Mathematical Algorithms

SWE2016-44

Sieve of Eratosthenes

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

Example:

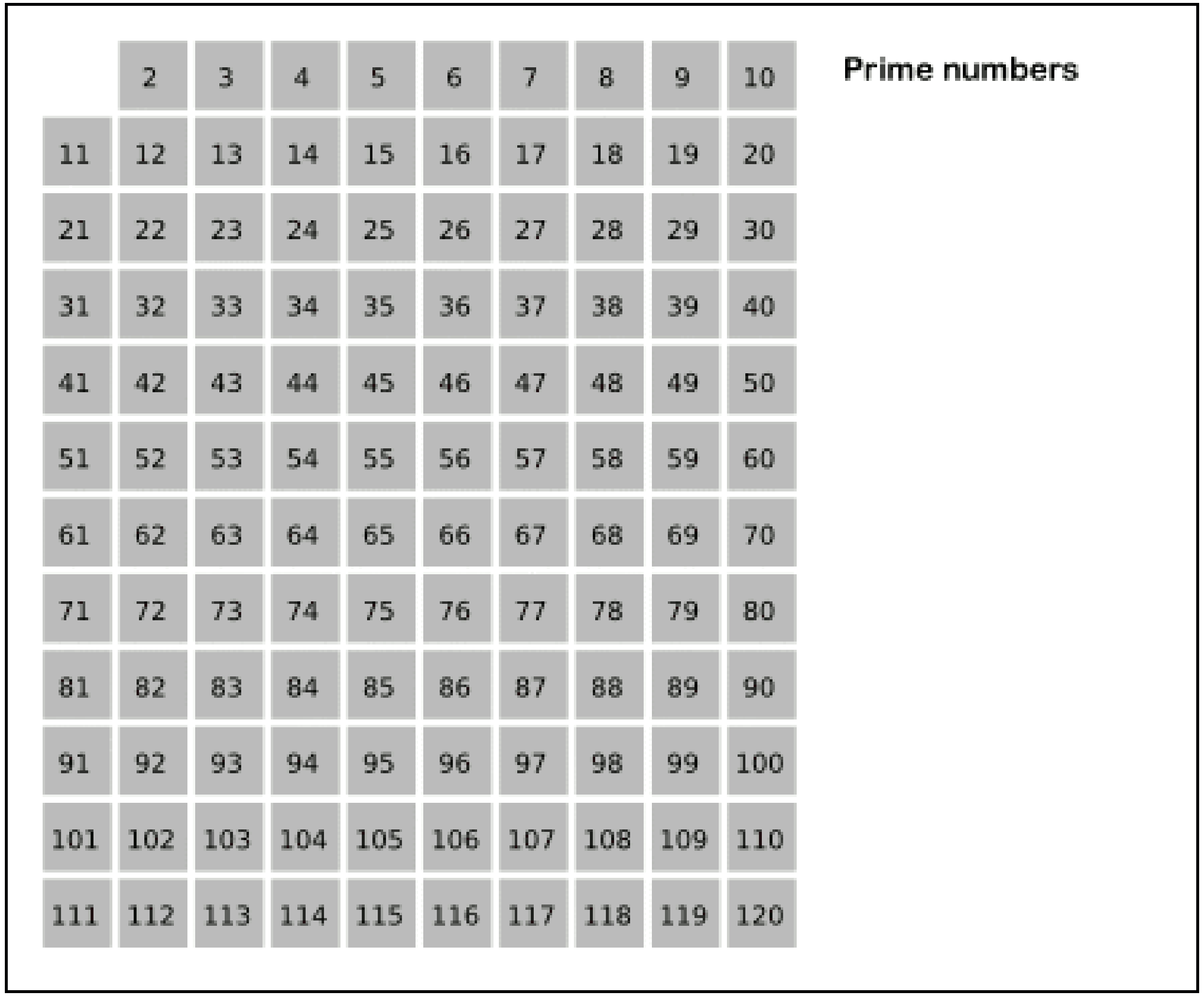
```
Input : n =10
Output : 2 3 5 7

Input : n = 20
Output: 2 3 5 7 11 13 17 19
```

Sieve of Eratosthenes

Algorithm:

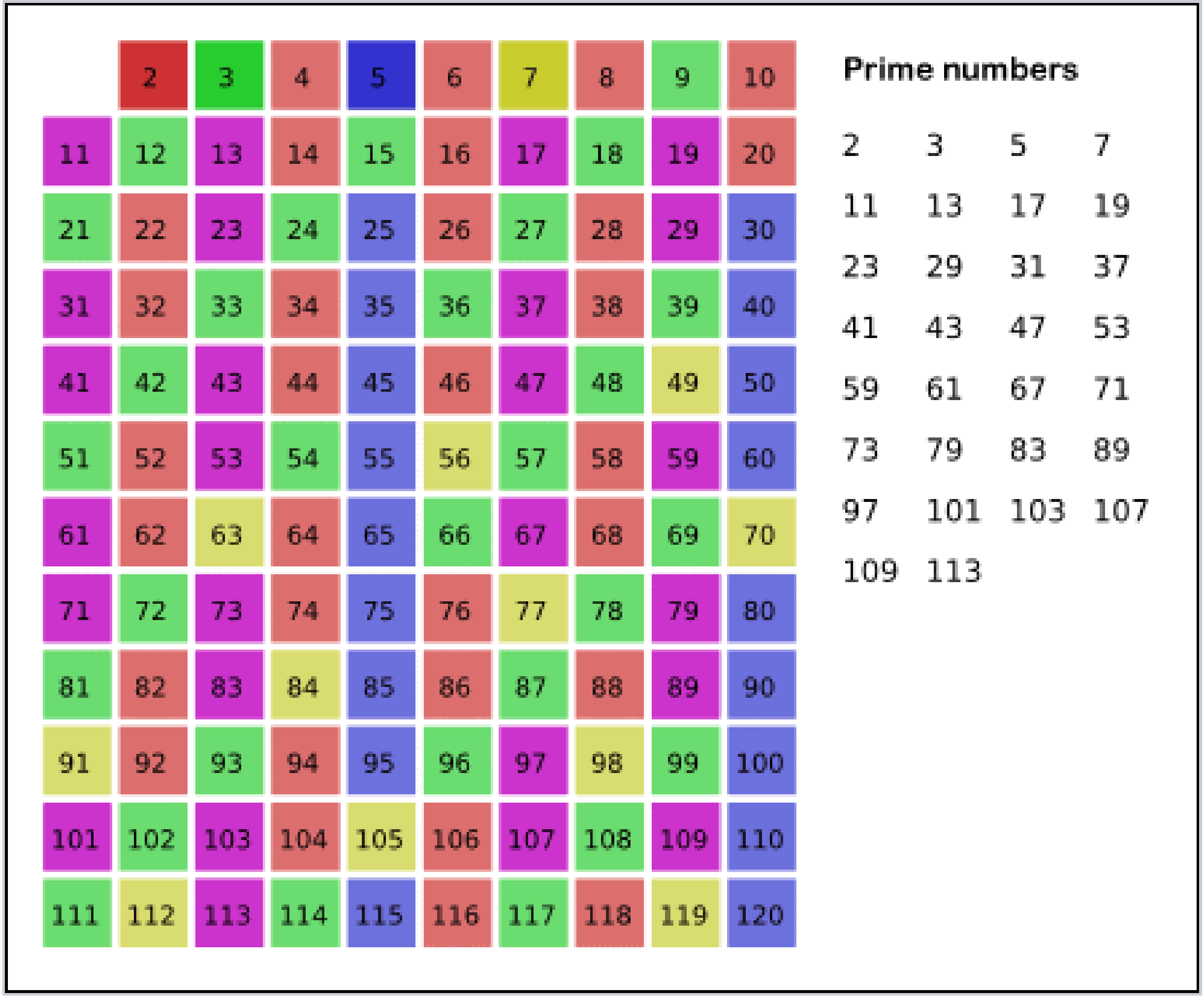
1) Create a list of consecutive integers from 2 to n: (2, 3, …, n).
2) Initially, let p equal 2, the first prime number.
3) Enumerate the multiples of p by counting in increments of p from 2p to n, and mark them in the list (2p, 3p, 4p, ...).
4) Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (the next prime), and repeat from step 3.
5) When the algorithm terminates, the numbers remaining not marked in the list are all the primes below n.

Sieve of Eratosthenes



https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Sieve of Eratosthenes



https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Sieve of Eratosthenes

```
void SieveOfEratosthenes(int n)
{
    // Create a boolean array "prime[0..n]" and initialize
    // all entries it as true. A value in prime[i] will
    // finally be false if i is Not a prime, else true.
    bool prime[n+1];
    memset(prime, true, sizeof(prime));

    for (int p=2; p*p<=n; p++)
    {
        // If prime[p] is not changed, then it is a prime
        if (prime[p] == true)
        {
            // Update all multiples of p greater than or
            // equal to the square of it
            // numbers which are multiple of p and are
            // less than p^2 are already been marked.
            for (int i=p*p; i<=n; i += p)
                prime[i] = false;
        }
    }

    // Print all prime numbers
    for (int p=2; p<=n; p++)
       if (prime[p])
          cout << p << " ";
}
```

Sieve of Eratosthenes

Time Complexity:

$$\boldsymbol{n\sum_{p\leq n}\frac{1}{p}\approx n\Big(log\big(log\,(n)\big)\Big)+M}$$

where M is the Meissel–Mertens constant

(Mertens' second theorem)

→ O(n*log(log(n)))

Finding all prime factors of a given number

Given a number n, write an efficient function to print all prime factors of n.

For example, if the input number is 12, then output should be “2 2 3”. And if the input number is 315, then output should be “3 3 5 7”.

Finding all prime factors of a given number

Algorithm 1:

1) **While n is divisible by 2, print 2 and divide n by 2.**
2) **After step 1, n must be odd. Now start a loop from i = 3 to square root of n. While i divides n, print i and divide n by i. After i fails to divide n, increment i by 2 and continue.**
3) **If n is a prime number and is greater than 2, then n will not become 1 by above two steps. So print n if it is greater than 2.**

Finding all prime factors of a given number

Algorithm 1:

```
void primeFactors(int n)
{
    // Print the number of 2s that divide n
    while (n % 2 == 0)
    {
        cout << 2 << " ";
        n = n/2;
    }

    // n must be odd at this point. So we can skip
    // one element (Note i = i +2)
    for (int i = 3; i <= sqrt(n); i = i + 2)
    {
        // While i divides n, print i and divide n
        while (n % i == 0)
        {
            cout << i << " ";
            n = n/i;
        }
    }

    // This condition is to handle the case when n
    // is a prime number greater than 2
    if (n > 2)
        cout << n << " ";
}
```

→ **Time Complexity: O(sqrt(n))**

Finding all prime factors of a given number

Algorithm 2:

1) **To calculate to smallest prime factor for every number we will use the sieve of eratosthenes. In original Sieve, every time we mark a number as not-prime, we store the corresponding smallest prime factor for that number.**
2) **After we are done with precalculating the smallest prime factor for every number we will divide our number n by its corresponding smallest prime factor till n becomes 1.**

Finding all prime factors of a given number

Algorithm 2:

```
#define MAXN   100001

// stores smallest prime factor for every number
int spf[MAXN];

// Calculating SPF (Smallest Prime Factor) for every
// number till MAXN.
// Time Complexity : O(nloglogn)
void sieve()
{
    spf[1] = 1;
    for (int i=2; i<MAXN; i++)

        // marking smallest prime factor for every
        // number to be itself.
        spf[i] = i;

    // separately marking spf for every even
    // number as 2
    for (int i=4; i<MAXN; i+=2)
        spf[i] = 2;
```

```
    for (int i=3; i*i<MAXN; i++)
    {
        // checking if i is prime
        if (spf[i] == i)
        {
            // marking SPF for all numbers divisible by i
            for (int j=i*i; j<MAXN; j+=i)

                // marking spf[j] if it is not
                // previously marked
                if (spf[j]==j)
                    spf[j] = i;
        }
    }
}

// A O(log n) function returning primefactorization
// by dividing by smallest prime factor at every step
vector<int> getFactorization(int x)
{
    vector<int> ret;
    while (x != 1)
    {
        ret.push_back(spf[x]);
        x = x / spf[x];
    }
    return ret;
}
```

→ **Time Complexity: O(log(n))**

Finding GCD or HCF of two numbers

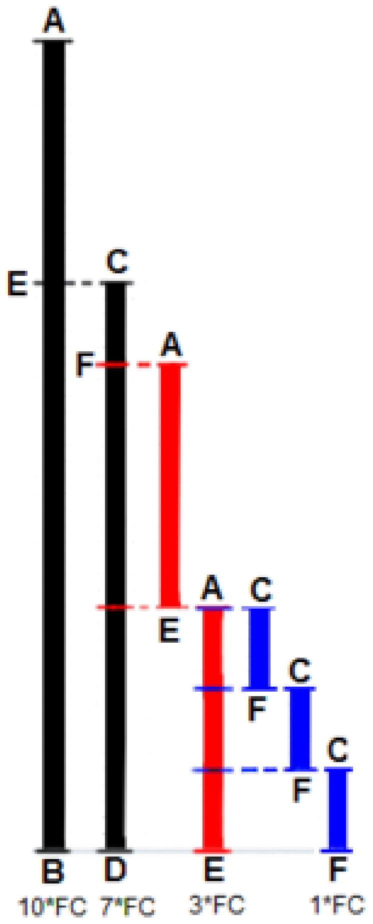
GCD (Greatest Common Divisor) or HCF (Highest Common Factor) of two numbers is the largest number that divides both of them. For example GCD of 20 and 28 is 4 and GCD of 98 and 56 is 14.

Finding GCD or HCF of two numbers

1) **<u>A simple solution</u>: find all prime factors of both numbers, then find intersection of all factors present in both numbers. Finally return product of elements in the intersection.**

Finding GCD or HCF of two numbers

2) <u>An efficient solution</u>: use Euclidean algorithm which is the main algorithm used for this purpose. The idea is, GCD of two numbers doesn't change if smaller number is subtracted from a bigger number.



Euclid's example

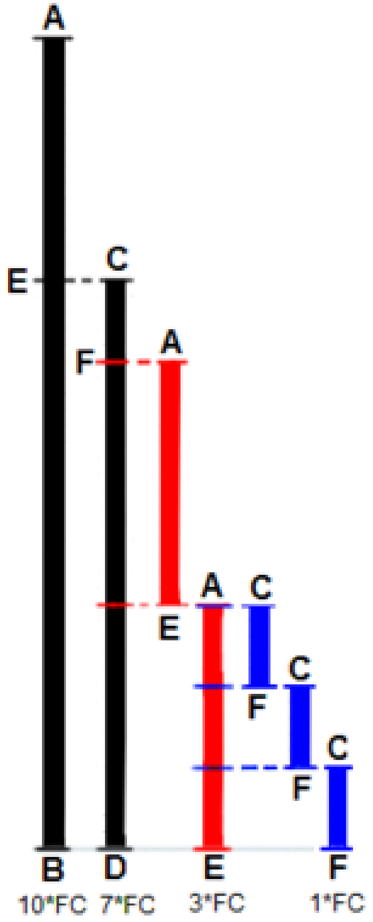
Finding GCD or HCF of two numbers

2) An efficient solution:

```
int gcd(int a, int b)
{
    // Everything divides 0
    if (a == 0)
       return b;
    if (b == 0)
       return a;

    // base case
    if (a == b)
        return a;

    // a is greater
    if (a > b)
        return gcd(a-b, b);
    return gcd(a, b-a);
}
```

Finding GCD or HCF of two numbers

3) <u>A more efficient solution</u>: use modulo operator in Euclidean algorithm



Euclid's example

Finding GCD or HCF of two numbers

3) <u>A more efficient solution (Euclidean Algorithm)</u>:

```
int gcd(int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);

}
```

Time Complexity: O(Log min(a, b))

Finding GCD or HCF of two numbers

Example: Find the GCD of 270 and 192

Method 2: GCD(270, 192) → GCD(192, 78) → GCD(114, 78) → GCD(78, 36) → GCD(42, 36) → GCD(36, 6) → GCD(6, 0)

Method 3: GCD(270, 192) → GCD(192, 78) → GCD(78, 36) → GCD(36, 6) → GCD(6, 0)

Finding GCD or HCF of two numbers

<u>Extended Euclidean Algorithm</u>

ax + by = gcd(a, b)

Examples:

```
Input: a = 30, b = 20
Output: gcd = 10
        x = 1, y = -1
(Note that 30*1 + 20*(-1) = 10)

Input: a = 35, b = 15
Output: gcd = 5
        x = 1, y = -2
(Note that 35*1 + 15*(-2) = 5)
```

Finding GCD or HCF of two numbers

Extended Euclidean Algorithm

The extended Euclidean algorithm updates results of gcd(a, b) using the results calculated by 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.

$$x = y_1 - \lfloor b/a \rfloor * x_1$$
$$y = x_1$$

Finding GCD or HCF of two numbers

Extended Euclidean Algorithm

```
int gcdExtended(int a, int b, int *x, int *y)
{
    // Base Case
    if (a == 0)
    {
        *x = 0;
        *y = 1;
        return b;
    }

    int x1, y1; // To store results of recursive call
    int gcd = gcdExtended(b%a, a, &x1, &y1);

    // Update x and y using results of
    // recursive call
    *x = y1 - (b/a) * x1;
    *y = x1;

    return gcd;
}
```

Time Complexity: O(Log min(a, b))

Matrix Chain Multiplication

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

Matrix Chain Multiplication

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

```
(ABC)D = (AB)(CD) = A(BCD) = ....
```

Matrix Chain Multiplication

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

```
(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations
A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.
```

Matrix Chain Multiplication

Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

```
Input: p[] = {40, 20, 30, 10, 30}
Output: 26000
There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.
Let the input 4 matrices be A, B, C and D.  The minimum number of
multiplications are obtained by putting parenthesis in following way
(A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30

Input: p[] = {10, 20, 30, 40, 30}
Output: 30000
There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.
Let the input 4 matrices be A, B, C and D.  The minimum number of
multiplications are obtained by putting parenthesis in following way
((AB)C)D --> 10*20*30 + 10*30*40 + 10*40*30
```

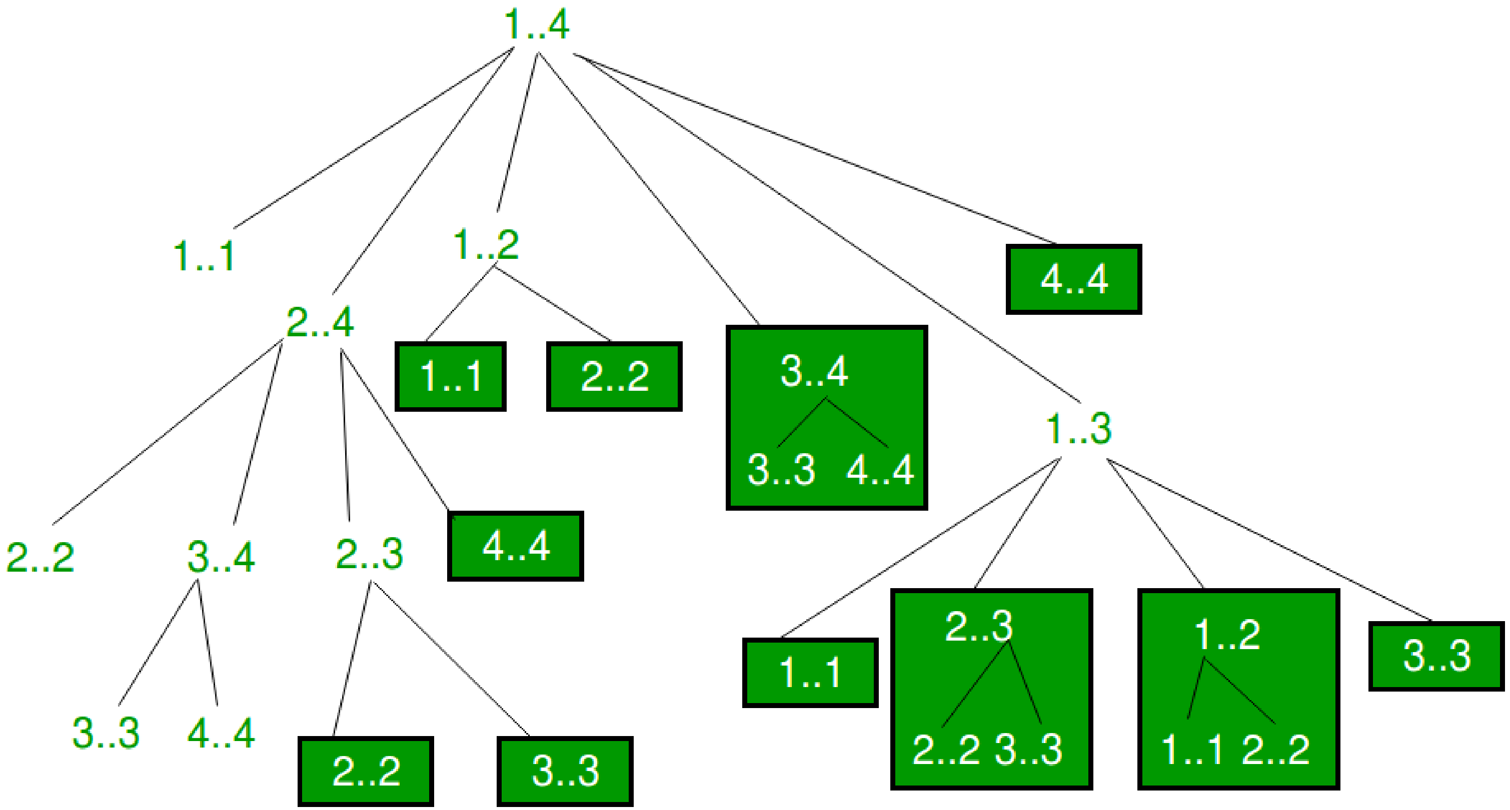
Matrix Chain Multiplication

Recursive Implementation

```
// Matrix Ai has dimension p[i-1] x p[i]
// for i = 1..n
int MatrixChainOrder(int p[], int i, int j)
{
    if(i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;

    // place parenthesis at different places
    // between first and last matrix, recursively
    // calculate count of multiplications for
    // each parenthesis placement and return the
    // minimum count
    for (k = i; k < j; k++)
    {
        count = MatrixChainOrder(p, i, k) +
                MatrixChainOrder(p, k + 1, j) +
                p[i - 1] * p[k] * p[j];

        if (count < min)
            min = count;
    }

    // Return minimum count
    return min;
}
```

Matrix Chain Multiplication

Overlapping Subproblems



Matrix Chain Multiplication

Dynamic Programming

```
int main()
{
    int arr[] = {1, 2, 3, 4};
    int size = sizeof(arr) / sizeof(arr[0]);

    cout << "Minimum number of multiplications is "
         << MatrixChainOrder(arr, size);

    getchar();
    return 0;
}
```

Time Complexity: $O(n^3)$

```
int MatrixChainOrder(int p[], int n)
{

    /* For simplicity of the program, one
    extra row and one extra column are
    allocated in m[][]. 0th row and 0th
    column of m[][] are not used */
    int m[n][n];

    int i, j, k, L, q;

    /* m[i,j] = Minimum number of scalar
    multiplications needed to compute the
    matrix A[i]A[i+1]...A[j] = A[i..j] where
    dimension of A[i] is p[i-1] x p[i] */

    // cost is zero when multiplying
    // one matrix.
    for (i = 1; i < n; i++)
        m[i][i] = 0;

    // L is chain length.
    for (L = 2; L < n; L++)
    {
        for (i = 1; i < n - L + 1; i++)
        {
            j = i + L - 1;
            m[i][j] = INT_MAX;
            for (k = i; k <= j - 1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k + 1][j] +
                    p[i - 1] * p[k] * p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

    return m[1][n - 1];
}
```

Reference

- Charles Leiserson and Piotr Indyk, “*Introduction to Algorithms”,* September 29, 2004

- https://www.geeksforgeeks.org

- https://en.wikipedia.org/wiki