UNIT-II

Sieve of Eratosthenes

Sieve of Eratosthenes is a simple and ancient algorithm that searches for all prime numbers in the given limit. It was developed by the Greek astronomer **Eratosthenes**. This algorithm is very simple to compute the prime number. In the beginning, we write all the numbers between 2 and n. We mark all appropriate multiples of 2 as a composite (because 2 is the smallest prime number), and then mark all appropriate multiples of 3, and this process runs up to n.

Algorithm of Sieve of Eratosthenes

input: an **int** n > 1

output: Each prime numbers from 2 to n

Let A be an array of Boolean values, indexed by integers 2 to n.
initially all set to **true**.
for i = 2, 3, 4, ..., not exceeding ?n **do**
if A[i] is **true**
for j = i2, i2+i, i2+2i, i2+3i, ..., not exceeding n **do**
A[j]: = **false**
return all i such that A[i] is **true**

Given a number n, print all primes smaller than or equal to n. It is also given that n is a small number.

Example:

Input: *n =10* ***Input:*** *n = 20*

Output : *2 3 5 7* ***Output:*** *2 3 5 7 11 13 17 19*

The sieve of Eratosthenes is one of the most efficient ways to find all primes smaller than n when n is smaller than 10 million

Following is the algorithm to find all the prime numbers less than or equal to a given integer n by the Eratosthene's method:

When the algorithm terminates, all the numbers in the list that are not marked are prime.

Explanation with Example:

Let us take an example when n = 50. So we need to print all prime numbers smaller than or equal to 50.

We create a list of all numbers from 2 to 50.

	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
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

According to the algorithm we will mark all the numbers which are divisible by 2 and are greater than or equal to the square of it.

	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
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Now we move to our next unmarked number 3 and mark all the numbers which are multiples of 3 and are greater than or equal to the square of it.

	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
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

We move to our next unmarked number 5 and mark all multiples of 5 and are greater than or equal to the square of it.

	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
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

We continue this process and our final table will look like below:

	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
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

So the prime numbers are the unmarked ones: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47.

Implementation:

Following is the implementation of the above algorithm. In the following implementation, a boolean array arr[] of size n is used to mark multiples of prime numbers.

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
void SieveOfEratosthenes(int n)
{
        bool prime[n + 1];
        memset(prime, true, sizeof(prime));
        for (int p = 2; p * p <= n; p++)
        {
        if (prime[p] == true) // If prime[p] is not changed, then it is a prime
          {
            for (int i = p * p; i <= n; i += p)
                prime[i] = false;
           }
        }
        for (int p = 2; p <= n; p++)  // Print all prime numbers
                if (prime[p])
                        printf("%d ",p);
}
int main()
{
        int n = 30;
        printf("Following are the prime numbers smaller than or equal to %d \n", n);
        SieveOfEratosthenes(n);
        return 0;
}
```

Output

Following are the prime numbers smaller than or equal to 30

2 3 5 7 11 13 17 19 23 29

Time Complexity: O(n*log(log(n)))

Auxiliary Space: O(n)

Sum of all Primes in a given range using Sieve of Eratosthenes

Given a range [L, R]. The task is to find the sum of all the prime numbers in the given range from L to R both inclusive.

Examples:

Input: L = 10, R = 20
Output : Sum = 60
Prime numbers between [10, 20] are:
11, 13, 17, 19
Therefore, sum = 11 + 13 + 17 + 19 = 60

Input : L = 15, R = 25
Output : Sum = 59
Prime numbers between [15, 25] are:
17,19,23
Therefore, sum = 17 + 19 + 23 = 59

A **Simple Solution** is to traverse from L to R, check if the current number is prime. If yes, add it to sum. Finally, print the sum.

An **Efficient Solution** is to use Sieve of Eratosthenes to find all primes up to a given limit. Then, compute a prefix sum array to store sum till every value before the limit. Once we have prefix array, We just need to return **prefix[R] – prefix[L-1].**

***Note**: prefix[i] will store the sum of all prime numbers from 1 to i*

Code:

```
#include <bits/stdc++.h>
using namespace std;
const int MAX = 10000;
int prefix[MAX + 1];
void buildPrefix()              // Function to build the prefix sum array
{
        bool prime[MAX + 1];
        memset(prime, true, sizeof(prime));
        for (int p = 2; p * p <= MAX; p++)
        {
        if (prime[p] == true)
        {
        for (int i = p * 2; i <= MAX; i += p) // Update all multiples of p
                prime[i] = false;
        }
        }
        prefix[0] = prefix[1] = 0;              // Build prefix array
        for (int p = 2; p <= MAX; p++)
         {
                prefix[p] = prefix[p - 1];
                if (prime[p])
                        prefix[p] += p;
        } }
```

```
int sumPrimeRange(int L, int R)    // Function to return sum of prime in range
{
        buildPrefix();
        return prefix[R] - prefix[L - 1];
}
int main()
{
        int L = 10, R = 20;
        cout << sumPrimeRange(L, R) << endl;
        return 0;
}
```

Output

60

Time Complexity: *n*log(log(n)) (where n = MAX, because we are building sieve to that limit)*

Auxiliary Space: *O(n), since extra space taken is 10000.*

Approach: Here's another approach to solve the problem using the Sieve of Eratosthenes

Algorithm:

1. Create a boolean array of size (R+1) to mark all numbers as prime initially. We will use the index of the array to represent the numbers. For example, index 2 represents the number 2, index 3 represents the number 3, and so on.
2. Mark 0 and 1 as not prime since they are not prime numbers.
   For each index i from 2 to sqrt(R), if the number at index i is marked as prime, then mark all multiples of i as not prime and this can be done by iterating over all multiples of i and marking them as not prime in the boolean array.
3. Iterate over the boolean array from index L to R, and if a number is marked as prime, add it to the sum.
4. Return the sum as the output.

Code:

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
int sieve_of_eratosthenes_sum(int L, int R)
 {
      vector<bool> is_prime(R + 1, true);    // Step 1
      is_prime[0] = is_prime[1] = false;    // Step 2
      for (int i = 2; i <= sqrt(R); i++)       // Step 3
      {
```

```
            if (is_prime[i])
            {
                for (int j = i * i; j <= R; j += i)
                {
                    is_prime[j] = false;
                }
            }
        }
        int s = 0;       // Step 4
        for (int i = L; i <= R; i++)
       {
            if (is_prime[i])
            {
                s += i;
            }
        }
        return s;        // Step 5
    }
    int main()
    {
        int L = 10;
        int R = 20;
        cout << sieve_of_eratosthenes_sum(L, R) << endl; // Output: 60
        return 0;
    }
```

Output

60

Time Complexity: O((R-L+1)log(log(R))) ,where R-L+1 is the range of numbers and log(log(R)) is the time complexity of the inner loop of the Sieve of Eratosthenes algorithm.

This is because we are iterating over the range of numbers from L to R and performing the Sieve of Eratosthenes algorithm on this range to find the prime numbers.

The time complexity of the Sieve of Eratosthenes algorithm is O(nlog(log(n))) where n is the maximum limit of the range of numbers. In our case, the maximum limit is R, so the time complexity of the Sieve of Eratosthenes algorithm is O(Rlog(log(R))). Since we are only performing the algorithm on a range of (R-L+1) numbers, the time complexity becomes O((R-L+1)log(log(R))).

Auxiliary Space: O(R+1), this is because we are storing a boolean value for each number in the range from 0 to R. In practice, this space requirement can be reduced by using a bit array instead of a boolean array to represent the numbers, which would reduce the space requirement to O(R/32) for a 32-bit system. However, the time complexity would remain the same.

Least prime factor of numbers till n

Given a number **n**, print **least prime factors** of all numbers from 1 to n. The least prime factor of an integer n is the smallest prime number that divides the number. The least prime factor of all even numbers is 2. A prime number is its own least prime factor (as well as its own greatest prime factor).
Example:

Input: 6
Output: Least Prime factor of 1: 1
Least Prime factor of 2: 2
Least Prime factor of 3: 3
Least Prime factor of 4: 2
Least Prime factor of 5: 5
Least Prime factor of 6: 2

We can use a variation of sieve of eratosrhenes to solve the above problem.

1. Create a list of consecutive integers from 2 through n: (2, 3, 4, …, n).
2. Initially, let i equal 2, the smallest prime number.
3. Enumerate the multiples of i by counting to n from 2i in increments of i, and mark them as having least prime factor as i (if not already marked). Also mark i as least prime factor of i (i itself is a prime number).
4. Find the first number greater than i in the list that is not marked. If there was no such number, stop. Otherwise, let i now equal this new number (which is the next prime), and repeat from step 3.

Below is the implementation of the algorithm, where least_prime[] saves the value of the least prime factor corresponding to the respective index.

```
#include<bits/stdc++.h>
using namespace std;
void leastPrimeFactor(int n)
{
        vector<int> least_prime(n+1, 0);
        least_prime[1] = 1;   // We need to print 1 for 1.
        for (int i = 2; i <= n; i++)
        {
                if (least_prime[i] == 0)
                {
                        least_prime[i] = i;
                        for (int j = i*i; j <= n; j += i)
                                if (least_prime[j] == 0)
                                least_prime[j] = i;
                 }
```

```
        }
    for (int i = 1; i <= n; i++)
        cout << "Least Prime factor of "<< i << ": " << least_prime[i] << "\n";
    }
int main()
{
    int n = 10;
    leastPrimeFactor(n);
    return 0;
}
```

Output

```
Least Prime factor of 1: 1
Least Prime factor of 2: 2
Least Prime factor of 3: 3
Least Prime factor of 4: 2
Least Prime factor of 5: 5
Least Prime factor of 6: 2
Least Prime factor of 7: 7
Least Prime factor of 8: 2
Least Prime factor of 9: 3
Least Prime factor of 10: 2
```

Time Complexity: O(n*log(n))
Auxiliary Space: O(n)

Prime Factorization using Sieve O(log n) for multiple queries

We can calculate the prime factorization of a number **“n”** in **O(sqrt(n))**. But O(sqrt n) method times out when we need to answer multiple queries regarding prime factorization.
An efficient method to calculate the prime factorization using O(n) space and **O(log n)** time complexity with pre-computation allowed.

Approach:

The main idea is to precompute the Smallest Prime Factor (SPF) for each number from 1 to MAXN using the sieve function. SPF is the smallest prime number that divides a given number without leaving a remainder. Then, the getFactorization function uses the precomputed SPF array to find the prime factorization of the given number by repeatedly dividing the number by its SPF until it becomes 1.

Now, after we are done with precalculating the smallest prime factor for every number we will divide our number n (whose prime factorization is to be calculated) by its corresponding smallest prime factor till n becomes 1.

Pseudo Code for prime factorization assuming
SPFs are computed :

```
PrimeFactors[]   // To store result
i = 0            // Index in PrimeFactors
while n != 1 :
      PrimeFactors[i] = SPF[n]  // SPF : smallest prime factor
   i++
   n = n / SPF[n]
```

Step-by-step approach of above idea:

- Defines a constant **MAXN** equal to **100001**.
- An integer array spf of size **MAXN** is declared. This array will store the smallest prime factor for each number up to **MAXN**.
- A function **sieve()** is defined to calculate the smallest prime factor of every number up to **MAXN** using the Sieve of Eratosthenes algorithm.
- The smallest prime factor for the number **1** is set to **1**.
- The smallest prime factor for every number from **2** to **MAXN** is initialized to be the number itself.
- The smallest prime factor for every even number from **4** to **MAXN** is set to **2**.
- The smallest prime factor for every odd number from **3** to the square root of **MAXN** is calculated by iterating over all odd numbers and checking if the number is prime.
- If a number **i** is prime, then the smallest prime factor for all numbers divisible by **i** is set to **i**.
- A function **getFactorization(int x)** is defined to return the prime factorization of a given integer **x** using the spf array.
- The **getFactorization(int x)** function finds the smallest prime factor of **x**, pushes it to a vector, and updates **x** to be the quotient of **x** divided by its smallest prime factor. This process continues until **x** becomes **1**, at which point the vector of prime factors is returned.
- In the main() function, the **sieve()** function is called to precalculate the smallest prime factor of every number up to **MAXN**. Then, the prime factorization of a sample integer **x** is found using the **getFactorization(int x)** function, and the **result** is printed to the con

Code:

```
#include "bits/stdc++.h"
using namespace std;
#define MAXN 100001
int spf[MAXN];        // stores smallest prime factor for every number
void sieve()
{
spf[1] = 1;
for (int i = 2; i < MAXN; i++)
spf[i] = i;
for (int i = 4; i < MAXN; i += 2)
```

```
spf[i] = 2;
for (int i = 3; i * i < MAXN; i++)
 {
        if (spf[i] == i)            // checking if i is prime
 {
        for (int j = i * i; j < MAXN; j += i)    // marking SPF for all numbers divisible by i
if (spf[j] == j)
spf[j] = i;
}
}
}
vector<int> getFactorization(int x)
{
vector<int> ret;
while (x != 1)
{
ret.push_back(spf[x]);
x = x / spf[x];
}
return ret;
}

int main(int argc, char const* argv[])
{
sieve();         // precalculating Smallest Prime Factor
int x = 12246;
cout << "prime factorization for " << x << " : ";
vector<int> p = getFactorization(x);        // calling getFactorization function
for (int i = 0; i < p.size(); i++)
        cout << p[i] << " ";
        cout << endl;
        return 0;
}
```

Output:

- prime factorization for 12246 : 2 3 13 157
- **Time Complexity:** O(log n), for each query (Time complexity for precomputation is not included)
  Auxiliary Space: O(1)

Euclidean algorithms (Basic and Extended)

The Euclidean algorithm is a way to find the greatest common divisor of two positive integers. GCD of two numbers is the largest number that divides both of them. A simple way to find GCD is to factorize both numbers and multiply common prime factors.

```
36   = 2 x 2 x 3 x 3
60   = 2 x 2 x 3 x 5

GCD = Multiplication of common factors
      = 2 x 2 x 3
      = 12
```

Basic Euclidean Algorithm for GCD:

The algorithm is based on the below facts.

- If we subtract a smaller number from a larger one (we reduce a larger number), GCD doesn't change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
- Now instead of subtraction, if we divide the smaller number, the algorithm stops when we find the remainder 0.

Below is a recursive function to evaluate gcd using Euclid's algorithm:

```
#include <stdio.h>
int gcd(int a, int b)    // Function to return gcd of a and b
{
        if (a == 0)
                return b;
        return gcd(b % a, a);
}
int main()
{
        int a = 10, b = 15;
        printf("GCD(%d, %d) = %d\n", a, b, gcd(a, b));          // Function call
        a = 35, b = 10;
        printf("GCD(%d, %d) = %d\n", a, b, gcd(a, b));
        a = 31, b = 2;
        printf("GCD(%d, %d) = %d\n", a, b, gcd(a, b));
        return 0;
}
```

Output

```
GCD(10, 15) = 5
GCD(35, 10) = 5
GCD(31, 2) = 1
```

Extended Euclidean Algorithm:

Extended Euclidean algorithm also finds integer coefficients x and y such that: ax + by = gcd(a, b)

Examples:

Input: *a = 30, b = 20* ***Input:*** *a = 35, b = 15*

Output: *gcd = 10, x = 1, y = -1* ***Output:*** *gcd = 5, x = 1, y = -2*

*(Note that 30*1 + 20*(-1) = 10)* *(Note that 35*1 + 15*(-2) = 5)*

The extended Euclidean algorithm updates the results of gcd(a, b) using the results calculated by the recursive call gcd(b%a, a). Let values of x and y calculated by the recursive call be x_1 and y_1. x and y are updated using the below expressions.

$ax + by = gcd(a, b)$

$gcd(a, b) = gcd(b\%a, a)$

$gcd(b\%a, a) = (b\%a)x_1 + ay_1$

$ax + by = (b\%a)x_1 + ay_1$

$ax + by = (b - [b/a] * a)x_1 + ay_1$

$ax + by = a(y_1 - [b/a] * x_1) + bx_1$

Comparing LHS and RHS,

$x = y_1 - [b/a] * x_1$

$y = x_1$

Below is an implementation of the above approach:

```
#include <stdio.h>
int gcdExtended(int a, int b, int *x, int *y) // C function for extended Euclidean Algorithm
{
        if (a == 0)      // Base Case
        {
                *x = 0;
                *y = 1;
                return b;
        }
        int x1, y1; // To store results of recursive call
        int gcd = gcdExtended(b%a, a, &x1, &y1);
        *x = y1 - (b/a) * x1;  // Update x and y using results of recursive call
        *y = x1;
        return gcd;
}
int main()
{
        int x, y;
        int a = 35, b = 15;
```

```
    int g = gcdExtended(a, b, &x, &y);
    printf("gcd(%d, %d) = %d", a, b, g);
    return 0;
}
```

Output : gcd(35, 15) = 5

Time Complexity: O(log N)

Auxiliary Space: O(log N)

How does Extended Algorithm Work?

As seen above, x and y are results for inputs a and b, $a.x + b.y = gcd$ *—-(1)*

And x_1 *and* y_1 *are results for inputs b%a and a*

$(b\%a).x_1 + a.y_1 = gcd$

When we put $b\%a = (b – (\lfloor b/a \rfloor).a)$ *in above, we get following. Note that* $\lfloor b/a \rfloor$ *is floor(b/a)*

$(b – (\lfloor b/a \rfloor).a).x_1 + a.y_1 = gcd$

Above equation can also be written as below

$b.x_1 + a.(y_1 – (\lfloor b/a \rfloor).x_1) = gcd$ *—(2)*

After comparing coefficients of ‘a’ and ‘b’ in (1) and (2), we get following,

$x = y_1 – \lfloor b/a \rfloor * x_1$

$y = x_1$

How is Extended Algorithm Useful?

The extended Euclidean algorithm is particularly useful when a and b are coprime (or gcd is 1). Since x is the modular multiplicative inverse of “a modulo b”, and y is the modular multiplicative inverse of “b modulo a”. In particular, the computation of the modular multiplicative inverse is an essential step in RSA public-key encryption method.