Laboratory 1

Mos Daniela, 935

The purpose of this application is to present three methods to compute the greatest common divisor of two integer numbers.

1. Repeated divisions

The Euclidean algorithm works by successively dividing the numbers and swapping them with the quotient and the remainder until the second number reaches the value 0. So, the steps to this algorithm are the following: 1. Divide the larger number with the smallest number and keep the quotient and the remainder 2. Swap the places of the divisible with the quotient and the divisor with the remainder and repeat until the remainder is 0.

Then the last quotient will be the greatest common divisor of the two numbers.

To illustrate the algorithm, I will take an example. Let a = 2256, b = 345

2256 = 345 * 6 + 186
345 = 186 * 1 + 159
186 = 159 * 1 + 27
159 = 27 * 5 + 24
27 = 24 * 1 + 3
24 = 3 * 8

then we find gcd(2256, 345) = 8

Proof

I will start the proof by stating the following properties: 1. if b < a and b|a, then gcd(a, b) = b 2. gcd(a, b) = gcd(b, a mod b)

The first property is straight forward, that if a smaller number divides a larger one, their greatest common divisor will be the smaller number, but the second one states that if a number divides a and b, it will divide their modulus aswell. From the left hand side we can write that $a = d * m, b = d * n$, where d is their common divisor and m, n integers and gcd(b, a mod b) = d. We also know that d | d, then from the right hand side we have $ a mod b = (d * m) mod (d * n)$ which is divisible by d. By repeatedly computing the quotiend and the remainder in this way, the remainder will reach 0, stopping the algorithm, and the last equation will be in the form of $q_{k-1} = q_k * r_{k-1}$, where the divisor is the last quotient.

Implementation

```
<<Euclid>>=
def euclid(a, b):
    if a == 0:
        return b
    if b == 0:
        return a
    a, b = max(a, b), min(a, b)
    while b != 0:
        aux = b
        b = a % b
        a = aux
    return a

@
```

2. Repeated subtractions

This algorithm is a variation of the repeated divisions algorithm, but this one consists of subtracting the smaller number from the bigger one, until they are equal and the resulting number is the greatest common divisor. The proof is also similar, since the second property can be proved in the same way for the subtraction of the two numbers.

Example for a = 539, b = 928

928 - 539 = 389
539 - 389 = 150
389 - 150 = 239
239 - 150 = 89
150 - 89 = 61
89 - 61 = 28
61 - 28 = 33
33 - 28 = 5
28 - 5 = 23
23 - 5 = 18
18 - 5 = 13
13 - 5 = 8
8 - 5 = 3
5 - 3 = 2
3 - 2 = 1
2 - 1 = 1

We find that a = b = 1 and we stop the algorithm. Then gcd(a, b) = 1. It is clear that the number of steps for this algorithm is larger than the one with repeated divisions.

Implementation

```
<<Subtraction>>=
def repeated_sub(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return abs(a)

@
```

3. Factorization algorithm

The idea of this algorithm is to find the common prime factors of each of the two numbers. The greatest common divisor is the product of these terms, at the smallest power.

For example, I will compute the greatest common divisor of the numbers a = 456 and b = 782.

$456 = 2^3 * 3 * 19$
782 = 2 * 17 * 23

After taking each common factor at the lowest power, the gcd(456, 782) = 2.

Proof

It is easy to see that the algorithm only takes the common primes (divisors) of both numbers and multiplies them in order to obtain the greatest common divisor.

Implementation

Each iteration of the program below consists of choosing the smallest common prime number lower than the square root of the smallest value of the two numbers.

```
<<Factorization>>=
def factorization(a, b):
    gcd = 1
    nr = 2
    while nr < int(math.sqrt(a)) + 1 and nr < int(math.sqrt(b)) + 1:
        while a % nr == 0 and b % nr == 0:
            gcd *= nr
            a /= nr
            b /= nr
        nr = next_prime(nr + 1)
    return gcd
@
```

In order to find the next prime number, I used the trial division algorithm which consists of checking if the given number is composite by finding divisors smaller than its square root. The limitation of the search to the square root of the number comes from the fact that after finding the greatest divisor, there can be another one as large only if the number is the square of that divisor.

```
<<Find next prime>>=
def next_prime(a):
    while not check_prime(a):
        a += 1
    return a
@
```

```
<<Check if prime>>=
def check_prime(a):
    a = abs(a)
    if a == 0 or a == 1:
        return False
    for i in range(2, int(math.sqrt(a)) + 1):
        if a % i == 0:
            return False
    return True
@
```

Testing

The testing function will take the two numbers and print their greatest common divisor, as found by each algorithm, and their running time. The observations that can be made are that the repeated divisions algorithm's running time is way below 1 second even for larger numbers, whereas the factorisation algorithm takes a lot of time to finish. The repeated subtractions algorithm's running time depends on the number of steps it needs to reach the equality of the two numbers.

```
<<Testing>>=
def testing(a, b):
    print(f'Running for: a={a}, b={b}')

    start = time.time()
    eu = euclid(a, b)
    end = time.time()
    eu_time = format(end - start, '.8f')
    print(f'Euclidean algorithm: {eu}, run for {eu_time}')

    start = time.time()
    repeated = repeated_sub(a, b)
    end = time.time()
    repeated_time = format(end - start, '.8f')
    print(f'Repeated subtractions algorithm: {repeated}, run for {repeated_time}')

    start = time.time()
    fact = factorization(a, b)
    end = time.time()
    fact_time = format(end - start, '.8f')
    print(f'Factorization algorithm: {fact}, run for {fact_time}')

    print()
@
```

Putting everything together:

```
<<*>>=
import math
import time

<<Euclid>>
<<Subtraction>>
<<Factorization>>
<<Check if prime>>
<<Find next prime>>
<<Testing>>

if __name__ == "__main__":

    testing(2 ** 35, 2 ** 33)
    testing(2 ** 24 - 1, 2 ** 20 - 1)
    testing(2 ** 25 - 1, 2 ** 23 - 1)
    testing(2 ** 45, 2 ** 51)
    testing(2 ** 154 - 1, 2 ** 157 - 1)

@
```