🌟 Prime Number Counting Approaches in C++ 💡

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

🧠 What Are We Solving?

We are exploring **multiple ways to count the number of prime numbers** less than a given number `n` or in a range `[L, R]`. This includes both **basic** and **optimized** techniques, culminating in the mighty ⚔️ **Segmented Sieve**, ideal for large ranges like `1e9`.

🔢 [1] Naive Prime Check — "Brute Force Knight" ⚔️

💭 Analogy:

Think of checking if a person is unique in a room by comparing them to **everyone else** in the room. That's what this approach does.

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

int countPrimesNaive(int n) {
    int count = 0;
    for (int i = 2; i < n; i++) {
        if (isPrimeNaive(i)) count++;
    }
    return count;
}
```

🔍 **Explanation**:

- Check every number from `2` to `n-1`.
- For each number, check if it is divisible by any number less than itself.
- If no such divisor is found, it's prime.
- Count it.

🧮 **Time:** `O(N^2)` 📦 **Space:** `O(1)` 🚫 **Downside:** Extremely slow for large `n`.

🔢 [2] Square Root Optimization — "Smart Detective" 🕵️

💭 Analogy:

Why check all the way to `n-1` when we know a non-prime must have a factor ≤ √n? Just like you'd only check a few people if you knew the tallest person possible.

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

int countPrimesSqrt(int n) {
    int count = 0;
    for (int i = 2; i < n; i++) {
        if (isPrimeSqrt(i)) count++;
    }
    return count;
}
```

🔍 **Explanation**:

- A number `n` is non-prime only if it has a factor between `2` and `√n`.
- So, we only check till `i * i ≤ n`.
- Faster than naive, but still checks each number individually.

🔢 **Time:** `O(N√N)` 📦 **Space:** `O(1)` ☑️ **Better**, but still not optimal.

🧵 [3] Sieve of Eratosthenes — "Efficient Gardener" 🌱

💭 Analogy:

Imagine crossing out all the multiples of each small number to clean up a list — like removing weeds with a filter.

```
int countPrimesSieve(int n) {
    if (n <= 2) return 0;
    vector<bool> isPrime(n, true);
    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;
        }
    }
```

```
    return count(isPrime.begin(), isPrime.end(), true);
}
```

🔍 **Explanation**:

1. Start with a `true` array assuming all numbers are prime.
2. For each number `i` starting from 2:
    - If it's still marked prime:
        - Mark all multiples of `i` (starting from `i*i`) as `false`.
3. Finally, count the number of `true` entries.

🚀 **Time:** `O(N log log N)` 📦 **Space:** `O(N)` 💥 **Best for moderate values of `n` (like up to 1e7).**

🛰️ [4] Segmented Sieve — "Satellite Surveyor" 🗺️

💭 Analogy:

You want to find prime houses in a far-away city block. You use a list of known small prime addresses and rule out non-primes in your remote block.

```
vector<int> segmentedSieve(long long L, long long R) {
    long long limit = sqrt(R) + 1;

    // Step 1: Base sieve to get small primes
    vector<bool> isPrime(limit + 1, true);
    isPrime[0] = isPrime[1] = false;

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

    vector<int> basePrimes;
    for (int i = 2; i <= limit; i++) {
        if (isPrime[i]) basePrimes.push_back(i);
    }

    // Step 2: Mark primes in [L, R]
    vector<bool> isPrimeSegment(R - L + 1, true);

    for (int prime : basePrimes) {
        long long firstMultiple = max(prime * prime, (L + prime - 1) / prime *
prime);
        for (long long j = firstMultiple; j <= R; j += prime) {
```

```
                isPrimeSegment[j - L] = false;
            }
        }

        if (L == 1) isPrimeSegment[0] = false;

        vector<int> primesInRange;
        for (long long i = L; i <= R; i++) {
            if (isPrimeSegment[i - L]) primesInRange.push_back(i);
        }

        return primesInRange;
    }
```

🔍 **Explanation**:

1. Find all primes up to `√R` using a basic sieve (base primes).
2. Create a boolean array for the range `[L, R]` initialized as `true`.
3. Use base primes to mark non-primes in that range.
4. Return all indices that are still marked as prime.

🧮 **Time:** `O((R-L+1) log log √R)` 📦 **Space:** `O(R-L+1)` ✅ **Best for large ranges** (e.g., `[1e9, 1e9 + 100000]`)

🏃 [5] Master Runner — Testing All Approaches 🧪

```
void runAllApproaches(int n) {
    cout << " 🔹 Naive Approach Count      : " << countPrimesNaive(n) << " (O(N^2))\n";
    cout << " 🔹 Square Root Approach      : " << countPrimesSqrt(n) << " (O(N√N))\n";
    cout << " 🔹 Sieve of Eratosthenes     : " << countPrimesSieve(n) << "  (O(N log log N))\n";
}

void runSegmentedSieveDemo(long long L, long long R) {
    cout << "\n📦 Segmented Sieve Range [" << L << ", " << R << "]\n";
    vector<int> primes = segmentedSieve(L, R);
    cout << " 🔹 Primes in range: ";
    for (int p : primes) cout << p << " ";
    cout << "\n🔹 Count: " << primes.size() << endl;
}
```

✅ **Utility Functions**:

- Run and compare different methods on the same input.
- Useful for performance checks and learning.

🔚 [6] Main Function — Time to Run the Show 🎬

```cpp
int main() {
    Solution sol;

    int n = 30;
    sol.runAllApproaches(n);

    long long L = 100, R = 150;
    sol.runSegmentedSieveDemo(L, R);

    return 0;
}
```

🧪 **Final Test Run**:

- Run all approaches for `n = 30`.
- Test segmented sieve for range `[100, 150]`.

🎯 Summary Table

🔢 Method	🕒 Time Complexity	💾 Space	⚙️ Ideal Use Case
Naive	$O(N^2)$	O(1)	Small values of `n`
Square Root	$O(N\sqrt{N})$	O(1)	Slightly better for `n ≤ 10⁴`
Sieve of Eratosthenes	O(N log log N)	O(N)	Fast and efficient for `n ≤ 10⁷`
Segmented Sieve	$O((R-L+1) \log \log \sqrt{R})$	O(R−L+1)	Large ranges like `[1e9, 1e12]`

✨ Bonus Tips

- 🔁 Use `isPrimeNaive()` or `isPrimeSqrt()` **only for educational or small cases**.
- 🚀 For coding contests or large data, **Sieve methods are a must!**
- 📦 **Segmented Sieve** shines when the range is **huge**, and a regular sieve won't fit in memory.