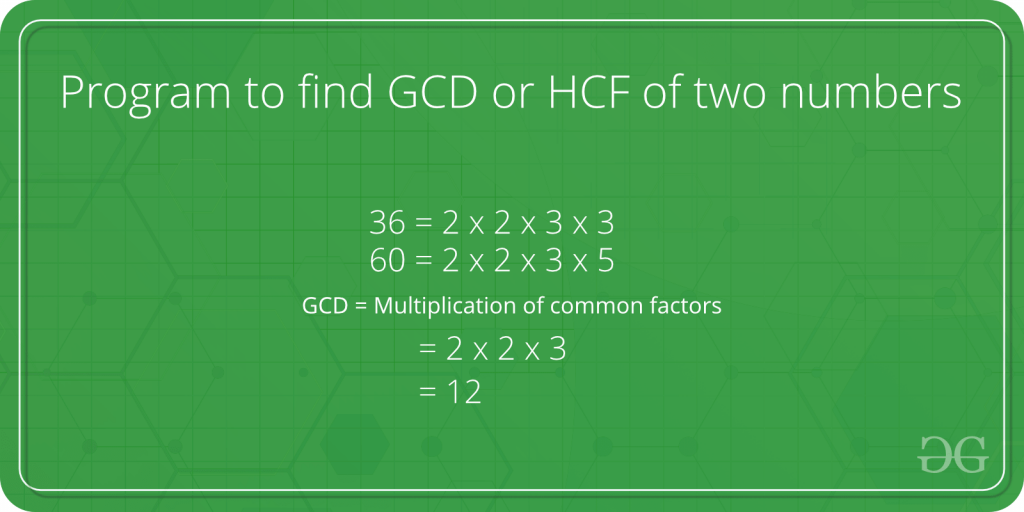
***GCD or HCF of two Numbers***

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

// Java program to find GCD of two numbers

class Test

{

// Recursive function to return gcd of a and b

static 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 method

public static void main(String[] args)

{

int a = 98, b = 56;

System.out.println("GCD of " + a +" and " + b + " is " + gcd(a, b));

}

}

**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

// Java program to find GCD of two numbers

class Test

{

// Recursive function to return gcd of a and b

static int gcd(int a, int b)

{

if (b == 0)

return a;

return gcd(b, a % b);

}

// Driver method

public static void main(String[] args)

{

int a = 98, b = 56;

System.out.println("GCD of " + a +" and " + b + " is " + gcd(a, b));

}

}

**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)).