Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Matemáticas e Informática

Trabajo Fin de Grado

Desarrollo de una Aplicación Didáctica: Aritmética Entera y Modular

Autor: Ilay Galron Chocron
Tutor: Alfonso Zamora Saiz

Madrid, Mayo 2023

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Matemáticas e Informática

Título: Desarrollo de una Aplicación Didáctica: Aritmética Entera y Modular

Mayo 2023

Autor: Ilay Galron Chocron
Tutor: Alfonso Zamora Saiz
Departamento de Matemáticas Aplicadas a las Tecnologías de la Información y las Comunicaciones
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Resumen

En este trabajo de fin de grado se va a explicar todo el proceso que se ha seguido para desarrollar una interfaz que resuelve problemas relacionados con la matemática discreta. Para su completo entendimiento, este trabajo guiará al lector por los diferentes pasos que se han seguido, comenzando con la teoría que se envuelve alrededor de la aritmética entera y modular, en el que se han descrito los temas más relevantes de cada área. Para ello, cada tema constará de una definición matemática seguida de algunos ejemplos para que el lector lo pueda comprender.

Basándose en esta teoría descrita en la primera parte del trabajo, se enseñará el código escrito en Python para resolver estos problemas de manera automática. Para cada uno de los temas desarrollados se explica el funcionamiento del código, así como la explicación de un código adicional que devuelve al usuario por consola los pasos que se han seguido a la hora de resolver el problema.

Estos pasos están explicados en el siguiente capítulo, que se ha dividido siguiendo el mismo formato que la teoría y el código. Por cada tema que se ha desarrollado se realizan ejemplos de una llamada a la función (pretendiendo querer resolver un problema). En cada ejemplo se indicará al lector lo que irá obteniendo teniendo en cuenta los pasos.

Cabe destacar que este TFG se enmarca dentro de la actividad del Grupo de Innovación Educativa 'Desarrollo de Tecnologías en la Enseñanza de las Matemáticas', formando parte de un proyecto que pretende construir una aplicación didáctica para resolver problemas de las diferentes asignaturas de matemáticas de la ETSIINF. Al igual que este TFG (que está centrado en la matemática discreta), se están desarrollando otros trabajos que desarrollan otras áreas de las matemáticas (como cálculo, topología, etc), que posteriormente se implementarán en una aplicación también desarrollada por otros alumnos que cursan la asignatura del Prácticum. El objetivo es que se pueda desarrollar una aplicación para ayudar a los alumnos a comprender y a resolver los problemas matemáticos que tengan.

Abstract

In this Final Degree Project, the whole process that has been followed to develop an interface that solves problems related to discrete mathematics will be explained. For its complete understanding, this project will guide the reader through the different steps that have been followed, starting with the theory that is wrapped around integer and modular arithmetic, in which the most relevant topics of each area have been described. To this end, each topic will consist of a mathematical definition followed by examples for the reader to understand.

Based on the theory described in the first part of the project, the code written in Python to solve these problems automatically will be taught. For each of the developed topics the operation of the code is explained, as well as the explanation of an additional code to return to the user by console the steps that have been followed at the time of solving the problem.

These steps are explained in the following chapter, which has been divided following the same format as the theory and the code. For each topic that has been developed, examples of a function call (pretending to want to solve a problem) are given. In each example the reader will be told the steps to obtain the solution.

It should be noted that this Final Degree Project is part of the activity of the Educational Innovation Group 'Development of Technologies in the Teaching of Mathematics', being part of a project that aims to build a didactic application to solve problems of the different mathematics subjects of the ETSIINF. As well as this Final Degree Project (which is focused on discrete mathematics), other projects are being developed on other areas of mathematics (such as calculus, topology, etc.), which will later be implemented in an application also developed by other students taking the 'Practicum' course. The objective is to develop an application to help students understand and solve mathematical problems.

Tabla de contenidos

Capítulo 1

Introducción

El propósito de este Trabajo de Fin de Grado es desarrollar una API (Interfaz de Programación de Aplicaciones) para resolver problemas relacionados con el área de las matemáticas discretas, especialmente con la aritmética entera y modular.

Este TFG se enmarca dentro de la actividad del Grupo de Innovación Educativa 'Desarrollo de Tecnologías en la Enseñanza de las Matemáticas' y forma parte de un proyecto en el cual se pretende construir una aplicación didáctica donde vengan implementados la mayoría de los algoritmos que aparecen en las asignaturas de matemáticas del Grado de Matemáticas e Informática (GMI) de la ETSIINF, con el fin de ayudar a los estudiantes en el aprendizaje.

Este grupo está formado por varios estudiantes del GMI divididos en dos grupos: los encargados de programar las APIs con los algoritmos necesarios en cada módulo y los encargados de implementar dichas APIs y crear la aplicación didáctica, que se está desarrollando en el momento de escribir este TFG. Cabe destacar que es el primer cuatrimestre en el que se está realizando este tipo de trabajos de fin de grado.

La matemática discreta es una disciplina de las matemáticas que se enfoca en estudiar los procesos matemáticos discretos (en oposición a los procesos continuos que estudia el análisis matemático o el cálculo). En concreto, la aritmética trata sobre las operaciones de suma y multiplicación y sus inversas en el anillo más representativo, que es los enteros, y en sus cocientes módulo un entero fijo, la aritmética modular.

Mi motivación para escogerlas como área para aportar en el proyecto viene dada por su sencillez pero a la vez por su importancia en el mundo de las matemáticas, siendo uno de sus pilares. La lógica y la combinatoria están basadas en ella, siendo la lógica la que aporta los principios del razonamiento, así como su solidez para demostrar teoremas, y la combinatoria, que permite profundizar en las diferentes maneras de aportar orden a grupos finitos de objetos. Ambos conceptos están presentes de alguna manera en mí, soy una persona que se rige por la lógica para resolver problemas diarios y una persona que se esfuerza por controlar las situaciones de manera ordenada, por lo que veo mayor razón para desarrollar este área.

Independientemente del módulo que tengo asignado, otra razón para realizar este TFG es el hecho de colaborar en la creación de una aplicación dirigida a los futuros alumnos de mi mismo grado, el Grado de Matemáticas e Informática, y a ser posible de otros grados, para ayudarles con todos los problemas que puedan tener resolviendo los mismos ejercicios que yo intenté resolver, añadiendo las explicaciones más detalladas posibles.

La aritmética entera y la aritmética modular son las dos principales partes de la matemática discreta que se han desarrollado en este TFG. Dentro de la parte de la aritmética entera se han implementado el algoritmo de Euclides para el cálculo del máximo común divisor, el cálculo de coeficientes del Teorema de Bezout, la divisibilidad de números enteros, la comprobación de números primos, la conversión de números en diferentes bases, la descomposición en factores primos de acuerdo con el Teorema Fundamental de la Aritmética y la resolución de ecuaciones diofánticas. Por otro lado, dentro de la aritmética modular se han implementado el cálculo en anillos de congruencias (unidades, divisores de cero e inversos) y la resolución de ecuaciones de congruencias y sistemas mediante el Teorema Chino del Resto.

A continuación se muestra un resumen del contenido que se desarrolla en cada capítulo.

1. Marco teórico: En él se incluyen todos los teoremas matemáticos y los algorítmos implementados en los módulos de la matemática discreta descritos previamente.
2. Código: Explicación del desarrollo de los módulos mediante el lenguaje de programación Python, haciendo énfasis en las funciones que se han programado.
3. Resultados: Funcionamiento general de la aplicación, explicando con ejemplos lo que devuelven las llamadas a las distintas funciones.
4. Impacto: Explicación de los objetivos de desarrollo sostenible de la Agenda 2030 que tengan relación con éste TFG

Capítulo 2

Marco Teórico

En este capítulo se explica la teoría en la que se basa el código Python que se ha desarrollado. Se ha dividido en dos grandes bloques: la aritmética entera, que incluye desde el algoritmo de Euclides para el cálculo del máximo común divisor hasta los sistemas de numeración en diferentes bases, y la aritmética modular, incluyendo el cálculo de unidades e inversos, divisores de cero y ecuaciones y sistemas lineales de congruencias. En cada apartado del marco teórico se incluyen unos ejemplos que se asocian a la teoría descrita.

2.1. Aritmética Entera

En este apartado se van a describir las principales operaciones y algoritmos que se incluyen dentro de la aritmética entera, es decir, la rama de las matemáticas discretas encargada del estudio y manipulación del anillo $\mathbb{Z}$ de los enteros mediante las operaciones de suma y multiplicación y sus operaciones inversas (la resta y la división).

Se va a definir el concepto de máximo común divisor y su cálculo mediante el algoritmo de Euclides, el teorema de Bezout, los criterios de divisibilidad, la resolución de ecuaciones diofánticas, el teorema fundamental de la aritmética, los números primos y los sistemas de numeración en base b.

2.1.1. Máximo Común Divisor

Sean dos números enteros no nulos a y b. Se dice que $d > 0$ es el **máximo común divisor** de a y b si cumple con las siguientes condiciones:

$$\text{(i) } d \mid a \text{ y } d \mid b \qquad \text{(ii) } c \mid a \text{ y } c \mid b \implies c \mid d$$

El máximo común divisor es único y se representa por $\mathrm{mcd}(a, b)$. Cuando $d = 1$ decimos que los números a y b son **primos entre sí**.

Algorítmo de Euclides para el cálculo del máximo común divisor

Si dividimos a y b y denotamos el cociente de la división como q y el resto como r podemos afirmar que $a = bq + r$. Por lo tanto cualquier divisor de a y b también va a ser divisor de r y cualquier divisor de b y r también lo va a ser de a. Es decir, tenemos que $\mathrm{mcd}(a, b) = \mathrm{mcd}(b, r)$, como se muestra en el siguiente algoritmo:

$$\begin{cases} a = bq_1 + r_1 & \implies \mathrm{mcd}(a,b) = \mathrm{mcd}(b, r_1) \\ b = r_1 q_2 + r_2 & \implies \mathrm{mcd}(b,r_1) = \mathrm{mcd}(r_1, r_2) \\ r_1 = r_2 q_3 + r_3 & \implies \mathrm{mcd}(r_1,r_2) = \mathrm{mcd}(r_2, r_2) \qquad \implies \mathrm{mcd}(a,b) = r_k \\ \dots \quad \implies \dots & \\ r_{k-1} = r_k q_{k+1} + 0 & \implies \mathrm{mcd}(r_{k-1}, r_k) = \mathrm{mcd}(r_k, 0) \end{cases}$$

De esta forma, el máximo común divisor de a y b es el último resto que no es nulo. Para el algoritmo podemos usar el siguiente esquema:

	q_1	q_2	q_3	$\dots$	$\dots$	q_k	q_{k+1}
a	b	r_1	r_2	$\dots$	$\dots$	r_{k-1}	r_k
r_1	r_2	r_3	$\dots$	$\dots$	r_k	0	

$\implies \mathrm{mcd}(a,b) = r_k$

[1]

Ejemplo 1: Hallar el máximo común divisor de 18 y 122.

	6	1	3	2
122	18	14	4	2
14	4	2	0	

$\implies \mathrm{mcd}(18, 122) = 2$

Ejemplo 2: Hallar el máximo común divisor de 35787 y 38.

	941	1	3	4	2
35787	38	29	9	2	1
29	9	2	1	0	

$\implies \mathrm{mcd}(35787, 38) = 1$

2.1.2. Teorema de Bezout

Sean dos números enteros a y b, se cumple que $\mathrm{mcd}(a, b) = 1$ si y sólo si existen números enteros m y n tales que $am + bn = 1$. Los números m y n no son únicos, como podemos ver en el siguiente ejemplo:

$$1 = \mathbf{4} \cdot 2 + (\mathbf{-1}) \cdot 7 = (\mathbf{-3}) \cdot 2 + 1 \cdot 7 = (\mathbf{-10}) \cdot 2 + \mathbf{3} \cdot 7 = \mathbf{25} \cdot 2 + (\mathbf{-7}) \cdot 7 = ...$$

En el caso que $\mathrm{mcd}(a, b) = d > 1$, se va a cumplir de la misma forma que existen m y n enteros que verifican la igualdad $am + bn = d$, justificándose de la siguiente forma:

$$\mathrm{mcd}(a,b) = d > 1 \implies \mathrm{mcd}(\tfrac{a}{d}, \tfrac{b}{d}) = 1 \implies \exists m, n \in \mathbb{Z} : \tfrac{a}{d}m + \tfrac{b}{d}n = 1 \implies am + bn = d$$

Algorítmo para calcular los coeficientes del teorema de Bezout

El siguiente algoritmo calcula los coeficientes del teorema de Bezout y se basa en el algoritmo de Euclides que proporciona el máximo común divisor. Sean a y b con máximo común divisor $d = \mathrm{mcd}(a, b)$. En cada iteración del algoritmo vamos a

calcular el cociente y el resto de la división entre a y b y vamos a calcular una variable compuesta por dos valores. Inicialmente tenemos las variables $\mathbf{v}_0 = (1, 0)$ y $\mathbf{v}_1 = (0, 1)$. En cada iteración restamos la primera variable menos la segunda multiplicada por el cociente obtenido en la iteración anterior. Se repetirá el proceso hasta que el resto sea 0.

[1]

Ejemplo 1: Sean $a = 122$ y $b = 18$ con $d = \text{mcd}(a, b) = 2$, calcular los coeficientes de Bezout.

1312		$\mathbf{v}_0$ = (1, 0)
800	1	$\mathbf{v}_1$ = (0, 1)
512	1	$\mathbf{v}_2$ = (1, 0) - 1(0, 1) = (1, −1)
288	1	$\mathbf{v}_3$ = (0, 1) - 1(1, −1) = (−1, 2)
224	1	$\mathbf{v}_4$ = (1, −1) - 1(−1, 2) = (2, −3)
64	3	$\mathbf{v}_5$ = (−1, 2) - 1(2, −3) = (−3, 5)
32	2	$\mathbf{v}_6$ = (2, −3) - 3(−3, 5) = **(**11**,** −18**)**
0		

$$\Longrightarrow 1312 \cdot \mathbf{11} + 800 \cdot (\mathbf{\text{-}18}) = 32$$

2.1.3. Criterios de divisibilidad

Exponemos en esta sección los criterios de divisibilidad por los enteros entre 2 y 10.

Criterio del 2: Un número es divisible entre 2 si termina en 0, 2, 4, 6 u 8.

Criterio del 3: Un número es divisible entre 3 si la suma de sus dígitos es un múltiplo de 3.

Criterio del 4: Un número es divisible entre 4 si sus últimos dos dígitos son 00 o un múltiplo de 4.

Criterio del 5: Un número es divisible entre 5 si termina en 0 o en 5.

Criterio del 6: Un número es divisible entre 6 si cumple con los criterios de divisibilidad del 2 y del 3.

Criterio del 7: Para saber si un número es divisible entre 7 se realiza lo siguiente: se multiplica el último dígito por 2 y se resta al número que conforman los demás dígitos. Esto se realiza de manera iterativa hasta que quede un único dígito. Si ese dígito es un 0 o un 7 entonces el número es divisible entre 7.

Criterio del 8: Un número es divisible entre 8 si los últimos tres dígitos son múltiplo e 8 o iguales a 0.

Criterio del 9: Un número es divisible entre 9 si la suma de sus dígitos es múltiplo de 9.

Criterio del 10: Un número es divisible entre 10 si termina en 0.

[2]

Ejemplo 1: Comprobar la divisibilidad del número 378.

1. Es divisible entre 2 ya que es un número par.
2. Es divisible entre 3 ya que la suma de sus dígitos en un múltiplo de 3: $3 + 7 + 8 = 18 = 3 \cdot 6$.
3. Es divisible entre 6 ya que es divisible entre 2 y 3.
4. Es divisible entre 7 y se comprueba de la siguiente forma: Se toma el último dígito y se multiplica por 2: $2 \cdot 8 = 16$, a continuación se resta al número que conforman el resto de dígitos: $37 - 16 = 21$, y se vuelve a repetir el proceso: $2 \cdot 1 = 2$ y $2 - 2 = 0$. Como el último número es un cero, entonces el 378 es divisible entre 7.
5. Es divisible entre 9 porque la suma de sus dígitos es divisible entre 9: $3 + 7 + 8 = 18 = 9 \cdot 2$).

2.1.4. Ecuaciones Diofánticas

Una **ecuación diofántica lineal** es cualquier ecuación de la forma $ax + by = c$ donde a, b y c son números enteros. Para que podamos obtener sus soluciones enteras se debe verificar que $\text{mcd}(a, b) = d \mid c$. Una vez se cumple, se puede proceder a encontrar sus soluciones mediante el siguiente algoritmo:

1. Aplicamos el teorema de Bezout con a y b para encontrar m y n tales que $am + bn = d$.
2. Multiplicamos la ecuación por $\alpha = c/d$, obteniendo $\alpha am + \alpha bn = \alpha d$, por lo que podemos extraer las soluciones particulares $x_0 = \alpha m$ e $y_0 = \alpha n$.
3. Por lo tanto todas las soluciones de la ecuación diofántica lineal tienen la siguiente forma:

$$\begin{cases} x = x_0 + \frac{bt}{d} \\ y = y_0 - \frac{at}{d} \end{cases} \qquad \text{ó} \qquad \begin{cases} x = \frac{mc+bt}{d} \\ y = \frac{nc-at}{d} \end{cases}$$

[1]

Ejemplo 1: Estudiar la existencia de soluciones y resolver la ecuación diofántica lineal $36x + 9y = 10$.

No tiene solución, ya que $\text{mcd}(36, 9) = 9 \nmid 10$.

Ejemplo 2: Estudiar la existencia de soluciones y resolver la ecuación diofántica lineal $14x - 4y = 2$.

Dividiendo por 2, la ecuación es equivalente a $7x - 2y = 1$, que tiene solución ya que $\text{mcd}(7, 2) = 1 \mid 1$. Una solución particular sería $x_0 = 1$, $y_0 = 3$. Por lo tanto, todas las soluciones serían de la forma:

$$(x, y) = (1, 3) + \tfrac{(-2,-7)}{1}t = (1, 3) - (2, 7)t,\ t \in \mathbb{Z} \quad \text{ó} \quad \begin{cases} x = 1 - 2t \\ y = 3 - 7t \end{cases} \qquad t \in \mathbb{Z}$$

2.1.5. Números Primos

Sea un número natural $n > 1$. Se dice que n es un **número primo** si sus únicos divisores positivos son 1 y n. En caso contrario, se llama **número compuesto**.

[1]

Ejemplo 1: Comprobar si el número 324 es primo:

Como se ha visto en el apartado de la divisibilidad, el número 324 es divisible entre 2, por ser un número par, y entre 3, ya que la suma de sus dígitos es un múltiplo de 3 ($3 + 2 + 4 = 9$). Al tener divisores, podemos concluir que el número 324 es un número compuesto.

Ejemplo 2: Comprobar si el número 37 es primo:

El número 37 es un número primo, ya que no verifica ninguno de los criterios de divisibilidad y es sólo divisible entre 1 y él mismo.

2.1.6. Teorema Fundamental de la Aritmética

El teorema fundamental de la Aritmética afirma que cualquier número $n \in \mathbb{N}$ que sea mayor que 1 se puede expresar como producto de potencias de números primos:

$$n = p_1^{a_1} \cdot p_2^{a_2} \cdot \dots \cdot p_k^{a_k}$$

donde todos los a_i son números naturales y todos los p_i son números primos distintos. Esta expresión es una **factorización prima** o **descomposición en factores primos** del número n.

Para encontrar la descomposición se comienza dividiendo el número por el primo más pequeño, es decir, el 2. En caso de que sea divisible se añade el 2 a la lista de primos y se sigue con el cociente y, en caso contrario, se comprueba con los sucesivos primos hasta que sea divisible, repitiendo el mismo proceso hasta que el cociente sea 1. Solamente va a ser necesario realizar divisiones hasta $\sqrt{n}$.

[1]

Ejemplo 1: Descomponer en factores primos el número 324:

$$\begin{array}{r|l} 324 & 2 \\ 162 & 2 \\ 81 & 3 \\ 27 & 3 \\ 9 & 3 \\ 3 & 3 \\ 1 & \end{array} \implies 324 = 2^2 \cdot 3^4$$

Ejemplo 2: Descomponer en factores primos el número 37:

$$\begin{array}{r|l} 37 & 37 \\ 1 & \end{array} \implies \text{El número } 37 \text{ es un número primo.}$$

2.1.7. Sistemas de numeración en base b

Sea un número n representado por la concatenación de sus dígitos $d_1, d_2 \dots d_i$, con $i \in \mathbb{N}$, de derecha a izquierda y sea b una base. Podemos decir que n está representado en base b si $n = d_i \cdot b^{i-1} + d_{i-1} \cdot b^{i-2} \cdots d_2 \cdot b^1 + d_1 \cdot b^0$.

Para representar un número en diferentes bases se realiza el siguiente algoritmo:

1. Se divide n entre b.
2. Se recoge el resto de la división, representando el elemento más a la derecha del resultado.
3. Se vuelve a realizar los mismos pasos con los sucesivos cocientes, añadiendo el resto de las divisiones en el resultado de derecha a izquierda.
4. El algoritmo termina cuando el cociente de la división sea 0.

[1]

Ejemplo 1: Representar el número 35 en las bases 2, 8 y 12.

En base 2
$\frac{35}{2} = 17 \implies$ resto = 1 (posición más a la derecha)
$\frac{17}{2} = 8 \implies$ resto = 1
$\frac{8}{2} = 4 \implies$ resto = 0
$\frac{4}{2} = 2 \implies$ resto = 0
$\frac{2}{2} = 1 \implies$ resto = 0
$\frac{1}{2} = 0 \implies$ resto = 1 (posición más a la izquierda)
Por lo tanto $35 = 100011_2$

En base 8
$\frac{35}{8} = 4 \implies$ resto = 3 (posición más a la derecha)
$\frac{4}{8} = 0 \implies$ resto = 4 (posición más a la izquierda)
Por lo tanto $35 = 43_8$

En base 12 (donde representamos los restos por $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B$)
$\frac{35}{12} = 2 \implies$ resto = 11 (posición más a la derecha)
$\frac{2}{12} = 0 \implies$ resto = 2 (posición más a la izquierda)
Por lo tanto $35 = 2B_{12}$, con $B = 11$

2.2. Aritmética Modular

En este segundo apartado del marco teórico se van a definir los algoritmos más relevantes dentro de la aritmética modular, es decir, las operaciones de suma y multiplicación y sus inversos dentro del anillo $\mathbb{Z}_m$ de los cocientes de enteros módulo un número entero m fijo.

En concreto, se van a estudiar las unidades o elementos inversibles, los divisores de cero, la resolución de ecuaciones lineales de congruencias y la resolución de sistemas lineales de congruencias mediante el teorema chino del resto.

2.2.1. Unidades o elementos inversibles

Se dice que $\overline{a} \in \mathbb{Z}_m$ es un **elemento inversible** ó **unidad** si existe $\overline{b} \in \mathbb{Z}_m$ tal que $\overline{a} \cdot \overline{b} = \overline{1}$. Se denotará por $\overline{a}^{-1} := \overline{b}$. Denotaremos U_m al conjunto de todos los elementos inversibles de $\mathbb{Z}_m$.

[1]

Ejemplo 1: Hallar los elementos inversibles en $\mathbb{Z}_5$:

En $\mathbb{Z}_5$ todos sus elementos no nulos son inversibles:

$$\overline{1}^{-1} = \overline{1}, \overline{2}^{-1} = \overline{3}, \overline{3}^{-1} = \overline{2}, \overline{4}^{-1} = \overline{4}$$

2.2.2. Divisores de cero

Sean $\overline{a}, \overline{b} \in \mathbb{Z}_m$. Diremos que son **divisores de cero** si cumplen que $\overline{a} \neq \overline{0} \neq \overline{b}$ y que $\overline{a} \cdot \overline{b} = \overline{0}$. Podemos observar que:

$$\text{Existen divisores de cero en } \mathbb{Z}_m \iff m \text{ no es primo}$$

[1]

Ejemplo 2: Hallar los divisores de cero en $\mathbb{Z}_8$:

Puesto que 8 no es primo, $\mathbb{Z}_8$ tiene divisores de cero. Los únicos divisores de cero son los pares $(\overline{2}, \overline{4})$ y $(\overline{4}, \overline{6})$ puesto que:

$$\overline{2} \cdot \overline{4} = \overline{0} \text{ en } \mathbb{Z}_8 \quad \text{y} \quad \overline{4} \cdot \overline{6} = \overline{0} \text{ en } \mathbb{Z}_8$$

2.2.3. Ecuaciones lineales de congruencias

Sean a y $b \in \mathbb{Z}$ y $m > 1$. Se llama **ecuación lineal de congruencias** a cualquier ecuación de la forma $ax \equiv b \pmod{m}$ y se llama **solución** a los valores $x \in \mathbb{Z}$ que satisfacen la ecuación.

Para que $ax \equiv b \pmod{m}$ tenga solución, es necesario y suficiente que la ecuación $ax + my = b$ tenga solución en x e y, lo cual se cumple si y sólo si $\text{mcd}(a, m) \mid b$. Es decir,

$$ax \equiv b \pmod{m} \text{ tiene solución} \iff \text{mcd}(a, m) \mid b$$

Tenemos entonces dos casos:

1. Si $\text{mcd}(a, b) = 1$, entonces:

   $ax \equiv b \pmod{m} \iff \overline{ax} = \overline{b}$ en $\mathbb{Z}_m \iff \overline{x} = \overline{a}^{-1}\overline{b}$ en $\mathbb{Z}_m$ y la solución es única.

2. Si $\text{mcd}(a, b) = d > 1$, con $d \mid b$, entonces:

   $$ax \equiv b \pmod{p} \iff (a/d)x \equiv b/d \pmod{m/d} \iff (\overline{a/d})\ \overline{x} = \overline{b/d} \text{ en } \mathbb{Z}_{m/d}$$

   con $\text{mcd}(a/d, m/d) = 1$, por lo que existe inverso de $\overline{a/d}$ módulo $\overline{m/d}$. Se puede obtener la solución en $\mathbb{Z}_{m/d}$ multiplicando por el inverso:

   $$\overline{x} = (\overline{a/d})^{-1}\overline{b/d} \text{ en } \mathbb{Z}_{m/d}$$

La solución es única en $\mathbb{Z}_{m/d}$, y existen d soluciones en $\mathbb{Z}_m$:

$$\overline{x} = (\overline{a/d})^{-1}\overline{b/d} + t\overline{m/d} \text{ en } \mathbb{Z}_m,\ t = 0, 1, 2, \ldots, d-1$$

[1]

Ejemplo 1: Hallar las soluciones enteras de $2x + 3 \equiv 8 \pmod 9$.

Si pasamos el 3 al segundo miembro, obtenemos la ecuación $2x \equiv 5 \pmod 9$. Como $\mathrm{mcd}(2, 9) = 1$, la ecuación tiene solución:

$$2x \equiv 5 \pmod 9 \iff \overline{2}\overline{x} = \overline{5} \text{ en } \mathbb{Z}_9 \iff \overline{x} = \overline{2}^{-1}\overline{5} = \overline{5}\cdot\overline{5} = \overline{25} = \overline{7} \text{ en } \mathbb{Z}_9$$

ya que $\overline{2}^{-1} = \overline{5}$ en $\mathbb{Z}_9$. Por lo tanto la solución a la ecuación sería:

$$\text{En } \mathbb{Z}_9\text{: } \overline{x} = \overline{7} \qquad \text{En } \mathbb{Z}\text{: } x = 7 + 9t,\ t \in \mathbb{Z}$$

Ejemplo 2: Hallar las soluciones enteras de $5x + 2 \equiv 3 \pmod{10}$.

Pasando el 2 al otro miembro obtenemos $5x \equiv 1 \pmod{10}$, que no tiene solución ya que $\mathrm{mcd}(5, 10) = 5 \nmid 1$.

2.2.4. Sistemas lineales de congruencias y Teorema Chino del Resto

Sea una sistema de ecuaciones lineales de congruencias de la siguiente forma:

$$\begin{cases} c_1x \equiv a_1 \pmod{m_1} \\ c_2x \equiv a_2 \pmod{m_2} \\ \quad\vdots \\ c_ix \equiv a_i \pmod{m_i} \end{cases}$$

Si c_i, a_i, $m_i \in \mathbb{Z}$ con $m_i > 1$ y $\mathrm{mcd}(m_i, m_j) = 1$ para $i \neq j$ (es decir, que son primos entre sí dos a dos), entonces el sistema de congruencias tiene una única solución en $\mathbb{Z}_m$ con $m = m_1 \cdot m_2 \ldots m_n$. Sea $\overline{b_i} = a_i \cdot \overline{c_i}^{-1}$ en $\mathbb{Z}_{m_i}$, la solución es la siguiente:

$$x \equiv (\overline{b_1}q_1\tfrac{m}{m_1} + \overline{b_2}q_2\tfrac{m}{m_2} + \cdots + \overline{b_n}q_n\tfrac{m}{m_n}) \pmod m$$

donde q_i es tal que $q_i\frac{m}{m_i} \equiv 1 \pmod{m_i}$, $1 \leq i \leq n$. Es decir:

$$q_i = \overline{m/m_i}^{-1} \text{ en } \mathbb{Z}_{m_i}$$

[1]

Ejemplo 1: Resolver el siguiente sistema: $\begin{cases} x \equiv 1 \pmod 2 \\ x \equiv 2 \pmod 3 \\ x \equiv 5 \pmod 7 \end{cases}$

Se puede aplicar el Teorema chino del resto al sistema ya que 2, 3 y 7 son primos entre sí y tiene solución única en $\mathbb{Z}_m$ con $m = 2 \cdot 3 \cdot 7 = 42$. Dicha solución es:

$$x \equiv (1 \cdot q_1 \cdot \tfrac{42}{2} + 2 \cdot q_2 \cdot \tfrac{42}{3} + 5 \cdot q_3 \cdot \tfrac{42}{7}) = (1 \cdot q_1 \cdot 21 + 2 \cdot q_2 \cdot 14 + 5 \cdot q_3 \cdot 6) \pmod{42}$$

donde:

$$\begin{cases} q_1 \cdot 21 \equiv 1 (\text{mod } 2) \implies q_1 = 1 \\ q_2 \cdot 14 \equiv 1 (\text{mod } 3) \implies 2q_2 \equiv 1 (\text{mod } 3) \equiv q_2 = 2 \\ q_3 \cdot 6 \equiv 1 (\text{mod } 7) \implies q_3 = 6 \end{cases}$$

Por lo tanto la solución es:

$$x \equiv (1 \cdot 1 \cdot 21 + 2 \cdot 2 \cdot 14 + 5 \cdot 6 \cdot 6) = (21 + 56 + 180) = 257 \equiv 5 \text{ (mod } 42)$$

Capítulo 3

Código

En este capítulo se incluye todo el código desarrollado para resolver los problemas más comunes de la matemática discreta relacionados con la aritmética entera y modular. Dicho código está escrito en el lenguaje Python y consta de dos grandes ficheros: *discreta_I.py* y *getPasos.py*. El primero de ellos contiene todo lo relacionado con la resolución del problema, planteando los algoritmos descritos en el capítulo del marco teórico. A lo largo de su resolución, estos algoritmos van recogiendo los valores y resultados más importantes del proceso para pasarlos posteriormente a las funciones de *getPasos.py*. Las funciones de este fichero toman como parámetro los resultados obtenidos y construyen un JSON de respuesta con los pasos para poder enviarlo a la aplicación didáctica.

EL formato del JSON con la respuesta es similar para todas las funciones. Contiene un valor llamado *"pasos"* asociado a un array de pasos. Dicho array va a estar formado por otros JSON con una *"descripción"* y un *"pasoLatex"* cuando sea necesario. La *"descripcion"* contiene una explicación de cada paso y *"pasoLatex"* contiene el paso escrito en formato matemático. Cabe destacar que no todas las respuestas van a tener *"pasoLatex"*. Además del array de pasos, hay algunas funciones que devuelven otros valores que son usados para algunas funciones pero que no se van a mostrar en la aplicación didáctica (como por ejemplo el teorema de Bezout devuelve los valores *"m"* y *"n"* que serán usados para la resolución de ecuaciones diofánticas). Una representación del formato es la siguiente:

```
{
    "pasos":
    [
        {
            "pasoLatex": str,
            "descripcion": str
        }
    ]
}
```

3.1. Código principal

3.1.1. Máximo común divisor

Algoritmo

```
def mcd(a, b):
    """
    Comprobación previa para el cálculo del Máximo Común Divisor de dos números o polinomios.
    Lanza una Exception si no cumplen los requisitos o llama a la función
    mcd_aux() para su cálculo. Toma como parámetros dos arrays que corresponden con los coeficientes
    de un polinomio, siendo el primer valor el coeficiente con mayor exponente. En caso de querer calcular
    el mcd de dos número enteros se pasarían dos listas con un elemento cado uno.

    Retorna como resultado el siguiente JSON:

    {
        "pasos": [
            {
                "pasoLatex": str,
                "descripcion": str
            }
            {
            ...
            }
        ],
        "resultado": str
    }
    """
    # Los parámetros no pueden estar vacíos
    if (a == []) or (b == []):
        raise MyException("Los elementos no pueden ser vacíos")
    # Los parámetros no pueden ser nulos
    elif (a == [0]) or (b == [0]):
        raise MyException("Los elementos no pueden ser nulos")
    # Los parámetros deben ser enteros
    elif (all(isinstance(x, int) for x in a) is False) or (all(isinstance(x, int) for x in b) is False):
        raise MyException("Los elementos deben ser enteros")
    elif (a[0] == 0) or (b[0] == 0):
        raise MyException("El primer coeficiente de los polinomios debe ser un entero distinto de cero")
    else:
```

```
36          resultado = mcd_aux(a, b, [(a, b)])
37          return pasos.mcd(resultado)
38
39
40  def mcd_aux(a, b, array):
41      """
42      Función auxiliar que calcula el máximo común divisor de dos
    ↪   números o polinomios.
43      """
44      while (b != []) and (b[0] != 0):
45          a, b = b, [a[0] % b[0]] if (len(a) == 1 and len(b) == 1)
            ↪  else dividir_polinomios(a, b)[1]
46          array.append((a, b))
47      return array
```

El algoritmo que se presenta para calcular el máximo común divisor de dos numeros o polinomios está formado por dos funciones: *mcd* y *mcd_aux*. La función *mcd* recoge los parámetros a y b, ambos listas que contienen los coeficientes de un polinomio (en caso de usar la función con dos enteros, los arrays tendrán solamente un elemento). La tarea de dicha función es realizar varias comprobaciones sobre los parámetros introducidos, es decir, que no sean vacíos, que los dos sean distinto de 0 y que todos los coeficientes sean enteros.

Una vez se han pasado todas las comprobaciones, se llama a la función *mcd_aux* con los parámetros a, b y un array con la tupla (a, b) que se usará para ir guardando todas las iteraciones del cálculo de las sucesivos divisores. Esta función itera sobre la variable b hasta que esta sea 0 o vacía, asignando a la variable a la anterior variable b y a la nueva variable b, el resultado de realizar 'a mod b'.

La función *mcd_aux* devuelve una lista que se usa posteriormente para obtener los pasos del algoritmo.

Función para devolver los pasos

```
1   def mcd(resultado):
2       json = {}
3       pasos = []
4       resFinal = ""
5
6       for i in range(len(resultado) - 1):
7           jsonAux = {}
8           a = Expression(resultado[i][0]).toExpression()
9           if a == "":
10              a = "0"
11          b = Expression(resultado[i][1]).toExpression()
12          if b == "":
13              b = "0"
14          if i+1 < len(resultado) and resultado[i+1] != None:
15              r = Expression(resultado[i+1][1]).toExpression()
```

```
16          if r == "":
17              r = "0"
18          jsonAux["pasoLatex"] = "\mathrm{mcd}(" + a + ", " + b +
            ↪  ") = \mathrm{mcd}(" + b + ", " + r + ")"
19          jsonAux["descripcion"] = "El mcd de 'a' y 'b' es igual
            ↪  al mcd de 'b' y el resto entre 'a' y 'b'"
20          pasos.append(jsonAux)
21          resFinal = b
22      if len(resultado[0][0]) == 1 and len(resultado[0][1]) == 1
        ↪  and int(resFinal) < 0:
23          pasos.append({"pasoLatex": "\mathrm{mcd}(" +
            ↪  Expression(resultado[0][0]).toExpression() + ", " +
            ↪  Expression(resultado[0][1]).toExpression() + ") = "
            ↪  + resFinal + " = " + str(abs(int(resFinal))),
24                        "descripcion": "Por definición, el mcd de
                           ↪  dos números es positivo por lo que la
                           ↪  solución será el valor absoluto"})
25          resFinal = str(abs(int(resFinal)))
26      pasos.append({"pasoLatex": "\mathrm{mcd}(" +
        ↪  Expression(resultado[0][0]).toExpression() + ", " +
        ↪  Expression(resultado[0][1]).toExpression() + ") = " +
        ↪  resFinal,
27                    "descripcion": "Resultado del máximo común
                       ↪  divisor"})
28      json["pasos"] = pasos
29      json["resultado"] = resFinal
30
31      return json
```

La función *mcd* toma como parámetro la variable *resultado*, un array que contiene los números a y b iniciales y una lista de tuplas que contiene los sucesivos cálculos del algoritmo de Euclides (en cada tupla tenemos el divisor y el resto de la tupla anterior).

Por cada iteración del parámetro *resultado*, se construye una cadena de texto de la forma $mcd(a, b) = mcd(b, r)$ y se añade a un JSON auxiliar, junto con una descripción. Finalmente se añade una variable más al JSON llamada *resultado"* en el que se incluye sólamente el resultado final, ya que va a ser utilizado por otras funciones como por ejemplo para el cálculo de los coeficientes de Bezout.

Cabe destacar que todas las variables se representan usando la clase *Expression*, que construye una expresión matemática tomando una lista de coeficientes.

3.1.2. Teorema de Bezout

Algoritmo

```
1  def bezout(a, b):
2      """
```

```
3       Función que comprueba si se puede aplicar el teorema de
    ↪   Bezout para calcular los coeficientes
4       dando como parámetros dos arrays que corresponden con los
    ↪   coeficientes de un polinomio,
5       siendo las variables que representan ax + by = d, con d =
    ↪   mcd(a, b). En caso de cumplir las
6       condiciones llama a la función 'bezout_enteros()' si se
    ↪   pasan enteros como parámetros o a
7       la función 'bezout_polinomios' si se pasan polinomios.
8
9       Retorna como resultado el siguiente JSON:
10
11      {
12          "pasos": [
13              {
14                  "pasoLatex": str,
15                  "descripcion": str
16              }
17              {
18              ...
19              }
20          ],
21          "m": str,
22          "n": str
23      }
24      """
25      # Los parámetros no pueden ser vacíos
26      if (a == []) or (b == []):
27          raise MyException("Los elementos no pueden ser vacíos")
28      # Los parámetros no pueden ser nulos
29      elif (a == [0]) or (b == [0]):
30          raise MyException("Los elementos deben ser no nulos")
31      # Los parámetros deben ser enteros
32      elif (all(isinstance(x, int) for x in a) is False) or
        ↪  (all(isinstance(x, int) for x in b) is False):
33          raise MyException("Los elementos deben ser enteros")
34      elif (a[0] == 0) or (b[0] == 0):
35          raise MyException("El primer coeficiente de los
            ↪  polinomios debe ser un entero distinto de cero")
36      else:
37          if (len(a) > 1 or len(b) > 1):
38              resultado = bezout_polinomios(a, b)
39              return pasos.bezout_polinomios(resultado)
40          elif (len(a) == 1 and len(b) == 1):
41              resultado = bezout_enteros(a[0], b[0])
42              return pasos.bezout_enteros(resultado)
43
```

```
44 # Función auxiliar para calcular los coeficientes del teorema de
    ↪ Bezout para enteros después de comprobar que a y b son
    ↪ correctos
45 def bezout_enteros(a, b):
46     """
47     Función auxiliar que realiza el algoritmo para obtener los
    ↪ coeficientes
48     del teorema de Bezout para enteros.
49     """
50     # Inicializamos las variables
51     v0 = (1, 0)
52     v1 = (0, 1)
53     q = []
54     r = a % b
55     resultado = [(a, q, v0), (b, a // b, v1)]
56 
57     # Realizamos el segundo algorítmo para el cálculo de los
       ↪ coeficientes
58     while r != 0:
59         q = a // b
60         r = a % b
61         # Vk+1 = Vk-1 - Qk*Vk
62         qv = (q * v1[0], q * v1[1])
63         v = (v0[0] - qv[0], v0[1] - qv[1])
64         a, b = b, r
65         v0, v1 = v1, v
66         if (r != 0):
67             resultado.append((b, a // b, v))
68     return resultado
69 
70 
71 # Función auxiliar para calcular los coeficientes del teorema de
    ↪ Bezout para polinomios después de comprobar que a y b son
    ↪ correctos
72 def bezout_polinomios(a, b):
73     """
74     Función auxiliar que realiza el algoritmo para obtener los
    ↪ coeficientes
75     del teorema de Bezout para polinomios.
76     """
77     # Inicializamos las variables
78     v0 = ([1], [0])
79     v1 = ([0], [1])
80     q = []
81     r = dividir_polinomios(a, b)[1]
82     resultado = [(a, q, v0), (b, dividir_polinomios(a, b)[0],
       ↪ v1)]
```

```
83
84      # Realizamos el segundo algorítmo para el cálculo de los
        ↪  coeficientes
85      while r[0] != 0:
86          q = dividir_polinomios(a, b)[0]
87          r = dividir_polinomios(a, b)[1]
88          # Vk+1 = Vk-1 - Qk*Vk
89          qv = (multiplicar_polinomios(q, v1[0]),
            ↪  multiplicar_polinomios(q, v1[1]))
90          v = (restar_polinomios(v0[0], qv[0]),
            ↪  restar_polinomios(v0[1], qv[1]))
91          a, b = b, r
92          v0, v1 = v1, v
93          if r[0] != 0:
94              resultado.append((b, dividir_polinomios(a, b)[0],
                ↪  v))
95      return resultado
```

El algoritmo definido para calcular los coeficientes del teorema de Bezout está formado por una función llamada *bezout* que toma como parámetros a y b, dos arrays de enteros (que corresponden a los coeficientes del polinomio). La tarea principal de esta función es comprobar que se cumplen todas las condiciones para calcular m y n tales que $am + bn = d$ con $d = \text{mcd}(a, b)$. Dichas condiciones son que los parámetros no estén vacíos, que no sean nulos y que todos los coeficientes sean enteros. Una vez comprobados, si los parámetros a y b tienen solamente un coeficiente (números enteros) se llamará a la función *bezout_enteros* y en el caso de que sean polinomios se llamará a *bezout_polinomios*.

Tanto *bezout_enteros* como *bezout_polinomios* utilizan el mismo algoritmo para calcular los coeficientes del teorema de Bezout (el algoritmo descrito en el marco teórico), por lo que solamente se va a explicar el funcionamiento de uno de ellos (el de *bezout_enteros*). La única diferencia entre las funciones es a la hora de realizar operaciones de división, multiplicación y resta de elementos. Cuando estamos trabajando con los polinomios es necesario llamar a unas funciones auxiliares de operaciones de polinomios, que se explicarán posteriormente.

En cuanto al algoritmo, comenzamos inicializando una serie de variables, donde una de ellas es r que guarda el resultado del resto de la división entre a y b. A continuación se itera sobre r hasta que sea igual a cero. En cada iteración guardamos en la variable q el cociente de a entre b y se realiza la siguiente operación: $V_{k+1} = V_{k-1} - Q_k \cdot V_k$ siendo inicialmente $V_0 = (1, 0)$ y $V_1 = (0, 1)$ (como se describe en el marco teórico). Una vez realizada la operación asignamos la nueva a a la antigua b y la nueva b a la antigua r. Por otra parte asignamos $v0$ a la antigua $v1$ y $v1$ a v.

Al final de cada iteración añadimos el resultado de ese paso a un array, para posteriormente devolver el objeto JSON con todos los pasos del algoritmo.

Función para devolver los pasos

```
def bezout_enteros(resultado):
    json = {}
    pasos = []
    mcd = str(resultado[len(resultado)-1][0])
    tabla = "\\begin{array}{l|l|l} "

    for i in range(len(resultado)):
        paso = ""
        # Añadimos la primera columna (a, b, r)
        paso += str(resultado[i][0]) + " & "
        # Añadimos la segunda columna (q)
        if resultado[i][1] == []:
            paso += "\\text{ } & "
        else:
            paso += str(resultado[i][1]) + " & "
        # Añadimos la tercera columna (v)
        v = (str(resultado[i][2][0]), str(resultado[i][2][1]))
        paso += "\\textbf{v}_" + str(i) + " = " + str(v) +
        ↪  "\\\\"
        # Añadimos todos los pasos al array
        tabla += paso + "\\hline \\\\"
    tabla += "0 & \\text{ } & \\text{ } "
    tabla += "\\end{array}"

    pasos.append({"pasoLatex": tabla,
                  "descripcion": "Tabla con el algoritmo para
                  ↪  calcular los coeficientes de Bezout"})

    m = str(resultado[len(resultado)-1][2][0])
    n = str(resultado[len(resultado)-1][2][1])

    if resultado[len(resultado)-1][0] < 0:
        m = str(int(m) * (-1))
        n = str(int(n) * (-1))
        mcd = str(abs(resultado[len(resultado)-1][0]))
        pasos.append({"pasoLatex": "(" + str(resultado[0][0]) +
        ↪  ") \\cdot (" + m + ") + (" + str(resultado[1][0]) +
        ↪  ") \\cdot (" + n + ") = " + mcd,
                      "descripcion": "Puesto que según la
                      ↪  definición el mcd es positivo,
                      ↪  multiplicamos la solución por -1"})

    pasos.append({"pasoLatex": "(" + str(resultado[0][0]) + ")
    ↪  \\cdot (" + m + ") + (" + str(resultado[1][0]) + ")
    ↪  \\cdot (" + n + ") = " + mcd, \
```

```
38                  "descripcion": "Solución final del teorema de
                     ↪  Bezout"})
39
40      json["pasos"] = pasos
41      json["m"] = m
42      json["n"] = n
43      return json
44
45
46  def bezout_polinomios(resultado):
47      json = {}
48      pasos = []
49      tabla = "\\begin{array}{l|l|l} "
50
51      for i in range(len(resultado)):
52          paso = ""
53          # Añadimos la primera columna (a, b, r)
54          paso += Expression(resultado[i][0]).toExpression() + " &
                     ↪  "
55          # Añadimos la segunda columna (q)
56          if resultado[i][1] == []:
57              paso += "\\text{ } & "
58          else:
59              paso += Expression(resultado[i][1]).toExpression() +
                     ↪  " & "
60          # Añadimos la tercera columna (v)
61          v = (Expression(resultado[i][2][0]).toExpression(),
                     ↪  Expression(resultado[i][2][1]).toExpression())
62          paso += "\\textbf{v}_" + str(i) + " = " + str(v) +
                     ↪  "\\\\"
63          # Añadimos todos los pasos al array
64          tabla += paso + "\\hline \\\\"
65      tabla += "0 & \\text{ } & \\text{ } "
66      tabla += "\\end{array}"
67
68      pasos.append({"pasoLatex": tabla,
69                    "descripcion": "Tabla con el algoritmo para
                       ↪  calcular los coeficientes de Bezout"})
70
71      m =
         ↪  Expression(resultado[len(resultado)-1][2][0]).toExpression()
72      n =
         ↪  Expression(resultado[len(resultado)-1][2][1]).toExpression()
73
74      pasos.append({"pasoLatex": "(" +
         ↪  Expression(resultado[0][0]).toExpression() + ") \\cdot
         ↪  (" + m + ") + ("\
```

```
75      + Expression(resultado[1][0]).toExpression() + ") \\cdot ("
        ↪  + n + ") = "\
76      + Expression(resultado[len(resultado)-1][0]).toExpression(),
        ↪  \
77              "descripcion": "Solución del teorema de Bezout"})
78
79      json["pasos"] = pasos
80      json["m"] = m
81      json["n"] = n
82      return json
```

Como se ha mencionado anteriormente, para la resolución del teorema de Bezout se han creado dos funciones: *bezout_enteros* y *bezout_polinomios*. Dependiendo del número de coeficientes que tengan los parámetros iniciales se llamará a una función u otra. Ambas funciones realizan los mismos pasos y la única diferencia es que tratando con polinomios se usarán funciones auxiliares para operar con ellos. Por eso se va a explicar el funcionamiento de *bezout_enteros*, ya que se aplica también a *bezout_polinomios*.

La función *bezout_enteros* toma como parámetro un array llamado *resultado* que contiene los elementos de las columnas de la tabla descrita en el marco teórico para su resolución. Por ello lo primero que hace el código es iterar sobre ese array recogiendo cada elemento y concatenándola a una cadena de texto que construye una tabla en formato latex. Por cada iteración del bucle se crea un paso, por lo que se repetirá tantas veces como filas tenga la tabla.

En el caso de que alguno de los valores iniciales sea negativo, se ha añadido un paso para multiplicar la ecuación por -1 de modo que se tiene el máximo común divisor positivo, de acuerdo con su definición.

A continuación se recogen los últimos valores de la tabla formada y se concatenan en una cadena de texto conteniendo la solución. Dicha cadena se añade posteriormente al array de pasos.

Finalmente se crean dos variables adicionales en el JSON conteniendo las soluciones m y n que se han obtenido de la ecuación. Estos pasos son necesarios ya que se van a usar en otro momento para la resolución de las ecuaciones diofánticas.

3.1.3. Divisibilidad

Algoritmo

```
1  def divisible(a, b):
2      """
3      Función para comprobar si un número es divisible por otro.
       ↪  Se dan como parámetros
4      dos números enteros. En el caso de que el divisor tenga un
       ↪  criterio de divisibilidad
```

```
    (2, 3, 4, 5, 6, 7, 8, 9 o 10), se dará una explicación de
↪   dicho criterio.

    Retorna como resultado el siguiente JSON:

    {
        "pasos": [
            {
                "descripcion": str
            }
            {
            ...
            }
        ]
    }
    """

    if (type(a) != int or type(b) != int):
        raise MyException("Los elementos deben ser enteros")

    resultado = [a % b == 0, a, b]
    return pasos.divisible(resultado)
```

La función *divisible* toma como parámetros dos valores enteros, a y b. Este algoritmo primero comprueba el tipo de los parámetros, en caso de no ser ambos enteros lanza una excepción. A continuación crea un array con tres valores: un booleano (*True* o *False*), para comprobar si a y b son divisibles, y los mismos a y b. Dicho array se usa posteriormente para obtener los pasos del algoritmo que se pasarán a la aplicación didáctica.

Función para devolver los pasos

```
def divisible(resultado):
    json = {}
    esDivisible = resultado[0]
    a = resultado[1]
    b = resultado[2]
    pasos = []
    explicacion = ""

    b = abs(b)

    match b:
        case 2:
            explicacion = "Un número es divisible entre 2 si es
            ↪  un número par, es decir, que termina en 0, 2, 4,
            ↪  6, u 8."
        case 3:
```

```
15              explicacion = "Un número es divisible entre 3 si la
                ↪  suma de sus dígitos es igual a 3 o a un múltiplo
                ↪  de 3."
16          case 4:
17              explicacion = "Un número es divisible entre 4 cuando
                ↪  sus últimos dos dígitos son 0 o múltiplo de 4."
18          case 5:
19              explicacion = "Un número es divisible entre 5 cuando
                ↪  su último dígito en 0 o 5."
20          case 6:
21              explicacion = "Un número es divisible entre 6 cuando
                ↪  cumple con los criterios de divisibilidad del 2
                ↪  y del 3, es decir, "\
22              "que sea un número par y que la suma de sus dígitos
                ↪  sea múltiplo de 3."
23          case 7:
24              explicacion = "Para saber si un número es divisible
                ↪  entre 7 hacemos lo siguiente: se debe
                ↪  multiplicar el último dígito "\
25              "por 2 y restarlo al número que conforman los demás
                ↪  dígitos. Esto, hasta que queda un número de solo
                ↪  un dígito. Si este "\
26              "es un 0 o un 7, entonces el número es divisible
                ↪  entre 7."
27          case 8:
28              explicacion = "Un número es divisible entre 8 si los
                ↪  últimos tres dígitos son múltiplos de 8 o
                ↪  iguales a 0."
29          case 9:
30              explicacion = "Un número es divisible entre 9 si la
                ↪  suma de sus dígitos es un múltiplo de 9."
31          case 10:
32              explicacion = "Un número es divisible entre 10 si
                ↪  termina en 0."
33          case _:
34              explicacion = ""
35
36      if explicacion != "":
37          pasos.append({"descripcion": explicacion})
38
39      if esDivisible:
40          pasos.append({"descripcion": str(a) + " es divisible por
                ↪  " + str(b)})
41      else:
42          pasos.append({"descripcion": str(a) + " no es divisible
                ↪  por " + str(b)})
43
```

```
44      json["pasos"] = pasos
45
46      return json
```

El parámetro que recibe la función *divisible* para devolver los pasos consta de una lista con tres valores: a, b y un booleano que representa *True* en caso de que a sea divisible entre b y *False* en caso contrario.

En primer lugar se comprueba si el parámetro b tiene algún tipo de criterio de divisibilidad mediante un *switch* (llamado *match* en Python), en cuyo caso se guarda en una cadena de texto y se añade como un paso extra a JSON de los pasos.

Finalmente, se añade un paso con una respuesta que dice si a es divisible por b.

3.1.4. Ecuaciones diofánticas lineales

Algoritmo

```
1   def ecDiofantica(a, b, c):
2       """
3       Función que resuelve ecuaciones diofánticas. Los parámetros
        ↪ que admite son
4       tres enteros a, b y c que representan la ecuación diofántica
        ↪ lineal
5       ax + by = c.
6
7       Retorna como resultado el siguiente JSON:
8
9       {
10          "pasos": [
11              {
12                  "pasoLatex": str,
13                  "descripcion": str
14              }
15              {
16              ...
17              }
18          ],
19          "valores": [str]
20      }
21      """
22      # Comprobamos que se cumple que el mcd(a, b) divide c
23      if(type(a) != int or type(b) != int or type(c) != int):
24          raise MyException("Los elementos deben ser enteros")
25
26      # Aplicamos el algoritmo
27      # El resultado a pasar sera: [a, b, c, mcd(a, b), m, n, x1,
        ↪ y1, x2, y2]
```

```
28      resultado = []
29
30      mcdLocal = int(mcd([abs(a)], [abs(b)])["resultado"])
31      if(c % mcdLocal != 0):
32          resultado = [a, b, c, mcdLocal]
33          return pasos.ecDiofantica(resultado)
34
35      if (a < 0) ^ (b < 0):
36          mcdLocal = mcdLocal * (-1)
37
38      # Aplicamos Bezout para obtener m y n
39      m, n = int(bezout([a], [b])["m"]), int(bezout([a],
        ↪  [b])["n"])
40
41      # Calculamos la solución
42      # x = (mc/mcdLocal)+(bt/mcdLocal), y =
        ↪  (nc/mcdLocal)-(at/mcdLocal)
43      x1 = int((m * c) / mcdLocal)
44      y1 = int((n * c) / mcdLocal)
45      x2 = int(b / mcdLocal)
46      y2 = int((a * (-1)) / mcdLocal)
47
48      resultado = [a, b, c, mcdLocal, m, n, x1, y1, x2, y2]
49
50      return pasos.ecDiofantica(resultado)
51
52  # Ecuaciones diofánticas con restricciones
53  def ecDiofantica_res(a, b, c, resX, resY):
54      """
55      Algorítmo para resolver ecuaciones diofánticas con
    ↪   restricciones. Los parámetros
56      a, b y c son enteros que representan la ecuación diofántica
    ↪   lineal ax + by = c.
57      Los parámetros resX y resY corresponden a arrays con las
    ↪   restricciones
58      de X y de Y: [mayorQue, menorQue, mayorIgualQue,
    ↪   menorIgualQue]
59
60      Retorna como resultado el siguiente JSON:
61
62      {
63          "pasos": [
64              {
65                  "pasoLatex": str,
66                  "descripcion": str
67              }
68              {
```

```
69              ...
70              }
71          ]
72      }
73      """
74      if (resX[0] != None and resX[1] != None and resX[0] ==
        ↪  resX[1]) or (resY[0] != None and resY[1] != None and
        ↪  resY[0] == resY[1]):
75          raise MyException("Esa restricción no es posible")
76      elif (resX[2] != None and resX[3] != None and resX[2] ==
        ↪  resX[3]) or (resY[2] != None and resY[3] != None and
        ↪  resY[2] == resY[3]):
77          raise MyException("La resticción tiene que ser un
            ↪  intervalo")
78      elif (resX[0] != None and resX[1] != None and resX[0] >
        ↪  resX[1]) or (resY[0] != None and resY[1] != None and
        ↪  resY[0] > resY[1]):
79          raise MyException("Esa restricción no es posible")
80      elif (resX[2] != None and resX[3] != None and resX[2] >
        ↪  resX[3]) or (resY[2] != None and resY[3] != None and
        ↪  resY[2] > resY[3]):
81          raise MyException("Esa restricción no es posible")
82      elif (resX[0] != None and resX[3] != None and resX[0] >
        ↪  resX[3]) or (resY[0] != None and resY[3] != None and
        ↪  resY[0] > resY[3]):
83          raise MyException("Esa restricción no es posible")
84      elif (resX[1] != None and resX[2] != None and resX[1] <
        ↪  resX[2]) or (resY[1] != None and resY[2] != None and
        ↪  resY[1] < resY[2]):
85          raise MyException("Esa restricción no es posible")
86
87      resultado = [a, b, c, [], []] # [a, b, c, coeficientes
        ↪  diofántica, intervalo t]
88
89      # Recogemos la solución (si la tiene) de la ecuación
        ↪  diofántica
90      pasosDiofantica = ecDiofantica(a, b, c)["pasos"]
91      valEcuaciones = ecDiofantica(a, b, c)["valores"]
92      if valEcuaciones != []:
93          x1 = valEcuaciones[0]
94          y1 = valEcuaciones[1]
95          x2 = valEcuaciones[2]
96          y2 = valEcuaciones[3]
97          resultado[3] = [x1, y1, x2, y2]
98      else:
99          return pasos.ecDiofantica_res(resultado,
            ↪  pasosDiofantica)
```

```
100
101     # Calculamos las t para los valores de x e y
102     intervaloT_x = [None, None, None, None] # [mayorQue,
        ↪  menorQue, mayorIgualQue, menorIgualQue]
103     intervaloT_y = [None, None, None, None]
104
105     for i in range(len(resX)):
106         if resX[i] is not None:
107             x1 = x1 * (-1)
108             x = resX[i]
109             x = x + x1
110             t = x // x2
111             tAux = x / x2
112             if (tAux == t):
113                 if (x >= 0 and x2 >= 0) or (x >= 0 and x2 < 0):
114                     intervaloT_x[i] = t
115                 elif (x < 0 and x2 >= 0) or (x < 0 and x2 < 0):
116                     if (i == 0):
117                         intervaloT_x[1] = t
118                     elif (i == 1):
119                         intervaloT_x[0] = t
120                     elif (i == 2):
121                         intervaloT_x[3] = t
122                     else:
123                         intervaloT_x[2] = t
124             else:
125                 if (x >= 0) and (x2 >= 0):
126                     if (i == 0):
127                         intervaloT_x[2] = t + 1
128                     elif (i == 1):
129                         intervaloT_x[3] = t + 1
130                     else:
131                         intervaloT_x[i] = t + 1
132                 elif (x >= 0) and (x2 < 0):
133                     if (i == 0):
134                         intervaloT_x[2] = t
135                     elif (i == 1):
136                         intervaloT_x[3] = t
137                     else:
138                         intervaloT_x[i] = t
139                 elif (x >= 0) and (x2 < 0):
140                     if (i == 0):
141                         intervaloT_x[2] = t - 1
142                     elif (i == 1):
143                         intervaloT_x[3] = t - 1
144                     elif (i == 2):
145                         intervaloT_x[3] = t - 1
```

```
146                     else:
147                         intervaloT_x[2] = t - 1
148                 else:
149                     if (i == 0):
150                         intervaloT_x[3] = t
151                     elif (i == 1):
152                         intervaloT_x[2] = t
153                     elif (i == 2):
154                         intervaloT_x[3] = t
155                     else:
156                         intervaloT_x[2] = t
157 
158     for i in range(len(resY)):
159         if resY[i] is not None:
160             y1 = y1 * (-1)
161             y = resY[i]
162             y = y + y1
163             t = y // y2
164             tAux = y / y2
165             if (tAux == t):
166                 if (y >= 0 and y2 >= 0) or (y >= 0 and y2 < 0):
167                     intervaloT_y[i] = t
168                 elif (y < 0 and y2 >= 0) or (y < 0 and y2 < 0):
169                     if (i == 0):
170                         intervaloT_y[1] = t
171                     elif (i == 1):
172                         intervaloT_y[0] = t
173                     elif (i == 2):
174                         intervaloT_y[3] = t
175                     else:
176                         intervaloT_y[2] = t
177             else:
178                 if (y >= 0) and (y2 >= 0):
179                     if (i == 0):
180                         intervaloT_y[2] = t + 1
181                     elif (i == 1):
182                         intervaloT_y[3] = t + 1
183                     else:
184                         intervaloT_y[i] = t + 1
185                 elif (y >= 0) and (y2 < 0):
186                     if (i == 0):
187                         intervaloT_y[2] = t
188                     elif (i == 1):
189                         intervaloT_y[3] = t
190                     else:
191                         intervaloT_y[i] = t
192                 elif (y >= 0) and (y2 < 0):
```

```
193                     if (i == 0):
194                         intervaloT_y[2] = t - 1
195                     elif (i == 1):
196                         intervaloT_y[3] = t - 1
197                     elif (i == 2):
198                         intervaloT_y[3] = t - 1
199                     else:
200                         intervaloT_y[2] = t - 1
201                 else:
202                     if (i == 0):
203                         intervaloT_y[3] = t
204                     elif (i == 1):
205                         intervaloT_y[2] = t
206                     elif (i == 2):
207                         intervaloT_y[3] = t
208                     else:
209                         intervaloT_y[2] = t
210
211     # Agrupamos los intervalos de la t en un mismo array
212     intervalosT = [] # [mayorQue, menorQue, mayorIgualQue,
        ↪ menorIgualQue]
213     for i in range(len(intervaloT_x)):
214         if (intervaloT_x[i] == None and intervaloT_y[i] ==
            ↪ None):
215             intervalosT.append(None)
216         elif (intervaloT_x[i] != None and intervaloT_y[i] ==
            ↪ None):
217             intervalosT.append(intervaloT_x[i])
218         elif (intervaloT_x[i] == None and intervaloT_y[i] !=
            ↪ None):
219             intervalosT.append(intervaloT_y[i])
220         else:
221             if (i == 0 or i == 2):
222                 if (intervaloT_x[i] >= intervaloT_y[i]):
223                     intervalosT.append(intervaloT_x[i])
224                 else:
225                     intervalosT.append(intervaloT_y[i])
226             else:
227                 if (intervaloT_x[i] <= intervaloT_y[i]):
228                     intervalosT.append(intervaloT_x[i])
229                 else:
230                     intervalosT.append(intervaloT_y[i])
231     resultado[4] = intervalosT
232
233     # En el caso de que tenga soluciones finitas, las imprimimos
234     values = []
235     valoresT = []
```

```
236     for elem in intervalosT:
237         if elem != None:
238             values.append(elem)
239 
240     # Añadir todos los posible valores
241     if len(values) >= 2:
242         for number in range(values[0], values[1] + 1):
243             valoresT.append(number)
244         # Quitar los valores que esten en mayorQue o menorQue
245         for number in valoresT:
246             if number == intervalosT[0] or number ==
                ↪  intervalosT[1]:
247                 valoresT.remove(number)
248 
249     resultado.append(valoresT)
250     return pasos.ecDiofantica_res(resultado, pasosDiofantica)
```

El algoritmo para resolver ecuaciones diofánticas lineales comienza con la llamada a la función *ecDiofantica* que toma como parámetros a, b y c, los correspondientes a la ecuación $ax + by = c$.

En primer lugar se comprueba que todos los parámetros sean enteros, por lo que se lanza una excepción si no lo son. A continuación se comprueba si se verifica que el mcd$(a, b) = d$ divide a c ya que es la única condición para que la ecuación tenga solución. Si se cumple, calculamos m y n, los coeficientes del teorema de Bezout tales que $am + bn = d$. Finalmente construimos el sistema de ecuaciones asignando los siguientes valores: $x = x1 + x2t$ e $y = y1 - y2t$, con $x1 = \frac{mc}{d}$, $x2 = \frac{b}{d}$, $y1 = \frac{nc}{d}$ e $y2 = \frac{a}{d}$. Este resultado se pasa como array para crear el JSON de respuesta con los pasos del algoritmo.

En el caso de que se quiera resolver una ecuación diofántica con una serie de restricciones, se llama a la función *ecDiofantica_res*, cuya diferencia con *ecDiofantica* es que se añaden dos parámetros extra, la restricción para x y la restricción para y. Ambos parámetros son arrays con el mismo tipo de contenido (*mayorQue*, *menorQue*, *mayorIgualQue*, *menorIgualQue*) y son valores enteros, que representan su resstricción dependiendo de la posición en la que se encuentran.

Antes de empezar con su resolución, se comprueba que los parámetros *resX* y *resY* son posibles restricciones, en caso contrario se lanza una excepción. Se comprueba, por ejemplo, que no haya restricciones del tipo $x > 0$, $x < 0$ o $y < 2, y >= 3$ ya que no tendrían solución. Por ello se han cubierto todas las posibles combinaciones.

Una vez pasadas las comprobaciones, primero se resuelve la ecuación diofántica sin restricciones haciendo uso de la función *ecDiofantica*, creando las variables $x1$, $x2$, $y1$ e $y2$. Segundo, se crean dos arrays con los intervalos de las soluciones de x e y con el mismo formato que el de los parámetros (*mayorQue*, *menorQue*, *mayorIgualQue*, *menorIgualQue*). Se realizan posteriormente dos bucles iterando sobre *resX* y *resY* y en cada iteración se despeja la t de la ecuación diofántica dependiendo si existe una restricción (en este paso se tiene en cuenta todas las

posibilidades en cuanto a los signos de la inecuación, es decir, cuando ambas partes son positivas, una negativa y otra positiva, y ambas negativas).

En este punto tenemos dos arrays para los intervalos en los que existe una solución en función de t, por lo que realizamos otro bucle para agrupar ambos arays en uno. Para ello, por cada posición de los arrays primero comprobamos si ambos son de tipo *None*, en cuyo caso añadimos *None* al array final; segundo, en el caso de que uno tenga *None* y el otro tenga un valor, introducimos el valor; y tercero, en el caso de que tengamos dos valores, los separamos en dos tipos: los de las posiciones 0 y 2 y los de las posiciones 1 y 3 (agrupamos los 'mayores que' por una parte y los 'menores que' en otra). Cuando estamos en las posiciones 0 y 2, elegimos el mayor valor de los dos y lo introducimos en el array y cuando estamos en las posiciones 1 y 3 elegimos el menor valor. Por lo tanto ahora tenemos un array de cuatro posiciones en el que como máximo hay dos posiciones con valores (uno para '*mayor*' o para '*mayor o igual*' y otro para '*menor*' o para '*menor o igual*'.

Teniendo ahora un array con ls posibles intervalos de t, podemos tener dos tipos de soluciones: una solución finita o infinita. En el caso de tener infinitas soluciones (se puede dar en el caso de que sólo haya un intervalo de t o ninguno) se imprime solamente el intervalo de x e y que tenga solución. En el caso de que sean finitas, se imprimen dichas soluciones sustituyendo los posibles valores de t en la solución inicial.

Función para devolver los pasos

```
1  def ecDiofantica(resultado):
2      json = {}
3      pasos = []
4
5      if len(resultado) == 4:
6          pasos.append({"descripcion": "La ecuación no tiene
           ↪  solución ya que el $\mathrm{mcd}(" +
           ↪  str(abs(resultado[0])) + ", " +
           ↪  str(abs(resultado[1])) + ") = " +
           ↪  str(abs(resultado[3])) + "$ no divide a " +
           ↪  str(resultado[2])})
7          json["pasos"] = pasos
8          json["valores"] = []
9      else:
10         pasos.append({"pasoLatex": "\mathrm{mcd}(" +
           ↪  str(abs(resultado[0])) + ", " +
           ↪  str(abs(resultado[1])) + ") = " +
           ↪  str(abs(resultado[3])) + " \\mid " +
           ↪  str(resultado[2]), \
11                       "descripcion": "La ecuación tiene solución
                          ↪  ya que se verifica que:"})
12
```

```
13          pasos.append({"pasoLatex":  "m = " + str(resultado[4]) +
            ↪  " \\text{, } n = " + str(resultado[5]), \
14                        "descripcion": "Aplicamos el teorema de
                          ↪  Bezout para " + str(resultado[0]) + "
                          ↪  y " + str(resultado[1]) + " para
                          ↪  obtener los valores m y n:"})
15
16          expresion1 = "x = " + str(resultado[6])
17          expresion2 = "y = " + str(resultado[7])
18
19          if resultado[8] < 0:
20              expresion1 += " - "
21          else:
22              expresion1 += " + "
23
24          if resultado[9] < 0:
25              expresion2 += " - "
26          else:
27              expresion2 += " + "
28
29          expresion1 += str(abs(resultado[8])) + "t"
30          expresion2 += str(abs(resultado[9])) + "t"
31
32          pasos.append({"pasoLatex": "\\begin{cases} " +
            ↪  expresion1 + "\\\\" + expresion2 + " \\end{cases}",
33                        "descripcion": "Las soluciones de la
                          ↪  ecuación diofántica lineal son:"})
34          json["pasos"] = pasos
35          json["valores"] = [resultado[6], resultado[7],
            ↪  resultado[8], resultado[9]] # NO USAR PARA LA
            ↪  RESPUESTA, se usa para ecDiofantica_res
36
37      return json
38
39  def ecDiofantica_res(resultado, pasosDiofantica):
40      json = {}
41      pasos = []
42
43      if resultado[3] == []:
44          pasos.append({"descripcion": "La ecuación no tiene
            ↪  solución ya que el $\mathrm{mcd}(" +
            ↪  str(abs(resultado[0])) + ", " +
            ↪  str(abs(resultado[1])) + ")$ no divide a " +
            ↪  str(resultado[2])})
45      else:
46          for paso in pasosDiofantica:
47              pasos.append(paso)
```

```
48          expresion1 = "x = " + str(resultado[3][0])
49          expresion2 = "y = " + str(resultado[3][1])
50
51          if resultado[3][2] < 0:
52              expresion1 += " - "
53          else:
54              expresion1 += " + "
55
56          if resultado[3][3] < 0:
57              expresion2 += " - "
58          else:
59              expresion2 += " + "
60
61          expresion1 += str(abs(resultado[3][2])) + "t"
62          expresion2 += str(abs(resultado[3][3])) + "t"
63
64          if resultado[4] != []:
65              intervalo = resultado[4]
66              valoresT = ""
67
68              for i in range(len(intervalo)):
69                  if intervalo[i] != None:
70                      if i == 0:
71                          valoresT += ("t > " + str(intervalo[i]))
                             ↪  + ", "
72                      elif i == 1:
73                          valoresT += ("t < " + str(intervalo[i]))
                             ↪  + ", "
74                      elif i == 2:
75                          valoresT += ("t >= " +
                             ↪  str(intervalo[i])) + ", "
76                      else:
77                          valoresT += ("t <= " +
                             ↪  str(intervalo[i])) + ", "
78              valoresT = valoresT[:-2]
79              pasos.append({"pasoLatex": valoresT, \
80                            "descripcion": "Los valores de t en
                               ↪  los cuales existe solución:"})
81
82              valoresX = []
83              valoresY = []
84              for t in resultado[5]:
85                  x = int((t * resultado[3][2]) + resultado[3][0])
86                  y = int((t * resultado[3][3]) + resultado[3][1])
87                  valoresX.append(x)
88                  valoresY.append(y)
89
```

```
90              if (len(valoresX) != 0) or (len(valoresY) != 0):
91                  valoresVariableX = ""
92                  valoresVariableY = ""
93
94                  for i in range(len(valoresX)):
95                      if i == 0:
96                          valoresVariableX += "x = {"
97                      valoresVariableX += str(valoresX[i])
98                      if i != len(valoresX) - 1:
99                          valoresVariableX += ", "
100                     else:
101                         valoresVariableX += "}"
102                 for i in range(len(valoresY)):
103                     if i == 0:
104                         valoresVariableY += "y = {"
105                     valoresVariableY += str(valoresY[i])
106                     if i != len(valoresY) - 1:
107                         valoresVariableY += ", "
108                     else:
109                         valoresVariableY += "}"
110
111                 pasos.append({"pasoLatex": valoresVariableX + ",
                     ↪  " + valoresVariableY, \
112                                 "descripcion": "Posibles soluciones
                                     ↪  de las variables X e Y"})
113
114     json["pasos"] = pasos
115     return json
```

La función para devolver los pasos de la resolución de ecuaciones diofánticas sin restricciones se llama *ecDiofantica* y toma como parámetro un array llamado *resultado*. En función de la longitud de dicho array se comprueba si la ecuación tiene solución. Por ello se comprueba en primer lugar si la longitud es cuatro, en cuyo caso se devuelve un paso explicando que la ecuación no tiene solución debido a que el $\mathrm{mcd}(a, b)$ no divide a c (teniendo una ecuación de la forma $ax+by = c$).

En el caso de que la longitud del array *resultado* sea mayor que cuatro, el primer paso que se añade es que la ecuación tiene solución. El siguiente paso explica como se aplica el teorema de Bezout para obtener los coeficientes m y n. A continuación se comienza a construir el sistema con las soluciones en función de t. Para ello se toman las posiciones 6, 7, 8 y 9 del array *resultado*, que corresponden respectivamente con los coeficiente de la solución representadas por c_1, c_2, c_3 y c_4 en $x = c_1+c_2t$ e $y = c_3+c_4t$. Dependiendo del signo que tengan los coeficientes se van a construir las cadenas de texto con la solución de una manera u otra, que finalmente se van a añadir al paso final.

Por otro lado tenemos la función que contruye los pasos para la resolución de ecuaciones diofánticas con restricciones llamada *ecDiofantica_res*. Esta función

es similar a *ecDiofantica* ya que lo primero que realiza es comprobar la longitud del parámetro *resultado*. En caso de que la longitud sea igual a tres, el código entiende que no hay solución, por lo que ese será el único paso que se devuelva. En caso contrario, utiliza los mismos pasos para contruir la solución de la ecuación que sin restricciones. Para representar los intervalos en los que se encuentra la t, se toma la posición 4 del array *resultado*, que contiene una lista con dichos intervalos de la forma *[mayorQue, menorQue, mayorIgualQue, menorIgualQue]*. Iterando sobre dicho array se contruyen cadenas de texto para representar dichos intervalos e incluirlos en un paso. Finalmente, en el caso de que el intervalo de t sea finito, se sustituyen las posibles soluciones para obtener los respectivos x e y para poder incluirlos en un paso adicional.

3.1.5. Numeros primos

Algoritmo

```
1  def esPrimo(n):
2      """
3      Algorítmo para comprobar si un número es primo. Toma como
       ↪ parámetro
4      un número entero.
5
6      Retorna como resultado el siguiente JSON:
7
8      {
9          "pasos": [
10             {
11                 "descripcion": str
12             }
13         ],
14         "esPrimo": boolean
15     }
16     """
17     if n <= 1:
18         raise MyException("El número introducido tiene que ser
           ↪ mayor que 1")
19     elif type(n) != int:
20         raise MyException("El número introducido debe ser un
           ↪ número natural")
21     else:
22         isPrime = sympy.isprime(n)
23         return pasos.esPrimo(n, isPrime)
```

La función *esPrimo* toma como parámetro un número entero y comprueba si este es primo o compuesto. Inicialmente se comprueba si este parámetro es mayor que 1 o es un entero, en caso contrario lanza una excepción. Una vez que pasa las comprobaciones, se hace uso de la librería *sympy* que incluye un método para comprobar si un número es primo, guardando el resultado en la variable

isPrime. Finalmente se pasan el número y la variable a la función para devolver los pasos.

Función para devolver los pasos

```
def esPrimo(n, resultado):
    json = {}
    pasos = []

    if resultado == False:
        pasos.append({"descripcion": "El número " + str(n) + "
        ↪  no es primo, es compuesto"})
    else:
        pasos.append({"descripcion": "El número " + str(n) + "
        ↪  es primo"})

    json["pasos"] = pasos
    json["esPrimo"] = resultado

    return json
```

La función *esPrimo* para devolver los pasos toma como parámetro una n, que corresponde con el número que se quiere comprobar, y una variable booleana *resultado*, cuyo valor es *True* si n es primo y *False* si es compuesto.

Para cada uno de los casos se crea un JSON con una *"descripción"* que va a contener la respuesta de si es primo o compuesto. Además se añade una variable al JSON llamada *.esPrimo"*, conteniendo el valor booleano correspondiente a la solución. Esta variable se va a usar posteriormente en el teorema fundamental de la aritmética.

3.1.6. Teorema fundamental de la aritmética

Algoritmo

```
def tfAritmetica(n):
    """
    Función que comprueba si se cumplen las condiciones para
    ↪  calcular la descomposición
    en factores primos de un número. En tal caso llama a la
    ↪  función auxiliar tfAritmetica_aux()
    para realizar el cálculo. Toma como parámetro un entero n.

    Retorna como resultado el siguiente JSON:

    {
        "pasos": [
            {
                "pasoLatex": str,
```

```
13                  "descripcion": str
14              }
15              {
16              ...
17              }
18          ]
19      }
20      """
21      if n <= 1:
22          raise MyException("El número introducido tiene que ser
            ↪ mayor que 1")
23      elif type(n) != int:
24          raise MyException("El número introducido debe ser un
            ↪ número natural")
25      else:
26          resultado = tfAritmetica_aux(n)
27          return pasos.tfAritmetica(n, resultado)
28
29  # Función auxiliar para calcular la descomposición en factores
    ↪ primos
30  def tfAritmetica_aux(n):
31      """
32      Función auxiliar que calcula la descomposición en factores
    ↪ primos de n.
33      """
34      divisoresPrimos = []
35      divisor = 2
36      while n != 1:
37          if n % divisor == 0:
38              n = n / divisor
39              divisoresPrimos.append(divisor)
40          else:
41              isPrime = False
42              while isPrime == False:
43                  divisor = divisor + 1
44                  if esPrimo(divisor)["esPrimo"]:
45                      isPrime = True
46      return divisoresPrimos
```

El algoritmo para calcular la descomposición en factores primos de un número está formado por dos funciones, *tfAritmetica* y *tfAritmetica_aux*.

La primera es una función que comprueba que se cumplan las condiciones para descomponer un número en factores primos. Para ello recibe el parámetro n y comprueba si es mayor que 1 y que sea un número entero.

Una vez pasa las comprobaciones, se llama a la función *tfAritmetica_aux* con el mismo parámetro. Esta función itera sobre n hasta que esta sea igual a 1. En cada iteración se comprueba si n es divisible entre el primer número primo

guardado en la variable *divisor* y en tal caso se asigna a n el valor de la división entre n y el *divisor*. En caso de que no sea divisible, se realiza un bucle para encontrar el siguiente número primo a partir de *divisor*.

En cada iteración del bucle principal en el que se halla encontrado un divisor primo, este se introducía en un array que posteriormente se usa para construir el JSON de respuesta con los pasos del algoritmo.

Función para devolver los pasos

```
def tfAritmetica(n, resultado):
    json = {}
    pasos = []
    bases = []
    repeticiones = []

    # Buscamos todas las bases sin repetir
    for base in resultado:
        if base not in bases:
            bases.append(base)

    # Contamos cuántas veces aparece una base
    for base in bases:
        contador = 0
        for i in resultado:
            if base == i:
                contador = contador + 1
        repeticiones.append(contador)

    # Agrupamos las potencias y las multiplicamos
    expresion = ""
    expresionLatex = ""
    for i in range(len(bases)):
        expresion = expresion + str(bases[i]) + "^" +
        ↪  str(repeticiones[i]) + " * "
        expresionLatex = expresionLatex + str(bases[i]) + "^{" +
        ↪  str(repeticiones[i]) + "} \\cdot "

    expresion = expresion[0:len(expresion)-3]
    expresion = expresion.replace("^1", "")
    expresion += ""

    expresionLatex = expresionLatex[0:len(expresionLatex)-7]
    expresionLatex = expresionLatex.replace("^{1}", "")
    expresionLatex += ""

    if expresion == str(n):
```

```
36          pasos.append({"descripcion": "El número " + str(n) + "
             ↪  es primo, por lo que no tiene descomposición"})
37      else:
38          respuesta = str(n) + " = " + expresionLatex
39          pasos.append({"pasoLatex": str(n) + " = " +
             ↪  expresionLatex,
40                        "descripcion": "La descomposición
                           ↪  factorial de " + str(n) + " es: "})
41
42      json["pasos"] = pasos
43      return json
```

La función para obtener los pasos del teorema fundamental de la aritmética se llama *tfAritmetica* y toma como parámetros dos valores: n y *resultado*. El número n es el valor sore el cual se quiere representar como una descomposición en valores primos y *resultado* es una lista con dichos números primos.

Lo primero que hace la función es tomar ese array y guardar todos los primos que son distintos en una lista. Por cada uno de los primos se vuelve a recorrer *resultado* contando las instancias en las que aparece cada primo. En este momento tenemos dos listas: una con los números primos y otra con las veces que aparece, es decir, su potencia.

Recorriendo ambos arrays concatenamos en una cadena de texto los primos con la potencia que le corresponde y posteriormente eliminar las partes del texto que son irrelevantes o redundantes (como elevar un número a 1).

Finalmente construimos el JSON con la respuesta añadiendo la cadena de texto al *"pasoLatex"*, además de una descripción. En el caso de que el número inicial n sea primo, se añade una respuesta explicando que el número es primo y por lo tanto no tiene descomposición.

3.1.7. Sistemas de numeración en base b

Algoritmo

```
1   def base(n, b):
2       """
3       Función que comprueba si un número puede ser representado en
     ↪  base a otro.
4       En tal caso llama a la función auxiliar base_aux() para
     ↪  realizar el algoritmo.
5       Toma como parámetros dos enteros: un número y la base en la
     ↪  que se quiere representar.
6
7       Retorna como resultado el siguiente JSON:
8
9       {
10          "pasos": [
11              {
```

```
12                  "pasoLatex": str,
13                  "descripcion": str
14              }
15              {
16              ...
17              }
18          ]
19      }
20      """
21      if b < 2:
22          raise MyException("La base tiene que ser mayor o igual
            ↪  que 2")
23      elif type(b) != int:
24          raise MyException("La base tiene que ser un número
            ↪  natural")
25      elif type(n) != int:
26          raise MyException("El número tiene que ser natural")
27      else:
28          resultado = base_aux(n, b)
29          return pasos.base(n, b, resultado)
30
31  # Función auxiliar para calcular n en base t
32  def base_aux(n, b):
33      """
34      Función auxiliar para calcular un número en una base
    ↪   determinada.
35      """
36      resultado = []
37      while n >= 2:
38          resto = n % b
39          resultado.insert(0, resto)
40          n = n // b
41      if n % b != 0:
42          resultado.insert(0, n % b)
43      return resultado
```

El algoritmo para calcular un número n en base b comienza con la función *base*, que toma como parámetros dichas variables. Esta función comprueba que la base b sea mayor o igual que 2 y que tanto n como b sean enteros.

Una vez las comprobaciones son correctas, se llama a la función *base_aux* con los mismos parámetros. Esta función itera sobre la variable n hasta que sea menor que 2. En cada iteración se calcula el módulo entre n y b y se guarda en una variable llamada *resto*, que se añade a un array de resultados. A continuación asignamos a la variable n el valor del cociente de la división entre n y b. Finalmente añadiremos el módulo entre n y b al array de respuestas, en caso de que sea distinto de cero al terminar el bucle.

Posteriormente, este array de resultados se pasa a la función para obtener los

pasos completos del algoritmo.

Función para devolver los pasos

```
def base(n, b, resultado):
    json = {}
    pasos = []
    abecedario = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    # Cambiamos los valores que sean mayores a 10
    resultadoNuevo = []
    for i in range(len(resultado)):
        if resultado[i] >= 10:
            letra = resultado[i] - 10
            resultadoNuevo.append(abecedario[letra])
        else:
            resultadoNuevo.append(resultado[i])

    # Concatenamos todos los elementos del array
    resultadoATexto = "".join([str(_) for _ in resultadoNuevo])

    # Establecemos los valores del abecedario
    valores = []
    for i in range(len(resultadoNuevo)):
        if resultadoNuevo[i] != resultado[i]:
            valores.append((resultadoNuevo[i], resultado[i]))

    # Añadimos los valores del abecedario al JSON si los hay
    if len(valores) != 0:
        explicacionLatex = ""
        for i in valores:
            valor = i[0] + " = " + str(i[1])
            explicacionLatex += valor + ", "
        explicacionLatex = explicacionLatex[:-2]
        pasos.append({"pasoLatex": explicacionLatex, \
                      "descripcion": "Los valores del abecedario
                      ↪  que se van a usar son:"})

    #Añadimos el paso resultado al JSON
    pasos.append({"descripcion": "El número " + str(n) + " en
    ↪  base " + str(b) + " es igual a " + resultadoATexto})

    json["pasos"] = pasos
    return json
```

La función *base* para obtener los pasos toma como parámetros el número inicial n, la base en la que se quiere calcular b y un array *resultado* con los dígitos de n en base b.

En primer lugar el algoritmo recorre todas las posiciones del array *resultado* y cambia los valores que sean mayores que 10. Para ello usa una cadena de texto llamada *abecedario*, obteniendo la letra que le corresponde según su posición comenzando a contar por 10. A continuación concatenamos todos los elementos del array y lo guardamos en una variable llamada *resultadoATexto*.

Posteriormente, se comprueba si se ha tenido que cambiar algún dígito por un elemento de abecedario. Para ello se recorre el nuevo array y se compara posición a posición con el array original. En el caso de encontrar un elemento distinto, se añade ese elemento a la lista de los valores del abecedario usados. De esta forma obtenemos una cadena de texto con las letras y sus valores correspondientes y lo añadimos a un nuevo paso.

Por último, añadimos el abecedario (en caso de haberlo usado) y el resultado en dos pasos distintos, explicados con un *"pasoLatex"* y una *"descripcion"*.

3.1.8. Unidades

Algoritmo

```
1  def esUnidad(a, m):
2      """
3      Función para calcular si un número es un elemento inversible
       ↪  o unidad en Zm.
4      Toma como parámetros un entero y el módulo en el que se
       ↪  encuentra Z.
5
6      Retorna como resultado el siguiente JSON:
7
8      {
9          "pasos": [
10             {
11                 "descripcion": str
12             }
13         ]
14     }
15     """
16
17     if a >= m or a < 0:
18         raise MyException("El número " + str(a) + " no pertenece
           ↪  a Z" + str(m))
19
20     # Calculamos si es unidad
21     unidad = False
22     i = 0
23
24     while not unidad and i < m:
25         if (a * i) % m == 1:
26             unidad = True
```

```
27          i+=1
28
29      return pasos.esUnidad(a, m, unidad)
```

La función *esUnidad* se usa para comprobar si un número es un elemento inversible o unidad en $\mathbb{Z}_m$. Por ello se pasan como parámetros un número a y m.

Lo primero que hace el algoritmo es comprobar las excepciones. En este caso la única posibilidad de excepción es que $a \geq m$ o que $a < 0$.

En caso de que no salte la excepción, se itera sobre la variable booleana *unidad* hasta que sea verdadera o que hayamos recorrido todas las posibilidades hatsa m. En cada iteración comprobamos la definición de unidad ($a \cdot i \,\%m = 1$) y en caso de que se cumpla cambiamos la variable booleana *unidad* a *True*, en caso contrario seguimos iterando.

Finalmente devolvemos los pasos del algoritmo pasando como parámetros la a, la m y la variable *unidad*.

Función para devolver los pasos

```
1  def esUnidad(a, z, unidad):
2      json = {}
3      pasos = []
4      jsonAux = {}
5
6      if unidad:
7          jsonAux["descripcion"] = "El número $\\overline{" +
           ↪  str(a) + "}$ es un elemento inversible ó unidad en
           ↪  $\\mathbb{Z}_{" + str(z) + "}$"
8      else:
9          jsonAux["descripcion"] = "El número $\\overline{" +
           ↪  str(a) + "}$ no es un elemento inversible ó unidad
           ↪  en $\\mathbb{Z}_{" + str(z) + "}$"
10
11     pasos.append(jsonAux)
12     json["pasos"] = pasos
13
14     return json
```

La función *esUnidad* para devolver los pasos toma como parámetros el número a que se quiere comprobar, el conjunto z y una variable booleana *unidad* cuyo valor es *True* si a es una unidad en $\mathbb{Z}_z$ y *False* en caso contrario.

En ambos casos se crea un JSON con una *"descripcion"* que contiene la comprobación de la unidad escrito en formato Latex.

3.1.9. Divisores de cero

Algoritmo

```
def esDivisorDeCero(a, m):
    """
    Función para calcular si un número es un divisor de cero en Zm.
    Toma como parámetros un entero y el módulo en el que se encuentra Z.

    Retorna como resultado el siguiente JSON:

    {
        "pasos": [
            {
                "descripcion": str
            }
        ]
    }
    """

    if m <= 1:
        raise MyException("Z no puede ser menor o igual que 1")
    if a >= m or a < 0:
        raise MyException("El número " + str(a) + " no pertenece a Z" + str(m))

    # Calculamos si es divisor de cero
    divisor = False
    i = 1

    while not divisor and i < m and a != 0:
        if (a * i) % m == 0:
            divisor = True
        i+=1

    return pasos.esDivisorDeCero(a, m, divisor)
```

La función *esDivisorDeCero* toma como parámetros a y m para comprobar si el número a es un divisor de cero en $\mathbb{Z}_m$.

Para ello inicialmente comprueba si m es mayor que 1 y que a sea menor que m y mayor que 0. En el caso de que no se cumpla una de las condiciones se lanza una excepción. En caso contrario se crea una variable *i* que va a actuar como iterador y una variable llamada *divisor* en la que se guarda un valor booleano *True* si a es divisor de cero. A continuación comenzamos un bucle *while* cuyas condiciones son que *divisor* sea *False*, que *i* sea menor que m y que a sea distinto de 0. En cada iteración comprobamos si $(a \cdot i)\,\%m = 0$ y si se cumple cambiamos

la variable *divisor* a *True*.

Cuando termina el bucle usamos las variables a, m y *divisor* para construir el JSON con los pasos del algoritmo.

Función para devolver los pasos

```
def esDivisorDeCero(a, z, divisor):
    json = {}
    pasos = []
    jsonAux = {}

    if divisor:
        jsonAux["descripcion"] = "El número $\\overline{" +
        ↪  str(a) + "}$ es un divisor de cero en
        ↪  $\\mathbb{Z}_{" + str(z) + "}$"
    else:
        jsonAux["descripcion"] = "El número $\\overline{" +
        ↪  str(a) + "}$ no es un divisor de cero en
        ↪  $\\mathbb{Z}_{" + str(z) + "}$"

    pasos.append(jsonAux)
    json["pasos"] = pasos

    return json
```

La función *esDivisorDeCero* para devolver los pasos toma como parámetros el número a que se quiere comprobar, el conjunto z y una variable booleana *divisor* cuyo valor es *True* si a es divisor de cero en $\mathbb{Z}_z$ y *False* en caso contrario.

En ambos casos se crea un JSON con una *"descripcion"* que contiene la comprobación para saber si es un divisor de cero escrito en formato Latex.

3.1.10. Inversos

Algoritmo

```
def inverso(a, m):
    """
    Cálculo del inverso de un número en Zm si lo tiene. Toma
    ↪  como parámetros un entero y
    el módulo en el que se encuentra Z.

    Retorna como resultado el siguiente JSON:

    {
        "pasos": [
            {
                "descripcion": str
```

```
12              }
13          ],
14          "resultado": str
15      }
16      """
17
18      if m <= 1:
19          raise MyException("Z no puede ser menor o igual que 1")
20      if a >= m or a < 0:
21          raise MyException("El número " + str(a) + " no pertenece
            ↪  a Z" + str(m))
22
23      # Calculamos el inverso de a
24      unidad = False
25      i = 0
26
27      while not unidad and i < m:
28          if (a * i) % m == 1:
29              unidad = True
30              break
31          i+=1
32
33      return pasos.inverso(i, unidad, a, m)
```

Para calcular el inverso de un número a en $\mathbb{Z}_m$ se va a usar la función *inverso*, que toma a y m como parámetros.

El algoritmo es similar al algoritmo de la función *esDivisorDeCero*, ya que comprueba que m sea mayor que 1 y que a sea menor que m y mayor que 0. Luego usa el mismo bucle, comprobando si $(a \cdot i)\,\%m = 1$ con i inicialmente a 0. En el caso de que se cumpla la condición, se guarda la variable i que se usará para construir el JSON con los pasos del algoritmo.

Función para devolver los pasos

```
1  def inverso(i, unidad, a, z):
2      json = {}
3      pasos = []
4      jsonAux = {}
5
6      if not unidad:
7          jsonAux["descripcion"] = "El número $\\overline{" +
           ↪  str(a) + "}$ en $\\mathbb{Z}_" + str(z) + "$ no
           ↪  tiene inverso"
8      else:
9          jsonAux["descripcion"] = "El inverso de $\\overline{" +
           ↪  str(a) + "}$ en $\\mathbb{Z}_{" + str(z) + "}$ es:
           ↪  $\\overline{" + str(a) + "}^{-1} = \\overline{" +
           ↪  str(i) + "}$"
```

```
10
11      pasos.append(jsonAux)
12      json["pasos"] = pasos
13
14      if not unidad:
15          json["resultado"] = ""
16      else:
17          json["resultado"] = str(i)
18
19      return json
```

La función *inverso* para devolver los pasos toma como parámetros el número a cuyo inverso se quiere calcular, el conjunto z, una variable booleana *unidad* cuyo valor es *True* si a es unidad en $\mathbb{Z}_z$ y *False* en caso contrario y una variable i que es el resultado del inverso de a en $\mathbb{Z}_z$.

En ambos casos se crea un JSON con una *"descripcion"* que contiene la comprobación para saber si a tiene inverso en z escrito en formato Latex.

Por último, en caso de tener inverso, se crea una variable adicional llamada *resultado"* en el que se guarda el valor actual de dicho inverso. Este paso es importante ya que se va a tener que usar al calcular ecuaciones o sistema lineales de congruencias.

3.1.11. Ecuaciones lineales de congruencias

Algoritmo

```
1  def ecuacion(xIzquierda, valoresIzquierda, xDerecha,
   ↪  valoresDerecha, z):
2      """
3      Función para resolver ecuaciones lineales de congruencias.
   ↪  Toma como parámetros
4      cuatro arrays que corresponden, respectivamente, a los
   ↪  coeficientes a la izquierda
5      de la ecuación acompañados de una X, coeficientes a la
   ↪  izquierda que no están
6      acompañados, coeficientes a la derecha acompañados de una X
   ↪  y coeficientes
7      a la derecha no acompañados. Por último también recibo como
   ↪  parámetro el módulo
8      en el que se encuentra Z.
9
10     Retorna como resultado el siguiente JSON:
11
12     {
13         "pasos": [
14             {
15                 "pasoLatex": str,
```

```
16                  "descripcion": str
17              }
18              {
19              ...
20              }
21          ]
22      }
23      """
24
25      # Comprobación de condiciones
26      if z <= 1:
27          raise MyException("El módulo tiene que ser mayor que 1")
28
29      # Operamos y dejamos las X a un lado
30      simplificado = False
31      if (len(xIzquierda) + len(valoresIzquierda)) > 1 or
        ↪  (len(xDerecha) + len(valoresDerecha)) > 1:
32          simplificado = True # Para añadir un paso de
            ↪  simplificación
33      a = sum(xIzquierda) - sum(xDerecha)
34      b = sum(valoresDerecha) - sum(valoresIzquierda)
35
36      # Comprobamos que la ecuación tiene solución
37      mcdLocal = mcd([a], [z])["resultado"]
38      tieneSolucion = True
39      if b % int(mcdLocal) != 0:
40          tieneSolucion = False
41          az = 0
42          bz = 0
43          x = []
44          resultado = False
45      else:
46          # Calculamos a y b en Zm
47          az = a % z
48          bz = b % z
49
50          # Resolvemos la ecuación
51          x = []
52
53          x.append("x = " + str(az) + "^{-1} * " + str(bz))
54
55          inverso_az = int(inverso(az, z)["resultado"])
56          x.append("x = " + str(inverso_az) + " * " + str(bz))
57
58          multiplicacion = inverso_az * bz
59          resultado = multiplicacion
60          x.append("x = " + str(multiplicacion))
```

```
61
62          if multiplicacion >= z:
63              multiplicacionEnZ = multiplicacion % z
64              resultado = multiplicacionEnZ
65              x.append("x = " + str(multiplicacionEnZ))
66      return pasos.ecuacion(simplificado, a, b, z, mcdLocal,
        ↪  tieneSolucion, az, bz, x, resultado)
```

La función para resolver ecuaciones lineales de congruencias de llama *ecuacion* que toma cinco parámetros: *xIzquierda*, *valoresIzquierda*, *xDerecha*, *valoresDerecha* y *z*. Los cuatro primeros son arrays que corresponden con los coeficientes a la izquerda de la ecuación que están acompañados de una x (*xIzquierda*), los coeficientes a la izquierda que no están acompañados de una x (*valoresIzquierda*), los coeficientes a la derecha acompañados de x (*xDerecha*) y los coeficientes a la derecha no acompañados (*valoresDerecha*).

El algoritmo primero comprueba que *z* sea un valor mayor que 1 ya que es el módulo en el que se va a operar en la ecuación. Después se comienza a despejar la ecuación dejando la x a la izquierda, por lo que tenemos una ecuación de la forma $ax \equiv b \pmod{z}$. Para que la ecuación tenga solución se comprueba que $\text{mcd}(a, z)$ divida a b, en cuyo caso se despeja a pasándola al otro lado de la ecuación mediante el cálculo de su inverso (usand la función *inverso*). Finalmente el inverso de a se multiplica a b y se calcula el resultado en módulo *z*.

Todas las variables que se han usado se pasan a una función para calcular e imprimir los pasos del algoritmo en formato JSON.

Función para devolver los pasos

```
1  def ecuacion(simplificado, a, b, z, mcdLocal, tieneSolucion, az,
   ↪  bz, x, resultado):
2      json = {}
3      pasos = []
4
5      if simplificado:
6          pasos.append({"pasoLatex": str(a) + "x = " + str(b) + "\
           ↪  (\mathrm{mod}\ " + str(z) + ")", \
7                        "descripcion": "Siplificando la ecuación
                         ↪  tenemos:"})
8
9      if not tieneSolucion:
10         pasos.append({"pasoLatex": "\mathrm{mcd}(" + str(a) + ",
           ↪  " + str(z) + ") = " + mcdLocal + " \\nmid " +
           ↪  str(b), \
11                       "descripcion": "La ecuación $" + str(a) +
                         ↪  "x \\equiv " + str(b) + "\
                         ↪  (\mathrm{mod}\ " + str(z) + ")$ no
                         ↪  tiene solución porque:"})
12     else:
```

```
13          pasos.append({"pasoLatex": "\mathrm{mcd}(" + str(a) + ",
            ↪  " + str(z) + ") = " + mcdLocal + " \\mid " + str(b),
            ↪  \
14                        "descripcion": "La ecuación $" + str(a) +
                          ↪  "x \\equiv " + str(b) + "\
                          ↪  (\mathrm{mod}\ " + str(z) + ")$ tiene
                          ↪  solución porque:"})
15
16          pasos.append({"pasoLatex": str(a) + "x = " + str(b) + "\
            ↪  (\mathrm{mod}\ " + str(z) + ") \\Longleftrightarrow
            ↪  " + str(az) + "x \\equiv " + str(bz) + " \\in
            ↪  \\mathbb{Z}_{" + str(z) + "}", \
17                        "descripcion": "Representamos la igualdad
                          ↪  en la Z que le corresponde"})
18
19          despejado = ""
20          for paso in x:
21              despejado += paso + " \\implies "
22          despejado = despejado[:-10]
23          pasos.append({"pasoLatex": despejado, \
24                        "descripcion": "Despejamos la x de la
                          ↪  ecuación"})
25
26          pasos.append({"pasoLatex": "x = " + str(resultado) + " +
            ↪  " + str(z) + "t \\quad t \\in \\mathbb{Z}", \
27                        "descripcion": "Representamos la solución
                          ↪  en $\\mathbb{Z}$"})
28
29          pasos.append({"pasoLatex": "x = " + str(resultado), \
30                        "descripcion": "Solución de la ecuación en
                          ↪  $\\mathbb{Z}_{" + str(z) + "}$"})
31
32      json["pasos"] = pasos
33
34      return json
```

La función para devolver los pasos de la resolución de ecuaciones lineales de congruencias se llama *ecuacion* y toma diez parámetros: *simplificado*, que es un booleano que indica si la ecuación se ha tenido que simplificar; *a*, el coeficiente que acompaña a la x; *b*, el coeficiente que no acompaña a la x y que se encuentra en el lado derecho de la ecuación; *z*, el módulo en el que hay que resolver la ecuación; *mcdLocal*, que representa el máximo común divisor de *a* y *z*; *tieneSolucion*, un booleano que indica si la ecuación tiene solución; *az*, la representación de *a* en módulo *z*; *bz*, la representación de *b* en módulo *z*; *x*, un array con los pasos para despejar la x de la ecuación y calcular la solución; y *resultado*, un entero que representa la solución.

La función comienza comprobando el parámetro *simplificado* para comprobar si

se ha tenido que simplificar la solución. En tal caso se añade un paso representando la nueva ecuación simplificada.

A continuación se comprueba el parámetro *tieneSolucion*. En caso de ser *False*, se añade un único paso explicando que la ecuación no tiene solución debido a que el máximo común divisor de a y z, es decir *mcdLocal*, no divide a b. En caso de tener solución, comienza explicando que sí la tiene porque cumple con la condición necesaria y suficiente. A continuación se añade otro paso representando la ecuación en la $\mathbb{Z}_m$ indicada en el problema.

Por último se itera sobre el parámetro x para concatenar la secuencia de pasos para despejar la x de la ecuación, representando posteriormente dicha solución en $\mathbb{Z}$ y en $\mathbb{Z}_m$.

3.1.12. Sistemas lineales de congruencias

Algoritmo

```
def sistema(ecuaciones):
    """
    Función para resolver sistemas lineales de congruencias mediante
    el teorema chino del resto. La función toma como parámetro un array
    llamado 'ecuaciones' que contiene listas [c, a, p],
    respresentando el sistema de la siguiente forma:

        c1 = a1 (mod p1)
        c2 = a2 (mod p2)
        .
        .
        .
        cn = an (mod pn)

    Retorna como resultado el siguiente JSON:

    {
        "pasos": [
            {
                "pasoLatex": str,
                "descripcion": str
            }
            {
            ...
            }
        ]
    }
    """
    resultado = [ecuaciones]
```

```
30
31      # Comprobamos que todos los números son enteros
32      for i in range(len(ecuaciones)):
33          if type(ecuaciones[i][0]) != int or
            ↪  type(ecuaciones[i][1]) != int or
            ↪  type(ecuaciones[i][2]) != int:
34              raise MyException("Los elementos en la ecuación " +
                ↪  str(i) + " tienen que ser enteros")
35
36      # Comprobamos que los módulos son mayores que 1
37      for i in range(len(ecuaciones)):
38          if ecuaciones[i][2] <= 1:
39              raise MyException("El módulo en la ecuación " +
                ↪  str(i) + " tiene que ser mayor que 1")
40
41      # Comprobamos si los módulos son primos entre sí
42      numeros = []
43      for ecuacion in ecuaciones:
44          numeros.append(ecuacion[2])
45
46      resultado.append(numeros)
47      combinaciones = list(combinations(numeros, 2))
48
49      for pares in combinaciones:
50          i = pares[0]
51          j = pares[1]
52          mcdLocal = mcd([i], [j])["resultado"]
53          if mcdLocal != "1":
54              raise MyException("Los módulos no son primos entre
                ↪  sí, no se puede aplicar el teorema chino del
                ↪  resto")
55
56      # Quitamos los coeficientes de las x
57      newEcuaciones = []
58      for ecuacion in ecuaciones:
59          if ecuacion[0] == 1:
60              newEcuaciones.append(ecuacion)
61          else:
62              newInverso = int(inverso(ecuacion[0],
                ↪  ecuacion[2])["resultado"])
63              multiplicacion = (newInverso * ecuacion[1]) %
                ↪  ecuacion[2]
64              newEcuacion = [1, multiplicacion, ecuacion[2]]
65              newEcuaciones.append(newEcuacion)
66      ecuaciones = newEcuaciones
67      resultado.append(newEcuaciones)
68
```

```
69      # Aplicamos el teorema
70      pFinal = 1
71      for ecuacion in ecuaciones:
72          pFinal = pFinal * ecuacion[2]
73      resultado.append(pFinal)
74
75      q = []
76      for ecuacion in ecuaciones:
77          n = (pFinal / ecuacion[2]) % ecuacion[2]
78          inversoLocal = int(inverso(n, ecuacion[2])["resultado"])
79          q.append(inversoLocal)
80      resultado.append(q)
81
82      xFinal = 0
83      for i in range(len(ecuaciones)):
84          xFinal += int(ecuaciones[i][1] * q[i] * (pFinal /
            ↪  ecuaciones[i][2]))
85      resultado.append(xFinal)
86
87      xModulo = int(xFinal) % pFinal
88      resultado.append(xModulo)
89
90      return pasos.sistema(resultado) # resultado = [ecuaciones,
        ↪  numeros, newEcuaciones, pFinal, q, xFinal, xModulo]
```

La función para resolver sistemas lineales de congruencias mediante el teroema chino del resto se llama *sistema*, que recibe un array de ecuaciones como parámetro. Dichas ecuaciones son otros arrays de la forma *[c, a, p]* que corresponde con una ecuación del tipo $c = a$ *(mod* p*)*.

Primero mediente un bucle se comprueba que todos los elementos de las ecuaciones son enteros y que todos los módulos sean mayores que 1, en caso contrario se lanza una excepción. A continuación se comprueba la condición para aplicar el teorema chino de resto, es decir, que los módulos sean primos entre sí. Para ello se usa la función *combinations* de la librería *itertools*.

El siguiente paso es despejar las ecuaciones para que sean de la forma $x = a$ (mod p) de una manera similar a como se hacia en la función *ecuacion* (pasando la c a la derecha de la ecuación calculando su inverso). Este proceso se realiza en un bucle para que sea aplicado a todas las ecuaciones del sistema.

En este momento tenemos las ecuaciones del sistema de la forma $x = a$ (mod p) por lo que se procede a aplicar el teorema chino del resto. para ello calculamos el módulo que va a tener la solución multilicando todos los módulos de las ecuaciones (se va a guardar en la variable *pFinal*). A continuación se aplica el algoritmo para calcular la *q* correspondiente a cada ecuación de la forma en la que se explica en la teoría.

Para obtener el resultado final se realiza un bucle para realizar las sucesivas sumas de $a_i q_i \frac{pFinal}{p_i}$ para cada una de las ecuaciones. Esta suma se guarda en la

variable *xFinal* al que finalmente se le aplica el módulo *pFinal*, guardándolo en la variable *xModulo*.

Finalmente todos las variables se recogen en un array para constuir los pasos del algoritmo.

Función para devolver los pasos

```
def sistema(resultado):
    json = {}
    pasos = []
    modulos = ""
    solucionPLatex = ""
    paramQLatex = ""
    divisiones = []
    sistemaLatex = "\\begin{cases}"
    solucionLatex = ""

    # Añadimos si se ha simplificado el sistema o no
    if resultado[0] != resultado[2]: # Las ecuaciones se han simplificado
        ecuacionesLatex = ""
        i = 1
        for ecuacion in resultado[2]:
            ecuacionesLatex += "x \\equiv " + str(ecuacion[1]) + " \ (\mathrm{mod}\ " + str(ecuacion[2]) + ")\\\\"
            i += 1
        ecuacionesLatex = ecuacionesLatex[:-2]
        pasos.append({"pasoLatex": "\\begin{cases} " + ecuacionesLatex + " \\end{cases}", \
                      "descripcion": "Simplificando las ecuaciones tenemos el siguiente sistema:"})

    # Añadimos la comprobación de si son primos entre sí
    for i in range(len(resultado[1])):
        modulos += str(resultado[1][i])
        if (i != len(resultado[1]) - 2) and (i != len(resultado[1]) - 1):
            modulos += ", "
        elif (i == len(resultado[1]) - 2):
            modulos += " y "

    pasos.append({"descripcion": "Puesto que " + modulos + " son primos entre sí, se puede aplicar el teorema chino del resto."})

```

```
32      # Añadimos la solución única en Zp
33      for i in range(len(resultado[1])):
34          solucionPLatex += str(resultado[1][i])
35          if (i != len(resultado[1]) - 1):
36              solucionPLatex += " \\cdot "
37
38      pasos.append({"descripcion": "El sistema tiene solucion
        ↪  única en $\\mathbb{Z}_p$, con $p = " + solucionPLatex +
        ↪  " = " + str(resultado[3]) + "$"})
39
40      # Cadena con la solución en función de q
41      paramQLatex += "x = ("
42      for i in range(len(resultado[2])):
43          paramQLatex += str(resultado[2][i][1]) + "q_" + str(i+1)
            ↪  + " \\cdot \\frac{" + str(resultado[3]) + "}{" +
            ↪  str(resultado[2][i][2]) + "}"
44          if (i != len(resultado[2]) - 1):
45              paramQLatex += " + "
46      paramQLatex += ") = ("
47      for i in range(len(resultado[2])):
48          division = int(resultado[3] / resultado[2][i][2])
49          divisiones.append(division)
50          paramQLatex += str(resultado[2][i][1]) + "q_" + str(i+1)
            ↪  + " \\cdot " + str(division)
51          if (i != len(resultado[2]) - 1):
52              paramQLatex += " + "
53      paramQLatex += ") \ (\mathrm{mod}\ " + str(resultado[3]) +
        ↪  ")"
54
55      pasos.append({"pasoLatex": paramQLatex, \
56                    "descripcion": "Representación de la solución
                      ↪  en función de q"})
57
58      # Añadimos el sistema para calcular las q
59      for i in range(len(resultado[2])):
60          auxLatex = ""
61          auxLatex += "q_" + str(i+1) + " \\cdot " +
            ↪  str(divisiones[i]) + " \\equiv 1 \ (\mathrm{mod}\ "
            ↪  + str(resultado[2][i][2]) + ") \\Rightarrow "
62          divisionMod = divisiones[i] % resultado[2][i][2]
63          if (divisionMod != 1):
64              auxLatex += str(divisionMod)
65          auxLatex += "q_" + str(i+1) + " \\equiv 1 \
            ↪  (\mathrm{mod}\ " + str(resultado[2][i][2]) + ")
            ↪  \\Rightarrow q_" + str(i+1) + " = " +
            ↪  str(resultado[4][i])
66          sistemaLatex += auxLatex + "\\\\"
```

```
67      sistemaLatex = sistemaLatex[:-2] + "\\end{cases}"
68
69      pasos.append({"pasoLatex": sistemaLatex, \
70                    "descripcion": "Sistema y pasos para calcular
                       ↪  cada una de las q"})
71
72      # Añadimos el cálculo de la solución
73      solucionLatex += "x \\equiv ("
74      for i in range(len(resultado[2])):
75          solucionLatex += str(resultado[2][i][1]) + " \\cdot " +
             ↪  str(resultado[4][i]) + " \\cdot " +
             ↪  str(divisiones[i])
76          if (i != len(resultado[2]) - 1):
77              solucionLatex += " + "
78      solucionLatex += ") = ("
79      for i in range(len(resultado[2])):
80          multiplicacion = int(resultado[2][i][1] *
             ↪  resultado[4][i] * divisiones[i])
81          solucionLatex += str(multiplicacion)
82          if (i != len(resultado[2]) - 1):
83              solucionLatex += " + "
84      solucionLatex += ") = " + str(resultado[5]) + " \\equiv " +
         ↪  str(resultado[6]) + "\ (\mathrm{mod}\ " +
         ↪  str(resultado[3]) + ")"
85
86      pasos.append({"pasoLatex": solucionLatex, \
87                    "descripcion": "Cálculo de la solución del
                       ↪  sistema"})
88
89      pasos.append({"pasoLatex": "x = " + str(resultado[6]) + "\
         ↪  (\mathrm{mod}\ " + str(resultado[3]) + ")",
90                    "descripcion": "Solución del sistema"})
91
92      json["pasos"] = pasos
93      return json
```

La función *sistema* para devolver los pasos de la resolución de sistemas de congruencias mediante el teorema chino del resto toma como parámetro un array llamado *resultado* que contiene las ecuaciones iniciales, los módulos de las ecuaciones, unas nuevas ecuaciones en caso de que el sistema haya tenido que ser simplificado, el módulo final de la solución, las ecuaciones para resolver las q_i y el resultado.

Lo primero que hace el algoritmo es añadir un paso en el caso de que se haya tenido que simplificar el sistema, recorriendo con un bucle las diferentes ecuaciones y concatenándolas para dar lugar al nuevo sistema. El siguiente bucle recorre los módulos de las ecuaciones y crea un paso explicando la existencia de la solución, añadiendola en una cadena de texto. A continuación, se recorren

las ecuaciones de nuevo para representar la solución multiplicando cada uno de los módulos.

La siguiente parte del código utiliza bucles y concatenaciones de cadenas de texto para explicar el comienzo del cálculo de la solución según se vio el el marco teórico. De esta forma se representa la solución en función de las diferentes q_i, cuya resolución se muestra a continuación creando las diferentes ecuaciones para despejar dichas variables q.

La última parte del código crea una cadena de texto donde se sustituyen las diferentes q_i que se han obtenido en la solución expuesta previamente, añadiendo a su vez la solución en la $\mathbb{Z}$ que se ha determinado inicialmente en el problema.

3.2. Librerías y funciones auxiliares

3.2.1. Librerías

Para el desarrollo del código se han usado tres librerías externas. La primera de ellas es la librería *sympy* que se ha usado en el cálculo de los números primos de la función *esPrimo* [3]. Esta librería se ha usado principalmente porque ya tiene una función para comprobar si un número es primo, siendo esta una función optimizada para números elevados.

La segunda librería es *itertools* en la que se ha usado el método *combinations* [4]. Este método se usa únicamente en la resolución de sistemas lineales de congruencias mediante el teorema chino del resto, en la que la condición para la existencia de solución es que los módulos sean primos entre sí. Usando *combinations* generamos todas las combinaciones de pares posibles para poder comprobar si son primos dos a dos.

La tercera y última librería es la llamada *Fractions* [5] que se ha usado en el fichero de *operaciones_polinomios.py* para poder manejar fácilmente fracciones cuando están presentes en las operaciones.

3.2.2. Fichero *operaciones_polinomios.py*

División de polinomios

```
1  def dividir_polinomios(dividendo, divisor):
2      cociente = []
3
4      while len(dividendo) >= len(divisor):
5
6          cocienteAux = Fraction(dividendo[0], divisor[0])
7          cociente.append(cocienteAux)
8
9          temporal = []
10         for i in range(len(divisor)):
11             temporal.append(cocienteAux * Fraction(divisor[i]))
12         newCeros = len(dividendo) - len(temporal)
```

```
13          for i in range(newCeros):
14              temporal.append(0)
15          newDividendo = []
16          for i in range(len(temporal)):
17              newDividendo.append(Fraction(dividendo[i] -
                ↪  temporal[i]))
18          dividendo = newDividendo
19          while len(dividendo) >= 1 and dividendo[0] == 0:
20              dividendo.pop(0)
21
22      resto = dividendo
23
24      while len(resto) > 0 and resto[0] == 0:
25          resto.pop(0)
26
27      if resto == []:
28          resto.append(0)
29
30      return cociente, resto
```

Esta función se usa para realizar divisiones de polinomios tomando como parámetros un *dividendo* y un *divisor*. Como se ha visto, dichos parámetros son arrays de enteros en el que cada número corresponde con un coeficiente de una expresión matemática (por ejemplo, el array $[3, -6, 1, 4]$ correspondería con la expresión $3x^3 - 6x^2 + x + 4$).

En esta función se itera con un *while* mientras se cumpla la condición de que la longitud del array del dividendo sea mayor o igual que el del divisor. En cada iteración se calcula el cociente de la división entre el elemento con mayor exponente del dividendo entre el mayor exponente del divisor. Una vez se tiene este cociente, se va multiplicando por cada coeficiente de divisor, añadiendo la expresión resultante un un array temporal, añadiendo con ceros las posiciones que no se han llenado. En este momento se realiza una resta de polinomios entre el dividendo y el array temporal, dando resultado a una nueva expresión.

Repitiendo este proceso y eliminando con otro bucle todos los ceros que se encuentran a la izquierda del resto, se da como resultado un cociente y un resto que se devolverán en el *return*.

Resta de polinomios

```
1  def restar_polinomios(a, b):
2      resultado = [0] * max(len(a), len(b))
3      for i in range(len(a)):
4          resultado[i] += a[i]
5      for i in range(len(b)):
6          resultado[i] -= b[i]
7      while len(resultado) > 1 and resultado[-1] == 0:
8          resultado.pop()
```

```
9       return resultado
```

Esta función se usa para realizar una resta de polinomios pasando como parámetros a y b, dos arrays de números que corresponden con sus coeficientes.

Primero se crea una variable *resultado* que va a ser un array con la misma longitud que el array de a, sólo que todo ceros. Luego iteramos sobre b, restando cada coeficiente a los coeficientes de *resultado*. Por último se eliminan todos los elementos que sean ceros empezando por la izquierda.

Multiplicación de polinomios

```
1  def multiplicar_polinomios(a, b):
2      resultado = [0] * (len(a) + len(b) - 1)
3      for i in range(len(a)):
4          for j in range(len(b)):
5              resultado[i + j] += a[i] * b[j]
6      while len(resultado) > 1 and resultado[-1] == 0:
7          resultado.pop()
8      return resultado
```

Esta función se usa para multiplicar polinomios usando los parámetros a y b, dos arrays de números que corresponden con sus coeficientes.

Primero creamos un array de ceros con una longitud igual a la longitud de a más la longitud de b menos 1. A continuación, se itera de manera simultánea sobre a y b, multiplicando sus posiciones *i* y *j* y sumándolo al array de resultado. Por último se eliminan todos los elementos que sean ceros empezando por la izquierda.

3.2.3. Fichero *Expression.py*

```
1  class Expression:
2      def __init__(self, coefs: list):
3          self.coefs = coefs
4
5      def toExpression(self):
6          exponente = len(self.coefs) - 1
7          expresion = ""
8          for i in range(len(self.coefs)):
9              if self.coefs[i] != 0:
10                 expresion += " + " + str(self.coefs[i]) + "x^" +
                       ↪  str(exponente)
11             exponente -= 1
12         expresion = expresion[3:]
13         expresion = expresion.replace("x^0", "")
14         expresion = expresion.replace("1x", "x")
15         expresion = expresion.replace("x^1", "x")
16         expresion = expresion.replace("+ -", "- ")
17         if expresion == "":
```

```
18                 expresion += "0"
19             return expresion
```

La clase *Expression* sirve para convertir un array de coeficientes en una expresión matemática con sus variables. Esta clase tiene un método llamado *toExpression*, que se usa para realizar dicha conversión.

Lo primero que hace el método es guardar en la variable *exponente* el exponente de la expresión, cuyo valor va a ser la longitud del array de coeficientes menos 1. A continuación, se itera sobre los coeficientes, concatenando en cada iteración el valor del coeficiente con un '+' y una 'x' elevado a una potencia (dependiendo de su posición en el array).

Por último se modifica el resultado eliminando las partes del *string* que quedan incoherentes o innecesarias, como x^0, sustituir $1x$ por x o x^1 por x.

3.2.4. Fichero *MyException.py*

```
1 class MyException(Exception):
2     pass
```

La clase *MyException* se usa en las funciones del fichero del código principal para lanzar excepciones cuando no se cumplen con unas condiciones. Dicha excepción va acompañada de un texto explicativo.

Capítulo 4

Resultados

En este capítulo se van a exponer los pasos que devuelven los algoritmos mediante varios ejemplos. Cada una de las secciones de este capítulo representa una función del código de Python y que contiene uno o dos ejemplos de su llamada para visualizar la respuesta.

Como se ha mencionado en el capítulo 3 sobre el código, una vez los algoritmos resuelven los problemas y obtienen una solución, llevan esa solución a un código adicional para construir los pasos. Dichos pasos tienen un formato JSON similar a la siguiente forma:

```
{
    "pasos":
    [
        {
            "pasoLatex": str,
            "descripción": str
        }
    ]
}
```

Para facilitar el entendimiento del lector, se ha descrito cada paso en un formato que sea comprensible, añadiendo finalmente una captura de lo que se obtiene en la web de la aplicación didáctica que se ha mencionado a lo largo de este TFG.

4.1. Máximo común divisor

Ejemplo 1: Llamada a la función $\mathrm{mcd}([122],[18])$ para el cálculo del máximo común divisor entre 122 y 18.

En primer lugar obtendríamos una secuencia de los sucesivos cálculos realizados mediante el teorema de Euler de la siguiente forma:

$\mathrm{mcd}(122,18) = \mathrm{mcd}(18,14) = \mathrm{mcd}(18,14) = \mathrm{mcd}(14,4) = \mathrm{mcd}(14,4) = \mathrm{mcd}(4,2) = \mathrm{mcd}(4,2) = \mathrm{mcd}(2,0)$

Finalmente se añade una variable más al JSON de respuesta solamente con la solución, es decir, tendríamos lo siguiente:

```
{
    "resultado": "2"
}
```

Este paso es necesario ya que se va a tener que recuperar el resultado del máximo común divisor en algunos de las siguientes algoritmos.

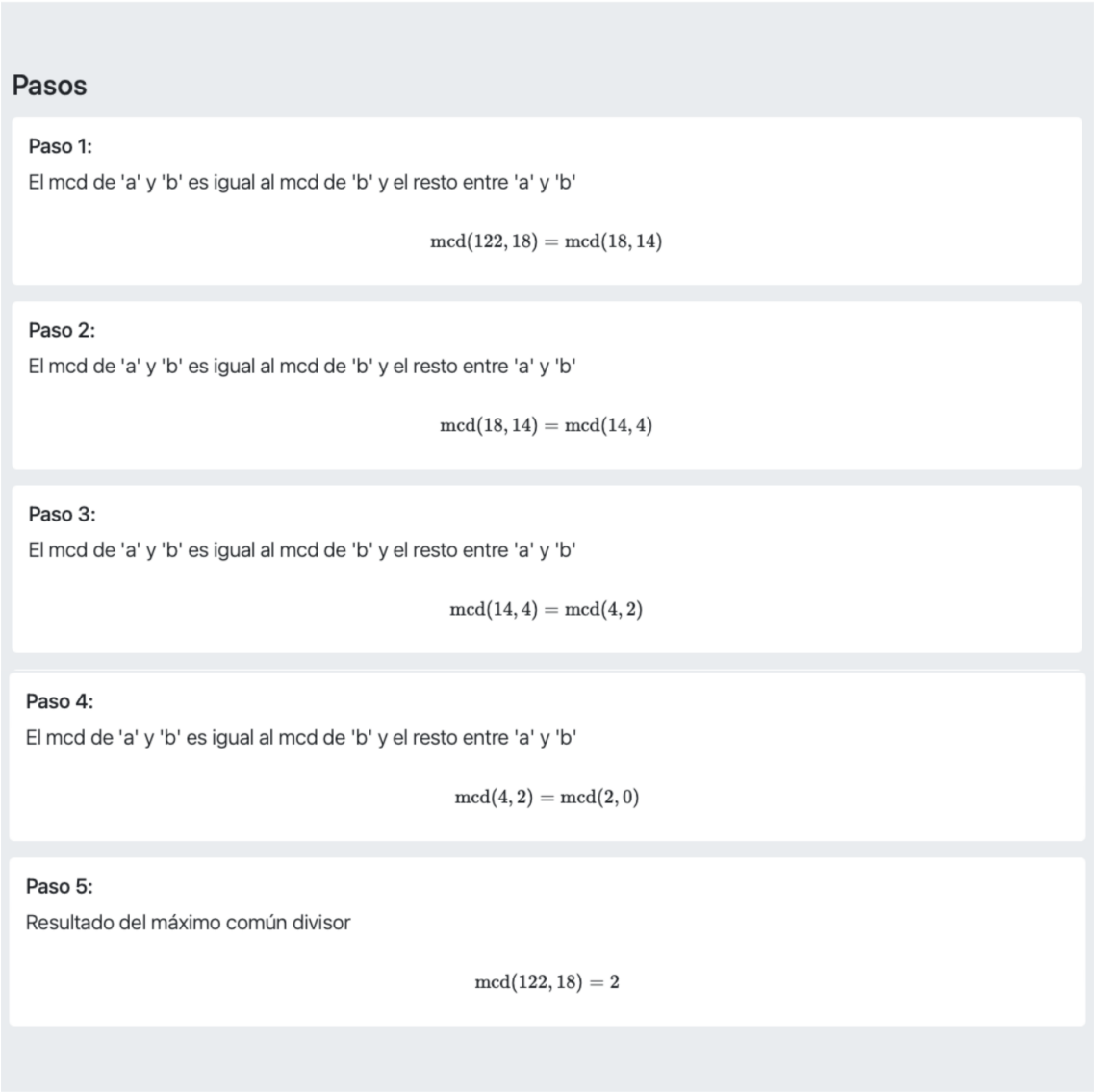

Figura 4.1: Resultado en la web para $\mathrm{mcd}([122], [18])$

Ejemplo 2: Llamada a la función $\mathrm{mcd}([1, 4, -5, 1], [1, -3, 1])$ para el cálculo del máximo común divisor de los polinomios $x^3 + 4x^2 - 5x + 1$ y $x^2 - 3x + 1$.

En este caso el resultado va a estar en el mismo formato que en el ejemplo 1, por lo que obtendremos esta secuencia:

$\mathrm{mcd}(x^3+4x^2-5x+1, x^2-3x+1) = \mathrm{mcd}(x^2-3x+1, 15x-6) = \mathrm{mcd}(x^2-3x+1, 15x-6) = \mathrm{mcd}(15x-6, -1/25) = \mathrm{mcd}(15x-6, -1/25) = \mathrm{mcd}(-1/25, 0)$

También obtendremos una última variable con el resultado final para su posterior uso:


```
{
    "resultado": "-1/25"
}
```


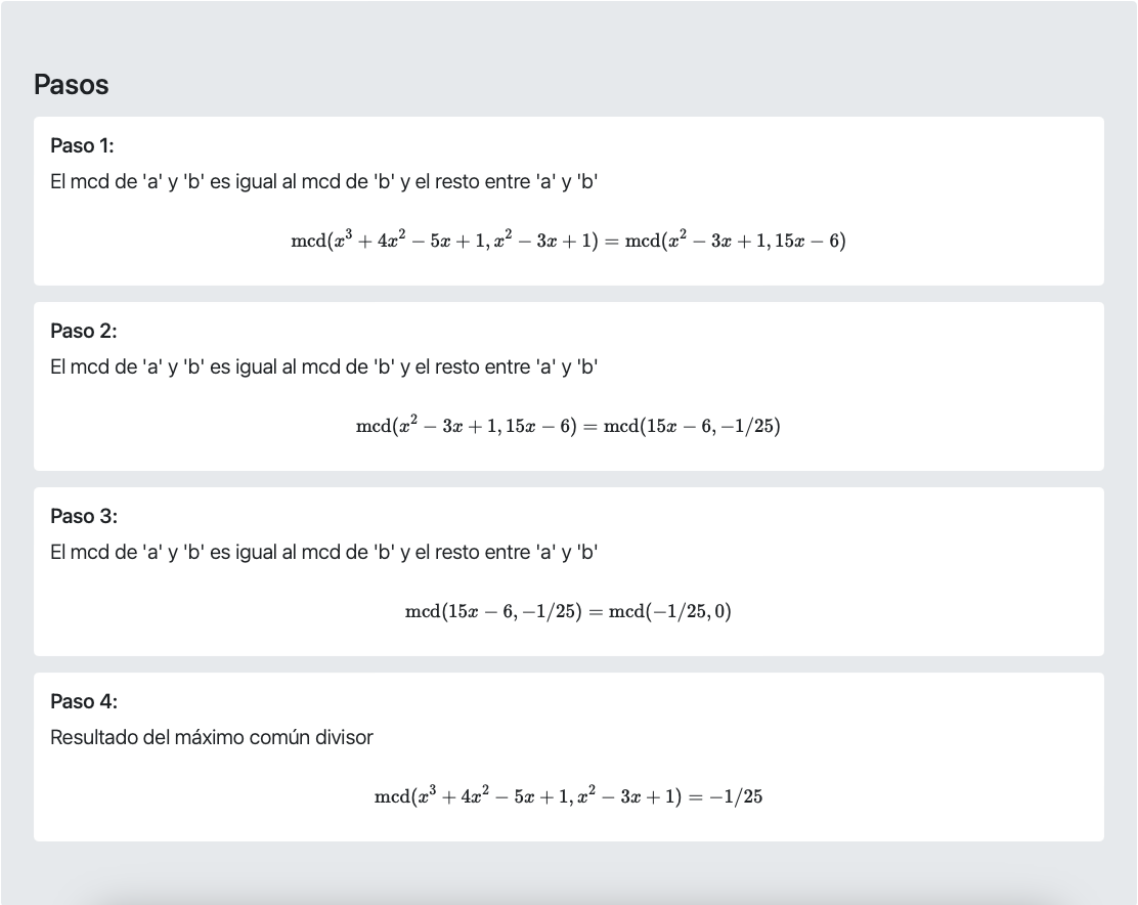

Figura 4.2: Resultado en la web para $\mathrm{mcd}([1, 4, -5, 1], [1, -3, 1])$

4.2. Teorema de Bezout

Ejemplo 1: Llamada a la función $bezout([122], [18])$ para calcular los coeficientes de Bezout para $122m + 18n = d$ con $d = \mathrm{mcd}(122, 18)$.

Para este tipo de problemas, la solución se expone describiendo los elementos de las columnas y de las filas que se han usado para la resolución del algoritmo

según se ha descrito en el capítulo del marco teórico. Cada paso se ha escrito en formato Latex para construir una tabla, resultando de la siguiente forma:

122		$\mathbf{v}_0 = (1, 0)$
18	6	$\mathbf{v}_1 = (0, 1)$
14	1	$\mathbf{v}_2 = (1, -6)$
4	3	$\mathbf{v}_3 = (-1, 7)$
2	2	$\mathbf{v}_4 = (4, -27)$
0		

A continuación tendíamos otro paso en el que nos da la comprobación de la solución del problema de la siguiente forma:

$$(122) \cdot (4) + (18) \cdot (-27) = 2$$

Finalmente obtenemos dos variables más que son, respectivamente, el valor de la m y el valor de la n que se van a usar posteriormente en la resolución de las ecuaciones diofánticas lineales. Para este ejemplo se obtendría:


```
{
    "m": "4"
}
```

```
{
    "n": "-27"
}
```

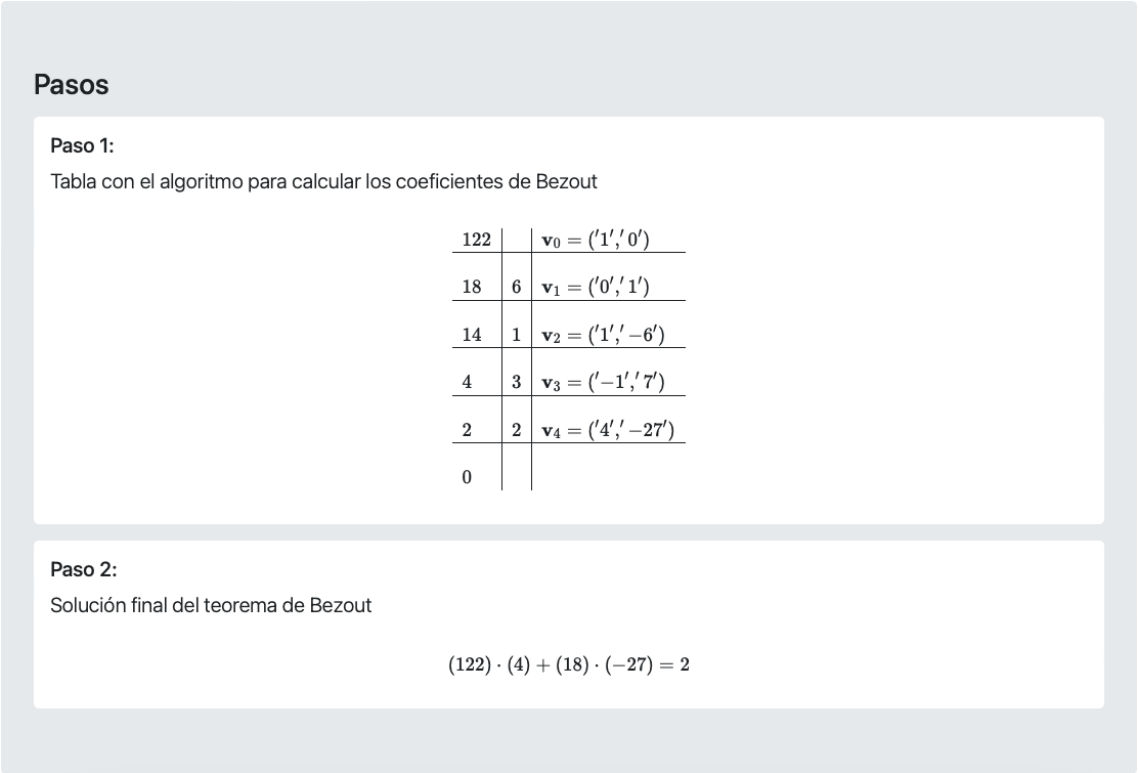

Figura 4.3: Resultado en la web para $bezout([122], [18])$

Ejemplo 2: Llamada a la función $bezout([1, 5, 7], [1, 2])$ para calcular los coeficientes de Bezout para $(x^2 + 5x + 7)m + (x + 2)n = d$ con $d = \text{mcd}(x^2 + 5x + 7, x + 2)$.

En este caso de llama a la misma función pero para el cálculo con polinomios. De la misma forma que en el ejemplo anterior, los pasos se dan describiendo las columnas de la tabla resultante:

$x^2 + 5x + 7$		$\mathbf{v}_0$ = (1, 0)
$x + 2$	$x + 3$	$\mathbf{v}_1$ = (0, 1)
1	$x + 2$	$\mathbf{v}_2$ = (1, $-x - 3$)
0		

Por lo tanto en el siguiente paso tendríamos:

$(x^2 + 5x + 7) \cdot (1) + (x + 2) \cdot (-x - 3) = 1$

Finalmente las últimas dos entradas del JSON que corresponden con los valores m y n serían:

```
{
    "m": "1"
}

{
    "n": "-x - 3"
```

}

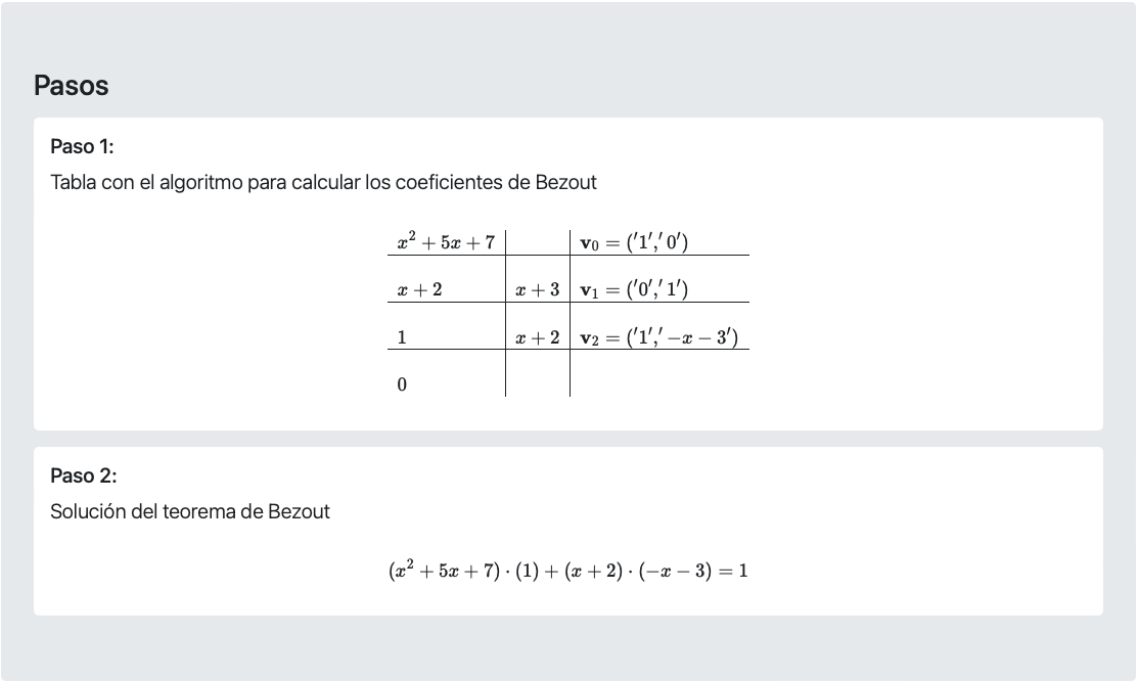

Figura 4.4: Resultado en la web para $bezout([1,5,7],[1,2])$

4.3. Divisibilidad

Ejemplo 1: Llamada a la función $divisible(245,7)$ para comprobar si el número 245 es divisible entre 7.

En este caso se devuelve un JSON únicamente con dos pasos. EL primero de ellos contiene la explicación del criterio de divisibilidad del 7 por lo que se devolvería:

Para saber si un número es divisible entre 7 hacemos lo siguiente: se debe multiplicar el último dígito por 2 y restarlo al número que conforman los demás dígitos. Esto, hasta que queda un número de solo un dígito. Si este es un 0 o un 7, entonces el número es divisible entre 7.

El segundo paso simplemente dice si un número es divisible entre otro, por lo tanto en este ejemplo tenemos la siguiente respuesta:

245 es divisible por 7.

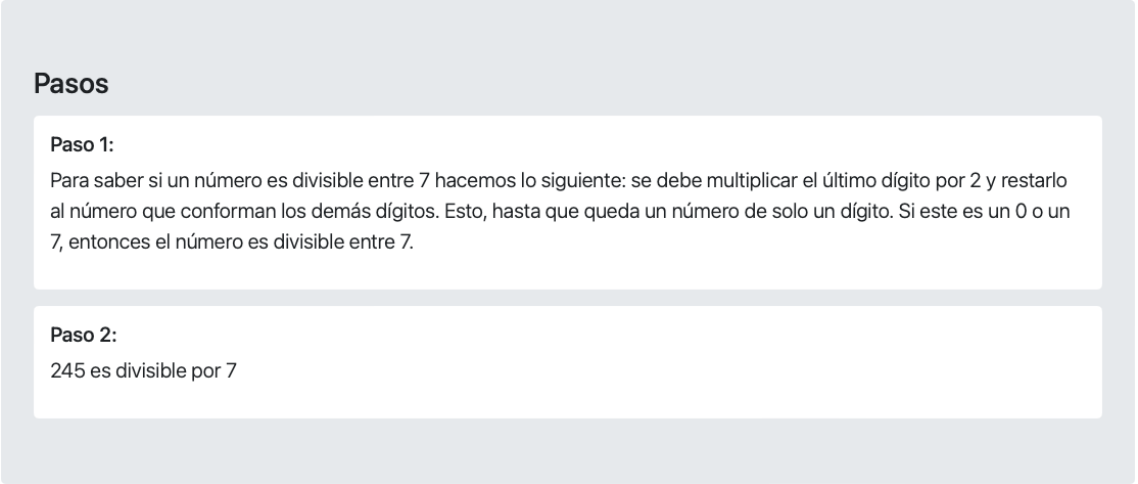

Figura 4.5: Resultado en la web para $divisible(245, 7)$

Ejemplo 2: Llamada a la función $divisible(5, 3)$ para comprobar si el número 5 es divisible entre 3.

Este caso sería igual que el anterior ejemplo. Se devuelve en primer lugar el criterio de divisibilidad del 3 y luego si 5 es divisible entre 3:

Un número es divisible entre 3 si la suma de sus dígitos es igual a 3 o a un múltiplo de 3.

5 no es divisible por 3.

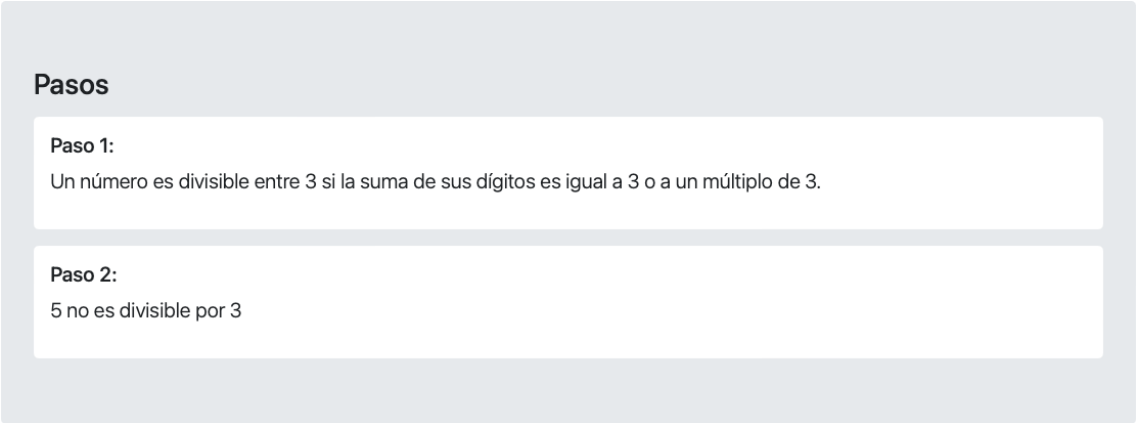

Figura 4.6: Resultado en la web para $divisible(5, 3)$

4.4. Ecuaciones diofánticas sin restricciones

Ejemplo 1: Llamada a la función $ecDiofantica(165, 48, -6)$ para obtener las soluciones a la ecuación $165m + 48n = -6$.

La resolución de este problema comenzaría con un paso exponiendo si la ecuación tiene o no solución. En este caso, al tener solución, el primer paso sería:

La ecuación tiene solución.

A continuación, en el segundo paso se tendría la explicación de por qué la ecuación tiene solución:

Se verifica que el mcd(165, 48) = 3 divide a -6

El algoritmo que imprime los pasos se detendría en este punto en el caso de que la ecuación diofántica no tuviese solución. En este caso, como la solución es existente, se continuaría con el cálculo de los coeficientes de Bezout:

Aplicando el teorema de Bezout para 165 y 48 obtenemos m=7 y n=-24

Tomando entonces los coeficiente de Bezout y aplicando el algoritmo descrito en el marco teórico respecto a la resolución de ecuaciones diofánticas lineales se obtendría la siguiente solución:

Las soluciones a la ecuación son x = -14 + 16t, y = 48 - 55t

Por último, el JSON con los pasos contendría una entrada adicional llamada "valores"que consta de un array con los valores de los coeficientes de la solución (en este caso $[-14, 48, 16, -55]$). Estos valores son necesarios a la hora de resolver ecuaciones diofánticas con restricciones.

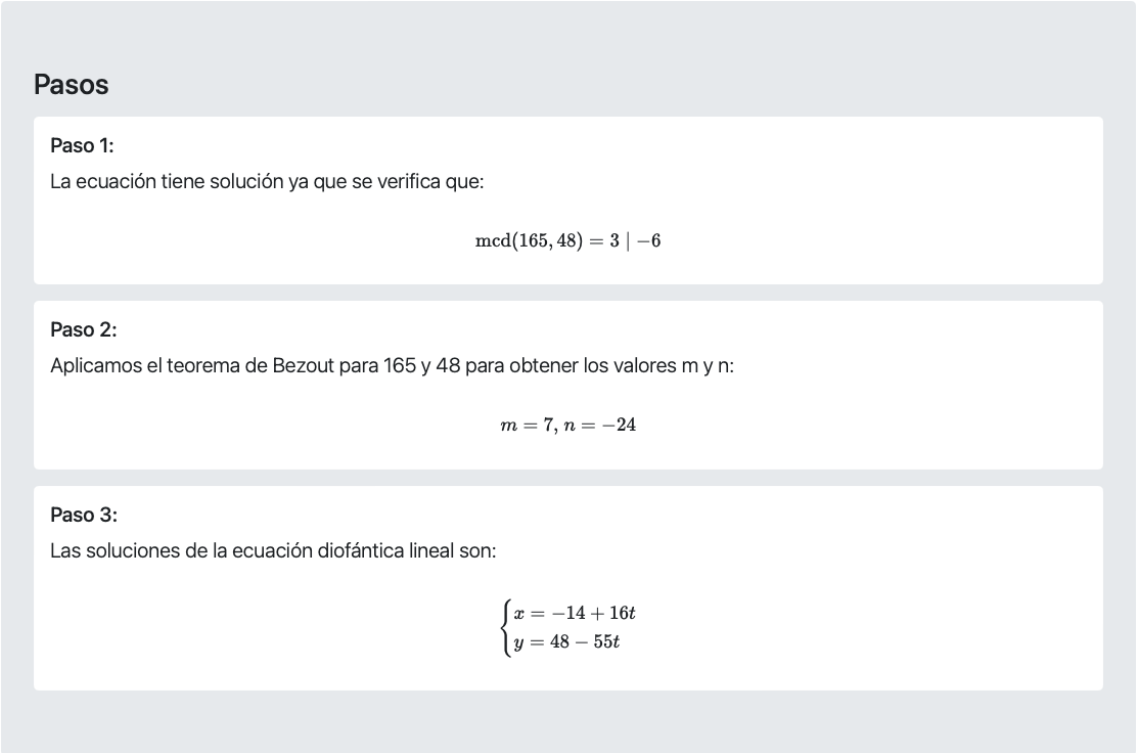

Figura 4.7: Resultado en la web para $ecDiofantica(165, 48, -6)$

4.5. Ecuaciones diofánticas con restricciones

Ejemplo 1: Llamada a la función $ecDiofantica\_res(19, 5, 100, [0, None, None, None], [0, None, None, None])$ para obtener las soluciones a la ecuación $19m + 5n = 100$ restringido a $x > 0$ e $y > 0$.

El algoritmo comienza con la resolución de la ecuación diofántica lineal ignorando las restricciones, por lo que los primeros pasos los mismos pasos que en las ecuaciones diofánticas lineales sin restricciones:

La ecuación tiene solución ya que se verifica que el mcd(19, 5) = 1 divide a 100

Aplicando el teorema de Bezout obtenemos m = -1 y n = 4

La ecuación diofántica tiene como solución: x = -100 + 5t, y = 400 - 19t con t perteneciente a Z.

A continuación se añadiría un paso exponiendo los intervalos de t en los que existe una solución, teniendo:

t >20, t <= 21.

Por último, teniendo en cuenta esos intervalos para t, el algoritmo comprueba si las soluciones son infinitas o finitas. En este caso, al tener soluciones finitas se añadiría el siguiente paso con las soluciones para x e y:

x = 5, y = 1

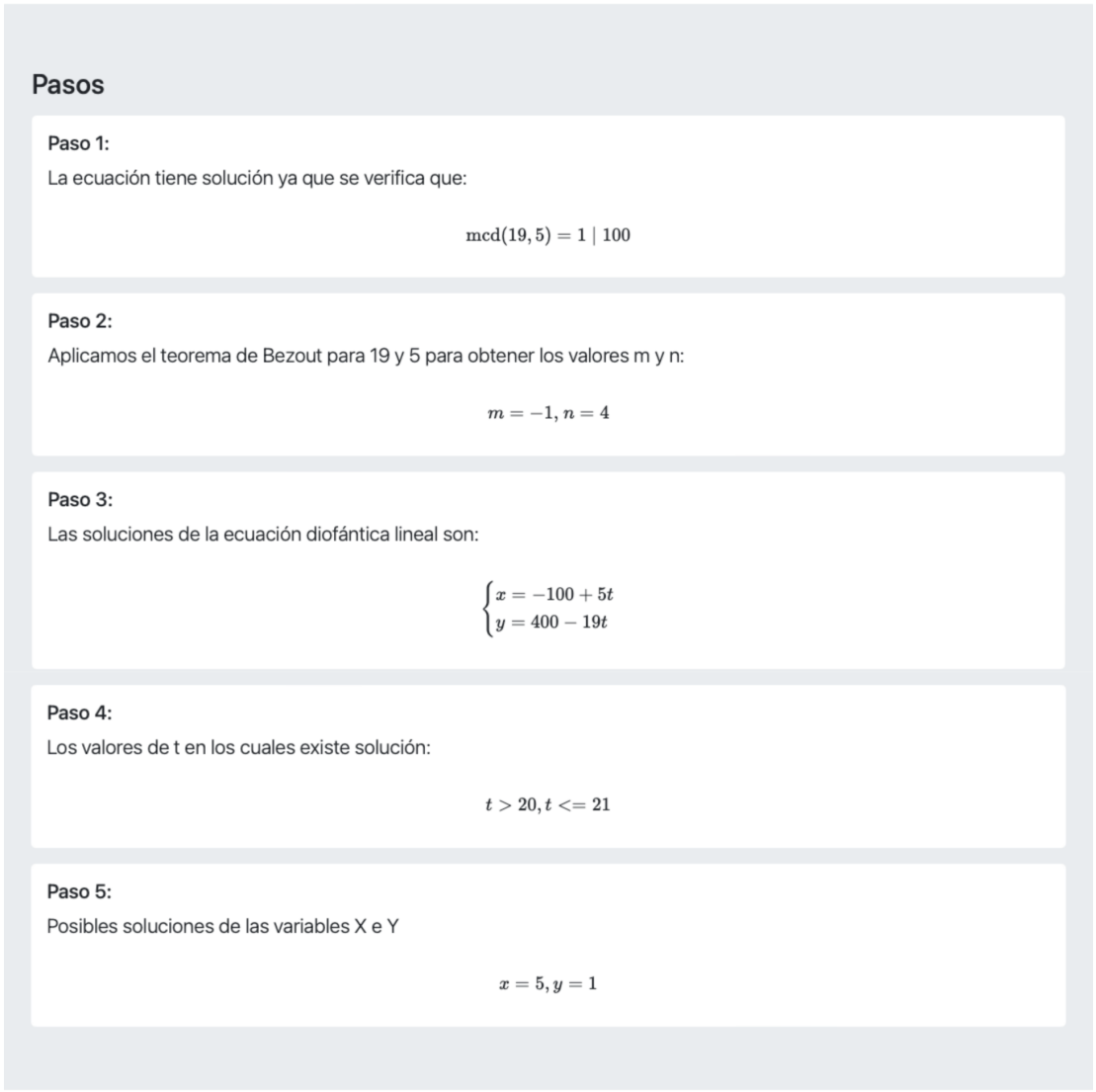

Figura 4.8: Resultado en la web para $ecDiofantica\_res(19, 5, 100, [0, None, None, None], [0, None, None, None])$

4.6. Números primos

Ejemplo 1: Llamada a la función $esPrimo(365)$ para comprobar si el número 365 es un número primo.

La respuesta de los pasos para comprobar si un número es primo es muy sencilla siendo de la siguiente forma:

El número 365 no es primo, es compuesto.

Por último se añadiría una variable al JSON con un booleano que respresenta si el número es primo. Este paso no se imprime pero se usa para otros cálculos,

siendo de la siguiente forma:

```
{
    "esPrimo": False
}
```

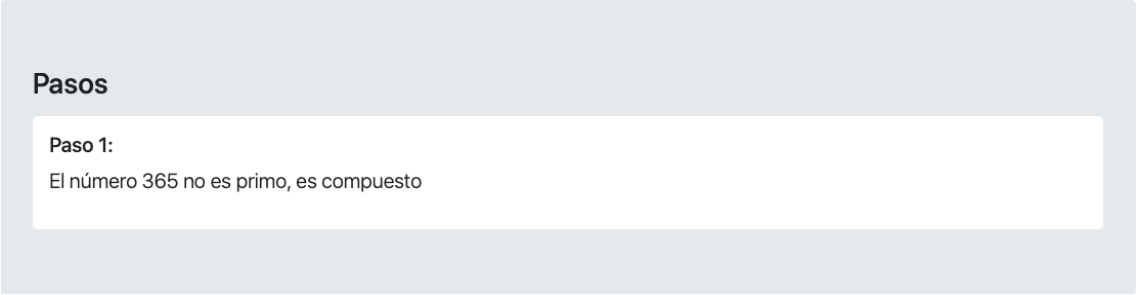

Figura 4.9: Resultado en la web para $esPrimo(365)$

Ejemplo 2: Llamada a la función $esPrimo(11)$ para comprobar si el número 11 es un número primo.

En este caso, al ser el 11 un número primo obtendríamos la siguiente respuesta:

El número 11 es primo.

AL igual que el anterior ejemplo, se añade una variable adicional al JSON con la respuesta.

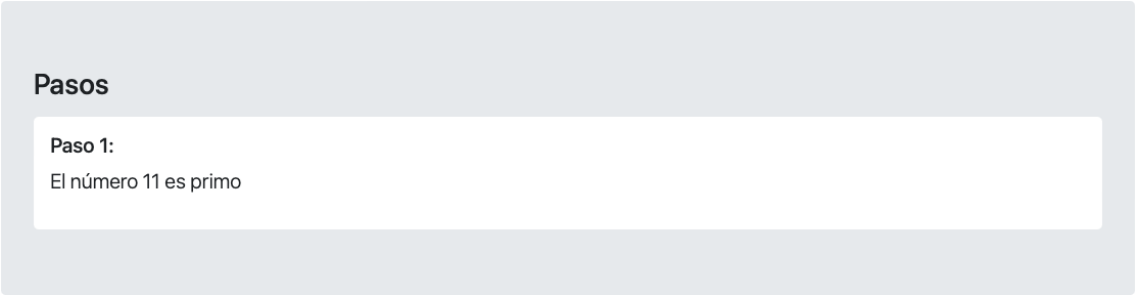

Figura 4.10: Resultado en la web para $esPrimo(11)$

4.7. Teorema fundamental de la aritmética

Ejemplo 1: Llamada a la función $tfAritmetica(100)$ para comprobar si el número 100 se puede descomponer en factores primos.

Esta función devuelve únicamente un paso, que sería la descomposición en factores primos del número pasado como parámetro. En nuestro caso, el paso que se devuelve es el siguiente:

La descomposición en factores primos de 100 es: $100 = 2^2 * 5^2$.

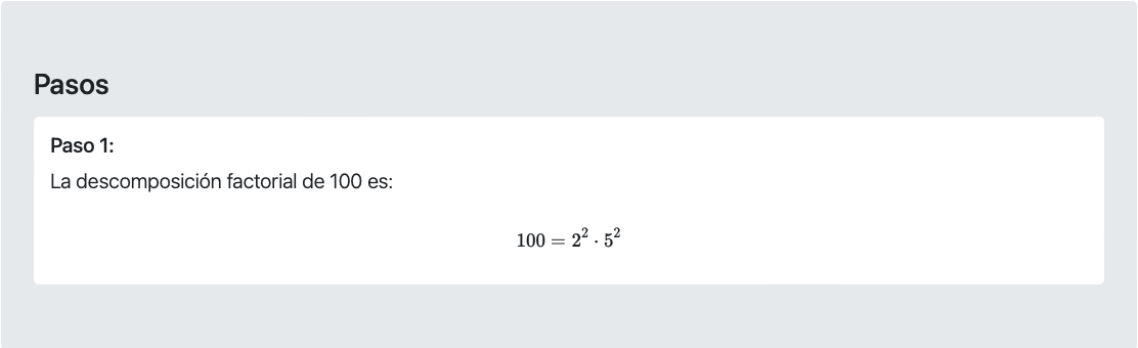

Figura 4.11: Resultado en la web para $tfAritmetica(100)$

Ejemplo 2: Llamada a la función $tfAritmetica(89)$ para comprobar si el número 89 se puede descomponer en factores primos.

Al ser el número 89 un número primo, el único paso que se devuelve es el siguiente:

El número 89 es primo, por lo que no tiene descomposición.

Figura 4.12: Resultado en la web para $tfAritmetica(89)$

4.8. Sistemas de numeración en base b

Ejemplo 1: Llamada a la función $base(26, 16)$ para representar el número 26 en base 16.

El primer paso que devuelve la función es el abecedario que se ha usado en la representación (en caso de tener letras). Por lo tanto en este ejemplo se tendría:

Los valores del abecedario que se han usado son: A = 10.

El segundo paso que devuelve es la representación del número dado, en este caso 26, en base 16. El paso contendría el siguiente texto:

El número 26 en base 16 es igual a 1A

Figura 4.13: Resultado en la web para $base(26, 16)$

4.9. Unidades

Ejemplo 1: Llamada a la función $esUnidad(5, 6)$ para comprobar si el número 5 es una unidad en $\mathbb{Z}_6$.

Dado que esta función se encarga simplemente de indicar si un número es o no invertible en un $\mathbb{Z}$ determinado, el único paso que se obtiene es el siguiente:

El número $\overline{5}$ es un elemento inversible ó unidad en $\mathbb{Z}_6$.

Figura 4.14: Resultado en la web para $esUnidad(5, 6)$

Ejemplo 2: Llamada a la función $esUnidad(3, 6)$ para comprobar si el número 3 es una unidad en $\mathbb{Z}_6$.

En este caso, al ser el 3 un elemento que no es inversible en $\mathbb{Z}_6$ obtendríamos el siguiente paso:

El número $\overline{3}$ no es un elemento inversible ó unidad en $\mathbb{Z}_6$

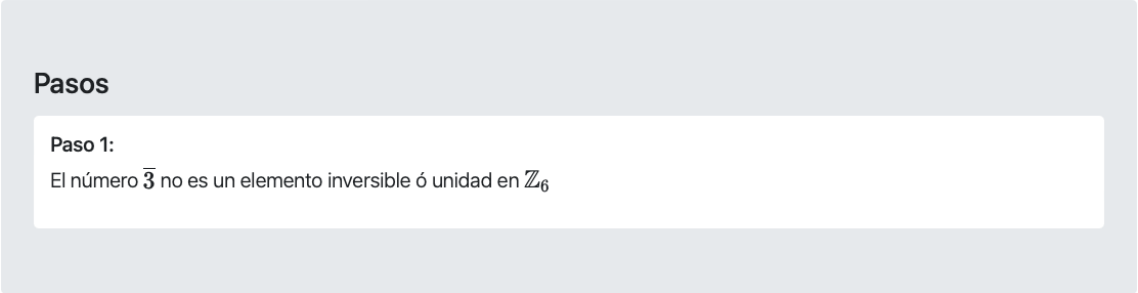

Figura 4.15: Resultado en la web para $esUnidad(3, 6)$

4.10. Divisores de cero

Ejemplo 1: Llamada a la función $esDivisorDeCero(3, 6)$ para comprobar si el número 3 es un divisor de cero en $\mathbb{Z}_6$.

Al igual que la función *esUnidad*, esta función simplemente indica si un número es o no divisor de cero en un determinado $\mathbb{Z}$. Por lo tanto en este caso se obtendría:

El número $\overline{3}$ es un divisor de cero en $\mathbb{Z}_6$.

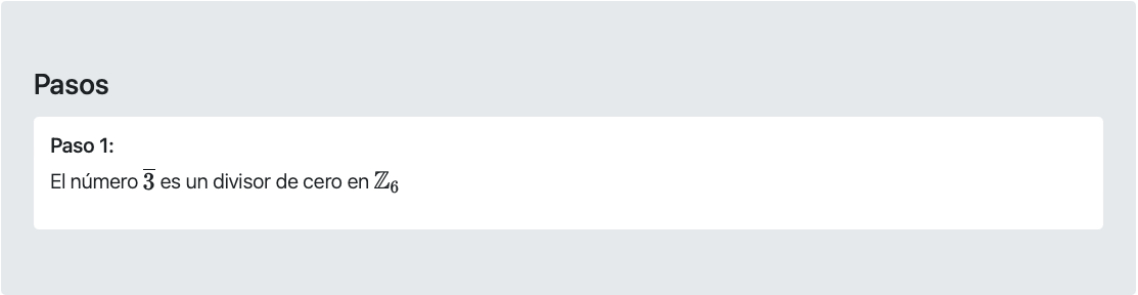

Figura 4.16: Resultado en la web para $esDivisorDeCero(3, 6)$

Ejemplo 2: Llamada a la función $esDivisorDeCero(5, 6)$ para comprobar si el número 5 es un divisor de cero en $\mathbb{Z}_6$.

En este caso, al no ser un divisor de cero, se obtendría el siguiente paso:

El número $\overline{5}$ no es un divisor de cero en $\mathbb{Z}_6$

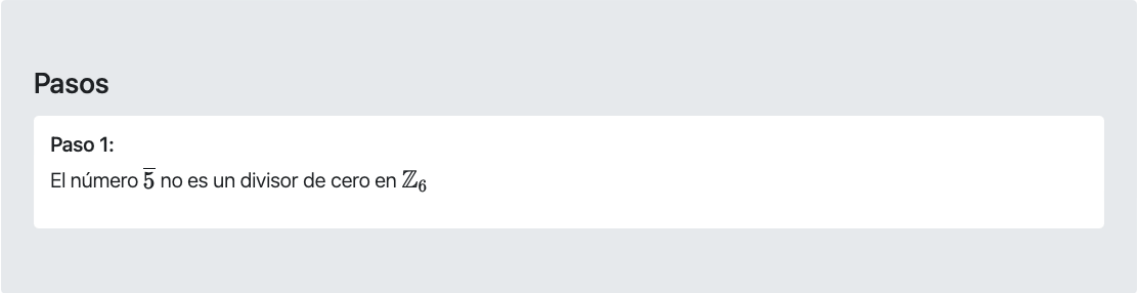

Figura 4.17: Resultado en la web para $esDivisorDeCero(5, 6)$

4.11. Inversos

Ejemplo 1: Llamada a la función $inverso(2, 7)$ para comprobar el inverso del número 2 en $\mathbb{Z}_7$.

A diferencia de la función *esUnidad*, esta función indica cuál es exactamente el inverso del número dado, por lo que en este caso se tendría:

El inverso de $\overline{2}$ *en* $\mathbb{Z}_7$ *es:* $\overline{2}^{-1} = \overline{4}$.

Esta función devuelve una variable adicional que se va a requerir para la resolución de ecuaciones y sistemas lineales de congruencias. En ella se obtiene el resultado directamente como se puede observar a continuación:


```
{
    "resultado": "4"
}
```


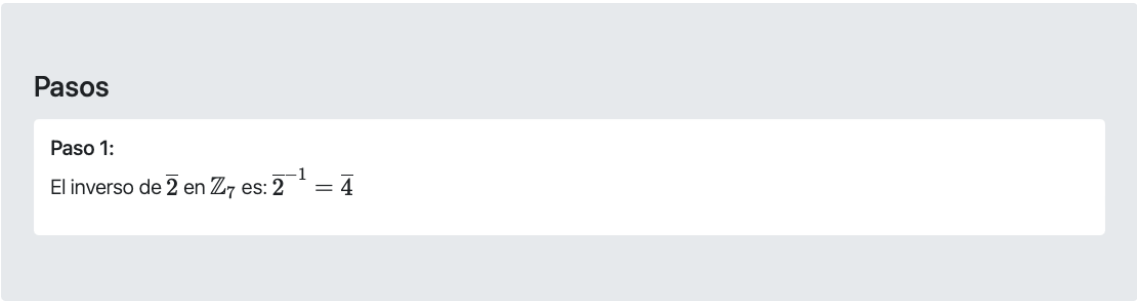

Figura 4.18: Resultado en la web para $inverso(2, 7)$

Ejemplo 2: Llamada a la función $inverso(3, 6)$ para comprobar el inverso del número 3 en $\mathbb{Z}_6$.

Como se comprobó en el apartado 4.9 en el ejemplo 2, el número 3 no es una unidad en $\mathbb{Z}_6$, por lo que se obtendría el siguiente resultado:

El número $\overline{3}$ *en* $\mathbb{Z}_6$ *no tiene inverso.*

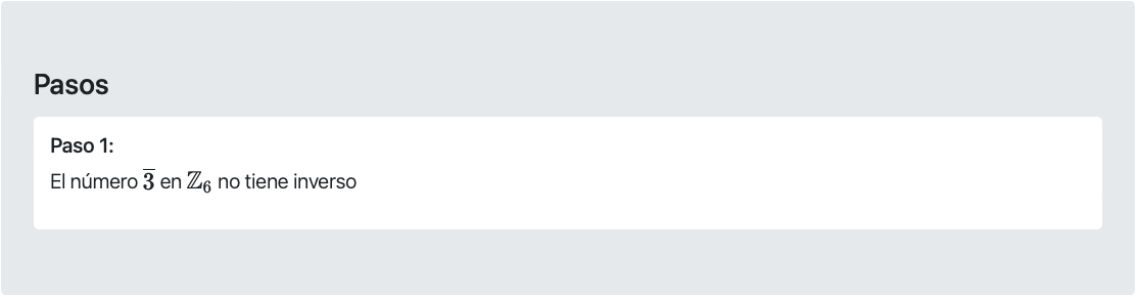

Figura 4.19: Resultado en la web para $inverso(3, 6)$

4.12. Ecuaciones lineales de congruencias

Ejemplo 1: Llamada a la función $ecuacion([7], [1], [2], [4], 7)$ para resolver la ecuación $7x + 1 = 2x + 4$ (mod 7).

El algoritmo que resuleve las ecuaciones lineales de congruencias comienza simplificando la ecuación en caso de ser posible. Al ser este el caso, el primer paso sería:

Simplificando la ecuación obtenemos: $5x = 3$ *(mod 7).*

A continuación se comprueba si la ecuación tiene solución devolviendo un paso como el siguiente:

La ecuación $5x = 3$ *(mod 7) tiene solución porque el* $\mathrm{mcd}(5, 7) = 1$ *divide a* 3.

Siguiendo con el algoritmo, se representa la ecuación en función de $\mathbb{Z}$, por lo que se obtendría esta respuesta:

$5x = 3$ *(mod 7) es equivalente a* $5x \equiv 3$ *en* $\mathbb{Z}_7$.

El siguiente paso contendría la resolución directa del algoritmo despejando la x de la ecuación:

$x = 5^{-1} * 3, x = 3 * 3, x = 9, x = 2.$

Finalmente se representa la solución de la ecuación en $\mathbb{Z}_7$ y en $\mathbb{Z}$:

En $\mathbb{Z}_7 : x = 2.$

En $\mathbb{Z} : x = 2 + 7t, t \in \mathbb{Z}.$

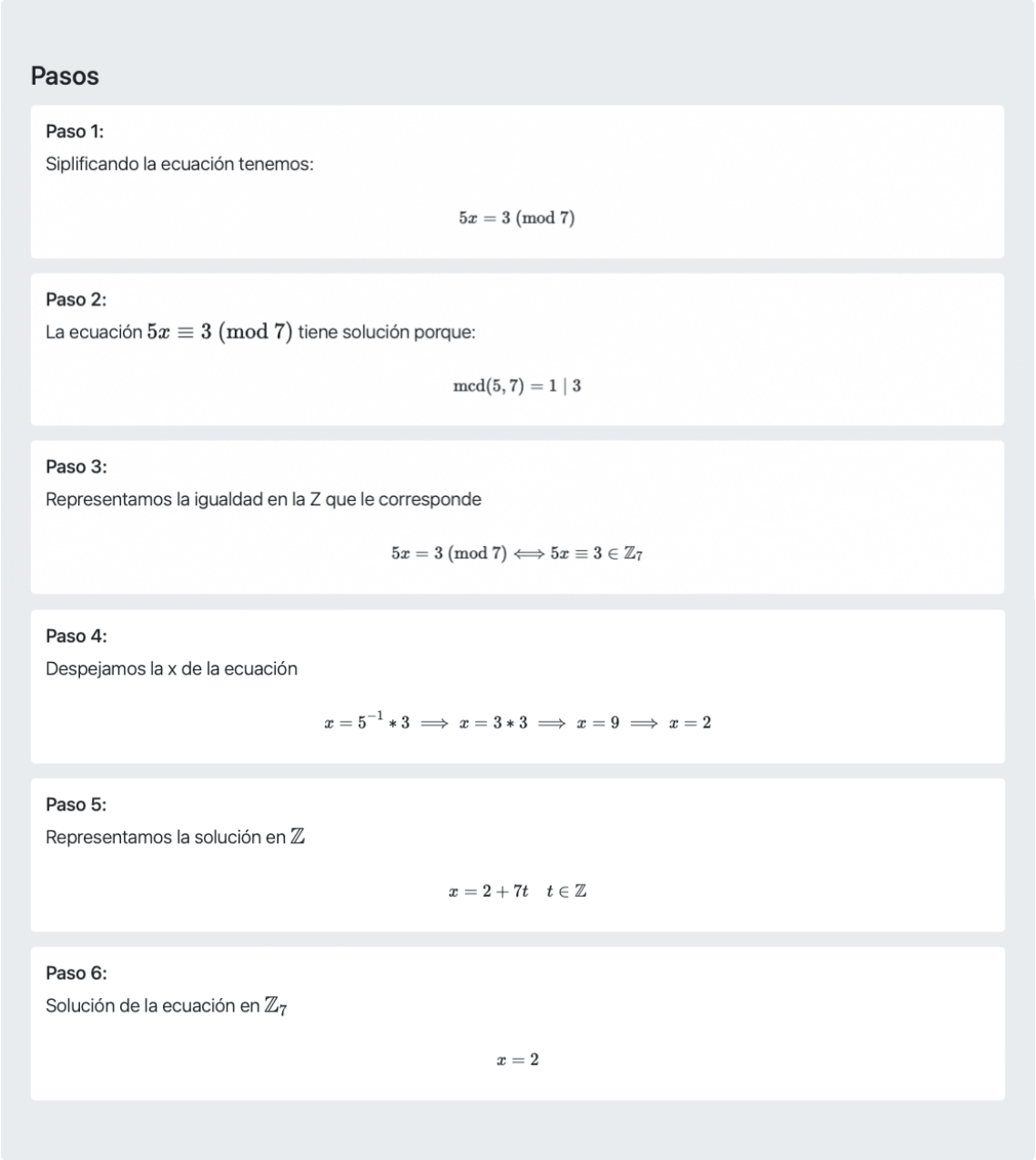

Figura 4.20: Resultado en la web para $ecuacion([7], [1], [2], [4], 7)$

4.13. Sistemas lineales de congruencias

Ejemplo 1: Llamada a la función $sistema([[1, 2, 5], [2, 1, 7], [3, 4, 11]])$ para resolver el sistema: $\begin{cases} x = 2 \pmod 5 \\ 2x = 1 \pmod 7 \\ 3x = 4 \pmod{11} \end{cases}$

El primer paso que realiza el algoritmo es simplificar y despejar las x de las ecuaciones, como se da en este ejemplo, devolviendo un array de ecuaciones que crean el siguiente sistema:

$$\begin{cases} x = 2 \text{ (mod } 5) \\ x = 4 \text{ (mod } 7) \\ x = 5 \text{ (mod } 11) \end{cases}$$

A continuación, el siguiente paso comprueba si el sistema tiene soluciónу da una explicación de por qué existe dicha solución:

Puesto que 5, 7 y 11 son primos entre sí, se puede aplicar el teorema chino del resto.

El sistema tiene solucion única en $\mathbb{Z}_p$*, con* $p = 5 \cdot 7 \cdot 11 = 385$

Siguiendo el algoritmo descrito en el marco teórico para resolver sistemas mediante el teorema chino del resto, el siguiente paso sería calcular la solución en función de los parámetros q_i:

$x = (2q_1 \cdot \frac{385}{5} + 4q_2 \cdot \frac{385}{7} + 5q_3 \cdot \frac{385}{11}) = (2q_1 \cdot 77 + 4q_2 \cdot 55 + 5q_3 \cdot 35)$ *(mod* 385*).*

De tal forma que se seguiría con la resolución de los valores de q_i. Este paso devuelve un array con las diferentes ecuaciones para calcular las q por lo que agrupándolas en un sistema quedaría de tal forma:

$$\begin{cases} q_1 * 77 = 1 \text{ (mod } 5) \implies 2q_1 = 1 \text{ (mod } 5) \implies q_1 = 3 \\ q_2 * 55 = 1 \text{ (mod } 7) \implies 6q_2 = 1 \text{ (mod } 7) \implies q_2 = 6 \\ q_3 * 35 = 1 \text{ (mod } 11) \implies 2q_3 = 1 \text{ (mod } 11) \implies q_3 = 6 \end{cases}$$

En el siguiente y último paso se sustituyen las q en la solución de x descrita, calculando el resultado final:

$x = (2 \cdot 3 \cdot 77 + 4 \cdot 6 \cdot 55 + 5 \cdot 6 \cdot 35) = (462 + 1320 + 1050) = 2832 = 137$ *(mod* 385*).*

Pasos

Paso 1:

Simplificando las ecuaciones tenemos el siguiente sistema:

$$\begin{cases} x \equiv 2 \pmod 5 \\ x \equiv 4 \pmod 7 \\ x \equiv 5 \pmod{11} \end{cases}$$

Paso 2:

Puesto que 5, 7 y 11 son primos entre sí, se puede aplicar el teorema chino del resto.

Paso 3:

El sistema tiene solucion única en $\mathbb{Z}_p$, con $p = 5 \cdot 7 \cdot 11 = 385$

Paso 4:

Representación de la solución en función de q

$$x = (2q_1 \cdot \frac{385}{5} + 4q_2 \cdot \frac{385}{7} + 5q_3 \cdot \frac{385}{11}) = (2q_1 \cdot 77 + 4q_2 \cdot 55 + 5q_3 \cdot 35) \pmod{385}$$

Paso 5:

Sistema y pasos para calcular cada una de las q

$$\begin{cases} q_1 \cdot 77 \equiv 1 \pmod 5 \Rightarrow 2q_1 \equiv 1 \pmod 5 \Rightarrow q_1 = 3 \\ q_2 \cdot 55 \equiv 1 \pmod 7 \Rightarrow 6q_2 \equiv 1 \pmod 7 \Rightarrow q_2 = 6 \\ q_3 \cdot 35 \equiv 1 \pmod{11} \Rightarrow 2q_3 \equiv 1 \pmod{11} \Rightarrow q_3 = 6 \end{cases}$$

Paso 6:

Cálculo de la solución del sistema

$$x \equiv (2 \cdot 3 \cdot 77 + 4 \cdot 6 \cdot 55 + 5 \cdot 6 \cdot 35) = (462 + 1320 + 1050) = 2832 \equiv 137 \pmod{385}$$

Paso 7:

Solución del sistema

$$x = 137 \pmod{385}$$

Figura 4.21: Resultado en la web para $sistema([[1, 2, 5], [2, 1, 7], [3, 4, 11]])$

Capítulo 5

Análisis de impacto

La Agenda 2030 es un plan de acción global creado por las Naciones Unidas para abordar los problemas mundiales (sociales, económicos y ambientales) con el fin de alcanzar unos objetivos para el año 2030. En este capítulo se analizará el impacto de este TFG en dichos objetivos, también conocidos como Objetivos de Desarrollo Sostenible u ODS.

Como se explicó en el capítulo de 'Introducción', este TFG forma parte de un proyecto llamado 'Proyecto calculadora' en el que varios alumnos desarrollan una plataforma para resolver problemas y ejercicios de las diferentes asignaturas del grado de Matemáticas e Informática, apoyándose en el código perteneciente a los diferentes trabajos de fin de grado. El simple hecho de estar desarrollando una plataforma didáctica se relaciona con el objetivo número cuatro de los ODS, 'Educación de calidad'. Esta plataforma se ha pensado para su uso por parte de los alumnos que necesiten ayuda resolviendo problemas matemáticos, promoviendo una educación inclusiva al ser accesible para todos (cumpliendo también co el objetivo número 10, 'Reducción de las desigualdades'). Además, facilita el aprendizaje de los alumnos en el campo de las matemáticas discretas debido al énfasis de describir los pasos necesarios para resolver los problemas.

El código desarrollado en este TFG cumple con el objetivo número 9 de los ODS, 'Industria, innovación e infraestructura' debido a que se va a integrar en una plataforma más amplia, por lo que se contribuye al desarrollo de infraestructuras digitales para el aprendizaje. Por otra parte este TFG promueve la innovación al brindar una alternativa para que los alumnos puedan resolver y entender problemas relacionados con la aritmética entera y modular.

Bibliografía

[1] Águeda Mata, *Matemática discreta I : guía de clase*. Madrid Fundación General de la UPM, 2013.

[2] G. Westreicher. (2020) Criterios de divisibilidad. [Online]. Available: https://economipedia.com/definiciones/criterios-de-divisibilidad.html

[3] Sympy.org. (s.f.) Number theory - sympy. [Online]. Available: https://docs.sympy.org/latest/modules/ntheory.html

[4] Python. (s.f.) itertools - functions creating iterators for efficient looping. [Online]. Available: https://docs.python.org/3/library/itertools.html

[5] ——. (s.f.) fractions - números racionales. [Online]. Available: https://docs.python.org/3/library/itertools.html

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Sun Jun 11 17:06:04 CEST 2023
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)