Asymptotic analysis

The outer loop starts at 2 and runs until i * i <= n, which means that it runs in O(√n) time.

The inner loop starts at i*i and runs until j is less than or equal to n, marking all multiples of i as not prime. The number of iterations of the inner loop is n/i.

The inner loop is executed for each value of i in the outer loop, and since i is increasing, the number of iterations of the inner loop decreases.

Therefore, the total number of iterations of the inner loop is the sum of the number of iterations for each value of i in the outer loop, which can be represented mathematically as:

Σ(n/i), where 2 <= i <= √n

So, Σ(n/i) = n/2 + n/3 + n/4 + n/5 +.......+ n/√n
= n(1/2 + 1/3 + 1/4 + 1/5 +.......+ 1/√n)

and (1/2 + 1/3 + 1/4 +.......+ 1/√n) ≈ log(log n)

Hence, Σ(n/i) = n(log(log n))

By using the property of prime number distribution, we can approximate this sum as n log log n. Therefore, the overall time complexity of this code is `O(n log(log n)).`

The space complexity of the algorithm is `O(n)` as we are using a boolean array of size n+1 to store the prime numbers.

The biggest weakness of this algorithm is it "walks" along the memory multiple times. This is not very cache friendly because of that, the constant which is concealed in `O(n log(log n))` is comparably big.

Besides, The space complexity of the algorithm is `O(n)` as we are using a boolean array of size n+1 to store the prime numbers.

Sieving till root :

The idea is to find all the prime numbers until root of n , it will be enough just to perform the shifting only by the prime numbers, which do not exceed the root of n.

```
int n;

vector<bool> is_prime(n+1, true);

is_prime[0] = is_prime[1] = false;

for (int i = 2; i * i <= n; i++) {

    if (is_prime[i]) {

        for (int j = i * i; j <= n; j += i)

            is_prime[j] = false;

    }

}
```

Such optimization doesn't affect the complexity (by repeating the proof presented above we'll get the evaluation (`O(n log(log n)) +O(n))`

which is asymptotically the same according to the properties of logarithms), though the number of operations will reduce noticeably.

Best, Average and Worst Case :

CASES	TIME COMPLEXITY
Best Case	O(1)
Average Case	O(n log(log n))
Worst Case	O(n log(log n))

Conclusion

The Sieve of Eratosthenes is an algorithm for finding prime numbers up to a given limit. It has a time complexity of `O(n log(log n))` and it's easy to understand and implement. However, it has a space complexity of `O(n)` which can be a problem for memory-constrained systems. The Sieve of Eratosthenes remains a popular choice for finding prime numbers due to its simplicity and wide range of applicability.