is more advanced but common in high-performance contest setups.

I provide a clear and maintainable bit-packed version (only odds packed) — typical fast implementation:

cpp Copy code

```cpp
// sieve_bitpacked.cpp
#include <bits/stdc++.h>
using namespace std;
using ull = unsigned long long;

// Only odds stored. index i -> number (2*i + 1)
// We'll pack bits into vector<ull>, 64 bits per word.
struct BitSieve {
    int n;
    int size; // number of odd entries
    vector<ull> bits;
    BitSieve(int n_=0){ init(n_); }
    void init(int n_){
        n = n_;
        if(n < 2){ size = 0; bits.clear(); return; }
        size = (n + 1) / 2;
        int words = (size + 63) >> 6;
        bits.assign(words, ~0ULL); // set all 1 => assume prime
        // mark index 0 (number 1) as not-prime
        bits[0] &= ~1ULL;
    }
    inline void clear_bit(int idx){ bits[idx >> 6] &= ~(1ULL << (idx &
    inline bool test_bit(int idx) const { return (bits[idx >> 6] >> (i

    vector<int> collect_primes(){
        vector<int> primes;
        if(n >= 2) primes.push_back(2);
        for(int i = 1; i < size; ++i) if(test_bit(i)) primes.push_back(
        return primes;
    }

    void run(){
        if(size == 0) return;
        int r = (int)floor(sqrt((double)n));
        for(int i = 1; 2*i + 1 <= r; ++i){
            if(test_bit(i)){
                int p = 2*i + 1;
                long long start = 1LL*p*p;
                int idx = (int)((start - 1) / 2);
                // mark bits idx, idx + p, idx + 2p, ...
                for(int j = idx; j < size; j += p) clear_bit(j);
            }
        }
    }
};

int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n; if(!(cin >> n)) return 0;
    BitSieve bs(n);
    bs.run();
    auto primes = bs.collect_primes();
    cout << primes.size() << "\n";
}
```

Comments:

- This is efficient in memory: ~ `N/16` bytes (since only odds stored and bits packed).
- `clear_bit/test_bit` are branch-light and fast.
- For very large `N` consider segmented sieving instead: this bit-packed sieve still requires `O(N)` contiguous memory.

---

6) Variations & related algorithms

- **Segmented sieve**: same idea applied on blocks — necessary when `N` too large to store whole array or when generating primes in `[L,R]`. (I covered it earlier in the document.)
- **Linear (Euler) sieve**: different algorithm producing primes and `spf[]` in `O(N)`. Use when you need smallest prime factor for every number.
- **Wheel sieves**: skip numbers divisible by first k primes (e.g., 2,3 → 6k±1). Slightly complex, used for micro-optimizations.
- **Parallel sieving**: split segments across threads (requires attention to memory and marking overlaps).

---

7) Use-cases (where Eratosthenes is the right tool)

- Precompute primes up to N for many queries (is-prime, prime counts, prime lists).
- Precompute prime list to use later in segmented sieve, factorization of small numbers, phi computation, mobius computation.
- Competitive programming problems like:
  - Count primes ≤ N (multiple queries).
  - Sum of primes ≤ N.
  - Prime gaps & twin primes within range.
  - Use primes to factor many small numbers quickly when combined with SPF.

---

8) Framed CP problems with precise constraints (examples)

1. **Count primes**

   Input: `N (1 ≤ N ≤ 10^8)` — output number of primes ≤ N.

   Use: C++ bit-packed or odd-only sieve.
2. **Primes in range**

   Input: `L, R` with `1 ≤ L ≤ R ≤ 10^12` and `R-L ≤ 10^7`.

   Use: segmented sieve (precompute primes up to `√R` with Eratosthenes).
3. **Multiple primality queries**

   Input: `q (≤ 5e5)`, then `q` numbers `x_i ≤ 1e7`. Answer whether each is prime.

   Use: sieve up to max x and answer O(1) queries.

Each problem requires selecting the right sieve variant and memory layout.

---

9) Common pitfalls & how to avoid them

- **Overflow of p*p**: cast to 64-bit before squaring (C++: `1LL * p * p`).
- **Wrong slice count in Python marking**: when using slice assignment `is_prime[start:n+1:step] = ...`, ensure the right length calculation; using the pattern I supplied avoids mistakes.
- **Using `vector<bool>` in C++**: `vector<bool>` is a specialisation that may be slower; prefer `vector<char>` or bit-packed `vector<ull>`.
- **Assuming boolean takes 1 bit**: typically booleans take 1 byte in arrays, so memory is larger than `N/8`.
- **Not handling n < 2**: handle small `n` explicitly.

---

10) Testing & benchmarking tips

- Test correctness on small `N` by comparing to known primes.
- Edge tests: `N=0,1,2,3`, `N=10`, `N=10^6`.
- Large test: time-run `N=5e7` (if your environment allows) for practical speed test.
- Measure memory: `sizeof` arrays or use OS tools.
- If C++ is too slow for large N in contest, use bit-packed/odd-only version.

---

11) Diagrams (how marking proceeds)

Simple visual (N=30):

mathematica

```
Numbers: 2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
Start:   P  P  C  P  C  P  C  C  C  P  C  P  C  C  C  P  C  P  C  C  C

p=2: mark 4,6,8,10,12,14,16,18,20,22,24,26,28,30
p=3: mark 9,12,15,18,21,24,27,30 (12,18,24,30 already marked)
p=4: skip (composite)
p=5: mark 25,30
...
Stop p when p > sqrt(30) ≈ 5
Remaining unmarked: primes {2,3,5,7,11,13,17,19,23,29}
```

Odd-only mapping:

- Index i → number 2*i+1
- Marking arithmetic converts to indexes accordingly.

---

12) Quick CP template (use the odd-only C++ sieve in most contests)

cpp

```cpp
// cp_sieve_template.cpp
#include <bits/stdc++.h>
using namespace std;

struct Sieve {
    int n;
    vector<char> is_prime; // full sieve, or use odd-only variant
    Sieve(int n_): n(n_), is_prime(n_+1, true) {
        if(n >= 0) is_prime[0] = false;
        if(n >= 1) is_prime[1] = false;
        int r = (int)floor(sqrt((double)n));
        for(int p=2;p<=r;++p) if(is_prime[p]) {
            for(long long j = 1LL*p*p; j <= n; j += p) is_prime[(int)j]
        }
    }
    bool check(int x) const { return x >= 0 && x <= n ? is_prime[x] : f
};
```

```
int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n; cin >> n;
    Sieve s(n);
    int q; cin >> q;
    while(q--){
        int x; cin >> x;
        cout << (s.check(x) ? "YES\n" : "NO\n");
    }
    return 0;
}
```

Use `Sieve` to precompute once and answer many queries in O(1).

---

13) When to choose something else

- Need SPF for all numbers → use **Linear sieve** (Euler) — gets `spf[]` in `O(N)`.
- N too large for memory → use **Segmented sieve** (only store a block).
- Need to test primality for single gigantic numbers (≥ 64-bit) → use **Miller–Rabin** or combined MR + Pollard's Rho for factorization.

---

14) Short FAQ

Q: Should I always use bit-packed?

A: If memory matters or `N` is large (≥ 1e8) use bit-packed. For simpler code and speed-to-write, use odd-only `vector<char>`.

Q: Which is fastest in practice?

A: C++ bit-packed / odd-only > C++ simple > Python odd-only > Python simple.

Q: Is the linear sieve always better?

A: Linear sieve is `O(N)` and great if you need SPF or primes+SPF. For basic prime listing, Eratosthenes with good constants is simpler and competitive.

---

15) Example runs / sample inputs

Python odd-only:

makefile Copy code

```makefile
Input:
30
Output:
10
(primes are 2 3 5 7 11 13 17 19 23 29)
```

C++ bit-packed:

makefile Copy code

```makefile
Input:
10000000
Output:
664579   // number of primes <= 10^7
```

If you want, next I can:

- Paste a single downloadable ZIP with all code files and a README with run instructions.
- Produce a visual interactive diagram / SVG that animates the marking (helpful in teaching).