ChatGPT 5.2

how to check if number is prime
give all the algrithm from basic to advance
give thier all proof(code in cpp)
all methods to determine number is prime or not
basic to advance level algorithms

Below is a **complete, structured guide** to **checking whether a number is prime**, starting from **basic ideas** to **advanced, competitive-programming / number-theory algorithms**.

I'll give for **each method**:

- 💡 **Idea**
- 📐 **Why it works (proof / reasoning)**
- ⏱️ **Time complexity**
- 💻 **C++ code**

1️⃣ Definition (Base Concept)

A number **n > 1** is **prime** if it has **exactly two divisors**:

- 1
- itself

2️⃣ Method 1 — Brute Force (Check all divisors)

💡 Idea

Try dividing `n` by every integer from `2` to `n-1`.

📐 Proof

If any divisor divides `n`, then `n` is composite.
If none divide, only `1` and `n` divide it → prime.

⏱️ Complexity

- **Time:** `O(n)`
- **Space:** `O(1)`

💻 C++ Code

```cpp
bool isPrime(int n) {
    if (n <= 1) return false;
    for (int i = 2; i < n; i++) {
        if (n % i == 0)
            return false;
    }
    return true;
}
```

Copy code

3️⃣ Method 2 — Optimized Brute (Up to √n)

💡 Idea

If `n = a × b`, then **at least one of a or b ≤ √n**.

So, check divisors only till `√n`.

📐 Proof

If both `a > √n` and `b > √n`, then
`a × b > n` → impossible.

⏱️ Complexity

- **Time:** `O(√n)`
- **Space:** `O(1)`

💻 C++ Code

```cpp
bool isPrime(int n) {
    if (n <= 1) return false;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0)
            return false;
    }
    return true;
}
```

Copy code

4️⃣ Method 3 — Skip Even Numbers

💡 Idea

- `2` is the only even prime
- Skip all even numbers

📐 Proof

Any even number > 2 is divisible by 2 → composite.

⏱️ Complexity

- **Time:** `O(√n / 2)`
- **Space:** `O(1)`

💻 C++ Code

cpp

```cpp
bool isPrime(int n) {
    if (n <= 1) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;

    for (int i = 3; i * i <= n; i += 2) {
        if (n % i == 0)
            return false;
    }
    return true;
}
```

Copy code

5️⃣ Method 4 — 6k ± 1 Optimization (CP Favorite)

💡 Idea

All primes > 3 are of form:

python

```python
6k - 1 or 6k + 1
```

Copy code

📐 Proof

Any integer can be written as:

```
6k, 6k±1, 6k±2, 6k+3
```

Copy code

- `6k`, `6k±2` → divisible by 2
- `6k+3` → divisible by 3

  Only `6k±1` remain possible primes

⏱️ Complexity

- **Time:** `O(√n / 3)`
- **Space:** `O(1)`

💻 C++ Code

cpp

```cpp
bool isPrime(int n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;

    for (int i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0)
            return false;
    }
    return true;
}
```

Copy code

6️⃣ Method 5 — Precomputation Using Sieve (Multiple Queries)

💡 Idea

Mark multiples of primes as composite.

📐 Proof

If `p` is prime, then all multiples `p×k` are composite.

⏱️ Complexity

- **Precompute:** `O(n log log n)`
- **Query:** `O(1)`
- **Space:** `O(n)`

💻 C++ Code (Sieve of Eratosthenes)

```cpp
const int N = 1e6;
vector<bool> isPrime(N + 1, true);

void sieve() {
    isPrime[0] = isPrime[1] = false;
    for (int i = 2; i * i <= N; i++) {
        if (isPrime[i]) {
            for (int j = i * i; j <= N; j += i)
                isPrime[j] = false;
        }
    }
}
```

Copy code

Usage:

```cpp
if (isPrime[x]) cout << "Prime";
```

Copy code

7️⃣ Method 6 — Fermat Primality Test (Probabilistic)

💡 Idea

If `n` is prime:

```scss
a^(n−1) ≡ 1 (mod n)
```

Copy code

📐 Proof

Based on **Fermat's Little Theorem**.

⚠️ **False positives exist** (Carmichael numbers).

⏱️ Complexity

- **Time:** `O(k log n)`
- **Space:** `O(1)`

💻 C++ Code

cpp

```cpp
long long power(long long a, long long n, long long mod) {
    long long res = 1;
    while (n) {
        if (n & 1) res = (res * a) % mod;
        a = (a * a) % mod;
        n >>= 1;
    }
    return res;
}

bool isPrimeFermat(long long n, int k = 5) {
    if (n <= 4) return n == 2 || n == 3;
    for (int i = 0; i < k; i++) {
        long long a = 2 + rand() % (n - 3);
        if (power(a, n - 1, n) != 1)
            return false;
    }
    return true;
}
```

Copy code

8️⃣ Method 7 — Miller–Rabin (Industry Standard)

💡 Idea

Improves Fermat using **strong witnesses**.

📐 Proof

Uses number-theoretic properties of modular exponentiation.

Deterministic for 64-bit integers with fixed bases.

⏱️ Complexity

- **Time:** $O(k \log^3 n)$
- **Accuracy:** 100% for 64-bit (with fixed bases)

💻 C++ Code (Deterministic for 64-bit)

cpp

```cpp
using u128 = __uint128_t;

long long modmul(long long a, long long b, long long mod) {
    return (u128)a * b % mod;
}

long long modpow(long long a, long long d, long long mod) {
    long long res = 1;
    while (d) {
        if (d & 1) res = modmul(res, a, mod);
        a = modmul(a, a, mod);
        d >>= 1;
    }
    return res;
}

bool millerTest(long long a, long long s, long long d, long long n) {
    long long x = modpow(a, d, n);
    if (x == 1 || x == n - 1) return true;
    for (int i = 1; i < s; i++) {
        x = modmul(x, x, n);
        if (x == n - 1) return true;
    }
    return false;
}

bool isPrime(long long n) {
    if (n < 2) return false;
    for (long long p : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}) {
        if (n == p) return true;
        if (n % p == 0 && n != p) return false;
    }

    long long d = n - 1, s = 0;
    while ((d & 1) == 0) {
        d >>= 1;
        s++;
    }

    for (long long a : {2, 325, 9375, 28178, 450775, 9780504, 1795265022})
```

```
        if (a % n == 0) continue;
        if (!millerTest(a, s, d, n)) return false;
    }
    return true;
}
```

Copy code

9️⃣ Method 8 — AKS Primality Test (Theoretical)

💡 Idea

First **deterministic polynomial-time** primality test.

📐 Proof

Based on:

```
(x + a)^n ≡ x^n + a (mod n)
```

Copy code

⏱️ Complexity

- **Time:** `O(log^6 n)`
- ✕ Not used in CP (too slow constants)

⚠️ No practical C++ implementation

🔥 Summary Table

Method	Type	Time	Use Case
Brute Force	Deterministic	O(n)	Learning
√n Check	Deterministic	O(√n)	Single query
6k ± 1	Deterministic	O(√n)	CP
Sieve	Deterministic	O(n log log n)	Multiple queries
Fermat	Probabilistic	Fast	Quick check
Miller–Rabin	Deterministic (64-bit)	Very fast	Industry / CP

Method	Type	Time	Use Case
AKS	Deterministic	Polynomial	Theory

✅ For Competitive Programming

- Single number → **6k ± 1**
- Many queries → **Sieve**
- Very large (≤ 1e18) → **Miller–Rabin**

If you want, I can:

- 📌 Compare sieve variants
- 📌 Explain Carmichael numbers
- 📌 Show how CF / LC problems choose methods
- 📌 Convert all code to **Go / Python**

Just tell me 👍

give all codes in one file
with proper comments and formating

Below is **ONE single, clean, well-formatted C++ file** that contains **ALL prime-checking algorithms from basic → advanced**, with **proper comments**, **clear naming**, and **CP-ready structure**.

You can **copy-paste this file directly** and use any method you want.

✅ Contents of This File

✔ Brute Force
✔ √n Optimization
✔ Skip Even Numbers
✔ 6k ± 1 Method
✔ Sieve of Eratosthenes

✔ Fermat Primality Test

✔ Miller–Rabin (Deterministic for 64-bit)

✔ Notes for AKS (theory only)

---

cpp

```cpp
/*****************************************************
 *  PRIME NUMBER CHECKING – ALL METHODS (BASIC → ADV)
 *  Author: Divyansh Rawat (CP Ready)
 *
 *  This file contains:
 *   1. Brute Force Method
 *   2. √n Optimization
 *   3. Skip Even Numbers
 *   4. 6k ± 1 Optimization
 *   5. Sieve of Eratosthenes (Multiple Queries)
 *   6. Fermat Primality Test (Probabilistic)
 *   7. Miller–Rabin (Deterministic for 64-bit)
 *
 *  Usage:
 *   - For single number ≤ 1e12 → 6k ± 1
 *   - For many queries ≤ 1e7 → Sieve
 *   - For very large numbers ≤ 1e18 → Miller–Rabin
 *****************************************************/

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

using ll = long long;
using u128 = __uint128_t;

/*-----------------------------------------------
  1️⃣ Brute Force Method
  Check divisibility from 2 to n-1
-----------------------------------------------*/
bool isPrime_BruteForce(ll n) {
    if (n <= 1) return false;

    for (ll i = 2; i < n; i++) {
        if (n % i == 0)
            return false;
```

```
    }
    return true;
}

/*-----------------------------------------------------
  2️⃣ √n Optimization
  Check divisors only till sqrt(n)
-----------------------------------------------------*/
bool isPrime_Sqrt(ll n) {
    if (n <= 1) return false;

    for (ll i = 2; i * i <= n; i++) {
        if (n % i == 0)
            return false;
    }
    return true;
}

/*-----------------------------------------------------
  3️⃣ Skip Even Numbers
  Only odd divisors after checking 2
-----------------------------------------------------*/
bool isPrime_SkipEven(ll n) {
    if (n <= 1) return false;
    if (n == 2) return true;
    if (n % 2 == 0) return false;

    for (ll i = 3; i * i <= n; i += 2) {
        if (n % i == 0)
            return false;
    }
    return true;
}

/*-----------------------------------------------------
  4️⃣ 6k ± 1 Optimization (CP Favorite)
  All primes > 3 are of form 6k ± 1
-----------------------------------------------------*/
bool isPrime_6k(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;
}

/*-------------------------------------------------
  5️⃣ Sieve of Eratosthenes
  Precompute primes up to MAXN
-------------------------------------------------*/
const int MAXN = 1000000;
vector<bool> isPrimeSieve(MAXN + 1, true);

void buildSieve() {
    isPrimeSieve[0] = isPrimeSieve[1] = false;

    for (int i = 2; i * i <= MAXN; i++) {
        if (isPrimeSieve[i]) {
            for (int j = i * i; j <= MAXN; j += i) {
                isPrimeSieve[j] = false;
            }
        }
    }
}

/*-------------------------------------------------
  6️⃣ Fermat Primality Test (Probabilistic)
  Based on Fermat's Little Theorem
-------------------------------------------------*/
ll modPower(ll a, ll d, ll mod) {
    ll res = 1;
    while (d) {
        if (d & 1) res = (res * a) % mod;
        a = (a * a) % mod;
        d >>= 1;
    }
    return res;
}

bool isPrime_Fermat(ll n, int iterations = 5) {
    if (n <= 4)
        return (n == 2 || n == 3);
```

```
    for (int i = 0; i < iterations; i++) {
        ll a = 2 + rand() % (n - 3);
        if (modPower(a, n - 1, n) != 1)
            return false;
    }
    return true; // Probably prime
}

/*-------------------------------------------------
  7️⃣ Miller–Rabin (Deterministic for 64-bit)
-------------------------------------------------*/
ll modMul(ll a, ll b, ll mod) {
    return (u128)a * b % mod;
}

ll modPow(ll a, ll d, ll mod) {
    ll res = 1;
    while (d) {
        if (d & 1) res = modMul(res, a, mod);
        a = modMul(a, a, mod);
        d >>= 1;
    }
    return res;
}

bool millerTest(ll a, ll s, ll d, ll n) {
    ll x = modPow(a, d, n);
    if (x == 1 || x == n - 1)
        return true;

    for (ll i = 1; i < s; i++) {
        x = modMul(x, x, n);
        if (x == n - 1)
            return true;
    }
    return false;
}

bool isPrime_MillerRabin(ll n) {
    if (n < 2) return false;

    // Small primes check
```

```
    for (ll p : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}) {
        if (n == p) return true;
        if (n % p == 0 && n != p) return false;
    }

    ll d = n - 1, s = 0;
    while ((d & 1) == 0) {
        d >>= 1;
        s++;
    }

    // Deterministic bases for 64-bit
    for (ll a : {2, 325, 9375, 28178, 450775, 9780504, 1795265022}) {
        if (a % n == 0) continue;
        if (!millerTest(a, s, d, n))
            return false;
    }
    return true;
}

/*-------------------------------------------------
  8️⃣ AKS Primality Test (Theory Only)
  Deterministic polynomial-time algorithm.
  Not practical for competitive programming.
-------------------------------------------------*/

/*-------------------------------------------------
  MAIN FUNCTION (Sample Usage)
-------------------------------------------------*/
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    buildSieve(); // Only needed if using sieve

    ll n;
    cin >> n;

    cout << "Brute Force        : " << isPrime_BruteForce(n) << '\n';
    cout << "Sqrt Method        : " << isPrime_Sqrt(n) << '\n';
    cout << "Skip Even          : " << isPrime_SkipEven(n) << '\n';
    cout << "6k ± 1 Method      : " << isPrime_6k(n) << '\n';
```

Copy code

```
    if (n <= MAXN)
        cout << "Sieve              : " << isPrimeSieve[n] << '\n';

    cout << "Fermat Test        : " << isPrime_Fermat(n) << '\n';
    cout << "Miller-Rabin       : " << isPrime_MillerRabin(n) << '\n';

    return 0;
}
```

🔥 What You Should Use in CP

Situation	Best Method
One number ≤ 1e12	`6k ± 1`
Many queries ≤ 1e7	`Sieve`
Large number ≤ 1e18	`Miller-Rabin`

If you want next:

- ⚡ **Benchmark comparison**
- 🧠 **Proof of Miller–Rabin correctness**
- 🧩 **CF/LC problems using each method**
- 🔁 **Same file in Go / Python**

Just tell me 👍