


Dev Raj Kandel

DipHE Mathematics with Computing, Middlesex University, United Kingdom

Article History : Submitted 05 Sep. 2023; Reviewed 19 Sep. 2023; Accepted 01 Oct. 2023

Corresponding Author : Dev Raj Kandel, **Email** : geniusppdevraj@gmail.com

DOI : <https://doi.org/10.3126/kdk.v4i1.64567>

 Copyright 2024 © the Author(s) and the Publisher

Abstract

In this article, the prime motivation is to demonstrate how calculating modular multiplicative inverse can be simplified computationally with the help of Euclid's algorithm, which is usually attributed to finding the Greatest Common Divisor.

Keywords: Algorithm, conjugacy class, data structure, java, modulus, number theory

Introduction to Modular Arithmetic

Quite central to the study of Number Theory[10], Modular Arithmetic provides a unique way of understanding different number systems, mainly the **Integers** [2][4][6][8]. Unequivocally, this further emphasizes the need for asking and solving different problems within “Modular Arithmetic”, eventually anticipating huge insights into solving big questions within both Pure and Applied Mathematics. The whole point of this report is to solve one such problem.

It all starts with understanding the basic “modulo operation” [9]. If “a” and “b” are any two integers, the “a mod b” returns the remainder when “a” is divided by “b”. Mathematically speaking,

$$a \bmod b = r \text{ iff } a = qb + r \text{ where } q, b, r, a \in \mathbf{Z}$$

Now, though the operation basically looks very simple, it can be quite a powerful tool for understanding the whole set of Integers (\mathbf{Z}). For example, take any natural number, $n \in \mathbf{N}$, then for any non-negative integer, $a \in \mathbf{Z}^+$, $0 \leq a \bmod n \leq (n - 1)$. This interesting discovery can, infact, be written in set notation as $\mathbf{Z}_n = 0, \dots, n - 1$. Now all the elements of this set are called **conjugate classes** of the equivalence relation $\equiv a \bmod n$, which also implies, $a = c + nk, \forall k \in \mathbf{Z}$. Now, all the possible values of ‘a’, given particular value of ‘c’ and ‘n’, can be called as the equivalence class of ‘c’, also written as [c]. Finally with different values of ‘c’, the equivalence relation partitions the whole of \mathbf{Z} . All these rigorous ideas can be understood in the example provided below.

Example: For $n = 5$,

Set of conjugacy classes = $Z_5 = \{0, 1, 2, 3, 4\}$

Examples of Equivalence classes:

Example 1: $[2] = \{\dots, -18, -13, -8, -3, 2, 7, 12, 17, \dots\}$.

Example 2: $[3] = \{\dots, -17, -12, -7, -2, 3, 8, 13, 18, \dots\}$

Note: $\{[0], [1], [2], [3], [4]\}$ partitions Z .

Now, normal arithmetic operations like $(+, -, *)$ can be easily defined and performed as:

Given, Z_n ,

$$[a] + [b] = [a + b]$$

$$[a] - [b] = [a - b]$$

$$[a] * [b] = [a * b]$$

Examples:

Given, Z_5 ,

$$7 + 8 = 15$$

This shows: $[2] + [3] = [0] (5 \text{ mod } 5 = 0)$

Also,

$$7 * 8 = 56$$

This shows:

$$[2] * [3] = [1] (1 = 6 \text{ mod } 5)$$

Modular Multiplicative Inverse and its Essence

Unlike the operators discussed in previous section, finding modular multiplicative inverse isn't straightforward. But what is modular multiplicative inverse? Mathematically speaking,

In Z_n ,

The inverse of $[a]$ is $[b]$ if $[a][b]=1$.

For example,

In Z_5 ,

The inverse of $[2]$ is $[3]$ because $[2]*[3] = [1]$.

We can see that inverse, if it exists, is always unique. Also, there is no inverse of 0 in Z_n . This fact will be used, later in producing efficient solution to the problem being discussed. However, it rather makes much sense to answer why do we care a lot about this?

Apart from being an interesting problem in mathematical theory, the concept of modular multiplicative inverse is used heavily in public-key cryptography- RSA being a major one. To be specific, a number and its modular multiplicative inverse are used as key pairs to encrypt and decrypt the data simultaneously. Usually in cryptographic procedures, computer scientists always aim to produce algorithms which are computationally cost effective. And, it turns out that manipulating **Euclid's Algorithm**, which will be discussed later, is the most effective one.

Euclid's Algorithm

In the previous page, it is claimed that "Euclid's" Algorithm can be utilized in producing a solution for finding the modular multiplicative inverse of a number 'a' in \mathbf{Z}_n . Before that, it is useful to know how Euclid's algorithm works and what it is primarily used for.

Basically, Euclid's algorithm, which culminated from Euclid's and Bezout's Lemma, is used for finding the greatest common divisor ($\gcd(a,b)$) of a and b assuming $a < b$. The algorithmic procedure works like this:

1. Express $b = qa + r$ where q is the quotient, r is the remainder, a the divisor.
2. If 'r' is 0, then terminate the algorithm and record $\gcd(a,b) = a$. [7]
3. Else if 'r' isn't 0, then set new dividend (b) to the previous divisor (a) and new divisor (a) to the previous remainder(r). Repeat step 1.

Now, this algorithm is not only the part of finding the modular inverse, but also a way to avoid errors as it tells if inverse exists or not. **There exists inverse of 'a' in \mathbf{Z}_n , if and only if $\gcd(a,n)=1$. Equivalently, if a and n are co-prime.**

So we can use this algorithm to prevent errors before attempting to find inverse. On practical terms, if $\gcd(a, n) \neq 1$, then we can computationally terminate the process of finding inverse.

Example: *Aim: To find $\gcd(7, 23)$.*

$$23 = 3 \times 7 + 2$$

$$7 = 3 \times 2 + 1$$

$2 = 1 \times 2 + 0$: In this stage, the algorithm terminates recording $\gcd(a,b) = a = 1$
Apparently, $\gcd(7,23) = 1$. So inverse of [7] exists in \mathbf{Z}_{23}

Why $\gcd(a,n) = 1$ is compulsory for inverse to exist?

First we need to understand **Bezouts Lemma**.

Bezouts Lemma [11]

Let a, b be non-zero integers.

Then there exists r, s $\in \mathbf{Z}$ such that $\gcd(a, b) = ra + sb$.

Now,

To prove: In \mathbf{Z}_n , the inverse of [a] exists if and only if $\gcd(a,n)=1$.

Proof: Suppose [a] has inverse [b] in \mathbf{Z}_n . That means $[a][b]=1$

$$\Leftrightarrow ab \bmod n = 1$$

$$\Leftrightarrow ab = 1 + kn$$

$$\Leftrightarrow ab - kn = 1$$

From Bezouts Lemma.

$$\Leftrightarrow \gcd(a, n) = 1$$

Extending Euclid's Algorithm- Full Solution

Euclid's algorithm is just a half-way to the process of finding the inverse. Now, to find the inverse of a in \mathbf{Z}_n , we need to express '1' in the form of $k \times a + l \times n$. This can be

done by keeping 1 as the subject and reverse engineering the process of Euclid's algorithm. Once completed, the value of k is the required inverse value of $[a]$ in Z_n . Mathematically, $[k]$ is the inverse of $[a]$ in Z_n . Or, $[k] = [a]^{-1}$

We can explain why this is the case.

We know: $k \times a + l \times n = 1$

Now, taking (mod n) on both sides,

$$(k \times a + l \times n) \text{ mod } n = 1 \text{ mod } n$$

Or, this implies

$$k \times a = 1$$

Obviously, in Z_n , we know $[k] = k + g \times n$ where g is any integer.

Also $[a] = a + h \times n$, where h is any integer.

$$\begin{aligned} [k][a] &= k \times a + k \times h \times n + g \times a \times n + g \times h \times n^{(2)} \\ &= 1 + n(kh + ga + ghn) \end{aligned}$$

Now,

$$[k][a] \text{ mod } n = 1 \text{ mod } n$$

$$[k] \times [a] = 1$$

Let's see the example here how it works:

We are going to find the inverse of 7 in Z_{23} .

Using information from Page 6,

$$1 = 1 \times 7 + (-3) \times 2$$

$$1 = 7 - (23 - 7 \times 3) \times 3$$

$$1 = 10 \times 7 + (-3) \times 23$$

Finally, we have $1 = k \times a + l \times n$

Ultimately $[10]$ is the inverse of 7 in Z_{23} .

How will we transform this process to an algorithm?

Although, it is easy to manipulate mathematical expressions in paper, we need to find 'k' and 'l' algorithmically.

Note: We have already run the Euclid's algorithm before executing this algorithm.

Step 1

First we need to store all the quotients during Euclid's algorithm in each step to some dynamic array variable named "quotient Stack" variable [1]. We are going to exclude quotients of those steps when remainder is either 0 or 1.

For example: QuotientStack (while finding gcd (7, 23)) = [3].

$$\text{QuotientStack (while finding gcd (20, 73))} = [3 \ 1 \ 1]$$

Step 2

Initialize two new variables "ksequence" to 1 and "lsequence" to $(-1 * (\text{last quotient in Euclid's algorithm when remainder is 1}))$.

Step 3

Loop from bottom of the QuotientStack till you reach the first element.

```
ksequence=ksequence+(-1)*lsequence*<current quotient>.
```

Then on second run(in the loop) execute

```
->lsequence =lsequence - <current quotient>*ksequence
```

<currentquotient> means the quotient at the particular run of the loop. We know the loop starts from backwards. So at first run, we access the last element and then second last element and so on.

Keep running these two commands alternatively till you reach the end of the loop. Note: The number of times the loop runs depends on the size of the Quotientstack. However, running two different commands alternatingly requires a different variable which we can call "alternator". The "alternator" just makes sure that 2 different commands run alternatingly inside the loop.

Step 4: After Exiting the loop:

```
IF ksequence*a+lsequence*n=1 THEN
RETURN ksequence
ELSE
RETURN lsequence
```

There is no definite mathematical justification on how this algorithm works. I just conjectured how the "quotients" we calculated during the "Euclid's" algorithm affects the numbers during the reverse process. In short, I have just tried to look at the pattern and conjectured accordingly. Despite all this, this algorithm works on all values of 'a' and 'n'. We will see, how this algorithm runs perfectly with different values in upcoming sections.

Working Demonstrations:

Example 1) Inverse of 12 in Z_{31}

Here $a = 12, n = 31$

$$31 = 2*12+7$$

$$12 = 1*7+5$$

$$7 = 1*5+2$$

$$5 = 2*2+1$$

$$2 = 1*2+0 \text{ (Terminate)}$$

We can also terminate the procedure already when we get remainder 1.

In the Java Programming Assignment, we terminate once we get remainder 1 or 0.

$\text{Gcd}(12, 31) = 1$. So inverse exists.

Quotientstack: [2 1 1] (Note: We exclude quotients in those lines where remainder is 1 or 0).

Now, $k_{sequence} = 1$ and $l_{sequence} = -2$ (Because, 2 is the quotient where the remainder is 1). (To be continued).

Looping from backwards:

Run in the loop	Current Quotient	ksequence	lsequence
1	1	$1 + (-1) * 2 * 1 = 3$	-2
2	1	3	$-2 - (1 * 3) = -5$
3	2	$3 + (-1) * (-5) * 2 = 13$	-5

Now, $13(k_{sequence}) * 12(a) + l_{sequence}(-5) * 31 = 1$. So inverse is ksequence which is 13. Or equivalence class of 13 which is [13].

Sometimes we might get a different case where $k_{sequence} \times a + l_{sequence} \times n \neq 1$. In those cases, inverse is the value of lsequence. One such example is: Finding inverse of 7 in Z_{12} . The lsequence in this case is -5. So, inverse is -5. We can convert this negative number into its positive equivalent by using a different function named “PositiveEquivalent”. Finally the answer will be $7(-5 + 12 * 1)$. This will be discussed later in upcoming sections.

Critical Analysis of the Proposed Method- Including Algorithmic Complexity

From computational point of view, manipulating Euclid’s algorithm is the most effective way. It runs in $O(\log(n)^2)$ given that we are finding the inverse of ‘a’ in Z_n assuming $a < n$. However, if computational running time complexity isn’t a big deal, the problem of modular multiplicative inverse can be solved in two other ways as well. The first one is the direct one:

1. Direct Algorithm.

```

INPUTS: N ,A
FOR X = 0 TO N-1
IF (X*A) MOD N = 1 THEN
INVERSE=A
EXIT LOOP
END IF
END LOOP
OUTPUT INVERSE
    
```

This algorithm has run-time complexity of $O(n)$, which is bad when n tends to be very large.

The second way is to use **Euler’s theorem**.

Applying Eulers’ theorem, if ‘a’ and ‘n’ are co-prime, then,

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

Here ϕ is euler totient function which basically gives the number of positive integers less than n which are co-prime to ‘n’.

Now multiplying a^{-1} on both sides we get,

$$a^{\phi(n)-1} \equiv a^{-1} \pmod{n}$$

Now $a^{\phi(n)-1}$ is the required modular multiplicative inverse of a in \mathbb{Z}_n . It can be observed that this method is slightly slower than the method proposed in the solution because it takes long polynomial time to calculate the value of Euler totient function. Overall, on comparison, using Euclid's algorithm seems practical for industrial purposes. But one major disadvantage of this method is that it is hard to explain the "Reverse process" and algorithmic logic to others.

Full Algorithmic Implementation in Java

The theoretical implementation of Euclid's algorithm is given in previous sections. Here, in this section, we will see the how the solution is implemented practically in Java.

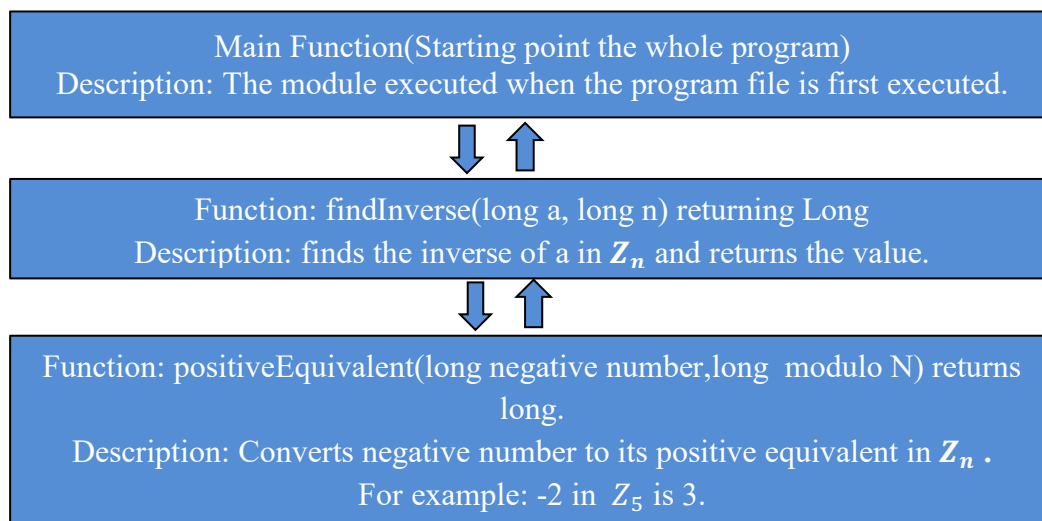


Figure 1: Functions in Rectangles and Arrows Signifying flow of data in-between functions.

Explanation:

The main program is divided into 3 function /procedure. The first one is a default procedure which is automatically executed when the program is run for the first time. The "main" function calls second function called `findInverse(long a, longn)` which finds the inverse of a in \mathbb{Z}_n . Then the output is returned back to the main function. However, for 'findInverse' to be executed successfully **may** require the invocation of third function 'positiveEquivalent'. The 'positiveEquivalent' function returns value to the 'find Inverse' function, if called depending upon the situation.

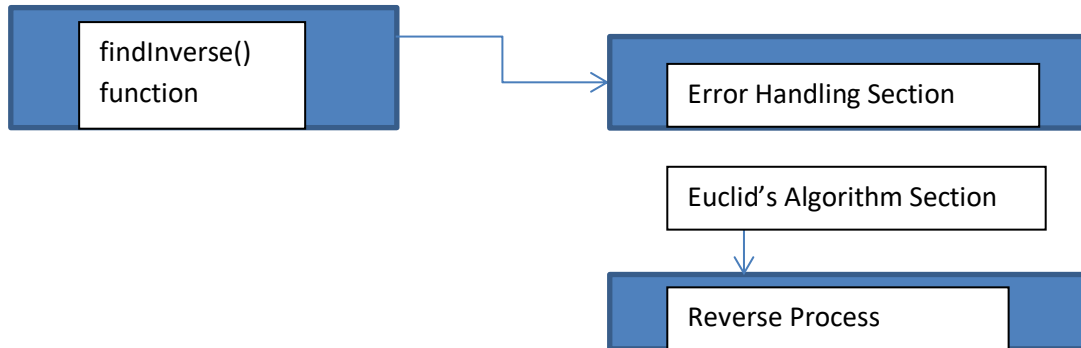
Breakdown of (find Inverse (long a, long n) returns long) function

Figure 2: Schematic Breakdown of findInverse function and arrows showing the flow of the program

Description:

The findInverse function contains 3 sections. These are: Error Handling Section, Euclid's Algorithm Section and Reverse Process Section.

The 'findInverse' function takes two inputs 'a' and 'n'. It ultimately returns the inverse of 'a' in \mathbb{Z}_n .

Error Handling Section

Before allowing the inputs 'a' and 'n' to go through the real algorithm, there are certain values of 'a' and 'n' which can create problem or create incorrect result while applying Euclid's algorithm and Reverse Process Section. So, we need to detect the error and throw exceptions. These will be discussed later in Error Detection Section.

Euclid's Algorithm Section

Coming after Error Handling Section, the main purpose of this section is to find the $\gcd(a,n)$ and to store the quotients(for further use in reverse process section). Note that we will only go to reverse process section if $\gcd(a,n)=1$. If $\gcd(a,n)$ doesn't equal to 1, an error(exception) will be thrown.

The variables declared/intialized in this section are:

1. *mainA*(for storing the value of a for further use),
2. *mainN*(for storing the value of n for further use)
3. *QuotientStack*(a dynamic array variable for storing quotients),
4. *gcd* (storing the gcd),
5. *ksequence*(equivalent of k in $ka+ln$)
6. *lsequence*(equivalent of l in $ka+ln$)

7. r (for remainder)
8. q (for quotient).

The method in which we carry out Euclid's algorithm is same as the procedure given in previous sections. We start with 'n' being the dividend and 'a' being the divisor. We keep looping using 'Do—while' loop structure until we get remainder 'r' to be either equal to 0 or 1. To calculate remainder, modulus in built operator('%') is used. Then to get the quotient we subtract remainder 'r' from 'n' and do the integer division of the result by 'a'. We then, keep changing the next divisor and dividend as according to the instruction in sections. Also, when we calculate the quotient every time, we add quotients to the quotient stack if the remainder isn't 0 or 1. Finally, when the loop is exited, the latest remainder becomes the gcd. Then, if the gcd is 1 then we set the lsequence to latest value of quotient(q)*-1. On contrary, if gcd isn't 1, we throw an error saying gcd isn't 1. The whole program comes to halt in such situation. The user must enter new values of 'a' and 'n' and then rerun the whole program.

Reverse Process Section

This is the final and the most important section. The main purpose of this section is to calculate the inverse of a in Z_n . We only enter this section if we are sure that the inverse exists that is $\gcd(a,n) = 1$. We declare a variable 'alternator' and initialize it to 1.

The reverse process happens like this. We first calculate the length of quotientstack and store it in sizeofStack variable of type, 'integer'. We then create a for-loop running from bottom of 'quotientStack' variable. Since, the index of array variable starts from 0, we start the for-loop from sizeof Stack-1 to 0. (that is in reverse order). We execute two statements alternatively within the loop. The first statement modifies the value of ksequence and the second statement modifies the value of lsequence. As seen in the statements themselves, the values of ksequence and lsequence depend on the value of current value of quotient accessed from quotientStack variable and the values themselves. Since, nature of odd and even number alternates, we use modulus(%) operator on alternator variable to run the statements alternatively during the loop. The alternator variable keeps increasing by 1 during the loop.

After the loop exits, our required answer is either the value of ksequence or lsequence. We need to check that by checking if $ksequence * a + lsequence * n = 1$. If this evaluates to true, our required answer is ksequence. Otherwise the inverse of a is lsequence. For example, while finding the inverse of 7 in Z_{12} , the inverse value is lsequence which is -5.

Now, in such cases when the inverse comes in negative, we need to convert it to positive equivalent value in modulo n. We can check if the inverse is negative by using If conditional statement. We have created positive Equivalent function for this.

The positive Equivalent function takes the two parameters 'negativeValue' and 'moduloN'. We will then keep increasing the value of a new variable k, which starts form 1. This is done, until the value of $negativeValue+k*moduloN$ is non-negative. Finally the function returns the positive value. To understand, why we do this, we need to understand the concept of equivalence class from the previous sections. Since the negative number is in

the same equivalent class as the positive numbers, we can use the formula above to the equivalent positive value.

Some examples where we get negative inverse is finding the inverse of 7 in Z_{12} the inverse of 5 in Z_6 .

Error Handling

The program might encounter different errors (some runtime errors) or incorrect outputs. To prevent that error handling has been implemented [5]. Basically, we run the program by calling the function `findInverse(a,n)`. If any developer wishes to use this function with incorrect inputs, then custom errors with messages will be thrown. Here are some the error handling cases:

- 1) If user/developer puts data with wrong type. For example, in `findInverse(a,n)` the expected data type is long for both a and n. If double data is entered, the program will throw type error.
- 2) If user/developer inputs n less than 1. We know that 'n' is a natural number. If user/developer puts n less than 1, then a custom error will be thrown saying that "n" should be greater than or equal to 1.
- 3) If user/developer inputs n equal to 0. We know that when n is 1, the conjugacy classes will only be {0}. We also know 0 has no inverse. So, a custom error will be thrown saying "No element with inverse exists in the conjugacy class of modulo 1".
- 4) If user/developer inputs a as 0, we know 0 has no inverse in any value of Z_n . A custom error will be thrown saying "0 has no inverse in $Z<N>$ ".
- 5) If user/developer inputs pair of a and n whose $\gcd(a,n)$ isn't 1. In such cases a custom error will be thrown saying "Gcd isn't 1. So, inverse doesn't exist."

Note: When error is thrown, the whole program will not proceed further. The user is either expected to use try catch statement or rerun the whole program with different appropriate input. If any developer wishes to use this function, he can easily see the error message while developing his/her respective application.

Conclusion/ Successful Testing of Algorithm

Here, we will see how the algorithm works when putting different values of 'a' and 'n' in the `findInverse()` function.

Case Scenario 1) When user/developer input $a < 0$: In this case, first 'a' is converted to its smallest positive equivalent in modulo n. For example, -2 in mod 5 will be first converted to 3 and then inverse is calculated using 'a' as 3

Case Scenario 2) When user/developer inputs $a > n$: Ideally a is supposed to be in between and including 0 to n-1. So, it should be a incorrect input. However, in modulo n, a can be represented to be a value in between 0 and n-1. And then inverse can be found. The java program produces correct input even when $a > n$. No further checking or manipulation is required.

Other cases:

findInverse (-5,12): $a < 0$. The program produces correct answer 7.

findInverse (7,12): The program produces correct answer 7.

findInverse(3,9): Error is thrown saying "Inverse doesn't exist.GCD isn't 1. "

findInverse(7,1111):The program produces correct answer 635.

findInverse(0,3): The program throws custom error as described in previous sections.

findInverse(4.2,9): Type error will be thrown.

findInverse(5,6): The program produces correct answer 5

When error is thrown, it is the responsibility of user/developer to put correct inputs.

References

- [1] *Stack data structure*. (n.d.).
https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm
- [2] *Modular arithmetic | brilliant Math & Science* (n.d.).
<https://brilliant.org/wiki/modular-arithmetic/>
- [3] Ramuglia, G. (2023, November 17). *Java functions explained: your ultimate guide*. Linux Dedicated Server Blog. <https://ioflood.com/blog/java-functions/>
- [4] Libretexts. (2022b, May 19). *3.1: Modulo operation*. Mathematics LibreTexts.
https://math.libretexts.org/Courses/Mount_Royal_University/MATH_2150%3A_Higher_Arithmetic/3%3A_Modular_Arithmetic/3.1%3A_Modulo_Operation
- [5] *Exception handling in java | Java exceptions - javatpoint*. (n.d.). www.javatpoint.com.
<https://www.javatpoint.com/exception-handling-in-java>
- [6] Libretexts. (2022, April 17). *7.3: Equivalence classes*. Mathematics LibreTexts.
[https://math.libretexts.org/Bookshelves/Mathematical_Logic_and_Proof/Book%3A_A_Mathematical_Reasoning_Writing_and_Proof_\(Sundstrom\)/07%3A_Equivalence_Relations/7.03%3A_Equivalence_Classes](https://math.libretexts.org/Bookshelves/Mathematical_Logic_and_Proof/Book%3A_A_Mathematical_Reasoning_Writing_and_Proof_(Sundstrom)/07%3A_Equivalence_Relations/7.03%3A_Equivalence_Classes)
- [7] Jafar, W., & Jafar, W. (2023, July 5). *How to find Greatest Common Divisor (GCD)? Definition, examples*. Splash learn - Math vocabulary.
<https://www.splashlearn.com/math-vocabulary/greatest-common-divisor-gcd>
- [8] *Number theory - Modular arithmetic*. (n.d.).
<https://crypto.stanford.edu/pbc/notes/numbertheory/arith.html>
- [9] Geeks for Geeks. (2023, July 20). *Operators in java*.
<https://www.geeksforgeeks.org/operators-in-java/>
- [10] *Number systems: naturals, integers, rationals, irrationals, reals, and beyond*. (n.d.).
https://www.varsitytutors.com/hotmath/hotmath_help/topics/number-systems
- [11] Libretexts. (2023, July 28). *4.2: Euclidean algorithm and Bezout's algorithm*. Mathematics LibreTexts.
https://math.libretexts.org/Courses/Mount_Royal_University/MATH_2150%3A_Higher_Arithmetic/4%3A_Greatest_Common_Divisor_least_common_multiple_and_Euclidean_Algorithm/4.2%3A_Euclidean_algorithm_and_Bezout's_algorithm