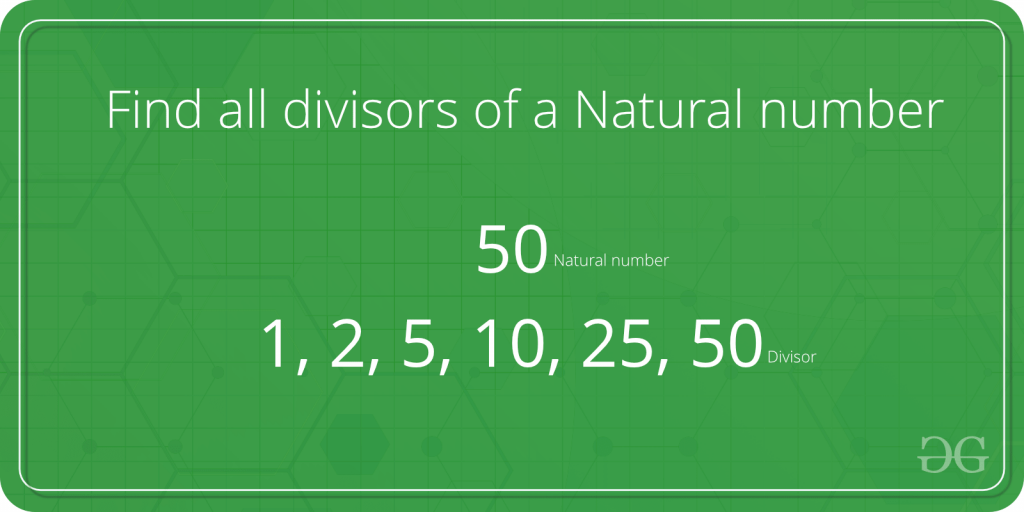
**All Divisors of a Number**

Given a natural number n, print all distinct divisors of it.



**Examples:**

**Input :** n = 10

**Output:** 1 2 5 10

**Input:**  n = 100

**Output:** 1 2 4 5 10 20 25 50 100

**Input:**  n = 125

**Output:** 1 5 25 125

A **Naive Solution** would be to iterate all the numbers from 1 to n, checking if that number divides n and printing it. Below is a program for the same:

C++Java

// C++ implementation of Naive method to print all

// divisors

#include <iostream>

using namespace std;

// function to print the divisors

void printDivisors(int n)

{

for (int i = 1; i <= n; i++)

if (n % i == 0)

cout <<" " << i;

}

/\* Driver program to test above function \*/

int main()

{

cout <<"The divisors of 100 are: \n";

printDivisors(100);

return 0;

}

**Output:**

The divisors of 100 are:

1 2 4 5 10 20 25 50 100

**Time Complexity :** O(n)   
**Auxiliary Space :** O(1)

**Can we improve the above solution?**   
If we look carefully, all the divisors are present in pairs. For example if n = 100, then the various pairs of divisors are: (1,100), (2,50), (4,25), (5,20), (10,10)  
Using this fact we could speed up our program significantly.   
We, however, have to be careful if there are two equal divisors as in the case of (10, 10). In such case, we’d print only one of them.

Below is an implementation for the same:

C++Java

// A Better (than Naive) Solution to find all divisors

#include <iostream>

#include <math.h>

using namespace std;

// Function to print the divisors

void printDivisors(int n)

{

// Note that this loop runs till square root

for (int i=1; i<=sqrt(n); i++)

{

if (n%i == 0)

{

// If divisors are equal, print only one

if (n/i == i)

cout <<" "<< i;

else // Otherwise print both

cout << " "<< i << " " << n/i;

}

}

}

/\* Driver program to test above function \*/

int main()

{

cout <<"The divisors of 100 are: \n";

printDivisors(100);

return 0;

}

**Output:**

The divisors of 100 are:

1 100 2 50 4 25 5 20 10

Time Complexity: O(sqrt(n))   
Auxiliary Space : O(1)

**Printing all the divisors in sorted order:**

C++Java

// A O(sqrt(n)) program that prints all divisors

// in sorted order

#include <iostream>

#include <math.h>

using namespace std;

// Function to print the divisors

void printDivisors(int n)

{

int i;

for (i = 1; i \* i < n; i++) {

if (n % i == 0)

cout<<i<<" ";

}

if (i - (n / i) == 1) {

i--;

}

for (; i >= 1; i--) {

if (n % i == 0)

cout<<n / i<<" ";

}

}

// Driver code

int main()

{

cout << "The divisors of 100 are: \n";

printDivisors(100);

return 0;

}

// This code is contributed by siteshbiswal

**Output:**

1 2 4 5 10 20 25 50 100

Time Complexity: O(sqrt(n))   
Auxiliary Space : O(1)

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*

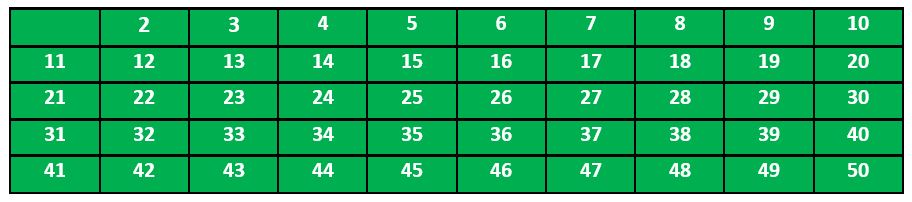
The sieve of Eratosthenes is one of the most efficient ways to find all primes smaller than n when n is smaller than 10 million or so.

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.

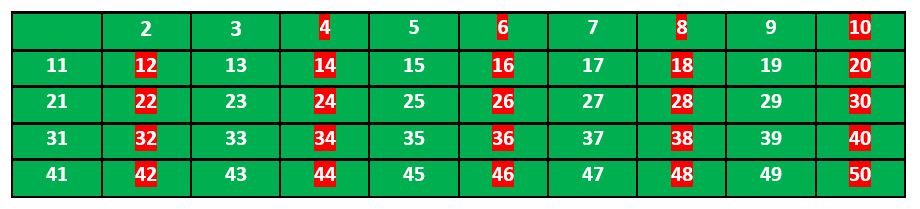
**Explanation with Example:**

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

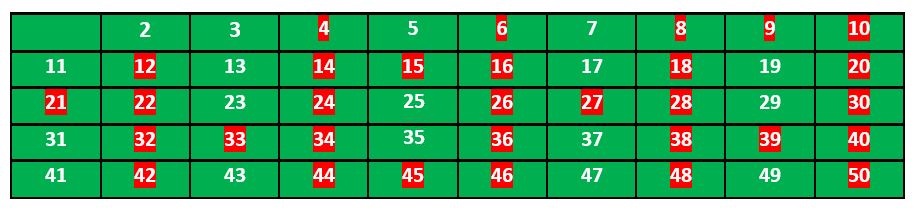
We create a list of all numbers from 2 to 50.



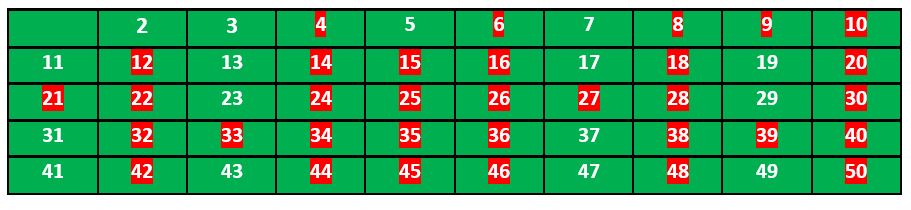
According to the algorithm we will mark all the numbers which are divisible by 2 and are greater than or equal to the square of it.



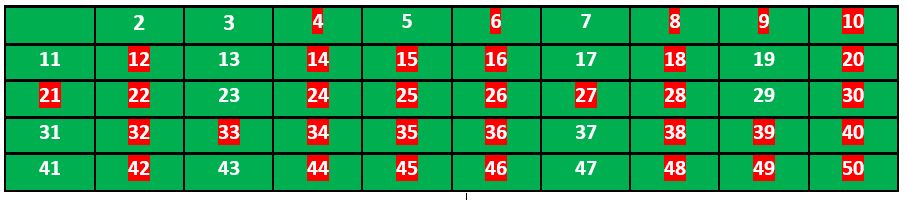
Now we move to our next unmarked number 3 and mark all the numbers which are multiples of 3 and are greater than or equal to the square of it.



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.



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



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.

C++Java

// C++ program to print all primes smaller than or equal to

// n using Sieve of Eratosthenes

#include <bits/stdc++.h>

using namespace std;

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 << " ";

}

// Driver Code

int main()

{

int n = 30;

cout << "Following are the prime numbers smaller "

<< " than or equal to " << n << endl;

SieveOfEratosthenes(n);

return 0;

}

**Output**

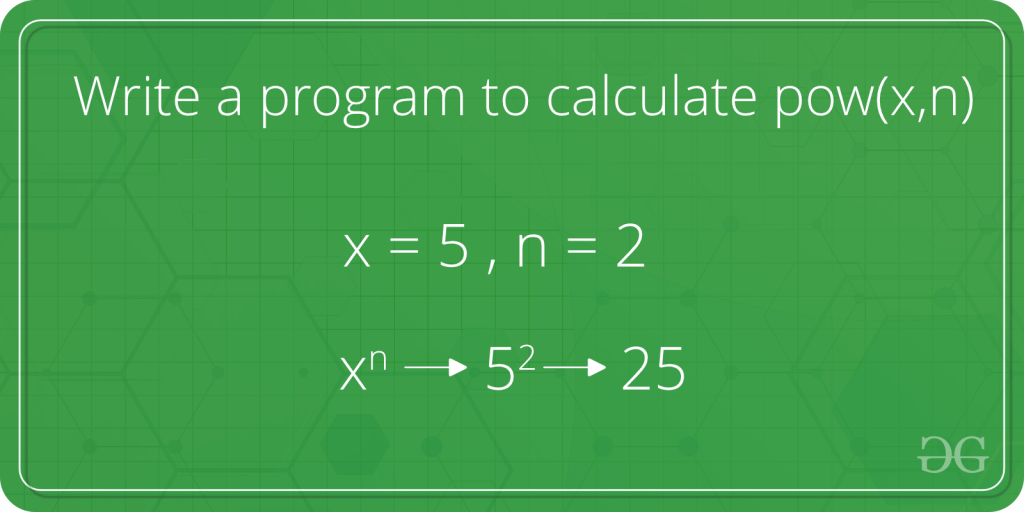
**Following are the prime numbers smaller than or equal to 30**

**2 3 5 7 11 13 17 19 23 29**

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

Computing Power

Given two integers**x**and **n**, write a function to compute **xn**. We may assume that x and n are small and overflow doesn’t happen.

****

**Examples :**

***Input :****x = 2, n = 3*  
***Output :****8*

***Input :****x = 7, n = 2*  
***Output :****49*

**Naive Approach:** To solve the problem follow the below idea:

*A simple solution to calculate pow(x, n) would multiply x exactly n times. We can do that by using a simple for loop*

Below is the implementation of the above approach:

C++Java

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

// Naive iterative solution to calculate pow(x, n)

long power(int x, unsigned n)

{

// Initialize result to 1

long long pow = 1;

// Multiply x for n times

for (int i = 0; i < n; i++) {

pow = pow \* x;

}

return pow;

}

// Driver code

int main(void)

{

int x = 2;

unsigned n = 3;

// Function call

int result = power(x, n);

cout << result << endl;

return 0;

}

**Output**

8

**Time Complexity:**O(n)  
**Auxiliary Space:** O(1)

**An Optimized Divide and Conquer Solution:**

*The problem can be recursively defined by:*

* *power(x, n) = power(x, n / 2) \* power(x, n / 2);        // if n is even*
* *power(x, n) = x \* power(x, n / 2) \* power(x, n / 2);    // if n is odd*

However there is a problem with the above solution, the same subproblem is computed twice for each recursive call. We can optimize the above function by computing the solution of the subproblem once only.

Below is the implementation of the above approach:

C++Java

/\* Function to calculate x raised to the power y in

\* O(logn)\*/

int power(int x, unsigned int y)

{

int temp;

if (y == 0)

return 1;

temp = power(x, y / 2);

if (y % 2 == 0)

return temp \* temp;

else

return x \* temp \* temp;

}

int main()

{

int x = 2;

unsigned int y = 3;

// Function call

cout << power(x, y);

return 0;

}

**Output**

8

**Time Complexity:**O(log n)  
**Auxiliary Space:** O(log n), for recursive call stack

Modular Arithmetic

Modular arithmetic is the branch of arithmetic mathematics related with the "mod" functionality. Basically, modular arithmetic is related with computation of "mod" of expressions. Expressions may have digits and computational symbols of addition, subtraction, multiplication, division or any other. Here we will discuss briefly about all modular arithmetic operations.

## [**Quotient Remainder Theorem:**](https://www.geeksforgeeks.org/quotient-remainder-theorem/)

It states that, for any pair of integers a and b (b is positive), there exist two unique integers q and r such that:

a = b x q + r where 0 <= r < b

**Example:** If a = 20, b = 6 then q = 3, r = 2 20 = 6 x 3 + 2

## [**Modular Addition:**](https://www.geeksforgeeks.org/modular-addition/)

Rule for modular addition is:

(a + b) mod m = ((a mod m) + (b mod m)) mod m

**Example:**

(15 + 17) % 7= ((15 % 7) + (17 % 7)) % 7= (1 + 3) % 7= 4 % 7= 4

The same rule is to modular subtraction. We don't require much modular subtraction but it can also be done in the same way.

## [**Modular Multiplication:**](https://www.geeksforgeeks.org/modular-multiplication/)

The Rule for modular multiplication is:

(a x b) mod m = ((a mod m) x (b mod m)) mod m

**Example:**

(12 x 13) % 5= ((12 % 5) x (13 % 5)) % 5= (2 x 3) % 5= 6 % 5= 1

## [**Modular Division:**](https://www.geeksforgeeks.org/modular-division/)

The modular division is totally different from modular addition, subtraction and multiplication. It also does not exist always.

(a / b) mod m is not equal to ((a mod m) / (b mod m)) mod m.

This is calculated using the following formula:

(a / b) mod m = (a x (inverse of b if exists)) mod m

## [**Modular Inverse:**](https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/)

The modular inverse of a mod m exists only if a and m are relatively prime i.e. gcd(a, m) = 1. Hence, for finding the inverse of an under modulo m, if (a x b) mod m = 1 then b is the modular inverse of a.

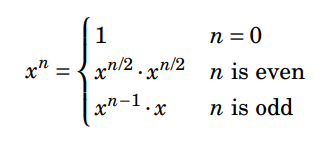
**Example:** a = 5, m = 7 (5 x 3) % 7 = 1 hence, 3 is modulo inverse of 5 under 7.

## [**Modular Exponentiation:**](https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic/)

Finding a^b mod m is the modular exponentiation. There are two approaches for this - recursive and iterative. **Example:**

a = 5, b = 2, m = 7(5 ^ 2) % 7 = 25 % 7 = 4

There is often a need to efficiently calculate the value of xn mod m. This can be done in O(logn) time using the following recursion:



It is important that in the case of an even n, the value of xn/2 is calculated only once.

This guarantees that the time complexity of the algorithm is O(logn) because n is always halved when it is even.

The following function calculates the value of xn mod m:

|  |
| --- |
| int modpower(int x, int n, int m)  {    if (n == 0)         return 1%m;    long long u = modpower(x,n/2,m);                                                  u = (u\*u)%m;    if (n%2 == 1)         u = (u\*x)%m;    return u; } |

C++Java

#include<bits/stdc++.h>

#include<iostream>

using namespace std;

//function that calculate modular exponentiation x^n mod m.

int modpower(int x, int n, int m)

{

if (n == 0) //base case

return 1%m;

long long u = modpower(x,n/2,m);

u = (u\*u)%m;

if (n%2 == 1) //when 'n' is odd

u = (u\*x)%m;

return u;

}

//driver function

int main()

{

cout<<modpower(5,2,7)<<endl;

return 0;

}

output:4

Time complexity: O(logn), because n is always halved when it is even.

Fermat’s theorem states that

                                              xm−1 mod m = 1

when m is prime and x and m are coprime. This also yields  
                                                                                              xk mod m = xk mod (m−1) mod m.

Iterative Power

Given an integer x and a positive number y, write a function that computes xy under following conditions.   
a) Time complexity of the function should be O(Log y)   
b) Extra Space is O(1)

**Examples:**

Input: x = 3, y = 5

Output: 243

Input: x = 2, y = 5

Output: 32

The recursive solutions are generally not preferred as they require space on call stack and they involve function call overhead.

Following is implementation to compute xy.

C++Java

// Iterative C program to implement pow(x, n)

#include <iostream>

using namespace std;

/\* Iterative Function to calculate (x^y) in O(logy) \*/

int power(int x, unsigned int y)

{

int res = 1; // Initialize result

while (y > 0) {

// If y is odd, multiply x with result

if (y & 1)

res = res \* x;

// y must be even now

y = y >> 1; // y = y/2

x = x \* x; // Change x to x^2

}

return res;

}

// Driver program to test above functions

int main()

{

int x = 3;

unsigned int y = 5;

cout<<"Power is "<<power(x, y);

return 0;

}

**Output:**

Power is 243

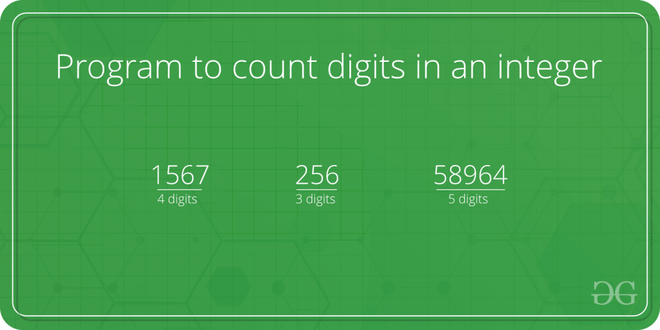
***Time Complexity:****O(log y), since in loop each time the value of y decreases by half it’s current value.*

***Auxiliary Space:****O(1), since no extra space has been taken.*

Count Digits

Given a number N, the task is to return the count of digits in this number.

**Example:**



*Program to count digits in an integer*

## **Simple Iterative Solution to count digits in an integer**

The integer entered by the user is stored in the variable n. Then the while loop is iterated until the test expression n != 0 is evaluated to 0 (false).   We will consider 3456 as the input integer.

1. After the first iteration, the value of n will be updated to 345 and the count is incremented to 1.
2. After the second iteration, the value of n will be updated to 34 and the count is incremented to 2.
3. After the third iteration, the value of n will be updated to 3 and the count is incremented to 3.
4. In the fourth iteration, the value of n will be updated to zero and the count will be incremented to 4.
5. Then the test expression is evaluated ( n!=0 ) as false and the loop terminates with final count as 4.

Below is the implementation of the above approach:

C++Java

// Iterative C++ program to count

// number of digits in a number

#include <bits/stdc++.h>

using namespace std;

int countDigit(long long n)

{

if (n == 0)

return 1;

int count = 0;

while (n != 0) {

n = n / 10;

++count;

}

return count;

}

// Driver code

int main(void)

{

long long n = 345289467;

cout << "Number of digits : " << countDigit(n);

return 0;

}

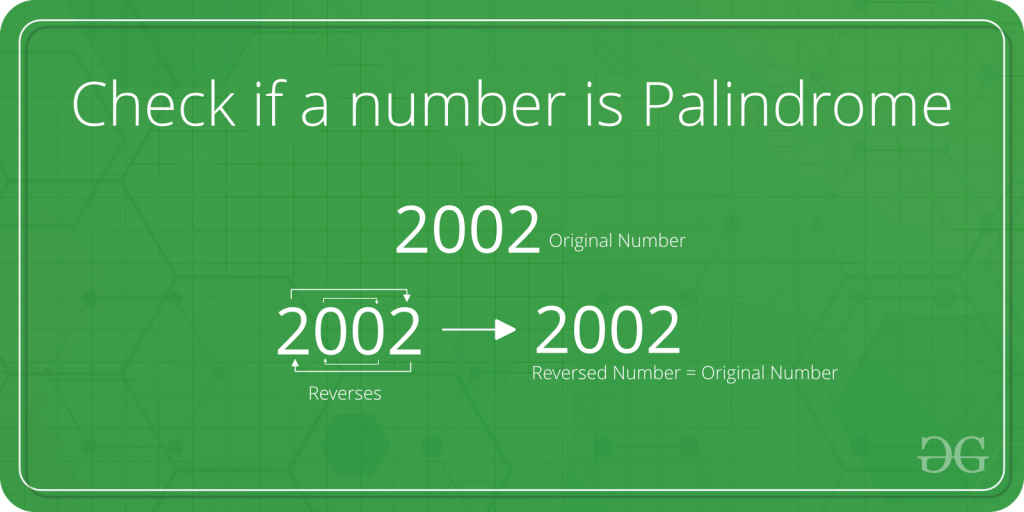
**Output**

Number of digits : 9

**Time Complexity :**O(log10(n)) or θ(num digits)  
**Auxiliary Space:**O(1) or constant

Palindrome Numbers

Given an integer, write a function that returns true if the given number is palindrome, else false. For example, 12321 is palindrome, but 1451 is not palindrome.



A simple approach to check if a number is Palindrome or not . This approach can be used when the number of digits in the given number is less than 10^18 because if the number of digits of that number exceeds 10^18, we can’t take that number as an integer since the range of long long int doesn’t satisfy the given number.

To check whether the given number is palindrome or not we will just reverse the digits of the given number and check if the reverse of that number is equal to the original number or not . If reverse of number is equal to that number than the number will be Palindrome else it will not a Palindrome.

C++Java

// C++ program to check if a number is Palindrome

#include <iostream>

using namespace std;

// Function to check Palindrome

bool checkPalindrome(int n)

{

int reverse = 0;

int temp = n;

while (temp != 0) {

reverse = (reverse \* 10) + (temp % 10);

temp = temp / 10;

}

return (reverse

== n); // if it is true then it will return 1;

// else if false it will return 0;

}

int main()

{

int n = 7007;

if (checkPalindrome(n) == 1) {

cout << "Yes\n";

}

else {

cout << "No\n";

}

return 0;

}

**Output**

Yes

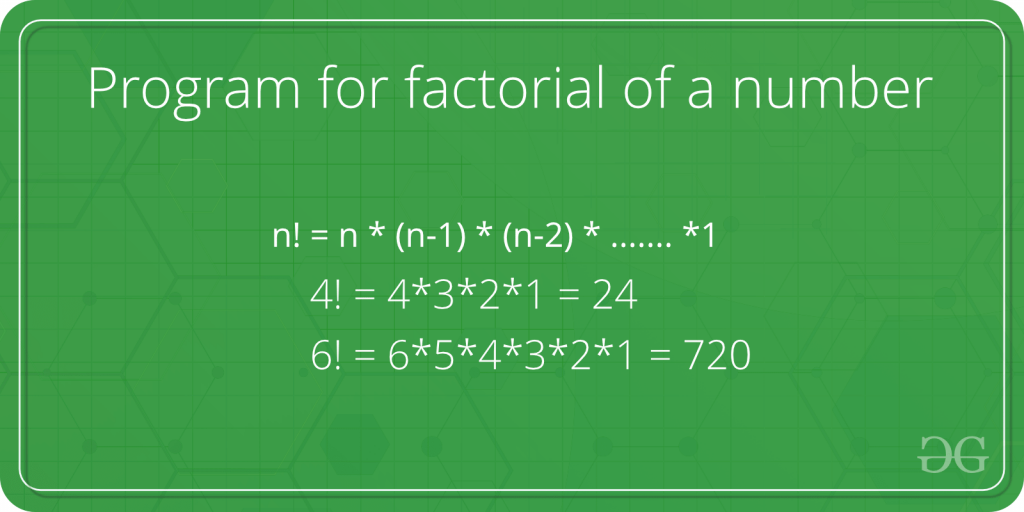
**Time Complexity :**O(log(n)) or O(Number of digits in a given number)

**Auxiliary space**: O(1) or constant

Factorial of a Number

## What is the factorial of a number?

* Factorial of a non-negative integer is the multiplication of all positive integers smaller than or equal to n. For example factorial of 6 is 6\*5\*4\*3\*2\*1 which is 720.
* A factorial is represented by a number and a  ” ! ”  mark at the end. It is widely used in permutations and combinations to calculate the total possible outcomes. A French mathematician Christian Kramp firstly used the exclamation.



Let’s create a factorial program using recursive functions. Until the value is not equal to zero, the recursive function will call itself. Factorial can be calculated using the following recursive formula.

*n! = n \* (n – 1)!*  
*n == 1 if n = 0 or n = 1*

Below is the implementation:

C++Java

// C++ program to find

// factorial of given number

#include <iostream>

using namespace std;

// Function to find factorial

// of given number

unsigned int factorial(unsigned int n)

{

if (n == 0 || n == 1)

return 1;

return n \* factorial(n - 1);

}

// Driver code

int main()

{

int num = 5;

cout << "Factorial of "

<< num << " is " << factorial(num) << endl;

return 0;

}

**Output**

Factorial of 5 is 120

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

## **Iterative Solution to find factorial of a number:**

Factorial can also be calculated iteratively as recursion can be costly for large numbers. Here we have shown the iterative approach using both for and while loops.

**Approach 1:**Using For loop

Follow the steps to solve the problem:

* Using a for loop, we will write a program for finding the factorial of a number.
* An integer variable with a value of 1 will be used in the program.
* With each iteration, the value will increase by 1 until it equals the value entered by the user.
* The factorial of the number entered by the user will be the final value in the fact variable.

Below is the implementation for the above approach:

C++Java

// C++ program for factorial of a number

#include <iostream>

using namespace std;

// function to find factorial of given number

unsigned int factorial(unsigned int n)

{

int res = 1, i;

for (i = 2; i <= n; i++)

res \*= i;

return res;

}

// Driver code

int main()

{

int num = 5;

cout << "Factorial of "

<< num << " is "

<< factorial(num) << endl;

return 0;

}

**Output**

Factorial of 5 is 120

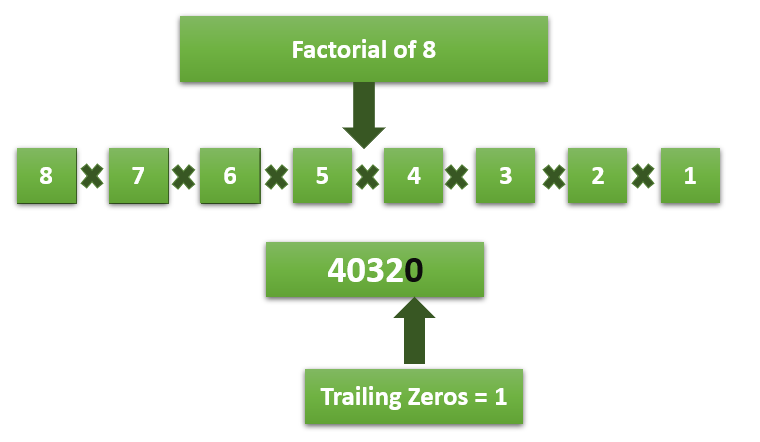
**Time Complexity:**O(n)  
**Auxiliary Space:** O(1)

Trailing Zeros in Factorial

In a realm where numbers hold secrets, a captivating challenge awaits, which is to, **Count Trailing Zeros in Factorial !!!**

**Our Task:** We are given a number. The task is to find the Number of Trailing Zeros in the factorial of the number.

The **Trailing Zeros**are the Zeros, which appear at the end of a number(factorial in that case)



**Examples :**

**Input:** 5

**Output:** 1

// Factorial of 5 = 5\*4\*3\*2\*1 = 120, which has one trailing 0.

**Input:** 20

**Output:** 4

// Factorial of 20 = 20\*19\*18\*.... 3\*2\*1 = 2432902008176640000 which has 4 trailing zeroes.

**Input:** 100

**Output:** 24

We have 2 approaches to solve the problem: **Naive Approach** & **Efficient Approach**

**1) Naive Approach**

A simple method is to first calculate the factorial of n, then count trailing 0s in the result (We can count trailing 0s by repeatedly dividing the factorial by 10 till the remainder is not 0).

But, this method can cause overflow for slightly bigger numbers as the factorial of a number is a big number. So, we prefer the Efficient Approach

**2) Efficient Approach**

The idea is to consider prime factors of a factorial n. A trailing zero is always produced by prime factors 2 and 5. Our task is done if we can **count the number of 5s and 2s**. Consider the following examples:

* **n = 5:** There is one 5 and 3 2s in prime factors of 5! (2 \* 2 \* 2 \* 3 \* 5). So a count of trailing 0s is 1.
* **n = 11:** There are two 5s and eight 2s in prime factors of 11! (2 8 \* 34 \* 52 \* 7). So the count of trailing 0s is 2.

We can easily observe that the number of 2s in prime factors is always more than or equal to the number of 5s. So if we **count 5s in prime factors**, we are done.

Following is the summarized formula for counting trailing 0s:

**Trailing 0s in n! = Count of 5s in prime factors of n! = floor(n/5) + floor(n/25) + floor(n/125) + ....**

**The implementation is shown below:**

C++Java

// C++ program to count

// trailing 0s in n!

#include <iostream>

using namespace std;

// Function to return trailing

// 0s in factorial of n

int findTrailingZeros(int n)

{

if (n < 0) // Negative Number Edge Case

return -1;

// Initialize result

int count = 0;

// Keep dividing n by powers of

// 5 and update count

for (int i = 5; n / i >= 1; i \*= 5)

count += n / i;

return count;

}

// Driver Code

int main()

{

int n = 100;

cout << "Count of trailing 0s in " << 100 << "! is "

<< findTrailingZeros(n);

return 0;

}

**Output :**

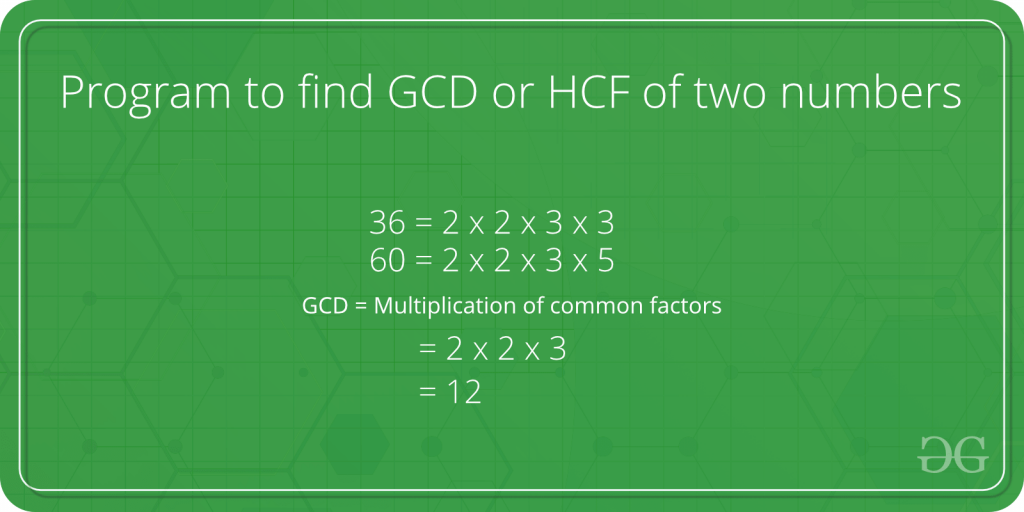
Count of trailing 0s in 100! is 24

**Time Complexity: O(log5n)**

**Auxiliary Space: O(1)**

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.

A simple and old  approach is the **Euclidean algorithm by subtraction**

 It is a process of repeat subtraction, carrying the result forward each time until the result is equal to any one number being subtracted. If the answer is greater than 1, there is a GCD (besides 1). If the answer is 1, there is no common divisor (besides 1), and so both numbers are coprime

pseudo code for the above approach:

*def gcd(a, b):*  
*if a == b:*  
*return a*  
*if a > b:*  
*gcd(a – b, b)*  
*else:*  
*gcd(a, b – a)*

At some point, one number becomes factor of the other so instead of repeatedly subtracting till both become equal, we check if it is factor of the other.

**For Example**  suppose a=98 & b=56  a>b so put a= a-b and b remains same. So  a=98-56=42  & b= 56 . Since b>a, we check if b%a==0. since answer is no we proceed further. Now b>a  so  b=b-a and a remain same. So b= 56-42 = 14 & a= 42   . Since a>b, we check if a%b==0 . Now the answer is yes. So we print smaller among a and b as H.C.F . i.e. 42 is  3 times of 14  so **HCF** is 14.

likewise  when a=36  & b=60  ,here b>a  so b = 24 & a= 36 but a%b!=0.  Now a>b so a= 12 & b= 24  . and b%a==0. smaller among a and b is 12  which becomes  HCF of 36 and 60.

The idea is, GCD of two numbers doesn’t change if a smaller number is subtracted from a bigger number.

C++Java

// C++ program to find GCD of two numbers

#include <iostream>

using namespace std;

// Recursive function to return gcd of a and b

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);

}

// Driver program to test above function

int main()

{

int a = 98, b = 56;

cout<<"GCD of "<<a<<" and "<<b<<" is "<<gcd(a, b);

return 0;

}

**Output**

GCD of 98 and 56 is 14

**Time Complexity:**O(min(a,b))  
**Auxiliary Space:**O(min(a,b))

Instead of Euclidean algorithm by subtraction, a better approach is present. We don’t perform subtraction here. we continuously divide the bigger number by the smaller number. More can be learned about this **efficient solution**by using modulo operator in Euclidean algorithm

C++Java

// C++ program to find GCD of two numbers

#include <iostream>

using namespace std;

// Recursive function to return gcd of a and b in single line

int gcd(int a, int b)

{

return b == 0 ? a : gcd(b, a % b);

}

// Driver program to test above function

int main()

{

int a = 98, b = 56;

cout<<"GCD of "<<a<<" and "<<b<<" is "<<gcd(a, b);

return 0;

}

**Output**

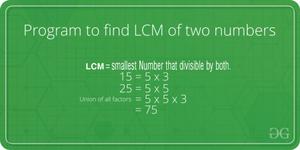
GCD of 98 and 56 is 14

**Time Complexity:** O(log(min(a,b))|  
**Auxiliary Space:**O(log(min(a,b))

The time complexity for the above algorithm is O(log(min(a,b))) the derivation for this is obtained from the analysis of the worst-case scenario. What we do is we ask what are the 2 least numbers that take 1 step, those would be (1,1). If we want to increase the number of steps to 2 while keeping the numbers as low as possible as we can take the numbers to be (1,2). Similarly, for 3 steps, the numbers would be (2,3), 4 would be (3,5), 5 would be (5,8). So we can notice a pattern here, for the nth step the numbers would be (fib(n), fib(n+1)).  So the worst-case time complexity would be O(n) where a>= fib(n) and b>= fib(n+1).

Now Fibonacci series is an exponentially growing series where the ratio of nth/(n-1)th term approaches (sqrt(5)+1)/2 which is also called the golden ratio. So we can see that the time complexity of the algorithm increases linearly as the terms grow exponentially hence the time complexity would be log(min(a,b)).

LCM of Two Numbers



LCM (Least Common Multiple) of two numbers is the smallest number which can be divided by both numbers.

For example, LCM of 15 and 20 is 60, and LCM of 5 and 7 is 35.

A **simple solution** is to find all prime factors of both numbers, then find union of all factors present in both numbers. Finally, return the product of elements in union.

An **efficient solution**is based on the below formula for LCM of two numbers ‘a’ and ‘b’.

a x b = LCM(a, b) \* GCD (a, b)

LCM(a, b) = (a x b) / GCD(a, b)

We have discussed function to find GCD of two numbers. Using GCD, we can find LCM.

Below is the implementation of the above idea:

C++Java

// C++ program to find LCM of two numbers

#include <iostream>

using namespace std;

// Recursive function to return gcd of a and b

long long gcd(long long int a, long long int b)

{

if (b == 0)

return a;

return gcd(b, a % b);

}

// Function to return LCM of two numbers

long long lcm(int a, int b)

{

return (a / gcd(a, b)) \* b;

}

// Driver program to test above function

int main()

{

int a = 15, b = 20;

cout <<"LCM of " << a << " and "

<< b << " is " << lcm(a, b);

return 0;

}

**Output**

LCM of 15 and 20 is 60

***Time Complexity:****O(log(min(a,b))*

***Auxiliary Space:****O(log(min(a,b))*

Check for Prime

## What are prime numbers?

* A prime number is a natural number greater than **1**, which is only divisible by 1 and itself. First few prime numbers are: 2 3 5 7 11 13 17 19 23…..



*Prime numbers*

* In other words, the prime number is a positive integer greater than 1 that has exactly two factors, 1 and the number itself.
* There are many prime numbers, such as 2, 3, 5, 7, 11, 13, etc.
* Keep in mind that 1 cannot be either prime or composite.
* The remaining numbers, except for 1, are classified as prime and composite numbers.

## **Some interesting facts about Prime numbers:**

* Except for 2, which is the smallest prime number and the only even prime number, all prime numbers are odd numbers.
* Every prime number can be represented in form of **6n + 1** or **6n – 1** except the prime numbers 2 and 3, where n is a natural number.
* 2 and 3 are only two consecutive natural numbers that are prime.
* [Goldbach Conjecture:](https://en.wikipedia.org/wiki/Goldbach%27s_conjecture)Every even integer greater than 2 can be expressed as the sum of two primes.
* [Wilson Theorem](https://www.geeksforgeeks.org/wilsons-theorem/): Wilson’s theorem states that a natural number p > 1 is a prime number if and only if

(p - 1) ! ≡ -1 mod p

OR (p - 1) ! ≡ (p-1) mod p

* [Fermat’s Little Theorem](https://en.wikipedia.org/wiki/Fermat's_little_theorem): If n is a prime number, then for every a, 1 <= a < n,

an-1 ≡ 1 (mod n)

OR

an-1 % n = 1

* [Prime Number Theorem](https://en.wikipedia.org/wiki/Prime_number_theorem): The probability that a given, randomly chosen number n is prime is inversely proportional to its number of digits, or to the logarithm of n.
* [Lemoine’s Conjecture](https://www.geeksforgeeks.org/lemoines-conjecture/): Any odd integer greater than 5 can be expressed as a sum of an odd prime (all primes other than 2 are odd) and an even semiprime. A semiprime number is a product of two prime numbers. This is called Lemoine’s conjecture.

## Properties of prime numbers:

* Every number greater than 1 can be divided by at least one prime number.
* Every even positive integer greater than 2 can be expressed as the sum of two primes.
* Except 2, all other prime numbers are odd. In other words, we can say that 2 is the only even prime number.
* Two prime numbers are always coprime to each other.
* Each composite number can be factored into prime factors and individually all of these are unique in nature.

## Prime numbers and co-prime numbers:

It is important to distinguish between prime numbers and co-prime numbers. Listed below are the differences between prime and co-prime numbers.

* A coprime number is always considered as a pair, whereas a prime number is considered as a single number.
* Co-prime numbers are numbers that have no common factor except 1. In contrast, prime numbers do not have such a condition.
* A co-prime number can be either prime or composite, but its greatest common factor (GCF) must always be 1. Unlike composite numbers, prime numbers have only two factors, 1 and the number itself.
* **Example of co-prime:**13and 15 are co-primes. The factors of 13 are 1 and 13 and the factors of 15 are 1, 3 and 5. We can see that they have only 1 as their common factor, therefore, they are coprime numbers.
* **Example of prime:**A few examples of prime numbers are 2, 3, 5, 7 and 11 etc.

## **How do we check whether a number is Prime or not?**

**Naive Approach:**A naive solution is to iterate through all numbers from 2 to sqrt(n) and for every number check if it divides n. If we find any number that divides, we return false.

Below is the implementation:

C++Java

// A school method based C++ program to

// check if a number is prime

#include <bits/stdc++.h>

using namespace std;

// function check whether a number

// is prime or not

bool isPrime(int n)

{

// Corner case

if (n <= 1)

return false;

// Check from 2 to square root of n

for (int i = 2; i <= sqrt(n); i++)

if (n % i == 0)

return false;

return true;

}

// Driver Code

int main()

{

isPrime(11) ? cout << " true\n" : cout << " false\n";

return 0;

}

**Output**

**true**

**Time Complexity:** O(sqrt(n))  
**Auxiliary space:** O(1)

**Efficient approach:**To check whether  the number is prime or not follow the below idea:

*In the previous approach given if the size of the given number is too large then its square root will be also very large, so to deal with large size input we will deal with a few numbers such as 1, 2, 3, and the numbers which are divisible by 2 and 3 in separate cases and for remaining numbers, we will iterate our loop from 5 to sqrt(n) and check for each iteration whether that  (iteration) or (that iteration + 2) divides n or not. If we find any number that divides, we return false.*

Below is the implementation for the above idea:

C++Java

// A school method based C++ program to

// check if a number is prime

#include <bits/stdc++.h>

using namespace std;

// function check whether a number

// is prime or not

bool isPrime(int n)

{

// Check if n=1 or n=0

if (n <= 1)

return false;

// Check if n=2 or n=3

if (n == 2 || n == 3)

return true;

// Check whether n is divisible by 2 or 3

if (n % 2 == 0 || n % 3 == 0)

return false;

// Check from 5 to square root of n

// Iterate i by (i+6)

for (int i = 5; i <= sqrt(n); i = i + 6)

if (n % i == 0 || n % (i + 2) == 0)

return false;

return true;

}

// Driver Code

int main()

{

isPrime(11) ? cout << "true\n" : cout << "false\n";

return 0;

}

**Output :**

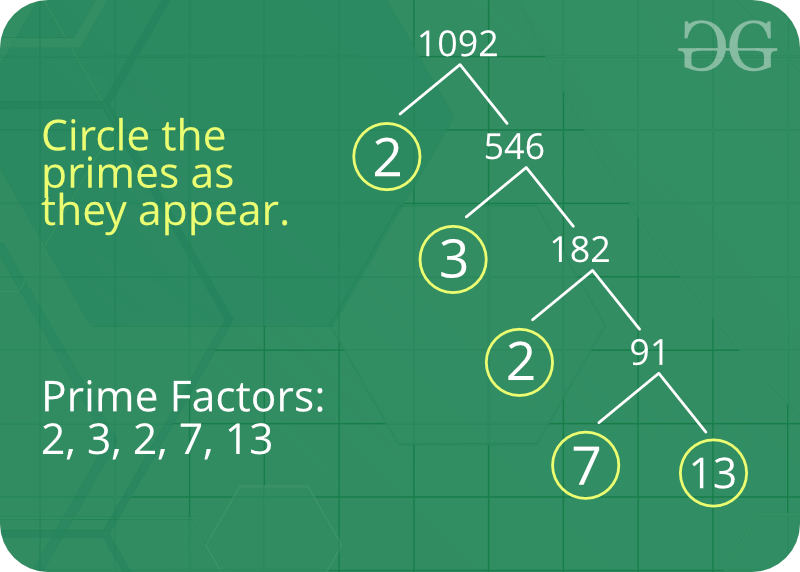
true

**Time complexity:**O(sqrt(n))  
**Auxiliary space:** O(1)

Prime Factors

Prime factor is the factor of the given number which is a [prime number](https://www.geeksforgeeks.org/prime-numbers/). Factors are the numbers you multiply together to get another number. In simple words, prime factor is finding which prime numbers multiply together to make the original number.

**Example:** The prime factors of 15 are 3 and 5 (because 3×5=15, and 3 and 5 are prime numbers).



**Some interesting fact about Prime Factor :**

1. There is only one (unique!) set of prime factors for any number.
2. In order to maintain this property of unique prime factorizations, it is necessary that the number one, 1, be categorized as neither prime nor composite.
3. Prime factorizations can help us with divisibility, simplifying fractions, and finding common denominators for fractions.
4. [Pollard’s Rho](https://www.geeksforgeeks.org/pollards-rho-algorithm-prime-factorization/) is a prime factorization algorithm, particularly fast for a large [composite number](https://www.geeksforgeeks.org/composite-number/) with small prime factors.
5. Cryptography is the study of secret codes. Prime Factorization is very important to people who try to make (or break) secret codes based on numbers.

**How to print a prime factor of a number?**  
**Naive solution:**   
Given a number n, write a 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”.

Following are the steps to find all prime factors:

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*, 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.

C++Java

// Program to print all prime factors

# include <stdio.h>

# include <math.h>

// A function to print all prime factors of a given number n

void primeFactors(int n)

{

// Print the number of 2s that divide n

while (n%2 == 0)

{

printf("%d ", 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)

{

printf("%d ", i);

n = n/i;

}

}

// This condition is to handle the case when n

// is a prime number greater than 2

if (n > 2)

printf ("%d ", n);

}

/\* Driver program to test above function \*/

int main()

{

int n = 315;

primeFactors(n);

return 0;

}

**Output:**

3 3 5 7

Time Complexity: O(sqrt(n))

Auxiliary Space: O(1)

**More efficient solution:**

C++Java

#include <iostream>

#include <limits.h>

using namespace std;

void printPrimeFactors(int n)

{

if(n <= 1)

return;

while(n % 2 == 0)

{

cout<<2<<" ";

n = n / 2;

}

while(n % 3 == 0)

{

cout<<3<<" ";

n = n / 3;

}

for(int i=5; i\*i<=n; i=i+6)

{

while(n % i == 0)

{

cout<<i<<" ";

n = n / i;

}

while(n % (i + 2) == 0)

{

cout<<(i + 2)<<" ";

n = n / (i + 2);

}

}

if(n > 3)

cout<<n<<" ";

cout<<endl;

}

int main() {

int n = 315;

printPrimeFactors(n);

return 0;

}

**Output:**

3 3 5 7