Sieve of Eratosthenes

The Sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to a given limit n efficiently.

Key Idea:

Start with a list of numbers from 2 to n.

Mark multiples of each prime number starting from 2 as composite.

The remaining unmarked numbers are primes.

Algorithm

```
vector<bool> sieve(int n) {
    vector<bool> is_prime(n + 1, true);
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i <= n; i++) {
        if (is_prime[i]) {
            for (int j = 2 * i; j <= n; j += i) {
                is_prime[j] = false;
            }
        }
    }
    return is_prime;
}
```

Time Complexity: O(nloglogn) (almost linear).
Space Complexity: O(n).

Different optimizations of the Sieve of Eratosthenes

Start marking multiples from i^2 (smaller multiples of i are already marked by smaller primes).

```
vector<bool> sieve(int n) {
    vector<bool> is_prime(n + 1, true);
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i <= n; i++) {
        if (is_prime[i]) {
            for (int j = i * i; j <= n; j += i) {
                is_prime[j] = false;
            }
        }
    }
    return is_prime;
}
```

The standard Sieve of Eratosthenes has two main limitations:

1. Memory access patterns are cache-inefficient

2. Memory usage becomes problematic for very large n (e.g., n > 10^9)

1. Sieving Only Up to √n

Observation: Any composite number n must have at least one prime factor ≤ √n.

Optimization:

-> Only mark multiples of primes up to √n

-> All remaining unmarked numbers are automatically prime

Algorithm:

```
vector<bool> sieve_sqrt(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;
            }
        }
    }
    return is_prime;
}
```

Impact:

-> Reduces number of operations by about 50%

-> Maintains same asymptotic complexity O(n log log n)

-> For n = 10^6, standard sieve checks 1000M numbers, optimized version checks 10k primes

2. Odd-Only Sieve

Observation: All even numbers > 2 are composite.

Optimization:

-> Handle even numbers separately

-> Only process odd numbers in the sieve

-> Reduces memory usage by half

Algorithm:

```
vector<bool> sieve_odd_only(int n) {
    vector<bool> is_prime(n+1, true);
    is_prime[0] = is_prime[1] = false;

    // Handle even numbers
    for (int j = 4; j <= n; j += 2)
        is_prime[j] = false;

    // Only check odd numbers
    for (int i = 3; i * i <= n; i += 2) {
        if (is_prime[i]) {
            for (int j = i * i; j <= n; j += 2*i) {
                is_prime[j] = false;
            }
        }
    }
    return is_prime;
}
```

Enhanced Sieve of Eratosthenes

1.Modified Sieve for Prime Factorization

The standard Sieve of Eratosthenes can be enhanced to store the smallest prime factor (SPF) for each number. This allows:

-> Instant primality checks (sieve[x] == 0 means x is prime).

-> Efficient factorization by repeatedly dividing by sieve[x].

Algorithm


```
vector<int> preprocess_sieve(int n) {
    vector<int> sieve(n + 1, 0);
    for (int x = 2; x <= n; x++) {
        if (sieve[x] != 0) continue;  // Not prime
        for (int u = 2 * x; u <= n; u += x) {
            if (sieve[u] == 0) sieve[u] = x;  // Mark smallest prime factor
        }
    }
    return sieve;
}
```

Factorizing Numbers Using the Sieve

Given the sieve array, factorizing any number k ≤ n takes O(log k) time:

```
vector<int> factorize(int k, const vector<int>& sieve) {
    vector<int> factors;
    while (sieve[k] != 0) {  // While k is composite
        factors.push_back(sieve[k]);
        k /= sieve[k];
    }
    if (k > 1) factors.push_back(k);  // Add remaining prime
    return factors;
}
```

Example: Factorize k = 12:

sieve[12] = 2 → factors = [2], k = 12 / 2 = 6.

sieve[6] = 2 → factors = [2, 2], k = 6 / 2 = 3.

sieve[3] = 0 → push 3 → factors = [2, 2, 3].

Output: 12 = 2 × 2 × 3.

2. Sum of Divisors Sieve (O(n log n))

Problem: For all numbers from 1 to n, compute the sum of their divisors.

Algorithm:

```
vector<int> sum_of_divisors(int n) {
    vector<int> sumdiv(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        for (int j = i; j <= n; j += i) {
            sumdiv[j] += i;  // Add divisor i to all its multiples
        }
    }
    return sumdiv;
}
```

2. Biggest Prime Divisor Sieve (O(n log log n))

Problem: For all numbers from 1 to n, find their largest prime factor.

Algorithm:

```
vector<int> largest_prime_factor(int n) {
    vector<int> big(n + 1, 1);
    big[0] = big[1] = 1;
    for (int i = 2; i <= n; ++i) {
        if (big[i] == 1) {  // i is prime
            for (int j = i; j <= n; j += i) {
                big[j] = i;  // Update largest prime factor
            }
        }
    }
    return big;
}
```