1. Thuật toán vét cạn (Brute Forces)

Đây là thuật toán sử dụng kỹ thuật vét hết tất cả các số lẻ và kiểm tra tính nguyên tố của nó theo định nghĩa. Độ phức tạp $O((n\sqrt{n})/4)$.

Brute Forces

```cpp
#include <bits/stdc++.h>
using namespace std;
void BruceForces(long limit) {
    long count = 1;
    bool prime = true;
    for(long i = 3; i <= limit; i += 2) {
        for(long j = 3; j * j <= i; j += 2) {
            if (i % j == 0) {
                prime = false;
                break;
            }
        }
        if(prime) ++count;
        else prime = true;
    }
    cout << count << endl;
}
int main() {
    long limit = 10000000;
    BruceForces(limit);
    return 0;
}
```

2. Sàng Eratosthenes

Eratosthenes Cyrene (cổ Hy Lạp: 276 – 195 TCN) là Nhà toán học, địa lý, nhà thơ, vận động viên, nhà thiên văn học, sáng tác nhạc, nhà triết học. Eratosthenes là người đầu tiên tính ra chu vi trái đất và khoảng cách từ Trái đất tới Mặt trời với độ chính xác ngạc nhiên bằng phương pháp đơn giản nhất là đo bóng Mặt Trời tại hai địa điểm khác nhau trên Trái Đất.

Sàng Eratosthenes hoạt động theo tư tưởng loại bỏ dần bội số của các số nguyên tố thỏa một giới hạn nhất định. Khi kết thúc thuật toán, các số chưa bị loại bỏ chính là các số nguyên tố cần tìm.

Ta có thể liệt kê mọi số nguyên tố không quá 10^9 bằng sàng Eratosthenes, tuy nhiên chỉ hiệu quả khi $N<=10^7$.

Chi tiết thuật toán :

Bước 1: Tạo 1 danh sách các số tự nhiên liên tiếp từ 2 đến n: (2, 3, 4,..., n).

Bước 2: Giả sử tất cả các số trong danh sách đều là số nguyên tố. Trong đó, p = 2 là số nguyên tố đầu tiên.

Bước 3: Tất cả các bội số của p: 2p, 3p, 4p,... sẽ bị đánh dấu vì không phải là số nguyên tố.

Bước 4: Tìm các số còn lại trong danh sách mà chưa bị đánh dấu và phải lớn hơn p. Nếu không còn số nào, dừng tìm kiếm. Ngược lại, gán cho p giá trị bằng số nguyên tố tiếp theo và quay lại bước 3.

Khi giải thuật kết thúc, tất cả các số chưa bị đánh dấu trong danh sách là các số nguyên tố cần tìm.

Sieve of Eratosthenes

```cpp
#include <bits/stdc++.h>
using namespace std;
bool prime[10000001];
void SieveOfEratosthenes(long n) {
    memset(prime, true, sizeof(prime));
    for(long p = 2; p * p <= n; p++) {
        // Nếu prime[p] không đổi, p là số nguyên tố
        if(prime[p] == true) {
            // Loại bỏ tất cả các bội của p
            for(long i = p * p; i <= n; i += p) prime[i] = false;
        }
    }
    long count = 0;
    for(long p = 2; p <= n; p++) {
       if(prime[p]) count++;
    }
    cout << count << endl;
}
int main() {
    long n = 10000000;
    SieveOfEratosthenes(n);
    return 0;
}
```

Độ phức tạp:

Ta nhận xét vòng lặp for bên trong:

✓ Với i = 2, vòng lặp sẽ chạy N/2 lần.

✓ Với i = 3, vòng lặp sẽ chạy N/3 lần.

✓ Với i = 5, vòng lặp sẽ chạy N/5 lần.

➔ Ta có tổng số lần thực hiện: $N * (1/2 + 1/3 + 1/5 + \dots)$

➔ Độ phức tạp thuật toán: O(nloglog n).

2.1. Cải tiến Sàng Eratosthenes 1

Ta thấy rằng ngoại trừ 2, tất cả các số chẵn đều không phải là số nguyên tố. Vì vậy ta sẽ không xét đến các số chẵn trong thuật toán, khi đó không gian lưu trữ sẽ giảm xuống còn n/2. Điều này sẽ làm tăng tốc độ xử lý và rút ngắn thời gian thực hiện thuật toán.

Sieve of Eratosthenes 1

```cpp
#include <bits/stdc++.h>
using namespace std;
long n = 10000000;
bool *prime = new bool [n / 2];
void SieveOfEratosthenes(){
    memset(prime, true, n / 2);
    //cout << 2 << "\n";
    long count = 1;
    for (long long i = 3; i < n; i += 2) {
        if (prime [i / 2]) {
            // cout << i << "\n";
            count++;
            for(long long j = i * i; j < n; j += 2 * i) {
                prime [j / 2] = false;
            }
        }
    }
    cout << count << endl;
}
int main() {
    SieveOfEratosthenes();
    return 0;
}
```

2.2. Cải tiến Sàng Eratosthenes 2

Ta có thể cải tiến thêm bằng cách giảm số lượng các số cần xét xuống nữa. Ta nhận thấy không chỉ các số chẵn, mà các số là bội của 2 hoặc 3 đều không phải là nguyên tố. Do đó, ta tiến hành loại bỏ các số là bội của 2 hoặc 3 ra khỏi danh sách xét.

Các số không phải là bội của 2 hoặc 3 sẽ có bước nhảy lần lượt là +2 và +4 bắt đầu từ 5. Cụ thể ta có: 5 (+2) 7 (+4) 11 (+2) 13 (+4) 17 ... Đây là các số có khả năng là số nguyên tố. Ta sẽ giảm không gian lưu trữ từ n/2 xuống còn n/3. Dẫn đến tốc độ thuật toán tăng lên.

Sieve of Eratosthenes 2

```cpp
#include <bits/stdc++.h>
using namespace std;
long n = 10000000;
bool * prime = new bool [n / 3];
void SieveOfEratosthenes() {
    memset(prime, true, n / 3);
        //cout << 2 << "\n";
    long count = 2;
        for(long long i = 5, t = 2; i < n; i += t, t = 6 - t) {
            if (prime [i / 3]) {
                // cout << i << "\n";
                ++count;
                for (long long j = i * i, v = t; j < n; j += v * i, v = 6 - v) {
                    prime [j / 3] = false;
                }
            }
        }
        cout << count << endl;
}
int main() {
    SieveOfEratosthenes();
    return 0;
}
```

2.3. Cải tiến Sàng Eratosthenes 3

Xét code C++, ta có nhận xét sau:

- Với n = 10000, ta phải sử dụng một mảng có kích thước 40000 bytes (4 bytes hay 32 bits cho một giá trị kiểu int).

- Thay vì sử dụng 32 bits để đánh dấu một số nguyên là nguyên tố hay hợp số, tại sao ta không sử dụng 1 bit? Điều này là hoàn toàn có thể.

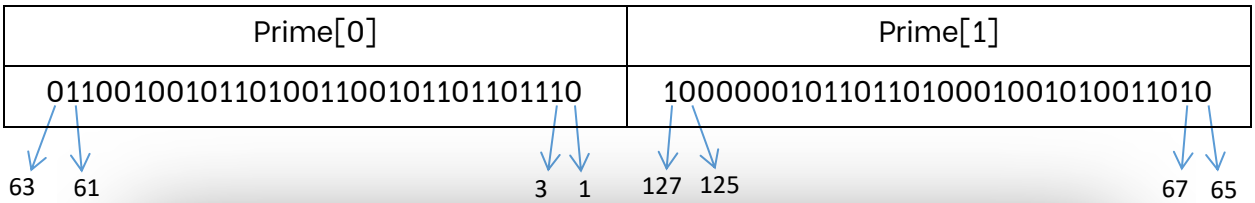
- Ta sẽ sử dụng phần tử prime[0] để lưu trữ các số từ 1 đến 32, phần tử prime[1] để lưu trữ các số từ 33 đến 64, và cứ thế tiếp tục...

Prime[0]	Prime[1]
01010000010001010001010001010110	00010100000100000100010100010000

32 31 2 1 64 63 34 33

→ Điều này sẽ làm giảm bộ nhớ, nhưng vẫn chưa cải thiện được thời gian thực hiện.

- Ta tiếp tục cải tiến. Ta nhận thấy các số chẵn đều là hợp số. Vì thế ta sẽ không lưu trữ số chẵn, mà chỉ lưu trữ các số lẻ mà thôi. Ta sẽ dùng prime[0] để lưu trữ các số 1,3,5,..., 63; prime[1] để lưu trữ các số 65, 67, 69,..., 127; và tương tự... Phương pháp này giúp ta tiết kiệm được nhiều bộ nhớ hơn và tốc độ xử lý cũng tăng lên. Cụ thể, lượng bộ nhớ sử dụng cho 1000 số nguyên là 4000/64 = 62.5 bytes.



Sieve of Eratosthenes 3

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define MAX 100000000
#define SQ 1000000
#define check(n) (prime[n >> 6] & (1 << ((n & 63) >> 1)))
#define set(n) prime[n >> 6]|=(1 << ((n & 63) >> 1))
ll prime[MAX >> 6];

void eratosthene(){
    for(ll i = 3; i <= SQ; i += 2) {
        if(!check(i)) {
            ll tmp = 2 * i;
            for(ll j = i * i; j <= MAX; j += tmp) {
                set(j);
            }
        }
    }
}
int main(){
    eratosthene();
    //cout << 2 << " ";
    ll cnt = 1;
    for(ll i = 3; i <= SQ; i += 2) {
        if(!check(i)) {
            //cout << i << " ";
            ++cnt;
        }
    }
    cout << cnt;
    return 0;
}
```

3. Sàng Atkin

Đây là một thuật toán nhanh và hiện đại để tìm tất cả các số nguyên tố thỏa một giới hạn nào đó. Thuật toán được tối ưu từ sàng Eratosthenes bằng cách đánh dấu các bội số của bình phương của các số nguyên tố chứ không phải là bội của các số nguyên tố. Thuật toán được xây dựng bởi A. O. L. Atkin và Daniel J. Bernstein.

Trong thuật toán có áp dụng phương pháp "Wheel factorization" (Bánh xe phân tích) giúp giảm đáng kể số lượng số cần xét, từ đó làm giảm lượng bộ nhớ lưu trữ.

Phương pháp Wheel factorization:

Đây là một phương pháp giúp tạo ra một danh sách nhỏ các số "cận" nguyên tố từ các công thức toán học đơn giản. Danh sách này được sử dụng để làm tham số đầu vào cho các thuật toán khác nhau trong đó có sàng Atkin.

Các bước thực hiện:

1. Tìm một vài số nguyên tố đầu tiên để làm cơ sở cho phương pháp.
2. Nhân các số nguyên tố cơ sở ở 1 lại với nhau được giá trị là n^2 chính là chu vi của bánh xe phân tích.
3. Viết các số nguyên từ 1 đến n theo một vòng tròn. Đây là vòng tròn bên trong nhất thể hiện một vòng quay của bánh xe .
4. Với các số từ 1 đến n, ta đánh dấu bỏ các bội của các số nguyên tố cơ sở vì các số này không phải là số nguyên tố.
5. Gọi x là số lượng vòng quay hiện tại, ta tiếp tục viết các số từ x*n + 1 đến x * n + n theo các vòng tròn đồng tâm với vòng tròn ở 3. Sao cho số thứ x*n + 1 kề với số thứ (x + 1) * n.
6. Lặp lại bước 5 cho tới khi đạt giới hạn kiểm tra.
7. Đánh dấu bỏ số 1.

8. Đánh dấu bỏ các số là bội của số nguyên tố cơ sở nằm trên cùng một phần rẽ quạt của bánh xe.
9. Đánh dấu bỏ các số là bội của số nguyên tố cơ sở nằm khác phần rẽ quạt với các số cơ sở, và trên các vòng tròn còn lại tương tự như bước 4.
10. Các số còn lại trên bánh xe là các số "cận" nguyên tố. Nghĩa là đa số là các số nguyên tố, còn lại xen lẫn một vài số là hợp số. Ta có thể sử dụng các thuật toán để lọc lại như sàng Eratothenes hay Atkin,...

Ví dụ minh họa:

Áp dụng phương pháp Wheel factorization với n = 2x3 = 6

1. Hai số nguyên tố cơ sở: 2 và 3.
2. n = 2 x 3 = 6.
3. Tạo vòng tròn các số : 1 2 3 4 5 6.
4. Đánh dấy loại bỏ 4 và 6 vì là bội của 2; 6 là bội của 3:

   1 2 3 4 5 6

   - x = 1
   - x * n + 1 = 1 * 6 + 1 = 7.
   - (x + 1) * n = (1 + 1) * 6 = 12.

5. Viết các số từ 7 đến 12 vào vòng tròn thứ 2, với 7 thẳng hàng với 1.

   1 2 3 4 5 6

   7 8 9 10 11 12

6. Viết lại các số từ 13 đến 18 vào vòng tròn thứ 3, với 13 thẳng hàng với 7.
7. Lặp lại cho các vòng tròn tiếp theo, ta được:

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30

8. Loại bỏ các số là bội của 2, 3 trên cùng phần rẽ quạt:

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30

9. Loại bỏ các số là bội của 2, 3 trên tất cả các vòng tròn của bánh xe:

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30

10. Danh sách kết quả chứa một số không phải là số nguyên tố là 25. Sử dụng các thuật toán khác để loại bỏ các phần tử như các sàng số nguyên tố.

2	3	5	7	11	13	17	19	23	29

Trở lại với thuật toán Sàng Atkin

Ta cần lưu ý:

- Tất cả các số dư đều là số dư trong phép chia cho 60, nghĩa là chia cho 60 lấy dư.
- Tất cả các số nguyên, kể cả x và y đều là các số nguyên dương.
- Phép đảo số là chuyển trạng thái của một số từ không là số nguyên tố thành số nguyên tố và ngược lại.
- Bánh xe phân tích được sử dụng trong Sàng Atkin là 2x3x5 = 30. Xét 2 vòng quay là 60.

Thuật toán:

1. Tạo bảng kết quả, điền vào 2, 3 và 5.

2. Tạo bảng sàng nguyên tố với các số nguyên dương, tất cả các số đánh dấu là false (không nguyên tố).

3. Với tất cả các số trong sàng:

- Nếu số đó chia 60 dư 1, 13, 17, 29, 37, 41, 49, hoặc 53, đảo đánh dấu các số có dạng $4x^2+y^2$. Hay số đó chia cho 12 dư 1 hoặc 5.
- Nếu số đó chia 60 dư 7, 19, 31, hoặc 43, đảo các số có dạng $3x^2+y^2$. Hay số đó chia cho 12 dư 7.
- Nếu số đó chia 60 dư 11, 23, 47, hoặc 59, đảo các số có dạng $3x^2-y^2$ (với $x > y$).
- Còn lại, không làm gì cả.

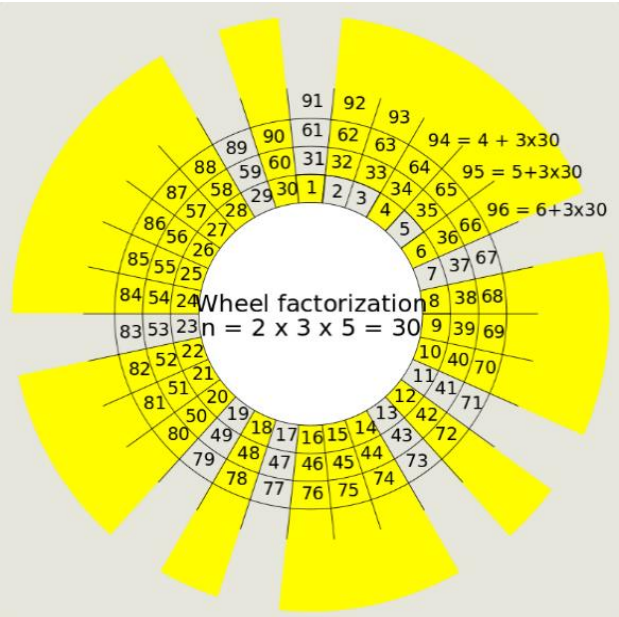
4. Bắt đầu từ số nhỏ nhất trong sàng.

5. Lấy các số tiếp theo trong sàng được đánh dấu là Prime.

6. Thêm vào danh sách kết quả.

7. Bình phương số đó và đánh dấu các bội số của bình phương số đó là không phải số nguyên tố.

8. Lặp lại bước 5 cho tới bước 8.



Sieve of Atkin

```cpp
#include <bits/stdc++.h>
using namespace std;
bool sieve[10000001];
void SieveOfAtkin(long limit) {
// 2 và 3 là số nguyên tố đã biết trước, ta đánh dấu true
    sieve[2] = true;
    sieve[3] = true;
//Đánh dấu các số có dạng phương trình bậc 2 có thể là nguyên tố,
//Đây là các giá trị còn lại khi phân tích bằng phương pháp wheel
//factorization.
    for(int x = 1; x * x < limit; x++) {
        for(int y = 1; y * y < limit; y++) {
            int n = (4 * x * x) + (y * y);
            if(n <= limit && (n % 12 == 1 || n % 12 == 5)) {
                sieve[n] ^= true;
            }
            n = (3 * x * x) + (y * y);
            if(n <= limit && n % 12 == 7) {
                sieve[n] ^= true;
            }
            n = (3 * x * x) - (y * y);
            if (x > y && n <= limit && n % 12 == 11) {
                sieve[n] ^= true;
            }
        }
    }
    //Loại bỏ tất cả các bội số của bình phương các số nguyên tố
    for(long r = 5; r * r < limit; r++) {
        if(sieve[r]) {
            for (int i = r * r; i < limit; i += r * r) {
                sieve[i] = false;
            }
        }
    }
    long count = 2;
    for(int a = 5; a < limit; a++) {
        if(sieve[a]) ++count;
    }
    cout << count << endl; // Số lượng số nguyên tố từ 1 đến n
}
int main() {
    long limit = 10000000;
    SieveOfAtkin(limit);
    return 0;
}
```

Độ phức tạp: O(n)

3.1. Cải tiến Sàng Atkin

Thuật toán đã được trình bày ở phía trên. Song ta vẫn có thể cải tiến code.

Nhận xét:

- Ở 2 vòng lặp for ban đầu ta phải xét tất cả các cặp x,y (với x*x < limit và y*y < limit). Do đó có một số cặp x, y không phù hợp với các giá trị có dạng phương trình bậc hai.
- Ở vòng lặp cuối khi đếm các số nguyên tố, ta phải xét hết tất cả các số trong đoạn.

Cải tiến:

- Ứng với mỗi phương trình bậc hai, ta sẽ xét các cặp (x, y) với các điều kiện nhất định.

Cụ thể:

```
// Xét tất cả các giá trị x (chẵn, lẻ) và y (lẻ).
for n ≤ limit, n ← 4x²+y² trong đó x ∈ {1, 2, …} và y ∈ {1, 3, …}
     if n mod 12 ∈ {1, 5}:
          prime(n) ← ¬prime(n) // Đảo đánh dấu
// Chỉ xét các giá trị x lẻ và y chẵn.
for n ≤ limit, n ← 3x²+y² trong đó x ∈ {1, 3, …} và y ∈ {2, 4, …}
     if n mod 12 = 7:
          prime(n) ← ¬prime(n) // Đảo đánh dấu
// Xét tất cả các cặp (chẵn, lẻ), (lẻ / chẵn) và x > y.
     if n mod 12 = 1:
          prime(n) ← ¬prime(n) // Đảo đánh dấu
```

- Khi loại bỏ các bội số của các số nguyên tố, ta sẽ loại bỏ từ 7 do đã xét chính xác các cặp (x,y) cần thiết ở bước trên.

- Khi đếm các số nguyên tố, ta chỉ đếm đúng các số còn lại trên bánh xe phân tích mà thôi, không xét hết các số trong đoạn, cụ thể:

```
output 2, 3, 5
for 7 ≤ n ≤ limit, n ← 60 × w + x where w ∈ {0, 1, …}, x ∈ s:
     if prime(n): output n
```

Sieve of Atkin cải tiến

```cpp
#include <bits/stdc++.h>
using namespace std;
#define boost std::ios::sync_with_stdio(false);
long limit = 100000000;
bool sieve[100000000];
void SieveOfAtkin(long limit) {
    for(int x = 1; x * x ≤ limit; x++) {
        for(int y = 1; y * y ≤ limit; y += 2) {
            int n = (4 * x * x) + (y * y);
            if (n ≤ limit && (n % 12 == 1 || n % 12 == 5)) {
                sieve[n] ^= true;
            }
        }
    }
    for(int x = 1; x * x ≤ limit; x += 2) {
        for(int y = 2; y * y ≤ limit; y += 2) {
            int n = (3 * x * x) + (y * y);
            if (n ≤ limit && n % 12 == 7) {
                sieve[n] ^= true;
            }
        }
    }
    for(int x = 2; x * x ≤ limit; x++) {
        for(int y = x - 1; y ≥ 1; y -= 2) {
            int n = (3 * x * x) - (y * y);
            if (n ≤ limit && n % 12 == 11) {
                sieve[n] ^= true;
            }
        }
    }

    //Danh dau boi cua binh phuong cac so nguyen to
    for(long r = 7; r * r < limit; r++) {
        if(sieve[r]) {
            for(int i = r * r; i < limit; i += r * r) {
                sieve[i] = false;
            }
        }
    }
    long count = 3; // 2, 3, 5 la cac so nguyen to

    //So du quan trong khi chia so n cho 60
    int s[16]={1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59};

    for(long w = 0; w * 60 ≤ limit; ++w) {
        for(int x = 0; x ≤ 15; ++x) {
            long n = 60 * w + s[x];
            if (n ≥ 7 && n ≤ limit && sieve[n]) ++count;
        }
    }
    cout << count << endl;
}
int main() {
    SieveOfAtkin(limit);
    return 0;
}
```

4. Sàng Sundaram

Đây là một thuật toán đơn giản và nhanh giải quyết bài toán tìm các số nguyên tố thỏa một giới hạn nguyên nào đó. Nó được khám phá bởi nhà toán học người Ấn Độ S.P. Sundaram năm 1934.

Ý tưởng thuật toán

❖ Bắt đầu với dãy số từ 1 đến n. Từ dãy số này, ta loại bỏ tất cả các số có dạng $i + j + 2ij$ trong đó:

- $i, j \in N, 1 \le i \le j$
- $i + j + 2ij \le n$

❖ Tất cả các số k còn lại trong dãy số sẽ được tính thành 2k + 1, sau khi tính ta được một danh sách các số nguyên tố lẻ (trừ số 2) nhỏ hơn 2n + 2.

❖ Ý tưởng này xuất phát từ việc các số nguyên tố, trừ số 2 ra, thì đều là số lẻ. Bên cạnh đó các số có dạng i + j + 2ij, khi được tính thành 2(i + j + 2ij) + 1 sẽ tạo thành một hợp số chứ không phải số nguyên tố. Do đó ta phải loại bỏ các số này đi. Cụ thể ta có:

Giả sử q là một số nguyên lẻ có dạng 2k + 1, số k sẽ bị loại bỏ khi k có dạng i + j + 2ij. Vì khi đó:

$$q = 2(i + j + 2ij) + 1 = 2i + 2j + 4ij + 1 = (2i + 1)(2j + 1)$$

➔ Rõ ràng q khi này là một hợp số.

Cách cài đặt thứ nhất (Độ phức tạp O(nlogn)):

Sieve of Sundaram 1

```cpp
#include <bits/stdc++.h>
using namespace std;
bool marked[5000001];
void SieveOfSundaram(long n) {   //Giảm n xuống 1 nửa, vì các số nguyên tố
thuộc nửa còn lại sẽ được tạo ra sau đó khi áp dụng công thức 2k+1.
    long nNew = (n - 2) / 2;
    //Loại bỏ các số có dạng i+j+2ij
    for (long long i = 1; i <= nNew; i++) {
        for (long long j = i; (i + j + 2*i*j) <= nNew; j++) {
            marked[i + j + 2*i*j] = true;
        }
    }
    //if (n > 2) cout << 2 << " ";
    long count = 1;
    for(long i = 1; i <= nNew; i++) {
        if(marked[i] == false) {
            //cout << 2*i + 1 << " ";
            count++;
        }
    }

    cout << count << endl;
}
int main() {
    long n = 10000002;
    SieveOfSundaram(n);
    return 0;
}
```

Cách cài đặt thứ hai (Độ phức tạp O(nlogn)):

Sieve of Sundaram 2

```cpp
#include <bits/stdc++.h>
using namespace std;
bool marked[10000005];
void SieveOfSundaram2(long n) {
    for(long long i = 3; i * i <= n; i += 2) {
        for(long long j = i ; i * j <= n; j += 2) {
            marked[i * j] = true;
        }
    }
    /*if (n > 2)
        cout << 2 << " ";*/
    long count = 1;
    for(long i = 3; i <= n; i += 2) {
        if(marked[i] == false) {
            //cout << 2*i + 1 << " ";
            count++;
        }
    }
    cout << count << endl;
}
int main() {
    long n = 10000000;
    SieveOfSundaram2(n);
    return 0;
}
```