Lesson:

Problems based on Recursion - 4

Pre-Requisites

- Recursion basics
- Working rules of recursive functions

List of Concepts Involved

- GCD using Recursion

Problem
Find the greatest common divisor (GCD) or HCF (Highest Common Factor) for the given two numbers by using the recursive function.

Understanding the Problem
What is GCD or HCF?
GCD or HCF of two numbers is the largest number that divides both of them.

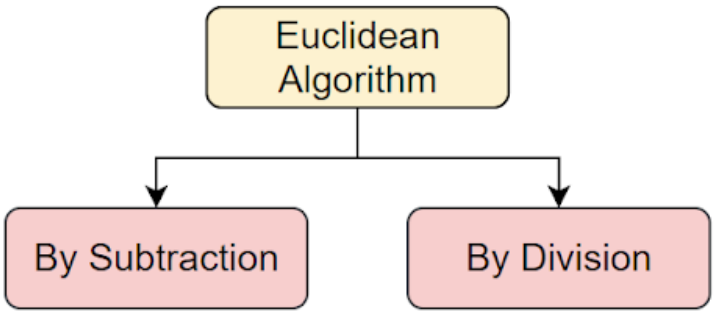
For example
GCD of 40 and 48
They have 4 common factors: 1, 2, 4, 8
Out of these 8 is the greatest and hence GCD = 8

GCD of 54 and 72
They have 6 common factors: 1, 2, 3, 6, 9, 18
Out of these 18 is the greatest and hence GCD = 18

Solution
We can solve this problem by Euclidean Algorithm.

Euclidean Algorithm
The algorithm is based on the below logic:

- 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.



Euclidean Algorithm by Subtraction

Say we need to find GCD(54, 72)

Steps:

1. 72 is greater than 54, so subtracting the smaller from the greater.
   72 - 54 = 18
2. Updating the greater value with this new value. So now it's GCD(54, 18).
3. 54 is greater than 18, so subtracting the smaller from the greater.
   54 - 18 = 36
4. Updating the greater value with this new value. So now it's GCD(36, 18).
5. 36 is greater than 18, so subtracting the smaller from the greater.
   36 - 18 = 18
6. Updating the greater value with this new value. So now it's GCD(18, 18).
7. At this point both values are equal, hence we can return any of them.
8. Returning 18 as the GCD.

Verification:

Factors of 54: 1, 2, 3, 6, 9, **18**, 27, 54
Factors of 72: 1, 2, 3, 4, 6, 8, 9, 12, **18**, 24, 36, 72
Common factors: 1, 2, 3, 6, 9, **18**
Greatest Common Factor: **18**

Java Code

https://pastebin.com/zuTrfZn1

Output

```
GCD(54, 72) = 18
```

Time Complexity: O(max(a,b)), Linear
Reason: Say in the worst case scenario, the smaller number is 1, then the recursive function will be called the max between the two numbers times.

Space Complexity: O(max(a,b)), Linear
Reason: Since we are using recursion, a call stack will be created, hence if we take that into account, space complexity won't be constant but linear.

Euclidean Algorithm by Division

Say we need to find GCD(40, 48)

Steps:

1. a = 40 & b = 48
   => a % b = 40 % 48 = 40
2. By updating values, we get, GCD(48, 40).
3. a = 48 & b = 40
   => a % b = 48 % 40 = 8
4. By updating values, we get, GCD(40, 8).
5. a = 40 & b = 8
   => a % b = 40 % 8 = 0
6. By updating values, we get, GCD(8, 0).
7. Now, b = 0, so we will return a which is 8.

Verification:

Factors of 40: 1, 2, 4, 5, 8, 10, 20, 40
Factors of 48: 1, 2, 3, 4, 6, 8, 12, 16, 24, 48
Common factors: 1, 2, 4, 8
Greatest Common Factor: 8

Java Code

https://pastebin.com/Z6nqiXJs

Output

```
GCD(40, 48) = 8
```

Time Complexity: O(log2(max(a,b))), Logarithmic
Reason: Here, at each recursive step, the gcd function will cut one of the parameters in at most half (that's why it is base 2).
For example: in the example above, we started with:
=> GCD(48, 40) => GCD(40, 8) => GCD(8, 0)

Comparison: Euclidean Algorithm Subtraction VS Division

Euclidean Algorithm by Subtraction

=> GCD(24, 2)
=> GCD(22, 2)
=> GCD(20, 2)
=> GCD(18, 2)
=> GCD(16, 2)
=> GCD(14, 2)
=> GCD(12, 2)
=> GCD(10, 2)
=> GCD(8, 2)
=> GCD(6, 2)
=> GCD(4, 2)
=> GCD(2, 2)
Ans = 2
Number of Steps = 12

Euclidean Algorithm by Division

=> GCD(24, 2)
=> GCD(2, 0)
Ans = 2
Number of Steps = 2

We can clearly see, how many steps are being reduced when we used the division algorithm.
Hence, in this case, the time complexity is the log of the time complexity in the subtraction algorithm.

Space Complexity: O(log2(max(a,b))), Logarithmic
Reason: Clearly in the worst case our recursive function will be called log2(max(a,b)) times, and each time it is called, in the backend a recursive stack will be created whose size will be of the number of times the function is called. Hence, if we take into account that call stack our space complexity won't be constant.

Upcoming Class Teasers

- Problems based on recursion.