We use Euclid's division algorithm in order to find the numbers that are respectively prime (their gcd(a,b) = 1)

This algorithm works by dividing the greater number by the smaller* and taking the remainder, we do this by using %. We do this until the b becomes 0. Our gcd value will be found in variable a. {* we know this because if we use Smaller%Greater the modulo is the smaller one and they are simply switched} (I used the explanation from the last lab)

```
<<Gcd Division>>=
def gcdDivision(a, b):
    while b != 0:
        a, b = b, a % b

    return a
@
```

This is needed for the testing of the relation one. We just iterate through [1,n] and we append them if they divide n We return the list of divisors

```
<<Get Divisors>>=
def getDivisors(n):
    divisors = []
    i = 1
    while i <= n:
        if (n % i == 0):
            divisors.append(i)
        i = i + 1
    return divisors
@
```

This is the implementation of our 'star' function. The function computes the cardinal of the set composed by numbers less than n that are respectively prime with n We iterate [1,n] and each time the number is relatively prime with n, we just increase the cardinal value;

```
<<Phi>>=
<<Gcd Division>>
def phi(n):
    cardinal = 0
    for k in range(1, n + 1):
        if (gcdDivision(k, n) == 1):
            cardinal += 1

    return cardinal
@
```

This function test if the implemented phi function holds the Gauss's property. This function checks if the sum of all values of the phi function for n's divisors

is equal to n. We get the n's divisors, plug them in the phi function and add each to the sum We return a boolean representing the equality of the sum to n.

```
<<Test1>>=
<<Get Divisors>>
def test1(n):
    sum = 0
    divisors = getDivisors(n)
    for item in divisors:
        sum += phi(item)

    return sum == n
@
```

I'm not really sure how this test works(or what it should test), but I was able to implement it We iterate from 1 to 999 (I couldn't find a way to do sum of a series in python so I improvised) the value of phi of each number and divide it to 2^n-1. At the end we check if the rounded sum is equal to 2.

```
<<Test2>>=
def test2():
    sum = 0
    for n in range(1, 999):
        sum += (phi(n) / (pow(2, n) - 1))


    #this needed to be rounded, it did converge to 2 (it was 1.(9))
    return sum.__round__() == 2
@
```

Auxiliar function for finding the most common value; It receives a dictionary that contains pairs of (value,frequency) We sort it based on the frequency in a list To get the wanted result we just take the last element

```
<<Most Common Value>>=
def getMostCommon(dictNumberOccurence):
    sortedDict = sorted(dictNumberOccurence.items(), key=lambda x: x[1])
    return sortedDict[len(sortedDict)-1][0]
@
```

Auxiliar function for showing the data ~~~{.python} <>= def plotHistogram(dictNumberOccurence): import matplotlib.pyplot as plt

```
lists = sorted(dictNumberOccurence.items())

x, y = zip(*lists)

plt.plot(x, y)
plt.ylabel('values of phi funct')
```

```
plt.show()
plt.show()
```

@ _~~

Auxiliar function for showing the data

```
<<*>>=
<<Phi>>
<<Test1>>
<<Test2>>
<<Most Common Value>>
<<Plot Histogram>>

def run():
    value = int(input("please input the value: "))
    bound = int(input("please input the bound: "))
    dictNumberOccurence = {}
    listOfNumbersWithV = []

    for i in range(1, bound + 1):
        dictNumberOccurence[i] = 0

    for index in range(1, bound + 1):
        if (phi(index) == value):
            listOfNumbersWithV.append(index)
        dictNumberOccurence[phi(index)] = dictNumberOccurence[phi(index)] + 1

    mostCommonV = getMostCommon(dictNumberOccurence)

    plotHistogram(dictNumberOccurence)

    print("The numbers that have v as the value are: ")
    print(listOfNumbersWithV)
    print("the most common value is: " + str(mostCommonV))

    print("test1 is " + str(test1(20)))
    print("test2 is " + str(test2()))

run()
@
```

Regarding the discussion:

I did not make a statistic of what are the most common values for each run. But the most common value is printed individually for each run. And I did make a histogram for when b is 1000, what I did observe is that the closer the values get to the bound, the phi function values get smaller and smaller. Also, I do not

know if this is true or not, but it seemed like there was some kind of simetry in how the values were distributed. This might be random, but it looked like everytime there was a value that had a large value for the phi function, the values before and after it were simetrically small. Maybe I should've noticed something else in addition, but that's it.